Cognitive Robotics
Seminar report

Matúš Harvan

Spring Semester 2006

Abstract

Cognitive robotics is concerned with enabling robots to perform higher level cognitive functions sucha as perception, reasoning and action in complex, unknown and changing environments. The goal of the RoboCup Rescue robots in the Robotics lab is one day to be used in search and rescue operations after disasters, i.e. after a building collapses the robots should be able to explore the collapsed building and find victims inside. In particular, robots should be capable of exploring and mapping an unknown environment, i.e. a crash site and look for victims. Several subtasks needed for such autonomous runs of these robots have been investigated and dealt with. A basic occupancy grid has been implemented as a starting point for mapping an unknown environment surrounding the robot. The usage of a gyro for odometry has been investigated, where double integration of the acceleration (as measured by the gyro) would be used to determine displacement from starting point. Finally, a laser scanner has been used to detect the largest opening among obstacles surrounding the robot. This would aid the robot in an autonomous exploration of unknown environments. The capability of the scanner to detect obstacles of various materials has been investigated while evaluating the implementation.

1 Introduction

Cognitive robotics is concerned with endowing robots with higher level cognitive functions, enabling them to perceive, reason and act in complex, unknown and changing environments, understanding the high level features of an environment.

The particular application scenario of cognitive robotics will be the RoboCup Rescue robots in the Robotics Lab at International University Bremen. The goal of the robots is to be used in search and rescue operations after disasters, i.e. after a building collapses the robots should be able to explore it and find victims inside. In particular, robots should be capable of exploring and mapping an unknown environment, i.e. a crash site. Within this crash site robots have to locate (human) victims, determine their health status and report this information including the victim position within the environment. Ideally, the robots should operate autonomously, without the need for a human operator. Such autonomy includes several several areas, such as exploring an unknown (and maybe changing) environment, creating a map of the explored environment, localizing the robot itself within the map, detecting and avoiding obstacles in the environment and detecting victims and checking their status. Within the scope of cognitive robotics, focus will be on higher level tasks such as mapping the environment and localing the robot itself within the environment rather than low level sensor operation or motor activation.

Specific subtasks of the autonomous exploration and mapping of the environment have been been examined and dealt with in more detail. For simplicity the environment was assumed to be only 2-dimensional. Usage of an occupancy grid has been considered as a way of creating a map of the environment. Several options to produce an occupancy grid have been considered and a basic algorithm has been implemented. The advantage of occupancy grids is that they can take into account uncertainties in the scanner data and a changing environment. The occupancy grids are discussed in more detail in section 2. So far, shaft encoders on robot wheels have been used for odometry. These, however are inaccurate due to slip of the wheels on the surface. Given that the robot turns by making the wheels slip, this method get especially inaccurate for turns. Therefore, the usage of a gyro mounted on the robot has been considered. Double integrated acceleration values measured by the gyro could be combined with the information from shaft encoders on the wheels to get more accurate odometry. The feasibility of this approach approach and results from experiments are described in more detail in section 3. Using information from a laser scanner to look for "holes" between obstacles around the robot as a basic way of autonomous environment exploration has been investigated and implemented. Details can be found in section 4.

2 Occupancy Grid

Occupancy grids are used to express which parts of the world around a robot (grid cells) are free and which are occupied. For that probabilities are used instead of a binary yes or no. Unknown cells are marked with values between free and occupied (usually exactly in

the middle between free and occupied) [7].

The occupancy grid is updated using information from sensors. The simplest way is to overwrite the old value of the corresponding cell with new reading from the sensor. However, better results can be obtained by using the Bayes Theorem and taking into account also previous value of the cell and sensor models when updating the occupancy grid [10]. To use Bayes Theorem, the sensor measurements are assumed to be independent. This approach copes better with sensor inaccuracies. Furthermore, readings from several sensors can be fused together to increase accuracy of the occupancy grid. [9] describes how data from sonar and stereo have been integrated. Occupancy grids can also be used for collision avoidance [4, 5]. A review of certainty grids usage is presented in [12]. An overview of the mapping methods and certainty grids as used so far by the Robotics Group at International University Bremen can be found in [3].

In general, sensor usage can roughly be divided into:

1. **localization** – to determine the current pose of the robot within the map (occupancy grid)

2. **obstacle detection** – to detect obstacles in the environment

Due to inaccuracies (e.g. from odometry) it is hard to determine the exact pose of the robot. Instead of struggling for exact values probabilistic localization can be used. In the occupancy grid framework, this can be accounted for by "blurring" older values in the grid. The idea is that current pose of the robot is known (in the robot-centric frame) and hence recently recorded values near the robot are accurate. However, previous positions of the robot are becoming less accurate (due to accumulated errors) as the robot moves and hence older values in the grid are being blurred (converging to values representing unknown grids).

As the sensor for creating an occupancy grid a laser scanner (`HOKUYO URG-04LX`) mounted on the robot has been used. The (2-dimensional) grid was based on a 2-dimensional array. There are two methods for updating the grid:

1. **sample along beam lines**
   Updates to grid cells are done by sampling along each beam of the scanner as shown in Figure 1. Cells covered by the beam are marked free. The cell corresponding to the end of the beam is marked occupied (that's where the beam was reflected by an obstacle). Values of cells after the end of the beam are not changed (as it is not known what is behind the obstacle). A disadvantage of this approach is that if the grid has finer resolution, only thin lines within the grid (corresponding to the beams) are updated. This could be mitigated by assuming cones instead of lines for the updates (see Figure 2).

2. **consider area around robot**
   Instead of sampling along beam lines, one could consider only a bounding box on the beams and construct a bounding polygon of the beams (i.e. end vertexes of the

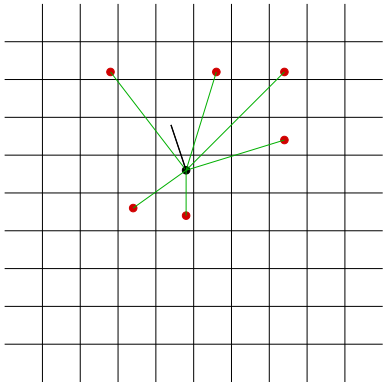


Figure 1: Sample along beam lines.

