AI in Robotics

TIENGRAU - Q4 2015

Project Report

Authors:
Jacob Harpøth Hansen | 10424
Kasper Nielsen | 10731
Kristian Mogensen | 10826

Supervisor:
Henrik Karstoft

Aarhus University - Department of Engineering

June 22, 2015

Contents

Introduction

This report contains a summary of the course material and a description of the project work from the Aarhus University reading course "AI in Robotics". The first part of the report covers the course theory, which is mainly based on the Udacity course named "Artificial Intelligence for Robotics". The Udacity course explains the concept of: localization, motion planning, motor control and SLAM, and presents some methods of how a robot can do these.

The second part covers the project work, where the theory is put into practice. Here, a robot that is capable of localizing itself in a pre-defined environment and drive to a pre-defined goal, is to be constructed. SLAM will not be implemented in the project, due to the limited time this course has and due to the complexity of the SLAM theory.

The robot is build upon the Baron-4WD mobile platform from DFRobot. This platform is by default equipped with four wheels that can be controlled individually, and is an open source hardware platform which means that there is room for mounting sensors and other things of interest.

LIDAR (LIght Detection And Ranging) is used to provide range information to the robot. Using a rotating Laser Distance Sensor (LDS) it is possible to collect range measurements in a horizontal plane with a high resolution of 1°, with a single sensor.

The project is built on a ZYBO platform from Digilent, which contains a Xilinx Zynq-7000 Processing System with a dual-core ARM processor and reprogrammable FPGA logic. Because the platform have FPGA logic, it is possible to build customized hardware acceleration cores that can decrease execution time of the system. Therefore the challenge of the project is not only to build and program a robot that can do localization, motion planning and control, but also to learn the chosen platform.

Part I

Theory

1 Localization

In the first lesson, the problem of localization is covered. Localization is defined as “The ability for a machine to locate itself in an environment". GPS is a solution to this problem that is often used when precision is not essential, but for navigation inside a house, GPS is not precise enough. Therefore alternative sensors like a LIDAR or an ultrasonic sensor can be used. These sensors only provides raw data, so localization is necessary to convert the raw sensor data and prior knowledge about the environment, that is navigated in, into estimates of the most likely current location of a robot.

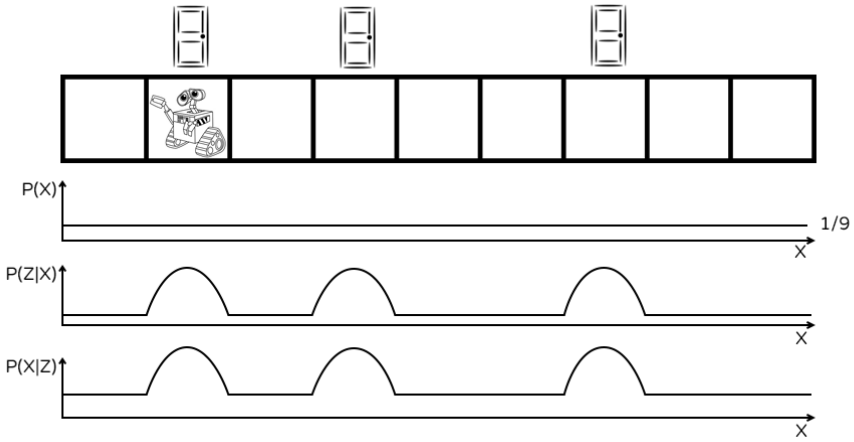
Two primary steps in a localization problem are introduced; Motion and Sensing. To improve the estimate of the robots position, sensing is performed. When sensing, raw data from a sensor is obtained, and compared with a map of the current environment to estimate the most likely current position of the robot. This is done by using Bayes rule:

$$P(X_{xy}|Z) = \frac{P(Z|X_{xy}) \cdot P(X_{xy})}{P(Z)} = \frac{P(Z|X_{xy}) \cdot P(X_{xy})}{\sum_x \sum_y P(Z|X_{xy}) \cdot P(X_{xy})} \tag{1}$$

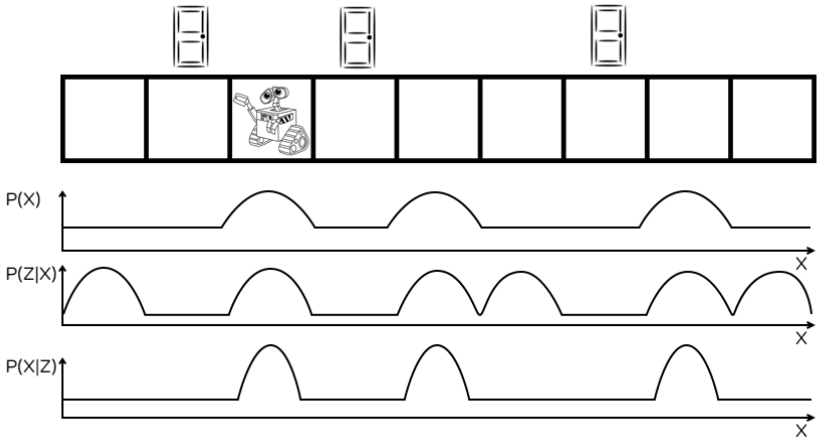
Here, $P(X_{xy})$ is the prior probability of the gridcell, X_{xy}, before the measurement, Z. $P(X_{xy}|Z)$ is the posterior probability after the measurement. $P(Z|X_{xy})$ is the probability of the measurement belonging to the specific gridcell and $P(Z)$ is a normalization factor, which can also be calculated by $\sum_x \sum_y P(Z|X_{xy}) \cdot P(X_{xy})$.

Motion is what happens when the robot moves. This adds uncertainty to the current estimate of the robots position, as there might be errors associated with moving. When turning, for example, a robot might turn 91° instead of 90°. Therefore, moving a robot makes its position more uncertain, as there is a chance of over- or undershooting a movement.

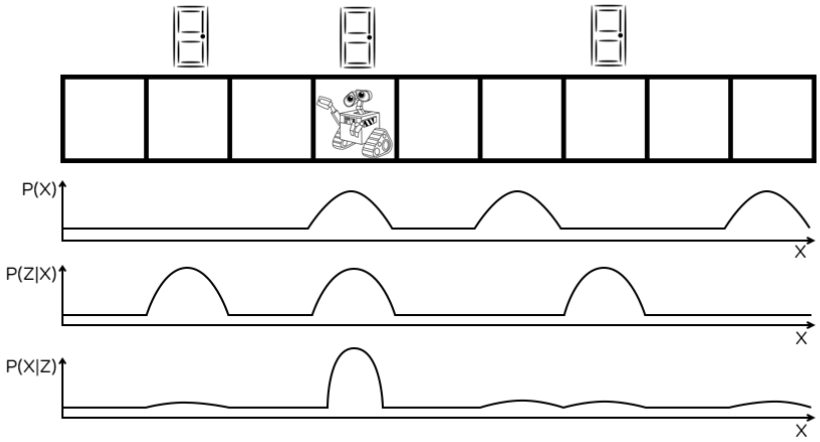
The concept of localization is best explained with an example, shown in Figure 1. These figures describes three cycles of localization. Notice how, between each figure, the previous posterior has become the new prior, with a movement and some uncertainty added. After the second movement and sensing, the robot is fairly certain it is located at the second door to the right, simply by using Bayes Theorem.



(a) Initial sensing



(b) First move and second measurement



(c) Second move and third measurement

Figure 1: Bayesian localization in three steps

2 Kalman

This lesson covers the theory about the basic Kalman Filter (KF) and the Extended Kalman Filter (EKF).

2.1 The Kalman Filter

The KF is a technique for filtering and predicting a linear system. It can be extended to also work on non-linear system, thus the EFK. The system in both KF and EKF is presented as a state-space model. Equation 2 and Equation 3 show the linear state-space model equations that can be used in the KF.

$$\mathbf{x}_k = \mathbf{A}_k\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k + \mathbf{w}_k \tag{2}$$

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k \tag{3}$$

In these equations k is the time index, $\mathbf{x}_k$ and $\mathbf{x}_{k-1}$ are state vectors at the different time instances, $\mathbf{u}_k$ is the control vector, $\mathbf{z}_k$ is the measurement vector, $\mathbf{A}_k$ is the state transition matrix of the system, $\mathbf{B}_k$ is the control matrix, $\mathbf{H}_k$ is the output measurement matrix, $\mathbf{w}_k$ is the process noise vector and $\mathbf{v}_k$ is the measurement noise vector. The two noise vector are assumed to be Gaussian zero-mean white noise with the covariances $\mathbf{Q}$ and $\mathbf{R}$, where $\mathbf{Q}$ describes the covariance for the process noise and $\mathbf{R}$ for the measurement noise. It is also assumed that the noises are uncorrelated, [5].

The basic KF operates in a cyclic manner between two states; the time update state and the measurement update state, sometimes also called the prediction state and the correction state. These two state can be compared to the ones described in Section 1; sensing and moving. Sensing matches the measurement update state, and moving matches the time update state. This and the following description of the cyclic behavior, is based on [5]. The cyclic behavior is shown in Figure 2.

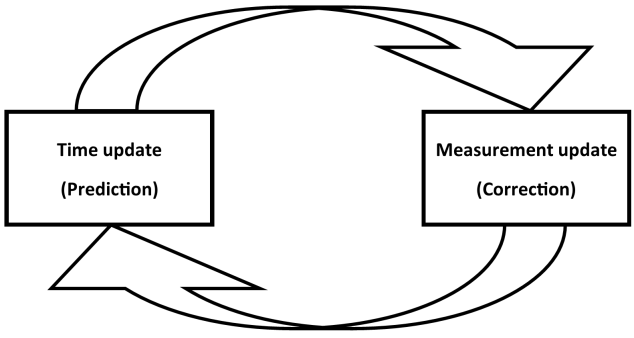


Figure 2: The cyclic behavior of the Kalman Filter

Each of these states consists of several equations. The prediction state uses the state-space model to calculate a prediction of the system state, called the a priori state, from the previous corrected system state, called the posterior state, Equation 4. A priori is marked with a $^-$, and posterior is marked with a $^+$.

$$\mathbf{x}_k^- = \mathbf{A}\mathbf{x}_{k-1}^+ + \mathbf{B}\mathbf{u}_k \tag{4}$$

Here it is assumed that the system and control matrices are constant at all time instances. It also predicts an a priori covariance of the state, that describes how certain the prediction is, Equation 5.

$$\mathbf{P}_k^- = \mathbf{A}\mathbf{P}_{k-1}^+\mathbf{A}^T + \mathbf{Q} \tag{5}$$

The correction state then starts by calculating the Kalman Gain. The Kalman Gain is a matrix of weights, that show how much belief there is in the prediction and the measurement. Equation 6 is the calculation of it.

$$\mathbf{K}_k = \mathbf{P}_k^-\mathbf{H}^T(\mathbf{H}\mathbf{P}_k^-\mathbf{H}^T + \mathbf{R})^{-1} \tag{6}$$

The Kalman Gain is then used to correct the states as shown in Equation 7, thereby calculation the posterior state.

$$\mathbf{x}_k^+ = \mathbf{x}_k^- + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}\mathbf{x}_k^-) \tag{7}$$

The a priori covariance is also corrected to the posterior covariance by using the Kalman Gain, Equation 8.

$$\mathbf{P}_k^+ = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_k^- \tag{8}$$

$\mathbf{I}$ is an identity matrix. By looking at these equations, it can be seen that a KF actually describes how the mean and the covariance of the states propagates in time. This becomes obvious when looking at an 1D example from [6].

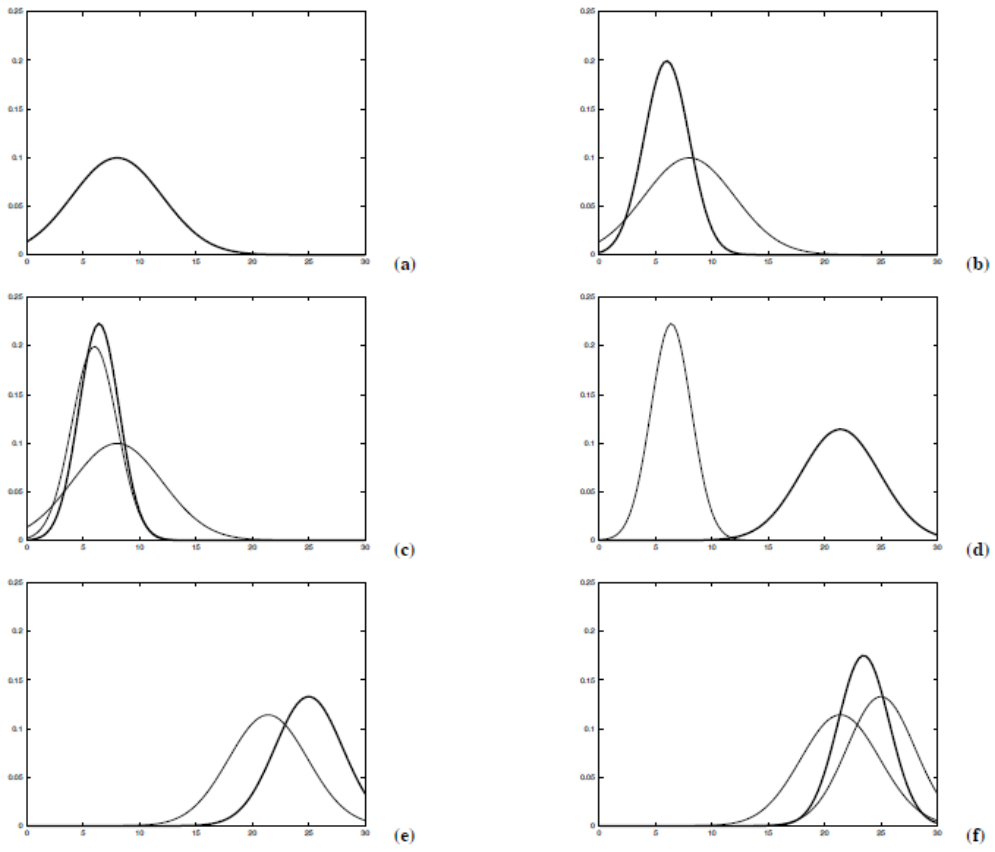


Figure 3: Kalman Filter 1D example, borrow from [6]

In this example a robot is moving in 1D. Figure 3a shows the a priori state and uncertainty, variance, of the robot. This mean that the KF is in its prediction state. The robot measures its position at Figure 3b, and the uncertainty here is described by the measurement noise variance, $\mathbf{R}$. The correction state calculates the Kalman Gain and finds the posterior state and uncertainty as seen in Figure 3c. Notice that the uncertainty now has a higher peak and smaller width. The robot then moves in Figure 3d, thereby adding uncertainty to its position; like it does in Section 1. This uncertainty is, as described in Equation 5, the process noise variance, $\mathbf{Q}$; KF is back in the prediction state. In Figure 3e, it repeats the process from Figure 3b. And in Figure 3f, it corrects itself like in Figure 3c.

