Fast Local Mapping For Mobile Robots

Erik Govers

March 6, 2007

Universiteit Utrecht

PHILIPS

Concluding thesis for Cognitive Artificial Intelligence

Utrecht University

Philips Applied Technologies

1^{st} Supervisor: Dr. M. Wiering
2^{nd} Supervisor: Ir. B. Verhaar
3^{d} Supervisor: Prof. Dr. J.-J.Ch. Meyer

Abstract

The focus of academic research to map building for mobile robots lies most often on the algorithms that use accurate sensor data, obtained by e.g. a laser scanner, to reduce uncertainty about movement, thereby facilitating the construction of large consistent maps. This thesis describes the research that is done to the possibilities concerning fast, local map building using inaccurate sensors.

The characteristics of several sensors are analysed, both individually and together in a reconstructed setup and algorithms are evaluated to study their potential for dealing with noisy sensors in real time. A simulator is written in C++ to integrate these matters and a robot platform is constructed to test the sensors in combination with the algorithm in a real life application.

Acknowledgements

First of all, I would like to thank Philips Applied Technologies in general for giving me the opportunity to graduate on the fascinating subject of robotics. More specifically I would like to thank Boudewijn, my supervisor at Philips, for the patience that is often required when supervising a student of philosophy. Bart Dirkx and Thom Warmerdam, thanks for the useful input at our meetings. Also, I would like to thank Dr. Marco Wiering, my supervisor at Utrecht University, for all the help and for forcing me sometimes to take perspective and Prof. Dr. J.-J.Ch. Meyer for his time to review my thesis. Harry Broers, thank you for helping me out, and Sander Maas for getting me in. Further I would like to thank my parents, of course, for all the support not only during my graduation period but during my whole career as a student. A lot of thanks go out to my friends who kept asking me when I would finally finish this assignment. And last but not least, Renske, thank you for your patience and kindness especially during the last months.

Contents

Chapter 1

Introduction

When autonomous mobile robots are to be used to perform certain tasks in domestic environments, the least they should be able to do is navigate through it. Moreover, performing a task usually involves planning for which the robot needs a map; to navigate from A to B, the robot should know where A and B are. And if the locations are known, it needs information about the objects between them to avoid bumping in to things. In other words, it would help a robot a great deal if it would know more about its environment. This conclusion brings us to one of the major subjects in the field of mobile robotics: map building.

1.1 Problem Description

Problem - Intuitive Constructing a map from scratch is not as trivial as one might think. Humans tend to underestimate the difficulty of tasks they are really good at. And without any effort at all we are capable of moving through both small and large open spaces using the information from our senses, the most important of which (at least for navigation) are the eyes. The 'task' of realizing where you are given a known environment - like for example your house - is especially easy; one quick look around will tell you your location.

This last task though, which for mobile robotics is called self-localization, has provided researchers with food for thought for many years. Even when (a map of) the environment is known, finding your own position is a difficult task indeed for a robot; If the "eyes" of a robot are a 180° laser scanner, measurements can be noisy. If less accurate sensors are used the noise is worse. Maps are often ambiguous. Wheels can skid. People move through the environment. In other words, a robot has to deal with the uncertainty that arises when moving through a noisy environment doing measurements using noisy sensors. Studies to both algorithms and sensors were part of the research conducted in fulfilment of this graduation thesis.

Algorithms For decades now a lot of progress is made in the inspiring research field of robotics. The problem known as localization in which a robot has to find and/or track its own location is practically solved, e.g. [11] [2], at least under certain conditions. Mapping, or map building, is the more difficult problem of building a map of the surroundings and is often mentioned together with localization since in practice both problems have to be solved together; to build a map, the robot needs to know where it is within this map. Simultaneous Localization and Mapbuilding is often abbreviated as SLAM.

Several techniques have been unleashed onto the problem of robotic mapping. One of the leading researchers in the field, Sebastian Thrun, has made an excellent survey [10] in which a lot of these techniques are discussed, several of which have proven to be useful. Some algorithms work better under specific conditions and others are useful under more circumstances but are too slow to work online. There's one thing however all algorithms have in common: They try to filter out as much uncertainty as possible from the continuous stream of data that comes from the sensors, which makes them dependent on the accuracy of the sensors.

Sensors There are several common sensors that are used in robotics today. From [10] it is clear that it's already possible to build quite accurate maps with expensive hardware like a SICK laserscanner or (stereo) vision. It is even possible to build maps on the fly, enabling the robot to detect objects at the moment they get close, and simultaneously adjust the map. However, map building gets challenging when less accurate sensors are used. Using inaccurate sensors such as ultrasound or infrared range sensors is interesting from two points of view; a Sharp infrared range sensor is typically 100 or more times cheaper than a 180° laser scanner, and then of course there's the scientific challenge to create better algorithms to deal with the increase in the amount of noise.

Research This graduation thesis is a report of the research, conducted by the author, to the possibilities concerning fast and local map building in mobile, autonomous robots using cheap hardware. Research is conducted to the nature of several sensors and to map building algorithms with potential to be versatile and potential to be fast. The results of this research are combined to study the possibilities of constructing local maps using existing algorithms adapted for use with inaccurate sensors.

1.2 Relevance to AI

Artificial Intelligence is a very broad field of research. Anything remotely related to human intelligence, including logical reasoning, speech recognition and synthesis, but also the biology of the brain and autonomous robotics in general, can be said to belong to AI.

The reason why a non-autonomous robot (e.g. a welding robot in a car factory) doesn't have to be intelligent might be obvious, but for the sake of clarity it's stated here anyway: Since the robot's working environment is known with a high degree of certainty, there's no need for the robot to respond to this environment. It is possible to make it perform its job using time as the only variable. To continue our example, the robot knows the location of the object it needs to weld at any point in time and uses this information to determine where and when it should start. In other words, it has no need to extract and use information from the environment. In contrast to an autonomous mobile robot, that is supposed to perform tasks in a completely undefined and dynamical environment.

From the advent of robotic research until not such a long time ago, a lot of effort was being put in studying only the theoretical reasoning behavior of a robot. Several kinds of modal logic and agent technology for example are not only applied for software entities; (Academic) research is conducted to use these techniques in real mobile robots. Intensive pondering about the usage of information however is of no use if the information concerned is not readily available. To enable high level reasoning about a noisy, unpredictable world, some gaps have to be closed. And this is why research about the use and integration of sensors is so useful. It is a major challenge to clean up the noisy data gathered by sensors, and to provide the (high level software of the) robot with accurate and clean information about its surroundings. It's inherent to measuring in the real world that sensor data is noisy, even when the sensor itself is quite accurate. Skidding of wheels can be enough to send a robot completely astray. The only way to deal with this uncertainty is to analyze large amounts of raw sensor data. Smart algorithms are obviously necessary to do this in realtime. In this thesis a description of several of such algorithms is given. Both sensors and algorithms are tested to discover the best combination(s).

So in fact, the relevance of this graduation thesis to the field of artificial intelligence is twofold: First of all, the algorithms that deal with real world data have to be 'intelligent' by themselves in order to be able to extract the useful information. This is what the algorithms discussed in this thesis do: They implicitly remove unwanted data and incorporate what they 'think' is good data in their world model. Second, because of these algorithms and the information they can provide, more high level reasoning becomes possible; accurate data about the real world is of great use to a mobile robot.

1.3 Project Description

This graduation assignment consists of three tasks that are to be performed in parallel: The first assignment is to do a literature study about Simultaneous Localization And Mapping (SLAM) by mobile robots. The focus of the literature study will be on algorithms suitable for fast and local map building in domestic environments, and on sensors suitable for this task. The second assignment will elaborate on the subject of sensors; several (infrared, ultrasonic

and perhaps other) sensors are to be tested in practice using a simple setup, evaluating them on their potential for use in mobile robots for domestic environments. Requirements: A typical refresh rate of 1 Hz, with an accuracy of 0.1 m. The third assignment is building a simulator for a standard PC, paying special attention to accurately mimicking the behavior of the tested sensors. When these three assignments are (at least partially) completed, the algorithm used in the simulator is implemented in a real-life application for testing purposes. Both simulator and real-life platform can assume a global, top level map is available; The platform should be able to map unexpected objects within the global map.

1.4 Approach

Ultrasound and infrared range sensors are tested in a simplified environment. The infrared sensor was tested using both rectangular and round objects at varying distances to find out at what distance the sensor is useful. The ultrasound sensor was tested in a different way; due to the expected sensitivity to reflections, care had to be taken to place the robot in the right setting. The shape and reach of the ultrasonic beam is investigated, as well as the influence of reflections on measurement results. Results of this research can be found in chapter 2. Different algorithms are studied and discussed in chapter 3. From the potential candidate algorithms one was chosen to be adapted to match the needs of fast and local map building. The algorithm in question, DP-SLAM, and its adjustments are explained more elaborately in chapter 4. When the sensors and algorithms were selected, construction of the simulator and the robot could commence. In practice, work on the latter started even earlier in order to finish this project in time. The software was written in such a way that it could be used for both the simulator and the robot. In chapter 5 a full description of both hardware and software of the mobile robot platform is given. The results section shows both maps produced by the simulator as well as maps built with the real robot. The outcomes of the simulator are compared to those of the robot to gain more knowledge about the capabilities of the robot as a whole. Also, DP-SLAM is applied to the raw sensor data acquired by the robot in an attempt to improve the accuracy of the resulting map. Results are given in chapter 6 and the conclusion can be found in chapter 7.

Chapter 2

Sensors

2.1 Introduction

Many different sensors can be used for dealing with all sorts of (localization and mapping) problems by researchers with all kinds of needs. There are infrared sensors, ultrasonic sensors, cameras, inductive sensors, capacitive sensors, laser scanners, and so on and so on. The choice of the right sensor is important since it strongly influences the choice of the mapping algorithm. It is not within the scope of this assignment to test every possible sensor using several kinds of algorithms. Choices had to be made, and since the main interest of this article is a fast local map building robot, accurate expensive sensors like laser scanners were not considered.

In literature, a sensor that is often used for robotic map building is the SICK laser scanner, see for example [13] [4] [1]. For academic purposes, this is indeed a plausible choice. A lot of accurate range information can be obtained by the laser scanner so the map building algorithms can focus on compensating for motion error, the subject of interest of most map building algorithms. See [10] for a summation and examples. A major disadvantage of this sensor is that it is too expensive for use in a commercial product; the exact price depends on the model, but such a device costs several thousands of euros at the time of writing [18]. To process the significant amount of data the scanner generates, a pc or the like is needed that contributes to the overall costs as well. Moreover, from a scientific point of view, it is a major challenge to try and overcome the assumption that measurements need to be as accurate as they are with the laser scanner, and still be able to build maps.

Years ago researchers, among whom Moravec [6], started doing research to localization - using ultrasound sensors. Technology has advanced since then and now most often the laser scanner is used in combination with a fast computer. During this graduation project, an attempt is made to adjust the current laser scanner algorithms for use with ultrasound sensors. To do this, a specific ultrasound sensor and its behavior in a complex environment is studied. Also, an

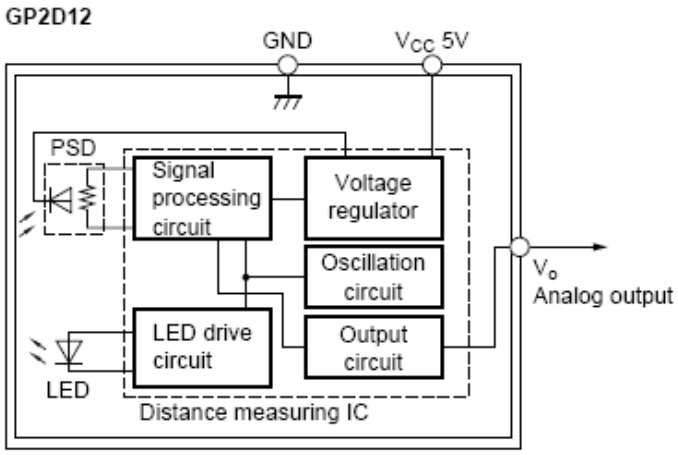


Figure 2.1: Internal workings of the Sharp GP2D12 infrared sensor. This figure comes from the datasheet, see appendix

infrared sensor, the Sharp G2DP12, is studied in the same manner to find out if it is possible to use this sensor for local mapping as well. Below is a description of the results of these studies[1].

2.2 Infrared

2.2.1 Operation & Use

There are several common principles used for infrared distance measurements, two of which (based on either triangulation or light intensity measurement) are found in relatively cheap distance sensors. A third method is based on time of flight of light, used for example in the laser scanners from SICK. Since this last method works on the principle of measuring the time an infrared burst of light takes to return an 'echo', the device's electronics have to be very accurate and very fast (certainly compared to the hardware of the sensors we're about to review), which makes it a commercially unattractive mobile robot.

An example of a sensor that *is* commercially attractive is the Hamamatsu [16] P5587, which is a very simple sensor using a photosensitive IC that measures infrared light intensity to output a high or low signal to indicate presence of an object. Sensors like this can be used e.g. for paper detection in printers, or end

[1] Originally the robot was to be equipped with a pair of Laser Beetles, laser based computer mice that could be able to replace the conventional wheel encoders. Some time has been put into getting the mice to work in software, unfortunately without success. The basic framework is present in the software and the Laser Beetles are physically attached to the robot, so perhaps in the future it is possible to test the sensors individually and integrate them into the platform completely.

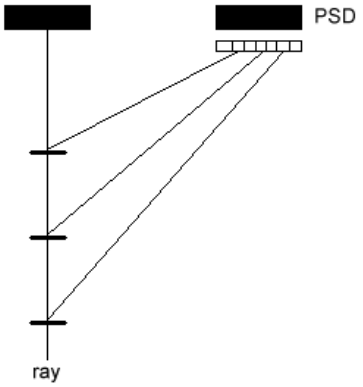


Figure 2.2: The principle of triangulation. The infrared ray of light falls on an object and is reflected through the lens to the PSD.

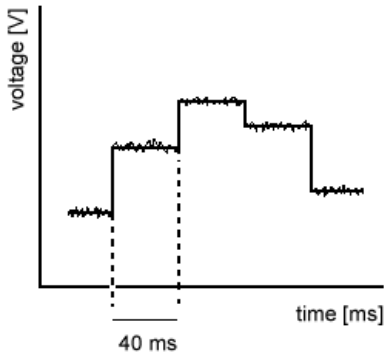


Figure 2.3: Reconstructed example of analog output from the Sharp GP2D12 infrared sensor. The analog output, which is updated with 25Hz, suffers from electric noise.

of tape detection in VCR's. An application in the area of robotics in which this sensor would be useful is an encoder. However, for measuring distance, we need something more sophisticated, that is nevertheless cheap enough. Fortunately, Sharp [17] offers such a sensor. The sensor used in the experiments is the Sharp GP2D12 infrared distance sensor. The electronics are relatively simple, making it a cheap and easy-to-use sensor for range sensing. It works by emitting a beam of infrared light, and uses a position sensitive device (PSD) to do triangulation, see Schwope [12]. Figure 2.1 depicts a block diagram of the internal workings of the Sharp sensor.

The PSD is a photosensitive semiconductor device [12]. Light emitted by an infrared LED that is present in the sensor unit reflects from an object back on to the PSD, right before which it passes through a lens, see figure 2.2. The location on which the ray of infrared light hits the PSD depends on the angle of incidence on the lens, which in turn depends on the distance between the object and the lens. Altering this distance means a change in the flow of currents in the PSD which in turn can be translated into a distance to the measured object using the formula $C * (I_A - I_B)/(I_A + I_B)$ where I_A is the output current on one side of the photo sensitive layer, and I_B the output on the other side. C is a scaling factor. The technique of passing a ray of light through a lens on to a PSD (or sometimes a CCD) is called *triangulation*.

The analogue output from the GP2D12 is not updated continuously. Instead, the output value is refreshed at a rate of 25Hz. After each 40ms, the V_output line is set to a new value. A (reconstructed) output sample is pictured in figure 2.3. Unfortunately, as the reader can see, there's some random electric noise on the output which we try to cancel out by taking the median of three readings.

2.2.2 Performance

The GP2D12 should be able to measure a distance from 10cm to 80cm according to the company's data sheet (see appendix). This is tested in a simple setup. Different objects at varying ranges were placed in front of a sensor attached to the robot. A DLP245PB microcontroller (see chapter 5 and the appendix) polls the sensor every 8ms as mentioned. The latest three measurements are stored in the microcontroller's onboard memory. When a request (from a pc or laptop) is received, it sends the median of these three values. In the experiment, ten of these requests were made for every tested distance, using a laptop attached to the microcontroller. Tests were performed using a flat white surface (copier paper) measuring 10cm x 10cm. The surface was measured in two ways: One series of experiments was done with the surface perpendicular to the sensor's beam, for another series it was kept at an angle of 45◦. A third series of experiments was performed using a curved surface, covered with the same white paper, see also figure 2.4. Figure 2.5 shows the results of the experiments.

In practice, the minimal distance is somewhat better than the 10cm mentioned in the company's specifications. Figure 2.6 depicts the distance estimates based on the median of 10 requests (i.e. 30 measurements) to the microcontroller, which are evidently more accurate than distance estimates based on 1 or 3 requests, see figure 2.8 and 2.7. As one can see in figure 2.5 and 2.6 sensors readings become consistent from about 8cm - which is good news for robotics. Readings below 8cm produce ambiguous results, which are potentially disastrous for map building, but this problem can be solved by placing the sensors at least 8cm into the robot. The robot used in the experiments (see chapter 5) has not been altered to solve this issue. However, no experiments were performed in which the distance of the sensor to an obstacle became less than 8cm.

The maximum distance is significantly less than 80cm. It's hard to estimate what the maximum useful range is, since accuracy gradually declines, but figure 2.6 suggests output is consistent up to about 60cm. Figure 2.8 however indicates that measurement error can become more than 5cm for distances further than 40cm. This graph is based on medians of 3 measurements, which is the same information density the robot uses. Taking the median of more measurements might seem the solution here. More measurements however mean more delay. As mentioned in the "Operation & Use" section the sensor updates its value every 40ms. Taking the median of 9 measurements (figure 2.7) or even 30 measurements (figure 2.6) is bound to produce better results due to the fact that this simple 'filter' does not only cancel out the electric noise; 9 measurements are done in $9 * 8ms = 72ms$, which means that they cover two or three sensor updates. This is not a problem for a static experimental setup, but for a moving robot it could be, depending on the speed of the robot and the desired spatial resolution.

From figures 2.6, 2.7 and 2.8 we gather that information is consistent at least between 8cm and 40cm. Between 40cm and 60cm output seems to decline gradually, but still appears to be (albeit less) consistent. When we look more closely to the data by calculating the standard deviations (see figure 2.9), output noise

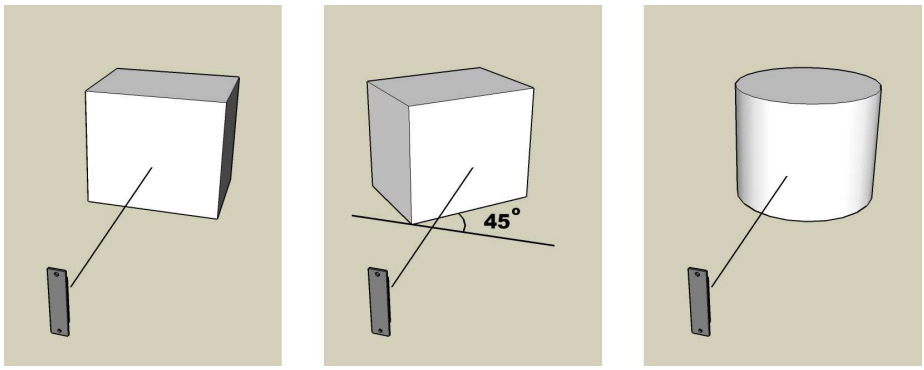


Figure 2.4: Measurement setup for the Sharp GP2D12 infrared sensor. The objects were tested at various distances.

indeed seems to increase rapidly for distances over 400mm. When the object is more than 60cm from the sensor, output is too much affected by noise to provide enough information for use in mapbuilding. In short, the sensor appears to be sufficiently accurate between 8cm and 60cm. The black line in the graph is the trendline of the blue line (which shows measurements of an object perpendicular to the sensor's beam). It indicates that standard deviation doubles when the object distance increases from 400mm to 600mm. After 600mm, standard deviation increases even faster. Perhaps it is possible, in future research, to investigate the possibility of using the data originating from distances between 400mm and 600mm by using an accurate lookup table or a better function fit.

All of the above holds for the objects from the experiments. Since the infrared sensor works on the principle emitting and receiving light, it's inherent to this kind of measurement that errors come to bare when the light beam is too heavily altered or influenced by a measured object. Altered, for example, when the surface of the object is black. Some experiments were performed with black objects. An effect noted during experiments is the 'invisibility' of an object when it is at a distance of about 30cm. It is the author's experience that when a black object is near, output is as consistent as with a white object. Further away, the amount of reflected light comes below a certain threshold which causes the sensor to stop detecting the object. The distance threshold for becoming invisible to the sensor will probably vary with the type and color of the material. As mentioned, no real data is available at the time, but it is advised to research this in future experiments.

An object that doesn't reflect enough light can form a problem. However, the opposite is true as well; if the object reflects light almost perfectly without divergence the sensor is only able to detect its presence if the measured surface is (nearly) perpendicular to the emitted light source. An object having an oblique angle will reflect the light away from the infrared PSD, thereby creating the same effect as a black surface would; Again, the sensor will tell us there is no object.

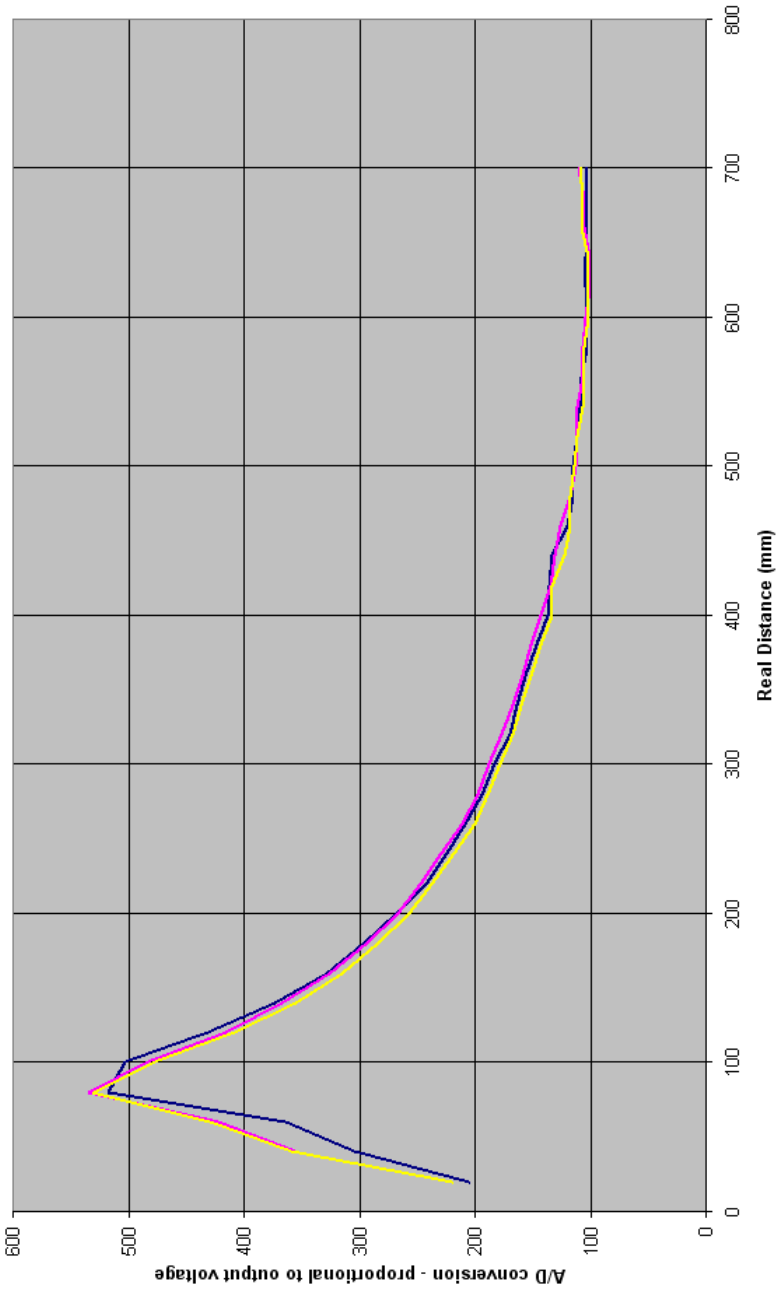


Figure 2.5: Infrared sensor output (using the median of 30 distinct measurements) from distance 0cm to 70cm. The blue line represents data gathered with the flat surface perpendicular to the sensor beam, the purple line shows the data from the same surface under an angle of 45o and the yellow line is the curved surface data (see figure 2.4).

Figure 2.6: ADC values converted to distances using the medians of 10 requests. Raw sensor data appears to be easily translated to centimeters using the equation $Distance = 50000/(Value - 15)$, at least for the distance between about 10 and 40 centimeters.

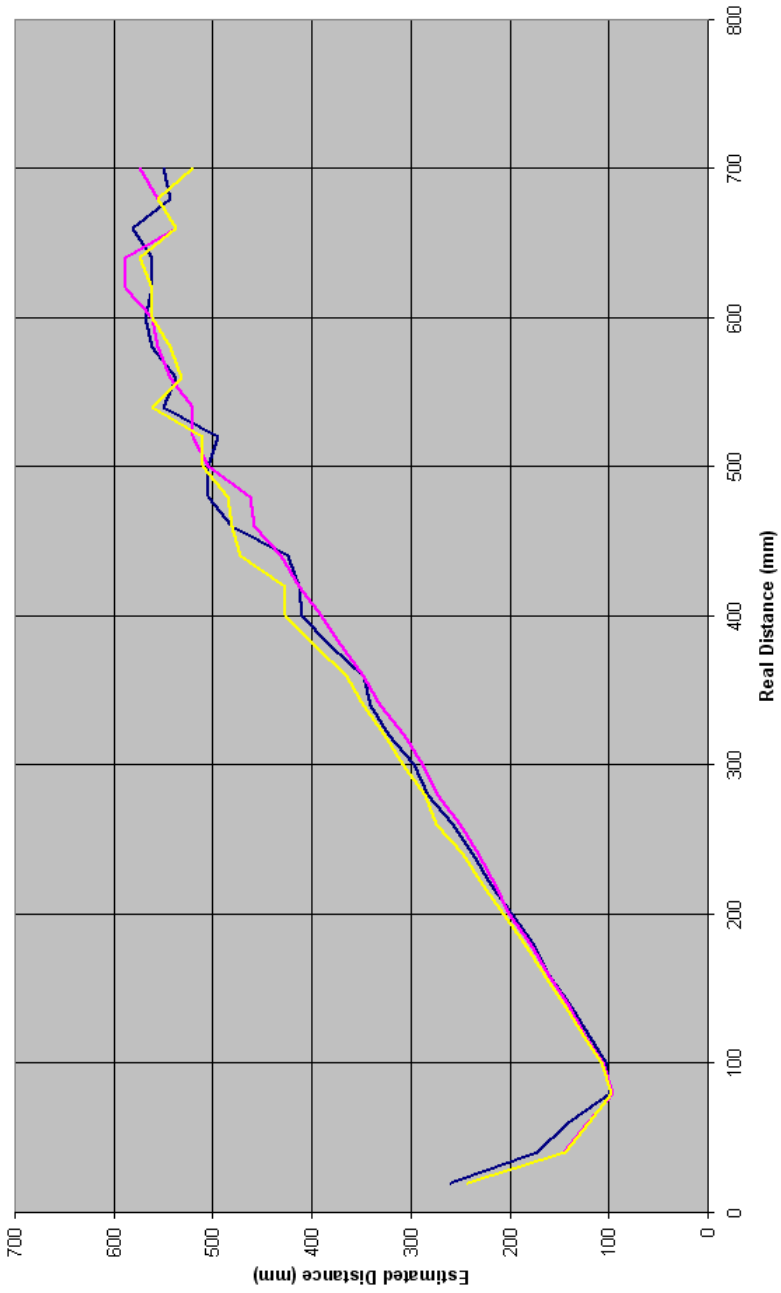


Figure 2.7: ADC values converted to distances using the medians of 3 requests.

