Autonomous Mobile Robots Final Project

Group 8

07. June 2022

Navn	Studienummer
Rasmus Carlsen	201805557
Peer Weidemann	201807779
Jonas Lund	201906201
Lucas Bjarke Jensen	201907355

Introduction

The purpose of this report is to document the development of a program capable of autonomously providing instructions to a robot, navigating it throughout a building and solving various tasks. A description of the robot system is first presented, after which a description of the principles of the robot oriented software framework called ROS is given. The algorithms that the program is composed of is then explained, and the results provided by this program are presented and discussed.

Robot System Description

This section gives an overview of the general system setup. There will be a general description of the robots features, a description of the communication with the robot and the software used.

Setup

The robot used is a Turtlebot 2, which is a robot development kit using open-source software, following the REP 119 (ROS Enhancement Proposals) specifications. It consists of a Kobuki mobile base, which essentially is similar to a robot vacuum cleaner. On top of the base there is mounted a structure with module plates containing mounting holes and a mount for a Microsoft Kinect-like camera. An Asus Xtion Pro Live module is mounted on the Kinect mount, which contains an RGB camera and a depth sensor. The depth sensor consists of an infrared projector and camera. For data processing and controlling the robot, a Raspberry Pi 3B(+) is used. It interfaces between the robot and the machine controlling the robot by creating a Wi-Fi network.

The Turtlebot is controlled using ROS (Robot Operating System), which will be introduced in detail in the next section. There are two ways to use the robot. One of the ways

of using the robot is by simulating it in a Simulator called Gazebo. Here a model of the robot is used which includes simulating some of the optical sensors such as cameras. It can drive around in a world you can create yourself with an editor built into the simulator or pre made worlds. The other way of using the robot is in the real world with the real hardware. Controlling the robot is done via the software MATLAB. Here the necessary code is written and can be executed using a ROS toolbox to be able to interface with the robot. This works for the simulated robot as for the real robot.

ROS

The Robot Operating System (ROS) is, contrary to what its name implies, not just an operating system. Rather, it is a open-source software platform that facilitate the development of robot applications. ROS achieves its functionality on three distinct levels. the Filesystem Level, the Computation Graph Level and the Community Level.

The Filesystem level deal with all the resources found on disk, such as runtime processes, ROS libraries, data and configuration files. These are in turn encapsulated by the concept of a Package.

The Computation Graph Level is a peer-to-peer network that links the runtime process nodes together, which enable communication across the network. The Computation Graph Level concepts that are the most relevant for this project include nodes, messages, the publish-subscribe pattern, topics and services.

The Community Level is the furthers from the implementation level and include concepts like the ROS Wiki, ROS distributions, code repositories and various forums.

The explanation of the Computation Graph Level will now be expanded upon.

Relevant Components of the Computation Graph Level

- **Nodes:** A ROS node is process that performs some computation at run-time. These are usually simply in nature and only perform more complicated tasks by communicating with other nodes.
- **Messages:** A ROS message is the type of input and output that nodes exchange to communicate with each other. A message can have any given number of fields, containing any number of types and the usual standard types as well as arrays and custom types are supported.
- **The Publish-Subscribe Pattern:** The passing of messages between nodes is facilitated by the publish-subcribe pattern. The result of using this pattern is that the communication between nodes is abstracted, leading to a publisher node having no information about where the data it outputs is going and subscriber node to have no information about where the data it receives is coming from. This results in the nodes being decoupled from each other, which is necessary given the modularity of the Computation Graph network. The relation between publishers and subscribers is a many-to-many relationship

- **Topics:** A topic is an identifier used by the publish-subcribe pattern. The topic contains information about the content of the message, which the pattern uses to organize where to send messages. A publisher will publish data to a topic, and only nodes that subscribe to that topic will get that message.
- **Services:** Services exists as an alternative to the publish-subcribe functionality, and is useful for request/reply interactions. A node that wants to use a service sends a special request message type containing a string name, after which it waits for a special reply message from a node that offers that service under the same requested string name.

