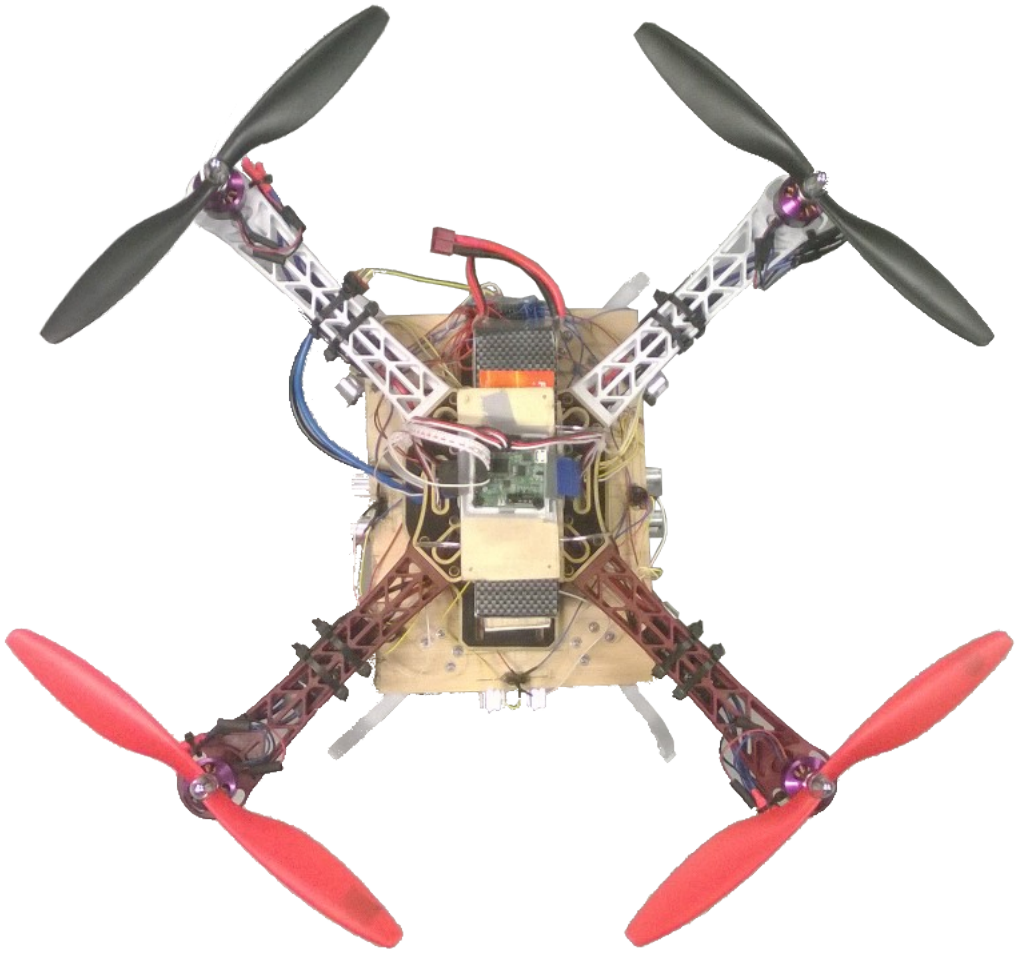
Autonomous Quadcopter
- For mapping an indoor space



21. december 2015

Group SW511-E15

Aalborg University

AALBORG UNIVERSITY

STUDENT REPORT

Institut for Datalogi
Selma Lagerløfs Vej 300
9220 Aalborg Ø
Telefon (+45) 9940 9940
Fax (+45) 9940 9798
http://cs.aau.dk

Title:
Quadcopter

Theme:
Embedded system

Project Period:
Fall 2015

Project Group:
SW511E15

Participant(s):
Alexander Kaleta Jensen
Andi Rosengreen Kjærsig Aaes
Kaare Bak Toxværd Madsen
Malthe Dahl Jensen
Niklas Ulstrup Larsen
Thomas Bjelbo Thomsen

Supervisor(s):
Xike Xie

Copies: 8

Page Numbers: 175

Date of Completion:
December 20, 2015

Abstract:

This paper covers the development of an automated quadcopter that should have the ability to automatically fly in an intelligent manner, while mapping its environment and avoiding obstacles. The development of the system is divided into four main tasks with each task focusing on a different part of the final system. The four tasks are completed in order of importance with each task dependent of the previous task. The first task being stabilization of the quadcopter, second task being movement and obstacle avoidance, the third task being the system having knowledge of its own position and the fourth task being mapping of the environment.
The system hardware in this project has been chosen from an analysis of the requirements and a comparison of different hardware opportunities to find the hardware components that best satisfy a solution to the system requirements. Further tests on the different hardware and software components have been performed to modify them to give the best results as possible.

Signatures

Alexander Kaleta Jensen

Andi Rosengreen Kjærsig Aaes

Kaare Bak Toxværd Madsen

Malthe Dahl Jensen

Niklas Ulstrup Larsen

Thomas Bjelbo Thomsen

Preface

This report is written and developed by project group SW511E15 from the fifth semester of the software study at Aalborg University. The project began on the 2. September 2015, and was completed the 21. December 2015. The theme of the project is *Embedded Systems* and the specific problem is to create and implement an autonomous quadcopter.

This report uses `monospace` for visualization of functions, methods and variables relevant to the software used in this project. For citation, the Vancouver method is used, where sources is indicated with a number in square brackets (i.e. [2]), and comma separation if multiple sources. The title, author(s) and other relevant information can then be found in the bibliography.

Throughout this report several abbreviations will be used. The abbreviations will be described when they are used in the report, but for good practice the most frequently used will be stated here as well:

ESC Electronic Speed Controller

IMU Inertial Measurement Unit

PID Proportional–Integral–Derivative controller

Contents

Part I

Introduction

UAVs (unmanned aerial vehicles) have been around for many years, mostly used by enthusiasts as small toys, but in the recent years, UAVs have been used in a wide variety of situations. In several countries, including the United States, UAVs are used for military purposes[1], and Amazon, one of the largest internet-based retailers in the world, recently announced that they are working on a system, using drones for delivering packages on very short notice, called Amazon Prime Air[2].

The prices for UAVs, especially small quadcopters, have been dropping drastically in the recent years, making them more available to the general public. This opens for new opportunities to use them for more than just toys and fun. The idea for this project is to implement an embedded system using a quadcopter as the platform. The purpose of this embedded system, at the final stage, is to map out rooms in a building, by itself, without the assistance of human operators controlling the quadcopter.

This will require several pieces of hardware equipment, that must be able to communicate with each other. Things like Arduino, motors, Electronic Speed Control (ESC), inertial measurement unit (IMU), proportional–integral–derivative (PID) controller, flight controller, gyroscope, accelerometer, battery, Bluetooth, distance sensors and software will all have to connect some way or the other, for this project to succeed.

Embedded Systems

Technical equipment and devices today is mainly controlled and regulated by embedded software, which can be anything from refrigerators to small wrist watches to intelligent software in modern cars. It is therefore essential, when developing embedded software, to take into account some of the boundaries that exist, as the small computers have a limited amount of computational power. Embedded systems cover areas such as robots, control systems, sensors, actuators and more. Embedded systems such as these must be efficient, as they may include real-time constraints, whether this may be code size efficiency, run-time efficiency, cost efficiency or energy efficiency. The embedded system is a microprocessor based system, that is built to control a range of functions, and as such it is not designed to be used as a platform for end user programming as with PCs.

For embedded systems in this report, the Berkeley definition, by Edward A. Lee, is used; "An embedded system is software integrated with *physical* processes. The technical problem is managing *time* and *concurrency* in computational systems".[3, 4]

"... An embedded system is designed to perform one particular task albeit with choices and different options."[5]

0.1 Initial problem

Designing for embedded systems is different than developing for modern personal computers for several reasons. First of all, the limited memory and processing power makes it very important to make the system run efficiently. Furthermore there must be both sensors and actuators to make it possible for the system to interact with its surroundings, and as the sensors and actuators can be inaccurate, they must therefore

be tested extensively. Given these premises and the description in section I this project focuses on the following problem:

How can we implement a quadcopter as an embedded system, capable of flying around and exploring an area in an effort to map the surroundings, while being able to stabilize and avoid obstacles, by itself?

0.1.1 Tasks

In order to implement this embedded system, we have concocted four sub-tasks that together mount up to the desired embedded system. These tasks will serve both as a guide to the progress of the project, as well as milestones which we will try to achieve. The list of tasks is a prioritized list with task number one being prioritized as the most important task. Each of these tasks will lead to an increment of the project, based on the incremental development model as described in section I, meaning we will start by making a working version of the first task, before moving on to the next task, which will utilize the existing implementation in order to implement a new increment. Each task will now be formulated:

1. **Quadcopter stabilization.** In order to fly without crashing, the quadcopter must be able to stabilize itself, meaning it needs to be able to stay relatively still while in the air.

2. **Quadcopter movement.** The quadcopter must be capable of flight in all directions (up, down, left, right, forward, backward), as well as yaw rotation. The quadcopter must be able to do this automatically, and by external input. In order to test this task, quadcopter takeoff and landing must be accomplished as well.

3. **Obstacle avoidance.** The quadcopter must be able to detect obstacles, and avoid them as necessary. Being able to detect obstacles is very important, as the quadcopter is supposed to fly around without help from any users. This is especially true if the quadcopter is flying indoors, as there are a lot of close obstacles, which increases the likelihood of the quadcopter crashing.

4. **Automatic exploring and mapping flight.** Using all the previously completed tasks, we also want the quadcopter to explore an area somewhat smart by covering the area without flying over the same spot more than necessary. This area will then be mapped, determining where there may be obstacles.

This prioritization has been made based on the following thoughts: In order to implement and test the movement of the quadcopter, it is important that it can stabilize itself. Without this feature it would be very difficult for the quadcopter to fly around. The quadcopter being able to move around is a prerequisite to being able to detect obstacles, since without this, the quadcopter would never be in a situation where detecting obstacles would be useful. The first two tasks would also need to be completed before automatic exploring flight can be tested, since it has to be able to stay in the air, and move around. Obstacle avoidance is also a prerequisite of doing automatic exploring flight, since being

able to avoid objects automatically, will allow the quadcopter to fly in areas which are not void of obstacles.

Each task also serves as a natural project boundary, as each task has some prerequisites that have to be fulfilled before the task may be implemented in a complete embedded system. However, if at some point a task is in danger of not being implemented, it is possible to instead simulate that specific task, and use that simulation as a basis for tests. E.g. task 3 simulation could be achieved by walking around with a setup approximating that of the embedded system setup, instead of testing with the quadcopter flying around.

0.2 Software requirements specification

In this section the requirements for the system will be stated. The requirements are found using the methods described by Sommerville[6]: *"Discovering requirements by interacting with stakeholders (elicitation and analysis); converting these requirements into a standard form (specification); and checking the requirements actually define the system that the customer wants (validation)."*

Since this system is not being developed in cooperation with a consumer, and the only stakeholders is this project group, the elicitation and analysis is very straight forward.

The requirements specification, also helps to eliminate the risk of misunderstandings towards what we are working to achieve. We will divide the requirements specification into two parts; the functional and non-functional requirements. The functional requirements are statements of services the system should provide or do, and the non-functional requirements are constraints of the solutions or functions.

Functional requirements:

- The system must be easy to setup and use.
- The quadcopter must be able to takeoff and land on a plane surface.
- The quadcopter must be able to fly in all direction.
- The quadcopter must avoid obstacles on its way.
- The system should discover and map the room structure on its way.
- The system should prioritize safety over speed.

Non-functional requirements:

- The system should work with no human interaction, besides the command to takeoff and land.
- The system should only map free areas and obstacles.

- The quadcopter should adjust the motor speed to stay to its current direction at least every 50 ms. or less.

- The quadcopter should respond quickly to a panic or emergency stop, within 50 ms.

- The software must not exceed the memory limit of the Arduino MEGA.

Scenario

In this subsection we will describe an idea for what the quadcopter could be used for.

A quadcopter controlled by a micro-controller and distance sensors, to be used for clearing and searching out empty, destroyed and/or dangerous buildings before people go in. The quadcopter will gather information and be able to give back a map/overview of the building, to the people using it as a reference for when going in. It will communicate with a computer, controlled by a person, where it will be able to deliver the map/overview. A helpful tool to make the job safer for certain people.

0.3 Process model

In this section, the process model used to write, design, implement and test this project, will be described. The advantages and disadvantages of the chosen model will be discussed, and the reasons for why we have chosen it will be stated. The foundation of our choice is based on Ian Sommerville's description of the different process models and development approaches[6].

For the overall approach to this project, we have decided to focus on a plan-driven approach. We will split the project into smaller secluded tasks, where each task will be less plan-driven and more based on tests, but for solving the problem statement and the overall project, we will be using a plan-driven approach. This means that we will have a requirements specification, as used in traditional plan-driven approaches.

As for the process model, the incremental development model is used. This model combines the elements of the waterfall model, with an iterative philosophy, as seen in figure 0.1. This means that we get through some of the same stages as in the waterfall method, but do it multiple times. We have chosen this model, because it fits well with our expectation of developing this system in multiple tasks. The aim is therefore to have each task corresponding to one iteration. This way the product will improve with each iteration, and each subsequent task is an increment of the previous task, thus responding well with the incremental development model.

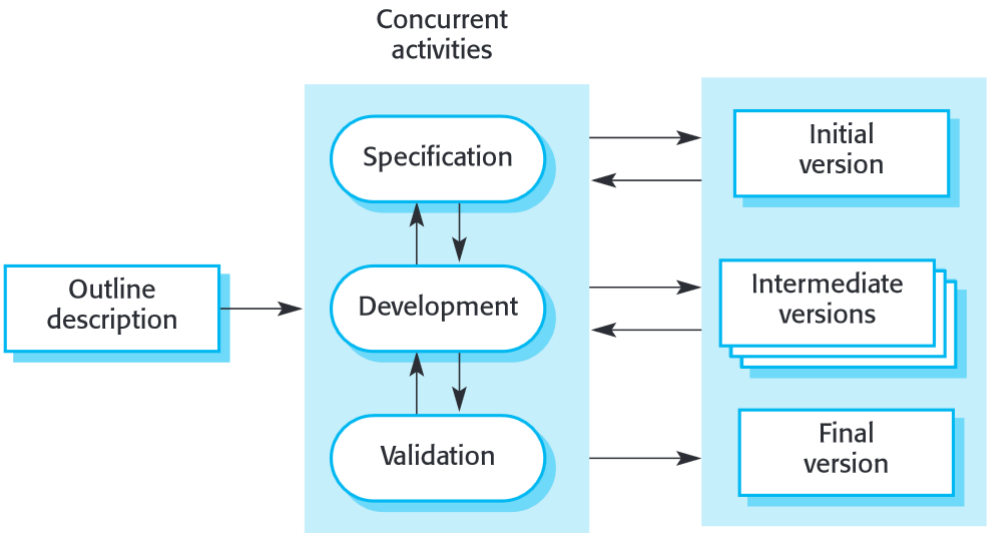


Figure 0.1: The incremental development model[6, p. 50]

The advantages of the incremental development model include:

- After each iteration, testing will be conducted. This allows for quick identification of the problems, because fewer changes are made within a single iteration.
- Initial product is faster, enabling for sooner testing and analysis of the product.
- Each iteration serves as a natural project boundary, where argumentation for changes can be made for each task, based on project status, if there are issues.

The disadvantages of the incremental development model include:

- Issues with a task will always delay the subsequent tasks, as these tasks require the complete implementation of the previous tasks. These issues therefore create bottlenecks that have to be solved as soon as possible.

In our project, each iteration will represent one of the tasks described in section 0.1.1. Looking at figure 0.1 the *intial version* will be after task 1, the *intermediate versions* will be task 2 and 3, and the *final version* will be task 4. We modified the model to fit our process better, and will now describe these modifications.

Looking at figure 0.1, the outline description includes the initial project domain specifications and the problem statement, as well as the conclusion of the project as a whole. The specification part in concurrent activities includes analysis, the development part includes design and implementation, and the final part, validation, will be a verification part instead, as this project focuses not on customers, but rather on project requirement specifications that have to be met.

Part II

Task One

Introduction 1

In this chapter, the first task, which is to get the quadcopter to stabilize in the air, will be described. There will be an elaboration of the challenges in this task, followed by an analysis of the Arduino microcontroller board, that will be used to control the quadcopter.

We will analyze how stabilization can be accomplished, based on the hardware that is chosen to complete this task. This chapter includes different kinds of tests of the used hardware pieces such as the chosen IMU, the PID and the flight controller.

This part will determine the system design for task one, and it will include relevant theory and argumentation for the task. Each sensor and actuator used in task one will be tested, and this part will result in an implementation of a subsystem of the complete embedded system, described in section I, that fulfills the requirements of task one.

1.1 Requirements

In this section the requirements for achieving the desired functionality in task one will be stated.

- The system should be functional using nothing but the hardware and software placed on the quadcopter frame itself.
- The quadcopter should be aware of its alignment to gravity, and then use that information to change the speed of the motors, based on the difference between the targeted alignment and the actual alignment.
- The system should be able to carry and lift the hardware needed to complete this task and future tasks.

These are the requirements that have to be fulfilled in task one, before it is possible to further implement task two.

1.2 System design

This section will describe the system of task 1. This includes choosing the various components of the system, such as the quadcopter frame, the computational platform,

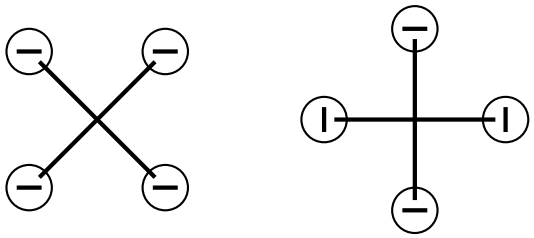


Figure 1.1: The two quadcopter configuration options

various sensors and actuators. After components have been chosen, they will be assembled into a single system.

1.2.1 Basic quadcopter physics

In this section information about the physics that concern the quadcopter will be described, as well as how it is possible for it to move around in the air.

As a source of information for this whole section, these two references have been used: [7, 8]

Quadcopters are mechanically simple, in the sense that they have four motors and four propellers, hence the name **quad**copter, where "quad" means four. For them to fly and move around, all that has to be altered is the rotational speeds of the propellers, each powered by a motor.

The configuration of the motors, is either in the formation of a '+' or an 'x', see figure 1.1 for clarification. This makes it somewhat easy to find the center of mass of the quadcopter, if the controller is placed in the middle. If the center of mass is not in the intersection of the configuration, the motors will have to produce different thrust to balance the quadcopter, which can be somewhat difficult to calculate.

A quick way to figure out if the center of mass of the quadcopter is at the intersection, is by grabbing on to two opposite ends of the quadcopter and see if it tilts to either side. By doing this on each set of ends, you will have an idea of where the center of mass is.

It is not only the center of mass that is important in making a quadcopter capable of flying. The direction the propellers rotate is also important. It is required that one pair of propellers give thrust by rotating clockwise and the other pair counter-clockwise. This is needed so the quadcopter will remain still and not spin in circles, which will happen if all propellers rotate in the the same direction. This is because of Newton's third law[9] that states *"if a body exerts a force on a second body, the second body exerts a force that is equal in magnitude and opposite in direction to the first force. So for every action force there is always a reaction force."*, which applied to rotational force, is called torque.

This means that the pair of propellers that rotates counter-clockwise, applies torque to the air in a counter-clockwise direction and air applies an equal and opposite reaction torque, which will push the quadcopter in a clockwise direction, see figure 1.2. It is also needed for the one opposite pair of propellers to have opposite blade pitch with regards

to the other pair, resulting in the thrust having the same direction. This is because of the two pair of propellers rotating in different directions.

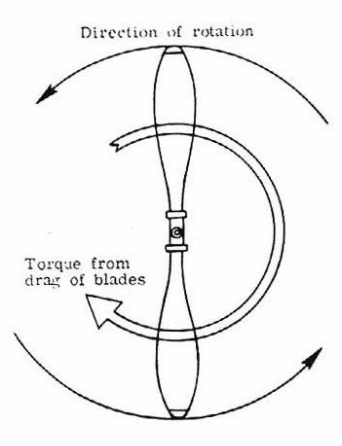


Figure 1.2: Showing propeller rotation and torque motion

Newton's third law also comes into effect when the quadcopter needs to hover. When the propellers spin, they push downward on the air around them, and the air applies an equal and opposite reaction force on the propellers. When this force equals the force of Earth's gravity, the quadcopter will hover.

When the quadcopter hovers, all it takes for it to move in any direction is to increase the power of one motor, and decrease the power of the opposite motor. This will produce a horizontal force, that moves the quadcopter in the opposite direction of the motor with increased power.

1.2.2 Choosing of components

One of the first problems when choosing components for a quadcopter, was the size of the quadcopter. The size should be large enough to support the weight of the microcontroller, the actuators and the sensors; and it should be affordable, as we will not focus on the quadcopter, but on the code to make it stabilize. Another thing about the size is that the further away the motors are placed from each other, the more stability. On the quadcopter we have different components, the needed components to achieve stabilization are the following; a frame, four motors, four ESCs (Electronic Speed Control), a battery, sensors to read the movement of the quadcopter, and a computing device to run the software, to control the ESCs.

The computing device and the sensors.

Arduino: The computing device of the quadcopter is the core of the system; all commands and controls that are initialized originate or have to be accepted by the computing device. Such a computing device could be an Arduino board.

Arduino is a software and hardware company which manufactures and produces open-source programmable microcontroller boards and an associated IDE, which uses a language

Table 1.1: Specifications for Arduino MEGA

Microcontroller	ATmega1280
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	128 KB of which 4 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

based on C, used to write and upload computer code to the microcontroller boards. Arduino's wish was to make a microcontroller board that was easy-to-use for both advanced and amateur computer programmers, with the intent that it would be widely used by students[10, 11]. Arduino has since 2005 created a lot of different series and models of microcontroller boards, which they call Arduino boards[12]. Arduino also uses some of its own terminology, such as a program running on Arduino board is not called a program but a sketch. The Arduino board that will be used and focused on in this project is the Arduino MEGA.

Arduino boards have the ability to either register inputs from the physical world by e.g. capturing light or tracking movement by utilizing sensors or inputs from the digital world as e.g. retrieving messages from a website, and then turn it into output which could be anything from starting a motor to sending an email[10]. In this task, the Arduino MEGA will be used to stabilize the quadcopter.

Arduino MEGA

The most significant part of the Arduino MEGA is its microcontroller ATmega1280. In a microcontroller board like the Arduino MEGA, the microcontroller is the part everything is based on and can be said as the core of the microcontroller board. The Arduino MEGA has 54 digital input/output pins, 16 analog inputs, 4 hardware serial ports, a 16 MHz crystal oscillator, 8 KB RAM, 128 KB flash memory, a USB connection, a power jack, an In-Circuit Serial Programming (ICSP) header, and a reset button. There is a list of the specifications for the Arduino MEGA in table 1.1.[13]

The frame

To select a frame we looked on frames that could support the size of an Arduino. We found multiple frames that could do this, though not any that were specifically made to support an Arduino, within a reasonable price range. So we found frames that could be modified so that an Arduino could fit. The two most appropriate frames were a medium 25 cm diagonal frame and a 45 cm diagonal frame. We chose the bigger frame over the smaller frame because of a few reasons; more modification possibilities, seemed more robust, and more stability based on size.

The motors, the battery, and the ESCs

Motors can use different kinds of propellant, but making a quadcopter with liquid fuel, controlling the motors as well as getting a lightweight motor that can lift itself, including extra weight, will be hard and expensive. The best choice of propellant will be electricity, because it is controllable, relatively cheap and there is a big market for it. The choice is then whether the motor should be a brushed AC motor, or a brushless DC motor, The speed of an AC motor is mechanically defined by the number of coils, the friction of the work, the voltage given and the current available. It is possible to manipulate the speed of a brushed AC motor with voltage, but by using a brushed AC motor there is no guarantee on the Rounds Per Minute(RPM) achieved at different voltage levels. The RPM will still depend on the friction, how worn out the motor is, the load on the motor, and the differences between the motors, thus making the AC motors hard to control.

A brushless motor's RPM is defined by an ESC, the ESC can also give feedback on how big of a load is on the motor. Furthermore the brushless motor is weighing less per torque, consuming less power per torque, in need of less maintenance, and has lower friction in the motor, because the AC is dragging the brush against the moving part. The brushless DC motor might be more expensive and require an ESC, though when making objects fly the brushless DC motor is definitely the best choice, for better control, less weight and less power consumption.[14, 15]

Most brushless DC motors, used with a remote control (RC) quadcopter, can use 2-3 cell batteries. The battery needs to have enough battery time to land and stabilize, and the flight time on a battery does not need to be high. Considering that the battery could be reused later, we chose a battery with 3 cells and 5500mAh, which is frequently used by RC quadcopters. The battery is made for RC hobby vehicles. The DC motors used for quadcopters are almost always brushless, because of its properties and advantages over brushed motors. There is no need for an open casing, because the windings are placed in the outer casing, which means it can be cooled by conduction.[14, 15]

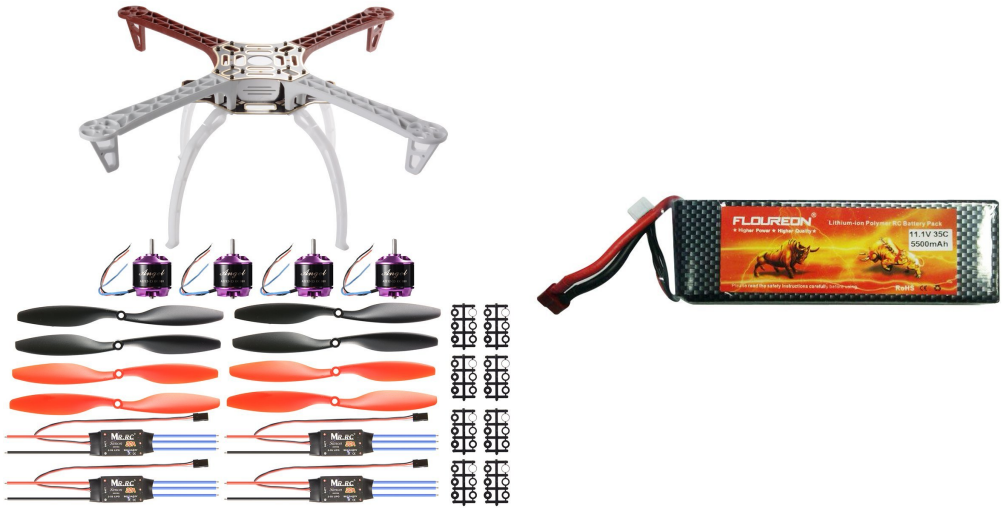
To choose a motor we would need to know an estimated weight of the whole system. We already know the weight of the components we have chosen; the frame weighs 326 g, the battery weighs 458 g and the Arduino MEGA weighs 37 g. This adds up to 821 g, then we need to add the motors', the ESCs', and the sensors' weight to that. When choosing motors we looked for brushless DC motors that had enough force to at least make 3 kg of thrust combined. We found a package with the frame, the motors, the ESCs and 10" $\times$ 4.5" propellers. The kit had combined thrust at about 4 kg at 16A with a 3 cell battery and 10" $\times$ 4.5" propellers, so that was a sufficient and cheap option. The ESCs can take up to 30A, it is advised to use ESCs with capabilities higher than that of the motors' requirements. ESCs can melt down if used at max current for too long. The ESCs within the kit comes with SimonK firmware, which is firmware made for multirotors, like a quadcopter. One of the pros with this is being able to maneuvers faster than most other firmwares, because of SimonK's faster speed adjustment time. With those motors and ESCs the final weight went up; motors weighs $60g \times 4$ and ESCs weighs $37g \times 4$.

All in all the quadcopter would weigh about 1,200 g including sensors and mounting for

the Arduino and sensors. With this setup the motors should be able to take off and stabilize in mid air, and for future use also be able to make maneuvers or ascend.

1.2.3 Building the quadcopter

This section will cover how the quadcopter used, in the project, was assembled for task one. As seen on figure 1.3a, the assembly includes a quadcopter frame, landing gear, 4 motors, 4 electronic speed control units for the motors, and 8 propellers for the motors. The kit also includes a 5500mAH Lipo battery, as seen on figure 1.3b. There are two different types of propellers; one type for clockwise rotation, and the other type for counter-clockwise rotation, which generates lift.



(a) All the parts included in the quadcopter assembly.

(b) The battery used with the quadcopter assembly.

Figure 1.3: quadcopter parts

The quadcopter frame includes 4 mounting points for the motors, and each motor includes different parts, as seen on figure 1.4. These parts include mounting screws, a mounting platform, propeller mount for the motor shaft, as well as shrink tubes and wire housing units for the motor and ESC wires.

Figure 1.4: All the parts included with the motors.

When assembling the quadcopter, each motor has to be mounted on each of the 4 mounting platforms, as seen in the corners of the quadcopter frame displayed on figure 1.5a, with one of the mounting platforms on the quadcopter frame being displayed in the lower left part of the picture set. For this project, it was decided to mount the motors directly onto the frame, instead of using the mounting platforms that were supplied with the motors, as this was deemed sufficient.

(a) Mountpoints on the quadcopter frame. (b) Sideview of the quadcopter frame.

Figure 1.5: quadcopter frame

Between the upper and lower metal parts of the quadcopter frame, at the quadcopter frame's center, the Arduino unit controlling the quadcopter itself is mounted. On the upper metal part, the battery displayed on figure 1.3b is mounted. A better angle can be seen on figure 1.5b, where the red area designates the battery mounting position, and the blue area designates the Arduino mounting position.

To control the speeds of the motors, the ESCs have to be connected to the Arduino unit, and the motors. For optimal efficiency, these ESCs were mounted below the arms between the center of the quadcopter frame, and the motor mounting points on the frame, as seen on figure 1.6 by the spots marked with the green color.

Figure 1.6: ESC mounting topview of the quadcopter frame.

After mounting everything, the last thing that had to be set up, was power distribution to each of the ESCs and their individual motor. It was discovered that the lower metal part of the quadcopter frame had mounting positions for 5 different power connections. Each mount had a positive (+) and a negative (-) indicator, and these mounts completed a circuit between the battery and the 4 ESCs. This setup is displayed on figure 1.7. The red color on the picture designates the positive (+) mount, and the blue color designates the negative (-) mount for the power connections. The green box contains the power connection for the battery.

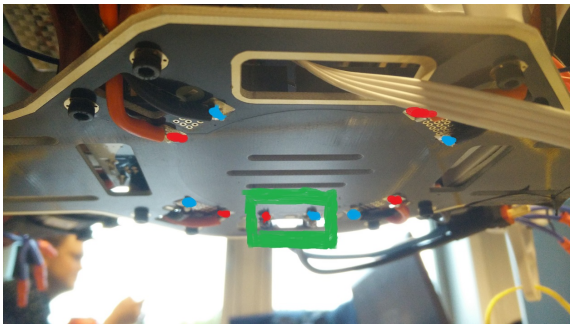


Figure 1.7: View of the power connections below the quadcopter frame.

The assembly of the quadcopter, for task one, can be seen on figure 1.8.

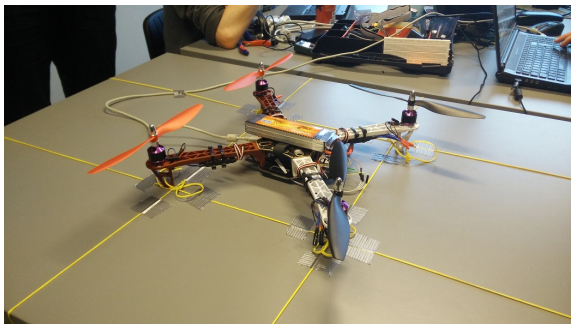


Figure 1.8: View of the assembly of the quadcopter for task one.

The completed assembly is 49 cm in diameter from the edge of the white to the red motor mounting positions on the quadcopter frame. Each propeller has a diameter of 25.5 cm, and after accounting for the motor mounting position, the wingspan of the propellers increase the distance of the assembly to roughly 73 cm in diameter.

The weight of the individual parts and a combined weight:

- **Battery:** 458g
- **Quadcopter frame:** 326g
- **Arduino MEGA:** 37g
- **Motor:** 60g each (240g total)
- **ESC:** 37g each (148g total)
- **Landing gear:** 78g
- **TOTAL:** 1131g without landing gear, and 1209g with landing gear

Analysis and tests 2

In this chapter the sensors and actuators will be described and analysed. First a description of the sensors, followed by an analysis of the actuators. Then the different methods for sensor fusion will be analysed and tested, then a description of how the PID is setup, and finally a description and test of the flight controller.

2.1 IMU

This task requires the addition of the Inertial Measurement Unit (IMU). The IMU includes an accelerometer, a gyroscope and a magnetometer. The IMU is used to measure and report a craft's, which in this case is the quadcopter, acceleration and orientation, using a combination of the three specified sensors.

The IMU chosen for this task, is the *GY-85* IMU for Arduino, which uses an ADXL345 accelerometer to measure proper acceleration, an ITG3200 gyroscope to measure or maintain orientation based on the principles of angular momentum, and a HMC5883L magnetometer to measure the magnetization of a magnetic material, or to measure the strength and the direction of the magnetic field at a point in space. All information gathered is from a GitHub implementation of the IMU, which includes datasheets for all three sensors.[16]

The IMU uses horizontal X-axis and Y-axis, and a vertical Z-axis to measure rotation in space relative to gravity. A technical description of each sensor can be found in appendix A.1.

2.2 Actuators

The only actuators on the quadcopter are the brushless DC motors where the propellers are mounted. A brushless DC motor can be difficult to control with a normal computing device, if the device should do other tasks at the same time, therefore an Electronic Speed Controller (ESC) is used to control the motors. This section will be about describing the motors, how they operate, what they are capable of and if they can do the task of lifting the quadcopter, to the extent that it can take off and stabilize in mid air.

2.2.1 Operating the motors

To operate the motor an ESC is used. The ESC we are using comes with pre-installed SimonK firmware, and it has five inputs and three outputs. All three outputs should be linked to the motor, and will control the coil circuits in the DC motor. By activating the coil circuits at the right time, the motor will spin, by applying power on each coil circuit consecutively and afterwards the same order but this time the power must be negative. The ESC will control this for us, so we only have to focus on the inputs. Two of the five inputs are power for the motors, the three others are for controlling the ESC: one of them are +5V and one is ground, those are used to power the device that controls the speed of the motor. The last input is used to give the ESC speed commands, this is the input we use to control the motor. Note that there is always a max current and voltage limitation on the ESC, in this case it is 30A and 2-3 cell battery (7.4V or 11.1V), and a maximum of 10 seconds of runtime on max current.

To operate the ESC we use a Servo library from Arduino. By giving an integer from 0-180, the servo library will convert it and send it to the ESC. The integer 0 will give full speed one direction, 90 will stop the motors, and 180 will give full speed the opposite direction. The ESCs can be calibrated by signalling highest speed (int = 0 or int = 180) to the ESC, when connecting the battery to the ESC, this will trigger a sound code from the ESC, after the sound code, the signal turn the speed to lowest speed (int = 0), and a new sound code will signal calibration successful. After a calibration to the Servo library, the motors are ready to be tested, and the calibration data will remain in the ESC, also after the battery is disconnected from the ESC.

While testing on the motors we change the speed gradually, never start on speed, and never jump more then 1/90 part speed as it can be hard on motors and the ESC itself.

A description of the configuration of the tests done on the motors can be found in appendix A.3.

After testing, we discovered that our ESCs were not able to change direction when using the integer ranges 0-90 and 90-180 as described. In order to do this, we instead had to rearrange two of the wire connections, which manually reversed the spin direction of the motors.

2.2.2 Results

For the first parts of the results we will look on the motors individually, to see if the motors are consistent. The results of all the motors can be found in appendix A.4. Motor number 2, 2.1, had the least consistency, but from 30% throttle and up the biggest inconsistency was at 3.3%. All the different lifts are very close to each other, see figure 2.1, so a little difference in the lifting power at same throttle should be expected with the motor sets. The lifts at 20% had a high difference, motor 2 even had 8.33% difference, where the highest lift was 39 g and lowest was 36 g, one good reason for this could be the digital scale, as the scale is only precise to whole digits.

We know that the individual motor's throttle gives the same lifting power with under 5%

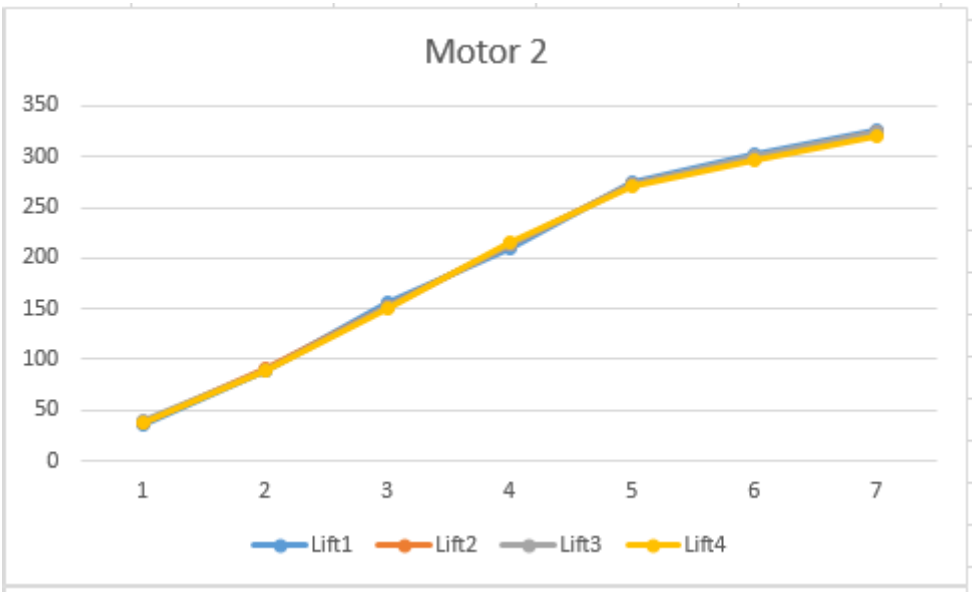


Figure 2.1: Results from motor 2, which had the least consistency

Motor 2	20%	30%	40%	50%	60%	70%	80%
Lift 1	36g	89g	156g	210g	274g	303g	326g
Lift 2	38g	91g	153g	213g	273g	299g	325g
Lift 3	39g	89g	153g	214g	272g	300g	324g
Lift 4	38g	89g	151g	215g	270g	296g	320g
Average	37,75g	89,5g	153,25g	213g	272,25g	299,5g	314,5g
Highest/lowest diff.	8.3%	2.2%	3.3%	2.4%	1.5%	2.4%	1.9%

Table 2.1: Result of motor 2

Total	20%	30%	40%	50%	60%	70%	80%
Average	38,75g	84,5g	148,13g	215,06g	265,88g	303,25g	332g
Min	36g	80g	135g	201g	238g	282g	314g
Max	42g	91g	157g	230g	283g	322g	344g
Max diff.	6g	11g	22g	29g	45g	40g	30g
Max diff.	16,7%	13,8%	16,3%	14,4%	18,9%	14,2%	9,5%

Table 2.2: Total result

deviation. But if we compare the motors to each other, the results are a bit different, as seen in table 2.2 and in graph 2.2. The results shows that there is a big spike at 60% throttle, where the motor's differs with 18.9%. This difference is very big, and we should expect when using the motors, that they will not deliver same lifting power, when at same throttle. The thrust should not be trusted across the different arms on the quadcopter, the throttle could be set by a PID, that uses the output of the IMU as input to change the throttle. With a PID the difference in the motors should not matter.

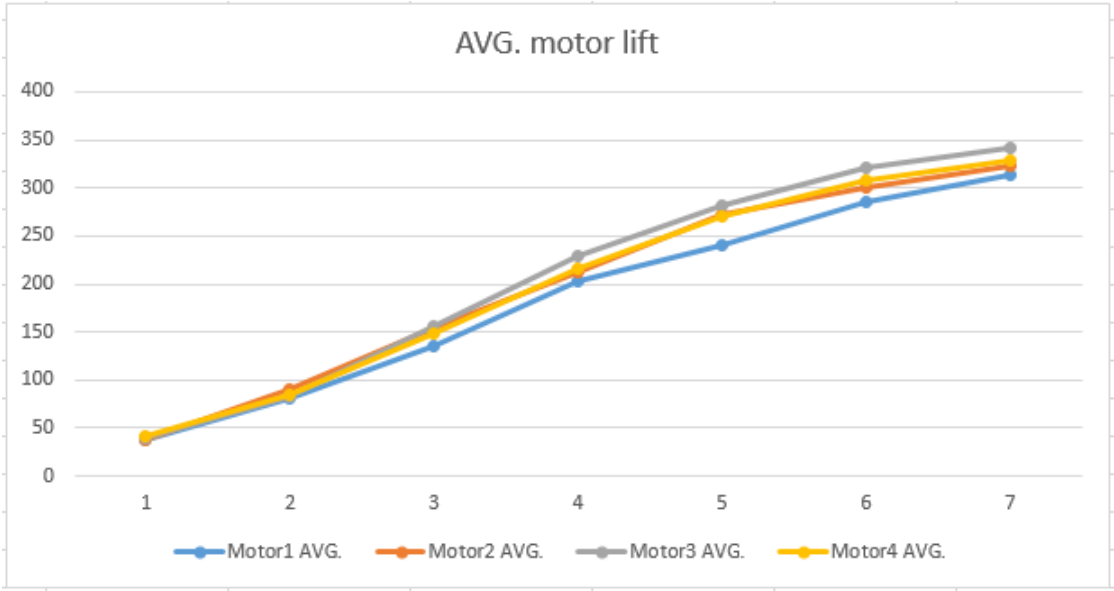


Figure 2.2: The four motor sets average lifts.

2.3 Sensor fusion

In the next section, section 2.4, we will describe the test of the sensor fusion. To be able to understand why we are performing the test, it is first necessary to describe what sensor fusion is, and the three different kinds of sensor fusion we will be using.

2.3.1 Reason and purpose

We need to know how the quadcopter's orientation compared to the surrounding world, and the IMU gives us two outputs that output data based on this orientation: one from the gyroscope and one from the accelerometer. We cannot just use the gyroscope, even though it handles shaking better than the accelerometer, because the gyroscope is not affected by the world, and therefore the gyroscope starts to drift. Drifting is when noise accumulates over time, and the gyroscope alone does not know that it has drifted. The accelerometer cannot be used alone either, because as seen on figure 2.3, it is affected by all accelerations, not only gravity. But when the IMU is still, the accelerometer is able to read the gravity vector, and therefore know the quadcopter's rotational relation to the world. To combine gyroscope and accelerometer is called sensor fusion, and it is done by using the accelerometer to calibrate the gyroscope readings as often as possible.

2.3.2 Complementary filter

The simple way to make a sensor fusion, is to use a complementary filter, it is implemented with a mathematical formula "$Angle = A \times (Angle + (Gyro \times Dt)) + (1 - A) \times Accel$", where $Angle$ is the quadcopter's current rotation in degrees, A is a constant from 0-1, $Gyro$ is the data from the gyroscope in degrees per second, Dt is the time since last sample in seconds, $Accel$ is the current rotational readings from the accelerometer in degrees. The gyroscope is returning the angle movement, this value times Dt is the number of degrees the IMU has rotated since last sample. To do this we add the last angle of the quadcopter to find

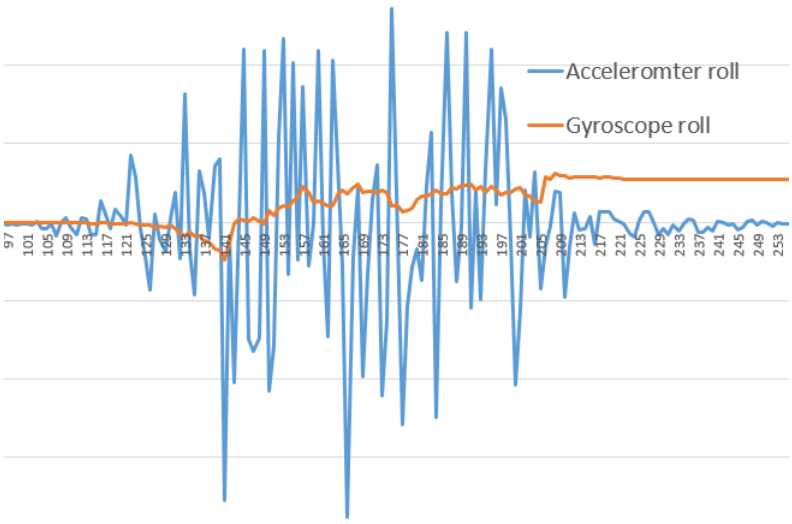


Figure 2.3: An example of the large fluctuations the accelerometer has

the current angle. Now the equation states that we take a percentage of this value, that represent the gyroscopes measured angle, and add it to a percentage of the accelerometers measured angle. This way we combine gyroscope and accelerometer, where we can choose how many percentages of the gyroscope and accelerometer we want. To choose the right A we would test different values, since the gyroscope is good at measuring angles but drift over time, we want a high percentage of the gyroscope's measured angle. This can also be seen as a high pass filter on the gyroscope and a low pass filter on the accelerometer because we let the long term accelerometer measurements stay by always adding a little of it to the equation. It also holds the short term changes from the gyroscope by holding on to a lot of the changes, but always using the accelerometers smaller percentage to calibrate a little for every sample taken.[17]

2.3.3 Kalman filter

The Kalman filter is a filter commonly used to guide, navigate and control vehicles, particularly aircraft, and it is also often used in robotic motion planning and control. It is a recursive algorithm, that can be implemented in real time, using only previous readings to filter out noise. It measures on inputs, and estimates a weight for each input, where a high weight means that the filter is certain that it is not noise. This weighing is dynamic and the filter will take it into account when calculating the predicted fusion. The filter is a more complicated version of the Complementary filter, where the previous values are taken into consideration when calculating the current value. Kristian Lauszus, TKJ Electronics, made an Arduino library specifically for gyroscope and accelerometer sensor fusion. This version of the Kalman filter will be the one used in our work with the quadcopter. [18, 19]

2.3.4 Our algorithm

The idea is to use the same equation as the complementary filter, and then tweak it with the knowledge we have about the quadcopter. We can dynamically measure how much

the quadcopter is accelerating at all times, by using the accelerometer. Since it is also known that the gravity vector has the length of exactly 1, if the vector is measured in gravitational pull, equal to roughly $9,82\ m/s^2$. So we know when the quadcopter is not accelerating, and this we can use to make the complementary filter better. To do this we add a new factor, which is called trust, and it is a value equal to A in the complementary filter. The trust indicates how much we trust the gyroscope at a given time. The trust is dynamically updated by calculating the current acceleration of the IMU, and if this acceleration is higher than or lower than 1, the trust is lowered accordingly.

