ISSN 0280-5316
ISRN LUTFD2/TFRT--5803--SE

Obstacle Avoidance for Mobile Robots

Jan Edhner

Department of Automatic Control
Lund University
September 2007

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
September 2007

Document Number
ISRNLUTFD2/TFRT--5803--SE

Author(s)
Jan Edhner

Supervisor
Peter Alriksson and Karl-Erik Årzén (Examiner) at Automatic Control in Lund.

Sponsoring organization

Title and subtitle
Obstacle Avoidance for Mobile Robots (Undvikande av hinder för mobila robotar)

Abstract
As a part of the RUNES project a robot has been developed and it has as a part of this theses been improved with an obstacle avoidance component. Work has also been done to make room for additional components on the Tmote sky (one of the micro controllers mounted on the robot) such as a power control component developed at KTH. Attempts has also been made to try to enhance the performance of the robot. Finally a program has been created so that an operator can control the robot from a PC.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
43

Recipient's notes

Security classification

Preface

I would like to thank Karl-Erik Årzén for giving me the opportunity to do this interesting thesis and I would also like to thank Peter Alriksson for all the help, especially for all his invaluable help with Matlab. Thanks also goes out to Rolf Braun, who has built all the hardware.

Contents

1. Introduction

1.1 Background

This thesis is done as a part of the European Integrated Project Reconfigurable Ubiquitous Networked Embedded Systems (RUNES) [1]. The project is focused on sensor networks and one of the main goals is to *"provide an adaptive middleware platform and application development tools that allow programmers the flexibility to interact with the environment where necessary, whilst affording a level of abstraction that facilitates ease of application construction and use."*

The motivation behind this is that, as networked embedded systems grow they get more complex and therefor it is necessary to simplify things if the full potential of such systems are to be realized. This is why RUNES wants to develop aids to help developers simplify development of such systems.

RUNES Participants

The RUNES consortium includes the following 21 partners from 9 different countries

- Australia
  - National ICT Australia
  - University of Queensland
  - Victoria University
- Canada
  - Communications Research Centre Canada
- Germany
  - Industrieanlagen-Betriebsgesellschaft mbH
  - LiPPERT Automationstechnik GmbH
  - Rheinisch-Westfaelische Technische Hochschule Aachen
- Greece
  - University of Patras
- Italy
  - Politecnico di Milano
  - Università di Pisa
- Hungary
  - Ericsson
- Sweden
  - ConnectBlue AB
  - Ericsson AB
  - Swedish Institute of Computer Science AB
  - Kungliga Tekniska Högskolan
  - Lund Institute of Technology

- United Kingdom
  - Kodak Ltd.
  - Lancaster University
  - University College London
- United States of America
  - University of California, Berkeley
  - University of California, San Diego

RUNES demonstrator

To illustrate one potential application of the RUNES middleware in greater detail a number of different scenarios where evaluated [2]. The one that was selected was a disaster relief scenario. This scenario takes place in a road tunnel where fire has broken out due to a car accident similar to what happened in the Mont Blanc tunnel in 1999. The result of the Mont Blanc accident was a fire that lasted for two days trapping around 40 vehicles in poisonous smoke, with a death toll of 37 people. The firefighters in such a situation want to know as much as possible about the status inside the tunnel such as exactly where and how big the fire is, how many people that are trapped, if there are any toxic gases and so on. In the Mont Blanc case the fire fighters did not have this information and as a result they went in so late that they did not make any difference and as a result they got trapped themselves.

In the RUNES scenario information mentioned above (temperature, toxic gases etc) are transferred through a wired network in the normal case but has a wireless network as a backup in case the cables burn in the fire (much like what actually happened in the Mont Blanc accident mentioned above). If the fire is serious enough it might also be possible that a part of the wireless sensors might get damaged as well. If that occurs the following might happen; one group of sensors (those that are close to the control room) are intact, one group (close to the fire) are damaged and can therefor not send data back and finally a third group that rely on the broken part of the network to send data to a control room. This means that we are missing parts of vital information that might possibly save lives. This is where a robot is sent in to act as a mobile sensor node so that the third group of sensors just mentioned can use the robot to route messages back to the control room. The robot can also send back data directly from the area that was covered by the now damaged nodes. Now the fire fighters can get complete information about the situation again and act according to that information.

1.2 Other Control Problems

Aside from the problem with localization (that is knowing where the robot is) and obstacle avoidance there are a number of control problems associated with the RUNES demonstrator; first you need to know where the robot is supposed go, this is solved with the network reconfiguration component (NetReC) [13] that uses the position of the nodes (that is, the software knows what nodes that report and those that does not) to calculate the optimal position of the robot.

An other component that is required is the Collision avoidance component [14]. It makes sure that robots will not collide with each other. The collision avoidance component is based on a reserved disk associated with each robot (meaning that the other robots knows the position of the other robots and the radius of the reserved disk for each robot).

The final component is the power control component [8]. It uses the received signal strength indicator (RSSI) to control the power level of the sensor nodes so the sensor nodes does not always send with maximum transmission power thus preserving essential battery power.

1.3 Purpose

The demonstrator (see above) was implemented and shown in a lab at Ericsson at two occasions. On the first occasion for reviewers and at a second time for the public (Ericsson staff, a few firefighters for example). The goal of this thesis is to improve the mobile robot that was developed as a part of [3] so that the robot can perform the tasks that are described in the RUNES demonstrator at the two demonstrations just mentioned.

The improvements include obstacle avoidance so that the robot can avoid cars, lorries and other material that is in its way due to the accident. It also involved writing a program that enables an operator to give the robot commands from a PC. Another thing that needs to be done is freeing up space on the sensor node so that different components developed by other RUNES partners (this includes for instance a power control component [8]) fits on the Tmote.

Another purpose of this thesis is to try to improve on the performance of the robot by replacing the angle estimation with a compass as was suggested as future work in [3].

1.4 Limitations

This thesis is limited to software since all the hardware needed was built by Rolf Braun, who works at the department as a research engineer.

1.5 Previous work

This is the second master thesis done at LTH as a part of RUNES. The first thesis focused on localization with ultrasound and the result from that work was a robot that could navigate with stationary ultrasound nodes positioned along its path. There was however only a very basic navigator and the only way to tell it where to go was to hard code the desired position.

2. Current System

This section describes both the hardware and software that was the basic of this thesis.

2.1 Hardware

Tmote Sky

The wireless sensor nodes are from a company called Moteiv [4] and are called Tmote Sky. The Tmote is equipped with a Texas Instruments MSP-430 microcontroller (with 48kbyte flash and 10kbyte RAM). It also features an onboard IEEE 802.15.4 Chipcon Wireless Transceiver.

The robot

There are two robots available that are basically the same. One big difference is that one of the robots can not only carry an extra mote on the back but can also drop it on the ground with a tipping mechanism (useing a servo to make it go up/down). Other than that one robot is slightly wider and higher than the other.

The robots are of dual-drive unicycle type and have two independently driven wheels and an AVR ATmega16 is connected to each wheel. The AVR's that are connected to the wheels measure and control (with a PI-controller) the speed of each wheel. There is also an additional ATmega16 that controls an ultrasound transmitter. That transmitter has a cone mounted on top of it to allow the ultrasound to spread 360° around the robot.

