10p Examensarbete
Version 1.0

Matthias Grimrath
Email: m.grimrath@tu-bs.de

June 3, 1999

Abstract

This 10p examensarbete describes the efforts to measure the distance and angel of a robot to a wall by image analysis. The wall was marked with a black-and-white stripe pattern. Passive CCD camera optics were used to sample pictures of the pattern. The distance and angel from these pictures was extracted by methods developed in this examensarbete. Further it explains the porting of the Khepera robot communication libraries from the Microsoft$^{(R)}$ Windows$^{(R)}$ to the Unix$^{(R)}$ environment.

Contents

Chapter 1

Background

1.1 Introduction

This "examensarbete" (Swedish for thesis project) was developed during the spring semester 1999 (mid January - beg June) at the Department of Computing Science, Umeå University, Sweden.

The author was an exchange student from the Technical University of Braunschweig, Germany. He can be reached by email at <m.grimrath@tu-bs.de>.

This examensarbete was supervised by Thomas Hellström <thomash@cs.umu.se>.

If you have any questions or wish to get the full sources please write an email to the author. I am always happy about feedback! :-)

1.2 Overview

Here is a little abstract of what you will find in the following chapters.

- Chapter 2 describes the environment and tools that have been used in this examensarbete. The description of the setup is followed by the tasks that were subject of this work and some definitions.
- Chapter 3 describes the effort to port over existing software that runs under MS Windows to Unix/Solaris.
- Chapter 4 describes various filters and processes that helped improving the overall results of the image analyzing algorithms.
- Chapter 5 describes in detail the distance measuring, what problems showed up and some approaches to accomplish this goal.
- Chapter 6 describes in detail the angle measuring, various problems and approaches.
- Appendix A includes all of the important source code developed during this examensarbete.

Chapter 2

Setup

2.1 The Khepera robot

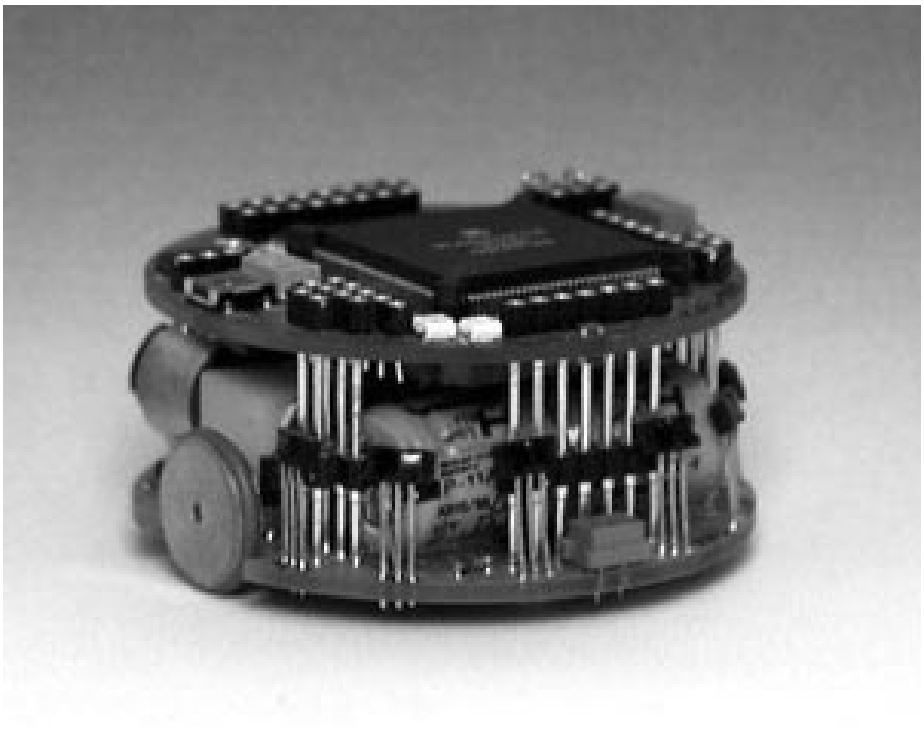

Figure 2.1: The Khepera Robot. Source: K-Team

Figure 2.1 and 2.2 shows the Khepera Robot used during this examensarbete. It is a small little robot with the ability to add extension modules to it. One extension available is a so-called "vision turret". The robot has other features and more extension are available, but they are left out here because they are not important for this examensarbete.

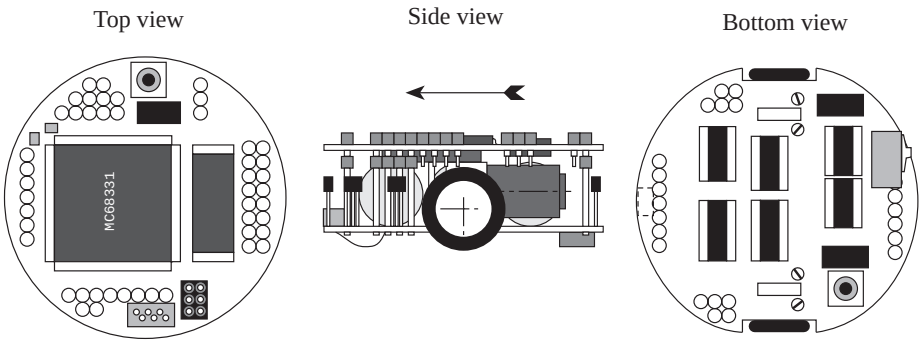

Figure 2.2: The Khepera Robot, schematic view. Source: K-Team

2.2 The vision turret

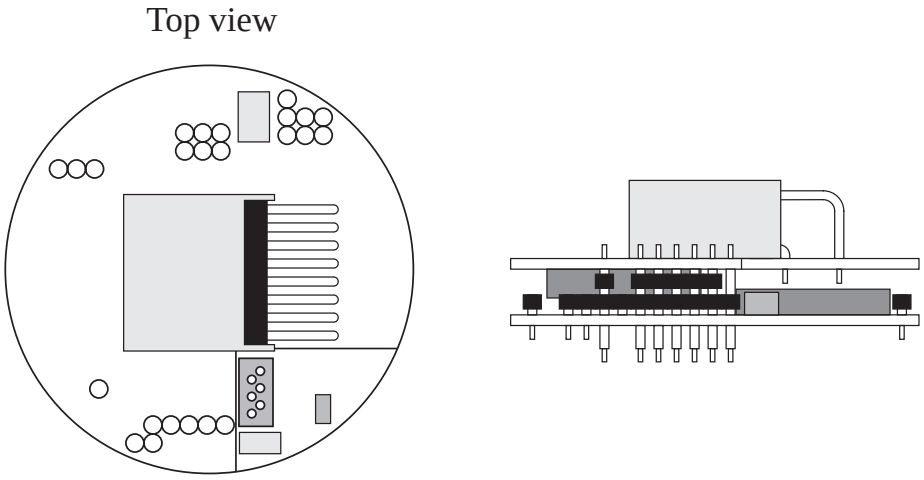

Figure 2.3: The Vision Turret, schematic view. Source: K-Team

The vision turret is an extension module with a camera on it. This camera has 64 pixel light sensors. Each sensor samples 256 different grey levels. It samples one-dimensional images along the field view, i.e. the field view gets rastered into 64 smaller field views. The whole field view has an angle of 36 degrees. The pixel sensor numbering starts from the left of the field view with number 0, i.e. pixel sensor 0 samples the light of the leftmost area of the whole field view.

There is also an ambient light sensor in this vision turret. It is used to adjust the sensitivity of the pixel sensors to the overall light level. This adaption to the ambient light is handled automatically by the vision turret's internal logic.

The manufacturer's[4] recommended operation's distance for a sharp picture range from 5cm to 50cm.

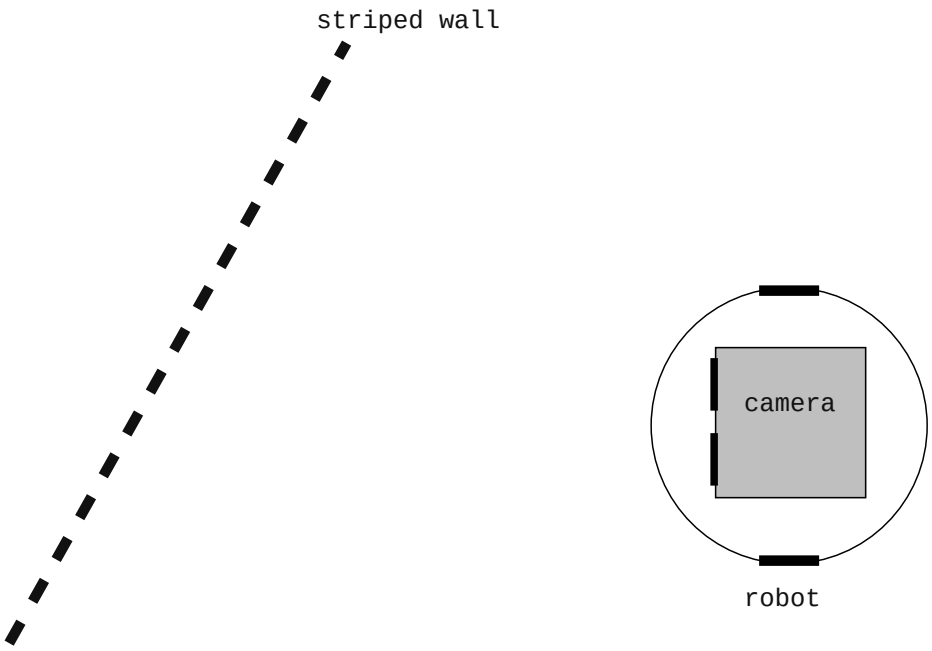

Figure 2.4: Sketch of the working environment used during this examensarbete

2.3 Configuration

Figure 2.4 shows a schematic illustration of the configuration/environment the robot was operated in.

The camera is looking at a "wall" with equally spaced black-and-white stripes printed on it, i.e. each black stripe has the same width than a white stripe. In theory this "wall" is unlimited, but in practice the robot was positioned in such a way the camera does not view beyond the stripes.

To have a simple start, it was made sure that no major disturbances occur, for example interruptions of the stripe pattern or non-uniform lumination of the scene. There also may only exist one "wall" so there is no need to take care of conditions looking at edges of two connected "walls".

Later in the development of the various analysis codes it was possible to tell at least under some but not all circumstances if the above constraints were not met.

2.4 Tasks for this examensarbete

When starting to work with the robot it was found that the number of existing Unix workstations in the Department of Computing Science outnumbered the amount of MS Windows PCs, the architecture the robot software already runs on. Furthermore the Windows PCs were frequently busy. Therefore it was decided to evaluate the possibility of porting the Khepera robot software to Unix as the first task.

The next decision was to make use of the vision turret. The one-dimensional,

grey scale optics of the camera (see also 2.2) led to the idea to evaluate the possibilities of display analysis of black and white stripes. After a little more investigating it was found it should be possible to measure both the distance to the stripes and the viewing angle to them.

To summarize, the tasks were:

- Implement robot communication API on Unix.
- Measuring the distance to a striped wall with the vision turret's camera.
- Measuring the viewing angle between the camera view line and the wall.

