AALBORG UNIVERSITY

STUDENT REPORT

Title

SUBTITLE

Project Group
SW506E15

Supervisor
SØREN KEJSER JENSEN

December 21st, 2015

This report was typeset using LaTeX.

DRAFT

AALBORG UNIVERSITY
STUDENT REPORT
Software - 5th semester
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
http://www.cs.aau.dk/

Title
Noget med lego

Theme
Embedded Systems

Project Period
Autumn Semester 2015

Project Group
sw506e15

Participants
Alexander Dalsgaard Krog
Alex Josefsen
Kasper Kohsel Terndrup
Mathias Plesner Bloch Kristensen
Lasse Just Petersen
Kenneth Husum Stick

Supervisor
Søren Kejser Jensen

Copies
2

Number of Pages
63

Attachments
0

Date of Completion
December 11, 2015

bla bla STUFF

This page intentionally left (almost) blank.

Preface

The following report is written by six 5th semester software engineering students, and is intended for the evaluation of a semester project. This entails that the reader is expected to be familiar with the subjects of real-time problems in embedded systems and machine intelligence. It is also expected that the reader is proficient in mathematical fields, such as linear algebra, trigonometry and calculus.

All sources will be cited before the sentence ends, such as it has been done in this sentence [0]. *Italicized* text is used to highlight to certain names, variables or terms - an example being "The *derivative* term is used to...".

Part of the challenge of this project was to find a compatible combination of available hardware and ideas, which would solve the problem. This was done through a process of going back and forth between ideas and hardware, examining hardware and brainstorming ways of applying its properties to a solution, and later repeating the process with an altered or entirely different approach. In order to try to convey this process in the report, the design was structured in two parts, a preliminary design in chapter 3 and a final design in chapter 5, with the hardware examination in between at chapter 4. The intention is to grant the reader insight into the reasoning, at the cost of not portraying the process chronologically.

Aalborg University, December 21st, 2015

Alexander Dalsgaard Krog
<akrog13@student.aau.dk>

Kenneth Husum Stick
<kstick13@student.aau.dk>

Lasse Just Petersen
<ljpe12@student.aau.dk>

Mathias Plesner Bloch Kristensen
<mpbk13@student.aau.dk>

Kasper Kohsel Terndrup
<kternd13@student.aau.dk>

Alex Josefsen
<ajosef13@student.aau.dk>

Contents

1 Introduction

Embedded systems have been in use for a long time, with one of the earliest examples of a microprocessor-based commercially available system, being the Busicom calculator based on the Intel 4004 microprocessor, which was released in 1971 [1]. Modern embedded systems are usually based on microcontrollers, which in addition to a processor also includes memory and input/output connections. Embedded systems are used in many everyday devices, and it is estimated that more than 90 percent of today's computer systems are embedded systems [2]. This of course means that a large part of future software engineering will be concerned with embedded systems, which is also the focus in this project.

One of the interesting areas in which embedded systems are in use today is tracking. Tracking is used in a many different systems, such as surveillance systems and military weapons, especially anti-air weapons. These systems can be very large in size, and have very high requirements for accuracy, reaction time, and constructional integrity. This project will focus on some of the underlying challenges of building a weapon system, but on a much smaller scale.

Motivation for this project is drawn from the popular genre of computer games called tower defense. The purpose of the game is to strategically position defensive towers, such that they can shoot moving targets to stop them from reaching their goal. A deeper explanation can be found in section 1.1. In a physical adaption of a tower defense game, the tower will have similar challenges to a military turret.

This leads to the following initiating problem.

How can a physical adaptation of a tower defense game, with focus on the tower, be developed?

This report will cover the development of the project that attempts to solve this initial problem.

FINAL: CHANGE MARGIN BACK!

1.1 Tower Defense

The tower defense genre is popular with countless individual games being produced, the most played game on Armorgames.com alone having been played 64 million times [3]. The purpose of the game is to prevent enemies from reaching a certain point on a map. This is done by placing defensive towers, usually different types, on the map, which will attack approaching enemies. Figure 1.1 is a screenshot of a tower defense game. The amount of towers available to the player is limited and because of this the challenge of the game is to strategically place the towers around the map, so that their different strengths are utilized.

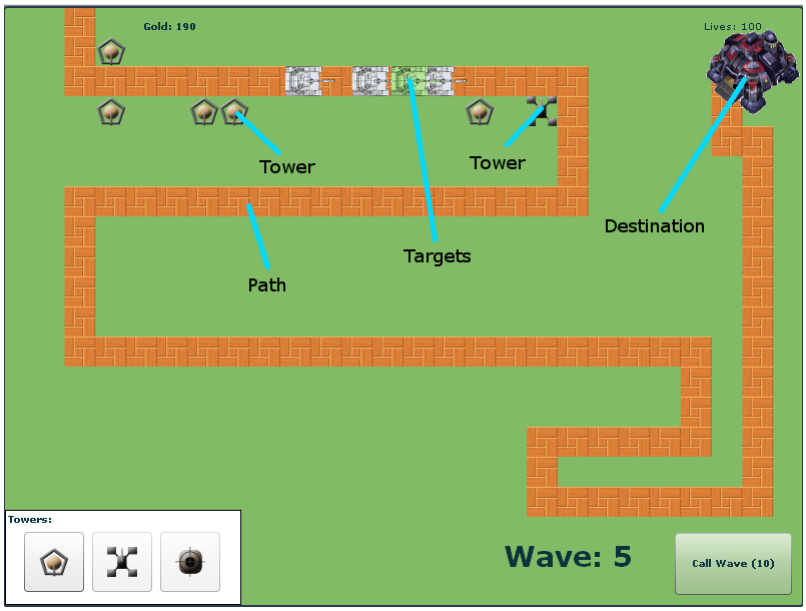

Figure 1.1: *Simple Tower Defense [4]*

Tower defense games often adopt different tower placement policies. Some games have maps with a predefined path that all enemies will follow. The player must then place their towers along this path. This is the policy used in the game seen in figure 1.1 where the path is represented by the red bricks, everything placed on the green area is a tower and the enemies are the tanks on the path. Other games do not have a set path, but instead rely on the player creating a maze using towers, the enemies must then navigate through the maze before they can make it to their destination, seen as a large building in the top right of figure 1.1.

This project will adopt the policy of a predefined path along which the enemies must travel. This means that the enemies will always be coming from the same direction, effectively allowing all towers to wait in the most optimal position for intercepting incoming enemies.

The target chosen for this project is a LEGO train. This was selected because it can travel along a predefined path of tracks. The speed of the train can also be adjusted, and because it is made of LEGO bricks the train can be modified to fit the needs of the project. These properties were deemed sufficient for a choice of target, since the movement of the target is not a primary concern of this project. The train is able to move with varying speed, but it will be limited to a constant speed while passing the tower.

2 Analysis

In order to create a real world implementation of a tower in a tower defense, we must examine the underlying problems. In computer games, the towers act autonomously, and they can instantly know of any targets within its range. It is also trivial to aim and hit targets, which in the games are mostly done for visual appeal, and not actually a requirement. In a real world implementation, the towers should remain autonomous, and as such, the points of surveying an area for targets, aiming, and hitting your targets, are much more complicated.

The processes of a tower can therefore be split into three parts: detection, tracking, shooting, which in turn can be divided further into sub-parts. This chapter examines the challenges that compose this problem.

2.1 Detection

An unavoidable task when creating a tower is to enable it to detect that an object is present in the observable area. The observable area is the entire area in which a given tower can possibly detect a target. This can be done with a wide array of sensors and techniques. However, just registering that an object is present somewhere is rarely enough, and as such, basic information is required. Firstly there is the matter of distinguishing between a target and a non-target. Considering the delimitation of the LEGO train, an example of a non-target could be the tracks which the train runs on. We must therefore define some properties of a target, from which we can identify it.
A target must always be:

1. Moving along a track inside the observable area.
2. Visually distinguishable from non-targets.

Detecting a target in the observable area will require monitoring of said area. The direct approach is to constantly cover the entire observable area, but a tower might not have such a large field of view. Field of view is defined as the area which a tower can observe at a given moment. If complete constant coverage is not a viable solution, sensors can either watch key points in the area, or have their field of view moved around the observable area in some manner.

2.2 Tracking

Tracking a target, is closely related to detection of a target. However they differ in the amount and type of information they require . Tracking a moving target can be a very complicated procedure as there are many factors to consider. These factors can be the speed of the target, the direction of the target, the speed of the tracking device, etc.

Alex: kilde

A target may be moving at a constant speed, but its speed can also vary. The tracking system needs to react to these changes in speed quickly to avoid losing sight of the target, especially if the target is moving at high speed. Part of this problem is removed in this project as the target will always remain at a constant speed while passing the tracking system, as described in section 1.1. However, this speed may change in-between passes. The direction of the target is also a part of the challenge. The tracking system will have to determine the direction in which the target is moving to ensure it isn't moving the offensive systems of the tower away from the target. The distance to the target also plays an important role. The closer to the target the tracking system is, the shorter the time span in which the target is within its observable area is.

kasper: HUSK: hvis vi ikke når at indføre variabel retning, så skal dette fjernes

The act of tracking ultimately boils down to detecting movement, analysing movement, and reacting accordingly.

2.3 Shooting a Target

For this project, the shooting part will be the launch of a LEGO projectile, since it fits the scope of the project and sufficiently illustrates common problems with reactions. The turret which fires the projectile is described later in the report, in section 5.1.
Certain factors have to be considered when firing projectiles at any given target. The most important factors are the velocity of both the target, and of the projectile, as well as the distance to the target, the accuracy of the projectiles and the turrets rate of fire. Another important factor is the predictability of the target, mostly in regards to the path which the target will take. However, since this project delimits itself to a tower defense scenario, a predictable path is assumed.
The projectile velocity, accuracy and rate of fire is covered in the testing of the constructed turret, see section 5.1, but some deductions can be made from the other factors alone.

2.3.1 Field of view

As the target enters the field of view, the shooting system has a limited time to respond, which depends on how fast the target is moving. A faster target will leave the field of view quicker, providing a shorter time frame for the projectile to hit the target. In this regard a faster target can therefore be interpreted as a smaller target. If we assume a degree of freedom in the distance from the target, we can increase the time a target spends in the field of view at the cost of increasing the travel time of the projectile to the target.

The distance from a tower to a given target inside its observable area, is not likely to be constant. In fact this is only the case when the target follows a path which curves in parallel to the tower. Assuming a straight path parallel to a tangent of the towers field of view, the distance to the target will increase polynomially in relation to the distance between the target and the tangents point of contact. This fact, in relation to velocity of the projectile, limits the maximum distance to the target, and such the effective observable area. This can

H

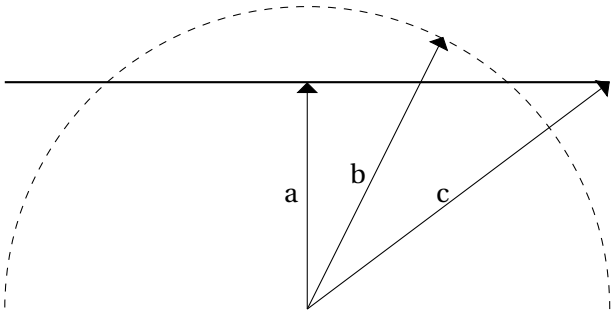

Figure 2.1: *find caption*

be seen on figure 2.1 where a at ninety degrees shows the shortest distance, b is at eighty degrees shows a maximum distance and c is at seventy degrees showing a distance out of reach. It can be seen that with the change in degrees the distance increase polynomial.

kasper: Dobbelttjek polynomial

The limit on the time a target spends in the effective range of a tower, also adds a requirement of time to the task of acquiring a target and releasing a projectile. If targets require multiple hits before being eliminated, which is the case in classical tower defense games, this will further increase the emphasis on time, since the faster the first projectile is fired, the sooner it will eliminate a target. However, for the purpose of a game, this could be considered a part of the challenge of managing your tower resources. So a tower should not nessesarily have the faster possible reaction, but rather a consistent one.

Alex: Caption til billede

2.3.2 Trajectory Prediction

When firing projectiles one must consider the correlation between the ratio of the projectile speed to the distance to the target, and the time it takes for the target to travel its own length. If for example the distance is 1 meter, and the projectile velocity is 1 meter per second, a target that moves faster than its own length in a second will not be hit by a projectile launched directly at the target. If we assume a passing target to move at a pace sufficiently high, that the turret can not successfully hit the target by firing immediately upon detection, it becomes necessary to be able to predict the movement of the target, and fire the projectile a distance in front of target.

In summation, the process of firing a projectile is a time sensitive task, in regards to both how long the target is in the effective range of the turret, how fast a projectile can be released, and how fast a prediction of the trajectory must be made.

2.4 Requirements

In light of the analysis, the following requirements for a turret in a tower defense game have been deduced. These requirements are considered the needs for any turret included in a final game. The lack of specific requirements reflect the fact that a key point of the game is to have different turrets, some which might have to shoot faster than a specific speed, and some which might require the ability to identify different targets, and so this list of requirements reflects the most basic type of turret.

1. A turret must be able to detect a target either through optics or by motion detection.

2. The turret must either constantly monitor the entire observable area, cover key points, or search the observable area as described in section 2.1.

3. A turret must be able to detect and analyze movement, and react accordingly.

4. Detection, analysis and reaction must be done faster than the movements of the target.

5. A turret must be able to predict the trajectory required to hit a target.

6. Prediction of trajectory, aiming and shooting a target must be done within a limited time frame.

Any solution to the initial problem does not necessarily have to involve all of these requirements, since some of the requirements are the byproduct of delimitations made in this project, such as requirement 1.

These requirements can be summarized into the question:

How does one implement a program on an embedded system, such that the system can detect, track, and shoot targets passing by?

This question, along with the requirements, will be the background for the design and implementation chapters.

3 Preliminary Design

This chapter will describe preliminary design ideas and techniques that can be used in a system which needs to track and hit a moving target. These solutions will conform to the requirements listed in section 2.4. The preliminary design will not take into account the limitations in hardware, but rather serve as a perspective on further design and to provide a standpoint for the hardware evaluation.

3.1 Detection

In section 2.1 two properties of a target was guaranteed. Those properties allow for two ways of detection: image recognition and motion-detection.

3.1.1 Image Recognition

Image recognition, as a detection method, involves analyzing a picture of the observed area in order to determine whether a target is present or not. There are two ways to do it: either the analysis looks for anomalies in the picture, or it looks for something that matches a preconceived signature of a target .

Alex: kilde

The first method compares any picture to a baseline, and reacts to changes. This requires a baseline to be established, which can prove to be quite complicated. If a static baseline is assumed, changes in light and shadows can potentially affect the analysis, resulting in false positives. Any instability in the camera mount can also result in the same problem. This could possibly be resolved through filters and margin tolerance. Another method is to have a more fluid baseline which continuously alters the baseline in accordance with new information. A fluid system would regulate the baseline such that small changes would not register as an anomaly, and only major disruptions from the baseline would. This would warrant a system that intelligently analyses the input. The idea of a baseline also requires the picture to be taken from a stationary position, or employ an elaborate system to map the 2-dimensional pictures onto a 3rd dimension.

The second method involves trying to match a preconceived signature to the picture. If the system is set up to recognize the shape of a square, the analysis will ignore everything that does not have a square shape. This would also allow for registration of multiple types of targets. This method also avoids the problem of the stationary camera and the baseline, but it is still vulnerable to image distortion and false positives. This could happen if squares seen from an angle are not registered or if something which is not a target happens to have a square on it.