An ATmega128 was later included as a computation engine because of the limited capacity of the MSP-430 (both in terms of memory and cpu power).

Finally there is a Tmote that among other things is used for radio communication. It also contained the Extended Kalman filter (the filter was however later moved, see below for details) that is used to calculate the position and the angle of the robot. A navigator was also included on the Tmote that outputs a speed and an angle that is used as input to a PD-controller that sends a reference speed to the wheel controllers. Since the navigator was updated to include obstacle avoidance it was moved to the Atmega128 (for more information on different controllers and other software see below). Figure 2.1 below shows what one of the robots looks like.

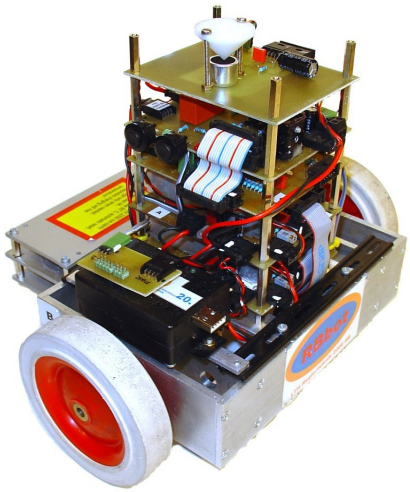

Figure 2.1: One of the robots

Ultrasound nodes

The Ultrasound nodes are basically Tmote Sky cards with a custom casing built at LTH. There is also an ultrasound receiver mounted on the Tmote. The card that holds the receiver is also custom built at LTH. There are currently eleven of these nodes available and they are placed at known positions throwout the course where the robot drives (more on how the robot determines its position from these nodes below). The image shows how one of the ultrasound nodes looks like

Figure 2.2: An ultrasound node

2.2 Controllers

The following image is an overview of the different controllers and how they interact in order for the robot to move to the desired destination.

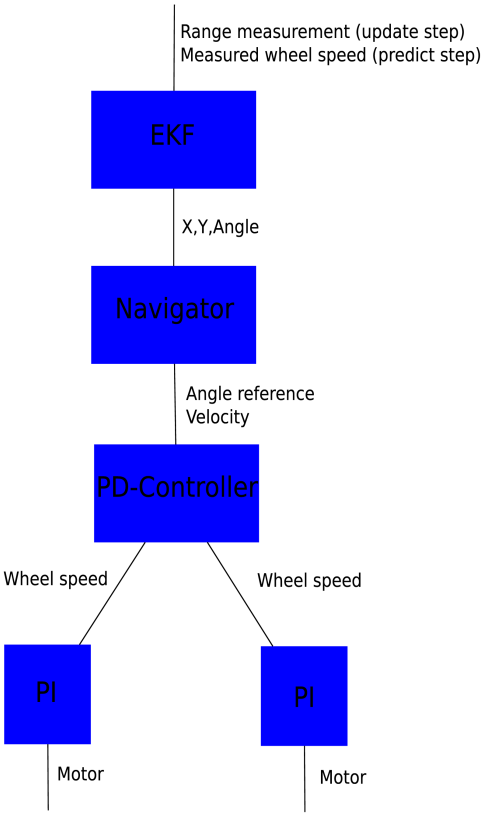

Figure 2.3: This image shows how the controllers are connected

The range measurements are gathered in the following manner; the robot sends out a radio broadcast and an ultrasound pulse at the same time[1] and when the nodes receive the radio (which happens immediately) it starts to count and when the ultrasound is detected it stops. Since the speed of sound is known we now have a distance between the robot and the stationary node. When a distance has been calculated the node sends the data back together with it is own (hard coded) position. That data is then used as an input to the Kalman Filter (see below).

The Extended Kalman Filter (EKF)

The model for the robot that the Kalman Filter uses is

1 Actually it's not at the exact same time, there is a 2ms delay between the radio and the ultrasound pulse.

$$\frac{d}{dt}\begin{pmatrix} x \\ y \\ \Theta \end{pmatrix} = \begin{pmatrix} \frac{1}{2}\cos(\Theta)(u_a + u_b) \\ \frac{1}{2}\sin(\Theta)(u_a + u_b) \\ \frac{1}{d}(u_a + u_b) \end{pmatrix}$$

where x and y are coordinates, Θ is the angle (where 0° means right, 90° is straight ahead and so on), d is the width of the robot and u_a,u_b are the velocities of the two wheels. Because of the low sample rate Tustin approximation is used in order to ensure correctness.

The model used in for the update step of the Extended Kalman Filter (originally defined in [3]) looks like this.

$$\tilde{y}_k = z_k - h(\hat{x}_{k|k-1}, 0)$$
$$S_k = H_k P_{k|k-1} H_k^T + R_k$$
$$K_k = P_{k|k-1} H_k^T S_k^{-1}$$
$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k$$
$$P_{k|k} = (I - K_k H_k) P_{k|k-1}$$

One of the challenges with this model is when calculating the Kalman gain (the K-matrix). This step involves calculating S^{-1} and when you get n measurements (in this case n range measures) you need to invert an n x n matrix (in this case that can easily mean that you need to invert a 5x5 matrix). In order to avoid that you can instead do the update step n times. By doing so you only need to invert a scalar each time you do update instead. This is the solution used on the robot.

The Robot Controller (PD-Controller)

The speed of the robot can be viewed as the average velocity of the two wheels and the angle velocity is proportional to the difference in velocity of the two wheels . It is therefore natural to design two controllers, one for the velocity and another for the angle where the output of the angle controller is added and subtracted from the two wheels respectively.

PI-controller

The last step is a PI-controller that given the velocity for that wheel controls the motor accordingly.

2.3 Other Software

Contiki

The RTOS that is used on the Tmote Sky is called Contiki [5] and is developed by SICS (Swedish Institute of Computer Science). Contiki is a light weight OS that contains protocols for upd/tcp communication.

It also features a special kind of processes called Protothreads [6]. The reason for using Protothreads is because of the small overhead that they require. Another feature of the protothreads is that there is no preemption which means that you have to manually define when the thread can be preempted. This ensures that the needed timing constraints are met. It is not always an advantage as it requires an higher level of discipline from the programmer as the thread will not release the processor unless it is explicitly told to by the programmer.

I^2C

The Tmote acts as a Master on the I^2C [7] bus while all the other micro controllers are slaves. The I^2C protocol allows multiple masters but due to the fact that the I^2C shares pins with the radio the Tmote has to initiate all I^2C communication so it does not confuse I^2C with radio communication. This causes a few problems; the main one is that since only the master can initiate communication over the bus it has to, for instance, periodically ask one of the slaves for a new direction (since the navigator which calculates the angle and the speed is on not on the Tmote) instead of being told when there is any change.

Timing

A very important part of any embedded system is the timing. On the robot one cycle is 1.2 seconds long and is divided into three subsections as shown by the figure below.