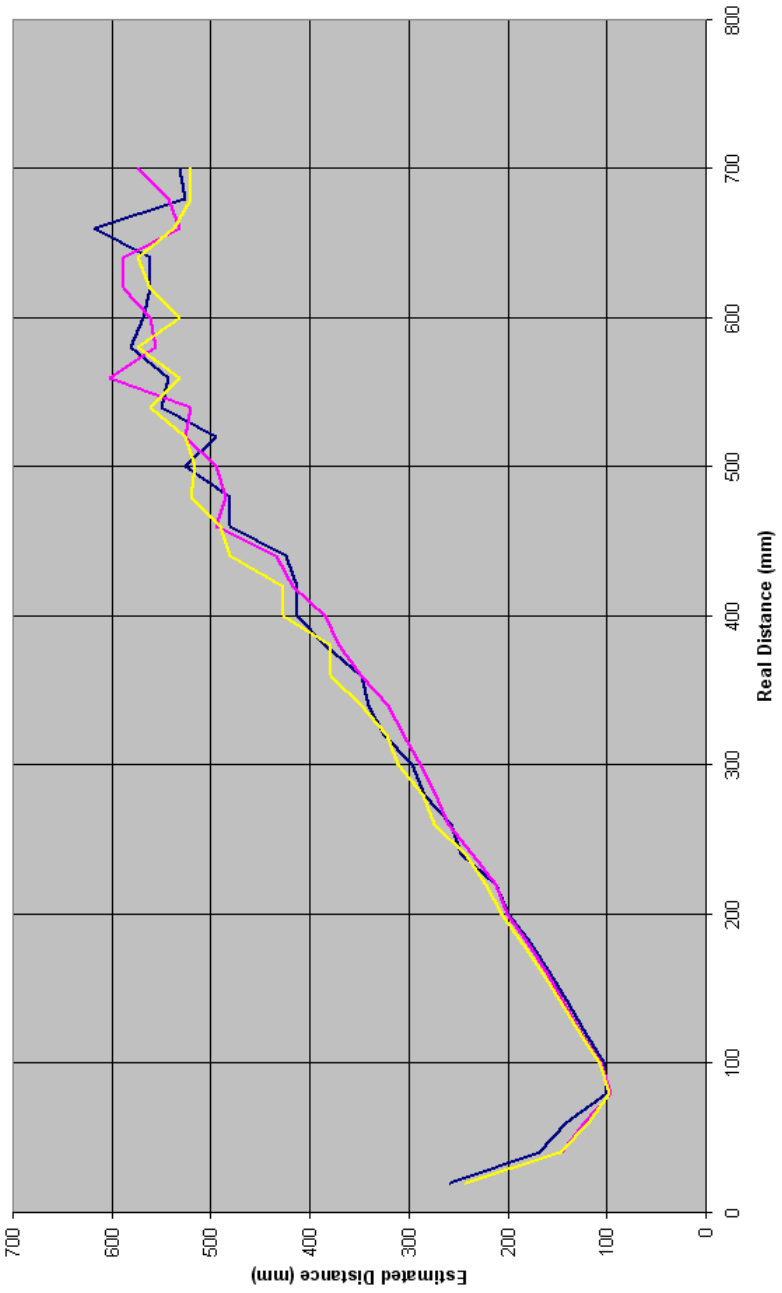


Figure 2.8: ADC values converted to distances using only 1 request, i.e. the median of 3 outputs.

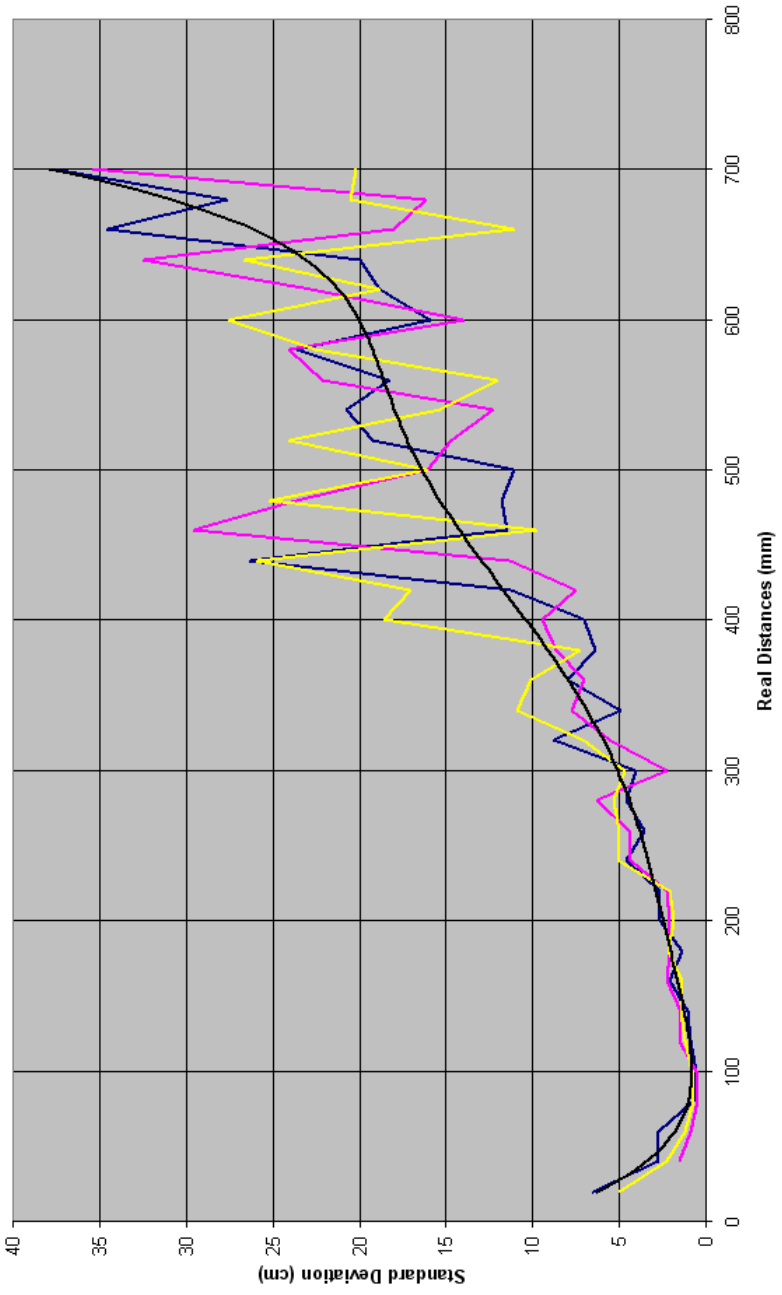


Figure 2.9: Standard deviations of 30 infrared range sensor measurements. This graph suggests that using e.g. a detailed lookup table instead of the described equation might increase accuracy for measured distances betweeen 400mm and 600mm.

To summarize, the sensor is blind for black and highly reflective objects, and objects that are further away than 60cm. Curved surfaces as well as surfaces that are not perpendicular to the sensor's beam produce . It is therefore advisable to be very careful when interpreting range information from an IR measurement. If the sensor outputs its maximum value -indicating that it didn't measure anything- it would be obvious though quite unwise to take this information for granted and update the grid cells of the map that the infrared beam has passed through as being empty; there are many circumstances in which the sensor is 'tricked' by the environment. However, when the sensor *does* measure something, it's relatively certain that an object exists at the concerned location; since the sensor is relatively 'picky' about the measurement conditions, an actual measurement -accurate or not- at least implies that there's some object within its line of sight. In short, information from an infrared sensor is only useful when a measurement indicates a nearby object. Using a maximum range measurement is *not* recommended since there's quite some chance that the sensor has missed an existing object.

Concerning speed of the robot and the ability to avoid objects using the Sharp GP2D12: Assuming electrical noise is less important and one is using only one measurement, object detection speed is limited by the sensor's refresh rate of 25Hz. Output is accurate enough for object avoidance from 8cm up to 60cm; a window of about 50cm. A robot with such a sensor attached to its front measuring straight ahead would therefore be able to drive[2] 50cm in 40ms, i.e. 12.5 m/s. However, due to the properties mentioned above, using this infrared sensor alone is not sufficient for mapping.

2.3 Ultrasound

2.3.1 Operation & Use

See also the datasheet in the appendix. For ultrasonic measurement, a Devantech ultrasonic ranging module, model SRF04 was used. Or more specifically, the 400ST160 (transmitter) / 400SR160 (receiver) combination residing on the module. The same PIC that was used to communicate with the infrared sensor is now used to trigger a sound burst on the SRF04 and to receive the echo. For more information about interfacing the PIC, see section 5.3.

Although the Sharp GP2D12 refreshes its output every 40 msec, it can be polled much more often. The output of an ultrasonic sensor depends on the time of flight of the sound burst. The sensor is able to measure up to quite a distance (see 2.3.2), but the echo that is to be received will take a long time to travel back to the sensor. To get data from the SRF04 we will need a totally different approach than with the ir sensor. The same holds for the interpretation of the data itself; integration into the map is totally different from the infrared sensor. More about the latter can be found in chapter 4. In this section, an explanation about the operation of the sensor is given.

[2]Making the unrealistic assumption a moving robot is able to stand still in .0s

Figure 2.10: The SRF04 ultrasound sensor

Figure 2.11: SRF04's backside

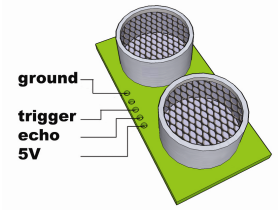


Figure 2.12: Pin out of the SRF04

Figure 2.10 shows a picture of the sensor, 2.11 shows the backside and figure 2.12 is a schematic drawing with labels at all the connections. "Trigger" and "Echo" are of interest here. The trigger pin is controlled by the microcontroller. A 5V pulse of at least 10 μs needs to be applied to the pin to start the ranging module. A burst of ultrasound is sent from the SRF04 and the echo line is raised. This is the signal for the microcontroller to start counting. If an echo is received, the echo pin is lowered again. Please see figure 2.13 for a schematic drawing of the timings.

A timing example: Assuming the speed of sound is 340 m/s, it will take about $(2 * 3)/340 \approx 0.018$ seconds (18 ms) for the sensor to measure an object at a distance of 3 meters. The sensor has a time out built in, in case the sound wave is absorbed or deflected completely and there's no echo at all. That means in the worst case scenario we have to wait for the timeout to occur, which is 36 ms. The maximum range of the ultrasound sensors is 3.7 m (see 2.3.2). An object at this distance will return an echo after $(2*3.7)/340 \approx 0.022$ seconds (22 ms). The time out is therefore inconveniently long. And although it is possible to keep track of time and create a software based time out in the microcontroller, if the sensor doesn't receive an echo it does not reset before the 36ms, plus an extra 10ms, are over. A robot traveling at a speed of 2 m/s, traverses 9.2 cm in 46 ms and can 'miss' objects of 9.2cm wide completely.

When using multiple sensors, the only way to be reasonably sure a sensor only receives the echo of its own sound burst, is to not trigger (adjacent) sensors simultaneously, but to spend at least 22ms on every sensor. This means for an array of 6 sensors the amount of time needed to poll the entire sensor array increases to 132ms. Fortunately, the difference between the required time out of 22ms and the real time out of 36ms plus 10ms is cancelled out by the fact that polling all sensors often takes more time.

2.3.2 Performance

On the internet, quite some unofficial information about the Devantech SRF04 range sensor can be found, see for example [9]. However, since this information is unofficial, several things needed to be tested in practice to make sure it was correct, including the shape of the ultrasonic beam and timing issues.

Tests were done using different objects. The same metal bin that was used

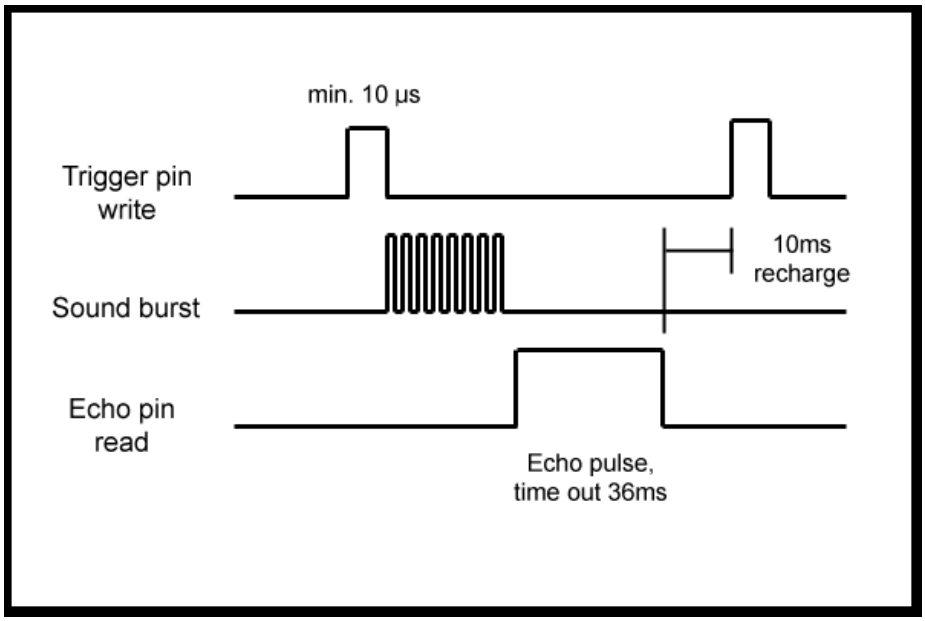


Figure 2.13: Timing manual of the SRF04 ultrasound module, from the datasheet, see appendix.

in the assault course (see chapter 6) is held perpendicular to the sensor's beam, and at an angle of 45◦ with the sensors's beam. A round metal bin of 26cm in diameter, also used in the assault course, is measured while keeping it exactly aligned with the sensor's beam.

Because of the wide area the ultrasonic sensor measures, one single measurement contains relatively much information about the environment, especially when the measured distance to an object is relatively large. If the sensor returns some distance value, it's relatively safe to assume that there's nothing occupying the space between the robot and the measured object. The setup used for the beam shape measurements is shown in figure 2.16. Figure 2.14 is a schematic drawing of the shape of the ultrasound beam. The larger the angle of measurement, the less the range of the sensor.

Since the sensor can measure up to 3.7m, one measurement tells a lot about empty space. Unfortunately - and perhaps initially a bit counter intuitive - it tells little about a measured object. The ultrasonic sensor returns a distance, but doesn't tell much about the width of the obstacle. An infrared sensor's beam has a certain width and an IR measurement result comprises a coordinate prediction along with some uncertainty about the prediction. However, the ultrasonic sensors beam is too wide to speak of a single location estimate; a complete line segment is needed to represent the possible locations of the measured object. Please look at figure 2.15 which illustrates the difference between the infrared and ultrasound measurements. A natural reaction to this problem would include using a Gaussian distribution to still be able to represent the uncertainty, and

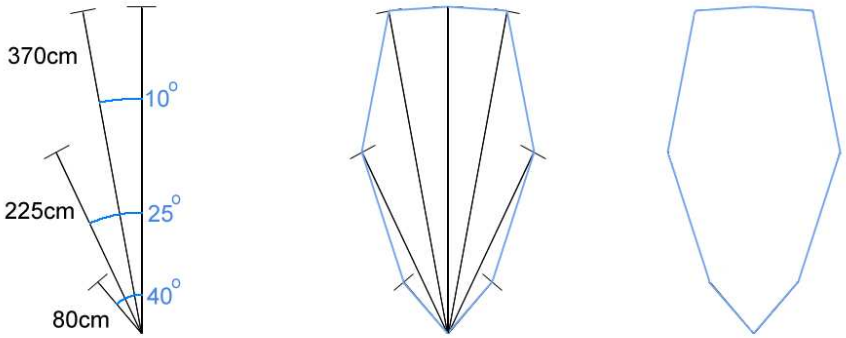


Figure 2.14: Field of view of an ultrasound sensor as measured in the experiments.

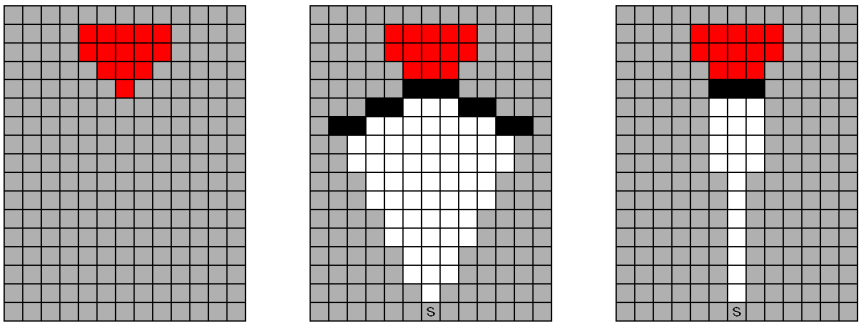


Figure 2.15: On the left is a grid with an object in it (red). The middle figure depicts a standard discretised ultrasound beam (originating from a sensor below), the rightmost picture is an example of an infrared beam. The black gridcells represent possible object locations. In the ideal case, only red gridcells are marked black. Both an ultrasound and an infrared beam have a certain width however which introduces uncertainty about the location of the measured object. Since an ultrasound beam is much wider than an infrared beam, more uncertainty is introduced.

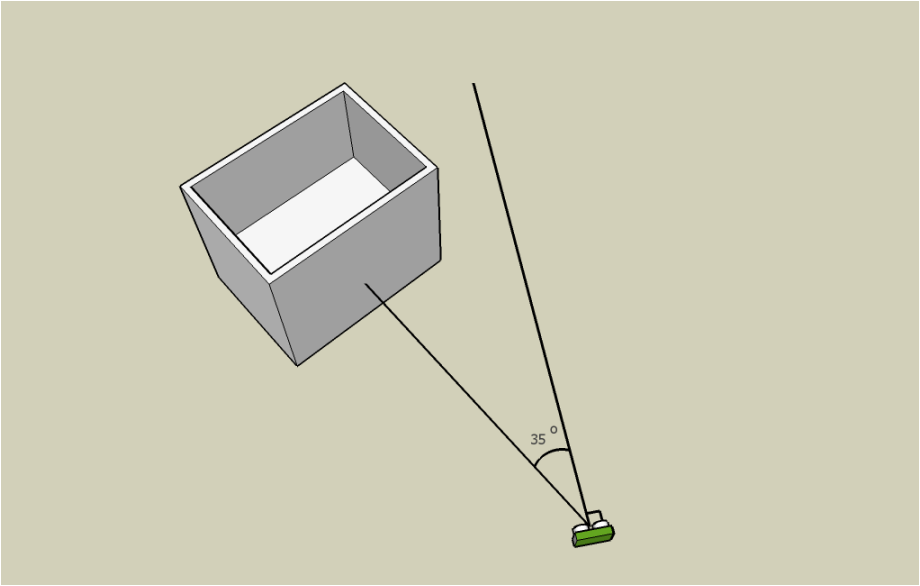


Figure 2.16: Measurement setup for the beam shape experiments. A rectangular, metal bin of 30cm wide was used to measure performance at 0°, 10°, 25° and 40° at various distances.

use the center of the line segment as the mean. Moravec and Elfes [6] use a scheme in which every cell in the occupancy grid contains two values: the certainty that the cell is occupied and the certainty that the cell is empty. They use this scheme to do exactly what has just been described; the line segment is represented by a Gaussian distribution.

There are reasons to doubt whether this is a good choice; First of all - especially when you assume that most objects are reasonably small - you are very likely to introduce more noise than real information (see figure 2.15). Second, it actually makes no sense to assume that one point on the line segment should be the mean; any point has just as much chance of being the real origin as any other. It's a pity to throw away information, but introducing a lot of noise might be even worse. It would be a better idea to either leave out any object related information or use a more intelligent scheme to introduce (uncertainty about) objects. Using particles, one might be able to represent multiple possible locations, because the particle filter offers the possibility of filtering out the introduced noise afterwards. This holds for the approach in which a lot of uncertainty is used to introduce possible object locations, but in combination with a more advanced scheme to represent the location of objects the particle filter should perform even better. Please take a look at section 4 for more information concerning the implementation of the sensormodel into the algorithms.

We've seen that due to timing delays, information isn't very dense; 132ms for an entire sensor array. The sensors are polled one by one, which means that every sensor operates at this refresh rate as well. If it is necessary to take the

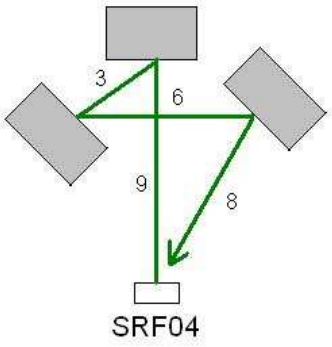


Figure 2.17: The numbers here are fictive. This is a phenomenon that is actually observed. With little effort a setup of 3 metal bins is built that can fool the SRF04 in a very consistent manner: The ultrasound sensor will return a number that is directly proportional to 9+3+6+8.

average or median of several measurements to cancel out noise there's not much play for fast mapbuilding.

Figure 2.20 shows us distance estimates based on a single ultrasound measurement. Figure 2.19 depicts the medians of 3 readings. This is the information density the robot uses. The microcontroller keeps the latest three measurements in its memory, and sends the median of those three measurements to the laptop when software indicates it needs information. Figure 2.18 shows the same graph, only this time one point represents the average of thirty measurements. Data looks reasonable consistent for figure 2.18. For figure 2.19 consistency seems to gradually decline. Figure 2.20 shows that consistency is even worse when using only one measurement.

Figure 2.21 shows the standard deviations in centimeters of the measurements as a function of the distance to the object. This graph is made with 30 measurements per point. Apparently, standard deviation is at a minimum (indicating consistent measurements) at 160cm.

A phenomenon that should also be noted is the property of a sound beam that it can reflect of certain surfaces. The metal bin that was used for testing for example bounces the beam away from the sensor. The angle under which the sound burst deflects from the smooth surface of the metal bin depends on the distance to the sensors. The exact circumstances under which this deflection can occur have been investigated, though measurements were inconsistent. More thorough investigation is needed to fully understand the reflections of the SRF04. For an observed example of the sensitivity of the sensor to reflections, see figure 2.17.

Concerning speed of the robot and the ability to avoid objects using the

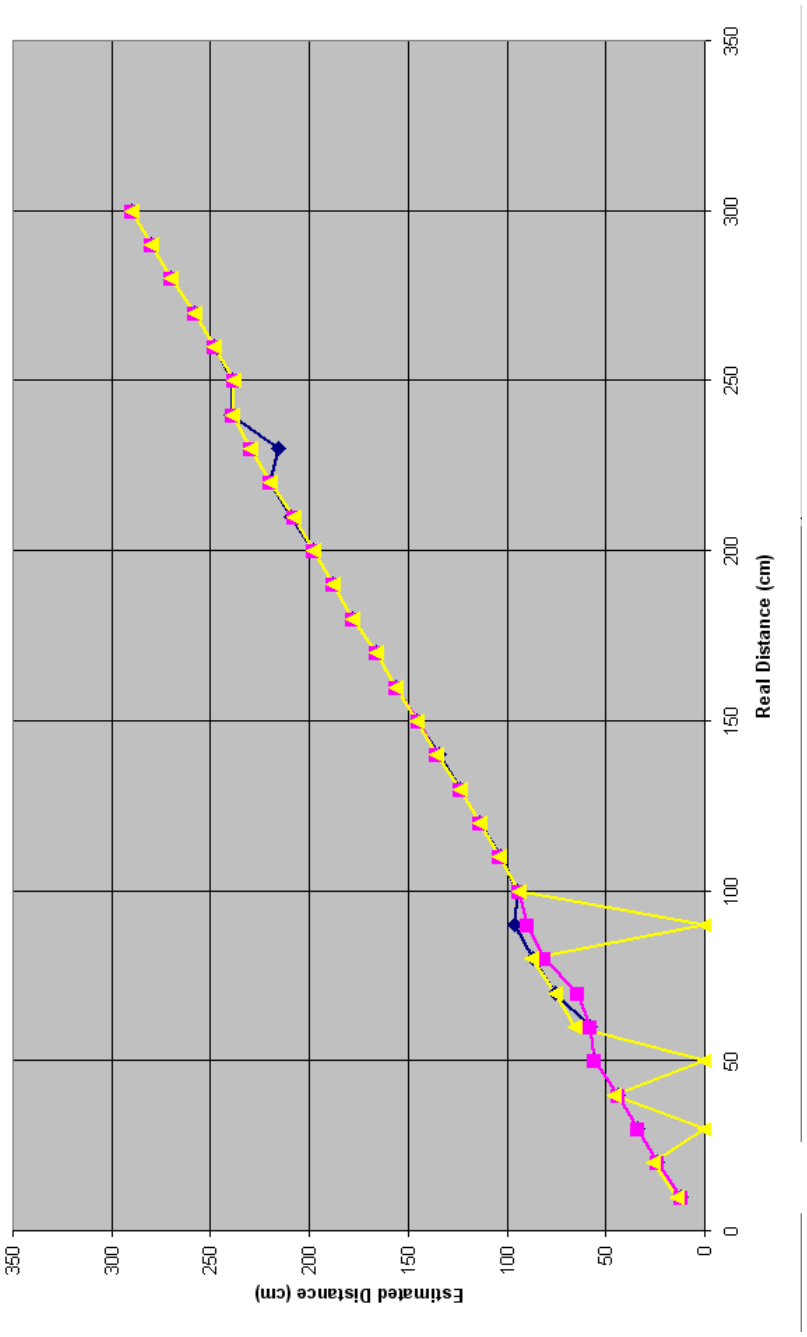


Figure 2.18: Ultrasound sensor output transformed to distance estimates. Every point is the median of thirty measurements. The blue line depicts the data gathered with the object right in front of the robot, the purple line is based on measurements of an object under 20◦ and for the yellow line the measured object had an angle of 30°. For an example please see figure 2.16.

