Minesweeper

Joakim Karlsson (joka2227@gmail.com)
Fredrik Olsson (frols88@gmail.com)
Martin Stålberg (mast4461@gmail.com)
Mikaela Åhlén (miahlen@gmail.com)

January 16, 2014

1 Abstract

This is a report from a project that was made in the course "Project in Systems and Control" at Uppsala University. The project involved the design of a robot, using the Lego NXT Platform. The idea was to have a robot navigate in an unknown environment with "mines" in the form of pieces of paper. These mines are detected with light sensors and placed in a map that the robot uses for navigation. To achieve this, algorithms for localization and pathfinding were implemented. Localization was acheived by using a camera to scan the environment and find the bearings to beacons set at known locations. These bearings were used to estimate the robot's position and orientation. To create a better estimate, the estimation from triangulation was fused with an estimate from odometry in a particle filter. Pathfinding was implemented as a modified A* search algorithm which returns the shortest mine-free path from start to goal. The robot proved to perform well with both a converging localization and a pathfinding that avoids the mines while keeping the path short.

Contents

2 Introduction

2.1 Aims and objectives

The goal is for a robot to traverse an unknown environment in a deterministic fashion based on mapping and pathfinding. To enable mapping and pathfinding the robot must know its position and to that end a local positioning system (LPS) is to be used. The LPS is to consist of visual beacons of known color and size placed at fixed positions in the environment. The beacons are always in line of sight of the robot which will locate them with a camera and estimate its position from the bearings to the beacons. The environment is to consist of a bright flat surface with "mines" marked with pieces black paper. The robot is to have an array of light sensors at its front arranged in a line perpendicular to the robot's forward direction and facing downwards. This will be used for detecting the mines. Detected mines will be added to the map at coordinates derived from the robot's estimated position and orientation. The robot will plan its path according to the information in the map so as to avoid all detected obstacles, and update the planned path whenever new map information is available.

3 Platform and Tools

3.1 NXT Intelligent Brick

The main component of the robot is the NXT Intelligent Brick [1], a small computer. It has four sensor ports through which it can receive analog sensor data or communicate with digital sensors via either of the standards I2C and RS-485. It has three motor ports through which it can control and communicate with motors. It has a 100x64 pixel monochrome LCD display and four buttons that can be used to navigate a user interface using hierarchical menus. The brick has a 32-bit ARM7 processor, 64 kb of RAM, 256 kb of FLASH memory. It also supports both Bluetooth and USB connection to a PC.

3.2 NXTCam v3

Figure 1. The NXTCam v3

The NXTCam [2], see *Figure 1*, is a camera with an on-board processor that handles image processing in real-time. The standard firmware operates at 30 frames per second and supports two tracking modes; line tracking and object tracking. In object tracking mode the camera tracks objects matching a predefined colormap and calculates their bounding boxes in the image. Up to eight different colormaps can be used and the camera can track up to eight objects for each colormap. The tracked image resolution is 88 x 144 pixels. The camera can connect to a PC via USB where the camera can be calibrated and tested in software NXTCamView [3].

3.3 NXT LineLeader

Figure 2. The NXTLineLeader

The NXTLineLeader [4], see *Figure 2*, contains an array of eight light sensors that measure the intensity of incoming light. In between the light sensors are red LEDs for illumination of the sensed surface. In default operation it is intended to detect dark areas on a bright background, e.g. black lines on white paper, or bright areas on a dark background. Before operation the sensor can be calibrated to a dark color and a bright color, supposed to be the dark and bright colors of the operating area. This allows the user to get calibrated sensor readings in the range 0 to 100 from each sensor, where 0 is dark and 100 is bright, or a result byte in which each bit is 0 or 1 depending on whether the calibrated sensor value was below or above, respectively, some unspecified threshold value (seemingly halfway) in between the dark calibration value and the bright calibration value.

3.4 GlideWheel-AS Angle Sensor

Figure 3. The GlideWheel-AS Angle Sensor

The GlideWheel-AS Angle Sensor [5], see *Figure 3* provides angle measurements of a connected axle with a precision of 0.5 degrees. It is made very compact to easily fit any axle; the axle only has to go through the hole in the middle of the sensor. The angle measured by the sensor is the angular offset to the axle's orientation at the time when the sensor was last powered up or reset.

3.5 LeJOS NXJ

LeJOS NXJ [6] is the programming environment chosen for this project. It is basically a smaller version of the Java Virtual Machine and comes with its own libraries for the NXT and most of the common sensors. LeJOS NXJ offers the following:

- Object oriented language (Java)
- Preemptive threads (tasks)
- Arrays, including multi-dimensional
- Recursion
- Synchronization
- Exceptions
- Java types including float, long, and String
- Most of the java.lang, java.util and java.io classes
- A Well-documented Robotics API

The main reasons for choosing LeJOS NXJ is that the authors are familiar with the Java language and that both Java and LeJOS NXJ have detailed and extensive online documentations.

LeJOS NXJ runs its own firmware on the brick which replaces the original NXT firmware. A LeJOS NXJ plugin for the integrated development environment Eclipse [9] quickly compiles and uploads programs to the brick via bluetooth or USB. There is also a feature that allows programs to run on the PC and make full use of the Java library and communicate with the NXT via bluetooth or USB.

3.6 Revision control system

When programming big and complex programs, it can sometimes be critical to have a revision control system, especially when the programming is done by a team of people where they may make changes in the same files. Revision control gives the user an opportunity to go back to an old and working revision if something went wrong after making changes in many different files and if it is difficult to redo these changes. One revision control system is Mercurial [8]. Mercurial is free, and can handle projects of any size. TortoiseHg [9] is a Mercurial revision control client and available for operating systems including Microsoft Windows, Mac OS X and Linux. It offers the advantages of revision control, where you can see which person made which changes, and also a merge tool. The merge tool simplifies the process of merging changes to the same file made by different persons. TortoiseHg will automatically merge the changes so that nothing is lost, and if two people have made changes on the same row in the code, TortoiseHg has support for a visual diff/merge tool so the user can merge the code manually.

4 Theory

4.1 Odometry

Odometry, sometimes referred to as dead reckoning, is a technique that utilizes data from moving sensors in order to estimate change in position over time. The technique can be used to estimate one's current position relative to a starting position. Odometry can be used on various types of systems but most common is perhaps wheeled robots. In this case, the sensors can be tachometers that measure the rotation of the robot's wheels. In the special case of a differential wheeled robot, it is sufficient to know the wheel diameter and track width in order to estimate the travelled distance

and heading of the robot from the data given by the tachometers. If the robot started in a known position, its current position can be estimated by advancing the starting position based upon estimated distance travelled and heading. In other words, the position of the robot X at time t is given by a sum of discrete displacements over time since the beginning as

$$X_t = \sum_0^t \Delta X_t = X_{t-1} + \Delta X. \tag{1}$$

As long as the position change ΔX is measured apace, the position estimate is capable of being quite accurate. If odometry is used for localisation in a two dimensional environment the position can be described by the three states