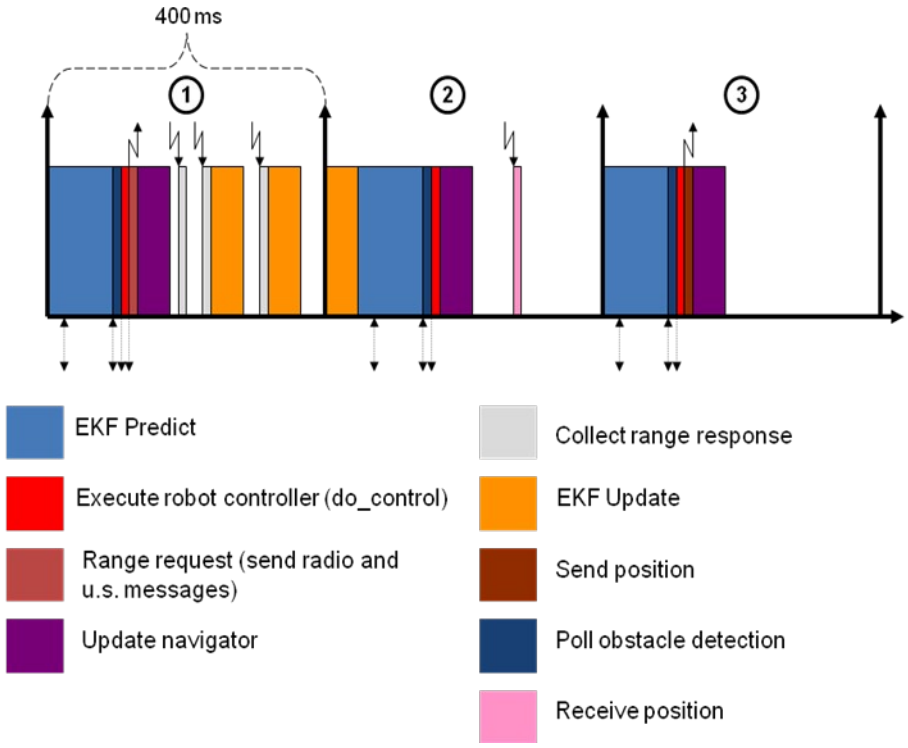

Figure 2.4: Image describing timing. Taken from [3]

The first thing that happens is a Predict followed by an ultrasound pulse. As those come back as range measurements (see Chapter 2.2 for details) Update is done with that measurement (together with the information about the position of the node). Note that the last update is left for the second subsection. This is because there are a lot of spare CPU power here whereas there is almost none to spare in the first subsection. This motivation is not true now with the ATmega128 introduced as a computation engine but that is the original motivation to keep one update for the second subsection.

3. Initial problems

3.1 Experiments

One of the first concerns was that if there is a lot of other traffic on the network the performance of the localization would be unacceptable. Therefor two experiments where done; one to check how many responses the robot got from the ultrasound sensors without any noise and a second to see how the robot behaved with background background traffic.

Updates

The first experiment that was done counted the number of responses that the robot got from the ultrasound nodes after sending out a radio/ultrasound pulse. The histogram shows the number of responses each time cycle.

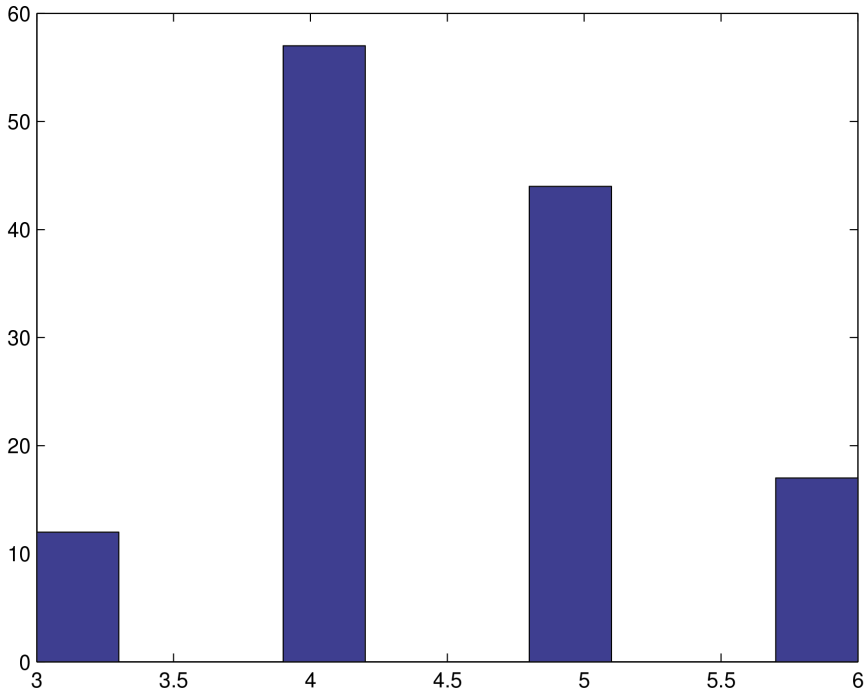

Figure 3.1: Histogram showing number of replies each time cycle

The results can be explained by the fact that the ultrasound nodes do not reply at the same time but delays the response according to the formula

$$delay = 8(ID \bmod 5)$$

where ID is between 1 and 7 (since there are seven nodes). The delay is therefore between 0 and 32 which mean the maximum delay is 32/256 = 125 ms (this since in Contiki a second is divided into steps that are 1/256 in size). This should make 5 the maximum number of responses but since there are a lot of random factors involved such as the fact that it is impossible for two nodes to start the delay at the exact same time, so sometimes we see more than five. Another reason is that the radio might not reach the ultrasound nodes due to for example fading. Therefore we might not always get 5 updates even in the best conditions.

Background traffic

A program was created that sent out background traffic via the radio to see what sort of impact this has on performance. The program was installed on each ultrasound node and was configured to broadcast two (empty) packets each second (with 7 motes this makes 14 packets/second in total). This was done since there will be a lot of unrelated (of no concern for the robot) traffic in the final demo and if the performance turned out unacceptable with the amount of background traffic that was expected something might have to be done. The resulting histogram looked like this (as above it is the number of range responses each time cycle).

