Making a Map-making Robot:
Using the IKAROS System to Implement the Occupancy Grid Algorithm.

Rasmus Bååth, Birger Johansson
Lund University Cognitive Science
Kungshuset, Lundagård, 222 22 Lund
rasmus.baath@lucs.lu.se, birger.johansson@lucs.lu.se

Abstract

This paper describes an implementation of the *occupancy grid algorithm*, one of the most popular algorithms for robotic mapping. The algorithm is implemented on a robot setup at Lund University Cognitive Science (LUCS), and a number of experiments are conducted where the algorithm is exposed to different kinds of noise. The outcome show that the algorithm performs well given its parameters are tuned right. The conclusion is made that, in spite of its limitations, the occupancy grid map algorithm is a robust algorithm that works well in practice.

1 Introduction

Maps are extremely useful artifacts. A map helps us relate to places we have never been to and shows us the way if we decide we want to go there. For an autonomous robot a map is even more useful as it could, if it is detailed enough, serve as the robot's internal representation of the world. The field of robotic mapping is quite young and started to receive attention first in the early 80s. Since then a lot of effort has gone into constructing robust robotic mapping algorithms, but the challenge is great as the way a human intuitively would build a map can not be directly applicable to a robot. Whereas a human possesses superior vision sensors and can locate herself by identifying landmarks, a robot, most often, only have sensors that approximates the distance to the closest walls. The conditions of robotic mapping actually closer resembles the conditions for a 15th century ship mapping uncharted water. Similar to the ship the robot only knows the approximate distance to the closest obstacles, it could happen that all obstacles are so far away that the robot senses void and it is often difficult for the robot to keep track of its position and heading. As opposed to the ship, a robot using a faulty map will bump into walls in a disgraceful manner, while the ship, on the other hand, might discover America.

A long-standing goal of AI and robotics research has been to construct truly autonomous robot's, capable of reasoning about and interacting with their environment. It is hard to see how this could be realized without general robust mapping algorithms.

1.1 The Approach of this Paper

This paper describes an implementation of a map building algorithm for a robot setup at LUCS. The main characteristics of the robot setup are that the environment is static and that the pose is given, therefore it does not induce all the difficulties mentioned above. The given pose is not without noise but there will never be the problem with cumulative position noise. Even if the problem is eased it is still far from trivial thus interesting in its own right. The setup will be further described in section 2. Given these precondition the *occupancy grid map* algorithm, first described by Elfes and Moravec [3], was chosen. The occupancy grid map algorithm was implemented and a number of experiments were conducted to investigate how it would perform given different types of sensor noise. The results of the experiments are presented in section 3.2.

1.2 The Occupancy Grid Map Algorithm

The occupancy grid map algorithm was developed in the mid 80s by Efes and Moravec and is a *recursive Bayesian estimation* algorithm. Here recursive means that in order integrate an nth sensor reading into a map no history of sensor readings is necessary. This is a useful property which implies that sensor readings can be integrated online and that the space and time complexity is constant with respect to the number of sensor readings. The algorithm is Bayesian because the central update equation is based on *Bayes theorem*:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

which answers the question "what is the probability of A given B", if we know the probabilities $P(B|A), P(A)$ and $P(B)$.

The map data structure is a grid, in 2D or 3D, that represents a region in space. This paper will treat the 2D case, thus the region is a rectangle. The value of each cell of the grid is the estimated probability that the corresponding area in space is occupied. The region corresponding to a cell is always considered completely occupied or completely empty. One can have different definitions regarding whether a region is free or occupied, but often a region is considered occupied if any part of it is occupied.

The algorithm consists of two separate parts: the update equation and a sensor model. The update equation is the basis of the algorithm and does not have to change for different robot setups. The sensor model on the other hand depends on the robot setup and each robot setup requires a customized sensor model. One can construct sensor models in many ways but the basic approach is described in section 1.3.

The computational complexity of the algorithm depends on the implementation of the sensor model. Apart from that, each update loop have time complexity $O(n'm')$, where n' and m' are the number of columns and rows of the grid that are affected by the current sensor reading. The space complexity is $O(nm)$ where n and m are the total number of columns and rows of the grid. An accessible introduction to occupancy grid maps is given by Elfes [2].

