Line Detection For Self Localization Of A Sony AIBO Robot

Peter Mörck

Examensarbete för 30 hp, Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet.

Thesis for a diploma in Computer Science, 30 ECTS credits, Department of Computer Science, Faculty of Science, Lund University.

Line Detection for Self Localization Of A Sony AIBO Robot

Abstract

This masters thesis investigates a sequence of methods and their usability for detecting lines with the built-in camera on a Sony AIBO robot. Primarily the goal was to develop a solution that eliminates the need for colour-coded landmarks in the RoboCup 4-legged soccer league in which AIBO robots are used. Using the framework Tekkotsu as a basis, camera image data is retrieved and then processed in a custom Tekkotsu module. This module performs edge detection, thresholding, thinning and line detection. The edge detection is a simplified version of the Sobel edge detector whose results in the next step are thresholded to produce a binary image. The binary image is passed on to the thinning algorithm which scans the surroundings of each pixel and either keeps or removes the pixel. Finally, the thinned image is sent to the line detector which uses the Hough Transform to extract the line information. The data of the detected lines is transformed from image frame coordinates to a global robot coordinate system. The line data can then be used in conjunction with a localization system, a very common part in mobile robot software. It turns out that the distance precision of the solution is acceptable, i.e. calculated distance to lines is only a few centimeters off, at distances up to at least half a meter, but gives too large errors at distances of one meter and above. At the same time the angular precision is very good across all ranges as long as the line is detectable by the image processing algorithms. The solution presented herein thus probably serves best as a complement to other localization methods due to its poor accuracy at longer distance, plus due to symmetry ambiguities of the lines of a soccer field, if one chooses to use the solution for the RoboCup or similar environments.

Linjedetektering för självlokalisering av en Sony AIBO robot

Sammanfattning

Detta examensarbete undersöker en följd av metoder och deras användbarhet för att detektera linjer med den inbyggda kameran på en Sony AIBO-robot. Det primära målet var att utveckla en lösning som eliminerar behovet av färgkodade landmärken i RoboCup 4-legged soccer league, i vilken Sony AIBO-robotar används. Genom att använda ramverket Tekkotsu som bas hämtas bilddata från kameran och behandlas därefter i en egenhändigt skriven Tekkotsumodul. Denna modul utför kantdetektering, tröskelvärdesbegränsning, förtunning och linjedetektering. Kantdetekteringen är en förenklad variant av Sobeldetektorn vars resultat i nästa steg kontrolleras mot ett tröskelvärde för att få en binär bild. Den binära bilden skickas till förtunningsalgoritmen, vilken kontrollerar varje pixels omgivning för att bestämma om pixeln i fråga skall behållas eller tas bort. Slutligen skickas den förtunnade bilden till linjedetektorn som använder Houghtransformen för att extrahera linjeinformation. De detekterade linjernas data transformeras från kamerabildens koordinatsystem till ett globalt robotkoordinatsystem. Linjedatan kan sedan användas tillsammans med ett lokaliseringssystem, en väldigt vanlig del av mjukvaran för mobila robotar. Det visar sig att avståndsprecisionen i lösningen är acceptabel, d.v.s. beräknade avståndet till linjerna har enbart ett fåtal centimeters avvikelse, på avstånd upp till minst en halvmeter men att lösningen ger för stora fel vid avstånd på en meter och uppåt. Samtidigt är vinkelprecisionen väldigt bra på samtliga avstånd så länge linjen kan detekteras av bildbehandlingsalgoritmerna. Lösningen som presenteras här fungerar således förmodligen bäst som ett komplement till andra lokaliseringsmetoder på grund av den dåliga precisionen vid högre avstånd och även på grund av tvetydigheter till följd av symmetrin av linjerna på en fotbollsplan, om man vill använda lösningen till RoboCup eller liknande miljöer.

Contents

Chapter 1

Introduction

1.1 Historical background

The first programmable robots appeared shortly after the advent of the first computers back around the 1950's. As advances in technology and computer science were made the robots too became more advanced. The first robot to be controlled by artifical intelligence, Shakey[1] (seen in Figure 1.1), was presented around 1970. It hosted a clever reasoning program fed very selective spatial data, derived from weak edge-based processing of camera and laser range measurements.

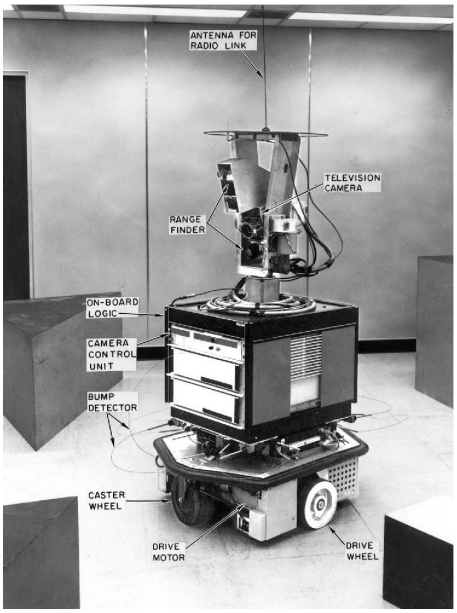

Figure 1.1: Shakey, the first mobile robot controlled by artificial intelligence.

Shakey did not use onboard computers but instead transmitted all sensor input over a radio link to stationary computers which sent commands back to Shakey. Although the computations did not take place on the robot itself, plus the fact that the robot reportedly moved with a speed of around two meters per hour, it was a first step.

Autonomous robots had been around for a while (for example Grey Walter's tortoises [2]), but programmable autonomous robots with onboard artificial intelligence now began to appear. During the 1980's the robot industry grew rapidly and although robotics were used in different areas such as manufacturing and space exploration, robots started appearing as toys.

In 1999 Sony introduced the AIBO[3] (Artificial Intelligence roBOt) entertainment robot, a dog-like mobile platform with a moderately powerful onboard computer and various sensors. It quickly became a common test-bed for artificial intelligence and related areas. It is also the platform used in this thesis.

1.2 The need for input

All robots, mobile ones in particular, that interact with their environment have a need to be able to analyze it. Gathering information about the world around a robot is done through various sensors. Common sensor types include touch, pressure, sound, visible light, IR light and so on.

The most useful sensory input to humans is vision using our eyes, which can be thought of as refined sensors for visible light. Our eyes are very important for us because they let us know what the environment we are in looks like and they provide valuable feedback when we perform different actions that might affect the state of our surroundings. We take our eyes and the information they provide for granted. We do not think about what we are looking at and how to try to interpret the image. We just do it.

For robots, this seemingly trivial task is very complex and difficult. It is not the eyes or the camera itself that is complex, but the analysis of the information it provides. Image analysis is still an area where advances need to be made.

1.3 Image analysis

The data that is gathered from a digital imaging device, such as a camera, is in practice just a matrix of values. A human looking at such values will probably have more problems analysing this data compared to looking at a photo of the same subject. For a computer to make any use of the data it has to apply calculations to the data, providing some results which hopefully can be classified according to some rules. Many different algorithms and techniques have been developed for analyzing images. These can all be categorized differently depending on use or theoretical base. Most, if not all, of them can be found at the HIPR [4]. As a small introduction and to give an overview of what types of algorithms exist, some briefly explained examples follow.

1.3.1 Filters

Filters have various uses but are mostly used to suppress noise (also known as smoothing) or to bring out details (like image sharpening) in an image. Some common filters are *Mean*, *Median*, *Gaussian*, *Laplacian* and *Unsharp* filter.

1.3.2 Morphology

Morphological algorithms generally alter the pixels of an image in a way to fix small errors or tidying the image up after previous algorithms. Some common morphological algorithms include *Erosion*, *Dilation*, *Opening*, *Closing*, *Thinning*, *Thickening* and *Skeletonization*.

1.3.3 Colour segmentation

Colours provide more information in an image but usually require more calculations. While it is sometimes practical to consider only gray-scale images in image analysis it is sometimes necessary to use all the colour information given. For example under certain circumstances two different colours might give the same gray tone when converted to gray-scale. It might be of interest to be able to separate different areas of colour from each other so that they can be analyzed relative to each other.

As an example, assume two squares of different colours have a certain size and have a known distance between them. Analyzing the image to calculate their orientation, size and the distance between them in the image is a good way to estimate how far away from the camera they are.

1.3.4 Feature detection

Edge detection

An edge detector reacts to changes in pixel intensity, producing an "edge" where the intensity changes. In a black and white image, such as the one in Figure 1.2, it would detect the "borders" between brighter and darker areas, yielding a result like the one seen in Figure 1.3.

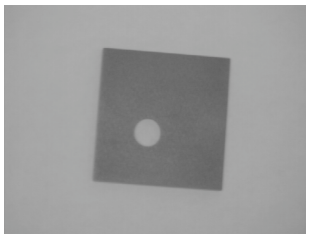

Figure 1.2: A low-contrast black and white image.

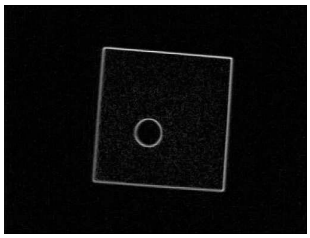

Figure 1.3: Figure 1.2 with edge detection applied.

Some well-known edge detectors are *Robert's Cross*, *Sobel* and *Canny*. Note that while the edges after edge detection are now much more clearly visible to the human eye, the image still consists of a matrix containing just numbers and as a whole still has very little meaning. Edge detectors are thus rarely used alone in image analysis applications but rather as preparation for more high level detectors.

Corner detection

While edges run along the borders of objects they don't give a lot of directly useful information. Corner detectors try to find "corners", such as line corners or any form of corners or junctions in the matrix of values that is the digital image, not necessarily the corners of a room. Figure 1.4 shows some simple figures with detected corners marked. Such corners can be considered to be reference points or points of interest. These can for example be used for tracking objects that move and rotate. One such corner detector is SUSAN [5].

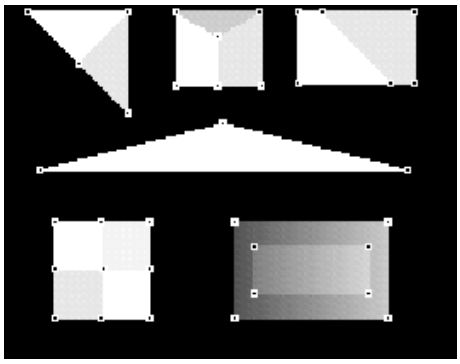

Figure 1.4: Sample image with detected corners marked by the SUSAN algorithm.

Shape detection

A higher level of detection, known as shape detection, tries to identify shapes such as lines and circles in an image. For each shape detected the the detectors will yield discrete parameters, like slope and intercept for lines or center coordinates and radius for circles. These parameters are now easy to classify. One example of a shape detector is known as the *Hough Transform* [6]. This detector serves as the basis for detecting lines in this thesis and is described more in detail in chapter 5.2.4.

Chapter 2

Problem definition

2.1 Purpose

The original purpose of this thesis was to contribute a solution to a localization problem for Team Sweden[7]. Team Sweden was participating in a competition called RoboCup[8] which is held annualy where university teams from all over the world compete in different events of which the biggest one is robot soccer. Team Sweden participated in the 4-legged league where Sony AIBO robots were used. Normally the playfield for the games resembled a traditional soccer playfield but was a bit simplified (See Figure 2.1).

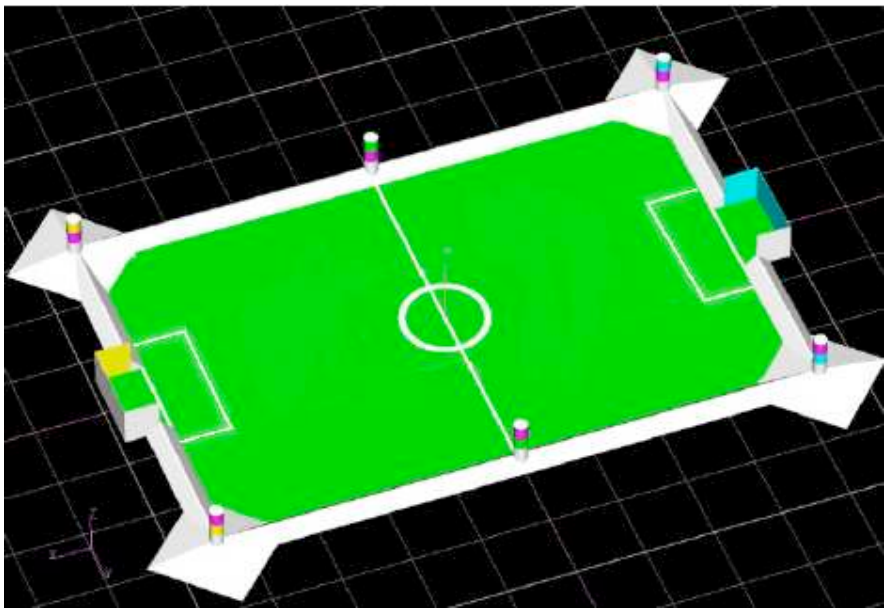

Figure 2.1: Standard playfield used in RoboCup 2003 World Championship.

To ease localization for the robots, the playfield was surrounded by six landmarks which can be seen in the four corners and on each side of the half-way line. Along with the goals, which were uniquely coloured, localization could be made relatively easy, for example using fuzzy localization as described in [9]. Just as in traditional soccer, the purpose of the game is to score a goal in the opponent's side of the field. The ball in this case was a brightly orange coloured ball about the size of an apple. The robots were allowed to communicate with each other via wireless networking but were not allowed to be aided by external computers, i.e. everything had to be processed on the robots themselves. By using the camera to detect the ball, the landmarks and the goals, the robots were supposed to act as a team each consisting of one goalie and three players.

2.2 Problem

The problem now given to the participating teams was that a localization challenge was going to be held without the six landmarks in place. Five points, whose coordinates on the field were unknown to the teams before the challenge started, would each be vaguely marked by a point and a copule of concentric circles on the playfield, barely visible to the human eye but indiscernible to the camera on the AIBO robot. The robot would be placed on the playfield and was then required to visit all five points by trying to navigate to the point and then placing its body on top of the mark. When the robot had reached a point and was satisfied with the position it would wag its tail to indicate it was done. A referee would judge how far from the point the robot was and award points accordingly. The robot would then go on to the remaining four points, repeating the process. The team with the highest score would win the challenge. In case of a tie, the time taken by the robot would determine the winner.

The robot was only allowed to use the coordinates given to it when the challenge started so there was no chance to pre-program it. By using the two remaining landmarks, the goals, the robots could still perform some localization but it would be very inaccurate. This is due to the fact that there were only two goals acting as landmarks, and to make it worse, only one goal at a time would be visible to the robot due to the limited field of view of the camera. Clearly other means for localization were needed. Looking at the playfield, the only remaining stationary features that could be used are the white markings on the playfield. These include the borders, the center circle, the half-way line and the goal lines.

As a demonstration, a movie clip showing the German Team from the RoboCup 2003 event can be found at http://ai.cs.lth.se/xj/PeterMorck/Ch2_gt.mpg .

