APPENDICES

CONTENT

I. MOTION MODEL

After each motion the set of particles used in localization has to be moved, this is done with the odometry measurements of the robot and the kinematic model of the AMR:

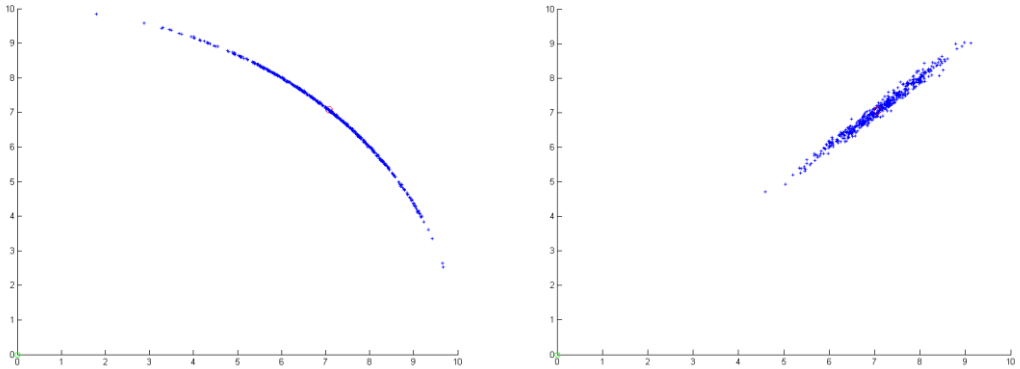
$$P' = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \Delta s \cdot \cos(\theta + \Delta\theta + \Delta\varphi/2) \\ \Delta s \cdot \cos(\theta + \Delta\theta + \Delta\varphi/2) \\ \Delta\theta + \Delta\varphi \end{bmatrix}$$

Where P' is the updated position, Δs is the distance traveled by the robot according to the odometry sensors, $\Delta\theta$ is the angle turned without moving and $\Delta\varphi$ is the angle turned during a circular motion according the odometry sensors. Because Monte Carlo Localization is a probabilistic model, the motion model has to be probabilistic as well. This algorithm is called a sample motion model and is implemented as described in (Thrun, Burgard, & Dieter, 2006, p. 136) . This model accepts an initial position x_{t-1} and an odometry measurement u_t and outputs a random x_t distributed according to $p(x_t|u_t, x_{t-1})$. The estimated position for all particles is guessed and this is done according to the algorithm in Pseudo code 2.

Pseudo code 2. Algorithm for sampling from $p(x_t|u_t, x_{t-1})$ based on odometry measurements

1: $\boldsymbol{Algorithm\ sample\_motion\_model\_odometry}(u_t, x_{t-1})$:
2: $\delta_{rot1} = atan2(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
3: $\delta_{trans} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$
4: $\hat{\delta}_{rot1} = \delta_{rot1} - \boldsymbol{sample}(\alpha_1\delta_{rot1}^2 + \alpha_2\delta_{trans}^2)$
5: $\hat{\delta}_{trans} = \delta_{trans} - \boldsymbol{sample}(\alpha_3\delta_{trans}^2 + \alpha_4\delta_{rot1}^2)$
6: $x' = x + \hat{\delta}_{trans}\cos(\theta + \hat{\delta}_{rot1})$
7: $y' = y + \hat{\delta}_{trans}\, sin(\theta + \hat{\delta}_{rot1})$
8: $\theta' = \theta + \hat{\delta}_{rot1}$
9: return $x_t = (x', y', \theta')^t$

In this algorithm a_1, a_2, a_3 and a_4 are noise parameters to model the error on the odometry, changing these parameters will change the behaviour of the particles when moved as can be seen on Figure 20.



A) High rotation error and low translation errror B) High translation error and low angular error

Figure 20. Plots showing impact of different parameters with start position = (0,0), step size = 10 and rotation step = 45 degrees. The noise added on these figures is Gaussian, but can be replaced by other types of noise.

II. SENSOR MODEL

The sensor model was shortly introduced in section 4.2 and is used to add weights to particles based on the probability that a measurement is correct: $p(z_t|x_t, map)$. This sensor has been implemented as described in (Thrun, Burgard, & Dieter, 2006, p. 153).
The probability of a sensor reading matching the state of a particle is modelled as a probability density function which is a sum of four different probability density functions and will be described separately:

- Correct range with local measurement noise
- Unmapped objects
- Failures causing maximum distances
- Random measurements