2.2 The Extended Kalman Filter

The EKF is, as mentioned earlier, a KF that takes into account that a system may not be linear, which is the case in most situations. The state-space model is now present in the form as shown in Equation 9 and Equation 10.

$$\mathbf{x}_k = g(\mathbf{u}_k, \mathbf{x}_{k-1}) + \mathbf{w}_k \tag{9}$$

$$\mathbf{z}_k = h(\mathbf{x}_k) + \mathbf{v}_k \tag{10}$$

Where g and h are the non-linear functions, that now describes the transform of the prior state and the control input instead of matrix $\mathbf{A_k}$ and $\mathbf{B_k}$, and the transform of the posterior state instead of $\mathbf{H_k}$ respectively. The downside of representing a system in the non-linear manner, is that the uncertainties no longer can be presented as Gaussians. The EKF takes this into account and tries to correct it by doing linearization, which is the key concept of the EFK. By linearizing the system at every iteration, the uncertainties can again be approximated to be Gaussians. The EKF linearize by making a First Order Taylor Expansion which calculates a tangent to the system at every iteration.

The EFK algorithm is shown in Equation 11 to Equation 15.

$$\mathbf{x}_k^- = g(\mathbf{u}_k, \mathbf{x}_{k-1}^+) \tag{11}$$

$$\mathbf{P}_k^- = \mathbf{G}_k \mathbf{P}_{k-1}^+ \mathbf{G}_k^T + \mathbf{Q} \tag{12}$$

$$\mathbf{K}_k = \mathbf{P}_k^- \mathbf{H}_k^T (\mathbf{H}_k \mathbf{P}_k^- \mathbf{H}_k^T + \mathbf{R})^{-1} \tag{13}$$

$$\mathbf{x}_k^+ = \mathbf{x}_k^- + \mathbf{K}_k(\mathbf{z}_k - h(\mathbf{x}_k^-)) \tag{14}$$

$$\mathbf{P}_k^+ = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\mathbf{P}_k^- \tag{15}$$

From these equations it is obvious that they are somewhat like the KF algorithm. The biggest changes are that the non-linear functions are used instead of the state-space matrices, and that $\mathbf{G}_k$ and $\mathbf{H}_k$ are the Jacobians of the non-linear functions.

3 Particle Filter

This lesson is about the Particle Filter. Particle Filters can be used for localization by spreading particles across the map and then use them as belief. This the last localization method presented in this course.

The basic idea with the particle filter in a localization application is to choose a number of particles, generate that many particles with random poses, and make them converges towards the actual position. The basic algorithm is the following:

1. Generate a particle set χ of particles with random poses
2. Calculate the importance weights of the particles
3. Resample the particle set based on the importance weights
4. Move the particles
5. Go to step 2.

In both Section 1 and Section 2 there was the two state of operation; sensing and moving. These two states also exists in the Particle Filter, where calculation of the importance weights and resampling matches sensing, and the movement of the particles matches the moving state.

In step 2 it can be seen that all the particles have their importance weight calculated at each iteration. The importance weight is a measure of how well a particle matches the measured pose. The weight can therefore be use as the belief of the robot pose. It can for an example be calculated using a Gaussian distribution:

$$p(\mathbf{x}|\mathbf{z}) = \frac{1}{\sqrt{2\pi^2 \cdot |\mathbf{C_z}|}} \exp\left(-\frac{1}{2} \cdot (\mathbf{z}-\mathbf{x})^T {\mathbf{C_z}}^{-1} (\mathbf{z}-\mathbf{x})\right) \tag{16}$$

Where $\mathbf{x}$ is a particles pose, $\mathbf{z}$ is the measured pose and $\mathbf{C_z}$ is the measurement noise. By changing the measurement noise, the spread of the particles is also changed after resampling; the smaller the measurement noise, the smaller the spread.

Resampling is the part of the algorithm where the particles should converges towards the true robot pose. The basic idea is that particles are chosen to be in the particle set, for the next iteration, with a probability which is proportional to the particles' importance weights. There are different way to do resampling, but the method presented in this course is based upon the Low Variance Resampling algorithm. This algorithm starts by sorting all the weights in the order of the particle index. It then randomly select a particle, where all particles have the same probability of being selected. Figure 4 is an example from [6] where the randomly selected particle is at index 6.

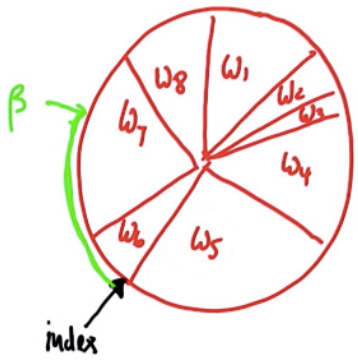


Figure 4: Low Variance Resampling example, borrowed from [7]

A β value is created using the largest importance weight and some randomness, Equation 17.

$$\beta = 2 \cdot w_{max} \cdot \mathcal{U}(0, 1) \tag{17}$$

The maximum weight is multiplied by 2 to make it possible to "jump over" the particle with that weight. The algorithm then checks if the β value is larger than the weight at index i, $\beta > w_i$. If this is the case, the weight is then subtracted from β and the index is incremented, otherwise the particle at the index is added to the particle set for the next iteration. This process repeats until a particle is added to the set, and then repeats until a full set is created. In this example β is larger than w_6, so β becomes w_6 smaller and the index become 7. β is now smaller than w_7 which means that the particle at index 7 is added to the next set. A new β is then calculate and it continues from index 7. Using this algorithm insures a circular selection of the particles, where particles with high weight have a larger chance to be selected. To summarize, the algorithm works in the following steps:

1. Randomly select an index where all particle have equal chance, $i = \mathcal{U}(1, N_{particles})$
2. Calculate β based on the maximum weight, Equation 17
3. Check if β is larger than the weight at the index, $\beta > w_i$
4. If true
   (a) Subtract the weight from β, $\beta = \beta - w_i$
   (b) Increment the index, $i = i + 1$
   (c) Go to step 3
5. If false
   (a) Add the particle to the particle set for the next iteration, *add* $particle_i$ *to* χ_{k+1}
   (b) Go to step 2

Now that the functionality of the particle filter have been described, an example of it is shown in Figure 5 where it is possible to see how the particles change over time. Figure 5a shows the initial particle set, where the height of the lines represent the importance weights. In Figure 5b a measurement have been made and the importance weight have been updated. Since the measurement shows a door, all the particles near door have a large weight compare to the rest.

Figure 5c shows the result after resampling and movement. Notice how there are three larger densities of particles that matches the particles at the different doors after being moved. Figure 5d shows the particles after another measurement and weight update. And Figure 5e is another resampling and movement. From this it is obvious to see how the particles converges toward the robot position.

The last thing that will be described is the so called "kidnapped robot" situation. This is where the particles converges toward a false pose, and the measurements suddenly does not match which means that the robot have no idea where it is since all the importance weight becomes small. The basic particle filter cannot handle this situation, but it can easily be extended so it can. One way to do it, is to only create, for an example, 90% of the particle set from resampling and add 10% newly generated particles. The number of new particles can also be chosen dynamic, like it is in the Augmented Monte Carlo Localization algorithm which is described in details in [6]. By adding new particles, there is a chance that one of these are near the true pose.

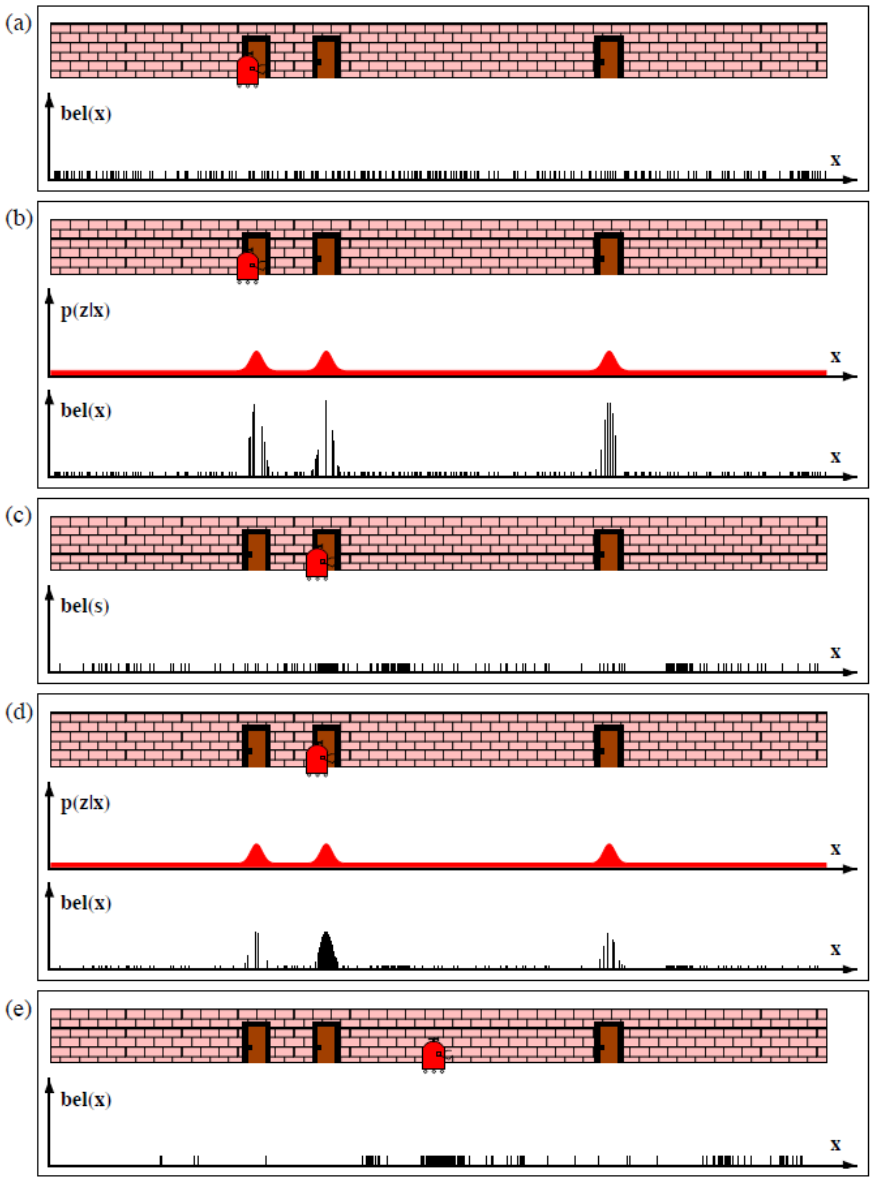


Figure 5: Particle filter example, borrowed from [6]

4 Search

This lesson covers the planning problem. The planning problem is defined as the task of finding the optimal or "minimum cost" path to reach a certain goal position, given a map, a starting location, a goal location and a cost of movement.

The cost of movement can be altered to prioritize specific paths. An example could be increasing the cost of moving next to a wall, to prioritize paths away from walls.

A naïve search algorithm, also known as Breath First Planning, is introduced, where the cost of moving from one cell to another is one. The map on which this algorithm will be demonstrated is shown below.

R	1	0	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0
0	0	0	0	0	G

Table 1: Grid map for demonstration

'0' is an empty cell, '1' is a cell containing a wall, 'R' is the location of the robot, and 'G' is the location of the goal.

The naïve search algorithm iteratively finds the next valid cell with the lowest cost, and expands it. The expansion order and the cost are shown below. Nodes that have not been visited have a value of '-1'.