Overview

In this section, we will give an overview of how the robot will execute its task in broad terms. Later sections will then describe each algorithm and its usage in greater detail.

The robot's task is to move from point A to point B and from point B to point C using a global path planning algorithm and a local path planning algorithm to dodge obstacles. On arrival at each point, the robot must find a green circle on a wall and position itself in front of it at a distance of 40 cm. Furthermore, it has to play a sound when the circle has been detected. The Shannon building and the placed goal points are visualised below in figure 1.

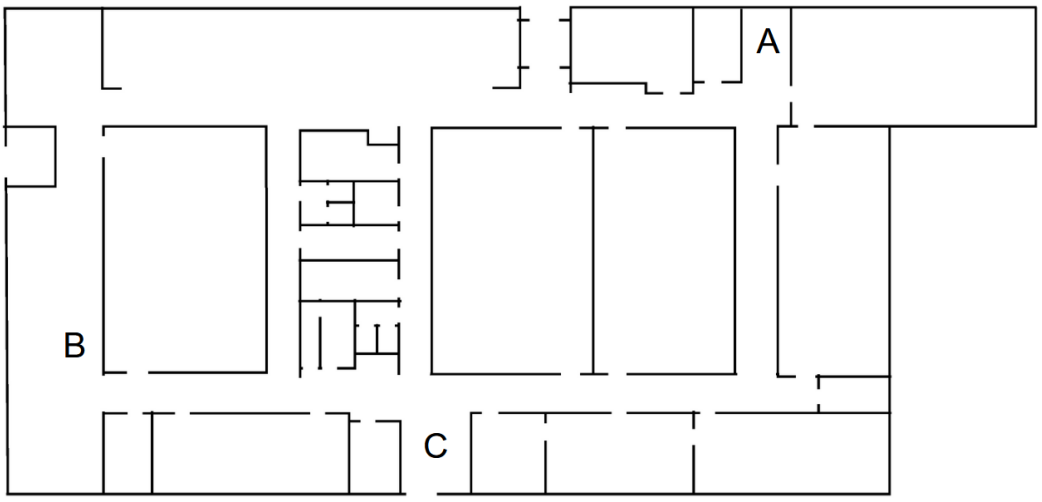

Figure 1: Overview of the Shannon building, with the goal points visualized

Overall, the execution loop of the robot can be expressed as the following pseudo-code:

```
load maps
initialize probabilistic road map
initialize pure pursuit controller
initialize vector field histogram controller
initialize monte carlo localization

// so the monte carlo localization can find its current position
spin 360 degrees

for each point do

    find path to point
    set pure pursuit waypoints to path

    // Get to the point
    while distance to point > 1.5 do

        receive odometry
        receive 2d scan

        update localization
        get desired direction from pure pursuit controller using estimated pose
        get target direction from vector field histogram controller

        calculate angular velocity
        send velocity data

    end

    // Find the circle
    receive picture
    find circle
    while circle not in center do
        if 360 degrees turned then
            print "No circle found!"
            return
        end

        turn right
        receive picture
        find circle
    end

    // Move up to the circle
    while distance to wall > 0.4 do
        move forward
    end

    // Announce
    make a noise

end
```

The overall idea is to find a suitable path inside the building by using the Probabilistic Road Map, drive to the point with Pure Pursuit using the pose found by the Monte Carlo localization and let a Vector Field Histogram controller do obstacle avoidance by overruling

the Pure Pursuit controllers direction if need be.

Once the robot is on/close to the point, it will turn and use the camera to find a green circle. The green circle is then centered on the camera. Lastly the robot moves forward to be 40 cm in front of the circle and outputs a sound.

The robot will then continue to the next point, until all points have been visited.

Probabilistic Road Map

The Probabilistic Road Map is a map that constructs a graph on a pre-existing Binary Occupancy Map. The Probabilistic Road Map enables the robot to find a path on said graph. The graph can be seen on fig. 6.

The graph is constructed by placing a number of random points on the Binary Occupancy Map supplied, and connecting the points, if they do not pass an occupied square on the map.