$$X_t = [x_t, y_t, \theta_t]^T = [x_{t-1}, y_{t-1}, \theta_{t-1}]^T + [\Delta, x_t, \Delta y_t, \Delta\theta_t]^T \tag{2}$$

where x_t and y_t are the robot's position in a Cartesian coordinate system and θ_t is the heading of the robot. $\Delta x, \Delta y$ and $\Delta\theta$ are the change in the robot's position and heading. For a differential wheeled robot the state equation becomes

$$X_t = [x_{t-1}, y_{t-1}, \theta_{t-1}]^T + [D\cos(\theta_{t-1}), D\sin(\theta_{t-1}), \Delta\theta]^T \tag{3}$$

where D is the distance travelled. D and $\Delta\theta$ can be calculated from the data given by the tachometers. If the tachometer counts for the left and right wheel, T_l and T_r, are given in degrees the distance becomes

$$D = \frac{D_w}{2} \cdot \frac{T_l + T_r}{2} \cdot \frac{2\pi}{360} \tag{4}$$

where D_w is the wheel diameter, assuming both wheels are equal in size. The change in orientation is given by

$$\Delta\theta = D_w \frac{T_r - T_l}{2L} \tag{5}$$

where L is the track width, the distance between the robot's two driving wheels.

A problem with odometry is that it is very sensitive to errors. In time, odometric localisation will accumulate errors due to wheel slippage, inaccurate measurements of physical parameters, discrete sampling of wheel rotation, etc.

4.2 Triangulation

Triangulation is the task of determining an unknown position and orientation from angle measurements between the unknown position and other known positions. The triangulation algorithm used in the system described in this report is based on the power center of three circles and uses angle measurements to three known positions [10]. As the angles are measured they inherently have a measurement uncertainty which propagates to the position and orientation, together labeled pose, calculated in the algorithm. However, the relation between the uncertainty in the angular measurements and the uncertainty in the pose is not trivial. The relation depends on how all the involved positions are located relative to each other, i.e. their relative distances and angles. The triangulation method used here, though, provides an error measure, which itself is used in the calculation of the position, to which the error in pose is proportional, that involves all aspects of the geometrical configuration that influence the error.

The triangulation algorithm from Pierlot *et al.* [10] works as follows:

Given three bearing measurements, α_i, sorted in ascending order, and the positions $\{x_i, y_i\}$ with which the bearings are associated:

1. Compute modified beacon coordinates:
   $x'_1 = x_1 - x_2, \qquad y'_1 = y_1 - y_2, \qquad x'_3 = x_3 - x_2, \qquad y'_3 = y_3 - y_2,$
2. Compute three cotangents:
   $T_{12} = \cot(\alpha_1 - \alpha_2), \qquad T_{23} = \cot(\alpha_3 - \alpha_2), \qquad T_{31} = \frac{1 - T_{12}T_{23}}{T_{12} + T_{23}}$
3. Compute modified circle center coordinates $\{x'_{ij}, t'_{ij}\}$:
   $x'_{12} = x'_1 + T_{12}y'_1, \qquad y'_{12} = y'_1 - T_{12}x'_1$
   $x'_{23} = x'_3 - T_{23}y'_3, \qquad y'_{23} = y'_3 + T_{23}x'_3$
   $x'_{31} = (x'_3 + x'_1) + T_{31}(y'_3 - y_1), \qquad y'_{31} = (y'_3 + y'_1) + T_{31}(x'_3 - x_1)$
4. Compute k'_{31}:
   $k'_{31} = x'_1x'_3 + y'_1y'_3 + T_{31}(x'_1y'_3 - x'_3y'_3)$
5. Compute denominator D (if $D = 0$, return with an error):
   $D = (x'_{12} - x'_{23})(y'_{23} - y'_{31}) - (y'_{12} - y'_{23})(x'_{23} - x'31)$
6. Compute the robot's position $\{x_R, y_R\}$:
   $x_R = x_2 + \frac{k'_{31}(y'_{12} - y'_{23})}{D}, \qquad y_R = y_2 + \frac{k'_{31}(x'_{23} - x'_{12})}{D},$

4.3 Particle filter

A particle filter is a Sequential Monte Carlo method and it is designed for hidden Markov Models, which means that its purpose is to estimate the current state of a system whose states are unknown. A particle filter consists of N particles, each associated with a weight based on the probability of that particle being the correct state estimate. So instead of having one estimation of the state, the particle filter provides N estimates. Using a weighted sum of all the particles and their weights, a final state estimate can be found.

In each time step, all the particles are traversed using the model, with noise added to it. Then one uses observations to reweight the particles. So particles that are more likely to be true given the observation get an increased weight, and vice versa. This means that eventually, we will have only a few particles with significant weight while the other particles will have a very low weight. If the number of particles with significant weight drops below a certain threshold, so called resampling is done. Resampling is replacing the old particle set with N new particles. Each new particle is placed in the proximity of one old particle, and a higher weight of an old particle makes it more probable for a new particle to be resampled from it. So instead of having particles with large differences in their weights, we will now have a new set of particles all having the same weight. The state estimate given from the weighted sum of the new particle set will more or less coincide with the state estimate from before the resampling, so the reason of performing resampling is never to change the state estimate, but to prevent the weights from being too big or small for computers to handle.

4.4 Pathfinding

Pathfinding deals with finding the optimal path from point A to B in an environment. Here the A* (pronounced "ay-star") search algorithm is used to do this. A* is a widely used search algorithm first described in 1968 [11], and is the foundation of many search algorithms today.

A* works by discretizing the environment into nodes, which can be connected to each other in various ways. Each connection of two neighboring nodes has a cost associated with it and the A* algorithm finds the path between any two nodes in the environment that has the minimal total cost. The algorithm works as follows:

1. Keep a visited set V for all nodes that have been expanded and a frontier set F for all neighboring nodes of V.

2. Evaluate the function $g(n) = c(n) + h(n, n_{goal})$, where $c(n)$ is the cost to node n, and $h(n, n_{goal})$ is an optimistic heuristic estimate of the cost from node n to the goal, for all nodes n in F.

3. Expand the node which minimizes g and add it to V.

4. Add new neighbors to F.

5. Repeat (2-4) until the goal is reached.

6. The optimal path is found by tracing back from the goal to the start.

5 Implementation

5.1 Design and physical configuration

This section describes the robot design, see *Figure 4*, how the program is structured and how constant parameters were chosen.

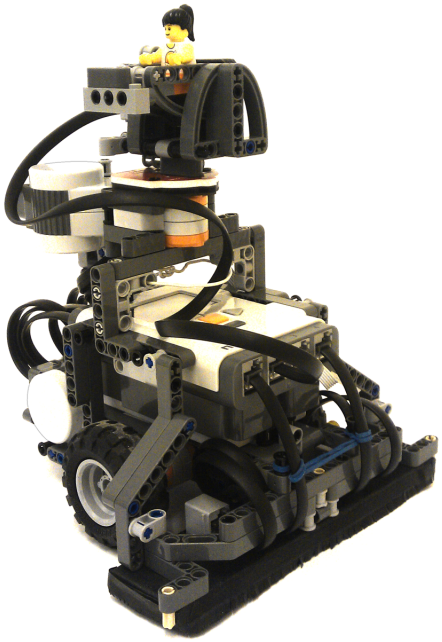


Figure 4. The robot.