0	-1	-1	-1	-1	-1
1	-1	12	-1	-1	-1
2	-1	9	13	-1	-1
3	-1	7	10	14	-1
4	5	6	8	11	15

(a) Expansion order

0	-1	-1	-1	-1	-1
1	-1	9	-1	-1	-1
2	-1	8	9	-1	-1
3	-1	7	8	9	-1
4	5	6	7	8	9

(b) Cost

Table 2: Breath First Planning

Once the algorithm reaches the goal, after 15 expansions, it stops. The path is then found by following the descending g values, the cost value, from the goal position to the start position. This creates the following path:

↓	1	0	0	0	0
↓	1	0	0	0	0
↓	1	0	0	0	0
↓	1	0	0	0	0
→	→	→	→	→	G

Table 3: Breath First Planning Path

This algorithm works, but in a large grid this algorithm becomes very ineffective.

Next A-star is introduced. A-star works in much the same way as the naïve algorithm, but the cost function is a little more nuanced. In the naïve algorithm the cost was only based on movement. In A-star the cost is the sum of the cost of movement, and a cost introduced by a heuristic function. In the lesson a simple heuristic function is proposed, where the distance-to-goal for every point in the map is calculated. The heuristic function for the presented map would look like this:

9	8	7	6	5	4
8	7	6	5	4	3
7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	0

Table 4: Heuristic map

This means the cost of cells close to the path is lower, than the cost of cells far away from the goal. The algorithm then explores the nodes with the shortest distance-to-goal first. This greatly reduces the number of nodes that have to be explored, and thus makes the algorithm more efficient. The cost and the expansion order of A-star in the map from before is shown below.

0	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1
4	5	6	7	8	9

(a) Expansion order

9	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1
9	-1	10	10	10	-1
9	9	9	9	9	9

(b) Cost

Table 5: A-star

This creates a path, which is exactly the same as the Breath First Planning, Table 3.

While the path of the naïve algorithm and A-Star is the same, the path finding algorithm of A-Star is much more efficient, as it only has to do 10 expansions, compared to the naïve approaches 15.

The last planning method is based on dynamic programming. Here the optimal path for any location on the map is pre-calculated. This might be very computationally heavy up front, but it only has to be done once. Dynamic programming would produce a map like the one below.

↓	1	↓	↓	↓	↓
↓	1	↓	↓	↓	↓
↓	1	↓	↓	↓	↓
↓	1	↓	↓	↓	↓
→	→	→	→	→	G

Table 6: Map for dynamic programming

5 PID

This lesson covers path smoothing, and control using PID controllers and a method for how the parameters can be estimated.

5.1 Path smoothing

Path smoothing is an extension to the material from previous lesson, where the generation of paths was covered. In this section it is explored how these paths can be smoothed to better fit the motion model of, for example, a car. An example of this can be seen in Figure 6.

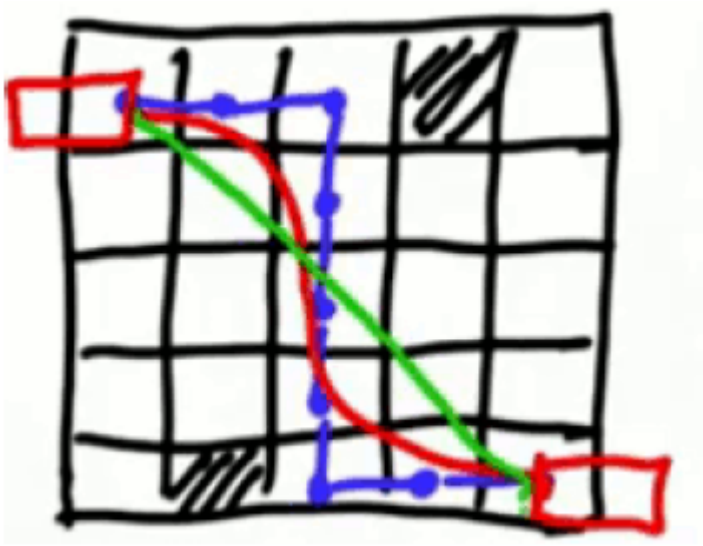


Figure 6: Path smoothing, borrowed from [7]

In this figure, the red and green paths are different smoothed versions of the blue path. The red path fits the motion model of a car much better than the blue or the green path, since a car cannot do perfect 45° or 90° turns.

The proposed smoothing algorithm works as follows:

- Given a original path, $\mathbf{p}$, which consists of multiple x,y coordinates, $p_i = [x_i, y_i]$
- Create a copy of the original path, which will hold the smoothed points, $\mathbf{s} = \mathbf{p}$
- Minimize the distance between the original path, and the smoothed path, $\min_{s_i}(p_i - s_i)^2$
- While also minimizing the distance between the consecutive points in the smoothed path, $\min_{s_i}(s_i - s_{i+1})^2$
- Using gradient decent, $s_i = s_i + \alpha \cdot (p_i + s_i) + \beta \cdot (s_{i+1} + s_{i-1} - 2 \cdot s_i)$
- Until the accumulated change is below a certain threshold

5.2 PID controllers

In this section the PID controller is covered and Cross Track Error, CTE, is introduced. Figure 7 shows a block diagram that illustrates the basic concept of a PID controller.

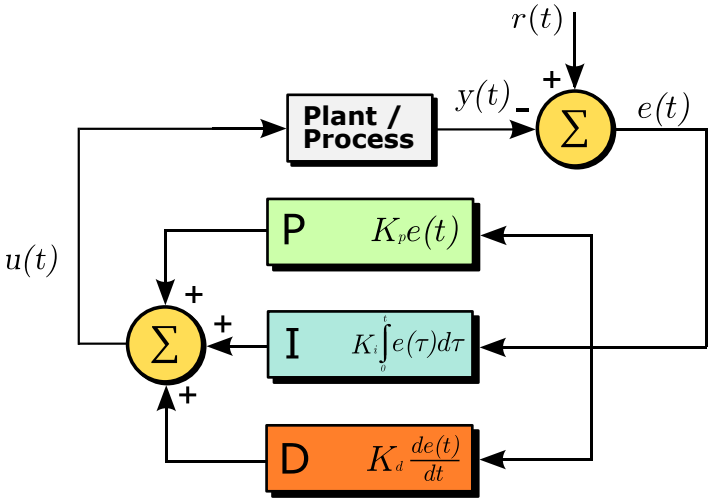


Figure 7: PID controller block diagram

A PID controller, is a controller that produces a control signal, u(t) that minimizes a certain error, e(t), by using P(proportionality), I(integration) and D(differentiation). In the examples in this course, the error is the CTE, which is the difference between the robots current position, and the expected position on the robots current path, calculated from the robots center of mass.

The different parts of a PID controlled assess different problems, and as such some parts, might be left out.

In a simple P controller, the error will only be scaled. Therefore a P controller will decrease rise time but will always introduce overshoot, and be marginally stable.

$$u = \tau_p \cdot CTE \tag{18}$$

By introducing a differentiator, the controller avoids overshoot. This is because the differentiator tries to predict "the future" and by doing this it tries to minimize the future error.

$$u = \tau_p \cdot CTE - \tau_d \cdot \frac{d}{dt} CTE \tag{19}$$

And by introducing an integrator, the bias, or *steady state error*, can be mnimized. This is done because the integrator uses "the past" to minimize the error. It therefore also helps in decreasing the rise time.

$$u = \tau_p \cdot CTE - \tau_d \cdot \frac{d}{dt} CTE - \tau_i \cdot \int CTE dt \tag{20}$$

5.3 Twiddle/Coordinate Descent

Coordinate descent is a proposed method for finding the parameters τ_p, τ_d and τ_i for the PID controller, where the parameters iteratively are corrected towards a better value.

The algorithm for coordinated decent is as follows:

1. Build a parameter vector, $\mathbf{p} = [0, 0, 0]$
2. Build a vector of potential changes, $\mathbf{dp} = [1, 1, 1]$
3. Add the potential change to the parameter at index i, $\mathbf{p}_i = \mathbf{p}_i + \mathbf{dp}_i$
4. Calculate the CTE
5. Check if the calculated CTE is smaller than the previous best, $CTE_{calc} < CTE_{best}$
6. if true
   (a) Set the calculated CTE to the best CTE, $CTE_{best} = CTE_{calc}$
   (b) Update the potential to a larger value, $\mathbf{dp}_i = \mathbf{dp}_i \cdot 1.1$
   (c) Go to step 3
7. if false
   (a) Subtract the potential change from the original parameter at index i, $\mathbf{p}_i = \mathbf{p}_i - 2 \cdot \mathbf{dp}_i$
   (b) Calculate the CTE
   (c) Check if the calculated CTE is smaller than the previous best, $CTE_{calc} < CTE_{best}$
   (d) if true
      i. Set the calculated CTE to the best CTE, $CTE_{best} = CTE_{calc}$
      ii. Update the potential to a larger value, $\mathbf{dp}_i = \mathbf{dp}_i \cdot 1.1$
   (e) if false
      i. Update the potential to a smaller value, $\mathbf{dp}_i = \mathbf{dp}_i \cdot 0.9$
   (f) Go to step 3

This is done while sum of potential changes is larger than a given threshold, and repeats for all the parameters. As can be seen, the algorithm basically start by adding a change to one of the parameter. If this gives a better CTE, it makes the change for the next iteration bigger. If it is worse, it tries to subtract the change instead. It then checks the CTE again and then either increase or the decrease the change.

6 SLAM

This lesson covers Simultaneous Localization and Mapping (SLAM). SLAM adds another layer of complexity to the problems that have been looked at so far. In the previous lessons, the map was assumed to be known. This known map has been used to lower the uncertainty of the robots current position, which the motion of the robot made larger. In reality the map is very seldom known. Therefore the robot not only has to localize itself, it also has to create the map it is localizing itself in.

In this lesson a specific form of SLAM is presented called graph SLAM. In graph slam, the initial location of the robot, relative movements of the robot and landmark locations are all used as constraints. These constraints are combined to find the most likely path the robot has traveled along with the location of the landmarks, hence SLAM.

Graph slam is a way of reducing SLAM to the solution of a linear system.

This is done by solving the following equation

$$\mu = \Omega^{-1} \cdot \xi \tag{21}$$

Here Ω is the graph slam matrix, ξ is the graph slam vector and μ is the estimated robots poses and landmarks. In the approach shown, the correspondences of landmarks are assumed to be known.

The advantage of graph slam is that it is very easy to add the constrains to the initial matrix/vector, using just additions. An initial graph slam matrix and graph slam vector can be seen in Table 7.

0	X0	X1	X2	L1	L2
X0	0	0	0	0	0
X1	0	0	0	0	0
X2	0	0	0	0	0
L1	0	0	0	0	0
L2	0	0	0	0	0

(a) Ω

X0	0
X1	0
X2	0
L1	0
L2	0

(b) ξ

Table 7: Initial Graph SLAM components

Initially there is no correlation between the points of the matrix. If the robot moves 5 distance units, and makes a new estimate of its position, the relationship in a 1D environment between the previous position X0 and the current position, X1, will be

$$X1 = X0 + 5 \tag{22}$$

Which can be written as

$$X1 - X0 = 5 \tag{23}$$

$$X0 - X1 = -5 \tag{24}$$

These constraints can be added to the initial graph components. The update components can be seen in Table 8.

0	X0	X1	X2	L1	L2
X0	1	-1	0	0	0
X1	-1	1	0	0	0
X2	0	0	0	0	0
L1	0	0	0	0	0
L2	0	0	0	0	0

(a) Ω

X0	-5
X1	5
X2	0
L1	0
L2	0

(b) ξ

Table 8: Graph SLAM components after first movement

A second move is made, moving the robot -4 distance units in the x plane. The two new equations become

$$X1 - X2 = 4 \tag{25}$$

$$X2 - X1 = -4 \tag{26}$$

And the components now look like Table 9

0	X0	X1	X2	L1	L2
X0	1	-1	0	0	0
X1	-1	2	-1	0	0
X2	0	-1	1	0	0
L1	0	0	0	0	0
L2	0	0	0	0	0

(a) Ω

X0	-5
X1	9
X2	-4
L1	0
L2	0

(b) ξ

Table 9: Graph SLAM components after second movement

It is easy to see that adding relative movement constraints can be done by using additions. Landmarks constraints are done in much the same way. A landmark is measure at X2 with a distance of 9. The two equations that represent this constrains are.

$$L1 - X2 = 9 \tag{27}$$

$$X2 - L1 = -9 \tag{28}$$

And the updated components become the ones seen in Table 10.

0	X0	X1	X2	L1	L2
X0	1	-1	0	0	0
X1	-1	2	-1	0	0
X2	0	-1	2	-1	0
L1	0	0	-1	1	0
L2	0	0	0	0	0

(a) Ω

X0	-5
X1	9
X2	5
L1	-9
L2	0

(b) ξ

Table 10: Graph SLAM components after second movement

Once all constraints has been added to the components, the solution for the best estimate can be found by solving the equation above. The material does not do a lot to actually explain how SLAM works, but a key point is that SLAM only really works when previously seen landmarks are revisited. This is shown in Figure 8, where, on the left, the uncertainty is quiet high, as the movement of the robot introduces a lot of noise. When a previous landmark is visited, and the path is closed, the uncertainty of all previous landmarks and locations are greatly reduced. This can be seen in the right part of the figure.