Chapter 3

Hardware and tools used

3.1 Sony AIBO ERS-7

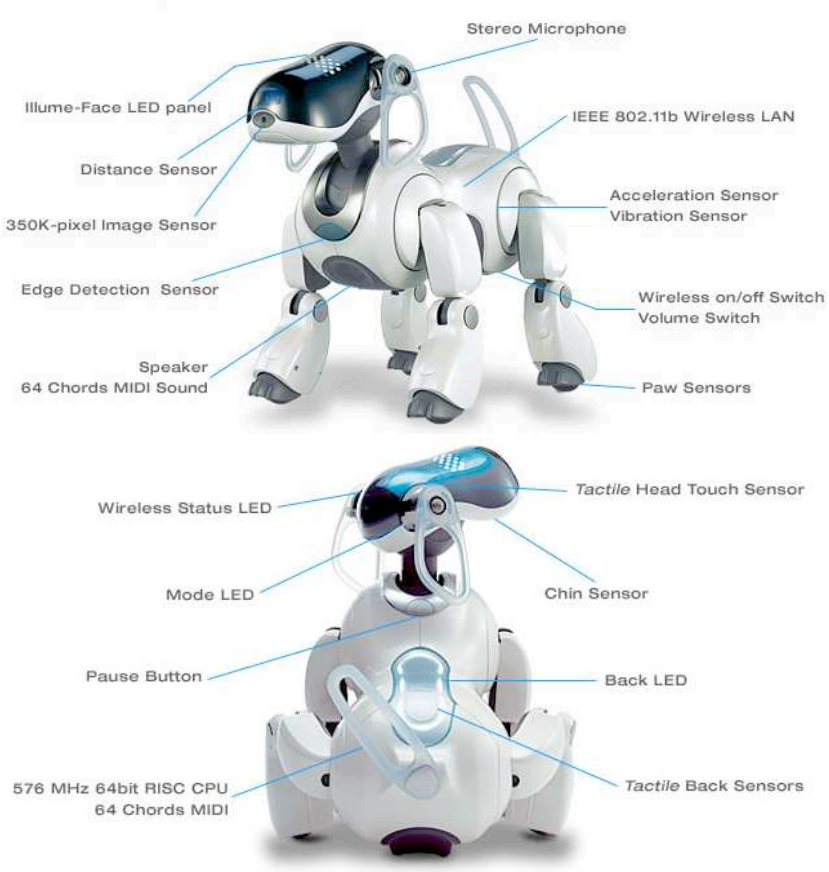

Figure 3.1: ERS-7 front and rear view.

As mentioned earlier, the Sony AIBO ERS-7 is the platform used for this thesis. An overview of the features can be seen in Figure 3.1. Important specifications for the onboard computer and sensors are

- CPU: 576MHz 64-bit MIPS R7000
- RAM: 64 MB

- Storage: Sony MemoryStick reader/writer using 16MB sticks
- Communication: 802.11b wireless ethernet
- Camera: 56.9° wide and 45.2° high
  30 FPS with resolutions 208x160 down to 13x10.
- Sensors update every 32 ms.

To be able to write programs for the AIBO Sony provides a development kit called Open-R [10] which is the API programmers must use to develop software for the AIBO. It is meant to be used with C++ and provides access to all the sensors, motors and so on. However, programming and configuring a project using Open-R directly can be complicated.

In an attempt to remedy this, a team from Carnegie Mellon University has developed a framework acting as a layer on top of Open-R. This framework is called *Tekkotsu* [11].

3.2 Tekkotsu

The word Tekkotsu, meaning *iron bones* in Japanese, gives some indication as to what the use of it is. It provides a basic structure for a project where everything is already setup and ready to run. Only minor configurations are needed and adding new custom program modules is easy. Tekkotsu comes with pre-made behaviour modules for things such as remote-controlled walking, remote head control, camera streaming, colour segmentation and so on. The modules communicate with each other by broadcasting *events* that any other module can choose to listen for.

For example, every time a new camera image is available from the AIBO a Tekkotsu module will use the underlying Open-R functions to retrieve the image data and post an event that this new camera data is available. Another module, say a colour segmentation module, could "listen" for this type of event and would receive it after it was posted. The event has information about the camera image and provides the functionality needed to extract the image data from it. The colour segmentation module could process the image and then post its resulting data as another type of event which in turn other modules can listen for. This approach using events provides a smooth and customizable flow of information through Tekkotsu and it means that the user can intercept and read data for his own modules at practically any stage.

For a large team of developers available to write specific code for different parts of a project Tekkotsu might not be the best choice, but for a small group or a single person wanting to test some approaches to solving a specific problem it is highly useful. The learning needed to get into Tekkotsu programming is relatively little compared to what you need to know to make the AIBO walk and use it's sensors if one were to use only Open-R. With Tekkotsu attention can be focused on the actual problem solving which can be implemented as behaviour modules and inserted into the already running framework. Thus, Tekkotsu was the obvious choice for the project involved in this thesis.

For full details, documentation and obtaining Tekkotsu, please visit the official Tekkotsu homepage [12].

3.3 Custom monitoring utilities

In order to effectively find bugs, change parameters on the fly and measure performance, some custom utilities had to be written. They were all written in java and instructions on how to install and use these utilities can be found in appendix B.

3.3.1 Houghmon

The *houghmon* client will connect to the AIBO and request data related to the Hough Transform to be sent to it. Three images will be streamed to the client:

- Camera image after edge detection and thresholding;
- Hough accumulator after Hough Transform has been applied;
- Y-channel image from camera with detected lines drawn on it.

3.3.2 CmdClient

To be able to change some parameters without having to shutdown, edit, recompile and reboot the robot, the *CmdClient* was developed. It allows the user to change the following parameters:

- Camera resolution (layer);
- Edge detection thresholds for each layer;
- r- and θ intervals for the accumulator;
- Enabling/disabling of the thinning algorithm.

3.3.3 LineMon

To compare the detected lines with the real world, the *LineMon* utility was written. It connects to the AIBO and retrieves the transformed line data. A top-down image of the AIBO is drawn in a window and the detected lines are drawn in their corresponding places.

Chapter 4

Choosing a solution

Given the localization problem, as described in 2.2, it was necessary to find a way to make use of the white markings on the playfield. The colour of the playfield was green and the lines and borders were white. It seemed to be possible to take advantage of the relatively high contrast between the two colours and get some information that way.

Two approaches to solving the problem for Team Sweden were started and evaluated but were later abandoned because the RoboCup competition was held before any solution was completed. The third approach was then started as an independant project.

4.1 Corner detection

The first approach was to try to use the intersections of the lines as landmarks. The fuzzy localization used in [9] required landmarks of point-type with a bearing and distance from the robot. A corner in an intersection of lines would indeed fulfill those requirements. Some development based on edge detection were made and underwent basic tests. Although the algorithm performed relatively well, some observations made showed that:

- A corner or intersection was not uniquely identifiable.
- Due to the low resolution and the close proximity of the camera to the ground, distant corners were very difficult to detect.
- If the robot was looking at the ground and there was no corner in the view then no information was available.

The idea of using corner detection was abandoned and unfortunately none of the original code or examples exist to use for comparison.

4.2 Indirect corner detection by line detection

During the development of the corner detection algorithm, the question about how to detect the corners was *What defines a corner and how does it occur?*. One answer was *Where two or more lines intersect.* Hence, the position of a corner should be easy to calculate once the equations for the two lines that the corner was a part of were known. The second approach to solve the localization problem thus became that instead of searching for corners directly the robot would instead attempt to detect the lines and borders around the playfield. Calculating the intersection between these would indirectly serve as corner detection.

Using lines for indirect corner detection provides some advantages over direct corner detection.

- Distant corners are easier to detect since the lines extend over a longer range.
- Corners obstructed from view, or even outside the camera view, can still be detected if the lines can be detected.

The two methods do however share the problem that the corners are not uniquely identifiable. Initially the idea was to make a tracking algorithm that would locate the contour of a line and follow it. This way the equation of the line would be found. After some consideration a tracking algorithm seemed too unreliable and unscientific so the choice was made to look for a more well-known algorithm. The Hough Transform mentioned earlier was chosen to find the equations of the lines.

A final note should be made regarding this approach. The nature of the Hough Transform is such that it detects *lines* that extend to infinity, not *line segments* with limited length. This means that where there are two actual line segments on the playfield that do not intersect, a false intersection between these could be detected. See Figure 4.1. The dashed lines represent the extension of some of the line segments around the goal. The dotted squares indicate where some of the true positives would have been detected. False positives are marked by dotted triangles and can be seen where the extension of the lines intersect with the border and the half-way line in this example.

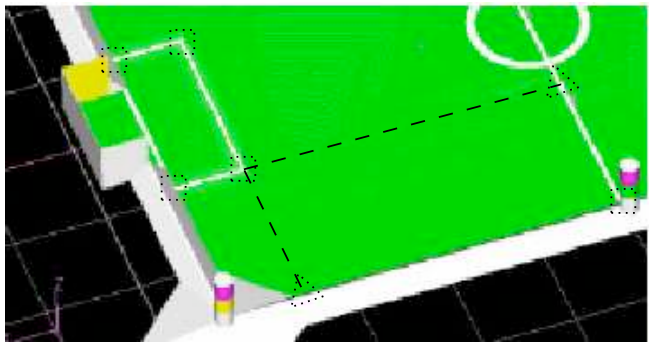

Figure 4.1: False positives when using lines to detect corners.

No real solution to the problem of false positives was made but it was hoped that the way the fuzzy localization worked, these false positives would be countered with the help of the goal landmarks and with repetetive detection where the true positives would be more frequent.

4.3 Line detection only

By this time the RoboCup competition was over and there was no further possibility to co-operate with Team Sweden. In fact every compatibility requirement was removed since the development turned into an independent project. The mapping module that was part of the Team Sweden solution was also missing. This meant that the need for point-landmarks was gone. The development was instead focused on using the lines themselves as landmarks.

For the actual localization, some external module will have to be used as it is beyond the scope of this thesis to provide all the needed components for such a task.

Chapter 5

Solution

5.1 Overview

A structural overview of the solution is given here, followed by more detailed sections for each part of the solution.

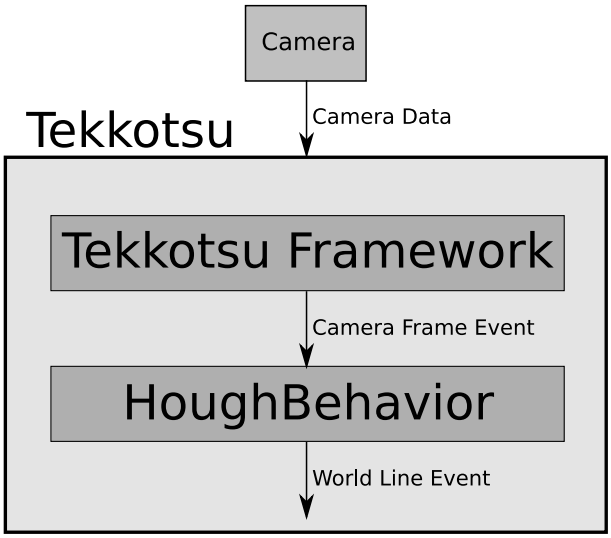

Figure 5.1: Block schematic overview of solution.

In Tekkotsu, messages between different program modules are sent as events. An event can either be of a predefined type or of a custom type that the developer would like to use instead. The main processing module in this project is called *HoughBehavior* and, like all other modules, can register listeners for specific events. For the camera, which is the sensor we are interested in here, there are many different events to listen for, depending on the desired information and format. The default Tekkotsu framework provides modules that among other things retrieve the camera image, scale it to desired resolutions and post the resulting image as an event.

In the HoughBehavior module, a listener is set up to receive events carrying certain camera frame information. When an appropriate event is posted by Tekkotsu, the listener receives it and the frame data is extracted from it. The data is then processed in the HoughBehavior module, producing a list of lines detected in the image. Those lines then go through a transform that filters out lines that are unlikely to be on the ground and transforms the remaining lines to global coordinates instead of camera coordinates.

Early on there was an idea to use a separate module, which would have been called *LineTransformBehavior*, to do the line filtering and transform. This module would receive events posted by the *HoughBehavior* module. However, since the line transform needs the current state of all the joints in the robot to accurately perform

the transform, this idea was scrapped, since the robot might change pose during the time the event is posted and until it is received and processed.

5.2 HoughBehavior

5.2.1 Overview

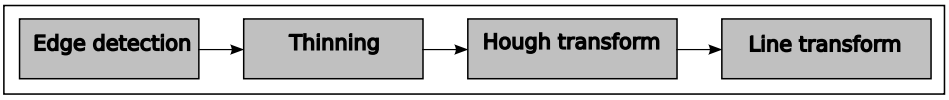

Figure 5.2: Sub-components in the HoughBehavior module.

The HoughBehavior module contains all the executable code needed. As can be seen in Figure 5.2, the Hough transform itself is actually only a part of the module.

In order for the Hough transform to operate properly the source image needs to be pre-processed. After pre-processing, the Hough transform can run and extract line features from the processed image. Once the line data has been extracted it is transformed from camera frame coordinate system to a global/world coordinate system.

5.2.2 Edge Detection

The edge detection algorithm used is a much simplified variant of the Sobel edge detector. In the Sobel mask the gradients around the middle pixel in the mask are calculated in two steps: horizontally (Table 5.1) and vertically (Table 5.2). While the Sobel mask, if used fully, calculates the direction of the gradients, the only information used in our solution is the magnitude of the edge.

-1	0	+1
-2	0	+2
-1	0	+1

Table 5.1: Gx

+1	+2	+1
0	0	0
-1	-2	-1

Table 5.2: Gy

The magnitude of the edge at a pixel in the Sobel mask is described by $G = \sqrt{Gx^2 + Gy^2}$. As square roots are computationally expensive the magnitude calculation is usually simplified to $G = Gx + Gy$. In this solution the mask is also simplified. The gradient calculation only uses the pixels directly above, below and to the sides, as in table 5.3.

0	+1	0
-1	0	+1
0	-1	0

Table 5.3: Simplified gradient mask (GxGy).

To make the edge response a bit steeper though, the actual calculation used in the solution looks like this: $G = Gx^2 + Gy^2$. An example of the result of using it on the Y-channel of an image, such as the one in Figure 5.3, can be seen in Figure 5.4.

Every pixel in the source image coming from the camera gets its gradient calculated. Finally a thresholding is applied so that the resulting gradient is either zero or 255, using a threshold value of 10. This value was chosen after a number of tests and proved to give satisfactory results. The resulting image after the threshold has been applied is shown in Figure 5.5.

Figure 5.3: Y-channel of camera image.

The source image itself is not modified during the procedure, but instead it's analyzed by the Sobel mask and the results are written to a destination image. This image can be viewed on a PC at run-time and is useful for testing purposes.

It should be noted that the source image referred to is actually the Y-channel (intensity) of the original YUV-image, not the complete colour image. There are two reasons for this. Firstly, the colour segmentation is not used because the tuning and configuration needed for it to function properly was not part of the project and could have been an additional source of errors. Secondly, this way, lines of any colour against any background colour (of course with a certain degree contrast) can be used. In this project the "ground" was green and the lines were white, but as long as there is contrast between the two the colours themselves are not important.

