Joachim, Dunkel

Controlling a Holonomic Mobile Robot for Automated Reconfiguratoin of Production Lines

Bachelor's Thesis

to achieve the university degree of

Bachelor of Science

Bachelor's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl-Ing. Dr.techn. Gerald Steinbauer-Wagner

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Franz Wotawa

Graz, June 2022

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum

Unterschrift

Abstract

In order to even more improve the flexibility of production lines an automated physical reconfiguration (e.g change of location of machines) is envisioned [Str22]. For this purpose automated mobile robots are needed that can precisely navigate and lift, move, and settle production machines.

The aim of this thesis is to develop a solution for approaching and maneuvering underneath these production machines, autonomously and without collision. A system with three main components was developed which uses the robots lidar sensors to precisely determine the machines position and to navigate beneath it. In the first part point cloud data is filtered using outlier removal techniques and clustering is used to extract the individual machine legs. In the second step the relative position of the machine w.r.t the robot is determined using a hand-crafted template and scan matching. To use the machine localization for navigating under it, a linear controller was created. The entire system was developed using ROS and PCL with Gazebo and tested on the real robot.

Acknowledgment

I wish to acknowledge the help provided by the technical and support staff at EvoCortex. I would also like to thank Daniel Strametz and the whole SmartFactory team who provided me with the robot system.

I especially want to express my gratitude to my primary supervisor Gerald Steinbauer-Wagner, who guided me throughout this project.

Contents

1 Overview

1.1 Introduction

SmartFactory@TuGraz is working on developing industry 4.0 applications. This work is part of a bigger project, which aims at developing a fully automatic dynamically configurable production line that lets a desired worker robot autonomously create a work piece without any human interference (See [Str22]). The production line involves specifically designed machine tables that house different machines such as a standing drill or a table grinder.

Figure 1: The SmartFactory production site.

The system's operation should work as follows. Once the production system receives an order for a specific work piece a plan for re-configuring the production line for efficient production is obtained. Based on that optimized machine layout an updated detailed production plan is generated.

For an automated reconfiguration the transport robot must fetch and place the correct machine tables next to each other in the correct configuration and then interlock them with the table's built-in interlocking system. This mechanism ensures the rigidity and sturdiness of the configured assembly line. Interlocking two tables involves lifting one table up, driving it next to another table and lowering it so that the dual interlocking parts match. When the assembly line is completed, the shuttle's job is done and the worker robot navigates to one table after another and operates the machines with its gripper. After the last machine table the work piece is finished.
One essential part of this autonomous system is the shuttle robot's ability to position itself correctly in order to pick up and set down the aforementioned machine tables.
Within the scope of this project a precise method to automatically navigate

beneath a production table, that does not require an external localization of the table or the robot, was developed.

1.2 The Robot System

The target robot platform for this project is the EvoRobot Industrial from EvoCortex [Evo22], which allows for precise navigation and lifting of loads.

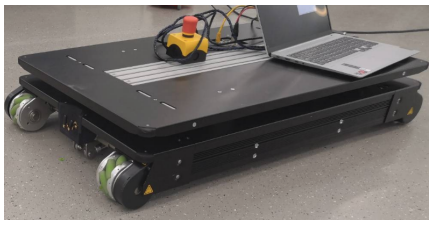

Figure 2: The EvoRobot shuttle.

As a versatile, autonomous transport and assembly platform, its main features are omnidirectional movement, 2-Sick-safety scanners as well as a fine-positioning sensor on the bottom. Figure 3 shows the robot and its dimensions (in mm).
We can see the robot's lidar-sensors are mounted diagonally inside it, between the upper and lower part of it's chassis (See Figure 2. For stability the upper and lower upper part of the robot's chassis are connected with two thin aluminum profiles. This design provides a 360 deg view around the robot. It's only downside is that objects may disappear in a blind-spot if they come too close (See Figure 4).

While navigating underneath a table, the table's legs are so close to the robot, that this happens on multiple occasions. For details how this issue is handled in the navigation please refer to Section 4.

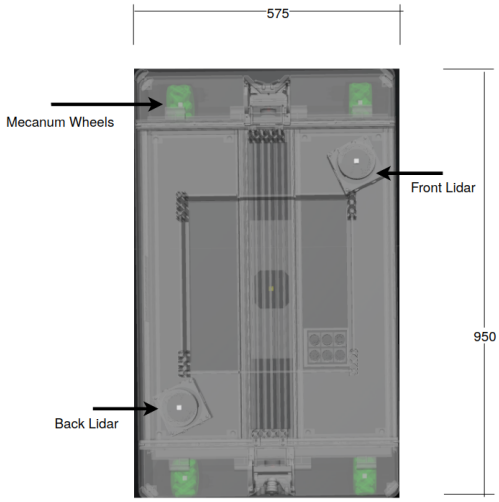

Figure 3: Bird eye view - Gazebo

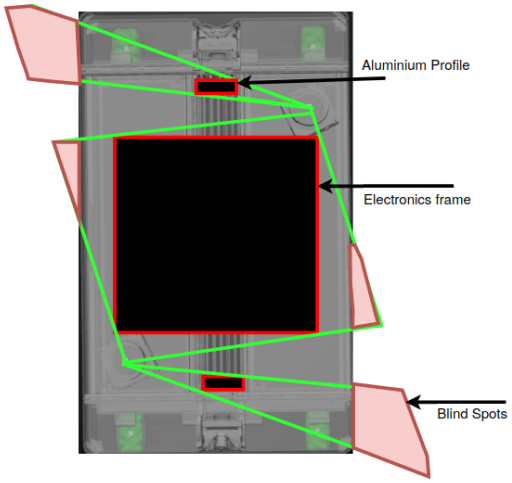

Figure 4: Blind spots of the robot's lidar-sensors

1.3 The machining table

As described in Section 1.1, this system was built to make the robot steer underneath specific machining tables. Figure and 6 show such a table in real life and simulation. Figure 7 shows its respective dimensions.
Also note the interlocking mechanisms mounted on all four sides of the table, circled red in Figure 5. Each interlocking mechanism contains a male and female counterpart, so that all the machining tables used in the envisioned production line can be connected (See [Str22]).