In order to accommodate the size of the robot, the Binary Occupancy Map is inflated before it is used in conjunction of the Probabilistic Road Map. The final code can be seen below:

```
%% Load occupancy map
img = imread("shannon.sdf(-1590,-728).png");
grayImg = rgb2gray(img);
bwImg = grayImg < 0.5;
xoffset = -2345;
yoffset = -1413;
resolution = 50; % 50 pixels per meter

map = binaryOccupancyMap(bwImg, resolution);
map.GridOriginInLocal = [xoffset, yoffset] / resolution;

% Inflate map to compensate for robot size (radius of 0.3m)
inflatedMap = copy(map);
inflate(inflatedMap, 0.7);

% Make Probabilistic Road Map with 3000 points
prmMap = mobileRobotPRM(inflatedMap, 3000);
prmMap.ConnectionDistance = 3;
```

Green Circle Detection

Detect Circle Loop

The circle detection loop consists of turning the robot while reading an RGB image from the ROS topic `"/camera/rgb/image_raw"`, calling the custom `findCircle()` function, and checking if a circle is found and if it is in the center of the image. If it is, the loop is broken out of.

The `findCircle()` function takes an RGB image and returns a boolean value `foundCircle`. It starts by blurring the image using a Gaussian filter. This is to remove noise and thus aid in the classification of the circle. In figure 2 is an image of a green circle from the robot's camera with a Gaussian filter applied.

Figure 2: Image of green circle with Gaussian filter applied.

First, we perform color classification. The filtered RGB image is converted to an HSV image and thresholds for hue, saturation and value are chosen. These thresholds are chosen based on what we want to consider green, but also what the lighting is like in the environment. These thresholds will represent a 'chunk' of the HSV colorspace (shown in figure 3) that classifies as our desired green. All indices receive a boolean value representing if a certain pixel is inside that threshold or not. These values are then combined into a black-white image shown in figure 4.

```
% Check the threshholds for hue, saturation and value
hueInds = hsvImg(:,:,1) ≥ (70 / 360) & hsvImg(:,:,1) ≤ (160 / 360);
satInds = hsvImg(:,:,2) ≥ (20 / 100);
valInds = hsvImg(:,:,3) ≥ (5 / 100);

% Combine and flip to get the black-white image
BW = hueInds & satInds & valInds;
```

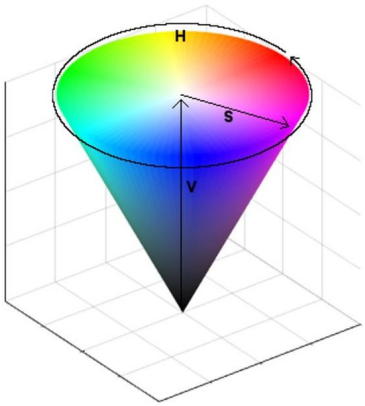

Figure 3: The HSV colorspace visualized.

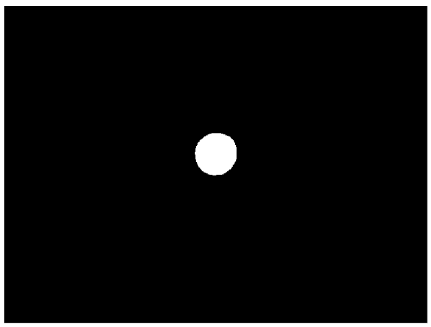

Figure 4: Color classification of the green circle image.

Now we need to classify whether anything on the black-and-white image is a circle. This happens through MATLAB's functions `bwconncomp()` and `regionprops().Circularity`. See following code snippet:

```
% Find the connected components
CC = bwconncomp(BW);

% Filter out all components with not enough area
CCSizes = arrayfun(@(c)(length(c{1})), CC.PixelIdxList);
CCSizeValids = CCSizes > 25*25;

% Filter out all components that are not circular enough
CCCircs = [regionprops(CC, "Circularity").Circularity];
CCCircValids = abs(CCCircs - 1) < 0.1; % 0.1 above or below 1 is valid.

% Combine
CCValids = CCSizeValids & CCCircValids;
```

Connected components are compiled using the `bwconncomp()` function on the black-and-white image. Since only components of considerate sizes are to be considered (as small components are not recommended for `regionprops().Circularity`), we compile the sizes of all connected components using an array function, and then sort out all components smaller than $25 * 25$ pixels.