The original algorithm is limited in several ways. It requires that the robot's pose is given, thus it can not rely solely on the odometry of the robot. It presumes a static environment or requires sensor readings where dynamic obstacles have been filtered. Finally the area to be mapped has to be specified in advance. This might sound like severe limitations but in many robot setups one can assume a static environment and that there is a way to deduce the robot's pose. The original algorithm has also been successfully extended to deal with e.g. unknown robot poses [7].

1.3 The Inverse Sensor Model

A *sensor model* is a procedure for calculating the probability $P(s_t|m, p_t)$, that is the probability to get sensor reading s_t given map m and pose p_t at time t. Therefore it follows that the procedure for calculating $P(m|s_t, p_t)$ is called an inverse sensor model, that is the probability of m given only one sensor reading. An inverse sensor model can be though of as function $\texttt{ism}(s_t, p_t)$ that returns a grid the size of g where the probabilities of $P(m|s_t,p_t)$ are imprinted. There is not only one correct way to construct $\texttt{ism}(s_t, p_t)$ for a given sensor, different approaches have different advantages.

An example of how the output of an inverse sensor model could look is given in figure 1.

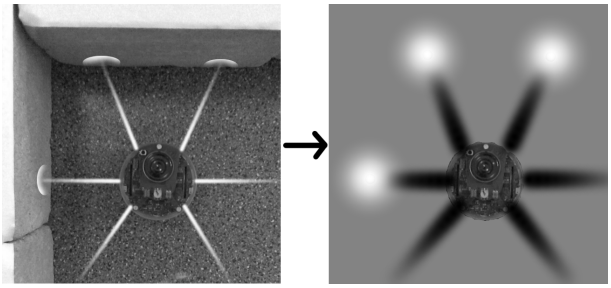


Figure 1: Illustration of an inverse sensor model for a robot equipped with infra-red proximity sensors.

The picture to the left show what the robot senses. The picture to the right is the resulting occupational probabilities. White denotes occupied space,black denotes free space and gray denotes unknown space. Notice how the black strokes fade with the distance to the robot. This indicates that the probability that a sensor detects an obstacle decreases with the distance to the obstacle.

An inverse sensor model can be built by hand or learned, for an example of the first see Elfes and Moravec [3] or the one described in section 2.1.5, for an example of the latter see Thrun et al. [6].

2 Implementation

In order to understand the design choices made a description of the robot setup will first be given, then the implementation will be described. The setup is currently used in the ongoing research regarding robot attention and one purpose of the implementation was that it should be possible to use in this context.

2.1 The Robot Setup

The robot used is the *e-puck*, a small, muffin sized robot developed by École Polytechnique Fédérale de Lausanne (*www.e-puck.org*). Its a differential wheeled robot boosting eight infra-red proximity sensors, a camera, accelerometer and Bluetooth connectivity. The e-puck also have very precise step motors to control its wheels. One problem is that no matter how

precise the e-pucks odometry is it can not solely be used to determine the robot's poses. Another problem is the proximity sensors of the e-puck. They have very limited range, roughly 10 cm, and are sensitive with respect to light conditions.

In order to remedy these problems a video camera has been placed in the ceiling of room where the robot experiments take place. The robots movements are restricted to a $2 \times 2\ m^2$ "sandbox" and objects in this area have been given color codes. Robots are wearing bright red plastic cups, the floor, the free space, is dark gray and obstacles are white. Images from the camera are processed in order to extract the poses of the robots and an image where only the obstacles are visible. Given this image and a robot's pose a circle sector is cut out of the image, its center being the robot's position and its direction being the robot's heading. By using this as the robot's sensor reading the robot can be treated as if it had a high resolution proximity sensor. The robots are controlled over Bluetooth link.

2.1.1 Ikaros

The whole system is implemented using Ikaros, a multi-purpose framework developed at LUCS. Ikaros is written in C++ and is intended for, among other things, brain modeling and robot control. The central concept in Ikaros is the *module*, and a system built in Ikaros is a collection of connected module's. An Ikaros module is simply put, a collection of inputs and an algorithm that works on these, the result ending up in a number of outputs. A module's inputs and outputs are defined by an Ikaros control file using an XML based language while the algorithm is implemented in C++.