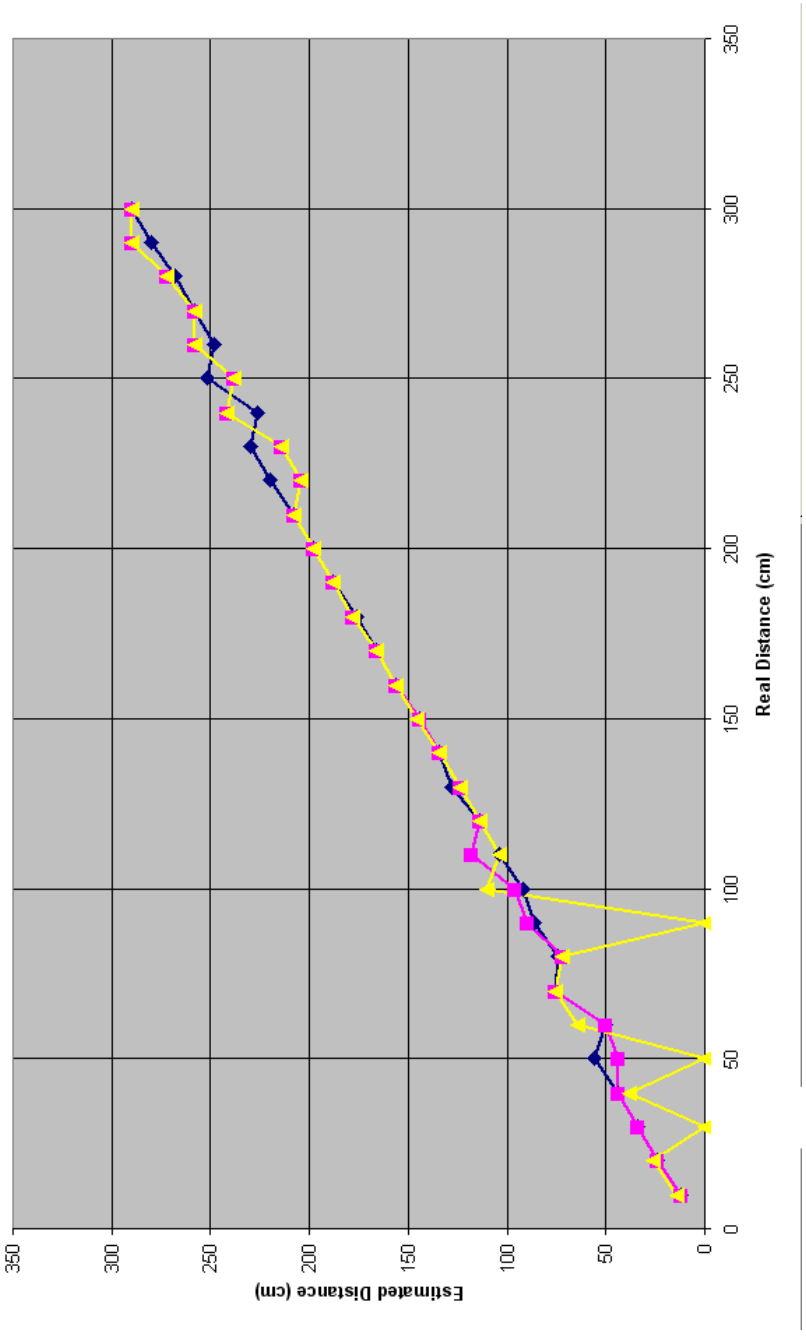


Figure 2.19: Ultrasound sensor output transformed to distance estimates. Every point is the median of three measurements.

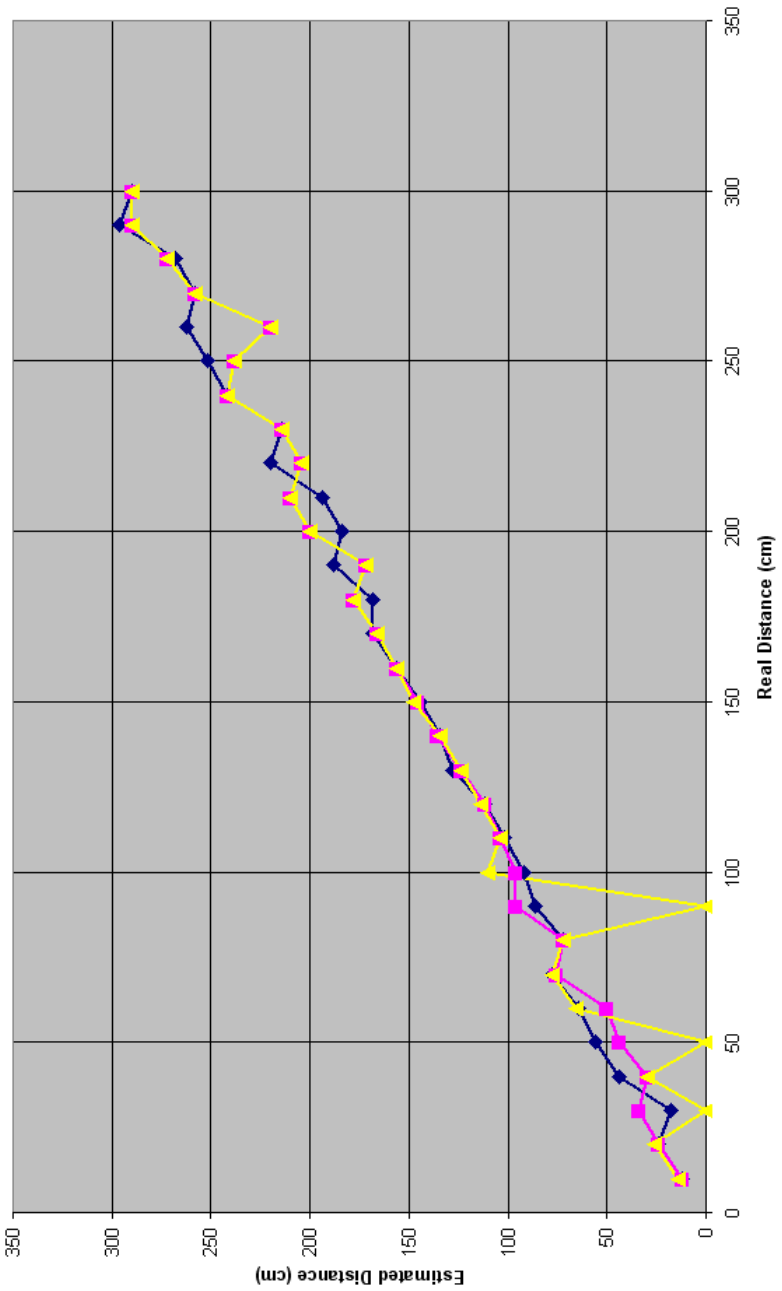


Figure 2.20: Ultrasound sensor output transformed to distance estimates. Every point represents one measurement.

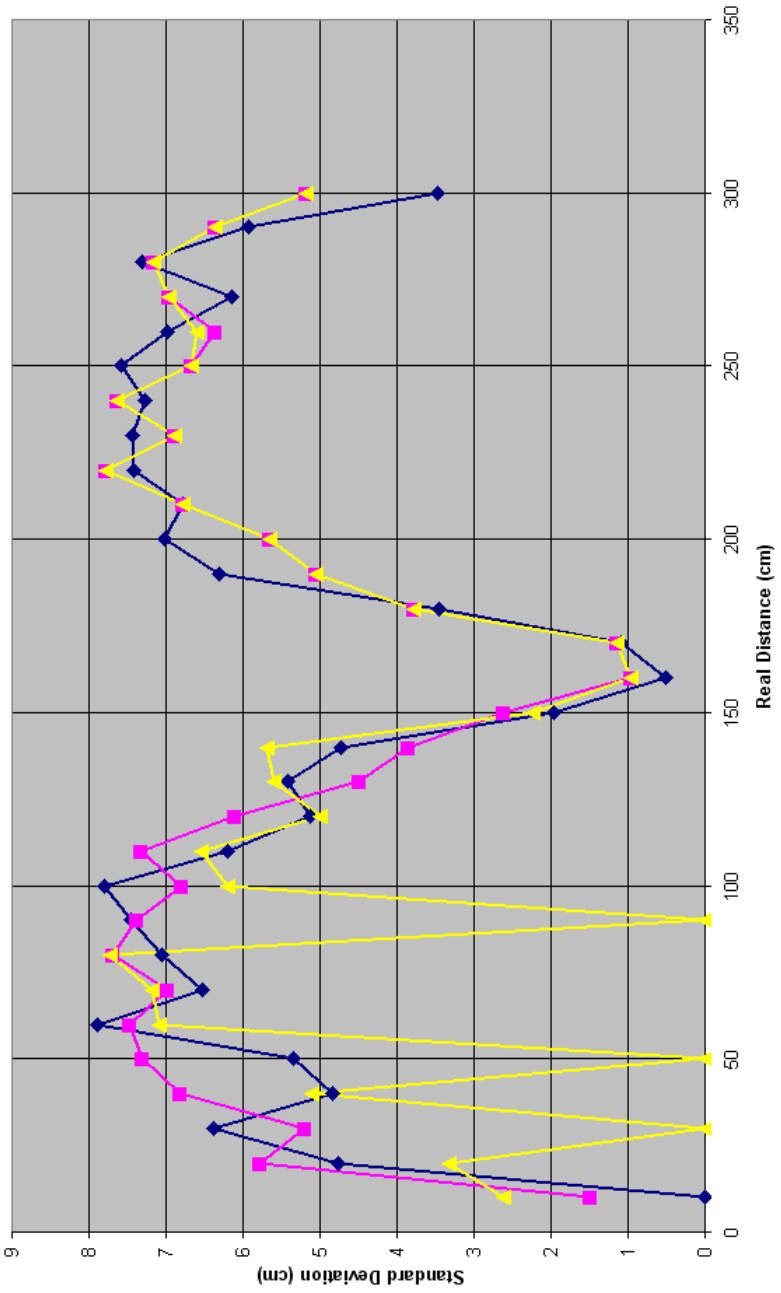


Figure 2.21: Standard deviations of 30 ultrasound range sensor measurements.

SRF04: object detection speed is limited by the sensor's time out of 46ms. Output is accurate enough for object detection from 0.1m up to 3.7m. A robot with such a sensor attached to its front measuring straight ahead would therefore be able to drive 3.6m in 46ms, i.e. 78.3 m/s. Despite from the fact that due to the Doppler effect the sensor would not be able to detect the echo of its own sound burst when the robot is driving at this speed, it is very likely that it is troubled by specular reflections. Moreover, a robot normally implements more than one ultrasound sensor thereby increasing the delay between consecutive measurements of the bumper sensor. Determining the maximum speed of a robot using one ore more ultrasound sensors depends on the amount of sensors used and the sensitivity of the sensor to reflections in a certain environment.

Chapter 3

Algorithms

3.1 Basics

3.1.1 Introduction

Map building research naturally evolved from research to robot localization, in which the robot has to find and track its location in an already available map. Localization is not the subject of this thesis. However, the algorithms used in localization have formed the basis for map building. The mapping algorithms are therefore understood better if the background about localization is known. In this paragraph, a very brief introduction is given.

Walking through a familiar room takes no intellectual effort at all for most people. Doing the same task blindfolded is a bit more difficult. By alternately taking steps and reaching out to find familiar objects however it is still possible to move through the room. By repeating this process of walking and 'measuring' one is able to keep track of the changing location. The problem of keeping track of your location is called *local positioning* in the field of mobile robotics. If a robot using the paradigm described here is equipped with range sensors, it should be able to maintain an estimation of its position at some degree of certainty. An algorithm implementing the paradigm is the Extended Kalman Filter (EKF) [8] for localization. The Kalman Filter however makes an important assumption: The initial position within a given map should be known for this solution to work. If the initial position is not known (*global positioning*), or if the robot loses track of its position due to an external influence (the so called *kidnapped robot problem*, see [11]), the EKF should be able to recover the right position by assuming an arbitrary position with a normally distributed uncertainty. The Kalman filter has several undesirable properties like a Gaussian noise assumption which caused another algorithm to enter the field of localization: The particle filter.

When a person is blindfolded and put in the middle of a familiar room without him or her knowing where exactly, that person -from his or her perspective- could be anywhere in the room. However, by repeating the walking and mea-

suring process once again, it is possible for the person to keep eliminating possibilities that are very unlikely, thereby eventually finding the location. Without an initial position it is necessary for a mobile robot to be able to keep multiple hypotheses about the current location; when it is, for example, switched on in a familiar room it needs to be able to 'think' it could be anywhere in the room and eliminate hypothesis untill it finds the right one. This is called *global positioning* [10]. With a particle filter this is possible. Solving the kidnapped robot problem is possible as well since it is actually almost the same as the global positioning problem; if the robot is tracking its location and time after time the sensor output seems too less correlated with the expected values, it has to assume the location is unknown again which brings the problem back to global positioning.

Although the global positioning is obviously harder than local positioning, the problem is practically solved under most circumstances. Localization however relies on one very strong assumption: an initially known map. Since this assumption is only rarely satisfied, an algorithm should actually be capable of building a map on its own. But before moving on to map building, the statistical framework of localization and mapping is given in the next few subsections.

3.1.2 Bayes

All discussed algorithms are in fact relying on one concept, the Bayes Filter, which in turn is based on Bayes' rule:

$$p(s|d) = \eta p(d|s)p(s) \tag{3.1}$$

where η is a normalizing constant that ensures the result of the equation is a valid probability distribution. This section will elaborate on the Bayes Filter. We will lean on the explanation from [11] but use the currently more common notation from e.g. [10].

A Bayes Filter is used for estimating the state of a *Partially Observable Markov Decision Process* (POMDP). Partially observable means some information about the world is available, though not enough to know the entire state. The classical example of a fully observable process is a game of chess. The world is of course the chessboard with all the pieces. A player can always see the position of the pieces on the board, in other words, the player always knows the state of the world. A pokergame however is only *partially* observable, since a cunning participant should be able to deduce at least some information about the distribution of cards in the game, but does not know the hands of his or her opponents with absolute certainty. Both the localization problem and the mapping problem are partially observable, and since in both cases we're estimating a state -localization means we're estimating position, during mapping position and the map itself are estimated- Bayes Filter based algorithms can be used.

We'll start by applying the Bayes Filter to the more intuitive problem of localization. Later on, it is shown how it forms the basis for map building algorithms. Localization addresses the problem of estimating your position given

(noisy) information about the traversed path and the measurements done during this movement. In formula:

$$p(s_t|z^t, u^{t-1}) \tag{3.2}$$

Here, s is the robot's pose; its position and rotation around the z-axis. z indicates measurement data and u is either a motor command or -when available- odometry data. Subscript $_t$ and superscript ^{t} are used here to indicate data at time t and all data up to and including time t, respectively. For example, u^{t-1} is a vector with chunks of movement data $\{u_{t-1}, u_{t-2}, \ldots, u_{t_0}\}$, whereas z^t holds the measurement data up to and including time t.

Now, the Bayes Filter is used to calculate the desired posterior $p(s_t|z^t, u^{t-1})$ from the previously gathered data by applying Bayes' rule. The resulting formula:

$$p(s_t|z^t, u^{t-1}) = \frac{p(z_t|s_t, u^{t-1}, z^{t-1})p(s_t|u^{t-1}, z^{t-1})}{p(z_t|u^{t-1}, z^{t-1})} \tag{3.3}$$

Since the divisor is constant with respect to s, and actually serves the same function as in equation 3.1, we replace is by η.

$$p(s_t|z^t, u^{t-1}) = \eta p(z_t|s_t, u^{t-1}, z^{t-1})p(s_t|u^{t-1}, z^{t-1}) \tag{3.4}$$

Now we expand $p(s_t|u^t, z^{t-1})$ by integrating over s_{t-1}, the state from the previous timestep.

$$\begin{aligned} &p(s_t|z^t, u^{t-1}) = \\ &\quad \eta p(z_t|s_t, u^{t-1}, z^{t-1}) \int p(s_t|s_{t-1}, u^{t-1}, z^{t-1}) p(s_{t-1}|u^{t-1}, z^{t-1}) ds_{t-1} \end{aligned} \tag{3.5}$$

Although this recursive update function enables us to calculate the posterior we want, using information that is available, it becomes more and more difficult. As the reader can see, according to this function *all* data up to time t has to remain available. This fact would render the Bayes Filter completely useless, if it wasn't for the Markov assumption. Bayes Filters assume the environment has the property of being *Markov*, which means that past and future data are conditionally independent given the current state. For our equation, it means the new posterior no longer depends on all data up to and including $t-1$, but only on the data from the previous timestep $t-1$. By applying the Markov assumption to the resulting equation, we end up with the desired Bayes Filter:

$$p(s_t|z^t, u^{t-1}) = \eta p(z_t|s_t) \int p(s_t|s_{t-1}, u_{t-1}) p(s_{t-1}|z^{t-1}, u^{t-2}) ds_{t-1} \tag{3.6}$$

The Bayes Filter forms the basis for several algorithms that are important to robotic localization and map building.

3.1.3 Particle Filtering

The result of equation 3.6 is a Probability Density Function. Since our state space is continuous, implementation is not straightforward; due to the continuity

there is an infinite number of possible poses, and computing the probability of being at the right location for an infinite number of locations is obviously not an option. Particle Filtering solves this issue by using a set of N samples to represent the posterior [7]. A sample is called a particle and represents the pose of the robot; a two dimensional coordinate and the heading of the robot. $\{s^i, w^i\}_{i=1,\ldots,N}$ denotes a random variable, where $\{s^i, i = 1, \ldots, N\}$ in combination with *importance factors* $\{w^i, i = 1, \ldots, N\}$ [11] (also called *weights* [7]) represent the posterior we're looking for. In formula:

$$p(s_t|z_t, u_{t-1}) \approx \sum_{i=1}^{N} w^i \delta(s_t - s_t^i) \tag{3.7}$$

For an accurate approximation, $\sum_{i=1}^{N} w^i = 1$ should hold, so the weights are normalized. Normalization is not a necessity for the Particle Filter, although it does keep the weights of the individual particles at a computable level[1]. Several versions of the particle filter exist, differing mainly in the way they resample particles or assign weights. The two most common filters are discussed here.

Sequential Importance Sampling The Sequential Importance Sampling (SIS) Particle Filter [7] is the most basic of all Particle Filters. It has some problems which are largely solved by the Sequential Importance Resampling (SIR) Particle Filter [7] which is discussed in the next paragraph.

To initialize the SIS filter, a predefined number of particles is drawn from a uniform distribution. To use an example from robot localization: The pose each particle represents gets random values for (x, y, θ) within the scope of the known map. Since nothing is known about the competence of each particle, all weights are initially set to $1/N$. During any other iteration but the first, new poses are sampled for the the set of particles that represents the posterior distribution.

All particles are sampled according to $p(s_t|s_{t-1}, u_{t-1})$, the *motion model* part of the right hand side of equation 3.6. In robot localization terms, every particle's pose (from time $t-1$) is updated to predict the actual pose of the robot at time t, using either movement commands or, when available, odometry information. In formula, what happens is:

$$s_t^i \overset{drawn from}{=} p(s_t|s_{i,t-1}, u_{t-1}) \tag{3.8}$$

Here, $s_{i,t-1}$ is the pose at time $t-1$ according to particle i. This step is repeated for every particle.

[1] When implementing a particle filter great care should be taken at all steps of the iteration process that variables containing information important to the filter -especially the particles' weights- are kept at values that are within the boundaries of the type of the variable. For example, the C++ data type *long double* holds a real number with a precision of approximately 18 digits. 18 digits may seem a lot, but when you're dealing with massive multiplications of real numbers > 0 and < 1 there's a good change you run out of precision!

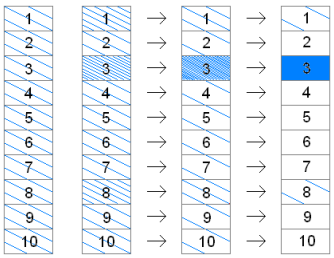


Figure 3.1: Sequential Importance Sampling. The amount of color represents the height of a particle's weight. The first column is the initial set of particles width equal weights. A few generations later, particle 3 has a weight of almost 1.

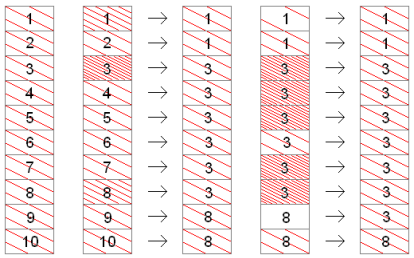


Figure 3.2: Sequential Importance Resampling. At each generation, new particles are drawn from the weighed set of particles of the previous timestep. The new particles' weights are reset. Due to the resampling particles diversity remains.

Next, the new weights of the particles are updated using the latest measurements, and the *measurement model*:

$$w_t^i \approx p(z_t|s_t^i) \tag{3.9}$$

In other words, for localization, for every particle the discrepancy between the measured value and the expected value, based on the predicted pose, is calculated and the resulting set of particles represents the new posterior.

Unfortunately, there's a problem with SIS, inherent to the sampling process. Variance of the importance factors can only increase over time [3] which results in what is commonly known as the *degeneracy problem*: After several iterations, one particle will have a weight associated to it that is close to 1, thereby outweighing the rest. Apart from the fact that most computational power is being put into useless samples, particle diversity is completely lost. This, obviously, is not good for our location estimator. Fortunately, *resampling* comes to the rescue.

Sequential Importance Resampling The Sequential Importance Resampling (SIR) algorithm works in a similar fashion as the SIS filter. But instead of using the same set of particles at every iteration, new particles are sampled from the posterior distribution. Particles with relatively high weights have a higher chance of being selected[2]. And as opposed to keeping the weights of their "parents", the "child particles'" weights are reset to $1/N$. Next, new poses are drawn using $p(s_t|s_{i,t-1}, u_{t-1})$, just as with the SIS filter. After this, also similar to what we've seen before, the weights of the new particles are adjusted using the measurement model $p(z_t|s_t^i)$. The difference between the SIS and SIR particle filter is visualised in figure 3.1 and 3.2.

[2]Drawing a parallel to the field of evolutionary algorithms, the posterior can be seen as the gene pool from which new children are selected

3.2 Map Building

3.2.1 Introduction

Building maps is all about uncertainty. In the localization examples we could see that uncertainty was constantly being reduced by comparing measurements to expected values given a known map. Since in this case we start out with an empty map we don't have any point of reference. An error in movement cannot be compensated by comparing measured and expected values.

Inherent to map building are several major issues. Since robotic mapping comprises a substantial part of this thesis, it's important for the reader to understand its difficulties. Below some of the major problems of mapping are explained.

Sensor Noise Dependency Statistically *independent* noise is normally easy to cancel out by taking a sufficient amount of measurements. Unfortunately, we are dealing with statistically *dependent* noise. During map building, all changes to the world model are related to movement, and movement contains error as well. Consequently, both errors in movement and errors in measurements accumulate over time. Even the slightest drift or slip can result in unrecoverable errors [14].

The most obvious way to cope with this problem is to try and reduce the chance of introducing motion error by using accurate motion sensors. However, this is not easy; wheel encoders, the most standard form of motion detection in mobile robots, are unable to compensate for slip since they only detect rotation of a wheel. However, as opposed to trying to cope with the problem of dependency of noise in a mechanical way, one can try to tackle it using a smart algorithm. Actually, tackling this problem is inherent to, for example, a particle filter based algorithm like DP-SLAM that keeps track of multiple possible robot paths. This enables us to assume noise independency. DP-SLAM is more elaborately discussed in chapter 4.

Data Association The problem that is currently known as the data association problem, is one of the most fundamental problems - or perhaps *the* most fundamental problem - of robotic localization and mapping. Associating a measurement with an object is a key feature of any localization and mapping algorithm. But even with an ideal, noise free range sensor, this association is far from trivial. Take for example a hypothetical infinitely accurate range sensor that tells us there's an object at a distance of exactly 40.0 cm. Then we move the range sensor 5.0 cm in a direction perpendicular to the sensor's beam, see figure 3.3 and 3.4. Again, our sensor returns 40.0 cm. The difficulty here is that the apparent correlation between these two measurements is not so apparent at all; the two measurements can be accounted for by several, quite distinct, explanations. If we assume there's no noise, we know at least that the two measurements are correct. What we don't know is whether they originate from

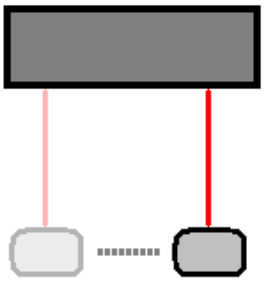


Figure 3.3: At time t and $t+1$ the sensor measures the same distance. Both measurements ought to be associated to one object.

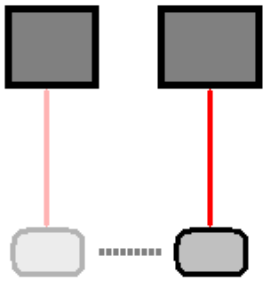


Figure 3.4: At time t and $t+1$ the sensor measures the same distance. Both measurements ought to be associated to two objects.

the same object which happens to be more than 5 cm wide (figure 3.3), or to two smaller distinct objects (figure 3.4).

If the Euclidian distance between the measured points is less than the desired resolution of the map, this is not a problem in practice. However, since the subject of interest of this thesis is cheap (i.e. *in*accurate) sensors, it does form a problem. Data association solely based on Euclidian distance is incorrect under noisy circumstances. As it happens, if the applied range sensor is somewhat noisy, the two measurements could very well belong to one single object that is only a few centimeters wide. Moreover, one (or two) measurement(s) might also belong to a dynamic object, for example a human being that is passing by. Matters are even worse, see chapter 2, for the ultrasound range sensor.

As the reader can see, data association is not as straightforward as it might seem at first glance. Measurements that seem to be correlated may have no relation at all, so every addition to the reconstructed map should be handled with care. Different mapping algorithms deal with these problems in different ways, some implicitly, some explicitly.

Circular Environments A circular environment is an environment in which you can leave in one direction from a certain location and return to that location from another direction. The reason that a robot should have an accurate motion model is that it should know, using only motion data, *whether* and if so *when* it has returned to a location it has already been before. This is where sensor noise dependency can really manifest itself. In localization, one can compensate for motion errors using measurements from sensors. In map building however, you are not only *not* compensating for noise, you're actually relying on motion to be accurate; its own position is the only point of reference the robot has. When the robot for example has a consistent deviation to the right that has not been accounted for in the motion model, and it drives through a rectangular

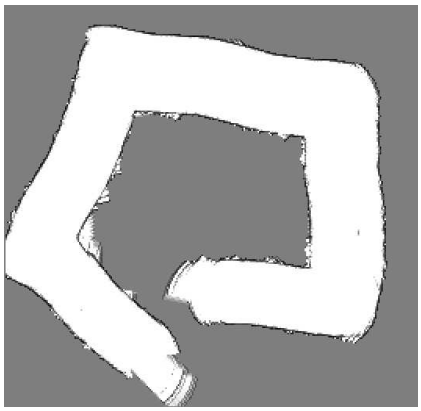


Figure 3.5: Motion error can lead to distorted maps. This is an example from the simulator. The robot moved through a circular hallway and eventually returned to its initial position. Due to motion errors however the robot is unable to connect its initial position in the map to the final position, thereby creating two loose ends.

corner clockwise, it will map the corner with a somewhat sloper angle (assuming zero-error measurements). If this continues for every one of the four corners in a circular hallway, the robot will not know that it has reached its initial position when it does. This deviation is of course clearly visible in the resulting map; The robot will not close the loop but continue mapping what it thinks is a new hallway, see figure 3.5. As with the data association problem, different algorithms deal with circular environments in different ways or, in some cases, not at all.

