University of Western Australia

Department of Electrical and Electronic Engineering

Hauptseminar

Informatik

Norman Apel

Index:

1 Preface

The intention of this work is the design and implementation of a system for navigation of an autonomous mobile robot in a maze.

In order not to collide with an obstacle on the way from start to goal, different navigation problems must be solved. Basically, there are two kinds of navigation, local and global navigation. Global navigation is associated mainly with routing on a map. Essential for this sort of navigation is, that it is based on data that is not producible by the actual sensor information. Therefore, the robot has to create a model of its environment. Self-localization is an essential part of that process. Local navigation is understood as the avoidance of obstacles within the sensor range.

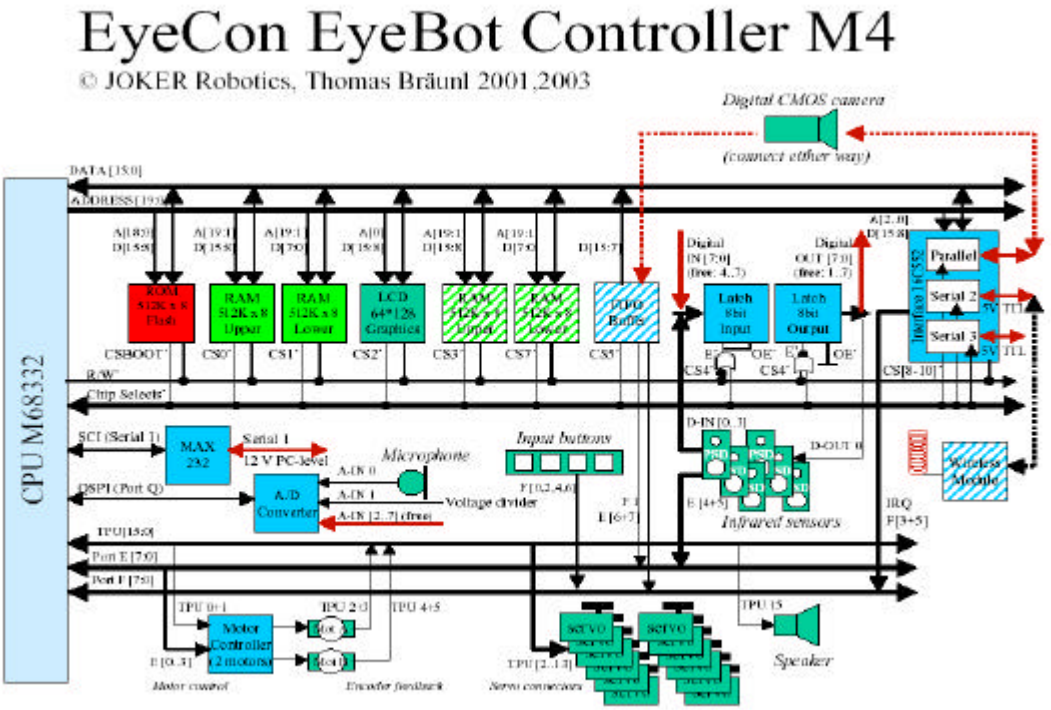
Navigation is a necessary task in mobile robotics, therefore there are many strategies dealing with the problem but the applicability of these strategies is mainly limited by the available hardware and software. This work focuses on a realization regarding the available resources.

