Improving Navigation and Mapping with Arduino robot

Jørund Øvsttun Amsen

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem description

The LEGO-robot project has been ongoing since 2004 and has been the subject of several master thesis and projects. During this semester it consisted of four robots and a server application. The robots all had motion sensors, wheels and four distance sensors rotating to cover 360 degrees. Using this, they were capable of recieving commands with heading and distance from the server and return information about distance from each sensor and its own estimated position. The goal was to navigate and map an unknown labyrinth. The server functions as the brain, issuing commands and communicating with all the robots involved, using an A* shortest path algorithm as well as optimally coordinating the different robots based on position to explore the unknown areas.

Of these robots the Arduino is the newest addition and the focus of this paper. The Arduino as with the other robots use a dead-reckoning method of position estimation using an IMU, encoder and magnetometer. The robot's ability to estimate and navigate according to the real world and server commands was found to be lacking, and caused mapping over large areas to be skewed and warped.

The purpose of this project was to focus on the two main points of accuracy; the navigation of the robot based on its position estimation, and the acquiring of this position estimation through sensor usage. The following tasks were agreed upon.

1. Aquire an overview of sensor capabilities and accuracies.
2. Improve sensors if possible.
3. Improve hardware faults.
4. Improve navigation algorithm.

Preface

This Master thesis is the culmination of 5 years at Norwegian University of Science and Technology (NTNU). It is a project covering 20 weeks of work on a subject of my own choosing. It follows in the footsteps of students on a project that has been on-going at NTNU since 2004. It has been an enriching and exciting experience to work on a project so many different students have put their mark on.

The work on this thesis has been individual, however I've had the pleasure of working with other students on the same project, creating a varied and interesting environment. During this project I've learned a lot about a large variety of engineering subjects, both hardware and software and how many small parts come together to make a large system.

Thanks to Kristian Lien and Lars Marius Strande, whom created a lively and enjoyable working area.

Thanks to Jan Leistad for great work on the plate, Stefano Bertelli and the other technical staff at D040 for their patience and help.

A big thanks to Tor Onshus, who has given more support than I ever expected, and has been a source of both knowledge and motivation every week.

[illegible]

Jørund Amsen

Trondheim, 2017-11-06

Summary and Conclusions

During this Master Thesis work has been made on the Arduino robot to improve its general capabilities. Work has also been made to further explore the possibilities of the sensors used on all robots in the system.

A hardware fault with the wheel armature cogs on the left side of the robot was discovered. This problem lead to the robot faultily estimating its position and be unable to drive in a straight line. This caused the robot to often drive into walls and not give reliable information about the mapping due to its corrupted position estimation. This problem was solved using a metal plate mounted on the armature to improve its stability and the tightness of each cog on the left side, so the cogs could not slip on each other. This solution completely removed the aforementioned problem and the robot now registers the correct wheel rotation. We do not expect a similar problem to occur on the right side as the work on the armature on that side is significantly better.

Each sensor was searched for faults and tested for accuracy. The encoders were tested mainly by the wheel factor; the number converting encoder "ticks" to distance moved. This number was found to be adequate, however caution must be made as to the possibility of differing wheel factors depending on the surface. The surface tested on was room B333, "Slangelabben" at NTNU.

The gyroscope was tested during rotation and compared to the encoder and the real-world value using the OptiTrack reflector-camera system. The gyroscope was found to be slightly off the real value each test. It was decided to poll for mean value more frequently to remove bias on the gyroscope on a more consistent basis. This seemed to improve the rotation estimation during our final tests.

The compass was also tested during rotation and found to be very lacking. The noise factor on the sensor proved too great to adequately filter out during rotation. Due to this, the compass values had a negative impact on the heading estimation, and was removed from the system. It is advised that more work be done to explore the possibilities of filtering the compass and use it to reset gyroscope off-set.

The accelerometer was found to give valuable, but very noisy data. Attempts were made at

filtering and using the data, however none proved usable enough to directly implement on the current position estimation. It was decided to not be included in the software. More work is advised as the data is valuable and could potentially be used to detect slippage of the wheels and crashes.

As the old navigation algorithm was not based on coordinates, but rather on distance and heading, it was decided that a new navigation algorithm be implemented in the robot. This algorithm is based on converting the heading and distance commands the server sends into specific coordinates, then use a similar controller to minimize the distance to the target. This controller is a two-step algorithm, which first rotates to the target, then drives forward while continuously controlling the heading towards the coordinates. A lot of focus was also put on smoother movements, as we witnessed a lot of overshoot on the robot both during rotation and forward movement due to the robot's mass. This new algorithm showed much greater results than expected and has almost eliminated the navigation error.

During the final testing with the other robots it was shown to work well even if it showed some expected faults. The navigation is smoother and more reliable than before, and the position estimation is adequate based on its current algorithm.

Oppsummering og konklusjoner

Under denne masteroppgaven er det gjort arbeid på Arduino-roboten for å forbedre dens generelle evner. Det har også blitt arbeidet for å undersøke mulighetene til sensorene som brukes på samntlige roboter i systemet. En maskinvarefeil med tannhjulene i hjulopphenget på venstre side av roboten ble oppdaget. Dette problemet førte til at roboten feilaktig estimerte sin posisjon og ikke klarte å kjøre i en rett linje. Dette førte til at roboten ofte kjørte inn i vegger og ikke ga pålitelig informasjon om kartlegging på grunn av sin korrupte posisjonsestimering. Dette problemet ble løst ved hjelp av en metallplate montert på opphenget for å forbedre stabiliteten og tettheten til hvert tannhjul på venstre side, slik at tannhjulene ikke kunne glippe på hverandre. Denne løsningen fjernet det tidligere nevnte problemet og roboten registrerer nå riktig hjulrotasjon. Vi forventer ikke at et lignende problem oppstår på høyre side, da arbeidet på opphenget på den siden er betydelig bedre. Hver sensor ble søkt etter feil og testet for nøyaktighet. Enkoderene ble testet hovedsakelig i forhold til hjulfaktoren; tallet som konverterer encoder "ticks" til avstand flyttet. Dette tallet ble funnet tilstrekkelig, men det må utvises forsiktighet om muligheten for forskjellige hjulfaktorer avhengig av overflaten. Overflaten som ble testet var rom B333, "Slangelabben" På NTNU.

Gyroskopet ble testet under rotasjon og sammenlignet med enkoderen og virkelig verdi ved hjelp av OptiTrack-reflektorkamera-systemet. Gyroskopet ble funnet å være litt unna den virkelige verdien hver test. Det ble besluttet å finne gjennomsnittlig verdi oftere for å fjerne bias på gyroskopet på en mer konsistent basis. Dette syntes å forbedre rotasjonsestimatet under de siste testene.

Kompasset ble også testet under rotasjon og funnet å være svært mangelfullt. Støyfaktoren på sensoren viste seg for stor til å filtrere tilstrekkelig ut under rotasjon. På grunn av dette hadde kompassverdiene en negativ innvirkning på kursestiamtet, og ble fjernet fra systemet. Det anbefales at det gjøres mer arbeid på å utforske mulighetene for å filtrere kompasset og bruke det for å tilbakestille gyroskopet sin off-set.

Akselerometeret ble funnet å gi verdifulle, men svært støyende data. Forsøk ble gjort på filtrering og bruk av dataene, men viste seg ikke brukbart nok til å implementere direkte på nåværende posisjonsestimering. Det ble besluttet å ikke inkludere akselerometeret i program-

varen. Mer arbeid er anbefalt da dataene er verdifulle og kan potensielt brukes til å oppdage glidning av hjulene og krasj.

Ettersom den gamle navigasjonsalgoritmen ikke var basert på koordinater, men heller på avstand og på kurs ble det bestemt at en ny navigasjonsalgoritme skulle bli implementert i roboten. Denne algoritmen er basert på å konvertere kurs- og avstandskommandoene serveren sender til koordinater, deretter bruke en lignende kontroller for å minimere avstanden til målet. Denne kontrolleren er en to-trinns algoritme, som først roterer til målet, og så kjører fremover mens den kontinuerlig regulerer kursen mot koordinatene. Det ble også lagt mye vekt på glattere bevegelser, da vi opplevde mye "overshoot" på roboten både under rotasjon og fremoverbevegelse på grunn av robotens masse. Denne nye algoritmen viste langt bedre resultater enn forventet, og har nesten eliminert navigasjonsfeilen.

Under den endelige testen med de andre robotene ble det vist å fungere godt, selv om det oppstod noen forventede feil. Navigasjonen er jevnere og mer pålitelig enn før, og posisjonsestimeringe er tilstrekkelig basert på den nåværende algoritmen.

Contents

Chapter 1

Introduction

1.1 Motivation

Autonomy is often viewed as the greatest achievement in robotics, and rightfully so. Robots are a steadily increasing part of our society, and with every innovation becomes less dependant on its user and thus often more effective in its work. They perform work in all sectors, from cutting the grass to aiding the elderly. The idea of self-navigating robots have been a steadily growing area of interest, with automatic cars and drones becoming a probable future. Mapping of areas where humans are unable to traverse, from the bottom of the ocean to large sewer systems are motivated by science as much as our own curiosity. Finally, the ability for cooperation between robots makes the sum of all parts larger than the parts themselves and open new possibilities as well as much more dynamic systems. This project combines all these areas and attempts to show the possibilities and versatility of cheap sensors, materials and software used well.

1.2 Earlier work

The LEGO robot project was originally started in 2004 at the Cybernetic Department, as an attempt at using cheap, simple sensors, actuators and material to map and navigate a labyrinth. Since then, the robot hardware, software and the server has gone through several iterations. More robots have been added to the project. Currently the robots consist of the AVR robot, NXT robot, EV3 robot and in 2016 the newest addition was the Arduino robot. The robots are named after their respective microcontrollers. Work has been done on both sensor-handling, navigation and map generation. Later the project focused on a shared server, capable of controlling all robots at the same time, and optimizing their search, exploring the possibilities, and improvements to such a system. The focus of these reports has been varied, and most attempt to add new functionality, or robots to the system. For a more in-depth overview of the projects, a summary can be found in Tusvik (2009).

1.3 Goals and limitations

The work on this project is primarily on the Arduino robot, its hardware and software. This does not include the software on the Bluetooth dongle attached to the robot. Our focus has been to improve existing functionalities with new and improved methods, of this sensor handling and navigation have been the main aspects, as well as continous improvements on the hardware and faults detected.

Chapter 2

Background and Theory

2.1 System Overview

The system as a whole today consist of four robots and a server software. The robots are built on different micro-controllers and different chassis, however they all contain the same sensors and are built to perform the same tasks. The software on each robot is supposed to be identical save the differences required by the micro-controllers and motors, and the server treats them identically. Each robot uses an array of motion sensors, wheels and IR-sensors to move, estimate position and map the surrounding area, continuously updating the server with its position and nearby surfaces. This information is then used by the server to generate a map and based on it, further command the robot's movement until the entire area is mapped. The robots function almost purely as slaves, without any autonomy in movement except anti-collision. They move based on commands given by the server containing relative heading change, and distance to move. The server is ran on a PC, and using Bluetooth dongles containing a simple software working as a middleman, communicate with the robots. The server uses an A* algorithm to estimate the best path for all robots, and the distance data combined with robot position to create a map of the surroundings.

Hardware and assembly For this paper, one robot has been the focus; the Arduino robot. It is so named because the microcontroller used is an Arduino Mega 2560 R3, using an ATmega2560 microcontroller. It is the newest addition to the team, created in 2016 by Andersen and Rødseth

(2016). As with the other robots, it is a differential drive robot, meaning it has two wheels capable of driving in separate speeds and directions. It also has a third point of contact, a small metal ball at the back. It is mounted by a sensor tower fitted with four IR distance sensors situated at 90 degree angle from each other, this sensor tower is fitted to a servo allowing it to rotate. The robot is also fitted with several sensors; a 6 degree of freedom IMU consisting of an accelerometer and gyroscope, a magnetometer and magnetic encoders fitted to the wheels. A Bluetooth dongle is used to communicate with the server. A DC motor is connected through cogs to each wheel. These cogs are due to the high minimum RPM of the motors and reduce the RPM of the wheels. These cogs are supported by an armature hand-glued to the chassis and is not very well fitted to the robot. The extension of the armature also causes the weight of the robot to be supported on the armature and the wheels to slightly bend in.

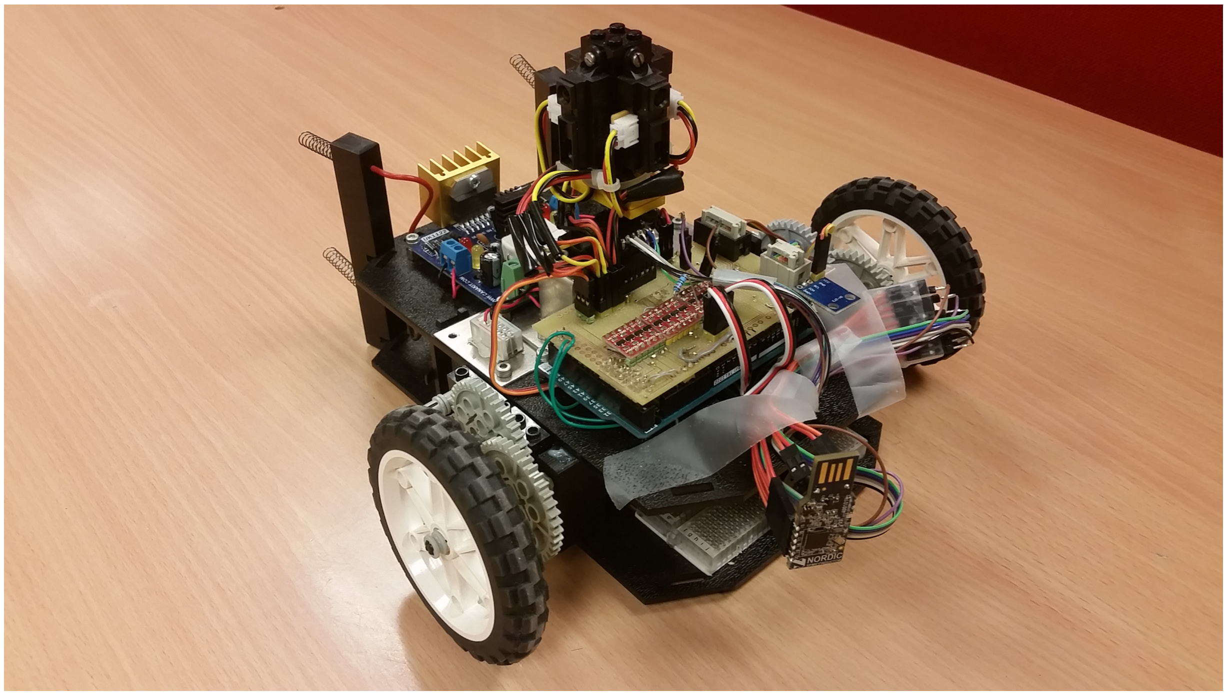
Figure 2.1: Arduino robot

Software and RTOS The brains of the robot is an ATmega 2560 microcontroller fitted to an arduino board. This microcontroller is programmed in C using Atmel Studio 7. The details of the software can be found in Ese (2016). However we mention it briefly here for the reader. The software on this controller serves several purposes. It contains software to gather data and communicate with the various sensors and actuators on the robot, as well as interfacing with the Bluetooth dongle. The software is built on a real-time operating system called FreeRTOS. This RTOS works using tasks and common real-time tools like semaphores, queues and mutexes. Tasks and real-time systems allows the robot to perform several tasks simultaneously. However they also pose a much greater possibility of complexity if not handled correctly. These tasks includes actions such as rotating the sensor tower, picking up sensor data, navigation, motor control, estimation of position, communication handling and more.

2.2 Status from earlier work

Problems with left motor At the end of Amsen (2016) it was noted that the performance of the left motor was lacking compared to the right motor. It was unable to keep the same speed as the right motor, resulting in a veering of the robot's movement to the left. Due to time constraints the problem was not properly diagnosed and it was assumed that something was wrong with the motor's ability to run at full speed. This problem also seemed to strongly affect the robots position estimate. Overall the performance of the robot due to this problem was very unsatisfactory.

Imprecise navigation At the start of this project an algorithm of polar navigation was used. This means the robot followed a two-step navigation style: First rotate to the correct angle, then drive in a straight line for the correct distance. This method however didn't take into consideration the actual end-point the server planned, and instead ran "blindly" on the two polar parameters. Meaning the robot was capable of knowing where it was, and that it hadn't reached the correct "point", but wouldn't correct. This in particular was a problem when the robot wasn't capable of running in a straight line for whatever reason. Because it didn't correct for errors in angle displacement during forward movement it would often miss the desired end-point.

Insufficient position estimation Based on the results from Amsen (2016) we expected the robot's position estimation to be quite satisfactory. However upon running some simple tests at the start of this project, it was shown that both the AVR and the Arduino robot were performing very poorly in their position estimation. Due to the importance in position estimation for both navigation and mapping it was concluded that the robot's sensors and algorithms should be fault checked and attempted improved.

Accelerometer The robot is outfitted with an accelerometer. This comes in combination with the gyroscope chip. In theory the accelerometer is an extremely useful sensor in regards to position estimation, especially in combination with the other sensors we use. However upon the start of this project, the sensor was not used at all, in fact it was set as deactivated by default. The reason for this was simple time constraint from earlier work. It is assumed that as much orthogonal sensor data as possible should be achieved, to further improve the robot's position estimation. No former work had been done to integrate the sensor into the system.

Compass The robot also carries a compass sensor also known as a magnetometer. Software work on this sensor was started, however not finished by the time this project started. An algorithm to find the compass bias, as well as merge the value with the gyro and encoder to more accurately find the rotation was planned by Ese (2016) and could be easily finished to improve the robot's performance.

Gyro The gyro, together with the encoder is one of the main sensors for this system's navigation and position estimation. It is highly weighted during the robot's turn phases, and is deactivated during forward motion. Due to its importance in position navigation, it should be prioritized when attempting to improve position estimation and navigation.

Encoder The main sensor for the robot's position and navigation system is the encoder. A Hall magnetic sensor encoder mounted on each wheel. The accuracy of the encoders are very important, as they are the robot's main method of navigating and position estimation. A constant converting each tick to a distance moved by the wheel is calculated in Ese (2016), but even small changes in this can yield inaccuracies. A more empirical approach may yield better results.

Master/Slave issues A problem encountered when the robot had issues with the left motor was a flaw in the master/slave motor controller. More specifically, the fact that the master would not regulate its speed if the slave couldn't keep up, which was the case when the left motor wasn't properly performing. Another issue with the PI controller showed that both motors always ran way above saturation levels, ultimately making the controller redundant.

2.3 Theoretical basis

2.3.1 Gyroscope

A gyroscopic sensor is a sensor used to measure rotational motion. The one used on the robot is a 3-axis gyroscope, capable of measuring angular velocities in all three degrees of freedom. However for our case we only use the z-axis, meaning we measure rotation parallel to the ground. Gyroscopes come mainly in two different types; optical and vibrating. The optical gyroscope uses the Sagnac effect and the difference in timing between two beams of light travelling the same path in different directions when the path is turning. The vibrating gyroscopes use the Coriolis effect on a resonating mass by measuring the Coriolis force applied perpendicularly to the mass when the system is rotating. Gyroscopes can be very accurate, even for a cheap price. However they are prone to bias, which is what we call the latent voltage when the gyro isn't moving at all. It manifests in a voltage, however this can also be seen as a constant error in the angular velocity. Bias is very dependant on several factors including both temperature and over time. This bias is important to measure and take into account when using the gyroscope. Especially seeing as we use the gyro by integrating the speed and acquiring total rotation. Integrating a bias causes what we call drift, and even a small bias will over time cause a very big error in the assumed rotation. Another error we encounter using gyros is white noise errors. Errors with zero mean. In most cases, white noise is not a problem, since they can simply be averaged out, but a problem arises when integrating white noise over a longer period of time. This is what we know as random walk, and could potentially accumulate more and more error. Figure 2.2 shows the effect of integrating white noise. Meaning even though gyroscopes can be very effective during short time spans we are guaranteed to see errors large enough to make any data unusable with enough time.