The robot is designed on the principle of a differential wheeled robot, which means that it has two wheels on a common axle controlled by separate motors. In the rear of the robot is one small support wheel that can rotate freely. The NXT Intelligent Brick is mounted in the center of the robot with sensor ports facing forward, motor ports facing backwards and the LCD screen facing up. Beneath the brick are the two driving motors which are positioned close to each other to keep the design compact. The center of the wheel axle is below the center of the brick. In front of the wheels are two NXTLineLeader light sensor arrays mounted side by side to cover as much of the area in front of the robot as possible while driving. Above the brick is a motor with its axle pointing vertically, a GlideWheel-AS angle sensor and a NXTCam camera are mounted on this axle.

Since the NXT Intelligent Brick has a significantly lower processing power than a standard PC the computationally heavy tasks are performed on the PC while the brick handles simple tasks such as sensor input and motor control. Data clustering, data association, triangulation, particle filtering, pathfinding and map drawing are the tasks that are handled by the PC. This works by passing necessary information back and forth between

the brick and the PC via bluetooth.

The following parameters were used to control the different motors:

- Wheel diameter = 5.5765 cm
- Track width = 11.0113 cm
- Robot acceleration = 10 cm/s^2
- Robot travel speed = 20 cm/s
- Robot rotation speed = 80 degrees/s
- Camera rotation speed = 67 degrees/s

The wheel diameter was found by first setting the parameter to a rough estimate, derived from measuring the wheel with a ruler, then commanding the robot to drive a certain distance and measure the real travelled distance. The calibrated value of the wheel diameter, D_w, was then calculated as

$$D_w = \hat{D_w} \cdot D/\hat{D} \tag{6}$$

where $\hat{D_w}$ is the current estimate of the wheel diameter, D is the real travelled distance and $\hat{D}$ is the estimated travelled distance. After only one iteration the robot's estimated travel distance and the real distance were within a few millimeters on a travel distance of four meters. With the wheel diameter calibrated, the track width was found in a similar way; the robot was commanded to rotate 3600 degrees around it's own center. Again a rough estimate of the track width obtained by ruler, was used when performing the maneuver. The new estimate of the track width, L, was calculated as

$$L = \hat{L} \cdot \theta/\hat{\theta} \tag{7}$$

where $\hat{L}$ is the current estimate of the track width, θ is the real rotated angle and $\hat{\theta}$ is the estimated rotated angle.

The travel speed, rotation speeds and accelerations were chosen based on iterative testing as the values that seemed to provide the best system performance.

5.2 Localization

5.2.1 Camera

The Camera unit consists of one NXTCam mounted on a motor through a GlideWheel-AS angle sensor. The camera is mounted in such a way that the internal y-axis of the NXTCam is aligned with the horizontal plane. Reasons for this is to keep the cable between the camera and the brick out of the field of view and that the NXTCam has a higher resolution in the y-direction than the x-direction when in object tracking mode.

When calibrating the colormap for the NXTCam it appeared to have a bias towards the red part of the color spectrum when taking images. This makes it easy for the camera to detect red objects but other colors were almost impossible to detect. For example blue objects were almost always seen as black. Therefore the beacons were chosen to be red and were constructed from red paper folded into 30 cm high pillars with equilaterally triangular cross sections with a side length of 5 cm. An example of an image captured by the camera can be seen in *Figure 5*. One thing to be careful with when choosing a red colormap though is that the camera may see red objects where there should not be any red objects due to its bias, these false objects must be discarded as noise.

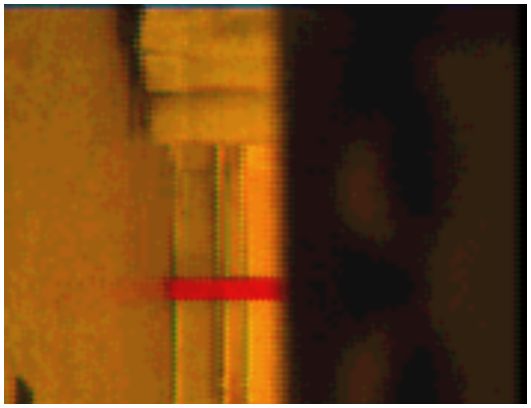


Figure 5. An image captured by the NXTCam. The red object is a beacon. The dark area to the right is intentionally obscured by a cover placed in front of the camera to avoid detecting objects above the horizontal plane.

Whenever the position needs to be triangulated the camera unit, consisting of the camera, camera motor and angle sensor, performs a horizontal 360 degree camera sweep and returns the bearings of all detected beacons. This sweep alternates between clockwise and counterclockwise to avoid winding up the cable. Bearings are calculated by translating the pixel coordinates of the center of the largest bounding box detected to an angle within the camera's field of view and adding this to the angle given from the angle sensor. This way of calculating the bearings makes it so that the beacons do not have to be in the center of the camera's field of view to be detected and returns more bearings for each beacon, which makes the final bearing estimation more precise. As a beacon is present in several sequential images as the camera sweeps across it, several bearing measurements are produced for each beacon, averaging these measurements gives a high precision bearing estimate.

The camera also have a few conditions for when to detect a beacon or not to avoid detection of false beacons. To detect a beacon the bounding box of that beacon has to be completely within the horizontal field of view. If the bounding box is at the edge that means that the camera only sees a smaller portion of the beacon and that would return bad angles. Another condition is that the area of the bounding box has to be larger than a certain area threshold to be detected. This threshold has to be larger than the smallest bounding box of a beacon in the environment, else the camera might miss a beacon during a sweep. A threshold of 100 pixels was chosen in our environment. Reasons for having this threshold is to avoid detecting false objects that are too small to be a beacon and are more likely to be

noise.

When the camera is done with a sweep it sends the detected bearings to the PC via bluetooth to be clustered and associated with the correct beacon and finally used to create an estimate of the robot's position via triangulation.

5.2.2 Bearings

The bearings to the beacons are calculated from data from the camera in combination with data from the angle sensor. When a beacon is seen in the image, the angle sensor measures the direction in which the camera is pointing and the image coordinates of the center of the beacon is used to determine the horizontal angle offset between the camera's optical axis (the line through the center of lens and the center of image) and the beacon. The sum of these two angles is the bearing to the beacon, i.e. the angle from the robot's forward direction to the beacon. The camera motor could also be used to measure the camera direction, but it has a lower precision than the GlideWheel-AS and also some hysteresis, possibly caused by the spacing between gear teeth within the motor. Assuming that the camera can be approximated by a pinhole camera model [12] horizontal image coordinates are related to the tangent of the horizontal angle to the optical axis, θ, as

$$x = f \cdot tan(\theta) \tag{8}$$

where f is the focal length of the pinhole camera model and x is the horizontal distance from the image center. With a small field of view the above equation can be well approximated by a first degree polynomial. This polynomial could be derived from the focal length and the size of the image sensor, but here it is instead estimated empirically by sweeping the camera at a known position in a known environment and then fitting the data from the camera and the angle sensor to the known environment. *Figure 6* shows raw data from such a sweep.