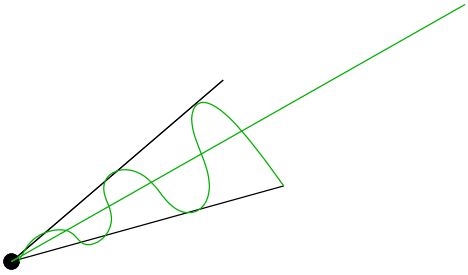


Figure 2: Sample along beam lines using cones.

beams are vertexes of the polygon). Then cells inside the polygon are considered free, cells on the sides of the polygon are considered occupied and cells outside the polygon are not updated. This is illustrated in Figure 3. In order to check whether a cell is inside the bounding polygon, it is sufficient to find points from closest scans (with y-coordinate) above and below given cell and then check if the cell is inside, outside or on the boundary of the polygon (on the line between the two scans).

For the implementation part the choice of data structures has been settled. The grid would be represented as a 2-dimensional array of unsigned 16-bit integers. The grid itself would be contained in a class `Map`. There would also be a class `Robot`, containing a `Map` and a class `Pose` to determine robot's current pose. `Map` would have a function for mapping real-world coordinates to grid cell indexes. A UML diagram of the classes are shown in Figure 4.

For updating the grid with new readings from the scanner, initially a robot-centric local grid considering the area around robot was to be used. In a second step the local grid would be merged into the global grid. This approach, however, has been abandoned due to complications with using a local robot-centric grid for updates from scanner. Furthermore, there would be inaccuracies merging the local grid with the global one. The decision was made to do the updates directly in the global grid. The updating has been implemented in one function of class `Robot`, taking readings from the scanner as an argument.

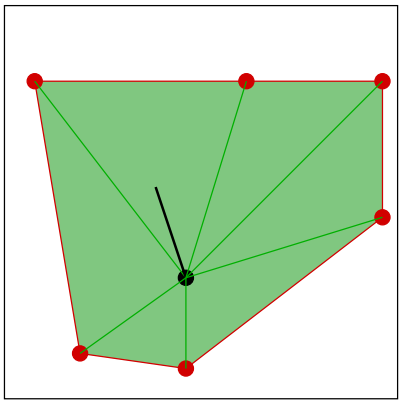


Figure 3: Sample whole area.

The updates of the grid with new readings from the scanner have been done by simply overwriting previous grid cell values with new readings from the scanner. The update algorithm could easily be changed later to use the Bayes Theorem if needed.

The implementation has been finished. It has been tested by simulating scanner values of an obstacle all around the scanner in the range of 5 cm. The robot was moved through the grid and the grid values have been dumped after each robot move by one cell. The field of view and detected obstacles around the robot matched what was expected. Another check was done by rotating the robot in one place and changing to a narrower field of view of the scanner. This checked if the orientation and angle calculations were done correctly. The purpose of the occupancy grid part was just to get a basic implementation working. Afterwards, focus has been shifted to other problems and no more work has been done on the occupancy grids.

3 Gyro

As one of the ways to do localization, the robot is using dead reckoning. For this shaft encoders on robot wheels are used. These, however, produce inaccurate results in case wheels of the robot slip. Given that the robot turns by rotating wheels one side at another speed than wheels on the other side, thereby making the wheels slip (skid steering), this method gets inaccurate rather fast. It is relatively accurate for driving straight, but the turns are a problem. To mitigate this, the usage of a gyro mounted on the robot has been considered. The gyro was an `Xsense GyroMT-I`, a 3 DOF gyro measuring orientation and acceleration. By merging information about orientation from the gyro with information from the encoders, accuracy of dead reckoning could be improved [1, 8, 11, 6]. This affects errors in heading, which in turn influence the position accuracy. For merging the information from various sensors (gyro and encoders), various forms of Kalman filters could be used (Indirect, Extended, Unscented,...).

Furthermore, double integrating acceleration values from the gyro could be used for