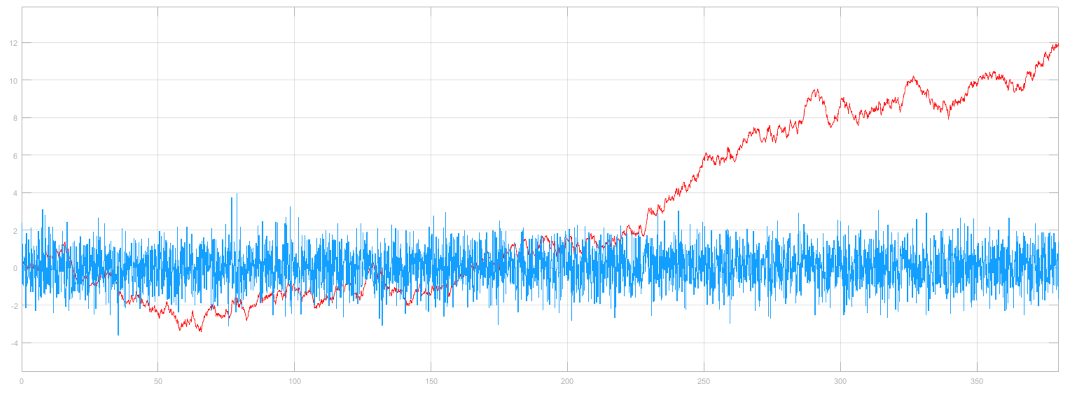

Figure 2.2: White noise and its integration. The red graph is often called random walk.

2.3.2 Compass

Compass, or as it is also known; a magnetometer is a sensor capable of detecting magnetic fields around it. It is most commonly used in for example mobile phones to detect the magnetic field of the earth and find north. It's built on the Hall effect; the electrons in the sensor are affected by the direction of Earth's magnetic field and by measuring this change we can calculate the compass heading. It is sensitive and picks up on all magnetic fields nearby, including metal beams in the floor and other components on the robot. This poses a problem as the magnetometer sums up all magnetic forces and calculates a vector in xyz direction representing the sum as opposed to treating them individually. This is why calibration of the compass is a vital part when using it. There are two main calibrations often used; orientation calibration and constant component calibration. In our case, we assume orientation calibration is negligible as long as the robot is running on a sufficiently level surface. However due to the density of components on the robot the second type should be taken care of. This is done by rotating the robot 360 degrees in place and recording the off-set values. Make sure this test is ran in different spots, to make sure no magnetic anomalies are present. This off-set can then be used until any new hardware is added. An algorithm for this was described in Ese (2016). Figure 2.3 shows the difference between calibrated compass measurements and uncalibrated. In this case, we can imagine some kind of constant magnetic field near the compass displacing what would otherwise show earth's magnetic field.

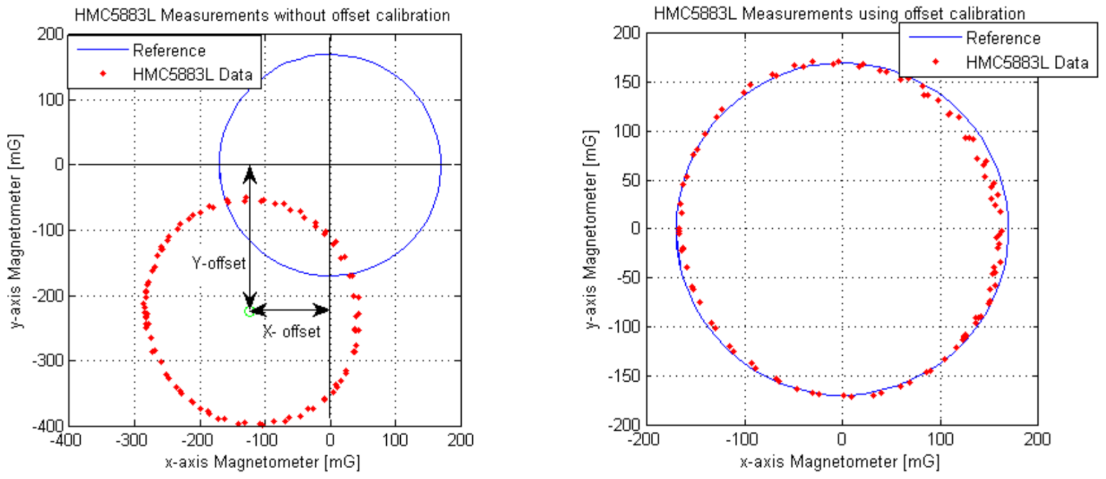

Figure 2.3: Before and after calibration Picture from Ese (2016)

2.3.3 Encoder

An encoder is in its most general term something that converts information from one format to another. In our case we are dealing with a magnetic rotary encoder. This means it uses magnetic forces and sensors to decode rotation into an electric signal. This is done using a disc fastened to the motor axle so it rotates in unison with the motor. This disc is divided into 4 negative and 4 positive poles. In close proximity to the disc, usually a few millimeters we place a Hall sensor, this is capable of detecting the difference between a negative and a positive pole nearby, giving a high or low signal depending on the pole nearest the sensor. Meaning for one rotation of the motor we would see 4 high and 4 low signals. Or as we call them in the code; 4 "ticks". Each tick corresponding to a distance moved with the wheel. It is important to note that since the ticks are only high or low, we cannot directly find out the direction of the rotation. This must be added by the software, with knowledge of what voltage is applied to the motor. Due to the gearing of the motors on the Arduino robot, every tick of the encoder yields very little motion and the reliability of the encoder is very sensitive to changes in the conversion constant from ticks to distance. Based on Ese (2016)'s work the constant was mathematically calculated, however improvements could be made empirically.

$$((pi * 81.6)/(198 * 40/16 * 40/8))/2 = 0.2072mm$$

Where 81.6 is the diameter of the wheel, 198 is number of ticks for one axle rotation. Then there are two gears with 40 teeth, one with 16 and one with 8 to gear down the rotation. This was due to the motor running about 1250 ticks/s or 150 revolutions/s.

An encoder solution is a robust method, however it is prone to slippage. This is when the wheel - and thus the magnetic disc - is turning, however it's slipping on the surface and not moving, or moving inconsitently. This is particularly common if the robot crashes with walls, and it has currently no method of detecting such an occurrence.

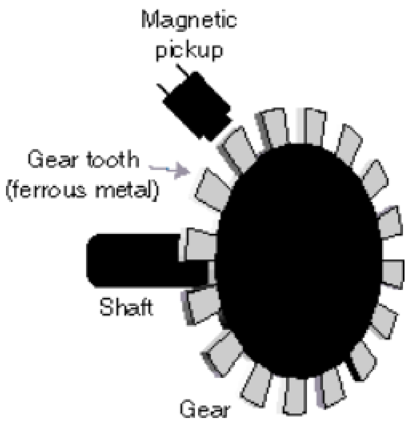

Figure 2.4: Magnetic rotary encoder. Illustration from NationalInstruments (2006)

2.3.4 Accelerometer

Most accelerometers are based on the principle of a tiny mass being suspended by small beams working as springs and some kind of fluid, usually air working as dampers. Thus creating a well known mass-spring-damper system. As the component undergoes acceleration, the mass is displaced due to the force working on it. This also explains the operational bandwidth. The two main ways of measuring this displacement are piezoelectric and capacitive material. The accelerometer is capable of measuring the acceleration in any of its 3 axis. It is important to note that it also measures the gravitational pull of the earth. Meaning we will always have a component from the earth. This is especially important when we are measuring in a vector not completely parallel to the ground. For example our robot running up a hill would show a constant acceleration component due to earth's gravity. This is also a factor if the component isn't fastened parallel to the robots point of contact with the surface. In addition to this, accelerometers are susceptible to bias, which manifest in a steady acceleration even when the component is lying still, this is not to be confused with earth's pull. This bias is due to the physical properties of the component, and is different for every chip, and can even change with time. Another common problem with accelerometer is that it picks up on all movement, even tiny ones. Vibration from the motor or uneven surface will both affect the accelerometer greatly, because even if the movements are small, the acceleration is big.

We mentioned earlier the issue with integrating sensor errors, in particular white noise, this problem in particular affects accelerometers. Double integration of the error causes an exponential increase in its effect on the system. To the point where mere seconds can yield several meters of error on the positioning of the robot. In fig 2.5 we can see the earlier plot of sensor error now double integrated. The earlier plot is now the thin line in the middle. This illustrates the problem of pure double integration when we try to use accelerometers for positioning.

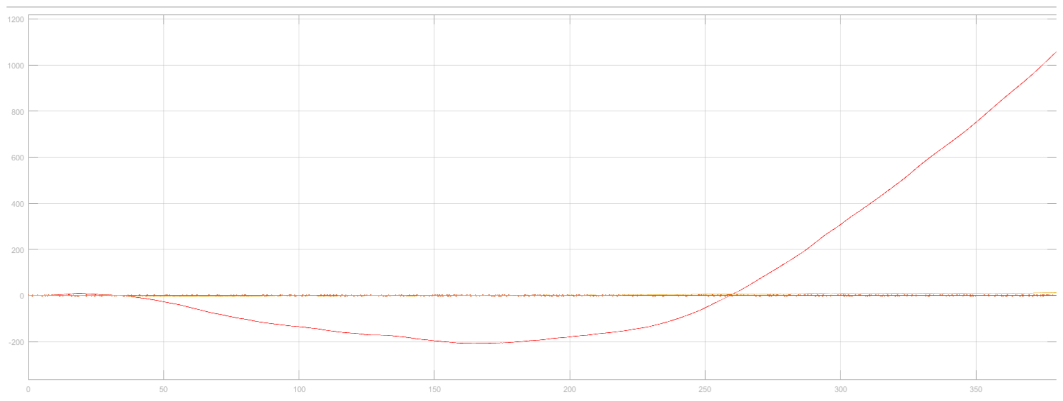

Figure 2.5: The integration of the earlier random walk.

2.4 Software

In the current software, we use several different methods to reduce the effects of the aforementioned sensor issues.

2.4.1 Gyroscope handling

Whenever the robot is not connected to the server for example during start-up, the robot will take 100 samples of the gyro, calculating a simple mean and applying this any time we read the sensor. This is to find an estimate for the bias error for the gyroscope. We also use a cut-off for the gyro. It was decided that the gyro would not be used when driving forward, but weighted heavily when in the rotate mode. Due to this, any gyro value under a certain threshold is ignored when calculating the rotation of the robot. This eliminates the effect of drift due to random walk integration when the robot isn't actually turning. Of course, it also means we will not pick up minor changes in rotation. Finally, during rotation the gyro is fused with the data from the encoder. At the time of writing, the gyro is weighted 85% of the final rotation prediction.

2.4.2 Compass handling

Similar to the gyro, we also calculate an off-set for the compass whenever the robot is not connected to the server. A similar method to the gyro of ignoring faulty high values of the compass was described in Ese (2016), by comparing before and after a forward movement section. However this was removed as the robot can't guarantee straightness in its movement. However a similar method to the gyro; where all compass rotational speeds that exceed the maximum rotation speed of the robot was implemented. Finally, a task that could be activated and deactivated when desirable is implemented to properly calibrate the compass in a 360 degree radius. This task is intended for use if new additions to the robot was made and is activated by defining the "COMPASS_CALIBRATE" variable. This will disable all other functionality, and acquire x-y compass off-set.

2.4.3 Encoder handling

Currently there is very little method of improving the handling of the encoder without the use of other sensors. The biggest problem we witness is when the wheels slip, either when the robot hits a wall of when the surface does not have enough friction for the sudden torque of the motor. Thus the current software is a simple interrupt counter connected to the signal on the encoder and the constant mentioned in the earlier section.

2.5 Navigation

There are many well-explored methods of navigation for a differential robot. Most of which use x-y coordinate goals. This means that a command from the server uses a point in the coordinate system to be the goal of the navigation. However at the start of this project a slightly different kind of navigation was used in the robot; polar navigation. This meant that the robot received only polar coordinates from the server, in the form of degrees and centimeters. The robot then used a two-step algorithm to first rotate to the correct heading, then drive in a straight line a certain distance.

The problem with this method was the inherent disregard for the actual goal. Instead of aiming for a specific point and a radius of accepted error, the robot attempted to first rotate correctly, and then have the correct distance from the starting point. This in theory sounds good if the robot can keep an almost zero error on the heading, and no heading error during forward movement. However the lack of heading control during forward movement shows its flaws in the physical world. An error of only 5 degrees (which at the start of this project was the error tolerance) would result in 4,5 cm if the distance driven was 50cm. The problem is, this 4,5cm error is an error the robot is aware of, but does not regulate for. On top of this, we often witnessed a change in heading during forward movement, further adding onto the end result. This also means that the further the robot drives, the bigger the error we can assume. It is also important to note that the distance the robot travels is from the starting point, and has nothing to do with the end-point or the distance the wheels have driven. This means that the robot will keep driving forward until it exits the distance radius from where it started, regardless of direction or distance from the actual target. A different method of control, in particular one that focuses on x-y control rather than polar coordinates, can drastically reduce the navigation error. Thus reducing the robots error in positioning to a matter of position estimation, rather than navigation.

Another problem was the motor controller described in Ese (2016) chapter 7.1.1. This controller was based around a master/slave PI controller between the two motors. This in theory sounds good, as even if the robot doesn't run as fast as possible, it still regulates the total distance the two wheels have driven. However an issue showed itself when the slave motor was the

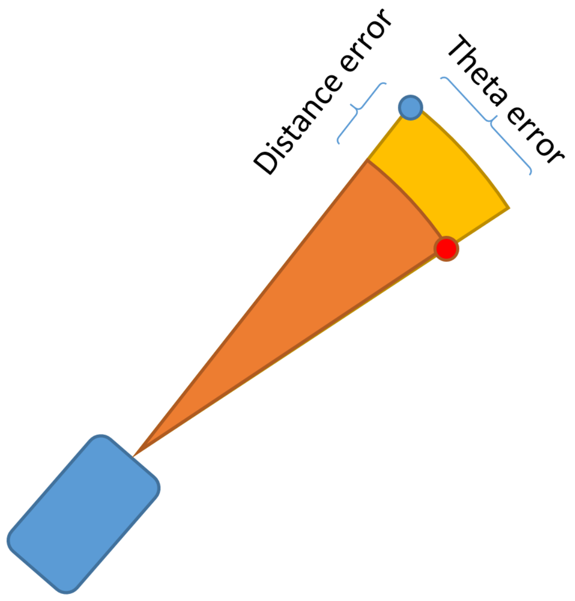

Figure 2.6: Visualisation of polar error

one not performing well enough. Since the master motor is not regulated based on the slave, but only the other way around, the entire motor controller was rendered obsolete, and incapable of regulating the speeds of the motors. A new method where both motors were equally reliant on the others speed/ticks was needed.

2.6 Position estimation

2.6.1 Current model

The position estimation is described in detail in Ese (2016). To sum it up; the robot uses mainly three sensors to calculate its position; the encoder, gyroscope and compass. It was shown that the encoder was inaccurate during turning phase due to a difference in wheel-factor constants. Instead of empirically estimating the different constants, a gyroscope was implemented to work as an orthogonal sensor input for the estimator. This means the two senors yield information about the same pose variable (rotation), but are independent of each other, and thus do not yield the same error components and can be used to eliminate errors. The gyroscope turned out to be a lot more effective during turning as it was independent of the surface and had no slippage and a simple percentage fuse between the two sensors was added. It was also noted that gyroscopes have drift and to counteract this a compass was added as well. The compass is used to find absolute value of rotation, whilst the encoder and gyro yields the rotational velocity. The compass was added to the calculations using a simple Kalman filter. The mathematical model and corresponding dead reckoning method are based on J. Borestein and Feng (1996) and we recount them quickly here. See the source for in-depth derivation of the formula.

Firstly the distance moved and the rotation made based on the encoders:

$$dRobot = \frac{dLeft + dRight}{2}$$

$$dTheta = \frac{dRight - dLeft}{L}$$

Where dLeft and dRight are the distance travelled based on encoder ticks since last timestep and L is the length between the wheels. This is then fused with the gyroscope values

$$dFusedTheta = (1 - gyroWeight) * dTheta + gyroWeight * gyroVal$$

Which is in turn combined with the compass through a Kalman filter. We will not get into the details of the Kalman filter, as they are described in chapter 7 of Ese (2016). Finally using dTheta and dRobot we calculate x and y:

$$\hat{x}_k = \hat{x}_{k-1} + dRobot * cos(\widehat{theta}_{k-1} + 0.5 * dTheta)$$

$$\hat{y}_k = \hat{y}_{k-1} + dRobot * sin(\widehat{theta}_{k-1} + 0.5 * dTheta)$$

$$\widehat{theta}_k = \widehat{theta}_{k-1} + dTheta$$

It is worth noting here that the fusing of the sensor data is very simple, and only the compass is actually included by the use of a Kalman filter. It is possible a Kalman filter including all the sensors, as well as the input to the motors could perform a lot better than the current one.

2.7 Equipment

2.7.1 Software

Windows 7 - Operating System

Atmel Studio 7.0 - Integrated development platform used mainly for Atmel microcontrollers. Combined with AVRDude to easily compile to the Arduino robot. See Andersen and Rødseth (2016) for details on setting it up.

SourceTree - Git client focused on a easy-to-use graphical user interface.

Termite - Simple RS232 terminal for serial communication with dongle.

MATLAB - vR2016a for modelling and graphing data.

Microsoft Project For project management. Overview of project progression, list of to-dos and time management and Gantt diagrams.

Powerpoint - Drawing vector graphics.

2.7.2 Hardware

Oscillioscope - Timing, debugging and measurements.

Multimeter - Hardware debugging and testing solder connections

Soldering equipment - For fixing loose and broken wires.

Optitrack Motion Capture System - Software used together with the "Slangelab" at NTNU to track and meassure reflectors for movement.

Chapter 3

Problems and solutions

3.1 Left motor problem

The problem

As written in Amsen (2016), at the end of Fall 2016 the Arduino robot started having problems with the left motor. It was at first uncertain what the exact problem was, but it seemed like the left motor was under-performing compared to the right motor, even at the same input from the software. This problem persisted, and despite there being a motor controller implemented, it was incapable of controlling the motors to the same speed due to the master/slave system and the slave being the broken motor thus the faster motor wouldn't regulate towards the slave. The robot also gave very confusing test results. During one test where the robot was held off the ground and two pieces of equally long tape were tied to each wheel, making a simple measurement of the total distance the wheels had travelled. However during this test it was shown that both wheels moved the exact same distance over the same time. Several tests were made, and they pointed toward the problem only arising when the robot was actually driving on the ground, and not during the tests where it was kept off the ground.

Finally it was discovered that the wheel cogs made of LEGO were having quite a bit of slack in the wheel axle, and at certain parts of the rotation of the one wheel the angle caused an extra bit of slack making the cogs slip their teeth. Figure 3.1 shows the loose cogs and how it could slip by the cog down to the right. It turned out that the reason for the inconsistency in errors was due to the weight of the robot when it was on the ground bending the wheel armature even more and making the slippage possible.

Another unexpected complication that arose from this and caused confusion was the sudden decline in position estimation exactness. This was due to the wheel turning less that the ticks by the motor, and thus what the position estimation thought. This manifested in an increasingly bad position estimation, both when turning and driving forward.

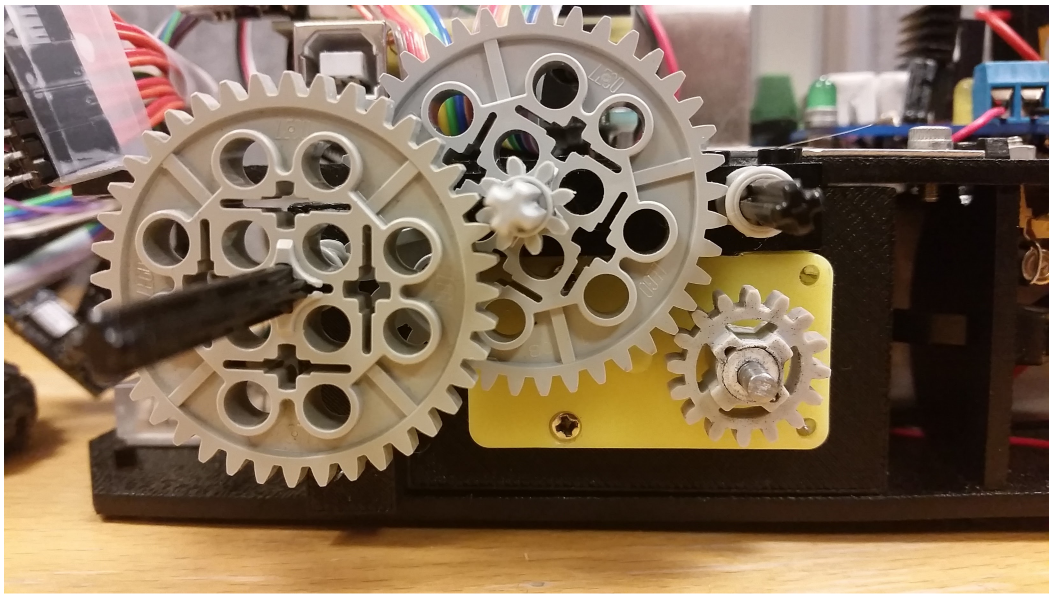
Figure 3.1: A demonstration of the slippage from the bottom right wheel.