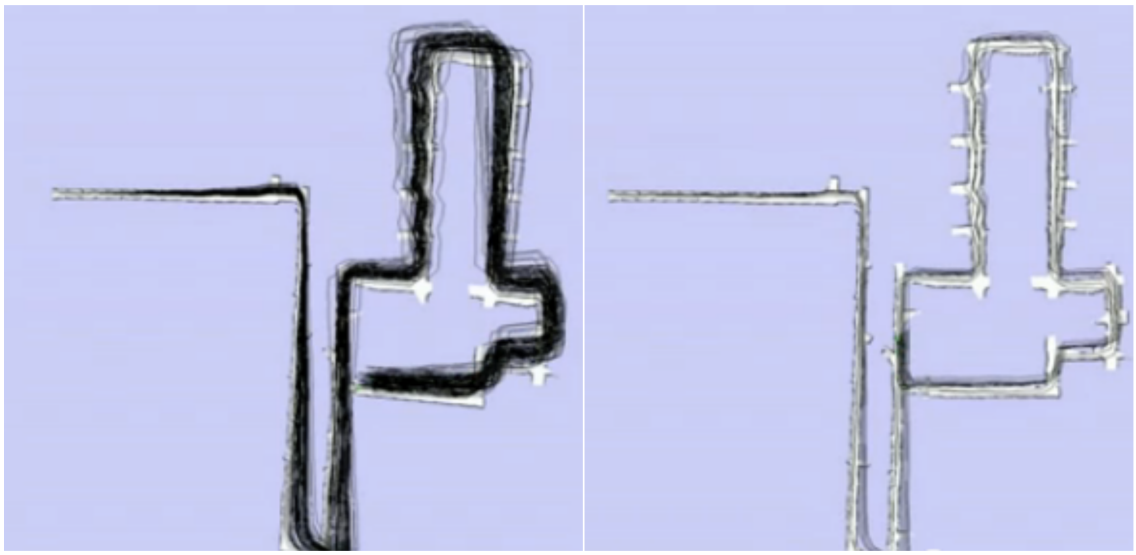


Figure 8: SLAM in action.

Part II

Project

7 System Description

This section uses SysML to describe the system from a structural and a behavioural perspective. For the structural description, a Block Definition Diagram and a Internal Block Diagram are used. For the behavioural perspective, an Activity Diagram is used.

7.1 Structure

The structural description is used to describe what parts the system is composed of, and how the parts communicate with each other. To depict this, a System Block Diagram is used to show composition, and a Internal Block Diagram is used to show dataflow. These can be seen in Figure 9 and Figure 10.

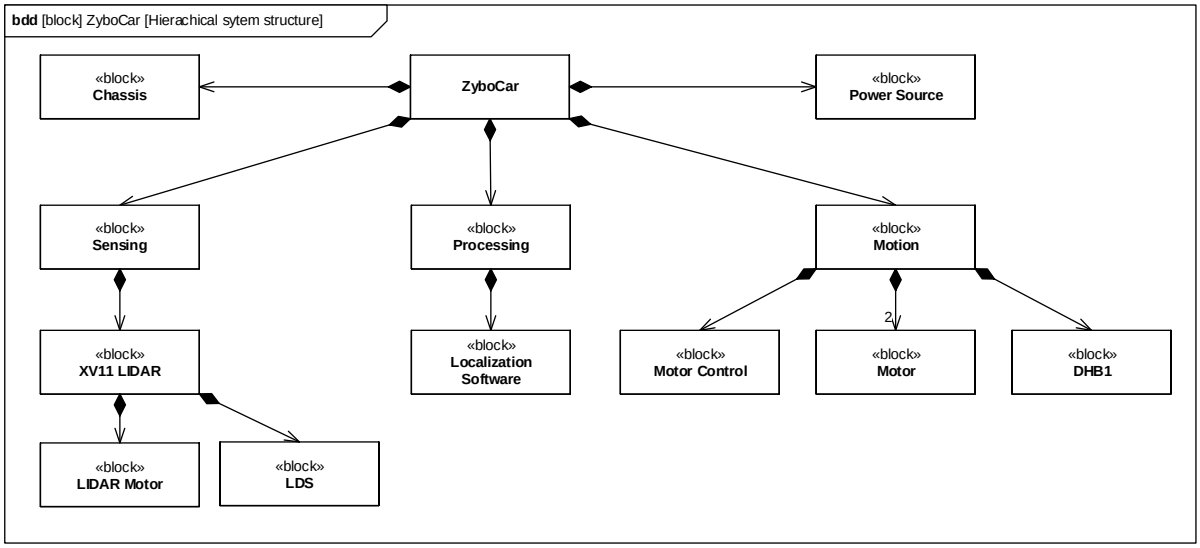


Figure 9: System Block Diagram

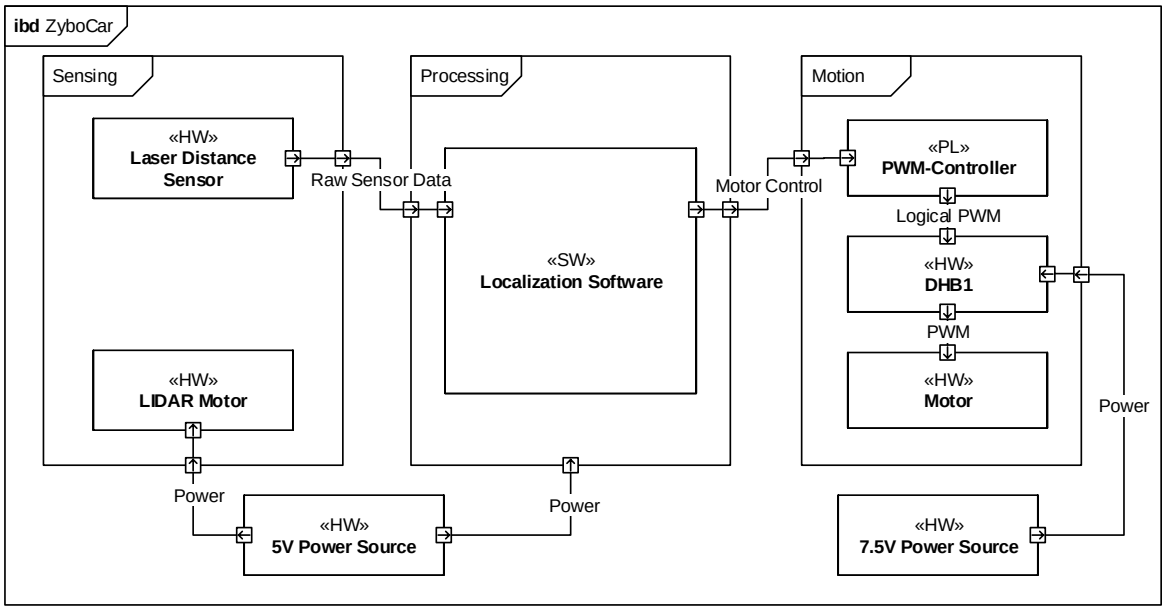


Figure 10: System Internal Block Diagram

The central block of the system is the ZyboCar, which all other blocks are composed of. The sensing block contains the XV11 LIDAR, which is the sensor used for measuring distances at 360° around the robot. This raw sensing data is transferred to the processing block, which uses the sensor data to construct a control signal for the motion block. The motion block contains a motor controller, two motors, and a Dual H-Bridge block. When the motion block receives a control signal from the localization software, the motor controller converts the signal to a logical PWM signal. The Dual H-Bridge then converts the logical PWM signal to an analoge PWM signal that can be used to control the Motor.

7.2 Behaviour

The behavioural description is used to describe what the system should do. At the behavioural level, composition is abstracted away. Instead, emphasis is put on what actions should be made and what order they should be made in. To depict this, a Activity Diagram is used. This can be seen in Figure 11.

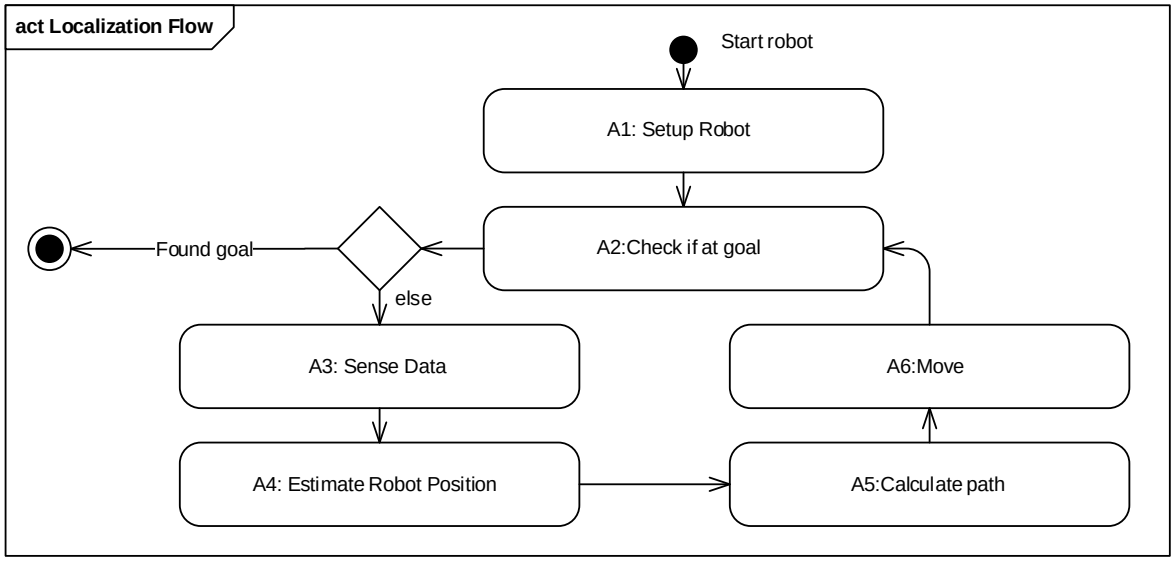


Figure 11: System Activity Diagram

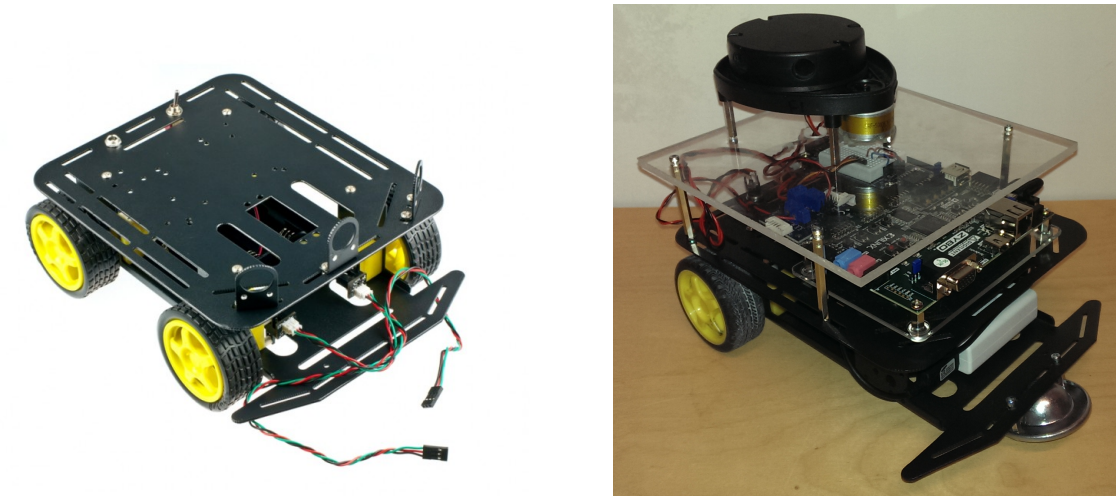
The system starts by setting up the robot, and then enters a loop that continues until the goal is found.

When the robot has not yet found its goal, it starts out by using its sensor. The sensor data is then used to estimate the most probable current location of the robot. With basis in this location, a path to the goal is generated. Finally, the robot moves a step along the found path, and checks if it has reached the goal. If not, it continues in the loop. Otherwise the activity ends.

8 Robot Mechanics

8.1 Chassis

The chassis of the robot is a Baron-4WD mobile platform bought from the website dfrobots.com. It consist of a lower frame where up to four DC motors can be attached, and a top plate with holes for mounting of other devices. Figure 12a shows a picture of the chassis where it is assembled with the default package items from dfrobots [1].



(a) The Baron-4WD platform

(b) The final robot configuration

Figure 12

In this project it was decide to only use two motors at the rear end of the chassis to make a simpler motion model for the robot; more about the motion model in Section 8.3. Instead of front wheels, a ball caster was attach to insure the turning ability. By only using motors in the rear end, the lower frame had a lot of room where the power supplies could be place and thereby leaving the top plate free for the processing platform, a Zynq development board, Section 9.1, and mounting of the sensors. The sensor used in this project is, as mentioned earlier, a LIDAR which required a specific mounting pattern, that filled out most of the top plate's area, thereby only leaving a small area for the ZYBO. More about the LIDAR in Section 9.2. The LIDAR was therefore mounted on a acrylic plate which then was mounted on top of the top plate. The final robot configuration is shown in Figure 12b.

8.2 Motor characteristics

In order to have an estimate of wheel speed related to input voltage for use when designing a motion model (see Section 8.3), a small experiment was carried out. In this, a motor was powered in the same way i would be in the final robot (see Section 9.3) and the steady-state rotation speed was recorded for different duty-cycles. The results of this is shown in Figure 13 below.