5.2.3 Thinning

The Sobel edge detection with thresholding does not leave clean one-pixel-wide edges. This is clearly visible in the magnified image in Figure 5.6. Wherever there is a gradient that is high enough to pass the thresholding there will be pixels on the resulting edge detection image, and the detected edges are as good as never only one pixel wide. Performing the Hough transform directly after edge detection would thus not be a good idea. The reason for this is that if a very soft edge is passed through the Sobel mask it would make a wide edge as a result. A wide edge can be thought of as a solid rectangle instead of a line, and the most prominent line (in Hough terms that could mean the longest one) in a solid rectangle is obviouisly one of the diagonals, which is not the desired line.

To remedy this, thinning is performed on the edge detection image. The purpose is to thin every edge down to one-pixel-wide edges to obtain cleaner results. Again a kind of convolution mask is used. Trying to decide whether or not to remove a pixel based on a convolution mask can be difficult since it is a local procedure, i.e. it is difficult to decide if a pixel is an unwanted "sprout" from a line when one is only looking at a 3x3 grid.

The thinning algorithm works like this: All the surrounding eight pixels of a pixel are scanned and their "on" or "off" status (since they thresholded) is converted into bits, conveniently forming an 8-bit code, which converted into a byte gives a number between 0 and 255. This number is then used as an index in a pre-computed array (see Figure 5.8) to determine if the pixel in the middle should be erased or not. This approach was chosen because it felt like a simple and, at the same time, fast way to do this work.

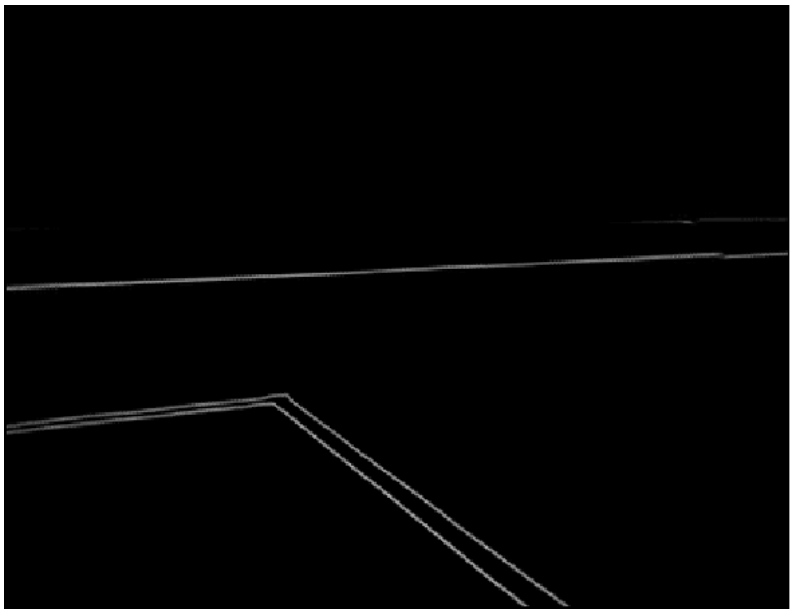

Figure 5.4: After edge detection only.

Normally when a convolution mask is used it is applied only once on each position. For the thinning algorithm this is not the case. There is no guarantee that the thinning has thinned an edge to be only one-pixel-wide the first time the algorithm is run if it runs once from top to bottom. Instead one would have to iterate over the image until no changes are detected, since a change in the image will change the results of the same test being performed again.

Iterating several times over the image could be costly. An optimization used in this case is based on the fact that if a pixel is removed during thinning, it can only affect the result of previous thinning calculations as far back as one row up and one column to the left. This is because the changed pixel would then appear in the lower right corner of the mask. Obviously there is no need to restart the entire iteration, but instead just step back and re-run it from where the latest change is made. This way the image is only processed once, albeit in possibly several minor loops. The results obtained by running the algorithm can be seen in Figure 5.7.

5.2.4 Hough Transform

Theory

The actual line detection is made using a variant of the Hough Transform [6]. There is a Generalized Hough Transform [13], which can be used to detect complex shapes. However, the variant used in this project is simply known as the Hough Transform [14].

The idea of the transform is to scan an image for pixels and see if they belong to a line. Any pixel found is possibly on a line, or actually an infinite number of lines with certain parameters. The possible line parameters (discretized of course) are recorded for every pixel found. When the entire image is scanned, the parameter combination with the most recorded hits is most likely a line.

A line may be described as $y = kx + m$, with k being the slope of the line and m the intercept on the y-axis. The problem with this form is that vertical lines cannot be described as they would have an infinite slope. Instead, polar coordinates are used. The two parameters r and θ represent the orthogonal distance from the origin and the angle of the line, respectively. Described in this way, the expression for a line is

$$r = \cos(\theta)x + \sin(\theta)y \tag{5.1}$$

where x and y are coordinates for a point on the line. This point is a pixel on the image being analyzed. To record the parameters for the lines of all the pixels, an *accumulator* is used, sometimes referred to as *Hough*

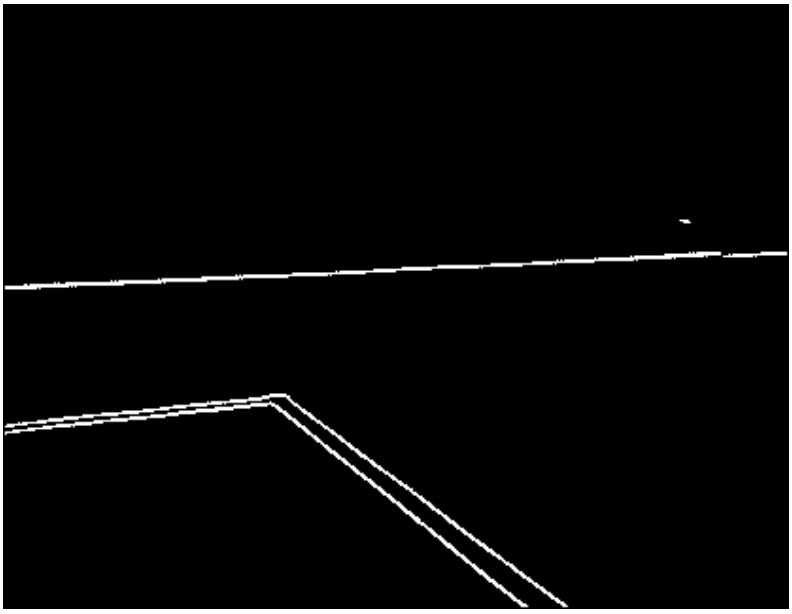

Figure 5.5: Edge detection image after thresholding.

space. In the case of the line detecting variant of the Hough Transform there are two parameters that need to be recorded, so the accumulator has two dimensions, namely r and θ. The accumulator is essentially a grid where every coordinate is known as a *bucket*.

When a pixel is found on the source image, the position of it is (x, y). By letting the angle θ vary and using expression (5.1), one will get a sinusoidal curve. This curve is plotted on the Hough space, and every bucket that gets *hit* increases its value by one.

A simple example image with six pixels arranged in a line with space between them and the resulting accumulator after running the Hough Transform can be seen in Figures 5.9 and 5.10.

This procedure is repeated for every pixel found in the source image. When all pixels have been processed there have been an equal number of curves drawn in the accumulator. The bucket with the highest value (most hits) represents the r and θ parameters of the most prominent line in the source image.

It should now be clear why it is important to have a clean edge detection image to work with as source image for the Hough Transform.

Example of a continuous straight line and the resulting accumulator after applying the Hough Transform can be seen in Figures 5.11 and 5.12.

Practical problems

As mentioned above there is an infinite amount of possible lines passing through a single point. At the same time the accumulator meant to record the parameters is a grid with discrete coordinates. This causes a quantization of the line parameters. The implementation in this solution only allows integer values since they are naturally easy to use as indices for the accumulator array.

So, when plotting the curve for a pixel in Hough space, the angle θ is stepped up by 1 degree at a time. Since the value of function (5.1) might be climbing more than 1 step at a time, the curve will be discontinuous in those cases in the accumulator. Considering that the method to detect lines is to find where several curves contribute to a high bucket hit count by intersecting each other, this causes some problems, since curves that intersect do not necessarily score hits in the same bucket.

For an example, consider Figure 5.13. The six curves plotted by stepping the θ value do not all generate a hit in the same bucket in the intersection. In reality, actually only three of the curves in this case happen to occur in the same bucket. Still, it is the bucket with the highest number of hits in the entire accumulator.

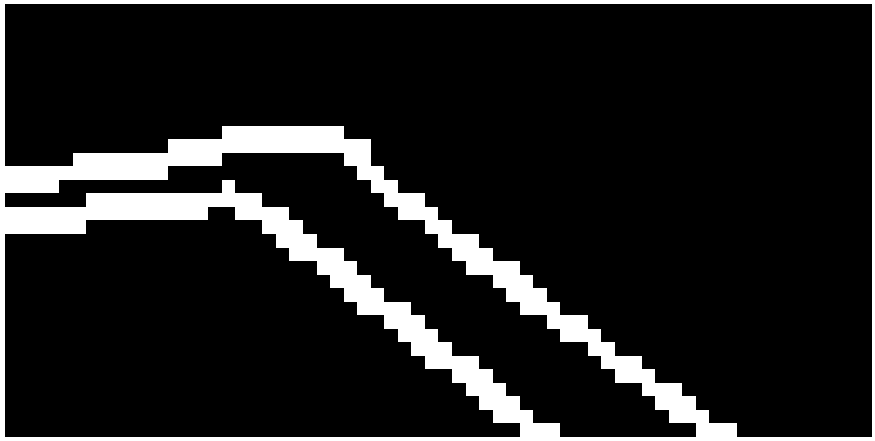

Figure 5.6: Blow-up of line corner in Figure 5.5.

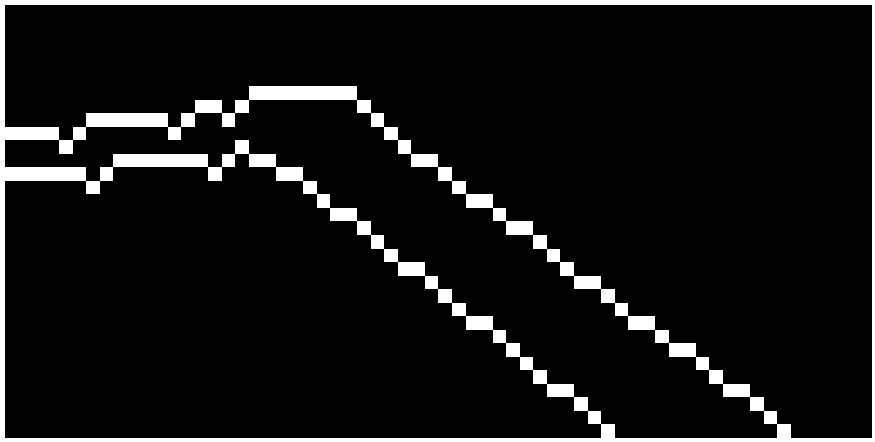

Figure 5.7: Figure 5.6 after thinning.

Optimizations

By varying the stepping of the accumulator axes, and thus the quantization of the line parameters, some optimizations can be made. If a stepping of 2 is used for the r-axis the number of discontinuities will be reduced to places where the function value r of function (5.1) jumps by more than 2 while the angle is stepped by 1. Increasing the stepping to 2 for example on the θ-axis also gives optimizations since there will be even more hits in the same bucket. At the same time the number of calculations used to plot the curve can be reduced because θ can be stepped by 2 as well. This gives yet another speed optimization when searching for the bucket with the most hits, since the search space (accumulator dimensions) will have decreased. The downside of these optimizations is loss of precision. When, for example, the θ-axis increases its stepping, the detected lines can only assume angles in intervals corresponding to the stepping. Hence, a stepping of 10 on the θ-axis would increase the calculation speed tenfold, as would it increase the accumulator scanning speed tenfold, at the same time as all lines would be limited to 10-degree steps.

When varying θ to plot the curve in the accumulator the initial thought is to let the angle vary through the interval $0 <= \theta <= 359$. However, it turns out that no lines in the interval $180 <= \theta <= 270$ will ever be present in the image. Knowing this, the accumulator can be reduced to range from -90 to 180 degrees instead, to get a continuous range. The size is reduced by 25%, and it also means that the calculations no longer need to run through the whole interval.

Additionally, the practical range of θ of a line is only really 180 degrees since the line has no "direction" like

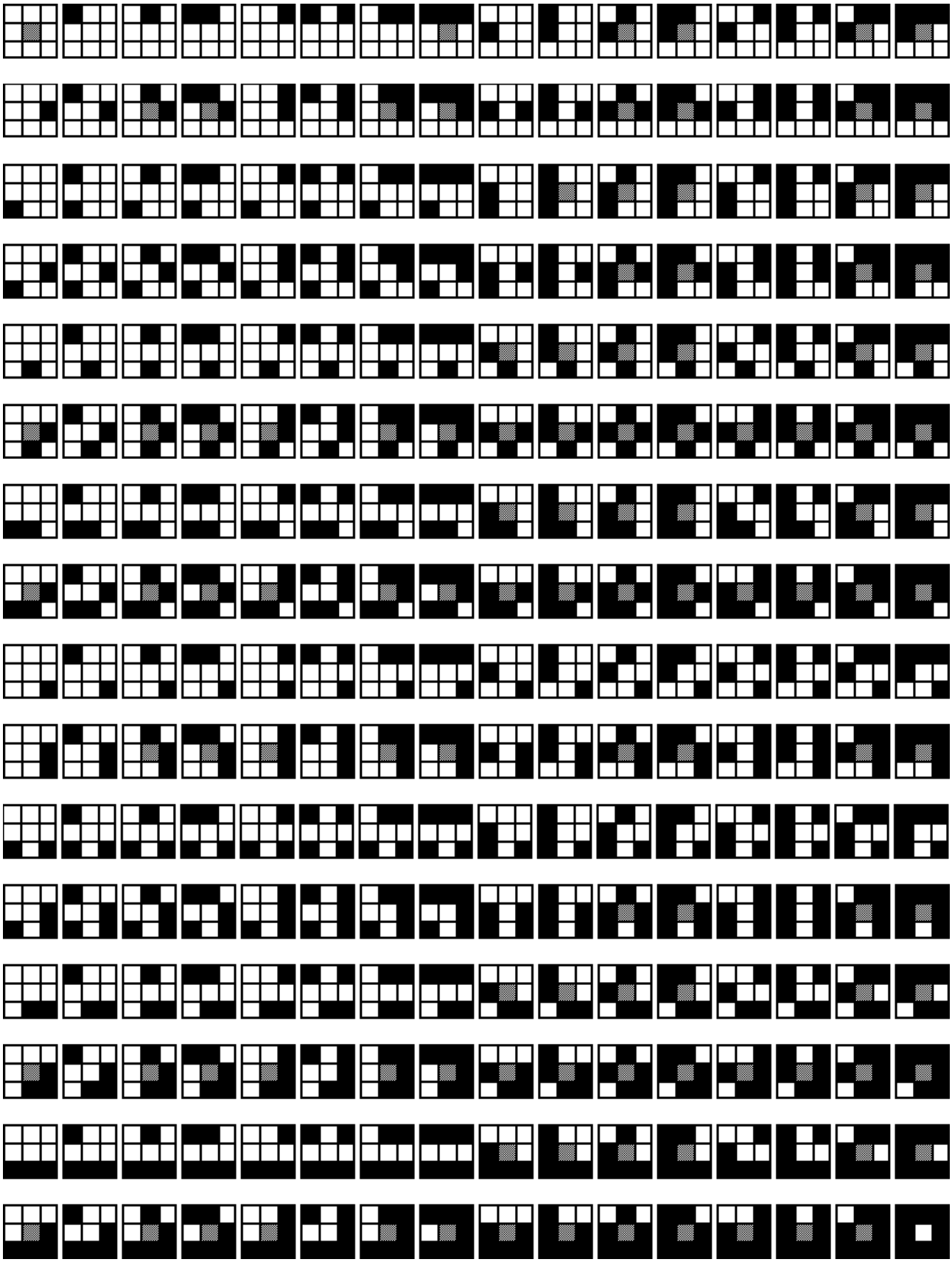

Figure 5.8: Thinning mask array. Grayed center means remove center pixel.