The solution

There were several proposed solutions. One was reattaching one of the fittings for the wheel axle, however since it was glued on this was seen as a last resort solution. Another was fixing the slight angle the small wheel had, which would reduce the slip, however it wouldn't really solve the problem of the loose fittings and could mean the problem would resurface at a later time. Finally it was decided that a metal mount fastened to each of the three wheel axles to more rigidly hold the cogs together was to be made. After a consultation with the hardware workshop team at Electrical Engineering NTNU, it was created from a holed out metal plate.

This solution immediately showed great results. The left motor returned to full capabilities and ran the same speed as the right motor. The problems that had earlier arisen with the po-

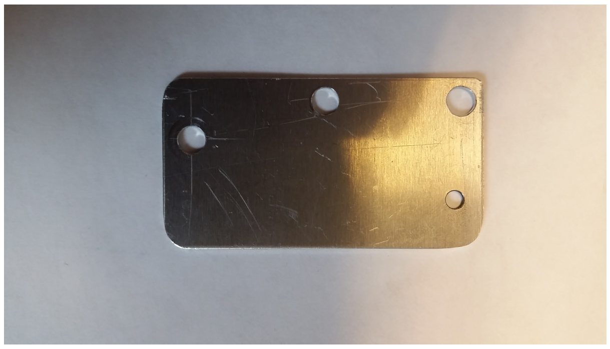
Figure 3.2: The finished metal plate.

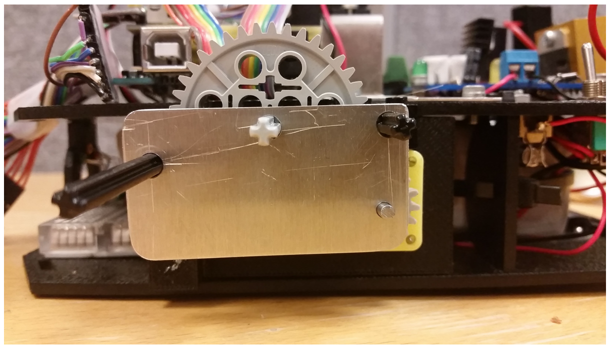
Figure 3.3: The plate mounted to the robot.

sition estimate also completely disappeared. The position estimate was not perfect, however it was clear that a lot of the errors had been caused by the differences between ticks and actual wheel turning. Furthermore, we concluded that the motor controller that had previously been unable to regulate the differences was not completely at fault, and could with the proper tuning work as intended. We also point out the problem with how dependant the robot is on the encoder being exact, and the necessity of fault detection with other sensors. In particular, a filter using both accelerometer and gyroscope would be able to detect the inconsistencies of the encoder using for example a Kalman filter. A natural question would also be if this problem could arise on the other side of the robot however and during the consultation with the workshop it was decided that the two wheel armatures have been slightly differently glued on and the right side is a much tighter fit. We do not expect this problem to occur unless serious strain is put on the cog mounts.

3.2 Sensor improvements

Part of improving the position estimates is having reliable and consistent data from the sensors. As a step to improve the overall functionality of the robot we decided that a good place to start was confirming the integrity of the different sensors, as well as improve on them if possible. Several tests have been made for each sensor through the semester, some more crude than others in order to establish at least a bottom line of functionality. We remind the reader that only the encoder, gyro and compass were used in the robot, of which the gyro and encoder did the lion's share of the sensor data.

3.2.1 Testing

Encoder In order to test the encoder we considered where faults might arise: The magnetic disc might not rotate at the same speed as the motor axle, or it might slip due to the way its been fastened, the encoder might not pick up on all the ticks, the conversion from motor rotation to wheel rotation - known in the code as WHEEL_FACTOR_MM might be off - or even different between the two wheels or the wheels might experience slippage on the surface.

The first test that was done was checking if all the magnetic poles on the disc worked as intended and that the encoder picked up the differences in magnetism as expected. It was done using a oscilloscope probing the signal wire on the encoder and marking the magnetic disk to count the rounds it rotated. This yielded the expected results of the encoder picking up every single change in magnetic pole to a high precision, not a single tick was lost over 100 ticks on both wheels.

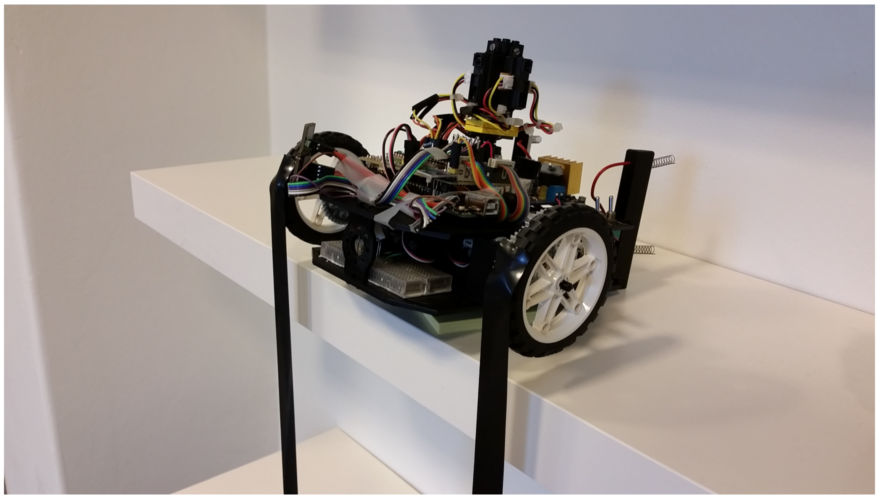

Figure 3.4: The setup for testing the encoder.

Further we tested the accuracy of the wheel factor. The test was done by fastening a piece of tape slightly longer than 2m, marking the 2m point on them, to each wheel. This sticky tape was used to eliminate the possibility of sliding on the wheel, as well as being able to measure the distance the wheels had rolled. A small weight was attached to the bottom of each tape to hold them taut, and left hanging from the robot. This was done with the wheels off the ground to so the robot wouldn't move and the tapes hanging off a ledge. The robot was then given a command to move either forward or backward for 2m, while continuously outputting the distance it estimated the wheels to have travelled.

Table 3.1 shows the results from the test. At first it would seem the robot is underestimating the wheel factor since it overshoots. However this test is important to look in comparison to the robot's estimated position, in which case we can see that the error between estimates are very low, under 1 cm over two meter distance. The reason for the overshoot however is the high speed of the wheels and the momentum they have when they reach the appropriate distance making the wheels roll slightly over the desired point before coming to a stop. The good thing is this distance rolled is picked up by the encoder and as we can see; very accurately estimated. It is worth noting this test is a very isolated case by being held off the ground, it only looks at the direct correlation between the encoder ticks and the distance the wheel moves without the effects of ground friction, in other words the value for the wheel factor in a perfect world.

[cm]	min	mean	max	max estimated error
Left forward	201,2	204,6	207,2	1,1
Right forward	200,8	204,3	207,3	0,7
Left backward	199,2	203,8	207,8	0,5
Right backward	200,4	204,0	207,9	0,5

Table 3.1: Results from the encoder tape-test

The second test we did was to add the effects of driving on a proper surface. The difference here is mostly the effect of friction, or rather slippage, on the wheels and how this might affect the difference of estimated distance and actual distance. The test was done on a linoleum floor at "Slangelabben" is which prone to dust, dryness and irregularities. This was our main test surface. Our hope with this test is seeing if the addition of driving on a surface might affect our accuracy in position estimation or change the wheel factor due to slippage. The test was the same distance as before; 2m driving. Due to the robot losing contact at the very end of the test, we can see that it doesn't actually hit the full 2m distance. However since our focus was on comparison to the OptiTrack, rather than the actual distance, the results are still perfectly viable. Figure 3.5 shows the results of the test with encoder values in blue and OptiTrack values in red. We see that after almost 2 meter we have an error between the OptiTrack and the estimated position of barely 1 cm.

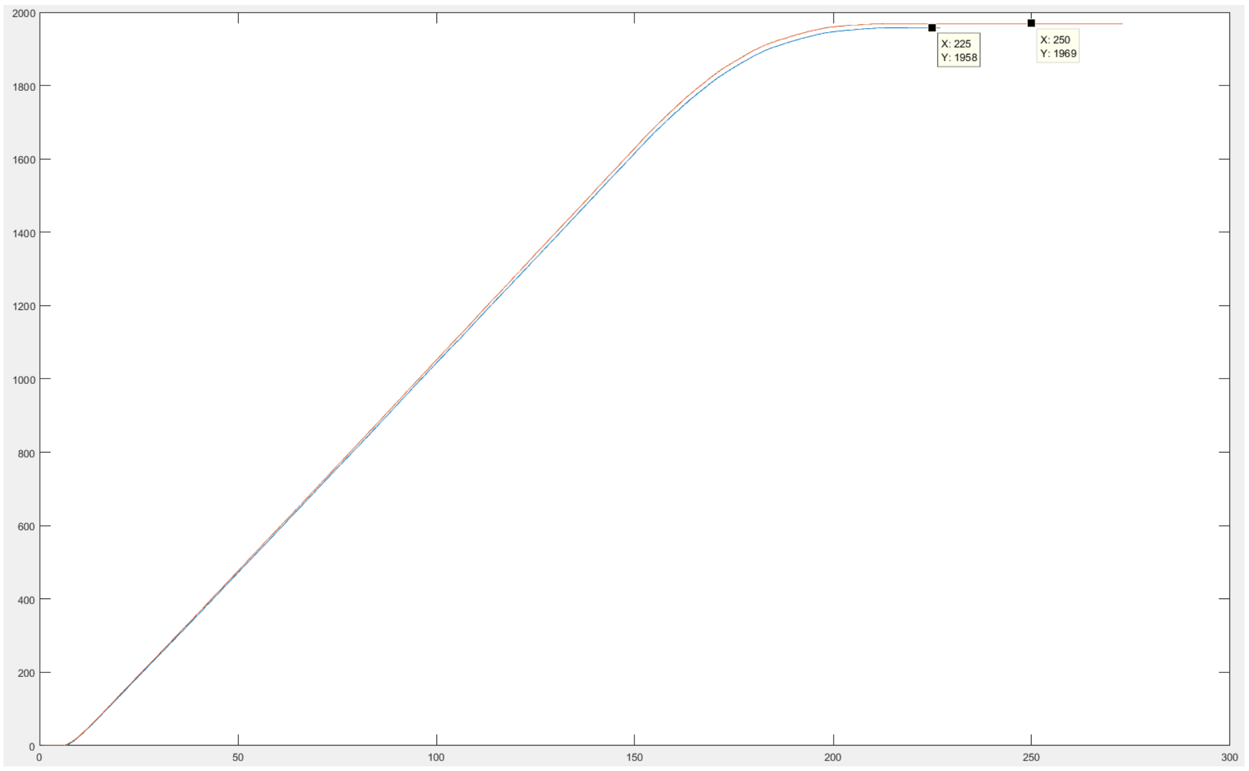

Figure 3.5: Comparison between encoder and real world. Encder blue, OptiTrack in red.

Gyroscope and encoder during rotation. To test the Gyro we wanted to test it compared to the encoder as well as the real world since it contributes through the fusing of gyro and encoder. Since measuring rotation accurately isn't as easy by hand, we decided to use the OptiTrack system, or "Slanglabben" to accurately track the real rotation of the robot. What we are looking for here is not the robot rotating exactly the amount of the server command, but rather to see that the rotation it does do is precisely measured. For this test, we did several rotations of 180 degrees, as this is the maximum the robot will ever do at a time and tracked pure encoder, gyro and Optitrack values. The results can be seen in figure 3.6. Here we can see the accuracy of the encoder values, as they are near indistinguishable from the optitrack values. The Gyroscope however seems to drift more and more with each rotation and ends up with a 25 degrees error after 900 degrees of rotation, which is within 2.7% accuracy. This doesn't sound like much, but it doesn't take long for either of the robots during navigation to acquire this amount of rotation. We also witness a very constant increase in errror, implying this is a matter of bias. The results from the encoder are also much better than expected, and backs up our earlier assumption that the encoder wheel factor is well calculated.

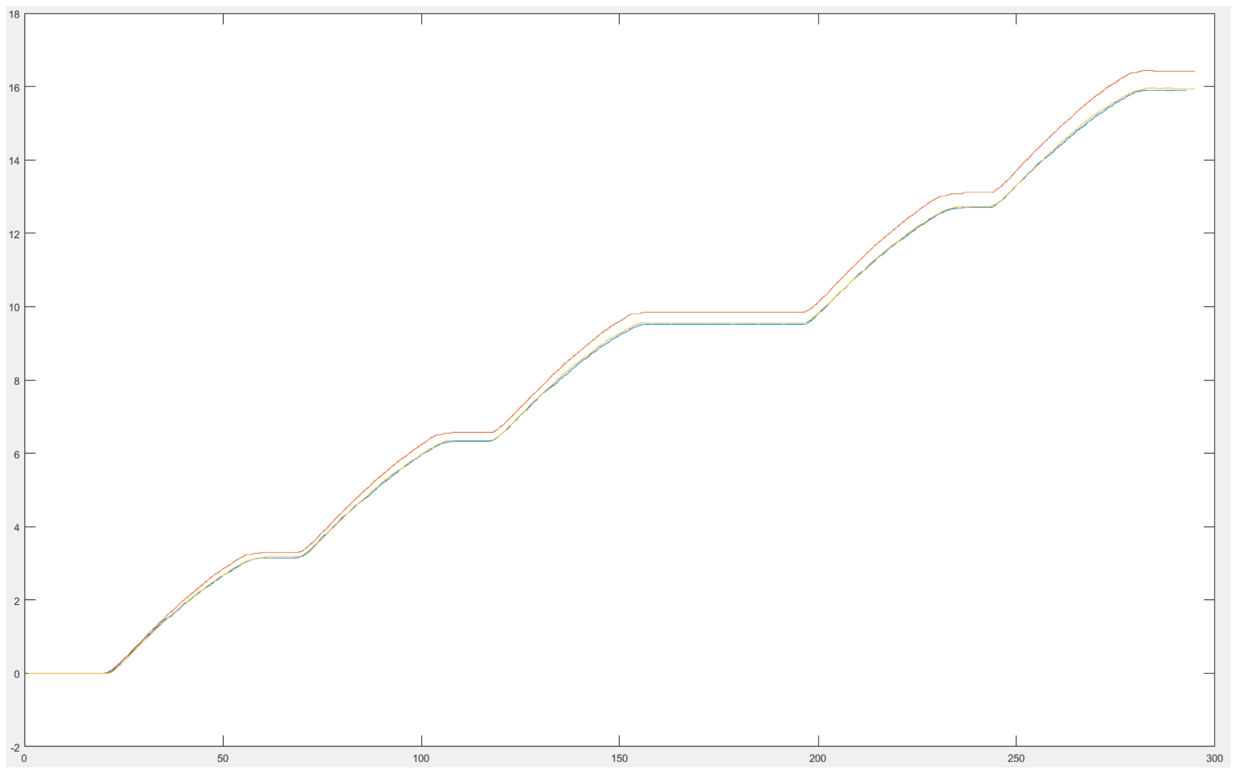

Figure 3.6: Encoder and gyroscope comparison to real world. Gyroscope in red, encoder in blue and Optitrack in green.

Compass. Testing the compass was important as it was mentioned in Ese (2016) that it could be unreliable if it was in the vicinity of magnetic areas. A simple solution to this problem was added by testing the change between each time-step and look for changes that were unreasonably large, discarding these. In this test we wished to test the reliability of the compass when isolated from magnetic disturbances as much as possible. The test was performed by placing the robot on a stack of books and other non-magnetic items to isolate it laterally from the ground and possible metal surfaces within it, as well as moving it into the middle of the room. It was calculated about a 70cm distance to any metallic surface not on the robot. The robot was then given two commands of 180 degrees and the raw data from the compass was logged. Figure 3.7 shows the results. In this plot we witness two kinds of disturbances. One is the large spikes visible around time-step 60 and 210 the other is the general constant sensor noise. The large spikes are actually not sensor noise in themselves, but rather the conversion from 0 to $2*\pi$ which flips back and forth due to sensor noise around the point of conversion. This is why we don't witness the large spikes during the middle of the graph. It is also interesting to note the noise when the robot stands still, in particular the lack of drift even if we witness variance. This is the strength of the compass. A simple mean value during no motion could reset both an accelerometer and gyroscope. However we find the noise during rotation to be too dominating to add valuable information to the gyroscope and encoder data we already use.

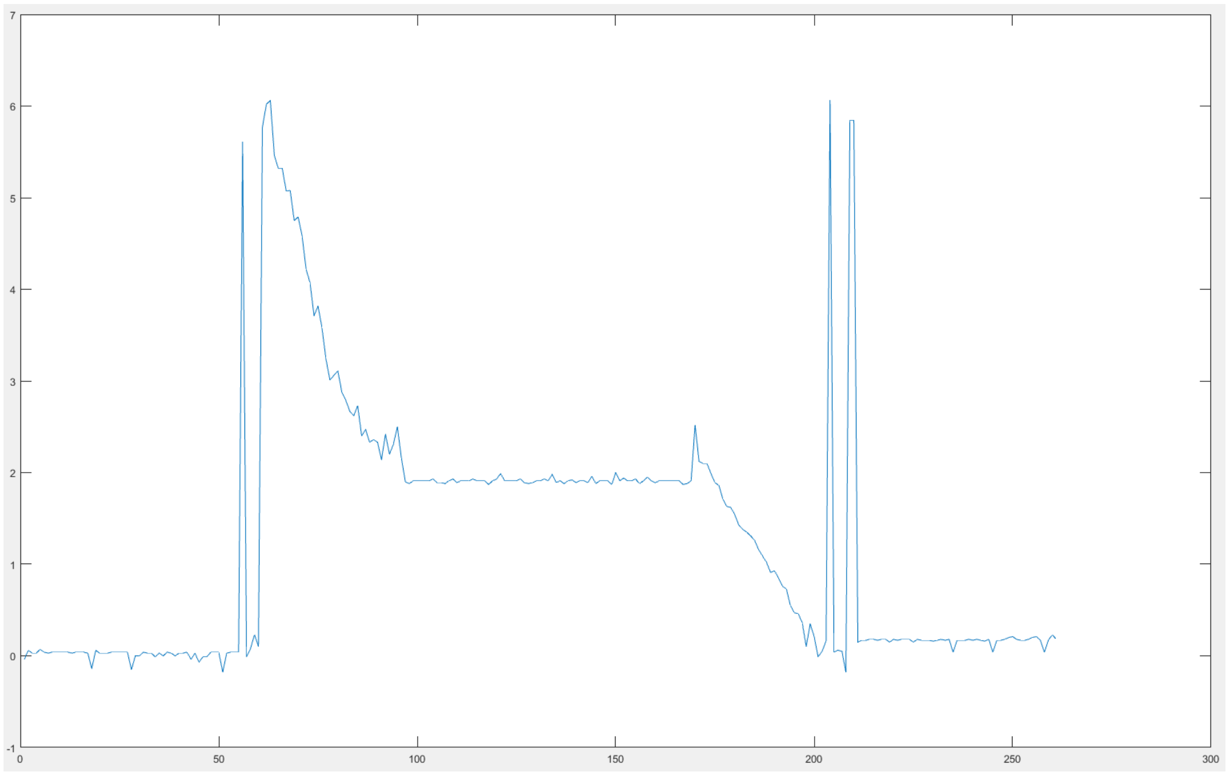

Figure 3.7: Raw compass values during 2 180 degree turns.

Accelerometer The accelerometer was an interesting sensor to test, as it had not been used in the software so far. Accelerometer data can be very valuable information, as it is the only other sensor than the encoder capable of measuring translational movement. Thus it's a vital sensor if we wish to cross-reference the encoder during running, for example to measure slippage or crashing into walls, something the encoder is currently very vulnerable to. The big problem with the accelerometer however is its sensitivity to measurement noise when we double integrate it, as shown in the earlier theory section. It's a known problem to calculate translational movement from an IMU only, and many papers have been written on the subject, suggesting different kind of filtering.