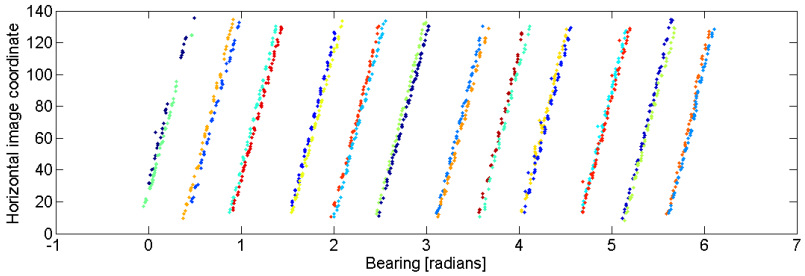


Figure 6. Raw data from the camera and angle sensor during a calibration sweep. Each slanted line is the result of sweeping across a beacon. Different colors signify different beacons or different directions of sweep, i.e. clockwise or counterclockwise rotation.

As visible in the image, straight lines are well suited to approximate the data. Since the beacons are static the true angle to the center of a beacon is constant during a sweep, provided that the robot is also static. The offset between the angle sensor reading and the real bearing to the center of a beacon is what is approximated by a first degree polynomial in the horizontal image coordinate. The bearings are calculated as the difference between the angle sensor reading and the offset. *Figure 7* shows a histogram of bearings calculated from the data in *Figure 6*.

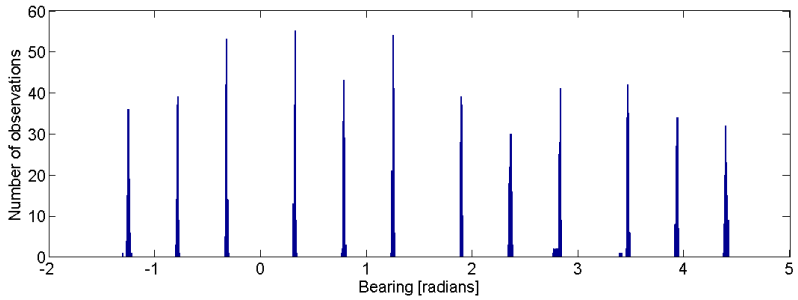


Figure 7. Histogram of bearings calculations from the data in *Figure 6*. Individual beacons are easily distinguishable. The data was collected over four 360 degree sweeps, meaning that the number of observations of each beacon are around four times greater than in the average single sweep performed during normal operation of the system.

The final estimate of the bearing to a beacon is calculated as the mean of all bearings associated with that beacon.

As can be seen in figure *Figure 6*, the lines have slightly different properties that can not simply be attributed to noise. In fact, they can be divided into two distinct groups based on whether the observations were made while sweeping in the positive or negative direction. A separate polynomial is derived for each case and the system switches between their use depending on sweep direction.

5.2.3 Clustering and data association

To associate individual bearings to the different beacons all the bearing measurements are first clustered. The clustering algorithm sorts the measurements in ascending order and loops through them, calculating the absolute difference between neighboring elements. The data is clustered so that no neighboring elements in the same cluster have a larger absolute difference than five degrees. If a beacon is located straight ahead of the robot, observations of it may be split across the two-pi-to-zero discontinuity. Therefore, if the difference between the first and the last measurement is less than five degrees, their respective clusters are merged. The mean of each cluster is passed to the data association algorithm.

Data association in this case means associating one of the clusters with each beacon. Estimated bearings are calculated from the pose estimate and the beacon locations and each beacon is associated with the cluster whose bearing is closest to the estimate.

5.2.4 Triangulation

As mentioned in the theory section, the size of the error in the triangulation depends on the relative positions of the robot and the three beacons used in the triangulation. The rest of the system is internally capable of handling any reasonable number of beacons. An upper limit set by the properties of the camera and other physical aspects of the system as each beacon must be distinguishable from the others. Whenever there are more than three beacons, the system can choose the combination of three of them that gives the lowest error measure. In the case of four beacons placed in the corners of a square area of operations (AO), the optimal choice is to use the beacons that are closest to, second closest to and farthest away from the robot's position, i.e. all but the second farthest beacon. With the robot's true position being unknown the choice of beacons is based on the current best estimate of the robot's position. *Figure 8* illustrates how the error factor behaves in this case and the optimal choice of beacons.

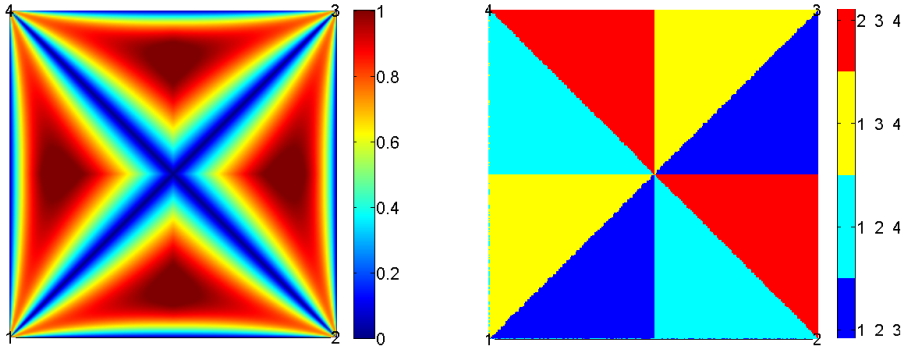


Figure 8. The images show the optimal selection of beacons (right) and resulting error factor (left) as a function of position inside a square with one beacon placed in each corner. The color in the left image is not in absolute scale.

5.2.5 Particle filter

To meet the objectives of this project an accurate estimate of the pose of the robot was needed. The robot had a quite accurate pose estimate itself using odometry, but only short-term. The odometry is provided by classes already implemented in the LeJOS library and utilizes the same equations as described in the theory-section.

For long-term, odometry cannot be used since the error will accumulate and hence, the estimate will diverge. So, one can use the estimate from the odometry, but then the estimate needs to be corrected somehow. This was done using the camera and triangulation, from which estimates of the pose was received. To fuse these estimates together to get a corrected and hopefully better estimate, a particle filter was used, since they are designed for hidden Markov Models. The unknown state in this case was the pose of the robot and the observations was the pose received from the triangulation.

Since we are not only interested in knowing the position of the robot, but also the heading, a choice had to be done about the particle filter. Either one particle filter could be used, with particles containing information about both the position and heading, or two different particle filters, with particles containing information about position in one of the filters and particles containing information about the heading in the other. The former was in the end chosen since it proved to give better results. Here all particles were traversed using odometry, as explained, but only the position from the triangulation pose was used as the observation to weight the particles.

This showed to be preferable and the reason for this is that given that the particles have converged to the right position at some time t, the next time we call the particle filter and have a new observation, particles that had the right heading at time t have a greater likelihood of being close to the position that was retrieved from the triangulation. This means that the particles with the correct heading now get a higher weight.