2.5 Definitions

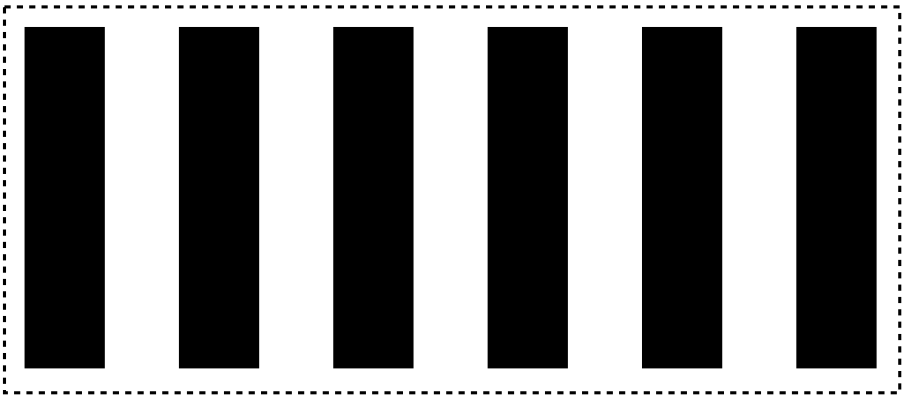

Figure 2.5: Equally spaced stripes used to measure the distance and angel

To make further explaining more comprehensive some definitions are introduced here. Since this examensarbete used a lot of signal processing methods its definitions are used here and adapted appropriately.

- The *Wall* is in practical terms a paper with equally spaced black-and-white stripes printed on it. See Figure 2.5 for an example. A more formal description is an unlimited line that has an alternating black and white pattern, where the black and white parts have the same width. Looking at it in signal-processing terms it is a periodic rectangular signal.
- The *Period* is the width of a white stripe plus the width of a black one. In signal-processing terms it is the period of the rectangular signal. Please note that the period here is measured in camera pixels when referring to sampled images.
- The *Width* is the width of one stripe. This is equal to half the period.
- The *Frequency* is the fraction how often a pair of black-and-white stripes is appears in a camera image. In signal-processing terms it means how often a periodic signal repeats in a fixed time. The equation is $period \cdot frequency = 64 pixels$ to describe how many stripes are sampled in a camera image.

At times it is important to distinguish between the *width* of the physically printed stripes (this is chosen at printing time and fixed) and the *width* of stripes (defined above as half the period) as it appears in the camera image. The first is measured as a length unit, while the second is measured in pixels.

2.6 FFT analysis

You will find many references to the "FFT" throughout this examensarbete. What follows here is a very brief description of the FFT for those readers that have not heart about it. If you would like to learn more about the FFT and get a better understanding of the theory behind and around it, read a book about signal processing, for example[5].

The "Fourier Transform" transforms a signal (for example a sinusoidal waveform) from the time domain into the frequency domain. The mathematical funding behind this transformation is that almost any signal can be decomposed into sinusoidal waves of different amplitudes, frequencies and phases. The sum ("overlapping") of all these waves results into the original signal.

The general formula for each sine wave is $y = a\sin(bx + c)$. a is called the amplitude, b is called the frequency and c is the phase.

The Fourier Transformation returns for a given signal in the time domain for every frequency the sine wave's amplitude and phase.

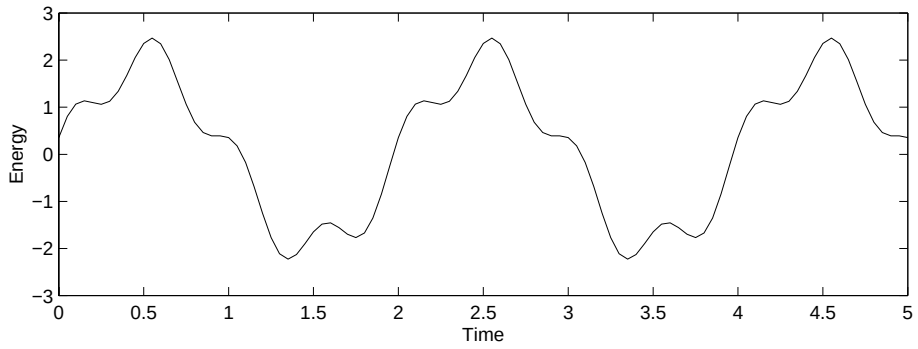

Figure 2.6: An example signal given in the time domain.

Figures 2.6, 2.7 and 2.8 give an example. The function to create this example signal is

$$y = 0.5\sin(2 \cdot 2\pi x + 0.25\pi) + 2\sin(0.5 \cdot 2\pi x + 0\pi)$$

"FFT" is an abbreviation for "Fast Fourier Transformation". It is an algorithm to transform a signal from the time into the frequency domain. The above example figures were produced with the FFT. There you can also see that the FFT mirrors the frequencies. This is a property of the FFT algorithm, and a "real" Fourier Transform would not show this behavior.

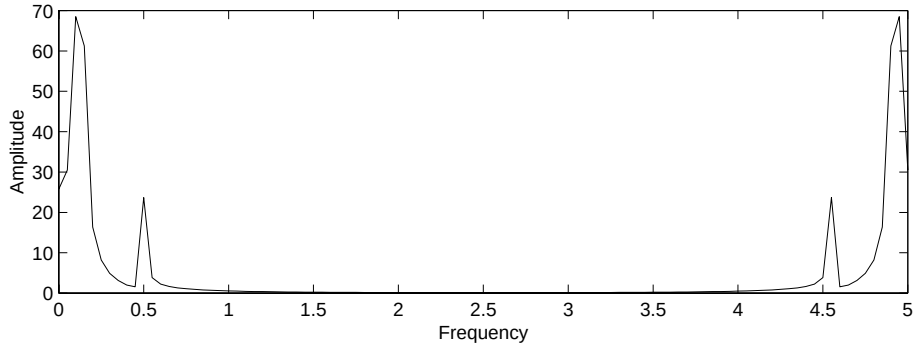

Figure 2.7: The signal from figure 2.6 decomposed into the frequency domain; amplitudes shown here

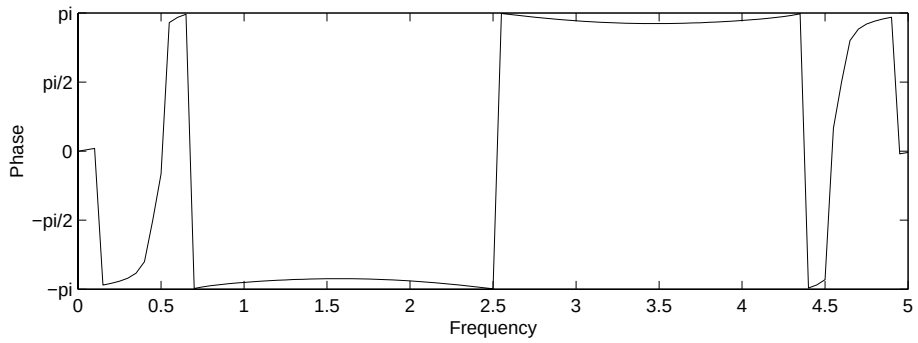

Figure 2.8: The signal from figure 2.6 decomposed into the frequency domain; phases shown here

Chapter 3

Porting to Unix/Solaris

Up to now the supplied MATLAB environment from the manufacturer to communicate to the Khepera robot only runs on MS Windows operating systems. Further investigation revealed that the porting effort was limited to operating system specific differences on accessing the serial port.

In-house developed software already written on Window PCs in MATLAB runned smoothly on the MATLAB version for Unix machines except for some minor glitches regarding case-sensitivity. These could be fixed very easily.

On the hardware side, the Khepera peripherals connect to a standard RS232 serial port. The Unix/Solaris workstation this examensarbete was developed on features such a serial port, thus no special adapter was needed to connect the robot to the workstation.

3.1 The Khepera Communication API

Examining the source and documentation of the supplied MATLAB environment shows that the communication to and from the robot goes through the following functions: `kopen`, `kclose` and `ksend`[1].

The task of porting the MATLAB environment could now be clearly defined:

1. Find out what the above mentioned functions are doing
2. Find out how these functions can be implemented in the Unix/Solaris version of MATLAB.
3. Find out how to access the serial port on Unix/Solaris and how to configure it to the correct baud rate, number of data bits, etc...

The first point was quite simple. The purpose and behavior of these three functions could be determined easily from the documentation from the manufacturer of the robot [4] and the already existing software making use of them.

The second point required more work. After some searching in the MATLAB online manuals it turns out that MATLAB provides a way to call binary code from within your own MATLAB programs. This makes it possible to write your

[1] In the Windows version of the communication software there also exists a function called `ksends`. Since this function only differs in implementation from `ksend` and is not used in in-house software its porting was omitted

MATLAB functions for example in C, compile it, and then make use of it in your MATLAB software. Having a way to incorporate programs written in C, the traditional Unix programming language, it is possible to use a lot of features Unix offers and at the same time providing them to the MATLAB software. This was necessary for the third point, accessing the serial port.

3.2 Interfacing to MATLAB

Writing a MATLAB function in C is not like writing a normal program in Unix, since the C code must run under control of MATLAB in order to access variables, return values and the like. For this special purpose MATLAB comes with a special compiling front-end called "mex" which takes care of linking special libraries against the code and producing a binary format understood by MATLAB. The C source itself needs to include special MATLAB library header files and must use special MATLAB functions to not interfere with MATLAB itself. For example, to allocate memory one must avoid the use of `malloc` and instead call a special MATLAB library function. Fortunately, with little careful programming, there are no real limitations for operations on serial ports, the interesting part in this context. For further information on how to include C code in MATLAB read [2], chapter "Creating C Language MEX files".

3.3 Interfacing to Unix

Serial ports in Unix system are represented as so-called device special files. If an application wants to send or receive data over the serial port, it has to open these special files. This means that the same system calls are used for I/O on a serial port as for a standard file.

Since a serial port has some properties that distinguish it from standard files, there exist special system functions to modify aspects of the serial communication such as the baud-rate. The terminology used for these functions is a little confusing and revealed that the serial port on Unix system is mainly used to hook terminals to it. You find the setting of the number of stop bits, for example, is specified along with that of I/O properties for terminals.

The function definitions for serial communication were taken from [1], topic "Low-Level Terminal Interface".

3.4 Portability and Compatibility

A lot of versions of Unix are around, and though the function definitions to alter the serial communication is covered by the POSIX standard, minor differences between different flavors exist. In order to make porting easier for other Unices (in particular Linux) the freely available software package minicom[3] was examined. This software runs on many platforms and thus served as a good source of information about incompatibility. Fortunately, for this special task of making the robot available to MATLAB, no special considerations regarding incompatibility need to be taken care of as long as the system is POSIX compliant.

Source code

The source code for `kopen` is on page 34, `ksend` on page 37 and `kclose` is on page 39.

Chapter 4

Camera Input Preprocessing

As was discovered during this examensarbete, the camera of the vision turret produces far from ideal image samples. Applying preprocessing to the image data of the vision turret's camera before it was further used notably bettered image quality and thus improved the output of the algorithms that extract distance and angel from the camera images.

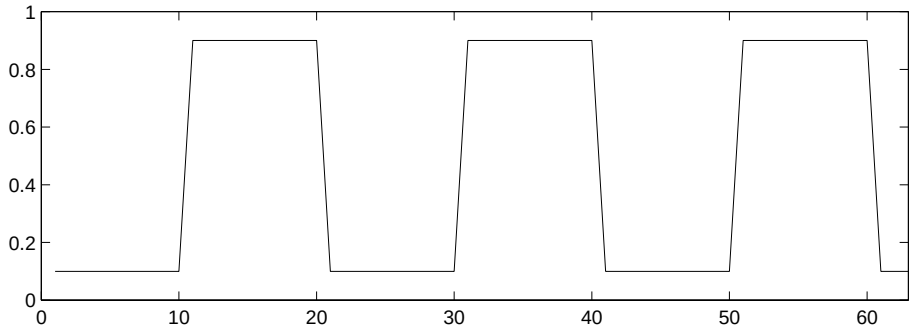

Figure 4.1: A theoretical, ideal image sample.

