KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

CAMPUS GROUP T LEUVEN

AUTONOMOUS MOBILE ROBOT NAVIGATION USING THE IROBOT ROOMBA

Kevin DENIS
Alexander MEYNEN

Supervisor: Prof. dr. ir. Peter Slaets

Co-supervisor: Prof. dr. Luc Vandeurzen

Master Thesis submitted to obtain the degree of Master of Science in Engineering Technology: *Master of Industrial Sciences: Electromechanical Engineering*

Academic Year 2014-2015

Master of Industrial Sciences 2014- 2015
Faculty of Engineering Technology, Campus Group T Leuven
This paper is written by students in the framework of a Master's Thesis

AUTONOMOUS MOBILE ROBOT NAVIGATION USING THE IROBOT ROOMBA

Kevin Denis[1], Alexander Meynen[1]

[1] Master student Electromechanical Engineering, Faculty of Engineering Technology,
Campus Group T Leuven
Vesaliusstraat 13, 3000 Leuven, Belgium

Supervisor: Peter Slaets
Faculty of Engineering Technology, Campus Group T, Leuven
Vesaliusstraat 13, 3000 Leuven, Belgium
peter.slaets@kuleuven.be

Co-supervisor: Luc Vandeurzen
Faculty of Engineering Technology, Campus Group T, Leuven
Vesaliusstraat 13, 3000 Leuven, Belgium
luc.vandeurzen@kuleuven.be

ABSTRACT

This thesis offers a description on how to select and implement different algorithms which gives an existing robot the ability to localize itself, plan a path to a destination and follow this path in a multi-floor building while avoiding obstacles. The purpose of this Autonomous Mobile Robot (AMR) is to guide visitors of our university campus to a desired room. To get information about the environment, the iRobot Roomba, the selected robot platform, has been equipped with a LIDAR sensor. The measurements gathered from the LIDAR are used for the Vector Polar Histogram algorithm which is selected as obstacle avoidance navigation method, as well as the Monte Carlo Particle filter which is used to localize the robot in its environment. Other inputs for the localization are the map of the building in which the robot has to function and the data from the onboard odometry sensor measurements. A Voronoi diagram is used to determine the possible paths to the destination and Dijkstra's algorithm is used to select the most optimal path. Combining all these methods results in a robust and accessible AMR, which is able to navigate in our university campus. The methods developed for autonomous navigation are however not specifically based on our campus or robot, which creates the opportunity for future uses.

Keywords

Autonomous Mobile Robot, iRobot Roomba, Localization, Obstacle Avoidance, Path Planning

1 INTRODUCTION

Indoor AMRs have numerous applications: wheelchair robots (RADHAR, 2015), household robots (Tse, Lang, Leung, & Sze, 1998) and museum tour-guides (Burgard, et al., 1998), to name just a few. All these applications have one common problem that needs solving, how to navigate in a smart way from one location to another without collisions. In this paper, the process of selecting and implementing the algorithms needed to design an AMR from an existing robot is described. In this application an iRobot Roomba has to be able to navigate without any collisions in Group T, our university campus, which is a multi-floor building, as can be seen in Figure 1. A possible application of the robot would be to assist visitors who are new to the building and lead them to a specific room. To make the robot deliver such a service effectively on a day to day basis, it is important that it is user-friendly. This is done by giving the robot the ability to communicate with the user and report useful information, such as progress made and when the destination is reached. The robot is also able to go to its charging station autonomously when required.

Figure 1. Picture from the university campus of Group T. All class rooms are connected with each other, without the need to take stares. This makes it particularly well adapted for mobile robot applications.

To make this project successful three main problems have to be solved: path planning, obstacle avoidance and localization. Path planning methods give an AMR the ability to get to its destination in an efficient and safe way. Obstacle avoidance deals with changes in this environment, it finds ways to overcome obstacles that the path planner did not know about at the outset. Localization is a critical part for both path planning and obstacle avoidance, for a robot to be able to make smart decisions it has to know where it is positioned currently in its environment.

This is why autonomous navigation can be divided in three main parts: path planning, localization and obstacle avoidance. A lot of research has already been done on each of these topics and new implementations are still being introduced. A global view of autonomous navigation can be found in “Introduction to autonomous mobile robots” (Siegwart, Nourbakhsh, & Scaramuzza, 2011), which was used extensively throughout this thesis. To get insight in the topic of localization, the book “Probabilistic Robots” (Thrun, Burgard, & Dieter, 2006) was used to design and implement the algorithms needed. A good and clear overview of particle filters can be found in (Rekleitis, 2002). In (Holm & Pedersen, 2008) inspiration was also found in a similar project about indoor localization.

More information about the iRobot Roomba platform is given in chapter 2. Methods used to perform autonomous navigation are described in chapter 3. To be able to use certain navigation methods, an additional platform for sensors had to be implemented, which is explained in chapter 4. Experiments were done to show the performance of the AMR, which will be described in chapter 5. A critical reflection is given in chapter 6 and possible improvements and future attention points are pointed out in chapter 7. A final conclusion is given chapter 8.

Enclosed with this paper are a number of appendices with more detailed information and practical guides on how to use the robot and its codes. There are also a number of digital appendices provided on the included CD-ROM such as the MATLAB code of the project, CAD files as well as videos of different test runs.

2 ROOMBA PLATFORM

The original iRobot Roomba 620 was designed to be a low-cost, off-the-shelf automatic vacuum cleaner. But why is it used more and more in an educational or research context, where its vacuuming functions aren’t even used? Firstly, the iRobot Roomba is one of the cheapest robotic off-the-shelf products. Secondly, and most importantly, iRobot shared the command interface to control the Roomba with a serial cable. (Tribelhorn & Dodds, 2007)

2.1 Hardware and models

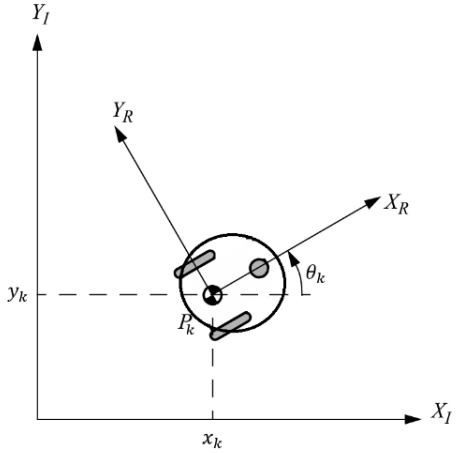


Figure 2. Model of the iRobot Roomba.
Y_I, X_I denotes the inertia coordinate system.
X_R, Y_R denotes the robot coordinate system.
$P_k = [x_k \quad y_k \quad \theta_k]^T$ fully defines the position of the robot.

The Roomba is a differential drive mobile robot; this means that it has two separately driven wheels. For stability reasons, there is also a third, little caster wheel at the front. The robot has a circular shape with a diameter of 34 cm. This simplifies a lot the geometric calculations, as the space taken up by the robot won't change depending on its orientation. The model used to represent the Roomba can be seen in Figure 2. The position of the robot is fully defined by its position (x_k, y_k) and orientation (θ_k). We define the position vector $P_k = [x_k \quad y_k \quad \theta_k]^T$.

The robot will make several types of motion: straight lines, turns and circular motions. The odometry data retrieved from the robot consists of a distance and an angle since last call. For the localization to interpret this correctly, equation (2.1) is used. Note that a movement cycle is either a straight line preceded with a turn without moving or a circular motion $(\Delta\theta \; or \; \Delta\varphi = 0)$.

$$P_k = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \Delta s \cdot cos(\theta_{k-1} + \Delta\theta + \Delta\varphi/2\,) \\ \Delta s \cdot sin(\theta_{k-1} + \Delta\theta + \Delta\varphi/2) \\ \Delta\theta + \Delta\varphi \end{bmatrix} \quad (2.1)$$

With:

$P_k = \; updated\; position$
$\Delta s \; = \; distance\; made\; by, the\; center\; of\; the\; robot$
$\Delta\theta = angle\; turned\; without\; moving$
$\Delta\varphi = angle\; turned\; during\; circular\; motion$
$k - 1 \; = precedent\; discrete\; position$

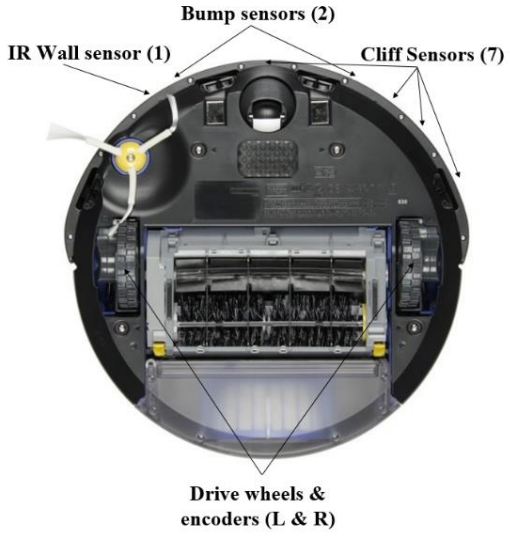


Figure 3. The iRobot Roomba 620 with an overview of its primary onboard sensors and actuators.

The Roomba comes with several onboard sensors (see Figure 3), to be able to vacuum a room autonomously. There are 2 tactile sensors, to sense if there is a collision at the left, right or front. There is an infrared sensor at the front right side of the robot, to sense if a wall is near or not. It has a range of +/- 10cm and can only send a true or false signal. There are two actuators for the wheels, with each an encoder to be able to have an estimation of the distance travelled and angle turned. Others sensors are available, but won't be discussed in the context of this thesis as they aren't used during the navigation. (Tribelhorn & Dodds, 2007)

2.2 Software

One of the main reasons why this project is possible on this robot, is because iRobot distributed the codes and parameters needed to communicate with the Roomba (iRobot, 2007). This has led to the creation of a MATLAB toolbox, which simplified the control by using functions instead of operation codes (Esposito, Barton, Koehler, & Lim, 2011). A simulation toolbox has also been developed, using the same syntax as the toolbox to control the robot, to make it possible to test navigation algorithms virtually before testing it on the actual robot. This simulation toolbox also has several sensor extensions, which can be used to extend the Roomba platform with more powerful sensors, like an ultrasonic sensor or a LIDAR sensor (Salzberger, Daisy Fan, & Kress-Gazit, 2014).

3 AUTONOMOUS METHODES

A truly autonomous robot has to have three main abilities: the ability to localize itself, the ability to navigate and to plan.

During the localization phase the robot finds an answer to the question where it is positioned in its environment without getting this information from an external source (e.g. user), this is an essential part of an autonomous vehicle (Cox, 1991). The robot in this project has to be able to navigate itself in a relatively large building without bumping into walls and has to reach its goal in an efficient way. To do this successfully localization is required. Note that this ability might not always be necessary for every robot, for example, the standard iRobot Roomba is not able to localize itself and can still perform its task autonomously by having a predefined behavior if a certain event materializes (if its left tactile sensor is triggered, it will turn right).