The `regionprops().Circularity` function is applied to all connected components, returning a value describing each components circularity/roundness, computed as `(4*Area*pi)/(Perimeter`^{2}`)`. The closer the absolute value of the circularity - 1 is to 0, the more circular the component is. When all components of valid sizes and roundness are found, they are combined and we have our valid circles.

If no valid circles exist, the function returns that no circle was found setting `foundCircle` to false. If valid circles are found, the biggest one is chosen and the center of it is found using `regionprops().Centroid`. The X and Y coordinates of the circle are returned, and `foundCircle` is set to true.

The Find Wall Component

The `Find Wall` component is responsible for placing the robot 40 cm in front of the circle and playing a sound. This happens after the circle has been detected, and the robot is, at this point, already facing the circle. In the pseudo-code presented in the overview section, the The `Find Wall` component is shown to consist of the following instructions:

```
 // Move up to the circle
while distance to wall > 0.4 do
    move forward
end

// Announce
make a noise
```

As can be seen from the pseudo-code, the `Find Wall` component therefore only deals with the distance to the wall, where the circle is placed. To get the robot in the right distance, the `getWalls` algorithm, which is explained below, is called. The distance is then used as part of the while loop condition, `distance < 0.7`. Meaning that, if the robot is already at a distance of 40 cm to the wall, taking the length of the robot into account, the while loop is not entered. The while loop itself, simply receives a scan message and uses the `getWalls` algorithm to read the distance to the wall. When the robot is at a correct distance, the

condition fails and the robot is instructed to stop and play a sound, using a ROS soundPub message.

Figure 5: Robot 40 cm in front of a green circle.

The getWalls Algorithm

As mentioned in the above section on the Find Walls component, the `getWalls` algorithm is used when finding the distance to the wall on which the green circle is placed.

The algorithm starts by creating the binary image matrix. This is done by rasterizing the known points. In order to rasterize the points, a grid must be defined. The grid has a fixed size of N times N. The resolution of the grid is determined by finding the smallest square that contains all points. The offset of the square on the X- and Y-axis are saved as `xoffset` and `yoffset` respectively. This square is divided into N by N sub-squares. If a square contains a points, it is a 1, otherwise 0. The points have thus been transformed from a point cloud to, essentially, a black and white image, the binary image matrix.

The binary image matrix is then fed to the `hough` function which performs the Hough transform. The `houghpeaks` then takes the Hough-space and the peaks matrix `P` is used to create the `houghlines` array struct.

The found line segments are then transformed back, such that they are relative to the robot. This means, that the resolution and offset applied during the rasterization is reverted for the 2 endpoints of all line segments.

Before processing the lines any further, they are converted from line segments defined by 2 points to a parametric line:

$$\overrightarrow{\mathrm{OP}_0} + \vec{\mathrm{r}} \cdot \mathrm{t}$$

Where:

$$\vec{\mathrm{r}} = \overrightarrow{\mathrm{OP}_1} - \overrightarrow{\mathrm{OP}_0}$$

With P_0 and P_1 being the 2 end points of the line segment and t being the position on the line segment, where $\mathrm{t} = 0$ is P_0 and $\mathrm{t} = 1$ is P_1. The variable t is then used to find points on the line between the start and end point.

The parametric representation of the line segment can now be used to find the distance to the segment itself, or the infinite line it can represent. Both distances are useful, since the scan of the robot is limited between $\pm 45°$ in the direction the robot is facing.

As shown on the image above, the robot does not see the wall that is 90° to it. This would mean that the distance to the wall is significantly closer than the distance to the line segment found by means of the Hough-transform.