Figure 5.9: Some points aligned in a line.

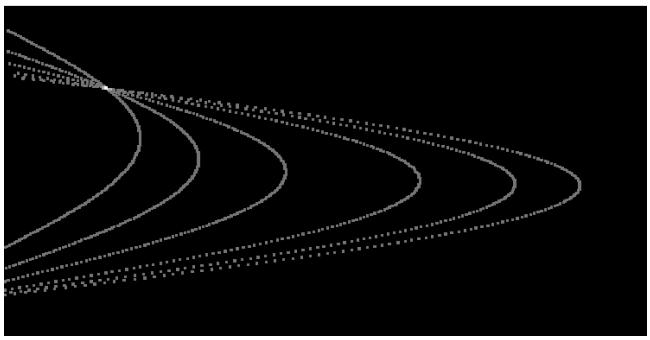

Figure 5.10: Accumulator contents after Hough Transform of Figure 5.9.

vectors do. This means that a line with $\theta = 45°$ has the same slope as a line with $\theta = 225°$. First the angle ϕ to the pixel being processed is calculated by $\phi = \arctan(\frac{p_y}{p_x})$, where p_x and p_y are the coordinates for that pixel. The interval θ is stepped through is obtained by $\phi - 90° <= \theta < \phi + 90°$.

For an illustration, see Figure 5.14. All lines in the example have positive r-values since they are the only ones recorded in this implementation. The upper right quadrant is the actual camera image. In the camera, line L1 would be just barely visible in the lower portion of the image and it would have a negative θ of around $-80°$. Line L2 runs almost diagonally across the image and has $\theta = 45°$. Line L3, like L1, is just barely visible, but in the left edge instead, and with a θ of around $170°$. Line L4 however can never be visible in the image, and with its θ of about $225°$ it is indeed in the interval $180 <= \theta <= 270$. The only time when a line with that angle can be visible is if it has a negative r-value, but then it can just as well be considered to be at $\theta \pm 180°$. One would then need to negate r to compensate for the rotation. This line would then still not be in the camera.

By using the source image seen in Figure 5.3, applying the edge detection, thresholding, thinning and finally the Hough Transform to it, the resulting accumulator can be seen in Figure 5.15.

Accumulator peak finding

After performing the Hough Transform the peaks must now be found since they represent the actual lines. The way this is made is by first setting a threshold value. This value is chosen depending on the source image resolution and the dimensions of the hough accumulator, since they influence the number of hits that will be generated in the buckets. For example, the highest resolution (416x320) will give roughly twice as many pixels

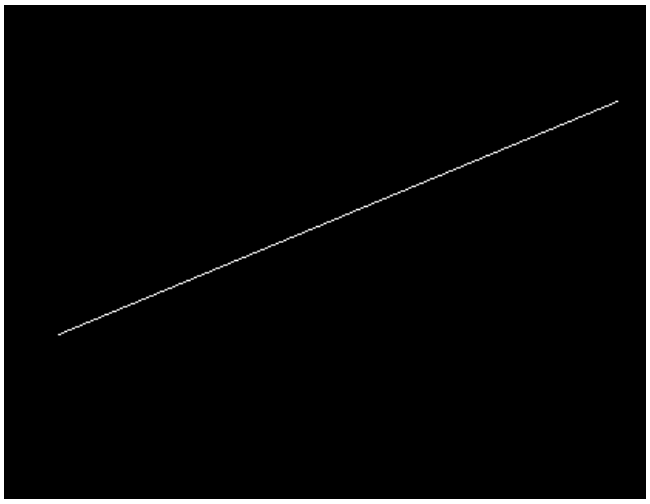

Figure 5.11: A line with the same parameters as the imagined line of Figure 5.9.

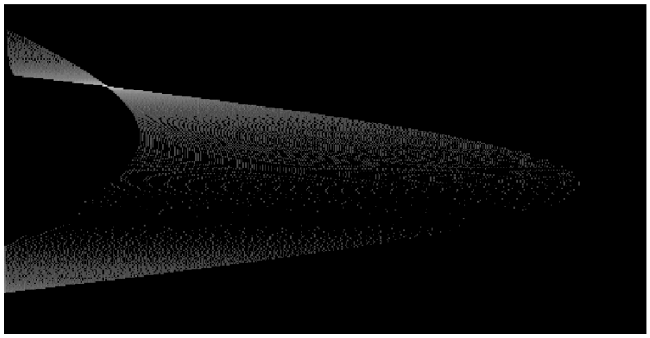

Figure 5.12: Accumulator contents after Hough Transform of Figure 5.11.

to run the Hough Transform on (after thinning), and setting the stepping to 2 on both axes of the accumulator will on average put four times as many hits in the same bucket. It turns out that this threshold value is one of the most important parameters in the entire process. Setting it too high will result in lines not being detected while a too low threshold will litter the results with a lot of false positives.

To try to make an efficient peak finder, all buckets with a number of hits equal to or higher than the threshold are copied to a separate data structure (a map, with the bucket "coordinates" as keys). From this point the accumulator is no longer used since all the interesting buckets are now stored in the map, which is significantly smaller and faster to search through than the entire accumulator. The peak detection then works like this:

1. Find the bucket with the highest number of hits.
2. Store the bucket values (θ, r, hits) in a list.
3. Remove all buckets in a rectangular area (in hough space) in the vicinity of the chosen bucket.
4. If there are more buckets in the map, go to step 1.

The reason for step 3 is that any lines that are detected so close to the line represented by the chosen bucket are most likely false positives due to the quantization effect, causing several peaks next to each other in the accumulator, which all originate from the same curve intersection. Another source for false positives is that a nearby peak might be the opposite edge of a real line. For example, the white line on the ground in Figure

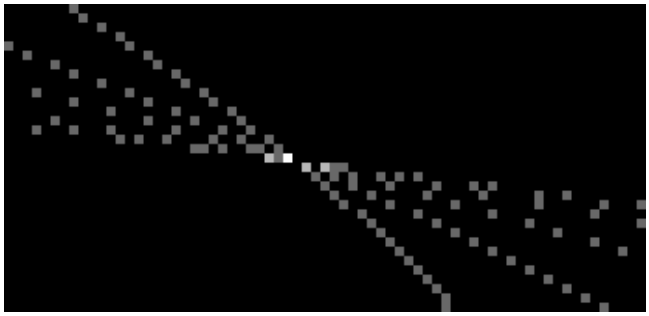

Figure 5.13: Magnification of curve intersection of Figure 5.10.

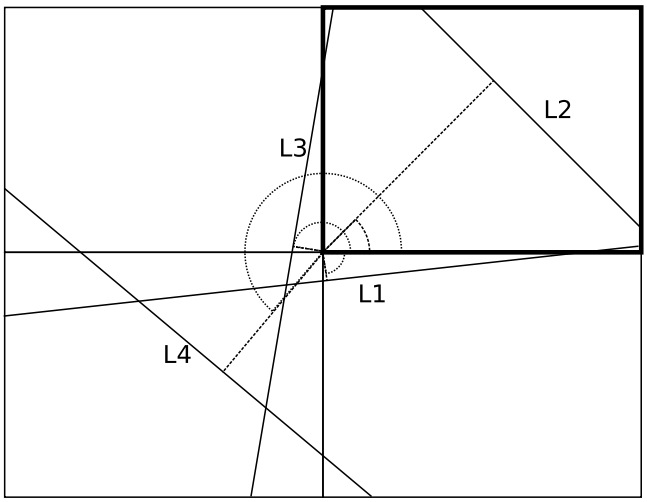

Figure 5.14: Lines and angles and their placements in the 4 quadrants.

5.3 produces two edge lines in the edge detection image. They both belong to the same physical line and it is probably preferred that only one of them is reported.

The size of the area to clear around a bucket is also a sensitive parameter. Currently it is constant but should probably be dynamic and be able to change depending on if the line represented by the bucket is close to or far away from the robot.

Having completed the peak detection, there is now a list of buckets representing lines in the image. The line detection in itself is now complete, but to make use of the information the lines must be transformed to robot coordinates so that they can be used for localization.

5.2.5 Line Transform

To transform the Hough line to robot coordinates the two points where the detected line intersects the camera border are calculated. Tekkotsu then offers a method to project each of these points to the ground. The resulting point coordinates are relative to the camera. In this step some checking can be made to see if the detected line is actually on the ground. Depending on the environment, lines that are not really on the ground might be detected but are not wanted. They can be removed by investigating the two intersect points. This is made by projecting them to the ground plane and looking at the point coordinates. If the coordinates are behind the camera, then the points must be above the horizon obviously, since they are actually in front of the camera.

Figure 5.16 illustrates the difference between a point on the ground and a point above the horizon. Since $p1$ is on the ground, the distance $d1$ from the camera turns out positive. However, when projecting $p2$ to the

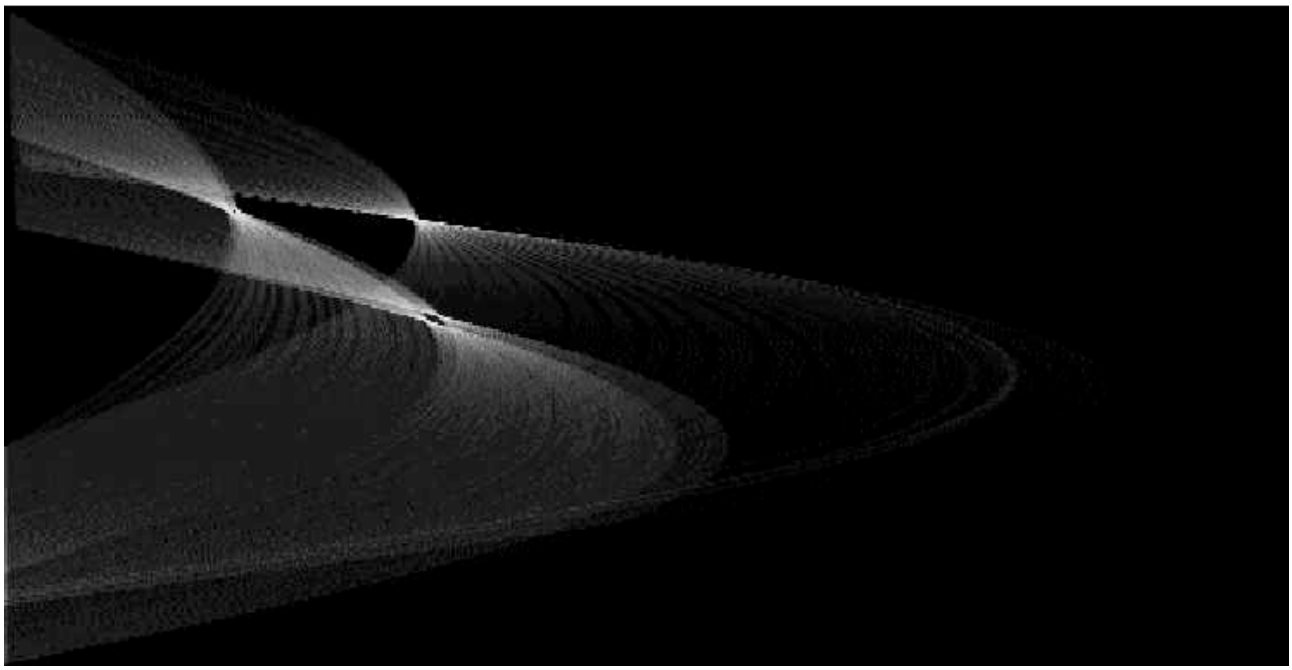

Figure 5.15: Accumulator contents after performing Hough Transform on Figure 5.5.

ground plane, yielding the false point pf, the distance $d2$ to pf turns out to be negative, but since we know the camera only can see in front of itself and as a consequence only positive distances exist, that must mean that the point is indeed not on the ground.

If both points appear in front of the camera the line is guaranteed to be entirely on the ground. This detection cannot determine whether a line is on the ground or not if one of the projected points appear to be above the horizon and the other on the ground. However, if the environment is controlled and well defined, all such lines should be on the ground.

An attempt is made to reduce the number of such lines anyway by moving the point that is above the horizon down along the line to the horizon. There is now a line segment from the edge of the camera up along the line to the horizon. If that line segment is shorter than a certain threshold value, then it is likely that most of the pixels that contributed to this line were detected above the horizon and thus the line is not on the ground. This threshold is set to be the same as for the peak detection to ensure that the pixels contibuting to the line are on the ground. In a way, one can think of the number of hits in a bucket as the "length" of the detected line. So, if the length "on the ground" is too short then the line is most likely a false positive and can thus be discarded.

Once these tests are passed, one point that is guaranteed to be on the ground (a point from the camera border intersect) and a point on the line just below the horizon are projected to the ground. One of them is used as a reference point and a directional vector is obtained by subtracting the coordinates of one from the other. This reference point and the directional vector describes the line completely and it's parameters are stored in a list. Finally, when all lines have been transformed, this list is broadcast as an event in Tekkotsu for other modules listening to receive.