A module's outputs can be connected to other module's inputs and to build a working system in Ikaros you would specify these connection in a control file. In this control file you could also give arguments to the modules. The data that can be transmitted between modules can only be in one format, that is arrays and matrices of floats. An Ikaros system works in discrete time-steps, so called "*ticks*". Each tick every module receives input and produces output.

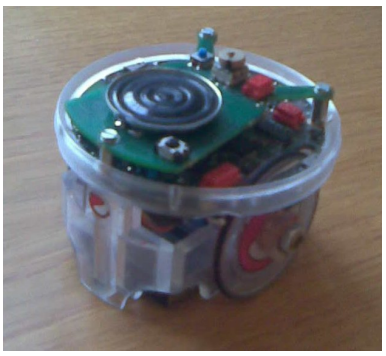


Figure 2: The e-puck.

Ikaros comes with a number of modules, both simple utility modules and more advanced such as several image feature extraction modules. Ikaros also includes a web interface that can display outputs in different ways. For a detailed introduction to Ikaros see Balkenius et al. [1].

2.1.2 Overview of the System

The core of the map drawing system consists of five modules: `Camera`, `Tracker`, `CameraSensor`, `SensorModel` and `OccupancyGridMap`. Further modules could be added to the system, e.g. a path planning module and a robot controller module. The connections between these modules are given in figure 3.

2.1.3 `Camera` and `Tracker`

The `Camera` and `Tracker` modules were already available and will only be described briefly.

The `Camera` module is basically a network camera interface and it is used to fetch images from the camera mounted in the ceiling. It outputs three matrices; `RED`, `GREEN` and `BLUE`, the size of the image, containing the corresponding color intensities of the image.

These matrices are fed into the `Tracker` module that extracts the poses of the robots and the positions of obstacles in the image. It outputs one array `POSITION` with the positions of the robots, one array `HEADING` with the headings of the robots and one matrix `OBSTACLES` with the obstacles extracted from the picture. `POSITION` is of the form $[r1_x, r1_y, r2_x, r2_y \ldots]$ where rn_x and rn_y is the nth robots x and y coordinate receptively. x and y are in the range 0.0 to 1.0 and the origo is in the upper left corner of the image. `HEADING` is of the same form as `POSITION` except for that rn_x and rn_y define a direction vector for the nth robot. The POSITION and `HEADING` will be referred to as the `POSE`. `OBSTACLES` is in the form of an occupancy grid over the area covered by the camera image, where 1.0 denotes an obstacle and 0.0 denotes free space.

2.1.4 `CameraSensor`

The `CameraSensor` module simulates a high resolution proximity sensor. It requires a matrix in the form of Tracker's `OBSTACLES` matrix and an array with the position of a robot as inputs. More specific we

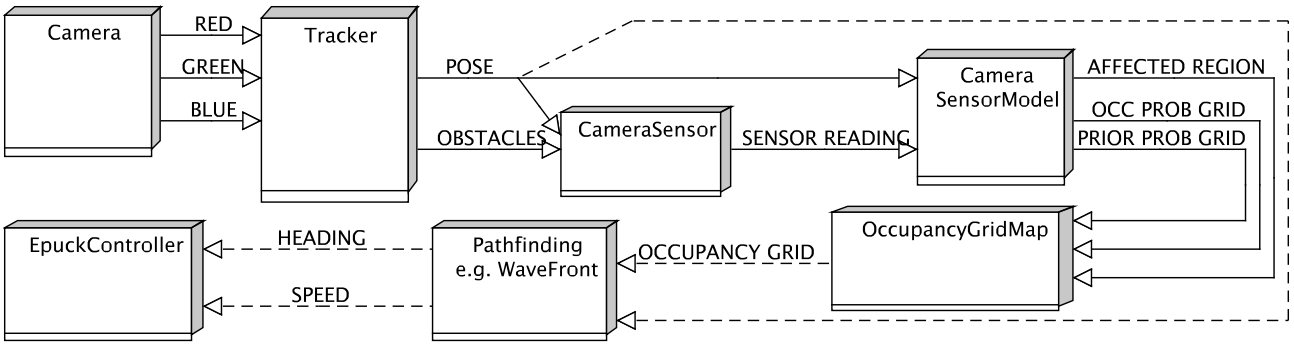


Figure 3: The connections between the modules of the map drawing system, with added path planning and robot control modules.