With the line segments relative to the robot, the distance from the robot to the line is calculated by means of the "closestT".

Motion Control

Pure Pursuit

The Pure Pursuit controller uses the Pure Pursuit algorithm to follow the path created using the Probabilistic Road Map. The controller is fed the nodes of the Probabilistic Road Map path along with parameters detailing desired linear velocity, maximum angular velocity and lookahead distance.

The algorithm works by following a point that travels along the generated path. The point will be ahead by the lookahead distance. This section illustrates how the Pure Pursuit controller is used. To start off, the controller is initialized and the maximum angular and linear velocities are configured as well as the lookahead distance:

```
16 ppcont = controllerPurePursuit;
17 maxLVel = 0.5;
18 maxAVel = 1.3;
19 ppcont.DesiredLinearVelocity = maxLVel;
20 ppcont.MaxAngularVelocity = maxAVel;
21 ppcont.LookaheadDistance = 0.5;
```

The waypoints given to to the Pure Pursuit controller is generated from the Probabilistic Road Map and is updated once per goal as can be seen below.

```
121 for j = 1:size(goals, 1)
122     goal = goals(j,:);
123     % Use find path function on the PRM map
124     path = findpath(prmMap,pose(1:2),goal);
125     ...
126     ppcont.Waypoints = path;
127     ...
```

The Pure Pursuit controller is also used, when deciding the exact linear and angular velocities that would move the robot closer to the current goal by following the lookahead points, as illustrated below.

```
155 [v, ¬, lookaheadPoint] = ppcont(pose);
156
157         lpr = lookaheadPoint - pose(1:2);
158         targetDir = angdiff(pose(3), atan2(lpr(2), lpr(1)));
159         steeringDir = vfhcont(scan, targetDir)
160         if isnan(steeringDir)
161             v = 0;
162             omega = 0.5;
163         else
164             omega = 2*sin(steeringDir) / ppcont.LookaheadDistance;
165             if abs(omega) > ppcont.MaxAngularVelocity
166                 omega = sign(omega) * ppcont.MaxAngularVelocity;
167             end
168         end
```

Figure 6 shows a planned Pure Pursuit path from the robots initial position to a goal further down a hallway.

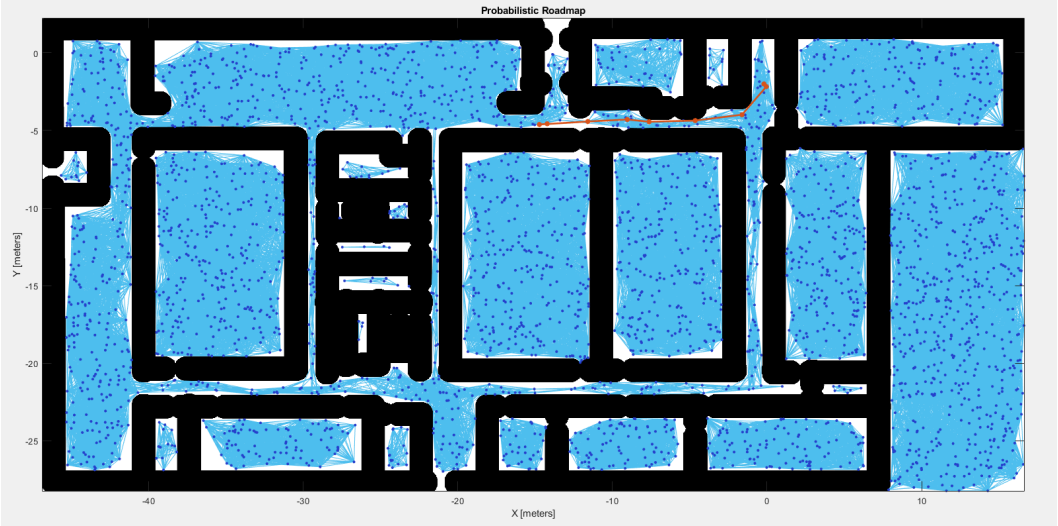

Figure 6: PRM with Pure Pursuit path from initial position to first goal.