2 Specification

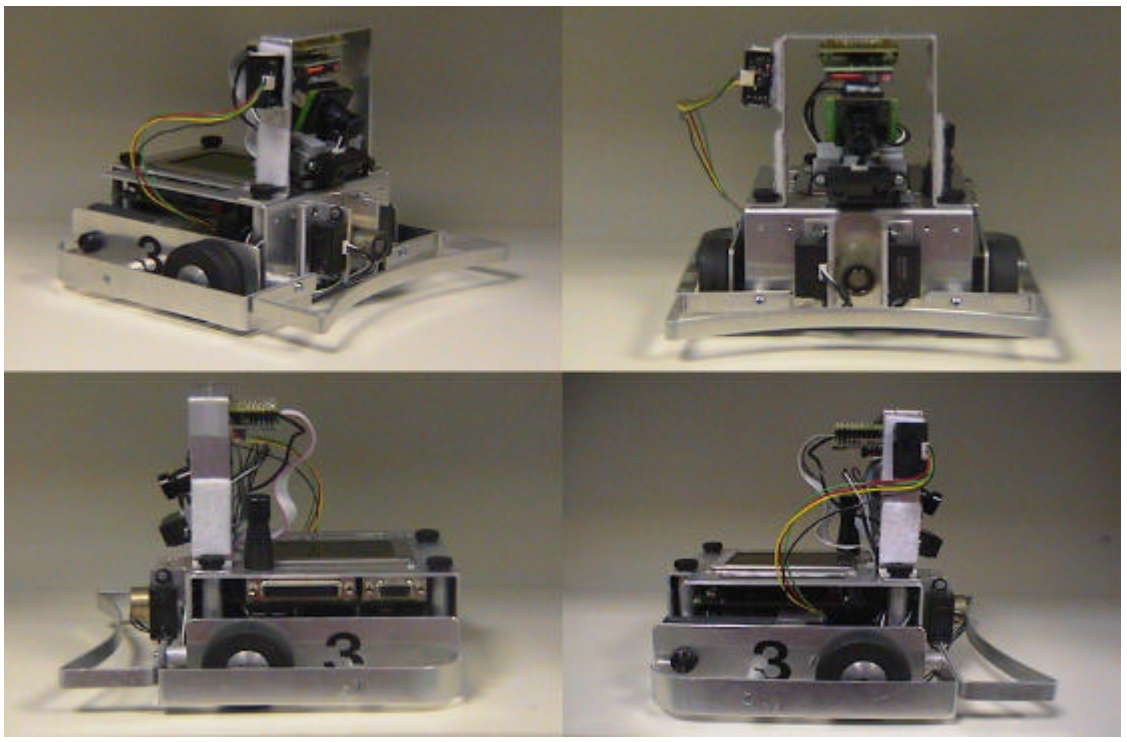
2.1 Hardware

Eyebot Controller:

- 25MHz 32bit controller (Motorola 68332)
- 1 MB RAM
- 512kB ROM (for system and user-programs)
- 1 parallel port
- serial ports
- 2 DC motor driver (L293D)
- background debugging module
- 8 digital inputs
- 8 digital outputs
- 8 analog inputs
- interface for color and b/w camera (real-time on board image processing)
- LCD display (128x64 pixel)
- input buttons
- speaker
- microphone
- battery level indication
- Plug and Play connectors for motors, sensors and camera
- integrated wireless communication system


EyeCon EyeBot Controller M4
© JOKER Robotics, Thomas Bräunl 2001,2003
Digital CMOS camera
(connect either way)
CPU M68332
DATA [15:0]
ADDRESS [19:1]
ROM 512K x 8 Flash
RAM 512K x 8 Upper
RAM 512K x 8 Lower
LCD 64*128 Graphics
RAM 128K x 8 Upper
RAM 128K x 8 Lower
FPGA Buffer
Latch 8bit Input
Latch 8bit Output
Parallel
Serial 2
Serial 3
CSBOOT'
CS0'
CS1'
CS2'
CS3'
CS7'
CS5'
CS4'
CS[8-10]
R/W
Chip Selects'
SCI (Serial 1)
MAX 232
Serial 1
12 V PC-level
Microphone
A/D Converter
Voltage divider
Input buttons
Infrared sensors
Wireless Module
TPU[15:0]
Port E [7:0]
Port F [7:0]
Motor Controller
Motor control
Encoder feedback
Servo connectors
SERVO
Speaker
IRQ
F[3+5]

Labcar:

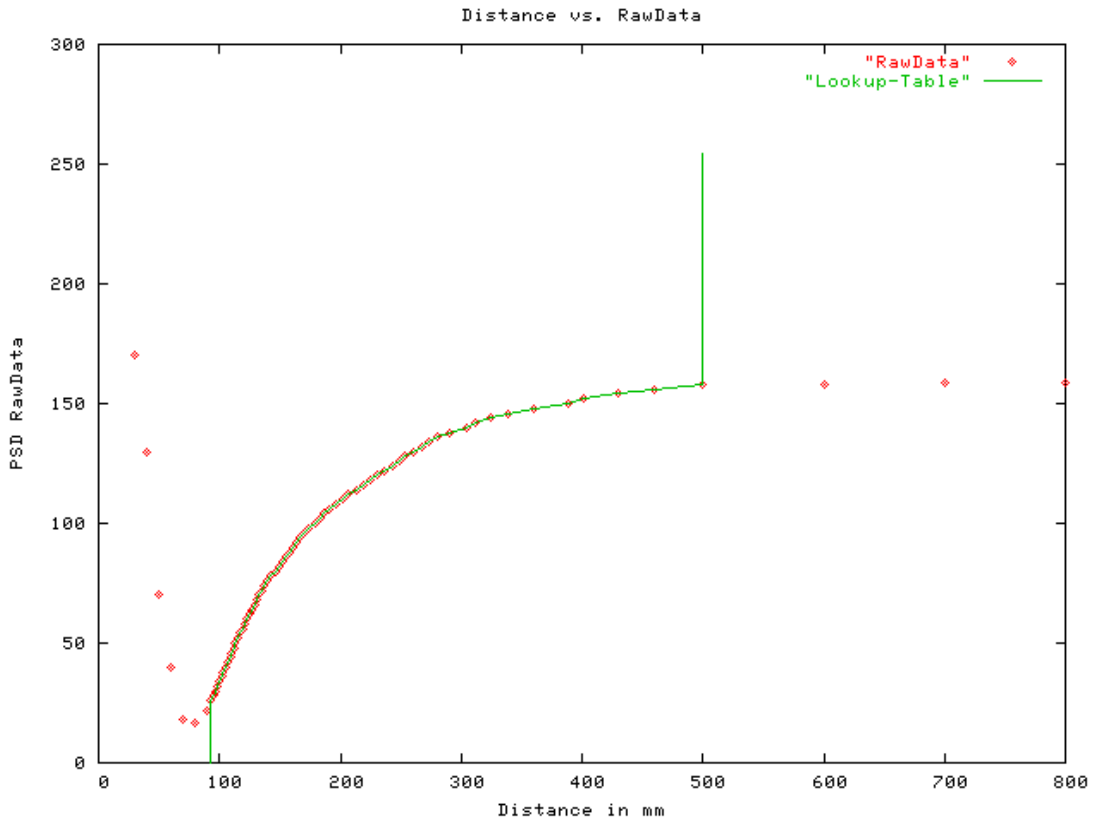


Configuration:

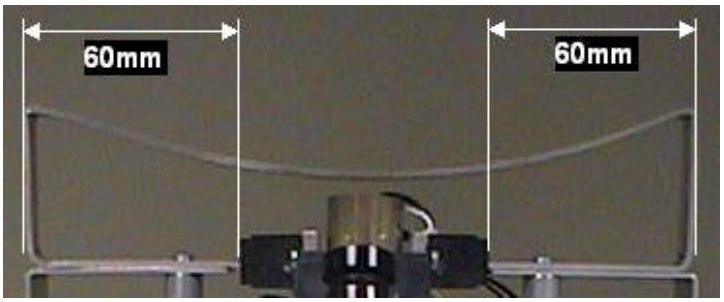
- EyeBot controller
- DC motors with encapsulated gears and encoders
- 1 servo for camera
- EyeCam digital camera
- LiIon battery with 220V charger
- EyeNet wireless communication module
- infra-red PSDs (Sharp GP2D02)

The raw data generated by the PSD is converted into a distance measurement by a lookup table. To get correct measurements in close-up range it was necessary to change the arrangement of the PSDs. Front, Left, and Right PSDs were calibrated separately and provided with appropriate lookup tables.