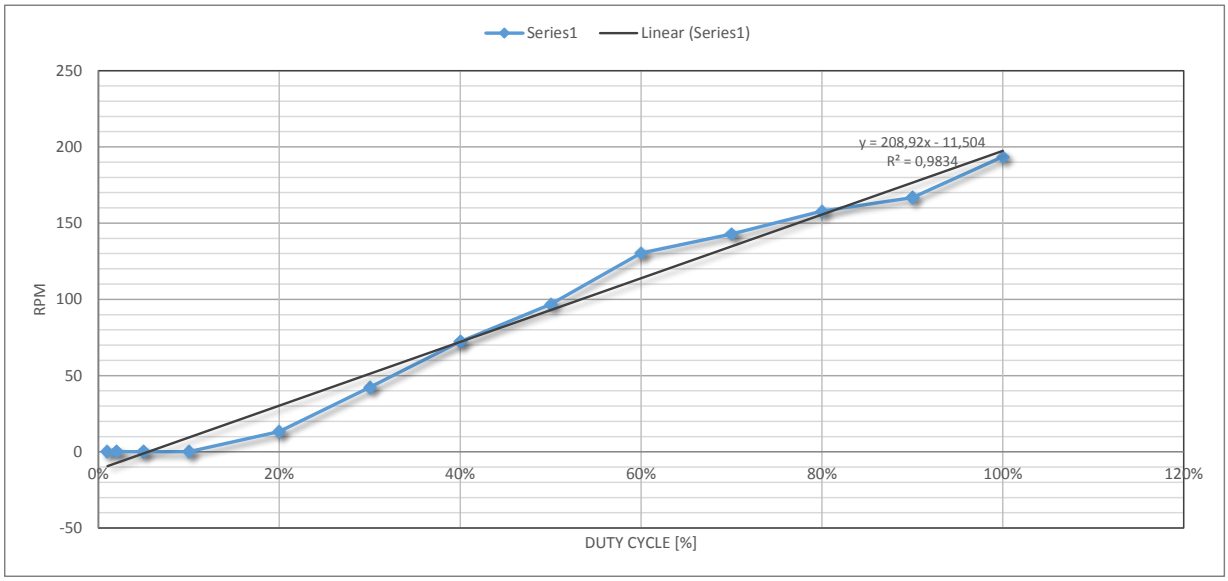


Figure 13: Motor rotational velocity [RPM] vs. PWM duty cycle [%]

Figure 13 shows that the speed of the motor, overall, changes linearly with input voltage. It is, however, apparent that at very low voltage the torque is not sufficient to get the motor to rotate.

The two major takeaways from this experiment and motor theory in general is:

1. It is most optimal to operate in the mid to high range of input voltage, as this is the most linear region of operation.
2. The motor input voltage may preferably be controlled by a PID controller to set a certain speed, instead of an open loop approach where transient acceleration due to inertia is more pronounced.

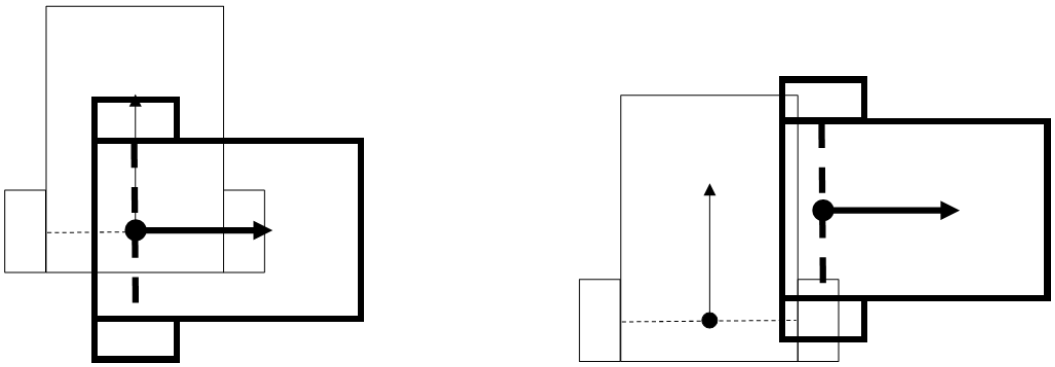
8.3 Motion model

The motion model used in this project is based on a simple rotation and the move model. The rotation is however not just a change in orientation since the robot is not rotating around it's center of mass, but around one of it's wheels. This means that during rotation the center of mass is also being moved. The difference between these two rotations is shown in Figure 14. Where the thin lined figures are the robot before rotation, and the thick figures are after.

To describe how the center of mass is being move when turning around the wheel, the following equation from [8] is used, which is just a 2D rotation matrix.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{29}$$

In this equation x and y are the coordinates before the transformation, x' and y' are after transformation, and θ is the angle of how much the orientation is being changed.



(a) Around center of mass (b) Around wheel

Figure 14: Rotation models

The movement is described simple by the change in the coordinates, Equation 30.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} d & 0 & x \\ 0 & d & y \end{bmatrix} \begin{bmatrix} \cos\phi \\ \sin\phi \\ 1 \end{bmatrix} \tag{30}$$

Where x, y, x' and y' is the same as previous, ϕ is the orientation and d is the distance of the movement.

To set a certain rotation (θ) or distance (d) one or two motors are set to run at a certain speed, and for a time linearly proportional to the change. This motion model assumes constant velocity, which means that the robot accelerates to its final speed instantly. This is of course impossible, but the assumption does break the model completely as the speed we are moving at is very low and so is the inertia of the robot.

Preferably the motor speed would be controlled by a PID controller, in order to have maximum torque when accelerating, making the true robot motion more similar to the model. This is however not implemented in this project because of time constraints. In stead the motion suffers from noise caused by the transient nature of the open loop control.

It was chosen to use this discrete model, where rotation and translation (movement) are done separately, above a continues model, where rotation is being done while translating. The reason for this was to keep model motion as simple as possible to begin with, and then extend it when a functional robot was made.

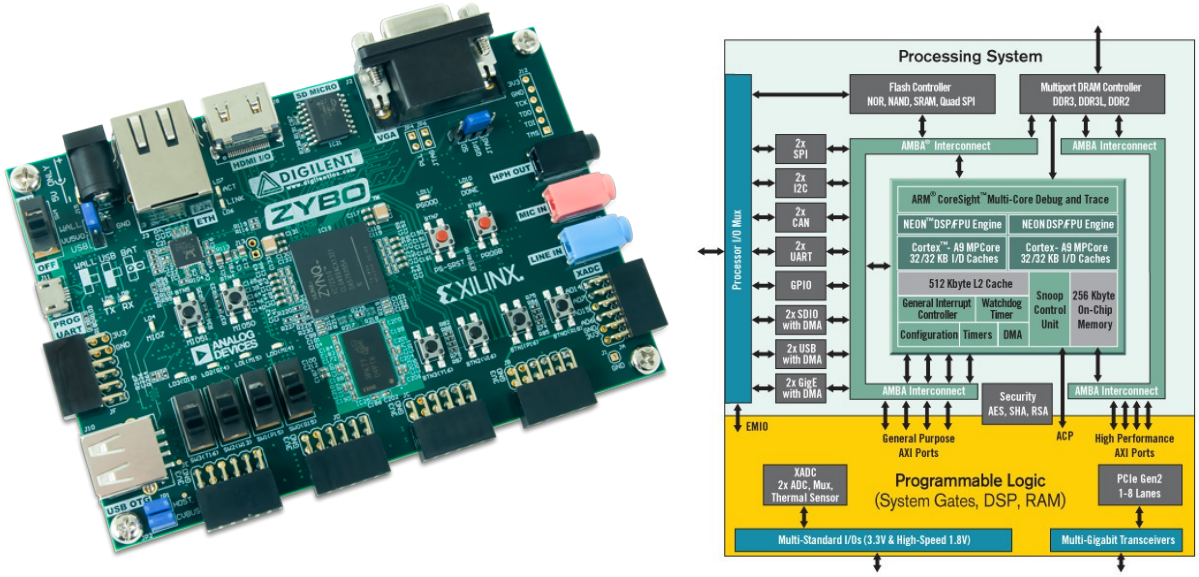
For simplicity, it was also decided that the change in orientation would by constrained to intervals of 90°. This meant that the robot in theory should only be able to drive horizontally and vertically in the grid map.

9 Hardware

9.1 Platform

The processing platform chosen for this project is a *ZYBO Zynq7000 Development Board* from Digilent (see Figure 15a). The brains of the ZYBO is a Xilinx Zynq-7000 All Programmable System on a Chip (AP SoC). The AP SoC (see Figure 15b) features both a Processing System (PS) consisting of a 650 MHz dual core ARM® Cortex A9 CPU with dedicated I/O, Memory and extensible FPGA Programmable Logic (PL) for HW synthesis.

This enables the construction of a system with software, written in C++, running on the CPU, and with dedicated hardware for acceleration of computation and I/O handling.



(a) Digilent Zynq Board (ZYBO)

(b) Zynq-7000 AP SoC Diagram

Figure 15: Figures borrowed from [3]

Alongside the AP SoC the ZYBO features a range of external hardware for power supply/management and interfacing. It has push-buttons, throw-switches and LEDs for direct interface, USB-OTG, UART-to-USB and Ethernet for communication and HDMI, VGA and an audio codec for Audio/Visual.

Lastly, the ZYBO features six PMOD GPIO ports along the perimeter (see Figure 15a), one directly connected to the PS and the others accessible through the PL. They deliver 3.3V power to external devices as well as eight customizable 1.8V or 3.3V digital I/O ports each (see Figure 16). These are used for interfacing with the sensor and motor (see Section 9.2 and Section 9.3, respectively).

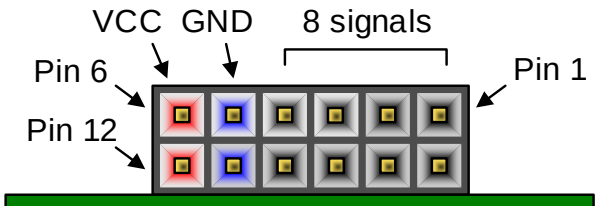


Figure 16: PMOD Connector. Borrowed from [3]

9.1.1 HW/SW Co-Design

The availability of FPGA logic on-chip enables rapid prototyping of HW acceleration blocks, through the use of High-Level Synthesis (HLS) tools. Using HLS one can specify some desired functionality in C++/SystemC code and directly synthesize it to HW-blocks to be implemented in FPGA logic, without needing to explicitly design HW in a Hardware Description Language like VHDL or Verilog.

This is utilized for creating a PWM motor controller, described in Section 9.3.2. A desciption of the final design of the processing hardware include PS and designs in PL is found Section 9.4.

9.2 Sensors

The robot is designed to use a LIDAR (Light Detection and Ranging) sensor for sensing its location. The sensor chosen for this robot is a Neato XV-11 LDS (Laser Distance Sensor) (seen mounted atop the robot in Figure 12b). This sensor consists of a 1 dimensional LDS mounted on a rotating platform, enabling it to take measurement in a 2 dimensional plane in 1° increments. The data is measured in mm at a numeric range of 0-16384 (14 bits) and an operating range of ~150-6000 mm depending on lighting conditions and reflectivity of objects.

The top houses the sensor and computation unit. This measures the distance, measures rotational velocity of the sensor related to the base and handles communication of data. It requires a 3.3V power supply and consumes about 145 mA of current when operating and communicates via serial-communication (see Section 9.2.1). The top has four wires, connected via a slip ring. The pinout is shown in Table 11a.

The base houses the motor that drives the rotating top. The motor is designed to be powered by a pulse width modulated 12V power supply, but in this project the motor is power by a 3.3V continuous supply This consumes approximately 60 mA.

Wire	**Signal**
Red	+3.3V
Brown	LDS_RX
Orange	LDS_TX
Black	GND

(a) LDS top pinout

Wire	**Signal**
Red	+3.3V
Black	GND

(b) LDS motor pinout

Table 11

9.2.1 Sensor data protocol

The LDS transmits data via 3.3V RS-232 at 115200 BAUD. It uses 8 databits, no parity and one stop bit (**8N1**). Data is transmitted continuously while the sensor rotates, in packages of four readings, totaling in 90 packages per turn [4].

Each package has 22 bytes in the following format:

```
<start> <index> <speed_L> <speed_H> [Data 0] [Data 1] [Data 2] [Data 3] <checksum_L> <checksum_H>
```

Where:

- `start` is always 0xFA
- `index` is the index byte in the 90 packets, going from 0xA0 (packet 0, readings 0 to 3) to 0xF9 (packet 89, readings 356 to 359)
- `speed` is a two-byte information, little-endian. It represents the speed, in 64th of RPM (aka value in RPM represented in fixed point, with 6 bits used for the decimal part)
- `[Data 0]` to `[Data 3]` are the 4 readings. Each one is 4 bytes long, and organized as follows
  - `<byte 0> : <distance 7:0>`
  - `<byte 1> : <"invalid data" flag> <"strength warning" flag> <distance 13:8>`
  - `<byte 2> : <signal strength 7:0>`
  - `<byte 3> : <signal strength 15:8>`

Data is only sensed and transmitted if the sensor is spinning at more than 180 PRM. If the sensor exceeds 320 RPM the data becomes sparse, as only every one in two measurements are non-zero and valid. Above 349 RPM the serial interface becomes a bottleneck and data is corrupted and packages are lost.

Because we operate at near or above the 320 RPM limit we only the even numbered measurements are used. This way corruption will not occur with a varying amount of valid measurements being collected.

9.3 Motor Control

The motor control hardware consists of two major blocks, an external Dual H-Bridge for driving the the motors, and an on-chip PWM controller for controlling the motor driver. The connection between the two blocks are shown in the SysML internal block diagram of the Motion block (Figure 17), along with the two motors. The *Power*-signal comes from a battery (see Section 9.5), and *Motor Control* is connected to the Processing System.