Figure 4.1 shows how the image data would like if the camera is an ideal device.

For a quick demonstration, figure 4.4 is a real image sample, far from being perfect. The following sections describe in detail what problems were discovered and how they were dealed with.

4.1 Noise

In figure 4.2 the camera was looking at a white piece of paper. Ideally every camera pixel would have sampled the same value somewhere in the bright. However, there are some minor deviations from the ideal line. Since the noise is not that strong it was a rather small problem compared to the others.

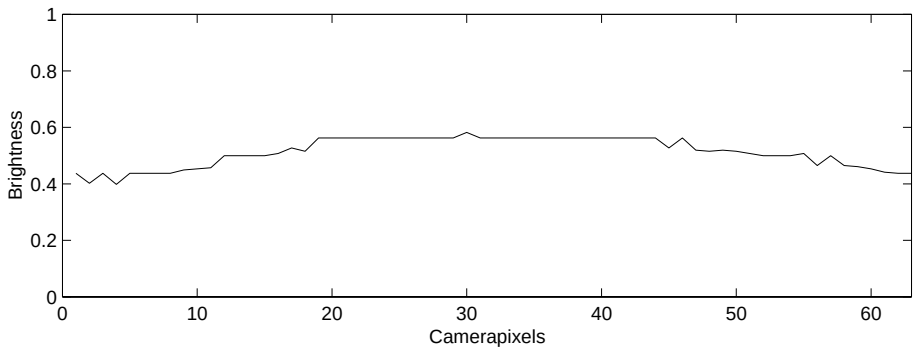

Figure 4.2: An image sample looking at a white paper. Pixel values have been "normalized" to [0 1] with increasing brightness towards 1. The quality suffers from a little brightness weakness in the corners. You can also see the noise while sampling the image.

4.2 Brightness weakness

Figure 4.2 also shows that the supposed to be line looks dragged down at both sides. Figure 4.3 is based on the same image data but visualized differently. There one can clearly see that the camera samples darker values at the edges. It may not look dramatically here, but this effect increases if the camera is looking at stripes. For an example, take a look at figure 4.5. The reason for this weakness was not explored further.

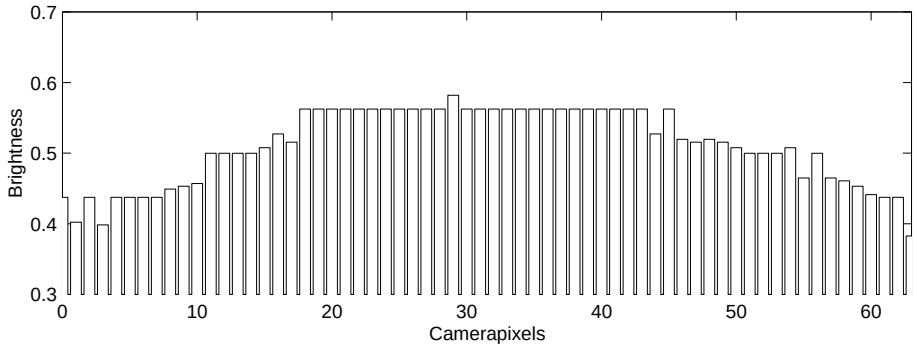

Figure 4.3: This figure is based on the same image date as those in figure 4.2. Here the image is printed as a bar graph and zoomed to better show the brightness weakness.

4.3 Low contrast

Figure 4.4 shows another weakness of the vision turret's camera. It doesn't have a sharp view on the stripes. Since the stripes are printed black on white, with no grey levels in between, the image values should "jump" up and down with no intermediate ones. However this is not the case. Especially when operating the

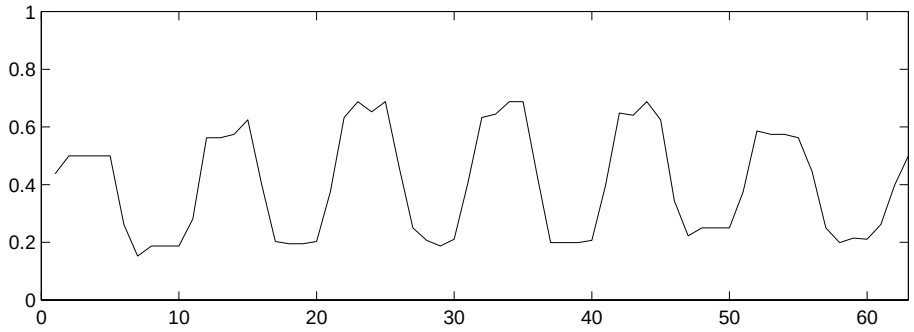

Figure 4.4: A real image sample from the vision turret's camera. The camera is looking perpendicularly at stripes of 1cm width from a distance of 20cm. The camera hasn't an ideal sharp view on the black and white stripes but some grey levels in between.

camera at very close distances, below the distance the manufacturer recommends (see also section 2.2), the stripes become smooth rounded hills. This is probably due to the fixed lens of the camera, therefore it cannot adapt to various distances like for example the human eye.

An interesting possibility would be to compensate for this systematic error. If the distance is somewhat know (this could be done as described in chapter 5 and 6) it might be possible to recalculate the image data, compensating the insharpness to a certain extent. However, this approach was not examined further in this examensarbete.

4.4 Gamma correction

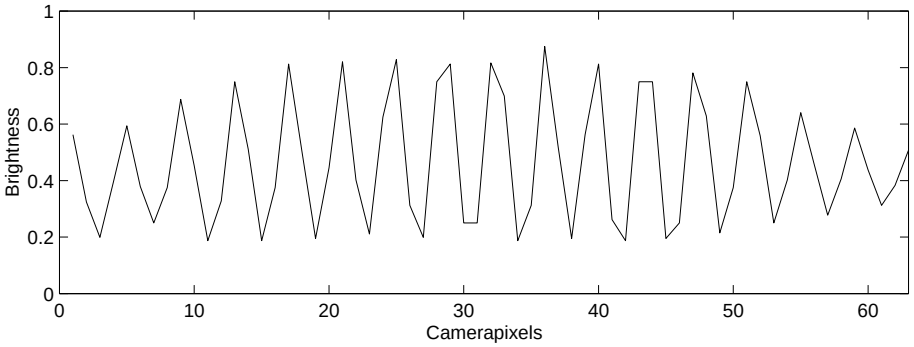

Figure 4.5: Another snapshot of stripes. This time so far away it comes close the cameras resolution. The brightness weakness is even more apparent here than in the previous figures.

In order to compensate for the brightness weakness, it was decided to gamma correct the image, i.e. increase low image values in the corners of the picture. The problem was to find a good correction method.

4.4.1 How to correct

Looking at the type of gamma error (for example figure 4.3) the error becomes stronger towards the corners while the middle of the image does not need gamma correction.

Also, gamma correction needs to be different for the black or white camera pixels. Figure 4.5 shows a stronger error on the white pixels than on the black. Therefore it is necessary to separate the white pixels from the black, and apply a different gamma correction to each.

4.4.2 Type of correction function

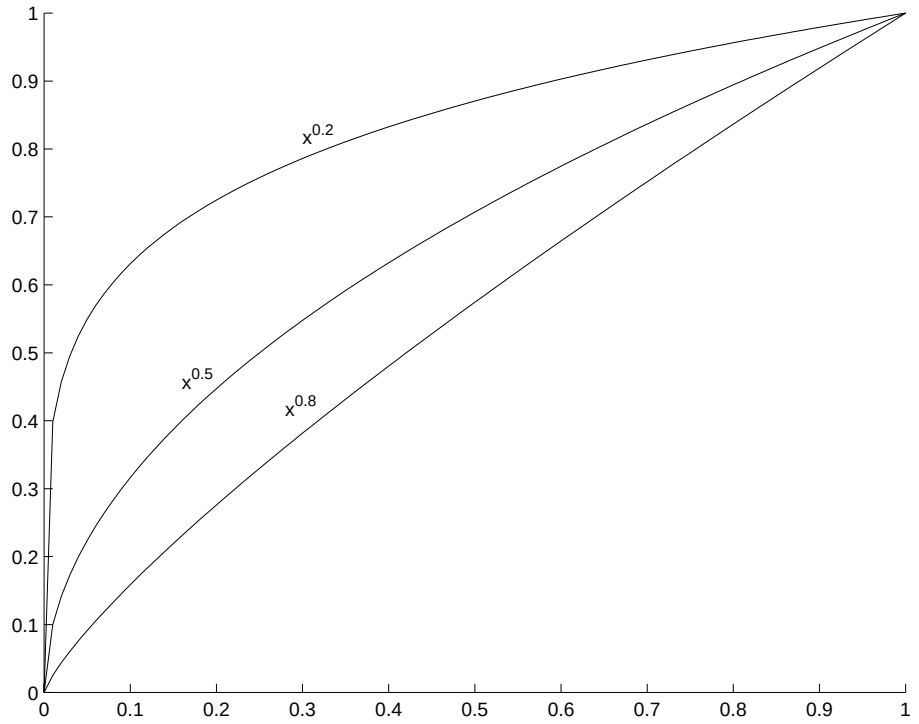

Figure 4.6: Some root functions used as an example to demonstrate the use of this kind of functions for gamma correction

The basic correction function chosen was the general root function, $y = x^a, 0 > a > 1$. To demonstrate how these functions work take a look at figure 4.6. These functions only produce good results with respect to gamma correction in the range from 0 to 1. Since our camera's pixel values have a limited range also it is easy to scale the pixel's value to the interval $[0; 1]$.

If a low pixel value is given as an input to the root functions, the resulting value will be relatively higher as if a high pixel value is given to a root function. For example, $x = 0.1$ for $y = x^{0.5}$ results in 0.31, whereas $x = 0.9$ gives 0.94. The first value is scaled by a factor of 3, while the second one hardly changes.

The effect of these functions is that values towards 0 are stretched more than

values towards 1, i.e. "dark" pixels are getting brighter while "bright" pixels almost stay the same.

4.4.3 Amount of correction

As described earlier, the brightness error get worse towards the corners. Therefore it was decided to apply a different gamma correction function for each pixel. The result is that the parameter a in the function $y = x^a$ varies from pixel to pixel. Based upon the look on how the error develops towards the corner and to have a simple function a parable aka polynomial of grade 2 ($y = c_2x^2 + c_1x + c_0$) was chosen, where $1/y$ is placed into the a parameter.

(Note: $1/y$ reflects how this gamma correction was developed. First it was thought of as the nth root, and this viewpoint was quite convenient to develop the parameters for the parable. Root and exponents can equally be converted into each other by the multiplicative inverse.)

This polynomial is shifted so that the y value of the apex has a value of 1 for the x value of 32, i.e. in the middle of the camera image which consists of 64 pixels.

Additionally, it should be controllable how much correction is applied in the corners of the camera image.

The general formula for this special purpose is

$$y = \frac{s}{32^2}(x - 32)^2 + 1, \qquad x = 0, 1, \ldots, 63$$

where s gives the maximal amount of correction +1, i.e. in the corners pixel values are corrected with the sth root. For example, if $s = 3$, $x = 1$ results in $y = 4$, thus the processed pixel value is the 4th root.