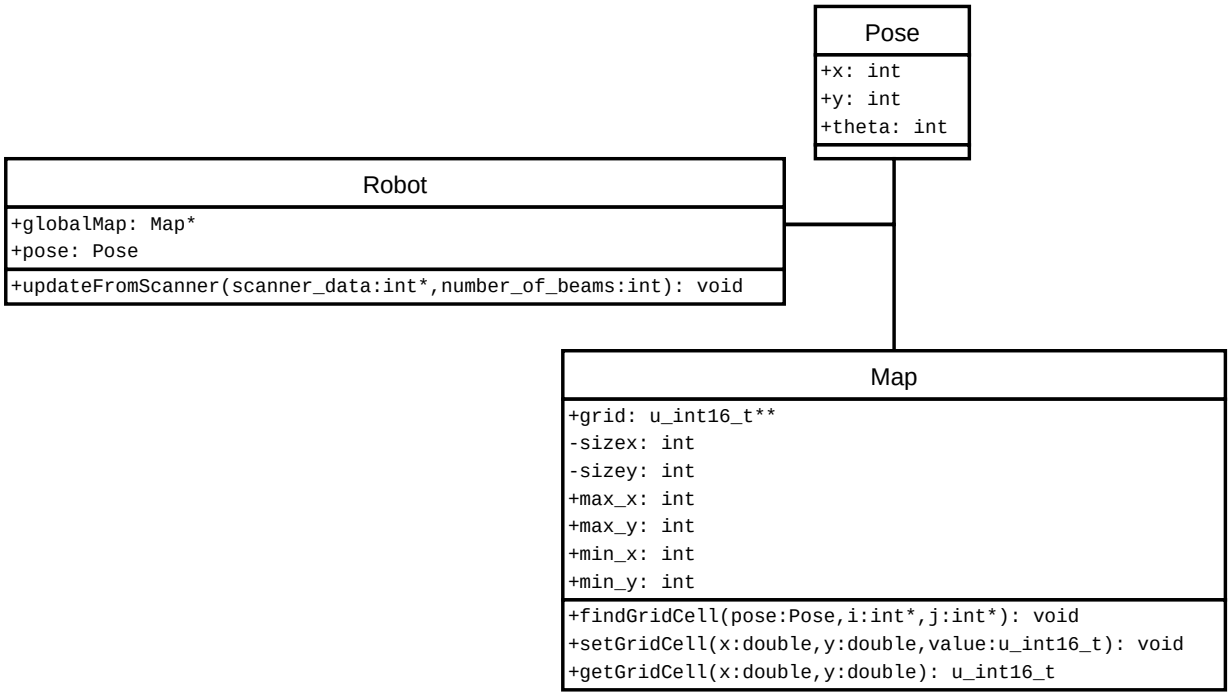


Figure 4: Classes used for implementation.

odometry. However, using double integration has the disadvantage of quickly accumulating errors. Furthermore, acceleration measurements are subject to fluctuations as the robot moves over uneven ground. A successful usage of this approach, called inertial navigation system, is described in [2]. It achieved a position drift rate of 1-8 cm/s. The conclusion was that position estimates using accelerometers were reliable only over short periods (5-10 s) and not better than using encoders on robot wheels.

The first problem turned out to be actually getting any data out of the gyro, so the first goal was to start reading data from the gyro. To start with, an older program written by Andreas Pfeil was found on the robot. However, this program was written for the driver infrastructure and hence could not be used directly. An attempt was made to delete all code which seemed player-related or unnecessary, but then the program was no longer getting any updates for the gyro data (i.e. it was reporting the same data all the time). The most probable reason for this was that the original code was also using threads for updates from the scanner and the code for starting the update thread has been removed.

Next attempt was made with sample code for the gyro downloaded from the robotics wiki[1]. This code worked without any problems and was showing showing readings for both acceleration and orientation.

The next goal was to try using the gyro for odometry. This would probably be inaccurate, but still useful for short time and/or coupled with other sensors. Initially, experiments

[1] `http://robotics.iu-bremen.de/docu/protected/XSENS/GenericDemoCode/SoftwareDevelopment/Xbusclass/Example-Linux/`

have been done with odometry only along x-axis to see whether using the gyro for odometry would be a viable approach. Using double integration to do odometry did not work well. The values for acceleration were constantly non-zero along x- and y-axis. This was probably because the gyro was not in level with water and hence part of the earth gravity was decomposed also into components along x- and y-axes, not only the z-axis. It was considered to mitigate this problem by calculating the component of earth gravity along x-axis from the pitch (as the gyro also reported the pitch). However, no agreement could be reached with fellow students about how to do the calculations. As a result, the problem was left to be tackled at a later point in time. Another alternative would have been to use a value from the matrix representation of orientation, where one would basically take the component of x-axis and see how it maps into the global z-axis (or the other way round) to determine the relevant component of Earth gravity onto the gyro's x-axis. The source code implementing ideas discussed so far can be found in `src/mt-gyro-example.1.tar.gz`.

Attempts to use the gyro for odometry have been continued, initially only along the x-axis. An agreement on determining the gravity component along x-axis was reached with fellow students. The gravity x-axis component would be obtained by multiplying g with the top-right value from the matrix representation of the gyro pose (the sample program for the gyro provided this). It was agreed that the value should be either top-right or bottom-left and the correct one was found experimentally by making the gyro face north and then rotating it.

The sample program has been modified to also do double integration of acceleration and remove the g-component from acceleration along x-axis. For translating the acceleration into odometry, the following formulas have been used

$$\begin{aligned}\ddot{x} &= a \\ \dot{x} &= a\Delta t + v_0 \\ x &= \frac{a\Delta t^2}{2} + v_0\Delta t + x_0\end{aligned}$$

giving

$$\begin{aligned}x &= x_0 + v_0\Delta t + \frac{a\Delta t^2}{2} \\ v &= v_0 + a\Delta t\end{aligned}$$

So at every update step the following was done

$$\begin{aligned}x &\leftarrow x + v\Delta t + \frac{1}{2}a\Delta t^2 \\ v &\leftarrow v + a\Delta t\end{aligned}$$

Readings of acceleration a were decreased by what was believed to be the component of g in the direction of x-axis. This component, however, was constantly changing (oscillating), so probably there were some non-negligible inaccuracies or instabilities within the gyro itself.

Afterwards, various techniques have been tried out to get some useful odometry data from the gyro. A description of these attempts follows. It should be noted that none of them was particularly successful. The testing was done by moving the gyro by hand for 20 cm and checking what distance the program (implementing the currently tested technique) has reported. These results were usually significantly off. As the acceleration was non-zero, the speed was constantly increasing or decreasing. After a short while the error has accumulated a large speed and there was no chance of getting any reasonable odometry as the relatively large speed was by no means affected by a small acceleration.

The values for the gyro at rest were dumped into a file for several hundreds of readings. Matlab was used to find the mean ($5.18 \cdot 10^{-5}$) and standard deviation (0.0080) of the gravity component along the x-axis.

To correctly determine the gravity component along x-axis the exact value of g was also needed, or at least the value that the sensor believed to be g. Hence, also the values of ($\sqrt{x^2 + y^2 + z^2}$) have been dumped, which should correspond to g when the gyro is at rest. The mean of this value was around 9.83. However, repeating this procedure revealed that the mean was changing for each run, i.e. for one run it was 9.8319, for next run it was 9.830091. To cope with the changing values a running average of g was used to cancel the component in x-axis direction. This, however, would be very accurate as the formula we have used would add also other acceleration to g such as starting to move forward. Therefore, this solution would only be useful if the robot were not moving – this is not terribly useful if the purpose is to determine how much the robot has moved.

Nevertheless, running average of the measured g values for the last 10 measurements was tried out, but it has not allowed for any reasonable odometry.