Two types of navigation can be distinguished in mobile robotics: Path Planning (PP) and Obstacle Avoidance (OA). Those two navigation methods can be perfectly combined as they work on different levels. PP needs a representation of the environment. OA requires only sensor data, together with start and goal positions. In sections 3.2 and 3.3, several methods of PP and OA respectively will be presented and the final selection retained will be explained. (Siegwart, Nourbakhsh, & Scaramuzza, 2011, p. 369)

3.1 Localization

There are multiple ways to solve the localization problem in robotics, the most intuitive way would be to use a GPS coordinate receiver. However a GPS receiver often fails in indoor situations and has a maximum accuracy of around three meters which is not satisfactory for indoor navigation (U.S. Department of Defence, 2008). Indoor positioning systems (IPS) could be used, however commercial systems are often very expensive (±15.000$) to implement (Randell & Muller, 2001). To assist robot localization, IR beacons or colored markers are very

useful, this however would mean modifying the building for our purposes and this was not an option.

In this application the robot needs to be able to localize itself from an unknown position and track its position while navigating. This last problem cannot be solved using dead-reckoning, where only odometry data is used. As each position estimate is relative to the previous one, this would result in a continuously cumulative error due to the imprecise odometry. (Tsai, 1998) Comparing the data of a laser sensor with the information given by a map of the building provides a much better result; this choice is explained in section 4.2.

An easy way to use a map in navigation problems is to create an occupancy grid map where each pixel of a bitmap picture corresponds with an area in reality, a zero on the map means the area is unoccupied while a one means there is an obstacle in that area. For each floor of the building an occupancy grid map was created. An example of an occupancy grid is given in Figure 4. For more information on how to generate these maps we refer to Appendix VI.

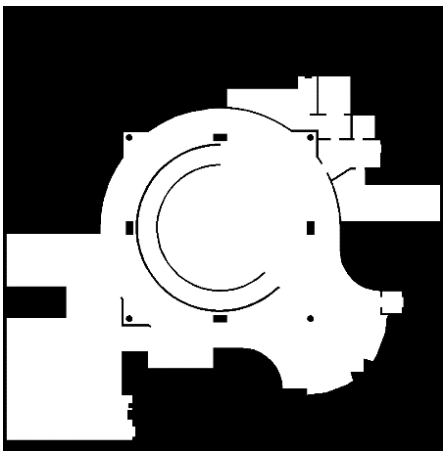


Figure 4. Example occupancy grid map of floor 0 of the university building. Black areas are occupied, white areas are free space.

Two probabilistic localization methods are often chosen for this localization problem: (Extended) Kalman filters (EKFs) and Particle filters. EKFs are performing well in tracking positions, but are unusable to perform global positioning (Fox D. , Burgard, Dellaert, & Thrun, 2001) (Dellaert, Fox, Burgard, & Thrun, 2001). When using EKFs the probability distribution has to be Gaussian and the uncertainty of the robot's position has to be low. Theoretically, in a global positioning problem, the initial uncertainty is equal to infinity, this is why particle filters were chosen in this application. (Kwok, Fox, & Meila, 2004) Instead of describing the required probability density function (PDF) as a function, in this scheme it is represented as a set of random samples of the PDF.

The accuracy of the localization can be improved by increasing the number of particles. Unlike the EKF this approach also works with non-linear and non-Gaussian assumptions. The basic version of this kind of filters is easy to implement, but has the disadvantage of being computationally intensive (Salmond & Gordon, 2005). One of the most popular methods to use particle filters for mobile robot localization is the Monte Carlo Localization Algorithm (MCL). Table 1 provides a comparison between the EKF and MCL approaches.

Table 1. Table with a comparison between EKF and MCL (Thrun, Burgard, & Dieter, 2006, p. 274)

	EKF	**MCL**
MEASUREMENTS	Landmarks	Raw measurements
MEASUREMENT NOISE	Gaussian	Any
POSTERIOR	Gaussian	Particles
EFFICIENCY (MEMORY)	++	+
EFFICIENCY (TIME)	++	+
EASE OF IMPLEMENTATION	+	++
RESOLUTION	++	+
ROBUSTNESS	-	++
GLOBAL LOCALIZATION	No	Yes

The basic version of Monte Carlo works according to a very simple principle, to localize itself the robot needs two things: a sensor measurement and a map to compare this measurements with. This comparison is done for a large number of particles in space and in this way the most likely correct particle can be chosen from the particle set. The map provided doesn't hold any measurement information, this is why a process called ray-casting is used to simulate sensor measurements. These simulated measurements are the measurements the sensor would measure when it has a certain position. A schematic image of this process can be seen in Figure 5. This process could be done in real time but because of its computational intensity is chosen to be done offline. In this way, the process of getting the simulated measurements is reduced to a much faster table lookup. More information about ray-casting can be found in appendix III (Thrun, Burgard, & Dieter, 2006, p. 168).

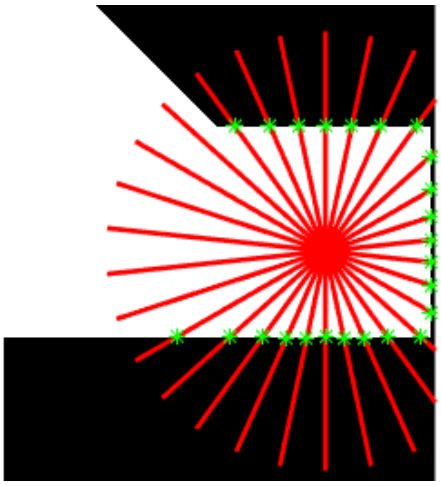


Figure 5. A schematic representation of the ray-casting process. The green marks are the end points of the casted ray which has the same range as the sensor. This end point is used to calculate the simulated measurement.

These simulated measurements can then be used to compare with the real measurements. When the two measurements correspond well, a weight is added to that particle, this is done for every particle and thus the particle with the highest weight can be chosen as the best estimate of the robot's position. When the robot moves, the cloud of particles can be moved based on the odometry data given by the wheel encoders of the AMR.

The particle filter algorithm is recursive in nature and operates in 2 phases: prediction and update. After each action of the AMR, each particle is modified according to an existing model which is the *prediction phase*, noise is added to simulate the effect of the odometry imperfections, for a more detailed description we refer to Appendix I. In the *update* phase the particles weight is re-evaluated based on the latest sensory information available (Rekleitis, 2002). After each cycle the particles with small weights, thus have a low probability to be correct, can be eliminated in a process called resampling.

The Monte Carlo algorithm was designed and implemented as explained in (Thrun, Burgard, & Dieter, 2006). The particle set of Monte Carlo localization consists of M particles, this is often a large number (e.g. 500). Equations (3.1) and (3.2) offer a description of the particle set and an individual particle:

$$S_t = \{s_1, s_2, \dots, s_M\} \tag{3.1}$$

$$s^{[m]} = \begin{bmatrix} x^{[m]} \\ w^{[m]} \end{bmatrix} for\ m = 1 \dots M \tag{3.2}$$

With:

$S_t = total\ particle\ set\ at\ time\ t$
$s^{[m]} = individual\ particle\ m$
$x^{[m]} = state\ guess\ of\ particle\ m$
$w^{[m]} = weight\ assigned\ to\ particle\ m$
$M = total\ amount\ of\ particles$

The regular MCL algorithm looks as shown in Pseudo code 1:

Pseudo code 1. Monte Carlo Localization (Thrun, Burgard, & Dieter, 2006)

1: **Algorithm MCL**(S_{t-1}, u_t, z_t, map):
2: $S_{temp} = S_t = \emptyset$
3: for $m = 1\ to\ M$ do
4: $x_t^{[m]} = \boldsymbol{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} = \boldsymbol{measurement\_model}(z_t, x_t^{[m]}, map)$
6: $S_{temp} = S_{temp} + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: endfor
8: $W_{total} = \sum_{m=1}^{M} w_t^{[m]}$
9: $w_t^{[m]} = \frac{w_t^{[m]}}{W_{total}}$
10: $S_t = \boldsymbol{resample}(S_t)$
11: $index = \max(W_t)$
12: $x_{est} = S_t(index)$
13: return S_t, x_{est}

The algorithm gets as inputs the previous particle set, S_{t-1}, in this project the initial samples are uniformly distributed around the start position, this start position is an inaccurate position given by the user and is not necessary for smaller areas. u_t are the odometry measurements provided by the onboard odometry of the AMR. z_t are the measurements provided by the laser sensor, map is the matrix with the simulated measurements from the ray-casting process as explained before.

The algorithm gives the estimated position x_{est} and the updated particle set S_t, this estimated position will be used for navigational purposes and the updated particle set will be used as an input in the next localization cycle.

In lines 3 to 7 the particle set goes through the prediction and update set. In the sample_motion_model each particle's old position and angle $x_{t-1}^{[m]}$ gets moved to its new position and angle according to the odometry data u_t, this is done in a probabilistic way where odometry noise is added to the particle positions and angles. The motion model is based on the kinematic model explained in section 2.1 and more detailed explanations of the implementation is given in appendix I. In the measurement_model each particle is assigned a weight by comparing the sensor measurements with the offline ray-casting data. This is done according to a sensor model as explained in chapter 4.2.1 and appendix II. The sensor measures one distance for every degree, these 360 measurements get reduced to 36 to lower the computational intensity, in appendix III this parameter is explained in more detail. The weight assigned to each particle is equal to the combination of the probability of each range sensor measurement, as shown in equation (3.3).

$$w_t^{[m]} = p(z_t|x_t, map) = \prod_{k=1}^{n_k} p(z_t^k|x_t, map) \tag{3.3}$$

With:

$w_t^{[m]} = weight\ of\ particle\ m\ at\ time\ t$
$z_t = set\ of\ sensor\ measurements$
$z_t^k = individual\ sensor\ measurement$
$map = ray\text{-}casted\ measurements\ from\ map$
$n_k = number\ of\ measurements$

Where $p(z_t|x_t, map)$ is the probability of the total sensor measurement being correct for a certain particle and a set of simulated measurements based on the map. $p(z_t^k|x_t, map)$ is equal to the probability of one single sensor measurement k being correct for a certain particle and the simulated measurement based on the map data.

In line 6 the updated particle will be added to the current set of particles S_{temp}. In line 8 and 9 the weights are normalized to ensure the particles are representing a probability distribution, so the area under its graph is equal to one. In line 10 the worst particles, those with the lowest weights are removed and the best ones get duplicated. This is done to avoid that the particles get scattered all around the state space after a couple of steps and allows for a more

accurate localization. More information about the resampling process can be found in appendix IV. In line 11 and 12 the best particle is selected, this can be done in different ways; the simplest way is to select the particle with the highest weight. Another method would be to use multiple points for a more robust result. The selected particle is returned in line 13 as well as the particle set which will be used for the next localization cycle.

3.2 Path Planning

PP is used to decide an optimal path given a map of the environment, a start position and a destination position. Even if some methods exist where the map is continuously updated with sensor data and executed several times during the navigation to provide a better path, this will not be applied here as static road blocking obstacles don't occur during normal circumstances in our environment.