A function to calculate trust is used, and needs to be tested with different values. This function is "$1 - (B \times (x-1)^2 + C)$" where B and C are constants, and x is the current accelerometer movement of the quadcopter. By changing B it will affect how much the quadcopter needs to be accelerating before the trust is lowered. By changing C the effect of the trust is changed, if C is 0.10, and the quadcopter is still, it would be the same as a complementary filter with $A = 90$. If the quadcopter is moving this algorithm might only use the gyroscope data, and leave out all the accelerometer, until the quadcopter stops accelerating, this is controlled by the constant B. An example is when B is set to 3.6, the algorithm will use the accelerometer data until the current acceleration vector is within the boundaries of 0.5 to 1.5, given that C is 0.10. This way we limit the use of the accelerometer in moments of known noise, and depend more on the gyroscope at those times. As with the complementary filter we will test on different constant values, and we still know that the accelerometer is going to have a lower affect on the output of the algorithm than the gyroscope at all times, because we still need the low-pass filter on the accelerometer and the high-pass filter on the gyroscope. Some of the issues with this algorithm is that if the quadcopter is always shaking and the B constant is too high, the shakes might be enough to get the trust to 1, thus preventing the accelerometer from calibrating the gyroscope. The opposite can happen if C and B are too high, the accelerometer would take over, and a lot of noise might slip through. That is the reasoning behind conducting tests with different constants, to see which constants would be best to use.

2.4 Sensor fusion test

In this section, the test of the sensor fusion and the IMU will be described. First the purpose of the test, including the reason we have performed it. Then the setup and execution of the test, and finally a conclusion based on the results of the test. It is important to notice that to understand the section completely, it is required to know about the sensor fusion described in section 2.3.

2.4.1 Purpose

The purpose of the test, is to test which algorithm will be best for sensor fusion. The optimal algorithm should cancel out noise, but not actual tilt. This is essential to keep the quadcopter level in the air.

The quadcopter should be able to fly more or less still in the air. To do that, we must know its angle compared to the world, so that it can adjust the thrust, to keep it leveled.

We must therefore test several parts of the IMU, including how fast it can update and respond, how accurate it is, and how sensitive it is to vibrations and other inaccuracies. In this first test we will focus on finding the best sensor fusion algorithm.

To get the angle of the quadcopter compared to the world, we must use input from two different parts of the IMU: the gyroscope and the accelerometer. Figure 2.3 shows an example of the output from the accelerometer and the gyroscope, the IMU is only moved in XY axis and have no rotations, therefore the desired output of this graph would be stable close to zero. It is clear that the accelerometer has large fluctuations, and it is therefore important to sort out the noise from the accelerometer. The two inputs from the gyroscope and the accelerometer must then be calculated together as one variable, to give a qualified guess at what the actual roll and pitch values are, this is called sensor fusion. There are many ways of implementing sensor fusion, and we have developed our own algorithm, which is a tweaked version of the complementary filter, and compared it to existing algorithms, see section 2.3. The purpose of the test, is to find the best algorithm for calculating the pitch and roll, by constraining the statistical noise and other inaccuracies from vibrations.

2.4.2 Setup and execution

The setup of this test was to have the quadcopter with the IMU on a plane surface, connected to the Arduino, ESCs and motors as seen on figure 2.4. We did three kinds of testing, one where the motors were not turned on, where we manually moved the quadcopter around on the plane surface, which should not give any pitch or roll. Another where we took the quadcopter in hand and moved it around, both pitch and roll, and one where the quadcopter was tied to a table, constraining it fly at most 15 centimeters above the surface, and then turned on to let it fly. The purpose of the test with no rotation, was to inflict as much noise on the accelerometer as possible, and test the algorithms on how well they handle movement in general, here we would like our algorithms to stay as close to 0 as possible. We collected data for about 30 seconds, before turning it off, and then put the results in a graph, to visualize them. We got 54 different outputs from each test, 4 of them were the raw angles from the gyroscope and accelerometer, both pitch and roll, and the other 50 were the algorithms. We used 25 different algorithms, with both pitch and roll. 4 different complementary filters, 1 Kalman filter and 20 different versions of our own algorithm. Because of the possibility of inconsistent movement, we used all the algorithms at each test, and only compared them within the same test.

2.4.3 Results

In this subsection we will conclude on which algorithm to use. It is important to note that the choice is not built only on the results shown in this subsection, but on several tests.

Due to limitations on the amount of data which can be communicated over a serial port at one time, we will only look at one value per 3 readings. This should be efficient enough though, and not result in any complications.

We have tested the 25 following algorithms:

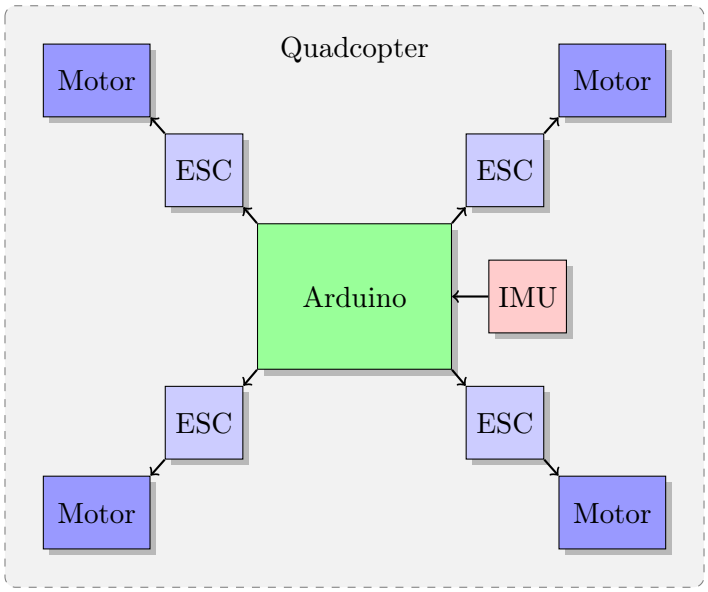


Figure 2.4: Component diagram over the Arduino, IMU, ESC and a Motor.

- Kalman filter
- Complementary filter, with 4 different A values: 0.995, 0.980, 0.950 and 0.900
- Our algorithm, with 5 different B values: 50, 100, 500, 1, 5000 and 4 different C values: 10, 5, 2 and 0.5, with a total of 20 different versions of our own algorithm.

The main goal of this test is to find the best algorithm for our sensor fusion, so to find the best, we started comparing the different algorithms, and continuously narrowing it down to the best. As seen on figure 2.5 where we have compared the 4 different C values with the same B value = 5000, all the algorithms are too dependant on the gyroscope, and are therefore not good to use. These will not be considered as usable algorithms for this project.

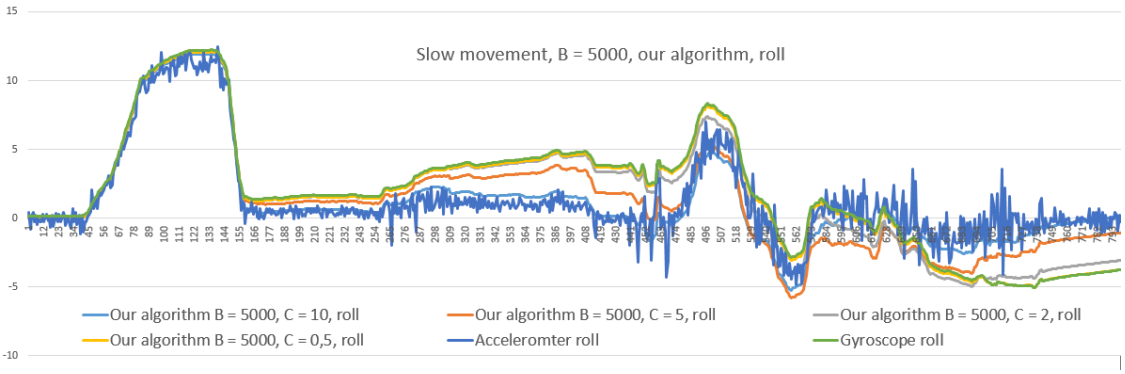


Figure 2.5: A graph showing the four different C values with the B value set to 5000, compared to the accelerometer and the gyroscope

Continuing like this, we narrowed it down to 4 algorithms, which all seemed to perform well:

- Kalman filter
- Complementary filter with $A = 0.98$
- Our algorithm, $B = 1$ and $C = 5$
- Our algorithm, $B = 1$ and $C = 2$

These last four algorithms were then compared to each other. As seen on figure 2.6 they all seem to perform fairly similar, with no clear favorite.

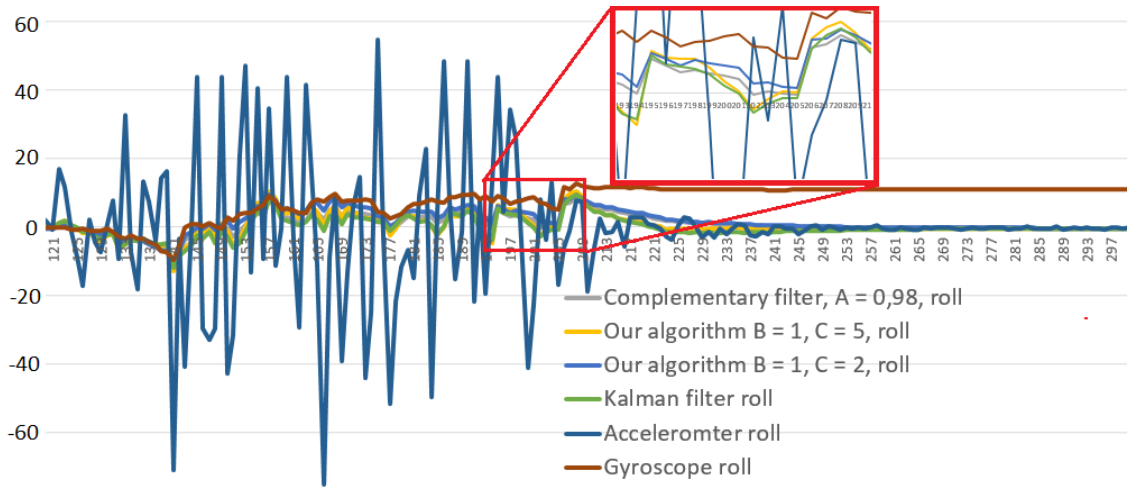


Figure 2.6: A comparison of the four best algorithms

Conclusion

The conclusion on this test is that our own algorithm with $B = 1$ and C either set to 2 or 5, the complementary filter and the Kalman filter all compare fairly similar. We have chosen to use the Kalman filter in this project, due to the fact that there is existing information stating that the Kalman filter performs best, and that it is already used in many real life vehicles.[19]

Even though the filters perform well, and are good at constraining the statistical noise and other inaccuracies from vibrations, the results are still not useful. In the next section we will describe how we reduced the large fluctuations even more.

2.5 Reduction of the fluctuations

In this section we will describe how we have limited the large fluctuations from the IMU. As seen on figure 2.6 even with the Kalman filter the fluctuations are still around $\pm$ 10 degrees which is way too much. The results are therefore still not usable for making the quadcopter fly.

To reduce the fluctuations from the accelerometer, we had to try several things.

2.5.1 Considerations

We considered different causes of the large fluctuations. It could be vibrations from the quadcopter, that made the accelerometer max out its readings. This could be solved simply by isolating the IMU with some shock- or vibration absorbent material, or by limiting how much the quadcopter vibrates. If it was not due to vibrations, it could be electronic noise from the battery, motors and Arduino that interfered with the readings. This could be solved by moving the IMU away from all the electronic devices, by placing it on the end of a stick, or at the end of a leg on the quadcopter. Lastly it could be that the IMU simply was too poor quality to make reliable readings. It was with these considerations in mind we performed the tests that should help. The first goal was to find out what caused the fluctuations.

2.5.2 Procedure for finding the cause

First we had to determine what the cause of the large fluctuations was. Vibrations, electronic noise, poor hardware quality or something else. To begin with we placed the IMU away from all the electronic devices, e.g. the battery, the motors and the Arduino. This was done by attaching a 30 cm aluminium stick to the quadcopter, and placing the IMU on the end, and using aluminium foil around the electronic devices as seen on the left side of figure 2.7. This solution didn't help much and therefore was not very useful. Secondly we tried attaching a cardboard box, and on the side of that the aluminium stick, see the right side of figure 2.7, this helped even more, giving fluctuations as small as $\pm$ 3 degrees, as seen on figure 2.8.

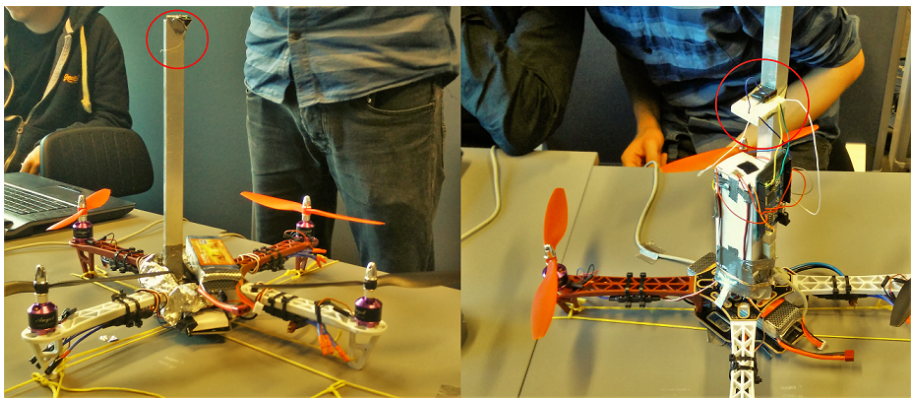


Figure 2.7: Moving the IMU away from the electronic noise

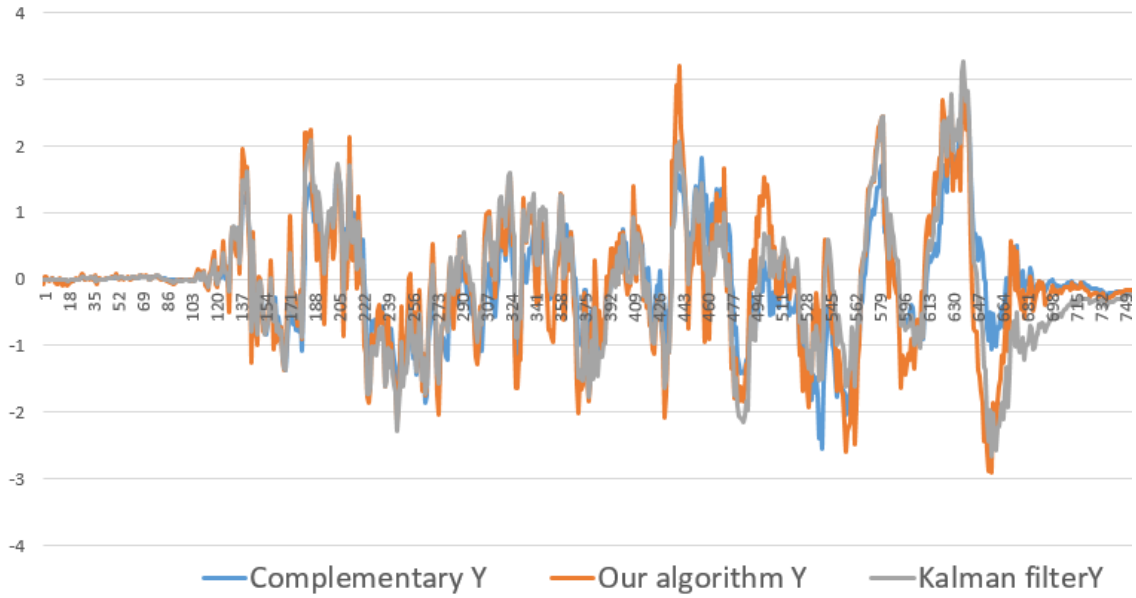


Figure 2.8: The output from the filters, with the IMU placed away from electronic noise

Placing the IMU as seen on the pictures 2.7, gave significantly better results, and made us believe that the large fluctuations was mainly caused by electronic noise. However the setup we made with the the cardboard box and the aluminium stick, was not sustainable, and we had to come up with something more permanent. We therefore tried mounting the aluminium stick horizontal to the quadcopter, making it stick out one of the sides. But this gave bad results again. After several attempts of placing the IMU different places on the quadcopter, we tried something new: placing the IMU close to all the electronic devices but instead holding the IMU in the hand of a group member. This showed good results with a minimum of fluctuations, and also convincing us that the fluctuations actually had nothing to do with the electronic noise, but that they were instead caused by vibrations, and the setup seen on picture 2.7 was working, not because of the placement away from the battery, but because it absorbed some of the vibrations. We then tried several things to isolate the IMU from the vibrations: mounting it on earplugs, tape, cardboard, sponges and so on. Nothing gave consistent usable results.

After finding the cause of the fluctuations, we tried to find a durable solution.

2.5.3 Reducing the vibrations

When working to reduce the amount of vibrations in the IMU, there are two approaches:

- Limit how much the quadcopter vibrates
- Isolating the IMU with shock- and vibration absorbing material

The first thing, limiting the vibrations from the quadcopter, is only part of the solution, since the quadcopter will always vibrate a certain amount, when the motors are running and the propellers are spinning. There is a fix which can limit the vibrations considerably;

balancing the propellers. If the propellers are not balanced, it may result in unnecessary vibrations, which can be removed by balancing the propellers. This is done by using a specific tool, and making sure both sides of the propellers are horizontal, thus making sure it is completely balanced, as seen on figure 2.9. If a propeller is not balanced it can be adjusted by adding a small piece of tape on the lighter side.

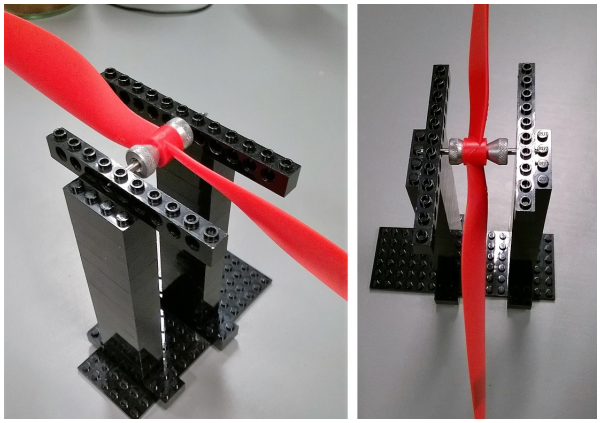


Figure 2.9: How to balance the propellers

The second part is to isolate the IMU from the vibrations by placing it in or on some material or device that can absorb some of the vibrations. To do this, we constructed a special device. The device is seen on figure 2.10. It consists of: two pieces of plywood, with four pieces of steel wire in each corner working like as a shock dampener. Underneath it has some pieces of firm foam with cotton wool between, and on the top the IMU is placed on top of a piece of led to give some weight, since the IMU does not weigh much itself. The cord from the IMU could also be a source of leading the vibrations to the IMU, so it is also placed between some pieces of firm foam and cotton wool, which is then attached to the bottom of the top piece of plywood.

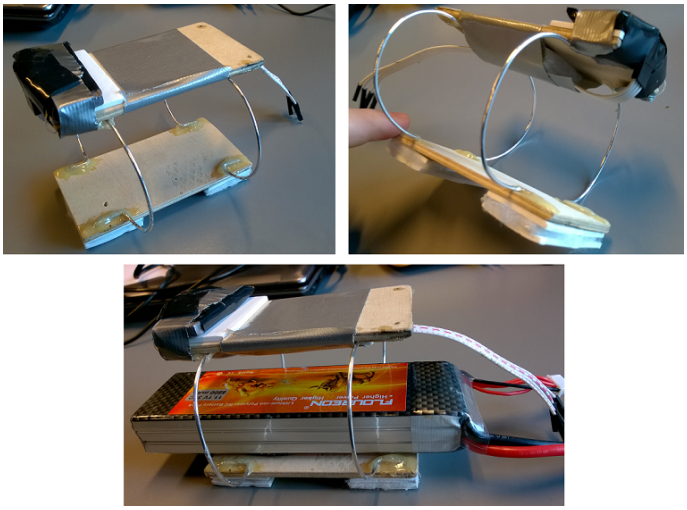


Figure 2.10: Our homemade vibration-absorption device

2.5.4 Results

After attaching our vibration-absorption device, and balancing the propellers, we made a new flight test, with the quadcopter tied to the table, only letting it fly 15 cm above the table. Figure 2.11 shows the results. The red circles mark the take-off and landing parts, which still show moderate spikes. The take-off has some spikes, because the motors do not start completely simultaneously, resulting in a few fluctuations. The green square marks the flying part, which shows good results, with fluctuations around ±1 degree. The fluctuations are reduced significantly, and considered to be at an acceptable level.

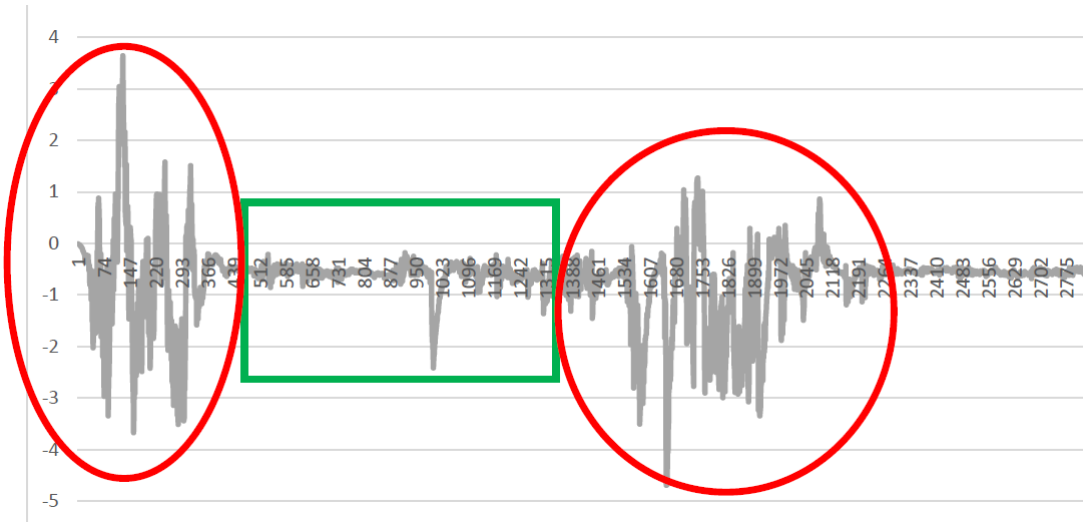


Figure 2.11: The result from the Kalman filter, after our modifications for reducing the vibrations. Green part is actual flight, and red part is take-off and landing.

2.6 PID Controller

A proportional-integral-derivative controller (PID controller) is a control algorithm which is used to help a system reach its desired status, and in this project it helps control the quadcopter to maintain its desired angle relative to the direction of the force of gravity. The desired status of a system is called the system's setpoint. The PID controller uses the system's setpoint and the current status of the system to continuously calculate the difference between the two which is said to be the error, the goal of this is to minimize the error so the current status is close or equal to the setpoint. The algorithm of the PID can be represented as following:

$$\mathrm{u}(t) = K_p e(t) + K_i \int_0^t e(\tau)\,d\tau + K_d e'(t)$$

$\mathrm{u}(t)$ represent the PID controller output, t is time, τ is the variable of integration, e is the error, and the K_p, K_i and K_d variables represent the three gain constants of the three control terms of the PID controller, the proportional term, the integral term and the derivative term. These three control terms are what define the algorithm and together should be able help the system achieve its setpoint at all times.

Proportional: Looking at the first part of the right hand side of the algorithm we see the proportional term $K_p e(t)$. K_p is known as the proportional gain constant and $e(t)$ is the size of error at a given time, and together they produce an output proportional to the error. The goal for the proportional term is to find a proportional gain constant that is not so high that it makes the system unstable, nor too low causing it to have no effect on the system. The proportional gain should be at a constant where the system obtains steady oscillations.

Integral: The second part of the algorithm, $K_i \int_0^t e(\tau)\,d\tau$, is the integral term, where K_i is the integral gain constant and $\int_0^t e(\tau)\,d\tau$ is the sum of the error from the initial time 0 to time t. The integral term uses previous errors and the present error to figure out overall how negative or how positive the total error is. This is to find accumulated error if the system has a tendency to gather overall more negative errors than positive or vice versa. Since the setpoint is when the error is zero the sum of all errors should also be as close to zero as possible, this is where the integral term is useful since its job is to make sure that there is an even amount of positive and negative errors at all times.

Derivative term: The last part of the algorithm, $K_d e'(t)$ is the derivative term, where K_d is the derivative gain constant and $e'(t)$ is the slope of the error at time t. The derivative term uses the slope of the error to predict the course of the system. The PID controller uses this prediction to eliminate or minimize any future errors. This term is the least used of the three terms, it is only useful for systems which cannot function if not remaining close to completely stable.[20, 21]

Not all systems that implement the PID controller require the use of all three terms, but when applying it to a quadcopter, all three terms should be in use. The purpose of the PID controller in a quadcopter is to achieve stabilization when in flight. It does this by minimizing the errors in the angles of both the roll and pitch individually so that the angles of both the roll and pitch are equal to each their setpoint. At still flight, the setpoints are the angles when the plane of the quadcopter is perpendicular to the direction of the force of gravity.

2.6.1 Implementation of the PID controller

Arduino PID Library is an Arduino Library created by Brett Beauregard that implements a PID controller designed specifically for Arduino boards. The library includes twelve functions where only a few of them are required to be implemented for the controller to work[22]. The following example code shows how some of these functions are implemented.

```
1 //PID values
2 double kp=4, ki=0.03, kd=23;
3
4 //PID setup
5 PID pidPitchStab(&kalAngleY, &pidPitch, &wantedPitch, kp, ki, kd, DIRECT);
6 PID pidRollStab(&kalAngleX, &pidRoll, &wantedRoll, kp, ki, kd, DIRECT);
```

Example code 2.1: Code from PIDtest

Example code 2.1 shows both the declaration and initialization of the proportional, integral, derivative gain constants and the declaration of two PID controllers, one for the pitch, `pidPitchStab`, and the other for the roll, `pidRollStab`. When declaring a PID controller the function requires that its seven parameters are passed arguments of the corresponding datatypes, the first parameter is the input we want to control, in this code it is the Kalman values for pitch and roll, the second is the output value from the controller, the third is the setpoint, the desired angles of pitch and roll, the fourth is the proportional gain constant, the fifth is the integral gain constant, the sixth is the derivative gain constant and the seventh is direction of the output depending on the error, the seventh parameter can only be given the arguments `DIRECT` or `REVERSED`, in this case, it is `DIRECT`.

```
1  void setup() {
2    Serial.begin(115200); // Settings for port communication
3    Wire.begin();         // Enables IMU communication
4    //...
5
6    ...//
7    kalmanX.setAngle(0);    // Starts Kalman
8    kalmanY.setAngle(0);    // Starts Kalman
9    CalibrateKalman(100);   // Finds offset
10   kalOffsetX = kalAngleX; // Saves offset.
11   kalOffsetY = kalAngleY; // Saves offset.
12
13   pidPitchStab.SetMode(AUTOMATIC);          // Marks PID as active
14   pidRollStab.SetMode(AUTOMATIC);           // Marks PID as active
15
16   pidPitchStab.SetOutputLimits(-100, 100);  // Sets max and min output for PID
17   pidRollStab.SetOutputLimits(-100, 100);   // Sets max and min output for PID
18
19   pidPitchStab.SetSampleTime(8);            // Settings for how often PID is
         computed
20   pidRollStab.SetSampleTime(8);             // Settings for how often PID is
         computed
21 }
```

Example code 2.2: Part of the `setup` function from PIDtest.ino

Example code 2.2 is part of the `setup` function from the PIDtest Arduino file. In this code we see three different functions from the PID controller library, `SetMode`, `SetOutputLimits` and `SetSampleTime`. `SetMode` is used to activate or deactivate the PID controller. If `SetMode` is passed the argument `AUTOMATIC` then the controller is set to on and activated, but if it is passed the argument `MANUAL` then the controller is set to off and deactivated. In this project the `SetMode` function is passed the argument `AUTOMATIC` for both roll and pitch at all times. `SetOutputLimits` controls the maximum and the minimum range of the PID controller output. This means that the PID controller only accepts output values within this range. For both the pitch and roll the `SetOutputLimits` has been given -100 as the minimum value and 100 as the maximum value. These two limitations have been chosen since the setpoint is when the output value is zero, and zero is the center value of this range, and also because the PID controllers for the pitch and roll should never be able to reach these possible output values. The last PID controller function in this example code is the `SetSampleTime`. This function sets the rate of which the PID controller algorithm should evaluate its input. In this example, both PID controllers are set to compute every 8 milliseconds meaning that the PID controller algorithm evaluates

at a frequency of 125Hz.

```
void loop() {
  for (int i = 0; i < 4; i++){ // Make 4 readings, each time it prints one.

    DeltaTime();  // Monitoring deltatime, and sets fail state if DT is higher then
         Deadling of 9ms.
    RawAccel();   // Reads Raw accel data
    RawGyro();    // Reads Raw gyro data

    IMUFusion();  // calculates estimated Quad position

    pidPitchStab.Compute(); // Computes how motors should react to IMUfusion
        changes.
    pidRollStab.Compute();  // Computes how motors should react to IMUfusion
        changes.

    StillFlight();          // Sets throttle speed.
    AdjustMotorSpeed();     // Adds PID speed to Stillflight
    SetMotorSpeed();        // Writes the speed to ESC/motors.
  }
  //...

  ...//
}
```

Example code 2.3: Part of the `Loop` function from PIDtest.ino

The `setup` function is used to setup and initialise all the information that is necessary for the system to know before flight, while another function `loop` is the function that will be running during flight. Example code 2.3 shows part of the code from the `loop` function. When the system enters the `loop` function it will repeatedly loop the function until it is interrupted by the user or some other unknown reason. In this function the `Compute` function is called, this function is probably the most important PID controller function since it is the function that contains the PID controller algorithm. The system runs the algorithm every time the function is called, which is at the rate set by the `SetSampleTime` function. After each time the `compute` function is called three new functions are called, `StillFlight`, `AdjustMotorSpeed` and `SetMotorSpeed`. `StillFlight` first sets the variables that hold the speed for each of the four motors to equal the same, then `AdjustMotorSpeed` is called which uses the output from the PID controller to adjust the value of each one of these four variables so that each motor can rotate at the right speed for the quadcopter to be able to achieve its setpoint. The last function `SetMotorSpeed` is then called to apply these new values to their corresponding motors.

2.6.2 PID Tuning

For a PID controller to work as optimal as possible it first has to be tuned by finding the best-suited proportional, integral and derivative gain constants, for the controller to eliminate the error in the angles given by the IMU. There are several ways of tuning a PID controller and in this project a form of manual tuning has been used. For the tuning, the setup of the Arduino, IMU, ECSs and the motors will be as demonstrated in the component diagram on figure 2.4.

Before the tuning of the PID controller could commence, the system had to be placed in a fixed setup so that the environment of every test remained consistent. The setup

consisted of the quadcopter hanging from a string that was attached to the ceiling and tied to two of the quadcopter's arms, the propellers of these two arms were removed while the other two remained. Five more strings were tied from the bottom of the quadcopter to a weight standing on the floor, and four of these strings were attached to each of the arms of quadcopter, where the two strings attached to the arms with propellers were more loose than the other two, and the last string was attached to the center of the quadcopter. The quadcopter was also configured so only the two opposite motors with propellers were set to run as the setup would not work if all four motors were to run. This should not affect the tuning in any negative way since the quadcopter would still be able to oscillate. Figure 2.12 shows a picture of the setup.

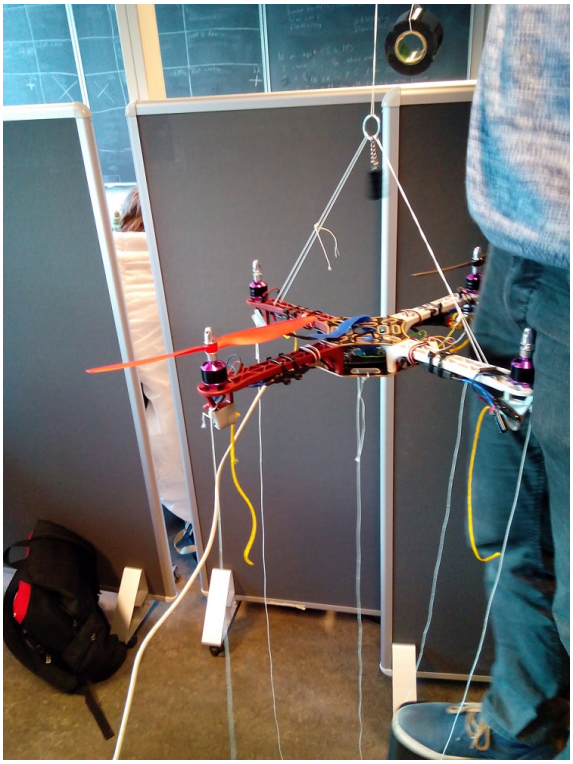


Figure 2.12: Picture of the setup when tuning the PID without the battery and IMU attached.

After the setup was complete the tuning process could begin. The process went as follows:

Step one was to set all gain constants to zero.

Step two the proportional gain was increased a little, after which the quadcopter was booted up. When the motors started spinning, an external force was applied to make the quadcopter start oscillating. After 8-10 seconds, the motors stopped and the data was recorded into a data sheet to see the fluctuation of the errors. This step was then repeated until reaching a proportional gain with steady oscillations in the errors.

Step three the proportional gain found in step two would remain the same, but instead the derivative gain would be increased a little at a time. As in step two the

quadcopter was booted up, an external force was applied and the data was recorded. But instead of repeating this step until we got steady oscillations, it was repeated until reaching a derivative gain where the oscillations were critically damped.

Step four was to repeat step two and three until reaching a point where increasing the derivative gain constant does not critically dampen the oscillations.

Step five is then to set the proportional and derivative gain constants to the combination values where the best data was recorded.

Step six is the last step in the process. Here the integral gain constant was increased until the PID controller brought the quadcopter to stabilize at its setpoint.[23]

Results

As a result from the PID controller tuning process, a combination of proportional, integral and derivative gain constants were achieved where the controller removed close to all errors in the system, and only very small steady oscillations remained. The best combinations of proportional, integral and derivative gains were when the proportional gain was 0.4, integral gain was 0 and derivative gain 0.125, and when the proportional gain was 0.4, the integral gain was 0 and derivative gain 0.1375. Figures 2.13 and 2.14 are two graphs generated from the data recorded from these two combinations of proportional, integral and derivative gain constants. The graphs show that the PID controller was able to quickly dampen the oscillations, caused by an external force, and then keep the quadcopter at a stable position, which is the goal of this task to keep the quadcopter stabilised. All the data from the tuning process can be found in appendix A.5.

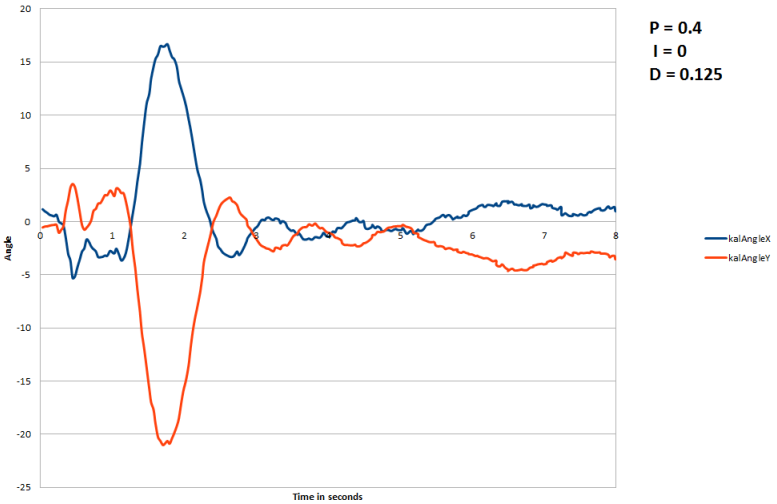


Figure 2.13: Proportional gain constant = 0.4, integral gain constant = 0, derivative gain constant = 0.125

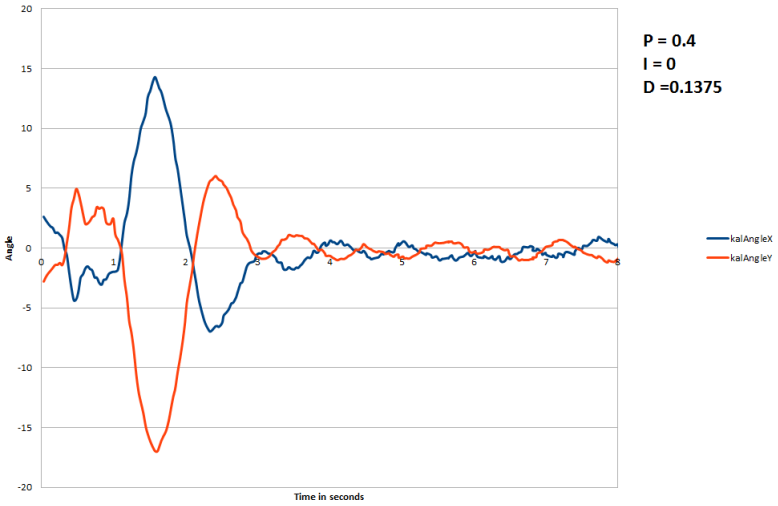


Figure 2.14: Proportional gain constant = 0.4, integral gain constant = 0, derivative gain constant = 0.3175

Problems started to occur when the quadcopter was removed from the test environment to a non-test environment. A single string was attached directly underneath the quadcopter with the quadcopter placed on the palm of a hand with the string between two fingers. Doing this the string can be given some slack and it will act like a completely free environment, but if the quadcopter started to oscillate the string could be tightened and the quadcopter would be pressed towards the palm of the hand. It turned out that the PID controller did not work as well in this environment. The quadcopter started oscillating in a random manner and was in no way able to stabilize itself.

These tests showed promise in the test environment but the PID controller worked nowhere near as well in a free environment. The conclusion from final results from the tuning of the PID controller is that the IMU cannot get proper readings when flying in a free environment because of the noise generated by the rest of the system. Because of these continuous problems with the IMU and having the need to better manage the remaining time, it has been decided that the IMU has to be removed from the system, and instead implement a flight controller in its place, which has a built-in PID controller and is designed specifically for multirotor crafts, such as quadcopters [24].

2.7 Flight controller

This section will contain information about what a flight controller is, what it can do, and what flight controller will be used in this project.

As for information about flight controllers the following references were used: [25, 26]. For information about the flight controller used in this project, the following reference is used: [24].

A flight controller is a board containing both a microprocessor with pre-programmed firmware and sensors for achieving flight. The software on a flight controller will utilize sensor fusion and often use PID to control its connected motors. The motors can often

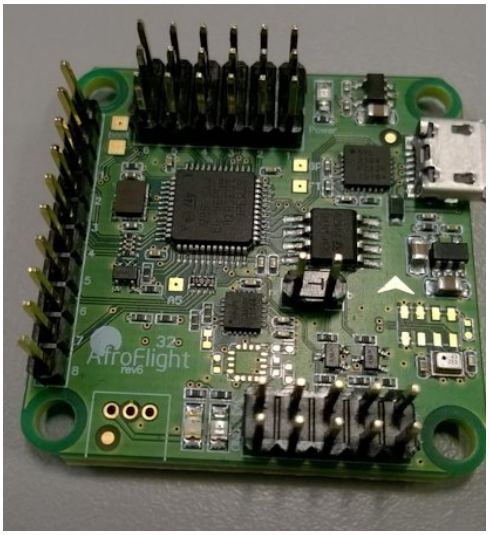
easily be connected to the flight controller, by connecting the wires from the ESCs directly to the flight controller board. When that is done, the quadcopter is more or less ready to fly. To control the flight controller, radio signals are used, often by using a remote control or by interfacing the Arduino Rx and Tx ports with the complementary ports on the flight controller.

The sensors used on flight controllers are usually five types of sensors:

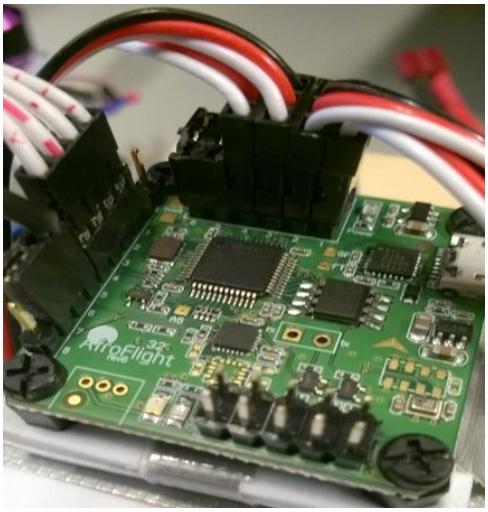
- **Gyroscope:** As mentioned in section 2.1, is used for measuring or maintaining orientation based on the principles of angular momentum. But a gyroscope can not alone detect change in direction.
- **Accelerometer:** To detect change in direction an accelerometer is used. Besides being able to tell about change in direction, an accelerometer can also measure the force of gravity.
- **Magnetometer:** Can detect variations in the earth's magnetic field, and work as a compass, which is used to counteract drifting in the yaw axis
- **Barometric pressure sensor:** Can detect change in altitude, which is used to tell if the quadcopter is moving up or down.
- **GPS antenna and receiver:** Which gives the flight controller the ability to tell where it is on earth in longitude and latitude.

2.7.1 Acro Naze32 V6

The flight controller chosen for this project ended up being the Acro Naze32 V6, which can be seen on figure 2.15a, which has got the essential features.



(a) With nothing connected.



(b) Connected with ESCs (white, red and black wires in the back) and RC input (white wires).

Figure 2.15: The Acro Naze32 flight controller

The features for this flight controller can be seen here[24]:

- Modern 32-bit ARM processor

- MEMS gyro, accelerometer, barometer.
- Several flight modes (auto-level, altitude hold etc.)
- Flexible RC input (supporting standard, CPPM, Spektrum satellite)
- Onboard USB for setup and configuration

It comes with pre-programmed firmware to process the sensor data (sensor fusion), controlling and adjusting the ESCs/motors, with PID that is configured by the flight controller automatically, process the radio signal, and giving several different flight modes. As mentioned in the previous text, the ESCs and RC input connects easily to the flight controller's designated pins. When this is done, the quadcopter is able to fly with the remote control. See figure 2.15b for the flight controller with ESCs and Arduino connected, where all the white wires are connected to the Arduino, and the black, white and red together are the four ESCs.

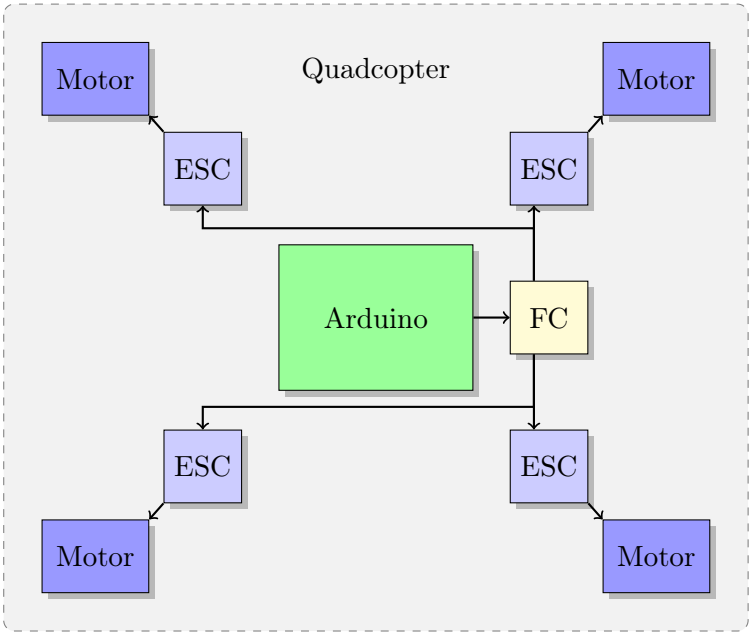


Figure 2.16: Component diagram over the quadcopter when using the Flight controller

With using the flight controller instead of the Arduino and IMU combination, the component design will now be as seen on figure 2.16.

2.8 Test of the flight controller

In this section we will describe how we configured and implemented the flight controller. First comes a description of the program used to control and check the flight controller. Then followed by a test performed with a remote control, to assure that everything works as planned.

2.8.1 Baseflight - Configurator

Baseflight - Configurator is an extension for the Google Chrome web browser, that according to the description is:

"Designed to simply updating, configuring and tuning your flight controller." - Baseflight description

This tool works well with the used flight controller, described in the previous section 2.7.1. When connecting the flight controller to a PC, with a micro USB cable, Baseflight quickly finds the flight controller. When connecting the flight controller, Baseflight shows a graphic of a quadcopter, to visualize how the flight controller is turning, as seen on figure 2.17. This tool is used to perform and set up the test described in the following subsection.