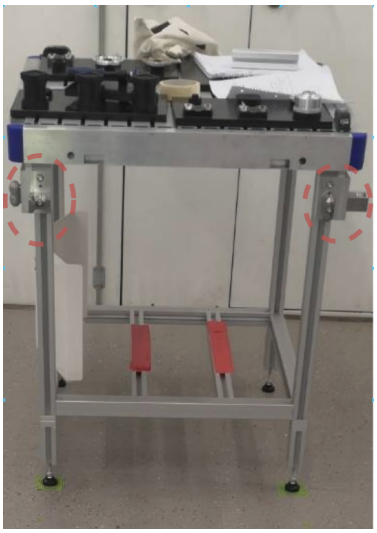

Figure 5: A machine table at SmartFactory

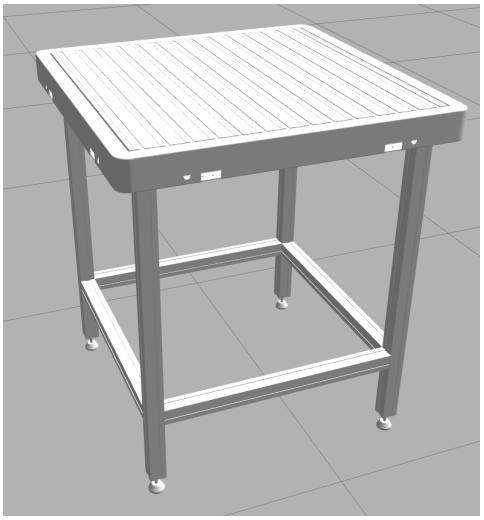

Figure 6: The table as seen in simulation

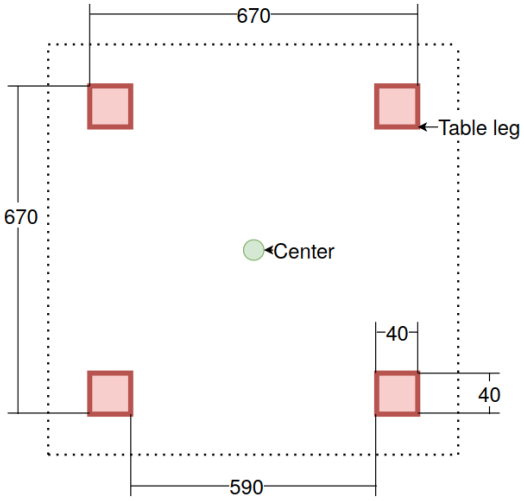

Figure 7: Table - Dimensions

1.4 Assumptions

Fine maneuvering in relation with the table without external positioning sensors for the robot and the table is only possible if the robot is already located in close proximity to the table.
In order to properly integrate this method with the bigger system, following preconditions for its execution were defined:

1. The robot needs to be within 10 to 150 cm in front of the table side it intents to enter.

2. There must not be any obstacles in the robot's way or underneath the table.

3. The robot's orientation should be no more than +- 25 degrees off from facing the table side directly.

1.5 Definitions

The proposed system is based on ROS (Robot Operating System) [Sta18] and uses PCL (Point Cloud Library) [RC11] for point cloud processing.
ROS uses special messages with its publish-subscriber protocol allowing individual components (nodes) to communicate with each other in real time.
Here we present some definitions, with their corresponding ROS & PCL message types, which will be used throughout this document.

entity	definition	ROS & PCL message type
position	2D Position (x, y) in m	geometry-msgs/Point
orientation	Yaw, (θ) in radiants	geometry-msgs/Quanternion
pose	position + orientation	geometry-msgs/Pose
point cloud	list of 2D Laser-scan points	pcl::PointCloud<T>
velocity command	3D Vector (v_x, v_y, v_θ)	geometry-msgs/Twist

2 System overview

This section will give the reader a high level overview on the whole system's implementation and how the individual parts work together.
The robot system sends continuous laser-scan data from its two Sick-Safety 2-D lidar sensors positioned on its diagonals. (See Section 1.1).

The Laser-scan multi-merger package [Bal+14] is used to combine both lidar sensors into one point cloud.

In order to start the system, an initial position located somewhere underneath the selected table is sent to the filtering module.

The Filtering module receives this initial pose and begins filtering laser-scan data.

Noise and other effects such as stringing on sharp edges, and problems when dealing with reflective surfaces create unwanted outliers in the robots point cloud data. Therefore outlier removal techniques are used. A heuristic is used to identify the table in the whole point cloud data. Every laser-scan point not associated with the table is removed. The resulting cleaned point cloud data which now only contains the table of interest is sent to localization.

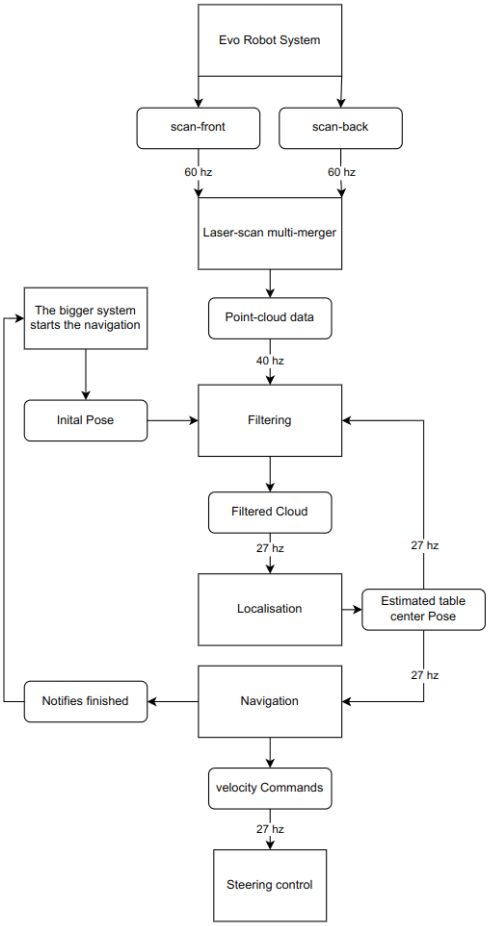

Figure 8: How the whole system works together.

The Localization module uses geometric reasoning to find out how many table legs are visible. Iterative Closest Point (ICP) plus a hand-crafted

template are used alongside a good initial guess to find out the table center pose. This pose is sent to both the filtering and the navigation node. Filtering needs this continuously updated position to find the nearest clusters representing our table legs.

The Navigation module uses the obtained table center pose to find a trajectory and to steer towards the table center. A linear controller is used to send the desired velocity commands to the underlying robot's steering control. An intermediate steering goal is defined to be in front of the table. Thus our holonomic robot drives in a rectangular path around the table legs. In order to prevent non-feasible velocity commands, ramping, cutting of above max-velocity and searching for a feasible trajectory underneath the table are performed.
If the goal is reached the navigation node notifies the bigger system of success.

A note on performance:
As seen in Figure 8 laser scan data is passed to the filtering module with a frequency of 40 Hz. There this data is processed with 27 Hz. All other involved parts can easily maintain this frequency and potentially process arriving data even faster. The filtering module is the systems main bottleneck because it has to reduce point cloud data of the robots whole environment to only containing points from the table of interest. After the filtering node is done all other modules only have to work with point cloud data that is up to 20 times smaller then before.

In the following sections we will present details about the different modules

3 Filtering

In order to use the point cloud data provided by the lidar sensors to localize the table of interest, the data needs to be filtered first. This is necessary because lidar sensor data contains noise that needs to be removed. Sources of such noise include stringing effects on sharp edges as well as problems when dealing with reflective surfaces. Both can therefore be caused by the legs of the machine tables we are trying to locate.

3.1 Statistical outlier removal

For removing such outliers the filter "StatisticalOutlierRemoval" [Rus+07] provided by PCL is used. Algorithm 1 describes this in more detail.