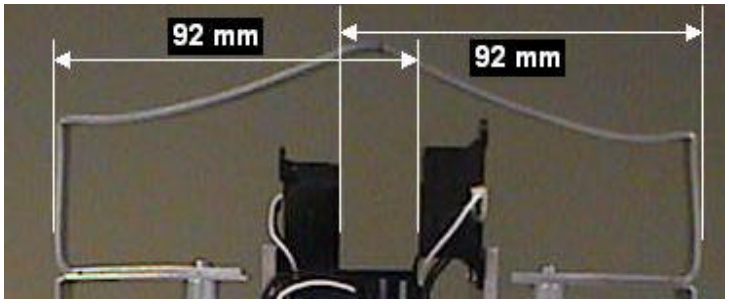
Example FrontPSD:



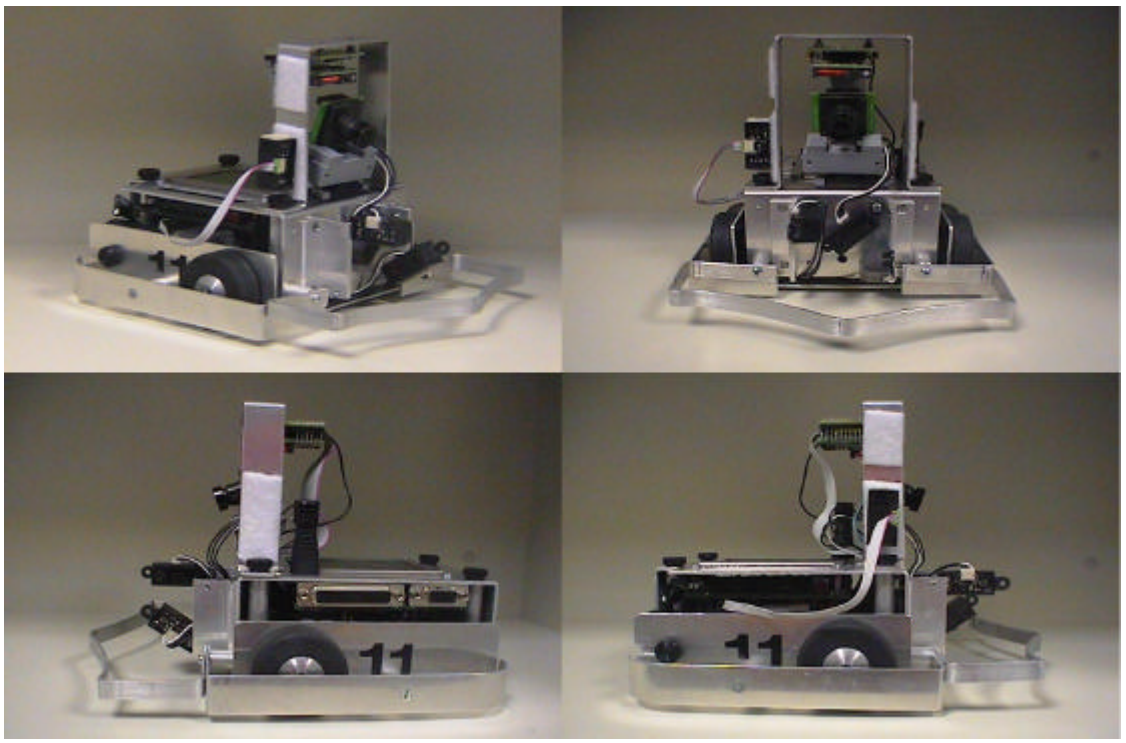
Arrangement before:



Arrangement after:



Modified Labcar:



2.2 Software

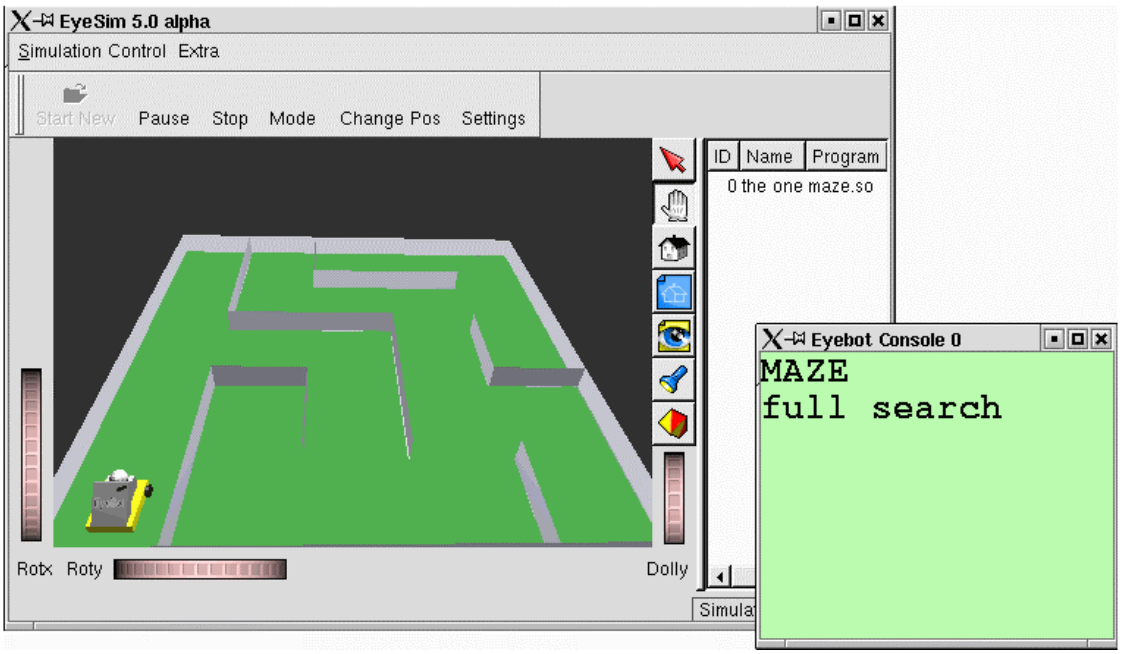
Operating system Eyebot:

RoBIOS V5.2

With inbuilt functionality for Image Processing, Key Input, LCD Output, Camera, System, Functions, Multi-Tasking, Semaphores, Timer, Download and RS-232, Audio, PSD Sensor, Servos and Motors, V-Omega Driving Interface, Bumper / Infrared Sensors, Latches, Parallel Port, A/D Converter, Radio Communication, Compass und TV Remote Control.

EyeSim simulator (Linux):

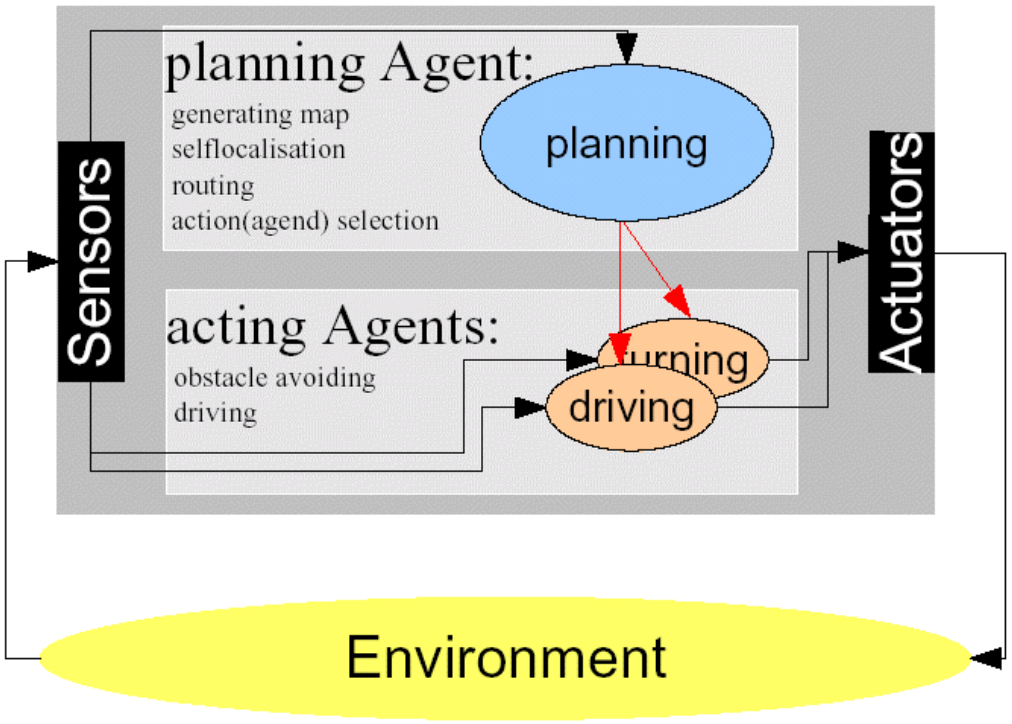
The EyeSim simulator is a multiple mobile robot simulator. It was designed for simulation of eyebot programs in a virtual environment. No changes in program code are necessary. The simulation includes the RoBIOS functions, sensors and actuators. The environment can be provided either in World format (Saphira) or in Maze format (Micromouse).