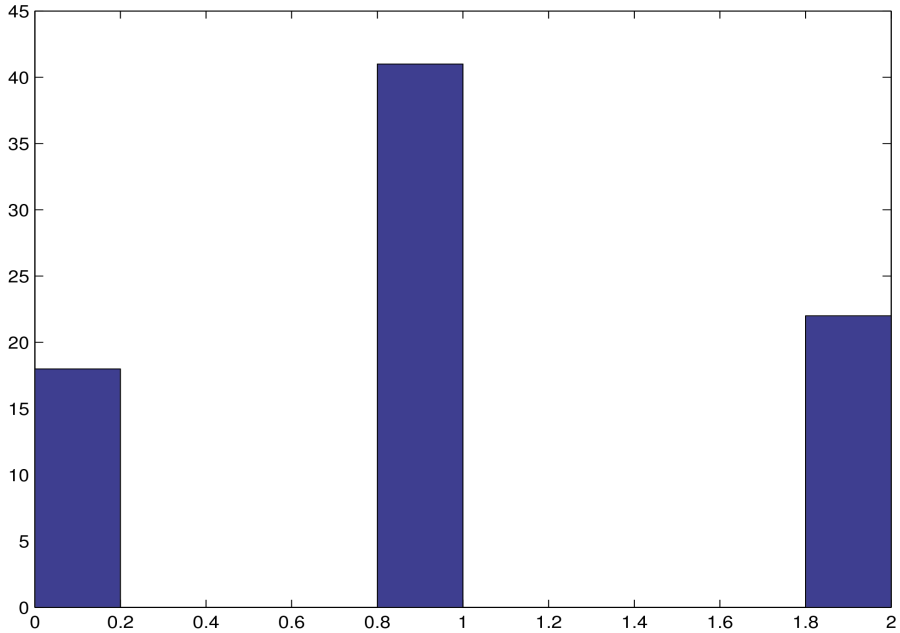

Figure 3.2: Number of updates each time cycle with background noise

The amount of traffic generated by the test program is quite a bit more than the unrelated traffic in the final demo but it is quite clear that the performance is such that something needs to be done in order to reduce unnecessary traffic. The solution to this problem is to use separate channels; one for the localization and another for the rest of the traffic. This ensures that all vital information reaches it's destination and that the localization performance is adequate.

3.2 Space issues

As already mentioned there is only 48 kbyte flash memory on a Tmote. The initial code that was used as a base for this thesis took 46 kbyte(including Contiki). Since there are a lot of other components that are supposed to fit on the Tmote (such as a power control component [8]) this obviously is a major issue that needs to be resolved. The solution was to mount an ATmega128 on the robot and move some of the code to it. The first part that was moved was the Kalman filter. Later the navigator was moved from the Tmote as well since it had to be rewritten to also include obstacle avoidance (see below for details). After the components mentioned was moved, 10kbyte was removed from the Tmote, more than enough to fit the rest of the components.

I2C protocol

When moving the Kalman filter to the ATmega128 an I^2C protocol had to be created in order to move data back and forth between the two micro processors. The data that had to be sent included the range measurements from the ultrasound nodes, the wheel velocities and obviously the result (x,y and the angle) has to be sent back. The protocol has the following structure.

1. The command that is to be performed or what type of data that is expected.
2. The number of bytes
3. The data (if the command is for example predict then the data is the wheel velocities)

3.3 The angle estimation

One of the conclusions of [3] was that the angle estimation had an accuracy of approximately 15°. To improve that a compass was mounted on top of the robot. Since the compass uses the magnetic field to get the compass direction this causes a problem due to the fact that the robot is mainly driving indoors and that means a lot of iron in the walls (radiators for example) etc. that interfere with the magnetic field. As a result, the compass has a systematic error of 20° or more in the lab, that is far worse than the estimation meaning that the compass is basically useless. A possible solution could be to replace the compass with a gyrocompass since it doesn't rely on a magnetic field but instead “uses an (electrically powered) fast spinning wheel and friction forces in order to exploit the rotation of the earth” (according to [11]).

4. Java program

A small command based java program was created so that an operator can control the robot from a PC. The commands was sent as udp-packets to the robot. This can be done if you configure a Tmote as a gateway and running a program called Tunslip (developed as a part of Contiki). What Tunslip does is to create a network device (much like eth0) and then Linux will use that to send data if the destination is one of the motes.
A protocol for this exists as a part of RUNES but considering the small amount of data being sent plus the fact that the program is not going to be integrated into any larger system this protocol was not used.

The features of the program include the following

- Sending the robot to a specific position
- Changing the radio channel that the robot sends/receives data on
- Rotating the robot (only when it is stopped)
- Dropping the “extra mote” from the tipping mechanism (see above for more information)
- Set the position on the ultrasound nodes.
- Emergency stop.

5. Obstacle avoidance

This chapter will present different obstacle avoidance methods and a brief motivation for the reason behind the choice of method. A more thorough description of the chosen method follows as well as a description of the used hardware and finally some implementation details followed by results.

5.1 Different methods

This section will describe different approaches to obstacle avoidance, such as the bug algorithm, a (simple) potential field method and the vector field histogram method. For more details on the different methods see [12]

The bug method

The bug method [9] does a full contour around the obstacle trying to find the point of minimum distance to the target. The robot then goes around the obstacle again until it reaches that point where it again moves towards the goal. This is a very slow method but it ensures that the robot will reach the goal.

There is an improved version on this method that starts by drawing a line between the starting point and the goal. When it detects an obstacle the robot goes around the obstacle until it reaches the line again. Then it continues towards the goal.

The vector histogram method

The vector histogram method [10] can be summarized in these three steps:

1. Create a 2D Cartesian histogram grid from each range sensor measurement
2. From the histogram created in step 1, consider a window around the robot and filter that grid onto a 1D polar histogram
3. Calculate the angle and velocity from the 1D polar histogram, as a result of an optimization procedure

Potential field method

One can see the robot as a particle that moves in a potential field generated by the goal and the obstacles. The goal generates an attractive potential and the obstacles generates a repulsive potential. A potential field can be viewed as an energy field so the gradients at each point is a force. The robot driving in the field is subject to a force driving it to the goal and at the same time forces that keeps it away from the obstacles.

Using The Dijkstra Algorithm

Together with a map (see Chapter 5.4 for an example) the Dijkstra algorithm can be used in the following manner; when the robot starts the algorithm runs and since no obstacles has been detected the result is go straight towards the goal. But when an obstacle is detected the corresponding vertexes are removed and Dijkstra runs again. If the obstacle detected is in the way of the goal the algorithm will get a path around it and the robot will avoid the obstacle.

Choosing a method

The obstacle avoidance algorithm that was implemented is a potential field method, meaning the robot sees the obstacles as a repulsive potential fields and the goal as an attractive field. The reason for choosing this algorithm are for two reasons; first because of the simplicity of the algorithm but also because it is the choice of the other universities that are developing robots as a part of RUNES.

5.2 Potential field method

Let $q=(x, y)$ represent the position of the robot, then the field where the robot moves is a scalar function U(q), where

$$U(q)=U_{att}(q)+U_{rep}(q)$$

that can also be written as

$$U(q)=U_{att}(q)+\sum U_{repi}(q)$$

That is the total repulsive potential is the sum of the individual repulsive potentials. The force that drives the robot is the negative gradient of the potential, that is

$$F(q)=-\nabla U_{att}(q)-\nabla U_{rep}(q)$$

where F(q) is a vector that points in the direction that the robot should have.