The number of particles that was used in this filter was N=500. The more particles used, the better the final state estimate gets since you then avoid noise due to the stochastic properties of the filter. On the other hand, the more particles you use, the more computationally heavy the program gets, naturally. So the number of particles that should be used completely depends on the system and is a trade-off between low execution time and precision.

5.3 Mine detection

About every 20 milliseconds, which corresponds to 4 mm when travelling at 20 cm/s, the robot uses its light sensors to investigate if any sensor sees a mine. If there are any indications of a mine under one of the sensors, the robot stops immediately. After stopping, the robot investigates if any sensor still sees the mine. If so, the pose and a boolean array containing information of which sensors sees the mine is sent to the computer via Bluetooth. Otherwise it investigates whether it was a false alarm, or if there really was a mine there but the robot drove too fast past it so the sensors cannot longer see it. If the robot was travelling forward, it slowly goes backwards to investigate if it can see the mine again (travels maximum 5 centimeters backwards), and if the robot was rotating into the mine, it slowly rotates back (rotates maximum 20 degrees or the maximum angle it rotated during the whole last movement). If it sees the mine again doing this, the robot stops again and the pose and boolean array is sent to the computer. It then rotates back so it has the same heading as it had before the movement during which the mine was found (if it was rotating into the mine), and then it travels backwards 10 centimeters. This is done to create some distance between the robot and the mine so it in the next step can find a path around the mine more easily, which the robot then starts to follow. Otherwise, if the robot never found the mine again, it sees it as a false alarm and continues travelling towards the goal it was about to go to, without sending any information about a mine to the computer.

5.4 Map representation

Visual feedback on the performance of the system is provided by an augmented reality video feed displayed on the computer (see *Figure 9*). The video feed is provided by a webcam, seeing the whole area of operations (AO), through the use of a Java webcam capture library [13] and is overlayed with graphical representations of various data computed by the system. The user marks the four corners of the AO in the image from the video feed and the system uses this information to sample the pixels in each frame according to a set of affine and projective transformations, based on the work by Zhang [14], that map the AO to the entire image. Assuming that the user marked the location of the corners accurately and precisely, the video feed and the graphical overlay match up perfectly.

Detected mines are entered into the map as the positions of the individual light sensors that saw the mine at the time of detection. These positions are derived from the physical configuration of the robot and the pose at the time of detection. A circular area centered on each detecting light sensor is marked as clearance for the pathfinding algorithm, meaning that the system can not produce paths going into or through those areas, decreasing the probability of the robot driving over a mine.

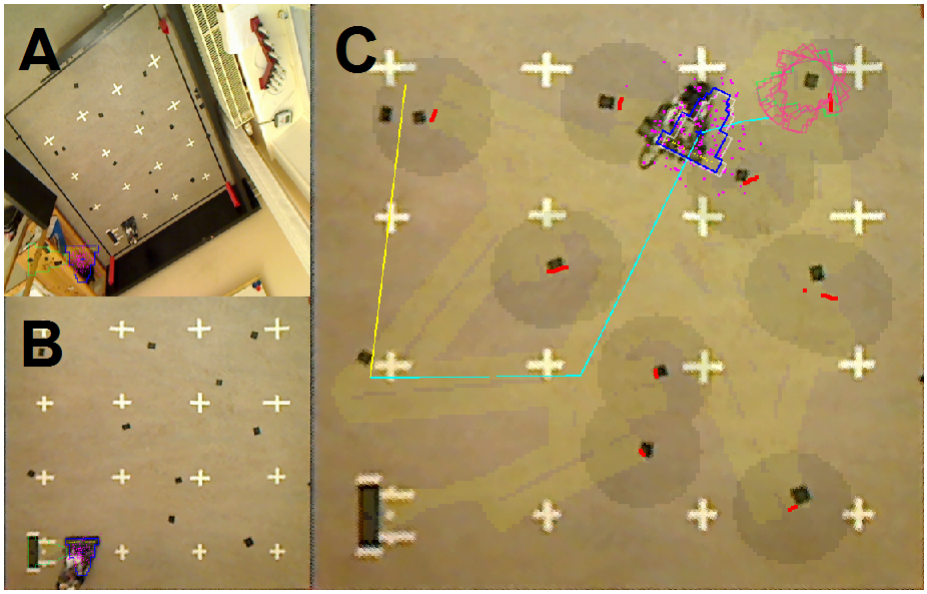


Figure 9. A: Image from video feed. B: Transformed image. C: After some driving around. Note the difference between the estimated pose and the actual pose. Contends described in *Table 1*.

Table 1. Legend to *Figure 9*.

Map item	**Description**
White cross	Physical reference marks
Black square	Mine or calibration area
Blue outline	Final pose estimate
Gray outline	Pure odometry pose
Green outline	Triangulated pose
Red dots	Marks detected mines
Cyan line	Current path
Yellow line	Line to next goal
Dark circles	Clearance
Magenta dots	Particle filter particles
Brightened area	Scanned area

5.5 Pathfinding

The pathfinding algorithm used to calculate the robot's path to a desired goal uses A* as a first step. The theory about A* is quite straightforward and is described in the theory part. The environment is discretized into nodes which are arranged in a four-connected grid mesh. This basically

means that each node except for the ones on the edges are connected to four other nodes. The nodes can be represented as squares with a certain side length. A suitable value of this side length was for our implementation 1 cm. This means that the path calculated by A* had a precision of 1 cm. The AO was in most cases $155x155$ cm, with a node size of 1 cm this means that the total number of nodes were $155 \cdot 155 = 24025$.

As stated in the theory part, each connection between two nodes has a cost associated with it. This cost is initially set to one for all connections since the robot has no prior knowledge of the environment. However, when the robot discovers a mine, the cost of the connections to the node corresponding to the position of the discovered mine is set to a high value (= a value higher than the total number of nodes). The cost of the connections to and between nodes corresponding to clearance is also set to an equally high value.

The A* algorithm also requires the use of an optimistic heuristic guess in order to estimate the cost from the evaluated node to the goal. This heuristic guess was implemented as the Manhattan-distance from the evaluated node to the goal divided by the node size.

The A* algorithm was implemented in Java. Before a path can be calculated, information about mines from the map must first be translated into the grid mesh. This is due to the fact that the map and the grid mesh are two different representations of the environment but where the grid mesh only stores the information needed to calculate a path. The output from the A* algorithm is a Path object (a set of waypoints) that leads to the goal. If the robot is only allowed to make 90 degree turns, this path is always optimal. However, since the robot has no restrictions on how it can turn, a second step was introduced. The purpose with this second step is to only keep those waypoints needed to maintain a "mine-free" path. This is done by systematically removing one waypoints at the time, beginning with the waypoint next to the goal, and check whether the new path intersects any nodes represented as a mine or clearance. If it does, the waypoint is kept in the path, otherwise it is discarded. To improve the solution even more, the second step is calculated twice. The second time waypoints are systematically removed from the start instead of from the goal of the path. The shortest of the two new paths is chosen as the final solution. However, if the two paths have common waypoints, then the shortest sub-paths are merged together into a final solution. This can be seen in *Figure 10*