The test we did was a very simple test where the robot was given two commands of 50 cen-

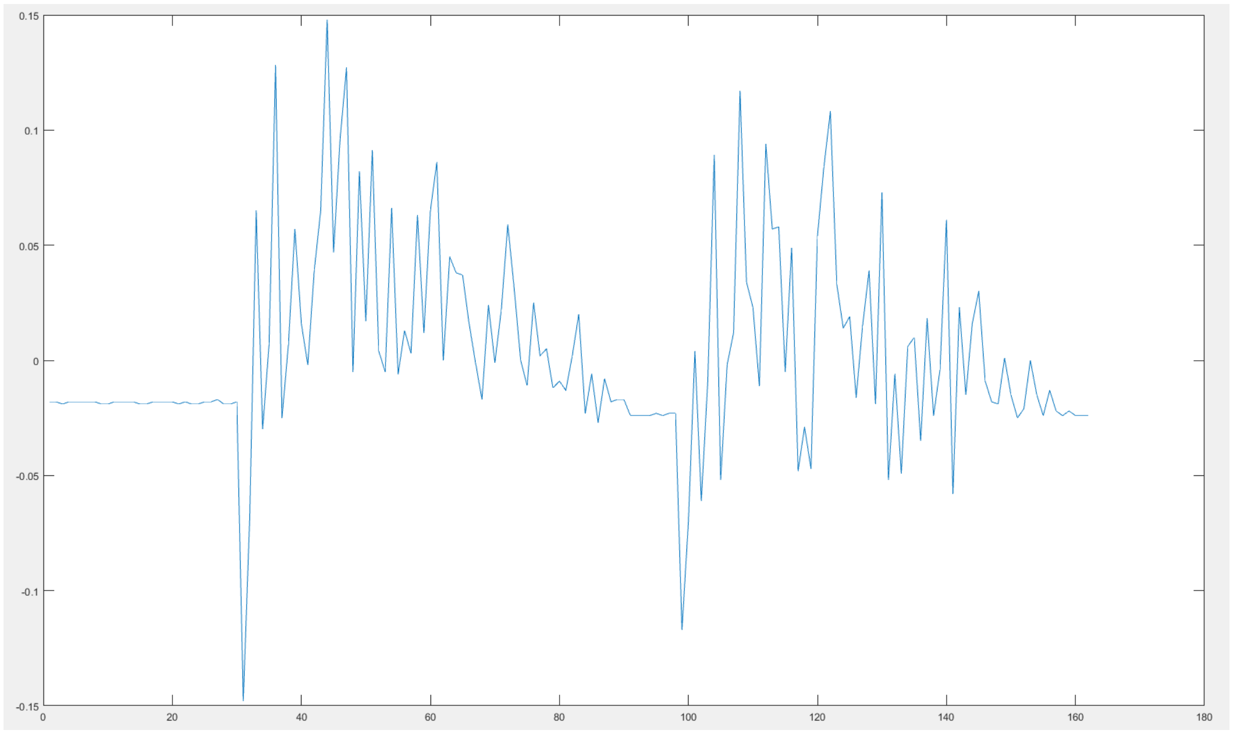

Figure 3.8: Raw accelerometer data

timeters with a short pause in between. Then we logged the raw data from the accelerometer, the results can be seen in figure 3.9. This example shows a lot of the problems we described in the theory section. We can see a constant bias of about -0.018 as well as a lot of noise. It is intuitively clear that this is not the true acceleration of the robot but rather noise. We can see the two points at about time-step 30 and time-step 95 where the robot starts moving forward. This is shown by the two high-value negative spikes. Then we have a constant but decreasing acceleration until the robot again is stationary. This coincides well with how the robot moves in the real world, where it has a smooth decrease of speed the closer the robot is to the goal. However the problem we are seeing is that the noise spikes are almost as large as the real acceleration spikes. This is most notably seen on the first negative spikes of both parts representing the forward acceleration, which in truth should dominate the other noise, if any meaningful data existed. We however make one valuable observation; during no motion the sensor noise is almost non-existent. This leads us to believe that it is not the accelerometer itself, but rather the motion of the robot and the way it has been fastened to the robot that creates the noise.

We did another test to try this hypothesis, this time the robot did not move of it's own mo-

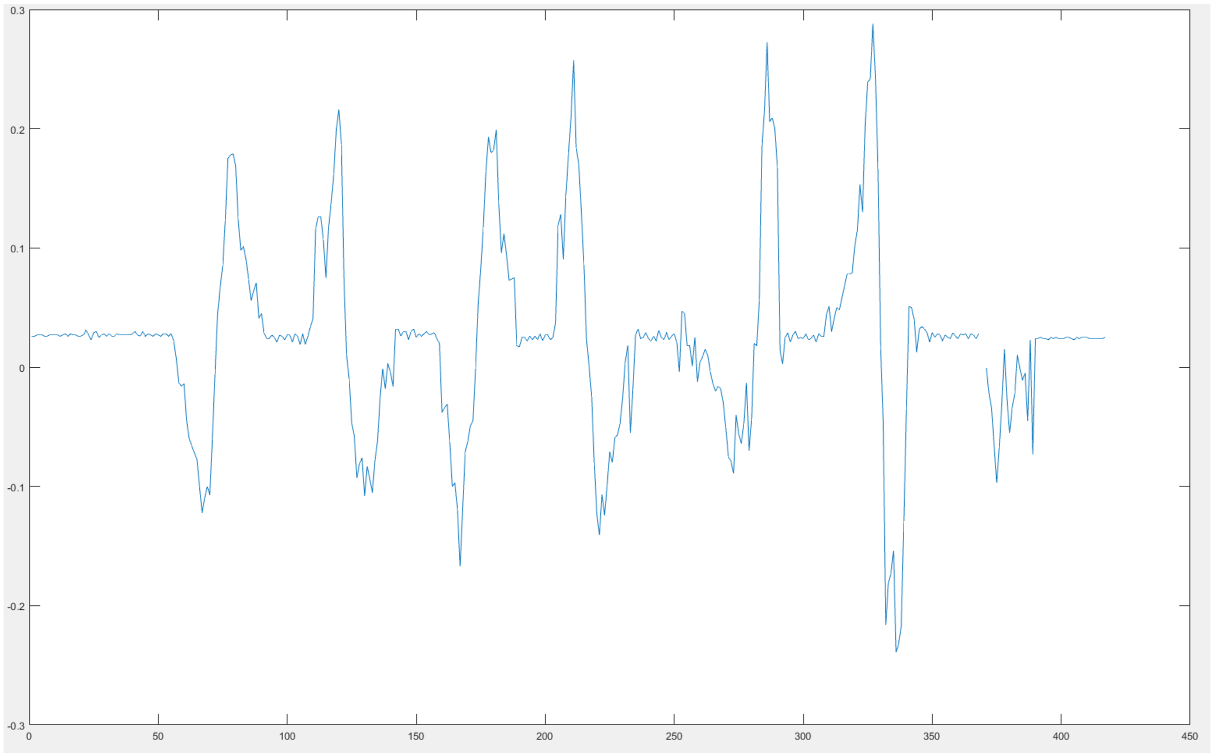

Figure 3.9: Raw accelerometer data with the robot on a board.

tors, but was rather put on a very slippery plate on a very low-friction surface and was dragged by hand. This was an attempt to reduce the vibration the motors might be causing when moving, as well as the vibration between surface and wheels. And the results were very promising. Figure 3.9 shows the results from the test. We clearly see much less noise affecting the signal, and the noise spikes are no longer dominating the true values. The movement we did was simple back and forth movement over the same distance with slightly varying acceleration. A few intuitive observations we make to further legitimize the measurements: the stronger the spike, the shorter the duration, as well as the spikes seem to be quite evenly spread around the zero-point, meaning it at least somewhat accurately describes the same distance moved for the robot each direction. the spikes comes in pairs of positive and negative, describing the acceleration then deacceleration for each stretch moved. To further explore this data we did a simple numerical integration of the acceleration to get an idea of the speed this acceleration described. We also decided to subtract the stationary bias to avoid it affecting the integration too much, the results are shown in figure 3.10.

Again these values are very promising, the main point we are trying to look for here is how the

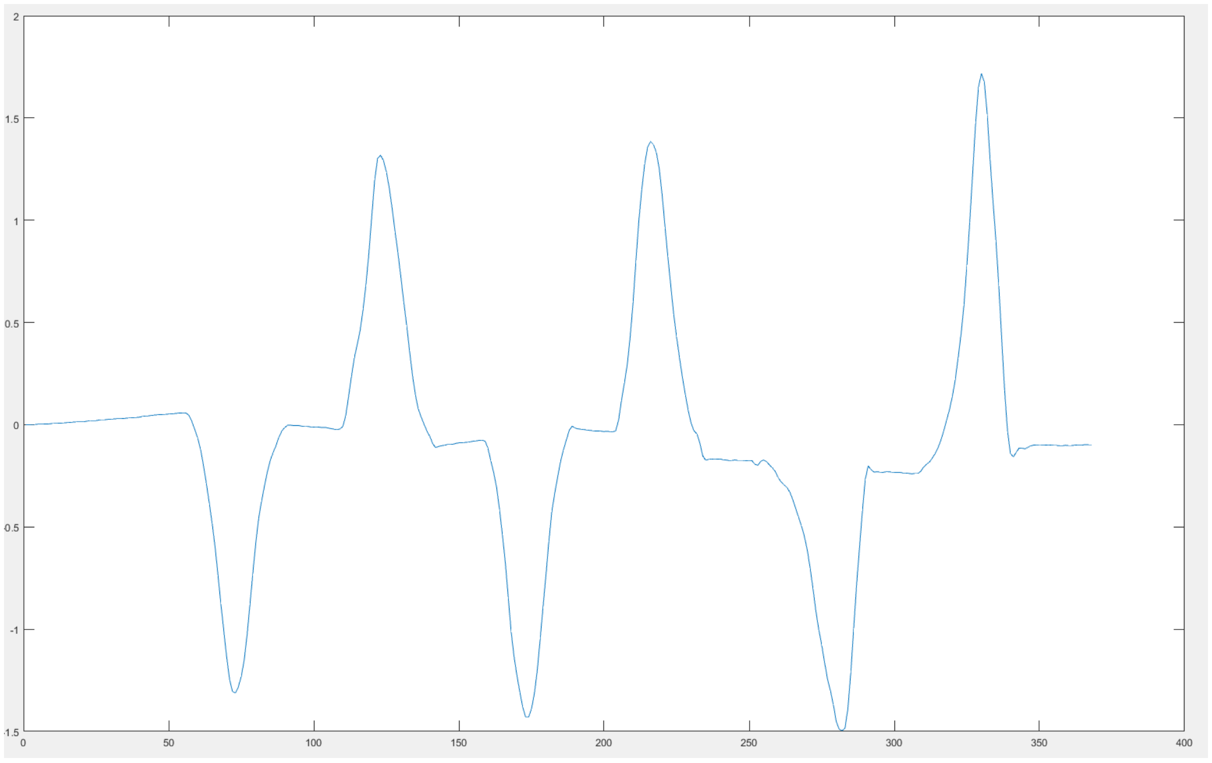

Figure 3.10: Simple integration of the raw acceleration

speed almost goes back to zero for every time it stops and changes direction. We can clearly see the speed isn't exactly zero for every time, but we believe this data can be much better treated with proper filtering and a smart-filter using the knowledge that the robot is standing still to reset the acceleration and speed. We make one last observation by integrating the speed and acquiring the estimated position seen in figure 3.11. We make a few observations here, mainly we see the dominance of double integrated noise just like explained in the theory section. However if we focus on the first section of movement from zero to -16 back to around zero, this is exactly the data we were hoping for. This shows the possibilities of the accelerometer under noise-reduced circumstances. We can see it estimating almost perfectly back to the original position. The latter sections are too dominated by drift, but the proof of concept over short stretches is very promising.

3.2.2 Improvements

Encoder. From our tests we concluded that the only real improvement we can make to the encoder is the varying wheel factor constant. However as was mentioned in Ese (2016) this

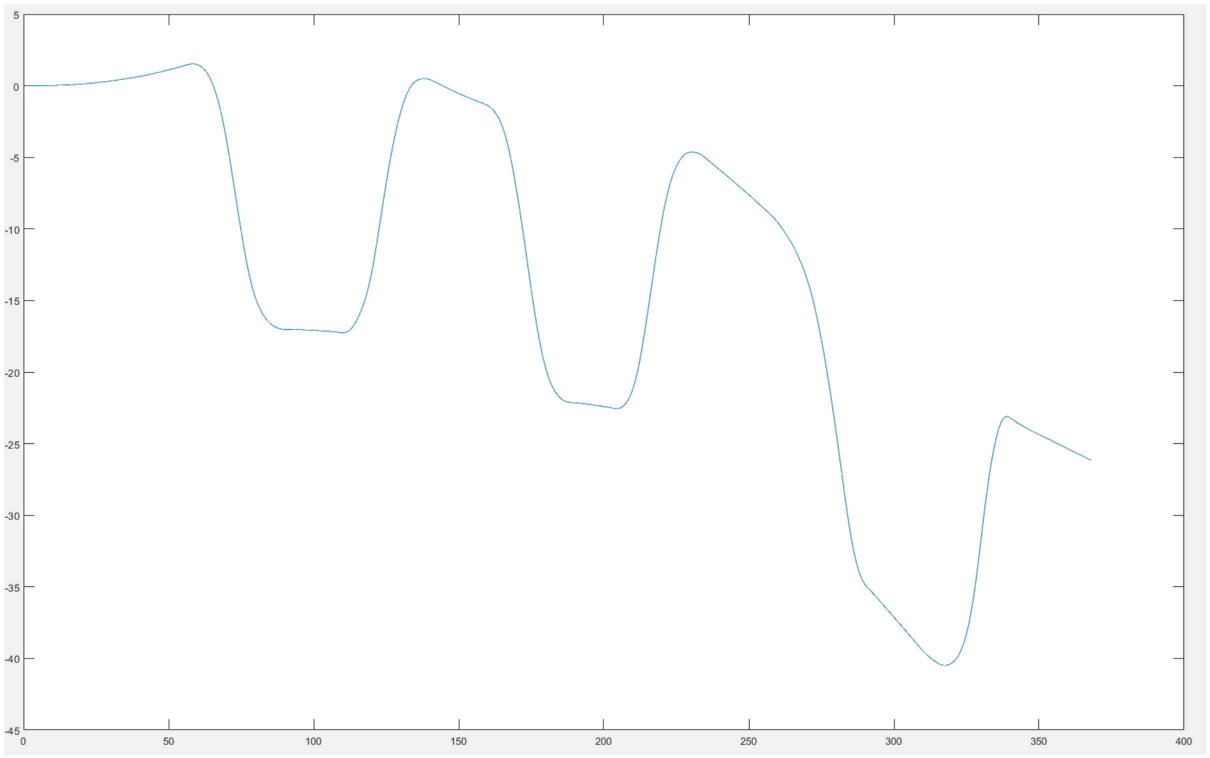

Figure 3.11: Simple double integration of the raw acceleration

constant varies depending on surface. During some of our tests, we did small adjustments to the wheel factor to compensate for the change in surface. Other than this, we see very little noise or faults with the encoder sensors themselves, and we believe they yield very reliable data when the wheel factor is correct and major slippage can be detected, with for example a gyro or accelerometer.

Gyroscope From different tests it seemed the off-set observed on the gyroscope varied from test to test. It was decided that a more frequent update of the gyroscope off-set would be valuable. A simple mean over 100 values every time the robot was between commands was implemented.

Accelerometer. As it is not used at the moment, we didn't make any specific improvements to the accelerometer. However we advice further work to isolate the accelerometer from the robot's vibration and more accurately determine the source of the noise factor. It is also possible to reduce the range of the accelerometer, thus reducing the noise factor compared to the values we are using. It is currently at $\pm 16g$ and could safely be turned down to $\pm 2g$ with the current speeds of the robot.

Compass. The compass data is useful, but only when used properly. During several tests of the compass it was shown to be unreliable during actual rotation and corrupted rather than improved the data we already had from the encoder and gyroscope. It was decided that the compass would not be used during rotation. But an inclusion of the compass during stand-stills to reset the values could be valuable.

3.3 Position control implementation

As was mentioned in the theory section, the navigation system on the robot wasn't satisfactory, and we found several faults with both the motor controller and the lack of constant heading-control during forward movement. It was therefore decided that a part of improving the robot's ability to perform its tasks was also improving it's ability to navigate properly.

Theory

Due to the system as a whole, taking into account the server software, how it navigates and commands the robots and the environment they would operate in, we decided to create our own control algorithm to more specifically suit our needs.

There were certain requirements to the algorithm:

- It should drive in a moderately straight line to the target to avoid hitting walls
- It should have smooth driving and rotation to avoid overshooting
- It should be fast, but also accurate
- It should be simple enough for the microcontroller to calculate
- It should be easily tuned and tweaked

These are all simple enough requirements, and most of them are already satisfied with most control laws for differential drive robots. The one requirement of "straight line" is especially important, and is an important design aspect. These are all already satisfied by the current control law, and are seen as a "bottom-line" of how our new controller should perform. As for the old controller we had a few problems we wanted to improve upon especially:

- Accuracy towards the end-point
- Coordinate point rather than polar navigation
- Smoothness of movement, less rigid
- Constant regulation of heading
- Ability to compensate for a under-performing motor
- Better speed regulation based on distance to target
- Should be useable with only polar coordinate information so the server does not need to be changed.

3.3.1 Design

The requirement of straightness was the main reason we decided not to use a traditional control law, but instead opted to use a two-step algorithm, quite similar to the one already implemented. It would consist of two states, either rotating or "driving", which we will call the step where the robot is moving towards the target. This way, the robot could rotate within a reasonable heading error, before starting forward movement. This way we minimize movement taken in the wrong direction, and thus also the "curve" of the trajectory. The rotation is a simple reverse-direction rotation, meaning each wheel rotates in the opposite direction at the same speed, with a simple p-controller on the actuation to avoid overshooting the rotation. Furthermore, there is a hysteresis mechanic between the two modes. This means that the rotation mode will rotate until a very small threshold of the goal heading(0-2 degrees), then the drive mode will be active until the heading error surpasses a much larger value (10-30 degrees). This way we avoid constant changing between the two modes and a smoother running robot. The drive mode constantly updates heading and heading error, as well as distance to target to calculate the difference in speed of the motors with a PI controller to drive the heading error to zero.

The integral component of this controller is vital in case one of the motors are not performing as well as the other. Since we wished to avoid the necessity of changing the server, we decided to do a conversion from polar coordinates to xy-coordinates internally. The question arose how much accuracy we'd lose converting from polar coordinates, but it was clear that with the floating point accuracy internally we wouldn't lose any noticeable accuracy in the real world due to our conversion.

```
if new setpoint received then
    xTargt = xhat + Setpoint.distance*cos(Setpoint.heading + thetahat);
    yTargt = yhat + Setpoint.distance*sin(Setpoint.heading + thetahat);
distance = $\sqrt{(xTargt - xhat)^2 + (yTargt - yhat)^2}$;
if distance < speedDecreaseThreshold then
    actuation = maxActuation*distance/speedDecreaseThreshold;
else
    actuation = maxActuation;
if distance > radiusEpsilon then
    Recalculate $\theta_e$ from target and position;
    if $\theta_e$ > rotateThreshold then
        doneTurning = FALSE;
    else if $\theta_e$ < driveThreshold then
        doneTurning = TRUE;
    if doneTurning then
        if $\theta_e$ >= 0 then
            LSpeed=currentActuation-$K_p$*$\theta_e$-$K_i$*leftIntError;
            RSpeed = currentActuation;
        else
            RSpeed=currentActuation-$K_p$*$\theta_e$-$K_i$*rightIntError;
            LSpeed = currentActuation;
        leftIntError += $\theta_e$;
        rightIntError -= $\theta_e$;
    else
        if $\theta_e$ >= 0 then
            LSpeed=-maxActuation*(0.3+0.22*($\theta_e$));
            RSpeed=maxActuation*(0.3+0.22*($\theta_e$));
        else
            LSpeed=maxActuation*(0.3+0.22*($\theta_e$));
            RSpeed=-maxActuation*(0.3+0.22*($\theta_e$));
```

Algorithm 1: Control algorithm pseudocode

3.3.2 Testing and simulation

Before implementing it on the robot, the algorithm was implemented on a model of the robot in MATLAB. This was done to test the proof of concept, while also having an environment where the parameters and algorithm could be easily tweaked. The model was created with a mix of MATLAB functions and simulink modelling for easy overview. This model was greatly simplified by removing the position estimation part of the robot. This meant that all position values used in the controller are real values. This was done because the the principles of the controller were based on the robot's estimated values, not the real values and comparing estimates to real values serves little purpose in this case.

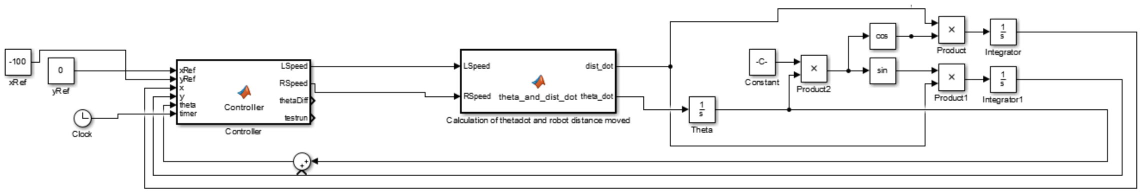

Figure 3.12: The MATLAB model. Controller to the left, robot model to the right

The first iteration of the controller encountered an unexpected problem. As we can see in figure 3.13 the plot seems to loose its smoothness towards the end of the convergence. At first it was assumed that this was a problem with the numeric attributes of MATLAB's solvers. However upon further investigation it was discovered that it was the controller flipping between rotation and driving mode constantly. We direct the readers attention towards the increasing duration of rotation as we get closer to the goal, and also the shorter duration of forward motion between each rotation. This is due to a very relevant problem; the closer the robot gets to the target point, the bigger the changes in θ_e is at the same forward speed. Meaning it will move only a fraction, then turn again constantly.