Correct range with local measurement noise

An ideal sensor measurement would be the exact distance from the sensor to the nearest object. However no such sensor exists and each sensor has imperfections due to its resolution or due to the environment it works in. This is why measurement noise is modelled by a narrow Gaussian with mean z_t^{k*} and standard deviation σ_{hit}. z_t^{k*} is considered the "true" distance to the object and is coming from the ray-casting process. σ_{hit} is the parameter which models the magnitude of the noise due to the reasons above. The values measured by the range sensor are limited to the interval $[0; z_{max}]$ where z_{max} is the maximum sensor range. The measurement probability is given by:

$$p_{hit}(z_t^k|x_t, m)ap = \begin{cases} \eta\mathcal{N}(z_t^k; z_t^{k*}, \sigma_{hit}^2)\ if\ 0 \le z_t^k \le z_{max} \\ 0\ otherwise \end{cases}$$

z_t^{k*}, as said above is calculated from x_t and map m by ray casting, $\mathcal{N}(z_t^k; z_t^{k*}, \sigma_{hit}^2)$ is a univariate normal distribution with mean z_t^{k*} and standard deviation σ_{hit}, an example of this distribution is given on Figure 21:

$$\mathcal{N}(z_t^k; z_t^{k*}, \sigma_{hit}^2) = \frac{1}{\sqrt{2\pi\sigma_{hit}^2}} e^{\frac{-1}{2}\frac{\left(z_t^k - z_t^{k*}\right)^2}{\sigma_{hit}^2}}$$

η is a normalizer which is given by:

$$\eta = \left(\int_0^{z_{max}} \mathcal{N}(z_t^k; z_t^{k*}, \sigma_{hit}^2) dz_t^k\right)^{-1}$$

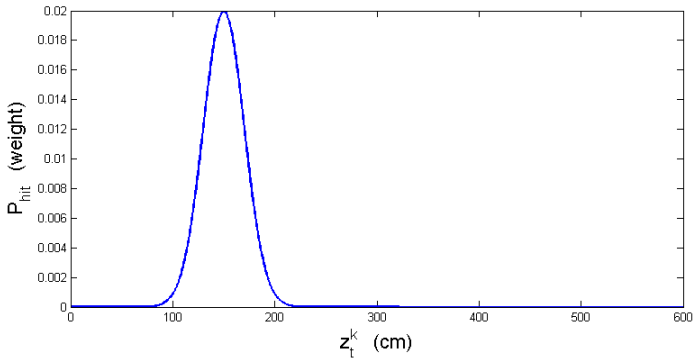


Figure 21. Correct hit distribution for a measurement of 150cm with added noise

Unmapped objects

The map used to ray-cast is static, while environments in which robots operate are dynamic due to unexpected objects. This will lead to a measurement lower than expected compared to the map. The likelihood of sensing an unexpected object decreases with range, this is why mathematically the probability of range measurements is described by an exponential distribution. This is because if two obstacles are in line, only the closest one gets detected. The parameters of this distribution is λ_{short} and is a model parameter and should be tuned depending on the amount of unmapped objects in the environment. The probability can be described as:

$$p_{short}(z_t^k|x_t, map) = \begin{cases} \eta\lambda_{short} e^{-\lambda_{short} z_t^k}\ if\ 0 \le z_t^k \le z_t^{k*} \\ 0\ otherwise \end{cases}$$

η again is a normalizer which is given by:

$$\eta = \frac{1}{1 - e^{-\lambda_{short} z_t^{k*}}}$$

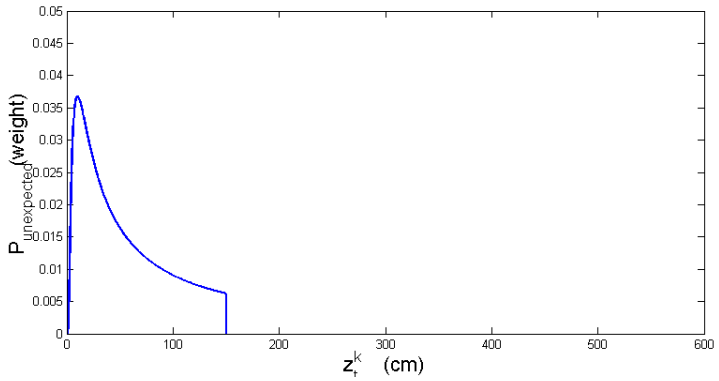


Figure 22. Unexpected hit distribution for a measurement of 150cm

Failures causing maximum distances

These failures have not been modelled in our implementation as they are filtered before they the localization. These failures occur for example when the laser measures a black object which the laser system is unable to detect and then the laser would return a maximum measurement, the sensor used in this project doesn't return any measurement due to high uncertainty. These failures would be modelled as:

$$p_{max}(z_t^k|x_t, map) = I(z = z_{max}) = \begin{cases} 1 \; if \; z = z_{max} \\ 0 \; otherwise \end{cases}$$

Random measurements

Range finders occasionally produce unexplainable measurements, these happen more often with sonar sensors but occur with laser sensors as well. These measurements are modelled using a uniform distribution spread over the entire sensor range $[0, z_{max}]$ as shown on Figure 23.