For a PP method to work, a representation of the "world" is needed. This section will discuss two algorithms, both based on graph theory. A graph is an abstract representation of a map, consisting of vertices (nodes) connected to each other with edges (links). Edges have an associated cost. In general this cost is a distance, but can contain other factors depending on what quantity needs to be optimized (difficulty of terrain or time needed to go from one vertex to another). Two vertices connected with an edge is called a path (road). (Corke, 2011, p. 97)

The two following methods are roadmap methods. When using a roadmap method, all possible roads are computed using the obstacles indicated on the map. Then, start and destination points are added. The roadmap is thus only dependent on the obstacles. Start and destination position will only change the chosen set of roads. The more obstacles there are, the more computational power a roadmap method will require. This can be solved by creating the roadmap of a specific map beforehand (offline) and just adding the start and destination position at the beginning (online) and then calculating the optimal path. With both of those methods, another algorithm has to be used to calculate the shortest distance between the added start and end position. Here, Dijkstra's algorithm is used for both methods, because of its optimal solution. (Siegwart, Nourbakhsh, & Scaramuzza, 2011, pp. 375-376)

The first method is the Visibility graph, (see Figure 6, upper part) this method minimizes the total length of the path, by placing vertices on each obstacle discontinuity and creating edges between each visible vertices. The biggest drawback to this method is that the optimal path is as close as possible to the objects, which tends to put the AMR in dangerous positions. In this method, the space taken up by the AMR has to be taken into account. This can be done by inflating obstacles by the AMR's radius. Doing so will make the proposed solution less optimal in path length. (Siegwart, Nourbakhsh, & Scaramuzza, 2011, p. 374)

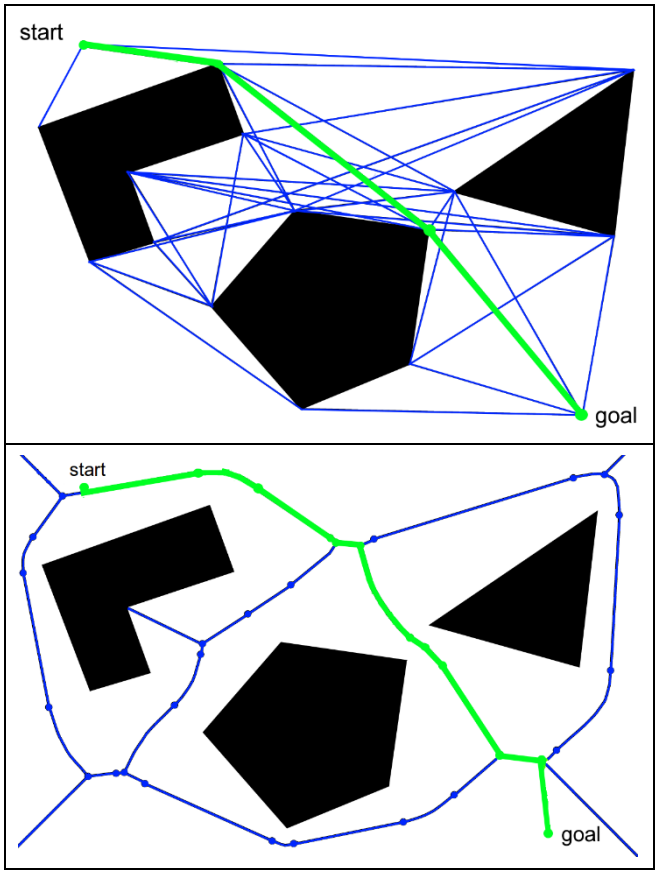


Figure 6. Visibility graph (above) and Voronoi Diagram (below). The obstacles are black. The blue line is calculated roadmap. The green line is the optimal path, using Dijkstra's algorithm, using the given start and goal position.

The second method is the Voronoi Diagram (see Figure 6, lower part) which consists of vertices placed as far as possible of each obstacle and edges that are equidistant to the objects. Taking the shape of the robot becomes then optional. The calculations of the vertices and edges is more demanding than the Visibility graph method and the resulting roadmap is far from optimal with regard to its length. The biggest drawback of this method is that a robot with only short range sensors could lose itself on its path if obstacles are too far from it. A review of both methods is shown in Table 2. (Siegwart, Nourbakhsh, & Scaramuzza, 2011, pp. 375-376)

Table 2. Review of both graph methods: the Visibility Graph (left column) and the Voronoi Diagram (right column)

Visibility Graph	Voronoi Diagram
Paths as close as possible to obstacles.	Paths as far away as possible from obstacles.
Taking shape of the robot is needed	Taking shape of the robot is optional
Quite simple to find vertices and edges	More complex to find vertices and edges
Shortest path	Safest path

After a careful assessment of both methods and their implementation in our university campus, the best method appears to be the Voronoi Diagram. This because of several factors:

1. As we are taking the largest possible distance to each obstacle, a lot of small obstacles can be omitted in the planning phase when they are close to the walls.
2. The primary goal of the AMR is to arrive at its destination, so it is better to arrive safely than to select a slightly shorter routing but thereby putting the AMR in a potentially dangerous position.
3. This method is more comparable to the intuitive method that a person choses when walking, it will therefore be more user friendly to follow this same trend, when guiding visitors in Group T.
4. A lot of documentation is already available on MATLAB about the Voronoi Diagram and Dijkstra's algorithm, which makes the implementation easier.
5. The only drawback is that a long range finding sensor is needed (e.g. LIDAR) for localizing the AMR, because the robot stays at a large distance with respect to the obstacles.

3.3 Obstacle Avoidance

OA only relies on sensor data, robot and goal position. This goal position does not necessarily need to be the destination position, but could also be an intermediary position on the path decided by the PP. OA will also take the shape of the robot into account and find a collision free trajectory, while following the path calculated by the PP. OA uses sensor data and will therefore sense static obstacles that could have been omitted/changed in the map and avoid dynamic obstacles.

Since the PP relies on long range data, we can also search for an OA method relying on this. One type of long range sensor that can be used is a LIDAR sensor. A whole set of OA algorithms exist, starting from the basic methods, so called 'bug' algorithms (due to their simplicity) that only rely on tactile sensors (Oroko & Nyakoe, 2012). But the goal is not to have the AMR bumping into every corner so those methods can't be used. A whole family of algorithms have been developed on the original idea of the Vector Field Histogram (VFH). The VFH is primarily designed for sonar sensors, but other methods have been adapted for laser range sensors (Borenstein & Koren, 1991). After a careful review of methods based on VFH (Babinec, Duchon, Dekan, Paszto, & Kelemen, 2014), the Vector Polar Histogram (VPH) seems a good choice, because:

1. It can take the geometry of the robot into account
2. With its variable threshold function, it can take the speed of the robot into account
3. A speed controller can be implemented (when too close to an obstacle or arriving at its destination)

The implemented OA described below will be based on the VPH (An & Wang, 2004) method and its upgraded version VPH+ (Jianwei, Yulin, Yiming, & Guangming, 2007). Several improvements have been made to those methods, mainly the possibility to make circular motions and an improved way to take the geometry of the robot into account, to avoid any collision with its environment.

The following section will describe a full cycle of the VPH method. The VPH has as input the scan data made by a LIDAR sensor, the robot position and the goal position and gives a desired heading angle $\theta_{desired}$, the speed for the next cycle and the Instant Centre of Rotation (IRC) radius r_c, if circular motion is desired.

Let i be the index of each measurement, $d(i)$ be the original measurement, $D(i)$ the modified measurement with the geometry of the robot, v the speed of the robot, T_{cycle} the total cycle time between two VPH runs and R the radius of the robot. An improved method has been implemented to fully take the shape of the robot into account, because when using the original method, the robot could collide with certain obstacles. Instead of just subtracting the radius of the robot of each measurement (as described in the original method of the VPH+), each obstacle has been inflated with the radius of the robot. This enables for the approximation of the robot by a single point for the remaining calculations. This is done by taking into account the width of the robot: an obstacle point will have an impact on the nearby measurement defined as $\theta_{influence}$. Let j contain all measurements between $i \pm \theta_{influence}$. If any measurement from $d(j)$ is bigger than $D(i)$, those measurements are modified as being equal to $D(i)$. (Al-Sagban & Dhaouadi, 2011). A comprehensive set of figures can be seen in Figure 7, explaining this method.

$$D(j) = \begin{cases} d(j), if\ d(j) < D(i) \\ D(i), if\ d(j) \geq D(i) \end{cases}$$

After this step, a threshold function $H(i)$ is constructed. This function will define if a direction can be selected later on in the cost function $C(i)$. A safe distance variable is defined $d_{safe} = v \cdot T_{cycle}$. If any $D(i) < d_{safe}$ then it is considered a dangerous angle and it can't be chosen as possible direction angle.

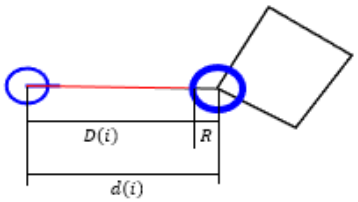
$$H(i) = \begin{cases} 0, if\ D(i) < d_{safe} \\ 1, if\ D(i) \geq d_{safe} \end{cases}$$

In the next step, an effort-goal function is constructed $S(i)$ in equation (3.4), comparing a goal seeking (factor k_1) and effort saving (factor k_2) behaviour. k_3 is needed to insure a non-zero value. Angles are defined as shown in Figure 8.

$$S(i) = k_1 \cdot hg + k_2 \cdot ho + k_3 \tag{3.4}$$

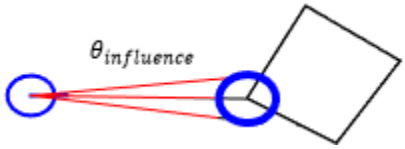
With:

$k_1 = goal\ seeking\ behavior\ factor$
$k_2 = effort\ saving\ behavior\ factor$
$k_3 = constant\ to\ insure\ a\ non-zero\ value.$
$k_1 > k_2, for\ a\ goal\ seeking\ behavior$
$k_1, k_2, k_3 > 0$
$hg = angle\ to\ goal$
$ho = angle\ to\ obstacle\ i\ (effort)$



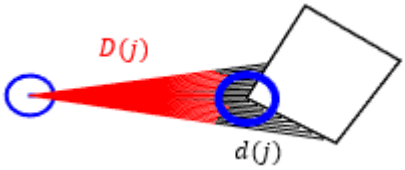
For a direct measurement, this method will just subtract the radius of the robot from the measured distance.

$$D(i) = d(i) - R$$



Each obstacle point will have a zone of influence, calculated from this formula:

$$\theta_{influence} = \tan(R/d(i))$$
$$j = i \pm \theta_{influence}$$



The resulting $D(j)$ is modified in such a way that the obstacles have been inflated by the robot's radius. The black lines (d_j) are the original measurements of the laser scanner the red lines($D(j)$) are the modified measurement, taking into account the shape of the robot

$$D(j) = \begin{cases} d(j), if\ d(j) < D(i) \\ D(i), if\ d(j) \geq D(i) \end{cases}$$

Figure 7. How obstacle inflation works. i is the index of a measurement, j contains every measurement between $i \pm \theta_{influence}$. $D(j)$ is the modified measurement after obstacle inflation.