The next approach was to try clipping the measured x-axis acceleration to get rid of small values (noise) with the idea that real robot movement acceleration gives higher values than the values at which clipping was done. This did not work well either, as with the clipping parts of the real acceleration were eliminated as well. This lead to some funny results, such as moving by 10 cm, i.e. accelerating, moving and decelerating to stop lead to a negative speed after stopping as parts of the "real" acceleration have been clipped away.

The source code implementing these attempts can be found in `src/mt-gyro-example.2`. The conclusion from the experiments is that it is not feasible to use double integration of acceleration (as measured by the gyro) to do odometry. The problem is that errors accumulate too fast to get any reasonable results, even for short time intervals like several seconds.

4 Largest Opening (Gaps From Laser Scanner)

A basic form of autonomous environment exploration can be achieved by a strategy of following the largest opening. Using information from a laser scanner one can look for openings (or gaps) between obstacles around the robot. Then the robot can be directed towards the largest identified opening. In this way a very basic autonomous environment exploration could be implemented. Within the framework of the robot autonomy, this

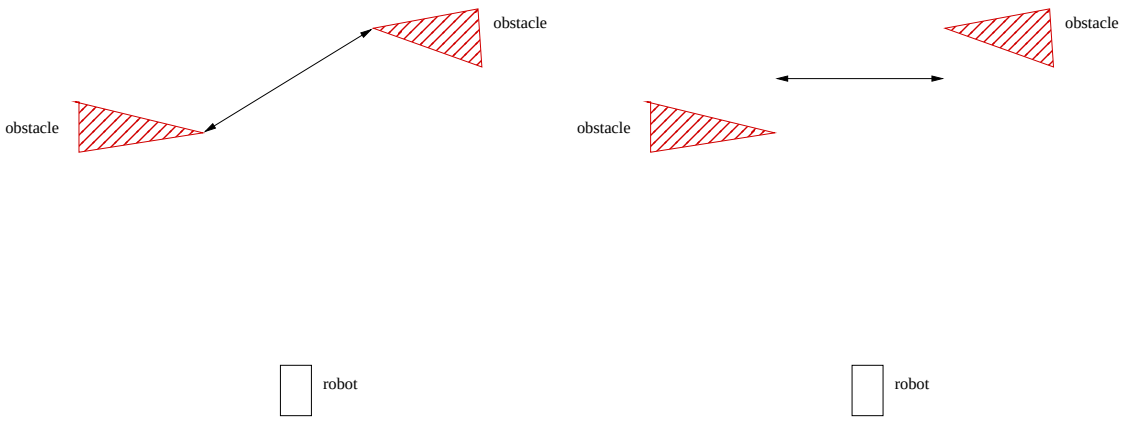
approach would be used in case the other autonomy approaches get "stuck". Then rotating the robot is expected to "unstuck" the other autonomy functions so that they could again take over. The task of the algorithm was only to report direction to the largest opening, but the task of actually navigating the robot to move would be taken over by other algorithms. From the discussion with the PhD student in the lab assigning this task, the understanding was that openings are identified by error beams of the laser scanner, i.e. beams which are emitted by the scanner but do not come back to the scanner in time. The understanding was that these beams represent free space within the range of the scanner (4 m). Therefore, a gap was defined as a sequence of consecutive error beams. Clearly, it only makes sense to consider gaps with sufficient width for the robot to pass through the gap. Already during the initial discussion it became clear that there were several approaches to solve the problem and probably several of them would have to be examined in more detail. During evaluation of the implemented solution, the capability of the laser scanner to detect obstacles composed of various materials has been investigated.

In order to pick the largest opening, a measure of the gap width is needed. Two possible measures have been considered.

1. distance between the obstacles forming the gap. Let l_1, l_2 be lengths of the beams to the obstacles bounding the gap and ϕ be the angle between the beams. Then the gap width is $\sqrt{l_1^2 + l_2^2 - 2l_1 l_2 \cos(\phi)}$.

2. perpendicular projection of the above measure. Let l_1, l_2 be lengths of the beams to the obstacles bounding the gap and ϕ be the angle between the beams. Then the perpendicular projection of the gap width is $2\min(l_1, l_2)sin(\frac{\phi}{2})$. This measure is roughly proportional to the number of laser beams covering the gap.

The two measures are graphically illustrated in figure 5.



(a) Measure 1 – distance between obstacles

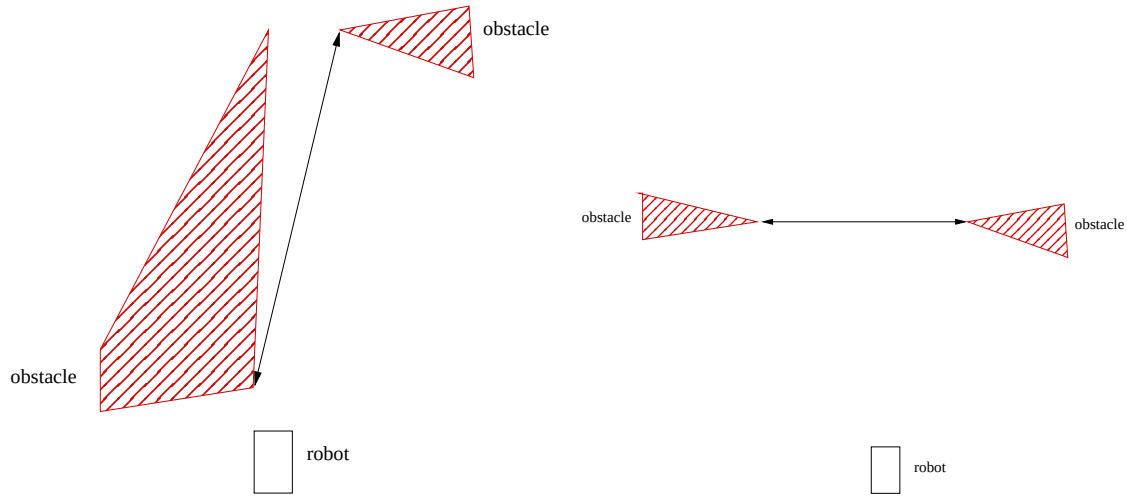
(b) Measure 2 – perpendicular projection of Measure 1