want to simulate a top mounted stereo camera. The `CameraSensor` module takes arguments specifying he range of the camera and the breadth of the view. Given the pose of the robot a square is cut out of the matrix, this square is rotated and projected onto another matrix representing the `SENSOR READING` of the robot. The `SENSOR READING` shows everything in the cut out square, even obstacles behind walls. Some simple ray-casting will solve this. Rays are shot from the center of the robot to the edge lying on the opposite side of the `SENSOR READING` matrix so that the cells touched by the rays form a circle sector. If a ray hits an obstacle the ray stops and all cells not touched by any ray obtains the value 0.5 indicating it's not part of the sensor reading. `CameraSensor` then outputs `SENSOR READING`.

2.1.5 `CameraSensorModel`

The `CameraSensorModel` is an inverse sensor model tailored to work with the output of the `CameraSensor`. `CameraSensorModel` has two outputs, both required by `OccupancyGridMap`: `AFFECTED GRID REGION` and `OCC PROB GRID`. `OCC PROB GRID` is a matrix the same size as the final occupancy grid that contains the probabilities $P(m|s_t, p_t)$. `AFFECTED GRID REGION` is an array of length four defining a box bounding the area of the occupancy grid that is affected by the `OCC PROB GRID`. The rationale behind this is that `OccupancyGridMap` should not have to update the whole occupancy grid when only a small area of it is affected by the current `SENSOR READING`.

The `SENSOR READING` from `CameraSensor` is already in the format of an occupancy grid, so transforming this into `OCC PROB GRID` in the format the `OccupancyGridMap` module requires, is pretty straight forward. First `OCC PROB GRID` is initialized with $P(m)$, the prior probability, given as an argument to `CameraSensorModel`. Then the `SENSOR READING` is rotated and translated, according to the robot's pose, so that it covers the corresponding area of the `OCC PROB GRID`. The `SENSOR READING` is then imprinted on the `OCC PROB GRID`. The values of `SENSOR READING`; 1.0, 0.5 and 0.0, should not be used directly as they do not correspond to the right probabilities. Instead 0.5 is substituted by the prior probability and 1.0 and 0.0 are substituted by two values `free_prob` and `occ_prob` given as arguments to `CameraSensorModel`. The values of `free_prob` and `occ_prob` should reflect probability that the information in `SENSOR READING` is correct. As the `Camera` and `Tracker` modules are quite exact good values seems to be; `free_prob`$= 0.05$ and `occ_prob` $= 0.95$. The performance of occupancy grid algorithm depends heavily on these values and they have to be adjusted according to the reliability of `SENSOR READING`. This will be further discussed in section 3.2.

2.1.6 `OccupancyGridMap`

The `OccupancyGridMap` take two inputs in the formats of `OCC PROB GRID` and `AFFECTED GRID REGION`. `OccupancyGridMap` also contains the state of the occupancy grid constructed so far; `MAP GRID`, and the prior probability; `pri_prob`, given as an argument. The `MAP GRID` is initialized by giving each cell the value of `pri_prob`.

The purpose of `OccupancyGridMap` is to update `MAP GRID` using the update equation of the occupancy grid map algorithm. This is done by applying this on all cells in `MAP GRID` that are inside the box defined by `AFFECTED GRID REGION`. Here follows the update equation taken directly from the code:

```
for(int i = affected_grid_region[2];
i <= affected_grid_region[3]; i++)
{
  for(int j = affected_grid_region[0];
  j <= affected_grid_region[1]; j++)
  {
```