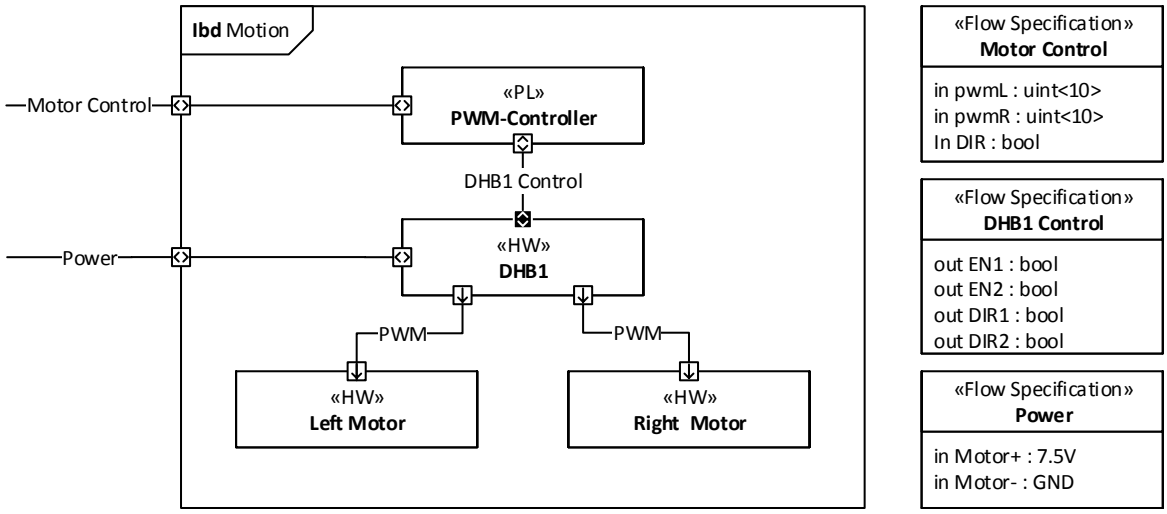


Figure 17: SysML Internal Block Diagram of the Motion Block

9.3.1 External Dual H-Bridge

The *DHB1* from Digilent is a dual- H-bridge motor controller that enables regular DC-motors to be driven in either direction with with a single ended power supply. Further it provides logic level transistor switching so that higher voltages (up to 11.8V) and higher currents (1.5A RMS, 2A peak) can be used to drive the motor without loading the logic circuit.

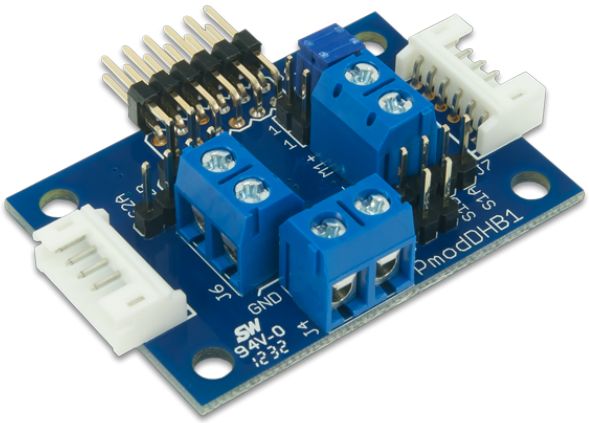


Figure 18: Digilent PmodPHB1 Dual H-Bridge Motor controller [2]

The DHB1 connects to the ZYBO via a PMOD connector. The individual signals are listed in Table 12a. The motor power source is connected to the DHB1 with screw terminals. The two motors are also connected with screw terminals, but the left motor is connected with reversed polarity. This is because the identical motors need to spin in different directions to achieve the same rotation in relation to the direction of travel.

Pin	Signal	Description
1	EN1	Motor 1 Enable
2	DIR1	Motor 1 Direction
3	S1A	Motor 1 Sensor A Feedback
4	S1B	Motor 1 Sensor B Feedback
5	GND	Power Supply Ground
6	V_{CC}	Power Supply (3.3V)
7	EN2	Motor 2 Enable
8	DIR2	Motor 2 Direction
9	S2A	Motor 2 Sensor A Feedback
10	S2B	Motor 2 Sensor B Feedback
11	GND	Power Supply Ground
12	V_{CC}	Power Supply (3.3V)

(a) PMOD connection description

DIR1	EN1	Result M1
0	0	Stop
0	1/PWM	Forward
1	0	Stop
1	1/PWM	Reverse

DIR2	EN2	Result M2
0	0	Stop
0	1/PWM	Forward
1	0	Stop
1	1/PWM	Reverse

(b) Truth table for DHB-1 input

Table 12

To control the direction of the motor one applies a logic level to the *DIR* pin for the corresponding motor; 1 for forward, 0 for reverse. The speed of the motors are controlled by applying a PWM signal to the EN pin for the corresponding motor (see Table 12b). The sensor feedback pins are not used as there are no sensors attached to the wheels.

For more information, see the reference manual [2].

9.3.2 PWM-controller

The PWM-controller is implemented as a hardware IP-Core in the FPGA logic of the AP SoC. It has been modeled in SystemC code in the Vivado HLS tool.

Listing 1 shows the header defining the PWM-controller module, consisting of a clock dividing thread and a PWM handling thread. It also defines the interface of `//Ports`, as designed in the IBD in Figure 17.

Listing 1: PWM-Controller SystemC Header

```
#ifndef _MAIN_
#define _MAIN_

#define ONE_TICK 25 // 50MHz / 25 = 2 MHz --> ~2 kHz, 10 bit PWM

#include <systemc.h>

SC_MODULE(MotorCtrl) {

  //Ports
  sc_in <bool> clk;
  sc_in <bool> reset;

  sc_in<sc_uint<10> > pwmR;
  sc_in<sc_uint<10> > pwmL;
  sc_in<bool> Direction;
  sc_out<bool> EN1;
  sc_out<bool> EN2;
  sc_out<bool> DIR1;
  sc_out<bool> DIR2;

  //Variables
  sc_uint<10> pwmCount;
  sc_uint<32> dividerCount;
  bool DIR;

  sc_logic pwmClock;

  //Process Declaration
  void pwmThread();
  void clockDividerThread();

  //Constructor
  SC_CTOR(MotorCtrl) {
    //Process Registration
    SC_CTHREAD(pwmThread, clk.pos());
    SC_CTHREAD(clockDividerThread, clk.pos());
    reset_signal_is(reset,true);
  }
};

#endif
```

Listing 2 shows the definition of the module declared in Listing 1. The `clockDividerThread` is responsible for dividing the 100 MHz clock of the PL down to a 2 MHz clock that can be used for ticks in the `pwmThread`. The use of a 2 MHz PWM-clock is based on the choice of a 10 bit resolution and a recommendation from [2] of a 2 kHz PWM-cycle for the DHB1.

$$f_{PWMclock}\frac{f_{PL}}{ONE\_TICK} = \frac{100\text{ MHz}}{25} = 2\text{ MHz} \tag{31}$$

$$f_{PWMcycle} = \frac{f_{PWMclock}}{2^{PWM\ resolution}} = \frac{2\text{ MHz}}{2^{10}} = \frac{2\text{ MHz}}{1024} = 1.953\text{ kHz} \approx 2\text{ kHz} \tag{32}$$

`clockDividerThread` signals `pwmThread` of a PWM clock tick by setting `pwmClock` for one PL clock-cycle every time a PWM clock tick is generated.

Listing 2: PWM-Controller SystemC Definition

```
#include "main.h"

void MotorCtrl::pwmThread() {
  //Group ports into AXI4 slave slv0
  #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=pwmR
  #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=pwmL
  #pragma HLS resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=Direction

  //Initialization
  wait();

  pwmCount = 0;

  while (true) {

    if (pwmClock == true) {

      pwmCount++;

      // Set DIR
      DIR = Direction.read();
      DIR1.write(DIR);
      DIR2.write(DIR);
      // DIR1.write(Direction.read());
      // DIR2.write(Direction.read());

      // Handle pwm count
      // Right motor
      if (pwmCount < pwmR.read()) {
        EN1.write(true);
      } else {
        EN1.write(false);
      }

      // Left motor
      if (pwmCount < pwmL.read()) {
        EN2.write(true);
      } else {
        EN2.write(false);
      }
    }
    wait();
  }
}

void MotorCtrl::clockDividerThread() {

  dividerCount = 0;

  while (1) {
    wait();

    if (dividerCount++ == ONE_TICK) {
      pwmClock = true;
      dividerCount = 0;
    } else {
      pwmClock = false;
    }
  }
}
```

`pwmThread` handles PWM output by counting up a 10 bit variable everytime it receives a PWM clock tick, and compares the count with the PWM settings for the left and right motor, `pwmL` and `pwmR` respectively. If the count is below the PWM setting, the enable signal is held high, and if it is above it is held low. This way the output duty cycle is determined by:

$$DC = \frac{PWM\ setting}{1024} \qquad \text{Example: } \frac{512}{1024} = 50\% \tag{33}$$

This model was then synthesized to VHDL code using the Vivado HLS tool and imported into the overall block design. Using special `#pragma`'s the setting variables `pwmL`, `pwmR` and `DIR` are mapped onto the AXI memory bus of the AP SoC. This way they are readily addressable from the PS as memory mapped registers. The procedure for controlling the motors from software is described in Section 10.3.3.

9.4 AP SoC Block Design

Figure 19 shows the final block design for the AP SoC designed in *Vivado*. In the top left corner the Processing system is shown and below it is a reset controller. In the middle is the memory mapped AXI Interconnect Bus.

In the top right< *MotorCtrl* is the PWM controller, described in Section 9.3.2. Below *MotorCtrl* is *AXI GPIO*; a GPIO controller used to interface with the push-buttons on the ZYBO. Lastly, in the bottom right corner, *AXI Uartlite* is a UART synthesized in PL. This is used to interface the LDS sensor with the PS via the AXI Interconnect bus.

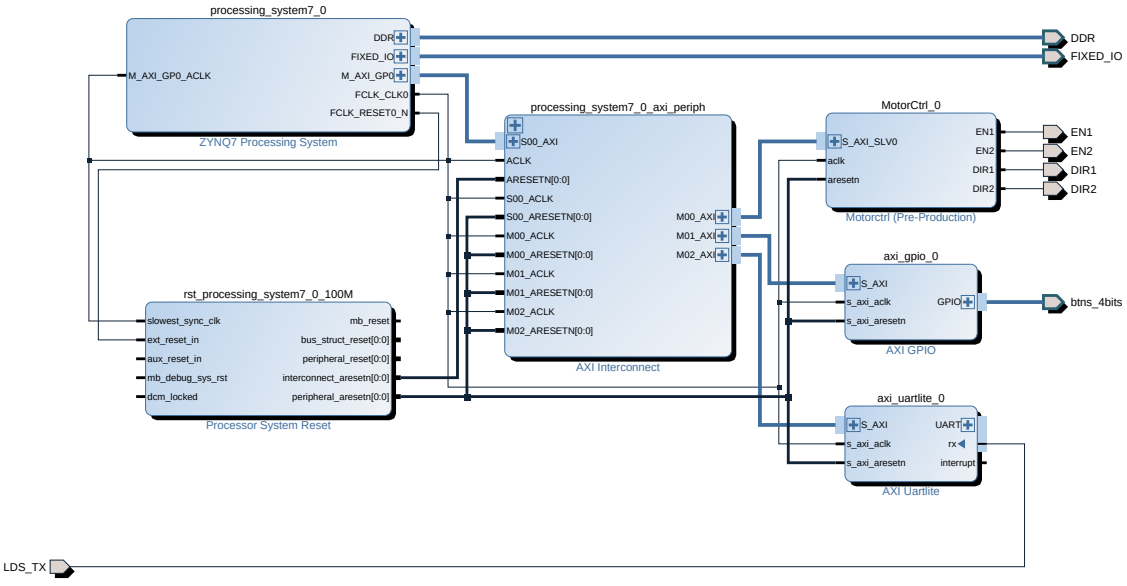
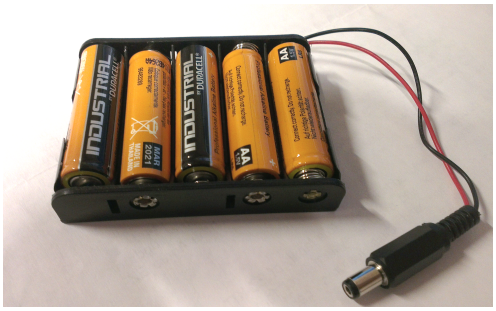


Figure 19: ZyboCar Hardware Platform Block Design

This block design is synthesized and a *hardware wrapper*, along with a *bitstream* for programming a device, is generated and exported for use with the *Xilinx SDK* IDE. The SDK is then used for writing the *bitstream* to the ZYBO and programming the PS.

9.5 Power

This project uses three different power sources: 7.5V, 5V and 3.3V. The 7.5V supply comes from a battery bank of 5 1.5V AA batteries (see Figure 20a). This is used solely for powering the motors, and is connected to the DHB1. The 5V supply comes from a micro-USB power bank (see Figure 20b). It is used to power the ZYBO.