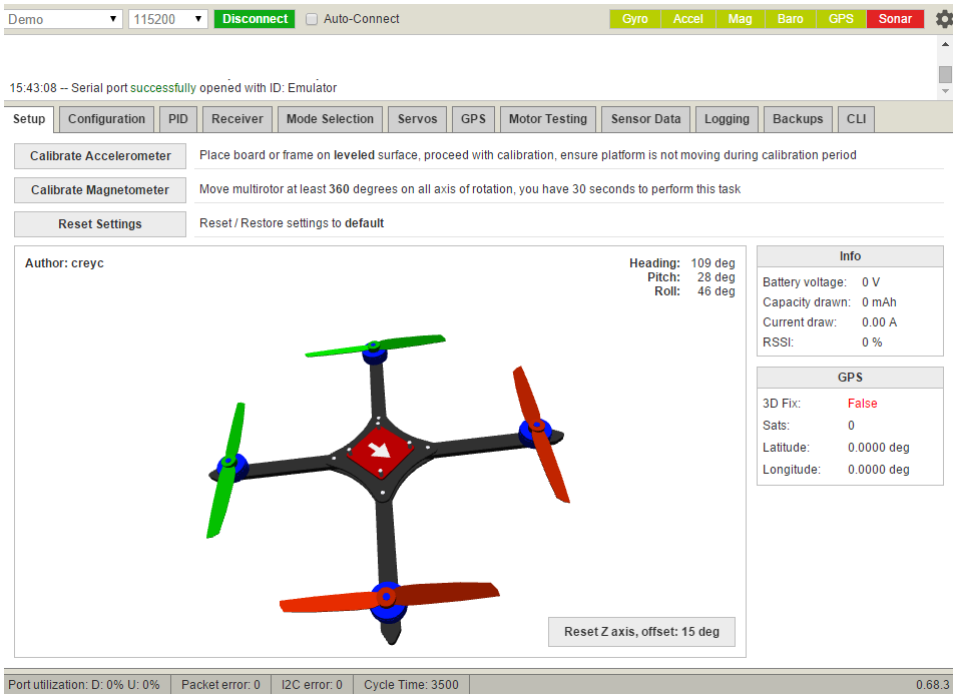


Figure 2.17: A screenshot of the setup tab in Baseflight after connecting the flight controller.

2.8.2 Purpose

The purpose of this test is to find out, if the flight controller is able to keep the quadcopter stabilized. By using a remote control, we hope to eliminate some of the uncertainties, there would have been in using software right away. It also gives us the power to turn it off, if something goes wrong, or it gets too close to an object. Thus we are able to test the flight controller, the motors, propellers and the quadcopter in whole, isolated from software, variables and error sources.

2.8.3 Setup and execution

First we connected the flight controller to a PC, and then turned on the remote control as seen on figure 2.18 and used Baseflight to calibrate the motors. This is done as seen on figure 2.19, by first turning the throttle all up, and then all down. This way it knows the whole span of possibilities. Then we made sure if we could control the different parameters like throttle, roll and pitch.

Figure 2.18: A picture of the remote control used in the test

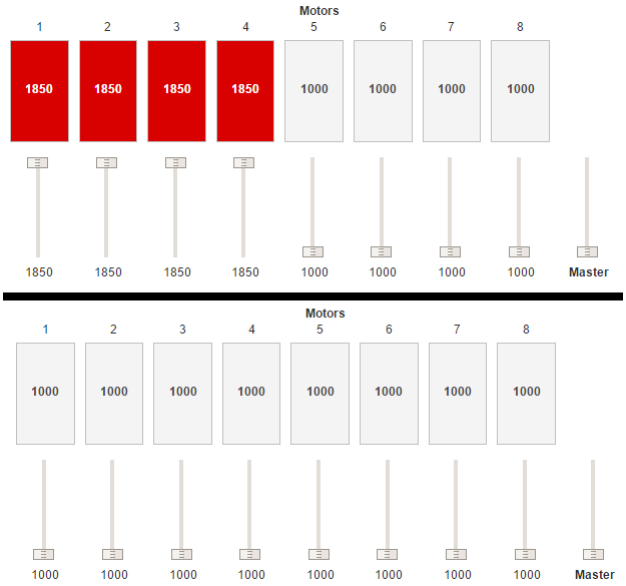


Figure 2.19: A screenshot of the motortesting in Baseflight. First all up, then all down.

Afterwards we tried powering the quadcopter with the battery, and see if we could control the motors with the remote control. Then we slightly increased the throttle, until almost liftoff, but then we experienced total failure. When one side of the quadcopter started rising up above the other side, instead of giving the opposite side more throttle, to level the quadcopter, the flight controller increased the already higher side, resulting in a flip-over, and an immediately end to the test. This turned out to be an error in the setup of the flight controller, so the motors were reversed. We then made sure that the motors were as seen on figure 2.20. This is a configuration tab in Baseflight, that enables you to adjust the placement and direction of the motors. After making sure the flight controller was setup to the correct motors, we tried again with the same approach. Slightly increasing throttle

until liftoff, this time with much better results. We managed to keep the quadcopter in the air, and control it with the remote control.

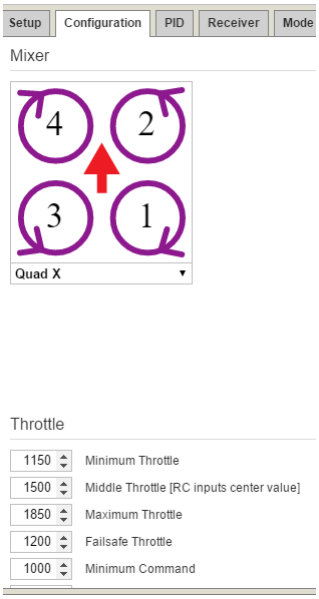


Figure 2.20: A screenshot of the configuration tab in Baseflight. Here you can choose the placement and directions of the motors.

2.8.4 Results

We managed to use the remote control and the flight controller, to get the quadcopter flying fairly stable and still in the air, and also to fly and turn around. This means that the flight controller is working as intended, and that it is able to adjust the motor speed properly to keep the quadcopter leveled at all times. The quadcopter should therefore be ready to the next step, which is to replace the remote control, with software.

2.8.5 Arduino flight controller communication

In this subsection we will describe how we used the Arduino for communicating with the flight controller, and how we made sure the flight controller responded correctly to our commands.

In the previous subsection 2.8.2 we succeeded to make the flight controller respond to the signals from a remote control. The next step was to replace the remote control with an Arduino, and use a program written in advance to give the input to the flight controller. To see how the flight controller behaved, we again used the Baseflight configurator as described in subsection 2.8.1.

The software running on the Arduino, used to give input to the flight controller, will be described in detail in section 7. In this part we will only focus on describing the part of getting the communication between the Arduino and the flight controller to work. The purpose is to test whether the flight controller will respond to the commands given by us through the Arduino.

First we connected the Arduino to the flight controller, and then we connected both the Arduino and the flight controller to a PC with two USB cables. In the time of actual flight the Arduino and flight controller will of course be powered by battery, but for the test, it was sufficient with an USB cable. Then we uploaded our software to the Arduino, and then opened the serial monitor window on the PC. Next we opened the Baseflight program, in the receiver tab, as seen on figure 2.21. The left part of the figure is the serial monitor of the Arduino software, and the right part is the Baseflight receiver tab. In the serial monitor screen we gave the command `arm` which then gave us the fluctation seen below the number '1' in the figure 2.21. This lasts for exactly 5 seconds, in which time a new command must be given, or the quadcopter will go back to an *unarmed* state from where it can not do anything. But during the *armed* state we gave the command `takeoff`, which is then seen at the number '2' in the figure, where throttle goes from 1000 (which is the lowest, and therefore the same as standing still) to 1400, and then is steady at 1400. This number is a fixed number in the Arduino code, but only works as a placeholder. The real throttle value for flying steady above the ground will have to be found by testing.

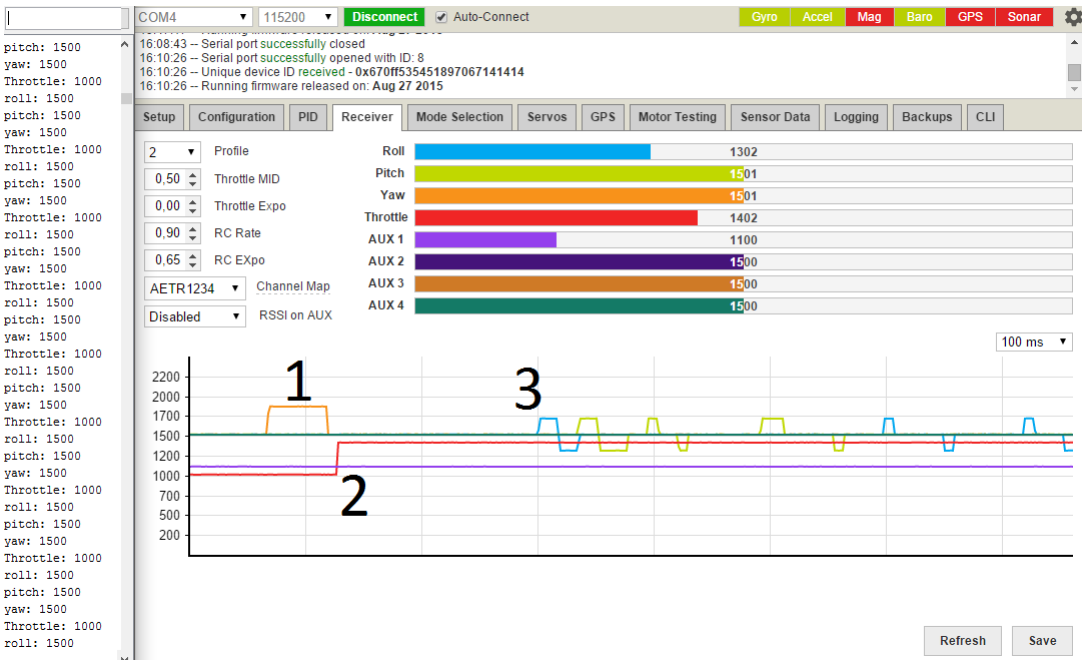


Figure 2.21: A screenshot of the receiver output tab from the flight controller, with the software running

At the number '3' in the figure, it gets the input `left`, and then the roll rises from 1500 to about 1700 in two seconds, before getting back to the default state of 1500. And then follows a row of commands like `right`, `backward` and `forward`. As we can see on the figure 2.21 the blue line represents the roll value, where when it gets above 1500 is `left`, and below is `right`. The same applies to the yellow line, which represents the pitch value, where above the line is `forward` and below the line is `backward`.

The flight controller works as expected, and this concludes this section about it.

Task conclusion 3

In this section, task one will be concluded upon, with regards to whether or not task one fulfills its requirements.

The requirements for task one are the following:

- The system should be functional using nothing but the hardware and software placed on the quadcopter frame itself.
- The quadcopter should be aware of its alignment to gravity, and then use that information to change the speed of the motors, based on the difference between the targeted alignment and the actual alignment.
- The system should be able to carry and lift the hardware needed to complete this task and future tasks.

Most of the components chosen in section 1.2 for task one, were within requirements. The quadcopter assembly kit that includes the frame, ESCs, and actuators quickly proved to be the correct choice. The actuators were very powerful, and as such there were no problems with the downward forces required for lift. The ESCs were able to receive input to designate speed, and hold that speed until a new input was received. The quadcopter frame's size was perfect, as it could contain everything needed for task one.

There was, however, one component that proved to be a bad choice for task one, which is the GY-85 IMU. The IMU failed to live up to expectations, even after a prolonged period of problem solving with the IMU, and the continuous problems with the IMU delayed the project for quite a while. Due to time constraints, it was decided after all the hard work trying to solve the problems with the GY-85, to replace it with a flight controller.

The flight controller chosen as a replacement for the GY-85, was the Acro Naze32 V6. Using this flight controller component, with firmware alone, we were able to utilize the sensor fusion between the gyroscope and accelerometer, and PID motor control, that we were trying to achieve with the GY-85. As seen in section 2.8, by using the flight controller we were able to achieve the stabilization needed to be able to conclude task one, and continue with the next task.

Part III

Task Two

Introduction 4

After stabilization of the quadcopter has been achieved, it is possible to move to task two, which is the next stage in the development of this project. The purpose of this task is to apply the ability of movement to the quadcopter while in flight, and make the quadcopter able to detect and avoid flying into obstacles. The requirements for task two is as following:

- **Quadcopter movement:** The quadcopter must be capable of flight in all directions (left, right, forward, backward). The quadcopter must be able to do this automatically, and by external input. In order to test this task, quadcopter takeoff and landing must be accomplished as well.
- **Obstacle avoidance:** The quadcopter must be able to detect obstacles, and avoid them as necessary. Being able to detect obstacles is very important, as the quadcopter is supposed to fly around without help from any users. This is especially true if the flying goes on indoors, as there are a lot of close obstacles, which increases the likelihood of the quadcopter crashing.

The quadcopter has to be able to fly on its own without crashing, this will be achieved using sensors that measure the relation between the quadcopter and its surroundings.

In this chapter, the specifications for the used sensors will be analysed, and the chapter will conclude on these specifications by testing the sensors. Finally, a test of the complete embedded system implemented in task two with regards to the fulfillment of the requirements for the task will be laid out.

4.1 Requirements

- The system must be able to hold itself at a fixed altitude.
- The system must be able to autonomously perform takeoff and landing.
- The system must be able to change its alignment to gravity, to achieve controlled movement by adjusting its motors speeds, both automatically and by external input.

- The system must be able to avoid obstacles, by reacting to its surroundings which are detected by sensors.
  - The reacting to the surroundings should include three different states; critically close, dangerously close, and cautious.
  - When in the critically close state, the system should shut down all motors within 50 ms.
  - When in the dangerously close state, the system should send a reaction, to move in the opposite direction of the obstacle, to the flight controller within 200 ms.
  - When in cautious state, the system should restrict incoming signals, from any external source, from giving a command to move in the direction of the obstacle.

It is important to design the software in a manner that will ensure the deadlines are met in time.

Analysis 5

The analysis of task two, includes determining what sensors should be used to detect the environment, and how we can communicate with the quadcopter when it is in the air.

5.1 Choosing distance sensor

In this section, different technologies available for sensing distance between a static and non-static position will be analysed, and one will be chosen as the distance sensing component for task two, so objects can be detected and avoided. The technology chosen is based on the pros and cons of the technology, and if it can be implemented using the Arduino platform.

Ultrasonic

The first technology is ultrasonic distance sensors. These sensors output a sound, which is almost constant in air, and the time between sending and receiving this sound is used to calculate the distance the soundwave has traveled. Sound waves bounce off walls and return to sender by this method, so they may be very useful for detecting objects that have to be avoided without the need for extra calculation. These sensors are most useful at the lower ranges.[27, 28, 29]

Laser

The second technology is photoelectric distance sensors, or lasers. These sensors transmit light, and the light reflected from this transmission by the target is analyzed. The distance is measured by calculating the speed of light and the time taken for the light to reflect back to the receiver, and can be very useful for long distances since the difference in position compared to the speed of light is very limited.[27, 28]

LIDAR

The third technology is LIDAR, and this technology works much like the laser sensor. A LIDAR instrument works by shooting a laser at a reflective material that can change its orientation after each light pulse, and thus one LIDAR is sufficient to get distance readings around the LIDAR. The LIDAR technology however requires knowledge about its current

position, so that has to be determined in order for the LIDAR to output reliable data, and this can be done by using data supplied by an IMU.[30, 31]

RADAR

RADAR is the fourth technology that could possibly be used to determine distance to an object. RADARs work by transmitting radio waves, and calculating the distance to objects that have reflected a radio wave back. RADARs operate by calculating the distance from the time traveled by the radio waves, much like LIDARs. RADARs are primarily used for long distance measurements, and for detecting moving objects by integrating doppler effect calculations in the distance measurement.[32, 33, 34]

Camera

The final technology that could be used to determine distance to an object is stereo cameras. By creating an illusion of depth using two pictures taken at different known positions, with the two cameras being horizontally aligned. The pictures have to be taken at the same exact time, if movement is involved, the measurement of distance between the cameras and the photographed objects can be imprecise.[35, 36, 37, 28]

Choosing a technology

Each technology has its own advantages and disadvantages when used in a project of this size. Laser sensors are focused, and can therefore precisely determine the distance to any point depending on how many laser sensors are used. Ultrasonic sensors require relatively few sensors to get full coverage, and several ultrasonic distance measurement components exist for the Arduino platform. RADAR is not as easily implemented as ultrasonic sensors on an Arduino, and as RADAR primarily operates at distances far greater than what is needed for this project, it is not a viable technology. Using stereo cameras to measure distance requires heavy computation, to distinguish the two pictures from one another, so this technology may require too much processing power as the distance has to be measured all around the quadcopter.

This leaves three technology choices; ultrasonic distance sensors, laser distance sensors, and the LIDAR that is essentially a spinning laser distance sensor. Ultrasonic distance sensors are very cheap, and widely used in conjunction with the Arduino platform, so placing enough sensors on the quadcopter to cover all sides will be relatively cheap. Laser distance sensors have to be placed all around the quadcopter as well, to get full coverage, but the laser has no angular coverage, so for a full coverage of the quadcopter a lot of sensors are needed, and that is not a viable choice. LIDAR is the only choice that can compete with ultrasonic distance sensors for this project, as it has no obvious disadvantages since a single laser is used, so it is relatively cheap, but the LIDAR would be hard to place on the quadcopter. The choice is therefore between ultrasonic distance sensors, and LIDAR.

Ultrasonic distance sensors will be used in this project, due to the availability of the sensors and Arduino implementation information.

5.2 Sonar sensor (HC-SR04)

This task requires the addition of the distance sensor. The distance sensor is used to measure the distance to nearby objects, and gives information that can be used in the embedded system to determine whether or not the quadcopter has to avoid an object.

The distance sensor chosen for this task, is the *HC-SR04* ultrasonic distance measuring module for Arduino, information gathered for the module, is from the vendor.[38, 39]

The specifications for the HC-SR04 are as following:

- **Working voltage :** 5V(DC)
- **Static current :** $<$ 2mA
- **Output signal :** High level 5V, low level 0V
- **Sensor angle :** Max 30 degrees
- **Detection range :** 2 cm to 450 cm
- **Precision :** 3 mm
- **Connection modes :** VCC / trig(T) / echo(R) / GND

The module works by supplying a sequence of at least 10 microseconds of a high level (5V) signal to the trig pin, after which the module will send out a signal and automatically detect a returned signal. If a returned signal has been detected, the output (echo) from the module will be high level (5V).

5.2.1 NewPing library

When using the HC-SR04 sensors, an Arduino library named NewPing is used to communicate with the sensors and will be described in this section.

The NewPing library has several features that helps a lot when communicating with the HC-SR04 sensors. The library fixes sensor response lag if there is no return signal, and makes it possible to set a max distance after which a received return ping from the sensor's ping is marked as not returned. The library can also convert the ping response time in micro seconds into distance in centimeters or inches.

When using the NewPing library, the ports used for the trigger and echo pins on each of the currently in use sensors must be initialized with the NewPing sonar constructor. The constructor requires information about the trigger and echo pin for the sensor being initialized. A max distance can also be set to any value in centimeters, and the default value is 500 cm. See figure 5.1, to see how the HC-SR04 sensor is connected to the Arduino, where the trigger-pin and echo-pin is pin-X and pin-Y on the figure.

An example constructor for a setup of a single sensor using pins X and Y as the trigger and echo pins on the Arduino, and a maximum distance of 200 cm: `NewPing sonar(X, Y, 200)`

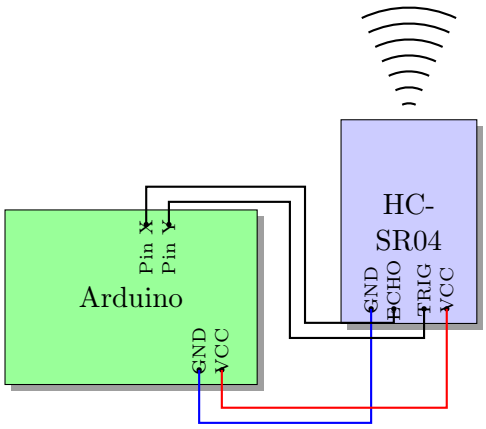


Figure 5.1: How the HC-SR04 sensor is connected to the Arduino

After the sensors have been initialized with the NewPing sonar constructor, several ping methods can be used from the library. The following method is the one used for this task, which sends a ping to the sensor, returns the echo time in microseconds or 0 (zero) if no ping echo within set distance limit:`sonar.ping()`

5.2.2 HC-SR04 testing

To make sure the HC-SR04 sensors are as specified, each sensor has to be tested to determine their individual specifications. At the same time, they will be tested to determine how they can be used optimally in this project.

The specifications that have to be tested for each sensor, and how this will be done, is as following:

Detection range and precision testing to affirm the specifications of the detection range from 2 cm to 450 cm. The sensor needs to be statically mounted, and an object has to be moved within the specified range to affirm the specification of each sensor's detection range.

Sensor angle of detection for each sensor has to be determined, in order for an optimal sensor placement on the quadcopter. An optimal test would include testing detection from objects being moved slowly into the field of view of the sensor at several distances, to determine the optimal angles at different distances for sensor detection.

Sensor interference has to be tested to see how several sensors interfere and affect each other. There might be a chance that a sensor can react on another sensors' signal, and thereby give back wrong information.

Measurement time to figure out how long it takes for a measurement to be made. This can be important for scheduling, depending on how long it takes to make a measurement.

Angled wall test was added after working with the sensors in task 4. It was discovered that when a sensor was pointed at an angled wall, it would return inconsistent and wrong results, and therefore had to be tested further.

How the sensors were tested can be read in appendix B.1.3, and the results can be read in the next section.

5.2.3 Results

For the tests of both range and precision, detection of angle and interference several measurements were made that gave data, which can all be seen in appendix B.2.

Range and precision

Starting by looking at the range and precision data it can be seen that for a distance at 1.5 m and below, that every sensor's precision is within 2 cm. At a 2 m distance the sensors' precision start to deteriorate just a little, seeing measurements from the sensors being within 4-5 cm of the actual distance. At measurements around 3 m the precision deteriorates even further, and the sensors' precision is within 6-7 cm. Since the sensors will be used to look for obstacles for when the quadcopter is flying, it is optimal to set the max distance at 1.5 m or below, given these results. This distance should be enough beyond the hardware of the quadcopter, and still be plenty for the quadcopter to react to the environment, should it detect something close.

Distance (cm)	**150**	**200**	**Diff.**	**150**	**200**
Sensors					
1	152	203		2	3
2	152	203		2	3
3	151	202		1	2
4	152	203		2	3
5	152	203		2	3
6	152	204		2	4
7	152	204		2	4
8	152	204		2	4
9	152	203		2	3
10	151	203		1	3
11	152	205		2	5

Table 5.1: Part of the results from distance testing

Angle of detection

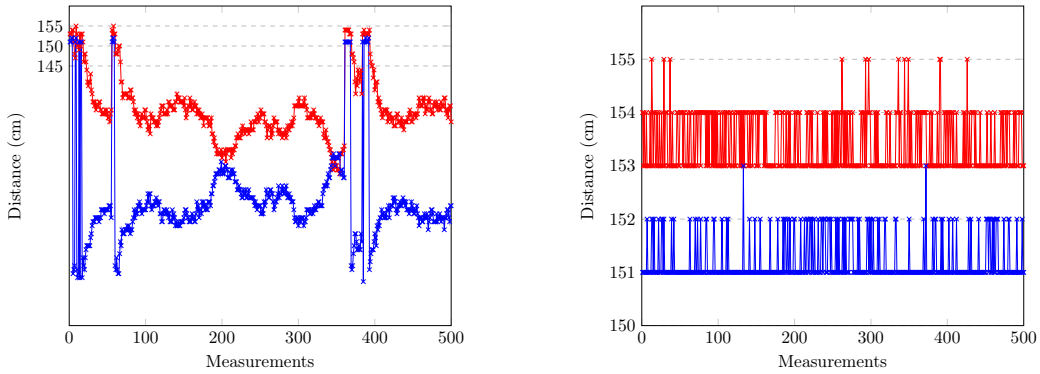
The test of the angle at which the sensors detect an object, was done with two different objects, as mentioned in the setup for testing in appendix B.1. Compared to the listed value of max 15 degrees to each side of the sensor in the datasheet, both tests can conclude that this is not entirely true. Looking at the data from both test objects, table 5.2 or table B.4 and B.5 in appendix, it can clearly be seen that it is rare for a sensor to detect an object at only 15 degrees or less in coverage, but that it mostly detects objects at a degree larger than this, even at a distance of 1.5 m. Looking at the round object, which had an overall lower angle of detection than the flat object, it will give a good detection angle at a measuring range of 1.5 m, which is in the optimal measuring distance of the previous test. Depending on the sensor chosen, there can be a detection angle from 14 to 23 degrees on each side, with only two measurements being at 14 degrees.

Object	Flat				Round			
Distance	150		200		150		200	
Sensors	left	right	l	r	l	r	l	r
1	40°	37°	34°	31°	22°	18°	12°	12°
2	38°	35	33°	30°	18°	20°	14°	12°
3	40°	40°	32°	35°	22°	23°	13°	14°
4	34°	35°	32°	30°	19°	21°	12°	10°
5	38°	35°	33°	31°	18°	15°	6°	8°
6	34°	33°	26°	32°	20°	19°	x	10°
7	47°	34°	34°	28°	22°	16°	12°	7°
8	37°	31°	32°	25°	16°	14°	4°	7°
9	35°	38°	30°	32°	15°	20°	11°	12°
10	35°	36°	25°	29°	14°	16°	0°	8°
11	40°	38°	33°	32°	19°	22°	1°	12°

Table 5.2: Part of the result from the angle of detection test

Interference

For testing interference between sensors, three tests were made. The first one, on a small distance, to verify that there is a possibility that the sensors might interfere with one another. After the test verified that there is a risk of interference between the sensors, the test was scaled up to 1.50 m, as that is the result from the precision test, and at 2 m. At 1.50 m the delay between the pings from the two sensors used were changed, until the data matched the data from when the sensors were measuring alone. This resulted in a delay between pings on 3.2 ms, which can be seen on the graph 5.2b, where the data is stable, compared to graph 5.2a with 2.9 ms delay, where it is clear that there is interference. The test with 3.1 ms delay almost showed stable, but as it can be seen on graph B.6b, interference still could not be ruled out. Using a delay of 3.3 ms, see graph B.6d, concludes that 3.2 ms is enough of a delay to avoid getting interference.



(a) 2.9 milliseconds delay

(b) 3.2 milliseconds delay

Figure 5.2: Graphs showing clear difference between different delays at 1.5 m

The test at 2 m showed difficulty, in that it was not possible to get data that showed interference no matter which delay was used between the pings from the sensors, using two sensors. Only when connecting a third sensor could interference be detected, which was only visible on one of the sensor's values, see graph B.7a and B.7b. As can be seen

on the last graph, a delay of 5.6 ms is needed at a distance of 2 m. Since the other test concluded that if a distance of 1.5 m is used, the sensors will still give good and reliable values when the used delay between the HC-SR04 sensors was 3.2 ms. In this project, the delay used will therefore be set to 3.2 ms, at a max distance of 1.5 m.

Measurement time

To find the worst-case measuring time for each sensor at 1.5 m, a single test was made for each sensor, which gave several measurement times, where the worst-case measuring time of each of the sensors can be seen in table 5.3

Sensors / **µSec.**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1	9100	9096	9096	9080	9112	9096	9076	9104	9080	9096	9104

Table 5.3: The worst-case measuring time for each of the 11 HC-SR04 sensors at 1.5 meters distance

As seen in table 5.3, the worst-case measuring time for the HCSR-04 sensor is 9.112 ms. Looking at the speed of sound, the results, table B.3, of this test makes sense. The speed of sound is defined as $340.29m/s$, and calculating the time it takes for sound to travel 1.5 m twice, from the sensor to the wall and back again, $((1.5/340.29) * 2) * 1000$ will give 8.816 ms. The result of the test will be used when scheduling between measuring distance and updating the PID on the quadcopter.

Angled walls

After doing the first physical test of task four, see section 17.2, it was discovered that the HC-SR04 had problems, when pointed at a wall with an angle. To make sure the issue was understood and investigated this addition was made to the sensor test.

The situation with angled walls makes sense since the sensors use sonar that will bounce from the objects it is hitting, which means if the angles of a corner is perfect, the ping might bounce from one wall to another, and then back to the sensor, which will then show a further distance to the first wall because of this. It was first thought that this problem could be solved by comparing the data from two sensors, to check if we were hitting an angle or not. But after doing some further testing, which can be read in appendix B.1.2, it can be concluded that with the sensors being used in this project, it is not possible to operate as wished with angled walls.

From the first test, figure B.5a, it can be seen from the graph showing the data for the 25 degree wall, figure B.8, that there is only a few data points at 50 cm where the sensor do not receive a sonar-signal back, which can be seen by a distance of 0 cm. But looking at the distance of 70 cm the results changes drastically, and there are several more measurements where the sensor does not receive a sonar-signal back. Another problem with an angled wall can be seen in figure B.9 at 50 cm, where the sonar receives signals back and measures longer than the actual distance. The sensor here receives signals that have bounced off the wall, might have hit some surrounding object or just hit the wall in such an angle,

that it bounces around before getting back to the sensor, and then representing a further distance.

For sharp corners, see figure B.5b, the sensors are not much better. Measurements at a 60 degree sharp corner shows that at a small distance, 50 and 60 cm, the returned distance from the sensors vary a lot. Looking at the graphs B.11a and B.11b, it can be seen that the distance returned from the sensors, at both 50 and 60 cm, mostly jumps from 0 to around 100 cm. At a 90 cm distance from the wall the results get better, but it is still often returning 0, which means the sensor does not get a signal back. The results are close to being the same with a 90 degree sharp corner. At a distance of 50 cm the results do not jump that often to 0, see graph B.12a, but at a further distance such as 100 cm the results looks a lot like the earlier tests, see graph B.12b. This shows that the sensors do not interact well with sharp corners and will often show a wrong distance.

As for the last test, with the sensor pointing into a 90 degree corner, see setup in figure B.5c and B.5d, it was much clearer that this is not a situation where these sensors work well. Pointing the sensors with a 45 degree angle at the wall, with a distance of 35 and 36 cm gave two very different results. For a distance of 35 cm, the sensor returned a distance of 59 to 60 cm, see graph B.13a. By moving it a 1 cm outwards, the returned distance was around 95 cm, see graph B.13b. These results show that the sensors will not show the exact, or close to, distance when close to a corner.

5.2.4 HC-SR04 tests conclusion

By doing these different tests with the HC-SR04 sensors we are able to determine how to use the sensors within our project. Looking at the results we can conclude that for the best range and precision, and still having good coverage from the quadcopter, a max distance of 1.5 m is the most optimal these sensors can give us. The precision at 2 m might not be that bad, only 1-2 cm more deviation than at 1.5 m, but when looking at the angle of detection of round objects, it is clear that it is preferable to use 1.5 m as max range. The difference from 1.5 m and 2 m is somewhat big, depending on which sensor is used, but the worst combined angle of detection at 1.5 m being 30 degrees and the best being 45 degrees compared to the angle of detection at 2 m where the worst is 8 degrees and the best is 27 degrees. As for interference, the test shows that by using a delay of 3.2 ms between sensors, when working with 1.5 m being max distance, is sufficient to avoid interference. For the test with angled walls it is clear that these sensors, no matter if it is a sharp corner, a wall or a open corner, do not handle them well. Therefore it is chosen to only focusing on rooms with no angled walls in this project, and only further working with the quadcopter to be able to fly in such rooms.

5.3 Bluetooth module (JY-MCU)

To be able to communicate wirelessly with the Arduino, it has been decided to use a Bluetooth module called JY-MCU HC06, see figure 5.3, that can be connected with the Arduino and used with a computer. This makes it able to get information from the Arduino and the sensors and give commands from the computer simultaneously. There were no specific need for fast or long distance communication with the Arduino, so this

module was chosen because it was at our disposal, and because it was easy to implement to the Arduino platform.

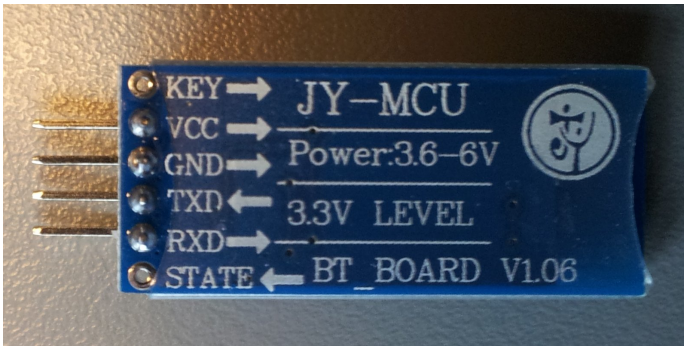


Figure 5.3: The JY-MCU HC06 Bluetooth module

As a source of information for technical specification and how to connect the JY-MCU to the Arduino the following references are used; [40, 41].

The technical specification of the JY-MCU is as following:

- Supply voltage between 3.6V to 6V DC
- Bluetooth SPP (Serial Port Protocol)
- Baud rate: 38400 bps.
- Easy to connect with any standard bluetooth module
- Requires no setup
- Connection Pins: Vcc, Gnd, TXD (TX), RXD (RX) (See figure 5.3)

The module works by connecting the TX and RX pins of the module with the opposites of the Arduino's, RX to TX, and TX to RX, to make serial communication possible with the Arduino's serial port[42]. The computer can then use Bluetooth to create serial communication to the JY-MCU.

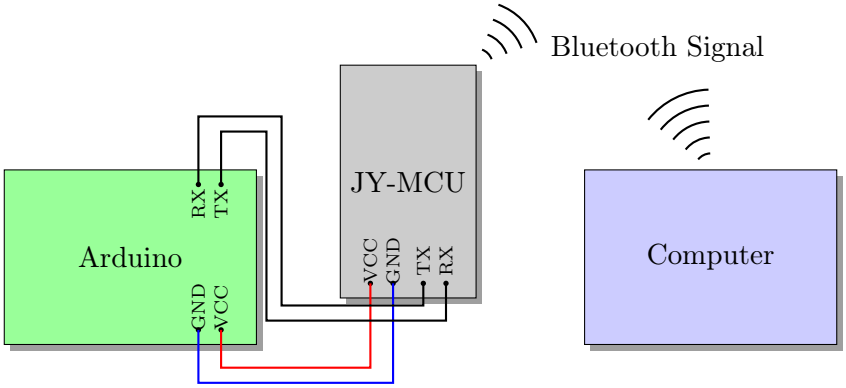


Figure 5.4: Diagram over the Arduino, JY-MCU and a computer

By doing this the function `Serial.print()` can be used on the Arduino to write to the JY-MCU module, which the computer then can read from. The computer can also

write to the JY-MCU module, which the Arduino can then read from with the function `Serial.read()`.

This way it is possible to send information from the sensors to a computer, and give commands to the Arduino and quadcopter while it is in the air simultaneously.

Design 6

When this task is finished, the quadcopter has to be able to fly by it self and avoid any obstacles in its way. The system will use distance sensors to register objects, but as the quadcopter itself is not designed to withhold nine sensors, an extra piece of material has to be created. This piece of material should be mounted to the quadcopter and at the same time be able to attach the nine sensors to it, with the sensors facing at the desired angles. This section will go through both the component design and code design of this task.

6.1 Component design

In this task new components has been added to the quadcopter. These components are the nine distance sensors (HC-SR04), four legs, a Bluetooth module (JY-MCU) and an extra piece of material designed to be an extension of space for the quadcopter, so all the new components can be mounted to the quadcopter. The reason there is exactly 9 sensors, is that there is 4 perpendicular (straight forward, left, right and backwards) and 4 diagonal in each corner, making a total of 8 sensors. The last sensor is pointing straight down, to measure the altitude. The new component design can be seen on figure 6.1

The extra piece of material is a board made of plywood with a surface area of 16×18 cm. The board has eight $13mm \times 3mm$ holes at the edge of the board with $45°$ between each hole, these holes are designed for eight of the nine distance sensors to be attached to. An extra of 16 holes have been drilled in the board, four holes near each corner of the board. These holes are for the legs which require four holes each to be mounted. Two additional smaller boards of same size are also created. These two boards are used to mount the larger board to the quadcopter as shown in figure 6.2. In figure 6.2a we see the two smaller boards on each side of the bottom layer of the quadcopter with the larger board attached to the bottom smaller board, the three boards are screwed together through the holes shown in figure 6.2b.

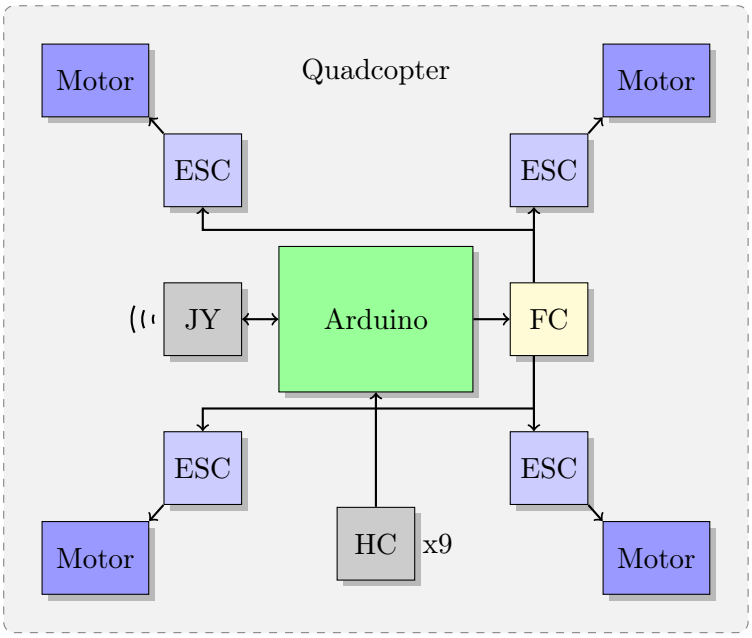


Figure 6.1: New component diagram over the quadcopter when using the Flight controller and HC-SR04 sensors

(a) (b)

Figure 6.2: The attachment of the wooden boards

In figure 6.3 we see all nine sensors and how they are mounted to the quadcopter. Here it is easy to see that the quadcopter can detect objects in all directions. The bottom sensor will be used to make sure that the quadcopter is at the same level above ground at all times.

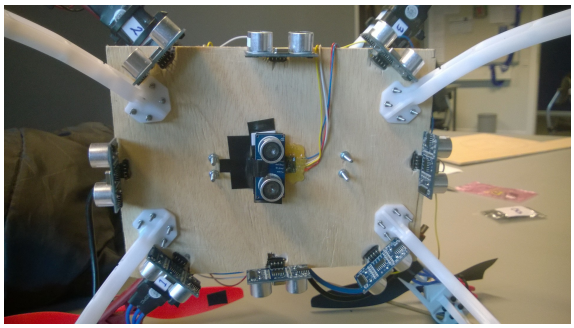


Figure 6.3: The quadcopter seen from the bottom with sensors attached

A Bluetooth module is also added to the quadcopter. This is to be able to communicate from a computer to the Arduino wirelessly and so the quadcopter can fly in a free environment. A diagram over the wiring of all the electronic components on the quadcopter can be seen in figure 6.4. The flight controller gets its power- and ground-connection from the ESCs while the analog connection is for the flight controller to control the ESCs, and the flight controller gets controlled by the analog connection from the Arduino. All the HC-SR04 sensors and the JY-MCU bluetooth module are all connected directly to the Arduino, where the JY-MCU module receives information from TX pin on the Arduino and deliver to the RX pin. The ESCs control the motors through the three wires between them.

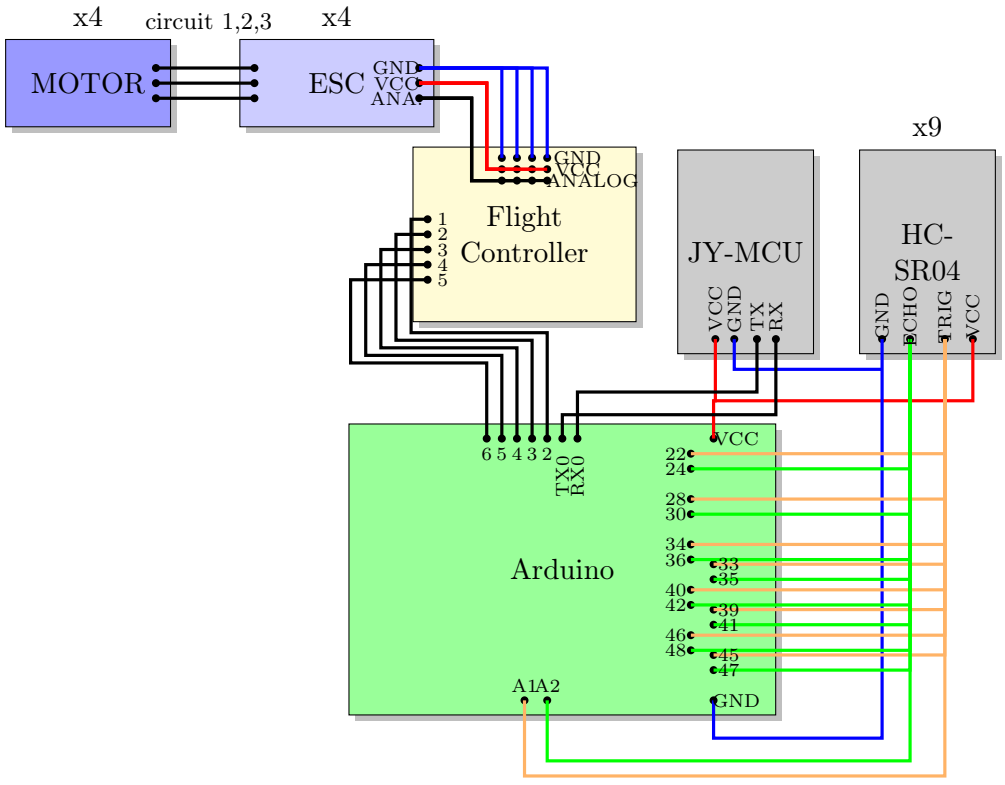


Figure 6.4: A wiring diagram over the electronic components on the quadcopter in this task

6.2 Code design

The code should be able to get information from the distance sensors and then make the quadcopter react in a proper manner when the sensors register a nearby object. The code should make the quadcopter stop or move away from an object after reaching a certain distance from the object. If the quadcopter has reached a distance where it can touch the object, then the code should be able to shut off all the motors, to limit the amount of damage done to the quadcopter or the object, as much as possible.

Even though it should be able to fly automatically and autonomously, it is also important the program has some user control functions, which the user can call at any time during flight. It should include a takeoff function which starts the motors and initiates the quadcopter to flight mode, move functions that can command the quadcopter to move forward, backward, right or left, a land function where the quadcopter performs a slow and steady landing, and a critical shutdown function that, when called, immediately turns off all motors. These functions are included for both safety precautions and if the user wished the quadcopter to perform a specific action.

The software has several different control constants, which should be easy to adjust. In the very top of the source code, the necessary definitions are therefore stated. This include things like the time the quadcopter should move when movement commands are given, the flight height for which it should stay above the ground, and the critical distance, for which it should turn off all motors. This will be explained in further detail in section 7. Some of these constants may need to be changed to the specific situation, like the flying height. By having these constants as definitions, it makes it easy and fast to make adjustments and changes.

To control things like how often the distance sensors are called, and how often the PID is updated, the program is designed in tasks, so that we can define the worst-case time it takes for running each task, and each part of the code. This is important for making sure we know how fast it finds obstacles, and that it strictly does it within the same time span, for each major cycle. We also do this to optimize how often we can read a distance sensor. To gain even more control, specific modes are designed, where each mode have special tasks, which can only be run in that particularly mode. As an example, the takeoff task can only be called when the system is in `takeoffMode`, where the systems waits for a command, to start flying. There are 6 different modes, and 14 different tasks. Some tasks will exclusively be called in one mode, whereas other tasks will be called in several modes. All the modes are designed as follows:

Modes relevant to the scheduler

deadMode No movement

takeoffMode This mode is a waiting mode, where the system waits for a person to give the takeoff command, which is sending the letter 't' from a computer to the Bluetooth module. This mode only runs for a maximum of 6 seconds, else it breaks out of `takeoffMode`. If the system receives the command 't' within the 6 seconds, it will shift to `inAirMode`.

inAirMode Gets the quadcopter up in the defined height above the ground. When the quadcopter has reached the desired height and is stable, it will shift to `flightMode`. The detection and avoidance of objects is also running during this mode.

flightMode In this mode the quadcopter should avoid obstacles if they come to close, as defined in the control variables. The system should also be able to receive and perform commands from the user, like left, right, forward and backward.

landMode It should slowly lower the setpoint for the PID, until it has landed, and then turn off the motors, when it has reached the ground.

dangerMode Should change the scheduler to perform the task of avoiding objects, until it breaks out of `dangerMode` again.

The scheduler itself is designed to do some busy waiting. The distance sensor needs a 3.2 ms wait time before a sensor can be pinged again, to eliminate interference. The rest of the tasks can be done within this time, and therefore some busy wait is needed to ensure that the waited time is over 3.2 ms. The wait is also made to rely on the clock of the Arduino, and therefore 1 ms is added to the wait, to account for drift in the clock. The major cycle of the scheduler is 13 ms, in which time all of the calculations can be completed, with no problems. The scheduler should also maintain the correct modes, and is responsible for only letting the appropriate tasks run in the correct modes.

This design will be the starting point for implementing the software for solving this task. The implementation will be described in the next section.