$$p_{rand}(z_t^k|x_t, map) = \begin{cases} \frac{1}{z_{max}} \; if \; 0 \leq z_t^k < z_{max} \\ 0 \; otherwise \end{cases}$$

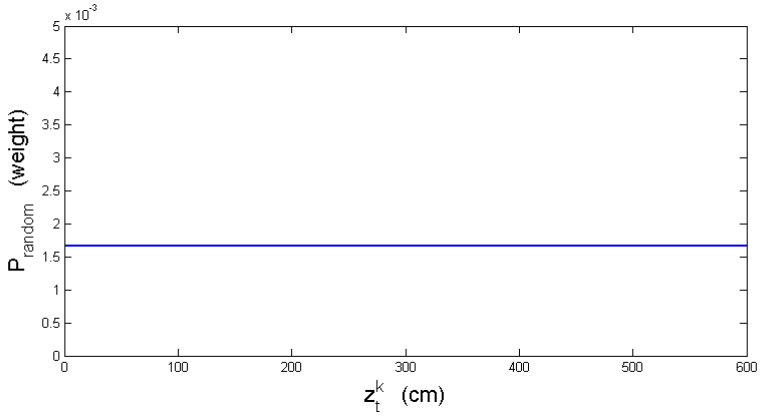


Figure 23. Random distribution

Total

The four different distributions are now mixed by a weighted average defined by parameters $z_{hit}, z_{short}, z_{max}$ and z_{rand} with $z_{hit} + z_{short} + z_{max} + z_{rand} = 1$. This results in the distribution are shown in Figure 24.

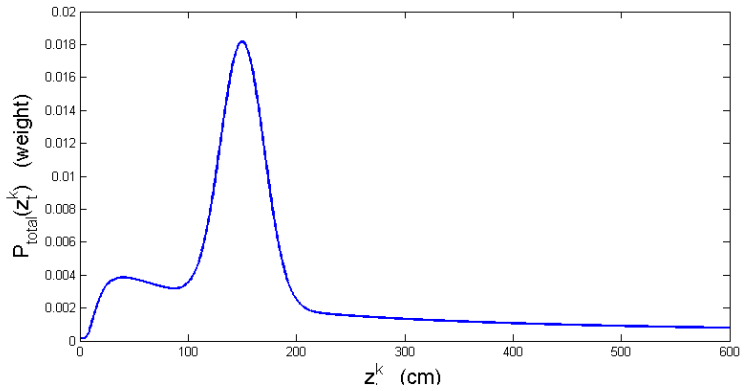


Figure 24. Total distribution of the sensor model

For each measurement the following algorithm is used to calculate the total probability:

Pseudo code 3. Sensor model algorithm for calculating the total probability for a particle assuming conditional independence between measurements.

1: $\boldsymbol{Algorithm\ beam\_range\_finder\_model}(\boldsymbol{z_t, x_t, m})$:
2: $q = 1$
3: $for\ k = 1\ to\ K\ do$
4: $\quad compute\ z_t^{k*}\ for\ the\ measurement\ z_t^k\ using\ ray\ casting$
5: $\quad p = z_{hit} \cdot p_{hit}(z_t^k | x_t, map) + z_{short} \cdot p_{short}(z_t^k | x_t, map) +$
6: $\qquad z_{max} \cdot p_{max}(z_t^k | x_t, map) + z_{rand} \cdot p_{rand}(z_t^k | x_t, map)$
7: $\qquad q = q \cdot p$
8: $\quad return\ q$

III. RAY-CASTING PROCESS

Ray-casting simulates range measurements for robots with a range sensor in a known environment, this step is not only used in Monte Carlo Particle filters but also in Extended Kalman Filters and localization. The inputs needed for this process are the robot's position, map its environment as an occupancy grid map, range of the range sensor and the amount of measurements to simulate (Shrestha, 2012). The process goes as explained in Pseudo code 4:

Pseudo code 4. Ray-casting algorithm for a single position.

```
1: Algorithm range_casting(X_j, m, r, n_m):
2: if Xj = in_obstacle
3:     range = 0
4: for each measument
5:     create ray with range r
6:     for ray 1 to n_m
7:         keep (rayElement < size(m) and rayElement > 0)
8:         if rayElement in obstacle
9:             rayElement = Y
10:            break
11:    endfor
12: calulateDist(X_j, Y)
```

This code is executed offline because this process is so computationally expensive. Because this process is so computationally expensive. This means that for each grid of the occupancy grid map this process is done with an angular resolution of 1 degree. The trade-off for the increased speed is the generation of very big matrices (e.g. in our project one floor has a matrix size of 1000x1000x360 rays). **Figure 25**Figure 25 shows the ray-casting process for a single point with an angular resolution of two degrees or 180 rays.