Figure 5: Two different measures for gap width

The implementation work has started with the scanner GUI program developed by Ivan Delchev (`src/scanner_new.tar.gz`). This program was capable of accessing the scanner,

reading data from it and displaying the beams in a semi-circle. Furthermore, error beams were shown with a different color than the normally reflected beams. This program has been used as a starting point for implementing the algorithm to locate the widest opening. The largest gap was chosen using measure 1. If found, the beams corresponding to it were marked with green color in the GUI. This allowed for an easy way to visually inspect how the algorithm was performing. A gap has been considered a sequence of consecutive error beams. Beams were processed sequentially from right to left. The first error beam in a sequence was opening a gap and the last one was closing it. When closing a gap, it's width was calculated and if it was larger than current maximum, information about the widest gap was updated. In this way the selection of the widest gap was done in time linear in the number of beams. Source code of the implementation can be found in `src/scanner\_new\_matus\_alex.tar.gz`.

After some experiments it has been noticed that measure 1 might not be as suitable as has been initially expected. The problem was that if there was a gap with a small perpendicular projection width (measure 2), but the boundary obstacles were far apart, i.e. one obstacle close to the robot and the other one far away, measure 1 was identifying this gap as the largest one. This went to extreme cases such as a gap of just one beam. However, there were several other gaps, which had the perpendicular projection of the width larger, i.e. were covered by more laser beams. An illustration of such a case can be found in Figure 6. In this figure (b) would seem as a better choice, but measure 1 clearly favors (a). Due to this the algorithm was changed to use measure 2. Initially only the



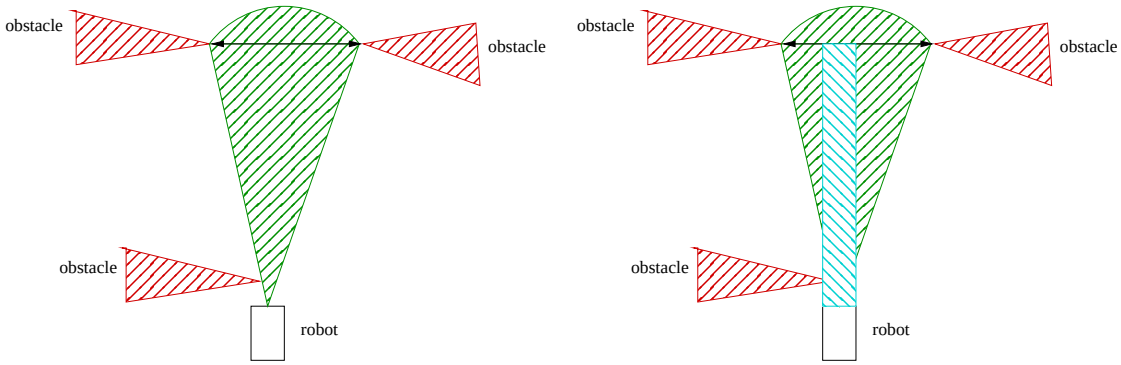
(a) bad (b) good

Figure 6: Two gaps to consider for measure 1 – (b) seems to be a larger gap, but (a) is found to be "wider"

number of beams was used, later on the proper formula was plugged in.

Another problem found during the implementation phase was that the laser beams forming a gap identify a cone of free space. However, it is desired to check whether the

robot could move in the direction of the gap. Therefore, a rectangular area has to be checked, part of which (close to the robot) is not covered by the free space cone. If only the cone of free space were checked, an obstacle could possibly exist close to the robot outside of this conical region. Hence, the rectangular region corresponding to the path of the robot going towards the gap in question should be checked. The argument is illustrated in figure 7. The first approach suggested was to assume that there would be no obstacle



(a) cone (b) rectangular

Figure 7: As shown in (a) it is possible to have an obstacle close to the robot in a way that the obstacle does not interfere with the conical free space region. Hence, the rectangular region corresponding to the planned robot path should be checked as well.

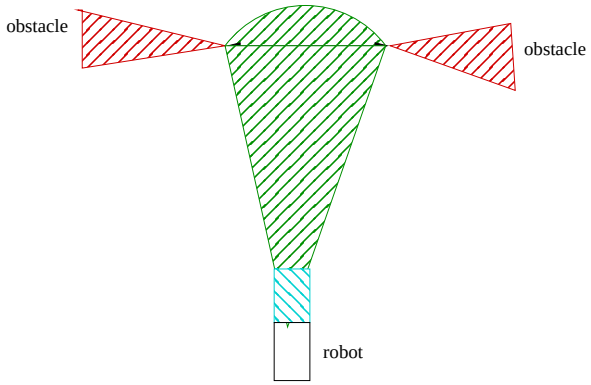


Figure 8: Assuming the cyan part is free, we no longer have to check the rectangular area outside the cone.

as close to the robot as to be outside of the cone and still prevent the robot from moving towards the gap, i.e. assume there are no obstacles in the cyan rectangle in Figure 8. The reason is that the motion planning and other autonomy functionality would not let the robot get so close to an obstacle. The largest opening algorithm only reports an angle to rotate the robot to face the largest opening, but the navigation towards the opening is then taken over by other parts of the robot software. Thess other parts would then make sure

not to hit any obstacles. Furthermore, the robot has also other sensors which should detect such close obstacles. This approach, however, was soon abandoned because choosing an opening as the largest one and then finding out the robot could not move in its direction does not get the robot any further in exploring the environment.