Vector Field Histogram

The Vector Field Histogram is an algorithm that allows fast obstacle avoidance. The idea is to generate a polar obstacle density histogram, which allows the algorithm to determine if it should turn in order to avoid an obstacle. Polar in this context means, that the data is represented in a polar coordinate system. VFH takes into account the proportions of a robot, as in its radius, and other parameters that describe the robots dynamics. The polar obstacle density histogram has been constructed from a histogram grid, which contains datapoints from scans of the environment. The MATLAB implementation has removed the 2 dimensional grid, and instead uses a lidar scan in order to generate the polar obstacle density histogram. As a consequence, the VFH implementation in MATLAB cannot take obstacles into account, that have been in a previous lidar scan, but are not in the current lidar scan.

The following explanation is therefore done from MATLAB's implementation and not the original paper.

On fig. 7, an example of the polar obstacle density histogram is show. The histogram on the left is made from the range scan readings, whilst the right is constructed from the left with a set of thresholds, which determine if a section is occupied.

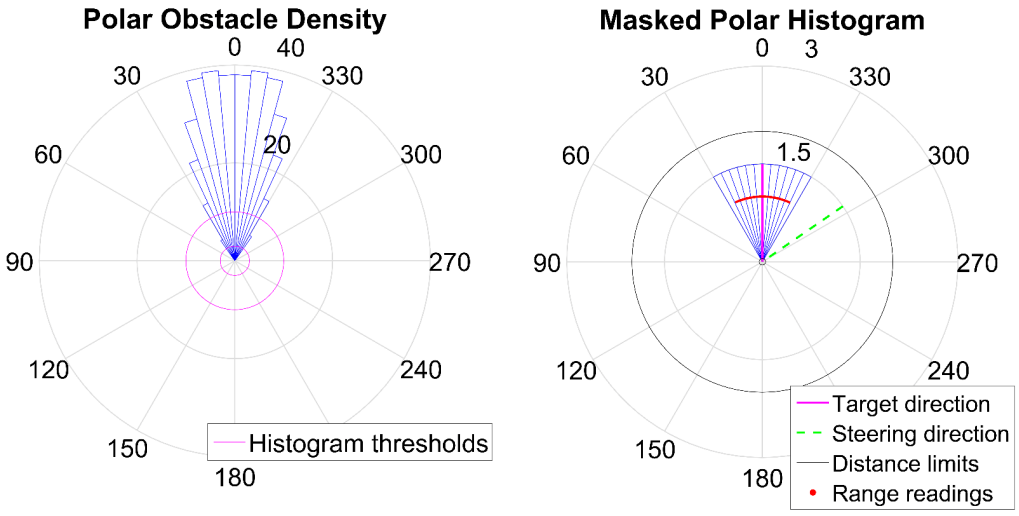

Figure 7: Visualization of the polar obstacle density histogram.

From the masked polar obstacle density histogram, the desired angle to continue along and the turning radius of the robot and the desired safety distance, the VFH controller can find a direction, which is suitable for the robot to continue along. If no direction can be found, the VFH controller will return `NaN`, in which case the robot will turn on the spot until a new suitable direction is found.

The configuration of the VFH controller can be seen in the code below.

```
%% VFH controller
vfhcont = controllerVFH;
vfhcont.UseLidarScan = true;
vfhcont.RobotRadius = 0.17;
vfhcont.SafetyDistance = 0.5;
vfhcont.MinTurningRadius = 0.1;
sensorOffset = 0.09;
vfhcont.DistanceLimits = [(0.17 + sensorOffset) (0.7 + sensorOffset)];
```

Monte Carlo Localization

Localization of the robot is to the success of the robot. Relying on the odometry data as only method for localization leads to a large difference between the robots estimated position and its actual position due to the accumulated error. It is therefore important to use an algorithm, that can correct this error by taking advantage of other sensors of the robot.