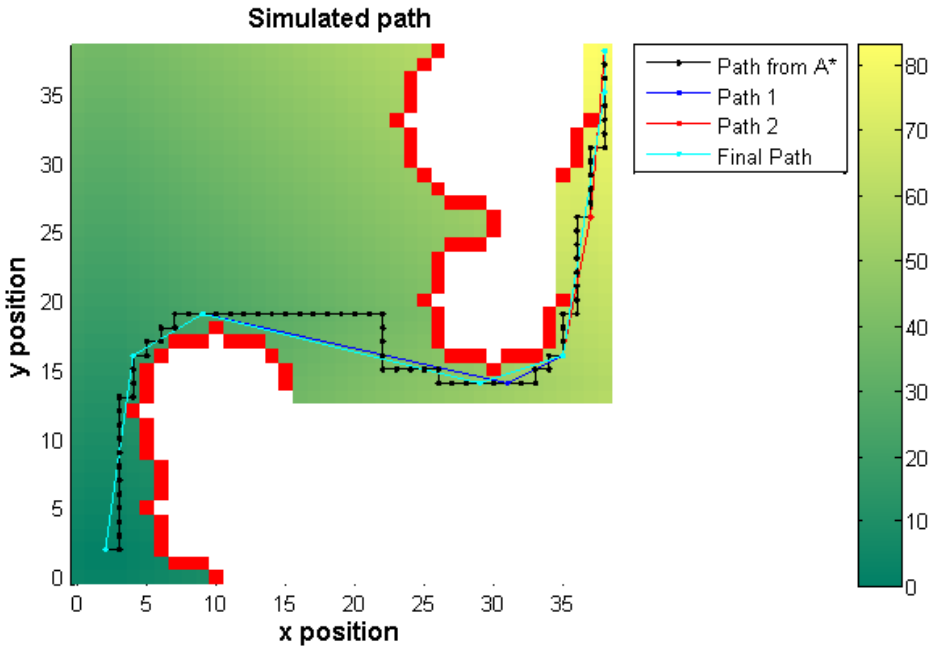


Figure 10. Simulated path where the start is in the bottom left corner and the goal is in the top right corner. Red areas indicates obstacles (mines or clearance) and the colourbar on the right shows the cost to reach evaluated nodes. White areas have not been evaluated by the A* algorithm. The cyan coloured path is created by merging the blue and red paths together.

When the A* algorithm has found the goal, the optimal path is found by tracing back from the goal to the start. The way this is implemented is that each node stores its neighboring node with lowest $g(n)$. Quite often more than one neighbor have an equally low value on $g(n)$, for these cases a random parameter was introduced in order to choose which one to store. This randomness seemed to improve the solution for some configurations of mines (compare *Figure 12* and *11*).

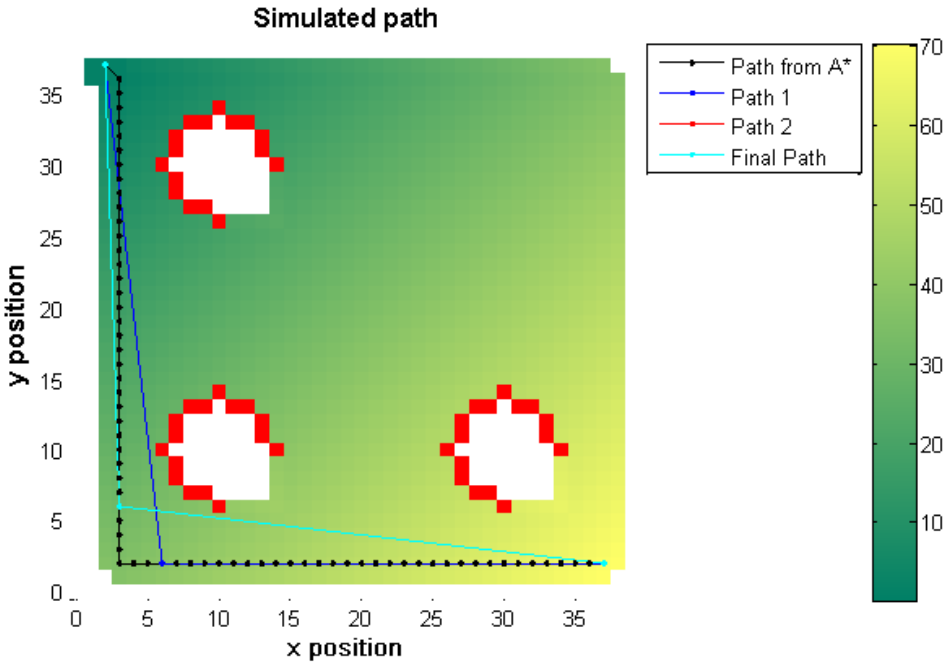


Figure 11. Simulated path where the start is in the top left corner and the goal is in the bottom right corner. Red areas indicates obstacles (mines or clearance) and the colourbar on the right shows the cost to reach evaluated nodes. White areas have not been evaluated by the A* algorithm. No randomness is here used to choose which neighbouring node to store.

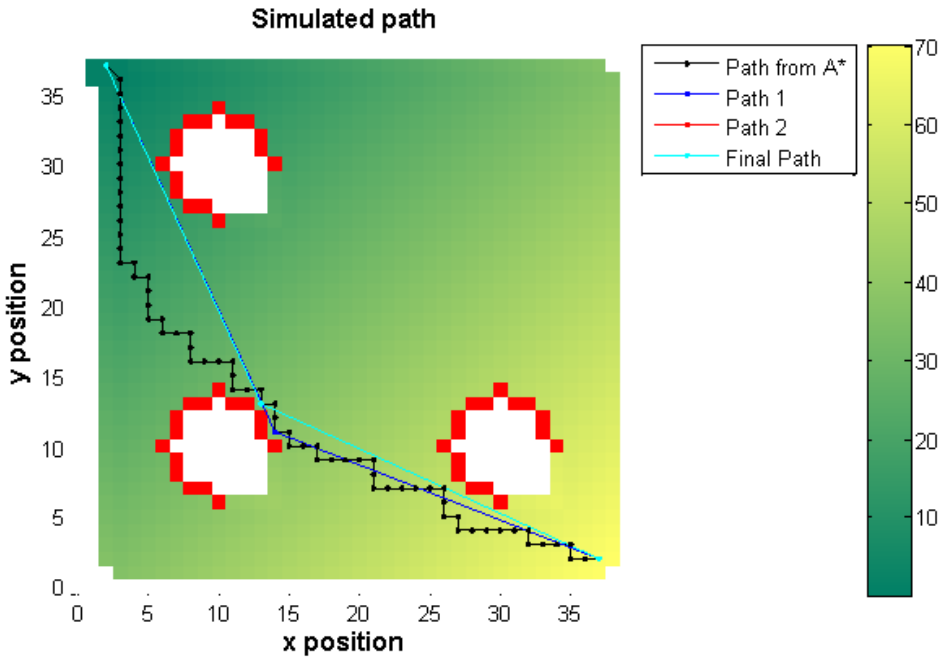


Figure 12. Simulated path where the start is in the top left corner and the goal is in the bottom right corner. Red areas indicates obstacles (mines or clearance) and the colourbar on the right shows the cost to reach evaluated nodes. White areas have not been evaluated by the A* algorithm. A random parameter is here used to choose which neighbouring node to store.