The next approach was to properly check if there are no obstacles also in the rectangular region near the robot. This checking will be referred to as corridor width checking. To

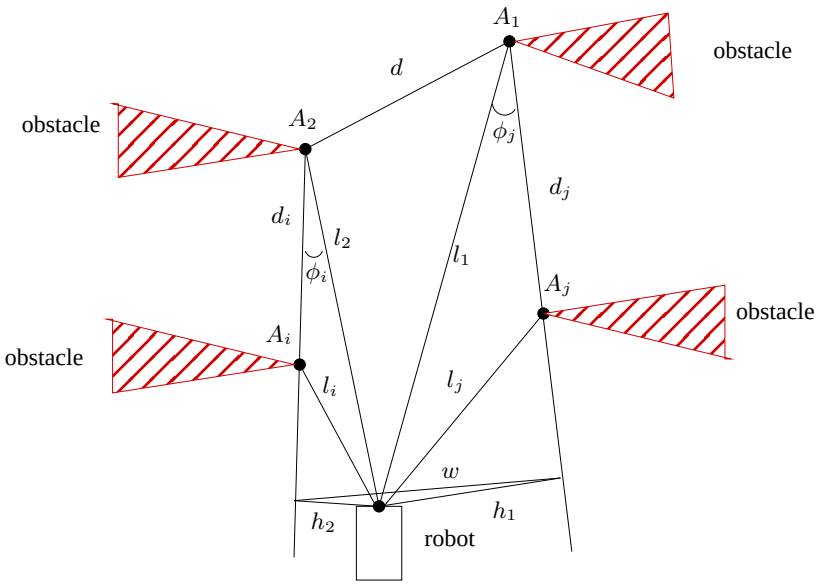


Figure 9: Corridor width calculation.

determine the corridor width w we need to find obstacles limiting it "the most". To do this following steps are performed (an overview of the variables can be found in Figure 9). Let A_1 be the obstacle limiting the gap from the right. Let the beam going to point A_1 that limits the gap have an angle α_1 and length l_1. For every beam to the right of this beam, going to obstacle A_j, having angle $\alpha_j \in [0, \alpha_1)$ and length l_j, we can determine distance d between obstacles A_1, A_j and angle ϕ_j as follows:

$$d_j = \sqrt{l_1^2 + l_2^2 - 2l_1 l_2 cos(\alpha_1 - \alpha_j)}$$

$$\phi_j = arcsin\left(\frac{l_j}{d_j} sin(\alpha_1 - \alpha_j)\right)$$

The beam corresponding to the obstacle limiting the corridor "the most" will be the one having minimum angle with the beam going to A_1. This will be $\phi_1 = \min_j(\phi_j)$. The right part of the corridor can be determined as

$$h_1 = l_1 sin\phi_1$$

Similar algorithm can be used to find the beam limiting the corridor on the left side. Let A_2 be the left bound of the gap, corresponding to the beam with angle α_2. Consider

all the beams to the left of this beam. These beams would have angle $\alpha_i \in (\alpha_2, \alpha_{\max}]$ and length l_i. Similar formulas as for the previous case can be used to determine ϕ_i. To find the beam corresponding to the obstacle limiting the corridor on the left, we need to find the beam with minimum ϕ_i. This would be $\phi_2 = \min_i(\phi_i)$ and $h2 = l_2 sin\phi_2$.

The corridor widht can the be determined from h_1 and h_2

$$w = \sqrt{h_1^2 + h_2^2 - 2h_1h_2cos\beta}$$

with $\beta = \pi + \alpha_2 - \alpha_1 - \phi_1 - \phi_2$. If the corridor width is less than the width of the robot, then the gap in question would not be considered as a candidate for the widest gap.

The above described algorithm has been implemented in the current widest gap detection. Using a `#define`, it can be enabled or disabled at compile time. Once a gap is identified (i.e. the closing beam is found), the corridor width is checked. Compared to the previous version, the additional corridor width checking implies looping over possibly all the beams for each gap. This worsens the run time to be worst case quadratic in the number of beams. However, usually there are not that many gaps and the number of beams from the laser scanner is constant. While evaluating the implementation, the run time did not seem to be in any way a limiting factor.

The implementation has been tested by moving cardboard boxes around the scanner and visually inspecting the GUI output of the program. The identified largest opening was marked green among the beams, so the the visual inspection was very convenient. During the testing, several shortcomings have been found and fixed. One particular problem which became obvious during testing was that only beams with angle $-\pi/2 \leq \alpha \leq \pi/2$ are compatible with the formulas. In other words, it does not make sense to check obstacles behind the robot for corridor width calculations. Furthermore, if a beam is longer than l_1 or l_2, it also does not make sense to consider it. Another bug was that the variable for storing maximum gap width was not initialized in the beginning, causing erroneous results like not identifying something which clearly would have been considered the widest gap after a visual inspection. At some point of time the scanner device could no longer be accessed. After some debugging the problem turned out to be that the device node representing the scanner (`/dev/ttyACM0`) changed from a devicce node to a regular file. Fortunately, there were similar device nodes still left intact (i.e. `/dev/ttyACM1`) to determine the type, major and minor numbers. The file was deleted and a proper device node was created by hand using the commands

```
cd /dev
rm ttyACM0
mknod ttyACM0 c 166 0
```