This can be seen in figure 3.15 where we have sudden spikes in the theta error over a short amount of time and the same speed. Figure 3.14 illustrates this problem for the reader. The orange angle is the original θ_e, and the red is the change after a distance moved; $d\theta_e$. The two robots show the difference depending on how close the robot is to the target. The closer the robot is to the goal, the greater the error will change per time-step due to the constant speed. To remedy this we need a speed controller. The immediate thought would be something like a

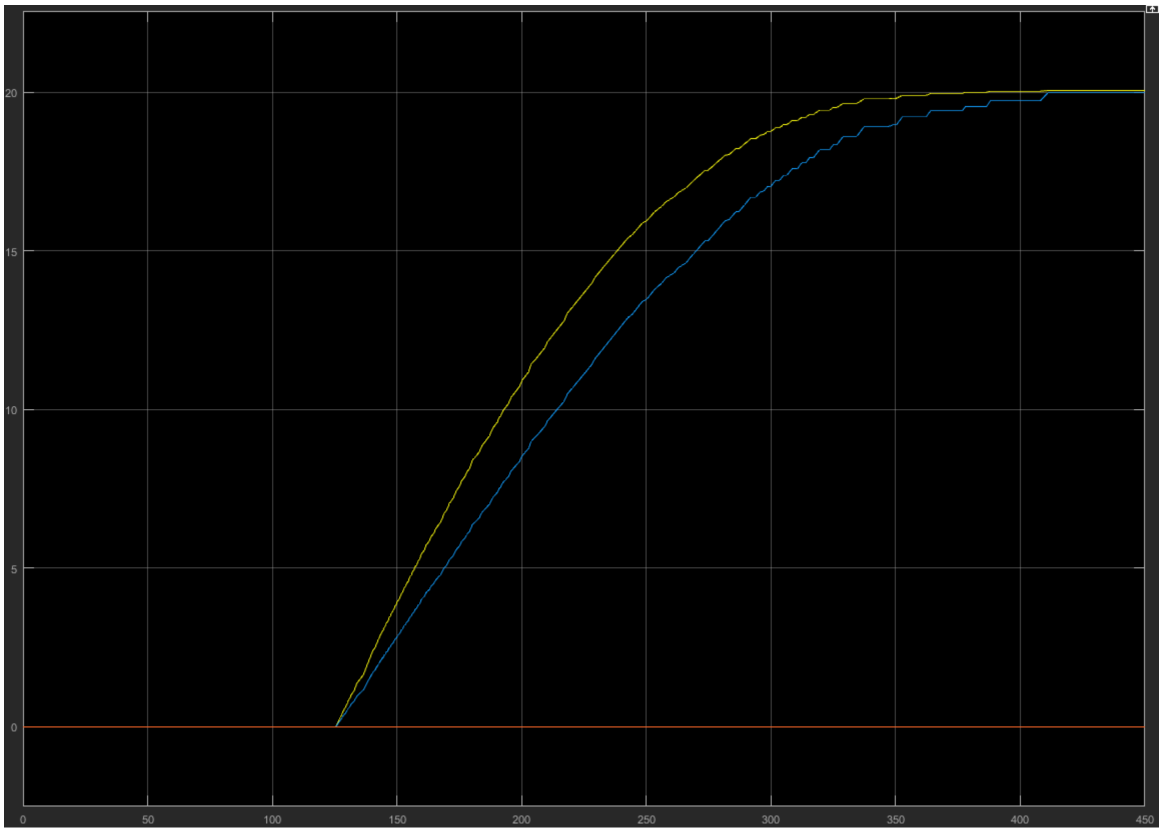

Figure 3.13: Plot of the x and y values during the simulation without speed control

PI-controller. However since we'd like more consistent control over the speed as the robot came closer (i.e. no integrated error that might build up more over longer distances) we decided on an inverse-proportional speed controller that kicks in at a certain distance from target. Which was later modified to include a small constant to make sure the robot reached its target.

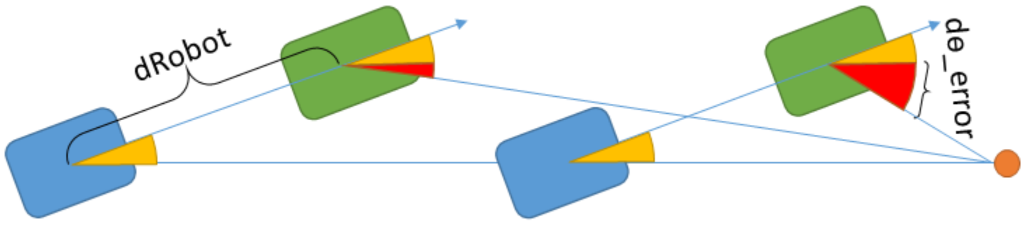

Figure 3.14: An illustration of the theta error problem

We also made a note of how the x and y curves didn't align, implying the robot didn't drive in a straight line toward the target. This was caused by an untuned controller variable, and by increasing the K_p of the heading controller we immediately saw a tighter fit of the x and y curves, implying the robot drove in a straight line, as opposed to a curved. However this tune

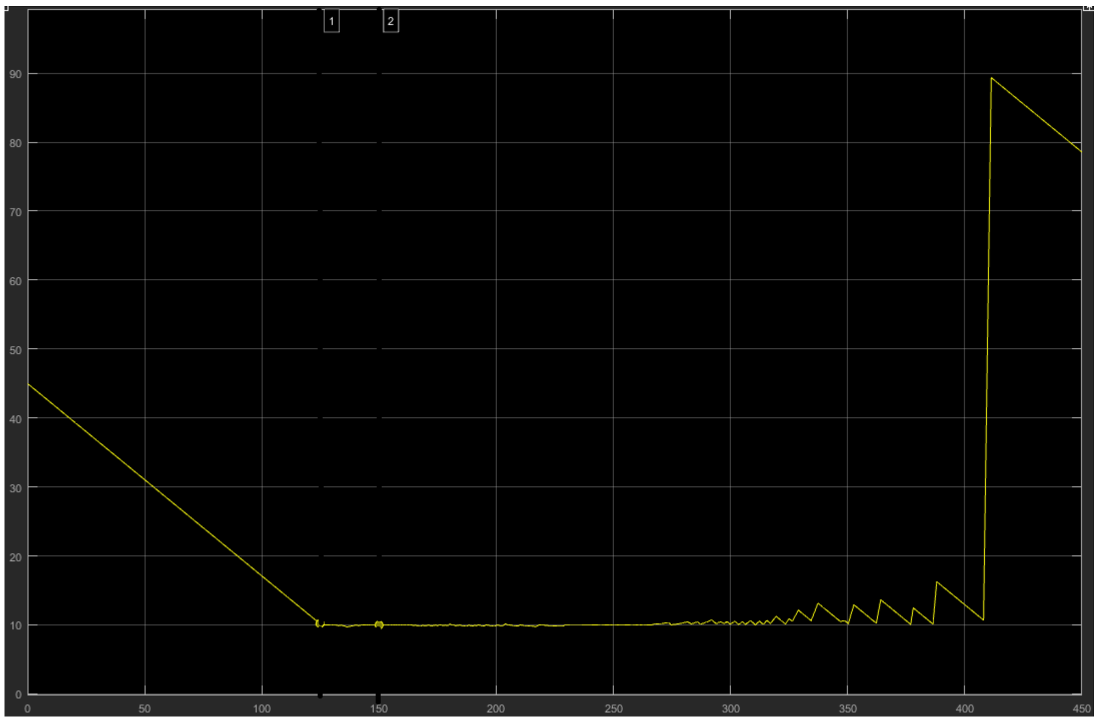

Figure 3.15: Plot of the theta error without speed control

alone would not solve the problem mentioned in the earlier paragraph, as the required K_p would go towards infinity the closer the robot came to the goal due to the aforementioned speed of divergence when the robot didn't have a speed controller.

Final simulations are shown in figure 3.16, 3.17 and 3.18. From this we can see all the earlier problems of jittering, as well as poor theta control have been solved. The X and Y values are almost completely aligned, this combined with the position plot in figure 3.18 error shows an almost perfectly straight line towards the goal. The smoothness of the curves, for both theta and position are also quite sufficient. It is important for a robot such as this to not attempt too rigid motion, due to it's mass, and overshooting a target can be very problematic if near walls. This test proved the concept of the regulator, however we also put it under some more rigorous testing to ensure it didn't have problems with sudden interrupts, changes of commands or changes in robot orientation as well as targets in all quadrants. We also mention that a lot of the tuning will probably have to be redone in real life due to the inaccuracies of the mathematical model.

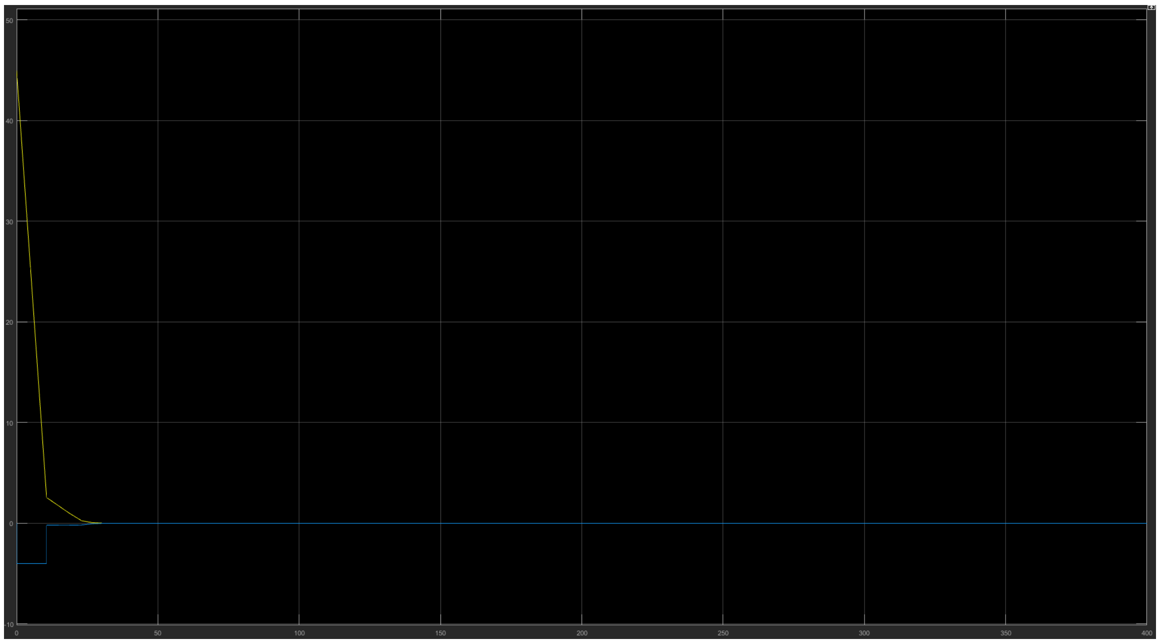

Figure 3.16: Plot of the theta error with speed control.

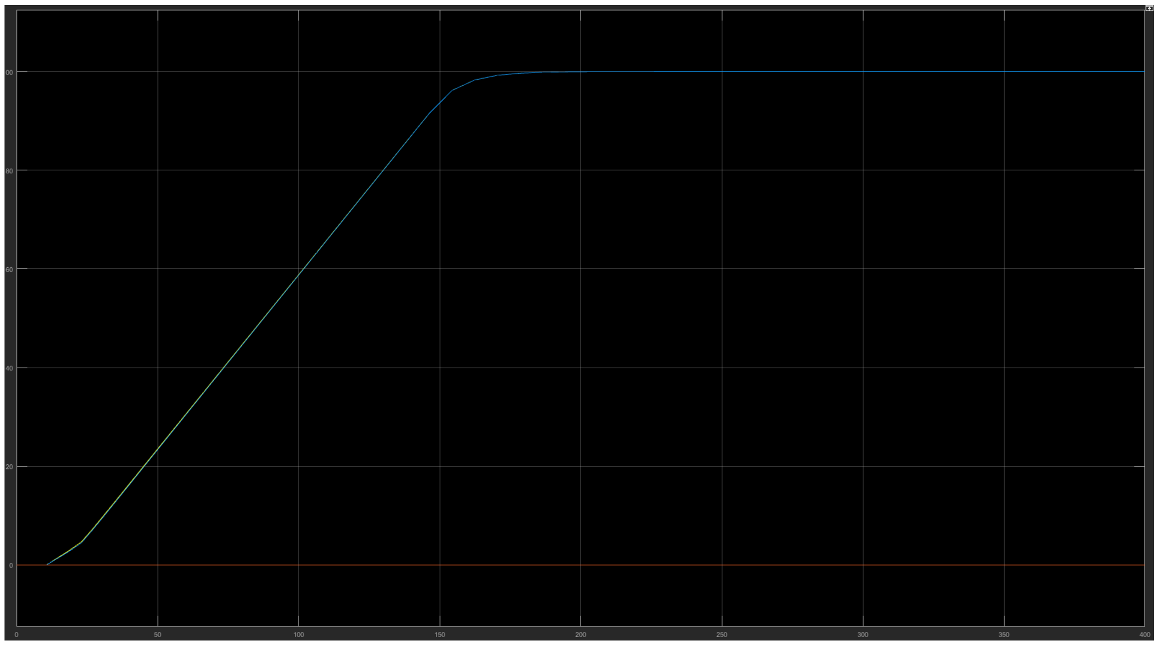

Figure 3.17: Plot of the XY values with speed control.

Step response on theta changes An important test we made which yielded some results we should comment on was when we added a step on the position estimate, simulating a sudden change in the robots heading, from for example an external source. What we wanted to see with this test was the robot's ability to regain the heading and how much such a disturbance could

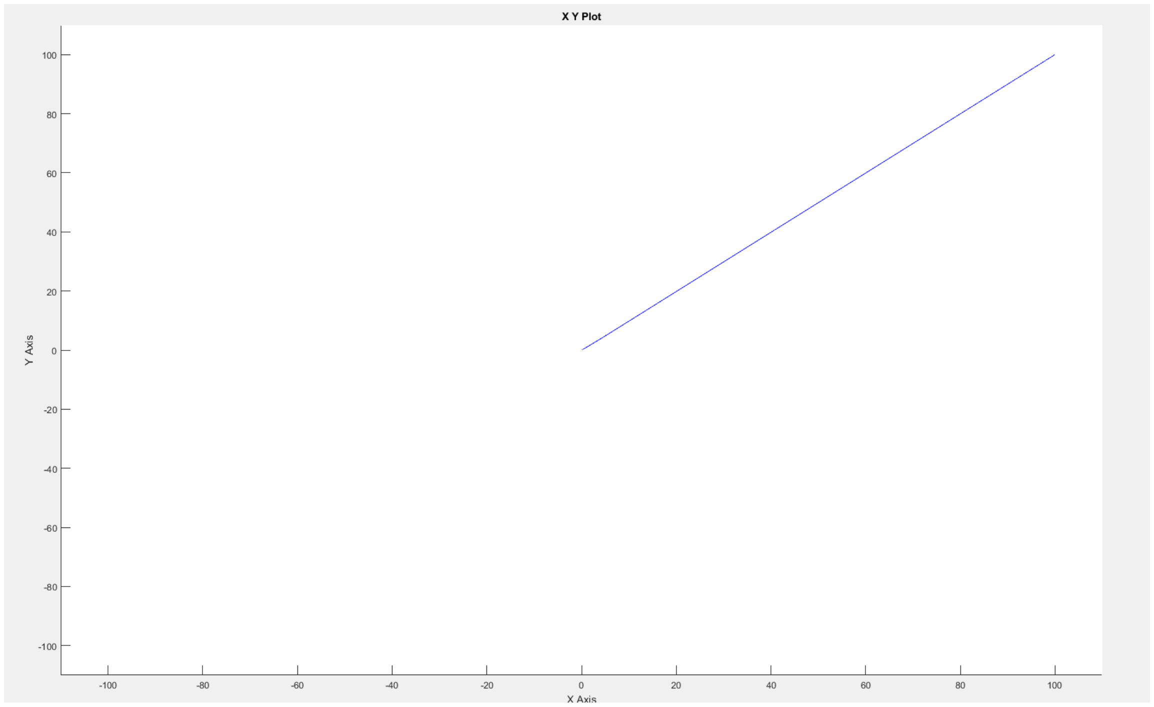

Figure 3.18: Plot of the robot trajectory with speed control

cause. The step had a value of 9 degrees at time-step 100. The importance of this test was to not trigger the rotation mode, but rather test the robot's drive mode during max theta disturbance. During this test, the threshold was 10 degrees before the rotation mode would have kicked in. Figure 3.19, 3.20 and 3.21 show the results from this test. The first thing the reader will notice is the seeming instability of the theta error halfway through the simulation. This in fact not instability of the controller itself, but a problem with MATLAB's numeric solver. More specifically it's the problem with integrating MATLAB functions in a simulink model. There exist inconsistencies in the values when one tries to use functions written in MATLAB code instead of pure simulink blocks. This is due to MATLAB not having optimized the user-written functions. However if we disregard the jagged behaviour caused by this problem, we can still see the controller properly realigning the robot to zero heading error. Another thing the reader might notice is the nonalignment of X and Y after the disturbance, and how the robot isn't "regulating" these towards each other, this is due to the robot not aiming for the same trajectory as the earlier simulation, but rather it recalculates and creates a new trajectory based on the disturbance. This can be more easily seen on figure 3.21. We can also quickly mention the reason for the curvature after the disturbance, one that is not visible during the first half. This is because the theta control is done during forward movement since we didn't breach the 10 degrees threshold, whilst if we had, as with the first half (where the starting error is 45 degrees) the rotation would have brought us to within 1 degrees before the forward motion began. This is a tradeoff, and the 9 degree error is almost as large as we can get it without starting the rotation. We believe this curvature is acceptable.

Other than this, we found the results to be very satisfying, and that an implementation on the robot, where the complexities of MATLAB solvers wouldn't be a problem would eliminate the problems shown earlier. Proper tuning of the parameters for straighter and faster regulation would also be done on the robot.