The outcome of these intertwined functions is that no gamma correction is done in the middle of the camera image ($y = 1$ for pixel 32) with increasing correction towards the corners. The highest correction is applied to the corner pixels, i.e. pixel number 1 and 64. The maximum correction is given by the above mentioned s.

4.4.4 Separating black from white pixels

As mentioned in 4.4.1, the brightness error is different for dark image values than for bright ones. By experimenting the following functions were found to be useful:

$$y = \frac{4}{32^2}(x - 32)^2 + 1, \qquad x = 0, 1, \ldots, 63$$

for the white pixels and

$$y = \frac{0.5}{32^2}(x - 32)^2 + 1, \qquad x = 0, 1, \ldots, 63$$

for the black ones.

"Separating" introduces another problem: Which pixel values should be treated as white, which as black, and which as nothing? For this examensarbete a rather simple method was used. This method and its associated parameters were - again - found by trial-and-error.

The pixel values must have already been scaled to $[0; 1]$. In brief, the algorithm is as follows:

1. A value is calculated that separates "black" from "white" pixels. This value is the average of all camera pixels.

2. Every pixel value that is too close to the separation value is regarded as neither white nor black and eliminated.

3. The remaining pixels are gamma corrected.

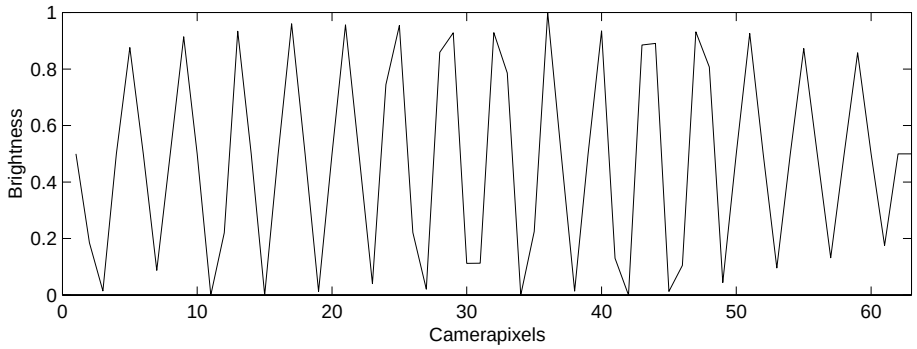

Figure 4.7: Based on the same input data than the figure 4.5, this picture is gamma-corrected to compensate for the brightness weakness

Some other (primitive) checks are done to avoid dividing by zero and gamma correcting images that are not images of striped walls.

Figure 4.7 shows the results of the above mentioned gamma correction.

To help understanding this gamma correction method, the important MATLAB source code is printed below.

'corrw' and 'corrb' are the gamma correction function tables for black and white pixel respectively.

```
corrw = ones(1,64)./((4/(32*32))*([0:63]-32).^2+1);
corrb = ones(1,64)./((0.5/(32*32))*([0:63]-32).^2+1);
```

'vis' is the camera image consisting of 64 pixels, already scaled to $[0; 1]$.

```
function [ret] = gamma_correct(vis,corrw,corrb)
  avg = mean(vis);
  vis = vis - avg;
  v1 = (vis > 0.1).*vis;
  v2 = (vis <= -0.1).*vis;

  if max(v1)>=0.1
    v1 = (v1./max(v1)).^corrw;
  end
  if min(v2)<=-0.1
    v2 = (v2./min(v2)).^corrb;
  end

  ret = v1 - v2 + max(v2);
```

```
if max(ret)~=0
  ret = ret./max(ret);
end
```

4.5 Low pass filtering

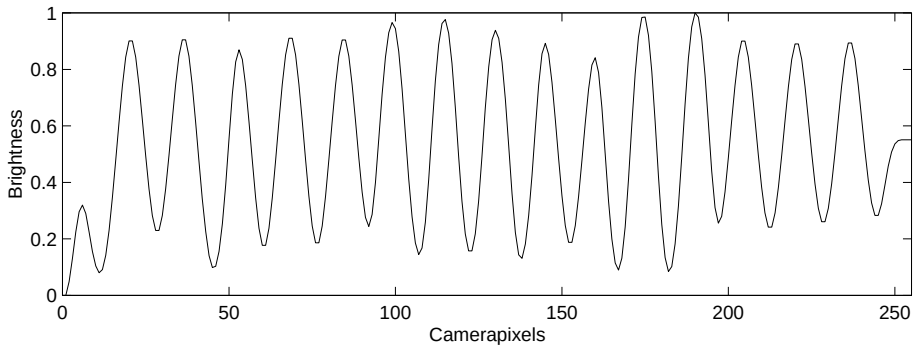

Figure 4.8: The next step after gamma-correcting the picture (see figure 4.7) is to eliminate noise. This is done by applying a low-pass filter to the data, "smoothing" the signal. Please note the changed x-axis labeling: The original input data was upsampled to better approximate the low frequencies of the filtered image.

Another less crucial step is to apply a low pass filter to the camera image. As mentioned earlier this step removes noise that typically appears in the higher frequencies. Since there is only one frequency that is interesting - the one that results from the printed stripes - and this frequency is expected to be low, higher frequencies can be safely cut off.

Low pass filtering is not really necessary, since a Fourier transform is used by the distance and angle measuring methods to transform the signal into the frequency domain. Once in the frequency representation, a similar high frequency cut can be achieved by ignoring higher frequencies.

Low pass filtering seemed to improve the results when measuring the angle (see chapter 6). The method used there requires a clear peak to produce good results. This seeming improvement was not explored further during this examensarbete however.

The MATLAB code that filters the input data is rather simple. `Filter` is a MATLAB provided function that performs filtering with the specified filter on digitized input data.

```
filtered_image = filter(hanning(8),1,image);
```

4.6 Preprocessed vs raw

Figure 4.9 and 4.10 shows the difference between the raw camera image (figure 4.5) and the gamma corrected and filtered image (figure 4.8). While the difference doesn't seem to be much, it is enough to extend the number of camera

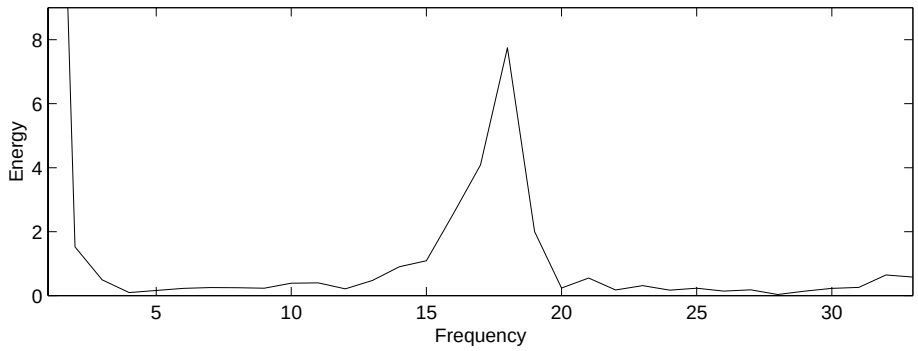

Figure 4.9: The FFT analysis of the data in figure 4.5.

images that can be analyzed. Especially when the images are less cleaner than the often referenced image in figure 4.5 the effort is worth it.

These graphs (4.9 and 4.10) demonstrate the effect of the gamma correction and low pass filtering. It is the FFT analysis of either the raw and the preprocessed image of figure 4.5. Both have their peak at the same index (17), and the peak of the filtered image is a little bit cleaner.

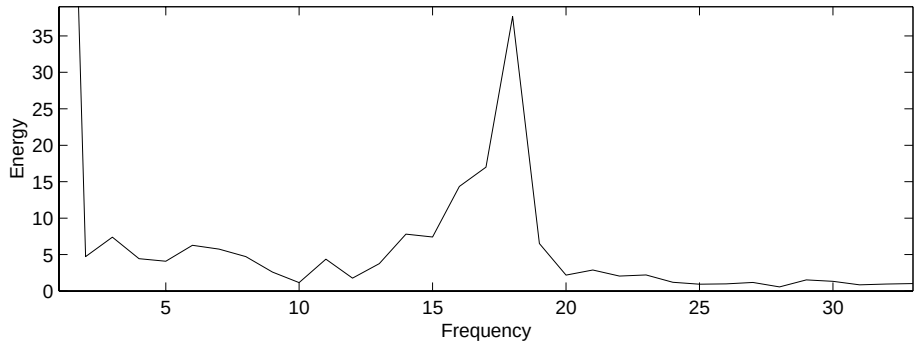

Figure 4.10: FFT analysis of the preprocessed image. Only the interesting part i.e. the first 32 frequencies of the analysis is printed here.

Chapter 5

Measuring Distance by Image Analysis

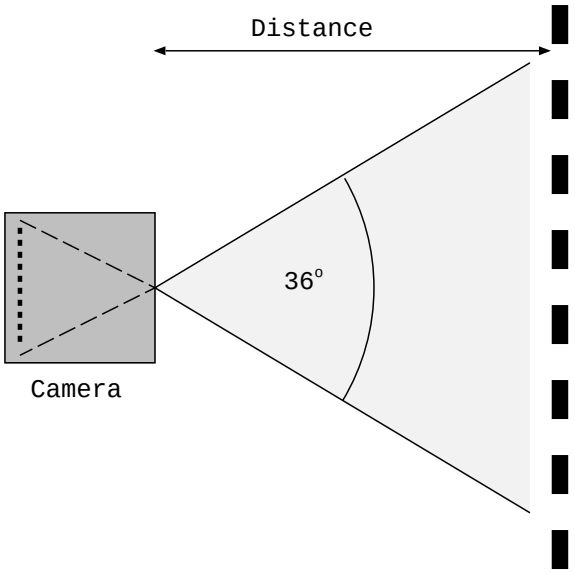

Figure 5.1: The setup for measuring the distance. The camera is looking perpendicular at a striped wall.

For the second task, measuring the distance, the setup was as shown in figure 5.1. The camera has to look perpendicular at the wall. This made it easier to get a working algorithm. In chapter 6 a different method was developed that can measure the distance even if the camera is not looking perpendicular. The method used there evolved out of the earlier written routines described in this chapter.

5.1 Methods used

The first and basic thing to do was to find out how to use the sampled image to calculate the distance. Two basic methods have been found, but other superior methods may exists nonetheless.

1. Measure the number of continuous pixels in the sample that supposedly belong to a stripe. From that measured "width" and the known printed "width" the distance may be calculated.

2. Count the total number of stripes, then multiply this number with the known printed width of each stripe. This gives the length of the area seen by the camera.

The above methods can also be explained by the equation given in definition section 2.5, $period \cdot frequency = 64 pixels$. The first method measures the period of the stripes as sampled in the camera image. From the period the frequency may be calculated. On the other hand, if the frequency is known (the second mentioned method), the period may be calculated.

Both approaches have their pros and cons. The first method has problems with the images not having a sharp contrast (see figure 4.4) and with noise. Also the accuracy drops with increasing distance, but on the other hand is quite accurate on short distances or big stripes respectively.

It should be noted that by applying intelligent filtering beforehand it should be possible to get good results with the first mentioned method too. It was not really explored further, due to the fact that the second approach gave surprisingly good results. That little that was done is described in 5.5. Nonetheless, further research may come up with equal results for the first method.

5.2 Distance calculation

Given the number of periods (which is to know the frequency), the length of the visible wall is *frequency*$\cdot 2 \cdot$*width-of-printed-stripe*. Using the known viewing angle of the camera, the formula to calculate the distance from the length of the visible area is

$$d = \frac{length\,of\,visible\,wall/2}{\tan 18^o}$$

See figure 5.2 and section 2.2 for reference.