Algorithm 1 Statistical outlier removal

1: Iterate over the input set P and build a kd-tree for distance search.
2: **for** point p_i in P **do**
3: Compute mean distance d_i from p_i and all of its K neighbors.
4: store d_i in D
5: **end for**
6: Compute μ and σ for all distances in D
7: outlier-threshold $\theta = \mu + \alpha * \sigma$ ▷ α = Sigma multiplier
8: **for** p_i in P **do**
9: remove p_i from P if $d_i > \theta$
10: **end for**
11: **return** P

This filter works in the following way. For each point, the mean distance from it to K of its neighbors is computed. By assuming the point-position distribution to be Gaussian, all points that are farther away than the defined standard deviation multiplier are removed from the point set.
In our case it was necessary to have different filter parameters for real life and simulation (See Table 1). This is necessary because real life data contains more outliers. (see Section 6.1.4 for detailed results).

	K	α
sim	2	2
real	3	3

Table 1: Different filter parameters for real world and simulation.

After using this filter most individual outliers are removed from the data.

3.2 Radius outlier removal

After applying "StatisticalOutlierRemoval" some outliers are still present in the point cloud. Especially stringing outliers are often at a close distance to the next point in the data set and therefore survive this Gaussian filtering technique.
For this reason "RadiusOutlierRemoval" [RC11] is used complementary (See Algorithm 2). This filter removes points that are surrounded by fewer than a defined number of points n_{min} within a specified radius r.

Algorithm 2 Radius outlier removal

1: Iterate over the input set P and build a kd-tree for distance search.
2: **for** point p_i in P **do**
3: Count all points p_r closer to p_i then defined radius r
4: **if** $count(p_r) \geq n_{min}$ **then**
5: Add p_i to P_{out}
6: **end if**
7: **end for**
8: **return** P_{out}

For our implementation a n_{min} of 2 and a r of $0.1m$ was chosen. Application of this filter ensures that only dense regions such as table legs remain in the point cloud.

3.3 Removing unwanted objects

At this stage recorded point cloud data still contains objects of the whole environment surrounding the robot, not just the table of interest. Every point belonging to an unwanted object is therefore filtered out here.
Assumption 2 ensures that there is no obstacle in between robot and table. The higher level system starts the filtering node by providing it with a rough estimation where the table should be. This initial position is used to find the four closest table legs that adhere to geometric constraints of our tables. Later the found table localization is used to update the initial position search point for filtering.
Firstly every point that is farther away than the diagonal length of the table from the initial position guess is removed. Based on the assumption that an initially guessed position is anywhere between the table legs, this will not remove any points from our desired table.
In order to identify which points belong to which table leg the data is clustered using "EuclidianClusterExtraction" [RC11].
This method works by dividing point cloud data into clusters based on the Euclidean distance between points. Clusters are built by searching the space based on a defined radius (in this case $r = 0.1m$).
After cluster extraction all clusters that do not belong to the table of interest are removed. This is done by performing geometrically constraint inclusion of clusters, based on their centroids.

Essentially this works by including the closest four clusters to our initial point that geometrically could form a table with each other.

Algorithm 3 Decide cluster inclusion

1: Get all centroids c of all clusters
2: Sort centroids C based on their distance to the initial point p
3: **while** C != empty or output-set $O \leq 4$ **do**
4: Pop centroid c_0 from C
5: Add c_0 to O if it's distance to any other point in O follows:

$$d\left(c_0, o_i\right) = \sqrt{\sum_{i=1}^{n} \left(q_i - p_i\right)^2} = D$$

$$D \in \{0.67 \pm 0.1, 0.9475 \pm 0.1\}$$

6: **end while**
7: **return** O

In this case the side length between two legs of our table is $0.67m$ and the diagonal length between two legs of our table is $0.9457m$.
Empirically an inclusion threshold of $0.1m$ was added since the position of extracted table-leg centroids is vulnerable to sensor noise.

The following image (Figure 9) shows an example of how filtering transforms point cloud data. All outliers and obstacles that are not part of the table are removed. Point cloud data before and after filtering is shown in green and brown here. Note: The green line represents a wall behind the robot, which was removed as well.

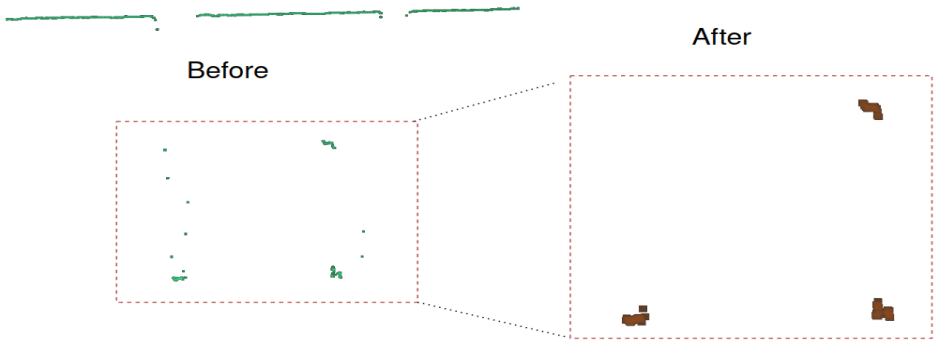

Figure 9: Filtering removes unwanted objects and outliers in the data.

3.4 Limitations

There are still some corner cases that need to be addressed when integrating this approach into a larger system. While the robot is navigating underneath the table, it can occur that only one leg is visible (See Section 1.2). It then can falsely keep table-leg clusters of neighboring tables inside our point cloud if their distances to the visible leg adhere to the geometric inclusion constraints. Localization performed on such a point cloud then produces wrong results that could lead to the navigation break down as a whole.

4 Localization

After filtering only the target table inside our point cloud data remains. The next step is to localize the position of the table as seen from the robot's local coordinate frame.
Essentially localization is done by performing scan matching on point cloud data using a hand-crafted template. This is done using "Generalized Iterative Closest Point" [SHT09], a more robust version of standard ICP. All ICP variants work by finding the best transformation that minimizes a distance function between two point clouds. While ICP always converges to a local minimum, an initial alignment (initial guess) is usually required to increase the likelihood of this minimum also being the global one.
Usually this is achieved by overlaying the centroids of both point clouds or similar methods. Scan matching accuracy heavily depends on the initial guess. In this particular case even more so, since scan matching is not performed on two point clouds produced by the same environment, but rather on a hand-crafted template with the current laser scan situation. Due to noise and blind spots in the sensor readings, the recorded point cloud and thus the number of visible table legs differ substantially while navigating underneath the table. (See Section 1.2) Both the chosen template and the initial guess method must account for this problem.

The following figure illustrates how blind spots affect our point cloud data while navigating underneath the table.