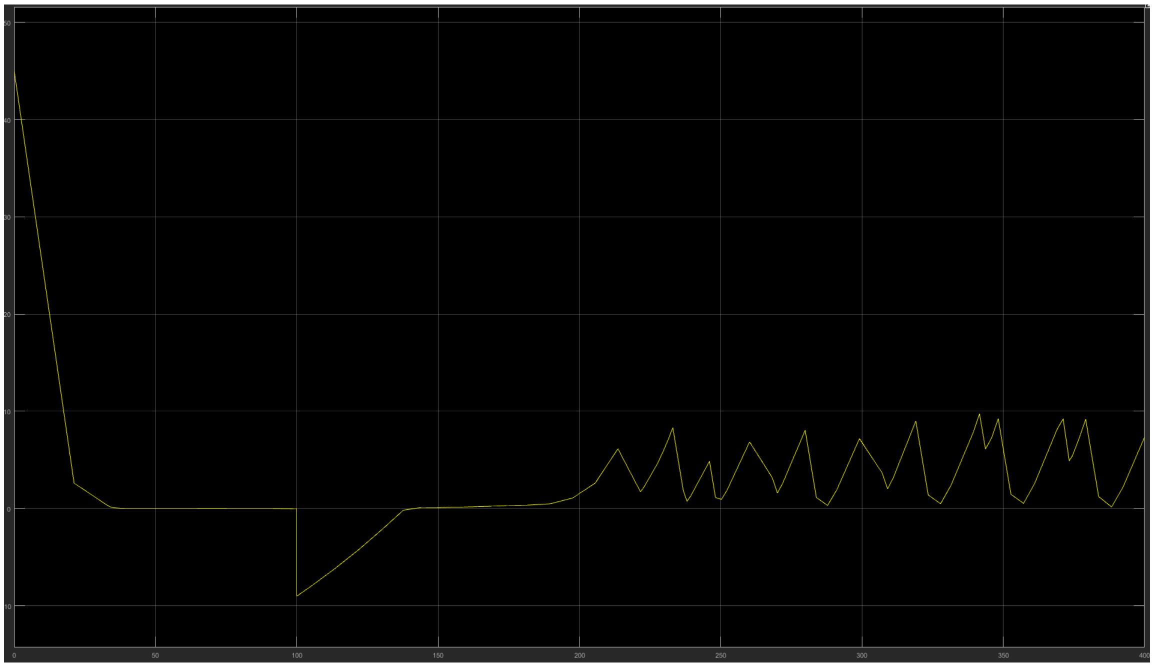

Figure 3.19: Plot of the theta error during a step disturbance

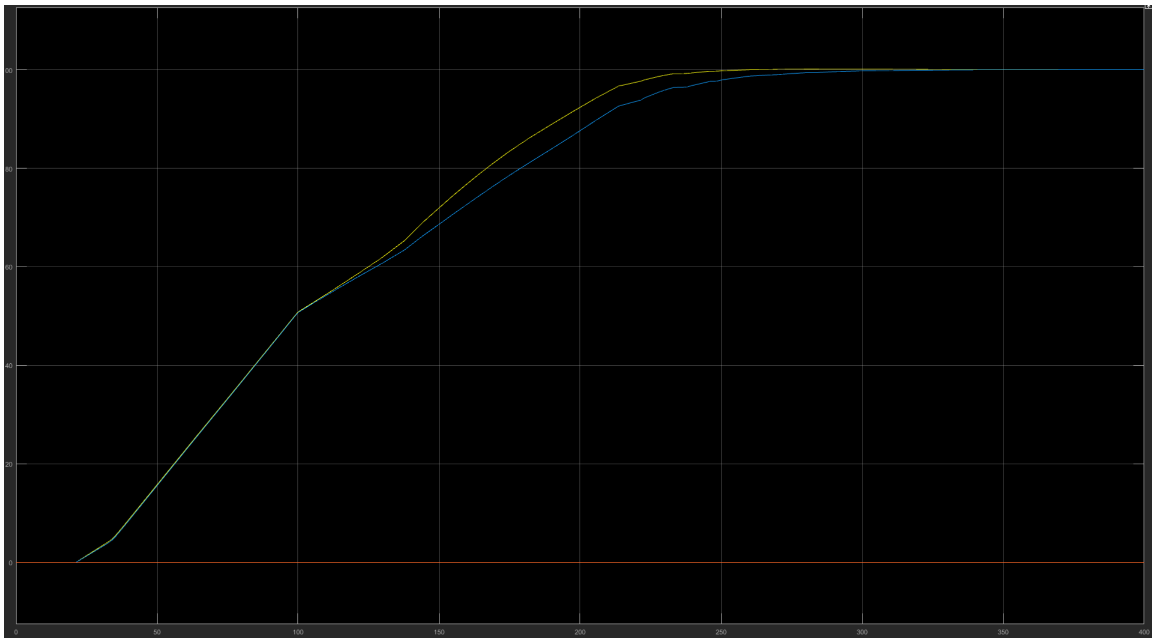

Figure 3.20: Plot of the XY values during a step disturbance

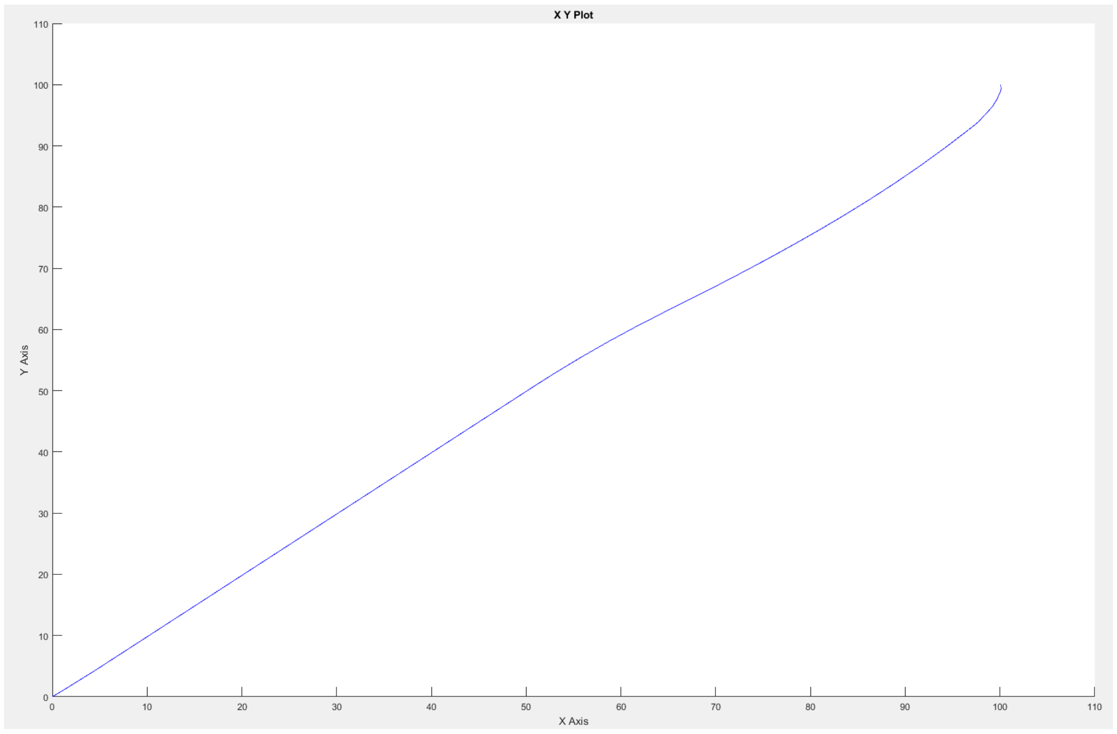

Figure 3.21: Plot of the robot trajectory during a step disturbance

3.3.3 Implementation on the robot

The implementation on the robot was a more arduous task than at first assumed. Although the only thing needed to change in theory was the control algorithm, it turned out that a lot of the system was built around the existing algorithm's properties. In this section we will go through the changes made to the code and the things one must keep in mind if one wishes to alter the control algorithm further.

Movement modes. The robot uses several status modes to describe its current action. These include, but are not limited to "moveStop", "moveForward", "moveClockwise" and "moveCounterClockwise" of which the last two describe pure rotation. These modes were in the original code used to send additional information between the tasks. More importantly, they were sent in such a way that prior knowledge of these states was needed by the other tasks when handling them. One good example would be the sensor tower task adjusting the speed of sensor tower rotation depending on whether it received the "moveForward" or "moveClockwise" / "moveCounterClockwise" status. However the speed of rotation for the robot was not included in the status message, and thus if the rotation speed changed, the speed of tower rotation had to be manually updated in the sensor tower task. Due to this we had to make ensure that all tasks who were dependant on this status did not use data from the earlier controller or was somehow dependant on the pure forward motion.

It was concluded that most of the tasks using this robot status would be replaced by the new controller and the only left-over still using the information would be the sensor tower task. We then decided that the practical differences between the earlier "straight" forward motion and our newer "curved" drive motion were similar enough concerning the sensor tower task that both could be described by "moveForward" in the new code. This meant we'd keep the motion status system for future use, as we saw it as a useful tool.

Merging MainMovement task and PoseController task. The old controller was mainly divided into two tasks: MainMovement and PoseController. MainMovement was in essence a motor PI-controller which simply controlled each motor to the same given speed. The problem with this task was that its entire function was to regulate the motors to the *same* speed, and all movement

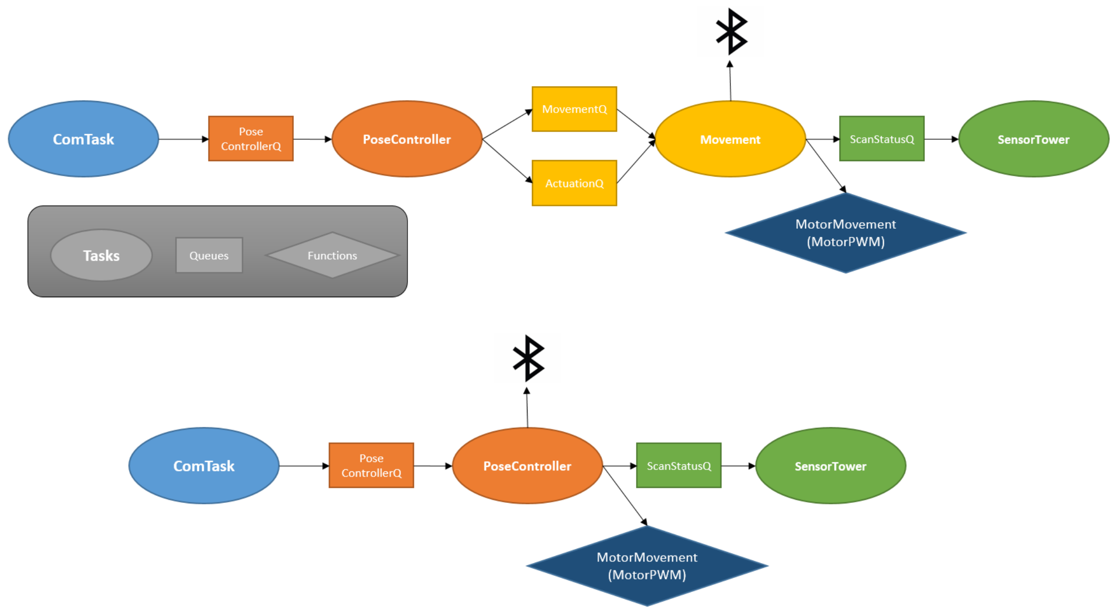

Figure 3.22: The old system around the pose controller over versus the new one under.

had to go through this task. With the old algorithm this was fine as it only needed the motors in the same speed but altering directions. However since we wanted to individually control the motors' speed, this entire task had to be redone. A more comprehensive description of this task is given in Ese (2016). The PoseController task contained most of the logic behind the controller itself; comparing current position with the set-point heading and distance, deciding whether the robot would turn or drive and at what speed, including a PI-controller for the distance. It also did important things like send "IDLE" messages and update the sensorTower task. However the way it was written was not easily reused and was centered around the idea of "first rotate, then drive a straight line". We concluded that these two tasks in essence was a part of the same job, and that the splitting of the two created a much more rigid system than we saw fit. We also concluded that they both needed to be completely rewritten for the new algorithm. Thus we removed the MainMovement task in its entirety and implemented the necessary functionality in a new PoseController task.

Global ticks handling. Another unexpected functionality of the main movement task was the handling of the global tick counter for the two wheels. For every iteration, it would take the values of a global interrupt counter, reset it and add the values to the global total ticks counter based on the direction of the motors. This was a simple addition to the new pose controller, but we mention that this might more suitably be added to a task of its own due to the separate nature.

Estimator synchronization Since the PoseController is most effective when synchronized with the PoseEstimator, we kept the earlier synchronization for the new task. We mention that this then also affects the frequency of the functionalities the Movement task had, since that task was independent of the synchronization between PoseController and PoseEstimator before. In particular this affects the update frequency of the global tick counters. However since the two task periods were so similar; 20ms vs 40ms we decided that this was a fair simplification. If it is found that this is not the case and the tick counters need more frequent resets, a reduction in the estimator period can safely be done, as we have a lot of computational power left according to the calculations in Ese (2016).

Chapter 4

Testing and Results

4.1 Left motor problem

We refer to the results from Amsen (2016) where it was shown that the robot was unable to drive in a straight line due to the cogs slipping as shown in figure 4.1. We show with the results following in this section that this problem has been completely eliminated after the added metal plate as can be seen from the straigh lines the robot drives in the tests.

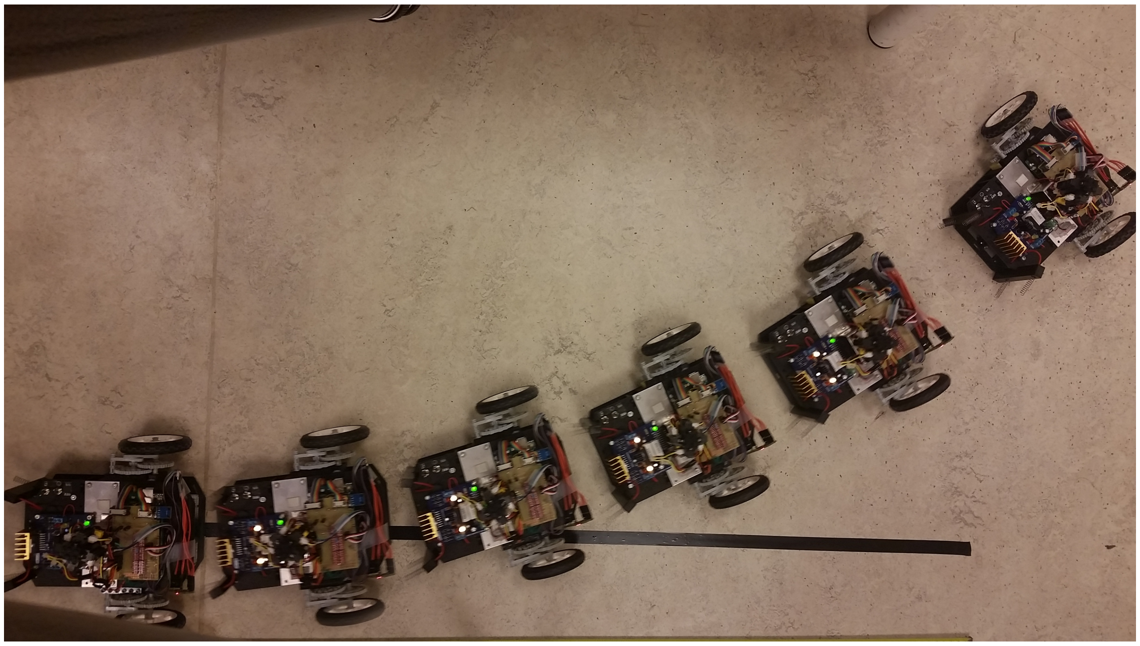

Figure 4.1: Picture showing the robot's inability to drive straight before the fix.

4.2 Rotation tests

After the slight improvements we made to the gyroscope, as well as removing the compass temporarily from the heading estimation, we ran a test comparing the robot's estimated heading. Figure 4.2 show the results of this test, the time has been skewed to better see the overlapping values. With the robot in blue and the Opti-track in red. During several executions of this test we observed varying accuracy and the one shown is of the worse results as we wished to shown the minimum accuracy we could expect from the robot. We can see that during the first turn we witness an error of 4 degrees on a 180 degree turn, or a 2.2% error. This doesn't sound like much, but can cause quite large errors the longer the robot drives. We would like to see better approximations, and we believe that with the right use of all sensors it is achievable.

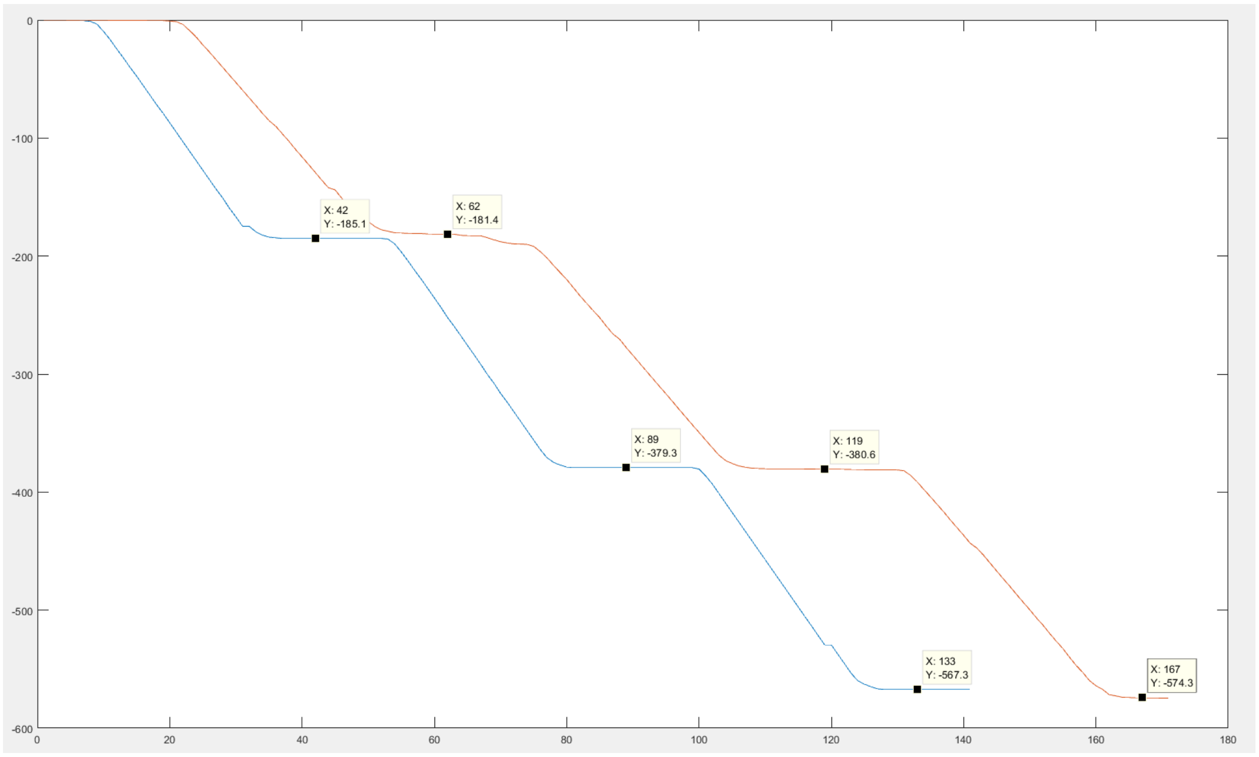

Figure 4.2: Comparison between theta estimation in blue and real theta in red.

4.3 Navigational tests

The main purpose of our new navigational implementation was to improve the robot's ability to utilize the knowledge it had. We found the earlier implementation to be insufficient in its navigational abilities. As a quick recap; the old navigational system worked by first rotating to a given heading, then moving forward a given distance. During the forward movement, no control of the heading was done to readjust any error in the heading. The new system is based on controlling the robot continuously during forward movement, to readjust any error in heading and utilize all knowledge of our position.

We also wished to benchmark the position estimation, thus all tests also contain the real world values.

The tests we performed were representations of the environment and the commands the robot could expect, while also being easily reproducible. They were all performed on the same surface at the "Slangelabb" room. All real values are measured using Optitrack.

It is important to remember how each target coordinate is calculated, as the commands of heading and distance can be misleading. The targets are calculated based on the robots last position and heading, and since the heading isn't a part of the last target/goal, the robot can find itself theoretically in any heading upon reaching the target coordinates. When the robot then calculates the next coordinate goal, it takes its current heading and finds the new target relative to it. Due to this confusion we advice the reader not to focus on the heading/degrees, but rather on the target coordinates the robot calculates and final coordinates it reaches, given in the tables with each test as they better represent the robot's performance.

4.3.1 Square tests

The square tests have been a very much used test with the LEGO-robot team. It makes for easy visualization of the robot's movement, cause no overlapping movement except at the end-points and is easily verifiable without the use of OptiTrack. The purpose with these tests was to get an overview of the robot's performance and see what further tests would be required. We mainly wanted to see how well the robot managed to navigate The first test was performed making a counterclockwise square starting in coordinate (0,0). The robot was then given the following commands:

1. Heading: 0 degrees Distance: 100cm

2. Heading: 90 degrees Distance: 100cm

3. Heading: 90 degrees Distance: 100cm

4. Heading: 90 degrees Distance: 100cm

The resulting goals and final destinations for each command is given in table 4.1, figure 4.3 and figure 4.4. Looking at 4.3, the first thing we notice is how smooth the square seems to be. Each line is straight, implying quick, accurate and constant regulation of theta. We also note that most of the edges are close to 90 degrees, and although this wasn't a requirement, it implies the robot needs very little final adjustments as it nears the target, rather hitting the target straight on. Each target is shown with a circle, and at a glance it seems all targets are hit very accurately. Looking at table 4.1 reinforces this notion. We see a maximum distance of 13 millimeters, and an average of about 8 millimeters. This is less than a centimeter and would be barely visible in the real world.

[mm]	x-goal	y-goal	x-final	y-final	Error
1st	1000	4	987	5	13.04
2nd	1000	1004	1000	999	5.00
3rd	0	992	4	992	6.40
4th	-6	-8	-7	-3	5.10
Avg					7.95

Table 4.1: Goal and final values of first square test.

Figure 4.4 shows the comparison with the real world position, the OptiTrack data shown in blue. We ask the reader to disregard the noise factors, as they are caused by errors in the OptiTrack system and does not reflect the robot's actual position. The cause for it is not known exactly, but we believe reflective surfaces on the robot have been mistaken for reflector points used to track the robot. We can see the first stretch being properly estimated by the robot, however a slight error in the heading estimation during the first turn propagates and causes some errors in the estimation further on. It's important that we don't mistake the propagation of one error, with steadily increasing error. This is the problem with dead-reckoning systems; one small error might cause great estimation errors as time goes on, even if everything goes perfect after it. After this, we can see from the almost parallel lines that the robot does a very good job of estimating the rest of the square.