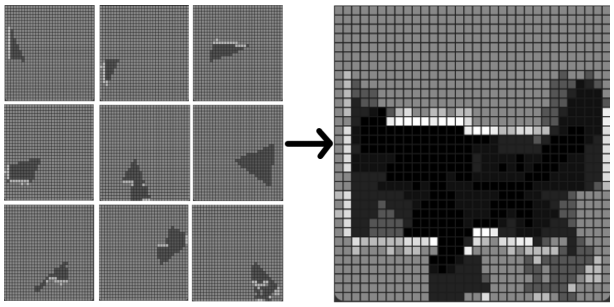


Figure 4: The image to the right shows the probabilities of a number of sensor readings and the image to the left shows the resulting occupancy grid map.

```
      float occ_prob = occ_prob_grid[i][j];
      map_grid[i][j] = 1.0 / (
        1.0 + (1.0 - occ_prob) / occ_prob *
        prior_prob / (1.0 - prior_prob) *
        (1.0 - map_grid[i][j]) / map_grid[i][j]);
    }
  }
```

An example of an how a `MAP GRID` could look is given in figure 4.

3 Evaluation

The implementation of the occupancy grid algorithm works very well on the robot setup. This is no big surprise as the conditions are ideal, there is practically no sensor noise nor pose uncertainty. In order to investigate how the implementation would handle different conditions a number of experiments were made, where noise was added to the sensor readings. How the implementation reacts to noise is highly dependent on the two parameters of `CameraSensorModel`; `free_prob` and `occ_prob`. Thus for each experiment, except for № 2, three different values of `free_prob` and `occ_prob` were used to illustrate this. The following values were used (using the notation [`free_prob`, `occ_prob`]): [0.01, 0.99], [0.2, 0.8] and [0.45, 0.55]. These values will be referred to as the *sensor weights,* as they reflect to what degree the occupancy grid map algorithm is persuaded by new sensor readings. All experiments used `pri_prob` = 0.5. The parameters `free_prob` and `occ_prob` might seem to be very specific for the `CameraSensorModel` but any sensor model will have parameters that governs to what degree the sensor readings should be trusted.

3.1 Experiment Setup

The experiments were setup in the following way: A robot was placed in the middle of the 2×2 m^2 "sandbox" and a number of obstacles were placed around it, the result is shown in figure 5 . The "camera" of `CameraSensor` was given a range of $\sqrt{2}$ m and a breadth of 32°. The robot does not move but each tick the heading of the robot is randomized, in this way the robot will eventually have "seen" the whole "sandbox" visible from the center. Four different experiments were then conducted:

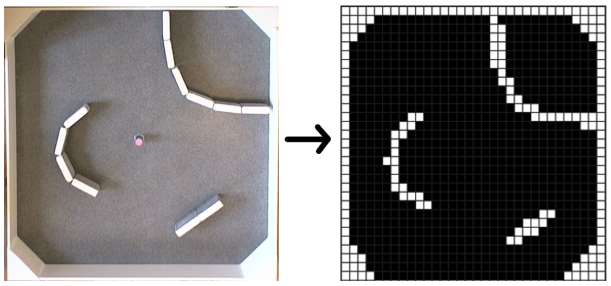


Figure 5: The experiment setup. The real world "sandbox" is to the left and the grid showing the extracted obstacles is to the right.

1. The ideal case. No noise was added, this is to get an measure to compare the other experiments with.

2. Gaussian white noise was added to the `OCC PROB GRID` of the `CameraSensorModel`. The noise had a variance of 0.1 and was applied to each cell `OPG`$[\mathtt{x}, \mathtt{y}]$ in the following way:

$$\mathtt{OPG}[\mathtt{x},\mathtt{y}] \quad = \quad \begin{cases} \mathtt{if\ OPG}[\mathtt{x},\mathtt{y}] < \mathtt{pri\_prob\ then} \\ \quad \mathtt{OPG}[\mathtt{x},\mathtt{y}] + \mathtt{abs(noise)} \\ \mathtt{if\ OPG}[\mathtt{x},\mathtt{y}] == \mathtt{pri\_prob\ then} \\ \quad \mathtt{pri\_prob} \\ \mathtt{if\ OPG}[\mathtt{x},\mathtt{y}] > \mathtt{pri\_prob\ then} \\ \quad \mathtt{OPG}[\mathtt{x},\mathtt{y}] - \mathtt{abs(noise)} \end{cases} .$$

   This experiment only uses `free_prob`=0.0 and `occ_prob`=1.0.