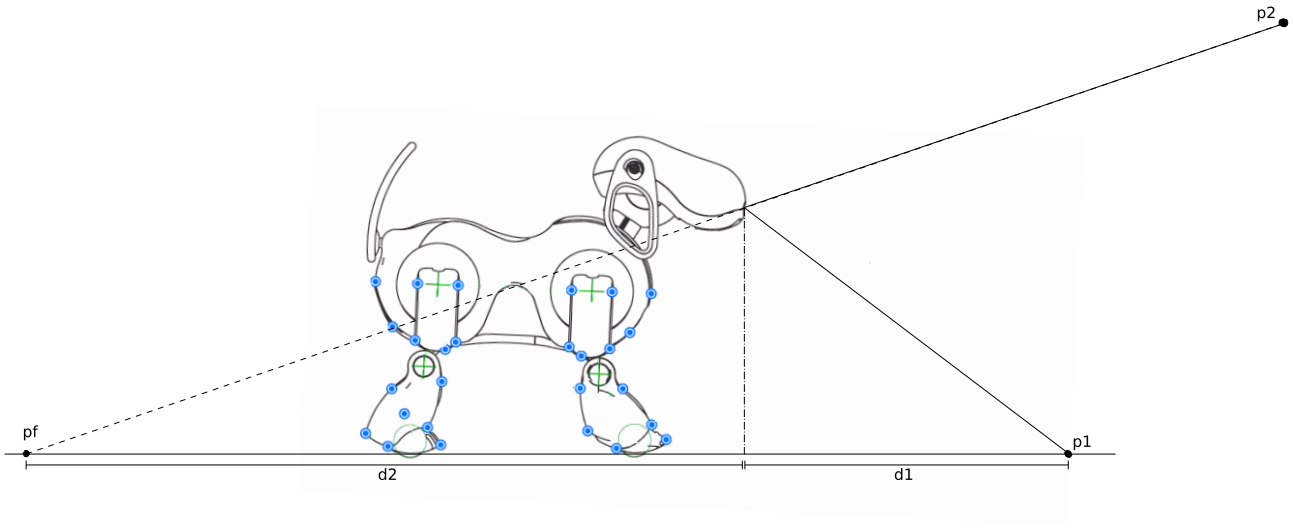

Figure 5.16: Two real points and how they are projected onto the groundplane.

Chapter 6

Results

A description of how performance was measured and the data collected is presented in appendix A. The detailed data can be found in appendix A.2 with an overview of the measurements presented in appendix A.3. Measurement numbers in the following sections refer to the data table mentioned in the beginning of each section.

6.1 Distance accuracy

The distance measurements are described in appendix A.4 and the data presented in table A.2.

Overall, the accuracy of the measured distance decreases with the actual distance, as might have been expected. The most reliable results are obtained when the distance is less than half a meter. See measurements 1-6. The first shows a relative error of 10%, probably due to poor manual measuring. An measure error of 15 millimeters at such a short distance most likely accounts for the high error. Counting from measurement 2 and up one can see that the relative error grows steadily from 0% at 250mm up to 37% at 1500mm. The 0% error at 250mm is just a fortunate average of the samples gathered but it is still only varying between 237 and 260mm. The high error at 1500mm though is a sure sign that the distance is too high for accurate readings.

The angle of the line does not seem to have any influence on the accuracy of the distance calculation, and it really shouldn't.

Image resolution does influence accuracy but mostly at the lowest resolutions (Compare measurements 7-12). The 4 highest resolutions have a relative error below 6%, while the two lowest have an error of 10 to 20 %. This may be due to both the camera resolution being lowered plus the lower resolution of the accumulator, since it is dependant on the image resolution.

Looking at measurements 22 and up, the relative error ranges from 28 to 38%. This is easy to explain for the times when the accumulator parameters are changed to lower the resolution of the accumulator. However it does not make sense for reading 22, which should have approximately the same errors as measurement 8, where the only difference is a line angle of 30° instead of 45°.

In practice, it would be advisable to keep the resolution to at least the quarter layer setting. Also, any lines detected above half a meter away should be deemed too inaccurate to be useful for any precision reading.

6.2 Angle accuracy

The angle measurements are described in appendix A.5 and the data presented in table A.3.

The angle accuracy seems to suffer less from both distance and resolution settings compared to the distance accuracy. Measurements 1-6 show only very small errors, with the greatest error at distance 1500mm, and then the error is only around 3°.

In the resolution test in measurements 7-12 the accuracy is good all the way down to 8th layer resolution. At 16th layer resolution the angle is suddenly off by over 25°. This is a sudden decrease in accuracy but still surprisingly good throughout the range until then.

At longer distances and when the line is not at 0° the accuracy drops somewhat but still remains acceptable. The really bad results are due to either too low image resolution for the distance (measurements 12 and 15), or too big θ-intervals of the accumulator, which directly influences the possible angles a line can assume (measurements 25 and 31).

Overall, the angle measurement values are more accurate and reliable as long the accumulator θ-interval is kept at around 5 or lower.

6.3 Time performance

The time measurements for the different algorithms are described in appendix A.6 and the data presented in table A.4.

6.3.1 Edge detection

It is obvious that the higher resolution parameter settings produce a much longer execution time. The source image at maximum resolution is 416x320 pixels and contains four times as many pixels as the next resolution, 208x160. Therefore, it should be expected that the edge detection should run about four times faster for each step resolution is lowered. Looking at the measurements with all parameters being equal except the resolution, and letting it vary between D (416x320) and F (208x160), this does seem to be the case. Compare measurements 7 and 8, 13 and 14, 17 and 18. Also it holds for the other measurements where only the resolution is lowered one step.

For double resolution layer measurements, the edge detection stands for about 16%-40% of the total execution time. For the lower resolutions this number drops and approaches 0% at 16th layer resolution. This could be due to inaccuracies in the timing functions of Tekkotsu and the AIBO itself, or simply that the algorithm runs so fast (there are only 130 pixels to process at 16th layer) that the time is not possible to measure.

6.3.2 Thinning

Just like the edge detection, the execution time for the thinning algorithm depends on the size of the camera image, but also on how well the edge detection performed. If there are no edges that need thinning, the algorithm will go through the image just once. If there are a lot of "thick" edges, the thinning will iterate over those places and will take longer time to finish. It is therefore not as easy to predict the execution time of the thinning algorithm.

Contrary to the edge detection algorithm, there are no multiplications or divisions used in the thinning itself (but there are in the supporting functions used by all algorithms processing pixels or accumulator buckets), which may explain the faster execution time. Tendencies for shorter execution times can also be seen for the lower resolutions, as well as times of zero at the lowest resolutions.

The thinning algorithm does take less time to execute than the edge detection. Relative to the total execution time of all algorithms, the thinning takes 5%-13%, which is about three times as fast as the edge detection. This is only a rough estimate though and there are a lot of contradictory examples in the table.

6.3.3 Hough Transform

Not completely as a surprise, the Hough Transform turns out to be the heaviest of all algorithms. The execution time relative to the total time varies between 27% and 64% across all different setups and parameter settings.

An interesting difference compared to the edge detection and thinning is that the relative execution time of the Hough Transform goes up, instead of down, as resolution is lowered. This does not of course mean that it takes longer to execute, only that the other algorithms take up less of the time while the Hough Transform is still somewhat slow. The explanation is that the HT execution time also depends on the θ-stepping in the accumulator (see the optimization of the HT discussed in 5.2.4). Comparing measurements 22-25 one can see that the execution time for the HT drops steadily as the θ-stepping goes up. The same can be seen in measurements 29-31.

Overall, the Hough Transform is the heaviest and slowest of all algorithms used, and so naturally is the one where more optimizations would improve total execution time the most.

6.3.4 Peak detection

The peak detection turned out to be the fastest of the four algorithms. The absolute execution time never exceeded 48ms and the relative execution time compared to total execution time varied between 4% and 26%.

Since the peak detection analyzes the contents of the accumulator it is expected that the execution time is directly proportional to the size of the accumulator. Since the size of the accumulator depends on three factors, namely image resolution, θ-stepping and r-stepping, it should be expected that the peak detection runs faster when image resolution drops, θ-stepping increases and r-stepping increases. For three series of measurements that show that this is the case, consider measurements 7-12, 22-25 and 26-28 respectively.

In short, the peak detection has very little impact on the total execution time.

6.3.5 Total execution time

Looking at table A.1, it is clear that image resolution has a large impact on total execution time. The double resolution (416x320) has execution times ranging from 360ms to 1155ms. That is over one second! If execution time is not a problem then it does not matter of course. However, for time critical applications, the screen resolution can be lowered to half layer or even quarter layer without degrading accuracy too much. If one compares measurements 7 and 10, representing double and quarter resolutions, the total execution time drops from 1009ms to just 50ms. That is an execution time improvement by a factor of 20. In practice, 50ms corresponds to a framerate of 20 frames per seconds, which should be acceptable even for real-time applications.

Increasing the θ and r-steppings of the accumulator also improves total execution time, but not nearly as much as the image resolution does. The best improvement can be seen if comparing measurements 22 to 31. The total execution times are 310ms and 110ms respectively. That corresponds to a factor of 2.8 improvement if one uses the parameters in measurement 31. Considering the horrible accuracy at those settings though, it would probably be better to choose a lower stepping, such as the one used in 30 (θ and r-steppings 5) or even 29(θ and r-stepping 2). Those settings would give speed improvement factors of 2.6 and 1.67 respectively.

A good combination for getting somewhat accurate readings while maintaining a high framerate might be to set resolution to half layer (104x80) while setting θ-stepping to 2 and r-stepping to 5.

Chapter 7

Conclusion and future work

While the Hough Transform is an effective line detection algorithm, it is lacking the ability to find line segments on its own. This may limit the usefulness of this solution depending on the environment it is intended to be used in. It should be possible to add functionality for such a feature by trying to locate the detected lines in the source image and tracing them to see where they begin and end, but if the accuracy of the detected lines is low then the chances of success for such a feature might be low. None such feature was added to this solution.

Another refinement to this solution could be to first do colour segmentation on the image and only analyse edges between white and other colours on the field, or maybe even only between white and green. This would eliminate some false lines generated by contours of other objects and other features that resemble lines, and instead limit the detection to only detecting lines along the white lines on the playfield, if the environment is the RoboCup soccer playfield.

As was mentioned in the problem definition in 2.2, the original use for this solution was to be used in conjunction with another localization system that would look for the goals on the RoboCup soccer field. This type of use is probably the best application for this solution, namely as a part of a localization system, using other algorithms as well. Relying solely on the line data extracted by the Hough Transform gives a lot of uncertainty, as well as symmetry problems for example in the RoboCup environment if the different coloured goals are not used for localization.

Another solution that seems to work well is the *Scan Lines*-approach presented by Matthias Jüngel in [15]. It would be interesting to compare the performance of that solution to the one based on the Hough Transform presented in this report.

To summarize, this solution is very accurate at shorter distances and fast enough for real-time use (assuming the robot does not move around too much since the joint data will change during the calculation), but it suffers from poor accuracy at medium and longer ranges, as well as the inability to detect line segments. Its best use would be as a complementary system to another localization method.

Appendix A

Measurements and tables

A.1 Method

The results that were discussed in chapter 6 and presented here were measured by placing the AIBO on a green mat in 31 different setups. A long flat piece of metal with white tape on it was then moved to different positions each of thedifferent setups. For each setup the following parameters were measured: Real distance to line, real angle of line, calculated distance to line and calculated angle of line. The execution times for the different algorithm were also measured.

The distance to the line was measured from the Tekkotsu AIBO coordinate system origin (which is beneath the "chest" of the AIBO) to the line using a long ruler. The angle was measured using a graduated arc. Due to the difficulty in finding the exact origin, the manually measured angles and distances may have some errors in them, which may in turn make the measurements seem more or less accurate than they are. Even if the measurements are a few millimeters and/or degrees off, the results are still useful for comparison.

The resolutions in the tables are marked as D,F,H,Q,8 or 16. These correspond to the following resolutions:

- D - Double layer (416x320)
- F - Full layer (208x160)
- H - Half layer (104x80)
- Q - Quarter layer (52x40)
- 8 - Eigth layer (26x20)
- 16 - Sixteenth layer (13x10)

A.2 Figures

All the data measured is presented in Figures A.1 to A.31. Each reading is marked in the figures, which have the measured distance to the line on the x-axis and the measured line angle (in the tables refered to as α) on the y-axis. Please note that although 31 measurements were made, two of them, namely measurements 6 and 16, did not produce any readings as no line was detected. For this reason only 29 figures are presented here.

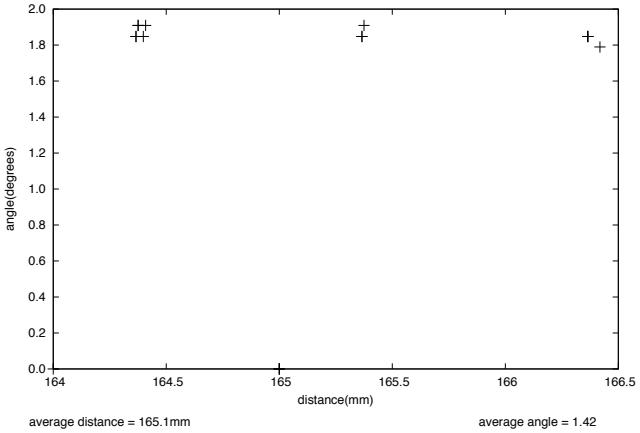

Figure A.1: d−150, α−0°, dbl res.

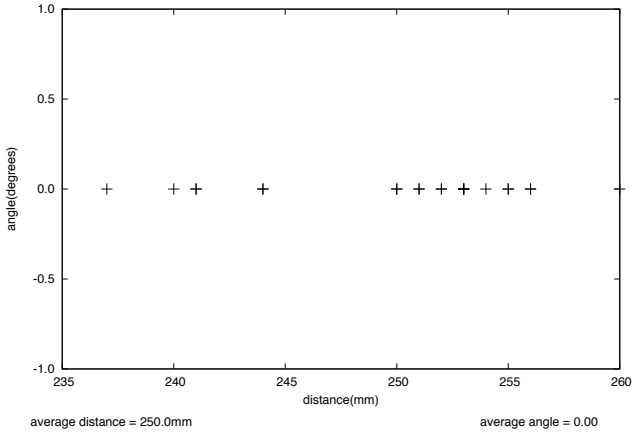

Figure A.2: d−250, α−0°, dbl res.

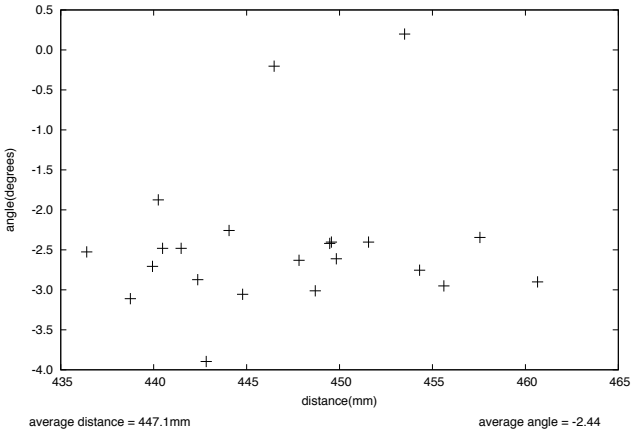

Figure A.3: d=500, α=0°, dbl res.

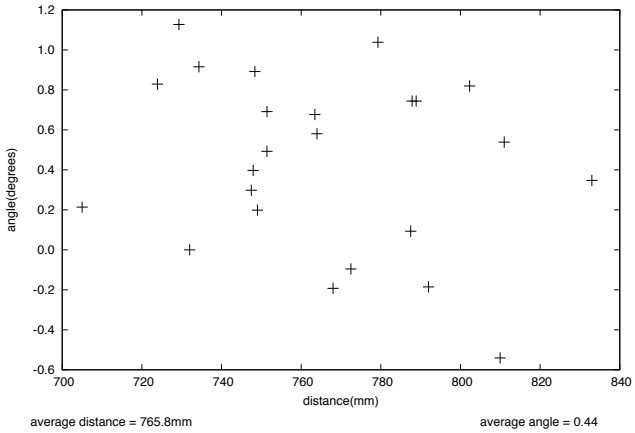

Figure A.4: d=1000, α=0°, dbl res.