There are situations where the robot cannot drive to the desired goal without passing through a mine or clearance. This can for example happen if the goal is right on a mine or if the robot is trapped inside a circle of mines and the goal is outside this circle. In these situations the pathfinding algorithm throws an exception which must be handled by the rest of the program.

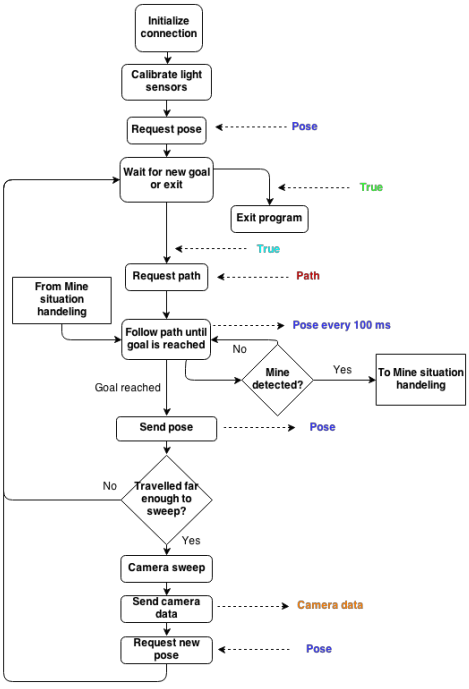
5.6 Bluetooth

The communication between the brick and the PC is implemented with a starting point in a client-server architecture where the brick is the client and the PC is the server. This is in the sense that the brick is in charge of the communication, e.g. the brick requests commands from the PC rather than waiting idly to receive commands and it voluntarily sends sensor information to the PC rather than doing so after a request from the PC. Each package of data is prefixed with an integer that tells the receiving part what type of

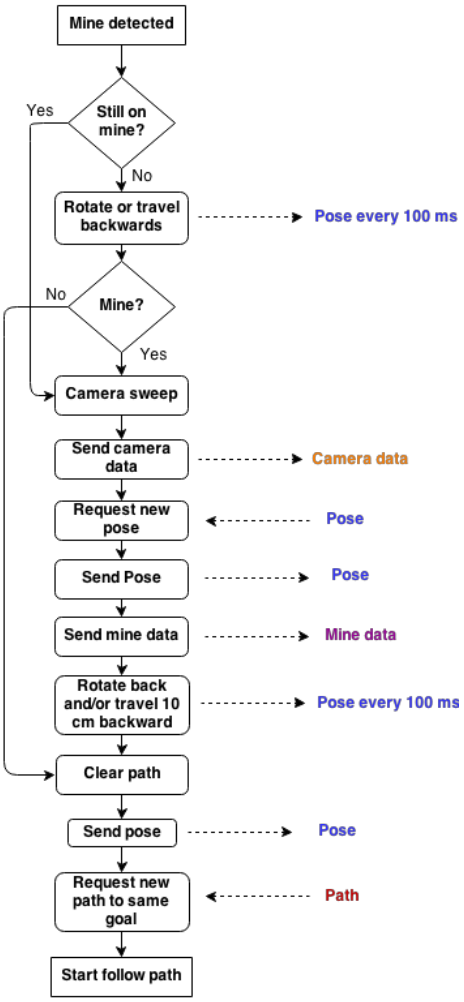
data it is.

5.7 Complete system description

The *Figure 13* and *14* illustrates a slightly simplified model of the complete system. *Figure 13* corresponds to events that occurs on the Brick while *Figure 14* corresponds to events on the PC. *Figure 13a* describes the general driving process while *Figure 13b* describes the actions taken by the robot if it observes a mine. The dashed arrows pointing to the right in the three figures represents information sent via bluetooth from the Brick to the PC and dashed arrows pointing to the left represents information sent from the PC to the Brick.



(a) Robot flowchart



(b) Mine handling flowchart

Figure 13. Flowchart of the program that runs on the NXT Intelligent Brick

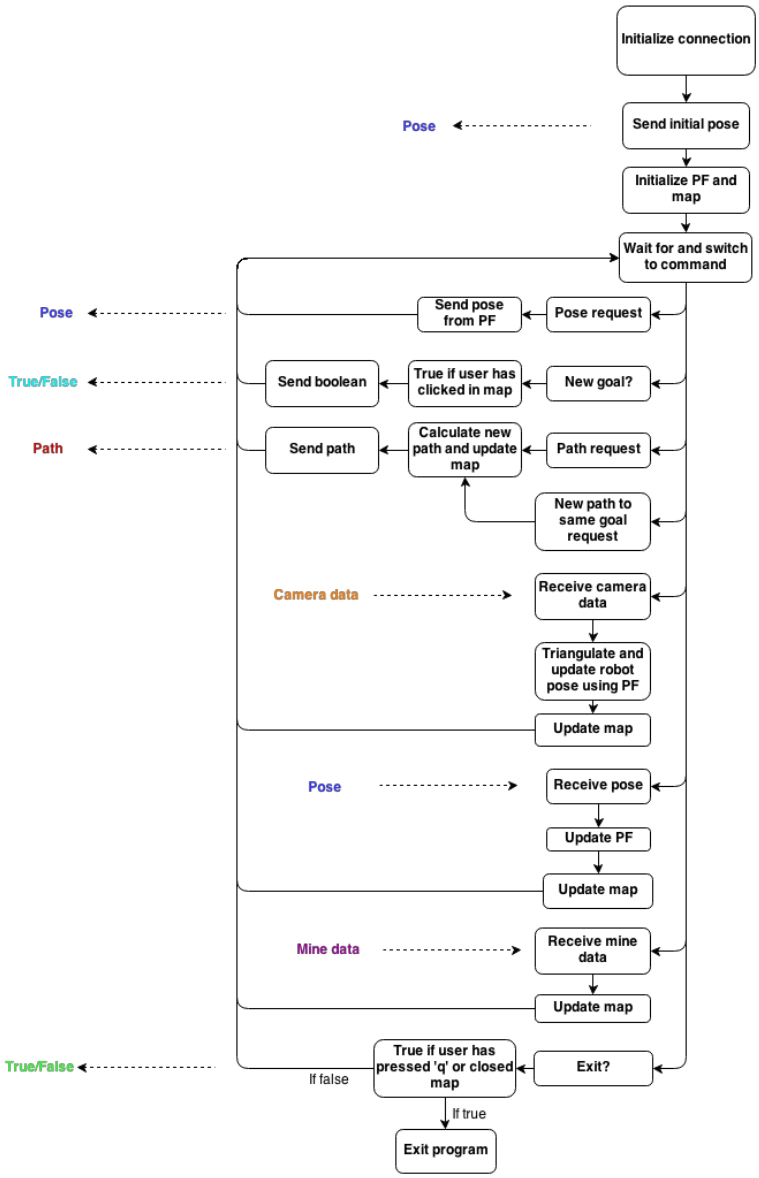


Figure 14. Flowchart of the program that runs on the PC

6 Results and Discussion

6.1 Localization

As expected the estimated position from odometry is very accurate in the beginning, but gets worse as the robot travels further. Also when performing quick maneuvers, such as an immediate stop when detecting a mine, the odometry becomes significantly worse.