This subject is less relevant for *local* mapping due to the fact that for mapping at short distances, motion error can be neglected. Moreover, when a global map -e.g. outline of borders and walls- is already known, it is theoretically possible to correct for motion errors using measurements. This issue is mentioned since it is a prominent subject in global mapping literature. See for example [14] [1].

Dynamical Environments Most algorithms are having enough trouble already with static environments, so in most cases it is assumed that environments are always static, i.e. the environment does not change over time. Moving objects like people are not taken into account. Even chairs and tables, which are potentially not static are mapped as being constant over time. Some algorithms can model partially dynamic objects like doors to a certain extent like algorithms based on the Kalman filter and algorithms based on occupancy grids. The former uses a covariance matrix to represent correlation between measured objects. Due to the Gaussian noise model that is inherent to the Kalman fil-

ter it is possible to model slow migration of objects [10]. Fast moving objects however cannot be modeled by such an algorithm. About the same holds for occupancy grid map based algorithms; slow changes can be represented by changing the assignment of grid cells, but representing fast movement is out of the question. Montemerlo, Thrun and Whittaker [5] have made an attempt to combine robotic localization with people tracking using a so-called conditional particle filter. Combining SLAM and detection and tracking of moving objects (DATMO) has been studied by [15] in an outdoor environment. The results of the experiments look promising. However, the sensor they used to get their outstanding results is again a SICK laser range finder (or actually, several laser range finders) which is able to accurately measure a distance up to 80 meters.

3.2.2 FastSLAM

Kalman filters are algorithms widely used in map building. This very well known filtering technique is used in a variety of research fields [8], among which is robotic self localization. Although Kalman filtering is not used in the experiments for this thesis, it is an important part of the FastSLAM algorithm [4] which in turn is an important advancement in map building.

A Kalman filter is a state estimator, and in robotic localization, the state to be estimated is of course the location of the robot. Given an initial position, the robot tries to compensate for motion noise using measurement data. Based on motor commands or odometry measurements, it predicts its next position and with measurements it compensates for possible motion error. In mapping, the robot should not only estimate its location within a known world, but the world itself as well. Kalman filter based mapping algorithms assume that the environment can be represented by landmarks. Every time the robot encounters an unidentified object it stores its position in memory. Uncertainty about the map i.e. the positions of the landmarks and the robot's location in it is represented by a matrix that holds the covariances between the estimated locations that are represented by normal distributions, the so called covariance matrix. More about the Kalman filter for localization can be read in [8]. This way of using the Kalman filter has two problems. First, it is sensitive to failures in data association (see section 3.2.1), which is the process of assigning a measurement to a certain object. Second, complexity (and thereby computational costs) increase quadratically with the number of landmarks, due to the covariance matrix that holds the pose of the robot and the position of all landmarks. In 2002, Montemerlo et al. [4] designed an online algorithm for map building, that combined the power of particle filters and Kalman filters.

In subsection 3.1.3 we saw that a particle represents x and y position, as well as the angle of the robot's heading. In FastSLAM, a particle consists of a hypothesis for the robot pose and Gaussians representing the coordinates of every landmark that is observed [4]. Thus, every landmark is a separate Kalman filter. FastSLAM therefore solves at least one problem inherent to any Kalman filtering technique: Quadratic complexity no longer exists since covariance (and thereby the covariance matrix) is no longer used. It is assumed here that fewer

particles are needed to accurately build the map and keep track of the location than landmarks are needed to represent the environment, since an increase in the number of particles has an immediate impact on computational complexity. Data association is done by every particle separately and possible association errors are (hopefully) filtered out. Thrun and his colleagues have improved their algorithm, which they now call FastSLAM 2.0, to use less particles in certain situations. One disadvantage that is inherent to (any) FastSLAM is the use of distinguished landmarks. FastSLAM produces satisfying results [4] for a number of outdoor environments where it is able to accurately map the locations of e.g. blocks (indoor) or trees (outdoor). Indoor environments however do not contain either trees or little blocks, but walls and cabinets. FastSLAM evidently works for environments that can be easily described by a set of landmarks. The legs of tables and chairs in a domestic environment are expected to be mapped well with FastSLAM. However, walls, cupboards or couches for example are not readily converted to landmarks. A large object could be described by a series of landmarks, but doing this using a wide beam sensor like the ultrasound sensor used in this research project -as opposed to the commonly used laser range finder- does not seem wise. That is why DP-SLAM was elected to be implemented; This algorithm is based on a particle filter as well, but uses a grid-like approach to incorporate sensor information, which seemed a better methodology to use in combination with the ultrasonic sensor.

3.2.3 DP-SLAM

The holy grail of map building of course is being able to search, online, in the infinite space of all possible locations in every possible map. In localization, the (still infinitely large) search space is all possible locations given one known map. In a particle filter, a finite set of particles is used to represent a search space. The more particles used, and the more complex they are, the more computational power is needed to meet the speed requirements. Since with the problem of localization a particle only represents a pose, a filter using several thousands of particles[3] can still work in real time. With map building however, a particle is composed not only of a position vector but needs to represent an entire map. If we assume that a map is represented by a series of landmarks, computational costs are therefore relatively low when the number of landmarks is low. A set of landmarks however is not always sufficient, as mentioned in section 3.2.2. The most general map representation would be a discrete map. A particle using this kind of map would need to store a two dimensional array. A naive implementation of a such a particle filter would require too much computation for online mapping using a standard desktop PC or notebook.

In 2003, Eliazar & Parr [1] developed an algorithm they called *distributed particle SLAM* (DP-SLAM). The idea of DP-SLAM is to use an alternative methodology to handle large discrete maps, in order to speed up calculations and reduce the amount of memory needed to store all the particles. For local

[3] a typical number for a global localization particle filter

map building, the subject of interest of this thesis, there is less need to handle large maps. However, efficiently handling a lot of particles *is* important in this case since we're dealing with noisy sensors; DP-SLAM is designed for use with a laser range finder, and relies on its accuracy. To use this algorithm for local mapping, it needs to be adjusted for use with smaller maps but more noise.

Chapter 4

DP-SLAM

4.1 Overview

DP-SLAM uses several data structures that are tightly connected in a way that has not -to the author's best knowledge- been used before in robotic mapping. An overview of the algorithm is presented below. In the next sections, the data structures are discussed separately.

DP-SLAM uses an advanced occupancy grid to store particle additions. Instead of associating a grid with every particle, particles are associated to a single grid by adding an identification number (ID) to a certain particle whenever it is embedding measurements. The difference between these two methods concerning particle ID storage is illustrated in figure 4.1 and figure 4.2. To prevent the grid from growing infinitely large over time, Eliazar & Parr devised a strategy that works by keeping the ID of every particle that updates the grid in a separate tree, called the *ancestry tree*.

These two data structures, the complex occupancy grid and the ancestry tree, are discussed more elaborately in the next sections. Subsequently, it is explained how measurements are processed by DP-SLAM and how the algorithm -which is originally designed for use with a laser range finder- can be converted for use with ultrasound sensors.

4.2 Occupancy Grid

A conventional occupancy grid is a two dimensional array of either integers or floating point values. These values indicate the probability of a location, i.e. a grid cell, being full or empty. A measurement is integrated into the grid by updating the values at the grid cells that the 'sensor cast' presumably went through. From this short summary it is easily seen that pose uncertainty is not taken into account. Also, it is not possible to recover from spurious measurements since information about separate measurements is only stored implicitly in the grid cell values. The occupancy grid that DP-SLAM uses is

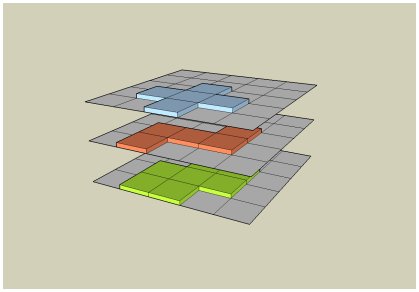


Figure 4.1: Implementation in which a particle holds a complete grid. This is a burden on memory usage as well as on computational costs due to the need to copy information during the resampling process.

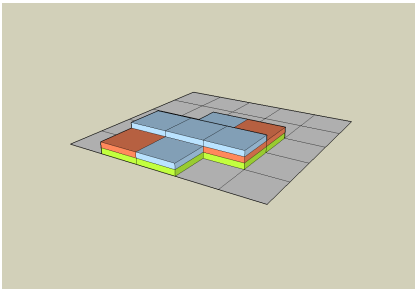


Figure 4.2: Occupancy grid using DP-SLAM. There's only one grid, in which all updates of all particles are stored.

able to keep track of distinctive measurements, thereby solving both issues just mentioned.

With DP-SLAM, every grid cell is actually a balanced tree of nodes holding the ID of a particle that updated this cell (see figure 4.3), and two values keeping track of how many times a lasercast[1] has passed through or stopped in this cell for this particular particle. The reason why this is a balanced tree, and not just e.g. a simple list, is that it is possible to find a node in such a tree in logarithmic time. The set of particles that is responsible for grid additions at a certain time t, will generate a new set of particles that update the grid at time $t + 1$. The ID's of the new particles are added to the grid in the same way as before; a new node is added to the tree of a certain grid cell for every particle that updates it.

Without further processing however the grid would eventually grow infinitely large. Consequently, there has to be a mechanism that deletes nodes as well. Preferably, nodes that are least likely to represent correct information should be deleted. How the weighting is done exactly is explained further on in section 4.4. Roughly it can be said that the weights, representing the probability of a particle's map being correct, need (a part of) this map to be calculated. This yields a need to (partially) reconstruct a map. DP-SLAM uses the information about a particle's ancestors, stored in the ancestry tree, to do exactly this.

4.3 Ancestry Tree

In the previous section it is explained that the unique ID's of particles that ever updated the grid are explicitly stored in the grid. This means we can recover the entire grid belonging to a particle by tracing the particle's lineage and checking for every grid cell whether the particle or one of its ancestors has updated the grid cell. To be able to trace all the ancestors for all particles DP-SLAM uses the *ancestry tree*. The ancestry tree is used to keep track of which particle

[1] Eliazar & Parr were using a laser range finder for their experiments.

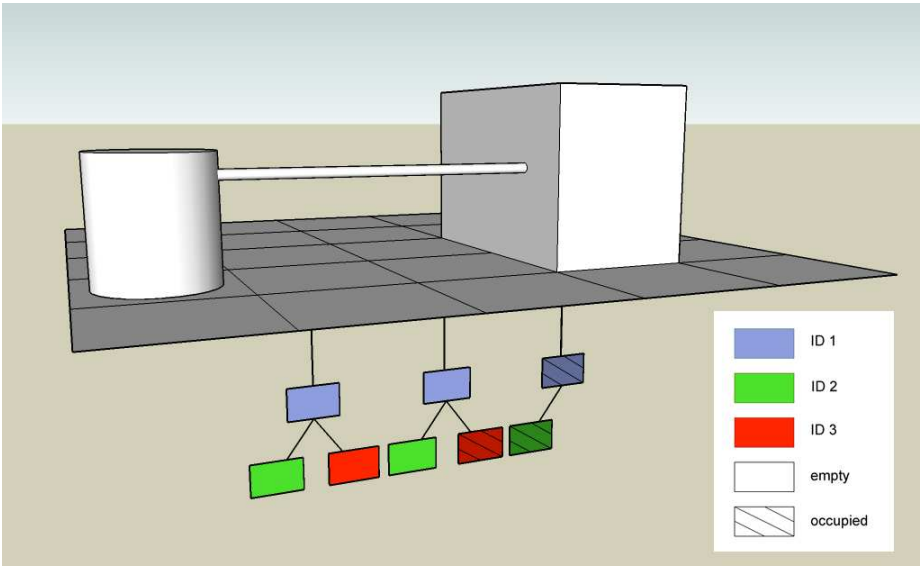


Figure 4.3: A measurement is added to the grid. There are three particles, two of which integrate the measurement as reaching as far as square 3. The third particle however assumes the laser cast was stopped at square 2.

'descended' from which particle by storing a particle and its children as nodes and subnodes. Eliazar & Parr [1] show that the tree can maintain a bounded size by pruning away unnecessary nodes. There are two situations in which a node is seen as unnecessary. First, when a node has no children it can obviously be removed, with the exception of nodes that are added last, i.e. nodes bearing particle ID's. If this means this node's parent is left without children the parent can be removed as well and so on. All nodes in the grid bearing the same ID as the nodes deleted from the ancestry tree are deleted from the grid. The second situation in which a node is unnecessary is when it has only one child. The parent in such a case is redundant and can be merged with its only child. In the ancestry tree, the child's ID is deleted and its subnodes, if present, become the parent's direct subnodes. The grid is updated by changing the ID of all nodes bearing the child's ID to that of the parent. Basically, by doing this the child's set of updated squares is incorporated in the parent's set. If there is a conflict, i.e. both the child and the parent made an update to a certain square, the parent's update is replaced with the child's.

Eliazar & Parr [1] state that independent of the number of iterations, a *minimal* ancestry tree is obtained after pruning, which has three useful properties: The tree

1. has exactly N leaves
2. has branching factor of at least 2, and
3. has depth no more than N

where N stands for the number of particles. From proposition 1 and 2 we also know that the maximum number of nodes in the tree is $N-1$ which in turn is equal to the maximum number of nodes in every cell in the occupancy grid.

4.4 Weighing the particles

Since the tasks of calculating the weight of the particles in DP-SLAM and incorporating the particle ID's into the grid are so tightly connected, they are performed simultaneously in practice. Therefore, they are discussed simultaneously in the next section.

DP-SLAM was originally designed for use with the SICK laserscanner. Measurement integration into the grid is therefore based on the "model" of a single laser cast, which is basically a single straight line from the origin of the sensor to the endpoint. From the chapters about sensors it is clear that the beam of an ultrasound sensor cannot be represented in the same way. The particle weighing mechanism therefore needs to be adjusted to handle ultrasound measurements. In section 4.4.1 the original lasercast method is explained, in 4.4.2 the conversion is made to ultrasound measurements.

4.4.1 Laser

When the robot is assumed to be at location (x, y, θ) by some particle, and the laserscanner returns a certain distance d at an angle a, the particle puts this measurement into the grid by updating all gridcells that are crossed by a straight line between $[x_a; y_a]$ and $[\hat{x}_a; \hat{y}_a]$, where $\hat{x}_a$ and $\hat{y}_a$ are $x_a + d\cos(\theta + a)$ and $y_a + d\sin(\theta + a)$ respectively. In figure 4.4 and figure 4.5 two examples are given of such a measurement. At every gridcell in the iteration, a new node is inserted in the tree at that point containing the particle ID along with two variables: the distance that is travelled through the square by the lasercast and a boolean variable keeping track of whether the laser has stopped in that square or not. Why the second variable is boolean is obvious. However, it might need some explaining why the first one isn't. In figure 4.4 it is clear that the laser traverses an equal distance through every square. Please note however that in figure 4.5 the lasercast goes straight through some of the squares but bearly touches others. In this case it is clearly desirable to distinguish between the two lasercasts.

By tracing back the ancestry, it is possible to produce a single map for each particle. Eliazar and Parr [1] use the accumulated distance travelled through the square by the lasercasts originating from the particle and all its ancestors d_τ and the number of times the particle or one of its ancestors had a lasercast that stopped h in that gridcell to calculate ρ, the *opacity* of the cell. Formally, $\rho = d_\tau / h$.

The opacity of a gridsquare is used in the construction of a map for the end user, as well as in the assignment of weights of particles. In the former case, it

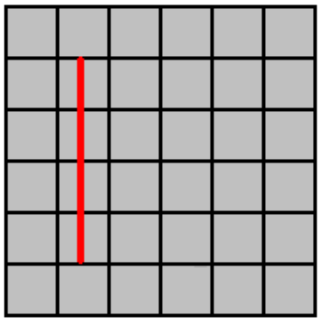


Figure 4.4: The laser travels an equal distance through all grid squares.

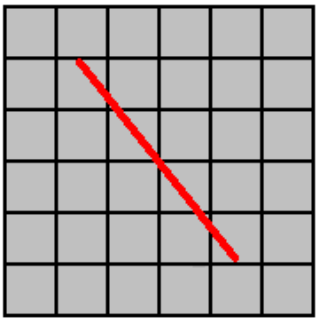


Figure 4.5: In this case the laser traverses some grid squares from corner to corner, while other grid squares are nearly touched.

appears useful to incorporate several 'tweaking variables' to get a clearer map. In the latter case, ρ is used in the following way:

$$P_c(x, \rho) = 1 - e^{-x/\rho} \tag{4.1}$$

Where $P_c(x, \rho)$ is the cumulative probability that the laser will have been interrupted after travelling distance x through a medium with opacity ρ. In the original DP-SLAM (2.0) article [1] several arguments are given for this formula. The total probability of the measurement is:

$$\sum_{i=1}^{n} P_L(\delta_i | stop = i) P(stop = i) \tag{4.2}$$

where $P(stop = i)$ is the probability that the lasercast is interrupted exactly at square i. The vector δ is a set of (normally distributed) distances to the endpoint of the lasercast. As Eliazar and Parr put it: "The probability of the measurement is then the sum, over all grid squares in the range of the laser, of the product of the conditional probability of the measurement given that the beam has stopped, and the probability that the beam stopped in each square.".

4.4.2 Conversion to Ultrasound

The robot used in the experiments for this thesis uses infrared and ultrasound sensors. Concerning measurement integration and particle weighting, infrared sensors can be treated more or less in the same way as the laserscanner since the shape of an infrared beam can be approximated by a straight line, as is the case for a laser. Ultrasound sensors however work differently (see chapter 2). The sensor can be modelled as a set of lines with the length of the measurement originating from the sensor, like a fan shape. From chapter 2 it is clear that

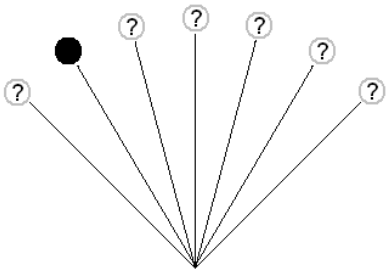


Figure 4.6: A single ultrasound measurement gives little information about the position of the measured object.

the fan shape is a correct approximation. However, this model needs some adjustments to enable the calculation of the probability $P_c(x, \rho)$ in the way we did before.

If an ultrasound measurement returns distance l we know that all lines should have length l since that is the distance to the nearest object and nothing is known yet about the space behind the nearest object. We also know that the endpoint of *one* of the lines represents (part of the outline of) an object. What we do not know is *which* line this is, see figure 4.6. During the determination of $P_c(x, \rho)$ it is therefore incorrect to calculate $P_L(\delta_i|stop = i)$ for all but one line. The method used in the experiments to cope with this problem determines $P_c(x, \rho)$ for every line despite the fact that this is incorrect in most cases. Next, the line with the highest probability is selected and given the benefit of the doubt; The endpoint of this line is assumed to represent the location of the measured object and is therefore assigned a 'full' tag in the grid. In the experiments, particle weights are assigned $P_c(x, \rho)$ of this line.

Chapter 5

Robot

5.1 Introduction

To test the SLAM algorithms not only in simulation but in real life as well, a custom built robot was used. Or rather, a custom adapted robot. The robot in question is a several years old prototype of an autonomous vacuum cleaner. The base frame, the wheels, and the DC motors with the gearboxes were recycled. Its electronic guts were removed completely, with the exception of the motor controller PCBs. Communication with these motor controllers is now done by a microcontroller. Six infrared and six ultrasound sensors are attached to the robot, and to interface them a second microcontroller is used. The microcontroller is constantly polling the sensors in order to keep the sensor data up to date. Both microcontrollers are connected to the laptop present on top of the robot. The software on the laptop running the DP-SLAM algorithm implements drivers to communicate with the microcontrollers in order to control the robot's movements and to obtain sensor data. Constant communication between the controller and the laptop ensures a tight connection between hardware and software. This software was written in Visual C++, as a partial fulfilment of the graduation assignment. A graphical user interface is provided to control both the physical robot and the simulator.

More detailed specifications about the individual components i.e. the motors, the motor controllers, the microcontrollers, and all the sensors can be found in the Appendix. In the next sections, the interconnectivity between these components is discussed, together with a description about how the connections on the robot were established.

5.2 Hardware - Platform

Figure 5.1 shows a photograph of our robot. As one can see, it has three relatively large (16cm diameter) omni-directional wheels (or omni-wheels for short). This typical kind of wheel has full grip perpendicular to the drive axis just as

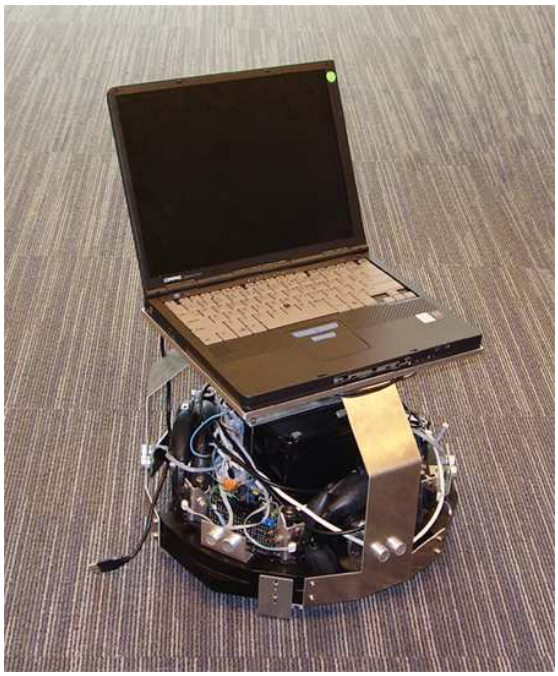


Figure 5.1: Photograph of the robot.

any other wheel, but unlike a normal wheel, due to small wheels distributed over the entire rim of the omni-wheel, it has almost no friction parallel to the drive axis. Attaching 3 wheels to a robot in the way shown below, results in a robot that is able to translate and rotate in every possible direction without having to turn, thereby minimizing the possibility of introducing location uncertainty by sticking behind an obstacle, as is the case with a robot using 4 normal wheels, or 2 normal wheels and a 3rd caster wheel which are both popular configurations.

By placing the motor controllers (figure 5.2) between the wheels alongside the rim, a space is formed on the inside of the robot. In this space both the low level part of the robot's brain and the battery that powers it reside. There are two PCBs: one for all the sensors and one for motor control. They are stacked on top of each other, and this way they are attached vertically to the frame (figure 5.3), parallel to the 12 Volt led battery. Attached to the cart is a metal frame (figure 5.4) to support the higher level brains of the robot, i.e. the laptop running the software. Because of the way the frame is constructed it is possible to drive around with the robot while keeping the screen of the laptop open. A USB hub is present to provide the necessary USB ports. The hub is powered by the DC/DC converter on one of the PCBs.

Our robot wouldn't be useful if it didn't contain lots of sensors. Hence, 6 infrared (figure 5.6) and 6 ultrasonic (figure 5.7) sensors are permanently attached to the base. That is, the ultrasonic sensors are distributed over the

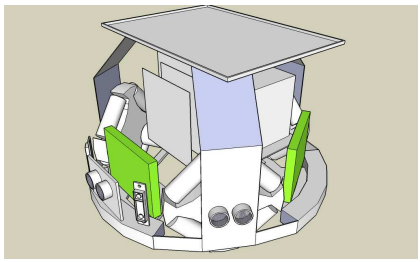


Figure 5.2: The motor controllers.

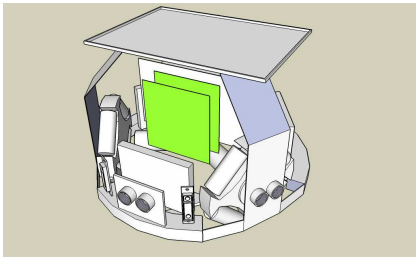


Figure 5.3: Circuit boards

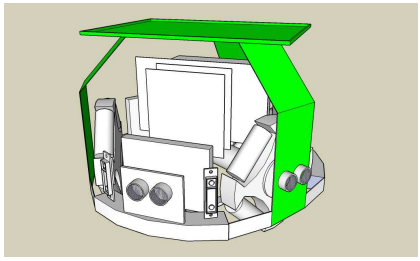


Figure 5.4: The frame.

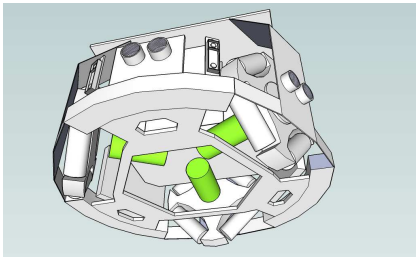


Figure 5.5: Motors and tachometers.

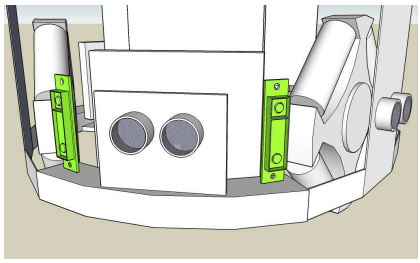


Figure 5.6: Infrared sensors.

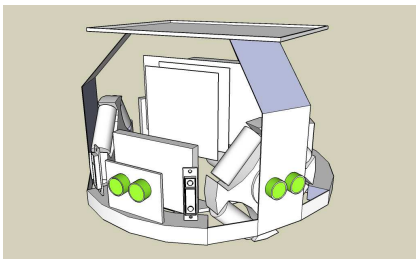


Figure 5.7: Ultrasound sensors.

rim at an equal distance from each other, all 'pointing' away from the center of the robot. Almost the same holds for the placement of the infrared sensors. Due to the position of the large omni-wheels however, the infrared sensors are placed somewhat more pairwise.

In fact, there are 3 other sensors, which are somewhat less relevant but should be mentioned anyway for the sake of consistency. All 3 motors get their setpoints from the motor controllers. Every motor has a tachometer (figure 5.5) attached to it, of which the output is fed back to the motor controller.

5.3 Hardware - Electronics

5.3.1 Microcontroller

The interface between the software part and the hardware part is formed by the DLP245PB microcontroller. Actually, two of these controllers are used; one mainly for motor control, the other for sensor control. The main components of the controller are the PIC16F877, a chip with multiple I/O ports including 6 A/D inputs, and an FTDI chip that is able to handle USB communication. This is a powerful combination, since we now have access to very elementary sensor information and control mechanisms by using high level USB communication. More specifically, communication is done by means of exchanging *unsigned char*s using the software libraries belonging to the DLP245PB. A DLP245 USB microcontroller can be powered by USB as well as by an external power source. In the former case it is adviced to keep in mind the fact that only 500mA can be drawn from a USB port.

5.3.2 Motors and Motor controllers

The omni-wheels are driven by 12 volt Maxon DC motors, which in turn are controlled by Elmo motor controllers. Attached to the motors are tachometers. Tachometer output is used by the control loop on the controllers, which are therefore capable of adjusting the rotational speed of the motors. To do this, the motor controllers use an input voltage of -10.0V to 10.0V (provided by converting the voltage from the led battery). For dynamical and fast adjustment of the voltage setpoints, we a use digital potentiometer, the AD7376 by Analog Devices. The potentiometer provides an SPI communication interface through which it is able to communicate with the microcontroller. The potentiometer is 7 bits, meaning that the 20.0V domain is divided in 128 steps, which is more than enough for our application. These electronic features are fitted onto one circuit board. In summary, the board holds several DC/DC converters providing the right voltages for the setpoints and power for the AD7376s and the microcontroller. These potentiometers provide the motor controllers with setpoints that can be set by the microcontroller (which in turn gets its commands from the software).