3. Salt and Pepper noise was added to 40 % of the `OCC PROB GRID` that represents the current sensor reading. That is, each cell that does no have the value `pri_prob` is given, by the toss of a coin, one of the values `free_prob` and `occ_prob` by a chance of 40%.

4. Gaussian white noise was added to he robot's position given as input to the `CameraSensorModel`. The noise had a variance of 0.001.

In order to compare the different experiment setups the comparison score measure described in Martin and Moravec [4] was used.

Let I be the ideal map over the same area as a constructed occupancy grid map m. I then only contains the values 1.0, 0.5 and 0.0, where 0.5 indicate that the value of the corresponding cell is unknown. The probability that a cell $m_{x,y}$ represents the same thing as $I_{x,y}$ is $I_{x,y}m_{x,y} + (1 - I_{x,y})(1 - m_{x,y})$. The probability that m represents the same as I is then:

$$\prod_{x,y} (I_{x,y}m_{x,y} + (1 - I_{x,y})(1 - m_{x,y}))$$

A problem is that this value will be very small for large maps. In order to remedy this the log_2 of this value is taken and $|I|$ is added. This results in the following score measure:

$$|I| + log_2 \left(\prod_{x,y} (I_{x,y}m_{x,y} + (1 - I_{x,y})(1 - m_{x,y})) \right)$$

The maximum score of m is $|I|$ minus the number of cells of I that are equal to 0.5. The ideal map was constructed by running experiment № 1 with `free_prob`=0.45 and `occ_prob`=0.55 for 2000 steps. The probabilities of this map was then rounded to the closest of the values 1.0, `pri_prob` and 0.0. Given this ideal map the possible maximum score is 640.

3.2 Results

Generally the implementation performed well in all four experiments but what became obvious is that the choice of sensor weights is important. Each experiment was run for a 1 000 ticks. As all of the experiments contain a randomized component a single run might not produce a characteristic result. To avoid this, each experiment was run ten times and the average of each tick was taken. The result of this is shown in figure 6. When interpretating these charts one should know that a score above 500 corresponds to a reasonably good map. Rather than looking for the sensor weights that eventually results in the best score one should look for the sensor weights that converge fast to a reasonable score. Most often a robot has more use for a good enough map now, that for a perfect map in five minutes. Because of this, the charts only display up to tick 500, even if the maps continue to converge after that.

Experiment № 1

This was the ideal case and as shown in figure 6a the algorithm performs well for both [0.01, 0.99] and [0.2, 0.8]. Even if [0.45, 0.55] surpasses them both eventually, it converges to slow to be practically useful.

Experiment № 2

The outcome of this experiment, as shown in figure 6b, show the strength of the probabilistic approach to robotic mapping. The algorithm handles the noisy sensor readings well and the map converges nearly as fast as [0.2, 0.8] from № 1.

Experiment № 3

Figure 6c show how to high or to low set sensor weight impacts the performance of the algorithm. While [0.2, 0.8] converges nicely, [0.45, 0.55] converges steady but too slow. As [0.01, 0.99] is the most sensible to noise, it converges slowly and never produces a reliable map.

Experiment № 4

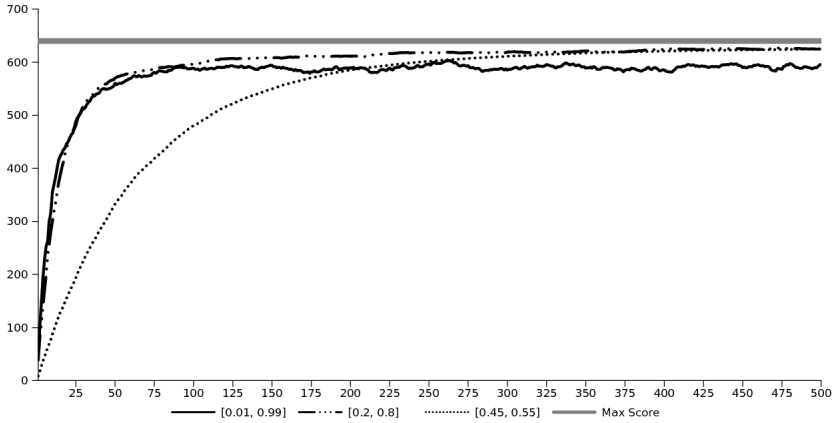
In this last experiment the score measure is a bit misleading. All three choices of sensor weights actually produces acceptable maps. What happens in the case of [0.01, 0.99] is that the edges of the obstacles get slightly displaced, which the score measure penalizes. Even though [0.01, 0.99] of № 3 and № 4 score the same, the map from № 3 is practically unusable, while the map from № 4 is OK.