Implementation 7

In this section the implementation, based on the design described in the previous section 6, will be described and explained. This section includes relevant pieces of the code, and will cover the most essential functions and parts of the program. The software is written in the Arduino language, which is very similar to C.

7.1 Definitions

As the very first thing in the source code, a lot of different definitions are made. The most relevant are seen on example 7.1. The first ones are labeled control variables, and this contains the following definitions: `STEERING_TIME` defines how long the quadcopter should fly in a given direction. This means that if it is set to 500, and receives the command to go left, it will do so for 500 ms. `FLIGHT_HEIGHT` defines the height for which the quadcopter will stay above the ground in centimeters. The `RESTRICTION_DISTANCE`, `DANGER_DISTANCE` and `ERROR_DISTANCE` states for which distance it should not be able to go any nearer an obstacle, for which distance it should fly away from an obstacle, and for which distance it is so close, that it should just turn off all the motors and do an emergency shut down. The `MAX_DIST_BETWEEN_UPDATE` defines the maximum difference between readings from a distance sensor. When a sensor is read, it is done 3 times after each other, and sometimes one reading may be incorrect, and therefore not give correct result. To minimize the risk of trusting an incorrect result, it will compare 3 readings, and only trust the newest within the margin set by the `MAX_DIST_BETWEEN_UPDATE`. The last control definition is `DEADLINE_MAJOR` which describes how long a major deadline is. It is set to 13, which has been tested to be an optimal time, but it can be raised to make sure that no interference between the sensors affect the readings. This time is based on the sensor section where the time it took to detect an obstacle was about 9.1 ms at 1.5 m, and the wait time, to limit interference between sensor readings, which is 3.2 ms. After rounding up to account for clock drift on the Arduino, the major cycle is calculated to 13ms. After the 13 ms has been tested, and proven to be sufficient, it is also tested that all other computation the quadcopter needs can be run in between the sensor readings, in the wait of 3.9ms.

The next group of definitions are flight controller modes. This describes the different modes the flight controller can be set to. `HORIZON_MODE` finds a fitting degree, for the roll value, and the `RATE_MODE` finds a fitting degree pr. second to the roll. The value is set

accordingly to the flight controller.

```
//Control
#define STEERING_TIME 500           //in ms.
#define FLIGHT_HEIGHT 25            //in cm.
#define RESTRICTION_DISTANCE 60     //in cm.
#define DANGER_DISTANCE 40          //in cm.
#define ERROR_DISTANCE 10           //in cm.
#define MAX_DIST_BETWEEN_UPDATE 20 //in cm.
#define DEADLINE_MAJOR 13           //in ms.

//Flight controller modes
#define HORIZON_MODE 1100
#define RATE_MODE 1900

//Commands
#define CMD_ARM "a"
#define CMD_TAKEOFF "t"
#define CMD_LAND "la"
#define CMD_STOP "s"
#define CMD_RIGHT "r"
#define CMD_LEFT "l"
#define CMD_FORWARD "f"
#define CMD_BACKWARD "b"
```

Example code 7.1: Control variables flight controller modes and commands

The last group of definitions is commands. These define the different commands the user can send from a computer, using Bluetooth. The commands themselves are described later in this section, but here it is possible to set how a user gives the command. This has been set to only a single letter, to make it faster to write and read.

7.2 Scheduler

The scheduler is a fixed priority scheduler with different modes. Each mode has some specific tasks which can be called in that particularly mode. Since the scheduler almost handles every aspect of the software, the implementation of the different modes and tasks will be described based on the scheduler. The whole scheduler will be shown and described as different parts in this section.

The first part of the scheduler is seen in code example 7.2. This part consists of an if statement, that, when the system is in `deadMode`, calls the `Steering()` function, then waits for 10 ms before checking again. The `Steering()` function is used to write values to the flight controller. The function takes five different inputs. Four integers for setting respectively the roll, pitch, yaw and throttle, and a final integer which is between 1000 and 2000, that represent the different modes on the flight controller. The roll, pitch, yaw and throttle all go from 1000-2000. The roll, pitch and yaw are therefore set to 1500 which is exactly in the middle, and therefore leveled, and the throttle is set to 1000 which is the same as turning off the motors.

```
void loop() {
  if (deadMode){ //If critical error occurred, do nothing.
    Steering(1500, 1500, 1500, 1000, HORIZON_MODE);
    delay(10);
  }
```

Example code 7.2: The first part of the scheduler - deadMode

The next part of the scheduler, is for when the quadcopter is operating, which is for all other modes than `deadMode`. This part, which is seen in code example 7.3, consists of four different functions. These functions should run at all times, in any mode, except `deadMode`. The first function `DeltaSteeringTime()` ensures that each cycle takes exactly the time for `MAJOR_DEADLINE` which in this case is set to 13 ms. This means that it will busy wait until the time is met. The function also checks and warns in case a deadline is exceeded.

The next function is `UpdateSensor()` which is used for getting distance from the sensors. Because the sensors are not always 100% reliable, we take 3 inputs, and compare them to each other, and only use the ones within the margin of `MAX_DIST_BETWEEN_UPDATE`. The `UpdateSensor()` counts up 3 readings, and then calls another function `CalcSensor`, which is used for the comparison. `CalcSensor` does three comparisons, so that all the readings are compared to each other. If one read is above the `MAX_DIST_BETWEEN_UPDATE` compared with the two other readings, it is marked as 'odd' and will not be used. The most recent reading which is not marked as 'odd' is the used reading.

`DistanceRestrictions()` keeps track of which directions are restricted or dangerously close and when the quadcopter is leaving a dangerous area.

The last function is `ReadBluetooth()` which is a function called every major cycle. This is to make sure that all `CMD_STOP` are found immediately. This function is one of the only functions including an `Error`, which will make the system stop immediately.

```
1  else {
2    DeltaSteeringTime();
3    UpdateSensor();
4    DistanceRestrictions();
5    String commmand = ReadBluetooth();
```

Example code 7.3: The four functions used when the quadcopter is operating

The next part of the scheduler is seen on example 7.4. This is for when the quadcopter is on the ground, and ready to receive commands. The `NotArmed(command)` arms the system if the `CMD_ARM` command is received, which makes it ready for another command which is `CMD_TAKEOFF`, which will then put it in takeoff mode. When not in `inAirMode` the steering is set to the standard for being on the ground, as previously described.

```
1    if (!inAirMode){
2      //On ground mode
3      NotArmed(commmand);
4      Takeoff(commmand);
5      Steering(1500, 1500, curYaw, 1000, HORIZON_MODE);
6    }
```

Example code 7.4: How the quadcopter should behave when not `inAirMode`

The most comprehensive part of the scheduler, is the `inAirMode`, it can be seen in code example 7.5. The first function is `MaintainHeight()`, which is called every major cycle when in `inAirMode`. This reads from the sensor pointing towards the ground, and computes the PID, that changes the throttle to keep the quadcopter at the altitude set at `FLIGHT_HEIGHT`.

The `counterSteeringMode` is used after avoiding an object. Without this, the quadcopter would build up momentum while avoiding an object, and when the quadcopter then leaves the dangerous area it would level, but keep the momentum, thus still fly away from the object. This is simply a normal steering movement in the opposite direction, but for a shorter period of time, as set by `COUNTER_STEERING_TIME`. This could be made dynamic as future work, but it would need to be tested with position determination to eliminate drift interfering with the test.

Then follows an if-else statement, which starts with `!dangerMode`. This would mean that the quadcopter is flying normal, and the quadcopter also has another mode, `takeoffMode`, `flightMode` or `landMode`, that determines what the quadcopter will do automatically and what commands the quadcopter will react to. The `takeoffMode` increases in altitude, and switches to `flightMode` when stabilized flight is achieved. The `flightMode` enables steering, and switches to `landMode` when `CMD_LAND` is received. In `landMode` the PID's setpoint is slowly lowered until the ground is reached, then turns off all motors. Lastly it resets all modes, so the system is ready to fly again.

If the first `if`-statement is not true, that would mean the system is in `dangerMode`, and then the `avoidObjects()` function is called. The `avoidObjects()` function changes the roll and pitch accordingly. If an object is too close in front of it, it will decrease the pitch value to fly away from it.

The last line is `inAirSteering` which resets the roll and pitch values after a steer has been made, and lasted for the amount of time defined in `STEERING_TIME`.

```
1   else {
2     MaintainHeight();     //Changes the curThrottle, with the PID.
3
4     if (counterSteeringMode){
5         CounterSteering();
6     }
7
8     if (!dangerMode){
9     //Normal flight
10      if (takeoffMode){
11        Takeoff(commmand);
12      } else if (flightMode){
13        Flight(commmand);
14      } else if (landMode){
15        Land();
16      }
17    } else {
18      AvoidObjects();
19    }
20    InAirSteering();
21  }
```

Example code 7.5: In air mode part of the scheduler

To visualize how the different modes are connected, a flowchart has been made as seen on figure 7.1. It is important to notice, that even though there are two `takeoffMode`s on the figure, it is only to visualize the fact that the system can be in `takeoffMode` out of `inAirMode`. It is also important to notice that there are three `dangerMode`s to show that when it enters `dangerMode` it goes back to the same mode.

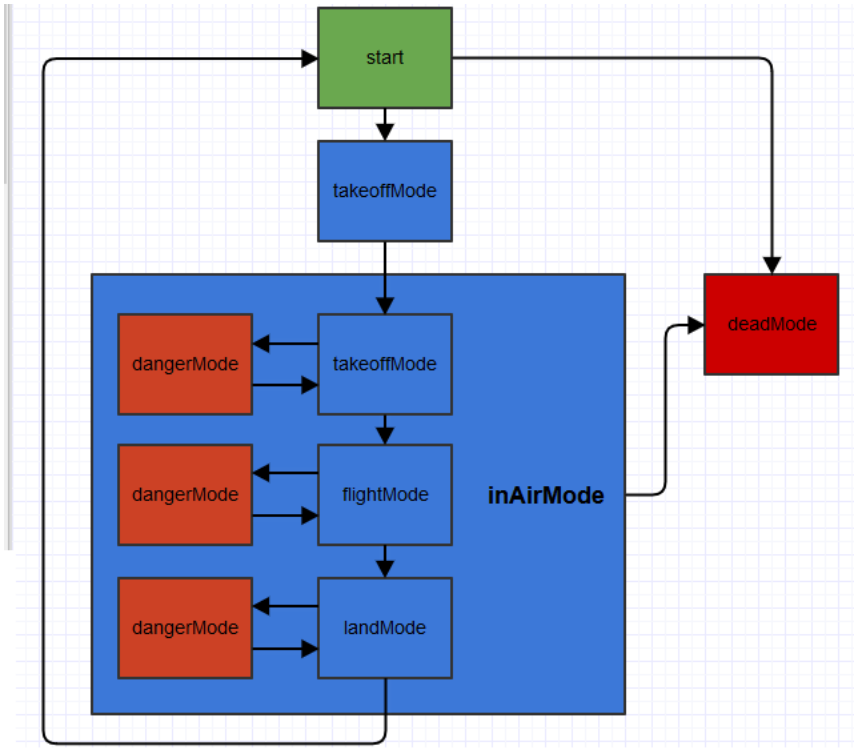


Figure 7.1: A flowchart of the different states the system can go through

Test 8

In this section, the test of the quadcopter's obstacle avoidance and controlled movement will be described. First the purpose of the test, including the reason we have performed it. Then the setup and execution of the test, and finally a conclusion based on the results of the test.

8.1 Purpose

The purpose of this test is to see if, and how well, the quadcopter is able to avoid obstacles and if the quadcopter can do controlled movement. Avoiding obstacles is essential for the quadcopter if it is to move around, and not collide with its surroundings. At the same time getting the quadcopter in the air, and the PID for holding the quadcopter at a steady altitude when in the air, will be tested, to find the values which makes it as stable as possible.

The quadcopter should be able to detect and react to obstacles in its way when it is flying. To do this it must use its sensors, the HC-SR04, to detect surroundings, and be able to receive movement commands through Bluetooth, and communicate between the Arduino to the flight controller to react to the obstacle. The purpose of this test is to test if the flight controller receives and reacts correctly to the commands given to it, both the automatically and user based commands from the computer to the Arduino.

8.2 Setup and execution

The testing of the obstacle avoidance and controlled movement was split up into three tests. The first with the quadcopter on the ground, motors off, and a PC connected to the flight controller to monitor it on the receiver tab in the Baseflight application, see section 2.8.1, to make sure whether or not the flight controller reacted to obstacles moved into its different distance-states, by either adjusting roll or pitch if an obstacle is at the given dangerous distance, such that it moved in the opposite direction of the obstacle, setting throttle to zero if the obstacle came into critical-distance, and not being able to move in the direction of an obstacle, with commands from the Bluetooth, when at restricted-distance.

The test started by testing the arm, takeoff and flight-mode, to make sure the commands and modes were as expected, such that the quadcopter is only able to receive the command

to takeoff, when it got armed with the arm command, and when it got to the wished altitude, by being lifted off the ground, that it entered flight mode. This was done first, since it is required for the quadcopter to be in flight mode before the quadcopter can execute movement commands could be used and tested.

It was chosen that this were to be done first, with the quadcopter on the ground, to eliminate the chance of the quadcopter colliding with an obstacle, if the flight controller did not react to the commands to avoid the obstacle, and getting damaged.

In the second test the quadcopter was also placed on the ground, this time with the feet tied to the ground with a string, giving the quadcopter just enough free room to hover over the ground. This was done to check if the results from the first test would be the same, if the quadcopter actually was in the air. The quadcopter was connected to a PC with Bluetooth and then brought to hover over the ground, using the commands arm and takeoff, so the PID used for holding a specific height could be tested. Under the PID test, the stop command was also tested, to make sure it was possible to stop the motors. After this was done, a piece of cardboard was moved in front of each sensor, to test obstacle avoidance and the different distance-states; restricted, dangerous and critical, and to make sure that the results from the first test matches the actual reaction of the quadcopter, by moving in the opposite direction of the obstacle.

The third test was an upscale of the second test, without the quadcopter being tied to the ground. A PC was again connected to the quadcopter and used for getting the quadcopter flying, with the arm and takeoff commands, and for safety-stop, with the stop command. The quadcopter was then placed in the middle of a room, where 4 people stood around it with moveable blackboards for walls, to try and control the quadcopter's movement and test obstacle avoidance. While testing the obstacle avoidance, the ability of the quadcopter to maintain a specific altitude was also watched and adjusted, by changing the PID's values and other defines, like `DANGER_DISTANCE` or `FLIGHT_HEIGHT`.

Both test two and three were repeated several times, to adjust the different values and making sure the results were consistent.

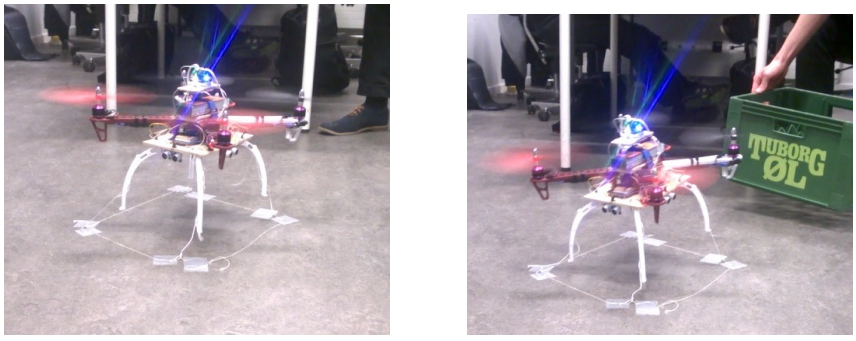
8.3 Results

The results from the first test, where the quadcopter was on the ground and lifted up to enter flight mode, showed that the commands for getting the quadcopter flying, with arm, takeoff and entering flight mode, worked as wished. Entering the arm command made the quadcopter go into arm mode, and making it possible to use the takeoff command, going into takeoff mode. In takeoff mode it could be seen on the receiver tab in the Baseflight application that the flight controller would increase the throttle to try and get the quadcopter off the ground. When the quadcopter then got lifted by hands off the ground to the wished altitude, it would enter flight mode successfully, and the quadcopter would be placed on the ground again. The throttle would still increase, since the quadcopter is placed on the ground and was not at the wished altitude, which was expected.

After being lifted to enter flight mode, and placed on the ground again, the different distance-states were tested, and showed that the quadcopter reacted as wished. When an obstacle was detected at the restricted-distance the quadcopter could not move in the direction of the obstacle, when an obstacle was at the dangerous-distance the quadcopter would try to move in the opposite direction, and the quadcopter would set the throttle to zero if an obstacle came into critical-distance. When the quadcopter was in flight-mode, the movement commands were tested, and was monitored on the receiver tab in the Baseflight application which showed that the flight controller gives the correct signals to the motors to move in the direction of the command. This all meant that the first test was successful, and we were ready to move to test two.

In the second test it was hard to test the PID for holding a specific altitude, since when the quadcopter was tied to the ground, the strings would restrict the quadcopter's movement, and hold it steady because of the quadcopter moving upwards and keeping the strings tight. This meant that there was not much visible movement in the altitude or any other directions, when the quadcopter was not given any commands. Therefore it could not, from this particular test, be concluded if the quadcopter had any drift in any directions.

The test continued with testing the obstacle avoidance and different distance-states an obstacle can be in. Since the first test showed that the flight controller actually signals the motors, to move in the opposite direction of an obstacle in order to avoid the obstacle in dangerous-distance, and that it could not move in directions where an obstacle was in restricted-distance of the quadcopter, the purpose of this test was more to make sure that it actually happened. The test was successful, and a situation with an obstacle moved into dangerous-distance of one of the sensors, the quadcopter can be seen moving away from the obstacle on figure 8.1. Both the situation with an obstacle in restricted- and critical-distance was a success, and the quadcopter would not move in a given direction, if an obstacle was in restriction-distance of the quadcopter in that direction, and the quadcopter would set the motors throttle to zero, if an obstacle got into critical-distance of any of the sensors, and then fall to the ground.



(a) The quadcopter without any obstacles near.

(b) The quadcopter trying to avoid the box.

Figure 8.1: Pictures taking under test two.

In the third test it was easier to test the PID for keeping the specific altitude, since the quadcopter was not tied to anything and had no restriction on movement besides the given altitude value and the different distance-states values. The first flight showed that the PID,

for altitude, had to be adjusted a little to keep the quadcopter more in balance, which resulted in the PID's D-value to be decreased, expecting this to help. The D-value was set too high, and the decreased D-value stopped the quadcopter from osculating around the given altitude. This problem was solved by lowering the D-value again, and slowly adjusting the I-value instead, which made the quadcopter, by eye measurement, seem to have better stabilization around the set altitude.

While flying the quadcopter and trying to adjust the PID, for keeping the altitude, obstacle avoidance was also tested. When the quadcopter took off and started flying, it would slowly move in a direction because of drift. The flight controller takes time to fully stabilize, and in this time the quadcopter drifted a lot. The blackboards were used to try and control its movement and test if it would avoid them and fly in the opposite direction, if one got into the given dangerous-distance from the quadcopter. When the quadcopter moved too close to a blackboard, such that it got in dangerous-state distance, it would, as seen in test one and two, avoid the obstacle by flying in the opposite direction, see figure 8.2, which concludes that the quadcopter is able to avoid obstacles, but another problem was discovered.

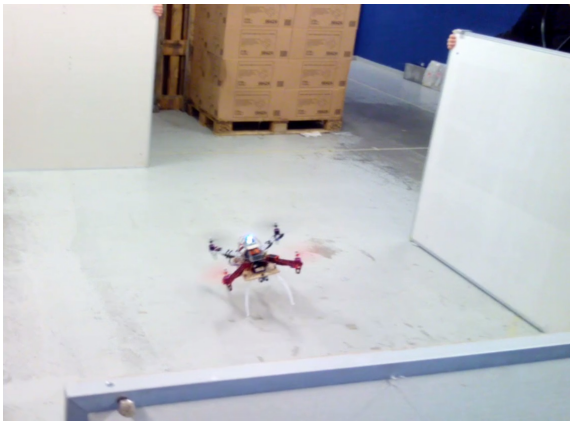


Figure 8.2: Picture of the quadcopter flying avoid of one of the blackboards

When the quadcopter avoided one obstacle and flew in the opposite direction, the momentum it got would make it keep flying in that particular direction for far to long, and sometimes getting too close to another obstacle shooting it back or sometimes even ignoring the second obstacle. To solve this problem, a counter-movement was added to the quadcopter for when it avoided an obstacle and it got out of the dangerous-state distance, see implementation 7. It would get a very short signal to fly in the direction of where the obstacle it avoided was. This helped the quadcopter from flying directly into another obstacle when it avoided the first, making avoiding obstacles more stable, but still not perfect.

There would still be a few times, where it had a hard time reacting to the second obstacle that it met, and would continue to fly towards it, even if it got into dangerous-state distance, where it should have avoided and flown in the opposite direction. It is believed this problem, with ignoring the second obstacle, might be influenced by the drift and high momentum, since the quadcopter should not continue to move in any direction, when it

has avoided the first obstacle, and comes out of the `dangerMode`. This problem is not particularly a part of this test or task, but it is a problem that needs to be fixed, and it can be fixed with position determination, which is the purpose of task three, by detecting drift and stopping it by opposite reaction.

Task conclusion 9

In this section task two will be concluded upon.

The PID maintaining the wanted altitude, has been tested to work, in environments with a high amount of drift, thus not tested in the final state, and might need tweaking of the different PID values.

The quadcopter can autonomously change between states based on sensors, and achieve takeoff and landing. As well as move away from obstacles, by communicating with the flight controller that then controls the motors. The system also reacts on inputs from an external source with Bluetooth, and the system can be commanded to move in directions, to arm, to initiate takeoff and landing, and to stop immediately. The movement might be restricted and thus not executed, if obstacles are within the restricted movement distance.

The sonar distance sensors does not work correctly when reading walls in angles, and might give faulty readings. Therefore until another solution is found, the system will not work well in rooms with angled walls. Therefore the four diagonal distance sensors pointing between the front-, back-, left- and right sensor, can give faulty data in a normal room.

One major cycle takes 13 ms, which means the software can react to a stop command within one major cycle. We have nine sensors, that needs three readings every time, which makes one sensor data cycle about 351 ms. If we take four sensors away, and leave one pointing downwards and four in perpendicular directions, then it would take 195 ms to get one trusted reading of the sensors. When entering a danger zone physically, it can take 351 ms or 195 ms for the quadcopter to discover the danger zone. By using only five sensors would hold us within the requirement for task two, that states we should react to walls within 200 ms. Though the deadzone of 50 ms when entering a critically close state cannot be accomplished with the sonar sensors, because the speed of sound is too slow when we need to account for interference.

Part IV

Task Three

Introduction 10

After having achieved flight in all directions, and automatic obstacle avoidance, it is time to proceed to task three. The purpose of task three is to implement movement by position, meaning the quadcopter is aware of how far it has moved in relation to any previous position, in order for task four to be integrated with the system. Knowing the quadcopter's physical movement, can help eliminating the quadcopter's drift and help making a controlled movement of a specific length.

10.1 Requirements

The requirements for task three will be defined in this section.

- The system should be able to determine the quadcopter's position in relation to the start-point, with a new position reading every 250 ms.

This is the requirement that has to be fulfilled before the quadcopter system can be integrated with task four. The new position readings are needed fairly often, to eliminate the quadcopter's physical drift. The precision of the position determination is directly proportional with the quality of the final product, and therefore it is important to make the precision determination as precise as possible.

Analysis 11

In this chapter, the methods for determining the quadcopter's position accurately will be determined, and one method will be chosen and expanded upon.

11.1 Position determination

In this section, the various means of determining the current position of the quadcopter will be described, and some will be chosen as candidates for implementation.

The position has to be determined, in order to integrate task four into the system. The purpose of task four is to explore and map a room for a building plan, and in order to do this accurately the position of the quadcopter is essential to know. In task two, it was discovered that the quadcopter has a drift in its position over time and the purpose of task three is therefore also to eliminate as much drift as possible, by sending counter reaction to actuators, when drift is detected.

Various technologies and options that can help achieve position determination can be the following:

- **Stereo cameras** where the quadcopter uses two cameras and some software to determine how far the quadcopter has moved, based on the depth perception achieved by using two cameras in conjunction.
- **GPS** Using a GPS tracker.
- **WiFi triangulation** where a WiFi module for the Arduino is used, and the difference in WiFi hotspots signal strengths is used to calculate an exact position based on known values of hotspot signal strengths.
- **IMU position determination** by using the data supplied by the IMU to exactly calculate how far the quadcopter has traveled.

Using a stereo camera setup to measure the current position, requires known points for referencing the distance traveled, and increases memory and computation requirements for the embedded system, as it has to keep track of these references and compare them to

each other. Using stereo cameras the quadcopter might also begin to drift, as it has no way of knowing if the position determination has drifted, if the system is not able to keep track of known points for referencing. GPS however has no risk of drift, but has precision issues. The distance accuracy needs to be within a safety margin of a couple centimeters, which GPS is not capable of, using relatively cheap hardware. WiFi triangulation would require placement of WiFi before using the system to map a building plan, and might be very inaccurate if there are walls or other interfering objects in the way. This severely limits the areas that may be explored and mapped, we would like our quadcopter to operate in a lot of different situations, so WiFi is not an option. The IMU is a hardware module on the quadcopter, and can therefore be used in all situations. The IMU position determination has no way of knowing if it drifted, and might drift a lot.

Based on these facts we choose to use the IMU for position determination, because the stereo camera will use a lot of memory and/or computational power. The GPS is too inaccurate, and the WiFi limits the situation severely. There might be better ways to determine position, with a combination of IMU and GPS, because GPS cannot drift, and the IMU can. But due to the fact that the GPS is so inaccurate that the data we would get from it would be practically useless, we have chosen not to use it.

11.2 Position determination using IMU

In this section, it will be determined how the quadcopter's position can be found using an IMU.

The IMU supplies acceleration data using an accelerometer, and using this data can in theory directly be computed into a change in position over time by integrating the acceleration data into velocity and then into distance.[43]

As described in task one, the data supplied by an IMU includes a lot of noise, and therefore sensor fusion has to be implemented to reduce the noise supplied by the IMU. The sensor fusion noise reduction is extremely important in the case of position determination, as integrating the acceleration data will result in quadratic growth of position error, due to the double integration of the acceleration data that may contain drift errors.[44]

A problem with the IMU's accelerometer is also that it measures gravity, and in this case we do not want gravity as a factor, but movement. The essential part to accurate position determination is down to having an accurate measurement of the orientation of the IMU. The orientation of the IMU is achieved by subtracting the gravitational acceleration from the total acceleration measurement, and combining this with high frequency angular rate supplied by a gyroscope using a sensor fusion algorithm.

It is also possible to combine magnetometer data through sensor fusion, but only in an environment which is free of magnetic anomalies, which is not the case with the system, as the motors and battery contain magnetic anomalies close to the frame of the quadcopter, where the IMU has to be mounted as part of the system, and because magnetic interference is often present indoors in lights, wires and more.[44, p. 3]

The sensor fusion of accelerometer and gyrosope, however, assumes that the data supplied by the gyroscope is extremely accurate, in order to accurately assume how much of the acceleration data is actually correspondent to the quadcopter's movement. Even a small error in the angle supplied by the gyroscope, will slowly accumulate noise over time and offset the position by an extreme amount, where an error of 1 degree accumulates to a position error range of 17 meters after 10 seconds.[45, p. 6]

To stop slowly accumulated noise, a high-pass filter can be applied, but this filter will make all other movement less significant, and make the position determination a little less accurate. So the IMU will be tested both with and without a filter made to filter out accumulated speed.

Test 12

12.1 Test precision

The test and the implementation of the tested software will be described in this section. The tests include a test without a drift filter, where all the accelerometer data minus the gravity is used to determine position, and also to test position determination with a high-pass filter, to eliminate drift, based on the velocity (the first integral of the raw accelerometer data).

12.1.1 Software

The software is developed in Arduino, so that it can be ported to the quadcopter if the methods are proven to be useful. Some of the software is reused code, from the sensor fusion section 2.3 and PID section 2.6. The communication with the IMU and the sensor fusion to determine the quadcopter's angular alignment with gravity, is mainly the code that has been reused.

The theory behind the position determination using the IMU, is that we know the IMU's angular alignment with gravity, and we know all the accelerations that affects the IMU. By calculating the effect gravity should have on the IMU (by using the angular alignment with gravity), the gravity can be subtracted from the raw accelerometer data, to achieve the accelerations affecting the IMU without the accelerations generated by gravity. After calculating the accelerations, we need to recalculate it to speed and then position, with the use of integral. We make the integral by accumulating every reading into speed, each time the accelerations are read, and then dividing both steps with the delta-time from last to current reading.

12.1.2 High-pass filter

The high-pass filter is adjusted with the use of a processing program that maps inputs in live graphs. The Arduino is connected to a computer via USB, and a live graph of 6 inputs are graphed at the same time, where acceleration, speed and position is used while adjusting and trying out new weights on the high-pass filter. The high-pass filter is applied to the speed, because the speed has a big risk of drifting, if the IMU's accelerometer data is a bit off. A drift on the speed, will be accumulated every reading to the position, where a little drift on the position will not be accumulated, but will only be a direct drift. The

accelerometer data cannot drift, but inaccuracy of the accelerometer can result in a faulty speed, and if the speed is faulty, the fault margin is accumulated in the position, and therefore a filter to prevent drift is placed on the speed.

The tested filter always made the speed variable go towards 0, this filter would eliminate a drift, but it would also make all data a bit faulty, with the amount that the speed is filtered towards 0 per second. If the IMU was only moved a little and then stopped, it would limit the amount of damage this can cause. And it should in practice be able to eliminate a lot of drift. The filter is applied accordingly: $Speed+ = (acceleration * dt * 0.98) + (0.02 * (0 - Speed))$ this will take 98% of the acceleration and apply it to the speed, and then 2% of the speed will be the speed negated. This way, if no acceleration is applied, the speed will go towards 0 with 2% every reading.

Another filter tested was a filter that only took the speed variable towards 0, when the IMU detected that there was no or little movement. This way the filter should not make faulty data, when the quadcopter is accelerating. Though if the quadcopter is at constant speed, the IMU will think the quadcopter is still, and the filter would be applied. The filter is the same as in the other test, but it is only applied at constant speed (as well during no movement).

12.1.3 Test and result

The test consists of moving the IMU 40 cm in one direction, and then move it back to the starting point, then stay at this point for about 5 seconds before repeating once more, so that the distance traveled is 160 cm. The end point for this process should be 0. This whole process is mapped in a graph, so it can be seen if and where the drift appears. This test was repeated 4 times, because this gave a clear picture of whether or not the IMU could be used.

The test results can be found in appendix C.1.

Test result without filter

The results as last position were; 5.7 m, -0.45 m, -10 m and 2,8 m. Two of the test results had a lot of drift between the movements, as seen on figure 12.1. The other two also drifted, it starts to drift a lot after the second movement, as seen on figure 12.2.

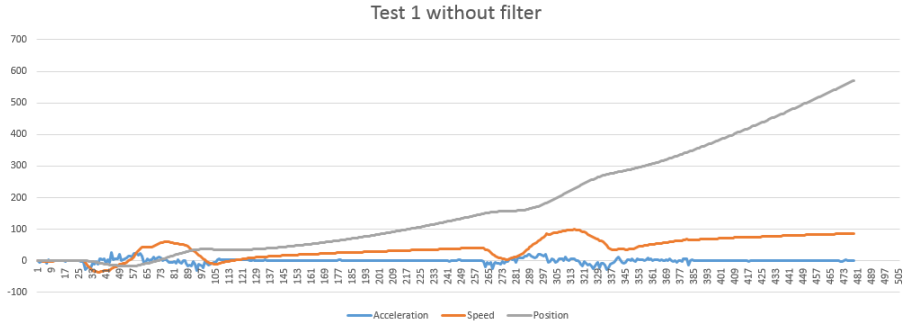


Figure 12.1: Test result of moving the IMU forward and back to the same starting point without filter

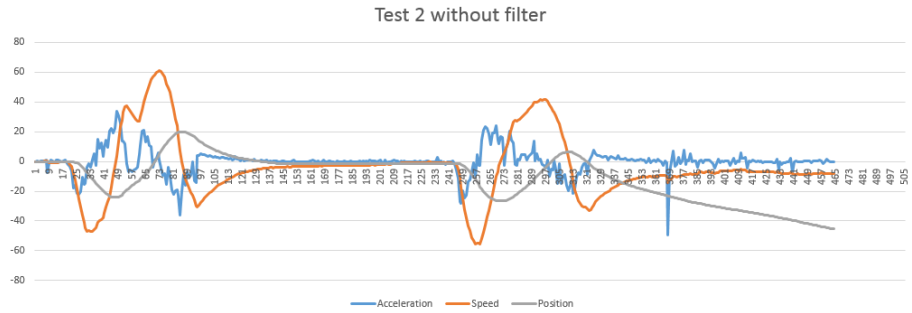


Figure 12.2: Test result of moving the IMU forward and back to the same starting point without filter

Using the IMU to determine movement, without filter, is not usable, as the drifts on a 7-8 second long test is between -10 m and to 5.7 m. Looking at the graphs the position is drifting a lot, and the drift is not going down its only getting bigger, and there is no sign that the speed would go towards the correct speed, thus the drift will keep growing.

Test result with filter

The results as last position were, with a high-pass filter: -19 cm, 171 cm, -71 cm and -51 cm. This is significantly lower than the results from without the filter, but still not good enough for use. The speed is also going towards still, and then the position does stop drifting, but there is still a lot of drift, as seen on figure 12.3. Furthermore, the IMU does not detect the movement correct, the graph shows the position reaches 20 cm in the correct direction and then moves back to the start point. This is recorded after a 40 cm movement in one direction. The graph shows that the speed is both negative and positive, while the IMU has only been moved in one direction. The same can be observed on the way back, the position moved 20 cm and then back to 0, while the movement never crosses the starting point, and the graph should never cross the marked x-axis.

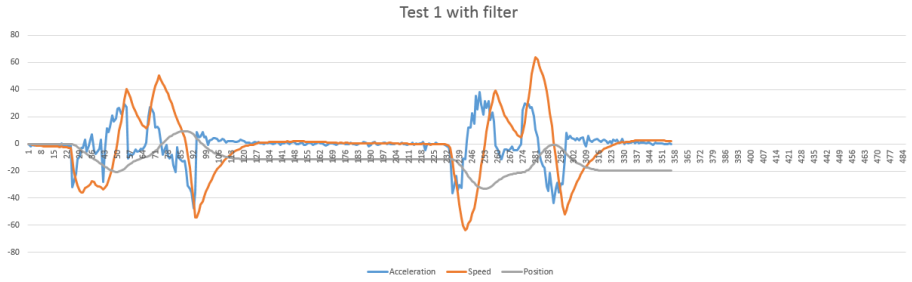


Figure 12.3: Test result of moving the IMU forward and back to same starting point without filter

Based on this test, we came to the conclusion that we would not be able to determine a precise enough location of the quadcopter, in order for task four to be integrated with the rest of the system. For this to happen, a better solution would be needed, perhaps using some hardware that we currently do not possess.

Task conclusion 13

Since the most reasonable option to determine the position of the quadcopter, was the IMU position determination, and because no other position determination reference can be tested without new hardware, task three is not within our ability to implement within project timeline at this time. The position determination, and therefore also the drift elimination, cannot be completed now, and will be the most needed future work. Without the drift elimination and position determination, task 4 cannot be tested with the quadcopter itself, and a manual position determination and/or simulation test would be needed.

There can be a lot of reasons why the IMU has a lot of drift; The data the IMU provides can be inaccurate, the delta-time used from the clock in the Arduino can be inaccurate, the sensor fusion can be inaccurate because it takes 2 faulty data and tries to make the best of it, the recalculation from bits to data, with scaling, can be slightly off. And lastly gravity is at 9.82 meters per second squared, which is a pretty big acceleration compared to the accelerations we want to detect, and therefore if only a small amount of this gravitational acceleration is applied as movement, it would change the speed and position a lot.

Part V

Task Four

Introduction 14

With the completion of the first two tasks and making the delimitation of task three's implementation, task four will be started. In this chapter it is assumed that the position and orientation of the quadcopter is known. The focus point in task 4 will be automatic exploration and mapping of a room/building. This means that this chapter will focus on creating a system which will allow us to draw a 2-dimensional building plan of any given building, using the quadcopter and its sensors. To make this possible, an algorithm will be explored and created that will give the quadcopter a form of intelligence. The solution to this task should be able to issue commands to the quadcopter about where to go, receive sensor and position information from the quadcopter, which it will use to draw a building plan. An example of what a building plan like this could look like, can be seen in figure 14.1, here the green grid cells represent areas that are free, and the red grid cells represent obstacles.

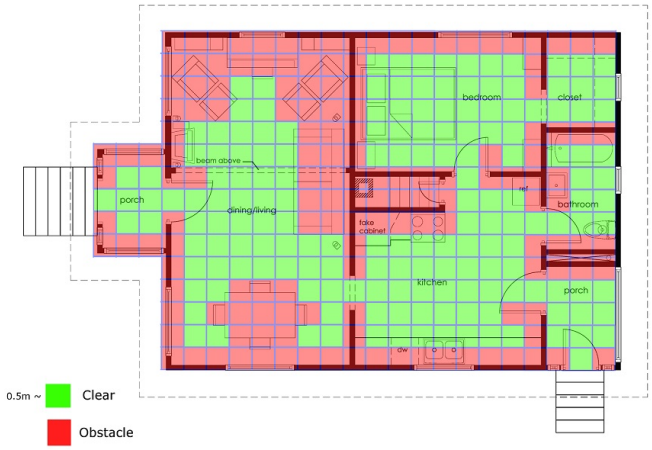


Figure 14.1: An example of what a drawn building plan could look like.

An idea of what the quadcopter movement and exploration could look like in the end, can be seen in figure 14.2.

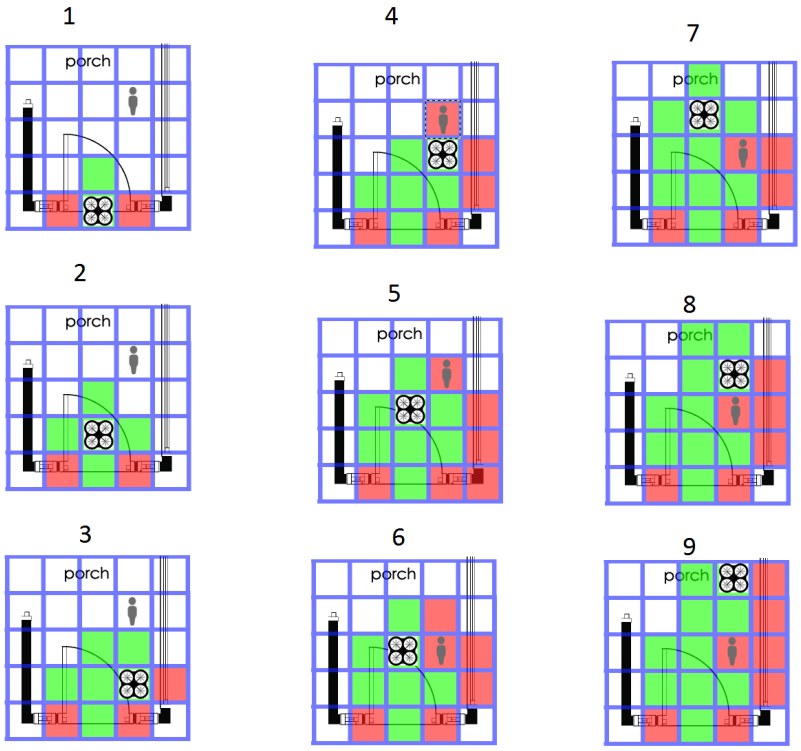


Figure 14.2: An example of step by step movement, and exploration

14.1 Requirements

In order to complete this task a number of requirements must be fulfilled:

- The system must be able to give a visual representation of a mapped building in form of a building plan.
- The system must be able to navigate the quadcopter through a room while avoiding obstacles.
- The system must be able to detect obstacles and add them to the building-plan.
- The system must be able to send and receive data to and from the quadcopter.

Analysis 15

In this chapter we will analyse task four and some of the possibilities which can be used to complete the task.

15.1 State space

In order to make this part of the system, it is important to first analyse the problem domain and what is relevant for the quadcopter to know. For this the state-space problem will be defined. This consists of, a set of states (which represents the relevant area of information needed), start states (which represents the state or states the quadcopter can start in), a set of actions (factors which can change the states), and the goal states (which will be the state which we are in when the quadcopter's objective has been completed).

State-space problem:

This will only give an estimate of how the state space problem will look like for the final system, since this can depend a lot on the path-finding algorithm used, how we choose to draw the building plan, and the fact that changes are sure to come up at a later time.

First of all a decision has to be made about how a room will be represented. We have chosen to represent it as a grid. Which is ideal to work with in a 2D space, if performance can be an issue.

Set of states

The set of states is defined by an infinite, but countable, number of positions in a grid and the number of features each grid cell has, as well as the domain of these features.

Start states

The start state, would have the quadcopter at a starting position in the grid (0,0), with a grid size of one, and the same cell would be assigned as free.

Set of actions

The quadcopter can move left, right, forward, backward. Grid pieces can be added to the

grid, and assigned as free or blocked.

Goal states

The goal state would be when an area has been explored and no unexplored grid pieces can be reached (meaning all added grid pieces will be either free or blocked, and the grid should be surrounded with blocked grid pieces).

This is an early assessment of the problem, and the actual state space is likely to change later.

15.2 Grid size

If the building plan is gonna be represented by a grid, it is important that each grid cell has an appropriate size. The smaller each grid cell is the more precise a room can be described, however the size of grid cells will also have an impact on the memory usage of the application, and on the speed at which a search algorithm can go through a room. This is the case, because the smaller each cell is, the more cells are needed to describe a room. This means that we will have to decide on a size which is small enough to represent a room at a reasonable precision, while not compromising the speed and memory usage of the application too much.

This means that we will have to decide on a size based on the narrowest environments that the quadcopter is supposed to be able to move through. The quadcopter itself is 60×60 cm, and to have some kind of safety margin we would not want it to move through areas that are less than a meter wide, which will give the quadcopter 20 cm space on each side. Some doors fit this size, but unfortunately a lot of doors are a bit more narrow than this. If a one meter space is split into five cells of 20×20 cm each, then the safety distance from an obstacle, in a situation with a one meter space, will be at best 20 cm and at worst 10 cm. The best case scenario is shown on figure 15.1, and the worst case can be seen in figure 15.2.

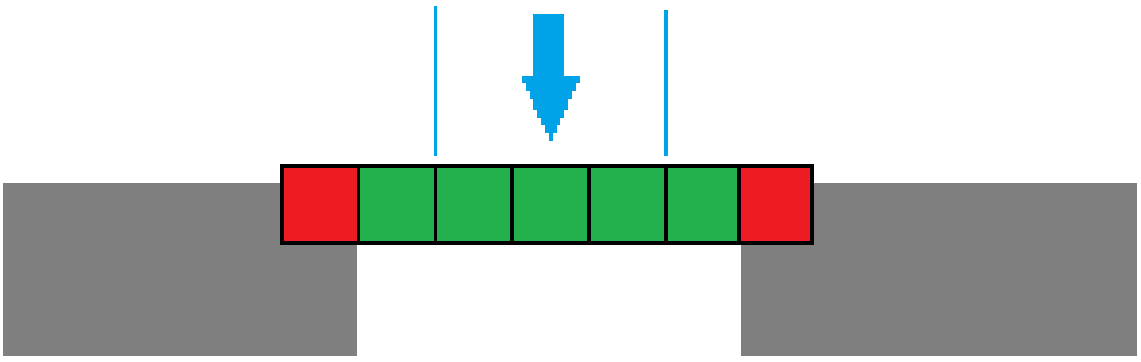


Figure 15.1: Best case scenario, of moving through a one meter space.

Each grid cell is 20×20 cm, the red ones are classified as obstacles and the green ones are free. The blue arrow is where the center of the quadcopter would move through the

space, and the blue lines represent the size of the quadcopter.

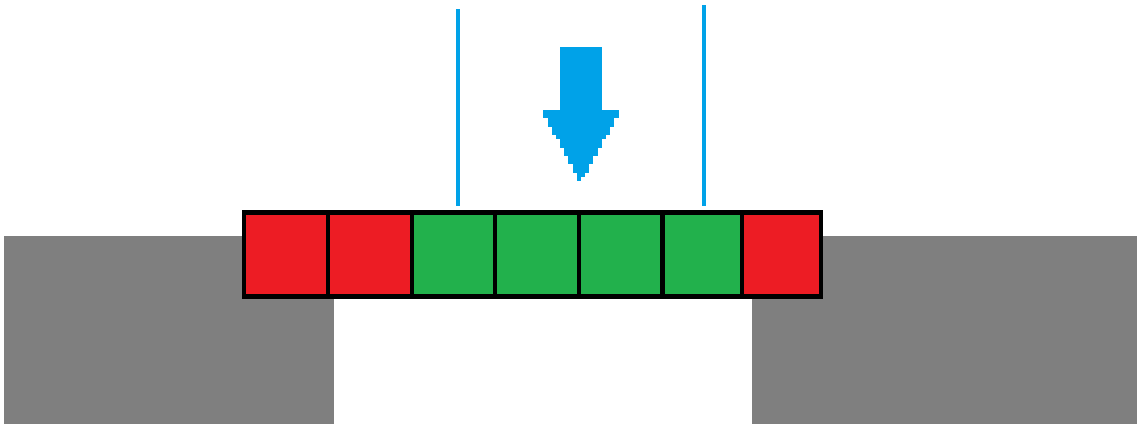


Figure 15.2: Worst case scenario, of moving through a one meter space.

15.3 Search algorithms

One of the things that the system needs is a way to do path-finding for the quadcopter. For this purpose we have examined two search algorithms. The A* algorithm and Lee's algorithm. These algorithms require a start position, and will search for a path to some goal position.