3 Approach and Realization

3.1 Structure

To solve the problem my decision was to use a hierarchical approach. A superior agent takes care of the routing and planning of action sequences. He activates the specific lower agents, which are actually doing something.



The advantage in this is that action sequences can be divided into elementary actions. These elementary actions are executed by agents, which are specially trained for their

tasks. Another big advantage is the modularity. This makes it easy to replace components and develop them separately.

3.2 Component Details

3.2.1 Routing

The used maze (real and virtual) has orthogonal walls and the wall units are equal in length. Therefore, it was self-evident to represent the map as a grid or a matrix. Further, it is assumed that the robot is aligned on the grid and starts at the left bottom corner.
In this scenario, three methods for routing were tested.

1. Left Wall Following:

This method does not need a map. If the maze is free of cycles it always finds a way. The advantage of this method is the easy implementation, but it is obvious that in most of the cases, the path found may not be the shortest one.
The algorithm is simple. The robot is confronted with four situations where he can take four different actions. These are executed as long as the goal is not reached.
Action table:

L	F	R	Action
0	x	x	turn 90 degree CCW, drive one square
1	0	x	drive one square
1	1	0	turn 90 degree CW, drive one square
1	1	1	turn 180 degree CW, drive one square

L – Left PSD

F – Front PSD

R – Right PSD

0 – PSD reading > Threshold (Threshold ~ grid size)