5.3 Determining the frequency

As can be seen in the previous section, the relationship between the length of the visible area and the distance is quite straightforward. The tricky part is to find out the actual length of the visible area. This is supposed to be done through the camera image.

The width of the stripes is known as it is chosen when the stripes are printed on the wall. The first attempt was to count the number of periods, i.e. find the sampled image stripe frequency.

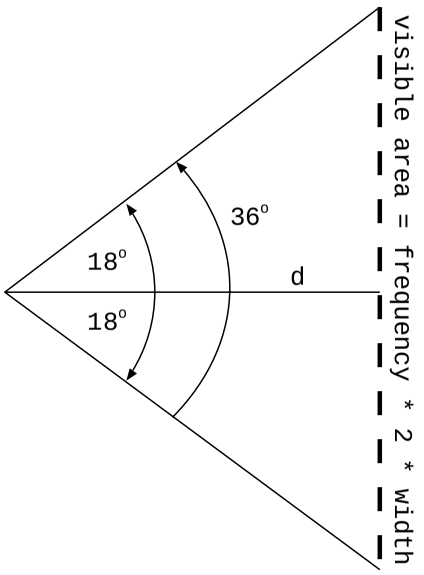

Figure 5.2: Basic geometry to calculate the distance from the length of the visible area.

Before the relationship between periods and frequency has been worked out in this examensarbete, only by pure looking at the camera image, it was wondered how to find out the numbers of periods in a camera image.

So in the beginning of the work for this examensarbete, while looking at a typical camera image (Figure 5.3), the similarities with a low frequency harmonic wave overlapped with other signals struck. This led to the idea of using the Fourier analysis (See section 2.6 for a short explanation of the FFT). Thus a Fourier analysis of such a camera image sample should give a peak somewhere in the low frequencies.

As expected experimenting showed that the assumption was correct. As an example, the Fourier analysis (using the FFT algorithm) of the data in figure 5.3 is shown in figure 5.4.

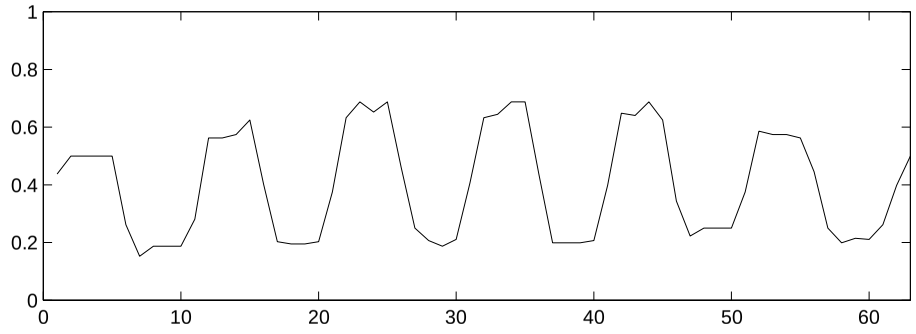

Figure 5.3: An example of a real image. The human eye clearly recognizes the 6 stripes, but not the computer without further analysis.

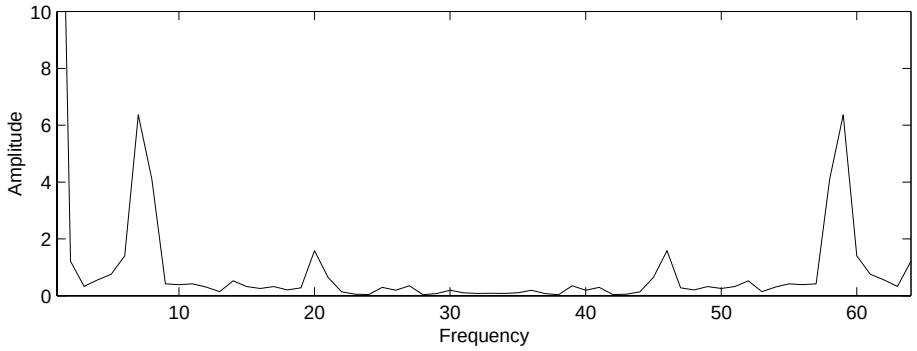

Figure 5.4: The FFT analysis of the data in figure 5.3 (absolute values). The "strongest" frequency caused by the stripes gives a clear peak. The mirroring of the frequency spectrum is caused by the properties of the FFT algorithm. Please note that the index of the highest peak (7) is one more than the number of seen stripes.

There is a clear peak at index 7. The actual number of stripes is one less. Sampling a camera image at various distances revealed that the index with the highest peak is always one more than the number of periods. This is for sure no coincidence, but further investigation into the FFT algorithm and the particular MATLAB implementation to validate this assumption was not done.

So the first and simple approach to get the number of periods was to Fourier transform the camera image, look for the maximum peak in the range of reasonable frequencies, and use that index as the number of periods. Once the frequency is known, the distance can be calculated easily as shown in the previous section.

5.4 Advantages, caveats and shortcomings of the FFT method

On the positive side, the following things were discovered:

- Transforming the camera image into the frequency/energy domain makes it quite robust against noise that is often high frequencies overlapping lower ones.
- The FFT "looks" at the whole image, not only a part of it, so sampling differences between the various stripes are averaged out.
- If the camera is not looking perpendicular at the wall, or not even looking at a wall at all, the peak in the FFT display disappears. Together with further processing on the FFT data this makes it possible to tell whether the artificial environment of a stripe wall is present. This does not work in all cases, but filters out a fair amount of invalid pictures.

One problem is that it only works for a limited number of periods. During experimenting it was found that the accuracy of the calculated distance

decreases when the camera stood closer to the wall. This is clear when one imagines that it takes a longer way to move until the number of periods change in the display if the camera is standing close to the wall. If it is further away it takes a shorter way until a new stripe moves into the visible area.

Likewise, the accuracy increases if more periods were in the camera image, but then the FFT data gets less accurate, because the peak moves more and more into the higher frequencies making it less distinguishable from noise. For practical reasons the number of periods regarded as resulting from a valid image ranged from 3 to 20.

Unfortunately the FFT algorithm returns discrete frequencies. Especially for the lower frequencies it would be nice if the "frequency resolution" would be finer. As a result from this limitation, when the number of visible periods are between two discrete values, the FFT data does not show a clear peak for one frequencies, but more like a hill ranging over 2 frequencies. To compensate for an image on a limit some extra code was written to discover such hills and take the average frequency, assuming the real frequency is between the two discrete values reported by the FFT analysis.

Source code

The MATLAB source code for the above described method can be found on page 43.

5.5 An alternative - Measuring the width

As mentioned earlier, this method was not really explored to its best.

5.6 Algorithm in brief

The main problem in developing a good algorithm was the insharp camera image and thus to make a good bet which pixel values belong to a white stripe and which to a black one.

Since the camera samples best in the middle, it was decided to start looking for a stripe in the middle and from there move away to the left and to right until the probable end of the stripe.

A careful reader may observe in the source code that there is also a procedure included that counts the total number of stripes and their width. In an early version instead of focusing on the stripe in the middle of the camera image all stripes in the image were measured and the average of all visible stripes was taken.

This approach was dropped because it provided hardly better results than only examining the stripe in the middle and tends to become even less accurate when not looking perpendicular to the wall. But the procedure that counts the total number of stripes is useful to make it more robust against bad camera images.

The first step was to separate pixels in black and white ones. All pixel values are "normalized" to the interval $[0; 1]$ first, 0 is absolute black and 1 brightest possible white. Then the separation was done using a hard coded threshold.

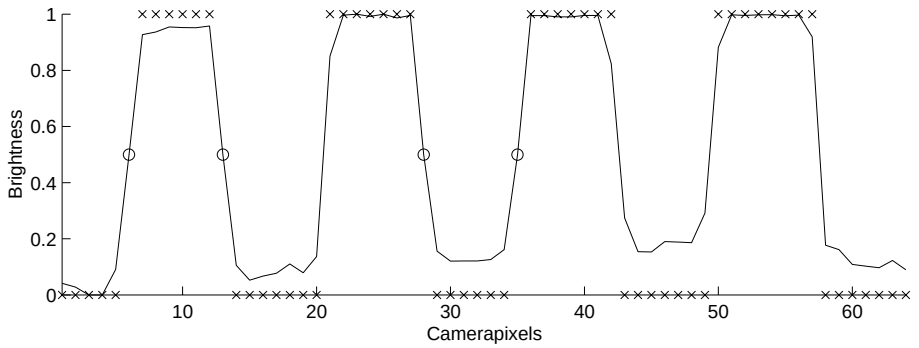

Figure 5.5: This figure visualizes the algorithm for direct measurement of the width of the stripes. '×' mark pixels that have been identified as white or black and '∘' that could not be classified. Please note that this image has been gamma corrected before.

Anything greater than 0.6 is regarded as white and anything lower 0.4 as black. Of course this is a very simple approach and it tends to count too few pixels that correctly would belong to a stripe.

For an example of how this algorithm analyses an image take a look at figure 5.5.

5.7 Results

Though the above mentioned algorithm is rather simple its accuracy was better than those of the FFT method if only a few stripes were visible. On the other hand, if the number of visible stripes increases the FFT method produced better results.

So it seems best to combine these two methods: FFT for long ranges and width measurement for close ones.

But the width measuring has some practical problems. It is less robust than the FFT method when fed with camera images that do not show the stripe pattern. With the FFT method it was easier to recognize bogus images. Also is the width measuring algorithm more complex than the FFT one.

Careful programming and further research may develop a more robust width measurement than the simple approach developed in this examensarbete.

It should ne noted that neither the width measurement nor the FFT method developed in this examensarbete are able to identify all bogus camera images.

Source code

The source code of the width measurement method described above is in the Appendix on page 40.

Chapter 6

Measuring Distance and Angle by Image Analysis

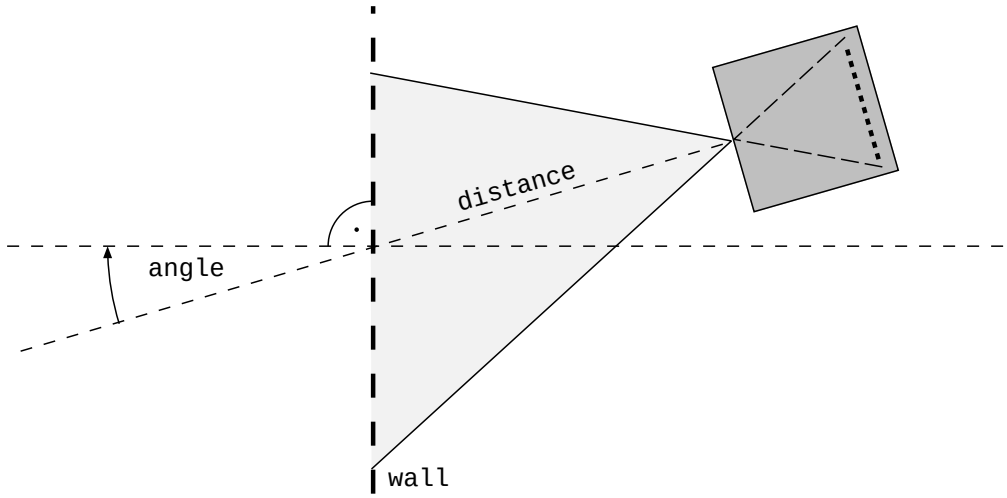

Figure 6.1: Sketch to visualize the camera's angel towards the wall

So far the setup was that the robot's camera is looking perpendicular at a striped wall. This made distance calculation easier, because perspective distortion of the stripes need not to be taken care of.