The attractive potential

The attractive potential is proportional to d_{goal}^2 , where d_{goal} is the euclidean distance from the robot to the goal. The function is as follows:

$$U_{att}(q)=\frac{1}{2}k_{att}\cdot d_{goal}^2$$

From that a gradient pointing from the goal to the robot can be calculated. The result is

$$\nabla U_{att}(q) = k_{att}(q - q_{goal})$$

The direction that the robot wants to go in can be viewed as a force F_{att}=-U_{att}, that is

$$F_{att}(q) = -\nabla U_{att}(q) = -k_{att}(q - q_{goal})$$

The repulsive potential

The repulsive potential is stronger when the robot is closer to the obstacle and will decrease as the distance between the robot and the obstacle increases. In this implementation the repulsive potential is the mean value of the repulsive effect of all the obstacles, i.e.,

$$U_{rep}(q) = \frac{1}{n}\sum U_{repi}(q)$$

Obstacles that are far away from the robot will not likely effect how the robot should move and as the robot gets closer the repulsive potential should increase. A possible repulsive potential for the obstacle *i* that takes this into account is

$$U_{repi}(q) = \begin{cases} \frac{1}{2}k_{obst}\left(\frac{1}{d_{obsti}} - \frac{1}{d_0}\right)^2 & if\ d_{obsti} < d_0 \\ 0 & if\ d_{obsti} \geqslant d_0 \end{cases}$$

where d_{obsti} is the euclidean distance between the robot and the obstacle *i*, k_{obst} is a constant and d_0 is the influence threshold.

The negative gradient of the repulsive potential, $F_{repi}(q) = -\nabla U_{repi}(q)$ is given by

$$F_{repi}(q) = \begin{cases} k_{obst}\left(\frac{1}{d_{obsti}} - \frac{1}{d_0}\right)\frac{1}{d_{obsti}^2} - \frac{q - q_{obst}}{d_{obsti}} & if\ d_{obsti} < d_0 \\ 0 & if\ d_{obsti} \geqslant d_0 \end{cases}$$

Drawbacks

The main drawback with this method is the sensitivity to local minima, usually caused by symmetry in the environment or concave obstacles. Another problem is oscillatory behavior in narrow spaces.

5.3 Hardware

The sensor used is an IR-sensor from Sharp (GP2D12) that is mounted on a Futaba FP-S17 servo. In order to find obstacles the servo moves approximately 1.6°. Then the sensor takes three measures and sends the median of those to the obstacle avoidance. The reason for doing this is to reduce the variance of the measures. After sending the measured value (if any) there is a delay of 17 ms (see below for details on the delay). After that the servo moves another step where the process is repeated. The servo has a 140° field of view meaning it takes approximately 1.5 sec for the servo to go from one endpoint to the other.

The servo

The servo is controlled by generating a 50Hz square wave and then in order to move it to a specific position you simply manipulate the duty cycle accordingly (see below). On the AVR this is done by first setting a register to a value according to the formula (where f_{oc} is the frequency of the external oscillator and f is the desired frequency).

$$\frac{f_{oc}}{8f} - 1 = \frac{14{,}7456 \cdot 10^6}{(8 \cdot 50)} - 1 = 36\,863$$

This will create a 50Hz pulse. If you then want, for example, a 10% duty cycle you simply set a second register to 10% of the value from the first one (in this case 3686).The servo used has the following relation between angle and duty cycle.

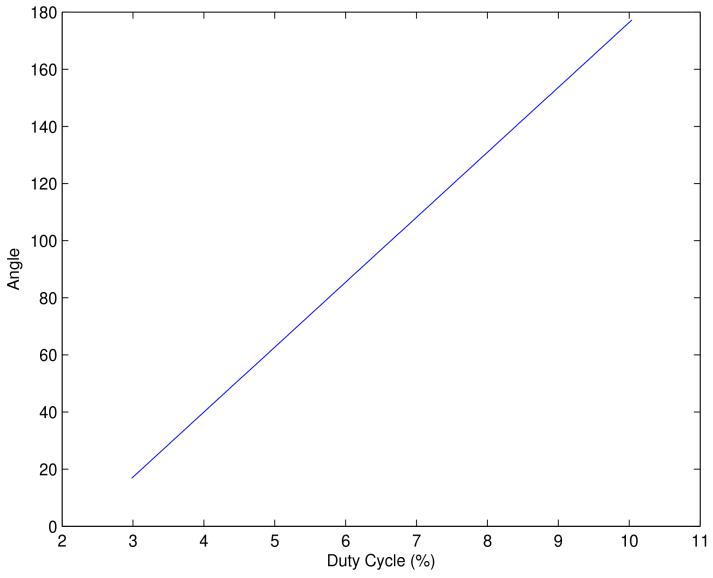

Figure 5.1: Relation between Angle and duty cycle

The sensor

The relation between the voltage (that is the output from the sensor) and the distance to the obstacle is exponential and has the following shape.

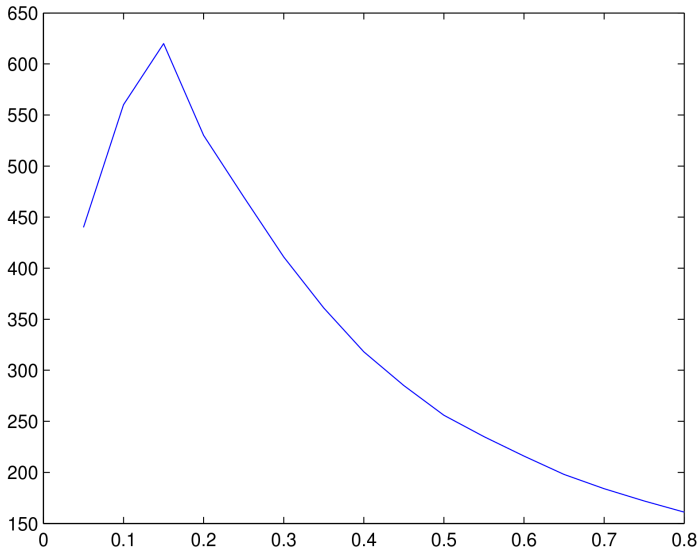

Figure 5.2: Relation between voltage and distance. The x-axis is represents range in meters and the y-axis is Output from the A/D converter

Notice that when you get closer than approximately 0.2m the voltage goes down again meaning that the sensor can mistake obstacles that are really close as further away than they really are.

When approximating a function the data closer than 0.2 m was disregarded, and then linear regression was applied with help of the fact that

$$y = Ce^{Ax} \Leftrightarrow \ln(y) = \ln(Ce^{Ax}) = \ln(C) + \ln(e^{Ax}) = \ln(C) + Ax$$

Drawbacks with the hardware

There is one major drawback with the servo; you can only tell it where to go, not measure where it actually is. The effect of this is shown in the two images below where the first one shows how the robot thinks a box looks like when the servo sweeps back and forth with a frequency of 0,5 Hz and in the second image the servo sweeps with 0,05 Hz.