3.1.2 Motion Detection

There are multiple ways of employing motion detection, however, the properties of the targets limit the sensor types to optical (which was covered as image recognition), ultrasonic and infrared.

No matter which is used, the methods are similar. One method is to establish a baseline, in the same fashion as image recognition. However, this baseline can be much simpler than the one used for image recognition. One example of a simpler baseline could be: "any object closer than 50 centimeters is considered a target". Once again a baseline could also be fluid, allowing for small changes while still detecting rapid changes. This could be useful in the case of a backdrop which curves or if the sensor is not always perpendicular to the backdrop. It is also possible to map the entire area, and register anomalies, with the same benefits and difficulties as with image recognition.

Using motion detection sensors do, however, require less computational power to analyze the data, in a trade-off with a limited perception of the actual composition of the area .

Alex: kilde

Summary

In summation, detection can be done with both image recognition and motion detection. Image recognition is however a more perceptive method and motion detection is a cheaper method in terms of required computational power.

3.2 Tracking

Tracking can be divided into two parts: determining position and reacting to movement.

3.2.1 Determining Position

Determining the position of a target, is a problem which is closely related to detection of a target. The problem of determining a position is a matter of interpreting the information acquired from the detection, and trying to infer a position based on this . In the case of image recognition it might be as straight forward as measuring the offset from the detected target to a known point.

Alex: kilde

When it comes to motion detection, the ability to figure out the position of a detected target can vary a great deal, based on the properties of the sensors. Two variables are to primarily be considered. One is the field of view of the sensors, and the other is the precision of the information regarding the position within the observed area . For example, imagine a sensor with a field of view matching the observable area, which can only give the information “detected” or “not detected”. In this scenario, pinpointing a

Alex: kilde

position is only possible if the velocity of the target is known, which can not be assumed. But let's imagine a very narrow field of view. Suddenly it becomes possible to estimate the position, by moving the field of view through the observable area and taking note of the position when a measurement is taken. The best case scenario however, is a sensor with field of view matching the observable area, which is also able to give precise information about the position of the detected object, in which case the process becomes trivial.

If the information acquired from a single sensor is limited, using multiple sensors can increase the amount of information gathered. When using multiple sensors, it is possible, depending on the properties of the sensors, to set them up in certain patterns which provide synergy. One such example is to let simple "detected"/"not detected" fields of view intersect. Such a setup will increase the information from the three possible areas, sensor A detects it, sensor B detects it, and neither detect it. With the fourth zone from the intersection: both sensor A and B detect it.

3.2.2 Reacting to Movement

Reacting to movement can be modelled as a two step process consisting of prediction and evaluation . The prediction step is the process of interpreting information on the position of target, and trying to predict its future position. The evaluation step is the process of evaluating a prediction, adding further information for the prediction step. The importance of the two processes varies based on different factors. If the information gathered through the sensors is considered accurate and plentiful, then the evaluation step will be a less important source of information to the point of not being of any support . But in cases with little or inaccurate information, the evaluation step can be of great importance, especially if the sensors are not covering the entire observable area, and are directed based on the prediction step. An example of this is the first measurement where a target is detected. At that point in time the prediction step can not know the speed of the target, and the prediction is likely to be wrong. In this case, it will often prove useful to evaluate the first prediction.

Alex: kilde

Someone: kilde eller omformuler

Unlike the evaluation step, the prediction step can not be marginalized. The problem with the prediction step is instead a problem of information dependence. It is obvious that a correct prediction will depend on some information, but the amount of information which is required for a correct prediction should be considered the benchmark for any tracking system. That said , the definition of "correct prediction" can vary, but in this project a correct prediction is considered to be on both position and velocity of the target, since these will be required for meeting all of the requirements. It is also important to note, that although correct predictions are always a priority, sometimes it can prove useful to initially focus on being in proximity of the target, such as with the narrow fielded but movable sensors.

Someone: talesprog

DRAFT

3.3 Trajectory Prediction

The last part of the preliminary design will consider trajectory prediction. The actual processes of aiming and shooting will not be covered in the preliminary since they are trivial when hardware factors are not involved, and are even then primarily implementation problems. Trajectory prediction does however deserve a mentioning. Given that the target is being tracked correctly, and thus our predictions on positions and velocity are correct, the right trajectory can be calculated given information on projectile speed and launch delays. But remembering that this process involves mechanical movement which is prone to inaccuracies (at least from a mathematical standpoint) a margin of error is expected. This includes the angle of the target relative to the turret, the width of the target, and whether the tracked point of the target is closer to the front of the target or the rear.

4 Hardware

This chapter examines the hardware that can be used for building the turret and the embedded control system. Different platforms and compatible components for embedded systems will be considered, and their limitations examined. A range of sensors will be examined in order to determine their behaviors and limitations. The chapter will also analyze actuators that can be used for the turret. The chapter will result in delimitations of possible designs.

4.1 Platform

The turret can be developed with one of several different platforms acting as the base. Aalborg University provides three primary platform choices, namely Arduino Uno, LEGO Mindstorms NXT and Raspberry Pi. The specifications of these three platforms are shown in table 4.1.

Platform	**Arduino Uno [5]**	**LEGO Mindstorms NXT [6]**	**Raspberry Pi 1 model A[7]**
CPU clock speed	16 MHz	48 MHz	700 MHz
Flash memory	32 KB	256 KB	SD card support
RAM	2 KB SRAM	64 KB RAM	256 MB SDRAM
Ports	14 digital I/O pins 6 analog input pins	4 input ports 3 output ports	17 I/O ports
Other	N/A	Separate I/O microcontroller Bluetooth support	250 MHz GPU

Table 4.1: *Hardware specifications for available platforms.*

4.1.1 Platform evaluation

When selecting a suitable platform for the project, there are several different criteria that must be taken into account. The importance of the individual criteria is not equal, as some will have a larger impact on this project than others. The four following criteria will be used for evaluation of the platforms.

Compatibility with sensors and actuators: In order to track objects, the platform must support input from sensors to observe the environment, and output to actuators to act on these observations. The ease of connecting sensors to the platform is also considered here, as easily connected sensors can decrease the time required to test and mount different sensors.

Range of available sensors and actuators: There are several different types of sensors and actuators, each with their own strengths and weaknesses. As several different types of sensors can be used, availability of a large range of sensors and actuators is desirable.

Computation power: The computational power of the platform will also be taken into consideration, as higher computation power might increase both the range of possible software design decisions, and the range of sensors available, such as cameras. Though more computation power might prove beneficial, this criteria is of lower concern, as it is assumed that limitations in computation power can be addressed in the design of the software.

Ease of constructing the turret: The focus in this project is the software aspect of the embedded system, and making use of sensors and actuators to interact with the physical surroundings, thus the construction of the turret itself is outside the main scope of the project. Having a platform that affords an easy construction of the turret is therefore important for the project.

The importance of each of the aforementioned criteria is listed in table 4.2. The criteria are assigned a value from one to three depending on their importance, with one being the most important and three the least important.

Alle:
Er i enige med vigtigheden af de forskellige kriterier?

Criteria	Importance
Sensor/actuator compatibility	2
Sensor/actuator availability	3
Computational power	2
Construction of turret	1

Table 4.2: *Importance of the evaluation criteria*

Table 4.3 shows the comparison of the platforms, where for each criteria, the platforms are ranked from one to three, where a ranking of one denotes the platform that best satisfies a given criteria.

Platform	Arduino Uno	LEGO NXT	Raspberry Pi
Sensor/actuator compatibility	2	1	3
Sensor/actuator availability	1	2	3
Computational power	3	2	1
Construction of turret	2/3	1	2/3

Table 4.3: *Platform ranking based on how well they satisfy the evaluation criteria.*

Lasse:
Hurtig forklaring af hvor de forskellige scores kommer fra

The Arduino Uno leads in the sensor and actuator availability criteria, with a very large range of sensors and actuators available. The sensors and actuators might have to be modified before use, resulting in a middle ranking for the compatibility criteria. The most important deficit of the Arduino Uno, lies in the ease of turret construction, which might

prove very time consuming with this platform.

The LEGO Mindstorms NXT is leading on compatibility of sensors and actuators, as sensors and actuators are easily connected to the NXT brick by cable. While trailing the Arduino on the availability criteria, there are still a large selection of sensors and actuators available for this platform. LEGO NXT ranks second in computation power, possibly allowing for more freedom in the software development over the Arduino. The LEGO NXT has its primary advantage in the construction of the turret, which is comparatively easy as the platform is designed to be used with LEGO bricks.

The main strength of the Raspberry Pi is its computational power, though this criteria is the least important for this project, while it is trailing behind the other platforms on the remaining criteria. Just as with the Arduino platform, the Raspberry Pi ranks low in respect to the ease of construction criteria, which is a very important limitation.

4.1.2 Final choice of platform

The Raspberry Pi is the least suitable platform for this project due to its primary strength being in the least important criteria, computational power, and having limitations compared to the other platforms in respect to the remaining criteria. The Arduino Uno would be a good platform choice, but the possibly very time consuming construction of the turret is seen as a severe limitation.

The LEGO NXT does not have any severely restricting limitations for this project, and has a large advantage in the ease of constructing the turret. This is seen as being very important for this project, as it allows the focus to be kept on the embedded software. The final choice of platform for this project will therefore be the LEGO Mindstorms NXT.

The choice of the LEGO NXT platform does, however, exclude the use of image recognition for tracking, as the process of acquiring and processing pictures with a high frequency can be very taxing on the system, in regards to both memory and computational power. In order to use image recognition with the NXT, it would be necessary to offload the storage and processing of images to another system with higher performance, and then communicate the results back to the NXT.

4.2 Sensors

The system will have to track and hit a moving target, namely a LEGO train. Given the nature of the project and the chosen platform, see section 4.1.2, two different sensor types will be considered: the LEGO NXT Ultrasonic Sensor [8] and the Mindsensors High Precision Medium Range Infrared Distance Sensor [9]. These sensors will be tested in order determine their condition and behavior, and to estimate their accuracy. The accuracy

of the sensors is important as they are used to determine the position of the target, as described in section 2.4, and irregularities in the readings can cause the system to be inaccurate when tracking the target, which would require compensation.

4.2.1 The Setup

To generate useful and comparable test results the testing setup will remain similar throughout all the tests. The setup consist of both the target and the turret. The testing setup was as follows.

- The target is a LEGO train with a white surface, width of X centimeter, and a height of Y centimeters.
- The target is traveling on a track with a height of one centimeter, and a width of 4.2 centimeters.
- The ultrasonic sensors are mounted on the turret, 16.5 centimeters above the ground.
- The infrared sensors are mounted on the turret, 12 centimeters above the ground.
- The ultrasonic sensors are located 45 centimeters away from the target.
- The infrared sensors are located 36 centimeters away from the target.
- Sensors in pairs are placed with a distance of four centimeters between them.

The difference in distance to the target between the two sensor types is due to optimal measurement distance of the infrared sensors being 10 to 40 centimeters [10]. The variation in height stems from the different sensor designs which means the infrared sensors have to be mounted differently, in a lower position. A testing was performed which concluded that this difference in sensor height between the two scenarios has no impact on the recorded data.

The sensors are first tested individually to ensure they are in working condition, and to determine their accuracy, then in pairs, as additional sensors will provide additional reference points, and thereby more information, as described in section 3.2, which might be necessary. The sensors are tested in both stationary mode, where the turret does not move in any way and remains pointing in the same direction, and in a sweeping mode, where the turret continuously alternates between turning left and right, between two specified boundaries. The purpose of this sweep mode is to simulate the movement of the turret tracking a target.

The data obtained from the tests will be presented in the form of a graph. The graph will contain the readings from the tested sensor, or sensors, and the position of the motor used to rotate the tracking system. The sensor data will be displayed with blue and green lines, which indicate their measured distance in centimeters. The red line represents the motors rotation in degrees. When this line has a y-value of zero, the sensors are pointing straight forward. Positive and negative values indicate a right or left rotation, respectively.

4.2.2 LEGO NXT Ultrasonic Sensor

The ultrasonic sensor measures distance by sending out a sound wave, and then calculating the time it takes for the sound wave to hit an object and return [11, p. 29]. According to the specifications the sensor works from a distance of 0 to 250 centimeters with a precision of ± 3 centimeters [11, p. 29]. The specifications note that the use of multiple ultrasonic sensors may interfere with the readings [11, p. 29]. The ultrasonic sensor has a field of view of approximately 30 degrees [12].

The first tests showed that the ultrasonic sensor registered the tracks resulting in inaccurate distance readings to the target, due to their large field of view. To counteract this property, a custom cover was manufactured for the ultrasonic sensor. The purpose of this cover is to minimize the field of view of the sensor, such that it doesn't register objects other than the intended target. A side-by-side comparison of the sensor with and without the cover can be seen in figure 4.1. The only tests described in detail are those performed with covers to avoid the detection of the tracks.

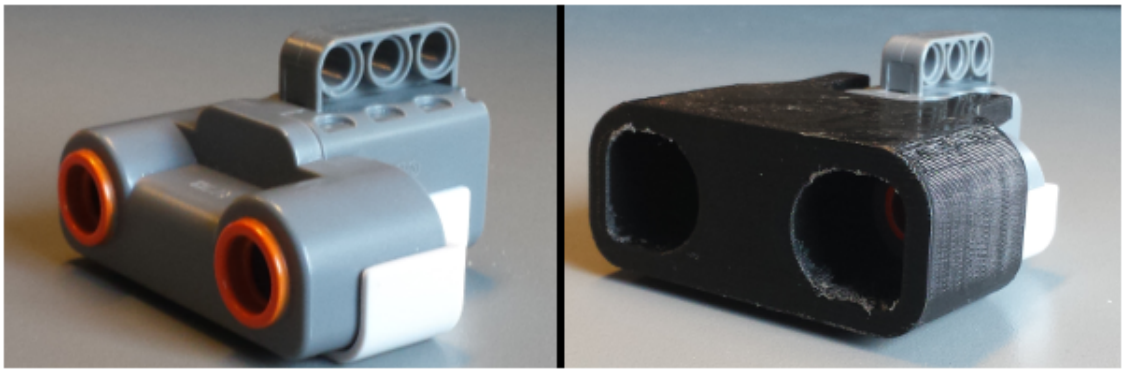

Figure 4.1: *The LEGO NXT Ultrasonic Sensor with (right) and without (left) the custom cover.*

Single Sensor

The sensors are tested individually in order to determine the accuracy and condition of the sensors. The sensors was tested in a stationary position, with the target also being stationary, in order to determine whether or not they were capable of producing accurate and consistent distance readings. The result of this test can be seen in figure 4.2. As the sensors performed identically, only one result is shown. The distance measured by the sensors was within three centimeters of the actual distance to the target.