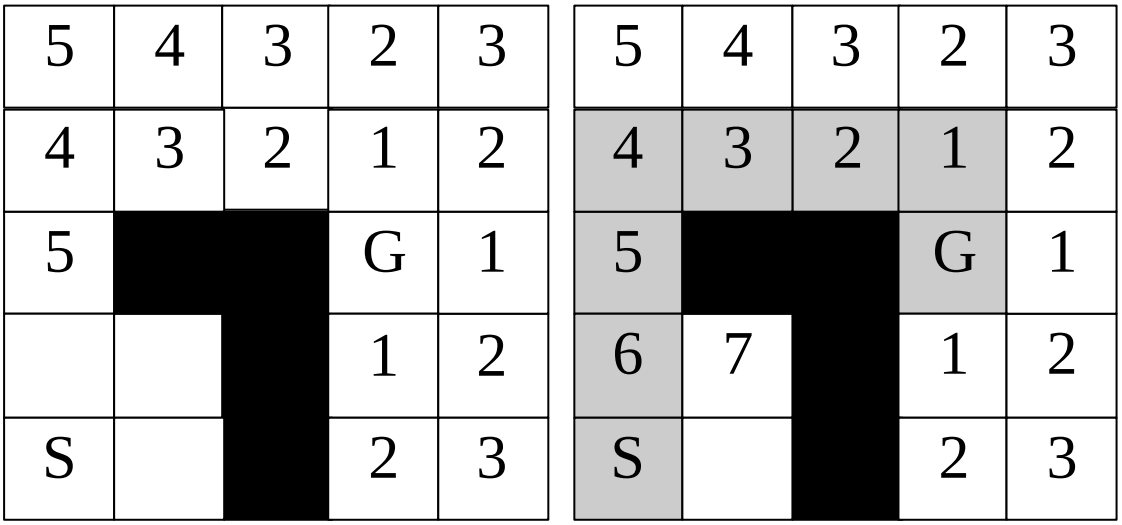
1 – PSD reading < Threshold

x – does not matter

CW, CCW – clockwise, counter clockwise

2. Lee-Moore Algorithm:

This algorithm needs a map. My implementation of the Lee-Moore algorithm represents the map as a grid, where each grid point can contain connections to adjacent grid points. It searches for a shortest-path connection between the start and goal by performing a breadth-first search and labeling each grid point with its distance from the goal. This expansion phase will eventually reach the start if a path exists. A second trace back phase then forms the shortest path by following any path with increasing labels. This algorithm is guaranteed to find the shortest path between a start and goal for a given map. The complexity of this algorithm is O(n3). For the used mazes, this was the algorithm with the best performance. Despite the maze being empty and hence there being many possible paths, this algorithm generates a solution quickly.



3. Reinforcement routing:

This algorithm works by reinforcing actions that lead to the desired behavior. This is achieved by using an extern reinforcement function. In the problem of routing, the reward is one when the goal is reached and zero in all other positions.

This algorithm needs a map as well. In my implementation, every grid position is represented with a state s. You can change states by taking actions a (e.g. go north, go west ...). State action pairs are called Q-Values and are stored in a table. In order to learn

faster, eligibility traces were used. That makes it pretty much the Sarsa Lambda Learning scheme, which works as follows:

Online Q-learning with Eligibility Traces:

Initialize every Q(s,a)=arbitrarily
X1 s = start
initialize every e(s,a)=0
observe current state s=st
repeat
- *select an action a=at and execute it*
  (max or random action, if random then Lambda = 0)
- *observe new state s'=st+1*
- *get max action a' in s'*
- *receive reward r=r(s')*
- *dQ = r + Gamma * Q(s',a') – Q(s,a)*
- *e(s,a) = 1 /*replacing trace*/*
- *do for all s,a*
  *Q(s,a) = Q(s,a) + Betha * dQ * e(s,a)*
  *e(s,a) = e(s,a) * Lambda * Gamma*
- *s <= s'*

until s is final
for X Epochs goto X1

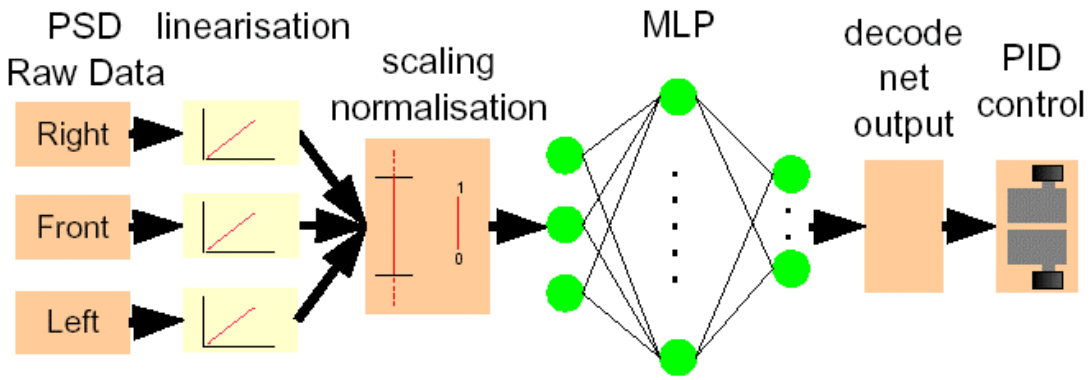
The state action table represents a police Q'. If you execute the action with the biggest value in each state, you follow the path to the goal. If every action in every state is executed infinitely, Q' converges to the ideal policy Q*, which is the shortest path.
The advantage of this method is that it is learning online which means that it can react to changes in the environment. Acquired knowledge about unmodified sections in the maze stays unchanged while just the knowledge of the modified part is adapted. This would lead to better performance than the Lee-Moore algorithm in very large mazes but has no effect at the size of the used mazes.