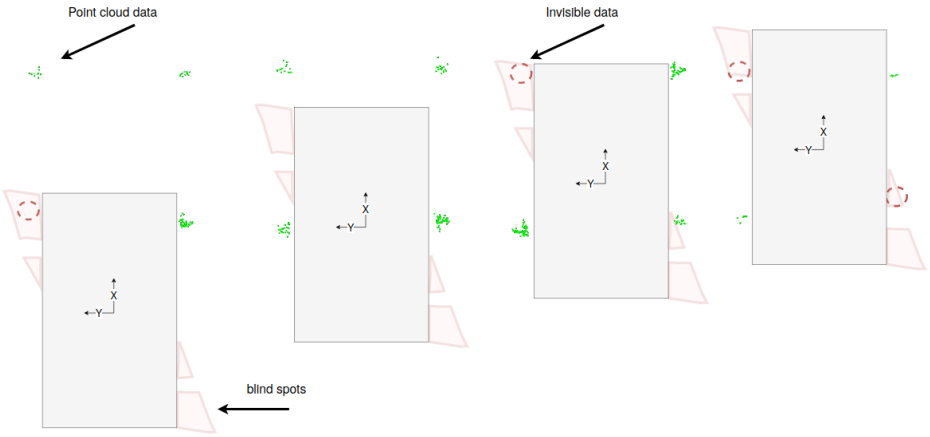

Figure 10: Visible legs during navigation underneath the table. The green dots represent visible point cloud data, red circles illustrate missing table legs due to sensor blind spots.

4.1 Crafting the template

An initial template was sampled by hand and stored inside a .pcd file. It is loaded during node startup.

As seen on Figure 11 the template is a point cloud with four sampled squares as table-legs. Points are equally distributed with $5mm$ of space between them. Since one table leg has a $40mm$ squared aluminum profile, there are eight points on each side of the leg. In total there are 32 points per leg, the whole template therefore consists of 128 points.
While having fewer points worsens scan matching, additional points do not further increase the scan

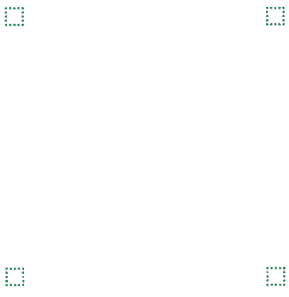

Figure 11: The template used for scan matching

matching accuracy, but do increase the computation
cost. Therefore a $5mm$ distance was chosen empirically. A square-shaped template was selected because it closely resembles the square shape of the table and its legs.

When a filtered point cloud arrives for localization, the system checks which table-legs are visible in the data. This information is used for removing any leg cluster in the template that does not have a corresponding leg-cluster in the received data.
This procedure works as follows. First, all cluster centroids from the sensor cloud data are retrieved. Then the number of legs (i.e. cluster-centroids) in the sensor cloud is counted and geometrical reasoning is used to identify which cluster should be removed from the template.
A rough overview of the algorithm is given here. More details about the implementation can be found in the project's github repository.

There are three situations that need different handling:

- There are 4 legs visible $\rightarrow$ the template can be left as is.
- There are 3 legs visible $\rightarrow$ there are four different cases to deal with.
- There are 2 legs visible $\rightarrow$ six distinct cases are possible.

If there are 2 or 3 legs visible, the x and y distances of the cluster centroids to the robot center are used to determine to which table legs the present clusters correspond. For instance if its y position is negative a left leg is found.
In the case of three visible legs, sorting legs by their x position w.r.t to the robot coordinate system ensures that the closest is always one of the lower legs, and the farthest is one of the upper legs.

This easy geometric reasoning is only possible because of Assumption 3. For instance, if the robot was rotated more than $45°$ degrees, the upper left leg could have a lower x-position.
After the present legs are identified in the input data, all legs that do not have a corresponding leg in the input date are stripped from the template.

4.2 Initial Estimate

As described above, scan matching with ICP requires a good initial guess in order to find the global minimum.
When crafting the template, geometric reasoning was used to find out which table legs are present in the point cloud data and our system takes advantage of this information here.

The algorithm is illustrated in Figure 12 and works as follows.

1. Connect all cluster centroids with each other. (If all four legs are present this makes 6 edges.)
2. Compute the center point of every edge.
3. Add an offset to every edge center point depending on which leg connection they represent. (For instance: Any diagonal connection center point does not get an offset at all. Any connection center point on the side receives half a table length offset so that it points to the center as much as possible.)
4. Record the center point's positions and the edges rotation w.r.t. the robot's coordinate frame.
5. The resulting rotation and position of this initial guess is the mean over all recorded ones.

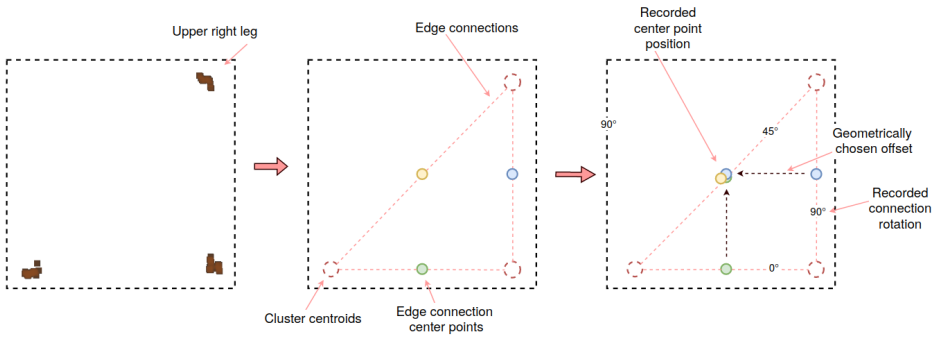

Figure 12: An illustration of initial guess estimation method.

Note: First a simpler approach was used, which consisted of just overlaying

the table and template cloud centroids. Using aforementioned method improved scan matching considerably (See Section 6.1.6).

4.3 Scan Matching

Before performing scan matching, the template is transformed by the initial guess $T_{initial}$ so that it is already close to the correct position.
Note: Transformation of a point cloud (in this case the template) refers to applying a 2D-Affine transformation to all of its points.

As discussed above, GICP is used for scan-matching.
This is an ICP variant that takes the planar structure of both scans into account. It can be viewed as an ICP variant with plane-to-plane error metric that assumes a Gaussian-distribution of points around the true pose. For more information we refer the reader to [SHT09].
GICP is then executed with the template cloud as source and the current sensor data as target.
Eventually it converges and finds a transformation T_{icp}.
T_{icp} is then multiplied with $T_{initial}$ to receive the full transformation T_{full}.
T_{full} presents the estimated transformation between the robot's local coordinate system and the table center. This transformation can be directly send to the navigation and filtering modules as desired robot pose P_{table}.

5 Navigation

5.1 Overview

After localizing the desired pose P_{table} through scan matching, the obtained information is used to move the robot underneath the table without it hitting a table leg.
That means to minimize the distance $P_{table} \Rightarrow 0$. To achieve that, a feasible trajectory towards the goal must be found.
Since this can generally be seen as a local path planning problem, existing path-planning methods were evaluated. In the case of the omnidirectional robot, "Time-Elastic-Band" Planner [RHB15] seemed especially promising and was therefore tested out. The TEB-planner along with the ROS-Navigation stack was integrated into our system by publishing our localization as a global reference pose. However, this approach did not work because the Y – accuracy of our localization has some outliers. These can sometimes mistakenly appear in the robot's path, prohibiting from moving forward.

While driving beneath the table, the robot only has $7,5mm$ space on either side (See Introduction). This narrow environment and the outliers make using the ROS-navigation stack unpractical for this task.