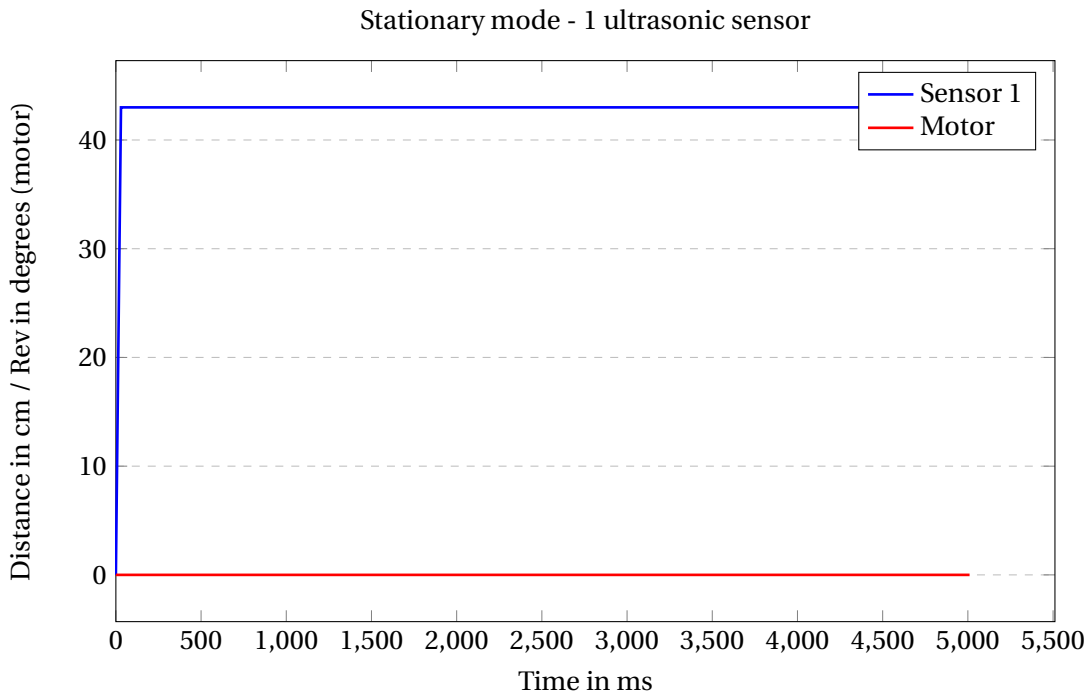

Figure 4.2: *Test with 1 ultrasonic sensor in stationary mode*

Multiple Sensors

Tracking using the ultrasonic sensors can be simplified using multiple sensors, as described in section 3.2. The two-sensor setup is tested with the turret in both stationary and sweep mode.

STATIONARY

With the target standing still directly in front of the turret in stationary mode the two ultrasonic sensors interfere with each other, creating small irregularities in the distance readings. These interferences can be seen in figure 4.3. The blue and green lines should both be straight as seen in figure 4.2.

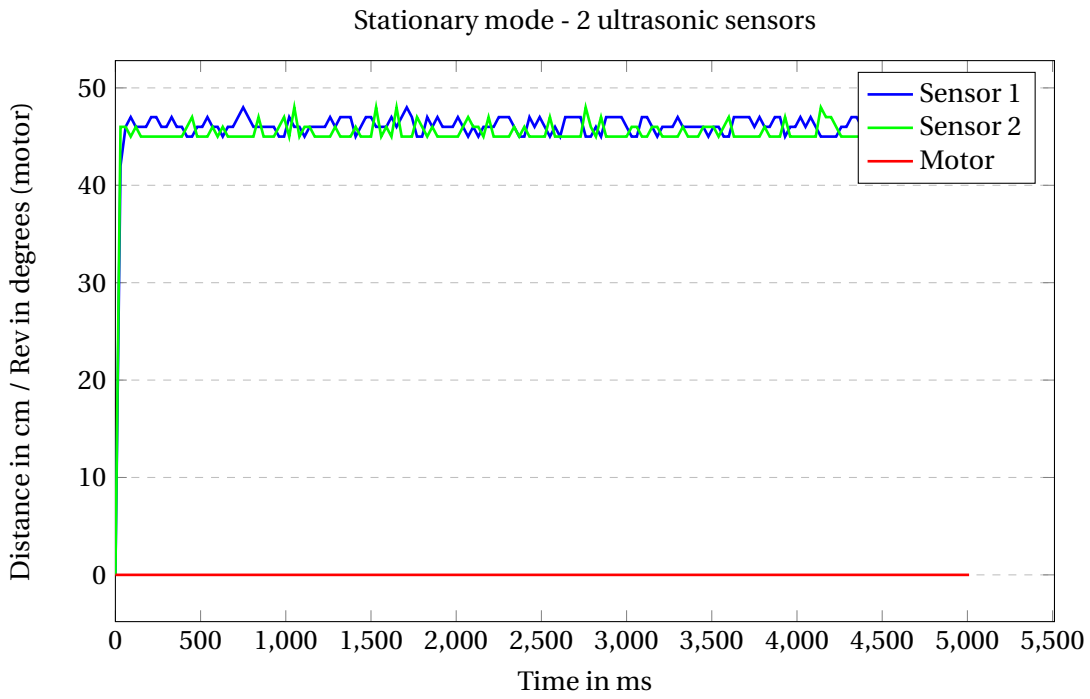

Figure 4.3: *Test with 2 ultrasonic sensors in stationary mode*

SWEEPING

Figure 4.4 shows the results from the test with the turret in sweep mode with two sensors mounted. Once again the two sensors interfere with each other. At times a sensor also sees the target for a longer time period than it should - this issue may be caused by it catching the reflected sound wave from the other sensor. The reflected sound wave may also behave irregularly if it hits the target at an angle, causing the sound wave to be reflected in a different direction than it originated from [13]. These inaccurate readings can result in the turret being unable to properly track the target, as the target may actually be in a different position than the one reported by the sensors.

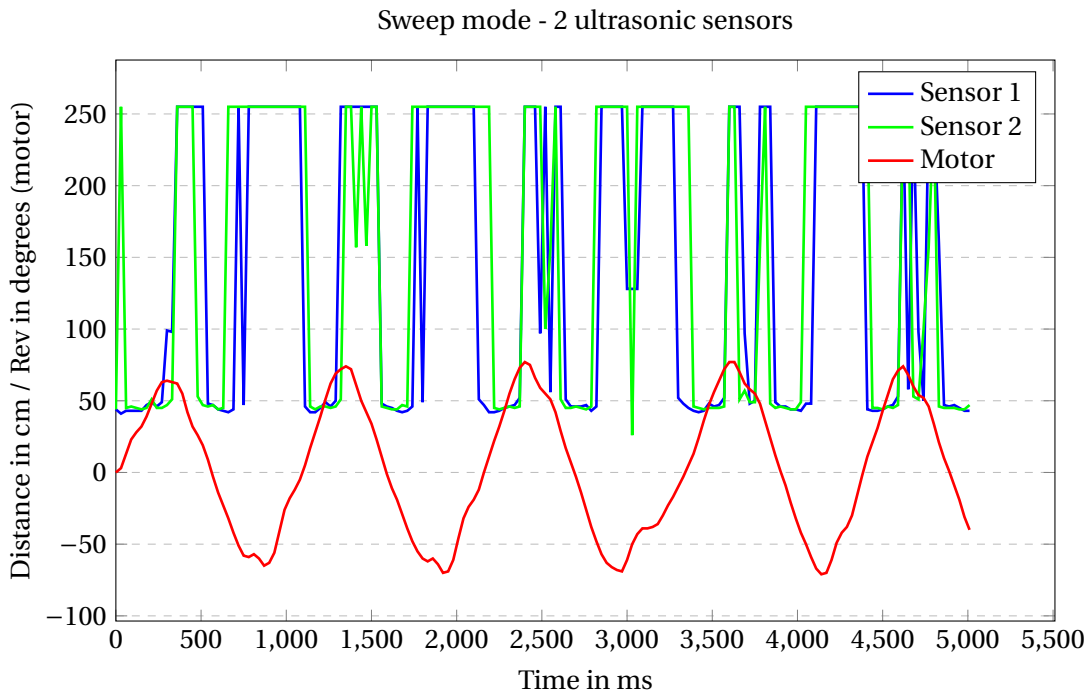

Figure 4.4: *Test with 2 ultrasonic sensors in sweep mode*

4.2.3 Mindsensors High Precision Infrared Distance Sensor

The infrared (IR) distance sensor from mindsensors measures distance based on the angle of the arriving reflected IR light it emits [14]. The sensor exists in three different variants, short, medium, and long, with different optimal ranges, shown in table 4.4. The medium range variants are provided by Aalborg University, and has a range suitable for this project. One of the infrared sensors can be seen in figure 4.5.

Figure 4.5: *An infrared sensor from mindsensors*

The precision of these sensors can be impaired when used in surroundings with high amounts of ambient light, and by surfaces that reflects light poorly [14]. The IR sensors measure distance in millimeters. According to the datasheet, the infrared sensor has a field of view spanning 12 centimeters when it is located 80 centimeters away from the target [15].

Range Variant	Distance
Long	20 - 150 cm
Medium	10 - 80 cm
Short	4 - 30 cm

Table 4.4: *The three range variants and their distances*

Faulty sensor

Again each sensor was tested individually, in order to determine their accuracy and condition. When testing these sensors, it was discovered that one of them was faulty, constantly measuring a distance of 9999 millimeters. As only two of this sensor type were available through Aalborg University, replacing it was not a feasible option, and instead further testing was performed, in an attempt to identify the error, and determine whether it could be mended.

In order to specify the origin of the fault, the architecture of the sensor was examined. A graphical representation of the internal communication of an IR sensor, along with its communication with the NXT brick is shown in figure 4.6.

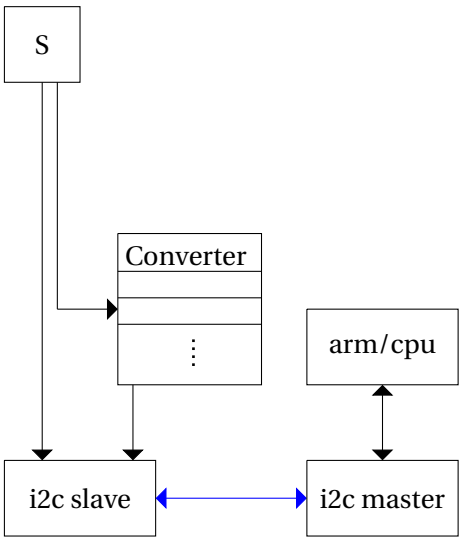

Figure 4.6: *find caption*

The blue arrow indicates communication between the IR sensor on the left side, and

the main processor in the NXT brick on the right side. The sensor denoted as *S* on figure 4.6 measures the voltage of reflected IR light, then uses a conversion table to map this voltage to a distance in millimeters. As shown on figure 4.6 both the voltage value, and the converted distance value, are stored in I2C registers. In order to test whether the fault was within the sensor module itself, the voltage reading was read directly. This test showed that the voltage reading was not constant, and thus the fault could be contributed to a fault in the conversion table. It was determined that this sensor could then still be usable, circumventing the fault by reading the voltage directly, and implementing a new conversion system on the NXT brick.

In order to implement this new conversion system, the voltage reading for each distance between 80 and 800 millimeters, in 20 millimeter intervals, was recorded, the results of which can be seen in figure 4.7.

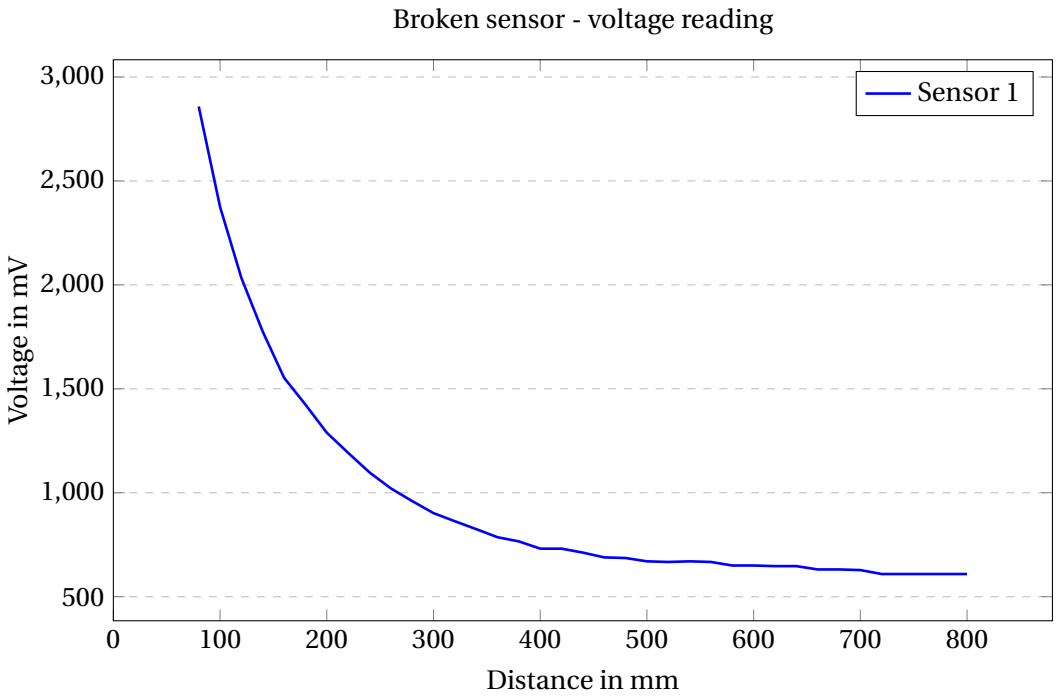

Figure 4.7: *Sensor voltage readings at different distances*

Based on these measurements, a power function that approximates the data was identified. This function acts as the base of the new conversion system, taking a voltage reading as input, and then returning a distance in millimeters. While the internal conversion table was calibrated according to general data from a new sensor, shown on figure A.1 in appendix A for comparison, this new conversion function is calibrated to this exact sensor and target, resulting in more precise distance readings, than a working internal conversion table would have afforded.

Furthermore, the possibly greatest benefit of using the custom conversion function on the NXT, is the possibility of increasing the sampling rate of the distance measurements, compared to using the internal conversion table. This is possible due to the voltage register of the I2C controller being updated more frequently than the distance register.

As the custom conversion function provided substantial benefits over using the internal conversion, it was decided to construct a custom conversion function for the other IR sensor as well, which was done using the same process. The data used in constructing the conversion function for this sensor can be seen in **??**, **??**.

The power functions follows the same general form, given by $f(x) = a/x^b$, where x denotes the voltage reading, and a and b are constants based on a nonlinear regression analysis on the recorded measurements.

Lasse:
Vi skal have grafen for voltage reading af den anden sensor ind i bilag.

Someone:
Vise beregning - how?

Single sensor

After implementing the custom conversion functions, the individual sensor tests was performed. Both the turret and the target were stationary for these tests. The result of the test of the faulty sensor can be seen in figure 4.8. The distance measured by the sensor was within three centimeters of the actual distance to the target. The other IR sensor performed very similarly, and as such the results of this test are not shown.