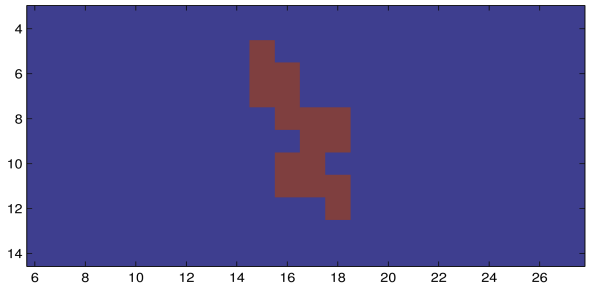

Figure 5.3: An obstacle when servo sweeps with 0,5 Hz

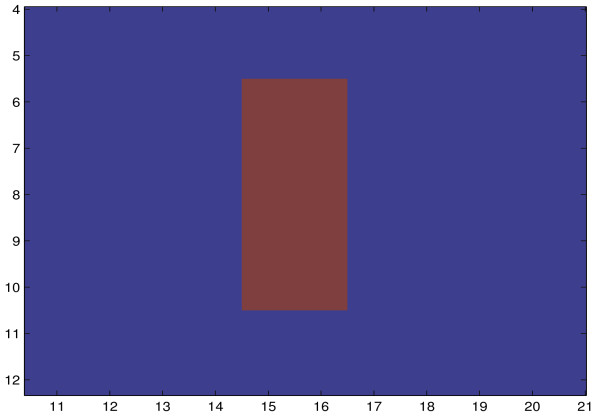

Figure 5.4: An obstacle when servo sweeps with 0,05Hz

What is clear by these images is that when you do not wait long enough the servo hasn't reached the desired position and the image of the box (the actual measurements of the box is 30cm x 20cm) is wrong and will result in odd behavior when navigating past the obstacles. On the other hand one can not move the servo too slow since then you have to compensate that by slowing down the robot (otherwise there is no chance to detect the obstacles) so there is a trade off there; accuracy of the map versus the speed of the robot.

5.4 Implementation Details

The map

There is a need to remember the obstacles so the robot will not forget them when it turns in such a manner that the sensor can not see the obstacle any more. This has been solved by implementing a simple map of the area that the robot drives in. The implementation uses a matrix where every entry is a flag that is 0 if no obstacle has been detected in that position and 1 if it has. If an obstacle is detected at position x,y the entry i,j in the matrix is set to 1, where i and j are defined by

$$i = \frac{x \cdot 100}{Resolution} \quad j = \frac{y \cdot 100}{Resolution}$$

then you simply round off i and j to an integer in order to get the correct values. Experimental results determined that a 10 cm resolution gave adequate results. It was also determined that there were enough memory left on the AVR to fit the entire map with a 10cm resolution.

The advantages of a matrix are many. First there is the simplicity, then there is the fact that it takes O(1) to read and write. Locating all obstacles on the map takes O(wl) where w is the width and l the length. This can however be improved by not searching for obstacles on the entire map but instead only search a window on both sides of the robot (the entire width has to be searched however). Experiments was done to determine that a window of 1,2 m (that is 0,6 m in front and 0,6 m behind the robot) was sufficient. The drawback is mainly the fact that it requires quite a lot of memory. This is however reduced by the fact that every entry only requires 1 bit. Also the fact that the robot will only drive in a very limited space (about 2x10 metes) helps reduce the required memory.

5.5 Results

The result of the obstacle avoidance is shown in the two images below (the blue dots represent obstacles and the green lines are repulsive gradients at each point)

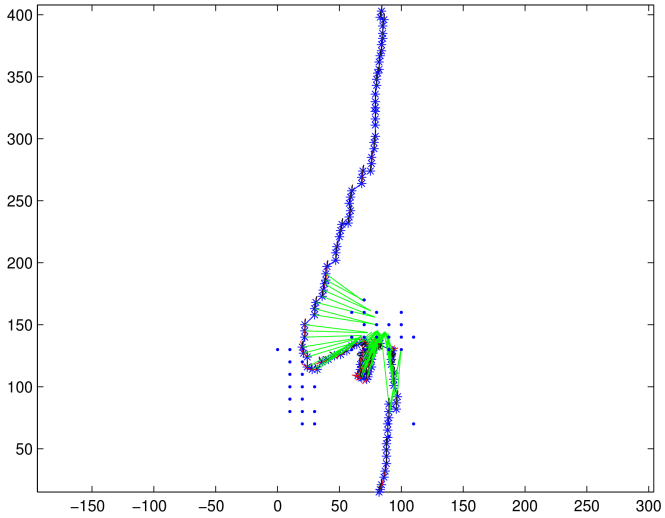

Figure 5.5: Problems with some oscillatory behavior

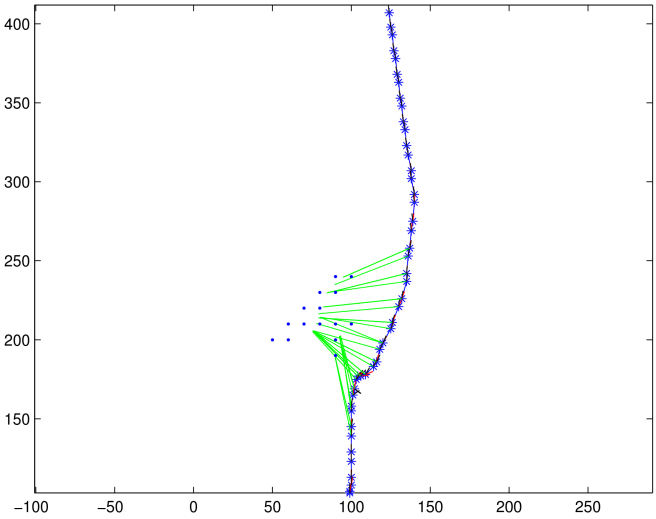

Figure 5.6: Good result when the obstacle is slightly to the side

Figure 5.5 shows one problem that still remains. As can be seen there is a lot of oscillatory behavior before the robot get passed the obstacle. The exact reason for this is unknown but a guess is that when there is an obstacle placed right in front of the goal like that the resulting gradient will point backwards and the robot will simply turn around until it reaches a point when the obstacle no longer has any impact and it will turn back towards the goal etc. On the other hand when the goal is slightly to the side, as is shown in Figure 5.6, the performance is a lot better.

6. Practical Experience

6.1 Bridging problems

A part of the scenario (see Chapter 1.1) is to use the robot to bridge two isolated islands of a sensor network. The first few attempts to do so was however unsuccessful. The first thought was that there where some problems with the fact that there is a plastic casing built around the mote that is mounted on the robot but that was ruled out when we tried remove the casing on the mote. Since the network was successfully bridged with a separate node and since the casing did not seem to make a difference the conclusion was that there was some sort of software problem. After some testing a bug was found in the i2c protocol; on the ATmega128 the number of bytes that was about to be sent was stored as an uint8_t but on the tmote stored as an int. Because of this (for reasons still not known) sometimes the data was interpreted wrong and was set to 127 instead of 12 (it was always supposed to be 12 since the problem was isolated to one function). This caused a memory leak (since the buffer the data was stored in was not 127 bytes long). When changing the data type on the mote from an int to an uint8_t that problem was solved but the problems with the bridging still remained.

A hardware problem with the mote that was mounted on the robot was found doing the following experiment; a mote was placed 1 m away from the robot acting as a gateway, then ICMP ping messages was sent from the PC through the gateway mote. After that the RSSI value for the last package was measured on the robot. The result was an RSSI value around -40. Since the uAODV implementation considers a route to poor (and therefor discards it) at -41 there was clearly a problem with the radio on the mote mounted on the robot. This was confirmed again when doing the same experiment with a different Tmote mounted on the robot. The RSSI value then changed to about -10.