5.3.3 Sensor Polling

As the reader can see in chapter 2 infrared sensors and sonar sensors, albeit it in a different manner, both need polling. The infrared sensors have an analogue output that is polled every 8ms. The ultrasonic sensors work in an entirely different way, but need to be polled as well. More about the workings of both sensors can be found in chapter 2.

In short, infrared sensor polling is as straightforward as can be; every 8ms the current sensor output (a voltage level between 0V and 2.5V)is stored in a variable. The output value can be read by a single A/D port on the microcontroller. Internally, the IR sensor updates its value every 40 milliseconds, but to cancel out some electrical and measurement noise it's better to perform a readout more often and use the mean or median of several values. Experiments pointed out that the mean of 3 measurements is already enough to cancel out the worst noise. Since the sensors are polled every 8 ms, the delay is never more than 24 ms.

The SRF04 ultrasonic sensor works quite differently from the infrared sensor. As the reader can see in chapter 2, one needs to send a trigger signal on one line, and receive an echo signal on another. In other words, we need two I/O pins per ultrasonic sensor on the microcontroller. We are using 6 ultrasound sensors, so in our case, this means that we need a total of 12 I/O pins. This is the main reason for using multiple microcontrollers; The infrared sensors use the 6 A/D pins so together with the ultrasound sensors they occupy all available I/O of one single microcontroller which leaves no more room for the motor control hardware.

Triggering is straightforward; The trigger pin on the sensor needs a short pulse, which the microcontroller can provide, to get started. After the pulse is given to the trigger line, we have to wait for the echo of the sound burst to return on the echo line. The best way to handle this would be using interrupt-on-change pins on the PIC. The PIC16F877 unfortunately does not provide an interrupt-on-change on all ports. In fact, the only port that *does* have this feature is already in use; the FTDI chip is using it to communicate with the PIC. Therefore, we need to poll the sensor. In our case, polling happens every $100\mu s$. The DLP microcontroller would have had the ability to use a different timer(interrupt) for both the ultrasound and infrared sensors, if it hadn't been for the fact that two I/O pins are disabled when the second timer is enabled. Since we need all 18 pins for our 12 sensors, using the second timer is not an option. However, this issue can be solved easily by using a counter in the ultrasound timer; 80 times $100\mu s$ equals 8ms, so by increasing a variable in the $100\mu s$ interrupt function that calls the actual infrared timer function everytime it reaches 80, it is possible to create a virtual second timer.

5.4 Software - PIC

On the DLP245PBs software is running to simultaneously gather information from the infrared sensors and the ultrasonic sensors, and at the same time communicate with the software to exchange information with the higher level software without interfering with the sensor readouts. Timer interrupts are used to constantly poll both kinds of sensors. It's already mentioned that control differs per sensor, but one principle remains the same: every fixed number of microseconds new sensor values are written to the PIC's limited memory. The newest values overwrite older values. The reader might want to take a look at the pseudocode algorithm [2]. When the software on the laptop needs measurement data of a certain sensor, it sends a request to the microcontroller, which in turn returns the contents of the associated memory. This way, communication can take place any time, sensors can continuously gather information and higher level software is sure to always get the latest sensor readings. Please see algorithm [1] for an explanation with pseudo code. The same principle holds on the microcontroller that runs the software that can adjust the setpoints for the motorcontrollers, see algorithm [3].

Algorithm 1: Sensor PIC - Main Loop

```
while ( true ) do
    if
    ( usb_data_available == true )&&
    ( retrieved_data[0] == sensor_request ) then
        switch ( retrieved_data[1] ) do
            case 0x01:  SendDataToLaptop( latest_ir_data )
            case 0x02:  SendDataToLaptop( latest_us_data )
        end
end
```

5.5 Software - Laptop

Attached on top of this collection of hardware is a laptop; an old 600Mhz Pentium III. Unfortunately, it is too slow to handle the computationally expensive algorithms that have to deal with the enormous amount of uncertainty. For that is its original function; the laptop should run the software that uses the information gathered by the robot to build a map in real time. In practice however, the laptop is not fast enough to do both the data acquisition and processing. Therefore, the laptop is used for gathering data after which it is processed on a faster desktop PC - a 2.8Ghz Pentium IV. The desktop is running the same software; the difference is that this software is compiled to process logged data.

Part of the graduation assignment was writing a simulator in C++ for mimicking the behavior of the sensors attached to the robot. Eventually, the results

Algorithm 2: Sensor PIC - Interrupt

```
BYTE counter = 1
BYTE us_polling_stage = 1

OnInterrupt()
    // IR sensor polling
    if ( counter == 80 ) then
        latest_ir_data = AD_ADR
        counter = 1
    else
        counter++

    // US sensor polling
    switch ( us_polling_stage ) do
        case 1:
            us_polling_stage++
            SetTriggerPinHigh()
        end
        case 2:
            us_times_polled = 0
            us_polling_stage++
            SetTriggerPinLow()
        end
        case 3 :
            if ( GetEchoPin() == high ) then
                us_times_polled++
            else
                latest_us_data = us_times_polled
                us_polling_stage = 1
        end
    end
end
```

Algorithm 3: Motor PIC - Main Loop

```
while ( true ) do
    if ( usb_data_available ) then
        switch ( retrieved_data[0] ) do
            case 0x01:
                SetPotentiometer( retrieved_data[1], retrieved_data[2] )
                SetPotentiometer( retrieved_data[3], retrieved_data[4] )
                SetPotentiometer( retrieved_data[5], retrieved_data[6] )
            end
            case 0x02:
                SetPotentiometer( 1, 0x40 )
                SetPotentiometer( 2, 0x40 )
                SetPotentiometer( 3, 0x40 )
            end
        end
end
```

should be compared to real data to gain better understanding of the deficiencies of the sensory (with a focus on the ultrasound sensors). It was decided that the best way to compare results is to use exactly the same software for both simulated and real data. In the software, the several types of I/O can be switched on and off. By setting a simple parameter the program can be compiled either as a simulator, taking only simulated data as its input, or as a true robot control center, thereby being able to control the robot and to use real data to work with.

Actually, a third option exists. When the robot does a run through the assault course (see chapter 6), the motor commands and all the gathered sensor data is stored in a data structure that can be saved to a file afterwards. By toggling another parameter, connection to the real hardware is cut off and bypassed to obtain data from this file. An apparent advantage of logging and reusing sensor data is the ability to reproduce real sensor noise. This way, it is possible to unleash various parameter settings onto the same data, which of course is a good way to benchmark the arsenal since the same can be done in simulation.

Figure 5.5 visualizes - in an extremely simplified way - what has just been explained; As far as the software is concerned, the only difference between simulated data, logged data and real data is the bottom layer[1] which can be selected by switching the right flags on or off.

[1] In theory. In practice, things are a little bit more complicated since e.g. sometimes it's convenient to let the class that provides the simulated data access parts of the graphical user interface.

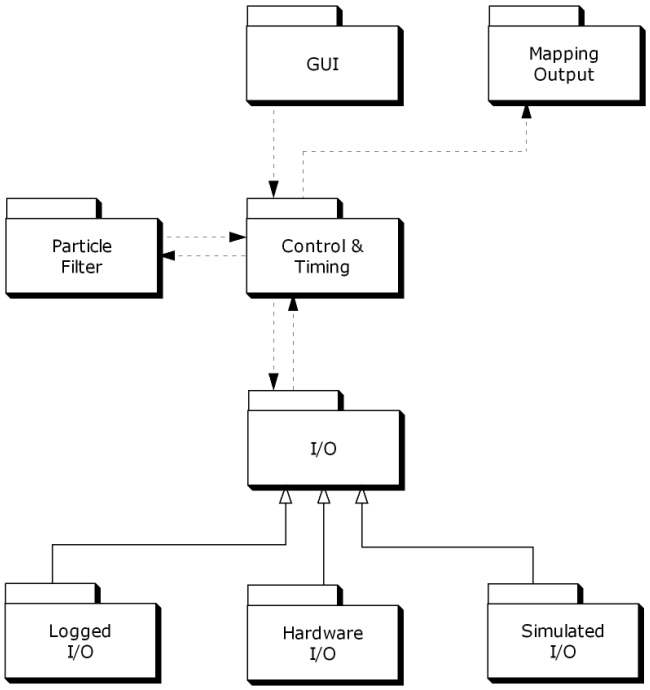


Figure 5.8: UML Class Diagram. Packages are used to present a general view of the software that runs on the laptop.

Chapter 6

Results

Both the infrared and ultrasound sensors are tested in isolation, see chapter 2. In chapter 5 the robot used in the experiments was described and in chapter 4 the algorithm running on the robot is explained. This chapter shows the results of the integration of these matters.

There are several kinds of results we're intersted in: How well do the sensors perform in a complex environment? Can the DP-SLAM style integration of ultrasound and infrared measurements into the grid create a (local) map that is good enough for the robot to avoid objects with? And is DP-SLAM able to compensate for motion error thereby facilitating better interaction with the global map? The first two questions are answered together in the next section. In the section after, DP-SLAM benchmarks are discussed to see how well the algorithm performs.

6.1 Assault Course

Experiments were performed using the robot on an 'assault course'; a representative environment to test its ability to detect (and even map the outlines of) potentially hard-to-detect obstacles. The testing environment was accurately measured by hand and this information was used to construct a 2D map for the simulator and a global map for the robot. Noise free simulation output is compared to the real life output to visualize the theoretical possibilities and practical limitations.

Figure 6.1 and figure 6.2 are the 2D and 3D representations of the real life testing environment. The three similar rectangles are the projections of cabinets in an office space, with a width and length of 45cm and 240cm respectively, and a height of about 1.5m. The large square at the top right forms the outlines of a wall. The distance between the bottom most cabinets is 1.70m. There are three obstacles: A round metal bin with a diameter of 26cm next to the bottom right cabinet, a rectangular metal bin measuring 22cm x 30cm next to the wall, and finally a hatstand of 2.5cm in diameter. This hatstand, which is basically

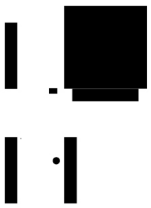


Figure 6.1: The assault course, 2D.

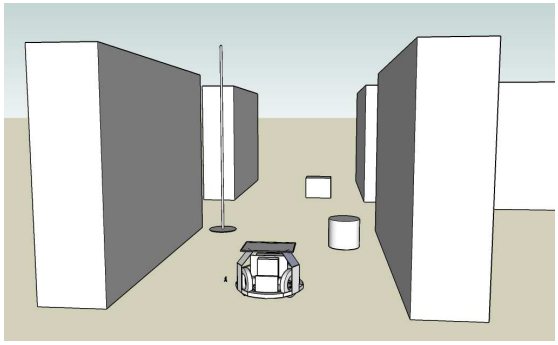


Figure 6.2: The assault course, 3D.

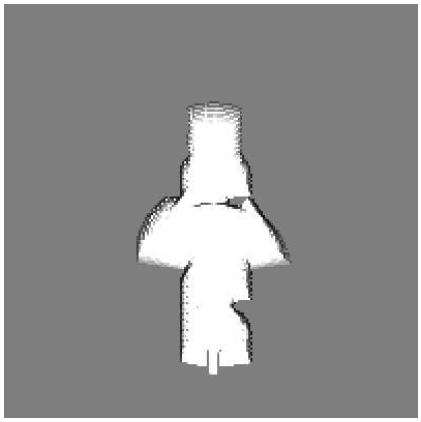


Figure 6.3: Map of the assault course, output of the simulation.

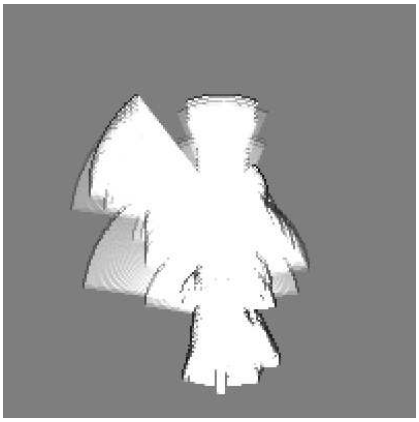


Figure 6.4: Map of the assault course, output of real data, gathered by the robot.

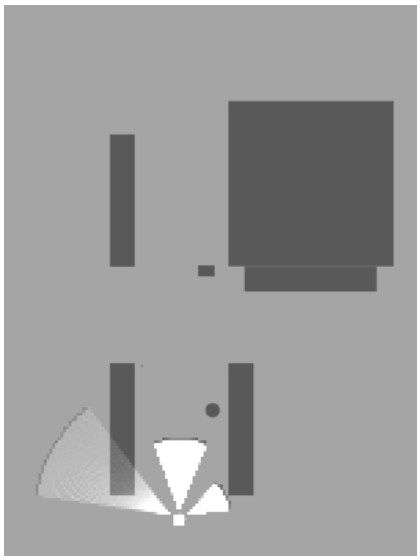


Figure 6.5: real ultrasound data, first measurement

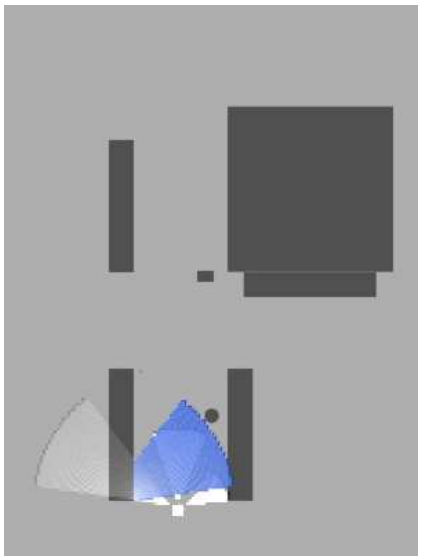


Figure 6.6: real data, plausible explanation for error in sensor A

a metal tube, is invisible in figure 6.1 due to its size but clearly visible in 6.2. Its location is at the top right of the bottom left cabinet. For the following experiments one particle is used, unless indicated otherwise.

Figure 6.3 is built by the simulator and figure 6.4 is a map that is created by the robot using ultrasound sensors. To emphasize the word *local* in local mapbuilding, only the three frontmost sensors on the robot are used in the construction of the maps. Map 6.3 is created without any simulated sensor or motion noise, and therefore represents the map that would be created by infinitely accurate ultrasound sensors on a robot without any motion error at all. Resolution of the maps is 5cm x 5cm. The simulator could be more accurate, but for the real application this would be overkill; the theoretical minimal error of the ultrasound sensors used in combination with the microcontroller is a little bit less than 2cm[1], but it appears to be the case that the error is often closer to 10cm, see chapter 2. In the same chapter it is also noted that the ultrasound sensor expected to be troubled by specular reflections in a more complex environment as the one used in the experiments. And indeed several discrepancies between figure 6.3 and figure 6.4 are visible, indicating errors in measurement. To see which errors in figure 6.4 can be accounted for by specular reflections, the output of several real test runs is compared to the simulator output, figure 6.3, step by step.

First it must be noted that the test run described below is one with as less

[1]Assuming the speed of sound is 340m/s and the ultrasound sensor polling frequency is 100 μs

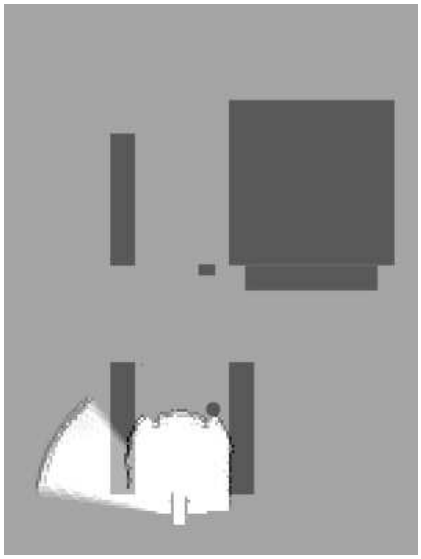


Figure 6.7: real ultrasound data, robot has travelled 1.2m.

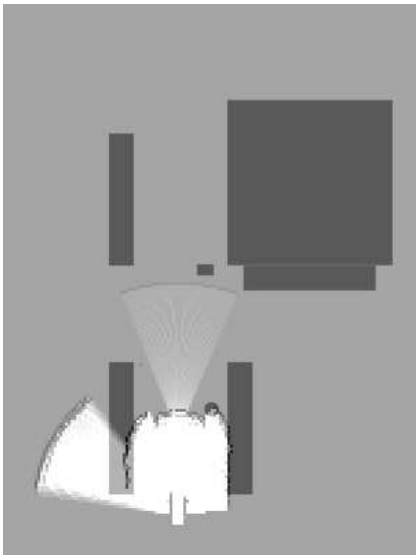


Figure 6.8: real ultrasound data

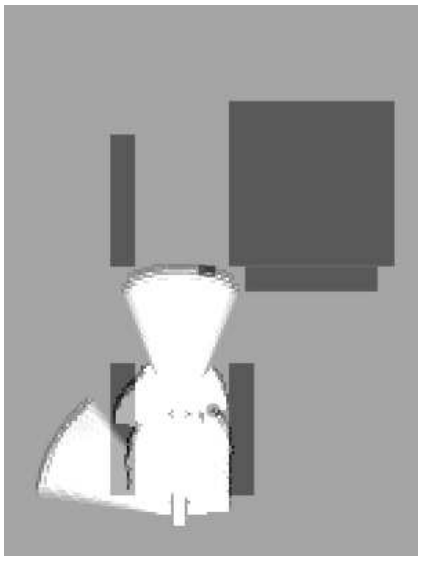


Figure 6.9: real ultrasound data

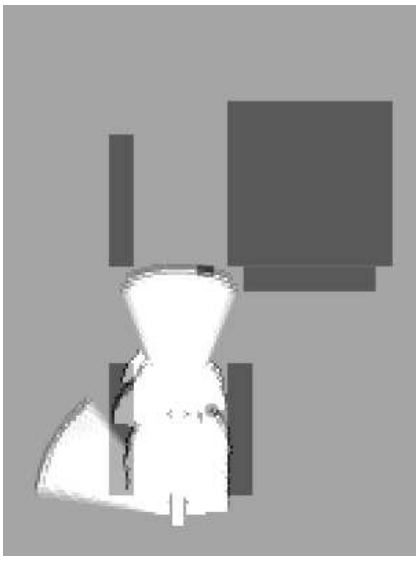


Figure 6.10: real ultrasound data

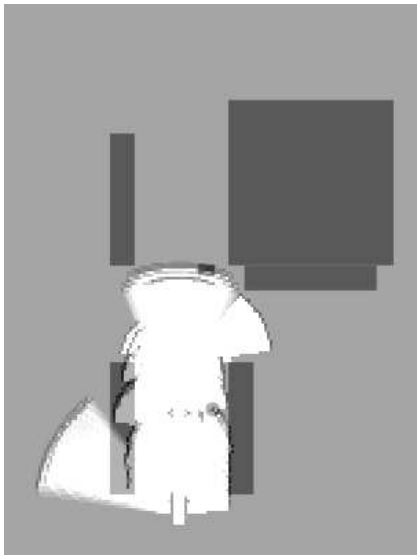


Figure 6.11: real ultrasound data

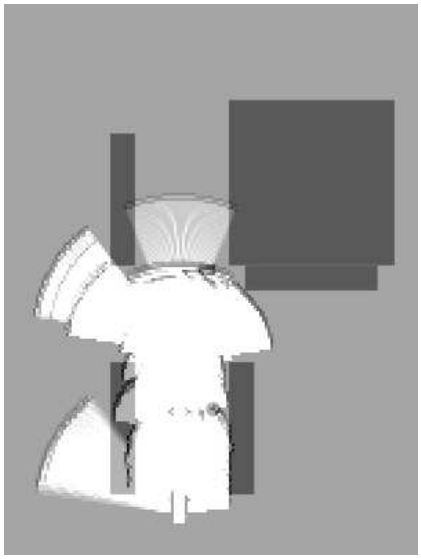


Figure 6.12: real ultrasound data

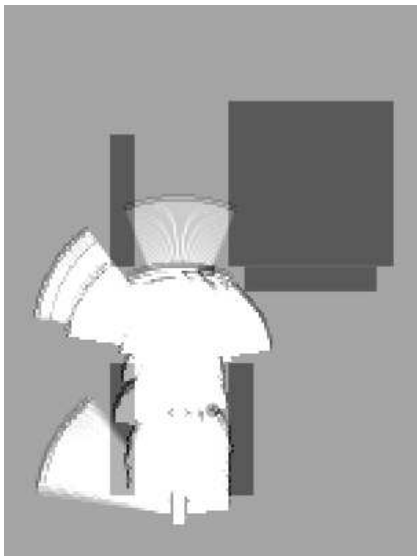


Figure 6.13: real ultrasound data

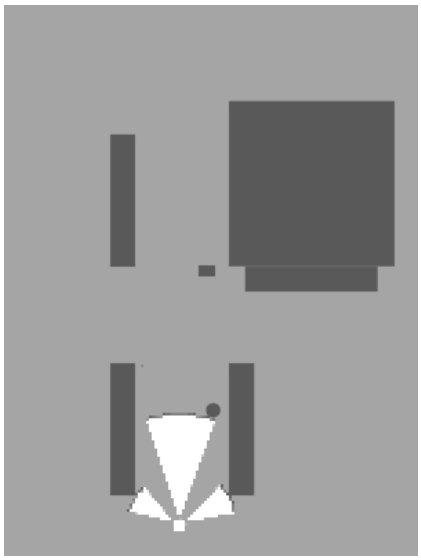


Figure 6.14: simulated ultrasound data, first measurement

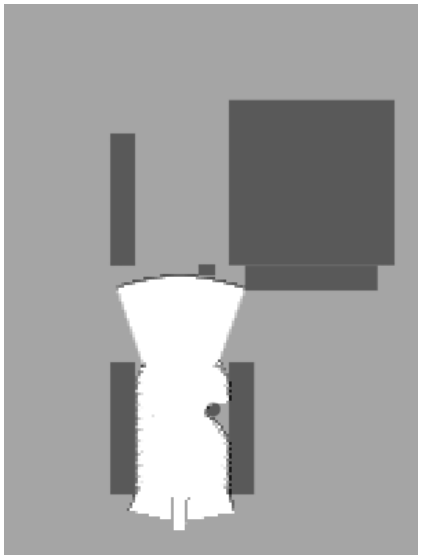


Figure 6.15: simulated ultrasound data

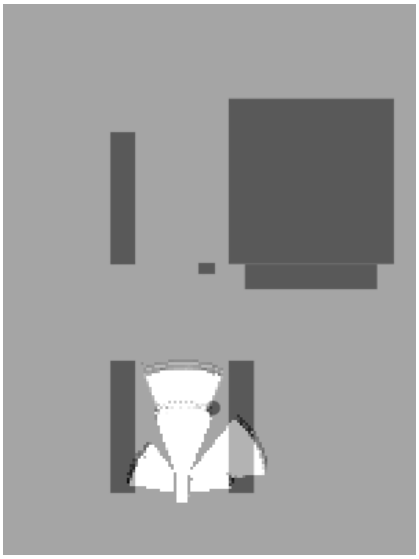


Figure 6.16: real ultrasound data

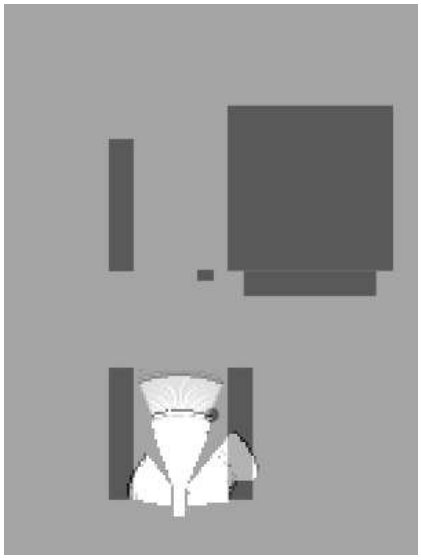


Figure 6.17: real ultrasound data

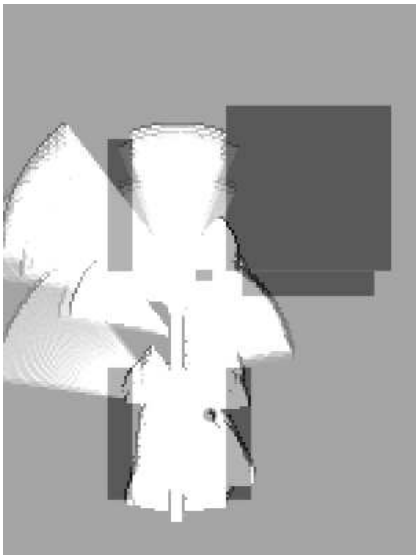


Figure 6.18: real ultrasound data

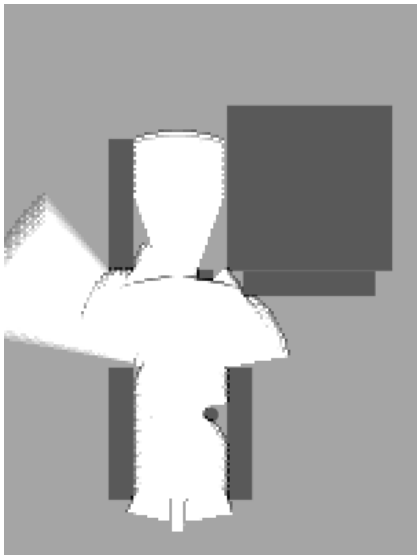


Figure 6.19: simulated ultrasound data

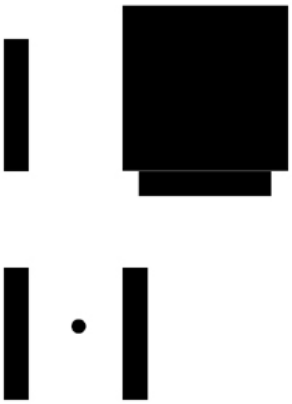


Figure 6.20: Alternative assault course, 2D.