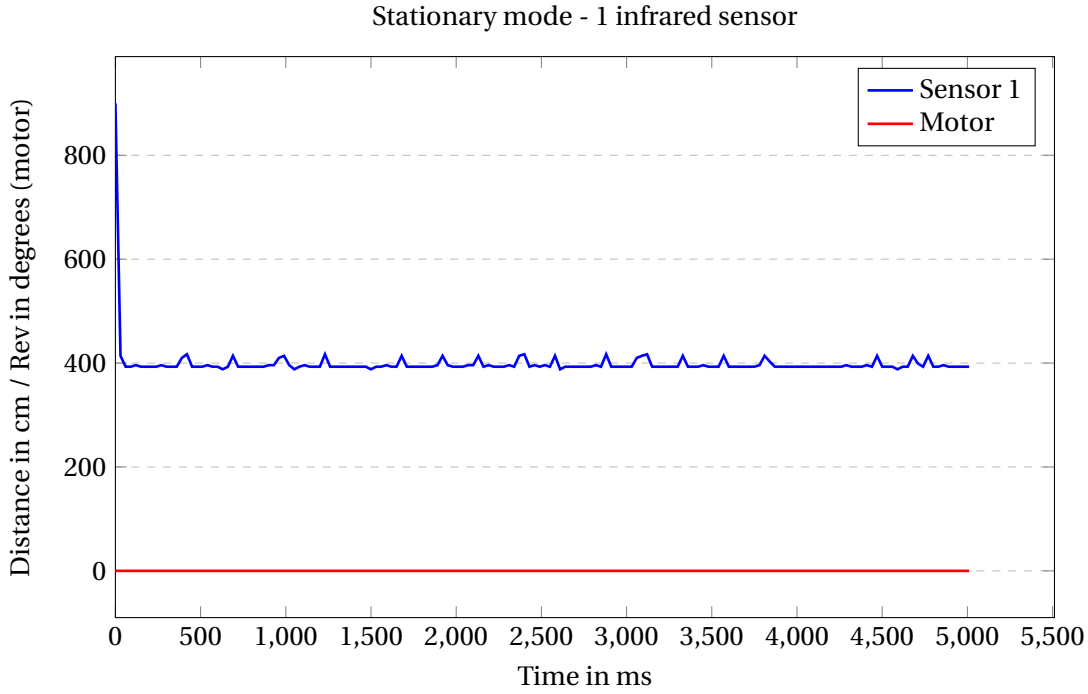

Figure 4.8: *Test with 1 infrared sensor in stationary mode*

Multiple Sensors

When using the infrared sensors for tracking, it is favorable to use multiple sensors as well, see section 3.2, why two sensors are tested. The two sensors are mounted on the turret, and are tested in both stationary and sweeping mode. As the custom conversion function outperformed the native function, both sensors use the custom function.

STATIONARY

The two infrared sensors do not interfere with each other when the target is stationary directly in front of the turret. Due to the width of the target only one of the two sensors is able to spot the target properly and measure the correct distance. The data collected from the test can be seen in figure 4.9.

Lasse:
Den her test skal nok tages om.

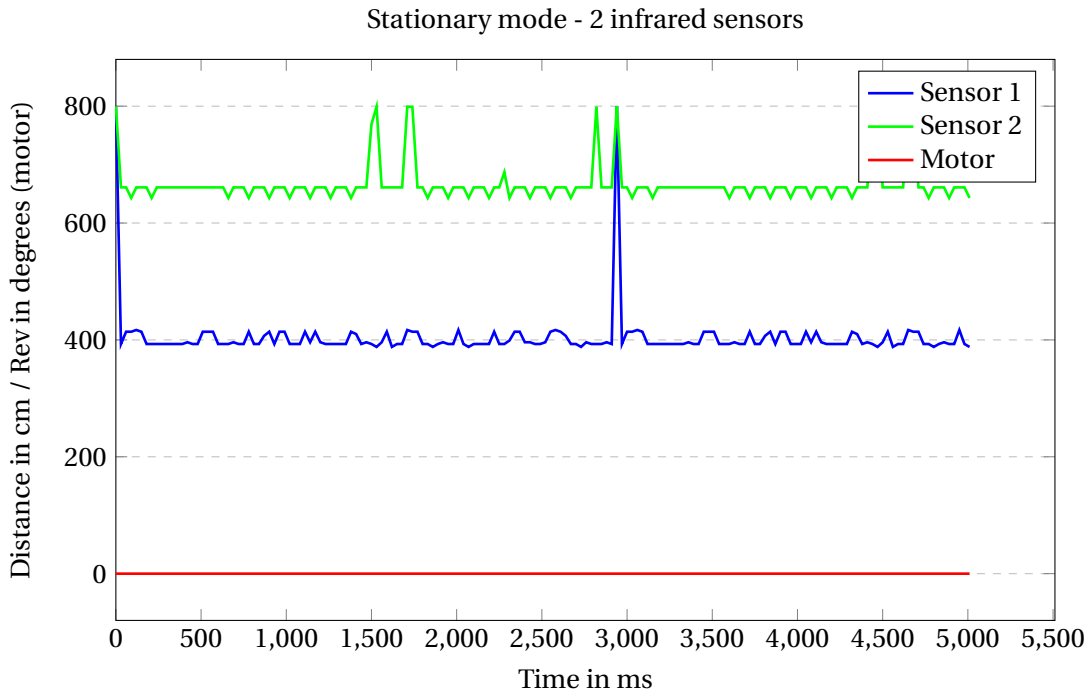

Figure 4.9: *Test with 2 infrared sensors in stationary mode*

SWEEPING

The results from the test with the IR sensors, and the turret in sweep mode, can be seen in figure 4.10. Here the distance readings are not very accurate. However, while the distance readings are not accurate, the sensors are still able to detect a change in the distance rather accurately. There are also minor overlaps in the readings from the two sensors, which indicates that both sensors have detected the target. As seen in figure 4.10 some irregularities exist, as such one more test to find out the workings of the cone for infrared sensors is done.

DRAFT

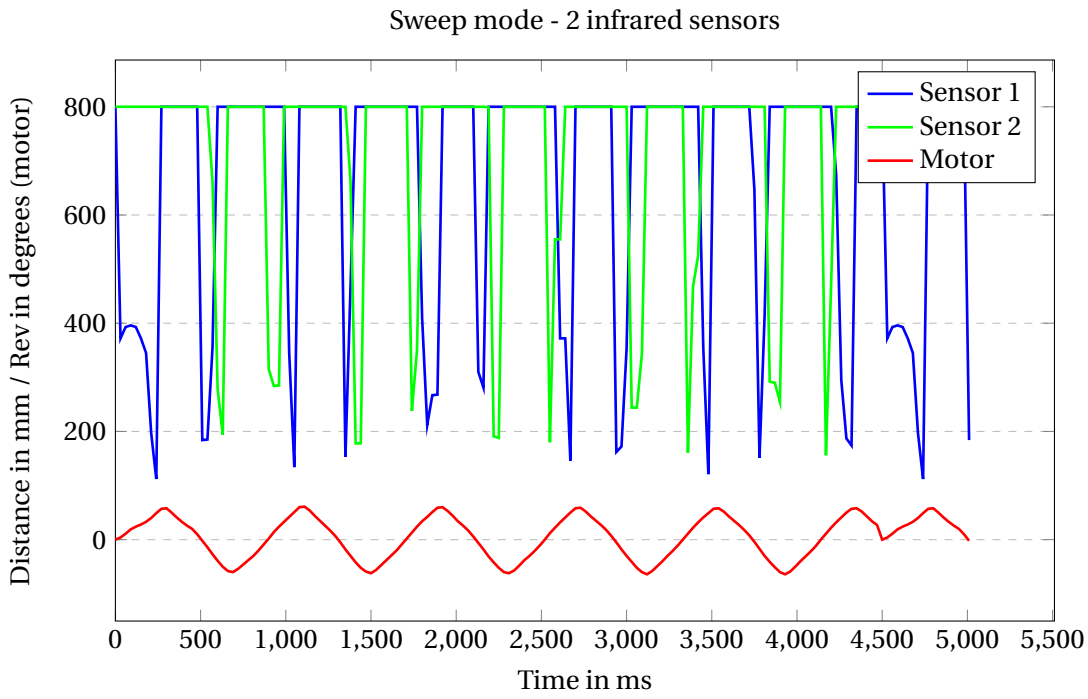

Figure 4.10: *Test with 2 infrared sensors in sweep mode*

Sensor Cone

A test of the IR sensors accuracy throughout its cone was also performed . In this test the turret was stationary, while a target was passing by. The resulting measurements are shown in figure 4.11 The distance measured by the IR sensors vary greatly depending on where in the cone the target was. These tests also show that the IR sensors distance measurements are imprecise in the majority of the sensors cone.

?:
Hvad var årsagen til at vi testede dette? Grafen med ens punkter kan være et tilfælde + den er semi-fake (den sidste del er gentaget, da der blev testet i lidt for kort tid)

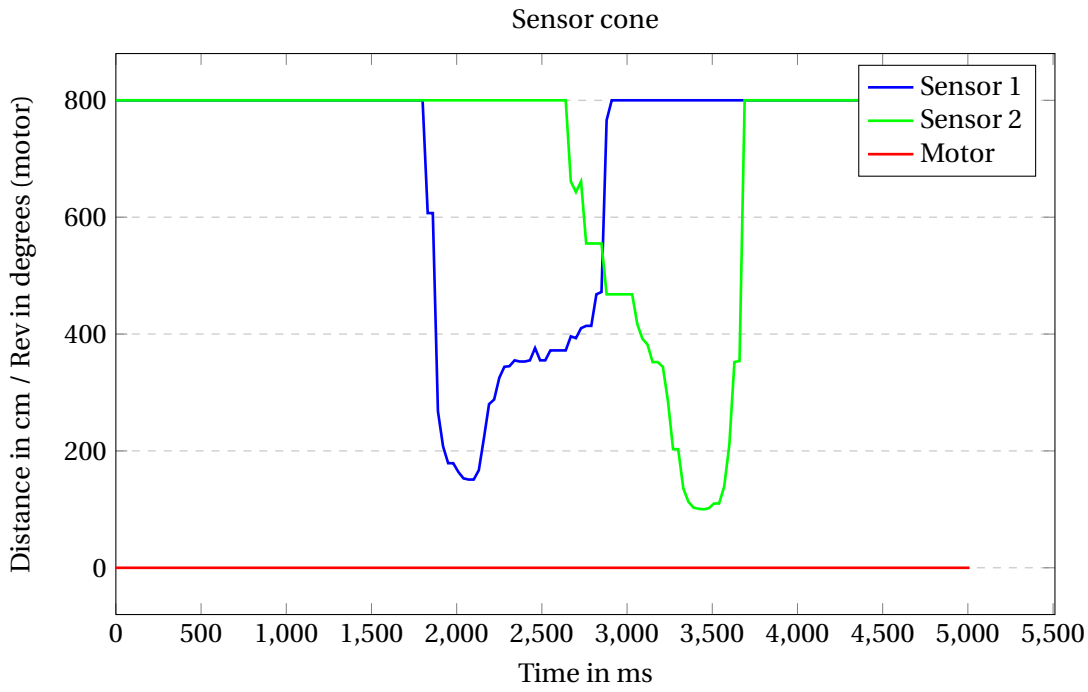

Figure 4.11: *Sensor cone*

4.2.4 Sensor Choice

The ultrasonic sensors have too much interference when multiple ultrasonic sensors are used. This was noted in the sensor specifications and was also confirmed during the tests. The infrared sensors have somewhat faulty measurements but this can also be seen as more information. With the infrared sensors providing more information and the fact that they have no noticeable interference with multiple sensors of the same type, the infrared sensors have been chosen as the sensors of choice for the turret.

4.3 Actuators

The system has to be able to aim at a target which requires the use of a motor that will allow the turret to rotate. Aalborg University provides the *Interactive Servo Motor* from LEGO Mindstorms. The motor has an encoder which tracks how far the motor has moved. The encoder supports two levels of precision: 360 ticks, which offers precision down to a single degree, and 720 ticks which offers precision down to half a degree. The standard library uses the 360 tick level of precision [16]. The standard nxtOSEK library has a function which instructs the motor to turn a given number of degrees. This function was tested and found to overshoot the target by approximately 70 degrees. As this level of inaccuracy is quite high, it is necessary to find a solution that allows the motor to turn more precisely.

The performance of the motors was tested by connecting them to an NXT and running a

program which counts the revolutions of the motor at variating power levels - from 10% to 100% in increments of 10%. The motor was running for 10 seconds at every interval. The motors were placed without any external load affecting them. Two different battery solutions were available for the NXT: a 7.4V rechargeable battery or six 1.5V batteries with a total output of 9V. The six battery solution was selected as the power of the motors was increased by 1.6V going from 7.4V to 9V.

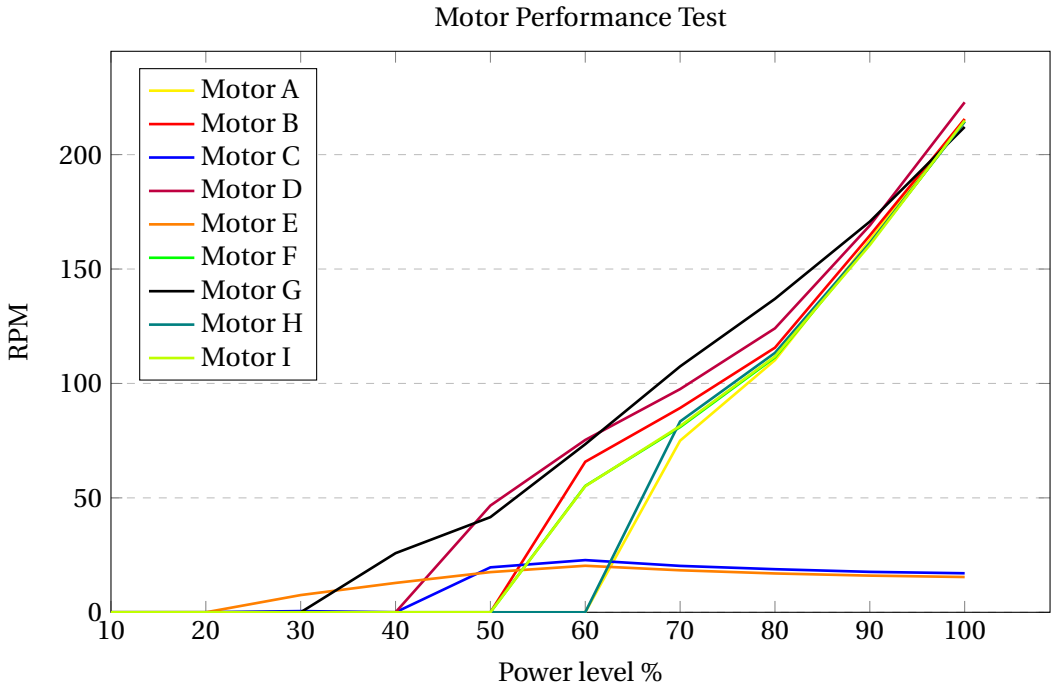

Figure 4.12: *Motor test*

Figure 4.12 shows the collected data from the motor tests. The x-axis denotes the power level percentage and the y-axis denotes the revolutions per minute (RPM) of the motor. The tests showed that motor *E* and *C* behaved abnormally compared to the other motors. At a certain point the motors started slowing down as more power was supplied rendering the two motors unusable. The other motors have different power levels at which they start rotating. The motors *D* and *G* have the lowest rotation starting points at 30% power and 40% power respectively. For all motors except *E* and *C*, the RPM rises steadily in a similar pattern as more power is supplied.

4.3.1 Motor Selection

Based on the motor test, see section 4.3 and the sensor test, see section 4.2, the most suitable components were selected for the turret. According to the motor test the motors *D* and *G* are the best with the remaining motors being nearly equal in performance - with the exception of *E* and *C*. Motor *D* has been selected as the rotation motor due to its performance, the early rotation starting point being important. Motors *G* and *B* have been

selected for the cannon. The cannon only requires motors with similar performance from above 60% power.