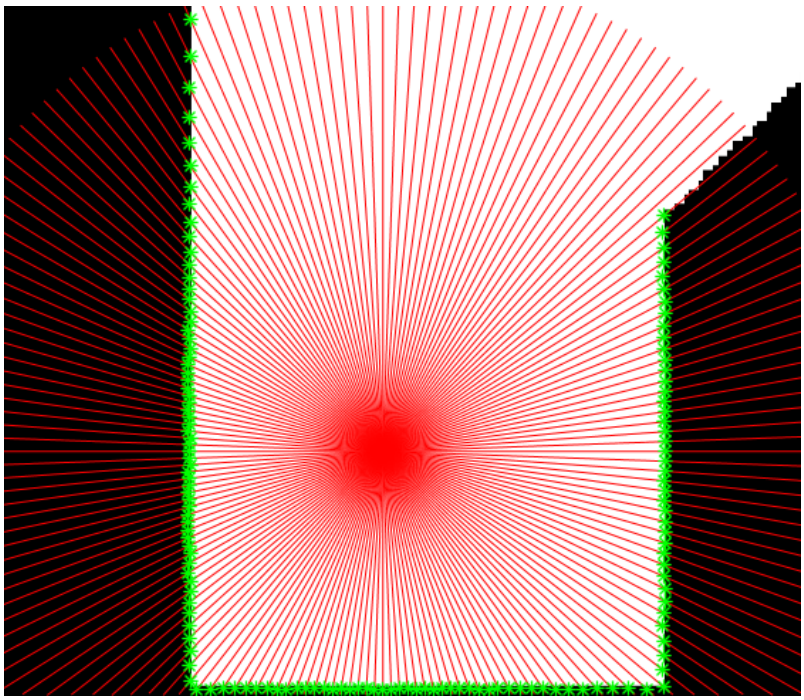


Figure 25. Detailed representation of the ray-casting process, in this figure an angular resolution of two degrees is obtained. The distance from the green crosses to the central point is calculated for each of these rays.

IV. RESAMPLING

One of the weak points of particle filters is that after a few iterations there is a depletion of population, this means that most of the particles have drifted far enough from the true position for their weight to be relevant. Resampling makes sure the computational effort is focused on the areas which are the most relevant. There are multiple ways to do this resampling process but the premises for algorithms is the same, particles with high weights are going to be duplicated and particles with low weights are eliminated (Rekleitis, 2002). This process can be done when the amount of useful particles has gotten to small. In this project systematic resampling is used, because it has lower computational requirements than other methods (Salmond & Gordon, 2005), a basic version MATLAB code is written out in Pseudo code 5:

Pseudo code 5. MATLAB code of systematic resampling algorithm from (Salmond & Gordon, 2005).

1: $\boldsymbol{Algorithm\ resampling}(\boldsymbol{X_{prior}}, \boldsymbol{weights}, \boldsymbol{n_{Samples}})$:
2: $addit = 1/nsamples;\ stt = addit * rand(1);$
3: $selection\_points = [\ stt : addit : stt + (n_{Samples} - 1) * addit\ ];\ j = 1;$
4: $for\ i = 1{:}\, n_{Samples}$
5: $\quad while\ selection\_points(i) >= weight(j);\ j = j + 1;\ end;$
6: $\quad x\_post(:, i) = x_{prior}(:, j);$
7: $endfor$

In systematic resampling the normalized weights are incrementally summed to form a cumulative sum of the weights. A "comb" is spaced at regular intervals of 1/N and the complete comb is translated by an offset chosen randomly from a uniform distribution over [1, 1/N]. There is only one random parameter, where as other algorithms often calculate multiple random parameters. The comb is then compared with the cumulative sum of the weights, this process can also be seen on Figure 26 (Salmond & Gordon, 2005).

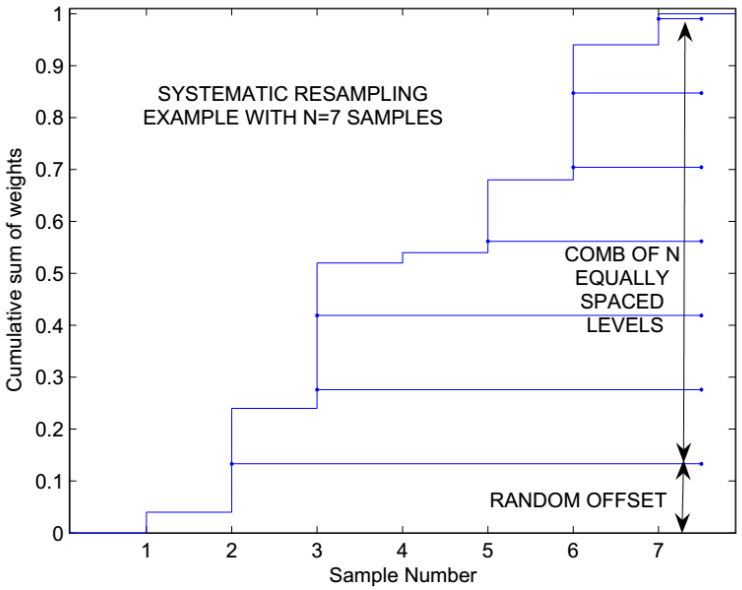


Figure 26. Figure illustrating systematic resampling (Salmond & Gordon, 2005)