A* Algorithm

The A* algorithm is the most popular choice for pathfinding. The A* algorithm can be used to find a shortest path between multiple points called nodes, using an heuristic estimate of the cost to reach the goal[46, 47]. A* uses a combination of Dijkstra's algorithm and best-first search, expanding on the most promising node based on a rule, which can be lowest cost to goal. Using an heuristic technique that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first, is called a greedy best-first search, however A* differs from greedy best-first by taking the distance already traveled into account. Efficient selection of the current best candidate for path extension is done by implementation of a priority queue ordering the nodes by a cost function.[47, 48, 49]

The A* cost function $f(n) = g(n) + h(n)$. $g(n)$ is the known cost of getting from the start node to node n, and is tracked by the algorithm. $h(n)$ is an heuristic estimate of the cost to get from node n to any goal node, and must be provided by the user of the algorithm, as it is problem specific.

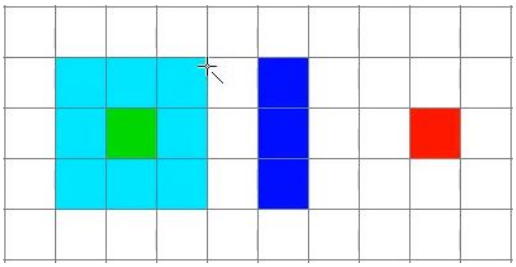
Following is a pseudo-code example[48] of the A* algorithm:

```
1 function A*(start,goal)
2     ClosedSet := {}        // The set of nodes already evaluated.
3     OpenSet := {start}     // The set of tentative nodes to be evaluated, initially
          containing the start node
4     Parent := the empty map    // The map of navigated nodes.
5
```

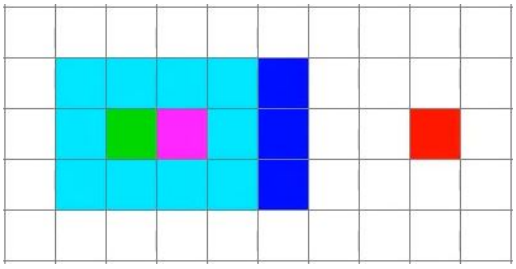
```
    g_score := map with default value of Infinity
    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal.
    f_score := map with default value of Infinity
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

    while OpenSet is not empty
        current := the node in OpenSet having the lowest f_score[] value
        if current = goal
            return reconstruct_path(Parent, goal)

        OpenSet.Remove(current)
        ClosedSet.Add(current)
        for each neighbor of current
            if neighbor in ClosedSet
                continue      // Ignore the neighbor which is already evaluated.
            tentative_g_score := g_score[current] + dist_between(current,neighbor)
                // length of this path.
            if neighbor not in OpenSet // Discover a new node
                OpenSet.Add(neighbor)
            else if tentative_g_score >= g_score[neighbor]
                continue      // This is not a better path.

            // This path is the best until now. Record it!
            Parent[neighbor] := current
            g_score[neighbor] := tentative_g_score
            f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(
                neighbor, goal)

    return failure

function reconstruct_path(Parent,current)
    total_path := [current]
    while current in Parent.Keys:
        current := Parent[current]
        total_path.append(current)
    return total_path
```

Example code 15.1: A* pseudo-code walkthrough

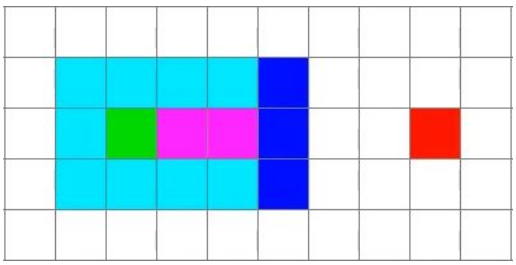
See figures 15.3a through 15.3h for a visual representation of the algorithm. Teal squares represent the open set, blue squares represent walls, purple squares represent the closed set, and the green square represents start while the red square represents the goal. Black squares represent the chosen path. In this representation, it is possible to move up, down, left and right for a cost of 10, as well as diagonally for a cost of 14.[50]



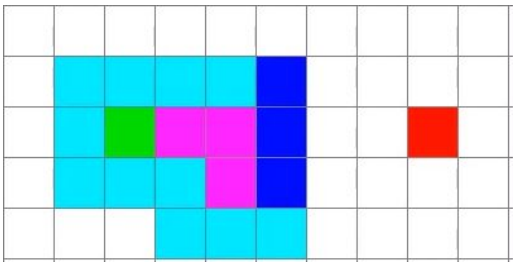
(a) Initial start



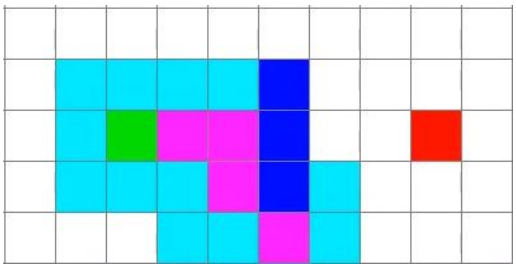
(b) Node from open set selected based on heuristic, and cost to move to start from this node determined



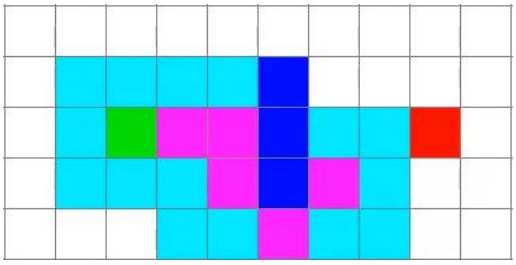
(c) New node based on heuristics, cost to move to start determined



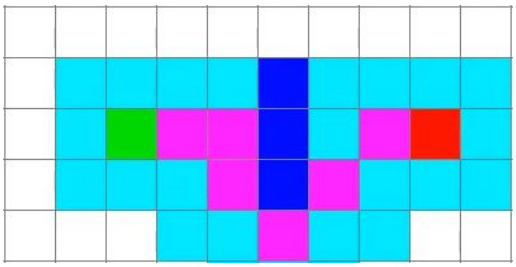
(d) wall nodes discovered, new node selected based on heuristic to move around wall, cost to move to start determined



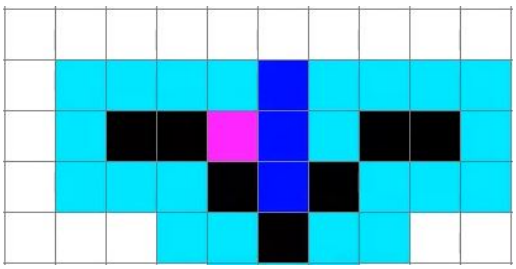
(e) New node based on heuristics to move around wall, cost to move to start determined



(f) New node based on heuristics, cost to move to start determined



(g) New node based on heuristics, goal added to open set, cost to move to start determined



(h) Backtrack nodes in closed set, based on cost to move to start node, lowest cost path determined

Figure 15.3: A* algorithm visualization

Lee's Algorithm

Lee's algorithm is a possible solution for a maze routing problem, which is a connection routing method that represents the routing space as a grid, where parts of the grid is blocked. The algorithm uses a breadth-first search approach.[51]

The algorithm is made up of 4 steps; initialization, wave expansion, backtrace and

clearence. First the routing layer is represented using a grid, with a starting point, then a breadth-first search from source to goal is performed, where each grid entry is labeled with the distance from the starting point, until the goal is found. When the goal is found, any path with a decreasing distance is chosen. After backtracing, all marks that were set during wave expansion are removed to allow for a new search.[51, 52]

1) Initialization:

```
Select start point, mark with 0
i := 0
```

2) Wave expansion

```
REPEAT
    Mark all unlabeled neighbors of points marked with i with i+1
    i := i+1
UNTIL ((target reached) or (no points can be marked))
```

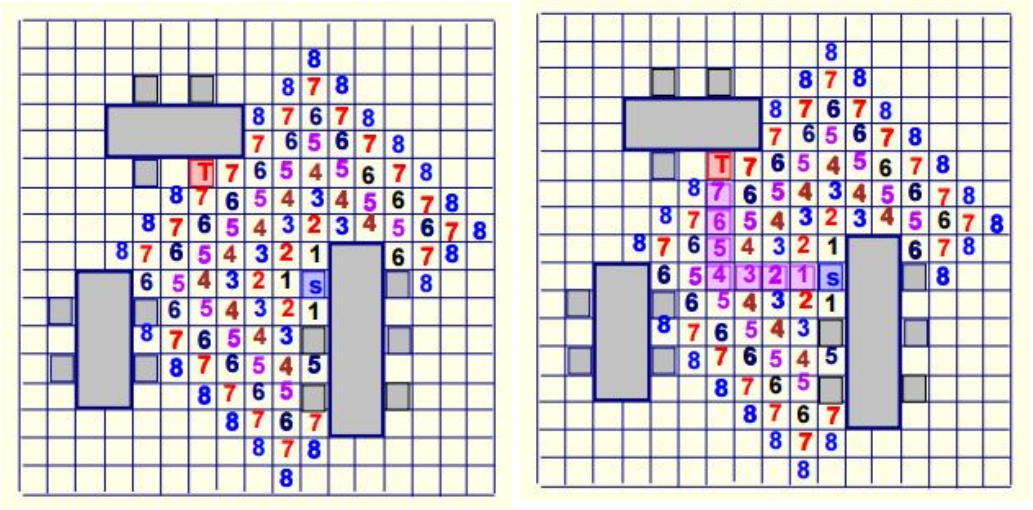
3) Backtrace

```
go to the target point
REPEAT
   go to next node that has a lower mark than the actual node
   add this node to path
UNTIL (start point reached)
```

4) Clearance

```
Delete all marks
```

A visual representation of Lee's algorithm can be seen on figures 15.4a and 15.4b.[53]



(a) Wave expansion until target is marked

(b) Backtracing from target to nodes with lower mark

Figure 15.4: Lees algorithm visualization

15.3.1 Choosing an algorithm

For general intents and purposes, the A* algorithm would be the choice at hand, but that is only if a consistent heuristic estimate can be achieved. For this project, where the purpose is to explore without any prior knowledge of the surroundings, such an heuristic estimate can not be achieved and therefore it will not be consistent. Lee's algorithm suits this project better, as it incorporates everything that is required of a search algorithm in this project, path finding and blocked obstacles for mapping, without requiring an heuristic as with A*.

Processing power

In order to complete the mapping of a room as well as doing pathfinding within a reasonable timeframe, an external processing unit is probably needed. Mapping a room, using Lee's algorithm, requires a lot of memory proportional to the size of the room. To ensure a reasonable timeframe, the mapping will be outsourced to this external processing unit, where we will use the Bluetooth module which was previously installed, to send information back and forth, between the external processing unit and the Arduino.

Design 16

The system that is to be designed needs to be able to communicate with the quadcopter (Arduino), giving commands and receiving data from the quadcopter's sensors. The system also needs to use the data received from the sensors to draw/visualize a 2-dimensional building plan representing the real world. All this has to happen autonomously, meaning that the system needs to be able to use the building plan to find its own way through a building while drawing it.

16.1 Component design

A PC has been added as a new component in this system, which will be used as an external processing unit to outsource the mapping and pathfinding of the quadcopter. The Bluetooth module which was added in task two will be used for sending and receiving information between the Arduino and the PC. This new addition can be seen in figure 16.1.

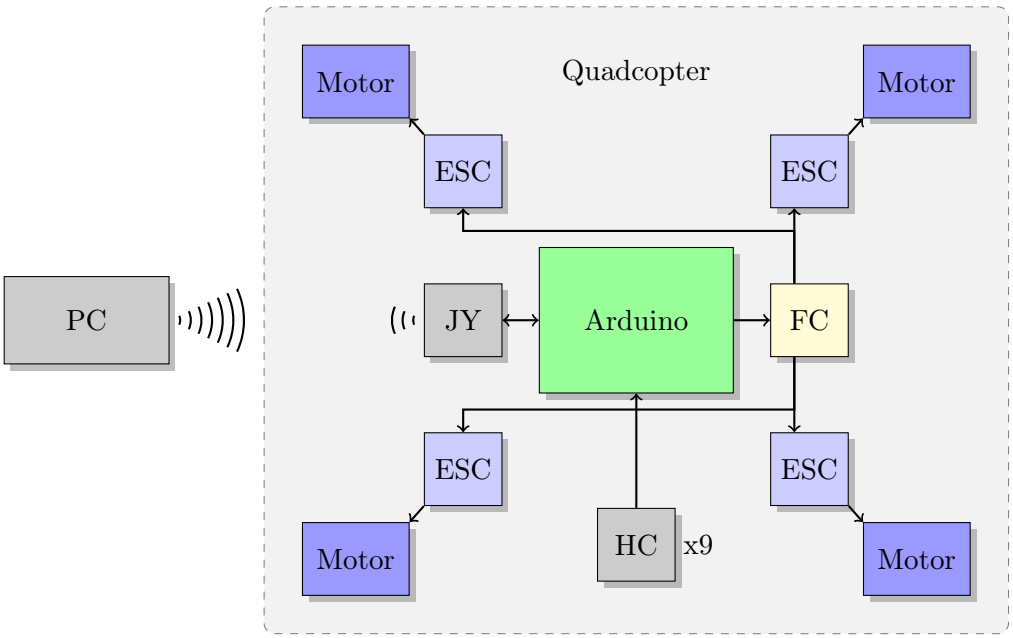


Figure 16.1: Diagram over the quadcopter representation and PC

16.2 Code design

This part of the system will be developed using C# in Visual Studio. For the GUI, WPF will be utilized. Elements of WPF will also be used for drawing the building plan. The building plan will be represented as a grid. For debugging purposes it would be ideal if the building plan can be created in real-time, since it would allow us to see if something goes wrong. Redrawing a bitmap in real-time would be very inefficient, and therefore we will instead have a fixed grid created with squares in the GUI. By not redrawing the squares every time something happens and instead just change the color/information on these squares, a lot of processing power can be spared. However this will also mean that the visible size of a map will be limited to this fixed size. Instead of being able to see the entire drawing at the same time, depending on its size, it will be made possible to move around the drawing using the mouse. A feature could also be added to save the building plan as an image in the end. In order to not slow down the main part of the system, the drawing of a building plan to the GUI will run on its own thread limiting the effect it will have on the main program.

A number of classes are needed to represent the rest of the system:

Quadcopter To keep track of the quadcopter's position, its sensors and path.

FloorCell To represent each cell of the grid (each 20×20 cm part of a room).

Room To keep track of all the information related to the building-plan.

Sensor To represent each individual sensor.

Using the `FloorCell` class, the system needs to be able to represent a room with obstacles and free spaces. For this a variable called probability will be used, this variable will be in the range of 0-99, the closer it is to 99 the more the system will believe that it is a free space, and the closer it is to 0 the more the system will believe it is an obstacle. To draw this the system will convert this number to a color, where the color will go from red(0) to yellow(50) to green(99).

A preliminary class diagram can be seen in figure 16.2.

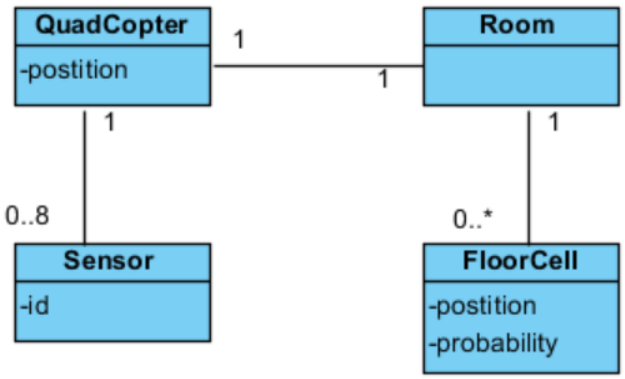


Figure 16.2: Preliminary class diagram

Implementation and test 17

In this chapter the implementation of task four will be described. The implementation itself happened in four steps:

Simulation In this first step, the system responsible for giving the quadcopter commands and drawing a building plan was developed, simulating the input from the quadcopter and its sensors. This method was chosen in order to be able to do tests on the system, without putting the quadcopter in any danger. This also allowed us to fine-tune some variables in order to get the best possible result, by doing thousands of simulations.

Physical test In the second step a representation of the quadcopter was made with the sensors attached, at approximately the same location to as they will be placed on the quadcopter. The method used in step one was then applied to the representation, with the input coming from the real life representation into the system. See section 17.2 for further information on how this test was done and the results of it.

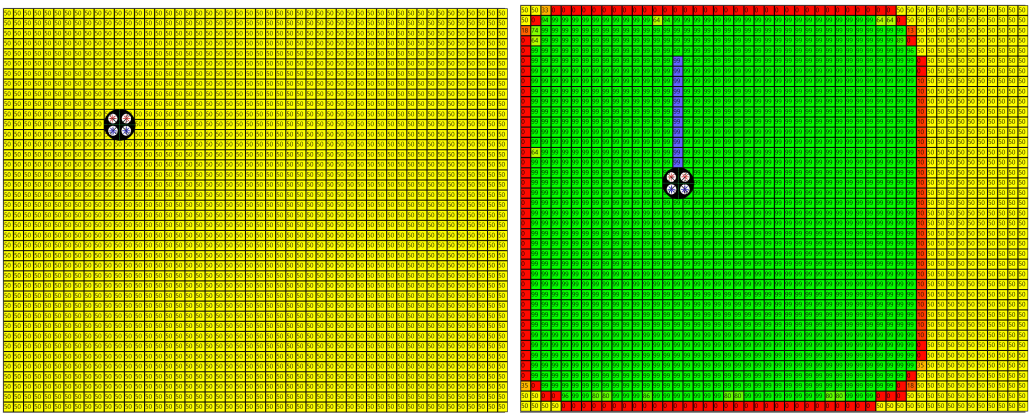
Updated simulation After the first physical test, it was discovered that the simulation software would need to be updated to account for a critical problem found in the first physical test, which was caused by the sensors being very inaccurate in some circumstances.

Physical test 2 After revising the simulation software it was time to do another physical test, which can be found in section 17.4.

The class diagram of the final system for task 4 can be seen in appendix D.4.

17.1 Step one: simulation

In this section the system used for simulating the drawing of a room using the quadcopter's sensors will be explained. The use of the word room will usually refer to the quadcopter's area of operation in this section. On figure 17.1a an image of how the grid representing a room looks in the final system before drawing anything, the figure 17.1b shows how the system represents a fully drawn room after completion.



(a) The grid (b) Representation of a room

Figure 17.1: Grid representations

Overview

The classes in the system are listed with a small description below.

MainWindow Is responsible for all drawing and all user actions in the system. This class also contains an instance of the `Room` class, used to save and access information about a building.

Room Is responsible for keeping track of all information the quadcopter needs to map an environment. The class contains an instance of the `QuadCopter` class and a list of the class `FloorCell`, where information about the room is stored.

FloorCell Each instance of this class represents a single grid cell in a room, meaning a 20×20 cm area in a room.

QuadCopter Keeps track of all information about the quadcopter. This class also contains an instance of the class `QCPath`, and a list of the class `Sensor`.

QCPath The `QCPath` class represents a path for the quadcopter, and contains methods for finding a new path.

Sensor Each instance of the `Sensor` class represents a single sensor on the quadcopter.

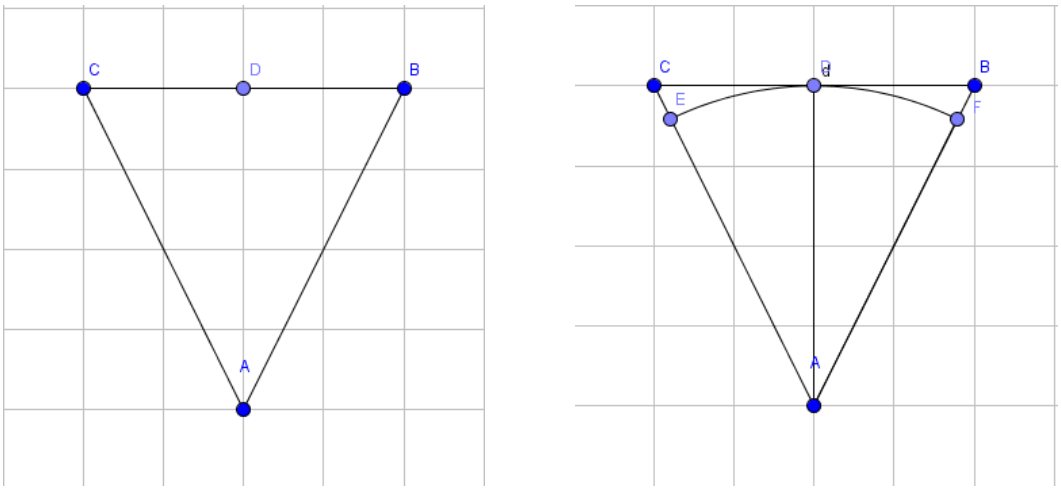
Simulation Is temporarily instantiated by a user using a button event from the `MainWindow` class. This method is used for running a simulation of drawing a room.

PointCalc Is a math class created for doing some basic calculations all around the system.

Sensor class

The `Sensor` class represents each of the eight sensors, used for mapping a room, on the quadcopter. Each sensor has a position, four points in the `Point[] viewPosition` which are used to store the positions for the minimum and maximum angle the sensors can detect. Index 0 and 1 represent the maximum angle while 2 and 3 represent the minimum angle. By using either the maximum or the minimum along with the sensors position,

a triangle can be made which is the view of each sensor. However a triangle is not an optimal way of representing the line of sight of a sensor. A visualisation of this can be seen on figure 17.2, and on figure 17.2a it can be seen that the distance between point A and point C or B, is longer than the distance between A and D. Given that the sensor has a maximum distance it is able to detect objects, this could create problems. To fix this the system checks if an object is within the triangle in figure 17.2a and also checks if the distance to any found object within the triangle is less than some specific distance, which we know is less than or equal to the distance between point A and D. This gives us a line of sight similar to what is seen on figure 17.2b.

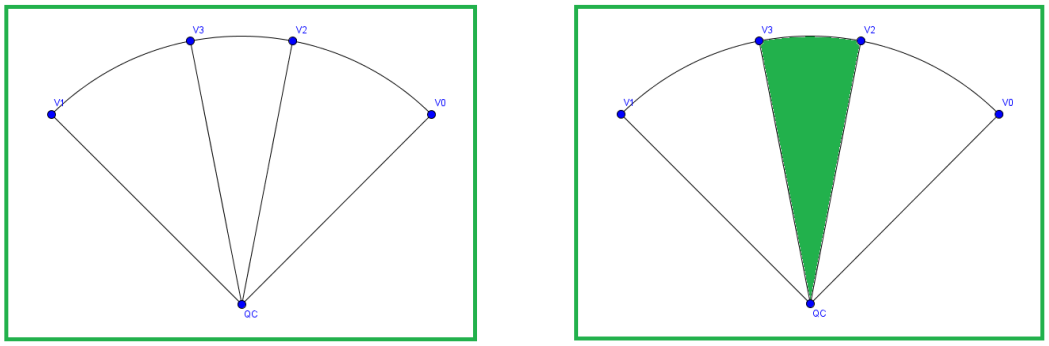


(a) Imprecise line of sight (b) Better line of sight

Figure 17.2: Sensor line of sight

The reason why both a maximum and minimum sensor view angle are needed, is that the system for all intents and purposes should try to avoid adding to `probability` of an obstacle, meaning that it should not mistake an obstacle for being a free cell. Both angles have been found during the sensor test in section 5.3, by looking at the smallest and biggest angles where thee sensors were able to detect an object. Since only eight sensors are needed for drawing a room, two of them can be removed and one, number 7, is already used in task two for measuring heights. We remove sensor 8 and 9 and only account for data between a distance of 50 cm to 150 cm because the quadcopter's sensors should not come within 50 cm of an obstacle, and the sensors have been limited to only account for objects detected at distances less than or equal to 150 cm. This leaves us with a minimum angle of 22 and a maximum of 90. This minimum angle represents an angle at which all of the sensors should detect any obstacle, while the maximum is the angle at which at least one of the sensors can detect obstacles. This is relevant because if we do not detect any obstacles then we can not conclude that no obstacles are within the maximum angle, since it is critical that no area with an obstacle is seen as a free area, only the area which the minimum angle represents will add to `probability`, this can be seen in figure 17.3. In the next three figures the left "situation" picture is what the sensor is actually looking at, the white part is free space, the red squares are obstacles. The "result" pictures on the right are what the sensor interprets the situation as, where green

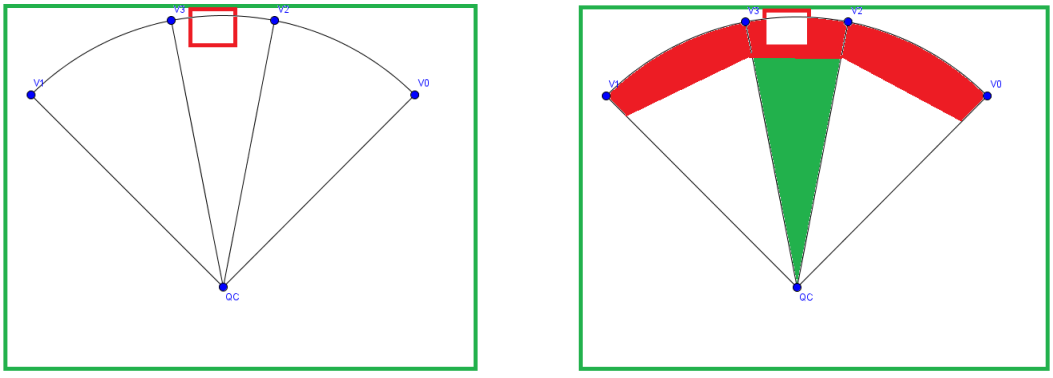
areas add to `probability` and the red areas plus the white squares, where the obstacle is, subtracts from `probability`.



(a) Situation (b) Result

Figure 17.3: Sensor behaviour with no obstacles detected

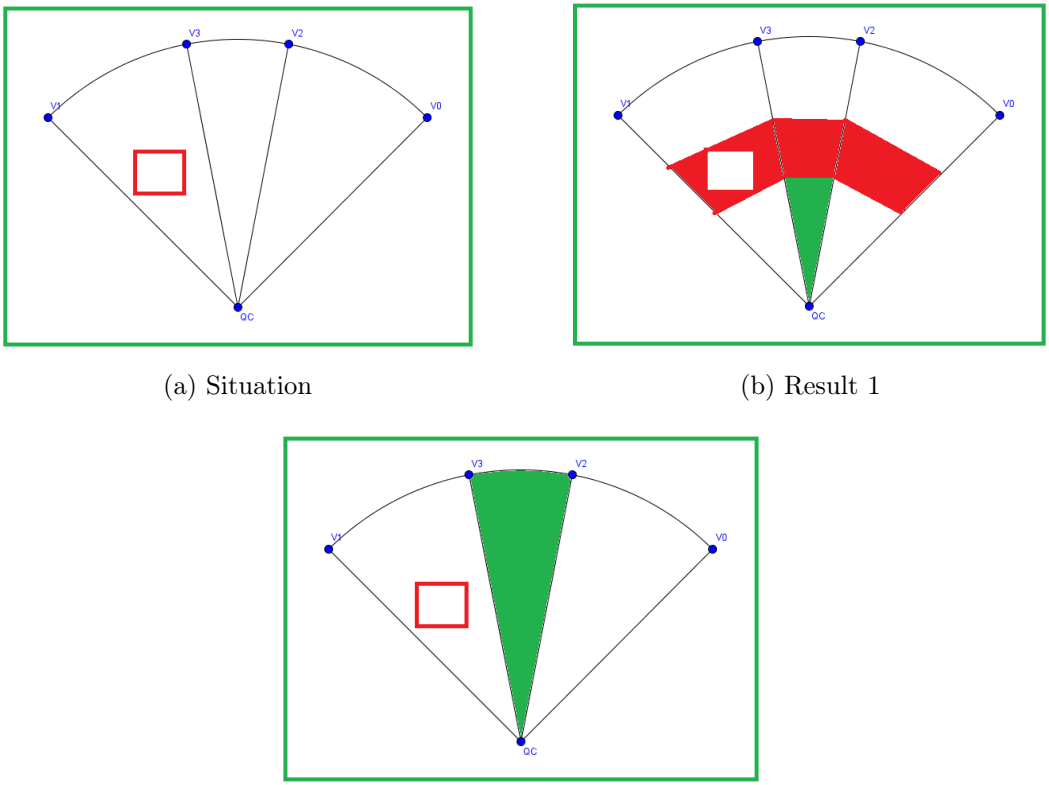
If an object is detected it will look a little different as seen in figure 17.4. It is not known where in the entire max angle the obstacle is, we only know the distance to the obstacle and therefore the entire arch has probability subtracted, while we only add probability to the area within the minimum angle which has a distance to it that is less than the distance to the obstacle.



(a) Situation (b) Result

Figure 17.4: Sensor behaviour with obstacle detected situation 1

This is also what happens in figure 17.5, however there are two possible outcomes, because it is not certain that the sensor will detect the obstacle, this is shown in said figure.



(a) Situation

(b) Result 1

(c) Result 2

Figure 17.5: Sensor behaviour with obstacle detected situation 2

QCPath class

The `QCPath` class is responsible for storing and finding the quadcopter's path. The path is saved as a stack of integers, where each integer is the id/index of a `FloorCell` in `gridList`.

`QCPath` contains a couple of methods, which purposes are to find a new path for the quadcopter. The method `FindPath()` is called whenever a new path is needed. This is where Lee's algorithm has been implemented. The four steps are done/called one at a time in the `FindPath()` method. Because we want to save the information found during the search, we start by doing step four (clearance) instead of ending with it, this allows us to see the breadth-first search after the search is done. This is done by going through `gridList` and each `FloorCell` has the variables `marked` set to false, and `steps` set to 0.

Next step one (initialization) is done, where we find the cell to start from using the quadcopter's position, as well as adding the `id` of this cell to the list `savedId`.

Then step two (wave expansion) is done by continuously giving the method `PathWaveExpansion()` `savedId` as input, which then returns a new list of id's and overwrites `savedId` with this list. If it is the first time going through the while loop in step two, then the new `savedId` list will be the id's of all cells with `steps` == 1, then 2 and so on. This happens until either a goal cell has been reached, or if the count of `savedId` has reached 0 which means that no path exists. If a goal cell is reached, `savedId` will be cleared, and the goal cell will be added to `savedId`. For each run of the while loop, as long as no path has been

found, the `CheckMargin()` method is called. This method makes sure that none of the cells which the list `savedId` refers to, are in too close proximity of an obstacle. This range is set to three meaning that the position of the quadcopter's center has to be at a distance of at least 4 cells (80 cm) from a wall, giving the quadcopter approximately 40 cm of safety distance from the edge of its propellers. When it is done it overwrites the `savedId` list with an updated list.

The last step that is done is step three (backtracing), where we find the actual path and save it. When this step is reached, the only id left in `savedId`, is that of the goal cell. First we push the id of the goal cell to `path`, which means that the goal cell will be at the bottom of the stack. Then a loop uses the method `PathBacktrace()` to backtrace through the wave expansion by always selecting a cell with a lower steps count, until the start cell is reached, while pushing the id's of these cells to `path`.

The system uses two different modes of selecting goal cells, these search modes will be explained now.

Search mode 1 In this mode the quadcopter tries to move along obstacles in order to map along the walls, but also to clean up walls which are imprecise.

Search mode 2 In the second search mode the quadcopter tries to find the nearest grid cell with a probability between 25 and 84 (uncertain cells).

The way these search modes work is that, depending on which one is chosen, they change what a goal cell is defined as. An illustration of the search modes can be seen on figure 17.6.

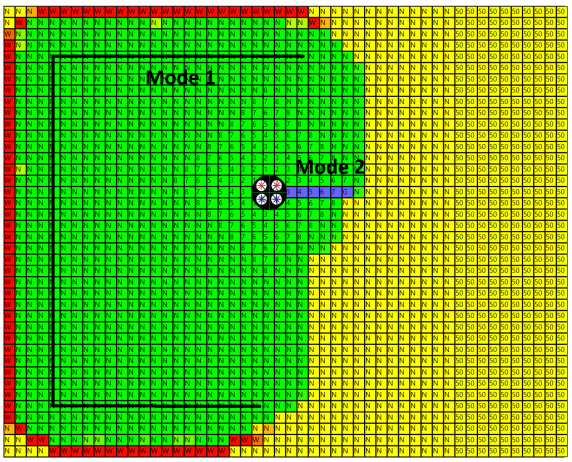


Figure 17.6: Search modes

Simulation class

The `Simulation` class represents a simulated situation for the quadcopter.

For simulating a room the `FloorCell` class has a boolean variable called `obstacle`, this variable is used to create simulated information for the quadcopter, instead of using data

from the sensors. Using this variable a fake room can be defined, which the quadcopter can then try to explore. Only the part of the code responsible for simulating the sensors will be aware of this variable, meaning that the quadcopter wont have any knowledge of which cells are obstacles other than from what information it has gathered from its "sensors".

Figure 17.7 is a graphical and simplified representation of the `Simulation_step()` method. The yellow ellipses represent "if control structures", where the text within is a logical expression. The red squares represent foreach loops. The blue rectangles represent assignments and method calls. The green hexagons represent the else part of an if-else statement meaning that if one of the yellow ellipses logical expression is false then the path with a green hexagon is chosen, if one exists. If no else statement is attached, then only one path is available. Indent is used to represent what is encapsulated by the different if and foreach structures. Each part of the figure has a number associated with it, this will be referred to as line numbers.

This method is executed for each step of the quadcopter. On line one and two a new path is found and the view of the sensors is calculated and set. On line three a foreach loop going through `gridList` is entered which continues until line six. Here an if statement checks all the cells if they are occupying the same location as the quadcopter and in this case sets their `probability` to 99 since they cant be obstacles, this is also the place where cells variable `visited` is set to true. If a cell is touching the quadcopter and the same cell is an obstacle, then the simulation has failed and true is returned ending the simulation. The cells that are not touching the quadcopter which are also obstacles, will for each sensor be checked if they are in the line of sight of the sensors.

For each execution of the `Simulation_step()` method the sensors save the shortest distance each of them has to a detected object, since this is the distance we want the sensor to believe it has detected. On line six we send the simulated information to quadcopter and set a local variable called `nextClear` to true, this variable is used when we decide if the quadcopter should move. On line seven, an if statement checks if a path has been found and if the path has a length greater than zero. If this statement is false we skip to line twelve, however if a path has been found then a foreach loop will go through `gridList` and an if statement will check each cell if it is within one cell of the next cell in the path. If a cell is in fact that and at the same time it has a `probability` of less than 75, then `nextClear` will be set to false and we break the foreach. This is done because the quadcopter should not move onto any cells with a `probability` less than 75. When the foreach loop is done an if statement will ask if `nextClear` is still true, if it is then the quadcopter moves else it just goes to line eleven. At this line we check if the quadcopter is stuck in a loop where it cannot decide between two paths, if this is the case then the quadcopter is forced to pick one of them, and continue down that path for the next couple of steps until the path has been taken to the end, or if any new information blocks that path.

Whether or not a loop is detected, this path of the code will result in the method returning false which means that the simulation is not done yet. Next line is number twelve which is accessed if and only if no path was found. At line twelve and thirteen we check which

search mode is active, switch the active search mode and count up the amount of times search modes have been changed. If search modes have been switched between more than 4 times then the method returns true and the simulation has been completed, if not then false is returned and the simulation continues.

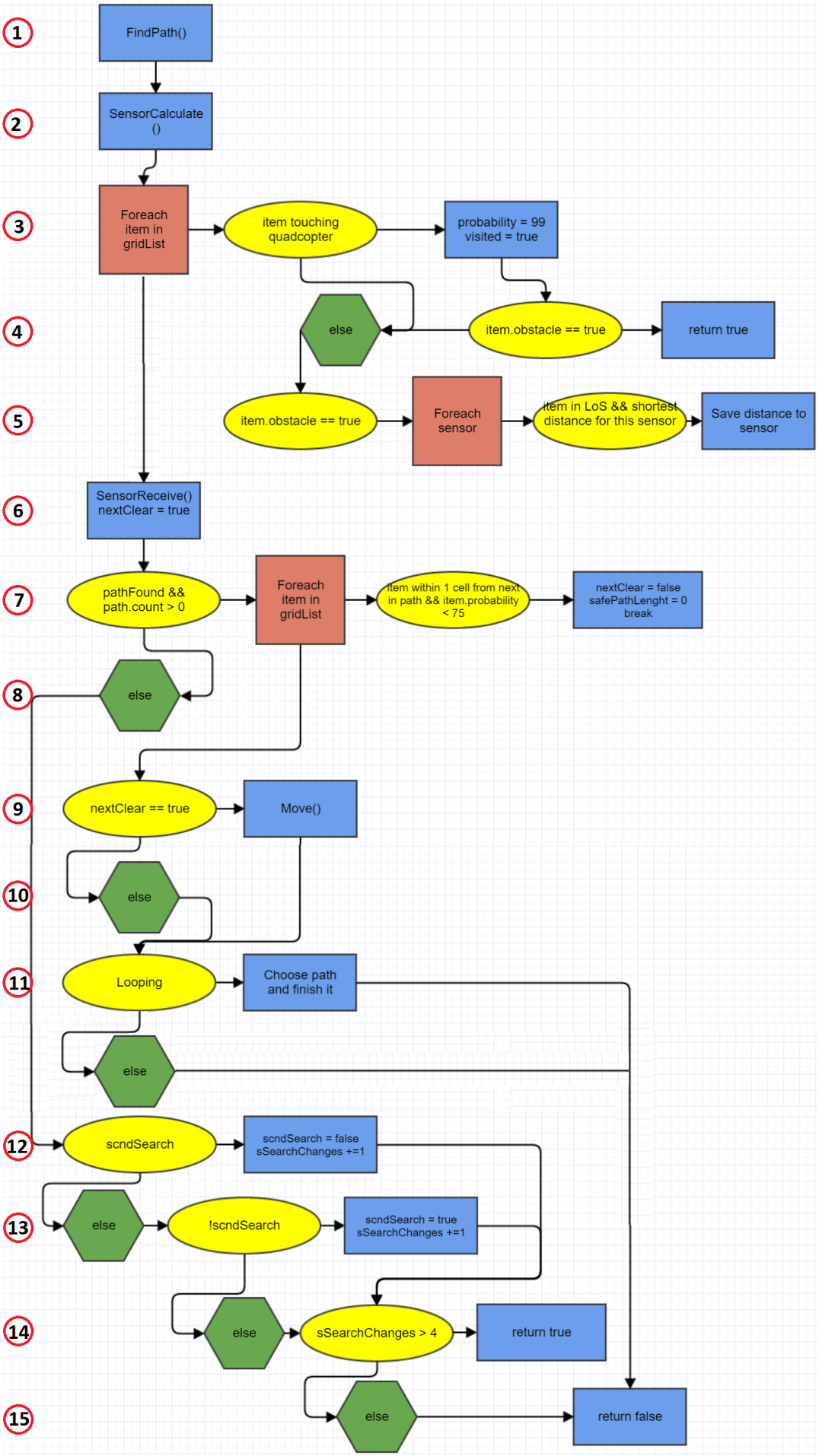


Figure 17.7: SimulationStep()

Step one test results

The values of two variables called `probChangeFree` and `probChangeObstacle`, that determine how much probability is added and subtracted from grid cells when they are detected by sensors, were found by simulating the mapping of multiple rooms in order to find the optimal values for these variables. These variable have a significant impact on the precision of the simulations. The simulations can be read about in-depth in appendix D.5.

The test and analysis of the results of this step is described in-depth in appendix D.6.

As seen in table 17.1, which contains the results from the test, there are two methods of calculating precision. Precision 1 is calculated by comparing how many grid cells are in the current map, with how many should be free and how many is above 75 in probability, which means that they are free. The same is done with obstacles, but instead of being above 75 they need to be below 25. In this method all cell with a probability between 25 and 75 are classified as errors. Precision 2 is a little less harsh, this method looks at each individual cells probability. For obstacles it takes the difference between zero and the cells probability, and for free cells it takes the difference between 99 and the cells probability, these differences are the amount of error in the drawing. With this method the correctness of drawings start at 50% since all cells has a probability of 50 from the start.

Maps	Precision 1	Precision 2	Steps
1	96.81%	95.07%	340
2	94.7%	92.67%	428
3	96.57%	94.66%	429
4	91.2%	88.45%	406

Table 17.1: Step one test results

Based on the results found during the simulations, we believe that the system is ready for its first physical test using the actual sensors, which can be read more about in the next section.

17.2 Step two: physical test

The physical test can be read in-depth in appendix D.7

During the test, a problem was quickly discovered. The sensors which had an angle of less than about 45 degrees from a wall would give very inconsistent results. Therefore a test was conducted, which can be seen in section 5.2 Angled walls, to determine how big a problem this is. This resulted in the quadcopter not being able to detect walls in most circumstances, and thereby giving an inconsistent result/mapping of a room. This meant that the test ended quickly, since the inconsistent results from the sensors with an angle, made the system automatically remove the walls made by the perpendicular sensors.

Distance sensor limitations

The distance sensors have various limitations that severely limits the accuracy of the building plan while the quadcopter is exploring. These various limitations will now be listed and described, and can be seen on figure 17.8

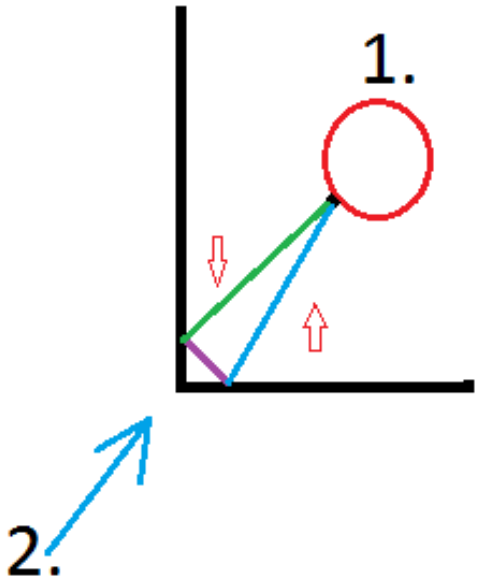


Figure 17.8: Problems with the distance sensors visualized

1. **problem** is seen on figure 17.8 at the top right side of the black wall. When the quadcopter is trying to map a corner, sometimes a distance sensor will output a distance value that is higher than the actual distance to the wall. This problem occurs when the pulse from a distance sensor hits one wall, bounces towards another wall, and then bounces back to the same sensor before it has received any other pulse. The distance value returned by the sensor in this case, will make the map inaccurate in the sense that the wall is displayed as being some distance further away than it actually is. This problem cannot be fixed by software, as it occurs due to a limitation of the distance sensor used, and so the case where the problem occurs cannot be predicted when mapping.

2. **problem** is seen on figure 17.8 on the lower left side of the black wall. If the quadcopter is flying towards a corner of a wall like displayed, none of the distance sensors are able to obtain a return pulse from the wall, as they will be redirected away from the quadcopter, if the quadcopter is flying towards the corner as displayed by the arrow away from 2. on the figure. Therefore, when mapping the room, the wall is non-existent to the quadcopter and the quadcopter will fly into the wall. This problem occurs because of how the distance sensor receives a distance value, and as such cannot be solved using software.

To address this some changes were needed in the system, for the quadcopter to be able to map a room, without to much inconsistency in the results. These changes can be read

about in the following section.

17.3 Step three: updated simulation

First the simulated data sent from the sensors in the system, have to more accurately model the sensor data supplied by the quadcopter, before it is possible to accurately simulate how the quadcopter actually maps a room. In our renewed test of the sensors it was discovered that all sensors with an angle less than 45 from wall would return distances approximately twice the real distance. In the simulation the quadcopter never turns, but at all times it has four diagonal sensors which are always pointed at walls with about a 45 degree angle. Outside of the simulation we can not know what angle any of our sensors have to walls, which poses a problem. If we can not ever know which sensors are giving us incorrect readings, then we need to take into account that all sensors might be doing this. In order to simulate this problem, we need to make sure that the system behaves the exact same way for all sensors, and that none of them has special cases when the system is handling diagonal sensors. However we can make changes to how we simulate sensor data, and this is how a more realistic world view will be created. This is done by telling the diagonal sensors in the system that they should all return twice their original reading. This approximately represents what the sensors did in the first physical test. After doing this the exact same problem occurred in the simulation as in the physical test, where walls are created by the perpendicular sensors and is then removed by the diagonal sensors.

The next step is to fix this issue, such that the simulation will again be able to draw a room. This problem will however reduce the quality of any produced drawings since the sensors are much harder to rely on.

Compensating for sensor limitations

The strategy used to solve this problem is that, the shorter a distance that is received from a sensor the more likely it is to be true, since the incorrect readings usually gives a longer distance than the correct one. This knowledge is gonna be exploited for fixing the issue.

Each cell in the grid have to hold more information about how far away from the quadcopter the cell has been read, to ensure that the accuracy of close readings do not get overwritten by sensor readings from further away.

Step three results

After the simulation code was changed, new values for `probChangeFree` and `probChangeObstacle` were needed. The test to find these new value can be found in appendix D.8.

The test and analysis of step three's results is described more in-depth in appendix D.9.

The results of the step three simulation test can be seen in table 17.2.

Maps	Precision 1	Precision 2	Steps
1	86.75%	85.93%	324
2	79.25%	80.7%	476
3	79.6%	84.86%	682
4	77.05%	80.17%	525

Table 17.2: Step three test results

Based on the results found during the new simulations, we believe that the system is ready for another physical test using the actual sensors, which can be read more about in the next section.

17.4 Step four: physical test 2

In this section step four, physical test two, will be explained and we will show the results of doing another physical test with the updated software, accounting for the problem with sensors which are diagonal compared to the walls, giving inconsistent and wrong results, see section 5.2.2.

The test were done with the same setup as in the first physical test, with the same Arduino code sending the sensor data to the JY-MCU Bluetooth module, which is then grabbed by the system. The only thing changed test-wise is the system software, which can be read about in the previous section, section 17.3.