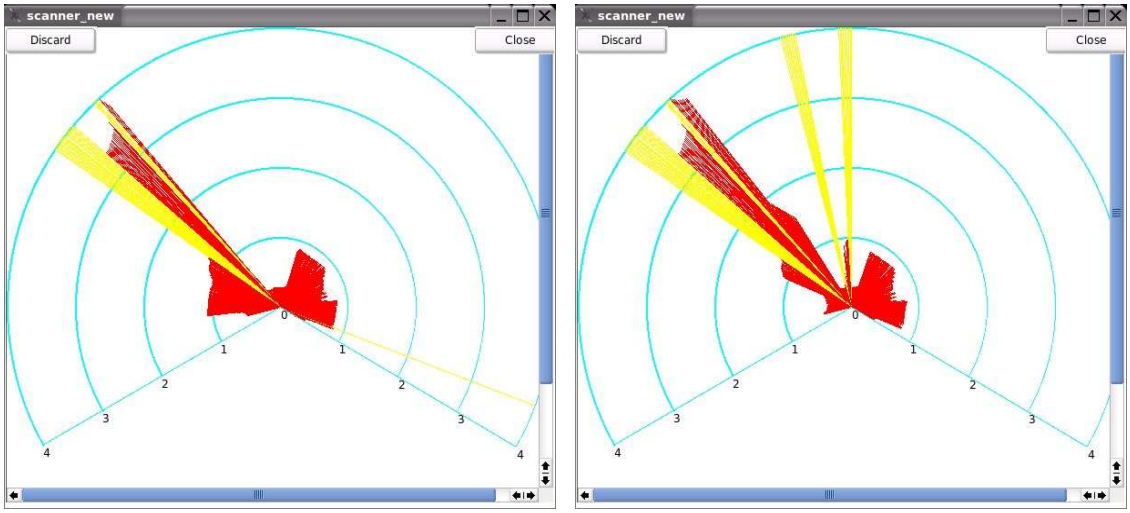
Afterwards, the scanner worked again. The problem with the device node was probably caused by a new installation or misconfiguration of `udev`, a framework for virtual `/dev` file system on Linux systems.

There was already another implementation of a widest opening selection algorithm on the robot by Ivan Delchev and Stefan Markov. A comparison with this implementation was

considered. However, due to differences in the algorithm, a discussion with the authors was initiated. Their algorithm completely ignores error beams (beams going to infinity) and divides the remaining beams to clusters where beams in a cluster have difference in length only within a certain margin. This leads to clusters roughly corresponding to obstacles detected by the scanner. The clusters are sorted comparing the sum of lengths of their beams. This approximates the free space area between the robot and the obstacle. The cluster with the largest area is then chosen as the widest opening. During the discussion the real meaning of error beams (beams going to infinity) was discussed. It became clear that they do not necessarily represent free space up to maximum scanner range. They are simply beams which have not been reflected back. As many materials absorb the beams, there could be obstacles not detected by the scanner. Therefore, it should be assumed to have no information about the area covered by error beams. This contradicts the basic assumption behind the design of our algorithm. Hence the algorithm could probably not be used on the robot. On the other hand, completely ignoring error beams would fail in certain scenarios, e.g. a simple room with an open door and no other way out. As there would be nothing to reflect the beams in the empty door frame, the robot would not detect the open door as an opening. It was considered to completely abandon the algorithm and work on another task. However, the prof said the algorithm should be tested with more materials and determine how feasible it is to use error beams.

To determine how well the laser scanner could detect various materials, several tests have been conducted by using obstacles from various materials and checking how they would be detected by the scanner. It turned out that shiny, reflexive materials and metals tend to produce a large amount of error beams. A paper notebook was detected without any problems. However, the effect of a reflexive surface could well be observed by holding a CD in front of the paper notebook. The CD not only produced a huge amount of error beams, but it also has distorted the image of the obstacle on the scanner. The CD was aligned with the paper notebook, but on the scanner it seemed as if the CD were 0.5m in front of the notebook. Screenshots from the scanner program are shown Figure 10. The red beams are detected obstacles while yellow beams are error beams. House keys (produced from metal) seem to produce error beams as well, even when put in a protective leather case. The heaters in the lab seemed to cause error beams as well. A black plastic box with rather shiny surface has been found to cause a number of error beams as well. For the shiny surfaces the distortion of beams (causing beams to show different distance) or production of error beams seemed to depend on the angle of the surface to the laser beams. Human body seemed to be detected by the scanner without problems. From the experiments it seems some materials cannot be well detected with the laser scanner and hence the feasibility of largest opening detection based on error beams is questionable.

The tests gave an incentive to redesign the algorithm developed so far. The definition of a gap has been changed from the sequence of error beams to a sequence of beams with length within a certain range. The algorithm was changed to look for gaps in iterations. The limit for considering a beam as belonging to a gap would dynamically change from iteration to iteration. The algorithm starts with a rather high limit and tries to find a gap wide enough for the robot to pass through it. If no such gap is found, the limit for

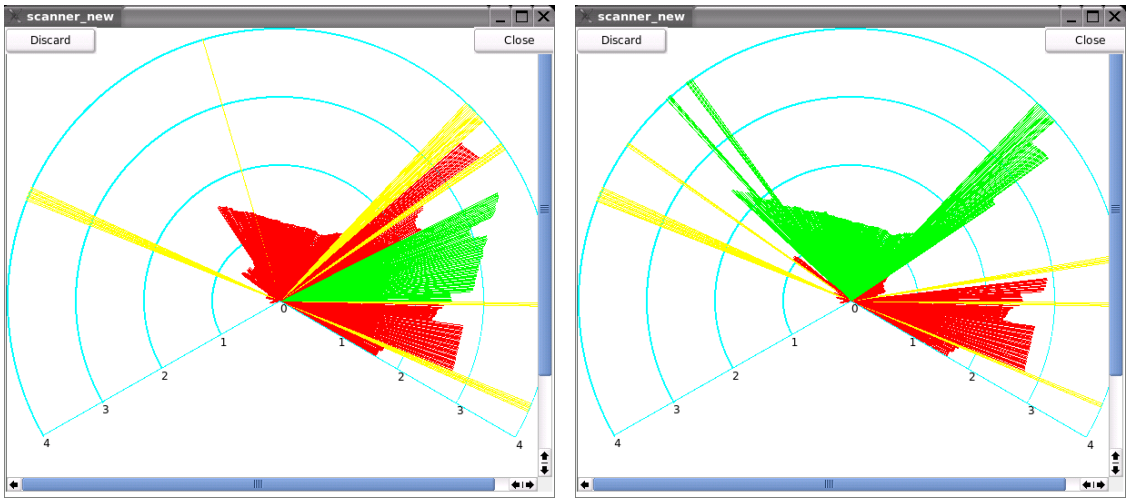


(a) Paper notebook (b) Paper notebook with CD in front of it

Figure 10: Screenshot of the scanner program showing scanner reading distortions caused by reflexive surfaces. Red beams are normal beams while yellow ones are error beams. In (a) a paper notebook is put in front of the scanner. It can be observed as an obstacle right in front of the scanner. In (b) a CD is held in front of the notebook. Besides producing significant amount of error beams, the CD also appears to be put further away from the notebook while in reality it is right in front of it.