5 Design

This chapter will describe and explain the design and the reasoning behind it. The design is based upon the requirements in section 2.4 and the possible solutions presented in chapter 3, delimited by the hardware analysis in chapter 4. The design is a conceptual solution, and will only address the implementation to the extent of it being a possibility.

5.1 The Turret

The turret is purpose-built using LEGO bricks following a custom design made by the project group. The turret consists of three main parts: the cannon, the sensory system and the rotation system. A total of 3 motors are mounted on the turret to aid the cannon and the rotation system, using the best three motors described in section 4.3. The tracking system is made up of two sensors mounted on the turret aimed in the same direction as the cannon. The turret with infrared sensors and motors mounted can be seen in figure 5.1. It was the decision of the project group to keep the turret as "one unit", meaning that all of the turrets components had to be in direct contact with the rest. This was chosen in order to keep true to the original genre, by making the placement of a turret a simple process.

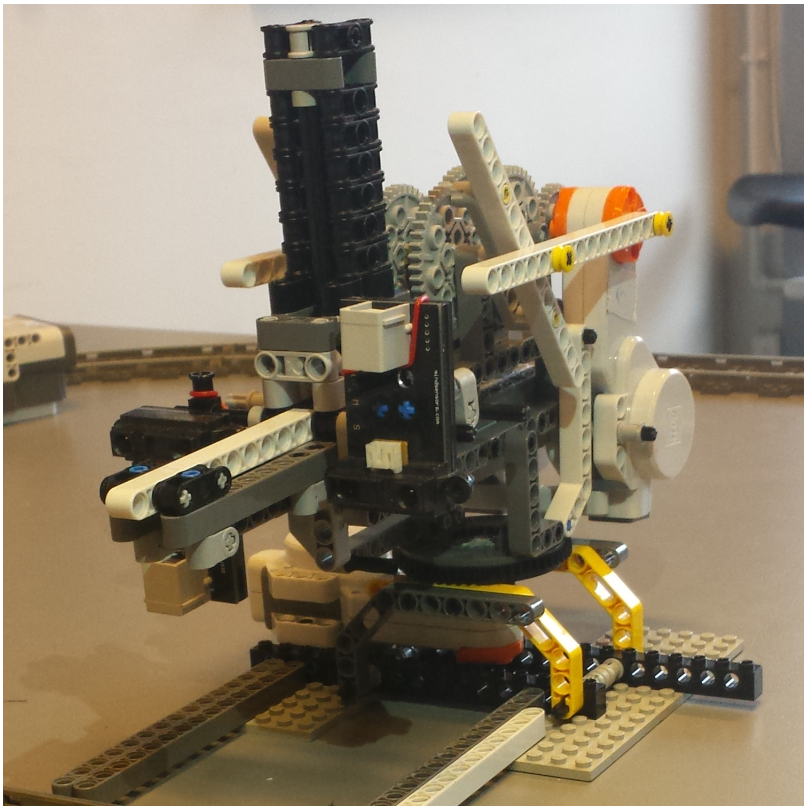

Figure 5.1: *The turret with infrared sensors mounted*

The turret uses LEGO bricks as projectiles. These projectiles are fired by releasing a piston bolt from a retracted position. The bolt, when released, is pulled into the barrel by stretched elastic bands which contracts and pushes the piston into the barrel and the loaded projectile, propelling it out of the barrel. The elastic bands are stretched using motor driven rotating rods, which in a circular pattern pulls the piston back to the apex of the stretch, and then sliding off the piston, releasing the bolt and allowing the elastic bands to contract. This allows for automatic fire of projectiles by continuously rotating the rods, however, it also makes priming the cannon in a cocked position somewhat difficult. The turret is loaded when the bolt is retracted as this allows a LEGO brick to drop down into the chamber just prior to the release of the bolt. The turret can hold a magazine with a capacity of 10 projectiles. The turret also allows for sensors to be mounted in various positions.

5.1.1 Projectile Accuracy

The accuracy of the fired projectiles was tested by placing the turret 75 centimeters away from a circular target with a diameter of 5 centimeters. At this distance a projectile drop off of 10 centimeters was recorded. A total of 10 projectiles were fired and all of them managed to hit the target. The hits were not centered in one location, but spread about the target within the 5 centimeter boundary. The projectiles had some difficulty penetrating the target, a sheet of paper, at the tested distance. This can be attributed to the shape of the projectiles as well as the loss of velocity at the tested distance. This accuracy was considered good enough for the project.

5.2 Proportional integral derivative controller

In this project the PID will be used to make the motors more precise and able to move to a specific amount of degrees to do this was found to be a problem in section 4.3. A proportional-integral-derivative (PID) controller is a commonly used control system. It can be used on a variety of systems and is a popular control system, partly because of its robustness and simplicity, but also because it can be used in many systems with only small changes. A PID controller is based on a control feedback loop, where the first execution of the loop only uses the current state and the goal state in the calculations. Subsequent executions of the loop uses information about the effects the previous loops had on the system. A PID consists of the terms P, I and D, which when summed gives the change that will be applied to the system. P is the proportional term, I is the integrale term and D is the derivative term.

A PID takes two inputs: the target state and the current state. The target state is the state that the PID is trying to reach, while the current state is the state measured by the sensors. A PID has 3 coefficients: K_p, K_i, K_d and two variables: *lasterror* and *integrale*. *Integrale* sums up all the errors and *lasterror* keeps the error from the last run. This information is used in the derivative term. The variable *error*, see code 5.1, is used to determine how far the system is from the target state; the bigger the *error* is, the more the system has to do to

reach the target state. The *error* influences all the terms. The P term is normally the most powerful term of a PID. Both I and D are used to stabilize the system in different ways, therefor, K_p is usually the biggest coefficient. The magnitude of the proportional term is decided by the size of K_p.

Integrale is used to make sure that if there is an error for a long time the change will keep rising. Both P and D can reach 0 if the error is sufficiently small, but if there is a small error, I is slowly growing and will eventually start changing the system - this kind of problem is called steady-state error. The magnitude of the *integrale* term is decided by the size of K_i.

Derivative is the changes that have happened since the last run. It is used to stabilize the system faster, as it attempts to predict the future. The magnitude of the derivative term is decided by the size of K_d. It is not used often in real world systems as its impact on the system varies [17].

```
1  //K_p, K_i and K_D are the PID coefficients that have been predefined
2  lasterror = 0
3  integrale = 0
4  PID(target, current)
5      error = target - current
6      integrale = integrale + error
7      derivative = lasterror - error
8      lasterror = error
9
10     return Kp * error + Ki * integrale + Kd * derivative
```

Code 5.1: Pseudo code of a PID

5.2.1 PID tuning

In order to make the PID controller work properly the three coefficients need to be tuned. There are several methods available for PID tuning, however, only two will be described here: manual tuning and the Ziegler-Nichols Method. The Ziegler-Nichols Method was chosen as it is a mathematically proven method and manual tuning because it is a good way to make changes to the PID based on specific demands.

Control Type	K_p	K_i	K_d
P	$0.50K_u$	-	-
PI	$0.45K_u$	$1.2K_p/T_u$	-
PID	$0.60K_u$	$2K_p/T_u$	$K_pT_u/8$

Table 5.1: *Ziegler-Nichols Method*

The Ziegler-Nichols Method is a mathematically well defined method used to tune a PID controller. The first step of the method is to set K_i and K_d to 0. K_p is then slowly increased until the motor oscillates. T_u is the rate at which it oscillates, and K_u is the K_p value at which

Mathias: Graf muligvis for bred

it oscillates. The three PID terms can then be calculated as shown in table 5.1. This way of tuning is aggressive and will result in a fast rise, it might even result in some overshoot.

	Rise time	**Overshoot**	**Settling time**	**Steady-state error**	**Stability**
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

Table 5.2: *Impact of increasing a parameter [17]*

Another method of system tuning is manual tuning. To use this method it is necessary to know which effect changing the coefficients has on the system. This information can be gathered from a tuning table, such as table 5.2, and then the coefficients can be tuned through a trial-and-error approach or meta optimization. Figure 5.2 shows a graph containing the properties of a PID. The rise time is the time it takes for a PID to reach the target state for the first time, a short rise time tends to result in overshoot. Overshoot is how much a PID misses the target by the first time, in some systems overshoot may be prohibited. The settling time is how long it takes to reach a steady state. There might be a steady state error when there is a small error when the steady state is reached. A PID can be unstable and never reach a steady state which means that the coefficients have to be changed in order to stabilize the system.

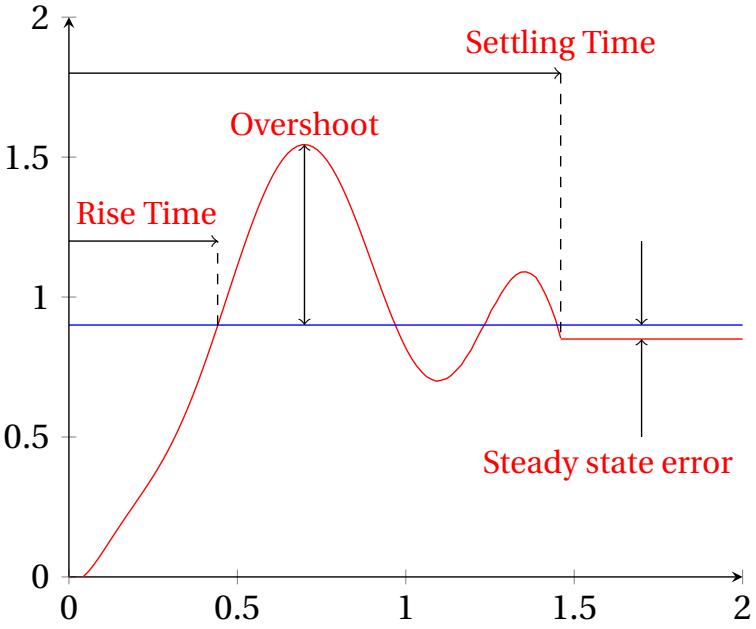

Figure 5.2: *PID settling time*

DRAFT

5.3 Detection

In section 3.1 of the preliminary design, the two methods of image recognition and motion detection were suggested. Then in section 4.1.2 of the hardware analysis, image recognition was deemed too computationally heavy for the embedded platform without off-loading the computations. This property resulted in image recognition being discarded, as the turret is to be autonomous according to its description in section 5.1. In light of this, this section will consider the design based on motion detection.

As written in section 4.2.3 the sensors have faulty measurements in certain areas of their fields of view. The sensors faulty measurements can be used to determine in which of three different areas of the field of view the target was detected in.

The areas, shown in figure 5.3, are determined by the distance reading. Firstly, any reading above 700 millimeters is considered noise, and is disregarded. If a reading occurs between 700 millimeters and 300 millimeters, the target is in one of the outer areas, denoted on figure 5.3 as O. If a reading is below 300 millimeters, it is in the inner zone, denoted by I.

Alex: noget omkring at bruge de 300-700 mm gør at tårnet helst skal stå i en bestemt distance fra track

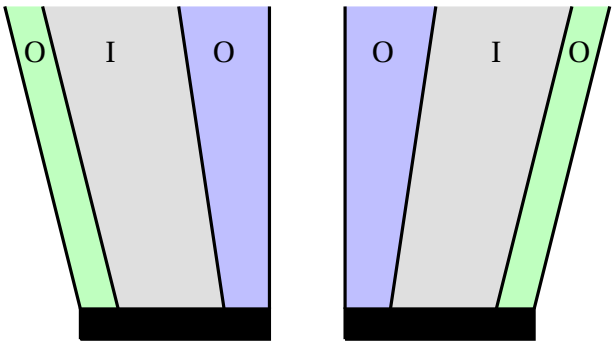

Figure 5.3: *Infrared sensor detection zones*

Interpreting the zones into offsets is met by two challenges. The first is to determine which of the outer zones the target was spotted in. Since the sensors allow for polling every 6th millisecond, it is very rare that a target will pass by the inner zone without being spotted. Therefore, any reading in an outer zone, which is not immediately preceded by its corresponding inner zone, is considered to be the outermost of the outer zones, unless the other sensor has already spotted the target.

The second challenge is to translate the zones into offsets. The offset of each area was found by measuring the distance from the rotation point of the tower to the train tracks and then measuring the offset of the train as it just shifted to a new area. Then inverse tangens was used to calculate the degree of each area, which can be seen in table 5.3. Since this calculation was done with the tracks at a distance of 470 millimeters from the rotation point, any deviation from this distance will make these exact numbers inaccurate.

kasper: præcis hastighed på polling + overvej at finde den præcise hastighed som skal til for at misse target i innerzonen

kasper: er det egentlig inaccurate?

area	-2	-1	0	1	2
degree	-17.37	-7.52	1.58	9.78	20.74

Table 5.3: *Sensor areas*

This gives the combined field of view illustrated in figure 5.4. Since the outermost outer zones were small areas, they were considered an extension of the inner zone. The inner zones are labelled “-2” and “2”, the innermost outer zones are labelled “-1” and“1”, with negative numbers for left side and positive for right side. If the target is spotted in both “-1” and “1”, it is instead read as a “0”, which indicates that the train is in the middle of the combined field of view of the two sensors.

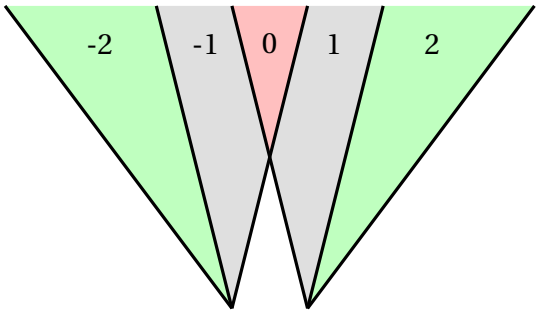

Figure 5.4: *Combined field of view*

5.4 Tracking

One of the primary tasks of the turret is tracking targets passing by in its observable area. Using the two-sensor setup, this tracking must preferably be done continuously, in order to obtain as much information about the target as possible, by keeping the target within the sensors field of view. This is done by having the rotation motor react to observations made by the sensors.

A very simple approach to this tracking problem, could be to simply activate the motor, turning it either left of right, based directly on each observation made by the sensors. The method could then be further refined by turning with different speeds, depending on the exact zone in which the target was observed. This approach might be effective if every observation provided by the sensors was perfectly accurate.

The sensors used for tracking are however not accurate, and produce noisy observations, which may result in a wrong estimate of the actual position of the target. If the turret were to act on these noisy observations, this might cause the turret to turn erroneously, possibly losing sight of the target, or overshooting and having to readjust immediately after. In order to improve the precision of turret, the noisy observations have to be taken into account.

5.4.1 Noise Handling

Certain noise handling methods can make it is possible to optimize the turrets tracking capabilities, by reducing the impact the noise of the observations will have on the system.

Naive Bayes Classifier

One possible approach for this, could be to use a naive Bayes classifier. with this model, instead of directly reacting on an observation, the position of the target can be represented by a probability, distributed over several different possible positions. The possible classes, contained in the class variable, would be all the possible positions of the target, which is equal to all the positions within the turrets observable area. A binary feature variable would then exist for each of the possible positions as well, where the binary value indicates whether the target was observed in the given position. This structure is shown in figure 5.5.