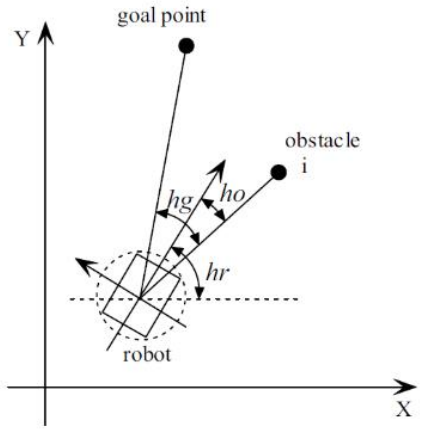


Figure 8. Angles used in cost function $S(i)$. hg is the angle to the goal, ho is the angle to obstacle i and hr the heading angle

The final step combines every factor described above. By finding the maximal value of $C(i)$ from equation (3.5) we get $\theta_{desired}$, the steer angle the AMR has to turn, to get closer towards its goal.

$$C(i) = \frac{H(i) \cdot D(i)}{S(i)} \tag{3.5}$$

With:

$C(i) = cost\ function$
$H(i) = threshold\ function$
$D(i) = distance\ to\ obstacle$
$S(i) = effort - goal\ function$

If smooth circular motion is desired instead of turns to achieve its goal, equation (3.6) can be implemented (Al-Sagban & Dhaouadi, 2011).

$$r_c = \frac{d_{goal}}{2 \cdot \sin(\theta_{desired})} \tag{3.6}$$

With:

$r_c = IRC\ radius\ for\ circular\ motion$
$d_{goal} = distance\ to\ goal\ (or\ next\ node)$
$\theta_{desired} = desired\ angle\ found\ by\ the\ VPH\ method$

4 IMPLEMENTATION

In this chapter the different additions to the iRobot Roomba and their implementations are explained. These additions were essential to apply the autonomous methods described in chapter 3.

4.1 Platform

An additional platform was necessary as the stock iRobot Roomba has no mounting positions for additional sensors and for carrying the laptop that is used to make computations and send commands to the AMR actuators. This was the main requirement for this project in addition to some minor necessities; the platform had to be designed with other projects in mind, had to be easy to extend or modify and easy to assemble with minimum changes to the original Roomba. Another concern was the amount of vibrations that propagate from the robot to sensitive parts like the range sensor and the laptop.

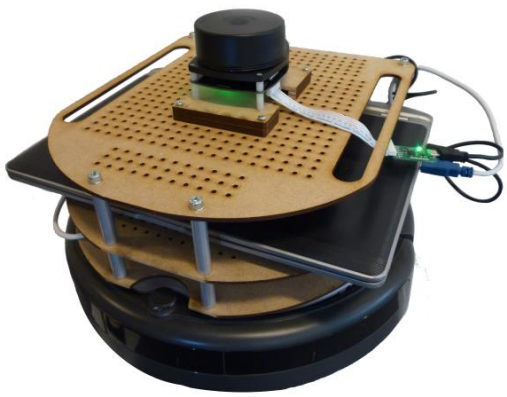


Figure 9. Photo of the final platform with the laptop and LIDAR sensor.

The design was inspired by the open-source design of the TurtleBot (Open Source Robotics Foundation, 2015) but was designed from scratch to incorporate our requirements. The main material for the platform is medium density fibreboard, due to its easy machinability, low cost and excellent damping properties. (Binggeli, 2013, p. 121) This material also allows other users to modify the design to fit their own needs or extend it. To extent the platform it was designed so the user can easily remove or add another level to the robot platform. Furthermore the mounting system was designed with other platforms in mind and allows the user to swap the complete platform quickly. To solve the issue of vibrations, anti-vibration rubber was used between the robot and the platform to lower vibration propagation.

4.2 Sensor

For obstacle avoidance and localization an additional sensor is needed to provide information about the robot's environment.

For this project a Laser Range Sensor (LRS) was chosen; another option would have been to choose a vision sensor like the Microsoft Kinect, but these weren't chosen due to their small range and the computational nature of vision systems. Range finding sensors require no additional computations and are in general easier to use. There are different kinds of range finding sensors, the most common ones are LIDAR (which use light to determine distances to objects) and sonar (which use sound). Sonar sensors are harder to model because they are more dependent on their environment and have complex noise characteristics. LIDAR sensors and more specifically laser sensors have good noise characteristics, are easy to model and scanning LIDAR sensors are able to take a large amount of measurements all around them, which is useful in obstacle avoidance and localization. The main disadvantage is however the price of some of these sensors. For this project a 360 degrees laser scanner was chosen made by RPLIDAR, which has excellent distance and angular resolutions and a high maximum range. In Table 3, an overview is given with the specifications of the RPLIDAR sensor.

Table 3. Table with LIDAR sensor specifications (RoboPeak, 2015)

Item	Typical
Distance Range	0.2-6 m
Distance Resolution	<0,5 mm | <1% Distance mm
Angular Range	0-360°
Angular Resolution	≤1°
Scan Rate	5.5 Hz

The RPLIDAR is controlled with a C++ code, which is available on the official webpage and extended with an additional UDP data sender to be able to continuously send up-to-date data, but only fetch it when required. For this to be possible, two programs have to be run simultaneously: the C++ code, which sends the LIDAR data and MATLAB, with the M-code which fetches and stores the sensor data in the workspace. In this application, the UDP is send to the localhost (the computer sends via the UDP protocol, data to itself), but one can imagine applications where this could be used to decentralized the computational power off the AMR and send all the data via UDP (sensor and actuator commands) (see section 7.2).

Each fetch cycle will return around 360 measurements (n), which are put in a matrix the following way:

$$z_t = \begin{bmatrix} d_1 & \theta_1 \\ \vdots & \vdots \\ d_n & \theta_n \end{bmatrix}$$

Where z_k is an individual measurement with distance d_k and angle θ_k. This matrix is used in both localization and obstacle avoidance as explained in section 3.1 and 3.3 respectively.

4.2.1 Sensor model

During the localizing of the robot a probabilistic model of the laser sensor is required to assign a weight based on the probability that a measurement is correct. The design of this sensor model is based on the information given in (Thrun, Burgard, & Dieter, 2006, p. 153). The two inputs for the model are the ray-casted measurements and the real measurements provided by the LRS, the output is the probability that a sensor reading is correct. During the localization the algorithm samples from the probability distribution.

Three different kinds of errors are incorporated into the model that was implemented for this project, small measurement noise, errors due to unexpected objects and random unexplained noise. These errors are described with probability density functions. The total probability density function $p(z_t|x_t, map)$ is a sum of these three individual density functions. The first density function resembles the density function of a correct measurement with local measurement noise added. In the second density function unexpected objects are modelled, maps are static, while the robot will be located in real environments where there are unmapped objects. This is modelled with an exponential distribution. Lastly there is the possibility to have random measurement noise which is modelled by a uniform distribution. Each of the individual distributions is normalized and summed so the total probability density resembling the entire sensor model as can be seen on Figure 10.

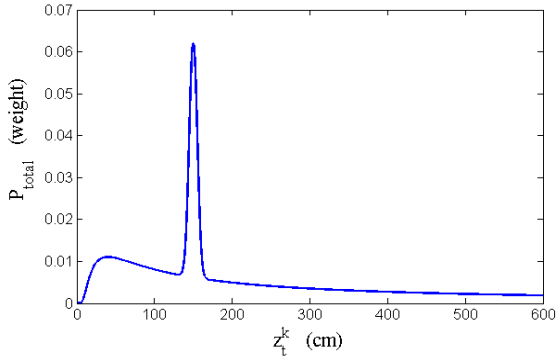


Figure 10. Total PDF of sensor model

Parameters can be changed to decide the weight each individual density function has on the total sensor model, for a more detailed explanation we refer to appendix II, for the adjustment of the parameters we refer to appendix III.

5 EXPERIMENTS

Two navigation objectives were imposed for the scope of this thesis. The first one was to show our understanding of the Roomba platform. The AMR should be able to navigate from one module to another, on the spiral of our campus, with basic navigation techniques. No real localization was needed here and the robot was allowed to bump into objects. The second objective was to implement extended navigation techniques that made localization and a global path planning possible. The usage of maps was allowed. The end objective of the extended navigation was to be able to guide a visitor of to our campus to a desired module and room, via a user friendly input.

5.1 Basic Navigation

The first objective was to explore basic navigation techniques with the AMR by using the MATLAB toolbox. Here, only onboard sensors and actuators were used (see chapter 2). By only using onboard sensors, localization becomes impossible, because the odometry is never precise enough to perform dead reckoning. A robust and simple navigation technique had to be implemented on the AMR, using only two tactile sensors and the IR-sensor. Using the latter as a guide to detect incoming walls on the front-right side of the AMR, a basic "wall-following" behavior is employed. This is implemented in a Finite State Machine, with 5 possible states, as shown in Figure 11.

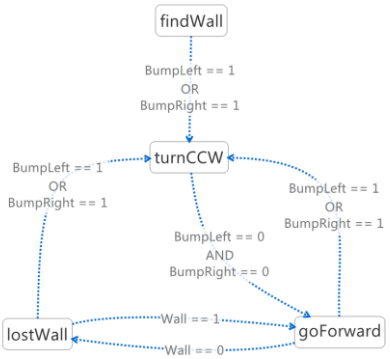


Figure 11. Finite State Machine of the wall following behavior.

- findWall: go straight, until bump sensors are active. If bump sensors active, turnCCW
- turnCCW: turn counter clock wise until no bump sensors are active. If no bump sensors active, goForward
- goForward: go in an arc away from the wall, until IR-sensor sends falls signal, then lostWall. If bump sensor active, turnCCW
- lostWall: go in an arc towards the wall, until IR-sensor sends true signal, then goForward. If bump sensor active, turnCCW.

Because the serial cable to connect the Roomba with a computer isn't provided, it had to be made separately according to the electrical schemes described in "Hacking Roomba" (Kurt, 2006, pp. 41-63). Several changes to the original MATLAB toolbox had to be made, the most important of which were the new serial communication settings. This is due to a change of Roomba version, as the Roomba versions below 500 use the Serial Command Interface (iRobot, 2005), for which the MATLAB toolbox had been designed, while versions above 500 use the Open Interface (iRobot, 2007). Another important change was made to speed up the process of receiving and sending information over the serial cable by eliminating unnecessary bit flushing, which takes a lot of time in serial communications. This improved the speed from 1 Hz to 20 Hz. Finally, a new odometry reading function has been implemented, using only the encoder values of both wheels which greatly improved the accuracy, compared to the original built-in codes.

5.2 Extended Navigation

To achieve the final objective of the extended navigation in Group T, experiments were first made in the robot lab of the Department of Mechanical Engineering, where a part of the lab was used as test room. There, testing was much easier because:

- Navigation techniques could be performed without disturbing class-changes
- The test room was much easier to set up
- The small test room made it possible to perform quicker tests

Once the extended navigation technique worked correctly in the robot lab, experiments were made at the Group T premises. The first difficulty in Group T is that a simple 2D approximation is not possible. Different modules and class rooms are on the top of each other, thus with the same 2D coordinates. Using a 3D map was not a realistic option, because only 2D maps are at our disposal and a 3D map would have taken much more memory space, than a set of carefully chosen 2D maps. Still, a work-around had to be found if only 2D maps are used. This is done by using a module navigation: e.g. to go from module 7 to module 11 in Group T (which have exactly the same 2D coordinates, as they have the same location but at different floors), the AMR has to pass by module 8,9,10. To be able to navigate with only 2D maps, a map changer algorithm has to be implemented, to be able to change at a specific point, from the current to a new map, given a current, start and destination position (See appendix XI).