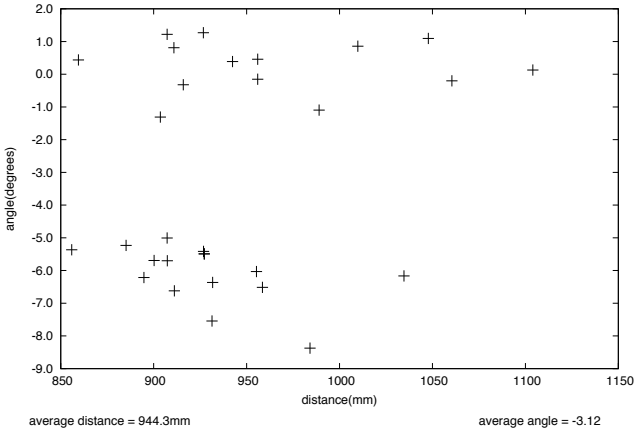

Figure A.5: d=1500, α=0°, dbl res.

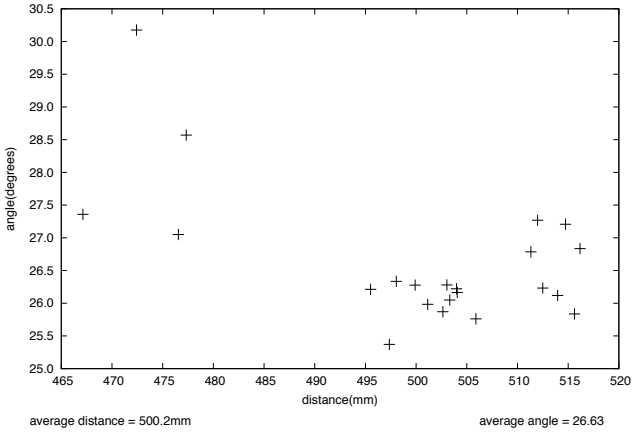

Figure A.7: d=500, α=30°, dbl res.

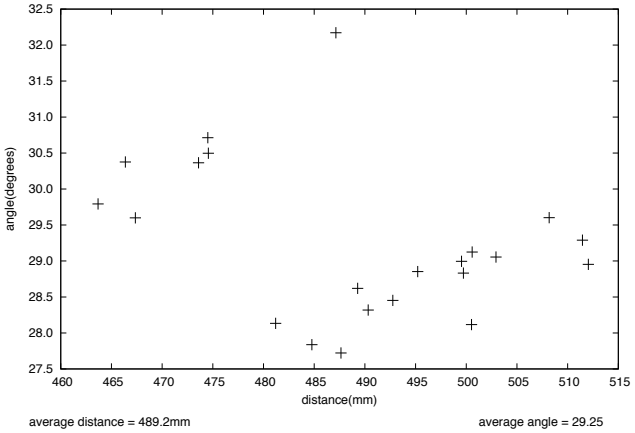

Figure A.8: d−500, α−30°, full res.

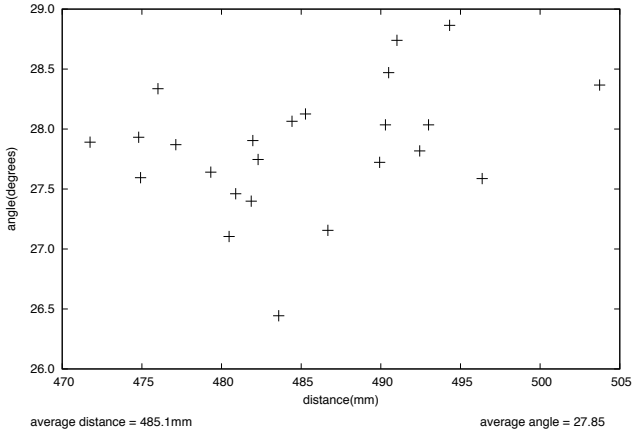

Figure A.9: d−500, α−30°, half res.

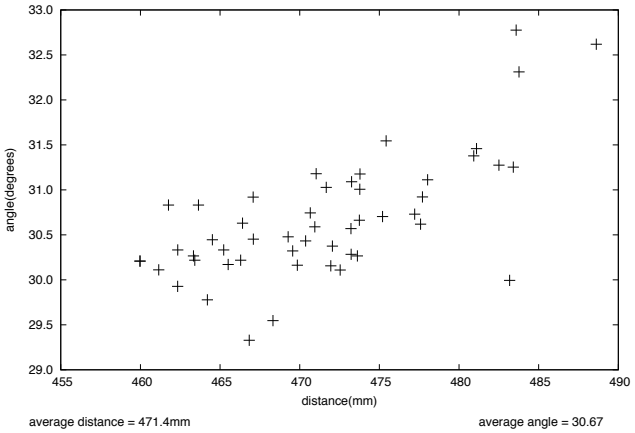

Figure A.10: d=500, α=30°, qrtr res.

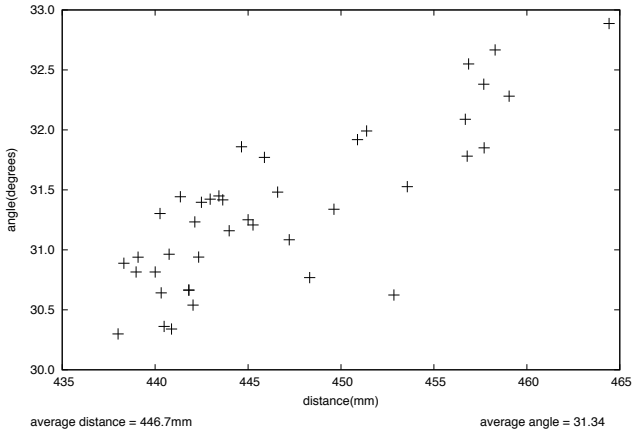

Figure A.11: d=500, α=30°, 8th res.

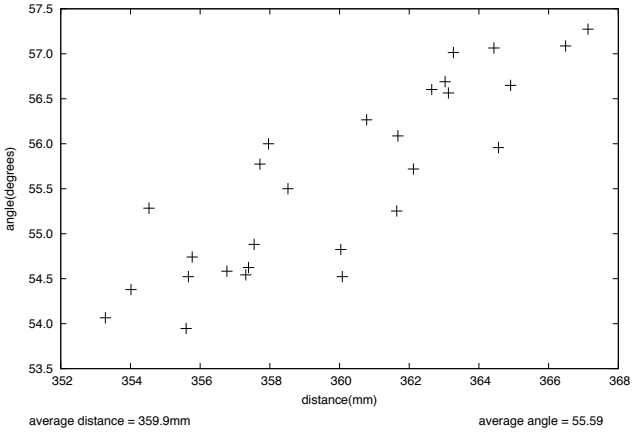

Figure A.12: d=500, α=30°, 16th res.

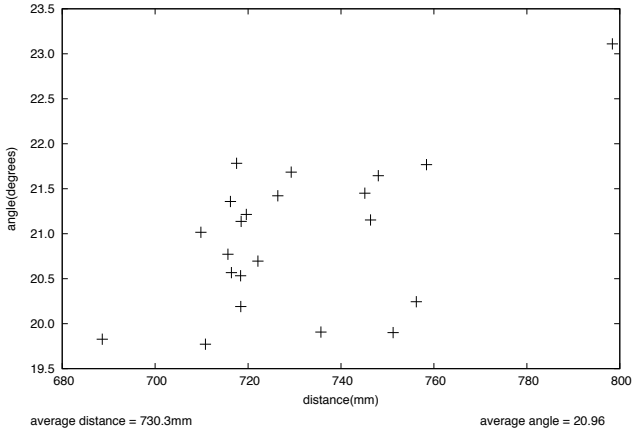

Figure A.13: d=1000 α=30°, dbl res.

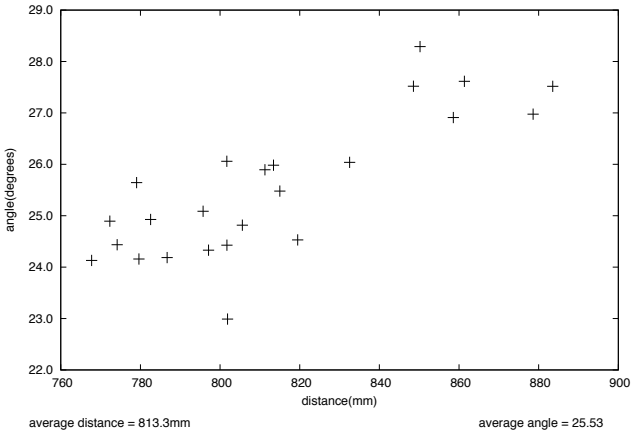

Figure A.14: d−1000, α−30°, full res.

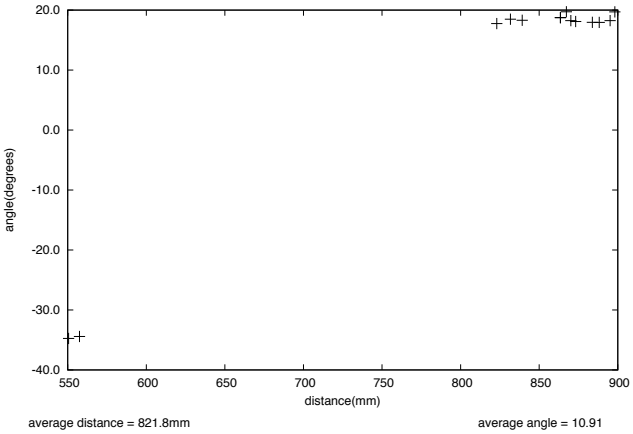

Figure A.15: d−1000, α−30°, half res.

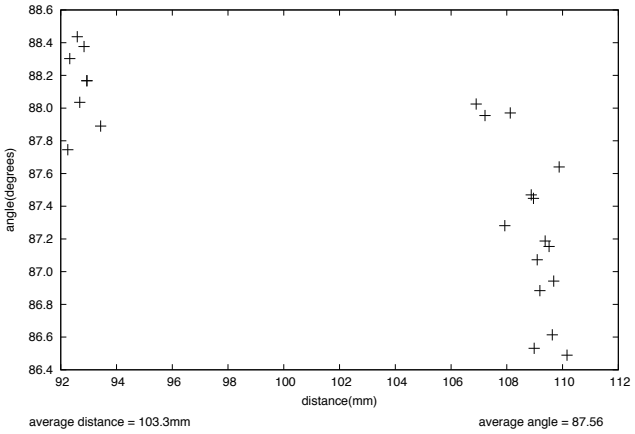

Figure A.17: d=100, α=90°, dbl res.

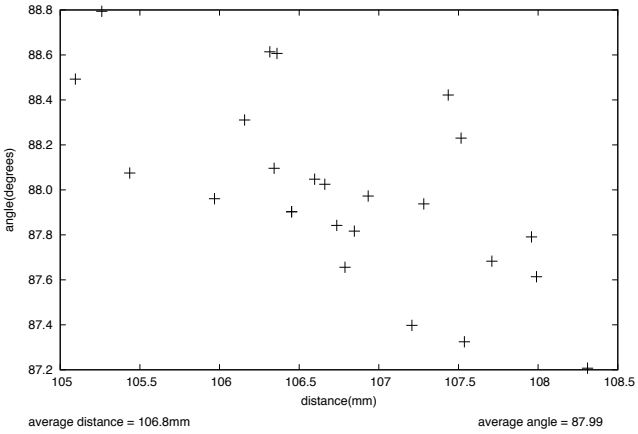

Figure A.18: d=100, α=90°, full res.

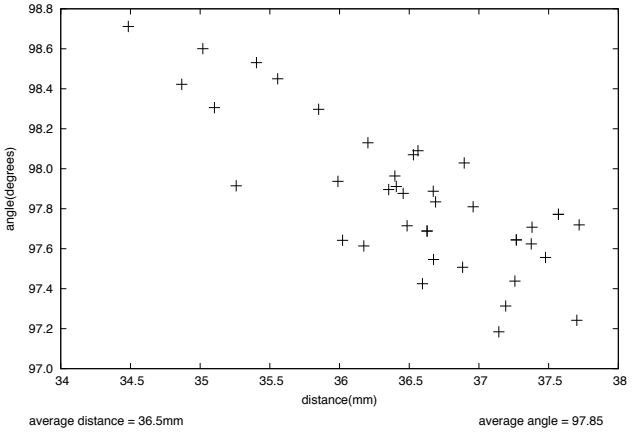

Figure A.19: d=100, α=90°, 8th res.

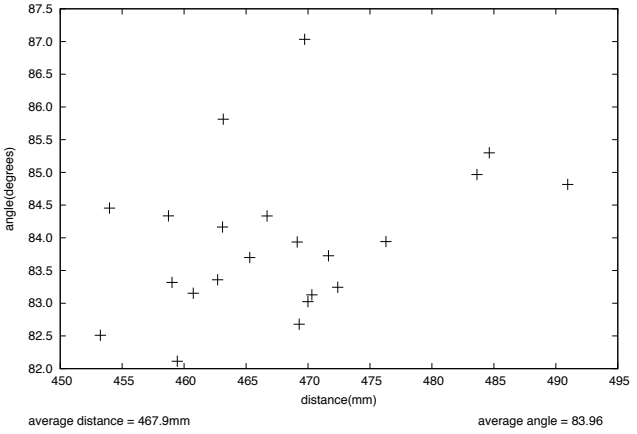

Figure A.20: d=500 α=90°, dbl res.

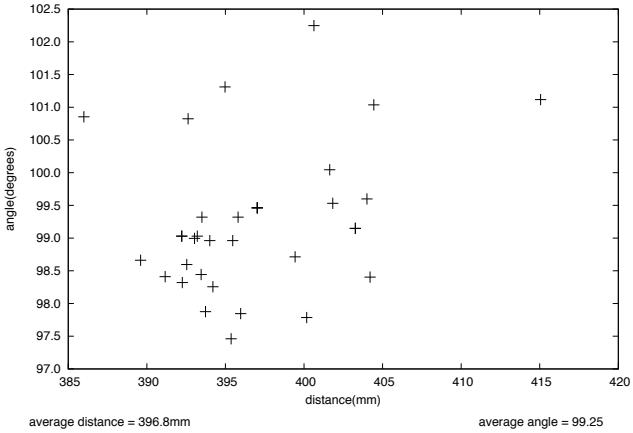

Figure A.21: d−500, α−90°, qrtr res.

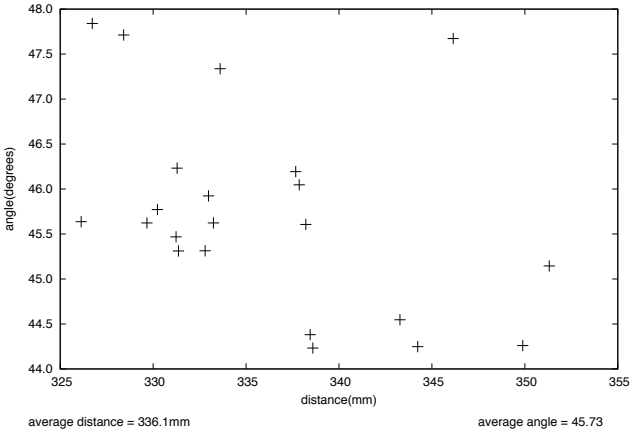

Figure A.22: d−500, α−45°, int_θ−1, int_r−1.