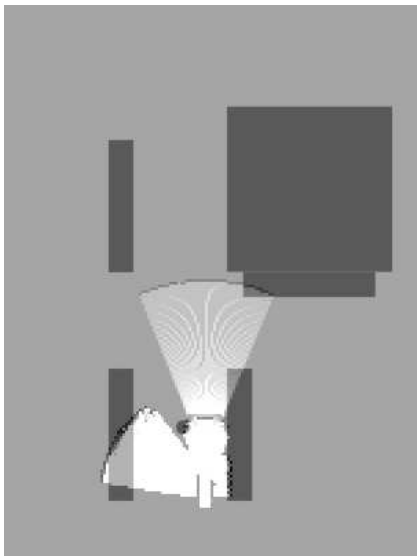


Figure 6.21: real ultrasound data

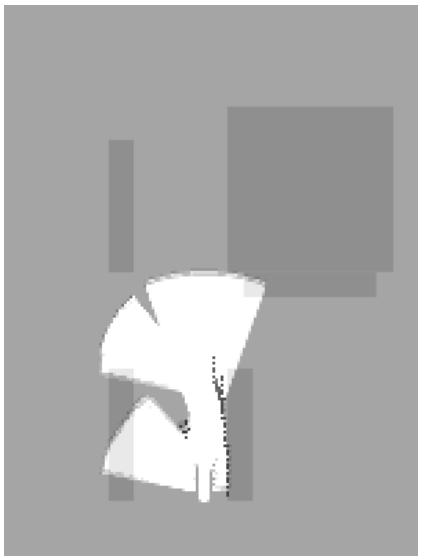


Figure 6.22: real data, integration of infrared and ultrasound

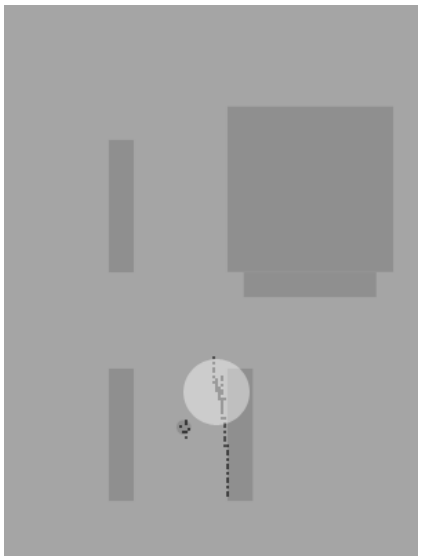


Figure 6.23: real infrared data, the clutter of points indicates motion error

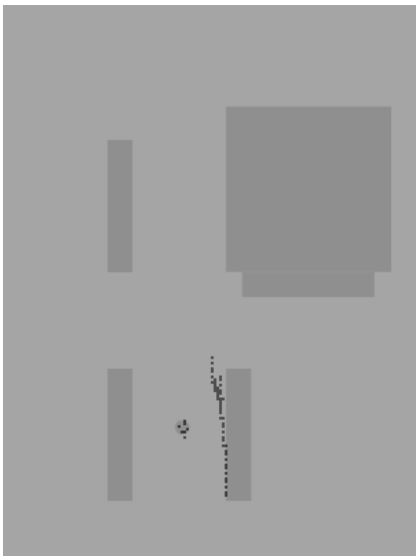


Figure 6.24: real infrared data, infrared sensor

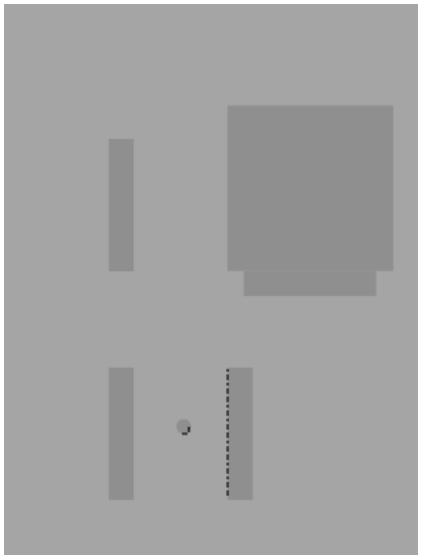


Figure 6.25: simulated infrared data, infrared sensor

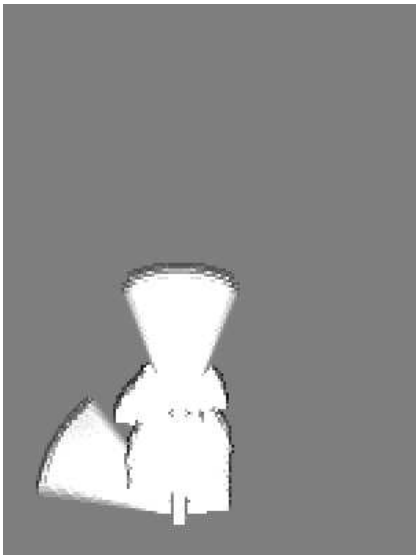


Figure 6.26: Map that is produced without any prior knowledge about a global map. Ultrasound data.

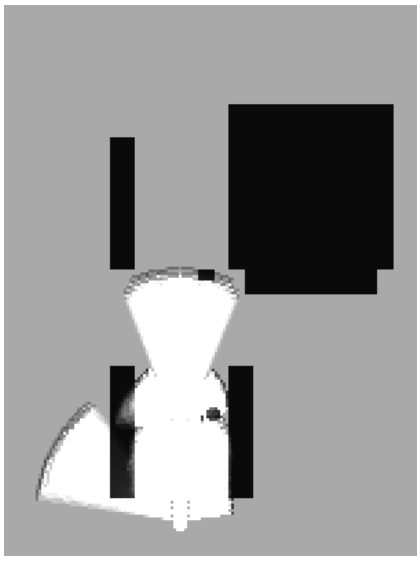


Figure 6.27: Here the global map (which matches with the real environment) was known to the robot on forehand. Ultrasound data.

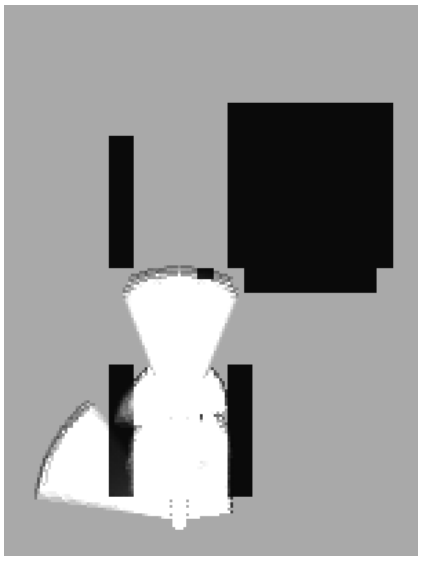


Figure 6.28: The erroneous global map; the location of the round bin in this map differs (figure 6.30) from the real location (figure 6.1). The algorithm is able to 'erase' the false location of the bin. Ultrasound data.

Figure 6.29: real infrared data without filter

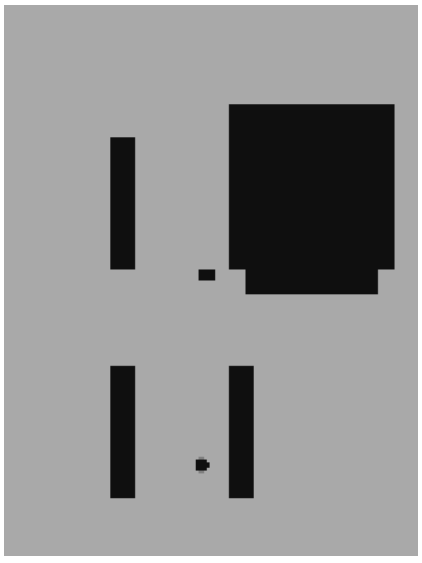


Figure 6.30: Erronous assault course given to the robot as global map.

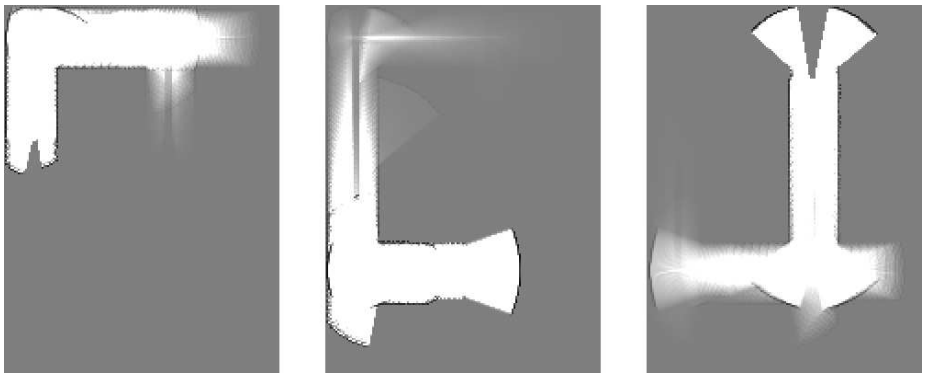


Figure 6.31: A global map is constantly available to the robot so the local map only has to keep track of the local environment and is therefore allowed to forget parts of the map it has visited. Here we see an implementation of an 'amnesia filter'; the robot gradually forgets parts of the local map it hasn't visited for a certain (adjustable) amount of time.

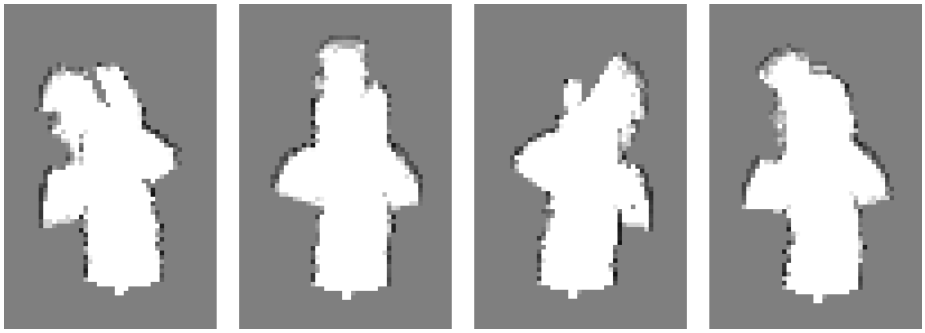


Figure 6.32: Diversity among particles. A lower spatial resolution than in the assault course experiments is used here in order to speed up the process, but the principle remains the same.

motion error as possible in order to focus on the measurements. During testing however it is almost impossible to avoid all errors. During this run on the assault course, the pose of the robot was measured by hand at several points. Whenever motion error interferes with the measurements, this is mentioned in the explanation.

Please have a look at figure 6.5 illustrating the measurement the robot takes at the very first step. We will refer to the leftmost, middle and rightmost sensor as A, B and C respectively. Figure 6.14 shows the simulated output, i.e. the expected output when no measurement noise is involved. Figure 6.5 clearly differs from 6.14. The leftmost sensor, sensor A, has made a measurement as though it didn't see the wall at all. Sensor B measured an object that isn't there. For the latter case, there is no obvious explanation. The nearest object to B is further away, so it detected an echo of another sensor that bounced of a distant object. Where the erroneous sensor reading really came from is hard, if not impossible, to recover. Sensor A has made an erroneous measurement as well. This measurement however *can* be explained. Apparently, it indeed failed to detect the wall. We've seen however in chapter 2 that it is quite possible for the sensor to detect an echo that has bounced of several objects. A plausible explanation for the measurement error is that it bounced of the wall, to the round metal bin and back via the wall to the sensor (figure 6.6). The robot however doesn't know anything about the map yet so it cannot correct the error, at least not without further information.

Fortunately, this first round of measurements does not seem to represent the overall correctness of the measurements. After driving about 1.2 meters the robot has made a small local map that is consistent with the surroundings, see figure 6.7. The very first measurement of sensor A is even partially compensated for by correct readings of sensor A; the wall that was undetected in the first measurement(s) was detected in following measurements. In the successive addition of correct information in the manner described in chapter 4, the wall has become clearly visible in the map.

From figure 6.8 it is clear that only a few centimeters further the robot detects the rectangular bin -which at that point is about 2.7 meters away. That this is not a one time event is evident from figure 6.9. The apparent variance in the measurements of sensor B is likely to originate from motion error; right after figure 6.8 the robot was paused for a moment to measure its pose by hand. At the moment the robot was stopped the robot swung around a bit, as was the case at the moment of continuing. Motion continued in the correct direction. Unfortunately, in contrast to the accurate measurements at 2.7m, figure 6.10 brings bad news for the accuracy of the ultrasound sensor. In figure 6.9 it appears to be the case that the ultrasound echo of sensor A bounces of either the wall or the hatstand. If the hatstand is mapped here we can conclude that the robot (i.e. the ultrasound sensor) is able to detect such a relatively small object in a complex environment. Figure 6.10 however shows us a map in which A failed to see the hatstand. So, whether we see a mapping of the hatstand or the wall in figure 6.9, the ability to detect an object like the hatstand in this environment is questionable at best. As a reference, figure 6.15 shows what the

robot should have mapped at this point.

The robot continues its course almost in a straight line. At about 2.6m of travelling, when the robot has just passed the hatstand, its orientation is altered by $-15\circ$ due to another pause-and-continue manoeuvre. Fortunately it recovers from this error, but not without updating the map with an incorrect measurement. In figure 6.11 the new update looks to be a correct one, measuring the distance to the corner of the cabinet that is standing to the wall. However, since the robot just had an error in its orientation clockwise, the echo cannot originate from the cabinet. Specular reflections inhibit the robot's ability to build an accurate map once more. In figures 6.12 and 6.13 we can see the evidence of *consistent* specular reflections; sensor A returns contradictive measurements and seems to 'doubt' between 2.0m and 2.6m. Sensor B also seems to return incorrect measurements. However, it is possible that the echos detected by sensor B originated from the door (frame) in the wall (not present in the hand drawn map) so no judgement is made about these measurements.

The sensors in combination with the DP-SLAM style measurement integration perform reasonably well; there are specular reflections that have an influence on the resulting map, but this map resembles the ideal map of figure 6.14 especially concerning the locations of the (larger) obstacles. Unfortunately, not all generated maps are as accurate as figure 6.13. Apart from possible motion errors, some maps generated on the same assault course show inexplicable and inconsistent errors in measurement. Moreover, some maps show very remarkably *consistent* errors; figure 6.16 and 6.17 are generated by two different test runs. Figure 6.21 is another example. This map is generated using the same environment, except that the only obstacle is the round metal bin which was placed in the middle 6.30. Although this kind of errors are likely to originate from echos that reflected of the same obstacles every run, which can even be reproduced under the right circumstances, it is impossible to predict the occurence of such a reroute of the soundburst in an initially unknown environment. Since no test run is the same, it is hard to reproduce errors. Based on these results, ultrasound sensors seem to perform well in most cases. Sometimes however maps are built that do not resemble the expected map.

The infrared sensor behaves according to expectations. Results from chapter 2 indicate that the reach is limited, from 8cm to maybe 60cm at most, but within this range it is accurate. Figure 6.20 depicts a short alternative assault course. Figure 6.20 is accompanied by the noise free simulation of a robot equipped with 4 infrared sensors (at $30\circ$, $90\circ$, $270\circ$ and $330\circ$) driving from south to north on this track (figure 6.25). Figure 6.24 shows the real measurement results. Figure 6.22 shows the integration of infrared and ultrasound measurements applied to this short assault course.

It seems that the sensor indeed lives up to expectations; the round bin is mapped, as is the wall to the right. Looking at figure 6.24 one would predict that the robot had an error in movement: towards the top, the robot maps the wall too far to the left which indicates a translational error to the right. The small clutter of points (figure 6.23) indicates a rotational error clockwise. Both predictions are correct according to the measurements that were done by hand.

The robot drifted 19cm to the right with a rotational error of $-15\circ$.

One filter is applied though to the map in figure 6.24: relying on the data collected in the individual sensor experiments from chapter 2, measurements indicating a distance of more than 60cm are removed from the process. Figure 6.29 is the resulting map if this filter is not applied. One can see that the cabinet on the left is noticed by the infrared sensor. The distance however is too great to accurately map the outlines.

Error in motion is a major problem in map building. Please take a look at figure 6.18. Sensor C has mapped the wall-cabinet combination almost exactly as in the ideal map. The location however is obviously wrong. In the middle part of the map one can see the robot drifted a bit to the left, since the map is somewhat skew to the right. At the end however, manual measurements indicate a small rotational error clockwise, meaning the robot drove in a slight curve. In the next section it is discussed whether DP-SLAM is able to compensate for the motion error using range measurements.

6.2 Error Compensation

DP-SLAM should be able to (i) compensate for motion noise and (ii) alter the global map if previously unknown obstacles are detected.

Figure 6.26 is the map that is based on the assumption that there's no motion error at all. However, as mentioned in the previous paragraph, by looking at the map it becomes clear an error in motion certainly occurred. The adjusted DP-SLAM algorithm used in this research has the advantage that it is allowed to use a global map which should make localization easier. However, the global map may contain errors.

Figure 6.27 shows basically the same map as figure 6.26. Or rather, they are based on the same data. The exception is that for figure 6.27 the algorithm knew the map on forehand. That is, the locations of the cabinets but also those of all the obstacles are known to the robot. For the first obstacle, the round bin, it is clear that robot indeed sees it, and maps around it when it goes by. The real question of these experiments however is whether the robot can also map the objects if the initial map is erronous; a phenomenon that could occur for example if someone would reallocate a bin. Figure 6.28 shows the result of this test. In the initial map available to the robot the round bin is at the wrong location (see figure 6.30). As the reader can see, the algorithm succeeds in 'erasing' the misplaced 'global' bin at one location, and adding an obstacle at the right location.

To compensate for motion error DP-SLAM relies on particles. Figure 6.32 shows a selection of 4 maps from 50 distinct particles. Evidently diversity exists. The particle filter is apparently capable of maintaining multiple maps. Unfortunately some problems arise when dealing with the algorithm in practice. The weights assigned to the particles have too little variance; no particle ever has more than 1 child. This can be an error in programming, it could be that the algorithm is sensitive to certain parameters and that weights will vary

perfectly under other conditions and it is even possible that it is inherent to the combination of the ultrasound sensor and the algorithm; More research is needed to make a sound judgement.

Another issue is that the algorithm appears to be quite slow. The PC running the software is an Athlon 2500+ at 1.83GHz with 256Mb of memory. Using a grid with a spatial resolution of 5cm x 5cm, the algorithms needs 5 seconds for every iteration with 10 particles. A spatial resolution of 4cm x 4cm requires about 8 seconds for the same number of particles, and a resolution of 10cm x 10cm requires only 1.5 seconds, suggesting processing time scales up quadratically with a linear increase in spatial resolution. The amount of time needed for one iteration for the last spatial resolution increases by a factor 4, to 6 seconds when the number of particles is doubled. By doubling the amount of particles for the 5cm x 5cm resolution the result is as the reader might have guessed: time increases by a factor 4, indicating that the amount of time increases quadratically with the amount of particles. Admittedly, it is hard to say without testing how many particles are needed exactly to produce a map, but bearing in mind that 'normal' DP-SLAM, using the laser range scanner, needs several thousands of particles, it is safe to say that the algorithm presented here will not run in real time, at least not in its current form.

Another way to deal with motion noise is to get around it by forgetting the past and rely on localization capabilities alone. A 'fade factor' in combination with time stamped particles can be used to eliminate information from the past. An example of this method is shown in figure 6.31.

Chapter 7

Conclusion

7.1 Conclusion

Sensors The accuracy and speed of sensors was tested in a simple setup. The range of the Sharp GP2D12 infrared sensor was put to the test and results indicate good performance from 8cm to 40cm, and reasonable performance between 40cm and 60cm. The sensor is robust within this range; even when the angle of the measured object is 45◦ performance does not decrease, see section 2.2.2. When the angle is too large however, the sensor will fail to see the object. Black objects can become invisible at a certain distance; the threshold depends on the material and the exact color. Summarized: The Sharp GP2D12 gives false negatives when measuring reflective materials or objects under a large angle, but rarely gives false positives.

The SRF04 ultrasound sensor has a much wider range: From 10cm to more than 300cm the sensor returns consistent measurements. Also, the SRF04's beam is much wider. Operating the ultrasound sensor is more complicated than operating the infrared sensor. Using several ultrasound sensors at once almost certainly gives rise to measurement errors due to the fact that a sensor can receive a signal that originated from another sensor. Since the sensor's microphone is sensitive enough to pick up an ultrasound signal that has bounced of several objects before returning, an individual sensor can even pick up an erroneous echo from itself. Summarized: The SRF04 is able to detect objects at a wider range than the Sharp GP2D12 infrared sensor, but is sensitive to reflections when the sensor is triggered at a fast pace, or when multiple sensors are used. In these cases, the SRF04 gives false positives.

Algorithms A promising mapping algorithm, Montemerlo's FastSLAM [4], is a hybrid algorithm; a combination of a particle filter and a Kalman filter makes FastSLAM fast and reliable. It is only suited for environments that can easily be represented by a collection of coordinate points, like a room full of small blocks or a park with trees. Also, to be able to distinguish between separate

landmarks, FastSLAM relies on the accurate laser scanner to provide data in the form of a coordinate. Based on literature study it is illogical to try and convert FastSLAM to an algorithm that builds small, local maps with the use of inaccurate sensors. In contrast, DP-SLAM is a good candidate for local map building; DP-SLAM is entirely particle filter based and makes no assumptions about the environment. Normally this would imply unrealistically high processing times, but Eliazar and Parr [1] claim their DP-SLAM can operate in real-time. Although originally designed to use for large maps using the accurate laser range finder, it is relatively easy converted for use with local maps using inaccurate sensors. Summarized: FastSLAM relies on accurate sensors and cannot be used for mapping domestic environments. Based on literature study, DP-SLAM is the best candidate algorithm for this graduation project.

Mapping Using the infrared and ultrasound sensors on a moving mobile robot platform has given insight in the use and reliability of both sensors. Ultrasound sensor data seems to be reasonably consistent. It sometimes 'misses' objects either due to reflections or the size of the object; small obstacles go by unnoticed. It is possible to use the ultrasound sensor for detecting obstacles with a diameter / width larger than 25cm. The sensor is less suitable for *mapping* obstacles due to the width of the ultrasound beam. In chapter 6 it is shown that a global map can be successfully overwritten by new data that is gathered by the SRF04 and integrated into the map using the DP-SLAM style grid update functions. The Sharp GP2D12 infrared sensor performs well within its range of 8cm to 60cm; it is possible to map objects using this sensor and the DP grid update functions. If a filter is applied that removes all >60cm measurements the sensor doesn't seem to have any false positives.

Unfortunately, the DP-SLAM implementation used in the experiments is impractically slow with only a few dozen particles. For actual real time performance -i.e. at most 1.0 second for every iteration of the algorithm- one would need an improvement factor in the order of thousands. Results of measurements with only a few dozen particles however *do* indicate that particle diversity can theoretically compensate for motion error under the right circumstances. Summarized: It is possible to build local maps using both the tested ultrasound sensor and the infrared sensor in combination with DP-SLAM. Due to computational limitations it is not possible to use the amount of particles required for compensation of motion errors in a real life application.

7.2 Recommendations

Improvements are recommended to both sensors and algorithms to improve the overall map building performance. The sensors used in these experiments are much less accurate than the sensors for which the algorithm were originally designed. Inaccurate sensors indirectly require more computational effort, so improvements in the performance of sensors will result in an improvement as a whole.

A possible improvement to the measurements of the infrared sensor is to simply use more sensors on a robot; The sensors don't easily interfere with each other, so it is a matter of having the right hardware that can poll more than 6 sensors and handle the communication between sensors and computer. Another beneficial alteration would be using a better function fit to convert infrared sensor output to real distances.

A way to improve the ultrasound sensors is to use different frequencies for every sensor on a platform to rule out false detections. Perhaps the hardware of the SRF04 can be altered to work with different frequencies. To the author's best knowledge no off-the-shelf sensors are available of which the frequency can be adjusted without changing the hardware, but further research to this matter is recommended. Another improvement would be to use an ultrasound sensor with *variable* frequencies to prevent it from interfering with itself. An obvious improvement concerning sensors would be to start using encoders on the omni-wheels. Perhaps it is possible to still use the Laser Beetles.

Improvements to DP-SLAM are needed to transform it into a genuine real-time algorithm. Speed improvements to the sensors are of no use if the algorithm using their data already has a significantly lower bandwidth. A better (implementation of the) model of the ultrasound sensor could result in more efficient updates to the complex occupancy grid, thereby increasing performance. Also, the current software implements the balanced trees in the occupancy grid as simple lists, due to the fact that the overhead, that is inherent to the use of a balanced tree, is only worth the computational payload for large trees, i.e. trees with a lot of particle ID's. Therefore, if the overall bandwidth of the implemented algorithm is improved and more particles are used, it could be interesting to start using balanced trees; When using balanced trees, the computational complexity scales up not quadratically, which is now the case, but logarithmically.

Appendix A

Application Manual

For anyone interested in getting the robot up and running again, or compiling the software, here's the application manual. In the following section, an explanation is given about which software you need to install on the laptop that is placed on top of the robot, how to apply the bluetooth connection, but also how to charge the battery, etc. In the section after that, just enough information is provided to be able to change important parameters and to learn how to compile the software for use on the robot or the computer, for simulation or logged data.

Appendix A.1 Robot

Starting from the bottom up, the first thing you need to do is check the battery. It needs to be fully charged! Driving time is limited, in the order of 15 minutes to half an hour with a new battery. Normally, the power to the USB hub should already be connected. The USB cables from the hub to the DLP's may remain connected at all times. The cable going from the hub to the laptop however should be disconnected before powering up the robot. Make sure the switch attached to the largest PCB is set to "laden" ("charge"). This feature is not used, it is just to make sure that the robot doesn't start moving when you're connecting the battery; when the digital potentiometers that provide setpoints to the motor controllers are powered, they are reset not to 0, but to the minimum output of -10V, which make the omniwheels turn at full speed in one direction.

Next, attach the wires to the battery. Connection has always been a bit provisionary. The blue wire connects to $-$, the brown wire to $+$. These wires power the motors and motor controllers. The easiest way to attach them is using the (somewhat less provisional) connectors that are attached to the wires that power the PCB's, thereby attaching them altogether. Attach the single black wire to $-$, the pair of black wires should connect to the $+$. The hardware should now be fully functional.

Appendix A.2 Laptop

To be able to communicate with the DLP245PB, some drivers need to be installed. These can be downloaded for free from the company's website[1]. An installation guide is available[2]. Included in the zip file are also the .h and .lib files used for programming. A programmer's guide can be downloaded[3]. An extremely basic but very useful test application is available, also for free[4]. If you are asked to, choose the DLL drivers, not the VCP (Virtual Com Port) drivers.

After installing the drivers, it is a good idea to first check the connection to both DLP245PB's with the test application. Run the application. Select "DLL" and hit the "search" button and make sure it finds 2 devices. If this is not the case, shut the robot down (i.e. pull the wires) and start over again. Choose one and click the "open" button. Now you are ready to send and receive chars to and from the robot. To check communication with the DLP controlling the sensors use for example "02 EE 00 00 00 00 00 00 — 02". Press the concerning "send" button and if everything is fine you will receive the number of seconds the device is up and running (or actually the number of seconds *mod* 60). To check communication with the motor PCB, send the command "02 90 00 00 00 00 00 00 — 02", which is the command that sets all potmeters to 0. As an acknowledgement, the device will send you "90".

The software, written for this graduation assignment, to control the robot can be copied to any directory on the laptop. How to exactly compile the software is explained in Appendix C. Important to know is that a very basic cursorcontrol is implemented; with the arrow buttons on the keyboard the robot can be sent to the left, right etc. The reason that this is so important is that the same functionality is used to control the robot via the bluetooth connection. Plug the bluetooth stick in the hub of the robot and let the laptop install the drivers. Do the same for the computer you want to use for steering the robot. By clicking on the Bluetooth icon in the Windows XP system tray at the bottom right of the screen, you can setup a Personal Area Network (PAN) by letting the two devices find each other. Now you can use your favorite Remote Desktop software. This functionality comes with Windows XP, but with this application for the combination of laptop and desktop used in the experiments it was impossible to get a connection. A program that *did* work was Remote Desktop Control[5]. This software is not free, but you are able to download an evaluation copy. With such software, you are able to take control of the laptop from your desktop. Key strokes are passed on from the desktop to the laptop. As mentioned, the same buttons, arrow left, arrow right, etc. can be used to steer the robot. Pressing any other key than the arrow keys will stop the robot.