5.2.1 Localization experiments and results

Global positioning. For the localization algorithm to work correctly it is important that it is able to localize itself in an environment, this is tested by putting the AMR in the environment and measuring its true position, then letting the algorithm localize the robot and see how this position compares to the true position. This can be done for multiple iterations to see if the algorithm is able to improve the result. It is important to note that the position of the AMR is not altered in between these iterations. This experiment is shown in Figure 12.

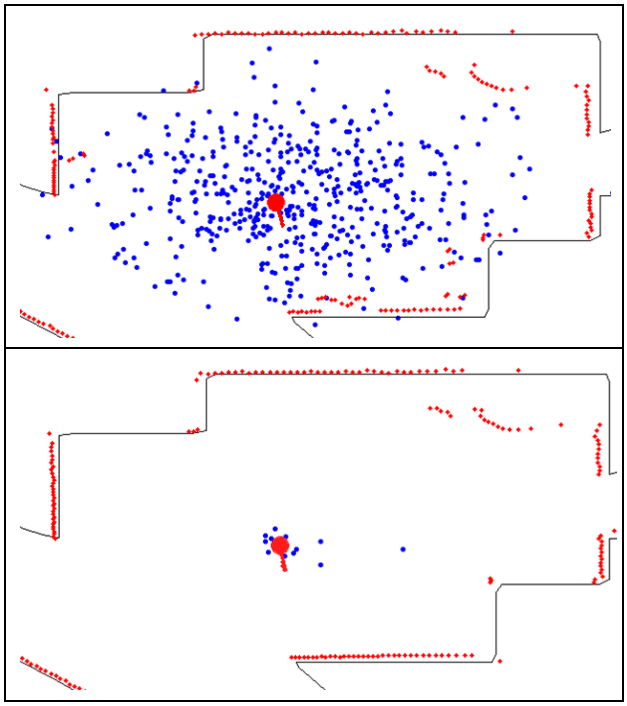


Figure 12. Image of localization experiments showing the estimated position and sensor data (red), on the first image the particles (blue) are initiated in a large area, the second image shows that the resampling algorithm eliminates the particles with the lowest weights.

Several of these global positioning tests were done and the robot was able to position itself within 15cm of its true position in all tests; this result is satisfactory for an indoor AMR. It can therefore be concluded that the MCL algorithm is able to successfully localize the robot.

Position tracking. Global positioning is just one step in the robot's algorithm. It is also important that the robot is able to localize itself after each motion and thus able to track its position after each motion. This process is done with the recursive MCL algorithm, this means that we do not restart the algorithm after each step but we use the information from the previous steps to estimate the new position. To test this ability the robot is moved between each localization step and we look if the robot is able to localize itself after each motion, this experiment is shown in Figure 13.
This test was done for different step sizes and rotations; the robot was always able to keep track of its position in standard environments.

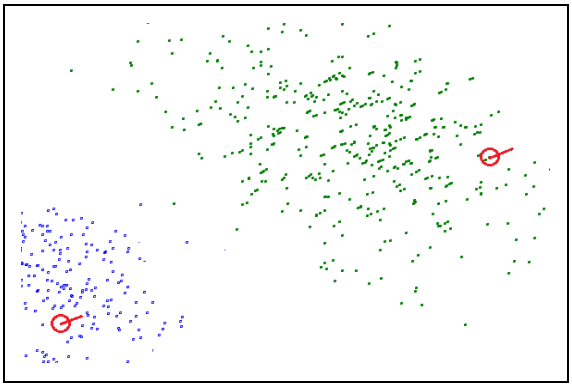


Figure 13. Plot showing the robot positioning between two consecutive steps, notice that the particles are more dispersed after the step (green point cloud > blue point cloud).

5.2.2 Path Planning results

During the path planning phase, the obstacles of the original maps are used. During the path planning test, some calculated roads couldn't be used by the AMR, even if those roads where as far as possible close to the walls. This happened mostly with pillars close to the walls, where the space between those two was too small for the robot (see Figure 14, top).

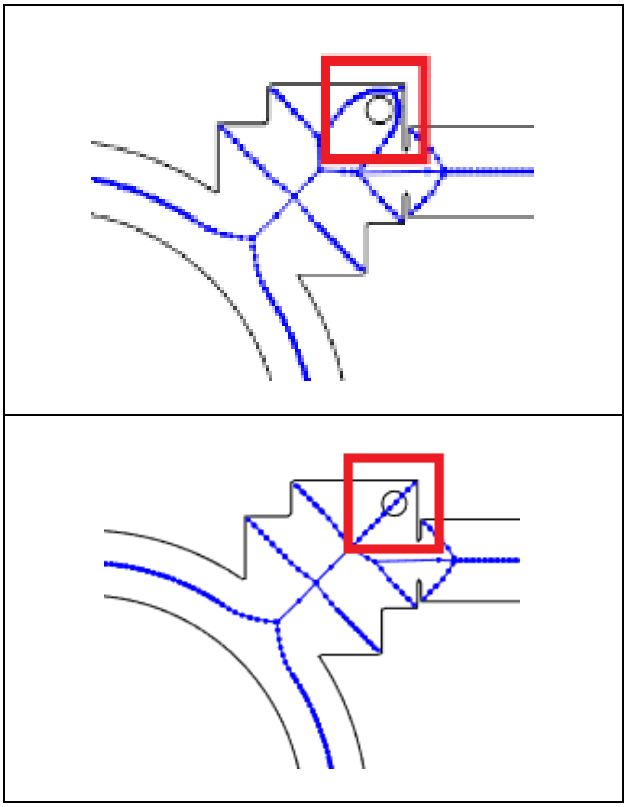


Figure 14. Roadmaps (blue lines) calculated from obstacles. Above, all obstacles are given to the PP algorithm. This creates roads that the AMR can't use (in the red box), because the space between the pillar and the wall is too small. In the bottom figure, the obstacles creating infeasible road have been omitted, resulting only in feasible roads. This assuming the user won't choose an invisible destination.

A simple solution to overcome the creation of those unfeasible roads was found by checking each map for objects that could be omitted, without endangering the path planning. This is shown in Figure 14 (bottom) and had to be checked manually, floor by floor.

5.2.3 Obstacle Avoidance results

The OA algorithm's goal is to find a collision-free path, while following the calculated path of the PP. Before discussing the tuning parameters of the VPH, two different simulation experiments are made, to prove its robustness and real world application.

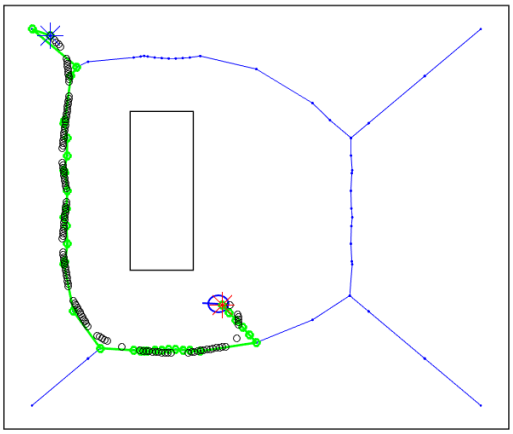


Figure 15. Proof that the OA can follow a path given by the PP. Here, the AMR navigates using the PP and the OA. The green path is the optimal path calculated by the PP, the black circles are the path taken by the robot. The start position is the blue star, and the destination is the red star. The OA leads the robot on the planned path, which is what had to be investigated.

The first one consists in testing if the OA is able to follow the path given by the PP. As can be seen from Figure 15, the VPH does lead the robot on the path calculated by the PP.

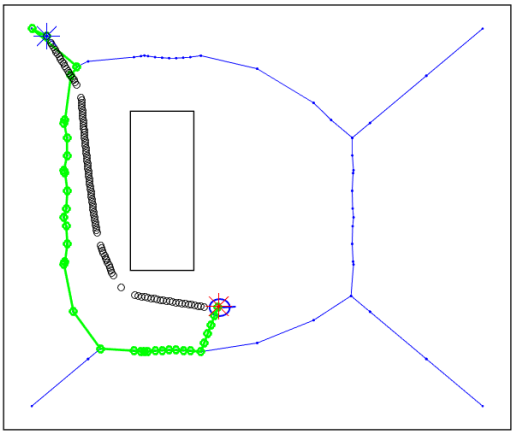


Figure 16. Proof that the OA will not collide with unplanned obstacles. The AMR navigates using only OA. The black circles are the path taken by the robot. There is also the path calculated by the PP in green. The start position is the blue star, and the destination the red star. As one would have expected, the AMR gets as close as possible to the wall, which is on the direct path between the start and destination position, but still without colliding with it. This proves that the OA can take the shape of the robot into account.

The second simulation experiment consists of turning off the PP and just giving a destination position to the VPH (this can be interpreted as an unknown obstacle, between two nodes of the path generated from the PP) to prove that the VPH will never make the robot collide with an unknown obstacle. The result of this experiment can be seen in Figure 16. The AMR arrives safely at its destination, which proves that the OA can take the shape of the AMR into account.

The most important parameter of the VPH algorithm is the weight given to the goal (k_1), in comparison to the weight given to the effort (k_2). In this experiment, a test scenario is investigated under two values $K = k_1/k_2$ (a high value in K means a high goal oriented behavior). This test scenario can be seen in Figure 17. Here the path with the higher value of K follows the safe road calculated by the PP, while the path with the lower K tends to cut several edges off. It should be noted that in terms of safety, cutting edges can put the AMR closer to obstacles. Because the primary goal of the AMR is to arrive safely at its destination, a value of $K = 10$ has been applied.

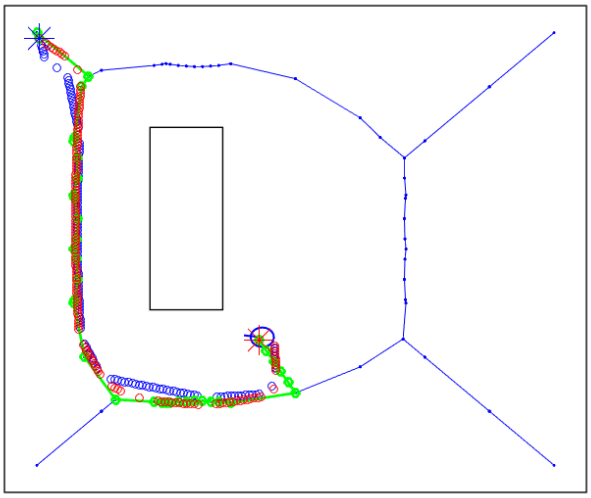


Figure 17. The AMR navigates with a PP (green path). The start position is the blue star, and the destination the red star. Two values of K are tested: the blue path has a $K = 1.5$, the red path has a $K = 10$. The red path stays closer to the green PP path (safest, because this is the part the furthest away from each obstacle). The blue path preserves the effort of the AMR by cutting off corners.