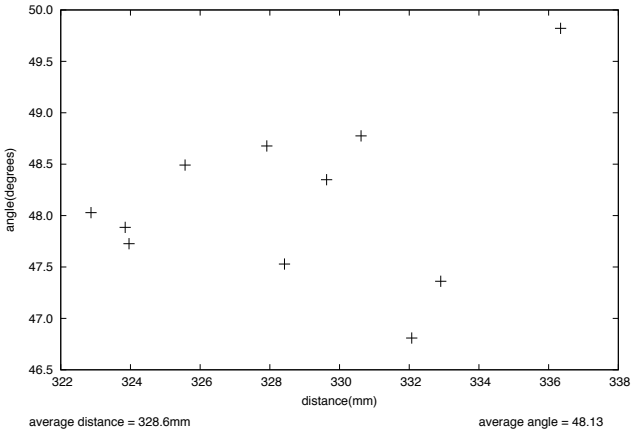

Figure A.23: d=500, α=45°, int_θ=2, int_r=1.

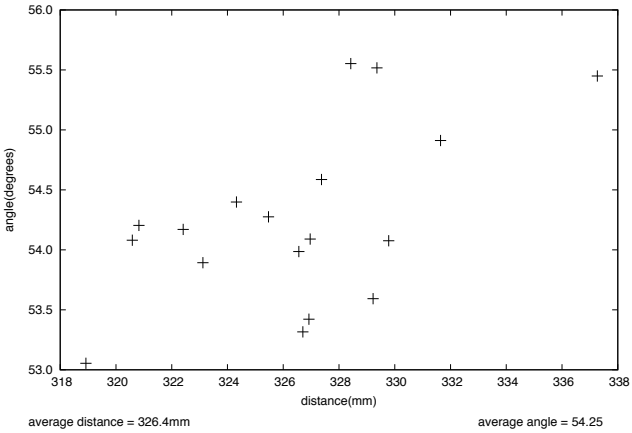

Figure A.24: d=500, α=45°, int_θ=5, int_r=1.

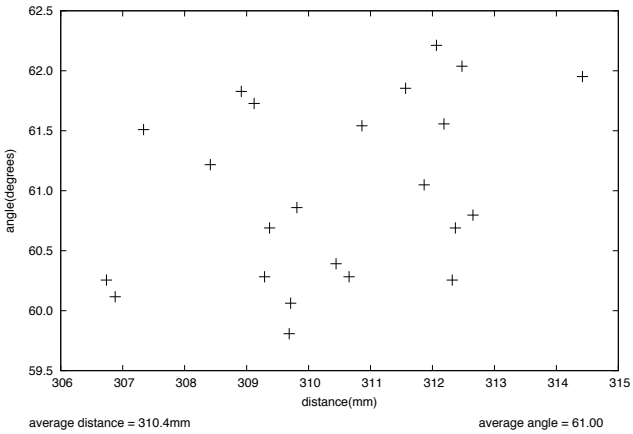

Figure A.25: d=500, α=45°, int_θ=10, int_r=1.

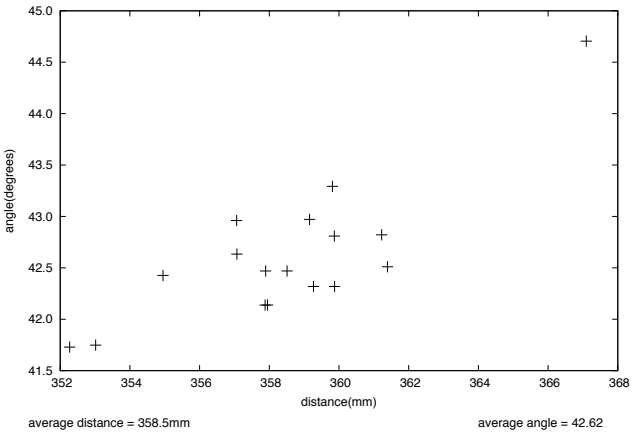

Figure A.26: d=500 α=45°, int_θ=1, int_r=2.

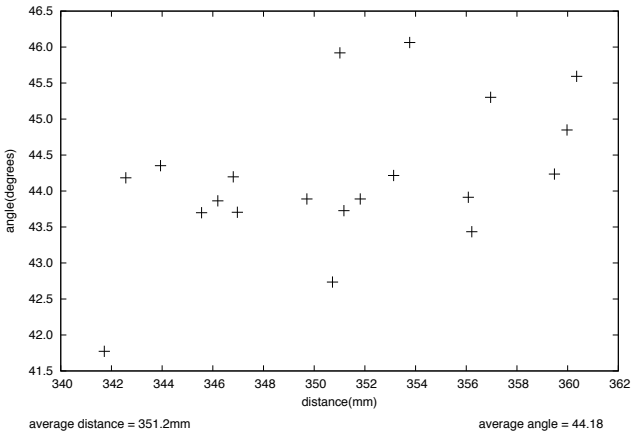

Figure A.27: d−500, α−45°, int_θ−1, int_r−5.

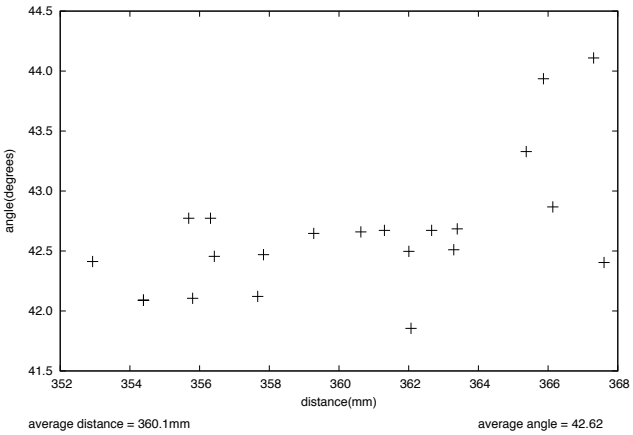

Figure A.28: d−500, α−45°, int_θ−1, int_r−10.

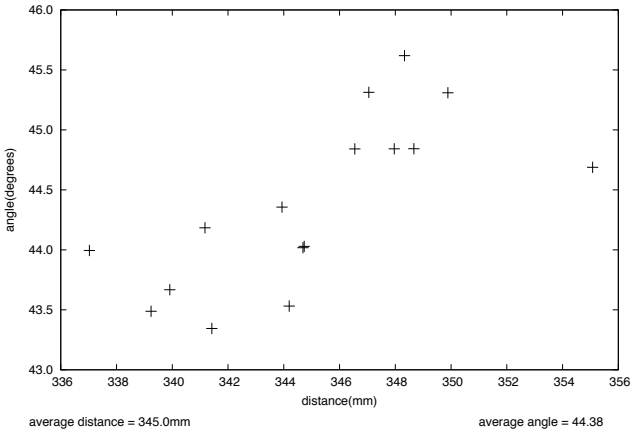

Figure A.29: d=500, α=45°, int_θ=2, int_r=2.

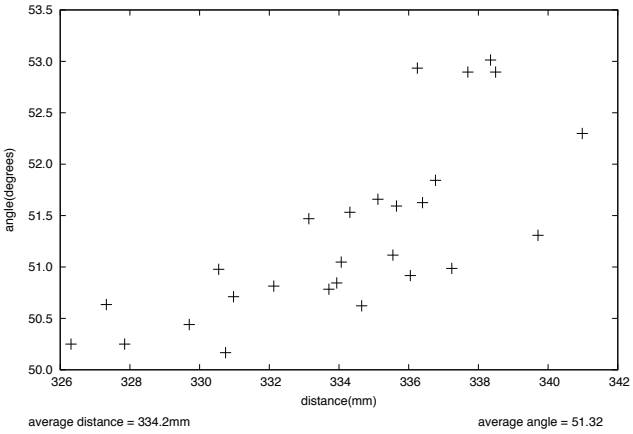

Figure A.30: d=500, α=45°, int_θ=5, int_r=5.

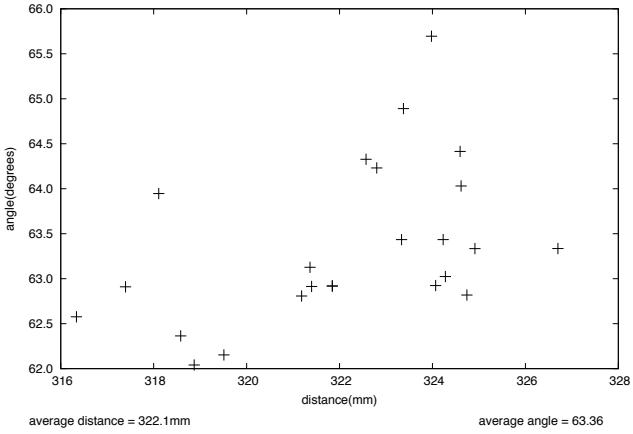

Figure A.31: d=500, α=45°, int_θ=10, int_r=10.

A.3 Data overview table

Table A.1 shows general values for each measurement. The actual distances and angles, measured distances and angles, as well as parameters for the Hough Transform are shown here. The table headers have the following meaning:

- Nr - Identification number of measurement (for referencing).
- Res - Camera resolution layer D,F,H,Q,8,16 for Double, Full, Half, Quarter, 8th and 16th respectively.
- int_θ - Interval of θ-axis in accumulator.
- int_r - Interval of r-axis in accumulator.
- Real dist (mm) - The actual distance to the line being measured.
- Real α (°) - The actual angle of the line being measured.
- Avg meas dist (mm) - Average distance to line from all readings.
- Avg meas α (°) - Average angle of line from all readings.
- Figure - Data figure showing all individual readings for this measurement.
- Comment - Additional comments. Sometimes the automatic threshold calculation did not work. There were problem with detecting a line at times as well. Such occations are noted here.

Nr	Res	int_θ	int_r	Real dist(mm)	Real $\alpha(°)$	Avg meas dist(mm)	Avg meas $\alpha(°)$	Figure	Comment
1	D	1	1	150	0	165.1	1.42	A.1	
2	D	1	1	250	0	250	0	A.2	
3	D	1	1	500	0	447.1	-2.44	A.3	
4	D	1	1	1000	0	765.8	0.44	A.4	
5	D	1	1	1500	0	944.3	-3.12	A.5	Problem seeing line
6	F	1	1	1500					No line detected
7	D	1	1	500	30	500.2	26.63	A.7	
8	F	1	1	500	30	489.2	29.25	A.8	
9	H	1	1	500	30	485.1	27.85	A.9	
10	Q	1	1	500	30	471.4	30.67	A.10	Problem seeing line
11	8	1	1	500	30	446.7	31.34	A.11	Manual threshold 10
12	16	1	1	500	30	359.9	55.59	A.12	Manual threshold 4
13	D	1	1	1000	30	730.3	20.96	A.13	
14	F	1	1	1000	30	813.3	25.53	A.14	
15	H	1	1	1000	30	821.8	10.91	A.15	Manual threshold 6
16	Q	1	1	1000	30				No line detected
17	D	1	1	100	90	103.3	87.56	A.17	
18	F	1	1	100	90	106.8	87.99	A.18	
19	8	1	1	100	90	36.5	97.85	A.19	
20	D	1	1	500	90	467.9	83.96	A.20	
21	Q	1	1	500	90	396.8	99.25	A.21	
22	F	1	1	500	45	336.1	45.73	A.22	
23	F	2	1	500	45	328.6	48.13	A.23	
24	F	5	1	500	45	326.4	54.25	A.24	
25	F	10	1	500	45	310.4	61	A.25	Manual threshold 30
26	F	1	2	500	45	358.5	42.62	A.26	
27	F	1	5	500	45	351.2	44.18	A.27	
28	F	1	10	500	45	360.1	42.62	A.28	Manual threshold 60
29	F	2	2	500	45	345	44.38	A.29	
30	F	5	5	500	45	334.2	51.32	A.30	
31	F	10	10	500	45	322.1	63.36	A.31	

Table A.1: Real angles and distances and their measured average values.

A.4 Distance error data table

In table A.2 the measured distances are shown. The minimum and maximum readings, as well as the errors of the average readings can be seen here.

- Nr - Identification number of measurement (for referencing).
- Real dist (mm) - The actual distance to the line being measured.
- Avg meas dist (mm) - Average distance to line from all readings.
- Rel error dist (%) - Relative error in percent between actual distance and average measured distance.
- Delta dist (mm) - Interval between minimum and maximum distance measured.
- Min dist (mm) - Minimum distance measured.
- Max dist (mm) - Maximum distance measured.

Nr	Real dist(mm)	Avg meas dist(mm)	Rel Error dist (%)	Delta dist (mm)	Min dist(mm)	Max dist(mm)
1	150	165.1	10.07	2.42	164.00	166.42
2	250	250	0.00	23.00	237.00	260.00
3	500	447.1	10.58	24.26	436.40	460.66
4	1000	765.8	23.42	127.98	705.00	832.98
5	1500	944.3	37.05	248.09	855.92	1104.01
6	1500					
7	500	500.2	0.04	49.04	467.13	516.17
8	500	489.2	2.16	48.36	463.68	512.04
9	500	485.1	2.98	31.99	471.75	503.73
10	500	471.4	5.72	28.64	459.98	488.62
11	500	446.7	10.66	26.42	438.00	464.42
12	500	359.9	28.02	13.85	353.28	367.13
13	1000	730.3	26.97	109.79	688.62	798.40
14	1000	813.3	18.67	115.77	767.75	883.52
15	1000	821.8	17.82	347.54	550.56	898.10
16	1000					
17	100	103.3	3.30	17.91	92.25	110.16
18	100	106.8	6.80	3.22	105.10	108.31
19	100	36.5	63.50	3.23	34.48	37.72
20	500	467.9	6.42	37.72	453.24	490.96
21	500	396.8	20.64	29.06	385.99	415.05
22	500	336.1	32.78	25.19	326.13	351.31
23	500	328.6	34.28	13.47	322.87	336.34
24	500	326.4	34.72	18.34	318.92	337.27
25	500	310.4	37.92	7.68	306.74	314.42
26	500	358.5	28.30	14.82	352.27	367.10
27	500	351.2	29.76	18.64	341.72	360.36
28	500	360.1	27.98	14.68	352.93	367.60
29	500	345	31.00	18.05	337.03	355.08
30	500	334.2	33.16	14.67	326.31	340.98
31	500	322.1	35.58	10.36	316.34	326.70

Table A.2: Real and measured distances with errors.

A.5 Angle error data table

- Nr - Identification number of measurement (for referencing).
- Real α (°) - The actual distance to the line being measured.
- Avg meas α (°) - Average distance to line from all readings.
- Abs error α (°) - Relative error in percent between actual distance and average measured distance.
- Delta α (°) - Interval between minimum and maximum distance measured.
- Min α (°) - Minimum distance measured.
- Max α (°) - Maximum distance measured.