3.3 Using the Implementation

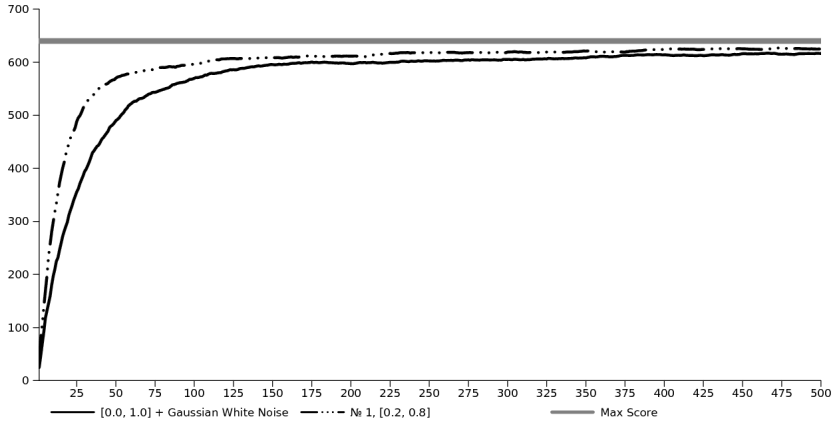
To show that the map drawing implementation can be used in practice an Ikaros system was setup to control an e-puck robot. Basically, this is the system shown in figure 3, including the dashed lines. The goal of the e-puck was to find another e-puck wandering randomly in a maze. The e-puck was not given a path to the other e-puck, only its position. In order to find a path to the other e-puck a wavefront algorithm as described in [5] was used. The e-puck would begin with an empty map, which it would build up gradually as it tried different paths to the other e-puck. Eventually, the map would be complete enough so that the e-puck would find a safe path to the other e-puck.

4 Discussion

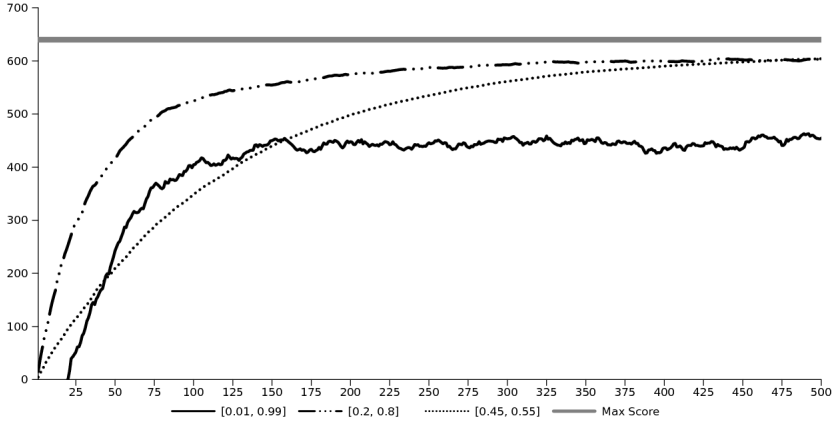
This paper has described an implementation of the occupancy grid map algorithm. This algorithm was implemented to be used with the e-puck robot, using the Ikaros framework. A derivation of the update equation, the basis of the algorithm, was given, as well as a measure for comparing maps. The implementation worked well. This was no surprise as the sensors and the pose tracking system produced very exact information. To investigate how noise would affect the performance of the algorithm a number of