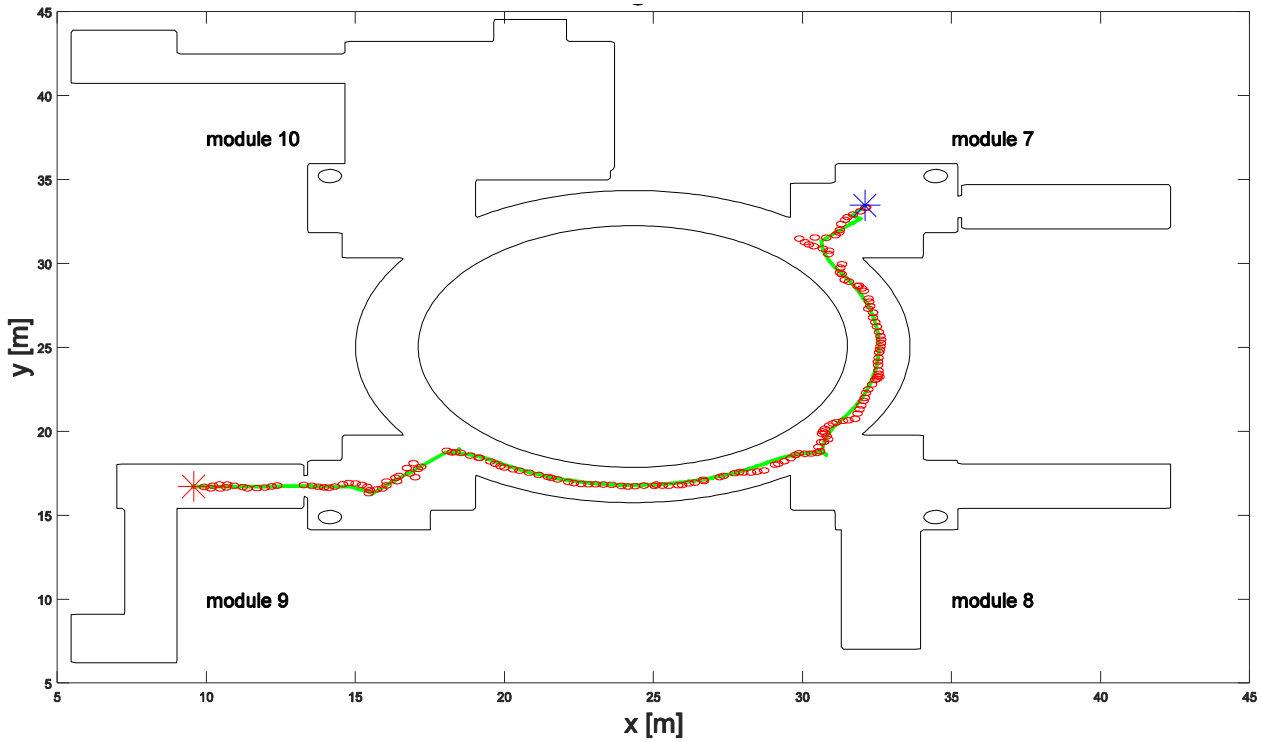


Figure 19. AMR Navigation result. The goal was to go from module 7 to module 9. The green line represents optimal path. The red circles the location and space taken by the AMR. The blue star is the start position and the red star the destination position. Note that the plotted location is the estimated position, not the true position.

5.2.4 Implementation results

In this section, comments are given on the interaction of the three main methods during the final experiments. These consisted of guiding a person from one location to another, in our university campus. Videos of those experiments can be found on the provided CD-ROM.

Figure 19 shows a successful navigation experiment. The goal was to go from module 7 to module 9. The cycle time is shown in Figure 18. Detailed explanation of the cycle time of each method can be found in appendix VII.

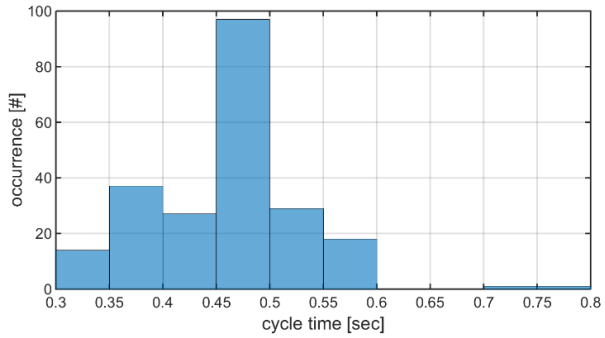


Figure 18. Total cycle time of the navigation algorithm. This cycle only contains LIDAR and onboard sensor fetch time, localisation and obstacle avoidance. The overall cycle frequency is 2 Hz.

The following parameters have been implemented to be able to have a robust navigation technique:

- Node Distance threshold: if the robot is within a certain distance of a node, it will look at the next one, until no node is found within this range. 0.5 meter distance provides satisfactory results.
- Angle threshold: if the VPH gives a $\theta_{desired}$ which is smaller than the angle threshold, circular motion will be used. If the angle is bigger, the robot will stop and turn this given angle. 45° provides satisfactory results.

Important remarks on the current flaws have to be made:

- The OA can't avoid highly dynamic obstacles (a person walking towards the robot), even worse, it will try to go backwards even if the obstacle has passed the robot.
- Because of the highly symmetrical shape of the building and lack of unique shapes on the spiral the robot has difficulties correcting its position. As a result, accurate encoder values become crucial, this has forced us to reprogram the default odometry function to obtain more consistent encoder measurements.

6 CRITICAL REFLECTION

During the selection and implementation of the project, a number of choices of algorithms had to be made, based on their respective advantages and disadvantages. This chapter provides a reflection on possible alternatives and under which circumstances such choices could achieve better results.

6.1 Localization

In general the Monte Carlo Localization algorithm proved to be an excellent choice, it was easy to understand and implement, delivers excellent performance and can be modified to fit the needs of the project. This however

doesn't mean the algorithm as it was implemented had no weaknesses.

The biggest shortcoming of the localization algorithm that is currently used is the amount of data and time needed due to the ray-casting process. If done in real-time we found the process to be too computationally intensive and if done offline it could cause memory issues. The matrices that store all the simulated measurements for a single floor need 720 megabytes of memory in MATLAB, knowing we use five different maps for each floor the total memory required is 3,6 gigabytes. For our implementation where we use a laptop with adequate available memory this isn't a big problem; however for many other implementations this requirement would not be acceptable. There are different possible solutions, one of them requiring minimum adjustments is the use of an occupancy grid map with a coarser grid. Currently a grid size of five centimeters is used. This would possibly require some interpolation method to calculate a more exact simulated measurement but would still be easily implementable. Another method would be to use a different type of map such as feature-based maps, which require much less memory and only depend on the amount of obstacles. However, such alternative requires a different approach to assign weights to particles or even different types of sensors (Andert & Goormann, 2007).

Another weak point of the algorithm is when the algorithm has to localize itself in cluttered environments, for example in locations with chairs and table, in such a location the algorithm has difficulties with the sensor readings. A solution for this problem, which also helps with the memory problem, is to use a likelihood sensor model. For more information we refer to (Thrun, Burgard, & Dieter, 2006, p. 169).

6.2 Path Planning

For some applications, the fact that the PP is only executed once can have a negative influence on the navigation: for instance, if one of the possible ways to go to a destination point is blocked. This type of algorithm, taking into account sensor data, could calculate a new road. Still, this kind of method would be excessive at Group T, where there is only a single path to go from one module to the next.

6.3 Obstacle Avoidance

The main reason why the VPH method was chosen is because of its specific design for the LRS, which made implementation of this method easier, as its calculations are made in polar coordinates (the same format as the LRS). However if a more careful study of the robot dynamics had been made, a more performant method would have been the Dynamic Window Approach (DWA). The DWA is designed to take the dynamics of the robot into account and is capable of calculating several steps in the future, resulting in a smoother navigation than the present one, a tuned VPH method. Due to a lack of time, DWA has not been implemented. (Fox, Burgard, & Thrun, 1997)

7 FUTURE WORK

Often algorithms or hardware have the possibility to be extended or changed for added functionalities, in this chapter possible extensions to the robot are proposed for future work.

7.1 Kidnapped robot + Monte Carlo extension

The selection of the Monte Carlo algorithm, as said above, was an excellent choice, notwithstanding some weaknesses. One of the additional advantages of this algorithm is the ability to extend it even further. Adaptive MCL (AMCL) and Augmented AMCL (AAMC) are two of the possible extensions. MCL has one major drawback, notably its computational complexity. AMCL tries to solve this by using a variable amount of particles. When the uncertainty of the position is big, like in the global positioning problem, the amount of particles is still big enough to make sure that a correct position estimate can be found. For position tracking the uncertainty is a lot lower, especially after a couple of iterations, so the amount of particles can be lowered.

AAMCL gives the AMR the possibility to solve the "kidnapped robot" problem in which the robot is moved during operation to a new unknown position, so the robot has to globally position itself again. This problem is said to be the most difficult localization problem (Thrun, Burgard, & Dieter, 2006, p. 232). This problem can be solved by injecting a number of random particles after each iteration. The implementation of this extension isn't difficult but was not a requirement for this project.

7.2 Embedded platform

Another interesting work could be to make the AMR truly embedded (not needing a laptop on the platform). This could be achieved in two different ways. One of them could be to use a single-board computer (e.g. a Raspberry Pi) sending and receiving data via UDP, from an external computer (for the computational power). Commands and sensor data could thus be transmitted wirelessly, which will make the platform more mobile than the current one.

A second method could be to use only a single-board computer on the mobile platform, making it completely autonomous. However for this to work, another localization method should be used because of the lower computational power, as described in section 6.1.

7.3 Better navigation

A possible way to improve the navigation of the AMR is to carefully study its dynamic model. To achieve this, another OA algorithm should be used (a possible method modeled is stated in section 6.3). This will make it possible to plan several navigation steps ahead in the future and make the navigation smoother. Another improvement could be to detect moving obstacles (and not considering them as static) and to plan the AMR movement in function of the movement of those dynamic objects, as described in (Babinec, Duchon, Dekan, Paszto, & Kelemen, 2014).

8 CONCLUSION

The goal of this project was to make an AMR which is able to guide visitors of our university to a desired room. The AMR that was designed is able to do this in a multi-floor environment and capable of avoiding expected and unexpected obstacles. To do this the robot only uses maps of the building and an additional LRS. The environment in which the AMR works needed no modifications. The AMR uses several advanced and modern algorithms like Monte Carlo algorithm for localization, Voronoi diagrams to layout possible paths, Dijkstra's algorithm to select the shortest path and a tuned VPH to avoid local obstacles. The selection and implementation of these algorithms was successful. All these techniques combined give the robot the ability to function autonomously. As the methods developed are not specific to the robot or its environment, they could form the basis for further applications.

In order to make the AMR accessible to people with no or limited knowledge of robotics, it was important to make the robot as user-friendly as possible. This has been achieved by implementing a simple start-up method, by adding sound feedback to update the user about its progress and by providing the robot the ability to go to its charging station automatically.

For our project it was important to use the iRobot Platform which can be applied for educational purposes; all of the features implemented in our project can be used for bigger, more advanced robots such as autonomous wheelchairs or other applications which can make a substantial difference to the life of many people.

9 AKNOWLEDGEMENTS

We would like to thank a couple of individuals and our institute which made this project successful.

First of all we would like to thank our supervisor, Peter Slaets, for his elaborate and precious advice, ample availability, and quick replies when needed.

We are grateful to our co-supervisor, Luc Vandeurzen. His insights in the UDP protocol helped us greatly to solve the struggles we were dealing with.

We also thank KU Leuven for the resources and material provided to make this project successful and more specifically the Department of Mechanical Engineering for access to their robot laboratory. A special thanks also to Lin Zang, who helped us implementing the LIDAR communication and who was always willing to help in the robot lab.