The test was done in the same way as the first test, by placing the representation of the quadcopter in a room, activating the Bluetooth communication, and letting the system step by step tell how to move the representation. To make sure the representation of the quadcopter were moved precisely two measuring tapes were used. Pictures of the end of the test, both of the presentation and the program, can be seen on figure 17.9.

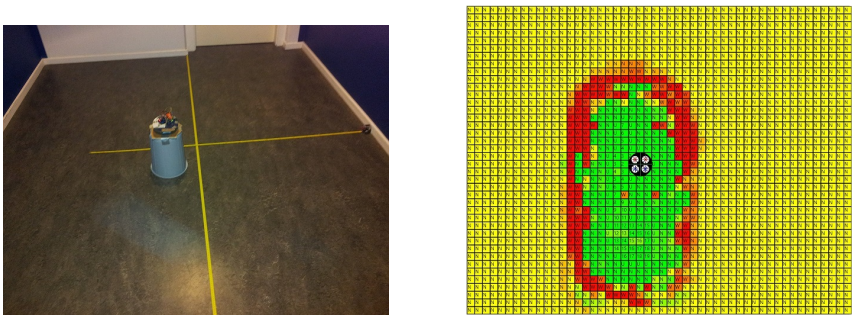


Figure 17.9: Pictures of the first physical test after the system got updated. In the left column the quadcopter representation can be seen placed in the room, and in the right column the system mapping the room.

As it can be seen on the pictures, the system is not that good at handling corners. This is because of two things; the first is that the diagonal sensors will give wrong results when pointed at a corner, as mentioned before. The second is that the room is small, being only 5.1×2.3 meters, which means that the sensors can hit both walls at the same time,

making the system use the data from the perpendicular sensors, since it is not possible for the representation to move along the wall.

Even with these problems the system still gives a good result, when counting out the corners. Counting each block from top to bottom and left to right, depending on from which block is counted as the wall, will yield results close to the actual length and width of the room. If counted as seen on figure 17.10, the room will be 24×11 cells, which gives a result in meters as 4.8×2.2 meters, which is within 30 centimeters of the actual length and width of the room.

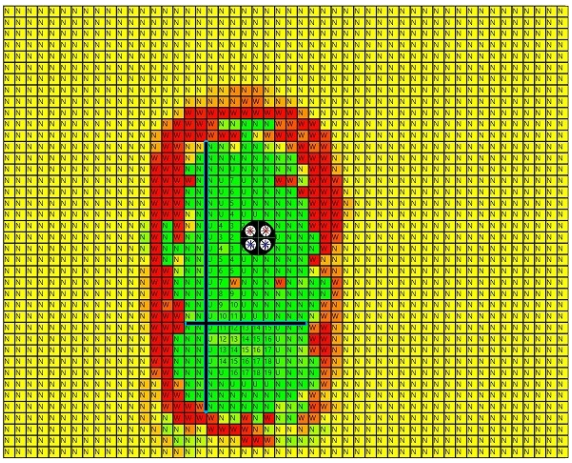


Figure 17.10: Lines showing a possible way to count from wall to wall

One of the reasons that the system is off by around 30 centimeters might be because of the way the representation were moved. Since it were moved by hand, depending on two measuring tapes, it can easily have been moved slightly less or more than 20 centimeters per step, or moved a little in another axis than it should have been.

With the result of this test being close to the actual size of the room, but having bad results with corners it were decided to do another test, in a bigger room to see if this helped with the result.

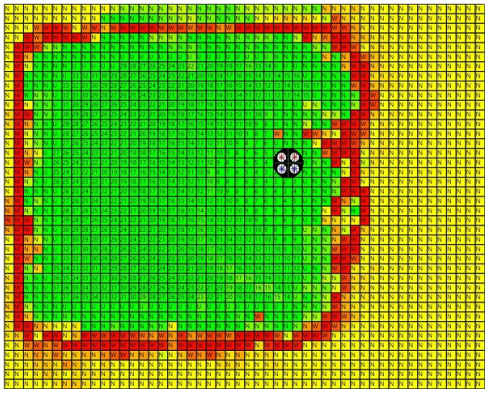


Figure 17.11: Pictures of the second physical test after the system got updated. The left of the quadcopter representation seen placed in the room, and the right of the system mapping the room.

The second room for testing was a lot bigger, being 6.5×7.2 meters. Again the test was done completely the same as the first two tests, the representation of the quadcopter was placed in the upper corner and the system was started, and the representation moved as told by the system. Step by step pictures of the test, both the physical placement of the representation and the system, can be seen in appendix D.2. The end result of the test can be seen on figure 17.11, where it can be seen that the top, left and bottom wall is more straight with this test, than the previous. The wall to the right is not straight, and this is because of the different search modes the system has. As the quadcopter representation were moving along the right wall, the systems search mode were in number 2, and it only searched to clear uncertainty, and thereby did not straighten out the wall, as seen with the other walls.

Looking at the precision of the system, it can be counted on figure 17.12, that the room should be either 6.2×6.4 m or 6.2×7 m, depending on which width is chosen. This is somewhat close to the actual size of the room, being off by either 30 and 80 cm or 30 and 20 cm. The reason for the result being off, is the same as with the last test, that moving the representation depending on two measuring tapes on the floor and moving it by hand is not optimal.

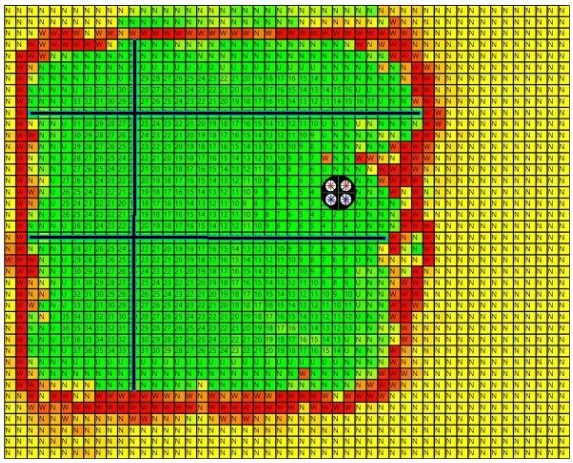


Figure 17.12: Lines showing a possible way to count from wall to wall

With the result in precision combined with the corners of the room in the system looking a lot better, the test can be concluded as being a success. There might be some adjusting to do with how the system chooses search mode, to make sure it covers and give consistent, straight walls every time.

Task conclusion 18

The requirements of task four has only been completed partially, due to unexpected difficulties with the distance sensors and the delimitation of task three's implementation. The system was supposed to be able to send commands to the quadcopter, and receive data from the sensors, however only the receiving part was implemented.

Although the system is able to figure out which commands the quadcopter should receive, the part about sending commands to the quadcopter would only be needed if we were to test the quadcopter flying by itself.

The data received from the distance sensors was intended to be used for drawing a building plan, this was achieved in the sense that the system was able to use the information from the sensors and make changes to the building plan based on the information.

The system is able to map and navigate the quadcopter through a room both in the simulations and by giving commands which we used to move it around a room by hand.

We were never able to do a test where we let the system from task four control the quadcopter, and drawing a map that way. This is mainly due to task three not being finished, since without knowing a somewhat precise position of the quadcopter, the system will not be able to properly navigate the quadcopter.

The drawings which we managed to create with the handheld test, did allow us to create a decent drawing of a room, however the drawing is still far from perfect.

Part VI

Evaluation

Reflection 19

In this section, the project process will be reflected upon, to determine and clarify the factors that were relevant to the process, when they occurred, their importance to the process as a whole, and what we learned due to these factors. Alternatives to what was done, will also be discussed.

Work process

Initially, in the project startup phase, the scope of the project was not very well defined. None of the group members really knew what an embedded system entailed, so brainstorming ideas that suited the theme were few. It was decided by vote to work on an idea that started out by using a quadcopter to pick up trash, and the quadcopter would then throw out the trash into a trash bin by itself. After some time of discussion and analysis of the project domain, it was decided that this initial idea did not really feel interesting to work with. Therefore, after some discussions, we decided that the project idea had to be configured a bit. The idea of using a quadcopter as the body for the embedded system seemed interesting and doable at the time, so it was just a matter of changing perspective with regards to which problem should be solved by the embedded system we were going to design and implement.

We ended up with the current idea for a problem domain, as stated in section 0.1.1, which was deemed interesting and enough relevant theory could be integrated into the embedded system. The idea was discussed for some time, and some concerns were raised with regards to the difficulty of the problem at hand; we had only limited knowlegde about how quadcopters worked, so how could we determine whether or not this problem domain was doable within the given timeframe? We initially tried to figure this out by analyzing embedded systems and quadcopters in general, and came to the conclusion that it would most likely be doable, but we were well aware that this project was not going to be an easy one. From the beginning, we were enthusiastic about the defined project domain, and as such we started off by covering the bases for the project process.

The project was analyzed, and the different tasks were defined. Based on experience from previous projects, we started out by defining the requirements for each task and quickly began researching possible hardware for the project's embedded system. Due to an early deadline for ordering hardware, and because of very little knowledge of the hardware

ordering process, the decisions had to be somewhat rushed to make the deadline. Looking back, not enough time was used researching quadcopters, which could have prepared us for future problems in the project timeline. The decision to try and create our own flight controller, that could gather the data needed for stabilization and flight, which could be used by the quadcopter embedded system, proved to be a problematic bottleneck for the project as a whole.

The first bottleneck happened due to how the project was planned, in conjunction with the decision made to create our own flight controller for the system. Even after researching the hardware thoroughly, and trying to implement various software fixes that could fix the problem that arose, we ran into a problem with noise in the data supplied by the IMU, the hardware component used to measure stability for flight. This hardware component was such an essential part of the embedded system, that the project as a whole simply could not continue until this problem was fixed, as all the defined tasks required the quadcopter to be able to stabilize or fly. Implementing the IMU as a data gathering component for the quadcopter embedded system proved to be such a time-consuming task, that deadlines were missed and only simple analysis and rough design of future tasks could be worked on beside trying to overcome the bottleneck that was the IMU.

The whole process that preceded the bottleneck did not seem to have prepared us for such a situation, which meant a lot of work hours were wasted on trying to find something meaningful, that the group members not working towards a solution for the data noise problem, could do to keep the project going during the bottleneck phase. As this bottleneck happened relatively early in the project's timeline, there were several subtasks for each task that could be worked on, while we were trying to find a solution to the noise problem, so at the time it did not seem like a huge bottleneck for the project had actually occurred, because we assumed that the problem would be fixed before most of the subtasks were completed. That, however, was not the case. We slowly completed the defined subtasks for the future tasks that had to be worked on after task one, where the objective was to make the quadcopter self-stabilize. When we reached the point where some members had nothing useful to work on, we became aware of the severity of the problem at hand and realized that we had to do something quickly in order to mitigate the fallout from this bottleneck as much as possible.

Mitigating the fallout started out by trying to design and implement whatever possible for the following tasks after task one, with the idea in mind that we would try to interface the tasks wherever possible, and redesign and reimplement where interfacing the tasks did not pan out. After discussing and analyzing the problem at hand, we came to the conclusion that the only task that could be designed and implemented in any meaningful fashion was the designated task four. Acting swiftly when we became aware of the problem at hand, resulted in task 4 being implemented in a fully simulated environment, that could later be interfaced with the quadcopter system through Bluetooth communication between the quadcopter system and an external machine simulating task 4.

Throughout the problematic process with the bottleneck, we became more consistent with regards to trying to determine possible bottlenecks later in the project, and work as hard as we could to try and analyze the problems thoroughly so another huge bottleneck would

not occur again.

Project development process

For this project, the development process model used, was the incremental development process model described in section 0.3. This model served us very well, with regards to defining the problem domain for the system as a whole, and splitting that into four tasks that had a natural flow of requirements, where each task required the previous task to be completed before it could be implemented.

Because of this observation early in the project, that each task required the previous task to be completed, we were focused on completing each task sequentially without wasting work and effort on the following tasks, as this work could be deemed worthless based on the findings for task one. In retrospect, this natural flow of tasks was also a negative factor that contributed to the severity of the bottleneck, as work effort and time was spent doing menial work in the project domain for task one, relative to what could have been achieved by spending that effort and time in the following tasks instead. This decision was made because at the time we were convinced that spending this effort and time on the following tasks would do more harm than good, as we were not sure what would change during our work, up to each task. Later in the project period, after finishing task one, the tasks two and three were redesigned to have better fitting problem domains, and as such our reluctance to spend more time on these tasks during the bottleneck was not entirely unfounded.

After working with the incremental development process model, we became aware of the severity problems in each stage had for tasks down the development road. Each task with a high dependency on the previous task(s) was delayed if the tasks beforehand were delayed, and mitigating this delay was only possible by risking work and effort being wasted, due to the fact that the project domain for the tasks could be changed at any point after the work was done. Due to previous experiences with regards to changed problem domains, we were reluctant to risk work being wasted because we were aware that the project as a whole required efficiency if we wanted to finish it, due to the difficult nature of the project domain.

The development process for this project showed how problems that seem small can escalate into project bottlenecks, and that a greater understanding of the hardware in question for a project is required in order to plan towards mitigating problems that arise due to the nature of the hardware used. Had we introduced a re-evaluation of the requirements for each task, after the analysis phase, we could probably have changed the requirements so they would have fitted each task better, so only requirements that could be fulfilled, would actually be worked on.

Conclusion 20

In this section the project as a whole will be concluded upon, with regards to the functional requirements defined in section 0.2, and the problem statement seen in the citation below. The process has been defined by the incremental model, with respect to the problem domains of each task. There was a natural flow from one task to the next, making the interfacing of each task simple.

"How can we implement a quadcopter as an embedded system, capable of flying around and exploring an area in an effort to map the surroundings, while being able to stabilize and avoid obstacles, by itself?"

Based on the subsystems concluded upon after each task, we were able to implement an embedded system capable of flight stabilization, movement, obstacle avoidance, takeoff and landing. We were not able to implement the subsystem tasked with integrating exploration and mapping, with the rest of the system, since implementing a fully functional position determination feature, was not achieved.

The sensors chosen for this project were inaccurate and gave several problems, but we were still able to use the information supplied by them to implement this system. The IMU, that was used to gather data in order to achieve stabilization, caused problems, so we decided to replace it with a flight controller that could perform the actions which was required of the IMU.

The subsystem implemented in task four was able to map a room created in a software simulation somewhat precisely, without human interaction, as well as in a physical test of the software moving the system by hand. Based on these results the requirements concerning the exploration and mapping of a room are fulfilled. Based on the pathfinding algorithm implemented, the subsystem is focused on ensuring the safety of the system as a whole, as it will avoid calculating paths that require the system moving close to walls.

We were not able to complete task four without exceeding the memory limit of the Arduino and therefore it was necessary to outsource the software responsible for mapping a room to a PC. A snippet of the memory usage from the software can be seen in figure 20.1, Visual Studio's diagnostic tools were used to determine the memory usage of the software.

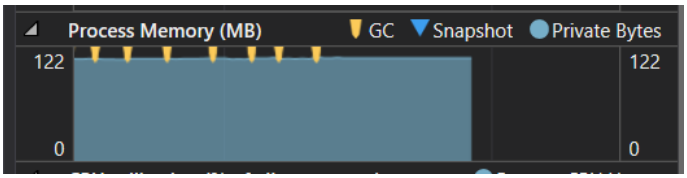


Figure 20.1: Memory usage of the software for task four, in a 8×8 m room

We have designed and implemented an embedded system that has a subsystem capable of stabilizing in the air, movement by commands and avoiding obstacles, and another subsystem capable of mapping a room based on data from distance sensors. However the two subsystems do not work together without position determination, though both systems are made to be compatible.

Future work 21

As there is a deadline to complete this project, there is not enough time to implement all features and ideas that we would want in the ideal system. This section will therefore describe ideas for additions to the system.

Implement and integrate task three and four into the system

Because of lack of time, task three and four never got implemented and integrated into the system and therefore these two tasks have to be categorised as future work.

3D mapping

As it is of now it is only required for the system to map a room as a 2-dimensional representation, meaning it does not have the ability to distinguish between objects of different heights. With 3D mapping the system would get a more precise image of a room, with it being able to register objects at different altitudes, if an object is too low to the ground or too high towards the ceiling a 2D mapping would not register it, depending on the altitude of the quadcopter. It would also have the ability to register the height of an object and therefore also be able to extent its area of flight, as it would be able to fly over and under objects as well.

Map an entire building

In the analysis of task four it is mentioned that the grid spaces' dimensions are $20cm \times 20cm$ so the quadcopter can fly through doors, but this is never tested with the distance sensors because of the errors in the readings from the sensors. In the future using different sensors of higher quality might give the opportunity to allow the quadcopter to fly through doors. With this ability the system would be able to map an entire floor of a building, and together with 3D mapping it would perhaps be able to map an entire building.

Distinguish between moving and still objects

Right now the system registers all objects as still objects even objects in movement, like a person. A future work could therefore be that the system could distinguish between moving objects and still objects. This could benefit in the way that if the object registers a moving object, instead of marking the position of the object as a permanent obstacle, it could mark the position as a temporary obstacle.

Improvement in registering objects with curves

The distance sensors in the system has trouble registering objects with curves, such as round objects. This is because the curves do not reflect the sound waves back to the sensors. This creates a problem where the sensors are not able to detect the object. A future work could therefore be finding a solution to this problem.

Implement a 360 degrees camera

Implementing a 360 degrees camera to the system that could take a picture every x seconds while mapping a room. This could give the opportunity to create a virtual replica of a room with it, giving the possibility to virtually explore the room.

Bibliography

[1] Wikipedia. *U.S. military UAS groups*. URL: https://en.wikipedia.org/wiki/U.S._military_UAS_groups.

[2] Amazon. *Amazon Prime Air*. URL: http://www.amazon.com/b?node=8037720011.

[3] Peter Marwedel. *Embedded System Design*. 2nd. Springer, 2011.

[4] Peter Marwedel. *Embedded Systems Design*. URL: https://ls12-www.cs.tu-dortmund.de/daes/media/documents/staff/marwedel/es-book/slides11/es-marw-1.1.pdf.

[5] Steve Heath. *Embedded Systems Design*. URL: http://books.google.com/books?id=BjNZXwH7HlkC&pg=PA2.

[6] Ian Sommerville. *Software Engineering*. 2016.

[7] Corrado Santoro. *How does a Quadrotor fly? A journey from physics, mathematics, control systems and computer science towards a "Controllable Flying Object"*. URL: http://www.slideshare.net/corradosantoro/quadcopter-31045379.

[8] Teppo Luukkonen. *Modelling and control of quadcopter*. URL: http://sal.aalto.fi/publications/pdf-files/eluu11_public.pdf.

[9] Carol Hodanbosi. *Newton's Third Law of Motion*. URL: https://www.grc.nasa.gov/www/k-12/WindTunnel/Activities/third_law_motion.html.

[10] Arduino. *Arduino - Introduction*. URL: https://www.arduino.cc/en/Guide/Introduction.

[11] B_E_N @Sparkfun. *What is an Arduino?* URL: https://learn.sparkfun.com/tutorials/what-is-an-arduino#re.

[12] Wikipedia. *Arduino*. URL: https://en.wikipedia.org/wiki/Arduino.

[13] Arduino. *Arduino MEGA*. URL: https://www.arduino.cc/en/Main/arduinoBoardMega.

[14] Wikipedia. *Brushleess DC electric motor*. URL: https://en.wikipedia.org/wiki/Brushless_DC_electric_motor.

[15] Mat Dirjish. *What's the difference between brush DC and brushless DC motors?* URL: http://electronicdesign.com/electromechanical/what-s-difference-between-brush-dc-and-brushless-dc-motors.

[16] Matthias 'madc' Esterl. *Arduino implementation for GY-85*. URL: https://github.com/madc/GY-85.

[17] Pieter Jan. *Reading a IMU Without Kalman: The Complementary Filter*. URL: http://www.pieter-jan.com/node/11.

[18] Gary Bishop Greg Welch. *An introduction to the Kalman filter*. URL: http://www.cs.unc.edu/~tracker/media/pdf/SIGGRAPH2001_CoursePack_08.pdf.

[19] Wikipedia. *Kalman filter*. URL: https://en.wikipedia.org/wiki/Kalman_filter.

[20] Wikipedia. *PID controller*. URL: https://en.wikipedia.org/wiki/PID_controller.

[21] Ill. Paul Avery Senior Product Training Engineer Yaskawa Electric America Inc. Waukegan. *Introduction to PID control*. URL: http://machinedesign.com/sensors/introduction-pid-control.

[22] Brett Beauregard. *Arduino PID Library*. URL: http://playground.arduino.cc/Code/PIDLibrary.

[23] hauptmech. *What are good strategies for tuning PID loops?* URL: http://robotics.stackexchange.com/questions/167/what-are-good-strategies-for-tuning-pid-loops.

[24] AbuseMark. *Flight Controller Naze32*. URL: http://www.abusemark.com/downloads/naze32_rev2.pdf.

[25] Lucien Miller. *Flight Controller Boards - The heart and soul of a multi-rotor machine*. URL: http://multirotorpilotmag.com/flight-controller-boards/.

[26] Adam Fabio. *Droning on: Choosing a flight controller*. URL: http://hackaday.com/2014/06/06/droning-on-flight-controller-round-up/.

[27] Thomas A. Kinney. *Proximity sensors compared: Inductive, capacitive, photoelectric, and ultrasonic*. URL: http://machinedesign.com/sensors/proximity-sensors-compared-inductive-capacitive-photoelectric-and-ultrasonic.

[28] robotplatform. *Types of robot sensors*. URL: http://www.robotplatform.com/knowledge/sensors/types_of_robot_sensors.html.

[29] Arduinobasics. *HC-SR04 Ultrasonic Sensor*. URL: http://arduinobasics.blogspot.dk/2012/11/arduinobasics-hc-sr04-ultrasonic-sensor.html.

[30] lidar-uk. *How LIDAR works*. URL: http://www.lidar-uk.com/how-lidar-works/.

[31] wikipedia. *Lidar*. URL: https://en.wikipedia.org/wiki/Lidar.

[32] Wikipedia. *Radar*. URL: https://en.wikipedia.org/wiki/Radar.

[33] Hermann Rohling Michael Klotz. *What are good strategies for tuning PID loops?* URL: 24%20GHz%20radar%20sensors%20for%20automotive%20applications.

[34] Christian Wolff. *Frequency-Modulated Continuous-Wave Radar (FM-CW Radar)*. URL: http://www.radartutorial.eu/02.basics/Frequency%20Modulated%20Continuous%20Wave%20Radar.en.html.

[35] Faris A. Kochery Manaf A. Mahammed Amera I. Melhum. *Object Distance Measurement by Stereo VISION*. URL: http://www.warse.org/pdfs/2013/icctesp02.pdf.

[36] Damir Vrančić Jernej Mrovlje. *Distance measuring based on stereoscopic pictures*. URL: http://dsc.ijs.si/files/papers/S101%20Mrovlje.pdf.

[37] Pirkko Oittinen Mikko Kytö Mikko Nuutinen. *Method for measuring stereo camera depth accuracy based on stereoscopic vision.* URL: http://www.helsinki.fi/~msjnuuti/pdf/EI_2011_kyto_preprint.pdf.

[38] DealeXtreme. *Distance Sensor HC-SR04.* URL: http://eud.dx.com/product/hc-sr04-ultrasonic-sensor-distance-measuring-module-844133696.

[39] DealeXtreme. *HC-SR04 datasheet.* URL: http://m5.img.dxcdn.com/CDDriver/CD/sku.133696.pdf.

[40] Guangzhou HC Information Technology Co. *JY-MCU Datasheet.* URL: http://silabs.org.ua/bc4/hc06.pdf.

[41] Core Electronics. *JY-MCU Bluetooth to UART Wireless Serial Port Module for Arduino.* URL: https://core-electronics.com.au/attachments/guides/Product-User-Guide-JY-MCU-Bluetooth-UART-R1-0.pdf.

[42] Arduino. *Serial.* URL: https://www.arduino.cc/en/Reference/Serial.

[43] CHrobotics. *Using Accelerometers to Estimate Position and Velocity.* URL: http://www.chrobotics.com/library/accel-position-velocity.

[44] David Vincent. *Accurate Position Tracking Using Inertial Measurement Units.* URL: http://www.pnicorp.com/wp-content/uploads/Accurate-PositionTracking-Using-IMUs.pdf.

[45] CHrobotics. *Estimating Velocity and Position Using Accelerometers.* URL: https://www.pololu.com/file/0J587/AN-1007-EstimatingVelocityAndPositionUsingAccelerometers.pdf.

[46] Amit Patel. *Introduction to A*.* URL: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[47] Rajiv Eranki. *Pathfinding using A* (A-Star).* URL: http://web.mit.edu/eranki/www/tutorials/search/.

[48] Wikipedia. *A* search algorithm.* URL: https://en.wikipedia.org/wiki/A*_search_algorithm.

[49] Wikipedia. *Best-first search.* URL: https://en.wikipedia.org/wiki/Best-first_search.

[50] Salad raider. *A* Algorithm - A Quick Guide to the A* Algorithm.* URL: https://www.youtube.com/watch?v=eTx6HQ9Veas.

[51] Wikipedia. *Maze Runner.* URL: https://en.wikipedia.org/wiki/Maze_runner.

[52] Stefan Kraemer Jianjiang Ceng. *Maze router with Lee algorithm.* URL: http://www.oop.rwth-aachen.de/documents/oop-2007/sss-oop-2007.pdf.

[53] haizhou. *Maze Router: Lee algorithm.* URL: http://www.ece.northwestern.edu/~haizhou/357/lec6.pdf.

[54] Matteo. *3d Accelerometer calculate the orientation.* URL: http://stackoverflow.com/questions/3755059/3d-accelerometer-calculate-the-orientation/10320532#10320532.

Part VII

Appendix

Task one A

A.1 Technical description of GY-85's sensors

Each sensor will be described as if it was a stand-alone sensor, as each sensor in the IMU can either be used by itself, or in conjunction with any of the other two sensors. The magnetometer will not be used or described.

A.1.1 ADXL345

The specifications for the ADXL345 accelerometer are as following:

- **Output resolution :** 10-13 bit depending on G-range (detects inclination changes <1.0 degrees in the X, Y and Z axis)
- **Activity sensing :** Detects presence or lack of motion, or if acceleration exceeds user-set level

The ADXL345 includes features like tap and double tap detection as interrupt triggers.

Value in G = $measurement\ value \times (Grange/(2^{resolution}))$

The G-range and resolution used in this formula depends of the configuration of the chip. The ADXL345 supports the ranges $\pm 2g, \pm 4g, \pm 8g\ or\ \pm\ 16g$. As resolution, 10 or 13 bit can be set. Measurement value is the raw value read from the chip for one axis. If the ADXL345 is used with the default settings, a resolution of 11 bit and a range of $\pm 4g$(8g range) is used, and the following formula can be used:

Value in G = $measurement\ value \times (8/(2^{10})) = measurement\ value \times (8/1024) = measurement\ value \times 0.0039$

This calculation must be done for each axis (X, Y and Z), and in this case the G values for each axis is as following:

- $xg = valX \times 0.0039$;
- $yg = valY \times 0.0039$;

- $zg = valZ \times 0.0039$;

Having calculated the x, y and z values in G, these can be used to further calculate the angles. Based on the aircraft principal axes, the rotation on the X-axis is called roll, the rotation on the Y-axis is called pitch. The rotation on the Z-axis would be called yaw, but a 3-axis accelerometer is not able to measure it, as the force vector of gravity does not change during the movement.

The formulas for calculating roll and pitch are supplied by a stackoverflow user[54], and are as following:

- Roll = $atan2(yg, zg) \times 180/PI$
- Pitch = $atan2(-xg, sqrt(yg \times yg + zg \times zg)) \times 180/PI$

A.1.2 ITG3200

The ITG3200 is used as a digital compass to sense the angle from magnetic north in degrees. The specifications for the ITG3200 gyroscope are as following:

- **Output resolution :** 16 bit ADC
- **Sensitivity Scale Factor :** 14.375 Least Significant Bits (LSBs) per degree per second
- **Full-scale range :** ±2000 degrees per second

It is necessary to divide the measured value by the "Sensitivity Scale Factor" to get the value in degree per second. This "Sensitivity Scale Factor" can be found in the datasheet.

$(Value\ in\ degree)/s = (measurement\ value)/(Sensitivity\ Scale\ Factor)$

$= (measurement\ value)/14.375$

This scale factor has to be applied to all axis values.

- $xds = valX/14.375$
- $yds = valY/14.375$
- $zds = valZ/14.375$

To calculate the angle from the degrees per second measured by the gyroscope, the following formula is used:

$Angle+ = ((Value\ in\ degree)/s) \times (Delta\ Time\ in\ Seconds)$ This has to be applied to each axis as well, to determine the angle of orientation in each axis.

- Roll $+ = xds \times dtime$

- Pitch $+= yds \times dtime$
- Yaw $+= zds \times dtime$

A.2 Configuration and results of motor tests

In this appendix you can find a description of the configuration of the tests performed on the motors and the results from these tests.

A.3 Test configuration

The tests will not include maximum thrust capacity of the actuators, because the motors and ESCs are not build to run at maximum power at all times, and our quadcopter should not compete in speed or in anyway need to use maximum power. Also the motor, propeller and ESC was bought to be able to lift the quadcopter. Instead the motor sets are tested on consistency, so that we would know, when programming the quadcopter, how much the motors can differ. E.g. When trying to make still flight, should the thrust of all motors be set to the same throttle percentage, or is there a slight difference in the motors causing them to need different throttle percentage to achieve the same lifting power? Normally only one factor would be changed in a test like this one, but because it is not only the motors we want to test, but the whole setup on each arm of the quadcopter. This means testing each arm of the quadcopter this includes; motors, the ESCs and the propellers. This way we can compare the setup for each quadcopter arm to each other.

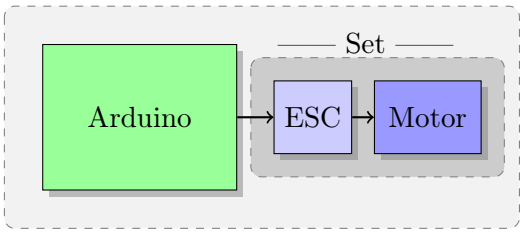


Figure A.1: Diagram over the Arduino, ESC and a Motor as used in the test.

To conduct the test, we made a special setup as seen on picture A.2. The setup consists of several parts, including: a digital scale, an Arduino, an ammeter, a bottle filled with water, a cardboard box, ESC and the motor with a propeller attached. The motor is fixed to the bottle filled with water, to ensure that it does not fly away. The arrangement is then placed on the digital scale, and then the cardboard box is placed around it. The reason the cardboard box is necessary, is so the air pressure when the propeller is spinning, is not interfering with our test, by pushing air onto the digital scale thus cancelling out the lift power of the motor. The cardboard box will take some of the lifting power away from the motor, but because of its placement under the propeller, the air cannot get away fast enough thus wasting power compressing air. It will not be a problem that the lifting power is lowered, if the motor sets are lowered with the same ratio. To ensure that they are lowered with the same ratio we only switch out the motor sets, between tests, see figure A.1. The digital scale is then reset just before every single testing, so that only

the lifting power of the motor setup is measured. When the motor has reached the given power percentage, and the digital scale shows a stable number, we note down the lifted weight. We did the test four times on each power percentage for each motor setup. We used 20%, 30%, 40%, 50%, 60%, 70% and 80% power.

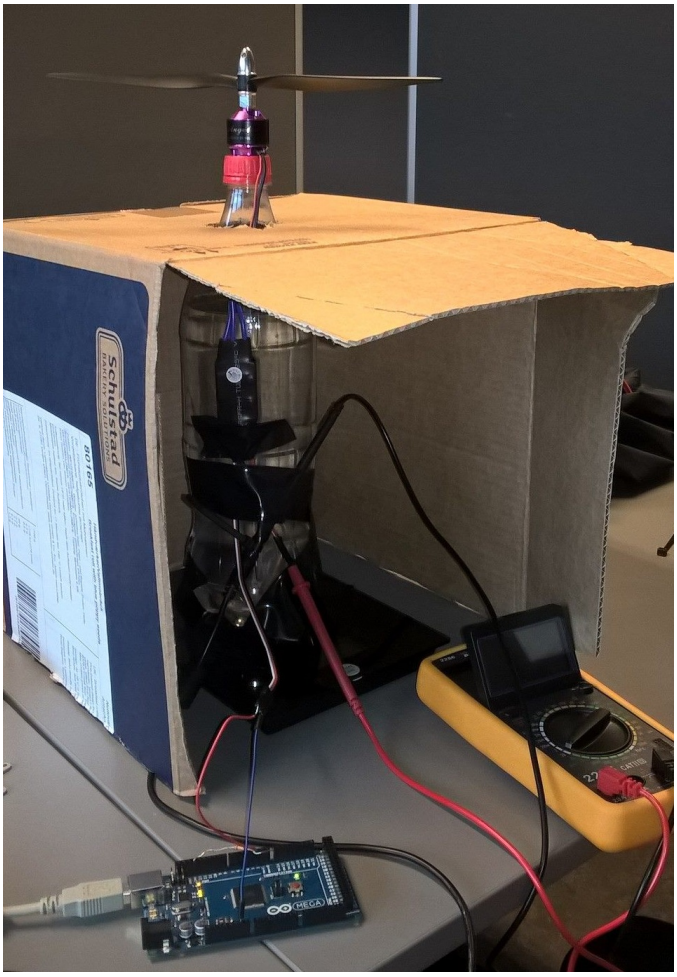


Figure A.2: Setup of the test

A.4 Test results

The results are given in grams (g), and the percentage is out of the maximum of the engines power.

Table A.1: Result of engine 1

Engine 1	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**
Lift 1	38g	82g	135g	203g	240g	288g	314g
Lift 2	37g	81g	136g	203g	240g	286g	315g
Lift 3	38g	80g	137g	205g	242g	283g	315g
Lift 4	36g	82g	135g	201g	248g	282g	314g
Average	37,25g	81,25g	135,75g	204g	240g	284,75g	314,5g
Highest/lowest diff.	5.5%	2.5%	1.5%	2%	1.7%	2.1%	0.3%

Table A.2: Result of engine 2

Engine 2	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**
Lift 1	36g	89g	156g	210g	274g	303g	326g
Lift 2	38g	91g	153g	213g	273g	299g	325g
Lift 3	39g	89g	153g	214g	272g	300g	324g
Lift 4	38g	89g	151g	215g	270g	296g	320g
Average	37,75g	89,5g	153,25g	213g	272,25g	299,5g	314,5g
Highest/lowest diff.	8.3%	2.2%	3.3%	2.4%	1.5%	2.4%	1.9%

Table A.3: Result of engine 3

Engine 3	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**
Lift 1	40g	83g	157g	230g	283g	322g	343g
Lift 2	40g	82g	156g	227g	281g	322g	344g
Lift 3	39g	84g	155g	230g	280g	320g	340g
Lift 4	39g	83g	154g	228g	281g	318g	337g
Average	39,5g	83g	155,5g	228,75g	281,25g	299,5g	323,75g
Highest/lowest diff.	2.6%	2.4%	1.9%	1.3%	1.1%	1.3%	2.1%

Table A.4: Result of engine 4

Engine 4	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**
Lift 1	39g	85g	146g	217g	269g	312g	332g
Lift 2	39g	85g	149g	216g	268g	308g	329g
Lift 3	42g	84g	149g	216g	271g	308g	328g
Lift 4	42g	83g	148g	213g	272g	305g	324g
Average	40,5g	84,25g	148g	215,5g	270g	308,25g	341g
Highest/lowest diff.	7.7%	2.4%	2.0%	1.9%	1.5%	2.3%	2.5%

Table A.5: Total result

Total	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**
Average	38,75g	84,5g	148,13g	215,06g	265,88g	303,25g	332g
Min	36g	80g	135g	201g	238g	282g	314g
Max	42g	91g	157g	230g	283g	322g	344g
Max diff.	6g	11g	22g	29g	45g	40g	30g
Max diff.(%)	16,67%	13,75%	16,30%	14,43%	18,91%	14,18%	9,55%

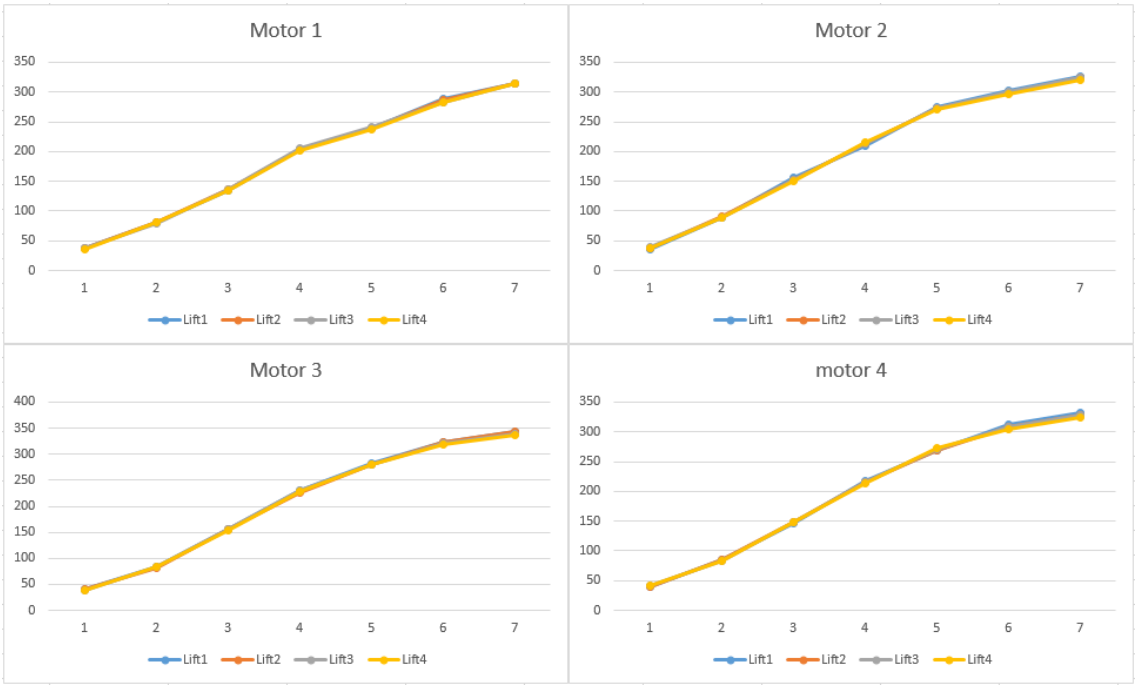


Figure A.3: The four individual motor lifts.

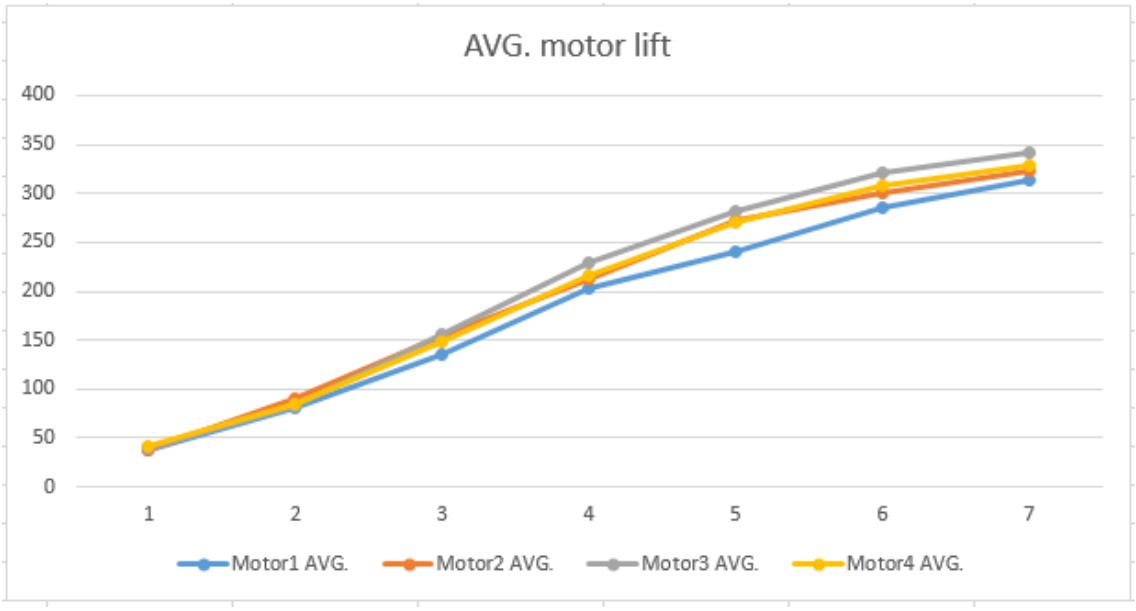
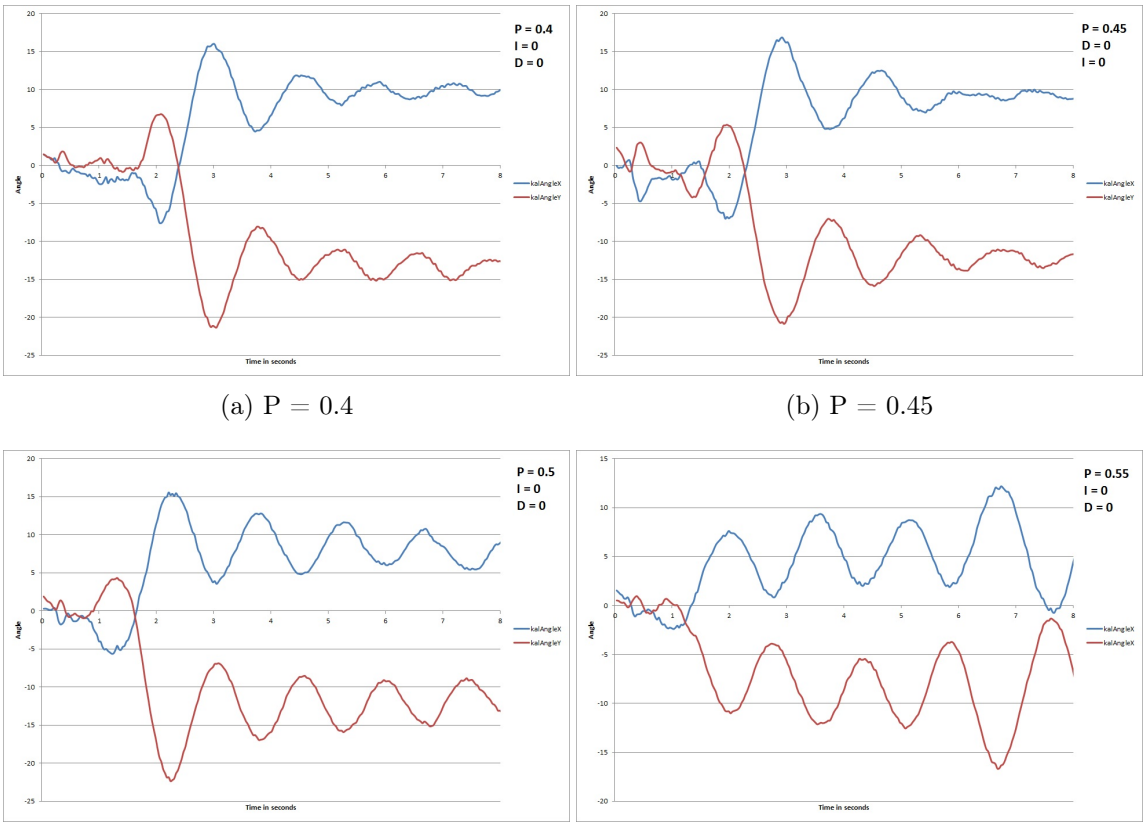


Figure A.4: The four motor sets average lifts.