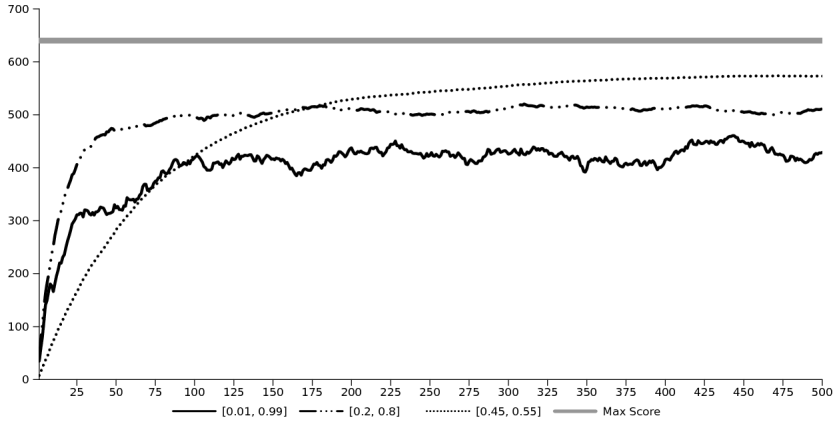
(a) Experiment № 1, the ideal case.



(b) Experiment № 2, gaussian white noise.



(c) Experiment № 3, salt and pepper noise.



(d) Experiment № 4, positional gaussian white noise.

Figure 6: Graphs showing how the estimated map converges to the ideal map given different noise and different sensor weights.

experiments were conducted. Gaussian white noise was applied to the sensors and the pose tracking system, and so called salt and pepper noise was applied to the sensors only. To show that the implementation was usable in practice a system was constructed that made an e-puck draw an occupancy grid map. The e-puck then used this map to find a path to another e-puck wandering randomly.

4.1 Evaluation of the Experiments

Experiment № 1 show that the algorithm works well given ideal preconditions. This is no surprise, but it is important note how the tuning of sensor weights impacts the performance. When the sensor weights are set so that the algorithm put little trust in the sensors, the map converges steadily but unnecessarily slow.

Experiment № 2 and 3 show the strength of the algorithm, its capability to handle independent noise. Both the sensor readings of № 2 and 3 are very noisy, indeed it is often hard for the human eye to separate true obstacles from noise. The algorithm manages this well, given that the sensor weights are set so that the algorithm does not put to much trust in the sensors.

Experiment № 4 show that the algorithm can produce an acceptable map when the position is noisy. The tuning of the sensor weights does not have such an impact as figure 6d might suggest. This is due to the fact that the score measure does not reward correctly identified obstacles that are off by a small distance. One problem with positional noise is that it does not lead to sensor noise that is statistically independent. If the positional noise is to large the algorithm will not be able to handle it no matter how the sensor weights are tuned.

The implementation of the e-puck control system described in section 3.3 worked well in simulation. The two robots steadily moved towards each other, drawing the map and avoiding obstacles as they went along. When trying this with the real robots there were some problems. The `Tracker` module sometimes confused one of the robots for the other one. Also there were some problems communicating with two robots over one Bluetooth connection. Nevertheless, the occupancy grid map algorithm, in combination with the wavefront path planner, always produced a correct path, even if the robot had troubles following it.

4.2 Conclusion

In spite of its limitation the occupancy grid map algorithm is, as this paper has shown, a robust and versatile algorithm. When in need for a robotic mapping algorithm one should have good reasons not to consider using it.

References

[1] C. Balkenius, J. Morén, B. Johansson, and M. Johnsson. Ikaros: Building cognitive models for robots. *Advanced Engineering Informatics*, 24 (1):40–48, 2009.

[2] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, June 1989. ISSN 0018-9162.

[3] A. Elfes and H. Moravec. High resolution maps fron wide angle sonar. *IEEE International conference on Robotics and Automation*, 1985.

[4] Martin C. Martin and Hans Moravec. Robot evidence grids. Technical Report CMU-RI-TR-96-06, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 1996.

[5] S. Russell and P. Norvig. *Artificial Intelligence - a Modern Approach, 2nd edition.* Prentice Hall, 2002.

[6] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Henning, T. Hofmann, M. Krell, and T. Schmidt. Map learning and high-speed navigation in RHINO. In D. Kortenkamp, R.P. Bonasso, and R Murphy, editors, *AI-based Mobile Robots: Case Studies of Successful Robot Systems.* MIT Press, 1998.

[7] S. Thrun, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A second generation mobile tour-guide robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1999.