[1] http://www.dlpdesign.com/D30104.zip
[2] http://www.dlpdesign.com/winxp_install_guide.pdf
[3] http://www.dlpdesign.com/drivers/D2XXPG21.pdf
[4] http://www.dlpdesign.com/usb/images/dlptest10c.zip
[5] http://www.remote-desktop-control.com/

Appendix B

Robot Hardware

In the following pages parts of data sheets and specifications are shown that could be useful to a person continuing work on the robot. Although the pages displayed here are -according to the author- the most useful pages for this thesis, they do not reflect or summarize the full content of the original articles. Someone interested in a certain device is therefore encouraged to download the original article.

PIC16F877 http://www.alldatasheet.com
DLP245PB http://www.dlpdesign.com
AD7376 http://www.alldatasheet.com
Elmo Motor Controller http://www.elmomc.com
Motor PCB -
Sharp GP2D12 From http://www.alldatasheet.com
SRF04 From http://www.datasheetarchive.com

PIC16F87X

28/40-Pin 8-Bit CMOS FLASH Microcontrollers

Devices Included in this Data Sheet:

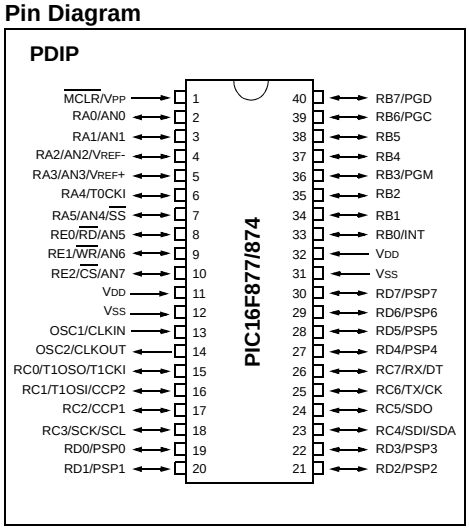
- PIC16F873
- PIC16F874
- PIC16F876
- PIC16F877

Microcontroller Core Features:

- High performance RISC CPU
- Only 35 single word instructions to learn
- All single cycle instructions except for program branches which are two cycle
- Operating speed: DC - 20 MHz clock input
  DC - 200 ns instruction cycle
- Up to 8K x 14 words of FLASH Program Memory,
  Up to 368 x 8 bytes of Data Memory (RAM)
  Up to 256 x 8 bytes of EEPROM Data Memory
- Pinout compatible to the PIC16C73B/74B/76/77
- Interrupt capability (up to 14 sources)
- Eight level deep hardware stack
- Direct, indirect and relative addressing modes
- Power-on Reset (POR)
- Power-up Timer (PWRT) and
  Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation
- Programmable code protection
- Power saving SLEEP mode
- Selectable oscillator options
- Low power, high speed CMOS FLASH/EEPROM technology
- Fully static design
- In-Circuit Serial Programming™ (ICSP) via two pins
- Single 5V In-Circuit Serial Programming capability
- In-Circuit Debugging via two pins
- Processor read/write access to program memory
- Wide operating voltage range: 2.0V to 5.5V
- High Sink/Source Current: 25 mA
- Commercial, Industrial and Extended temperature ranges
- Low-power consumption:
  - < 0.6 mA typical @ 3V, 4 MHz
  - 20 μA typical @ 3V, 32 kHz
  - < 1 μA typical standby current

Pin Diagram

PDIP

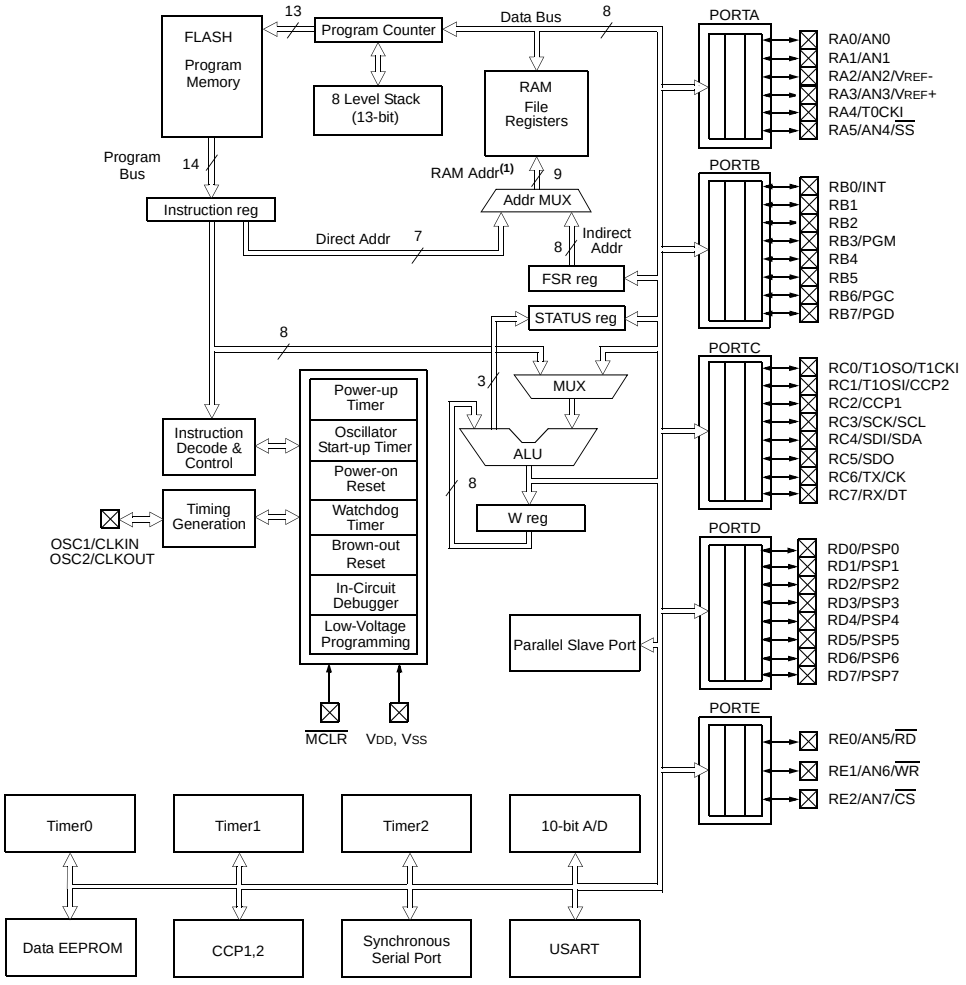


Peripheral Features:

- Timer0: 8-bit timer/counter with 8-bit prescaler
- Timer1: 16-bit timer/counter with prescaler, can be incremented during SLEEP via external crystal/clock
- Timer2: 8-bit timer/counter with 8-bit period register, prescaler and postscaler
- Two Capture, Compare, PWM modules
  - Capture is 16-bit, max. resolution is 12.5 ns
  - Compare is 16-bit, max. resolution is 200 ns
  - PWM max. resolution is 10-bit
- 10-bit multi-channel Analog-to-Digital converter
- Synchronous Serial Port (SSP) with SPI™ (Master mode) and I^2C™ (Master/Slave)
- Universal Synchronous Asynchronous Receiver Transmitter (USART/SCI) with 9-bit address detection
- Parallel Slave Port (PSP) 8-bits wide, with external RD, WR and CS controls (40/44-pin only)
- Brown-out detection circuitry for Brown-out Reset (BOR)

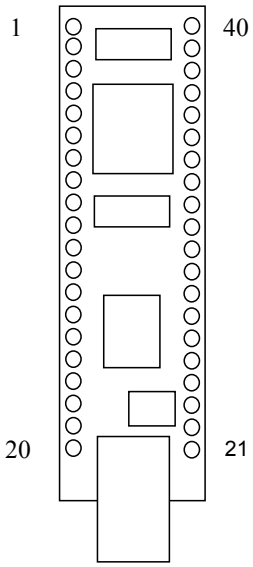
FIGURE 1-2: PIC16F874 AND PIC16F877 BLOCK DIAGRAM

Device	Program FLASH	Data Memory	Data EEPROM
PIC16F874	4K	192 Bytes	128 Bytes
PIC16F877	8K	368 Bytes	256 Bytes



Note 1: Higher order bits are from the STATUS register.

TABLE 1: DLP-245PB PINOUT DESCRIPTION



Pin #	Description
1	**GROUND**
2	**E0** (I/O) Port Pin E0 connected to the 16F877 microcontroller. A/D Channel 5.
3	**A0** (I/O) Port Pin A0 connected to the 16F877 microcontroller. A/D Channel 0.
4	**A1** (I/O) Port Pin A1 connected to the 16F877 microcontroller. A/D Channel 1.
5	**A2** (I/O) Port Pin A2 connected to the 16F877 microcontroller. A/D Channel 2.
6	**A3** (I/O) Port Pin A3 connected to the 16F877 microcontroller. A/D Channel 3.
7	**A4** (I/O) Port Pin A4 connected to the 16F877 microcontroller. Open drain output.
8	**A5** (I/O) Port Pin A5 connected to the 16F877 microcontroller. A/D Channel 4.
9	**UPRST** (In) 16F877 Reset input. Can be left disconnected if not used.
10	**GROUND**
11	**RESET#** (In) Can be used by an external device to reset the FT245BM. Can be left disconnected if not used.
12	**RESETO#** (Out) Output of the FT245BM's internal Reset Generator. Stays high impedance for ~ 2ms after VCC>3.5v and the internal clock starts up, then clamps its output to the 3.3v output of the internal regulator. Taking RESET# low will also force RESETO# to go high impedance. RESET0# is NOT affected by a USB Bus Reset.
13	**GROUND**
14	**3V3OUT** (Out) Output from the integrated L.D.O. regulator. Its primary purpose is to provide the

	internal 3.3v supply to the USB transceiver cell and the RSTOUT# pin. A small amount of current (<=5mA) can be drawn from this pin to power external 3.3v logic if required.
15	**GROUND**
16	**SWVCC** (Out) Power from EXTVCC (Pin 19) controlled via Pin 10 (POWERN#) of the FT245BM and Q1 MOSFET power switch. R5 and C3 control the power-up rate to help limit inrush current.
17	**GROUND**
18	**VCC-IO** (In) 4.4 volt to +5.25 volt VCC to the FT245BM's interface pins 10-12, 14-16, and 18-25. This pin must be connected to VCC from the target electronics or EXTVCC.
19	**EXTVCC** (In) Use for applying main power (4.4 to 5.25 volts) to the module. Connect to PORTVCC if the module is to be powered by the USB port (typical configuration).
20	**PORTVCC** (Out) Power from USB port. Connect to EXTVCC if module is to be powered by the USB port (typical configuration). 500mA is the maximum current available to the DLP-245PB and target electronics if the USB device is configured for high power.
21	**DB7** (I/O) Line 7 of the data bus between the 16F877 and the FT245BM USB-FIFO.
22	**DB6** (I/O) Line 6 of the data bus between the 16F877 and the FT245BM USB-FIFO.
23	**DB5** (I/O) Line 5 of the data bus between the 16F877 and the FT245BM USB-FIFO.
24	**DB4** (I/O) Line 4 of the data bus between the 16F877 and the FT245BM USB-FIFO.
25	**DB3** (I/O) Line 3 of the data bus between the 16F877 and the FT245BM USB-FIFO.
26	**DB2** (I/O) Line 2 of the data bus between the 16F877 and the FT245BM USB-FIFO.
27	**DB1** (I/O) Line 1 of the data bus between the 16F877 and the FT245BM USB-FIFO.
28	**DB0** (I/O) Line 0 of the data bus between the 16F877 and the FT245BM USB-FIFO.
29	**B5** (I/O) Port Pin B5 connected to the 16F877 microcontroller.
30	**B4** (I/O) Port Pin B4 connected to the 16F877 microcontroller.
31	**B0** (I/O) Port Pin B0 connected to the 16F877 microcontroller.
32	**C0** (I/O) Port Pin C0 connected to the 16F877 microcontroller.
33	**C1** (I/O) Port Pin C1 connected to the 16F877 microcontroller.
34	**C2** (I/O) Port Pin C2 connected to the 16F877 microcontroller.
35	**C3** (I/O) Port Pin C3 connected to the 16F877 microcontroller.
36	**C4** (I/O) Port Pin C4 connected to the 16F877 microcontroller.
37	**C5** (I/O) Port Pin C5 connected to the 16F877 microcontroller.
38	**C6** (I/O) Port Pin C6 connected to the 16F877 microcontroller.
39	**C7** (I/O) Port Pin C7 connected to the 16F877 microcontroller.
40	**GROUND**

±15 V Operation Digital Potentiometer

AD7376*

FEATURES
128 Position
Potentiometer Replacement
10 kΩ, 50 kΩ, 100 kΩ, 1 MΩ
Power Shutdown: Less than 1 μA
3-Wire SPI Compatible Serial Data Input
+5 V to +30 V Single Supply Operation
±5 V to ±15 V Dual Supply Operation
Midscale Preset

APPLICATIONS
Mechanical Potentiometer Replacement
Instrumentation: Gain, Offset Adjustment
Programmable Voltage-to-Current Conversion
Programmable Filters, Delays, Time Constants
Line Impedance Matching
Power Supply Adjustment

GENERAL DESCRIPTION

The AD7376 provides a single channel, 128-position digitally-controlled variable resistor (VR) device. This device performs the same electronic adjustment function as a potentiometer or variable resistor. These products were optimized for instrument and test equipment applications where a combination of high voltage with a choice between bandwidth or power dissipation are available as a result of the wide selection of end-to-end terminal resistance values. The AD7376 contains a fixed resistor with a wiper contact that taps the fixed resistor value at a point determined by a digital code loaded into the SPI-compatible serial-input register. The resistance between the wiper and either endpoint of the fixed resistor varies linearly with respect to the digital code transferred into the VR latch. The variable resistor offers a completely programmable value of resistance between the A terminal and the wiper or the B terminal and the wiper. The fixed A to B terminal resistance of 10 kΩ, 50 kΩ, 100 kΩ or 1 MΩ has a nominal temperature coefficient of –300 ppm/°C.

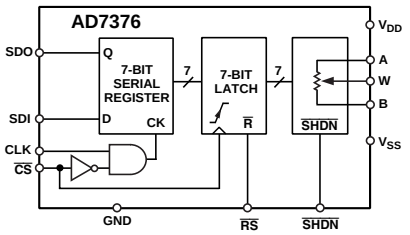
The VR has its own VR latch which holds its programmed resistance value. The VR latch is updated from an internal serial-to-parallel shift register which is loaded from a standard 3-wire serial-input digital interface. Seven data bits make up the data word clocked into the serial data input register (SDI). Only the last seven bits of the data word loaded are transferred into the 7-bit VR latch when the $\overline{CS}$ strobe is returned to logic high. A serial data output pin (SDO) at the opposite end of the serial register allows simple daisy-chaining in multiple VR applications without additional external decoding logic.

The reset ($\overline{RS}$) pin forces the wiper to the midscale position by loading 40_H into the VR latch. The $\overline{SHDN}$ pin forces the resistor

*Patent Number: 5495245

REV. 0

FUNCTIONAL BLOCK DIAGRAM



to an end-to-end open circuit condition on the A terminal and shorts the wiper to the B terminal, achieving a microwatt power shutdown state. When shutdown is returned to logic high, the previous latch settings put the wiper in the same resistance setting prior to shutdown as long as power to V_{DD} is not removed. The digital interface is still active in shutdown so that code changes can be made that will produce a new wiper position when the device is taken out of shutdown.

The AD7376 is available in both surface mount (SOL-16) and the 14-lead plastic DIP package. For ultracompact solutions selected models are available in the thin TSSOP package. All parts are guaranteed to operate over the extended industrial temperature range of –40°C to +85°C. For operation at lower supply voltages (+3 V to +5 V), see the AD8400/AD8402/AD8403 products.

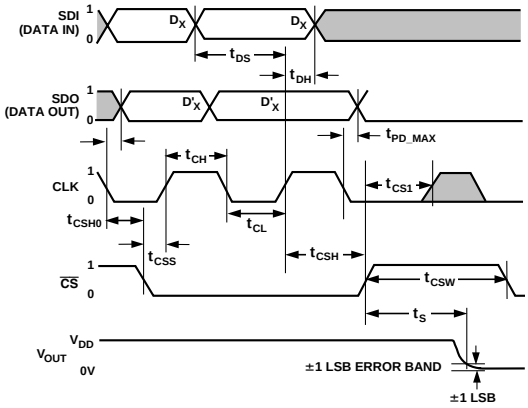


Figure 1. Detail Timing Diagram

The last seven data bits clocked into the serial input register will be transferred to the VR 7-bit latch when $\overline{CS}$ returns to logic high. Extra data bits are ignored.

One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.
Tel: 781/329-4700 World Wide Web Site: http://www.analog.com
Fax: 781/326-8703

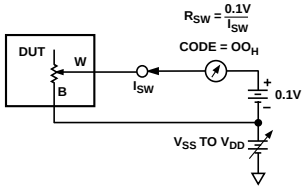


Figure 38. Incremental ON Resistance Test Circuit

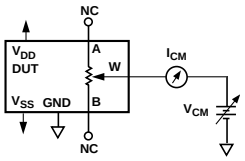


Figure 39. Common-Mode Leakage Current Test Circuit

OPERATION

The AD7376 provides a 128-position digitally-controlled variable resistor (VR) device. Changing the programmed VR settings is accomplished by clocking in a 7-bit serial data word into the SDI (Serial Data Input) pin, while $\overline{CS}$ is active low. When $\overline{CS}$ returns high the last seven bits are transferred into the RDAC latch setting the new wiper position. The exact timing requirements are shown in Figure 1.

The AD7376 resets to a midscale by asserting the $\overline{RS}$ pin, simplifying initial conditions at power-up. Both parts have a power shutdown $\overline{SHDN}$ pin which places the RDAC in a zero power consumption state where terminal A is open circuited and the wiper W is connected to B, resulting in only leakage currents being consumed in the VR structure. In shutdown mode the VR latch settings are maintained so that, returning to operational mode from power shutdown, the VR settings return to their previous resistance values.

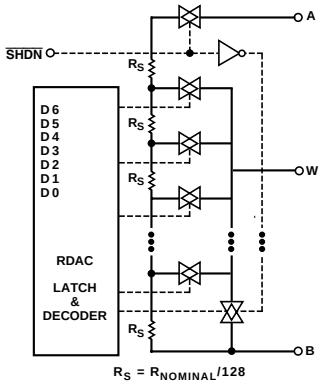


Figure 40. AD7376 Equivalent RDAC Circuit

PROGRAMMING THE VARIABLE RESISTOR

Rheostat Operation

The nominal resistance of the RDAC between terminals A and B are available with values of 10 kΩ, 50 kΩ, 100 kΩ and 1 MΩ. The final three characters of the part number determine the nominal resistance value, e.g., 10 kΩ = 10; 50 kΩ = 50; 100 kΩ = 100; 1 MΩ = 1M. The nominal resistance (R_{AB}) of the VR has 128 contact points accessed by the wiper terminal, plus the B terminal contact. The 7-bit data word in the RDAC latch is decoded to select one of the 128 possible settings. The wiper's first connection starts at the B terminal for data 00_H. This B–terminal connection has a wiper contact resistance of 120 Ω. The second connection (10 kΩ part) is the first tap point located at 198 Ω (= R_{BA} [nominal resistance]/128 + R_W = 78 Ω + 120 Ω) for data 01_H. The third connection is the next tap point representing 156 + 120 = 276 Ω for data 02_H. Each LSB data value increase moves the wiper up the resistor ladder until the last tap point is reached at 10041 Ω. The wiper does not directly connect to the B terminal. See Figure 40 for a simplified diagram of the equivalent RDAC circuit.

The general transfer equation that determines the digitally programmed output resistance between W and B is:

$$R_{WB}(D) = (D)/128 \times R_{BA} + R_W \quad (1)$$

where D is the data contained in the 7-bit VR latch, and R_{BA} is the nominal end-to-end resistance.

For example, when V_B = 0 V and A–terminal is open circuit, the following output resistance values will be set for the following VR latch codes (applies to the 10 kΩ potentiometer).

Table I.

D (DEC)	R_{WB} (Ω)	Output State
127	10041	Full-Scale
64	5120	Midscale ($\overline{RS}$ = 0 Condition)
1	276	1 LSB
0	198	Zero-Scale (Wiper Contact Resistance)

Note that in the zero-scale condition a finite wiper resistance of 120 Ω is present. Care should be taken to limit the current flow between W and B in this state to a maximum value of 5 mA to avoid degradation or possible destruction of the internal switch contact.

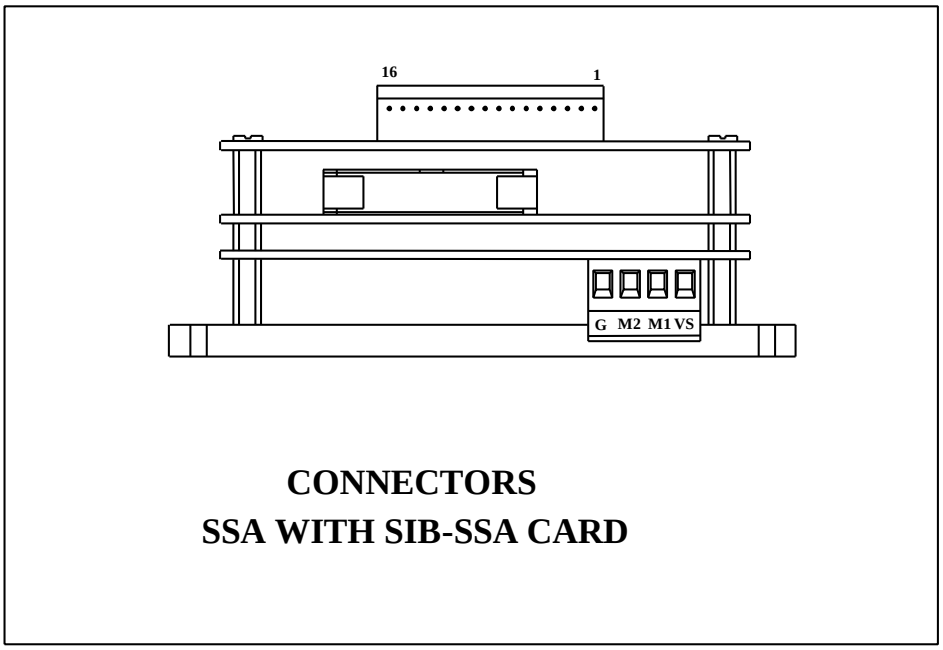
Like the mechanical potentiometer the RDAC replaces, it is totally symmetrical. The resistance between the wiper W and terminal A also produces a digitally controlled resistance R_{WA}. When these terminals are used the B–terminal should be tied to the wiper. Setting the resistance value for R_{WA} starts at a maximum value of resistance and decreases as the data loaded in the latch is increased in value. The general transfer equation for this operation is:

$$R_{WA}(D) = (128\text{-}D)/128 \times R_{BA} + R_W \quad (2)$$

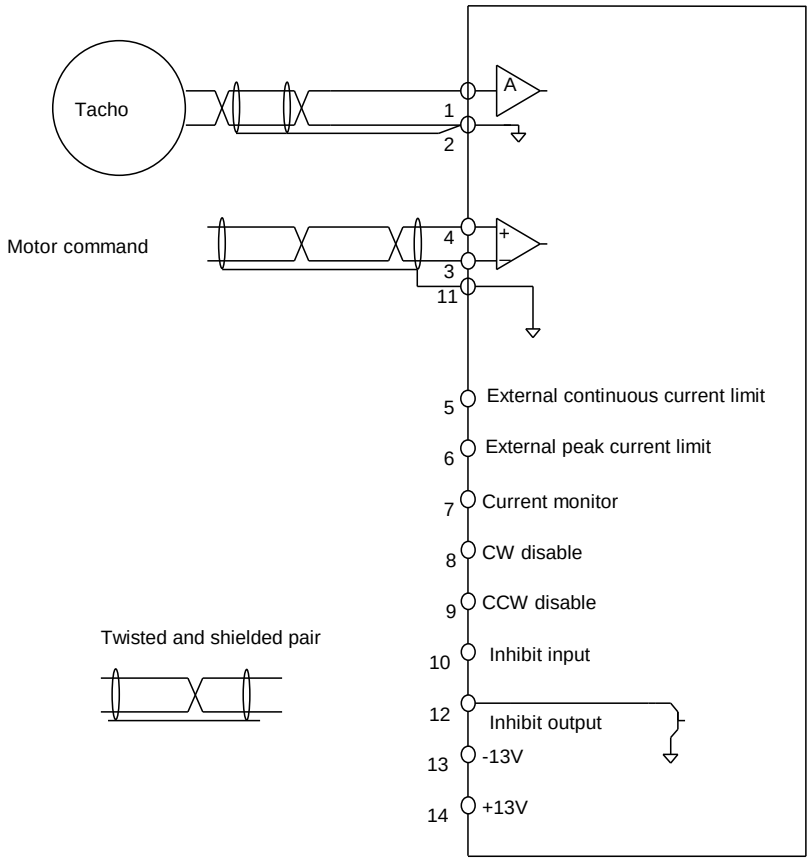
where D is the data contained in the 7-bit RDAC latch, and R_{BA} is the nominal end-to-end resistance. For example, when V_A = 0 V and B–terminal is tied to the wiper W the following output resistance values will be set for the following RDAC latch codes.

5.2 Terminals for SIB-SSA

The numbering of the SIB-SSA terminals (1-16) is identical to the numbering of the SSA control board connector.

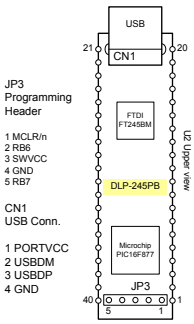


CONNECTORS
SSA WITH SIB-SSA CARD



SSA CONTROL CONNECTIONS

TACHOGENERATOR FEEDBACK

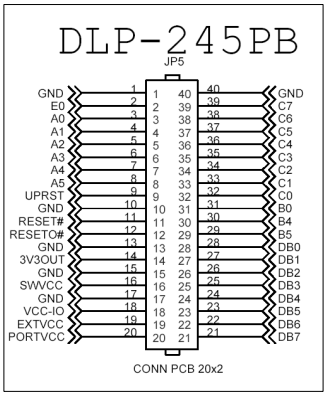


JP3
Programming
Header

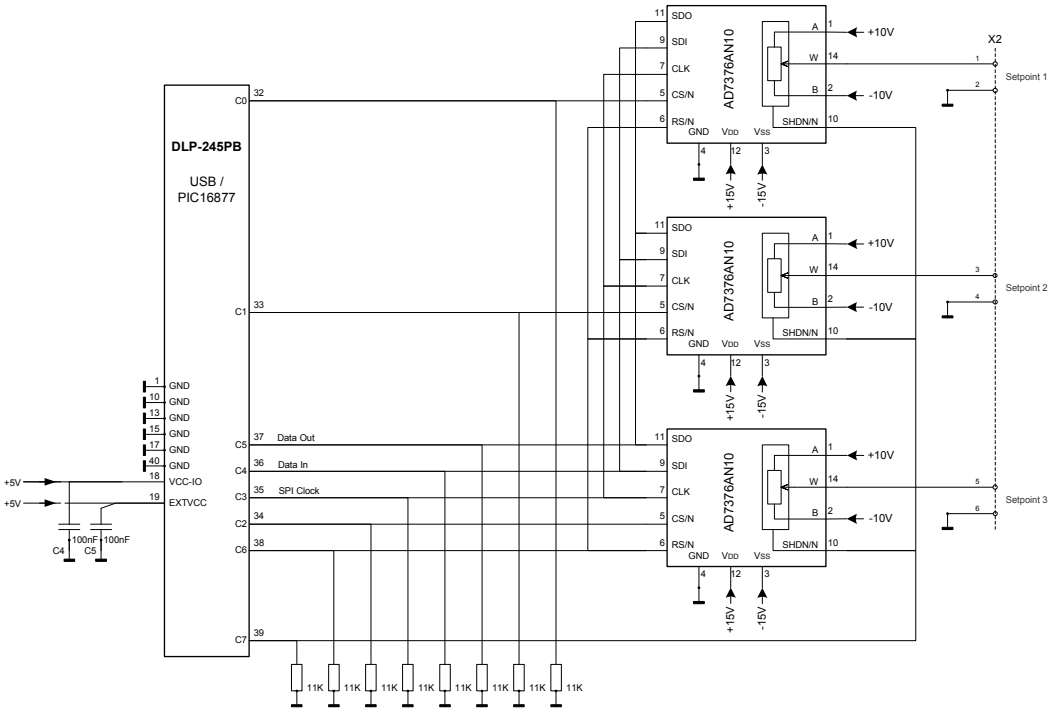
1 MCLR/n
2 RB6
3 SWVCC
4 GND
5 RB7

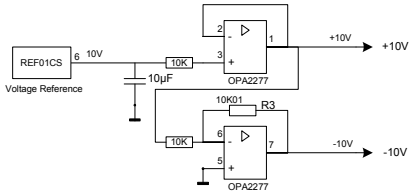
CN1
USB Conn.