A.5 PID tuning data



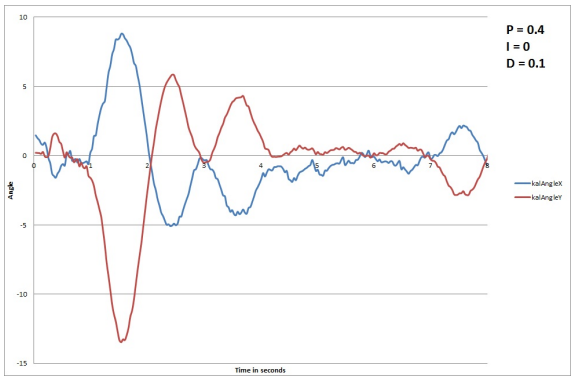
(a) P = 0.4

(b) P = 0.45

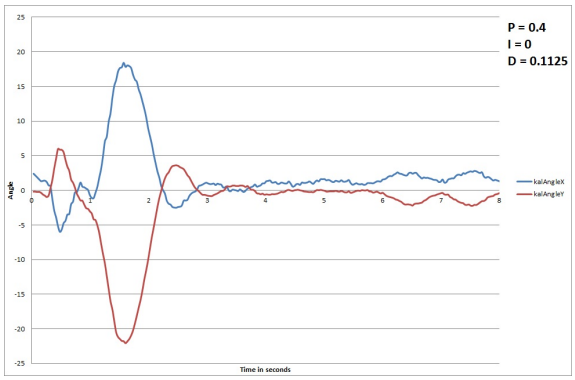
(c) P = 0.5

(d) P = 0.55

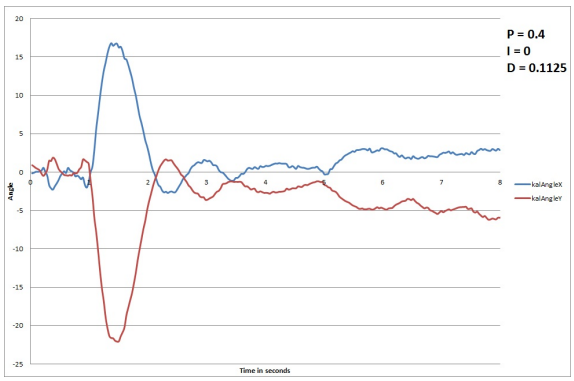
Figure A.5: I = 0 D = 0



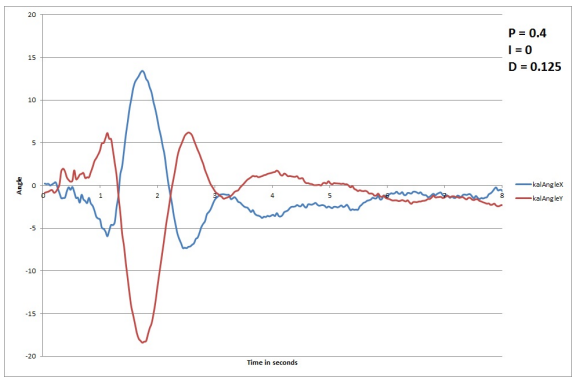
(a) D = 0.1



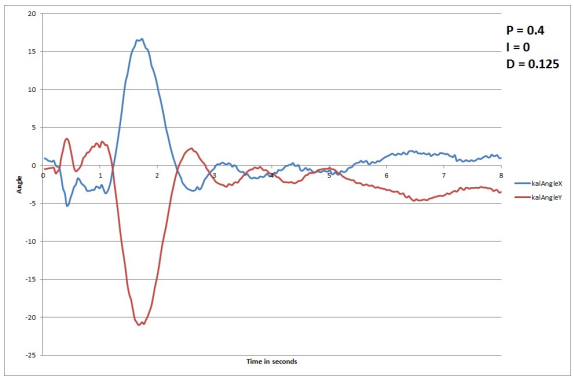
(b) D = 0.1125



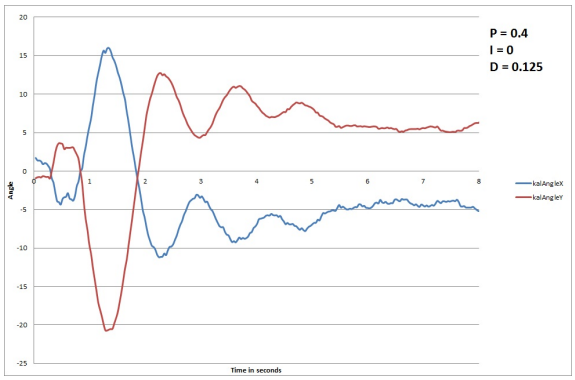
(c) D = 0.1125



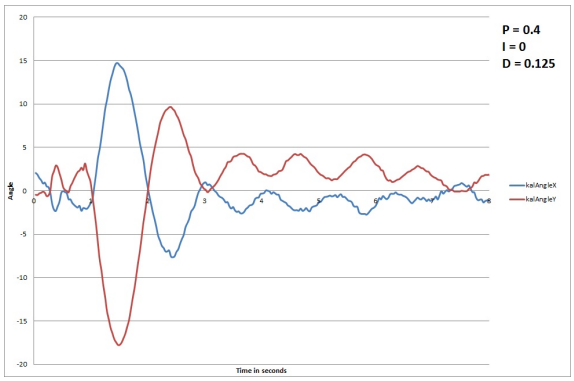
(d) D = 0.125



(e) D = 0.125

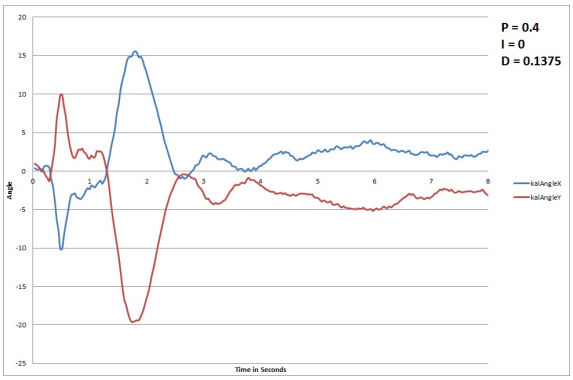


(f) D = 0.125

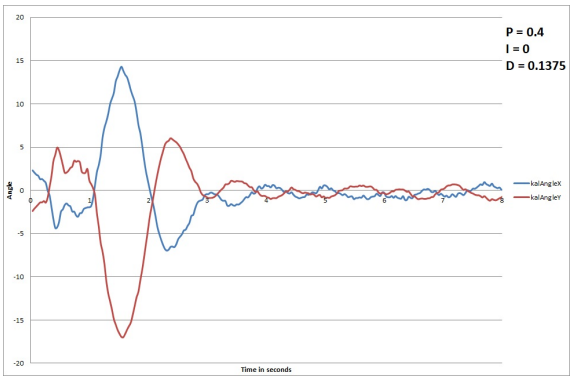


(g) D = 0.125

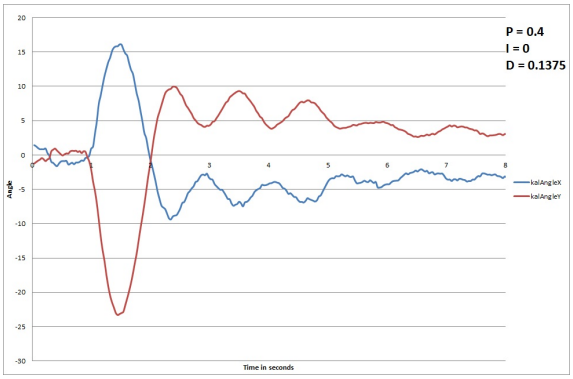
Figure A.6: P = 0.4 I = 0



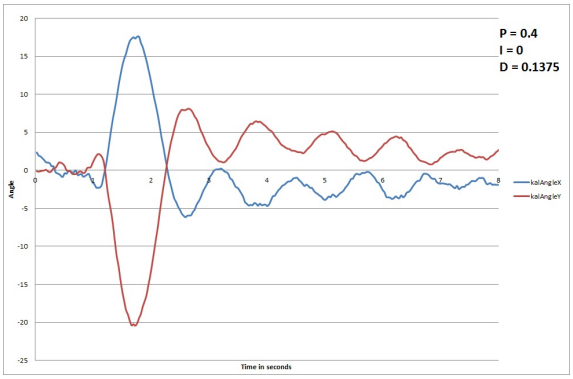
(a) D = 0.1375



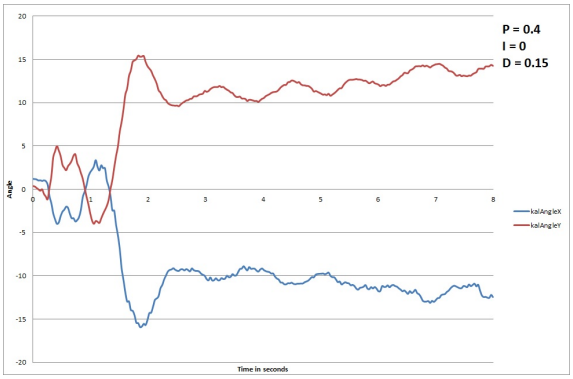
(b) D = 0.1375



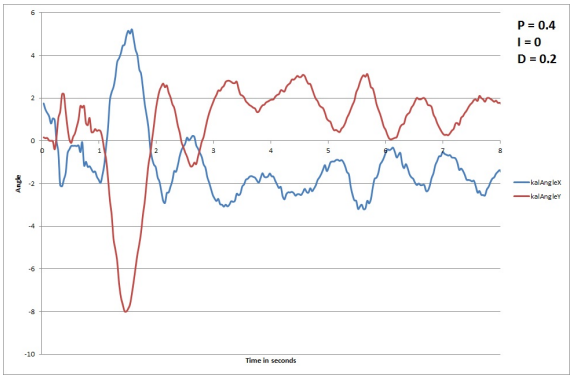
(c) D = 0.1375



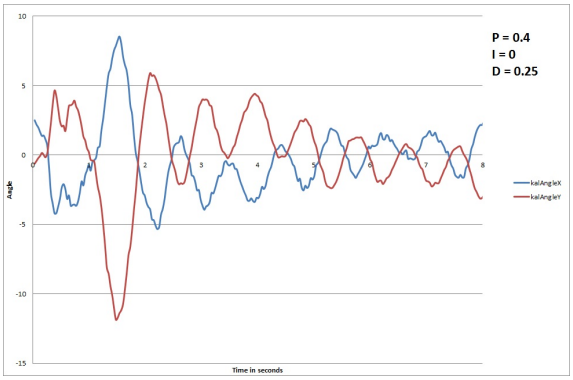
(d) D = 0.1375



(e) D = 0.15

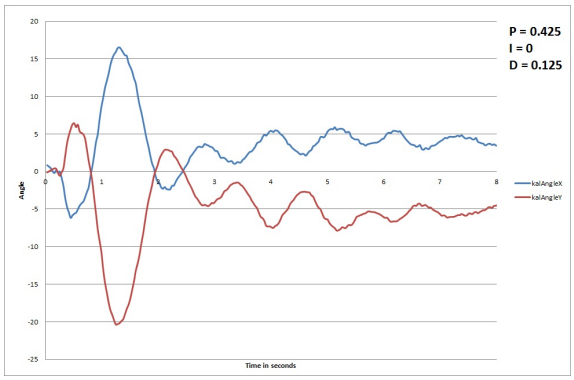


(f) D = 0.2

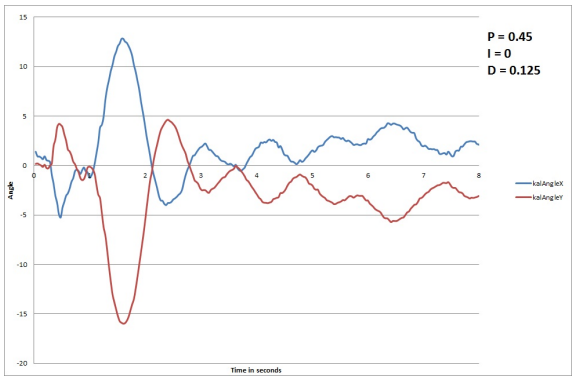


(g) D = 0.25

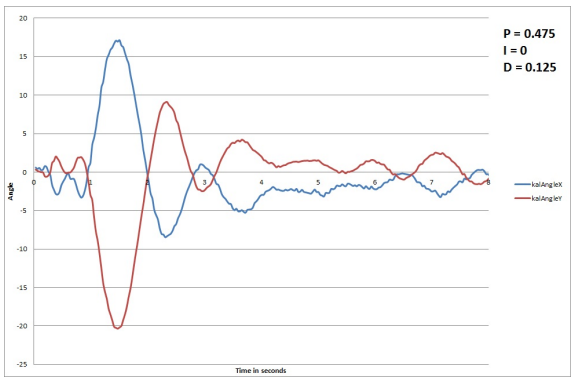
Figure A.7: P = 0.4 I = 0



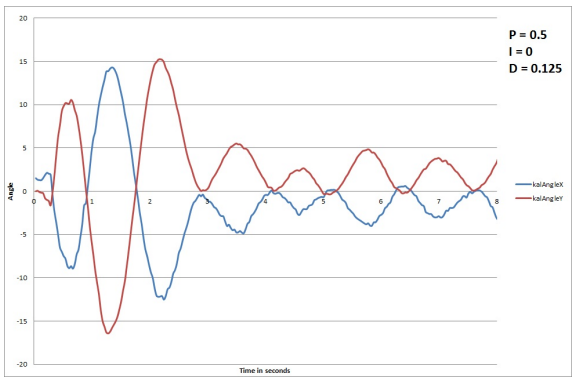
(a) P = 0.425



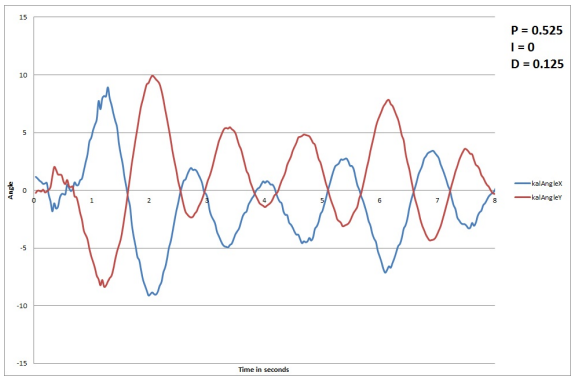
(b) P = 0.45



(c) P = 0.475

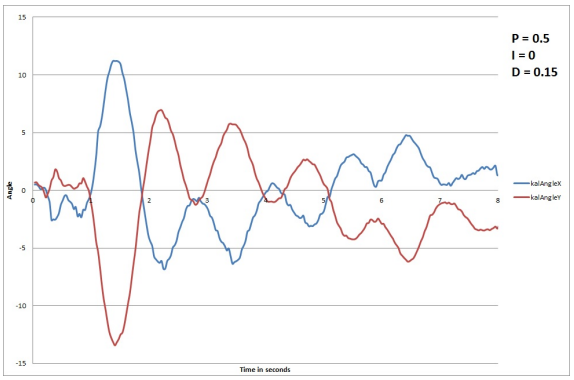


(d) P = 0.5

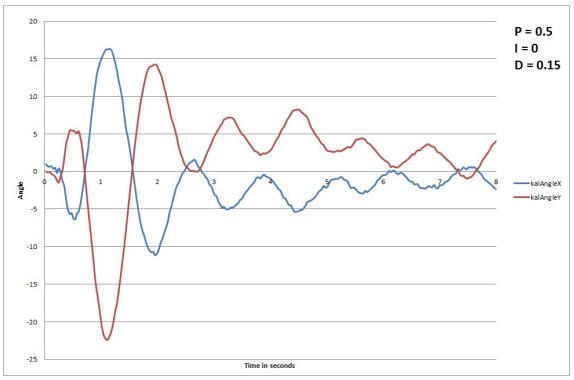


(e) P = 0.525

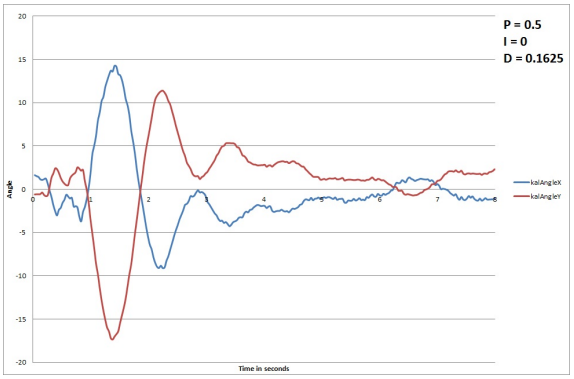
Figure A.8: I = 0 D = 0.125



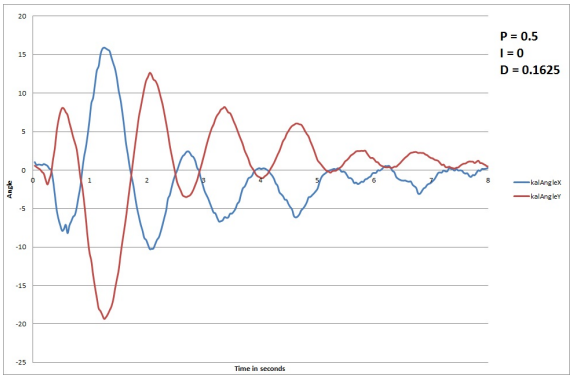
(a) D = 0.15



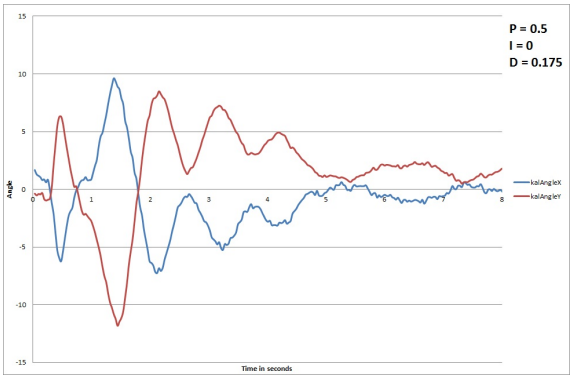
(b) D = 0.15



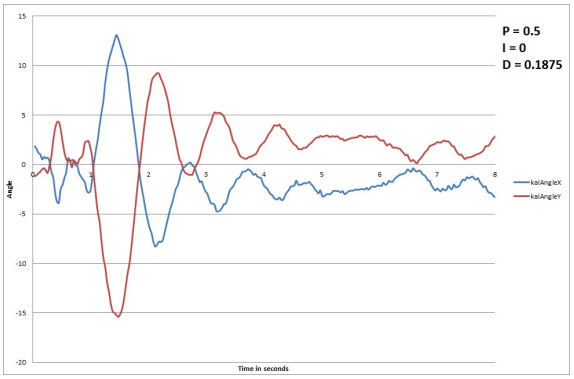
(c) D = 0.1625



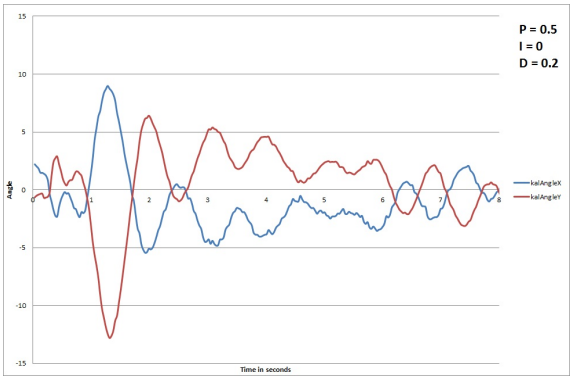
(d) D = 0.1625



(e) D = 0.175

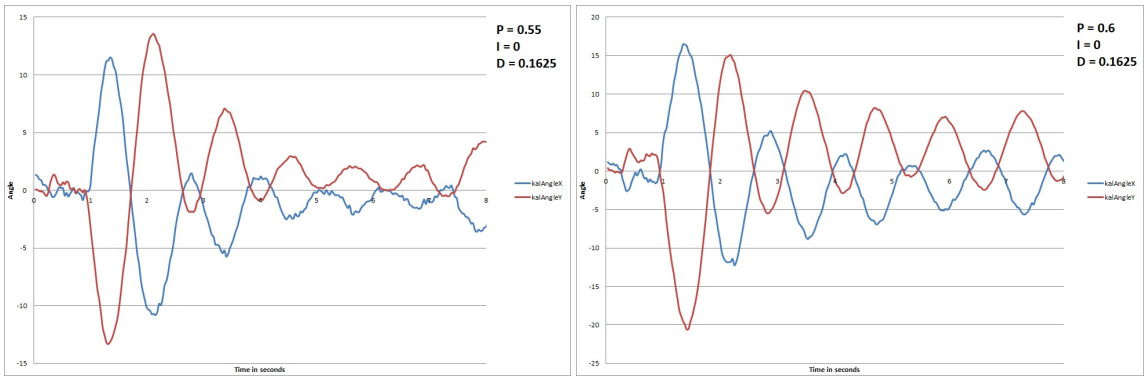


(f) D = 0.1875



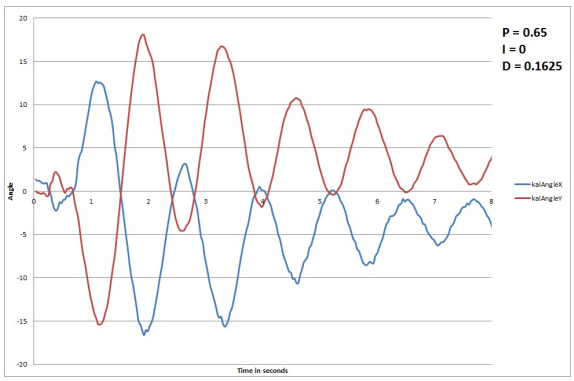
(g) D = 0.2

Figure A.9: P = 0.5 I = 0

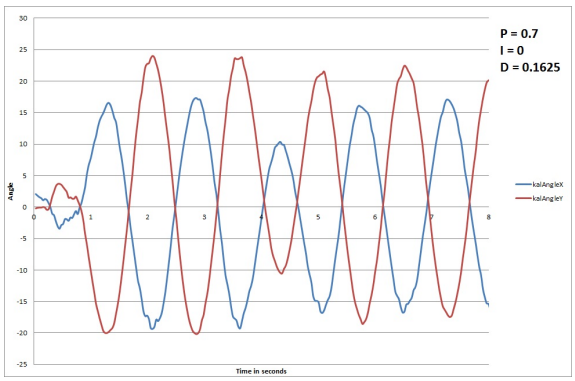


(a) P = 0.55

(b) P = 0.6

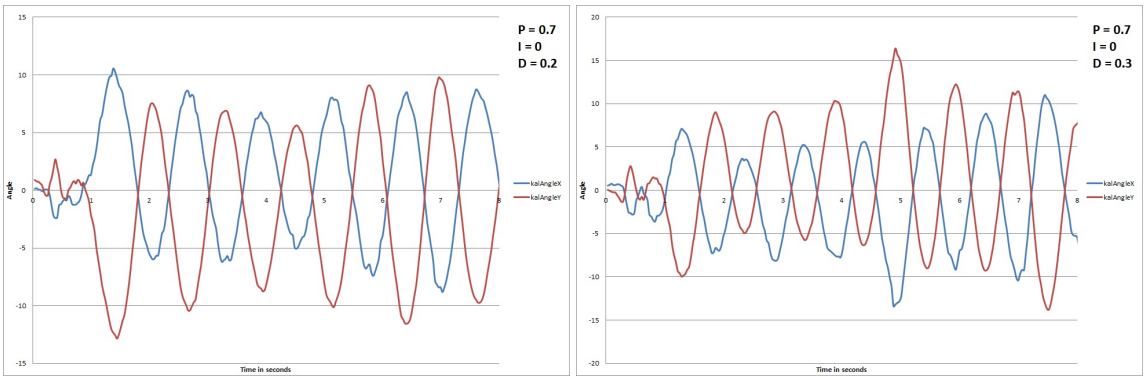


(c) P = 0.65



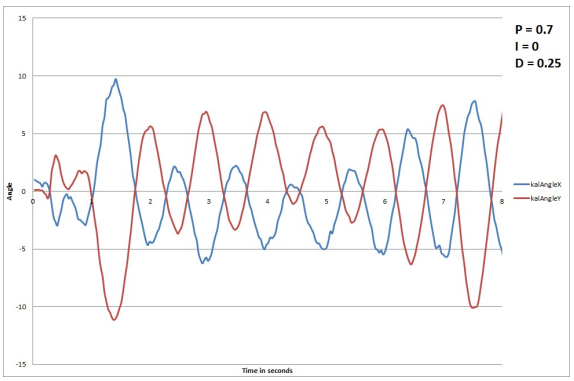
(d) P = 0.7

Figure A.10: I = 0 D = 0.1625

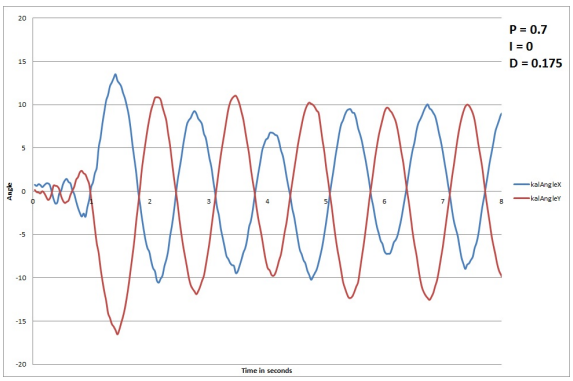


(a) D = 0.2

(b) D = 0.3

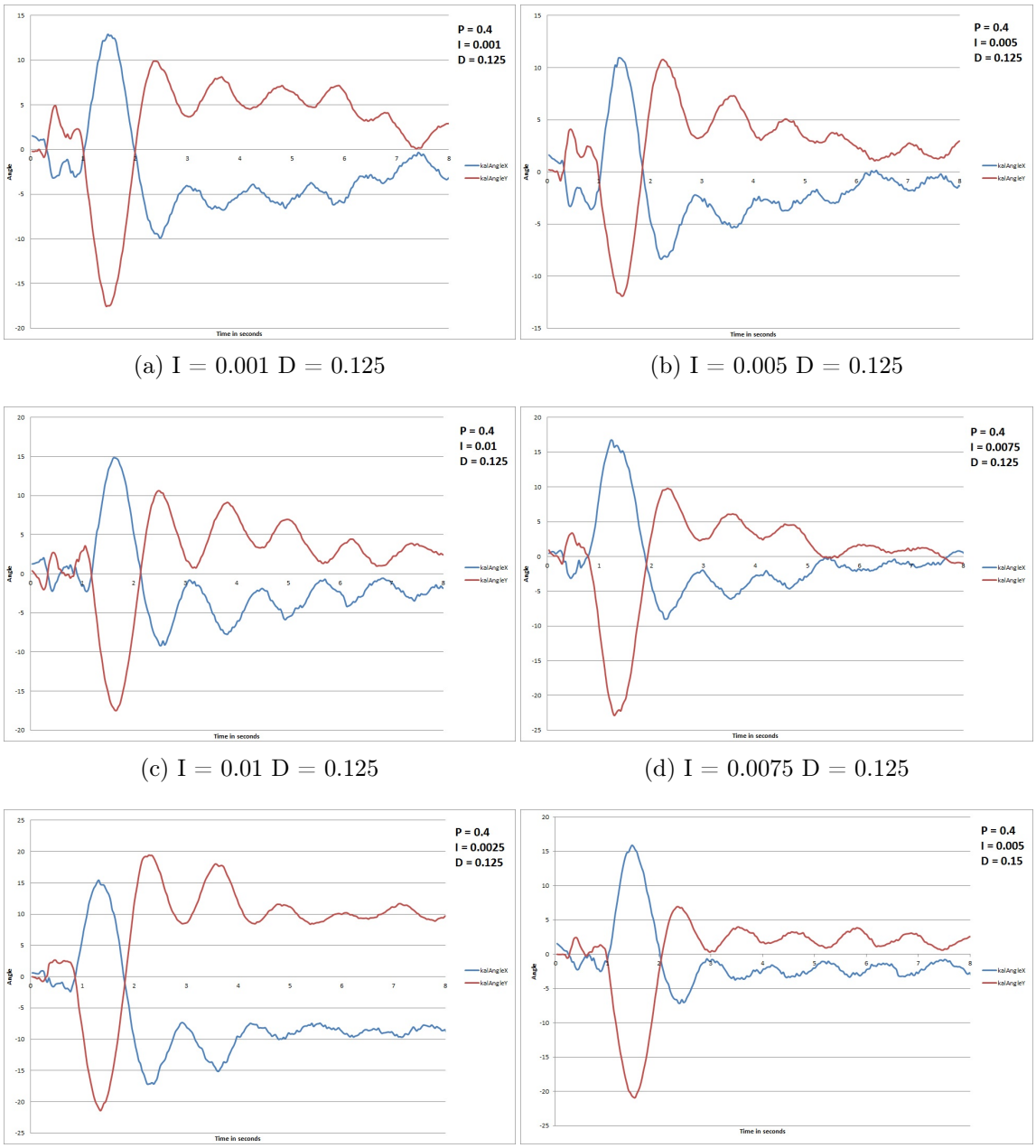


(c) D = 0.25



(d) D = 0.175

Figure A.11: P = 0.7 I = 0



(a) I = 0.001 D = 0.125

(b) I = 0.005 D = 0.125

(c) I = 0.01 D = 0.125

(d) I = 0.0075 D = 0.125

(e) I = 0.0025 D = 0.125

(f) I = 0.005 D = 0.15

Figure A.12: P = 0.4

Task two B

B.1 Testing of HC-SR04 sensors

This section will contain information on how the different tests with the HC-SR04 sensors were done. To be able to know which sensors were being tested, they were each marked with a number, which is referred to in the tables created from the test data.

```
#include <NewPing.h>

#define SONAR_NUM 2
#define MAX_DIST 400
#define PING_INTERVAL 100

unsigned int cm[SONAR_NUM]; // array to hold the different measurements for each sensor

String dataSend = "";

NewPing sonar[SONAR_NUM] = { //Initializing the sensors
  /* Setting the pins for which the Trigger and Echo pin on the sensors
  are connected with the Arduino's, and the max distance the sensors should work at */
  NewPing(13, 12, MAX_DIST),
  NewPing(11, 10, MAX_DIST)
};

void setup() {
    Serial.begin(9600);
}

void loop() {
  //Emptying the "dataSend"-string for earlier sensor-measurements
  dataSend = "";
  //for-loop going through every sensor in the sonar array
  for(uint8_t i = 0; i < SONAR_NUM; i++){
    //Pinging sensor number "i" and saving the distance measured in cm[i]
    cm[i] = (sonar[i].ping() / US_ROUNDTRIP_CM);

    //Appending the measurement of the sensor onto "dataSend"
    dataSend += String(cm[i]);
    //Making space between the measurements
    dataSend += ", ";
    //Delay used to make sure that there is no interference between measurements
    delay(PING_INTERVAL);
    //delayMicroseconds(PING_INTERVAL);
  }
  //Printing all the measurements gathered and appended in "dataSend" to the serial port.
```

```
39    Serial.println(dataSend);
40  }
```

Example code B.1: The primary Arduino code used for testing the HCSR04 sensors

The Arduino code used for testing the sensors can be seen in listing B.1. This code were changed a little based on which test were to be performed and how many sensors were needed for that particular test.

Precision

For measuring the max distance and the precision of the sensors, it was chosen the minimum range tested would be 20 cm, and not the minimum of the sensor. This is because of the size of the quadcopter, where measurements at 2 cm can not be used, since the distance from where the sensors are placed and out beyond the quadcopter's hardware is longer than 2 cm. Additionally, distance markers of 50 cm, 100 cm, 105 cm, 150 cm, 155 cm, 200 cm, 300 cm and 400 cm were chosen to be tested. For each sensor an flat object were placed at each marked up distance, and the distance-data from the sensor was noted. The flat object was the same used in the angle of detection test. If the object could not be detected at the 400cm mark, the object would be moved closer to the sensor until it was detected and the distance was noted. The data from this test can be seen in appendix section B.2

Angle of detection

To test at which angles the sensors detect an object, the detection angle's center had to be determined. As it is seen on figure B.1 the sensors have two heads from where the signal is transmitted/received, and since it is not stated whether the angle center of the detection range is between the two parts, or from the middle of both of them, a test to figure out which one of these propositions is true is needed. If it is known where the angle is centered, this can help determine how close sensors can be to each other, and at which angle they should be positioned, for optimal coverage.

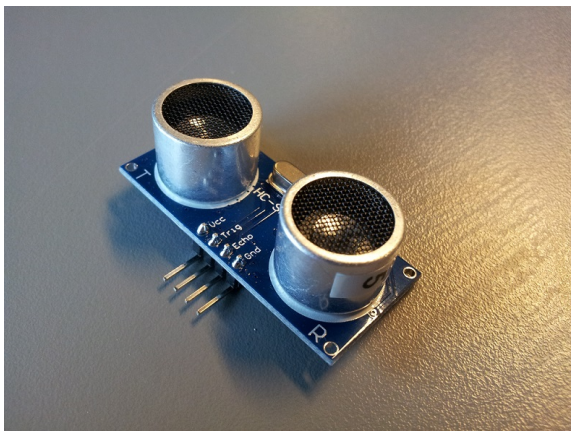


Figure B.1: Picture of a HC-SR04 sensor

For figuring out the angle's center point, the sensor being tested was taped to a piece of

paper and connected to the Arduino. The middle between the sensor heads was marked, and from that mark, lines that were five degrees apart were drawn, until 40 degree coverage was achieved on each side, which makes the total test coverage area an 80 degree cone. This line marking was also done with a new piece of paper that had two markings. A marking placed below the middle of each of the two heads that transmit/receive, again with lines drawn from these points outwards, with five degrees split between the lines, up to 40 degrees coverage from the middle of the sensor heads, to the side away from the other head. On each of the papers, a distance of 20 cm and 30 cm were drawn as an arch from the center markings, as seen on figure B.2, and they were used to make sure each sensor used for testing always tested at the same distance. When this was done, the testing began.

For each sensor, an object was slowly moved from the 40 degree line towards the centre on both the 20 cm and 30 cm arch. When the sensor registered the object, the degree it was registered at was noted. This test was performed with both papers and all of the sensors, to first figure out, as mentioned, if the angle of detection is measured from the middle between the sensor heads, or the middle of each head of the sender/receiver.

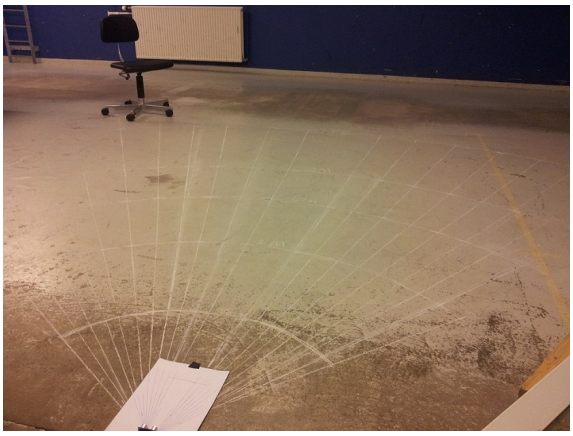


Figure B.2: Setup for testing angle of detection

After figuring out the angle of detection's center placement, the setup was scaled up. The distance of 20 cm and 30 cm is not enough to clearly conclude at which angle each of the sensors can detect an object. This is also because of the quadcopter, as it has to be able to detect object further away than only 30 cm. Therefore an upscale was done, where the angle of detection was, additionally, measured at the distances 0.5 m, 1 m, 1.5 m, 2 m and 3 m.

To try and imitate real world situations, both a square and a round object were used, as the detection unit that had to be detected by the sensors, to figure out how limited the sensors were based on the type of object. When testing a square object, the angle of the detection surface to the sensor had to be adjusted to allow for a more direct return of the sensor's detection waves, in order to fully determine the limitations of each sensor. When testing with a round and uniform object, the detection limitations were tested without moving the object, unlike the square object's angle to the sensors. The data from this test can be seen in appendix section B.2

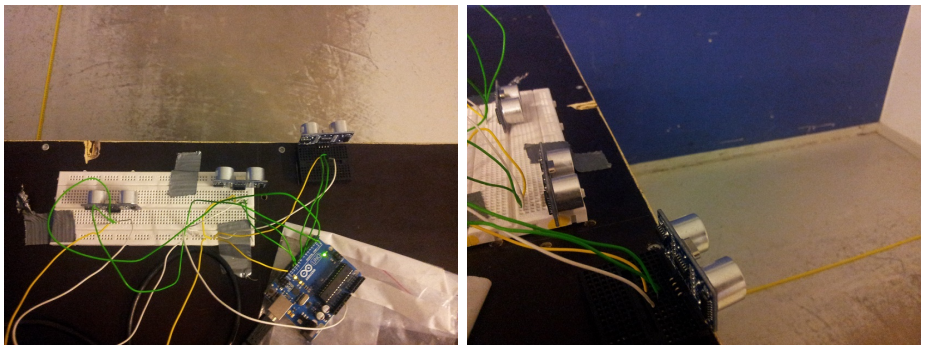


Figure B.4: Setup for testing interference at 1.5 m and 2 m

Interference

In order to test the interference between the sensors and at what delay it could be avoided, a setup where made where two sensors were placed close together on an object above the ground, as some sensors might be on the quadcopter, with a different distance to the same wall. By looking at the data the sensors returned and adjusting the delay between each sensor ping, see `PING_INTERVAL` in listing B.1, it was possible to see when and with what minimum delay between pings that there were no more interference.

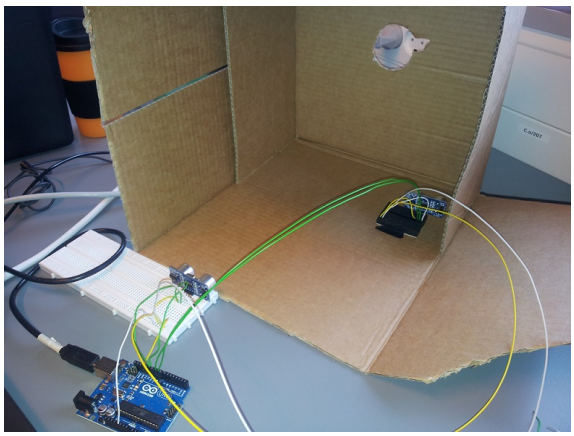


Figure B.3: Setup for testing if interference can occur

This was done three times, first as a test to see within close range, at 30 cm, when interference would occur, second to see at the concluded distance from the earlier test, at 1.50 m and third at 2 m. In the first test at 30 cm, one of the sensors were placed roughly 20 cm closer to the wall. This was done to make it easier to look at the returned data, and see when the data did not match the actual distance. At the second and third test the sensors difference to the wall were only 5 cm to closer match how the sensors will be on the quadcopter. The sensors were also placed above ground in the second and third test, this were done again to match how the sensors will operate on the quadcopter, and to make sure the floor does not have any influence on the test.

The data from the last two tests were saved and used to determine with which delay the variation in distance were accurate and there were no wrong data being outputted by the sensors. To help visualize this, graphs were made, which can be seen in appendix section

B.2. Not all tested delays have been made into a graph, as it was deemed unnecessary to include more than a few around the delay, where no interference is detected. Under the first test it was discovered that the sensors can did not handle angled walls and corners well, and that it was necessary to further test this.

B.1.1 Measurement time

To make sure the measuring time of the HC-SR04 sensors is known, a test were setup with a single sensor at a time, pointed at a wall with a distance of 1.5 m, since this were the best results of the precision test, while still giving a good distance. The reason the test only is done at 1.5 m is because it is enough to only know the worst-case measuring time of the sensors.

The test were done with all the sensors, to find the worst-case for all of the sensors, to make sure that when scheduling the different tasks the quadcopter got, that the time for making a measurement by the HC-SR04 sensor is know. The 20 worst-case measuring times of each of the HC-SR04 sensors, out of at least 300 measuring times per sensor, can be seen in table B.3.

B.1.2 Angled walls

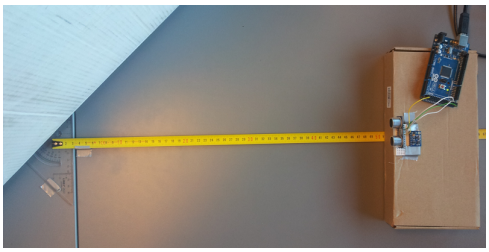
As stated in section 17.2 it was discovered that the sensors have difficulty in measuring distance when pointed at angled walls and corners. This section will test and document this further.

To test this several setups where made to test the sensors when pointed at an angled wall and different corners, see figure B.5 with different angels, and from different distances. This were all done to make sure that it is know how the sensors will react to these elements in the environment.

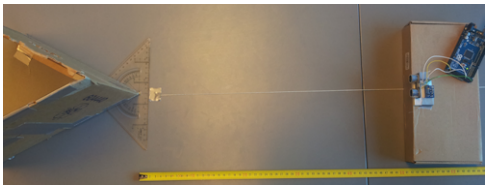
For the situation where the sensor would be pointed at one long wall at an angle, the setup in figure B.5a where made. The sensor were placed at different distances from the wall, and the wall were changed to have different angles compared to the sensor. Some of the results from this test can be seen in figure B.8, B.9 and B.10, showing results with the wall at three different angles, 25, 45 and 65 degrees, and two distances of 50 and 70 centimeters.

For the situation where the sensor is pointed at a sharp corner, or edge, a piece of cardboard where bend into two different angles and the sensor where placed at different distances from it, see figure B.5b. Some of the results from this test can be seen in figure B.11 and B.12.

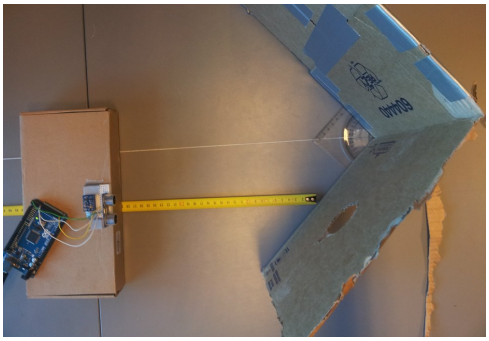
As a last test of the interaction between the sensors and angled walls, a setup where made, which can be seen on figure B.5c and B.5d, that would show how the sensors handles a 90 degree open corner. The sensor is pointed at the wall with a 45 degree angle, to see how the sensors handles this kind of corners, and to see what measurements get returned. Some of the results from this test can be seen in graph B.13a and B.13b.



(a) Single angled wall



(b) Sharp angled corner



(c) Angled cardboard corner

(d) Angled wall corner

Figure B.5: Different testing setups for testing Sensors interaction with angled walls

B.1.3 Source of error

The different sources of errors that could have impacted the tests will be listed in this section.

- **Human error** is the biggest potential source of error when testing the HC-SR04 sensors. Each sensor had to be manually placed by hand, and measurements had to be done by hand as well, so the precision of the data could be compromised to some degree.
  - **Sensor placement** could be compromised, as the setup had to be adjusted for each sensor, so their direction may not be equal across all tests.
  - **Measurement line drawings** were drawn by hand with a solid stick as support, but the accuracy of the measurement lines could also be compromised.
  - **Detection object placement** had to be placed by hand as well, and since the detection angle at a far away range needed an almost perfect return angle on the detection object, the placement by hand could compromise the specifications of each sensor, as the placement was not adjusted to a change in degrees until a detection occured.
- **Test setup** is a possible source of error, because the test possibly was not designed very well to accurately determine the specifications of each of the sensors tested.
- **Wrong detection** is a possible source of error, where the detection did not occur with the specific object used for testing. This source of error is however very slight to non-existent.

All these potential sources of errors were taken into account when considering the accuracy of the data, and it was decided that the setup and test, used to determine the specifications of each of the sensors, was sufficient enough to determine relative specification of each sensor, to use the sensors in conjunction with the quadcopter to complete the task 3 system.

B.2 Data from testing HC-SR04

Precision

Distance (cm)	20	50	100	105	150	155	200	300	400	Max
Sensors										
1	19	50	101	106	152	157	203	306	x	341
2	19	50	102	106	152	157	203	306	410	
3	20	50	101	105	151	156	202	306	409	
4	20	50	101	105	152	156	203	305	410	
5	20	51	102	106	152	157	203	306	410	
6	21	51	101	106	152	157	204	306	411	
7	20	50	101	106	152	156	204	307	x	362
8	21	50	103	107	152	157	204	306	x	381
9	20	50	101	106	152	157	203	307	x	374
10	19	50	102	105	151	154	203	306	x	370
11	19	50	103	106	152	157	205	306	x	397

Table B.1: Results from distance testing

Variation (cm)	20	50	100	105	150	155	200	300	400 or Max
Sensors									
1	-1	0	1	1	2	2	3	6	8
2	-1	0	2	1	2	2	3	6	10
3	0	0	1	0	1	1	2	6	9
4	0	0	1	0	2	1	3	5	10
5	0	1	2	1	2	2	3	6	10
6	1	1	1	1	2	2	4	6	11
7	0	0	1	1	2	1	4	7	8
8	1	0	3	2	2	2	4	6	9
9	0	0	1	1	2	2	3	7	9
10	-1	0	2	0	1	-1	3	6	8
11	-1	0	3	1	2	2	5	6	9

Table B.2: Variation from measured distance to actual distance

Measuring time

Sensors / **μSec.**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1	9100	9096	9096	9080	9112	9096	9076	9104	9080	9096	9104
2	9100	9092	9088	9064	9108	9088	9076	9104	9056	9092	9088
3	9080	9092	9088	9060	9108	9080	9076	9104	9056	9092	9084
4	9076	9072	9088	9060	9104	9080	9076	9104	9052	9092	9068
5	9076	9072	9068	9056	9084	9076	9076	9100	9052	9088	9068
6	9072	9072	9064	9056	9072	9076	9072	9100	9052	9088	9068
7	9072	9072	9064	9056	9064	9076	9072	9100	9052	9088	9064
8	9072	9068	9064	9056	9064	9076	9072	9100	9052	9088	9064
9	9072	9068	9064	9056	9064	9076	9072	9088	9052	9088	9064
10	9056	9068	9064	9052	9064	9076	9072	9088	9052	9088	9064
11	9056	9068	9064	9036	9060	9076	9072	9088	9052	9084	9064
12	9056	9068	9064	9036	9060	9076	9072	9088	9052	9084	9064
13	9052	9068	9064	9036	9060	9076	9072	9088	9052	9064	9064
14	9052	9068	9064	9036	9060	9076	9068	9088	9052	9064	9064
15	9052	9068	9064	9032	9060	9076	9068	9088	9052	9064	9064
16	9052	9068	9064	9032	9060	9076	9068	9088	9052	9064	9064
17	9052	9068	9064	9032	9060	9076	9052	9088	9032	9064	9064
18	9052	9068	9064	9032	9060	9076	9052	9088	9032	9064	9060
19	9052	9068	9064	9032	9060	9076	9052	9088	9032	9064	9060
20	9052	9068	9064	9032	9060	9072	9052	9088	9032	9060	9060

Table B.3: The 20 worst measuring times in microseconds at 1.5 meters. Taken out of 300 data points per sensor.