For this reason a new Tmote was fitted on the robot and with that together with the memory leak fixed the problem with the bridging. There was however one problem remaining; “the tower” on the robot seemed to have some sort of impact on the radio performance. This was solved by first including a possibility to rotate the robot with a command sent from the Java program. The second part of the solution was to add a second mote on the back of the robot and have a tipping mechanism controlled with a servo connected to the ATmega128. Just before stopping the robot then dropped the extra mote. This gave the sensors on the other side of the damaged part of the network a backup route back to the control room in case the robot stopped in such a way that “the tower” was in a bad position.

6.2 The Final Demo

At the final demonstration an obstacle course was set up for the robot to navigate through. This was done as a part of the RUNES demonstrator (see Chapter 1.1) and therefore the obstacles were model cars and a lorry. This obstacle course was to represent rubble caused by the accident. Apart from the obstacle course the entire lab was transformed into a mini tunnel (including a road made from carpet with white a tape marking the middle).

The obstacle course had to be laid out in such a way so the robot could navigate it despite the flaws in the implementation (see Chapter 5.5 for details). Therefor no obstacles were placed in such a way that oscillatory behavior as shown in Figure 5.1 occurred.

There were still some problems where the biggest one was that sometimes the robot just drove in the direction it was headed to (as if it stopped asking the obstacle avoidance component for new directions). The reason for this is still unknown but it is most likely a software bug (possibly another memory leak). Another problem we had when the demo was shown to the public was that a lot of people had cameras with them and cameras use infrared technology to determine distance to an object. This infrared light was mistaken as an obstacle so the robot had to be “manually reset” (that is carried to it starting position again).

7. Conclusions

The performance of the obstacle avoidance is far from perfect for a number of reasons. First the hardware; since you can not measure the position of the servo it is very hard to get any accuracy, second the shortcomings of the algorithm makes it hard to navigate through complex obstacles. However the robot made it through the rather simple obstacle course that was used at the RUNES final demo and the robot managed to bridge the two isolated islands created in the network successfully so in that regard the result was a success. This, since the RUNES project is not focused on control and robotics but rather on sensor networks.

The problems with the obstacle avoidance could be solved using a more advanced algorithm like maybe something based on the Dijkstra algorithm.

Due to the fact that the compass did not work in any of the labs we tested there was not any improvement in localization performance of the robot.

8. Bibliography

[1] RUNES, Reconfigurable Ubiquitous Networked Embedded Systems. Webpage, http://www.ist-runes.org

[2] RUNES Deliverable 2.1 "Application scenario building/definition". Webpage, http://www.ist-runes.org/docs/deliverables/D2_01.pdf

[3] Jerker Nordh, Ultrasound-based Navigation for Mobile Robots. Master thesis, Dept of Control, Lund Institute of Technology, Lund university, 2006

[4] Moteiv corporation. Webpage, http://www.moteiv.com/

[5] Adam Dunkels, Björn Grönvall, Thiemo Voight. Contiki – a Lightweight and Flexible Operating System for Tiny Networked Senors. In Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, November 2004

[6] Adam Dunkels, Oliver Schmidt, Themio Voigt, Muneeb Ali. "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems."

[7] I2C. Webpage, http://en.wikipedia.org/wiki/I2c

[8] Benigno Zurita Ares, Carlo Fischione, Alberto Speranzon, Karl Henrik Johansson, "On Power Control for Wireless Sensor Networks: System Model, Middleware Component and Experimental Evaluation"

[9] V. Lumelsky and T. Skewis, "Incorporating range sensing in the robot navigation function," IEEE Transactions on Systems Man and Cybernetics, vol. 20, pp. 1058 – 1068, 1990.

[10] J. Borenstein and Y. Koren, "The vector field histogram - fast obstacle avoidance for mobile robots," IEEE Transaction on Robotics and Automation, vol. 7, no. 3, pp. 278 – 288, 1991.

[11] Gyrocompass. Webpage, http://en.wikipedia.org/wiki/Gyrocompass

[12] Roble. Website, http://www.roble.info/robotics/motionplanning/html/ObstacleAvoidance-1.html

[13] A. Panousopoulou and A. Tzes, "Utilization of Mobile Agents for Voronoi–based Heterogeneous Wireless Sensor Network Reconfiguration"

[14] P. Alriksson, J. Nordh, K.-E. Årzén A. Bicchi, A. Danesi, R. Schiavi, L. Pallottino, "A Component-Based Approach to Localization and Collision Avoidance for Mobile Multi-Agent Systems"

9. Appendix A

9.1 Obstacle Detection

```
volatile int dir_value=2400;
volatile int dir=1;
volatile int distBuff[3];
volatile int conv = 0;

volatile int ivar = 0;


//To make things easier change angle so: 0<angle<360
void adjust_angle(float *angle){
        while ((*angle)<0){
                (*angle)+=(2*PI);
        }
        while ((*angle)>=(2*PI)){
                (*angle)-=(2*PI);
        }
}

//Move the servo
void move_servo(){
        if (dir<0){
                dir_value-=SERVO_STEP_SIZE;
                if (dir_value<=SERVO_MAX_RIGHT){
                        dir_value=SERVO_MAX_RIGHT;
                        dir=1;
           }
        }
        else if (dir>0){
           dir_value+=SERVO_STEP_SIZE;
           if (dir_value>=SERVO_MAX_LEFT){
                        dir_value=SERVO_MAX_LEFT;
                        dir=-1;
           }
        }
        OCR1A=dir_value;
}

//add an obstacle (if the distance is less than ~0.64)
```

```
void add_obstacle(int distance){
        float angle  = (dir_value*SERVO_STEP_ANGLE)-51.0424;
        angle*=(PI/180); //convert to radians

        //*** CONVERT RESULT FROM A/D INTO A DISTANCE IN
METERS ***
        float dist_m =(log(distance)-C2)/C1;

        //*** CONVERT TO A GLOBAL COORDINATE SYSTEM ***
        angle+= (X[2] - (PI/2) );
        adjust_angle(&angle); //0 < angle <= 2pi to make things easy
        float x = 0;
        float y = 0;

        //*** COMBINE angle AND dist_m TO GET x,y ***
        if (angle == 0){
                x=X[0]+dist_m;
                y=X[1];
        }
        else if (angle<(PI/2)){
           x=X[0]+(dist_m*sin((PI/2)-angle));
           y=X[1]+(dist_m*cos((PI/2)-angle));
        }
        else if (angle==(PI/2)){
           x=X[0];
           y=X[1]+dist_m;
        }
        else if (angle>(PI/2) && angle<PI){
           x=X[0]-(dist_m*sin(angle-(PI/2)));
           y=X[1]+(dist_m*cos(angle-(PI/2)));
        }
        else if (angle == PI){
           x=X[0]-dist_m;
           y=X[1];
        }
        else if (angle>PI && angle<(1.5*PI)){
           x=X[0]-(dist_m*sin((1.5*PI)-angle));
           y=X[1]-(dist_m*cos((1.5*PI)-angle));
        }
        else if (angle==(1.5*PI)){
           x=X[0];
           y=X[1]-dist_m;
        }
        else if (angle>(1.5*PI)){
           x=X[0]+(dist_m*sin(angle-(1.5*PI)));
```

```
            y=X[1]-(dist_m*cos(angle-(1.5*PI)));
        }
        add_to_map(x,y);
}

// a/d conversion complete interrupt
SIGNAL(SIG_ADC){
        int distance;
        unsigned char lbyte,hbyte;
   lbyte = inp(ADCL);
   hbyte = inp(ADCH);

   distBuff[conv] = (hbyte<<8) | lbyte;
   ++conv;

        if (conv<3){
                outp(BV(ADEN)|BV(ADSC)|BV(ADIE)|BV(ADPS2)|BV(ADPS1)|
                        BV(ADPS0),ADCSRA);
        }




        else if (conv==3){
                //*** FIND THE MEDIAN OF THE VALUES ***
                distance = MIN(MIN(
                        MAX(distBuff[0],distBuff[1]),
                  MAX(distBuff[0], distBuff[2])),
                      MAX(distBuff[1],distBuff[2]));

             /*
              * Notice the semi-hack (last bit of if-statement)!
              * I'ts there cuz of the poor angle estimation
              * when the robot is turning fast, so what the hack
              * does is it forces the robot not to update the
              * map when the robot is turning faster than
              * pi/4 radians/second!
              */
             if (distance >= MAX_DIST && mode==1
            && (ABS(u[0]-u[1])/L_BOT)<(PI/4)){
                add_obstacle(distance);
             }
             move_servo();
        }
}
```

```
//*** TIMER INTETRRUPT ***
SIGNAL(SIG_OVERFLOW0){
   //*** RESTART TIMER ***
        outp(0x01,TCNT0); //6F <=> 10 ms

#if BOT_ID==2
        if (drop_mote==1){
                if (ivar==0)
                        OCR1B=1100;
                else if (ivar==150){
                        OCR1B=3700;
                        ivar=-1;
                        drop_mote=0;
                }
                ++ivar;
        }
#endif
        if (mode==1){
                conv=0;
                outp(BV(ADEN)|BV(ADSC)|BV(ADIE)|BV(ADPS2)|BV(ADPS1)|
                        BV(ADPS0),ADCSRA);
        }
}
```