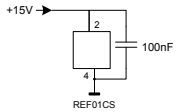
1 PORTVCC
2 USBDM
3 USBDP
4 GND

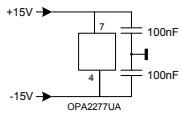


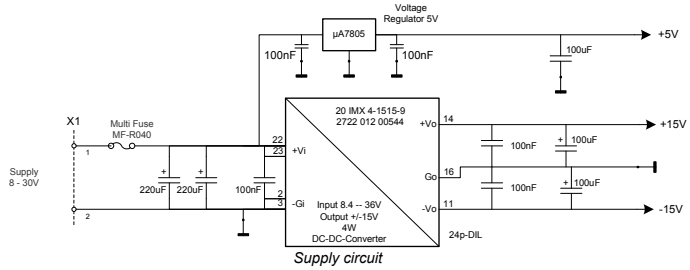
For schematics and source:
www.dlpdesign.com/dnlda











Supply circuit

Home Robotics
DLP-245PB Controller
Cor van der Klooster
2007-02-08
3_Axes_Omnidrive_Controller.vsd

GP2D12/GP2D15

General Purpose Type Distance Measuring Sensors

■ Features

1. Less influence on the color of reflective objects, reflectivity
2. Line-up of distance output/distance judgement type
   Distance output type (analog voltage) : **GP2D12**
   Detecting distance : 10 to 80cm
   Distance judgement type : **GP2D15**
   Judgement distance : 24cm
   (Adjustable within the range of 10 to 80cm)
3. External control circuit is unnecessary
4. Low cost

■ Applications

1. TVs
2. Personal computers
3. Cars
4. Copiers

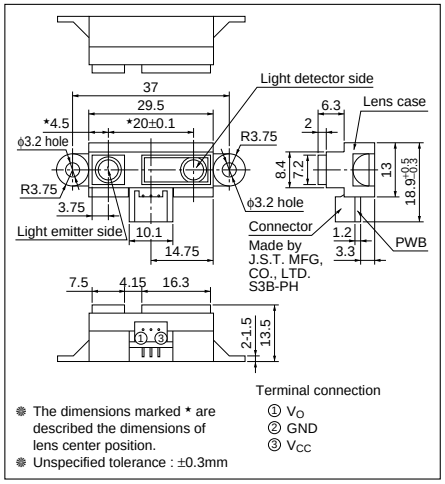
■ Absolute Maximum Ratings

(Ta=25°C, Vcc=5V)

Parameter	Symbol	Rating	Unit
Supply voltage	V_{CC}	−0.3 to +7	V
Output terminal voltage	V_O	−0.3 to V_{CC} +0.3	V
Operating temperature	T_{opr}	−10 to +60	°C
Storage temperature	T_{stg}	−40 to +70	°C

■ Outline Dimensions

(Unit : mm)



■ Recommended Operating Conditions

Parameter	Symbol	Rating	Unit
Operating supply voltage	V_{CC}	4.5 to +5.5	V

■ Electro-optical Characteristics

(Ta=25°C, V_{CC}=5V)

Parameter		Symbol	Conditions	MIN.	TYP.	MAX.	Unit
Distance measuring range		ΔL	*1 *3	10	–	80	cm
Output terminal voltage	**GP2D12**	V_O	L=80cm *1	0.25	0.4	0.55	V
	GP2D15	V_{OH}	Output voltage at High *1	Vcc –0.3	–	–	V
		V_{OL}	Output voltage at Low *1	–	–	0.6	V
Difference of output voltage	**GP2D12**	ΔV_O	Output change at L=80cm to 10cm *1	1.75	2.0	2.25	V
Distance characteristics of output	**GP2D15**	V_O	*1 *2 *4	21	24	27	cm
Average Dissipation current		I_{CC}	L=80cm *1	–	33	50	mA

Note) L : Distance to reflective object.

*1 Using reflective object : White paper (Made by Kodak Co. Ltd. gray cards R-27 · white face, reflective ratio ; 90%).
*2 We ship the device after the following adjustment : Output switching distance L=24cm±3cm must be measured by the sensor.
*3 Distance measuring range of the optical sensor system.
*4 Output switching has a hysteresis width. The distance specified by Vo should be the one with which the output L switches to the output H.

Fig.1 Internal Block Diagram

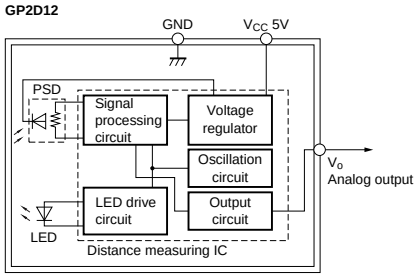


Fig.2 Internal Block Diagram

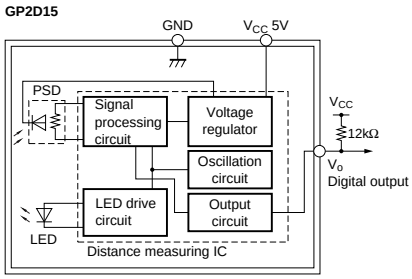
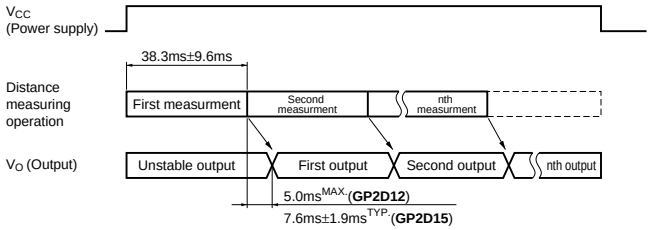


Fig.3 Timing Chart



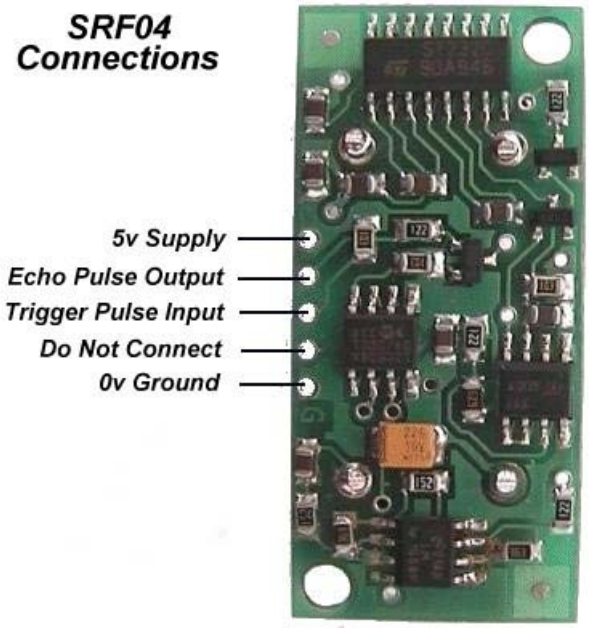
SRF04 - Ultra-Sonic Ranger

Technical Specification

This project started after I looked at the Polaroid Ultrasonic Ranging module. It has a number of disadvantages for use in small robots etc.

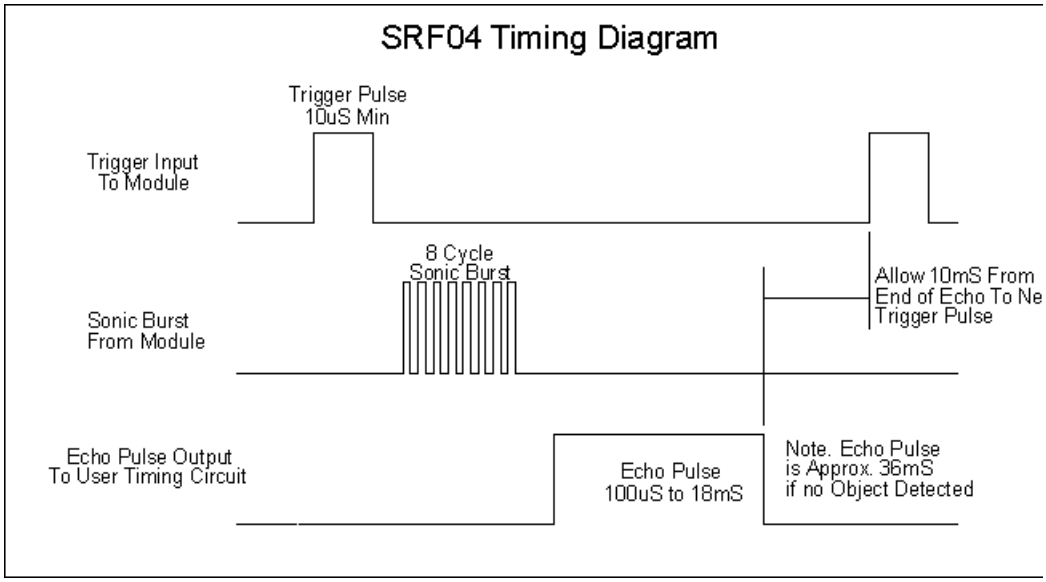
1. The maximum range of 10.7 metre is far more than is normally required, and as a result
2. The current consumption, at 2.5 Amps during the sonic burst is truly horrendous.
3. The 150mA quiescent current is also far too high.
4. The minimum range of 26cm is useless. 1-2cm is more like it.
5. The module is quite large to fit into small systems, and
6. It's EXPENSIVE.

The SRF04 was designed to be just as easy to use as the Polaroid sonar, requiring a short trigger pulse and providing an echo pulse. Your controller only has to time the length of this pulse to find the range. The connections to the SRF04 are shown below:

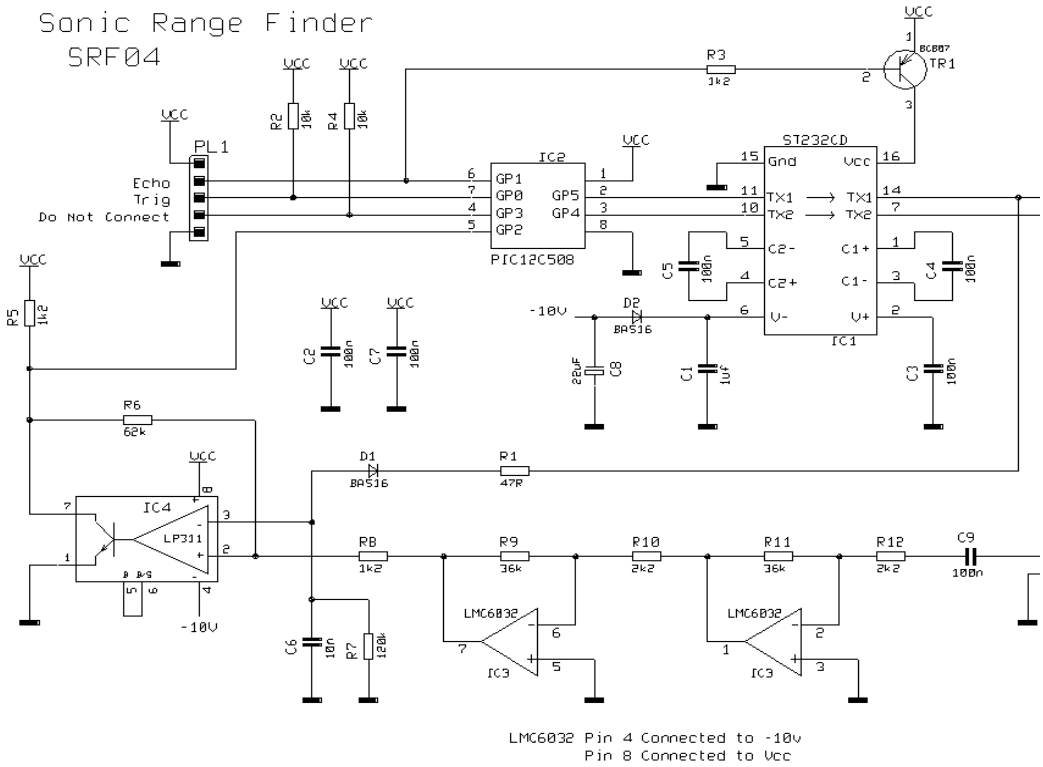


The SRF04 Timing diagram is shown below. You only need to supply a short 10uS pulse to the trigger input to start the ranging. The SRF04 will send out an 8 cycle burst of ultrasound at 40khz and raise its echo line high. It then listens for an echo, and as soon as it detects one it lowers the echo line again. The echo line is therefore a pulse whose width is proportional to the distance to the object. By timing the pulse it is possible to calculate the range in inches/centimeters or anything else.

If nothing is detected then the SRF04 will lower its echo line anyway after about 36mS.



Here is the schematic, You can download a better quality pdf (161k) version [srf1.pdf]



The circuit is designed to be low cost. It uses a PIC12C508 to perform the control functions and

Appendix C

Software

Appendix C.1 GUI

Software was written to control both the simulator and the real robot. In the section below it is explained how to compile the software in simulator, real life or logdata mode. In this section, the most important features are explained. Figure C.1 shows the basic Graphical User Interface (GUI). Figure C.2 is built in simulator mode and is, as the reader can see, almost the same, except for the window in the bottom left corner. This is the simulator map, which is actually just a bitmap file of 1500x1500 pixels. By pressing the "Print Grid" button after a few iterations of the algorithm, the map belonging to one particle will be printed on the large window on the right. To select which particle should be printed, click the drop down menu next to the print button, see figure C.4.

Figure C.3 shows the control center. To control the robot, click one of the arrow buttons. In simulator, a button click stands for 1 iteration. If the software is compiled for the robot, the robot keeps going in the chosen direction until the middle button, the stop button, is pressed. This also holds for the rotate left "L" and rotate right "R" buttons. The robot can follow a trail: when clicking in the large map at several locations the robot will go to those locations if the user presses the "Go To Target" button.

In real mode, all sensordata is logged. By pressing the "Save Log" button, this log is saved to file to the application directory. The "Toggle RT" but especially the "Toggle Graph" buttons will show interesting windows for testing the complete sensor array. They work in both real life mode and logdata mode.

Appendix C.2 defs.h

A lot of parameters have to be hardcoded, for example parameters defining the size of static arrays. All the hardcoded parameters can be found in one file: defs.h. The most interesting and important ones are listed here:

////////////////////////////////////

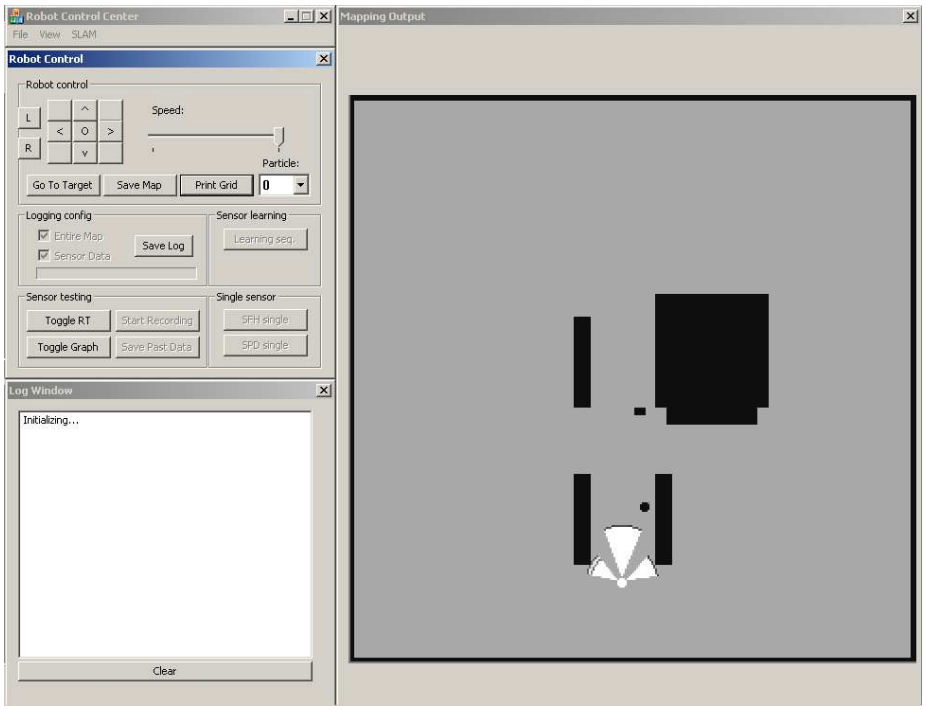


Figure C.1: Complete GUI.

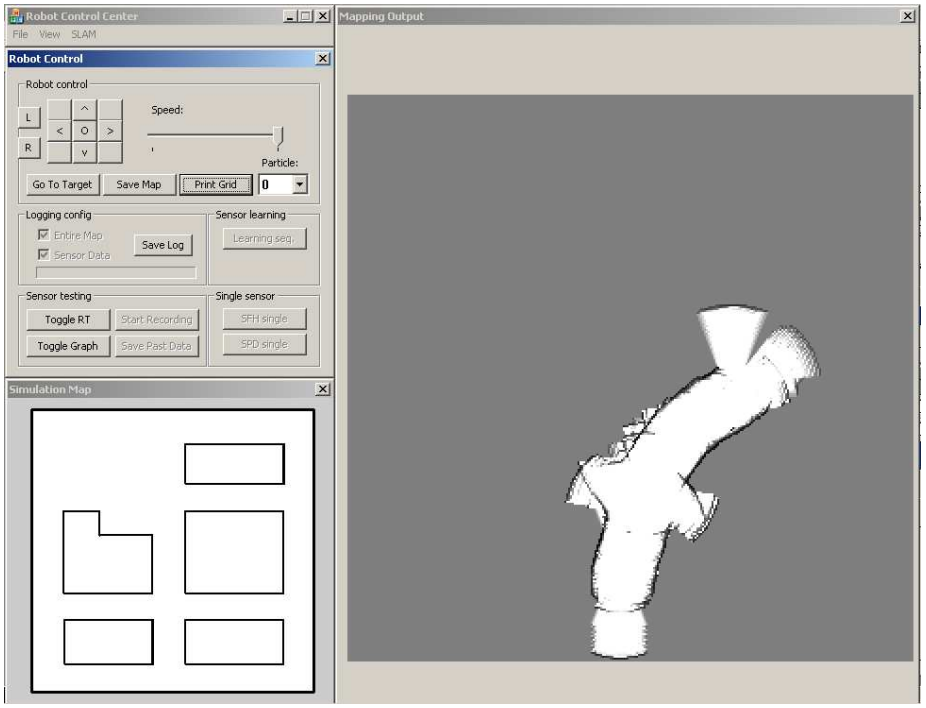


Figure C.2: GUI for simulator.

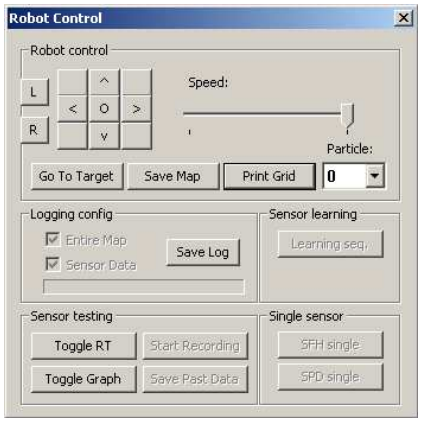


Figure C.3: Control of the robot or simulator.

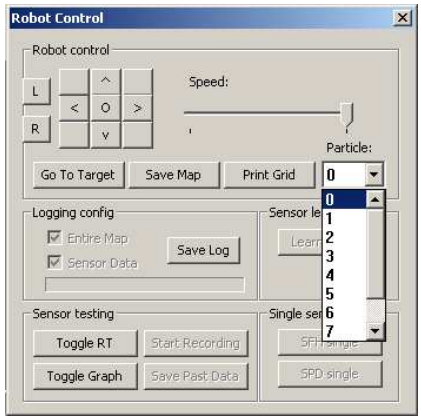


Figure C.4: Selecting the particle to print the map from.

```
// Flags
//

#define USELOGDATA       // These defines speak for
//#define SIMULATOR      // themselves. To activate
//#define REALTIME       // one, comment the others.


////////////////////////////////////
// Low level control
//

const unsigned char MOTORDEVICE[9]  = "DPBQMCX1";
const unsigned char SENSORDEVICE[9] = "DPCUMKY4";


////////////////////////////////////
// Basics
//

#define PARTICLES           20    // The total no of particles

#define TMR_MOVE_INTRV     300    // No. of ms between 2 motion commands
#define TMR_MEAS_INTRV     100    // No. of ms between 2 measurements

#define IR_VIEW_ANGLE        0.08 // Angle of the beam in rad.
#define IR_MIN_DIST          8    // Min and max measurement distance
#define IR_MAX_DIST         60    // Note: In sim this is actual maximum.
                                  //       In real, a > MAX is INF
                                  // These variables also exist for US


////////////////////////////////////
// Grid related
//
```

```
#define RESOLUTION          5    // Resolution in cm per grid square
#define GRIDCELLSX        300    // 1500/RESOLUTION // Note: Static 'cause
#define GRIDCELLSY        300    // 1500/RESOLUTION // they are array sizes

#define TWEAK_h             1    // For tweaking the opacity of grid cells.
#define TWEAK_d             3    // opacity = (TWEAK_d+d)/(TWEAK_h+h);

#define EMPTYCELL           1.0  // These 4 defines are for US. In defs.h
#define FULLCELL            1.0  // there are also defines for IR.
#define EMPTYCELLINF        0.2  // Used for tweaking relevance of certain
#define FULLCELLINF         0.2  // measurements. INF is less relevant.


#define GLOBALMAPFULL       7.0  // Relevance of the global map. A FULL
#define GLOBALMAPEMPTY      3.0  // cell in the global map equals 7 meas.


#define FADEFACTORFULL      0.98 // The amount of fading that takes place
#define FADEFACTOREMPTY     0.98 // every iteration of the algorithm.


////////////////////////////////////
// Noise related
//

const float ROBOT_US_DEV_EST   = (float)  0; // Particle's estimated noise
const float ROBOT_TURN_DEV_EST = (float)  0; // Particle's estimated noise
const float ROBOT_MOVE_DEV_EST = (float)  0; // Particle's estimated noise

const float ROBOT_US_DEV_SIM   = (float)  0; // % of US noise in sim
const float ROBOT_TURN_DEV_SIM = (float)  5; // % of turning noise in sim
const float ROBOT_MOVE_DEV_SIM = (float)  5; // % of motion noise in sim
const float ROBOT_US_BULL_FREQ = (float) 10; // 1/random measurements
```

Bibliography

[1] A. Eliazar and R. Parr, *DP-SLAM: Fast, robust Simultaneous Localization And Mapping without predetermined landmarks*, Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03), Morgan Kaufmann, 2003, pp. 1135–1142.

[2] D. Fox, *Markov localization: A probabilistic framework for mobile robot localization and navigation*, Ph.D. thesis, University of Bonn, 1998.

[3] J.S. Liu and R. Chen, *Sequential Monte Carlo methods for dynamic systems*, Journal of the American Statistical Association **93** (1998), no. 443, 1032–1044.

[4] D. Koller M. Montemerlo, S. Thrun and B. Wegbreit, *FastSLAM: A factored solution to the Simultaneous Localization and Mapping problem*, Proceedings of the AAAI National Conference on Artificial Intelligence (Edmonton, Canada), 2002, pp. 593–598.

[5] W. Whittaker M. Montemerlo and S. Thrun, *Conditional particle filters for simultaneous mobile robot localization and people-tracking*, IEEE International Conference on Robotics and Automation (ICRA) (Washington, DC), 2002, pp. 695–701.

[6] H.P. Moravec and A. Elfes, *High resolution maps from wide angle sonar*, Proceedings of the IEEE International Conference on Robotics and Automation, 1985, pp. 116–121.

[7] N. Gordon M.S. Arulampalam, S. Maskell and T. Clapp, *A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking*, IEEE Transactions on Acoustics, Speech, and Signal Processing **50** (2002), no. 2, 174–188.

[8] R. Negenborn, *Kalman filters and robot localization*, Master's thesis, Utrecht University, Utrecht, Netherlands, 2003.

[9] SRF04 technical specification Robot Electronics website, *http://www.robot-electronics.co.uk/htm/srf04tech.htm*.

[10] Thrun S, *Robotic mapping: A survey*, Exploring Artificial Intelligence in the New Millenium (G. Lakemeyer and B. Nebel, eds.), Morgan Kaufmann, 2002.

[11] W. Burgard S. Thrun, D. Fox and F. Dellaert, *Robust Monte Carlo localization for mobile robots*, Artificial Intelligence **128** (2000), no. 1-2, 99–141.

[12] S. Schwope, *Determination of limits and development of different sensors and algorithms for autonomous navigation*, Master's thesis, RWTH, Aachen, 2004.

[13] C. Stachniss and W. Burgard, *Mobile robot mapping and localization in non-static environments*, Proceedings of the AAAI National Conference on Artificial Intelligence (Pittsburgh, USA), 2005, pp. 1324–1329.

[14] H. Jans C. Matenar W. Burgard, D. Fox and S. Thrun, *Sonar-based mapping of large-scale mobile robot environments using EM*, Proceedings of the 16th International Conference on Machine Learning (ICML '99), 1999, pp. 67–76.

[15] C. Wang, C. Thorpe, and S. Thrun, *Online Simultaneous Localization And Mapping with detection and tracking of moving objects: Theory and results from a ground vehicle in crowded urban areas*, Proceedings of the IEEE International Conference on Robotics and Automation, 2003, pp. 842–849.

[16] Hamamatsu Website, *http://www.hamamatsu.com*.

[17] Sharp Website, *http://sharp-world.com/index.html*.

[18] SICK Website, *http://www.sickusa.com*.