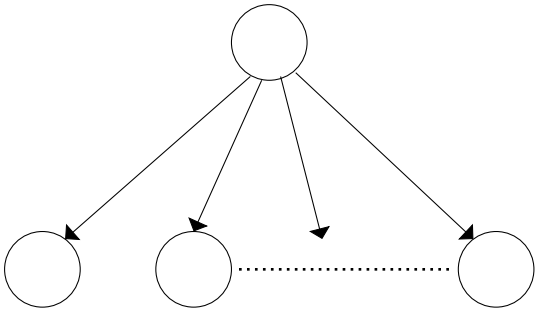

Figure 5.5: *Graphical representation of a naive Bayes classifier for identifying the targets position.*

Each time a new observation is made, the probability in the classification variable would be updated. This model does however have limitations, such as quite large probability tables as well as a lack of native support of the temporal aspect that exists in the project. Because of this, other models, with the temporal aspect included, will be considered instead.

Hidden Markov Model

A hidden Markov model (HMM) is a specialization of the more general dynamic Bayesian network, which is a Bayesian network where variables are related in respect to adjacent time steps. The model assumes the system to be a Markov process, where each state is dependent only on the previous state, meaning $P(x_k|x_0, x_1, ..., x_{k-1}) = P(x_k|x_{k-1})$, where x_k is the state of the system at discrete time step k. This relation can be seen in figure 5.6.

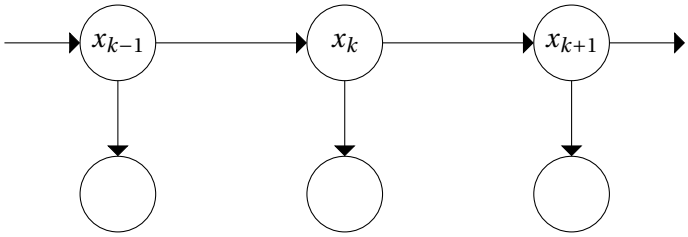

Figure 5.6: *Graphical representation of the hidden Markov model for the system.*

By making this assumption, it is possible to predict the next state of the system, given only the prior state. Additionally the model assumes that the state at each time step is not directly observable, instead a observation is associated with each time step. By using the observation, the values of the hidden state variable can be estimated. Hidden markov model reduces the size of the probability tables, but does still require one.

Kalman filter

The Kalman filter is based on a hidden Markov model, with the difference that it is used for linear systems only, and the values in the hidden state are continuous rather than discrete. While the Kalman filter does not require a Gaussian error distribution, in cases where the error is indeed Gaussian, the filter is an optimal estimator, meaning its estimate is as close to the. The Kalman filter is very suitable for use in this project, as the assumption of a linear system holds, and due to this limitation it outperforms the hidden Markov model. It also does not require training data, if the domain knowledge is sufficient, and does not require any probability tables.

In order to model the system for use with the Kalman filter, first the different equations used in the filter will be examined. As with the hidden Markov model, the state can be estimated using only the previous state. In the Kalman filter, this prediction is made according to the equation

$$x_k = F_k x_{k-1} + B_k u_k + w_k \,, \tag{5.1}$$

where F_k is a state transition matrix, which can be applied to the previous state x_{k-1}, in order to obtain a prediction for the state at the next timestep. u_k is a control vector, containing information about a known control input, such as acceleration, and B_k is a control-input matrix, which applies the vector u_k to the state transition. w_k is the process noise, assumed to be Gaussian distributed white noise with a covariance of Q_k, such that $w_k \sim \mathcal{N}(0, Q_k)$. Measurements are made according to the formula $z_k = H_k x_k + v_k$, where z_k is the measurement at time k, H_k is a measurement transformation matrix, that maps a state into a measurement, which is necessary when you can only partially measure the state. v_k is the noise in the measurement, also assumed to be Gaussian distributed white noise, with covariance R_k such that $v_k \sim \mathcal{N}(0, R_k)$.

The prediction step of the Kalman filter consists of two separate predictions. A prediction of the state estimate, seen in equation 5.2, where the notation $\hat{x}_{k|k-1}$ denotes the state estimate at time k, given the state estimate at time $k-1$, and a prediction of the covariance in this estimation, denoted P_k, seen in equation 5.3.

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k \tag{5.2}$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k \tag{5.3}$$

Compared to equation 5.1, the process noise is omitted in equation 5.2. This omission is due to the noise being assumed to be white noise, and as such, only the covariance in this noise, Q_k is of importance, and this covariance is included in the P_k calculation.

In the update step the state and covariance predictions are updated with information from a new measurement. Both these updates rely on a Kalman gain, that describes how much weight should be put on the prediction and the measurement. This Kalman gain K_k, is calculated using equation 5.4.

$$K_k = P_{k|k-1} H_k^T \left(H_k P_{k|k-1} H_k^T + R_k \right)^{-1} \tag{5.4}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k (z_k - H_k \hat{x}_{k|k-1}) \tag{5.5}$$

$$\begin{aligned} P_{k|k} &= P_{k|k-1} - K_k H_k P_{k|k-1} \\ &= (I - K_k H_k) P_{k|k-1} \end{aligned} \tag{5.6}$$

The state is updated with the new measurement according to equation 5.5, and the covariance is updated according to equation 5.6, where I is the identity matrix in the same dimensions as P_k.

The system model

In order to use the Kalman filter, it is necessary to model the system. The state of interest is the state of the target, and particularly its position and velocity. This results in the state vector shown in equation 5.7, with a 2-by-2 covariance matrix $P_{k_{2,2}}$.

$$\mathbf{x_k} = \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} = \begin{bmatrix} position_k \\ velocity_k \end{bmatrix} \tag{5.7}$$

The velocity of the target can not be measured by the sensors, so the measurement will consist of only a single value z_k, which is the targets measured position, with a single valued noise of v_k. The measured position maps directly into the state position, so disregarding the state velocity gives the measurement transformation matrix H_k shown in equation 5.8.

$$H_k = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{5.8}$$

Using this H_k matrix we obtain

$$\begin{aligned} z_k &= H_k x_k + v_k \\ &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_{pos} \\ \dot{x}_{vel} \end{bmatrix} + v_k \\ &= x_{pos} + v_k \end{aligned} \tag{5.9}$$

which describes the connection between a measurement and a state.

As each target is assumed to be traveling at a constant velocity, the system has no known control input, so equation 5.2 can be reduced to

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} \, . \tag{5.10}$$

Additionally it is assumed that the system will not be affected by any unknown control input, and as such the process noise covariance Q_k will be zero, and thus equation 5.3 can be reduced to

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T \, . \tag{5.11}$$

The state transition matrix F_k describes the change in the state between two timesteps. The change in the targets position depends on the elapsed time since the last prediction, and the velocity of the target, according to the equation $x_t = x_{t-1} + \dot{x}_{t-1}\Delta t$, while the velocity remains constant, given by $\dot{x}_t = \dot{x}_{t-1}$. This can be rewritten in matrix form as

$$\begin{aligned} \begin{bmatrix} x_t \\ \dot{x}_t \end{bmatrix} &= \begin{bmatrix} x_{t-1} + \dot{x}_{t-1}\Delta t \\ \dot{x}_{t-1} \end{bmatrix} \\ &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ \dot{x}_{t-1} \end{bmatrix} , \end{aligned} \tag{5.12}$$

giving the state transition matrix

$$F_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} . \tag{5.13}$$

While the state estimate and state covariance are calculated at each iteration of the filter, initial values must be specified. As the target are approaching from the same direction, and the range of which the turret can first detect the target is known, the initial position can be specified with high precision. The first position in which the target can be observed is 13 degrees left of the turrets initial position, zero. Any position to the left of zero will be a negative value, and the initial position will therefore be -13. The velocity of the target can, in general, have i possible discrete values $s_1, s_2, ..., s_i$, so the initial velocity is given by $s_{avg} = (s_1 + s_2 + ... + s_i)/i$. With the limitation of the three possible velocities 20, 50, and 80, this gives the initial state

$$\begin{aligned} \mathbf{x_0} &= \begin{bmatrix} x_0 \\ \dot{x}_0 \end{bmatrix} \\ &= \begin{bmatrix} -13 \\ (20+50+80)/3 \end{bmatrix} \\ &= \begin{bmatrix} -13 \\ 50 \end{bmatrix}. \end{aligned} \tag{5.14}$$

At which of the possible velocities a target is travelling, can not be determined before observations are made, thus the initial variance in the velocity will be given by $p_{0_{2,2}} = max(s_{avg} - s_1, s_i - s_{avg})$, where $max()$ is the maximal value function. The initial variance in position is set to $p_{0_{1,1}} = 0$, and as the the variances are independent of each other, the covariance is set as $p_{0_{1,2}} = p_{0_{2,1}} = 0$. This gives the initial covariance matrix

$$\begin{aligned} P_0 &= \begin{bmatrix} p_{1,1} & p_{1,2} \\ p_{2,1} & p_{2,2} \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 0 & max(s_{avg} - s_1, s_i - s_{avg}) \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 0 & max(50-20, 80-50) \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 \\ 0 & 30 \end{bmatrix}. \end{aligned} \tag{5.15}$$

DRAFT

6 Implementation

This chapter describes the implementation of the software components of the turret. The components are implemented based on the design in chapter 5. The components covered in this chapter are those involved in the three main processes of the turret: detection, tracking and shooting. These components include the PID, the Kalman filter, the custom voltage-to-distance converter described in section 4.2.3 and the scheduling. The implementation will also cover the selection of the firmware used on the LEGO Mindstorms NXT.

Someone: Tjek consistency med kursive termer/variable/etc. Bruger vi det eller gør vi ikke?

6.1 Operating System

In this section the operating system in use for the project will be described, and there will be reasoned in the choice of operating system. Furthermore the scheduler running on the OS will specified.

6.1.1 nxtOSEK

There are several operating systems for the nxt platform with different advantages and disadvantages, a comparison of these can be seen on Team Hassenplug's NXT Programming Software page [18]. For this project it ended with two possible OS, RobotC and LEJOS OSEK, both of them had floating points which will make some calculations easier to handle. RobotC got data logging and LEJOS OSEK dont haveing the ability to datalog would be preferable as it would make it easyer to find trends in the data if it was off loaded, LEJOS OSEK is able to use bluetooth to communicate with a PC and send data that way, bluetooth can then be used for data logging. RobotC got a file system and a memory of 561 bytes it might make a difference but not one that we plan to use.

Scheduling

nxtOSEK got 2 types of scheduling shiped with the OS the first one is event driven were tasks are activated by an event, this way is not scheduleable in rts. The other is rate monitoring scheduling were the tasks are given a priority and a period in ms, when a tasks period is reached it will be put in the que, if it got a higher priority the tasks running will be preempted. Periods are static and is not changeable, the only way a task can get a higher priority is by priority inheritance, if two tasks got a shared resource the resource get a priority equal to the highest priority of the two tasks, when a tasks get the resurce they also get the priority of the resource, this way a low priority task cant block a high priority one. Tasks are given priority based on their period the tasks with the lowest period is given the highest priority and so on. Rate monitoring is used in RTS systems as it is easy to implement and a guarantee can be given that is will never miss a deadline if the actual cost if found.

6.2 Detection

Alex:
better title

```
#define CENTER  0
#define RIGHT_1 1
#define RIGHT_2 2
#define RIGHT_3 3
#define RIGHT_4 4
#define UNKNOWN 5
```

Code 6.1: Areas outside and inside sensors field of view

The defines in code 6.1 shows which areas is implemented, from section 5.3 “0”, “1”, “2” and “3” the same exists in negative. The “4” and “5” is to manage the zone outside the sensors field of view. The difference between “4” and “5” is when it is “4” the train has been inside the sensors field of view whereas “5” the train has never been inside the field of view. This difference is needed by the Kalman filter cause else the first thing that will happen when the program is run, is that the Kalman filter will try to find the train by giving the PID a degree to sart rotating the tower. The Kalman filter need to know that we have never seen the train and therefor it should not start to look for it.

```
if(counter < 10 && prev != UNKNOWN)
        {
            counter++;
            prev = UNKNOWN;
        }
```

Code 6.2: Start Kalman

In code 6.2 a counter is used for the purpose of filtering out noise before starting kalman. This makes sure that the Kalman filter is not stated because the sensors get one reading which indicates something is in the field of view. At least “10” readings where a target is inside the field of view is needed before we trust that the sensors have in fact spotted the target and not made some faulty reading.

6.3 PID

The main PID function can be seen in code 6.3. The function takes 4 variables as input: the first two being the target position and the current position of the turret, and the final two being the integrale and the last error. The latter two are both variables used to store data from the last run of the PID. They are not stored in the function itself because that would cause a problem with the PID running multiple motors.

The PID utilizes four defines that are located in the PID.h file. These defines are K_p, K_i, K_d and MotorStartPower. The first three are coefficients for the PID, as explained in section 5.2, and MotorStartPower is the power required for the motor to start.

The S32 data type was chosen for most of the variables as it was sufficiently large, allowing for the values to run several thousand shots. A double was chosen as the data type for the calculation as the used coefficients are small which means the result is not always bigger than 1, a situation which may lead to a rounding error if an integer was used.

```
S32 PID(S32 target, S32 current, S32 *integrale, S32 *lastError)
{
  S32 error = target - current;
  S32 derivative = error - *lastError;
  *integrale = *integrale + error;
  double speed = error * Kp + Ki*(*integrale) + Kd*derivative;
  *lastError = error;

  return (speed > 0) ?
        (S32)(speed + MotorStartPower) :
        (speed < 0) ?
           (S32)(speed - MotorStartPower) :
            (S32)speed;
}
```

Code 6.3: PID code

Code 6.4 shows the wrapper for the PID function. The function has two arrays with a size of four. These are used to told the *integrale* and *lasterror* variables. The wrapper takes two variables as input: the first is the target position, of the motor, and the second is the motor number. The motor number can change depending on which port in the NXT the motor has been connected to.

The two arrays hold the integrale and last error that are used by the PID function. Because the NXT has four output ports the length of the array is four. The port signature is translated to integers in the range 0 to 3. The wrapper then retrieves the current position of the motor, calls the PID and finally, it sets the speed of the motor to the one returned by the PID function in code 6.3 and returns it.

Kenneth/Alexander: Update med tekst om lasttarget og flag

```
S32 MotorPID(S32 target, U8 motor, U8 flag)
    {
        static S32 getSpeed = 0;
        static S32 integrale[] = {0, 0, 0, 0};
        static S32 lastError[] = {0, 0, 0, 0};
        static S32 lastTarget[] = {0, 0, 0, 0};

        if(lastTarget[motor] != target)
        {
          integrale[motor] = 0;
          lastError[motor] = 0;
          lastTarget[motor] = target;
        }

      S32 current = nxt_motor_get_count(motor);
        if( flag || !((getSpeed <= 80 && getSpeed >= -80) && (current - target) <= 4
    && (current - target) >= -4)){
```

```
17          getSpeed = PID(target, current, &integrale[motor], &lastError[motor]);
18          nxt_motor_set_speed(motor, getSpeed, (!getSpeed));
19
20        }
21        else
22        {
23            nxt_motor_set_speed(motor, 0, 1);
24        }
25      return getSpeed;
26    }
```

Code 6.4: PID wrapper

The variables K_p, K_i and K_d, as seen in code 6.5, were found using the Ziegler–Nichols method and then tuned manually. K_i is much smaller than the other two because the I value in the PID is the sum of all the errors that the system has seen, and with a sample rate of 20 milliseconds it will sum up fast. MotorStartPower is the minimum power that is required for the motors to start. In this case the MotorStartPower is set to 80 which is the required power level when the motor is running under load.

```
1 #define Kp 0.075              // Kp
2 #define Kd 0.0375             // Kd
3 #define Ki 0.0000000075       // Ki
4 #define MotorStartPower 80
5
6 extern S32 PID(S32, S32, S32*, S32*);
7 extern S32 MotorPID(S32 target, U8 motor, U8 flag);
```

Code 6.5: PID.h

6.4 Shooting

This section covers the implementation of the software components involved in the turret's shooting process. Some of these components rely on the PID, whose implementation was explained in section 6.3, for their functionality.