9.2 Obstacle avoidance

```
//*** GLOBAL VARIABLES ***
volatile struct point goal = {0,8};
volatile int nbr_ref = 0;
volatile uint8_t obstacles[matrix_x][matrix_y];

//*** FILE-LOCAL VARIABLES ***
float k_att = 0.17; //attractive force constant
float max_speed = 0.2; // maximum speed for the robot
float d0 = 0.7; // max distance that the sensor can reach
float k_obst = 0.10; //repulsive force constant
float k_speed = 0.1; //constant for the speed

int map_test = 1;

//calculate the euclidian distance of the vector v
float get_distance(struct point v){
        return sqrt( SQR(v.x)+SQR(v.y) );
}

//calculate the attractive force gradient
struct point get_f_att(){
        struct point ret_gradient;
        ret_gradient.x = -(X[0] - goal.x);
        ret_gradient.y = -(X[1] - goal.y);

        ret_gradient.x=ret_gradient.x/get_distance(ret_gradient)*8.0f;
        ret_gradient.y=ret_gradient.y/get_distance(ret_gradient)*8.0f;

        return ret_gradient;
}

//calculate the repulsive force gradient
struct point get_f_rep(){
        int i,j;
        int t=1;

        int start = get_start_index();
        int stop = get_stop_index();

        struct point ret_gradient;
        struct point obst_pos;
```

```
        obst_pos.x = 0;
        obst_pos.y = 0;

        ret_gradient.x = 0;
        ret_gradient.y = 0;

        //calculate an avrage of all the obstacle gradients
        for (i=start;i<stop;++i){
                for (j=0;j<25;++j){
                        if (obstacles[j][i]>=1){
                                obst_pos.x = index_to_coordinate(j);
                                obst_pos.y = index_to_coordinate(i);

                                ret_gradient.x =
                        (((t-   1)*ret_gradient.x)/t)+(obst_pos.x/t);
                                ret_gradient.y =
                        (((t-1)*ret_gradient.y)/t)+(obst_pos.y/t);
                                ++t;
                        }
                }
        }
        //if no obstacles where found return 0
        if (t==1) {
                ret_gradient.x=0;
                ret_gradient.y=0;
        }
        else{
                ret_gradient.x=X[0]-ret_gradient.x;
                ret_gradient.y=X[1]-ret_gradient.y;
        }

   return ret_gradient;
}

//calculate the heading and the speed for the robot
void get_references(float *speed, float *heading){
        struct point heading_att_grad  = get_f_att();
        struct point heading_rep_grad  = get_f_rep();
        struct point heading_grad;
        float dist_m = get_distance(heading_rep_grad);

        // make the resulting gradient prop. to 1/distance^2 (that is   // distance to
the obstacle)
        if (dist_m<d0 && dist_m>0) {
          heading_grad.x = k_att*heading_att_grad.x +
```

```
                k_obst*( (1/dist_m)-(1/d0) )*
                ( 1/(SQR(dist_m)) )*
                        ( (heading_rep_grad.x)/dist_m);

        heading_grad.y = k_att*heading_att_grad.y +
                        k_obst*( (1/dist_m)-(1/d0) )*
                        ( 1/(SQR(dist_m)) )*
                        ( (heading_rep_grad.y)/dist_m);

        //if the obstacle is to far away ignore it
        } else {
                heading_grad.x=k_att*heading_att_grad.x;
                heading_grad.y=k_att*heading_att_grad.y;
        }
        *speed = get_distance(heading_grad);
        *heading = atan2( heading_grad.y,heading_grad.x );
        (*speed)*=k_speed;

        if ( (*speed) > max_speed){
                *speed=max_speed;
        }
        ++nbr_ref;
}
//couldn't find a round function so I made my own with ceil and //floor
int round (float x){
        float temp = x;
        temp-=(int)x;
        if (temp>=0.5)
                return (int) ceil(x);
        else
                return (int) floor(x);
}

void print_map(){
        int i,j;
        for (i=0;i<matrix_y;++i){
                for (j=0;j<matrix_x;++j){
                        put_int(obstacles[j][i]);
                }
        }
}

void add_to_map(float x, float y){
        int xIndex = round((x*100)/Resolution);
        int yIndex = round((y*100)/Resolution);
```

```
        if (xIndex<matrix_x && yIndex<matrix_y){
                obstacles[xIndex][yIndex]=nbr_ref;
        }
}

//returns a coordinate (in meters) given an array index
float index_to_coordinate(int index){
        float val;
        val=index*Resolution;
        val/=100;
        return val;
}

//returns the start index, that is an index corresponding to 0,2m beihnd the robot.
int get_start_index(){
        int index;
        float y;
        if (X[1]>0)
                y = X[1] * 100;
        else
                y=0; //simpler if the map starts at y=0

        index = round(y/Resolution);
        index-= 2;

        if (index<0){
                index=0;
        }

        return index;
}
//returns the stop index, that is an index corresponding to 0,6m in front of the
//robot.
int get_stop_index(){
        int index = get_start_index();

        index+=8;

        if (index>matrix_y){
                index=matrix_y;
        }
        return index;
}
```