The Monte Carlo Localization, or Particle Filter, is a localization method, where the estimates are made about the pose of the robot, which are the particles, and are checked in conjunction of a map, how likely the particle is to be the correct one. The first step is to create estimated poses for the robot. The probability of a pose to be correct is depending on how well the scanned points from the lidar scan match to the map from the poses perspective. The higher the overlap, the more likely the pose is to be the correct pose.

Unlikely poses are removed and new particles are added based on the measurement of the odometry. The new points have a spread depending on the noise of the odometry data. These values can be adjusted in MATLAB with the parameter vector `MeasurementNoise`. Finding the correct noise values is essential in order for the localization to work as best as possible. If the noise is set too low, the algorithm is unable to correct the error introduced by the odometry, if the noise is set too high, there is a risk of the localization to put estimates through walls, leading to the robot potentially guessing a completely wrong position, even if the current guess is correct.

The Monte Carlo Localization can start in two different modes. A local localization mode, where the initial pose is given and an initial covariance is given or a global localization mode, which evenly distributes the first guesses across the map. Using the global localization method in conjunction with the Pure Pursuit controller can lead to some unintended behaviour, where the estimated pose jumps around on the map, leading the Pure Pursuit controller constantly turning the robot. The covariance of the estimated poses could be used to determine if the robot should continue on with the Pure Pursuit or use a local path planning algorithm until the pose is more well known. We chose to use the local method, since its applicable to our task and easier to use. The poses (particles) distribution can be seen on fig. 8 and fig. 9.

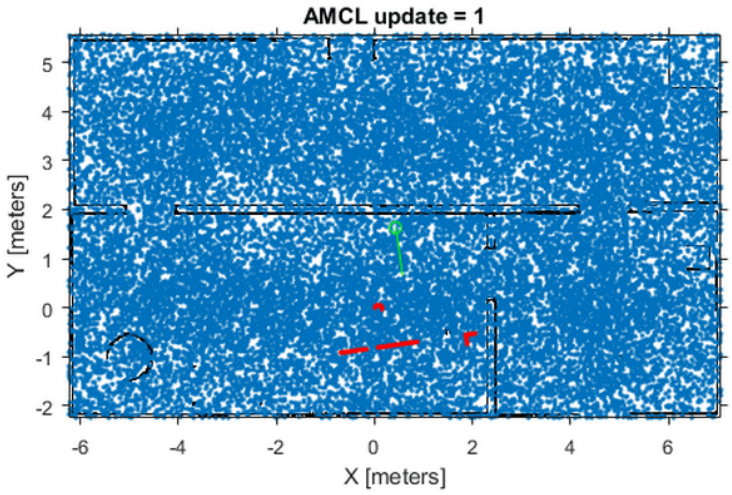

Figure 8: Distribution of initial pose estimates when using global localization.

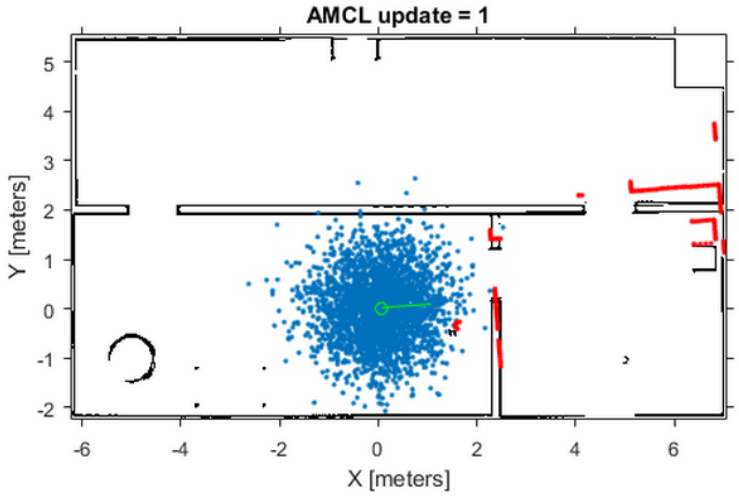

Figure 9: Distribution of initial pose estimates when using local localization.