6.4.1 Cock function

Before the turret can shoot at the target, it needs to prime its weapon system. This is done through the *cock* function which, when called, sets the *WSRotation* variable such that the weapon system will rotate 150 degrees into a "ready to fire" position. This is done by using the PID to make the rotating rods turn to the given position such that it neither overshoots nor undershoots the "ready to fire position". The *cock* function and the use of the *WSRotation* variable can be seen in code 6.6 and code 6.7.

```
1 void cock()
2 {
3   WSRotation = shotsfired * 300 + 150;
```

```
}
```

Code 6.6: Cock function

```
TASK(Task3)
{
  MotorPID(WSRotation,WSMOTOR1,  0);
  MotorPID(WSRotation,WSMOTOR2,  0);
    TerminateTask();
}
```

Code 6.7: WSRotation variable used in a PID function call

6.4.2 Fire function

The turret fires a projectile by calling the *fire* function which can be seen in code 6.8. The *fire* function also uses the *WSRotation* variable in the same manner as the *cock* function. The function also increments the *shotsfired* variable and returns its new value when called.

```
S32 fire() {
    ++shotsfired;
    WSRotation = shotsfired * 300 + 150;
    return shotsfired;
}
```

Code 6.8: Fire function

As seen in code 6.9 the turret may not shoot before the target has been seen as stated in the first if check, when this check is performed the tower is made ready to fire by calling “cock()” which rotate the weapon system 150 degrees making it ready. The PID is used to make the rotating rods turn to the right degree so as not to overshoot or undershoot the position for ready to shoot.Before shooting the tower needs to track the train for a while this is done by the first check in the second if sentence, the second premises makes sure that we will not fire a second time and the third premises makes sure we do not shoot unless the train is close to the middle of the combined sensors field of view. The “fire()” function both has a counter in it which is returned so that the “shotflag” is set and makes sure that we do not shoot more than one time, and it also send information to the PID to make sure that the rotating rods is rotated.

```
if (targetSeenFlag) {
    cock();
    if (nxt_motor_get_count(NXT_PORT_A) > 45 && !shotFlag && motor_in_range(3)) {
        shotFlag = fire();
        resetCounter = 1;
        }
    }
```

Code 6.9: Ready to fire

6.5 Kalman filter

The matrices and vectors used in the kalman filter are stored as two-dimensional arrays, of type *double*. The state vector, its covariance matrix, and the state transition matrix are shown in code 6.10. Compared to the values specified in eq. (5.14), the initial position has been set to -40, and the initial velocity set to 70. The purpose of the higher initial speed, is to make the filter slightly biased towards the higher velocities, as the turrets limited range of motion means the turret might have a very short time period to determine and adjust to a high velocity. When the target is moving at a low velocity, the initial velocity, and its variance, are inaccurate, but the turret has longer time to adjust these inaccurate values compared to the high velocities. The change in the initial position is to prevent the kalman filter from estimating estimating a very low velocity, which might happen due to the high polling rate of the sensors, which leads to the target being observed in the same zone several times before the turret start rotation, which is interpreted by the kalman filter, as if target standing still. Therefore the filter starts lowering the expected velocity until the target enters the next sensor zone, or the turret starts rotating, while still observing the target in the same sensor zone.

```
// Global variables
...
#define VK 8.0
...
double x[2][1] = {{-40.0}, {70.0}};
double P[2][2] = {{0.4, 0.0}, {0.0, 30.0}};
U32 last = 0;
...

void kalman(double zn){
    /* Constants and matrices used for kalman calculations */
    static double R = VK;
    static double I[2][2] = {{1.0, 0.0}, {0.0, 1.0}};
    U32 current = systick_get_ms();
    double t = (last == 0) ? 0.1 : (double)(current - last) / 1000.0;
    last = current;
    double h[1][2] = {{1.0, 0.0}};
    double hT[2][1] = {{1.0}, {0.0}};
    double a[2][2] = {{1.0, t}, {0.0, 1.0}};
    double aT[2][2] = {{1.0, 0.0}, {t, 1.0}};
    double K[2][1] = {{0.0}, {0.0}};

    /* Temporary variables used for storing intermediate results */
    double y[2][1] = {{0.0}, {0.0}};
    double temp[1][2] = {{0.0, 0.0}};
    double temp2v = 0.0;
    double (*temp2)[1] = (double ( *)[1])(&temp2v);
    double temp3[2][2] = {{0.0, 0.0}, {0.0, 0.0}};
    ...
}
```

Code 6.10: Matrices used in the kalman filter.

The state covariance matrix, and the state transition matrix are implemented as seen in equations 5.15 and 5.13 on line 2 and 4, respectively. The time between iterations Δt, is calculated in seconds on line 3, where the first iteration sets the time as 10 milliseconds.

The equations for the kalman filter uses several different matrix operations, but as these operations follow the same structure, they will not be explained in detail individually, but the general structure explained based on the function for matrix multiplication shown in code 6.11.

```
1  U8 matrixMultiply(U8 rowsA, U8 colsA, U8 rowsB, U8 colsB, double a[rowsA][colsA],
       double b[rowsB][colsB], double out[rowsA][colsB])
2  {
3    if(colsA != rowsB)
4      return -1;
5
6    double tempA[rowsA][colsA];
7    double tempB[rowsB][colsB];
8    matrixCopy(rowsA, colsA, a, tempA);
9    matrixCopy(rowsB, colsB, b, tempB);
10   double sum = 0;
11   U8 i, j, k;
12   for(i = 0; i < rowsA; i++){
13     for(j = 0; j < colsB; j++){
14       for(k = 0; k < colsA; k++){
15         sum += tempA[i][k]*tempB[k][j];
16       }
17       out[i][j] = sum;
18       sum = 0;
19     }
20   }
21   return 1;
22 }
```

Code 6.11: The matrix multiplication function

The first four parameters are the sizes of the two matrices, given as the next two parameters, with a return matrix as the last parameter. In operations that require the operands to be in the same dimensions, the function only takes one pair of dimensions. The function returns a value indicating whether the operations was successful. First the two operands are copied into two temporary arrays, seen in line 5-8, which are then used in the computation. By doing this, it is possible to save the results of an operation in one of the given operands, with no risk of overriding information. Giving the return matrix as a parameter is necessary in C, as it is not possible to return a reference type from a function.

The predict part of the kalman filter, equations 5.10 and 5.11, are implemented as seen in code 6.12.

```
1  void kalman(double zn){
2      ...
3      // Predict the current state
```

```
4   matrixMultiply(2,2,2,1, a, x, x);
5
6   // Predict the covariance in the current state
7   matrixMultiply(2,2,2,2, a, P, P);
8   matrixMultiply(2,2,2,2, P, aT, P);
9     ...
10 }
```

Code 6.12: The predict step of the kalman filter

Code 6.13 shows update step of the kalman filter. In the calculations, some temprary variables are used for intermediate calculations. The multiplication on line 4 results in a 1-by-1 matrix, here saved in *temp2*. In order to make operations with this matrix containing only a single variable easier, the variable *temp2v* is used, which contains the element from the *temp2* matrix as a simpler *double* datatype.

```
1  void kalman(double zn){
2      ...
3      /* Calculate kalman gain */
4      matrixMultiply(1, 2, 2, 2, h, P, temp);
5      matrixMultiply(1, 2, 2, 1, temp, hT, temp2);
6      temp2v += R;
7      temp2v = 1.0 / temp2v;
8      matrixMultiply(2, 2, 2, 1, P, hT, K);
9      matrixScale(2, 1, K, temp2v, K);
10
11     /* Update state estimate */
12     matrixMultiply(1, 2, 2, 1, h, x, temp2);
13     zn -= temp2v;
14     matrixScale(2, 1, K, zn, y);
15     matrixAdd(2, 1, x, y, x);
16
17     /* Update state covariance */
18     matrixMultiply(2, 1, 1, 2, K, h, temp3);
19     matrixSubtract(2, 2, I, temp3, temp3);
20     matrixMultiply(2, 2, 2, 2, temp3, P, P);
21     ...
22 }
```

Code 6.13: The update step of the kalman filter

6.6 RTS

6.6.1 Task structure

Oil file

6.6.2 Model

If a more accurate check of a set of tasks schedulability, then the normal tests, is needed a model can be contructed as it is not limited to a single uni-processor as the normal tests. To

make a model a modeling tool have to be used for this project Uppaal is used as it is made in collaboration with Aalborg Univerisity and it was easy to get help if there were problems. The problem with a model is that there is no method to mathematical prove that a given model models the given problem,
Figure 6.1 shows the task model made with Uppaal. The model has four states: idle, ready, running, and error. The error state does not exist in the CPU, however, it is required to show that a task missed a deadline and has entered an undefined state. Outside a hard real-time system a missed deadline can be handled by incrementing a counter, and as long as the amount of missed deadlines do not increase too much, nothing will happen. Ready is the initial state which a task will enter when it is released. From this state a task can enter either the error state, if it misses its deadline, or running if the task is allowed to run. When a task is done running it enters the idle state, and from here it can only enter the ready state. The last state available to a task is the running state which can enter all three other states. It will enter the error state if it misses a deadline or isn't finished yet, it will enter the ready state if a higher priority tasks comes along, and it will enter the idle state if it has finished running.

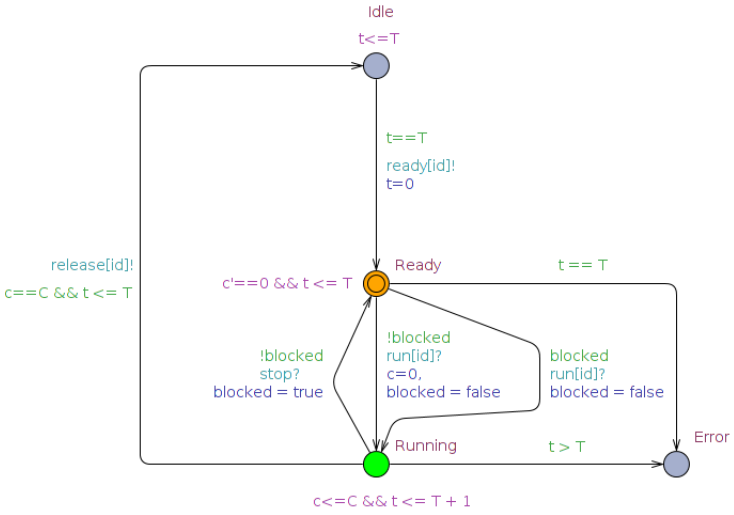

Figure 6.1: *Uppaal Task*

Figure 6.2 show the scheduler model, the scheduler got seven states, *free*, *switchingFree*, *Occ* , *switchingRunning* and three dummy states that it is not allow to stay in but is used as transit states. *switchingFree* and *switchingRunning* both handle context switching either in the case of from the free state and the running state from these states it is only allowed to go to the *Occ* state. When no tasks are running the model is in the free state, this state can only go the the *switchingFree* when a task is released. When a task is running the scheduler is in the running state in this state there is two things can happen either the task is allowed to run until the end or a new task is released. if the task is allowed to finish it can either run the next task or wait for a new task depending on if there is a task waiting. if a new task is released, the scheduler will put it in the que if it got a lower priority it will be ignored

and the current task will keep running if it got a higher priority the scheduler will enter *switchingRunning* and switch to the new task.

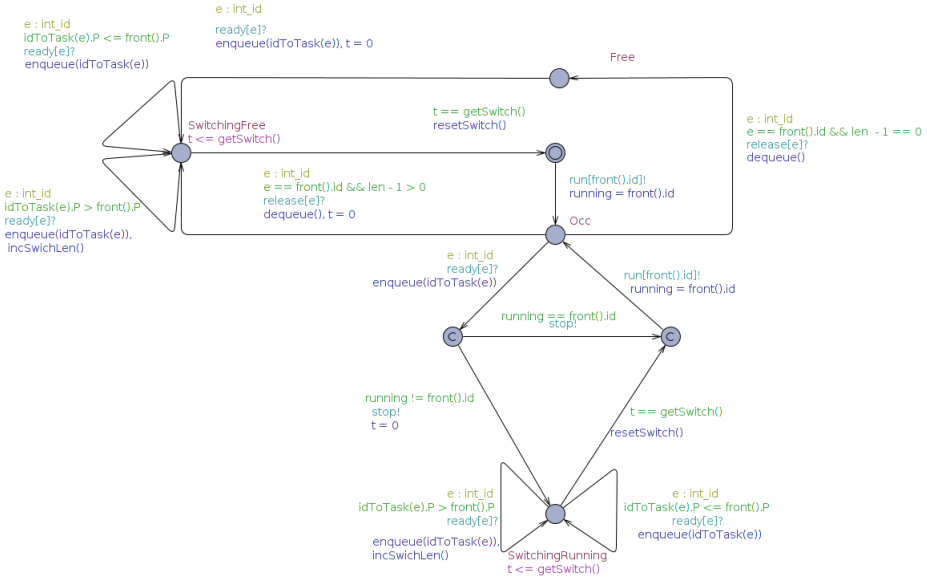

Figure 6.2: *Uppaal Scheduler*

7 Test

This chapter describes the tests of the completed turret, with all its components implemented as seen in chapter 6, and the real-time system components used to determine the schedulability and worst-case response time of the system. The tests are conducted in order to gain information regarding the performance of the turret such that it can be determined whether it fulfills the requirements listed in section 2.4.

7.1 Turret

The finished turret, designed in section 5.1 and implemented in chapter 6, is tested to determine if it meets the requirements in section 2.4. Due to the nature of the turret, unit tests are not conducted as this would require the creation of a test environment capable of simulating the moving target such that the turret knows when its projectiles hit the target. As such the tests have to be performed manually with hit detection reliant on visual and auditory feedback.

All the requirements share the fact that a functioning turret, capable of hitting a moving target, would fulfill all of the requirements as they are essential to the successful execution of the task. If the turret manages to hit the moving target it means it has detected the target by monitoring the observable area and detecting movement, predicted the trajectory of the target and hit the target before it left the range of the turret.

The turret tests will be performed using the same setup as the one used during the test of the infrared sensors, see section 4.2.1. The sensors are mounted as seen in figure 5.1. The tracks upon which the LEGO train travels is 160 centimeters long and 70 centimeters wide. The entire setup can be seen in figure 7.1.