V. PARAMETER SELECTION

During the experiments localization was tested and the different parameters were optimized. There are a number of different parameters which had to be optimized in the localization algorithm. These will now be explained and how they were adjusted.

Amount of particles and measurements

These two parameters decide for a big part the accuracy and robustness of the system, the more particles used in the localization the higher the amount of samples taken from the PDF and thus the higher the chance that the exact location can be found. The trade-off of course is speed, we found experimentally that a set of 500 is sufficient for this project.
All measurement retrieved from the sensor could be used for localization, this however would lead to a very slow localization, this why a number of measurements is chosen. An added advantage of this method is that these measurements are less susceptible to correlated noise since they aren't adjacent to each other. A higher amount of measurements means the system is more robust and less dependent on a single measurement which might be flawed or if a large number of measurements is blocked by obstacles it has a higher chance of localizing itself. Again a trade-off has to be made, in our project we chose to have 36 measurements spread evenly across all measurements. In an ideal scan 360 measurements would be returned, meaning that we use a measurement every 10 degrees.

Map Resolution

Map resolution plays an important factor in the localization algorithm as it defines the accuracy of the ray-casting process and thus the accuracy of the localization. During our experiments we tried both a map with a high resolution with a grid of one centimeter and a grid of 5 centimeter. The results of the 5 centimeter map were sufficient and made the ray-casting process 25 times faster at the cost of a bit of accuracy. An added advantage of using a coarse map resolution is that less data has to be stored thus having lower memory requirements. Even coarser grids can be used successfully.

Motion model noise

As explained in appendix I there are four different parameters which can change the noise characteristics of the motion model. These parameters have to be adjusted with a number of influences in mind:

- The slope of the floor and whether the AMR is going up or down
- The material and slipperiness of the floor
- Speed of motion
- Quality of odometry encoders

With these influences in mind the only way to reliably determine these parameters was to do a large number of tests and change the parameters until the results were satisfying. The reprogramming of the odometry encoders improved the odometry a lot and allowed us to lower the amount of noise drastically.

Sensor model parameters

Another set of parameters to determine are the constants explained in in appendix III these constants can be determined with an algorithm that determines these parameters automatically. (Thrun, Burgard, & Dieter, 2006, p. 159) In our project the results were satisfying even by adjusting these parameters by hand so there was no need to implement this algorithm.
These parameters are influenced by:

- Map resolution
- Surface material of the walls
- Amount of unexpected objects
- Amount of sensor noise

VI. PREPARING MAPS FOR ROBOTICS

There aren't many practical guides available on how to make an occupancy grid map from a real map of a building. This is why we decided to add a short guide on how to go from CAD map to an occupancy grid map in monochrome bitmap format. The idea is to give every pixel on the BMP image a corresponding dimension in reality. In the end of this appendix there is also an explanation on how to get coordinates from polygons which is used in our project in the path planning chapter.

Requirements

- 2D CAD drawing program such as DraftSight or AutoCAD
- CAD files in a format which is accepted by your CAD program
- Image editing software such as Microsoft Paint, Paint.net or Photoshop

Preparing and edit the CAD maps

CAD maps of buildings are often made by architects who do not make these with robotics in mind. Therefor it is important to do the following things:

1. Select all layers and move them to the origin (MOVE cmd)
2. Scale all layers with the correct scale in mind (Scale 0.2 makes every pixel 5 times the original dimension)
3. Create a new layer to work on and use the polyline command to trace the boundaries of the location the robot has to function in.
4. Hide all the layers except the one that was made with the polylines and make sure the layer only has the newly added polyline and this line is colored black.