considering a beam as part of a gap is decreased and the search is repeated. The particular scanner used had a range of 4m. The initial range started at the maximum scanner range (4m) and if no suitable gap was found, it was decreased by 0.5 m. Using this approach, the algorithm no longer had to rely on error beams only. Furthermore, the algorithm has been redesigned to support two policies for dealing with error beams. Using a `#define` one can choose at compile time whether error beams should be considered as representing free space or whether they should be avoided and no gap could contain an error beam. The latter approach is a more defensive one, where only areas would be considered about which the scanner is reasonably sure they are free. However, if this approach failed to find a gap, it would be possible to switch to the former approach. Switching during run time (i.e. for cases like a room with one open door to exit the room) has not been implemented. Basically. it would require changing the `#ifdef`'s to `if`'s and implementing the switching from one policy to another and then back. In the evaluation, where cardboard boxes have been moved around the scanner, the new algorithm seemed to identify gaps properly. Both policies for dealing with error beams have been tested. Two sample scenarios are shown in Figure 11.

As an additional optimization, a pseudo-inertia approach has been implemented. The algorithm has been biased to prefer gaps in front of it rather than turn to the side or backwards. This should prevent cases where the robot would in an autonomous run get stuck moving always between two gaps but on the way from one gap to the other always



(a) Dynamic limit avoiding error beams. (b) Dynamic limit ignoring error beams.

Figure 11: Screenshot of the scanner program showing a comparison of the algorithm when error beams are avoided, i.e. a gap is not allowed to contain error beams (a) and error beams are considered to represent free space (b). The latter approach is a bit riskier while the former one might miss certain gaps. In (b) it can be seen that the dynamic limit for considering a beam as part of a gap has been decreased to 1m.

change its preference. The bias has been implemented by multiplying the width of the gap with the triple power of cosine of the angle to the middle of the gap. The power was used to make the peak of the cosine function sharp enough for biasing gaps in front of the robot and has been found empirically. Furthermore, the angle had to be multiplied by $\frac{3}{4}$ to make the weight zero at the edges of the scanner field of view ($\pm 120°$). The formula for the weight of a gap at angle α then was $\cos^3(\frac{3}{4}\alpha)$.

5 Conclusion

Several subtasks needed for autonomous runs of the RoboCup Rescue robots have been investigated and dealt with in more detail. A basic occupancy grid has been implemented as a starting point for mapping unknown environment surrounding the robot. The implementation could be further upgraded using Bayes Theorem for updating the grid and fusing together readings from various sensors. This could significantly increase the accuracy of the grid.

The usage of a gyro for odometry has been investigated, where double integration of the acceleration (as measured by the gyro) would be used to determine displacement from starting point. It has been concluded that this is not a viable approach. Errors in acceleration measurements accumulated very fast and using them for odometry was not possible even for short time periods. Several filtering and smoothing techniques have been tested, but could not mitigate the large accumulating errors.

Finally, a laser scanner has been used to detect the largest opening among obstacles surrounding the robot. This would aid the robot in autonomous exploration of unknown environment. Initial work has been done on the incorrect assumption that error beams represent free space until the maximum scanner range. Later on an algorithm was developed that was looking for openings considering beams in iterations, where in each iteration shorter and shorter beams are considered. As an additional check, the width of the corridor leading to the opening is checked to be compliant with the robot dimensions. The algorithm implements two policies for dealing with error beams. They are either considered to represent free space or are avoided and no gap is allowed to contain an error beam. The algorithm has been evaluated by simulating obstacles using cardboard boxes. While evaluating the algorithm, the laser scanner has been found to perform very badly with reflective surfaces and metal materials.

References

[1] F. Azizi and N. Houshangi. Mobile robot position detrmination using data from gyro and odometry. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, volume 2, pages 719 – 722, May 2004.

[2] B. Barshan and H. F. Durrant-Whyte. Inertial navigation systems for mobile robots. *IEEE Transactions on Robotics and Automation*, 11:328 – 342, June 1995.

[3] A. Birk and S. Carpin. Rescue robotics - a crucial milestone on the road to autonomous systems. *Advanced Robotics Journal*, 20(5), 2006.

[4] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots in cluttered environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 572–577, 1990.

[5] J. Borenstein and Y. Koren. The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[6] H. Chung, L. Ojeda, and J. Borenstein. Sensor fusion for mobile robot dead-reckoning with a precision-calibrated fiber optic gyroscope. In *Proceedings of the IEEE International Conference on Robotics and Automation ICRA 2001*, volume 4, pages 3588 – 3593, 2001.

[7] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[8] K. Komoriya and E. Oyama. Position estimation of a mobile robot using optical fiber gyroscope (ofg). In *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, volume 1, pages 143 – 149, Sep. 1994.

[9] L. Matthies and A. Elfes. Integration of sonar and stereo range data using a grid-based representation. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, volume 2, pages 727 – 733, Apr. 1988.

[10] H. Moravec. Sensor fusion in certainty grids for mobile robots. *AI Mag.*, 9(2):61–74, 1988.

[11] K. Park, H. Chung, J. Choi, and J. G. Lee. Dead reckoning navigation for an autonomous mobile robot using a differential encoder and a gyroscope. In *Proceedings of the 8th International Conference on Advanced Robotics*, pages 441 – 446, July 1997.

[12] T. Wang and J. Yang. Certainty grids method in robot perception and navigation. In *Proceedings of the 1995 IEEE International Symposium on Intelligent Control*, pages 539 – 544, Aug. 1995.