Our approach involves using the obtained table center pose to find a trajectory and to steer towards the table center. Since EvoRobot is a holonomic platform a linear controller is used to directly send desired velocity commands to the robot's motion control. In order to drive toward the table center without hitting any legs, an intermediate steering goal is defined to be in front of the table. Thus the robot navigates in a rectangular path around the table legs. In order to prevent non-feasible velocity commands, ramping, cutting of above max-velocity and searching for a feasible trajectory underneath the table are performed.

5.2 The Linear Controller

For any holonomic platform, kinematics constraints do not have to be taken into account.

Our navigation system therefore uses a linear controller, to translate a desired table center pose P_{goal} into a velocity command $V_{desired}$ (v_x, v_y, v_θ). Each component of $V_{desired}$ is calculated as follows.

$$v_x = \begin{cases} \alpha * x_{Pgoal}, & \text{if } x_{Pgoal} \leq x_{Vmax} \\ x_{Vmax}, & \text{otherwise} \end{cases}$$

$$v_y, v_\theta = \text{—"—}$$

$V_{desired}$	(v_x, v_y, v_θ)Newly calculated velocity.
V_{max}	Maximal acceptable velocity.
P_{goal}	(x, y, θ) Desired 2D-pose as seen from robot coordinate frame
α	controller parameter for obeying acceleration constraints

5.3 Navigation Behavior

Since navigation has to perform many more tasks, our system internally uses a state-machine with the following states:

5.3.1 Ready

When the navigation system is started, it goes into the initial state "Ready". Upon receiving its first desired pose from the localization module it immediately proceeds to the next one.

5.3.2 Lift down

Before driving underneath the table, the robot has to be in lowered position. This state notifies the robot's lifting system, to go into lowered position.

5.3.3 Driving in front

Before navigating underneath the table, the robot must be positioned directly in front of it. For that an intermediate goal $P_{infront}$ is defined.

$$P_{infront} = P_{table} - \begin{pmatrix} 0,9m \\ 0 \\ 0 \end{pmatrix}$$

The empirically chosen distance of $0.9m$ ensures that the robot does not hit any table legs. Since this is a holonomic robot, the desired position can be directly fed to the linear controller defined above. This is possible because of Assumption 2.
While driving outside of the table, a higher V_{max} can be used.
A goal is reached if all components of P_{goal} (x, y, θ) are smaller than that of a defined threshold $T_{accepted}$.

$$T_{accepted} = \begin{pmatrix} 0.01m \\ 0.005m \\ 0.5° \end{pmatrix}$$

Eventually $P_{infront}$ is reached and the system switches to the next state.

5.3.4 Driving inside

The robot is now directly positioned in front of the table with only little y and θ deviation given by noise or mislocalization. In this stage the table legs also have to be avoided when driving towards the table center P_{table}.
First, our linear controller system is used to get the desired velocity $V_{desired}$. Then a feasible velocity $V_{feasible}$ that does not cause a collision with a table leg has to be found. This is done in the following way. An inflated robot

bounding box R_{hull} is created at the robots position. This bounding box is a rectangle with its length being the robot's length in lowered position and width being the robot's width plus an empirically-chosen buffer of $5mm$ to avoid collision. Using a lower buffer size increases the y-offset the robot can have to the table's center position while using a bigger one can result in the robot not being able to drive at all.

R_{hull} is then transformed by the x, y, θ components of $V_{desired}$. This can be seen as simulating the movement the robot would do if $V_{desired}$ was sent to the motion control system for one second.

With the currently observed point cloud data, the system then checks if there are any points within R_{hull}. If so, the tested velocity values are invalid. Seven different combinations of $V_{desired}$'s three components (x, y, θ) are tested this way. These combinations are:
| x | y | θ | x and y | x and θ | y and θ | x, y and θ |
If none of them are valid, $V_{desired}$ is not feasible. In this case the values of $V_{desired}$ are halved and the system tries again. This "drive-if-you-can" approach can be seen as doing a binary search for a feasible $V_{desired}$.
Whenever a valid $V_{desired}$ is found, it is sent to the motion control system.

Our navigation approach works well during the earlier stages of driving underneath the table, where all it's legs are still visible. Towards the end of reaching the table center, most of our table legs are barely or not visible at all (See 10). Therefore, after the robot (i.e. its base frame) is closer to the center than a specified threshold($x_P < 0.75m$), the robot ignores $V_{desired}$ and just drives straight until it reaches its goal P_{table}. Doing so decreased the amount of time navigation needs considerably, while not loosing any robustness. Fine-positioning already ensures that the robot is properly positioned and oriented underneath the table.

Eventually the robot reaches its goal P_{table} and proceeds to the last state.

5.3.5 Finished

The navigation node signals to the bigger system that the robot is now located close enough to the table center and navigation is finished.

6 Evaluation

In this section, the results obtained during testing the localization and navigation system used in our solution are presented and evaluated. To properly compare test results, performance metrics and test setups had to be defined. The system was developed with the help of a simulation. As discussed in Section 4 there are some differences in how the table is perceived by the robot versus the real-world.
This has an impact on the performance. Therefore the simulation and the real-world were evaluated separately.

6.1 Localization

Irrespective of the robot's position, localization should always yield the best possible table center pose estimation achievable.
Two different test-setups were created: One for testing localization accuracy in front and one for the same when under the table.

6.1.1 Performance metric

Our test setup involves locating the robot at a certain location and orientation, recording data and then comparing the localization result to the measured ground truth. Figure 13 should illustrate this.

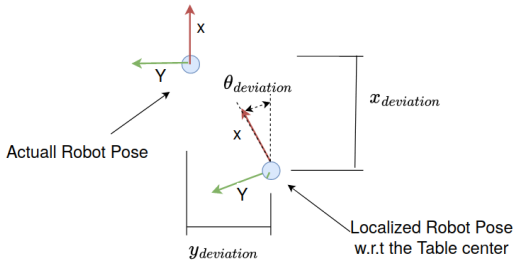

Figure 13: The taken localization error-measurements.

Each data sample we recorded with a bag-file consisting of 20 different laser-scan measurements. The evaluation for one configuration is done as follows:

$$\epsilon_{distance} = \frac{1}{20} \sum_{i=1}^{20} ||p_{localized} - p_{measured}||$$

$\epsilon_{distance}$	Absolute distance error.
$p_{localized}$	Position given by localization
$p_{measured}$	Measured ground truth position

The absolute distance deviation from the measured ground truth for all 20 sensor-readings is calculated and their mean is computed. The absolute distance is defined as the l2-norm.
For the robot orientation deviation, the absolute value between ground-truth and sensor-readings is taken as follows.

$$\epsilon_{orientation} = \frac{1}{20} \sum_{i=1}^{20} |\theta_{localized} - \theta_{measured}|$$

$$-25 \leq \theta_{(localized/measured)} \leq 25 \qquad \text{as given by Assumption 3}$$

For localization underneath the table, x-offset proved to be irrelevant and therefore only the y and θ deviation means are collected. Measurements taken in the real world are prone to error especially since a tape measure was used. Therefore a ground-truth inaccuracy of $\pm 0.5cm$ is assumed. On the other hand, ground truth measurements in the simulation are given by Gazebo and are therefore very accurate.