Creating and editing the image file

This option will vary depending on what software is used, a lot of information can be found on the internet about this. But in AutoCAD the option is found in the print menu, we recommend printing the file to a PNG file as this gave the best results which makes the following steps faster. It is important to print the drawing with scale 1:1 so each pixel corresponds correctly to the wanted dimension, this might require you to add a custom page size which is big enough for the drawing.
Editing the image can be done in a lot of different software but we found that Microsoft Paint works pretty well. The following things have to be checked:

1. Crop the image so that the outer edges of the drawing are the outer edges of the image, this is especially important at the origin. If done incorrectly an offset is created.
2. Make sure the polyline is closed in the image, sometimes especially at sharp corners these edges are not fully closed.
3. Fill the areas which are obstacles with the "bucket tool"
4. Save the image as a monochrome bitmap

Getting coordinates from maps

Getting coordinates from a CAD map in an automated way can be useful, in a room with only straight lines this task can be done relatively quickly by hand, but in a building with a lot of curved lines this task would be way too intensive to do by hand. We made use of a semi-automated way to generated points on curved lines and to export them to an Excel file. To do this we made use of two AutoCAD scripts:

- Polyfit from http://www.polyface.de/ allows the user to select a polyline, specify a distance between two nodes and creates a polygon that fits the (curved) polyline.
- PTexp which is a code made by Lee Mac from http://lee-mac.com/, it allows you to select an object to export the coordinates of every corner to a .CSV file which works with both Excel and MATLAB.

VII. CYCLE TIME MAIN FILE

This appendix analysis the cycle time of the main loop during the navigation of the AMR. The cycle time is divided in 4 parts: onboard sensors, LIDAR, localization and navigation. Note that the path planning is done beforehand and thus not shown here. Each part is performed once unless specified.

Total Cycle time

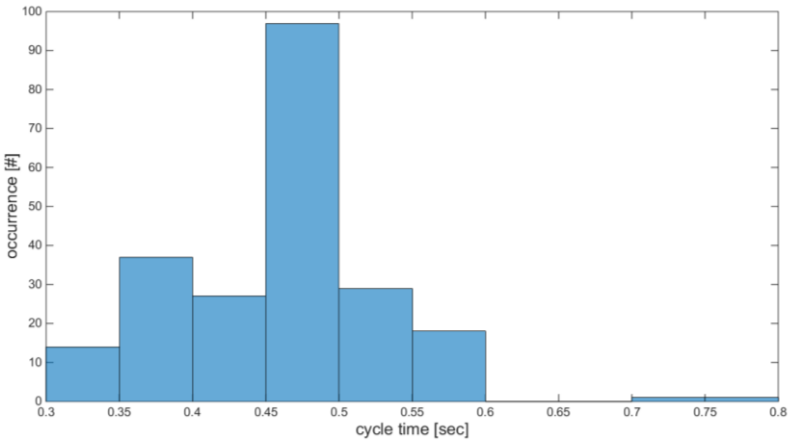


Figure 27. Total cycle time of the navigation algorithm. This cycle only contains LIDAR and sensor fetch time, localization and obstacle avoidance. Except for two long cycle times, this algorithm arrives at a cycle frequency of 2 Hz.

Onboard sensors

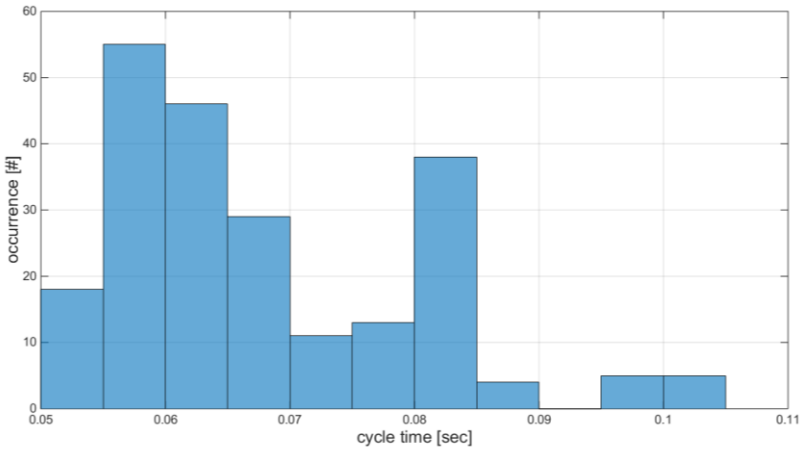


Figure 28. The cycle time to fetch the onboard sensors data is around 0.06s. The following onboard sensors are pulled: encoders, bump and drop. This is performed twice each cycle.

LIDAR

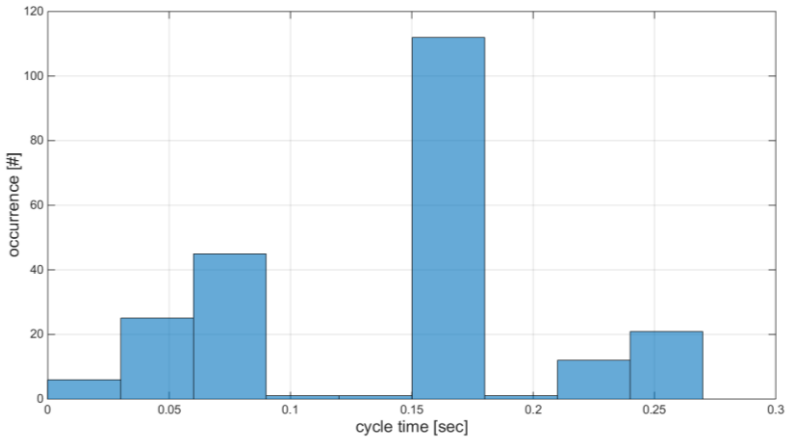


Figure 29. The LIDAR fetch cycle time is around 0.15 sec.
This cycle time is expected, since the LIDAR sends data at a rate of $\pm 5.5 Hz$.