The next task was to examine if it is possible to measure the distance if the camera is not looking perpendicular at the wall. Furthermore it was investigated if it is possible to measure the angel of the camera position.

6.1 Setup

Figure 6.1 shows a sketch of the robot setup. The *distance* is the shortest way from the camera's lens to wall. The *angle* is the angle between the distance line

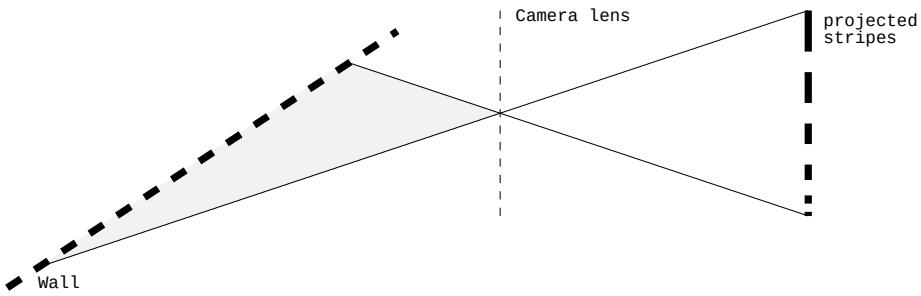

Figure 6.2: Sketch to demonstrate the perspective distortion.

and the line that is perpendicular to the wall. The rest of the setup is same as described in chapter 5.

6.2 Methods used

As in the chapter describing how to measure the distance, again two methods were found that could be used to measure the angle and distance.

1. Measure the number of continuous pixels in the sample that supposedly belong to a stripe. From that measured "width" and the known real "width" the distance and angle may be calculated.

2. Use again FFT analysis. How this is done in particular is described below.

Since the camera is not looking perpendicular anymore the image suffers from a perspective distortion. This distortion however corresponds to the angle. In particular, the width of the sampled stripes increases/decreases linearly, where higher increases/decreases indicate a steeper angle. So the predictable distortion of the stripes may be used to measure the angle. See also figure 6.2 to see how the stripes are projected into the camera.

6.3 Measurement of width

Despite being possible direct measuring was not explored further, again because the FFT analysis gave better practical results.

6.4 FFT analysis

The standard FFT analysis which looks at the whole image cannot be used unless the camera is looking perpendicular. If the camera looks with increasing angle towards the wall, more and more the peak in the FFT analysis becomes a flater and broader hill.

If the camera is not looking perpendicular the sampled width of the stripes changes constantly over the image (For an example take a look at figure 6.3). It can be seen as if the frequency of the stripes increases/decreases constantly. This explains why the FFT analysis does not show a clear peak, because there is more than one frequency in the image caused by the stripe pattern.

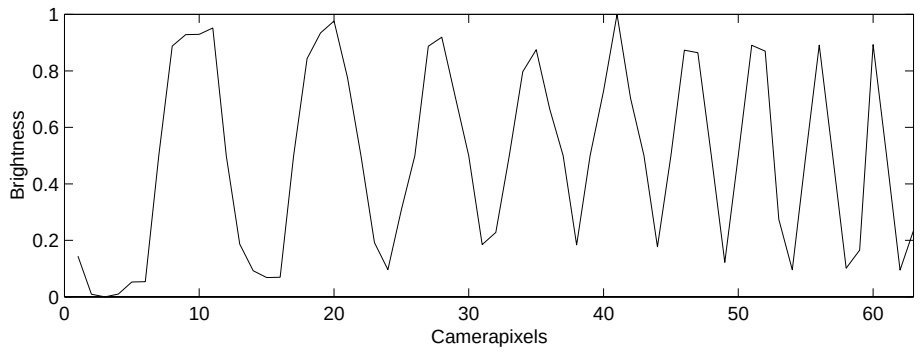

Figure 6.3: An example image (already gamma corrected) of the vision turret's camera looking inperpendicular to the wall. The width of the stripes increases from left to right indicating the camera is standing left from the wall.

6.5 Handling the changing width

A necessary precondition for analysing such image further with the method described below is that the width of the stripes changes linearly and monotonically over the image. This condition is met if one is looking at the projection geometry involved. See figure 6.2 for reference.

Based on this precondition the idea is not to FFT the whole image, but only a part of it, for example pixels 1 - 32 and pixels 33 - 64. This narrows the analysis window, thus making the increase of width in the stripes less and so reducing the frequency mix as described in 6.4. Since we know that the width of stripes in the image changes constantly, narrowing the analysis window leaves out the other stripe frequencies, thus the FFT analysis results in a cleaner peak. The downside is that the original image size (64 pixels) is reduced, making the measurement less accurate.

We know that the stripe width changes constantly. If we analyze the "left" (pixels 1 - 32) of the image and the "right" (pixels 33 - 64) and the frequency peak for either is different, the camera is looking sideways towards the wall. By comparing which side has the greater frequency it is possible to tell whether the camera is standing left- or rightwards to the wall.

See figures 6.4 and 6.5 for an example.

6.6 Continuous FFT analysis

The next step was to extend this method. Instead of analysing only the left and right part, a window of 32 pixels was moved continuously over the 64 pixel wide image, starting from the left. This means that first pixels 1 - 32 are Fourier transformed, then pixels 2 - 33, 3 - 34, ... until pixels 33 - 64. After each analysis the index of the highest frequency peak is stored.

Since the camera is looking at a straight wall printing these indices in a graph results in a line. Figure 6.6 shows this kind of graph for the image in figure 6.3.

As described later using a sliding window increases the robustness and ac-

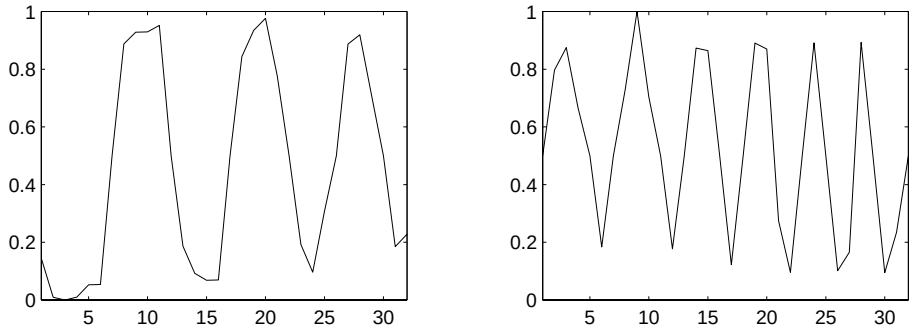

Figure 6.4: The image of figure 6.3 split up into a left and right part

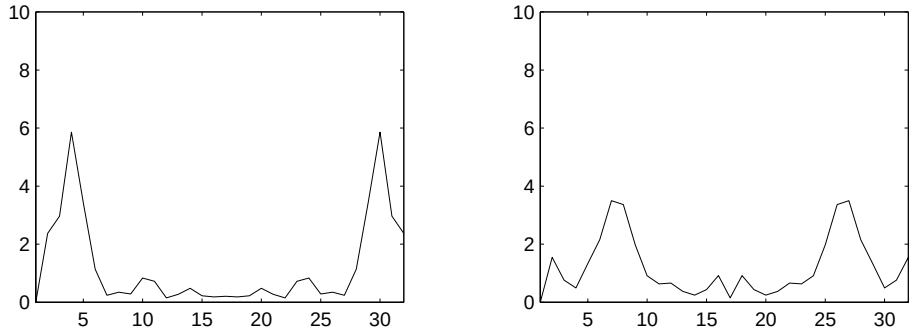

Figure 6.5: The FFT analysis of the left and right parts of figure 6.4. The left figure has its peak at index 4 and the right one at index 7

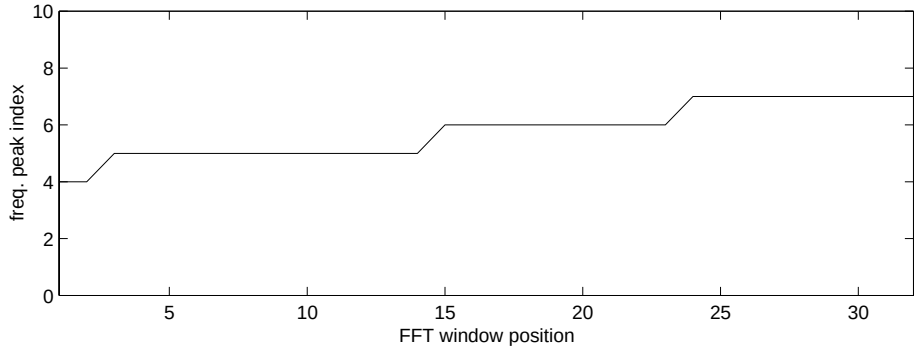

Figure 6.6: The indices of the frequencies peaks resulting from the continuous FFT analysis of the camera image in figure 6.3. The climbing line without disturbances indicates a good camera image and that the camera was standing left towards the wall.

curacy. It also enables more accurate measuring of the distance.

6.7 Approximation of the peak line

In theory this peak line that looks as a rough approximation of a line is indeed supposed to be a line, because the stripe "frequency" increases constantly (The reason for this is describe in section 6.5). Since the FFT analysis only returns discrete values this peak line only increases on ordinal number on the y-axis.

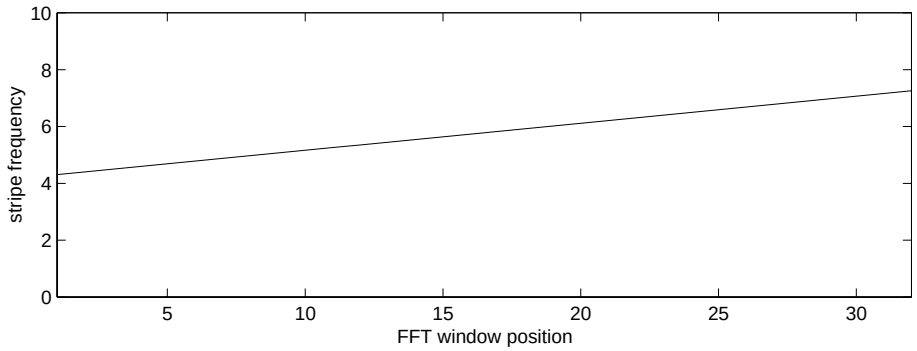

Figure 6.7: Reconstructed frequency increase line calculated from the peak line in figure 6.6.

However, the peak line data can be used to reconstruct the ideal, theoretical line to a certain extent. Provided in MATLAB is a function called `polyfit` that returns the approximate coefficients of a polynomial that fits closest the input data; in this case the peak line. The degree of the polynomial that should approximate the input data is given as a parameter. In this case it is 2 to find the closest line. Figure 6.7 shows this reconstruction.

6.8 Measuring distance

For distance measuring the same method of counting the periods was used as described in chapter 5.

Finding the "right" frequency is easy if the camera is looking perpendicular, since then there exists only one stripe frequency. However, in the case of non-perpendicularity, the stripe frequency is different at each position in the image – the stripe frequency for every position in the camera image is approximately reconstructed as described above. An example of such reconstruction is given in figure 6.7.

In order to calculate the distance as defined in figure 6.1 the frequency that is present in the middle of the camera image needs to be taken. This frequency is taken from the reconstructed stripe frequency change line (see figure 6.7) by taking the y-value at x-pixel position 16.5. This corresponds to the middle of the camera image.