6.1.2 Outside Table Test Setup

In order to properly represent multiple different poses inside our test setup, a grid with 12 equally distant points was defined in front of the table. As illustrated by Figure 15 the robot was placed in 3 different rotations onto the grid's points. For each of the 36 different configurations a bag-file was recorded and ground truths were collected.

As seen in Figure 14, the robot was placed on every position with its left corner. This was necessary to get a more accurate and reproducible measurement using tape-measures in the real-life scenario. Otherwise figuring out if the robot's center point is exactly on top of the position marker, would have been inaccurate and result in a lot of wasted time.

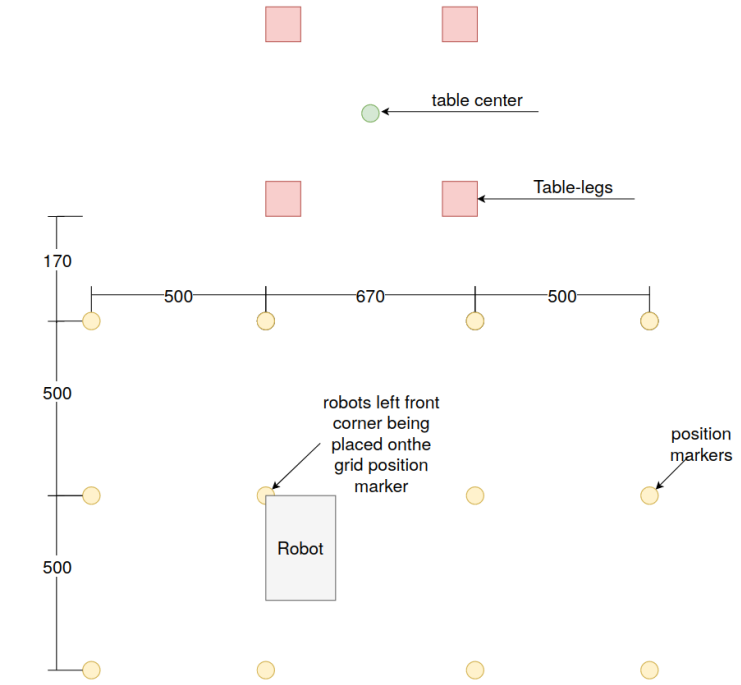

Figure 14: The position grid used for the test setup.

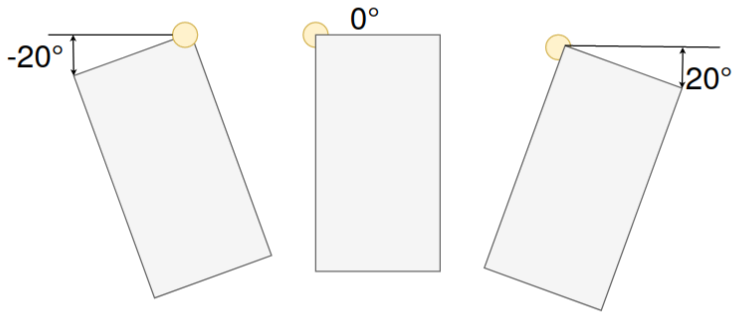

Figure 15: The three different orientation the robot was placed in for every position-marker

6.1.3 Inside Table Setup

With correct orientation θ and y-offset the robot was positioned with a specified distance from the table center. Starting from $1m$ away, in discrete $5cm$ steps a bag-file was recorded and ground-truth data was measured. In the last recording the robot-center being only $5cm$ away from the table-center.
See Figure 16.

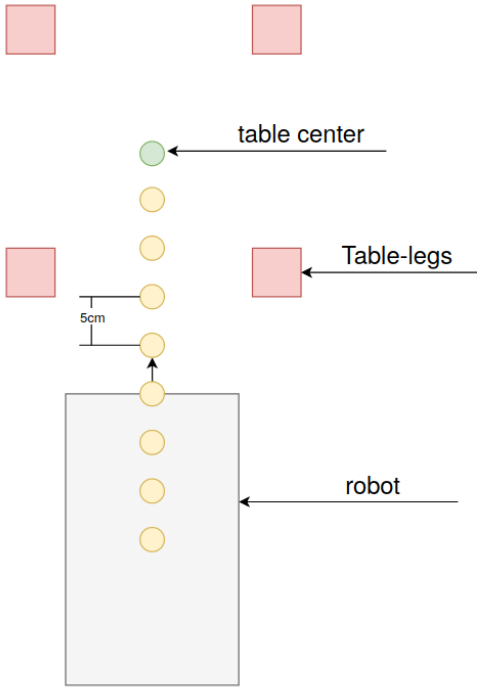

Figure 16: The test setup to measure accuracy underneath the table.

6.1.4 Results - Outside Table

In order to properly derive conclusions from the recorded data, different plots will be discussed here.

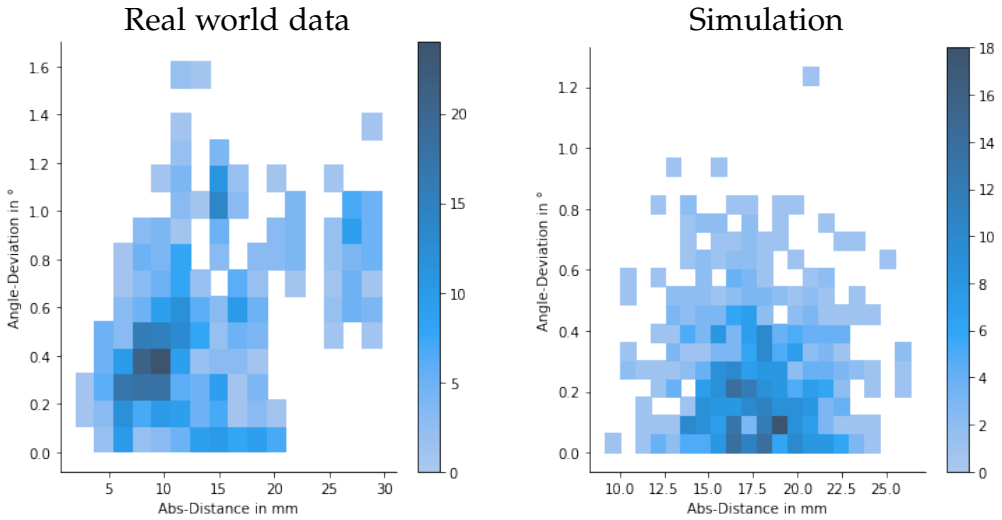

Figure 17: Outside table - localization error - comparison

Figure 17 shows the difference in distance and angle deviation error in real world and simulation with "2D-Histogram" plots. Color bars on the side of each plot show the number of points inside every bin.