Nr	Real α(°)	Avg meas α(°)	Abs Error α(°)	Delta α(°)	Min α(°)	Max α(°)
1	0	1.42	1.42	1.91	0.00	1.91
2	0	0	0.00	0.00	0.00	0.00
3	0	-2.44	2.44	4.09	-3.90	0.20
4	0	0.44	0.44	1.67	-0.54	1.13
5	0	-3.12	3.12	9.64	-8.37	1.27
6						
7	30	26.63	3.37	4.80	25.37	30.18
8	30	29.25	0.75	4.45	27.72	32.17
9	30	27.85	2.15	2.42	26.44	28.86
10	30	30.67	0.67	3.45	29.33	32.78
11	30	31.34	1.34	2.59	30.30	32.89
12	30	55.59	25.59	3.33	53.95	57.27
13	30	20.96	9.04	3.34	19.77	23.11
14	30	25.53	4.47	5.30	22.99	28.29
15	30	10.91	19.09	54.46	34.74	19.72
16	30					
17	90	87.56	2.44	1.95	86.49	88.44
18	90	87.99	2.01	1.59	87.21	88.79
19	90	97.85	7.85	1.53	97.18	98.71
20	90	83.96	6.04	4.92	82.11	87.03
21	90	99.25	9.25	4.79	97.46	102.25
22	45	45.73	0.73	3.61	44.23	47.84
23	45	48.13	3.13	3.01	46.81	49.82
24	45	54.25	9.25	2.50	53.05	55.55
25	45	61	16.00	2.40	59.81	62.21
26	45	42.62	2.38	2.98	41.73	44.70
27	45	44.18	0.82	4.29	41.77	46.06
28	45	42.62	2.38	2.25	41.86	44.11
29	45	44.38	0.62	2.27	43.34	45.62
30	45	51.32	6.32	2.85	50.17	53.01
31	45	63.36	18.36	3.66	62.04	65.70

Table A.3: Real and measured angles with errors.

A.6 Time performance data table

- Nr - Identification number of measurement (for referencing).
- Res - Camera resolution layer D,F,H,Q,8,16 for Double, Full, Half, Quarter, 8th and 16th respectively.
- int_θ - Interval of θ-axis in accumulator.
- int_r - Interval of r-axis in accumulator.
- Real dist (mm) - The actual distance to the line being measured.
- Real α (°) - The actual angle of the line being measured.
- $t_{Edgedet}$ Average time taken by edge detection algorithm during this measurement.
- t_{Thin} Average time taken by thinning algorithm during this measurement.
- t_{HT} Average time taken by Hough Transform algorith during this measurement.
- $t_{PeakDet}$ Average time taken by peak detection algorithm during this measurement.
- t_{Tot} Total time taken for the. Note that this includes overhead time and is not just a sum of the other times.

Nr	Res	int_θ	int_r	Dist (mm)	α(deg)	$t_{Edgedet}$ (ms)	t_{Thin} (ms)	t_{HT} (ms)	$t_{PeakDet}$ (ms)	t_{Tot} (ms)
1	D	1	1	150	0	191	34	208	48	684
2	D	1	1	250	0	182	149	411	47	1155
3	D	1	1	500	0	194	78	400	36	879
4	D	1	1	1000	0	204	94	335	36	893
5	D	1	1	1500	0	143	26	85	28	360
6	F	1	1	1500						
7	D	1	1	500	30	200	109	510	34	1009
8	F	1	1	500	30	45	42	224	23	373
9	H	1	1	500	30	10	6	94	19	146
10	Q	1	1	500	30	2	0	26	13	50
11	8	1	1	500	30	0	0	18	7	32
12	16	1	1	500	30	0	0	12	4	20
13	D	1	1	1000	30	192	61	198	37	587
14	F	1	1	1000	30	52	11	78	23	189
15	H	1	1	1000	30	11	2	18	9	52
16	Q	1	1	1000	30					
17	D	1	1	100	90	224	45	297	38	771
18	F	1	1	100	90	57	15	109	24	249
19	8	1	1	100	90	0	0	17	8	31
20	D	1	1	500	90	212	81	377	34	871
21	Q	1	1	500	90	2	0	24	12	48
22	F	1	1	500	45	49	20	174	25	310
23	F	2	1	500	45	46	15	113	15	222
24	F	5	1	500	45	44	18	67	12	168
25	F	10	1	500	45	43	17	43	6	127
26	F	1	2	500	45	48	15	164	19	279
27	F	1	5	500	45	41	15	159	16	256
28	F	1	10	500	45	40	14	158	10	251
29	F	2	2	500	45	40	16	95	12	186
30	F	5	5	500	45	38	16	49	6	118
31	F	10	10	500	45	39	15	42	4	110

Table A.4: Time performance measurements for the different algorithms and total execution.

Appendix B

User guide and installation

This project was done using Tekkotsu 2.4.1, meaning that it is verified to work with this version. It may or may not work with other versions but there are no guarantees for it. For the remainder of this installation and user guide it is assumed that the user already has Tekkotsu 2.4.1 installed and knows how to compile and install files on the memory-stick used by the AIBO robot.

B.1 Downloading and installing

B.1.1 Download and unpack

To install the solution along with the custom java monitoring utilities, first download the file:
`aibo_linedet.tar.gz`
It should be available at this location:
`http://ai.cs.lth.se/xj/PeterMorck/aibo_linedet.tar.gz`
Once downloaded, unpack it at a place of your choice using the command `tar`, like this:
`tar xvzf aibo_linedet.tar.gz`
That will produce the directory `aibo_linedet` in the same directory the command was executed in. Step into the `aibo_linedet`-directory with: `cd aibo_linedet`
The directory structure looks like this:

```
aibo_linedet/
                aibohough.tar.gz
                javamons/
                                cmdclient/
                                houghmon/
                                linemon/
```

The directory `javamons` contains the java monitoring utilities and the file `aibohough.tar.gz` contains the code for the AIBO.

B.1.2 Installing the AIBO files

The file `aibohough.tar.gz` is meant to be unpacked directly into the Tekkotsu project directory. To do this, follow these steps:

1. In the Tekkotsu root directory there is a directory named `project`. If you want, copy this to where you want to work with it, for example if you're standing in the Tekkotsu root directory, type:
   `cp -r project ~/aiboproj`
2. Go to the folder `aibo_linedet` that you unpacked earlier.
3. Copy the file `aibohough.tar.gz` into the project folder you copied, for example:
   `cp aibohough.tar.gz ~/aiboproj/project`

4. Go to the folder you just copied to, for example:
   `cd ~/aiboproj/project`
5. Unpack the file tar-file `aibohough.tar.gz` by using the following command:
   `tar xvzf aibohough.tar.gz`
6. To compile the project AIBO code and install it on the memory-stick, type: `make install`.

B.1.3 Installing the monitoring utilities

The java monitoring utilities need to be compiled before running. To do this, enter each of the three directories `cmdclient`, `houghmon` and `linemon` and in each directory run the command:
`javac *.java`. After this is done, they can be run from where they were compiled. The java version used during development was 1.5.0.

B.2 Activating the AIBO code

The code is contained in a module called `Exjobb` which needs to be manually started. To do this, start the `ControllerGUI` provided by Tekkotsu. It can be found in the `/tools/mon`-subdirectory of the Tekkotsu installation root. Locate the module `Exjobb` and double-click it to start it and double-click it again to stop it.

B.3 Custom monitoring utilities

These utilities require the `Exjobb` program module to be running on the AIBO since the server facilities are located in it. Note that both the houghmon requires the AIBO to send 3 images for each frame processed and is very heavy on the network. The CmdClient only sends a few characters as commands to the AIBO and the LineMon receives an array of line data in form of some numbers from the AIBO, hence they should both be very undemanding on the communication.

B.3.1 Using houghmon

The houghmon client is located in the `javamons/houghmon` sub-directory and has the following syntax:
`java Test <AIBO IP-address>`
When started, the AIBO will start streaming three images to the houghmon client in three separate windows. These are:

- Thresholded edge detection image.
- The grayscale image of the Hough accumulator with 1 pixel representing 1 bucket and the number of hits scaled between 0 and 255 where 0 is black and 255 is white.
- Y-channel of current camera layer with detected lines overlaid. The linedrawing takes place on the AIBO and the lines are drawn white and black to be visible on both white and black backgrounds.

Figure B.1 shows an example of what the houghmon will show when running. There is also a defunct test server (`TestServer`) used during development of the houghmon utility.

B.3.2 CmdClient

The CmdClient is located in the `javamons/cmdclient` sub-directory and is invoked by:
`java CmdClient <AIBO IP-address>`
Using the CmdClient one can change the camera resolution and also select whether thinning should be performed or not. The intervals of ϕ and r in the accumulator can also be changed. Furthermore, the accumulator peak threshold values for the Hough transform can be changed here. Note that only the last sent values are shown in the display, not the default starting values.

There is a test server available (`TestServer`) for the CmdClient as well but it serves no other purpose than just debugging the communication from the client.

Figure B.1: Example of houghmon running.

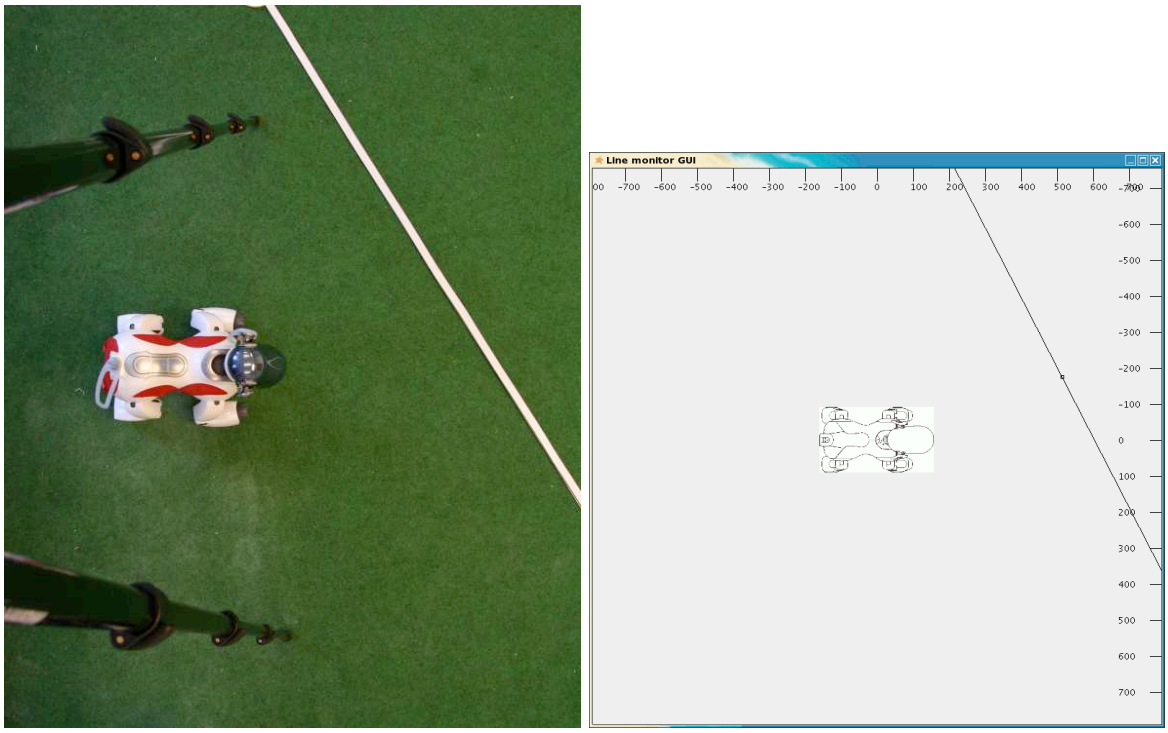

Figure B.2: Photo of AIBO in front of line and screenshot of LineMon util showing the same situation. Note: The black legs in the photo are from the camera tripod.

B.3.3 LineMon

The LineMon client is located in the `javamons/linemon` sub-directory and uses the following syntax:
`java LineMon <AIBO IP-address>`
When started the AIBO will start streaming line data which will be drawn onto a picture of the AIBO seen from above.

There is a record button available in a separate window which will capture line data in text-format and save it to a file, using the current date and time as filename; for example:
`20070416_131217.data`.
All data files are saved in the sub-directory `linemon/data`. There is also a test server (`TestServer`) available, which continuously generates a static pattern of lines sent to the client.

An example of the LineMon utility running can be seen in Figure B.2. It should be noted that the visible area of the LineMon is hardcoded in the program, but is easy to change. In the file `LineMon.java`, locate the line
`linegui = new LineMonGUI("Line monitor GUI",linedata,800,1600);`.
The two last arguments to the `LineMonGUI`-constructor are the sizes of the square area around the AIBO. The first one (800) is the number of pixels for the monitor, and the second one (1600) is the number of real millimeters the pixels correspond to. In effect, a scale is set by these two parameters.

Bibliography

[1] Nilsson, Nils J., *Shakey the Robot*, Technical Note 323, Institution: AI Center, SRI International, Apr 1984.

[2] W. G. Walter, *The Living Brain*, W.W. Norton, New York, 1953.
Also see *The Grey Walter Online Archive*, http://www.ias.uwe.ac.uk/Robots/gwonline/gwonline.html (verified November 6, 2007)

[3] *Sony Global - AIBO Global Link*, http://www.sony.net/Products/aibo/ (verified November 6, 2007)

[4] HyperMedia Image Processing Reference, http://www.cee.hw.ac.uk/hipr/html/hipr_top.html (verified November 6, 2007)

[5] S.M. Smith and J.M. Brady, *SUSAN - a new approach to low level image processing.* Int. Journal of Computer Vision, 23(1):45–78, May 1997.

[6] P.V.C. Hough, *Method and means for recognizing complex patterns.*, U. S. Patent 3,069,654, 1962

[7] K. LeBlanc, S. Johansson, J. Malec, H. Martínez, A. Saffiotti, *Team Sweden 2003*, pp. in proc. RoboCup 2003, Padua, Italy, 2003.

[8] *RoboCup Official Site*, http://www.robocup.org/ (verified November 6, 2007)

[9] P. Buschka, A. Saffiotti, and Z. Wasik, *Fuzzy Landmark-Based Localization for a Legged Robot.*, Int. Conf. on Intelligent Robotic Systems (IROS) Takamatsu, Japan, 2000.

[10] *AIBO SDE (Software Development Environment)*, http://openr.aibo.com (verified November 6, 2007)

[11] Ethan Tira-Thompson, *Tekkotsu: A Rapid Development Framework for Robotics*, master's thesis, Robotics Institute, Carnegie Mellon University, May 2004.

[12] *Tekkotsu Homepage*, http://www.tekkotsu.org (verified November 6, 2007)

[13] D.H. Ballard, *Generalizing the Hough Transform to Detect Arbitrary Shapes*, Pattern Recognition, Vol.13, No.2, p.111-122, 1981.

[14] Richard O. Duda, Peter E. Hard, *Use of the Hough transformation to detect lines in curves and in pictures*, Commun. ACM, Vol. 15, No. 1. (January 1972), pp. 11-15.

[15] Matthias Jüngel, *A Vision System for RoboCup: Diploma thesis*, master's thesis, Institut für Informatik, Humboldt-Universität zu Berlin, Feb 2004.