Localization

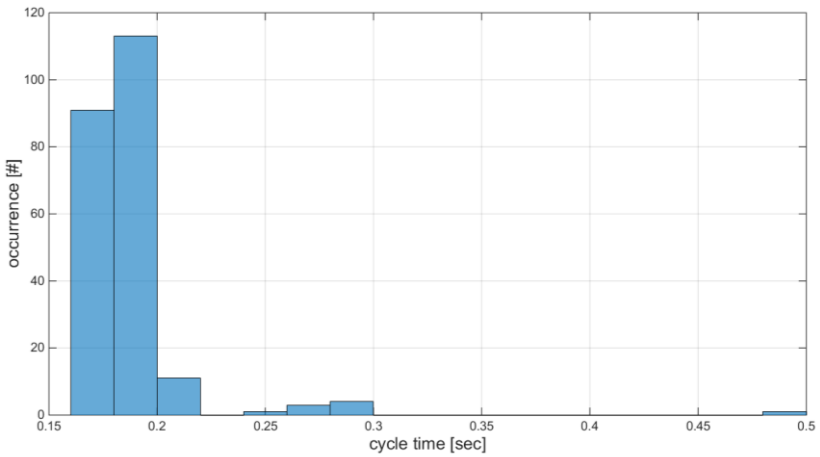


Figure 30. The localization cycle time is around 0.2 sec, this is an acceptable results for a computationally heavy algorithm. The most demanding step of the localization algorithm is the step where the real measurements are compared with the simulated ones according to the sensor model.

Navigation

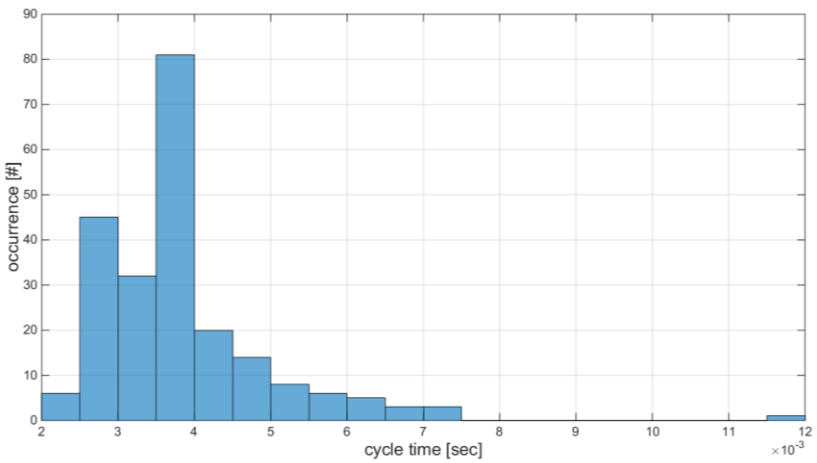


Figure 31. This time is nearly negligible, because of the low computational cost of the OA algorithm.

VIII. HOW TO SEND COMMANDS AND RECEIVE DATA FROM THE IROBOT ROOMBA

The goal of this step by step guide is to show the user how to connect the Roomba with a computer (Windows and Linux), and how to send and receive basic commands, using the Roomba MATLAB Toolbox. This toolbox can be found in the folder "Roomba". Several changes had to be made on the original toolbox, due to the version change (see section 2.2) and multi-platform adaptation (Linux and Windows). A few additional functions were also added. The user is expected to have the right USB cable to connect the Roomba to the computer. If this isn't the case, we refer to the book "Hacking Roomba" (Kurt, 2006, pp. 41-63).