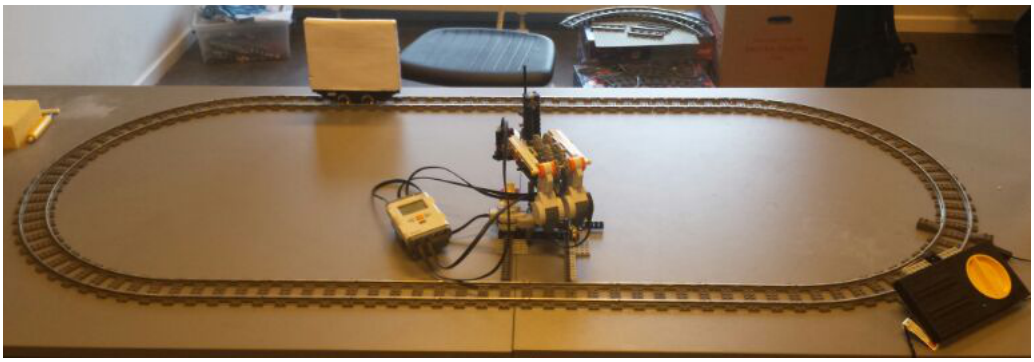

Figure 7.1: *The turret test setup*

The speed of the LEGO train can be controlled using the control panel seen in the lower right of figure 7.1. The speed can be changed in intervals with a total of six settings available, one being the slowest and six being the fastest. Only the first three settings are

used for tests as anything above that causes the train to tip over when going through the corners on the track. Testing the turret's ability to hit targets moving at different speeds verifies that it is capable of analyzing the target's movement and reacting to it while also ensuring the turret performs it calculations within a limited time frame, allowing it to hit a target moving at a relatively high speed.

SPEED 1

This test was performed at the slowest possible speed. The train did ten laps of the track, moving past the turret. The results of the test are noted in table 7.1.

Hit	Miss Direction
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—

Table 7.1: *Turret test at speed 1*

All the fired projectiles hit the target at speed 1.

SPEED 2

This test was performed at speed 2 which is approximately twice as fast as speed 1. The train lapped the track ten times while the turret was activated. The results of the test are noted in table 7.2.

DRAFT

Hit	Miss Direction
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—

Table 7.2: *Turret test at speed 2*

All the fired projectiles hit the target at speed 2.

SPEED 3

This test was performed at speed 3 which is ...

Mathias: Færdiggør og skriv mere tekst

Hit	Miss Direction
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
✓	—
×	←

Table 7.3: *Turret test at speed 3*

7.2 RTS

The finished system has 3 tasks with worst-case execution time, periods and priority as seen in table 7.4.

Kenneth: Tjek tallene når vi får dem

DRAFT

Task	WCET	Period	Priority
1	1	5	3
2	5	10	2
3	10	15	1

Table 7.4: *System tasks (WCET = Worst-case Execution Time)*

The second task runs the implemented Kalman filter, see **??**. The task is used to track the target and its run time is limited by the sensors. The third task controls the motors used for the weapon system by running the PID, explained in section 6.3, on the shared variable WSRotation which is set by the fire and cock functions. Both functions can write to their resource, the WSRotation variable, and the second task is reading it as seen in figure 7.2. There are only two functions that can write to the variable and both of them set their resource based on their shared variable shotsfired. This means that if the cock function is called before the fire function, cock will be overridden and it will continue with the input from fire. If the fire function is called right before the cock function then shotsfired will be incremented by the fire function and then the cock function will override the value of WSRotation. The period of the second task is decided by the PID function which has been designed to run with a period of 20 milliseconds.

Kenneth:
Add first task

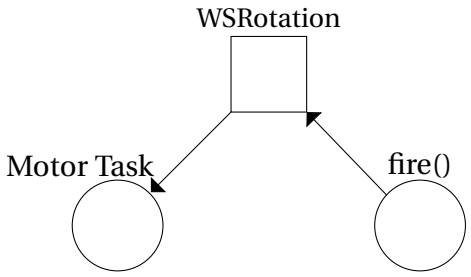

Figure 7.2: *Shared resource for the weapon system*

7.2.1 Utilisation Analysis

To check if the program is scheduable a utilisations analysis is made, as this is a fast way of learning that it is schedulable or if more test have too be made. The utilizationbound is as seen in table 7.5

Kenneth:
skrive færdig når vi har værdier

Mathias:
Ret når færdigt

Task	WCET	Period	Utilization Bound
1	1	5	20%
2	5	10	50%
3	10	15	66,6%
Total			136,6%

Table 7.5: *Utilization Bound (WCET = Worst-case Execution Time)*

7.2.2 Worst-case Response Time Analysis

As the utilisation test failed the tasks might not be schedulable but to determine that a worst-case response time analysis is made. The results of this analysis can be seen in table 7.6. Task 1 got a worst-case response time of x ms as this is lower then it's period it will never miss a deadline,

Kenneth:
skriv om de andre tasks

Task	WCET	Period	Priority	WCRT
1	1	5	3	5
2	5	10	2	5
3	10	15	1	5

Table 7.6: *Worst-case response time (WCET = Worst-case Execution Time, WCRT = Worst-case Response Time)*

7.2.3 Model check

DRAFT

8 Evaluation

DRAFT

9 Draft-only: TODOs

FINAL: CHANGE MARGIN BACK! at Internal conversion table data (page 1, line 17)
Alex: kilde at Internal conversion table data (page 3, line 2)
kasper: HUSK: hvis vi ikke når at indføre variabel retning, så skal dette fjernes at Internal conversion table data (page 4, line 4)
kasper: Dobbelttjek polynomial at Internal conversion table data (page 5, line 11)
Alex: Caption til billede at Internal conversion table data (page 5, line 28)
Alex: kilde at Internal conversion table data (page 7, line 5)
Alex: kilde at Internal conversion table data (page 8, line 16)
Alex: kilde at Internal conversion table data (page 8, line 5)
Alex: kilde at Internal conversion table data (page 8, line 7)
Alex: kilde at Internal conversion table data (page 9, line 12)
Someone: kilde eller omformuler at Internal conversion table data (page 9, line 12)
Someone: talesprog at Internal conversion table data (page 9, line 14)
Alle: Er i enige med vigtigheden af de forskellige kriterier? at Internal conversion table data (page 12, line 18)
Lasse: Hurtig forklaring af hvor de forskellige scores kommer fra at Internal conversion table data (page 12, line 31)
Lasse: Vi skal have grafen for voltage reading af den anden sensor ind i bilag. at Internal conversion table data (page 21, line 131)
Someone: Vise beregning - how? at Internal conversion table data (page 21, line 133)
Lasse: Den her test skal nok tages om. at Internal conversion table data (page 22, line 148)
?: Hvad var årsagen til at vi testede dette? Grafen med ens punkter kan være et tilfælde + den er semi-fake (den sidste del er gentaget, da der blev testet i lidt for kort tid) at Internal conversion table data (page 23, line 158)
Mathias: Graf muligvis for bred at Internal conversion table data (page 30, line 30)
Alex: noget omkring at bruge de 300-700 mm gør at tårnet helst skal stå i en bestemt distance fra track at Internal conversion table data (page 31, line 6)
kasper: præcis hastighed på polling + overvej at finde den præcise hastighed som skal til for at misse target i innerzonen at Internal conversion table data (page 31, line 11)
kasper: er det egentlig inaccurate? at Internal conversion table data (page 31, line 14)
Someone: Tjek consistency med kursive termer/variable/etc. Bruger vi det eller gør vi ikke? at Internal conversion table data (page 39, line 3)
Alex: better title at Internal conversion table data (page 40, line 1)
Kenneth/Alexander: Update med tekst om lasttarget og flag at Internal conversion table data (page 41, line 27)
Mathias: Færdiggør og skriv mere tekst at Internal conversion table data (page 51, line 27)
Kenneth: Tjek tallene når vi får dem at Internal conversion table data (page 51, line 2)
Kenneth: Add first task at Internal conversion table data (page 52, line 8)
Kenneth: skrive færdig når vi har værdier at Internal conversion table data (page 52, line 12)
Mathias: Ret når færdigt at Internal conversion table data (page 52, line 12)

DRAFT

Kenneth: skriv om de andre tasks at Internal conversion table data (page 53, line 16)

List of Figures

List of Tables

Bibliography

[1] Michael J. Miller. The Birth of the Microprocessor.
http://forwardthinking.pcmag.com/none/330368-the-birth-of-the-microprocessor, 2014.
Accessed September 16th, 2015.

[2] University of Notre Dame. Embedded System Design Group.
http://www3.nd.edu/~codes.
Accessed September 16th, 2015.

[3] Armorgames. Armorgames.com's list of Tower Defense games sorted by most plays.
http://armorgames.com/category/tower-defense-games?sort=plays&page=1, 2015.
Accessed November 25th, 2015.

[4] AGames. Simple Tower Defense.
http://agames.cc/strategy/simple-tower-defense/#tab5, 2015.
Accessed November 26th, 2015.

[5] Arduino. Arduino Uno.
https://www.arduino.cc/en/Main/ArduinoBoardUno#techspecs, 2015.
Accessed November 24th, 2015.

[6] LEGO Group. LEGO MINDSTORMS User Guide.
http://lego.brandls.info/ebooks/8547_ms_user_guide.pdf, 2009.
Accessed December 1st, 2015.

[7] RASPBERRY PI FOUNDATION. RASPBERRY PI MODEL SPECIFICATIONS.
https://www.raspberrypi.org/documentation/hardware/raspberrypi/models/specs.md, 2014.
Accessed December 2nd, 2015.

[8] LEGO. LEGO NXT Ultrasonic Sensor.
http://shop.lego.com/en-US/Ultrasonic-Sensor-9846, 2015.
Accessed September 10th, 2015.

[9] mindsensors. High Precision Medium Range Infrared distance sensor for NXT or EV3.
http://www.mindsensors.com/ev3-and-nxt/112-high-precision-medium-range-infrared-distance-sensor-for-nxt-or-ev3, 2015.
Accessed September 15th, 2015.

[10] mindsensors. DIST-Nx v2 User Guide.
http://mindsensors.ddns.net/index.php?module=documents&JAS_DocumentManager_op=downloadFile&JAS_File_id=895, 2011.
Accessed September 15th, 2015.

[11] LEGO. LEGO NXT Sensor Specifications.
http://www.cs.tau.ac.il/~eranhaba/SMLAB/Preliminaries/9797_LME_UserGuide_US_low.pdf, 2015.
Accessed September 10th, 2015.

[12] Claudia, Thomas. Ultrasonic Sensor.
http://www.tik.ee.ethz.ch/mindstorms/sa_nxt/index.php?page=tests_us, 2006.
Accessed December 1st, 2015.

[13] R. Nave. Reflection of Sound.
http://hyperphysics.phy-astr.gsu.edu/hbase/sound/reflec.html.
Accessed December 4th, 2015.

[14] mindsensors. DIST-Nx v2 User Guide.
http://www.robotshop.com/media/files/pdf/DIST-Nx-v20-User-Guide.pdf, 2015.
Accessed September 15th, 2015.

[15] Sharp. GP2Y0A21YK Optoelectronic Device.
http://www.sharpsma.com/webfm_send/1208, 2005.
Accessed December 1st, 2015.

[16] Philippe Hurbain. NXT® motor internals.
http://www.philohome.com/nxtmotor/nxtmotor.html, 2015.
Accessed October 13th, 2015.

[17] Kiam Heong Ang, Gregory Chong and Yun Li. PID Control System Analysis, Design, and Technology.
http://eprints.gla.ac.uk/3817/1/IEEE3.pdf, 2007.
Accessed October 14th, 2015.

[18] Steve Hassenplug . NXT Programming Software.
http://www.teamhassenplug.org/NXT/NXTSoftware.html.
Accessed December 11th, 2015.

A Internal conversion table data

Data used for the DIST-Nx-v2 medium range sensors internal conversion table.

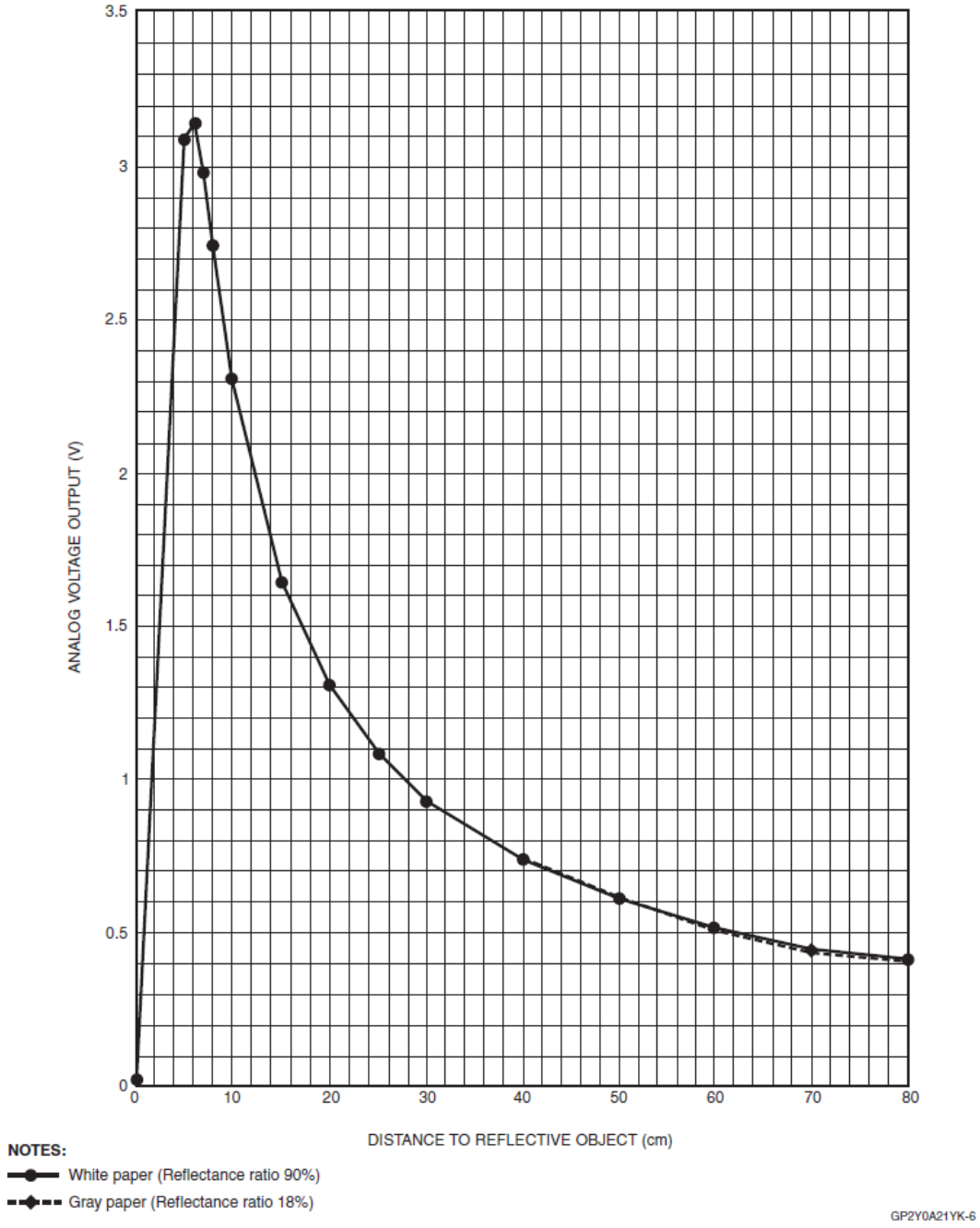

Figure A.1: *Voltage to distance data for the Sharp GP2Y0A021YK sensor module [15].*