The full configuration of the Monte Carlo Localization can be seen in the code below.

```
%% Setup odometry and range scan models
odometryModel = odometryMotionModel;
odometryModel.Noise = [0.2 0.2 0.2 0.2];

rangeFinderModel = likelihoodFieldSensorModel;
rangeFinderModel.SensorLimits = [0.45 8];
rangeFinderModel.Map = map;

%% Setup monte carlo
amcl = monteCarloLocalization;
amcl.UseLidarScan = true;
amcl.MotionModel = odometryModel;
amcl.SensorModel = rangeFinderModel;
amcl.UpdateThresholds = [0.01,0.01,0.01];
amcl.ResamplingInterval = 1;

amcl.GlobalLocalization = false;
amcl.ParticleLimits = [200 200];
amcl.InitialPose = pose;
amcl.InitialCovariance = 0.2;
```

Tests

During demo-day, the robot did not perform quite as expected. However, we went back and did some more testing while troubleshooting the algorithms. The most successful test is shown in the video found in the appendices. A flaw of the script is that the Monte Carlo algorithm would not always update the robot's localization correctly, and it would be off compared to the binary occupancy map. This is why in test we had the points B and C be closer to A to at least test the overall process of the robot.

During tests, the robot would also tend to stop, which can be accredited to the VFH controller lowering the refresh rate from time to time. Another frequent issue is that the VFH controller does not take off-sensor obstacles into account. This lead to the robot following walls and then turning into the wall once it was no longer visible on the range scan. Further complication of the issue is the fact that the sensor has a minimum range that is smaller than the distance from the sensor to the front of the robot. Once the robot turned into a wall, it was not possible to adequately handle situation from the range scan alone.

When coming near a goal, the robot would sometimes get stuck in its Pure Pursuit loop, which is caused by the fact that the difference between the estimated and actual pose is fairly large, due to the long hallways in Shannon not giving many key points for the Monte Carlo algorithm to use as reference points. The difference in the actual and estimated pose would lead to the robot to never get into the goal radius.

Discussion

In order to improve the robot and fix shortcomings, the first step would be to tweak the VFH controller and the turn rate. If the turning radius is larger than the distance to the last range scan visible of a wall (taking the robots radius into account), the robot could only turn around the wall and not into the wall. Furthermore, the robot could (and should) make use of its bumper sensor to detect, that it is driving into the wall and turn 180 degrees before continuing. Frankly, a sensor without the minimum distance would be desirable, but this is what we have to work with.

During the circle detection, the robot turns really slowly in order to take the images in time. It might have been possible to make the robot turn faster when a large circle component has been found already and thus save some time. The script also only takes into account the robot being further than 40 cm away from the goal, meaning it would not be able to adjust its distance backing up if it were to be too close the circle.

To further improve the performance of the robot, a faster refresh rate would be desirable. Unfortunately, there is only so much you can do, when using the algorithms included in MATLAB's toolboxes, but there is likely a reasonable amount of optimizations that can be made to our code.

The issues with the robot being stuck in the pure pursuit loop, never reaching its goal, could be fixed by defining a bounding box as goal instead of a singular point. The path planning could be done to the middle of the bounding box, but the robot would be at the desired point, once its position is within the defined bounding box.

In order to fix the robot's localization being off by a lot once reaching the end of a long hallway, it could be possible to restart the Monte Carlo Localization with the last estimated pose as initial pose. The robot could then potentially fix the large difference in actual and estimated position.

Conclusion

The robot demonstrated that it would be able to autonomously execute the task. This included the following cornerstones:

- Planning of a path to the tasks
- Following a path whilst avoiding obstacles
- Localize itself reliably
- Detect the task with sufficient reliability
- Position itself correctly in front of the task

Even though the robot is far from perfect, it can be concluded that the robot has the right basics, but needs a lot of tweaking. The algorithms by themselves work well, but the combination of them all is hard to get right.

The robots performance would be enhanced by quite a bit, when fixing the list of improvements made in the discussion section. Most notably, the issues regarding localization and obstacle avoidance.