(a) 5 AA-Battery Pack

(b) 5V USB Power Bank

Figure 20: Onboard power supplies

The 3.3V supply is supplied by the ZYBO's on-board power regulator. The ZYBO can supply a theoretical maximum of 2.5A (minus its own consumption ~ 150 *to* $400\ mA$) with a maximum of 1A supplied to each PMOD. The output is however limited by the output of the 5V power bank. Assuming little to no conversion loss the the maximum current from the ZYBO 3.3V supply is:

$$P_{PB} = I_{PB} \cdot V_{PB} = 500\ mA \cdot 5\ V = 2.5\ W \tag{34}$$

$$I_{3V3} = \frac{P_{PB}}{3.3\ V} = 757.57\ mA \tag{35}$$

The 3.3V supply is used to power the LDS top and motor through one PMOD connector and the DHB1 through another.

10 Software

In this chapter, the coding environment, design and implementation of the software will be covered. For reference, the entire codebase can be found in the attached .zip archive *ZyboCarSourceCode.zip*.

10.1 Environment

The software for this project was written in C++ using the Xilinx Software Development Kit. C++ was chosen because it is a object oriented language, which gives some clear advantages when working in a team. Furthermore, C++ has native support for an resizable array datatype known as a vector, which greatly reduced the complexity of manipulating arrays. The software was written for the C++11 standard, since this standard gave some advantages in initialization of vectors. GitHub was used for version control to manage and backup code.

10.2 Design

This section describes the design of the software system that will control the self-localizing robot. First, a description of the intended structure of the system will be presented, showing how the system level design maps to the software design. Second, a description of the software's overall behavioral design is presented.

10.2.1 Structure

A object-oriented approach was taken to the design, in order to modulize the different parts of the software.

This gave clear advantages when working multiple people on the same codebase, as the development of the different classes could be spread out between multiple people.

The structure of the software is depicted in Figure 21 as a UML class diagram.

The main components of the program is the LIDAR, PathMaker, RobotFrame and ParticleFilter classes. Each class has a distinct purpose, which maps directly to one of the four actions in the main loop of Figure 11. The mapping can be see in Table 13.

Action	Class
A2: Sense Data	LIDAR
A3: Estimate Robot Position	ParticleFilter
A4: Calculate Path	PathMaker
A5: Move	RobotFrame

Table 13: Action mapping of classes

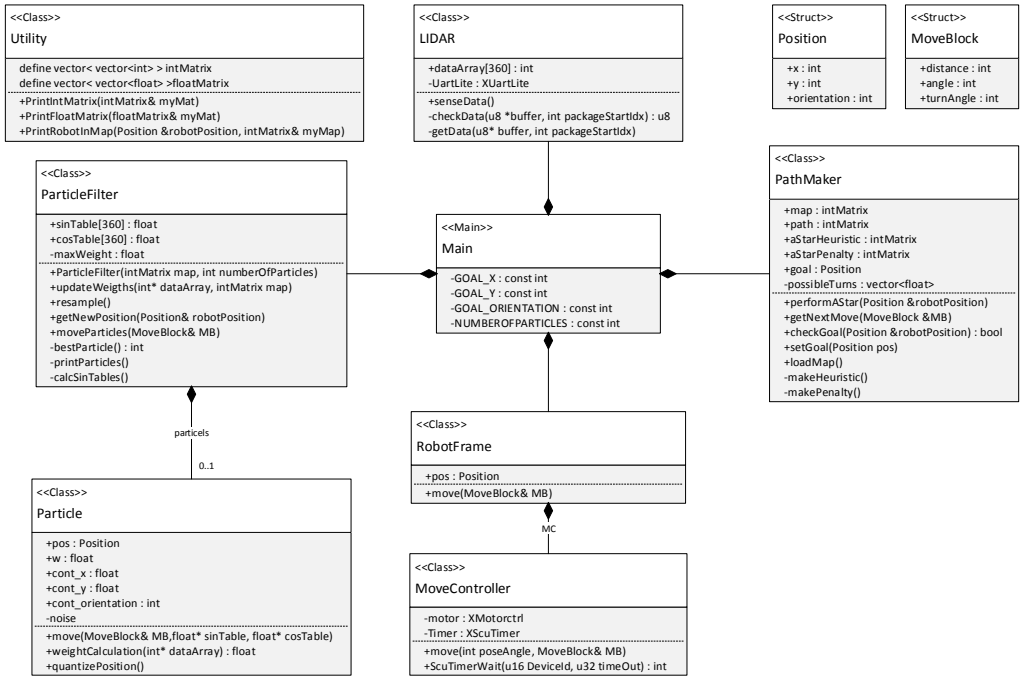


Figure 21: Class Diagram

10.2.2 Behaviour

The dataflow between the different classes is covered in the Internal Block Diagram in Figure 22.

The raw sensor data from the Sensing block in Figure 10 is sent to the LIDAR class, which converts the raw sensor data to 360 distance values. The sensor data is sent to the ParticleFilter class, which updates its current particle weights. These weights are used to resample the particles using the Low Variance Resampling technique presented in the course material, Section 3. The robots position is then estimated as position and orientation of the particle with the largest weight. This position is used in the PathMaker, which performs AStar to find the best path to the goal.

The next move from this path is sent to both the RobotFrame class and the ParticleFilter, to move both the robot and the particles along the expected path. The RobotFrame class uses the MotorControl class to send a control signal to the Motion block from Figure 10, which converts the control to a PWM signal that controls the motors of the robot.

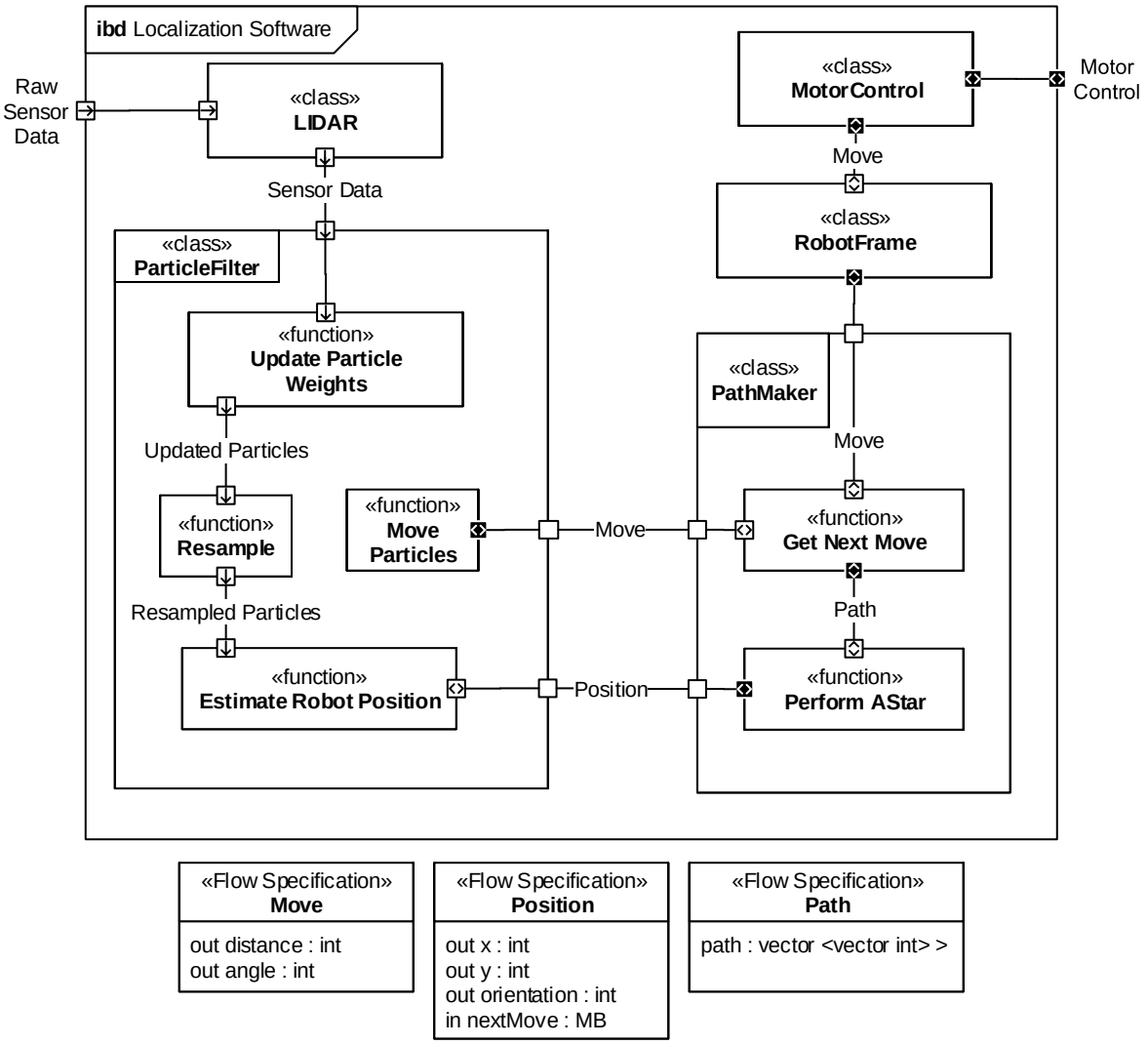


Figure 22: Software Internal Block Diagram

10.3 Implementation

This section covers the implementation specific details of the software. The software was implemented according to the design, as the programs main function, presented in Listing 3, makes apparent.

Focus in the following subsections has been put on the PathMaker, ParticleFilter and RobotFrame class, as they have required the most work during implementation.

Listing 3: Main Function of Progam

```
int main(void) {
  Position goal;
  goal.x = GOAL_X;
  goal.y = GOAL_Y;
  goal.orientation = GOAL_ORIENTATION;

  PathMaker* pathMaker = new PathMaker();
  pathMaker->setGoal(goal);
  pathMaker->loadMap();

  RobotFrame* robotFrame = new RobotFrame();
  robotFrame->pos.x = 4;
  robotFrame->pos.y = 4;

  ParticleFilter* particleFilter = new ParticleFilter(pathMaker->map);
  LIDAR* lidar = new LIDAR();
  MoveBlock nextMove;

  while (1) {
    if (pathMaker->checkGoal(robotFrame->pos) != true) {

      lidar->senseData();

      particleFilter->updateWeigths(lidar->dataArray, pathMaker->map);
      particleFilter->resample(pathMaker->map);
      particleFilter->getNewPosition(robotFrame->pos);

      pathMaker->performAStar(robotFrame->pos);
      pathMaker->getNextMove(nextMove);

      robotFrame->move(nextMove);
      particleFilter->moveParticles(nextMove);
    }
    else
    {
      std::cout << "Found Goal" << std::endl;
      return 0;
    }
  }
  return 0;
}
```

10.3.1 ParticleFilter

The ParticleFilter class is responsible for turning sensor data into an estimate of the robots current position. In this course, the two main techniques for doing so are particle filters and Kalman filters. Particle filters were chosen, which the name of this class implies, since particle filters are relatively easy to implement, and do not require landmarks; it can use raw measurements.

The ParticleFilter class uses a Particle object to represent a particle. A particle has attributes describing its position and orientation, much like the robot does. These particles are generated randomly in the constructor of the ParticleFilter class. The ParticleFilter class has three main functions. UpdateWeights, Resample and GetNewPosition. The UpdateWeights function updates the weights of the particles. The weights are calculated by using the weightCalculation function of the particle objects that can be seen in Listing 4.

Listing 4: weightCalculation function of particle class

```
float Particle::weightCalculation(int* dataArray){

  float prob = 0.0;
  int distDif, angleIndex;
  int measurement[NUM_ANGLES]; //[8];

  for (int i = 0; i < NUM_ANGLES ; i++ ) {
    measurement[i] = dataArray[2*i];
  }

  int n = 0;
  for(int i = 0; i < NUM_ANGLES; i++){
      if (measurement[i] != -1){

        angleIndex = modu(((pos.orientation/2)+i),180);

          distDif = RangeArray[pos.y][pos.x][angleIndex] - measurement[i];
          prob += (normalizer - 0.5*(pow(distDif,2))/(noiseSqr));
          n++;
      }
  }

  if(n == 0)
      return 0;
  else
      return exp(prob/(float)n);
}
```

The measured data is compared with the RangeArray. The RangeArray is a pre-computed array with 32x32x180 floats, that contains the expected measurement data at all grid cells and angels in the map.

As described in Section 3 the weight can be calculated using a Gaussian distribution. Since each particle compares 180 measurements with their expected ranges, the weights must be calculated as the product of likelihoods, that are found using a Gaussian. There is however a disadvantage using the likelihood directly due to that fact that it in numeric unstable because small numbers might be multiplied together. It is instead preferable to use the log likelihood while it turns the multiplications into additions, and then in the end find the likelihood by taking the exponential to the log likelihood value. In this project the log likelihood is being divided by the number of measurement used in the iteration before it is turned into likelihood. Thereby the weight is found as the likelihood given the exponential of the average log likelihood, Equation 36

$$w_i = \mathcal{L}(\mathbf{x}_i|\mathbf{z}) = \exp\left(\frac{1}{N}\sum_k \ln\left(\frac{1}{\sqrt{2\pi^2\cdot\sigma_{\mathbf{z}}^2}}\exp\left(-\frac{\frac{1}{2}\cdot(z_k - x_{i,k})^2}{\sigma_{\mathbf{z}}^2}\right)\right)\right) \tag{36}$$

Once the new weights have been calculated, the Resampling function is invoked. The resampling function is the heart of the particle filter. In this function, a new set of particles is constructed, by randomly choosing particles from the current set. When choosing which particles to include in the new set, the probability of picking a certain particle is proportional to its weight. This is accomplished by using the Low Variance Resampling technique presented in the course material.