10 REFERENCES

Al-Sagban, M., & Dhaouadi, R. (2011). Reactive navigation algorithm for wheeled mobile robots under non-holonomic constraints. *IEEE International Conference on Mechatronics* (pp. 504-509). Istanbul: IEEE.

An, D., & Wang, H. (2004). VPH: a new laser radar based obstacle avoidance method for intelligent mobile robots. *Intelligent Control and Automation* (pp. 4681 - 4685 Vol.5). Hangzhou, China: IEEE.

Andert, F., & Goormann, L. (2007). Combined Grid and Feature-Based Occupancy Map Building in Large Outdoor Environments. *Conference on Inmtelligent Robots and Systems.* San Diego: IEEE/RSJ International.

Babinec, A., Duchon, F., Dekan, M., Paszto, P., & Kelemen, M. (2014). VFH*TDT (VFH* with Time Dependent Tree): A new laser rangefinder based obstacle avoidance method designed for environment with non-static obstacles. *Robotics And Autonomous Systems*, 1098-1115.

Binggeli, C. (2013). *Materials for Interior Environments.* Wiley.

Borenstein, J., & Koren, Y. (1991). The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 278 - 288.

Burgard, W., Cremers, A. B., Fox, D., Hahnel, D., Lakemeyery, G., Schulz, D., . . . Thrun, S. (1998). The Interactive Museum Tour-Guide Robot. *AAAI-98* (pp. 11–18). Madison: AAAI.

Corke, P. (2011). *Robotics, Vision and Control: Fundamental Algorithms in MATLAB®.* Berlin: Springer-Verlag Berlin Heidelberg.

Cox, I. J. (1991). Blance - An Experiment in Guidance Navigation of an Autonomous Robot Vehicle. *IEEE Transactions on Robotics and Automation*, 193-204.

Dellaert, F., Fox, D., Burgard, W., & Thrun, S. (2001). Robust Monte Carlo localization for mobile robots. *International Conference on Robotics and Automation (ICRA-99).* Detroit: IEEE.

Esposito, J. M., Barton, O., Koehler, J., & Lim, D. (2011). *"Matlab Toolbox for the Create Robot"*. Retrieved from http://www.usna.edu/Users/weapsys/esposito/roomba.matlab/

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. *Robotics & Automation Magazine, IEEE (Volume:4 , Issue: 1)*, 23 - 33.

Fox, D., Burgard, W., Dellaert, F., & Thrun, S. (2001). Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. *Artificial Intelligence*, 99-141.

Holm, J. M., & Pedersen, S. L. (2008). *Autonomous Wheelchair Navigation.* Aalborg.

iRobot. (2005, December 8). *iRobot Roomba Serial Command Interface (SCI) Specification.* Retrieved from

http://www.ecsl.cs.sunysb.edu/mint/Roomba_SCI_Spec_Manual.pdf

iRobot. (2007, October 1). *iRobot Roomba 500 Open Interface (OI) Specification*. Retrieved from iRobot: http://irobot.lv/uploaded_files/File/iRobot_Roomba_500_Open_Interface_Spec.pdf

Jianwei, G., Yulin, D., Yiming, M., & Guangming, X. (2007). VPH+: An enhanced vector polar histogram method for mobile robot obstacle avoidance. *Mechatronics and Automation* (pp. 2784-2788). Harbin, China: IEEE.

Kurt, T. E. (2006). *Hacking Roomba*. Indianapolis, Indiana: Wiley Publishing, Inc.

Kwok, C., Fox, D., & Meila, M. (2004). Real-time Particle Filters. *Proceedings of IEEE*, 469-484.

Open Source Robotics Foundation. (2015, April 20). *Turtlebot 2*. Retrieved from Turtlebot: http://turtlebot.com/

Oroko, J. A., & Nyakoe, G. N. (2012). Obstacle Avoidance and Path Planning Schemes for Autonomous Navigation of a Mobile Robot: A Review. *Mechanical Engineering Conference on Sustainable Research and Innovation.* Juja Kenya: AICAD.

RADHAR. (2015, April 15). *About*. Retrieved from RADHAR: http://www.radhar.eu/about

Randell, C., & Muller, H. (2001). Low Cost Indoor Positioning System. *Ubicomp 2001: Ubiquitous Computing*, 42-48.

Rekleitis, I. M. (2002). *A Particle Filter Tutorial for Mobile Robot Localization.* Montreal.

RoboPeak. (2015, April 22). *Specification*. Retrieved from RPLidar RoboPeak: http://rplidar.robopeak.com/specification.html

Salmond, D., & Gordon, N. (2005). An Introduction to Particle Filters.

Salzberger, C., Daisy Fan, K.-Y., & Kress-Gazit, H. (2014, May 2). *MATLAB-based Simulator for the iRobot Create*. Retrieved from http://verifiablerobotics.com//CreateMATLABsimulator/createsimulator.html

Shrestha, S. (2012, May 29). Ray Casting for Implementing Map based Localization in Mobile Robots. MATLAB.

Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). *Introduction to autonomous mobile robots* (2nd ed.). London: MIT press.

Thrun, S., Burgard, W., & Dieter, F. (2006). *Probabilistic Robotics.* Cambridge, MA: The MIT Press.

Tribelhorn, B., & Dodds, Z. (2007). Evaluating the roomba: A low-cost, ubiquitous platform for robotics research and education. *Robotics and Automation* (pp. 1393-1399). Rome, Italy: IEEE .

Tsai, C.-C. (1998). A Localization System of a Mobile Robot by Fusing. *IEEE Transactions on Instrumentation and Measurements*, 144-149 .

Tse, P., Lang, S., Leung, K., & Sze, H. (1998). Design of a Navigation System for a Household Mobile Robot Using Neural Networks. *1998 IEEE International Joint Conference on Neural Networks Proceedings* (pp. 2151-2156). Anchorage, AK: IEEE.

U.S. Department of Defence. (2008). *GPS Performance Standard.* Washington: United States Department of Defense.

11 APPENDICES

Appendix I. Motion Model
Appendix II. Sensor Model
Appendix III. Ray-casting Process
Appendix IV. Resampling
Appendix V. Parameter selection
Appendix VI. Preparing maps for robotics
Appendix VII. Cycle time main file
Appendix VIII. Send commands and receive data from the iRobot Roomba
Appendix IX. How to send LIDAR data over UDP
Appendix X. How to run the main file
Appendix XI. MATLAB code
Appendix XII. CD-ROM content

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:

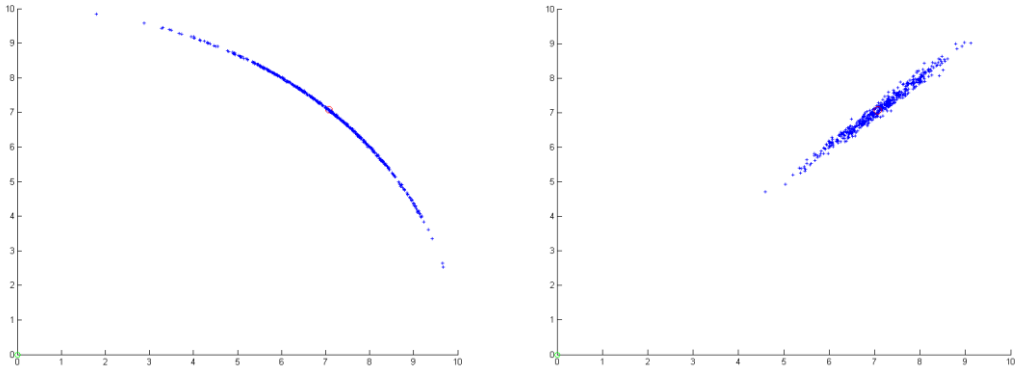
$$P' = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} \Delta s \cdot \cos(\theta + \Delta\theta + \Delta\varphi/2) \\ \Delta s \cdot \cos(\theta + \Delta\theta + \Delta\varphi/2) \\ \Delta\theta + \Delta\varphi \end{bmatrix}$$

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:

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

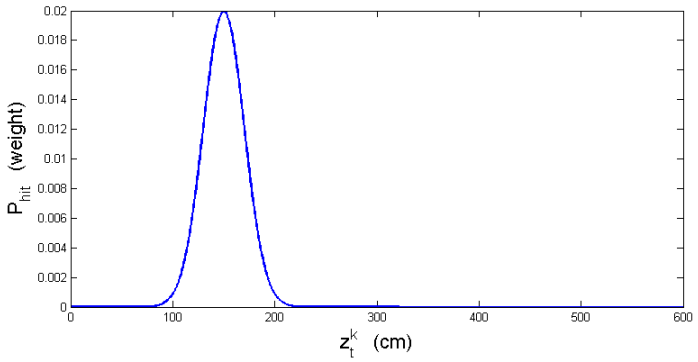


Figure 21. Correct hit distribution for a measurement of 150cm with added noise

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:

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

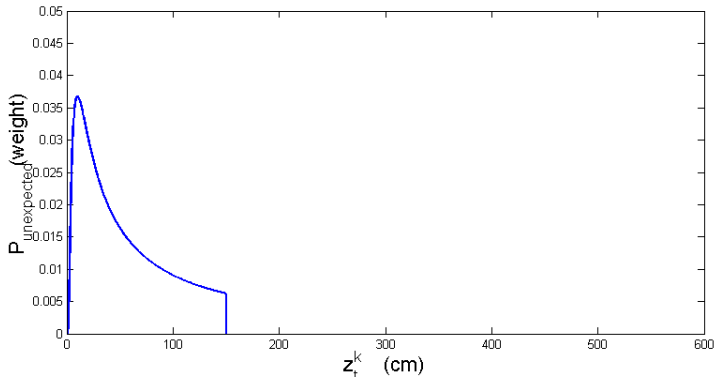


Figure 22. Unexpected hit distribution for a measurement of 150cm

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.

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

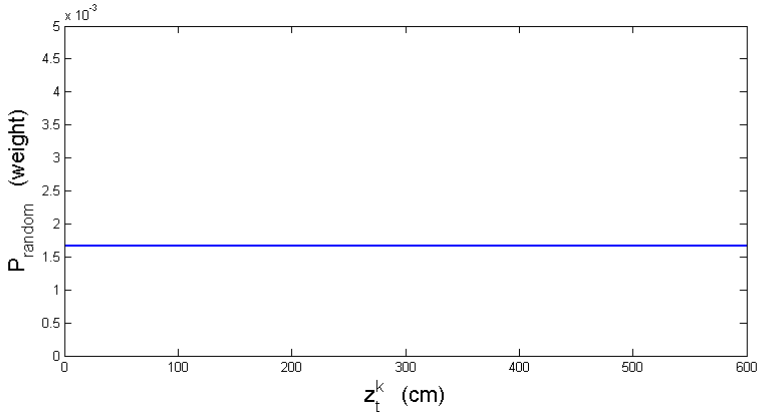


Figure 23. Random distribution

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.