6.9 Measuring angle

It was observed that whether the frequency peak line raises or falls the camera is either looking left or right towards the wall. It was further noticed that the amount of raising/falling is directly related to the size of the angle.

Measuring the angle in degrees proved to be difficult however. The relationship between the raise of the line and the angle is not straightforward. More important is that the amount of the raise measured is very inaccurate and changes rapidly when the robot is moved further away from the wall on the distance vector.

So calculating an actual degree was omitted. Instead it was only chosen to to see whether the peak line raises or falls, and based upon this it is decided whether the camera is looking rightwards (line raises or angle is positive) or leftwards (line falls or angle is negative).

6.10 Robustness

As described in the previous section measuring the angle works only to a limited extend. However, constructing a frequency peak line turned out to greatly increase the robustness against bogus[1] camera images.

The algorithm used to reconstruct the stripe frequency line from the FFT analysis returns a value called "norm of the residuals". Effectively this norm describes how good the input data matches an actual line. So this norm can be used to measure the quality of the camera image. If the input data to reconstruct the frequency line is too far away from being a line, this norm increases in value and as such can be used to sort out bad images. Figure 6.8 and 6.9 show an example.

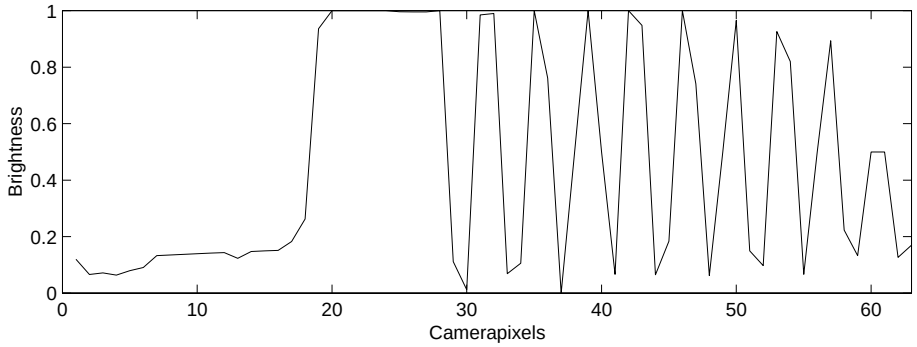

Figure 6.8: A sample image that is too bad to measure the distance from.

Source code

The source for the angle's measuring routines is printed on page 43.

[1] "bogus" in the sense that the camera image does not show an image that was taken from stripes or that the image of stripes is seriously distorted.

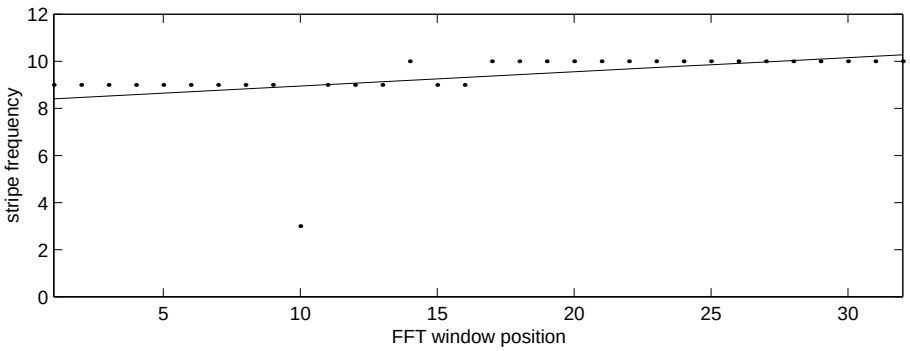

Figure 6.9: The frequency peak line and reconstruction. For some positions the continuous FFT analysis gave bad stripe frequencies. Reconstructing the frequency line from this data returns a too high deviation of the input data from the (ideal) frequency line. In this example the "norm of the residuals" is 6.26. As a reference norm values above 6 are considered to be the result of too bad input data.

Appendix A

Source Code

In this appendix all important source code is printed. Usually it is a direct inclusion of the files. If you would like to get the latest versions – or simply hesitate to type it in – contact the author to get it in electronic form. Future revisions of this document may include pointers to home pages where you may directly download the sources.

A.1 Robot API - kopen

```
/**
 * Opens the serial communication to the robot on Unix machines
 *
 * ref = kopen([portid, baudrate, timeout])
 *
 *                     portid: 0 = /dev/ttya (serial port A)
 *                             1 = /dev/ttyb (serial port B)
 *
 *                     baudrate: Must be 9600.
 *
 *                     timeout: Number of seconds to wait for the
 *                              respond of the robot.
 *
 *                     ref:  Handle to be used with the other serial
 *                           communication routines.
 *
 * (C) Matthias Grimrath <m.grimrath@tu-bs.de>
 */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <termios.h>
#include <fcntl.h>

#include <math.h>
#include "mex.h"
```

```
/* Settings for the serial communication */

#define ROBOTDEVICE "/dev/tty"
#define MAXTIMEOUT 10000000   /* equals 10s */

static int kopen(int portid, int baudrate)
{
    int fd;
    int i;
    struct termios tio;
    char devname[sizeof(ROBOTDEVICE)+10];

    /* create device name */
    strcpy(devname, ROBOTDEVICE);
    if (portid==0)
strcat(devname, "a");
    else
strcat(devname, "b");

    /* Open modem device for reading and writing and not as
     * controlling tty because we don't want to get killed if
     * linenoise sends CTRL-C.  */
    fd = open(devname, O_RDWR | O_NOCTTY);
    if (fd < 0)
return fd;
    tcgetattr(fd,&tio); /* save current modem settings */

    /*
     * Ignore bytes with parity errors and make terminal raw and dumb.
     */
    tio.c_iflag = IGNPAR|IGNBRK;

    /*
     * Raw output.
     */
    tio.c_oflag = 0;

    /*
     * Don't echo characters
     */
    tio.c_lflag = 0;

    /* Set bps rate and hardware flow control and 8n2 (8bit,no
     * parity,2 stopbit).
     */
    tio.c_cflag = CLOCAL|CREAD|CSTOPB|CS8;
    switch (baudrate) {
    case 9600:  baudrate = B9600; break;
```

```
    case 19200: baudrate = B19200; break;
    case 38400: baudrate = B38400; break;
    default:    baudrate = -1;
    }
    i =  cfsetospeed(&tio, baudrate);
    i |= cfsetispeed(&tio, baudrate);
    if (i)
        goto close_exit;

    tio.c_cc[VMIN]=0;
    tio.c_cc[VTIME]=0;

    /* now clean the serial line and activate settings */
    tcflush(fd, TCIFLUSH);
    tcsetattr(fd,TCSANOW,&tio);

    return fd;

close_exit:
    close(fd);
    return -1;
}

/* Gateway to MATLAB */

void mexFunction(int nlhs,       mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    int portid;
    int baudrate;
    int timeoutvalue;
    int retval;
    double *arg;

    /* Check for proper number of arguments */
    if (nrhs != 1) {
mexErrMsgTxt("KOPEN requires one input argument.");
    } else if (nlhs > 1) {
mexErrMsgTxt("KOPEN requires one output argument.");
    }

    /* Check the dimensions */
    if ((mxGetM(prhs[0])!=1) || (mxGetN(prhs[0])!=3)) {
mexErrMsgTxt("KOPEN requires argument to be [portid,baudrate,timeout].");
    }
    if (!mxIsDouble(prhs[0])) {
mexErrMsgTxt("The arguments must be of type double!");
    }

    /* Create a matrix for the return argument */
```

```
    plhs[0] = mxCreateDoubleMatrix(1, 2, mxREAL);

    /* Assign pointers to the various parameters */
    arg = mxGetPr(prhs[0]);
    portid        = (int)arg[0];
    baudrate      = (int)arg[1];
    timeoutvalue  = (int)(arg[2]*1000000);  /* timeout in s */

    if (portid!=0 && portid!=1)
mexErrMsgTxt("Illegal portid. Only ttya (=0) and ttyb (=1) are "
     "supported!");
    if (baudrate!=9600 && baudrate!=19200 && baudrate!=38400)
mexErrMsgTxt("Only baudrate of 9600, 19200 or 38400 is supported)!");
    if (timeoutvalue==0)
mexErrMsgTxt("You WANT to specify a timeoutvalue!");
    if (timeoutvalue>=MAXTIMEOUT)
mexErrMsgTxt("Timeoutvalue >10s, that's too much");

    /* Open the serial port */
    retval = kopen(portid, baudrate);
    if (retval < 0) {
        char buff[1024];
        snprintf(buff,1023,"KOPEN system error: %s",strerror(errno));
        buff[1023]=0;
        mexErrMsgTxt(buff);
    }

    arg = mxGetPr(plhs[0]);
    *(int *)(arg++) = retval;       /* Store as int in double to avoid */
    *(int *)(arg++) = timeoutvalue; /* conversion time */
}
```

A.2 Robot API - ksend

```
/**
 * See ksend.m
 *
 * (C) 1999 Matthias Grimrath <m.grimrath@tu-bs.de>
 */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <termios.h>
#include <fcntl.h>

#include <math.h>
#include "mex.h"

#define CMDLEN 1024
```

```
#define RCVLEN 1024

void mexFunction(int nlhs,       mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    int     m;
    int     ret;
    int     fd;
    int     cmdlen;         /* So 0-bytes can be sent as well */
    char    cmdbuf[CMDLEN];
    char    rcvbuf[RCVLEN];
    char   *rcvptr;
    int     rcv;
    double *arg;
    int     alwayswaittimeout;
    int     timeoutvalue;
    fd_set  readfds;
    fd_set  junkfds;
    struct timeval tv;

    /* Check for proper number of arguments */
    if (nrhs!=2 && nrhs!=3)
mexErrMsgTxt("KSEND requires two or three input arguments.");

    /* Check and eval string argument */
    m = mxGetM(prhs[0]);
    if (m!=1)
mexErrMsgTxt("Input must be a row vector!");
    if (!mxIsChar(prhs[0]))
mexErrMsgTxt("Input must be a string");
    cmdlen = m * mxGetN(prhs[0]);
    ret = mxGetString(prhs[0], cmdbuf, CMDLEN);
    if (ret)
mexErrMsgTxt("Command exceeds outputbufferlength");

    /* Check and eval fd argument */
    if ((mxGetM(prhs[1])!=1) || (mxGetN(prhs[1])!=2))
mexErrMsgTxt("'ref' must be reference obtained from KOPEN.");
    if (!mxIsDouble(prhs[1]))
mexErrMsgTxt("'ref' must be of type double!");
    arg = mxGetPr(prhs[1]);
    fd = *(int *)(arg++);
    timeoutvalue = *(int *)(arg++);

    /* Check and eval multiline, if present */
#if 0
    if (nrhs==3)
        mexErrMsgTxt("Multiline unimplemented!");
#endif
```

```
    /* Send the string */
    write(fd, cmdbuf, cmdlen);

    /* Wait for response */
    FD_ZERO(&readfds);
    FD_SET(fd, &readfds);
    tv.tv_sec  = timeoutvalue / 1000000;
    tv.tv_usec = timeoutvalue % 1000000;

    rcvptr = rcvbuf;
    while (rcvptr != rcvbuf + RCVLEN) {
junkfds = readfds;
rcv = select(fd+1, &junkfds, NULL, NULL, &tv);
if (rcv<0) {
    if (errno!=EINTR) {
char buff[1024];
snprintf(buff,1023,"KSEND select error: %s",strerror(errno));
buff[1023]=0;
mexErrMsgTxt(buff);
    } else
continue;
}
if (rcv==0)
    break; /* timeout */
rcv = read(fd, rcvptr, rcvbuf + RCVLEN - rcvptr);
        if (rcv<0) {
            char buff[1024];
            snprintf(buff,1023,"KSEND read error: %s",strerror(errno));
            buff[1023]=0;
            mexErrMsgTxt(buff);
        }
rcvptr += rcv;
/* The khepera sends both 'cr' and 'lf', despite the documentation
 * says only 'lf', but maybe just these stupid unix terminalmodes
         * mess it up
 */
if (rcvptr-rcvbuf>=2 && strncmp(rcvptr-2,"\r\n",2)==0) {
    rcvptr[-2]=0; /* Discard cr and lf */
    plhs[0]=mxCreateString(rcvbuf);
    return;
}
    }
    /* Come here if no valuable response received */
    plhs[0] = mxCreateString("");
}
```