One can observer that localization is significantly better in real life. Real-world data is mostly located around a distance deviation of 7.5mm. Data derived from simulation is more inaccurate and centered around a 17.5mm deviation. Interesting here is that it is apparent that simulated sensor noise follows a Gaussian distribution, as created by Gazebo. Orientation deviation is greater in the real-world and there are stronger outliers in the data.

If we take a closer look at Figure 18, we can see that the localization error decreases the closer the robot is in x and y direction in front of the table. Rotation does not effect the localization accuracy a lot.

Note: As seen in Figure 14 the robots left front corner was placed on the grid position markers, but recorded distance deviation were collected from the

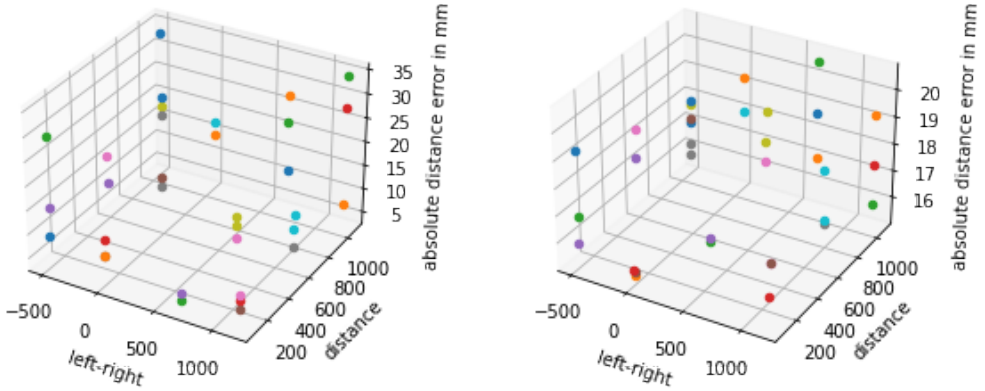

Figure 18: recorded error of all 36 different position- and orientation configurations outside the table - Real and simulation

robot's coordinate frame. Therefore Figure 18 does not have symmetrically placed points along its x and y axis.

The robot system tries to navigate towards the table's center based the on given localization. Navigation in front of the table could degenerate if localization is ever further away from the center than half a table's side length. Meaning that a localized pose always has to be inside the table's bounds. See section 3.4.
Based on presented findings we can observe that our approach works well enough to properly navigate in front of the table center.

6.1.5 Results - Inside table

The figure above shows heat plots of angle deviation in deg and y-deviation in mm. Here we can observe that most of our data is below the feasibility threshold of $7.5mm$. Here, simulation is more accurate and has less spread in contrast to recorded real world data.

As described in section 5, the robot moves towards the center if there are no laser points in its way. Whenever it receives a sensor reading that has a y offset higher than $\approx$ 3mm, the robot stalls. Therefore the robot's driving

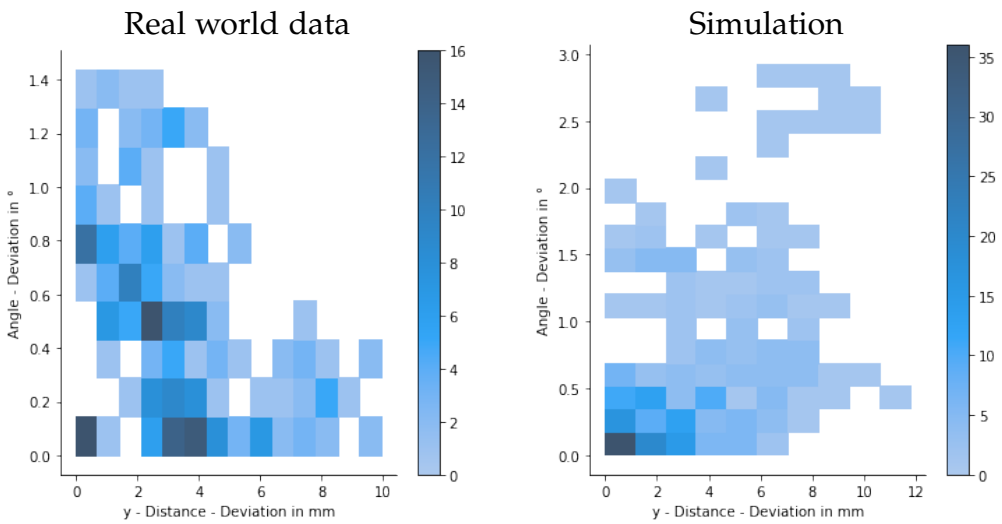

Figure 19: Inside table - localization error - comparison

speed is heavily influenced by the robustness of it's localization.

6.1.6 Scan Matching Robustness

In this section we present a brief discussion of how localization accuracy changed based on the initial guess used.

When comparing the findings, it can be seen that the data plot on the right has data centered more around one point and outliers are rather sparse. The left plot has data spread more evenly and there are generally stronger outliers. This is especially true when taking a look at the angle deviation. There, a small cluster around a 2.5 deg offset can be observed in the left plot. The one on the right does not contain any data with an angle deviation above 1.5. When taking a look at the absolute distance mean d_{mean} and the angle offset mean θ_{mean}, one can see the accuracy difference of both approaches.

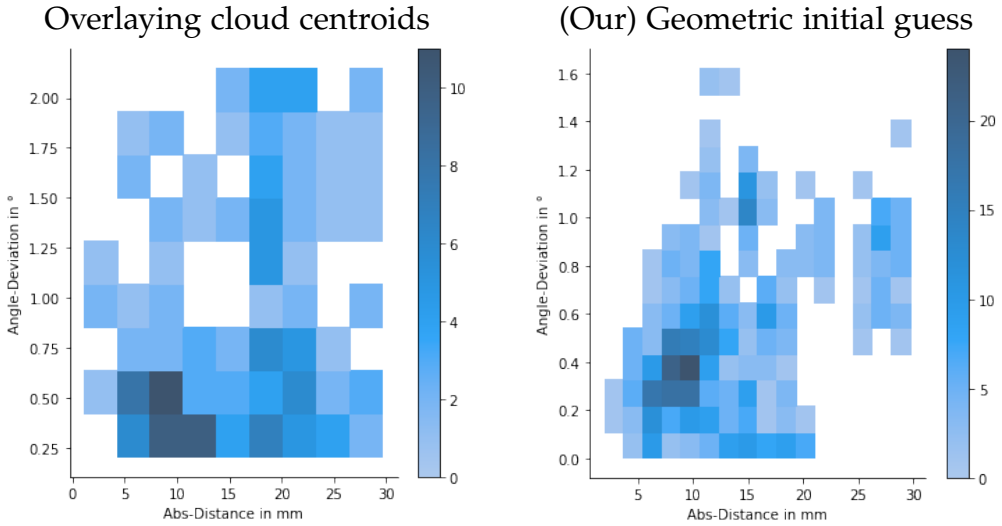

Figure 20: Error comparison of our proposed initial guess method to a standard approach

	θ_{mean}	d_{mean}
Overlaying cloud centroids	0.86 deg	$15.77mm$
Geometric initial guess	0.50 deg	$13.31mm$

6.2 Navigation

6.2.1 Test Setup

The Navigation system's goal is to successfully steer the robot towards the table center. Two major concerns here were robustness and maneuvering speed. To properly evaluate those parameters, the following test-setup was defined (See Figure 21):

The robot was placed in three different poses in front of the table. For every pose the system was run five times and it was recorded if the robot hit a table leg and how long it took to finish its navigation.