3.2.2 Collision Prevention

For the benefit of a fast reacting system, that is, driving in a corridor and avoiding walls, just the PSD sensors were used. The camera was not used, since image processing takes a lot of time. The main task was to drive in the middle of the corridor and around 90 degree

corners. A good approach to solve that problem is to use a Multi Layer Perceptron (MLP). It can be trained with back propagation. The training data was simply generated by pushing the robot through the maze and record PSD readings, the speed and the rotational speed. Then the MLP is trained with that data on the PC. Transferring the MLP back to the robot he is now able to drive on its own. This method is called expert cloning.
Of course, some data manipulation is necessary before and after the neural net, which is explained below:

Dataflow:



The used sensors (Sharp GP2D02) have a non linear-output. In order to get a distance measurement you have to do a linearization and a scaling first (detail 2.1). After that, I blend out distances that can either not occur or are not relevant for the task by clipping the distance measurements to the range from 92mm to 420mm. This range is than scaled to 0 to 1.
To enhance the performance, different setups were tested in the Neural Network module.

Setup 1:

· 3 input neurons

· 5,7,8 or 16 hidden neurons

· 2 output neurons

These Networks were trained with two different kinds of training data. The first approach was to control the speed of left and the right motor directly. After putting a PID controller between the net output and the motors, it worked well.
The second approach was to let the network control the speed (v) and the rotation speed (w). The advantage in this is that the task becomes easier because the speed is nearly constant. Furthermore, the V-Omega driving interface can be used which has an inbuilt PI controller.
Both approaches generate good results but using the V-Omega driving interface makes many thins much easier. That is the reason the second version was preferred. Increasing the number of hidden neuron to more than seven, led to no more improvement.

Setup 2:

- 9 input neurons
- 8,16 or 32 hidden neurons
- 2 output neurons

The input neurons obtain the sensor readings from the current and the previous two time steps. That is why nine input neurons are needed. This is a called sliding window technique, and it leads to an expansion of the feature space. The intention in this is to get more information out of the few sensors.
The training data corresponds to the second approach in setup 1.
Unfortunately, this setup did not perform very well. This can be caused by the complexity of the input space and the mapping function, which cannot approximated properly by a three layer MLP. The biggest problem was nevertheless the timings. It is essential that the sample rate is the same while recording the training data and while using the neural net. I could not find a method that ensures this in an appropriate way.

Setup 3:

- input neurons
- 5,7,8 or 16 hidden neurons
- 18 or34 Output neurons

This setup is just a little different from the first one. It is just got a different output coding, which is topological.

Topological coding:

w in rad/s

v in m/s

Because of the topological coding, the neural net should be more robust. This effect did not occur. In addition, the computing time was worse because of the higher number of neurons.

Conclusion:

The best result was achieved with setup 1. In a maze with no dead ends, the robot drove nearly collision free. Unfortunately, the neural net was not able to learn a U-turn. However, this is understandable, due to the limited sensor information and the complexity of the task.

4 Future Prospects

A big problem on the real robot was the self-localization. Using just odometer, the information gets more and more inaccurate the further the robot goes. Furthermore, the assumptions of the alignment and the orthogonal grid structure of the maze are very restricting.

An approach to ease the restrictions is to represent the map as a graph. The position could be mapped to the graph with vector-quantization. The drift of the odometer could be adjusted by that way, but now the problem is to define a discrete action space and the recognition of important landmarks, like crossings. This is hard to achieve without further sensors.

I tried a graph approach, keeping the restrictions. A Growing Neural Gas (GNG) generates the graph and the path is learned collateral with reinforcement learning. The reason why it was not working was the instability of the Neural Gas. The parameters of the GNG could not be adjusted in a way that new nodes were inserted for unknown positions in the map while the old nodes were stable enough to represent the visited positions.

Other approaches to build a graph representing a map could be Cohonen-Maps and ART-Nets. Especially ART-nets are good in stability and plasticity. A big advantage is also the ability of One-Shot-Learning.

5 References

[1] Prof. Dr. H.M. Gross, TU-Ilmenau, script “Lernen in neuronalen Netzen”, (2002)

[2] Dr. H.J. Boehme , Tu-Ilmenau, script “Kuenstliche Neuronale Netze”, (1998)

internet:

[3] http://robotics.ee.uwa.edu.au/eyebot/index.html

[4] http://www.geocities.com/fastiland/rlrobots.html#reinfo

[5] http://www.ece.iit.edu/~jnestor/java/MazeRouter.html