The position from the camera improved significantly when choosing optimally from four beacons instead of three, making the estimation error several orders of magnitude lower. The position received after particle filtering is better than the position from odometry most of the time since the error in the position from the camera is not accumulating. The error in the particle filter's estimated position is largely dependent on what situation the robot is in. In the worst case the robot has traveled far and accumulated a large error in the odometry and performs a scan in a position nearly concyclic with the beacons.

The precision in the bearing measurements is good. The accuracy, however, could be further improved. As can be seen in *Figure 7*, the coefficients of the polynomial used for calculating the bearings clearly vary with the direction of the camera in ways that can not be considered as simply noise. This may be due to the camera tilting in different directions, both sideways and back-and-forth, depending on the angle of rotation. Examples of possible causes for this is the camera axle not being exactly vertical or the connecting cable tugging in different directions as it is being wound up or down. This could be accounted for by compensation calculated from measurements of the roll and pitch angles of the camera, or by imposing more rigid physical constraints to prohibit the camera from moving in unwanted ways.

The current accuracy of the system is usually within one degree. This may seem good, and it is true that the triangulated position is often good. However, even a small error in the heading accumulates to a large error in the position while the robot is driving. This problem can be avoided for example by triangulating more often, or by getting more accurate bearings.

6.2 Mine detection

As said earlier, the robot uses its light sensors about every 20 milliseconds to investigate if any of the sensors sees a mine. However, sometimes this takes much longer. The reason for this is probably that each 100 millisecond we send information about the robot's pose to the computer via bluetooth.

This usually causes a delay so it takes about 50 milliseconds before the next sampling of the light sensors is done. This is however not an issue since this corresponds to 1 cm when travelling at a speed of 20 cm/s, and we can assume that the mines are bigger than 1 cm^2. However, sometimes this takes much longer than 50 milliseconds. We have seen that it can take up to 10 seconds! This corresponds to a travel distance of 2 whole meters without searching for mines. This happens very seldom, but it is still a big issue. A solution to this, something that has not been done but can quite easily be implemented, is to have an own thread for the mine detection. Then we could choose the sampling rate of the light sensors ourselves such that we would not miss a mine.

6.3 Pathfinding

The pathfinding algorithm that was implemented on the robot and that was described earlier does not guarantee that the resulting path is optimal in terms of Euclidean distance. This is due to the fact that it is based on a four-connected grid mesh which assumes that the robot only can make 90 degree turns. However, the systematic removal of waypoints and the introduced randomness improves the solution to such an extent that in a majority of all situations it is very hard to visually identify a shorter path. For less complicated mine configurations the final path is in fact very likely to be the optimal one, even in terms of Euclidean distance.

A pathfinding algorithm based on an eight-connected grid mesh would probably improve the pathfinding, especially when the mine configuration gets more complicated. However, this improvement would not be significant and it would also increase the complexity of the algorithm.

If one wants to improve the way the robots travels from one point to another it is better to improve the robot's navigation technique instead of further improve the pathfinding. As it is now, the robot must stop every time it should rotate, it cannot turn while it is driving forward or backward. This makes the robot unable to travel in an arc and it also stops unnecessary many times on its way to the goal.

7 References

[1] Xander. *Comparing the NXT and EV3 bricks.*

`http://botbench.com/blog/2013/01/08/comparing-the-nxt-and-ev3-bricks/` [2014-01-15]

[2] MindSensors. Vision Subsystem v4 for NXT or EV3 (NXTCam-v4)

`http://www.mindsensors.com/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=78` [2014-01-06]

[3] NXTCamView.

`http://nxtcamview.sourceforge.net/` [2014-01-08]

[4] MindSensors. Line Sensor for NXT (NXTLineLeader).

`http://www.mindsensors.com/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=111` [2014-01-06]

[5] MindSensors. GlideWheel-AS - Angle Sensor for NXT or EV3.

`http://www.mindsensors.com/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=173` [2014-01-06]

[6] LeJOS, Java for Lego Mindstorms.

`http://www.lejos.org/nxj.php` [2014-01-06]

[7] The Eclipse Foundation. Eclipse.

`http://www.eclipse.org/` [2014-01-08]

[8] David Soria Parra. Mercurial SCM.

`http://mercurial.selenic.com/` [2014-01-06]

[9] Atlassian Bitbucket.

`https://bitbucket.org/tortoisehg/thg/wiki/Home` [2014-01-06]

[10] V. PIERLOT, M. VAN DROOGENBROECK, and M. Urbin-Choffray. *A new three object triangulation algorithm based on the power center of three circles. In Research and Education in Robotics (EUROBOT), volume 161 of Communications in Computer and Information Science, pages 248-262, 2011. Springer.*

[11] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). *"A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100–107.*

[12] Fusiello, A. *Elements of Geometric Computer Vision.*

`http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FUSIELLO4/tutorial.html#x1-740008` [2014-01-06]

[13] Firyn, B. Webcam Capture, Generic Webcam Java Utility.

`http://webcam-capture.sarxos.pl/` [2014-01-09]

[14] Zhang, R. ECE661 Computer Vision Homework 2, Image Rectification: Remove Projective and Affine Distortions.

`https://engineering.purdue.edu/kak/courses-i-teach/ECE661.08/solution/hw2_s2.pdf` [2014-01-09]

8 Appendix A. Division of labour

8.1 Joakim Karlsson

- Implemented the pathfinding in Java.
- Worked partially on the particle filter.
- Worked on a simulation of a Kalman filter for the localization.
- Worked on implementing clustering and data association in java.
- Worked on the mine detection handling.

8.2 Fredrik Olsson

- Set up the LeJOS programming environment.
- Researched pathfinding (A*).
- Implemented a search algorithm for the camera.
- Performed tests of the camera's accuracy.
- Implemented Bluetooth communication.
- Worked on implementing clustering and data association in java.
- Worked on fusing the particle filter, bluetooth communication and camera code.
- Worked on the graphical user interface.
- Worked on the calibration of the light sensors.

8.3 Martin Stålberg

- Implemented a LeJOS driver for the GlideWheel-AS Angle Sensor.
- Researched the triangulation method.
- Developed a calibration algorithm for the camera
- Developed and implemented algorithms for processing the camera data.
- Came up with the odometry parameter calibration procedure.

- Worked on the calibration of the light sensors.
- Implemented the graphical user interface.
- Implemented webcam integration with image transformation.
- Worked on the Bluetooth communication.
- Iterated extensively on optimising the robot's design.

8.4 Mikaela Åhlén

- Set up the repository.
- Did a simulink model of the triangulation.
- Implemented particle filters.
- Worked on fusing the particle filter with the rest of the code.
- Implemented the Bluetooth communication between the brick and the PC.
- Implemented the mine detection and what the robot does when it detects a mine.
- Translated Martin's data association algorithm from Matlab code to Java.
- Worked a little bit on the calibration of the light sensors