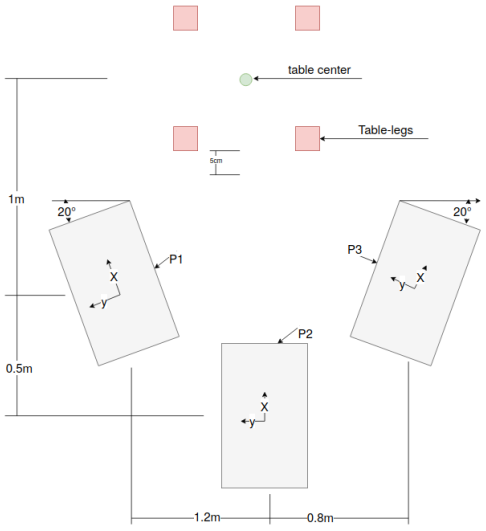

Figure 21: The navigation test setup.

The following tables show the test results for simulation (Table 2) and real world (Table 3). All data is given in seconds and was collected using our combined approach of obstacle avoidance and headless driving(See Navigation). Avoiding obstacles during the whole course of navigation resulted in a significantly longer maneuvering time, usually of around 1 - 2 min.

6.2.2 Simulation

	T_1	T_2	T_3	T_4	T_5	mean	deviation
P_1	53s	53s	53s	53s	53s	53s	0s
P_2	45s	45s	45s	45s	45s	45s	0s
P_3	52s	52s	52s	52s	52s	52s	0s

P_n Pose 1 to 3
T_n Test-run 1 to 5

Table 2: Time needed for navigation - Simulation

Most importantly, every test run succeeded by never hitting a table leg or failing due to localization errors.
As expected, navigation from P_2 was the fastest, since the robot does not have to account for rotation. Interestingly we always get the same time for every test run. This is probably due to the closed environment the simulation provides. Apart from the laser-sensor noise simulated with a Gaussian distribution, there is minimal inaccuracy or non-determinism that can impact the system's performance.

6.2.3 Real world

	T_1	T_2	T_3	T_4	T_5	mean	deviation
P_1	35s	22s	36s	48s	45s	37.2s	10.18s
P_2	35s	32s	40s	38s	36s	36.2	3.03
P_3	82s / Fail	45s	50s	50s	50s	55.4s	15.02s

Table 3: Time needed for navigation - Real World

Navigation succeeded every time besides T_1/P_3. In that particular case the localization diverged, leading to the system failing. After a restart of the localization node everything worked.

Unsurprisingly every test run took a different amount of time. Besides non-determinism, a lot of the differences are probably because always placing the robot on exactly the same pose is not possible in real world.

During most of the test runs our robot slightly touched the upper left table leg. This was probably because localization has a higher mean angle deviation in real world scenarios. Towards reaching the center the robot only drives straight. If the robot has a to large θ offset when entering the "only-drive-forward" state this can result in getting a little to far of on one side.
Fine tuning localization on real world data could improve this problem.

6.2.4 Improvements

Although the overall system works rather robust in simulation and real world, navigation could still be improved. A process time of 30s or more is still quite long.

For a navigation system to work for this task, it would have to continuously readjust its motion and not accumulate any point cloud data from the environment. This is necessary because consistently occurring outliers in the point cloud data would then block navigation completely. As discussed in Section 5, local planners like "Time-Elastic-Band Planner" [RHB15] are therefore not suited for this. Additionally, during the whole course of navigation every table leg is located in one of the robot's blind spots at some point. For this reason, approaches like "Vector Field Histogram" – Navigation [BK91], will not work, because driving towards the largest gap in the environment only works if all obstacles are visible.

A working solution could be to calculate a complete trajectory in front of the table and then follow a mental model while ignoring sensor readings related to the table.
Due to effects such as wheel slippage, global localization using odometry information alone accumulates error and is therefore not accurate enough.

The fine navigation sensor on the bottom of the EvoRobot could have provided enough local accuracy to make this work. However, the goal of this thesis was to develop a functional navigation system without using this sensor.

7 Conclusion

In this thesis we showed the development of a system that lets the EvoRobot shuttle robot navigate underneath extremely narrow machining tables, regardless of the robot's starting pose. Filtering was used to constrain what the robot perceives to only one table at a time. Localization of the table center was done using scan-matching. We showed how ICP can be used in conjunction with a hand-crafted template. If the target for scan-matching does not change, a geometrically crafted initial guess can heavily improve localization accuracy.

Since localization still contained some outliers, we therefore implemented a "drive-if-you-can" approach. Towards reaching the table center, simply driving in a straight line while ignoring sensor data showed to improve maneuvering time significantly. Therefore a combined navigation approach was used.

Evaluation showed the system's robustness. However, the overall navigation speed is still rather slow compared to what it could be. Possible improvements of the system's components were discussed.

References

[BK91] Johann Borenstein and Yoram Koren. "The Vector Field Histogram - Fast Obstacle Avoidance For Mobile Robots." In: *Robotics and Automation, IEEE Transactions on* 7 (July 1991), pp. 278–288. DOI: 10.1109/70.88137 (cit. on p. 38).

[Rus+07] Radu Rusu et al. "Towards 3D Object Maps for Autonomous Household Robots." In: Oct. 2007, pp. 3191–3198. DOI: 10.1109/IROS.2007.4399309 (cit. on p. 13).

[SHT09] Aleksandr Segal, Dirk Hähnel, and Sebastian Thrun. "Generalized-ICP." In: June 2009. DOI: 10.15607/RSS.2009.V.021 (cit. on pp. 18, 22).

[RC11] Radu Rusu and Steve Cousins. "3D is here: Point cloud library (PCL)." In: May 2011. DOI: 10.1109/ICRA.2011.5980567 (cit. on pp. 10, 14, 15).

[Bal+14] Augusto Ballardini et al. "ira-laser-tools: a ROS LaserScan manipulation toolbox." In: (Nov. 2014) (cit. on p. 11).

[RHB15] Christoph Rösmann, Frank Hoffmann, and Torsten Bertram. "Timed-Elastic-Bands for Time-Optimal Point-to-Point Nonlinear Model Predictive Control." In: July 2015. DOI: 10.1109/ECC.2015.7331052 (cit. on pp. 23, 38).

[Sta18] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org (cit. on p. 10).

[Evo22] EvoCortex. *EvoRobot Industrial*. 2022. URL: https://evocortex.org/products/evorobot/industrial/ (visited on 06/27/2022) (cit. on p. 7).

[Str22] Daniel Strametz. *Autonomous reconfiguration of mobile workstations*. 2022. URL: https://www.smartfactory.tugraz.at/de/aktivitaeten/aktuelle_projekte/autonome_rekonfiguration_von_mobilen_arbeitsstationen_/ (visited on 06/27/2022) (cit. on pp. 3, 6, 8).