1. Connect the Roomba to the computer
2. Find the port used by the Roomba
    - For Linux :
        - Launch terminal
        - ls /dev/ttyUSB*
        - normally, the location of the Roomba is in the following format /dev/ttyUSB0
    - For Windows :
        - Computer
        - Control Panel
        - View devices and printers
        - Here, find the COM* who are active
3. If both LIDAR and Roomba is connected, the user has to try both serial ports, to find the right one.
4. Initiate the connection with RoombaInit(COM). Normally the Roomba should beep.
5. Use the serial port reserved from RoombaInit (by default, serPort), to send commands.
6. You can now use the AllSensorsReadRoomba(serPort) to receive all the sensor data of the Roomba
7. You can now use SetFwdVelRadiusRoomba(serPort,v,r) to set the velocity (v), and radius (r) of the Roomba

IX. HOW TO SEND LIDAR DATA OVER UDP (USING LINUX MACHINE)

The goal of this step by step manual is to inform the reader how to correctly install and run the C++ code that will control and send the LIDAR measurement over UDP. The C++ code is made for a Linux machine, compatibility with a windows client is possible but has not been tested. A windows library of "pthread" (or equivalent to it) has to be found by the Windows user. The LIDAR source code, can be found on the CD with this thesis, under the folder name "LIDAR".

C++ project settings (using Eclipse)

8. Install from Eclipse
9. Unzip C++ code
10. Create a C++ project in Eclipse
11. Drag the unzipped files in the project
12. Add the necessary paths in eclipse.
    - right click project
    - C/C++ build
    - Settings
        - GCC C++
            - Include, add the following paths :
                - sdk/include
                - sdk/src
                - skd/src/arch
                - sdk/src/hal
13. Add the necessary libraries :
    - right click project
    - C/CC++ build
    - Settings
        - GCC C++
            - Linker libraries
                - pthread
14. Build
15. Left click on arrow next to run
    - Run configurations
    - new launch configurations
    - Arguments → /dev/ttyUSB0 (for example)

How to allow Eclipse to connect to LIDAR

1. sudo gpasswd --add ${USER} dialout

Important terminal commands:

To find the LIDAR COM port:

1. Launch terminal
2. ls /dev/ttyUSB*
3. normally, the location of the LIDAR is in the following format /dev/ttyUSB0
4. If both LIDAR and Roomba is connected, the user has to try both serial ports, to find the right one.

X. HOW TO RUN THE MAIN FILE

1. Follow the steps described in appendix VII and IX
2. Run the main file (note: the first time to run the code can take some time, because MATLAB has to load all the maps of the different floors)
3. Enter the start module in the MATLAB command (0-14)
4. Click on the map that appeared, the start position and orientation of the robot.
5. Enter your destination module in the MATLAB command (0-14)
6. Click on the map that appeared, your desired destination position.
7. The AMR will now calculate the optimal path. This will take up to 20 seconds (if you go from module 0 to 14)
8. Once the AMR starts moving, follow it! The AMR will tell you, when you are at 25%, 50% and 75% of the way, and when you arrived at your destination.

If an error occurs …

Sometimes connection problems occurs if the serial port of the Roomba is not shut correctly. If no connection to the Roomba can be made in the next run (no "beep" during initialisation), the serial cable connected to the Roomba has to be disconnected from the pc and reconnected after at least 3 seconds. Then, normally the COM port will change, this has to be changed in the template routine, according to the procedure described in appendix VII.
If the COM port doesn't change after several attempts of disconnecting and reconnecting, this means that the Roomba has a low battery and had to be charged.

How to hard reset the Roomba

If the Roomba is stuck in while its actuators are running, the best way to shut it down, it to manually hard reset the Roomba. This is done by taking the battery off the Roomba.

XI. MATLAB CODE

In this appendix a short description is given of the functions, the project is structured into maps. Each map has its own subject and contains the functions that are used to make that part function. All these functions are connected in the Main file as explained in Appendix X. A more detailed description of each individual function is given in the function's code.

Localization

The Localization folder contains all the files required to let the AMR localize itself, each step of the localization is separated into a different function. This folder also contains the simulated measurements which are a result of the ray casting process.

Navigation

The navigation directory contains all the functions needed to perform the PP and the OA.

Jukebox

The Jukebox map holds the files that provide the audio feedback to the user.

Maps

These map contain all the coordinate data of the obstacles of each floor, for each of the floors this data gets initiated with a floor specific function. Also an image of the floor can be found which is used for plotting purposes. This map also contains the map changing algorithm which is used in the multi-floor building.

Roomba

The Roomba folder contains all the function that interact with the Roomba. These functions comes mainly from the Roomba toolbox (Esposito, Barton, Koehler, & Lim, 2011).

Lidar

The LIDAR directory contains all the functions necessary to retrieve the sensor measurement from UDP.

MISC

A few cross-directory functions are written here

XII. CD-ROM CONTENT

This appendix describes the content of the digital appendix stored on the included CD-ROM

CAD Drawings
In this directory can be found the files used to reproduce or extend the platform.

CAD Maps
This directory has the original CAD maps of our university campus and modified files used in this project.

LIDAR Files
This map contains the files needed to control the LIDAR sensor and send the retrieved data via UDP

MATLAB Code
This map has all the MATLAB code described in Appendix XI.

Videos
This map encloses a link to a personal YouTube channel, containing videos taken of successful experiments.