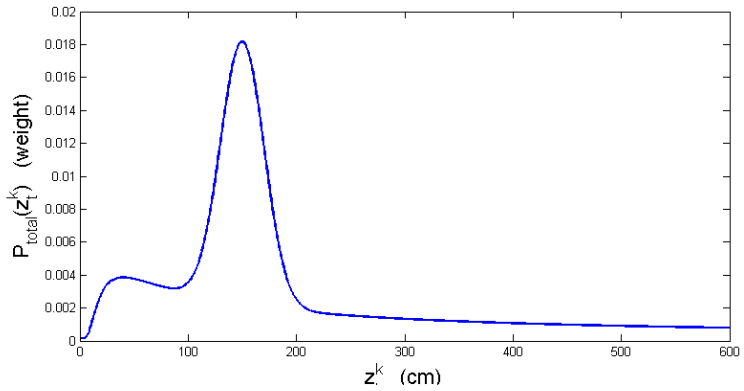


Figure 24. Total distribution of the sensor model

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.

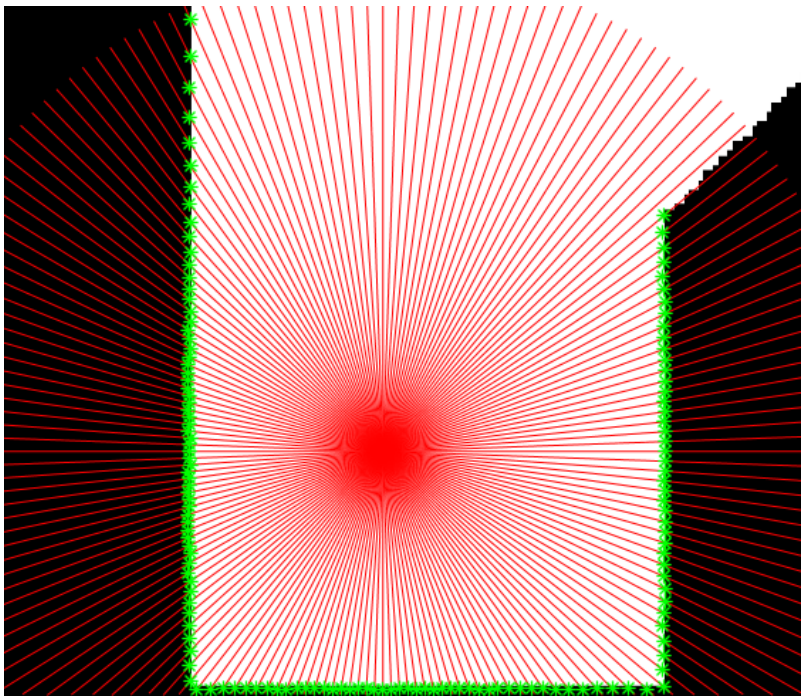


Figure 25. Detailed representation of the ray-casting process, in this figure an angular resolution of two degrees is obtained. The distance from the green crosses to the central point is calculated for each of these rays.

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).

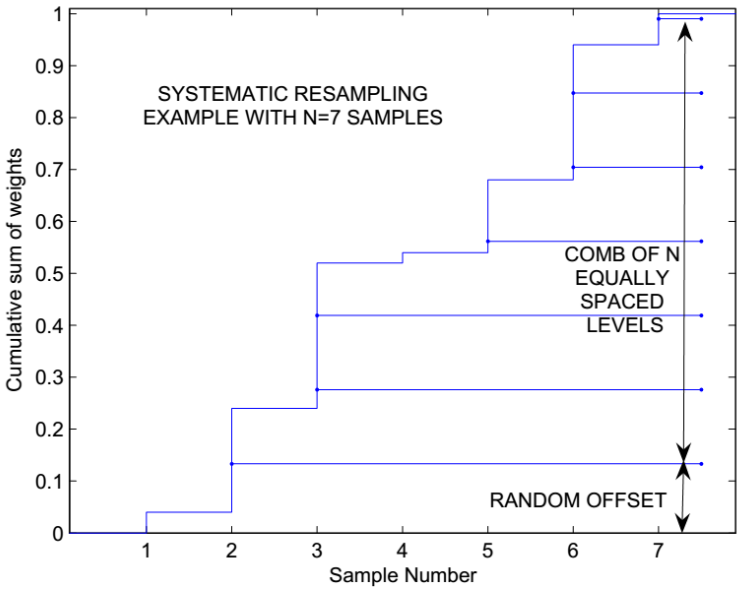


Figure 26. Figure illustrating systematic resampling (Salmond & Gordon, 2005)

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

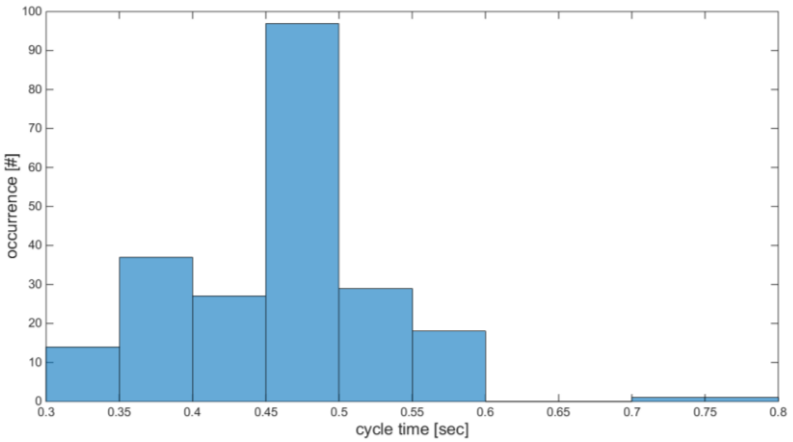


Figure 27. Total cycle time of the navigation algorithm. This cycle only contains LIDAR and sensor fetch time, localization and obstacle avoidance. Except for two long cycle times, this algorithm arrives at a cycle frequency of 2 Hz.

Onboard sensors

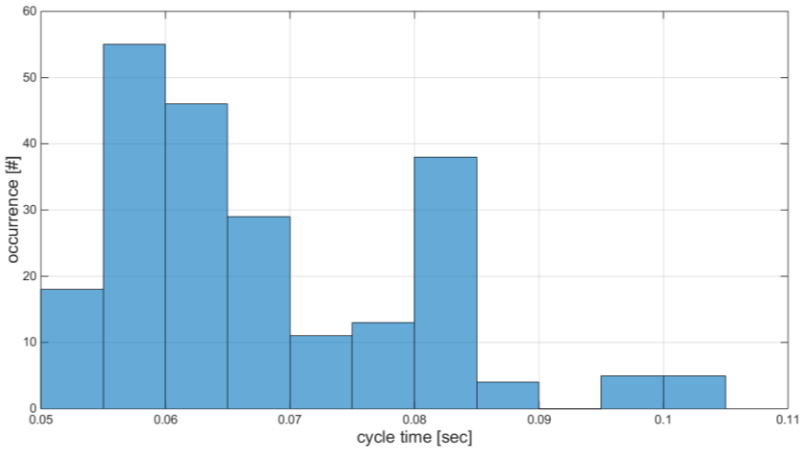


Figure 28. The cycle time to fetch the onboard sensors data is around 0.06s. The following onboard sensors are pulled: encoders, bump and drop. This is performed twice each cycle.

LIDAR

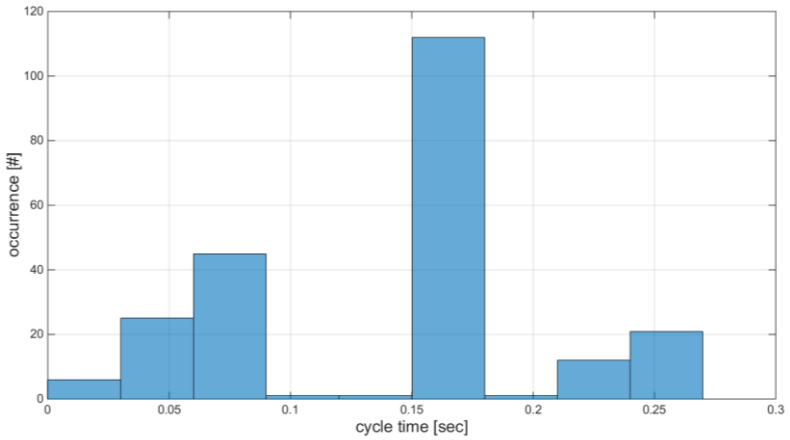


Figure 29. The LIDAR fetch cycle time is around 0.15 sec.
This cycle time is expected, since the LIDAR sends data at a rate of $\pm 5.5 Hz$.

Localization

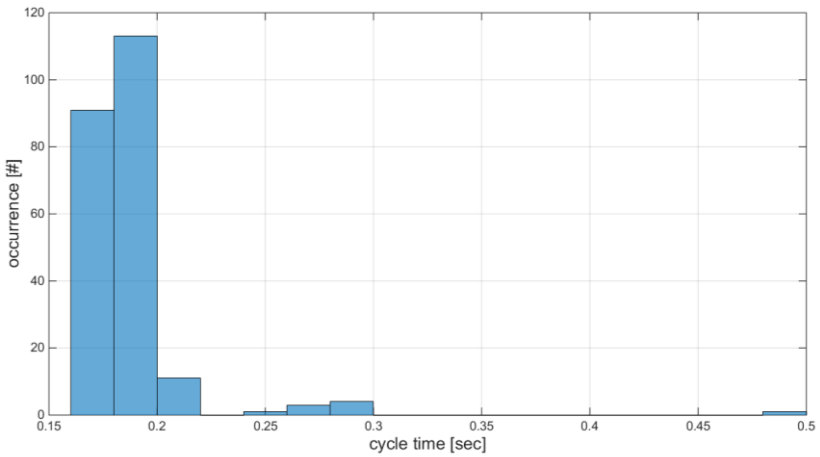


Figure 30. The localization cycle time is around 0.2 sec, this is an acceptable results for a computationally heavy algorithm. The most demanding step of the localization algorithm is the step where the real measurements are compared with the simulated ones according to the sensor model.

Navigation

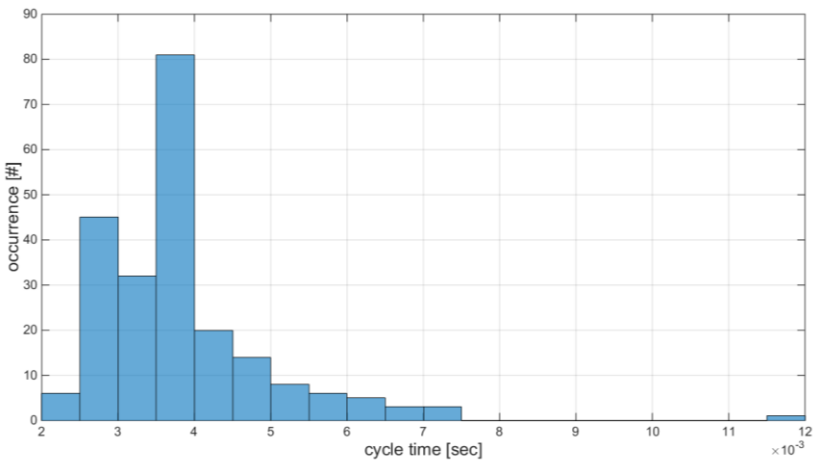


Figure 31. This time is nearly negligible, because of the low computational cost of the OA algorithm.

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.

FACULTY OF ENGINEERING TECHNOLOGY
CAMPUS GROUP T LEUVEN
Andreas Vesaliusstraat 13
3000 LEUVEN, Belgium
tel. + 32 16 30 10 30
fet.groupt @kuleuven.be
www.fet.kuleuven.be