Only 95% of the particles in the new set are made from resampling. The last 5% are new particles, which are randomly placed on the map. The randomly placed particles are needed if the robot misplaces itself in relations to its particle. This can happen if the movement of the robot does not fit the motion model, or if the robot is physically moved by someone else or "Kidnapped"; hence the "kidnapped robot" situation described in Section 3. The Resampling function can be seen in Listing 5

Listing 5: Resampling function of ParticleFilter

```
void ParticleFilter::resample(intMatrix map) {
  vector<Particle> tempSet;
  int index = rand() % NUMBEROFPARTICLES;
  float beta = 0;
  float r;

  //Resampling of old particles
  for (int i = 0; i < NUMBEROFPARTICLES-NUMBEROFRANDOMPARTICLES; i++) {
    r = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
    beta += r * 2 * maxWeight;
    while (beta > particles[index].w) {
      beta -= particles[index].w;
      index = (index + 1) % NUMBEROFPARTICLES;
    }
    tempSet.push_back(particles[index]);
  }

  //Newly generated random particels
  Particle tempPar;
  for(int i = 0; i < NUMBEROFRANDOMPARTICLES; i++){

    tempPar.cont_x = (static_cast <float> (rand()) / static_cast <float> (RAND_MAX/↩
        MAPWIDTH))*GRID_SIZE;
    tempPar.cont_y = (static_cast <float> (rand()) / static_cast <float> (RAND_MAX/↩
        MAPHEIGHT))*GRID_SIZE;

    tempPar.pos.x = (int)(tempPar.cont_x/GRID_SIZE);
    tempPar.pos.y = (int)(tempPar.cont_y/GRID_SIZE);

    if(map[tempPar.pos.y][tempPar.pos.x] == 0)
    {
      tempPar.cont_orientation = (rand()%4)*90; // Initialize particles in grid ↩
          orientations
      tempPar.w = 0;
      tempSet.push_back(tempPar);
    }
    else
    {
      --i;
    }
  }
  particles = tempSet;

}
```

Lastly the particle filter estimates the most probable current position of the robot. This is done by finding the particle with the largest weight. The course material proposes to do this by averaging the parameters of all particles. However, this does not work well with a multi-model distribution of particles. If the particles are clustered in two equally likely clusters, the resulting position would always be in between the two clusters, and thus wrong. By using the largest weight, the position would only be wrong half the time.

Once the PathMaker class has used the estimate of the current position to find the next movement instructions for the robot, the same instructions are sent to the ParticleFilter. The movement instructions are applied to all the particles, with a normal distributed noise with standard deviation of 10° added to the angle, and a normal distributed noise with standard deviation of 10 mm added to the distance.

10.3.2 PathMaker

The PathMaker class is responsibly for managing the map the robot is navigating in, finding the shortest path to the goal position and getting the next move instruction that moves the robot and the particles along the generated path. In the constructor of the PathMaker, a 32 x 32 grid is build, based on the real environment, shown in Figure 23.

Figure 23: Real-world environment

The grid map in which the robot will be driving, is divided into 10 x 10 cm squares. Open spaces are represented as '0', Obstacles are represented as '1' and the robot is represented as '**'. The grid map can be seen in Table 14.

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
4	0	0	0	0	0	0	0	0	0	0	**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
10	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
23	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
24	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
25	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
26	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
27	0	0	0	0	G	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
28	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
29	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
30	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
31	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 14: Robot enviroment

When the ParticleFilter has estimated the position of the robot, the PathMaker should use that position to calculate a path to the goal.

In this course, Breath First Planning, A-Star and Dynamic Programming were presented as ways of calculating a path. A-Star was chosen because it is faster than breath first planning, and it is better at handling unexpected obstacles than dynamic programming.

A heuristic map was made for the A-Star algorithm. This is added to the movement cost to make the nodes closest to the goal, the first ones to be explored. The heuristic map can be seen in Table 15.

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	31	30	29	28	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
1	30	29	28	27	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
2	29	28	27	26	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
3	28	27	26	25	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
4	27	26	25	24	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
5	26	25	24	23	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
6	25	24	23	22	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
7	24	23	22	21	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
8	23	22	21	20	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
9	22	21	20	19	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
10	21	20	19	18	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
11	20	19	18	17	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
12	19	18	17	16	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
13	18	17	16	15	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
14	17	16	15	14	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
15	16	15	14	13	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
16	15	14	13	12	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
17	14	13	12	11	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
18	13	12	11	10	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
19	12	11	10	9	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
20	11	10	9	8	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
21	10	9	8	7	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
22	9	8	7	6	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
23	8	7	6	5	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
24	7	6	5	4	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
25	6	5	4	3	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
26	5	4	3	2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
27	4	3	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
28	5	4	3	2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
29	6	5	4	3	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
30	7	6	5	4	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
31	8	7	6	5	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Table 15: Heuristic map from implementation. The number is the additional cost the heuristic adds to the cost of movement

In addition to the heuristic map, a penalty map was also created. The penalty map was also added to the movement cost. A penalty was given to the squares adjacent to walls, and therefore the robot should avoid moving next to walls. The penalty map can be seen in Table 16.

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
1	100	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	100	100	100	100	100	100	100
2	100	50	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	50	100	100	100	100	100	100
3	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
4	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
5	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
6	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
7	100	50	10	10	10	10	10	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
8	100	50	50	50	50	50	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
9	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
10	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100	100	100	100	100	100
11	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	50	50	50	50	50	100
12	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	10	10	10	10	10	50	100
13	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100
14	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100
15	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100
16	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100
17	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100
18	100	100	100	100	100	100	50	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	50	100
19	100	100	100	100	100	100	50	10	0	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	50	100
20	100	50	50	50	50	50	50	10	0	10	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	100
21	100	50	10	10	10	10	10	10	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
22	100	50	10	0	0	0	0	0	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
23	100	50	10	0	0	0	0	0	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
24	100	50	10	0	0	0	0	0	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
25	100	50	10	0	0	0	0	0	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
26	100	50	10	0	0	0	0	0	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
27	100	50	10	0	0	0	0	0	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
28	100	50	10	0	G	0	0	0	0	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
29	100	50	10	10	10	10	10	10	10	10	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
30	100	50	50	50	50	50	50	50	50	50	50	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
31	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Table 16: Penalty map from implementation. The number is the additional cost the penalty adds to the cost of movement

The final A-Star path is constructed as explained in the course theory, Section 4, and the resulting path can be seen in Table 17.

idx	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	↓	←	←	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	↓	←	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	↓	←	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	↓	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	G	←	←	←	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 17: Shortest path from implementation

Once the path has been generated, the next move instruction is sent to both the robot and the particles of the particle filter.

10.3.3 RobotFrame

The robot frame is a wrapper class for the movement control class that does the actual movement. It contains the current pose estimate, consisting of grid position and grid orientation, and is responsible for executing movement.

`MoveController` receives a requested move from the `PathMaker` class in a `MoveBlock` object. This object contains a heading, related to map coordinates, and a distance in grid cells. `MoveController` then calculates the required turn and translation to achieve the requested move. This entails figuring out how much to turn and in which direction (clockwise/counter-clockwise) to turn.

To execute the move the `MoveController` does the following:

1. Calculate *TurnTime*; the time a turn to the desired angle will take
2. Set speed of relevant wheel according to turn direction, by writing a PWM value in the range 0 - 1023 to the `pwmL` or `pwmR` registers, and setting the direction of travel by writing to the `DIR` register.
3. Set a timer for *TurnTime*, wait for timeout and set PWM values to zero to stop
4. Calculate *MoveTime*; the time a move to the desired distance will take
5. Set speed of the wheels by writing a PWM value in the range 0 - 1023 to the `pwmL` and `pwmR` registers, and setting the direction of travel by writing to the `DIR` register.
6. Set a timer for *MoveTime*, wait for timeout and set PWM values to zero to stop

In Listing 6 below, an example of a simple forward move i shown.

Listing 6: Example of simple forward movement

```
// Move
// Calc move time
moveTime = distance/GRID_SIZE*MOVETIME100;
// moveTime = int(((float) distance / MOVERATE50) * ONE_SECOND);
// Start move
XMotorctrl_SetPwmr(&motor, RIGHT_MOTOR_ON);
XMotorctrl_SetPwml(&motor, LEFT_MOTOR_ON);  //512);
XMotorctrl_SetDirection(&motor, 1);
// Set timer  // wait

ScuTimerWait(TIMER_DEVICE_ID, moveTime);

// Stop move
XMotorctrl_SetPwmr(&motor, 0);
XMotorctrl_SetPwml(&motor, 0);
```

11 Test

Because of the probabilistic nature of the robot and because we get no data back from the robot as it moves, it is hard to give an estimate of its performance. To be specific; it (sort of) works most off the time, eventually finding a goal. This goal is not always within the target distance of the true goal.

A common issue is that it gets stuck at a corner, or along a wall. This is likely due to the squarish shape the platform, and the less than ideal motion model. Solutions for this is discussed in Section 12.

As for the *Kidnapped Robot Problem*, it is handled without much issue. When the robot is moved during path-finding, the continuous addition of random particles seems to help it relocate itself within a few (2 to 5) sense/move cycles. Again, it is next to impossible to know what the robot is "thinking" as it moves along. Therefore we can only speculate as to the performance of the robot in the KRP from its subsequent actions. Here it seems 'confused' at first, but gradually starts moving in the right direction.

A large source of error in testing the robot is the surface of the test course. The concrete floor seems smooth at first, but is filled with dips and cracks, that the ball caster front wheel gets stuck in, resulting in over/under rotation on turning and heading changes during forward translation.

12 Improvements

This section highlights some of the improvements that could be made to the project. The main reasons these improvements were not implemented were lack of time, technology or working software licenses.

The motor controller is the first point that should be improved. Currently the motor controller works by applying a PWM signal for a certain period of time. This works in some circumstances, but factors like an uneven surface makes it very unreliable. The motors do contain an odometer which in theory would make it possbile to get a more precise movement, by making a PID controller.

With a more precise movement, the possible turn angle can also be expanded. Currently the robot is limited to 90 degree turns, which makes it hard for the robot to navigate back to its path. Quantisizing the movement to 10 degrees instead of 90 degrees could make the robot able to make up for small errors in rotation.

In the current motion model a rotation also involves a translation. This complicates the motion model, and adds a lot of uncertainty to a rotation. This uncertainty could be greatly reduced by changing the motion model to a pure rotation around the center of the robot. This could be done by making the wheels of the robot move in opposite directions when turning. Unfortunately, the license of the tool that was used for generating the hardware for the motor controller has expired, making this change impossible.

Another improvement would be to add a wireless connection between the robot and a PC. The wireless connection would provide two advantages. The first advantage would be monitoring, which would greatly speed up the debugging process. Currently the state of the robot is not easily accessible once the system is running. This makes debugging very hard, as it requires a physical connection to the device through a cable. The other advantage would be the ability to control the parameters of the robot. This could involve changing the heuristic or penalty map, changing the position of the goal, or changing the number of particles. Currently, implementing any change is a cumbersome process, as it involves re-flashing the device. This takes 5-10 minutes, making the debugging cycles very long.

A wireless connection could be made by buying an additional PMOD for the ZYBO, which adds support for a wireless connection.

A final improvement would be to get the robot to move more continuously. This would require the update rate of the particle filter to be improved, since it is the current bottleneck of the system. Especially the resampling can take a long time, because the current method of resampling takes a non deterministic time, with a worst case of $O(n^2)$ time.

If an approach like the tournament approach was used, the resampling would take $O(n)$ time.

Part III

Conclusion

In this course the theory of solving the localization problem for a robot in a known environment was covered.

Six main theory sections were used as the basis for a project that used the theory to solve the localization robot in a custom robot. A system was build using a chassis, a XV11 LIDAR Sensor, two motors and a ZYBO development board with accompanying modification boards. High Level Synthesis was used to produce custom hardware components for motor control in reprogrammable logic. The Xilinx SDK was used to program software, which was designed in an object-oriented manner.

In the end, a working system was created, though improvements, to among others the motion model, have been suggested, that would greatly improve the efficiency and precision of the robot.

References

[1] DFRobot. Baron-4WD Mobile Platform, 2015. URL http://www.dfrobot.com/index.php?route=product/product&keyword=baro&product_id=261#.VYP3qjCqpBc.

[2] Digilent. PmodDHB1™ Refernce Manual, 2013. URL https://digilentinc.com/Data/Products/PMOD-DHB1/PmodDHB1_rm_RevA.pdf.

[3] Digilent. ZYBO Reference Manual, 2014. URL http://www.digilentinc.com/Data/Products/ZYBO/ZYBO_RM_B_V6.pdf.

[4] OpenSource. xv11hacking. URL https://xv11hacking.wikispaces.com/.

[5] Dan Simon. *Optimal State Estimation: Kalman, H_infinite, and Nonlinear Approaches*. 2006. ISBN 0471708585. doi: 10.1002/0470045345.

[6] Sebastian Thrun. Probabilistic robotics, 2002. ISSN 00010782.

[7] Sebastian Thrun. Artificial Intelligence for Robotics, 2015. URL https://www.udacity.com/course/artificial-intelligence-for-robotics--cs373.

[8] Wikipedia. Rotation (mathematics), 2015. URL https://en.wikipedia.org/wiki/Rotation_(mathematics).