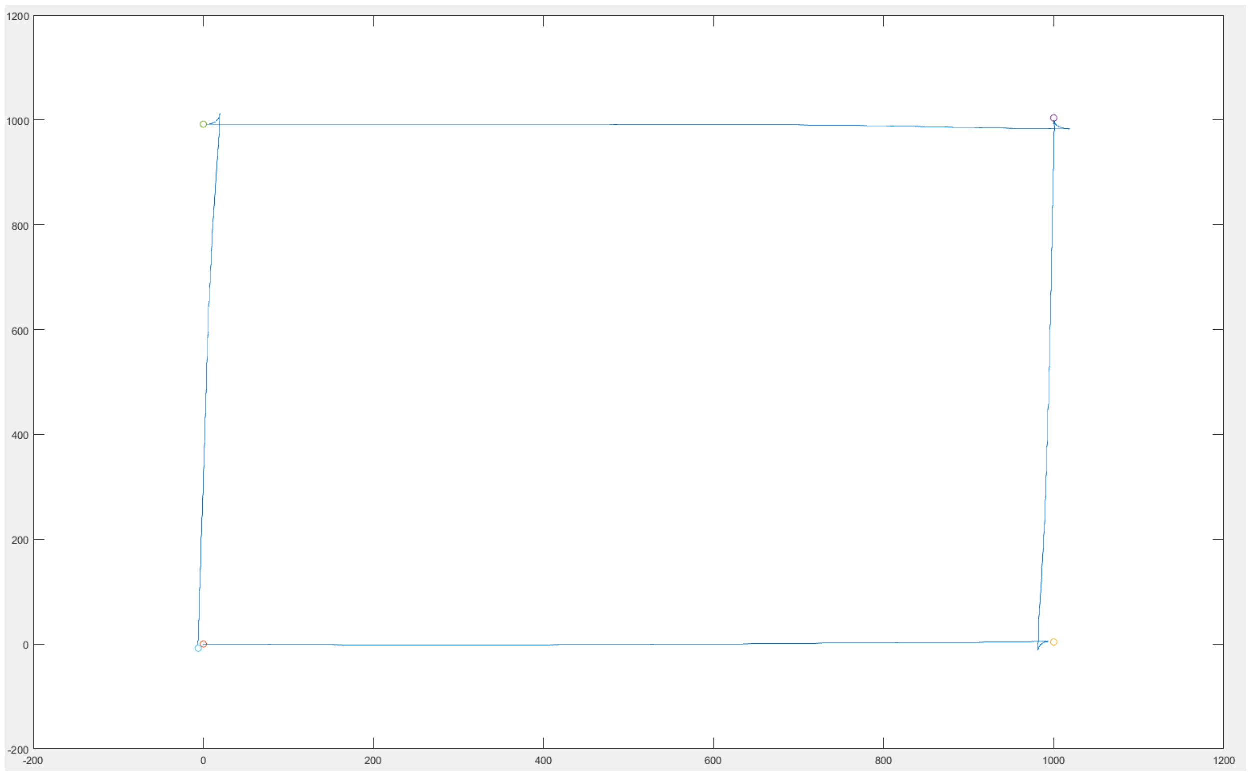

Figure 4.3: Counter-clockwise square test with navigational target goals as circles.

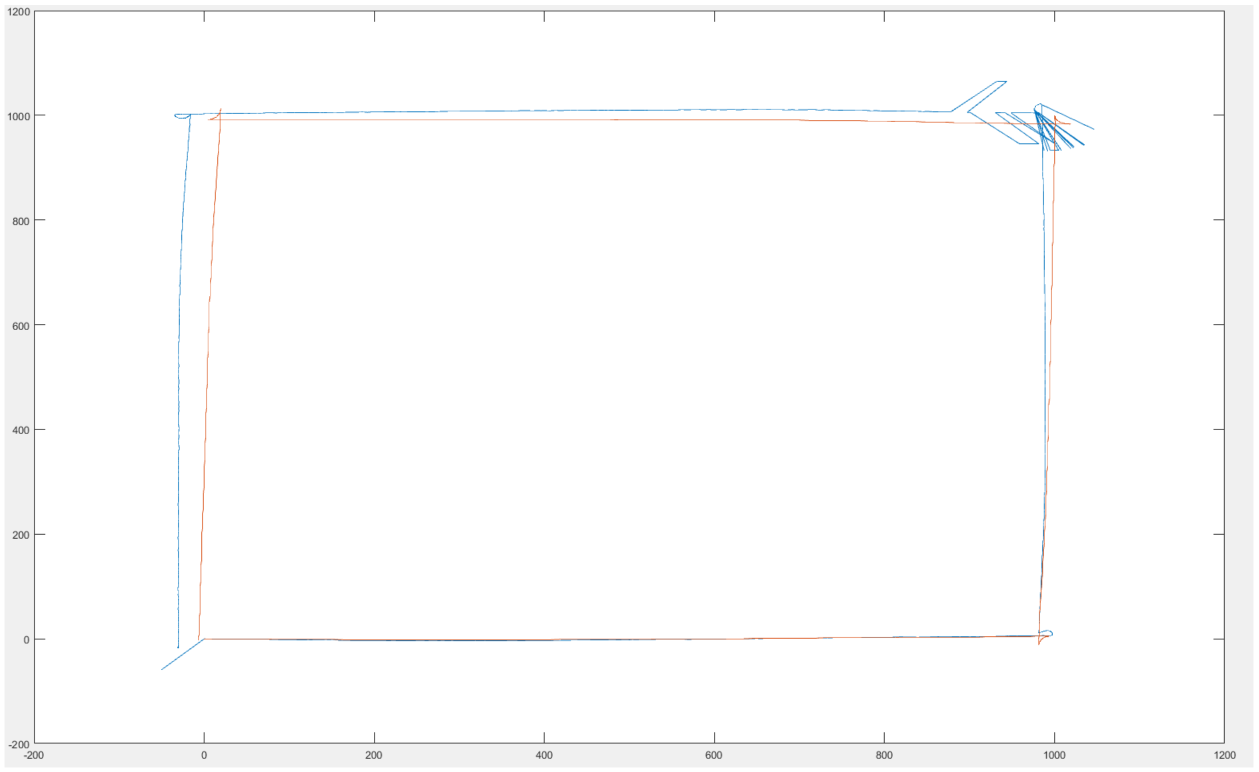

Figure 4.4: Counter-clockwise square test with OptiTrackin blue.

Due to the nature of the test, we decided that it was necessary to perform the same test the opposite direction, to ensure biases on the sensors don't yield false results. The commands given were the same as with the earlier tests, only each turn was now -90 degrees rather than 90. The results of the second test are shown in figure 4.5, figure 4.6 and table 4.2. As we can see, we achieve similar results as with the first test. All distances driven by the robot are almost perfectly 1m. As with the first test we see the robot make a small mistake in the heading estimation on the first turn, causing an estimation error of 6cm at the end point. However if we focus on the navigation performance, as shown in table 4.2 we see an average of 3.6 millimeter error. We can also see, as with the first test that all lines are almost perfectly straight, implying a shortest path taken between each point.

[mm]	x-goal	y-goal	x-final	y-final	Error
1st	1000	-3	998	-4	2.24
2nd	958	-1003	959	-1000	3.16
3rd	-41	-957	-37	-958	4.12
4th	6	42	6	37	5.00
Avg					3.63

Table 4.2: Goal and final values of the clockwise square test.

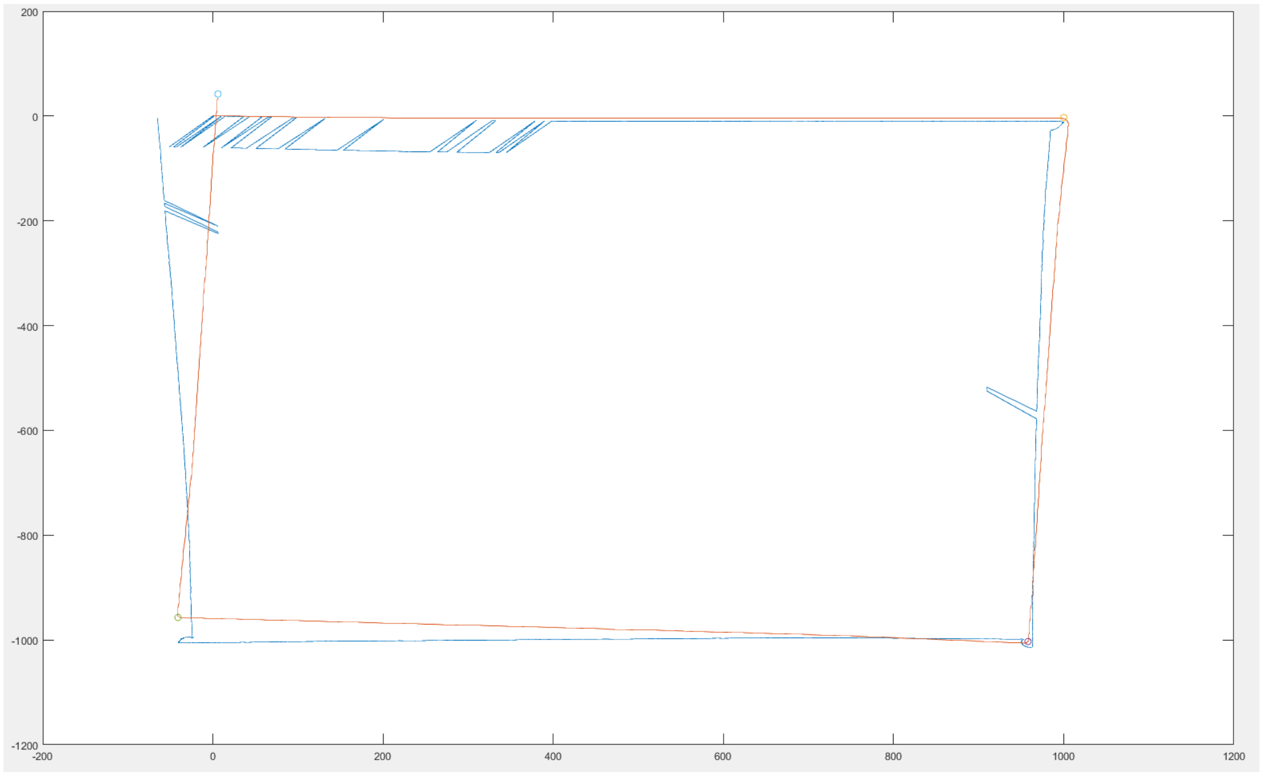

Figure 4.5: Clockwise square test with OptiTrack in blue.

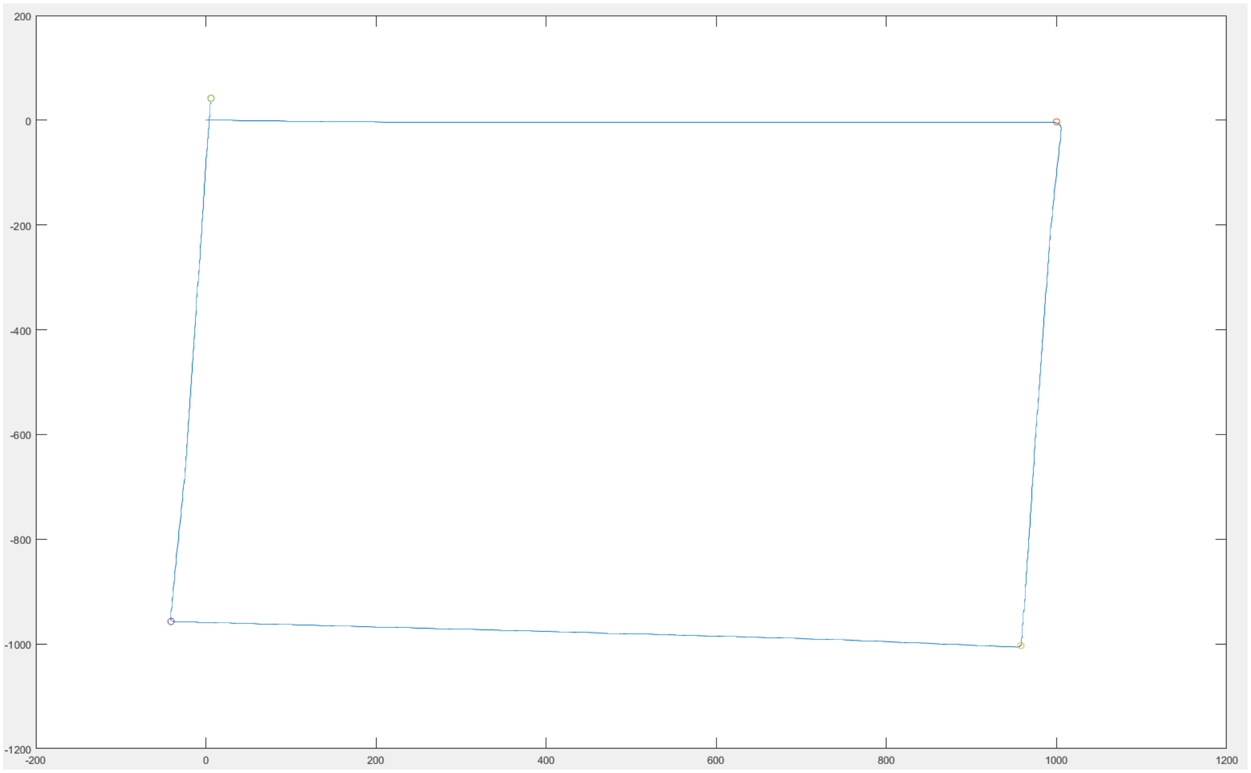

Figure 4.6: Clockwise square test with navigational target goals as circles.

To compare the new navigation algorithm with the old one, we ran the same test with the code implemented at the start of this project. The results are shown in figure 4.7 and table 4.3. At first glance it's easily seen that the results are very different from our other tests. Looking at the first move segment, we can see a curvature in the line. This shows the navigation algorithm did not sufficiently perform one of its two main tasks: making the line segments straight. The second thing we notice is that line segment #2 misses it's target goal by almost 25 centimeters. This as we see is mainly due to an error in the heading from the turning mode, but also is amplified by further curvature of the line segment. We observe the same faults in all later segments. The results of this is the robot not being capable of even remotely navigate according to the commands given by the server. If we consult table 4.3 we can see an average of 20 centimeter error, or 20% of the distance driven each segment, which is a substantial error, and will often cause crashing into walls and inability to navigate any tight areas.

[mm]	x-goal	y-goal	x-final	y-final	Error
1st	999	-35	1014	48	84.34
2nd	911	1042	667	1011	245.96
3rd	-244	598	-138	377	245.10
4th	557	-343	755	-115	301.97
Avg					219.34

Table 4.3: Goal and final values of the old navigator square test

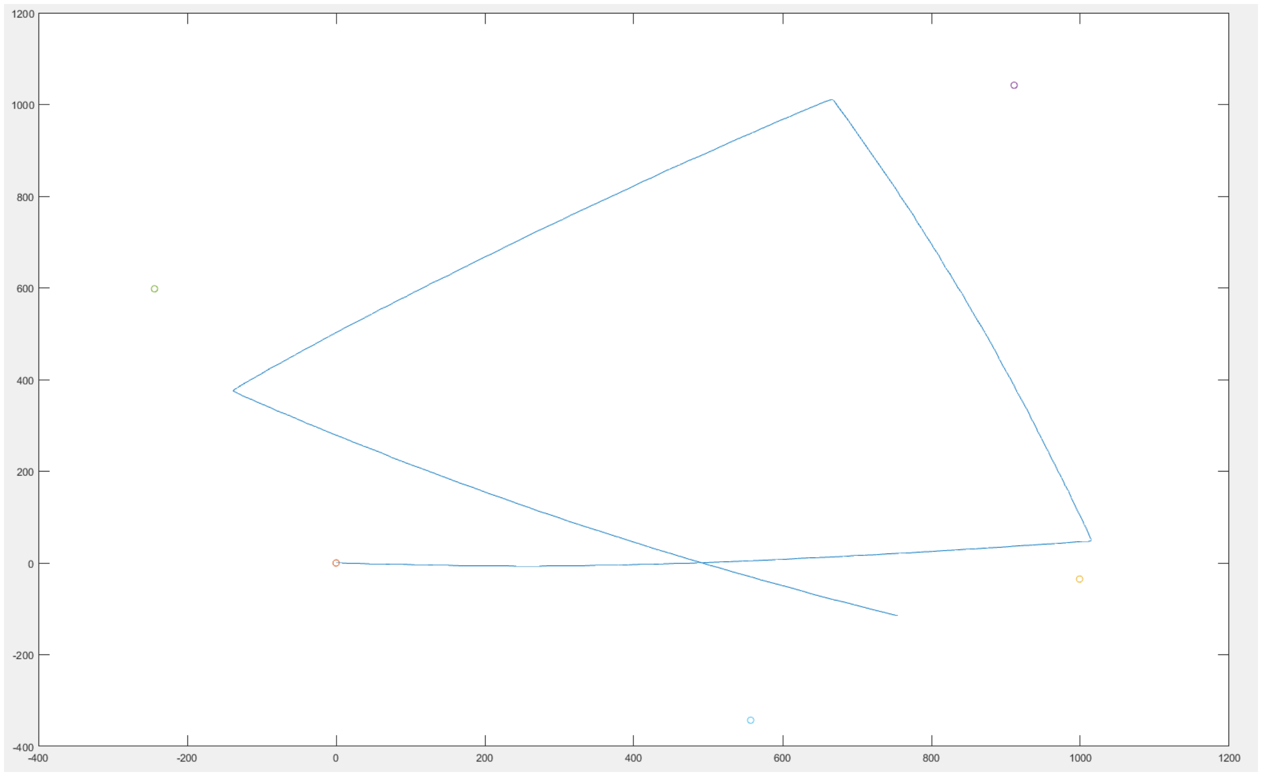

Figure 4.7: Square test with old navigational algorithm.

4.3.2 Longer movement tests.

A test more resembling the real world was also performed, this one with a focus on smaller distances and rotations in both directions. Below are the commands given to the server.

1. Heading: 0 degrees Distance: 50cm
2. Heading: -90 degrees Distance: 50cm
3. Heading: 90 degrees Distance: 50cm
4. Heading: 90 degrees Distance: 50cm
5. Heading: -90 degrees Distance: 50cm
6. Heading: 90 degrees Distance: 50cm
7. Heading: -90 degrees Distance: 50cm

8. Heading: 90 degrees Distance: 50cm

The purpose of the test was similar to earlier tests; to gauge the robot's ability to properly execute the commands given by the server, especially over longer distances. This time we put more focus on the error in heading, as it was shown in earlier tests to be the main source of error. We also wanted to see the effects of shorter distances moved, and if it made the navigational algorithm less effective. The results are shown in table 4.4, figure 4.8 and figure 4.9. We note that even though the distance driven for each move segment was halved, we don't see a reduction in the error. This implies that our controller works as well over great distances (1 meter at a time) as it does over shorter commands. This is important, as the server can give very varying distances depending on the area, and we need to have a dependable performance in accuracy.

Furthermore we focus on the heading estimation. We mention that the heading graph has been slightly modified to represent the real world, and can seem to give conflicting results from figure 4.8. As we see there seems to be an error already at the beginning of the run, however this is due to a fault in placement of the robot relative to the OptiTrack system and figure 4.9 shows the real error in theta. What we can see from this is that during the second turn we experience an error in the estimation of about 4 degrees, however this error is very consistent the rest of the run, implying that no further estimation errors are made. This is a promising result, as it implies not a consistent error in our sensors, but rather one-time occurrences we don't fix. This also further supports our idea of "resetting" the heading at certain points using a well-filtered compass measurements.

[mm]	x-goal	y-goal	x-final	y-final	Error
1st	500	-17	493	-17	7.00
2nd	476	-517	477	-512	5.10
3rd	977	-511	972	-512	5.10
4th	979	-11	980	-18	7.70
5th	1479	-33	1471	-34	8.60
6th	1481	467	1481	461	6.00
7th	1980	431	1974	432	6.08
Avg					6.51

Table 4.4: Goal and final values of long test.