A.3 Robot API - kclose

```
/**
```

```
 * Closes the serial communication to the robot on Unix machines
 *
 * kclose(ref)
 *
 *              ref: Reference obtained from kopen
 *
 * (C) Matthias Grimrath <m.grimrath@tu-bs.de>
 */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <termios.h>
#include <fcntl.h>

#include <math.h>
#include "mex.h"


void mexFunction(int nlhs,       mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    double *arg;

    /* Check for proper number of arguments */
    if (nrhs != 1) {
mexErrMsgTxt("KCLOSE requires one input argument.");
    } else if (nlhs > 0) {
mexErrMsgTxt("KCLOSE requires no output arguments.");
    }

    /* Check the dimensions */
    if ((mxGetM(prhs[0])!=1) || (mxGetN(prhs[0])!=2)) {
mexErrMsgTxt("KCLOSE requires argument to be reference obtained"
     "from KOPEN.");
    }
    if (!mxIsDouble(prhs[0])) {
mexErrMsgTxt("The argument must be of type double!");
    }

    /* Close down serial connection */
    arg = mxGetPr(prhs[0]);
    close(*(int *)arg);
}
```

A.4 Direct measuring of stripe width

```
function [dist,distw,distb,visw,visb] = distby_count(vis,thickness)
% [dist,distw,distb,visw,visb] = distby_count(vis,thickness)
%
```

```
% Measures the distance by counting the thickness of stripes
% closest to the middle of the camera view
%
% Input
%    vis: vision input vector from camera in the range [0 1]
%    thickness:  thickness of the stripes
%
% Output
%    dist:  distance to wall, -1 if it couldn't be determined
%
%    The rest should not be used except for debugging

  % separate in black and white
  visw = (vis>0.6);
  visb = (vis<0.4);

  % Handle it
  [visw, distw] = do_stripe(visw,thickness);
  [visb, distb] = do_stripe(visb,thickness);

  if distw==-1 | distb==-1
    dist = -1;
  else
    dist = (distw + distb)/2;
  end


function [ret,dist] = do_stripe(vis,thickness)

  ret = vis;
  dist = -1;

  stripes = count_stripes(vis);
  if length(stripes)==0
    return;
  end

  % Take a close look at the middle
  midblk = find(stripes(:,2)>33 & stripes(:,1)<=33);
  if length(midblk)==0

    % try looking for a block left or right
    blk = find(stripes(:,1)>33);
    if length(blk)==0
      blk = find(stripes(:,2)<=33);
      if length(blk)==0
        return
      else
        blk = 33 - stripes(blk(1),:);
        ps  = blk(1) - blk(2);
```

```
      end
    else
      blk = stripes(blk(1),:) - 33;
      ps  = blk(2) - blk(1);
    end

    p = 32 - ps;
    dist = (((p*thickness)/ps)+thickness)/tan(18*pi/180);

    return

  else
    mid = stripes(midblk,:);

    pl = 33-mid(1);
    pr = mid(2)-32;

    w = thickness*pr/(pr+pl);
    p = pr;
    ps = 32 - p;
    dist = (w+(ps*w/p))/tan(18*pi/180);

    return;
  end


function [widths] = count_stripes(vis)
  % counts

  widths = [];
  state = 0;
  cur  = 0;
  leadzero = 0;
  for i=1:length(vis)
    switch state

    case 0
    if vis(i) ~= 0
      start = i;
      state = 1;
    else
      leadzero = 1;
    end

    case 1
    if vis(i) == 0
      hill = vis(start:i-1);
      if leadzero~=0
        hill = hill./mean(hill);
        widths = [widths; start-hill(1) i-1+hill(end)];
```

```
      end
      leadzero = 1;
      state = 0;
    end
    end
  end
```

A.5 Measuring the distance

```
function [dist,frt_r,frt_a,ampl,ind] = distby_fft(vis, thickness)
% [dist,frt_r,frt_a] = distby_fft(vis)
%
% Measures the distance by counting the stripes
% trough a FFT transformation.
%
% Input
%    vis: vision input vector from camera in the range [0 1]
%    thickness:  thickness of the stripes
%
% Output
%    dist:  distance to wall, -1 if it couldn't be determined
%    frt_r, frt_a:  Distance and angel of the FFT transformation
%    ampl: internal, just for debugging purposes returned
%    ind:  same here

  frt = fft(vis);
  frt_r = abs(frt);
  frt_a = angle(frt);

  ampl = frt_r(1:32);  % Cut off high frequencies
  ampl(1) = 0;  % This contains a bogus value

  ampl = ampl./max(ampl);
  ampl = ampl.^3;
  ampl = (ampl > 0.4).*ampl;
  ind = find(ampl);
  ind = median(ind);

  if ind < 3.5
    dist = -1;
  else
    dist = (thickness*(ind-1)) / (tan(18*pi/180));
  end
```

A.6 Measuring the angle

```
function vision
```

```
global finished debugg

GUIsetup
debugg=0;

comm_open(2,9600,1,debugg);

finished=0;
fak=4;
datsize=64*fak;

cam_ax = subplot('Position',[0.1 0.8 0.8 0.15]);
cam_hd = plot(0:63);
set(cam_ax,'YLim',[0 1]);
set(cam_ax,'XLim',[0 63]);

cam2_ax = subplot('Position',[0.1 0.6 0.8 0.15]);
cam2_hd = plot(0:datsize-1);
set(cam2_ax,'YLim',[0 1]);
set(cam2_ax,'XLim',[0 datsize-1]);

pl1_ax = subplot('Position',[0.1 0.4 0.8 0.15]);
pl1_hd = plot(0:datsize/2-1);
set(pl1_ax,'YLim',[0 12]);
set(pl1_ax,'XLim',[0 datsize/2-1]);

pl2_ax = subplot('Position',[0.1 0.2 0.8 0.15]);
pl2_hd = plot(0:datsize/2-1);
set(pl2_ax,'YLim',[0 15]);
set(pl2_ax,'XLim',[0 datsize/2-1]);

% Gamme correction table for camera.  The contrast gets worse
% in the corners
corrw = ones(1,64)./((4/(32*32))*([0:63]-32).^2+1);
corrb = ones(1,64)./((0.5/(32*32))*([0:63]-32).^2+1);

wavelen = 0.01;
lowpass = hanning(8);

% Loop until global finished is set by the End command button:
while ~finished
  while ~finished

    [cameravals ok] = read_vt_cam;
    vis2 = cameravals'./256;

    vis = gamma_correct(vis2,corrw,corrb);

    vislin = [interp1(1:64,vis,1:1/fak:64) 0.5*ones(1,3)];
    vislin = filter(lowpass,1,vislin);
```

```
    vislin = vislin - min(vislin);
    if (max(vislin)~=0)
        vislin = vislin./max(vislin);
    end

    contfrq = zeros(datsize/2,1);
    contphi = zeros(datsize/2,1);
    for i=1:datsize/2
        a = fft(vislin(i+1:i+datsize/2));
        a = [0 0 a(3:datsize/4)];
        [dump,contfrq(i,1)] = max(abs(a));
        if dump < 6.4
            contfrq(i,1) = 0;
        end
    end

    [coef,s] = polyfit(0:datsize/2-1,contfrq',1);

    if (s.normr >= 6) | (coef(2) < 2.5)
     disp('Not a striped wall');
    else
        if coef(1) < -0.01
            disp('looking from the right');
        end
        if coef(1) > 0.01
            disp('looking from the left');
        end
        if (coef(1) >= -0.01) & (coef(1) <= 0.01)
            disp('looking straight');
        end
    end

    ind  = polyval(coef,datsize/4);
    tanphi = tan(18*pi/180);
    dist = (wavelen*(ind-1)) / tanphi;

    disp([dist coef(1)]);

%    disp([coef s.normr dump]);

    % Display it
    set(cam_hd,'YData',vis);
    set(cam2_hd,'YData',vislin);
    set(pl1_hd,'YData',contfrq');
    set(pl2_hd,'YData',polyval(coef,0:datsize/2-1));
    %while ~finished
      pause(0.5);
    %end
  end
```

```
  % If no evalution possible, just display camera input
  set(cam_hd,'YData',vis);
end

comm_close;
close all % close windows
return


function [ret] = gamma_correct(vis,corrw,corrb)

  avg = mean(vis);
  vis = vis - avg;
  v1 = (vis > 0.1).*vis;
  v2 = (vis <= -0.1).*vis;

  if max(v1)>=0.1
    v1 = (v1./max(v1)).^corrw;
  end
  if min(v2)<=-0.1
    v2 = (v2./min(v2)).^corrb;
  end

  ret = v1 - v2 + max(v2);
  if max(ret)~=0
    ret = ret./max(ret);
  end

  return;


function [ret] = linear_interpolate(vis,fak)

  ret = zeros(1,length(vis)*fak);
  for i=1:length(vis)-1
    d = (vis(i+1)-vis(i))/fak;
    ret((i-1)*fak+1:i*fak) = linspace(vis(i),vis(i+1)-d,fak);
  end

function reach_pos(dist,pos,err)

  tomove = dist - pos;

  if abs(tomove) > (err/2)
    if (tomove>0.1)
      tomove=0.1;
    end
    move(tomove*1000,1);
  end
```

```
  return


function GUIsetup
% Defines the graphical user interface.
global GUI xxxh  finished
global bck
bck=[1 1 1];

UIinit('- Khepera Vision output', 6, 16)
set(gcf,'Color',[1 1 1])
xxxh=gcf;

UIpos('rightmost','bottom')
UIdir('left');
UIctrl('cmdEnd','End','cmd','global finished; finished=1;');
UIctrl('cmdTurn','forward','cmd','move(10,1);');
UIctrl('cmdTurn','backward','cmd','move(-10,1);');
UIctrl('txtDist', '0.10','edit');

UIfinish
return %GUIsetup
```

Bibliography

[1] GNU Texinfo pages for the C library on Unix and the like. Available from the Free Software Foundation, any good FTP server and Linux distribution.

[2] Application Programming Interface Guide, Version 5. Online version found in {Matlab installation directory}/help/pdf_doc/matlab/api/apiguide.pdf

[3] Minicom - An Open Source software package that implements several data transfer protocols over modem lines amoung other things. Version 1.82.1 Homepage: <http://www.clinet.fi/~walker/minicom.html>

[4] The K-Team. Homepage: <http://www.k-team.com>

[5] Digital signal processing: principles, algorithms and applications / John G. Proakis, Dimitris G. Manolakis. – 2nd Edition, 1992, Macmillan Publishing Company.

[6] Signal Processing Toolbox User's guide / Thomas P. Krauss, Loren Shure, John N. Little. – February 1994, The Mathworks Inc.