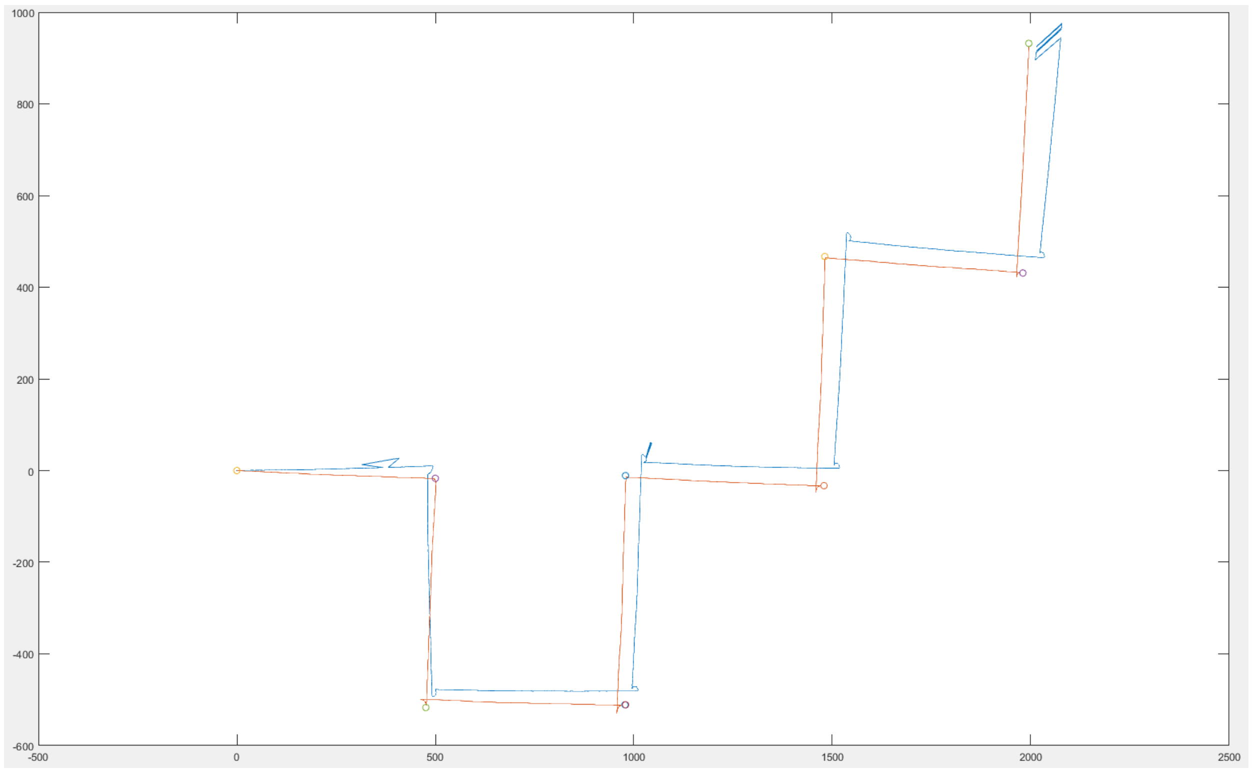

Figure 4.8: Position and real values of long test. Robot estimate in red, OptiTrack in blue.

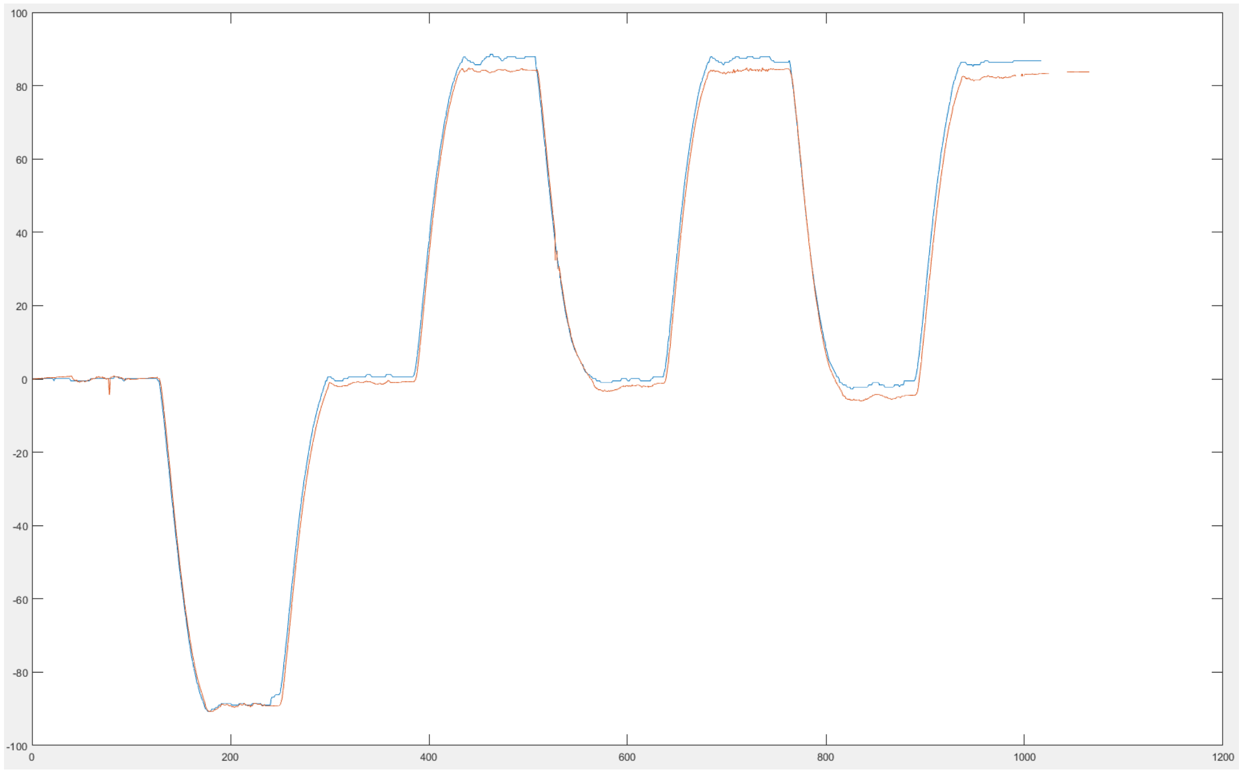

Figure 4.9: Theta estimate and theta real of long test. Robot estimate in red, OptiTrack in blue.

The same test with the same commands issued, was performed on the old navigation system for comparison. The results are shown in table 4.5 and figure 4.10. In this case we did not focus on the position and heading estimation, but rather on the differences between the old and new navigation algorithm. We see the same problems as with the earlier tests, where the initial heading is off, and the robot also has a constant increase in heading error as the distance is traversed, causing a curvature in the movement. We can also see that the distance traversed isn't as exact, especially from the first segment where it misses the distance by 1.4 centimeters.

[mm]	x-goal	y-goal	x-final	y-final	Error
1st	500	-15	514	-14	14.04
2nd	508	-513	446	-526	63.35
3rd	944	-575	959	-452	123.91
4th	863	40	740	17	125.13
5th	1181	251	1222	201	64.66
6th	1023	660	905	619	124.92
7th	1282	945	1334	908	63.82
Avg					82.83

Table 4.5: Goal and final values of long test for old system

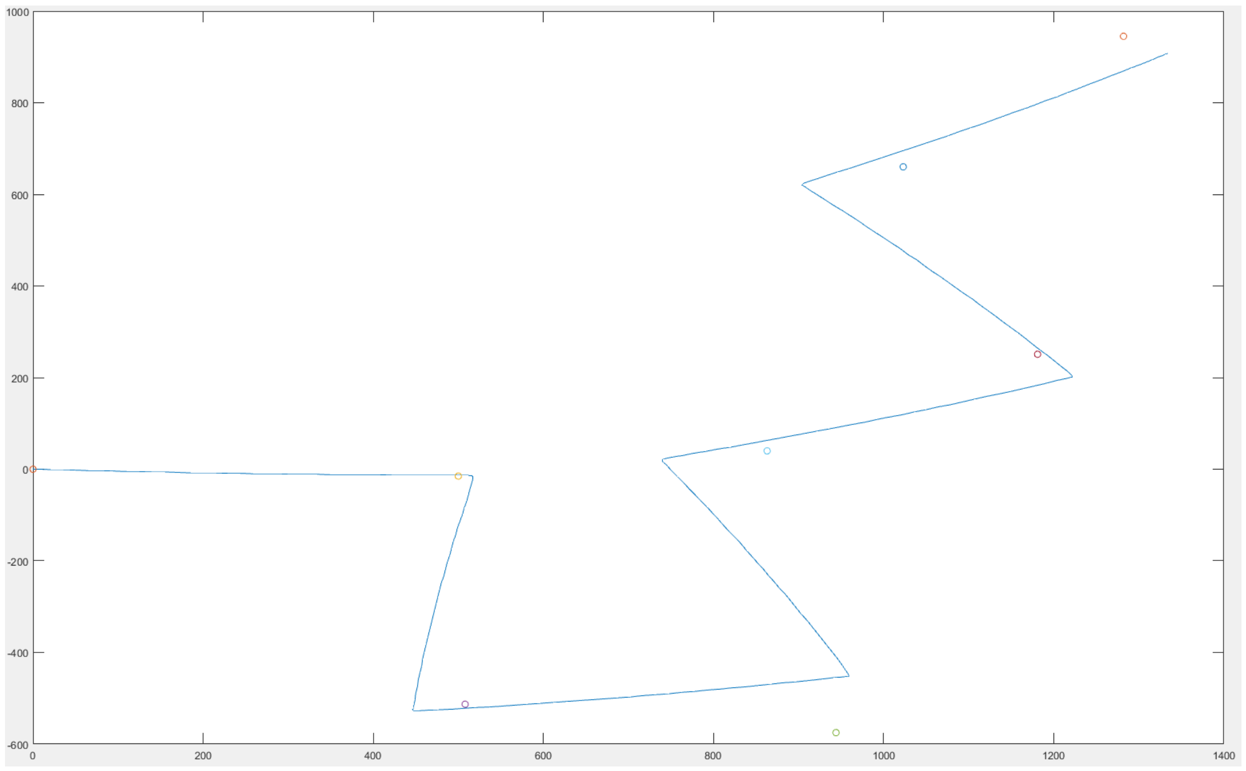

Figure 4.10: Position estimates and goal points as circles.

4.3.3 Full labyrinth test

A test was done to confirm the function of all robots and the server software in a real environment. A labyrinth was set up with focus on a size large enough for the robots to have to drive quite a bit, but also wide enough to ensure they got around. The labyrinth was set up in such a way for each robot to have their own area they could navigate and map before reaching the middle. Figure 4.11 shows the labyrinth used, the Arduino is the robot up to the right. The resulting map can bee see in 4.12. The AVR robot was disconnected before the picture was taken, but it was a part of the entire test. We can see from the map that the entire map was explored, meaning the robots found all openings and explored them fully. The reader might notice two mistakes in the map; the grey area between the Arduino and the rest, and the grey area to the right of the NXT. As can be seen in the picture, there are no walls here. These appear due to the robots discovering eachother and believing they are seeing a wall.

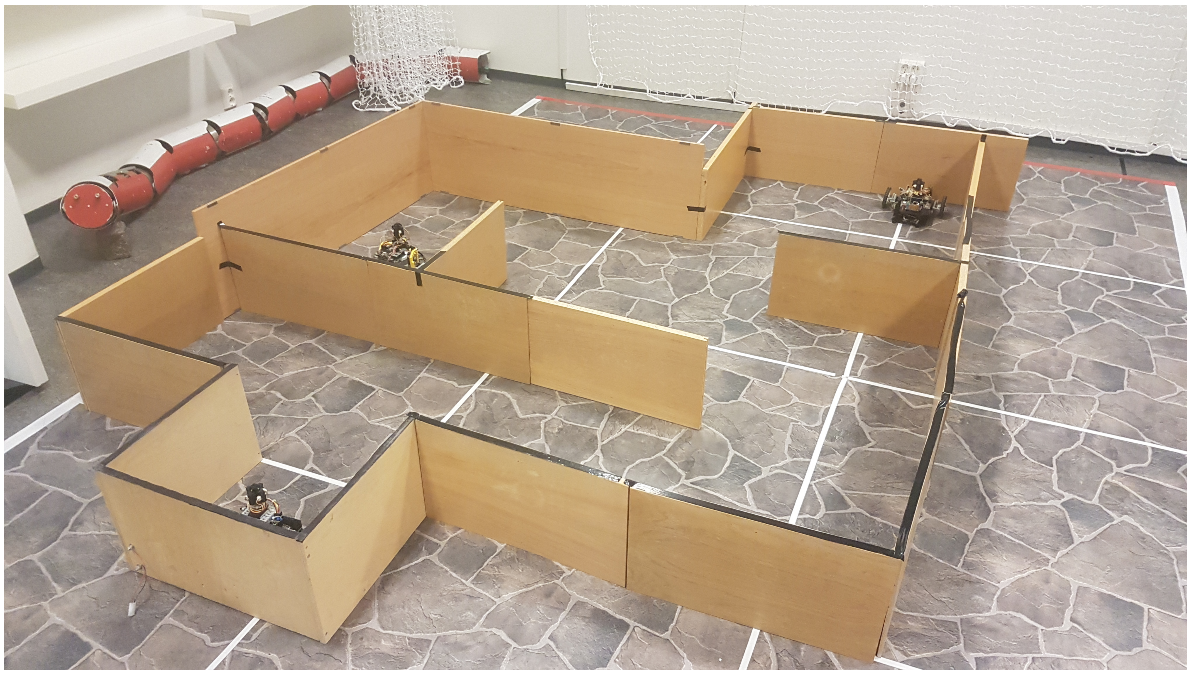

Figure 4.11: The labyrinth set-up with all the robots.

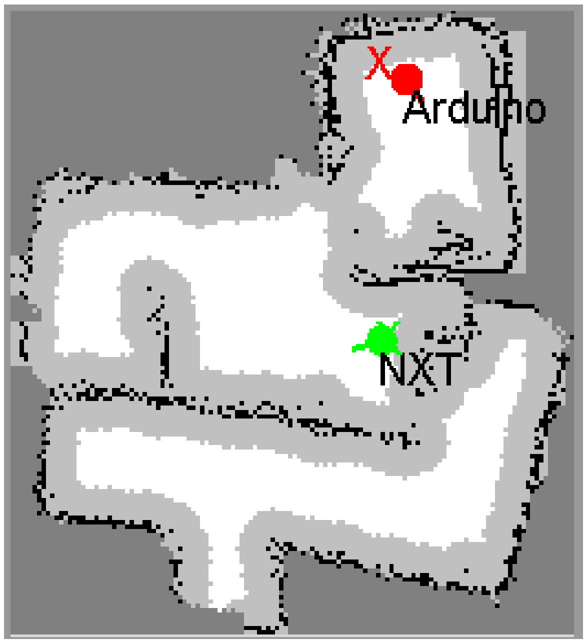

Figure 4.12: The resulting map from all robots.

Chapter 5

Discussion and further work

5.1 Discussion

5.1.1 Left motor fix

The problem of the left motor arose over time, and was observed in the middle of last semester. The exact cause for why it occurred suddenly is unknown, as no similar issue had been noticed earlier. We believe it was simply a matter of strain on the wheel armature over longer periods of time, and the initial configuration being inexact. This strain is something we've noticed before and voiced concern as the weight of the robot is being distributed on a longer arm, causing a greater moment on the armature. This can be easily seen by the wheels being skewed inwards when looking at from the front. It would seem that the metal plate completely fixes the issue of slippage, as it tightens the cogs and works as a sturdy support. However it does not solve the problem of strain on the armature. The question is of course if this strain can cause more problems in the future. We can for example see the possibility of the front of the robot no longer being supported enough to stay off the ground, causing it to catch on surfaces or small rocks. However as was discussed with the hardware shop at the Cybernetics department, we do not believe the same problem of slippage will occur on the right side, as their cogs have been much better fitted. We also believe disassembling and reassembling the armature more tightly is not worth the time. Finding a more secure solution to the down-gearing of the motors, or replacing the motors with ones having smaller RPM is something that could be considered in the future, however it is not at this point a necessity.

5.1.2 Improvements to the sensors

The work done during this project has not so much been to improve the sensors, as it has been to gauge and confirm their effectiveness and potential. From our results we've gotten a good insight into the encoder and gyroscope performances. In particular the encoder has been confirmed to be a much better source of data than we first assumed. The problems with slippage on the surface remain and cannot be removed, however we believe with more conservative uses of the motor power we have minimized the problem when no disturbances like walls are present. We also believe there exist several optimal wheel factors depending on the surface the robot works on and perhaps a system of discovering these could be considered.

The gyroscope was also confirmed to be very reliable, however we have kept it out of use during forward movement as drift seemed to make the results worse rather than better. We believe a proper intelligent filter instead of a simple fuse with the encoder could make the data valuable also during forward movement. It would be especially valuable to be used to filter out errors caused when the robot hits a wall, making the wheels slip on the surface and the encoder yield wrong heading, as this is currently the #1 problem with position estimation.

It was also decided that in it's current state the compass does not yield sufficiently accurate measurements to be used during rotation or forward movement. This is because it has a high noise factor, and since not enough measurements are made during rotation to be able to filter out the noise factor and also estimate the heading we decided to remove it all together. However the sensor itself is very valuable, and with proper filtering and usage can be used to great effect. We believe it can be effectively used to "reset" the gyro to avoid drift, but this should be used with a delay to allow for noise cancelling and thus might not be as valuable for instantaneous heading control.

The accelerometer is in much the same situation as the compass. We believe the data to be very valuable if used correctly. A simple double integration of the raw data proved to be almost usable only over short durations. We have theorized that a well-tuned low-pass filter could suffice, but from the few tests and analysis we did on the signal it seemed to be a complicated process, and too much high-frequency information will be lost. We also believe that a higher frequency of gathering accelerometer data might prove valuable as this might improve our estimations during the rapid movements. The true value in the accelerometer lies in short-duration use. During the start of movement or to detect crashes and prevent false data from the encoder due to slippage. It was mentioned in Ese (2016) that the accelerometer might be used to automatically detect the wheel factor of the encoder, however we believe that it is too inaccurate for this task.

5.1.3 Navigation algorithm

The changes in the functionality of the software has not been minor, but from tests of the system it has been shown that the new software is stable and reliable. The new algorithm has been implemented smoothly in the already existing code and works well with the system as a whole. There might still be improvements to the software, and certain functionalities, like the use of robot states should be replaced by more dynamic approaches, as changes in the control algorithm would require updates, and not all algorithms would work well with the current states.

When we try to improve the robot's ability to accurately follow server commands, there are two areas of fault; the navigation control and the position estimation. Each of these work independently and can add their own part to the error. This means that even if the robot has a perfect estimation, the control must still correctly navigate, and even if the controller properly navigates, if it navigates based on a faulty position estimate, it still won't hit the true target in the real world. In this project we have focused on the robot's navigational control, as we found that the existing controller did not use the information it had sufficiently. More importantly, it didn't navigate to a coordinate, but rather based on heading and distance. This, despite the server internally using coordinates to control the robots. Due to this we decided to change the control algorithm to reduce the *known* error to the target as much as possible. In the previous chapter we can see the difference in performance from the old and the new system. We can see that the error is down to only a few millimeters which is much better than we initially expected. The constant control of heading removes the problem where the forward movement curves away from the target. We also see quick, smooth and damped control, without any visible overshoot or jittering. The current tuning has been done according to the current speed, and it's worth noting that increasing the speed will increase the momentum of the robot and thus also its possible overshoot when stopping.

One can argue that the robot's ability to navigate very exact isn't as important since the mapping mainly relies on the position of the robot, regardless of that position is exactly what the server commanded. However after several tests in a proper labyrinth, we've observed the importance of accurate navigation and not hitting walls. Overshooting the server command by only 10cm can mean the robot crashes into the wall and corrupts its position estimate.

5.2 Recommendations for Further Work

Improve collision detection Currently the Arduino robot is a lot wider than the other robot, and often has trouble navigating corners in the test labyrinths, a better anti-collision should be implemented.

Improve position estimation Currently we have a lot of good sensor data, however the robot doesn't use this data in an intelligent way. a Kalman filter and some intelligent algorithms can greatly increase the position estimation, as well as reduce the error caused by crashing and surface slippage. Also focus on adding both the compass and the accelerometer data to the position estimate in a usefull way.

Isolate accelerometer Vibration of the robot might be the cause of a lot of the accelerometer noise, vibration isolation might help.

Hardware issues The robot has several hardware issues that should be fixed, however aren't at the moment critical.

- Fasten bluetooth dongle properly
- Reinforce the metal ball to lift the robot higher off the surface
- Reinforce the wheel armature
- Fasten connectors better, especially at the battery and sensor tower
- Fasten loose charging towers

Add the navigation algorithm to the other robots

Improve bluetooth connection The bluetooth connection seems to be very unreliable, and it doesn't take much distance or material between the dongles before the connection falls out.

Appendix A

Additional Information

A.1 Software

A.1.1 Termite

Termite is an intuitive and easy to use RS232 terminal. It focuses on string, in particular terminated with endline. It alllows us to communicate directly with a port of our choice connected to the computer. It is capable of sending and receiving string messages. In particular we can use this to manually communicate with the Bluetooth dongle in this project to easily debug and view the messages going on behind the scenes.

To connect with the Bluetooth dongle using Termite we used the following settings:

- Baud rate: 38400
- Data bits: 8
- Stop bits: 1
- Parity: None
- Flow control: None
- Transmitted text: Append LF

Bibliography

Amsen, J. (2016). Navigation and mapping with arduino robot.

Andersen, T. E. S. and Rødseth, M. G. (2016). System for self-navigating autonomous robots.

Ese, E. (2016). Sanntidsprogrammering på samarbeidande mobil-robotar.

J. Borestein, H. R. E. and Feng, L. (1996). *Where am I? Sensors and Methods for Mobile Robot Positioning*. University of Michigan.

NationalInstruments (2006). Magnetic Encoder Fundamentals.

Tusvik (2009). Prosjekt: Fjernstyring av legorobot.