Angle of detection

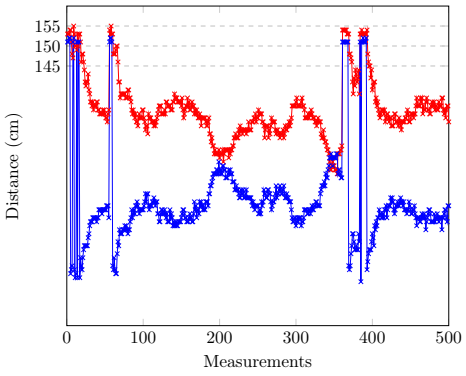
Distance (cm)	20		50		100		150		200		300	
Sensors	**left**	**right**	**l**	**r**	**l**	**r**	**l**	**r**	**l**	**r**	**l**	**r**
1	21°	22°	33°	26°	35°	32°	40°	37°	34°	31°	16°	20°
2	18°	21°	31°	29°	40°	41°	38°	35°	33°	30°	21°	20°
3	22°	29°	34°	35°	45°	45°	40°	40°	32°	35°	23°	23°
4	22°	22°	30°	30°	40°	40°	34°	35°	32°	30°	16°	16°
5	21°	21°	32°	30°	35°	33°	38°	35°	33°	31°	17°	16°
6	20°	21°	30°	30°	36°	40°	34°	33°	26°	32°	16°	20°
7	23°	20°	35°	28°	56°	42°	47°	34°	34°	28°	20°	16°
8	19°	15°	30°	25°	33°	27°	37°	31°	32°	25°	16°	15°
9	17°	25°	30°	34°	44°	47°	35°	38°	30°	32°	10°	16°
10	20°	20°	29°	30°	31°	31°	35°	36°	25°	29°	12°	10°
11	20°	21°	30°	30°	34°	32°	40°	38°	33°	32°	17°	22°

Table B.4: Angle test with **flat** object - Showing max angle in which object is detected

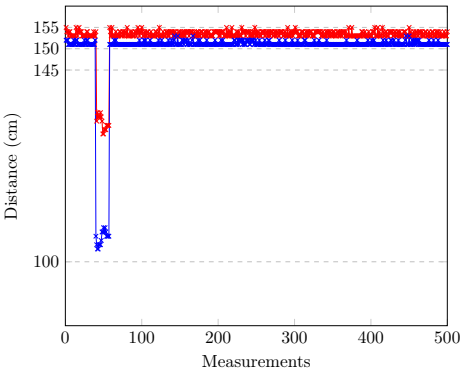
Distance (cm)	20		50		100		150		200		300	
Sensors	**left**	**right**	**l**	**r**	**l**	**r**	**l**	**r**	**l**	**r**	**l**	**r**
1	12°	14°	13°	13°	13°	13°	22°	18°	12°	12°	x	x
2	13°	16°	14°	17°	23°	22°	18°	20°	14°	12°	x	x
3	17°	21°	20°	22°	30°	31°	22°	23°	13°	14°	x	x
4	15°	17°	15°	18°	27°	26°	19°	21°	12°	10°	x	x
5	11°	12°	15°	14°	18°	13°	18°	15°	6°	8°	x	x
6	15°	15°	15°	18°	23°	27°	20°	19°	x	10°	x	x
7	14°	11°	17°	13°	28°	24°	22°	16°	12°	7°	x	x
8	11°	4°	12°	6°	6°	10°	16°	14°	4°	7°	x	x
9	9°	15°	12°	15°	25°	26°	15°	20°	11°	12°	x	x
10	7°	14°	10°	14°	16°	17°	14°	16°	0°	8°	x	x
11	10°	15°	12°	16°	11°	15°	19°	22°	11°	12°	x	x

Table B.5: Angle test with **round** object - Showing max angle in which object is detected

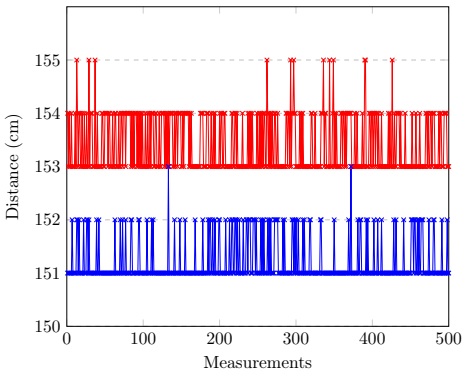
Interference



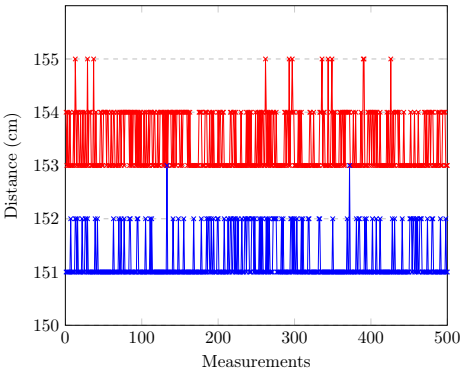
(a) 3 milliseconds delay



(b) 3.1 milliseconds delay

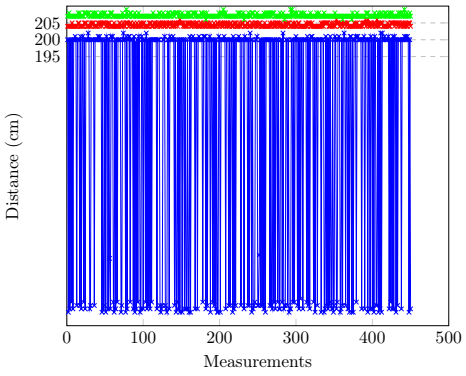


(c) 3.2 milliseconds delay

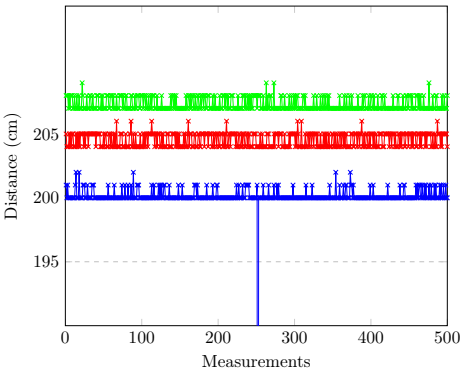


(d) 3.3 milliseconds delay

Figure B.6: Graphs showing the results from the interference test with two sensors, at 1.5 meters with different delays



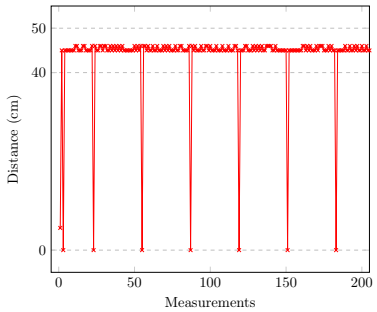
(a) 4 milliseconds delay



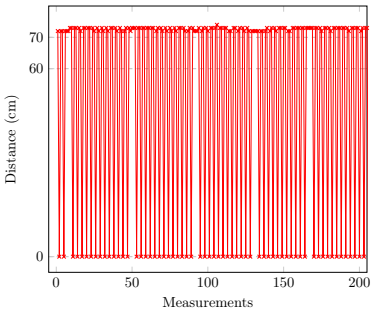
(b) 5.6 milliseconds delay

Figure B.7: Graphs showing the results from the interference test with three sensors, at 2 meters with different delays

Angled walls

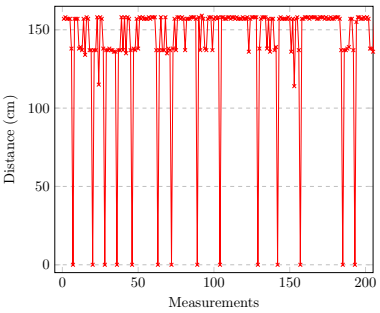


(a) 50 centimeters distance

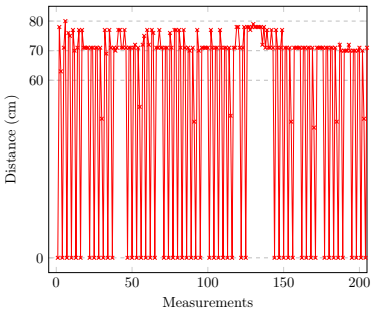


(b) 70 centimeters distance

Figure B.8: Data from the angled wall at 25 degrees with the sensor at different distances to the wall, see figure B.5a for the setup

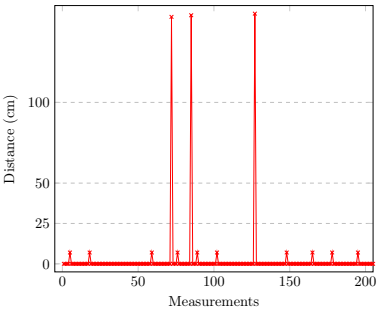


(a) 50 centimeters distance

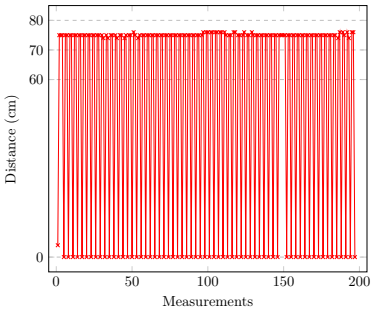


(b) 70 centimeters distance

Figure B.9: Data from the angled wall at 45 degrees with the sensor at different distances to the wall, see figure B.5a for the setup

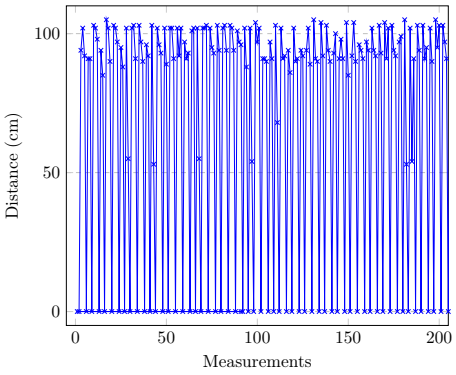


(a) 50 centimeters distance

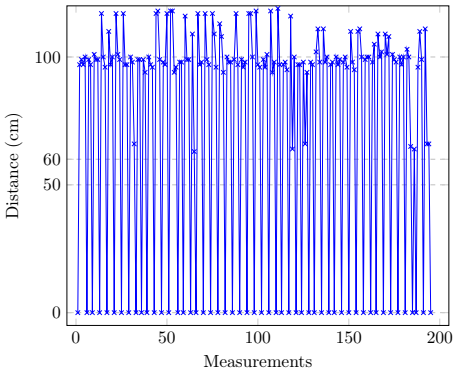


(b) 70 centimeters distance

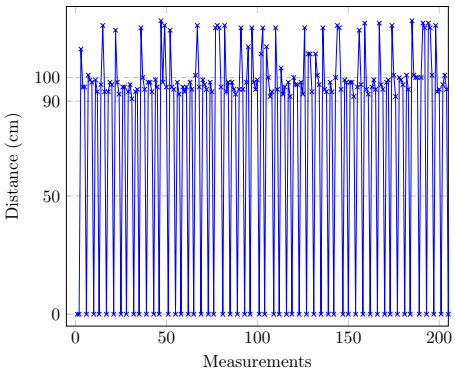
Figure B.10: Data from the angled wall at 65 degrees with the sensor at different distances to the wall, see figure B.5a for the setup



(a) 50 centimeters distance

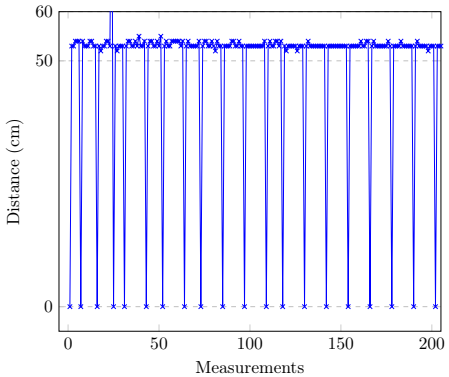


(b) 60 centimeters distance

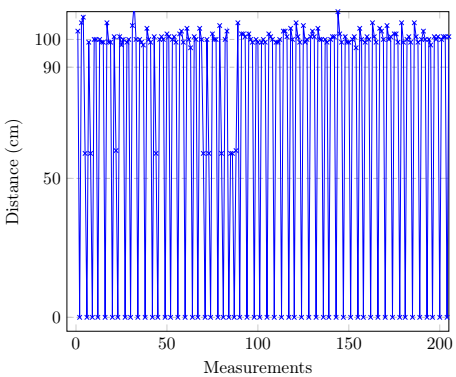


(c) 90 centimeters distance

Figure B.11: Data from the pointy cardboard corner at 60 degrees, with different distances to the wall, see figure B.5b for setup

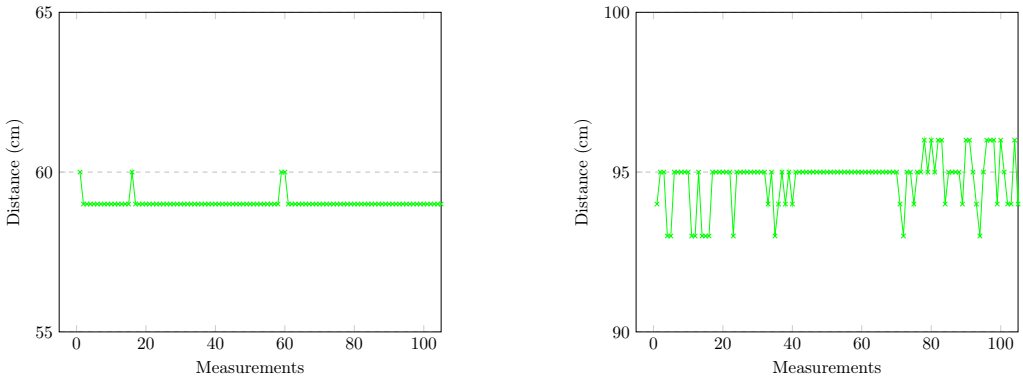


(a) 50 centimeters distance



(b) 100 centimeters distance

Figure B.12: Data from the pointy cardboard corner at 90 degrees with the sensors at different distances to the wall, see figure B.5b for the setup



(a) 35 centimeters distance

(b) 36 centimeters distance

Figure B.13: Data from the wall corner at 45 degrees with the sensor at different distances to the wall, see figure B.5c and B.5d for the setup.

Task three C

C.1 IMU distance determination

A test made, where the IMU is suppose to map movement of the IMU. In the test, the IMU is moved 40 cm forward, and then back to the start point, then a wait on about 5 minutes, and then the same movement again.

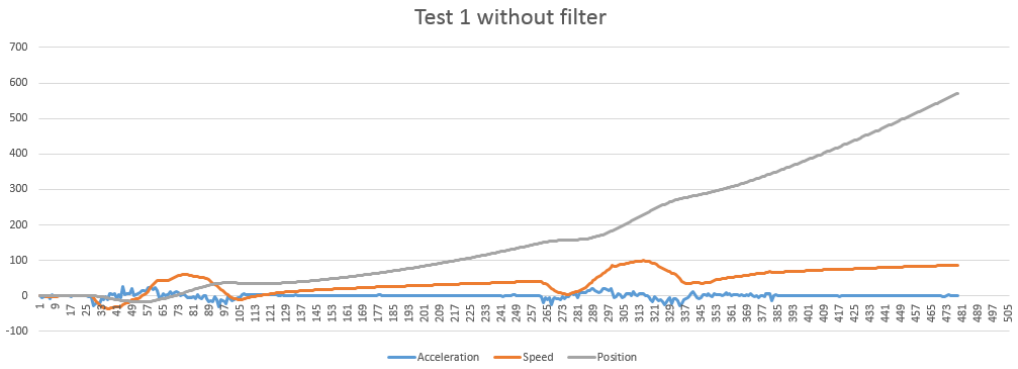


Figure C.1: Test result of moving the IMU forward and back to same starting point without filter

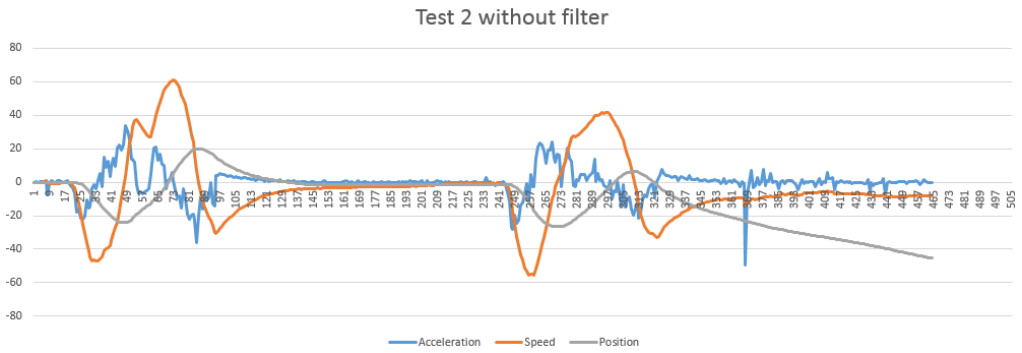


Figure C.2: Test result of moving the IMU forward and back to same starting point without filter

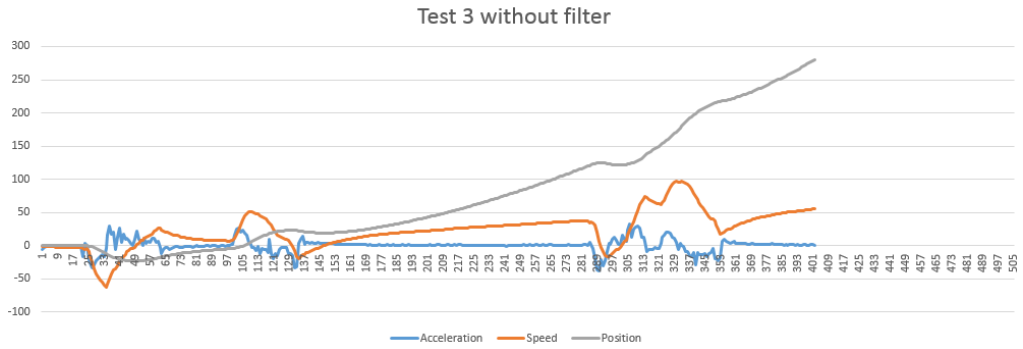


Figure C.3: Test result of moving the IMU forward and back to same starting point without filter

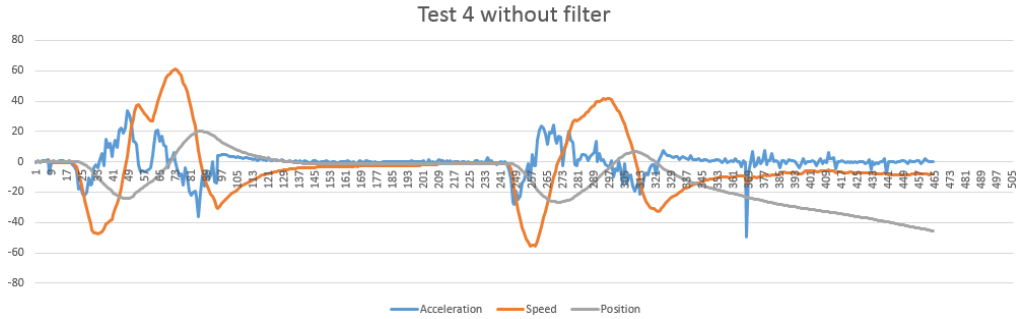


Figure C.4: Test result of moving the IMU forward and back to same starting point without filter

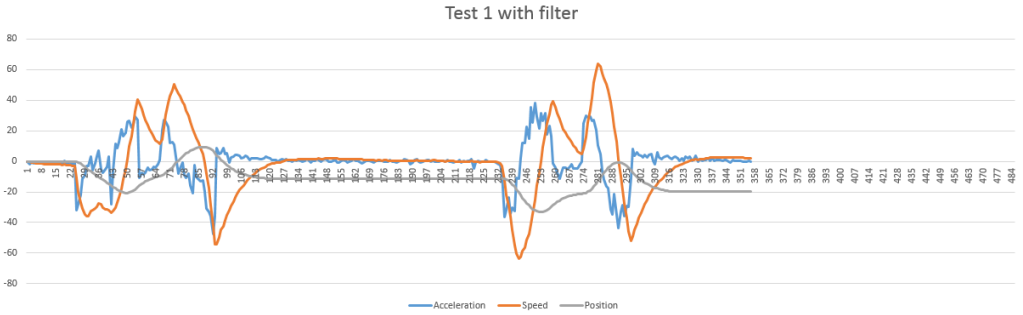


Figure C.5: Test result of moving the IMU forward and back to same starting point with filter

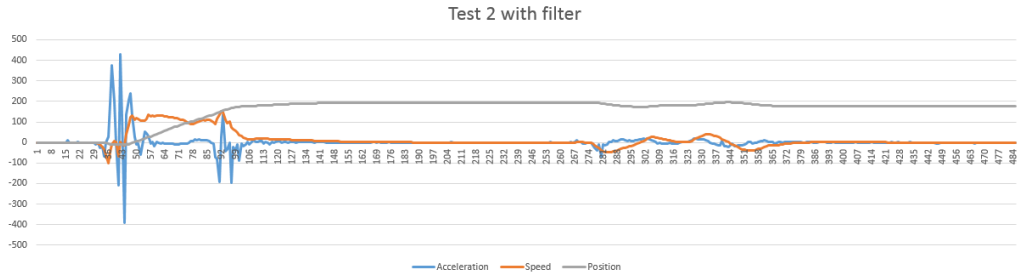


Figure C.6: Test result of moving the IMU forward and back to same starting point with filter

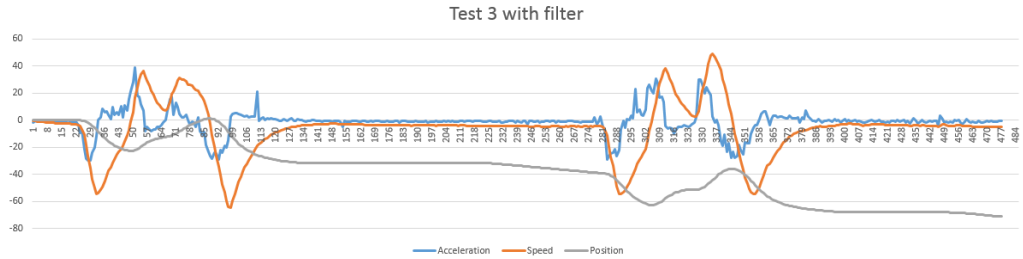


Figure C.7: Test result of moving the IMU forward and back to same starting point with filter

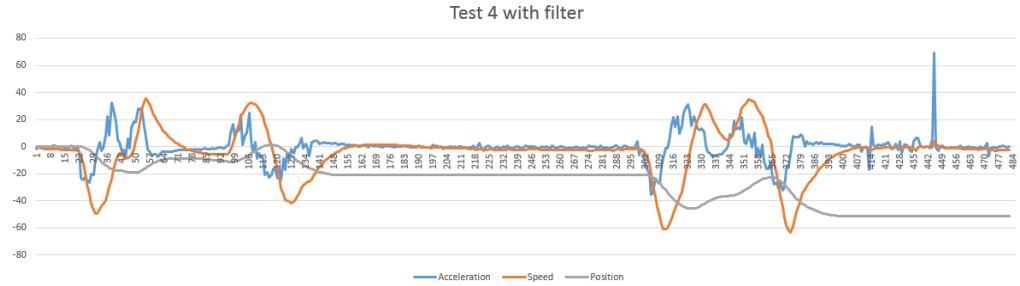
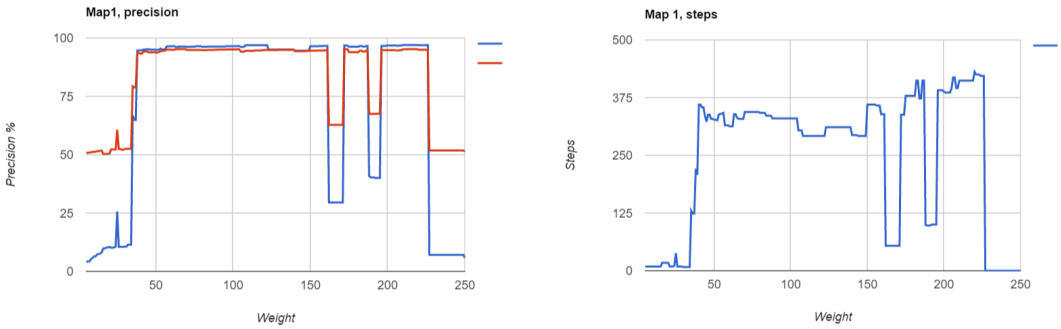


Figure C.8: Test result of moving the IMU forward and back to same starting point with filter

Task four D

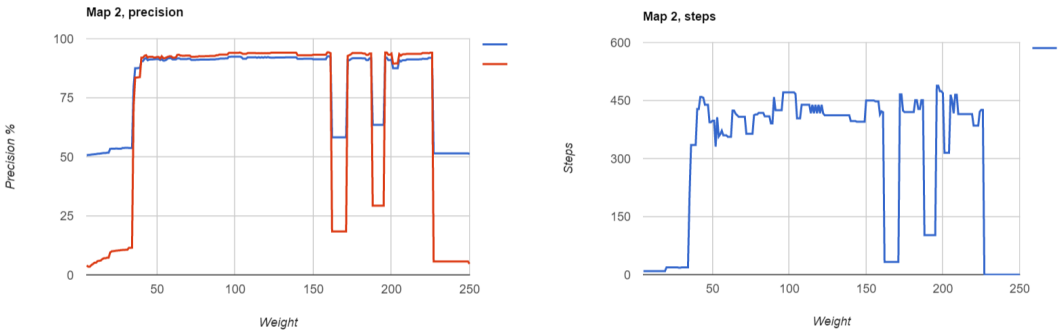
D.1 Weight test 1 data

D.1.1 Task 4, simulation test part 1



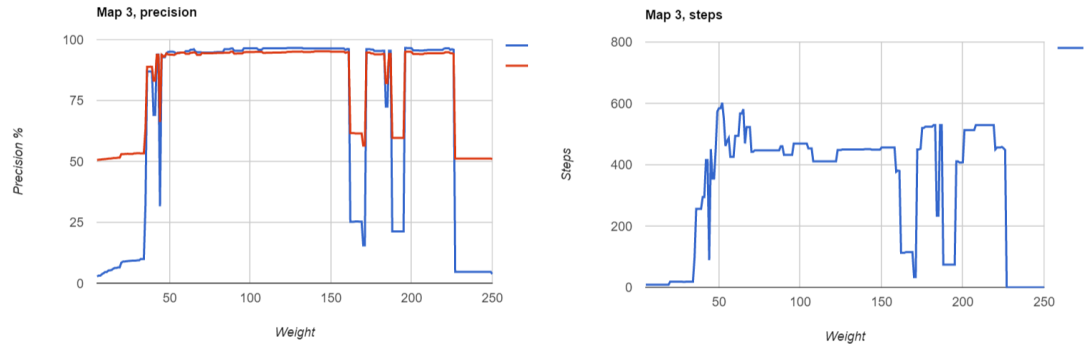
(a) Precision (b) Steps

Figure D.1: Map 1



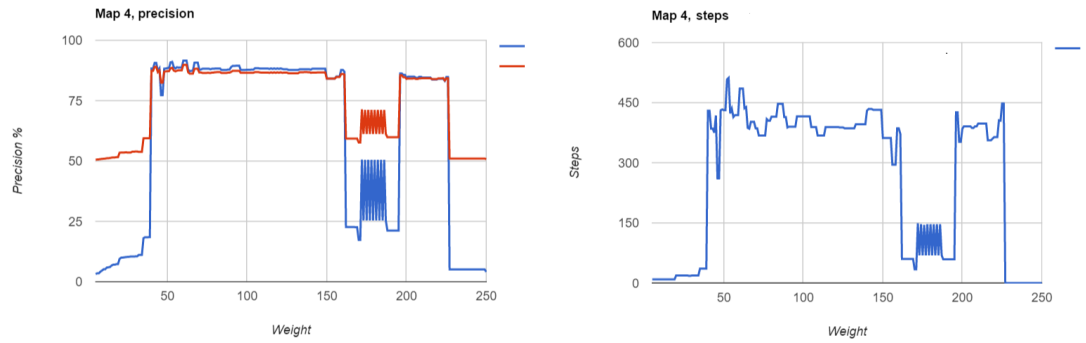
(a) Precision (b) Steps

Figure D.2: Map 2



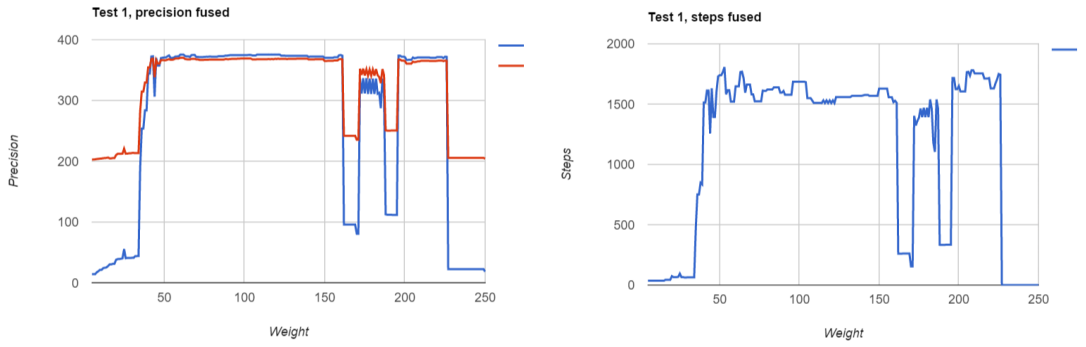
(a) Precision (b) Steps

Figure D.3: Map 3



(a) Precision (b) Steps

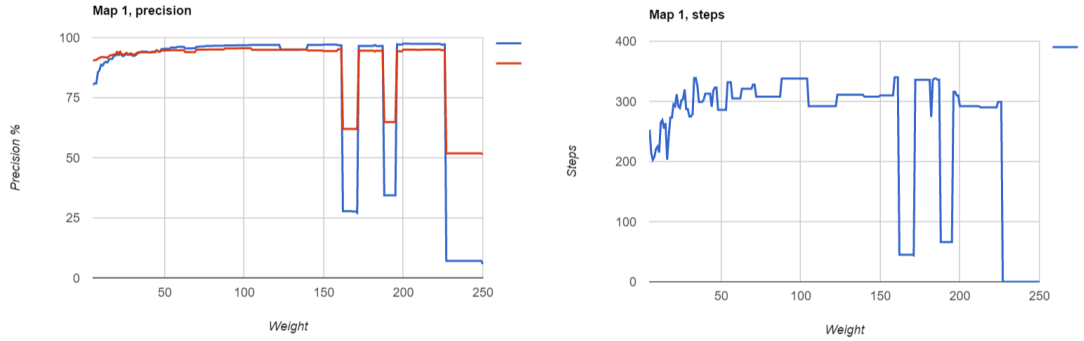
Figure D.4: Map 4



(a) Precision (b) Steps

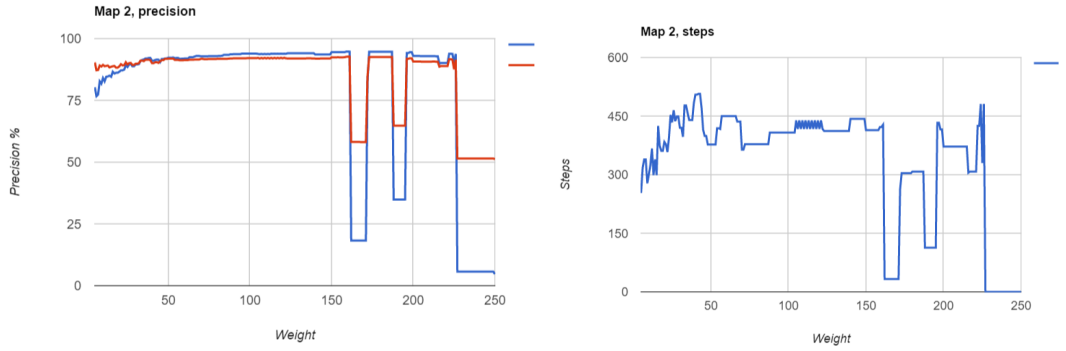
Figure D.5: Map 1, map 2, map 3 and map 4, fused together

D.1.2 Task 4, simulation test part 2



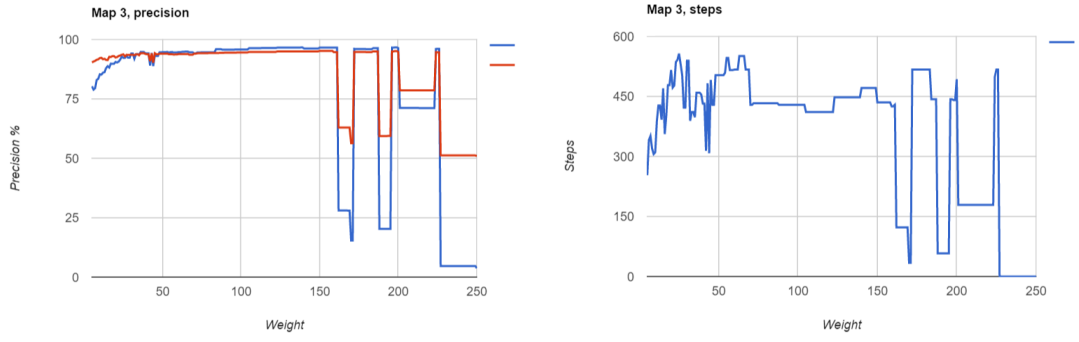
(a) Precision (b) Steps

Figure D.6: Map 1



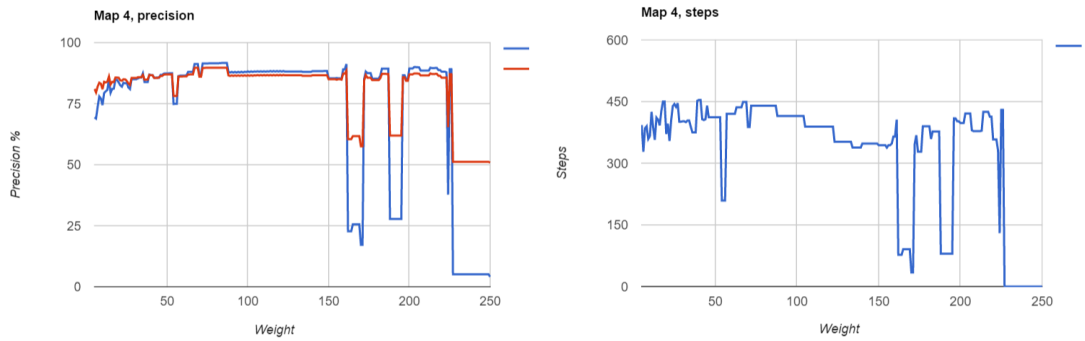
(a) Precision (b) Steps

Figure D.7: Map 2



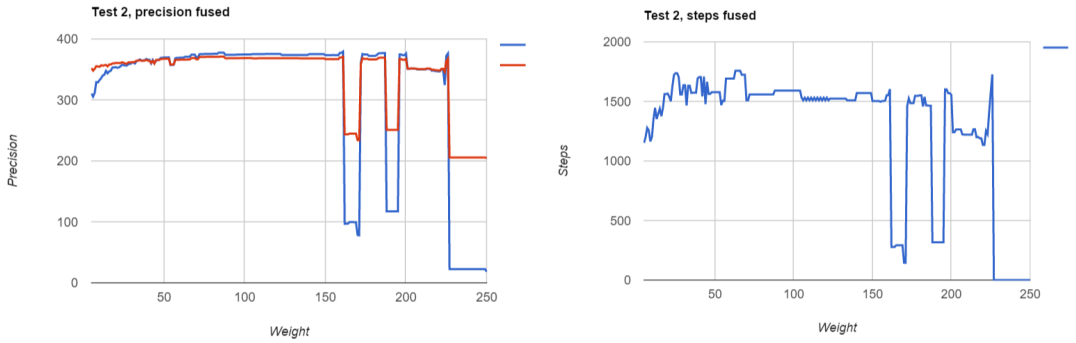
(a) Precision (b) Steps

Figure D.8: Map 3



(a) Precision (b) Steps

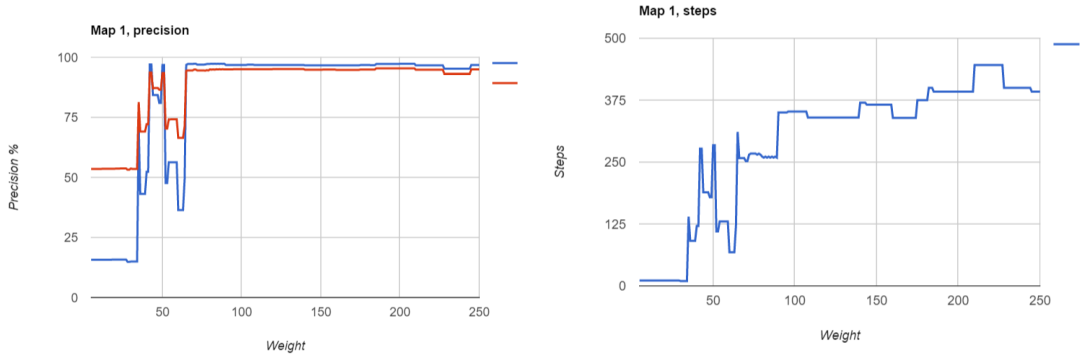
Figure D.9: Map 4



(a) Precision (b) Steps

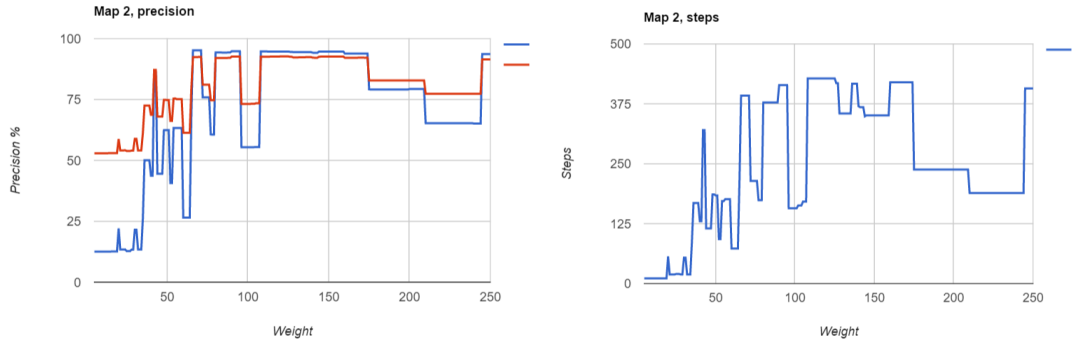
Figure D.10: Map 1, map 2, map 3 and map 4, fused together

D.1.3 Task 4, simulation test part 3



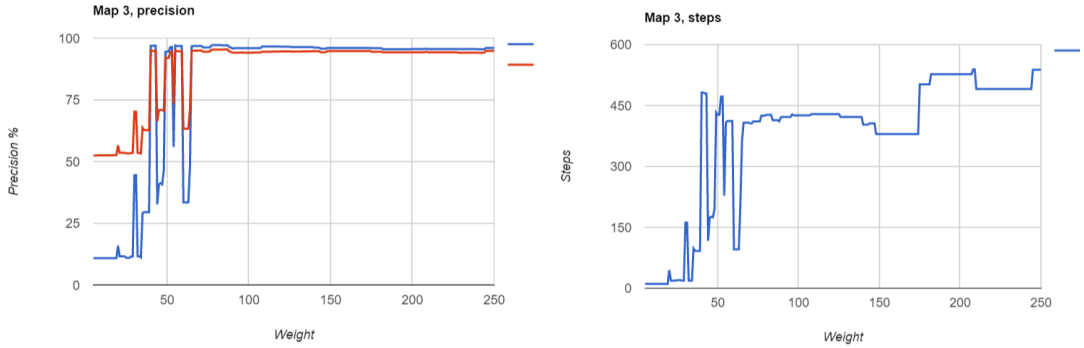
(a) Precision (b) Steps

Figure D.11: Map 1



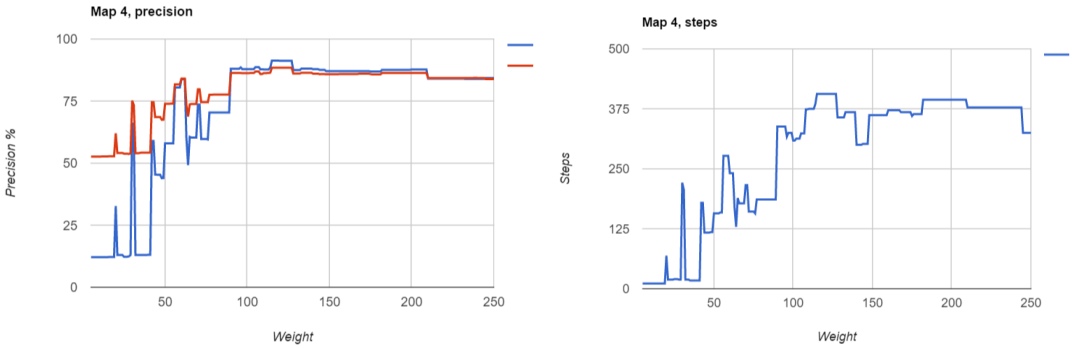
(a) Precision (b) Steps

Figure D.12: Map 2



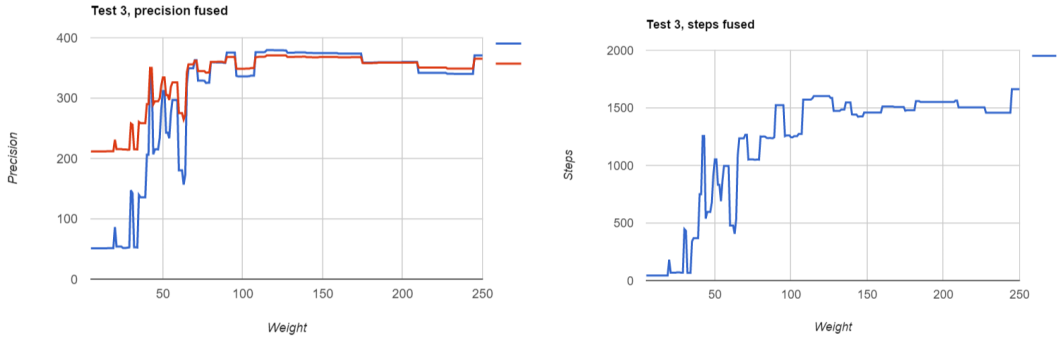
(a) Precision (b) Steps

Figure D.13: Map 3



(a) Precision (b) Steps

Figure D.14: Map 4



(a) Precision (b) Steps

Figure D.15: Map 1, map 2, map 3 and map 4, fused together

D.2 Pictures of physical test 2

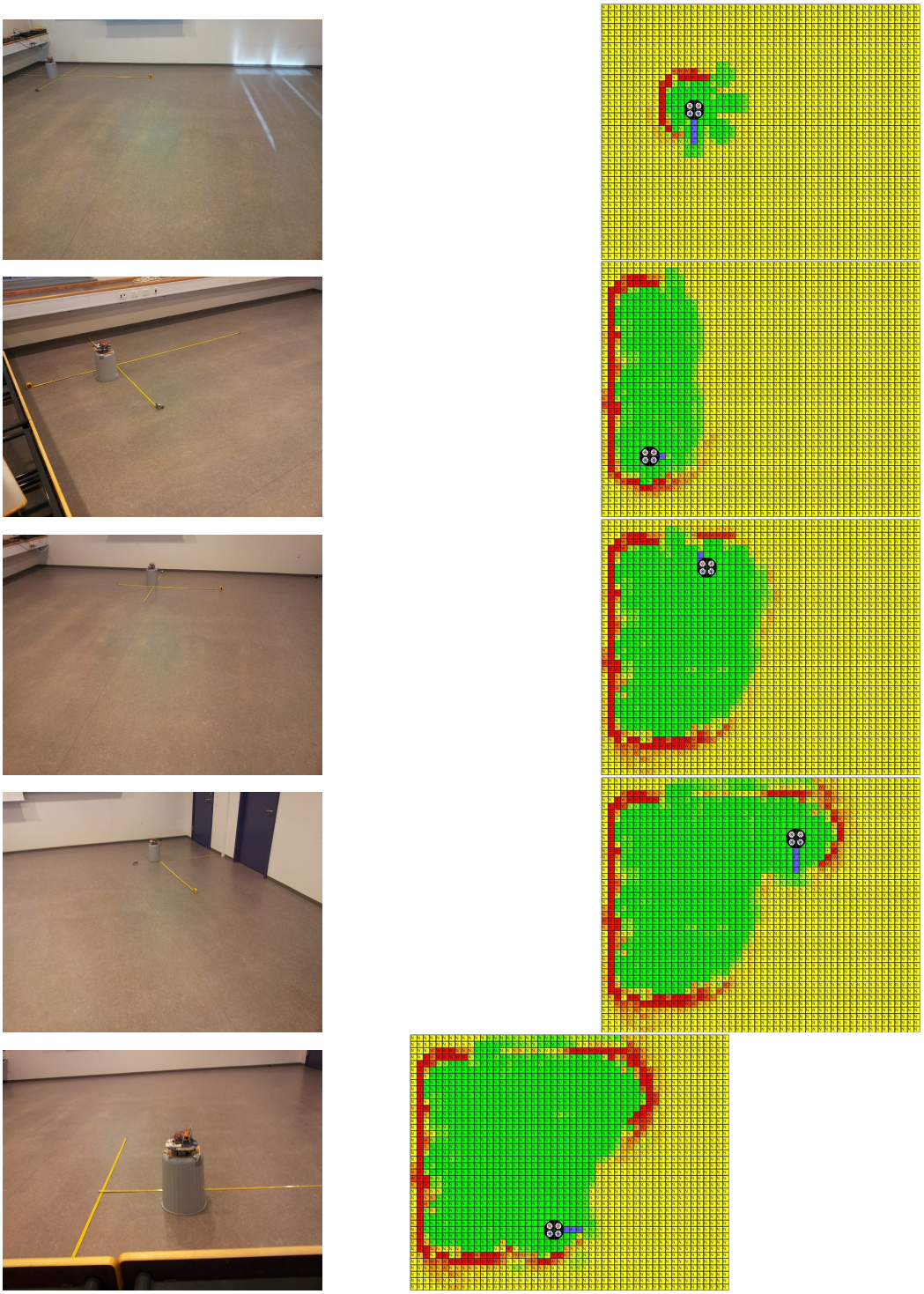


Figure D.16: Pictures of the placement of the quadcopter representation and the system, step by step

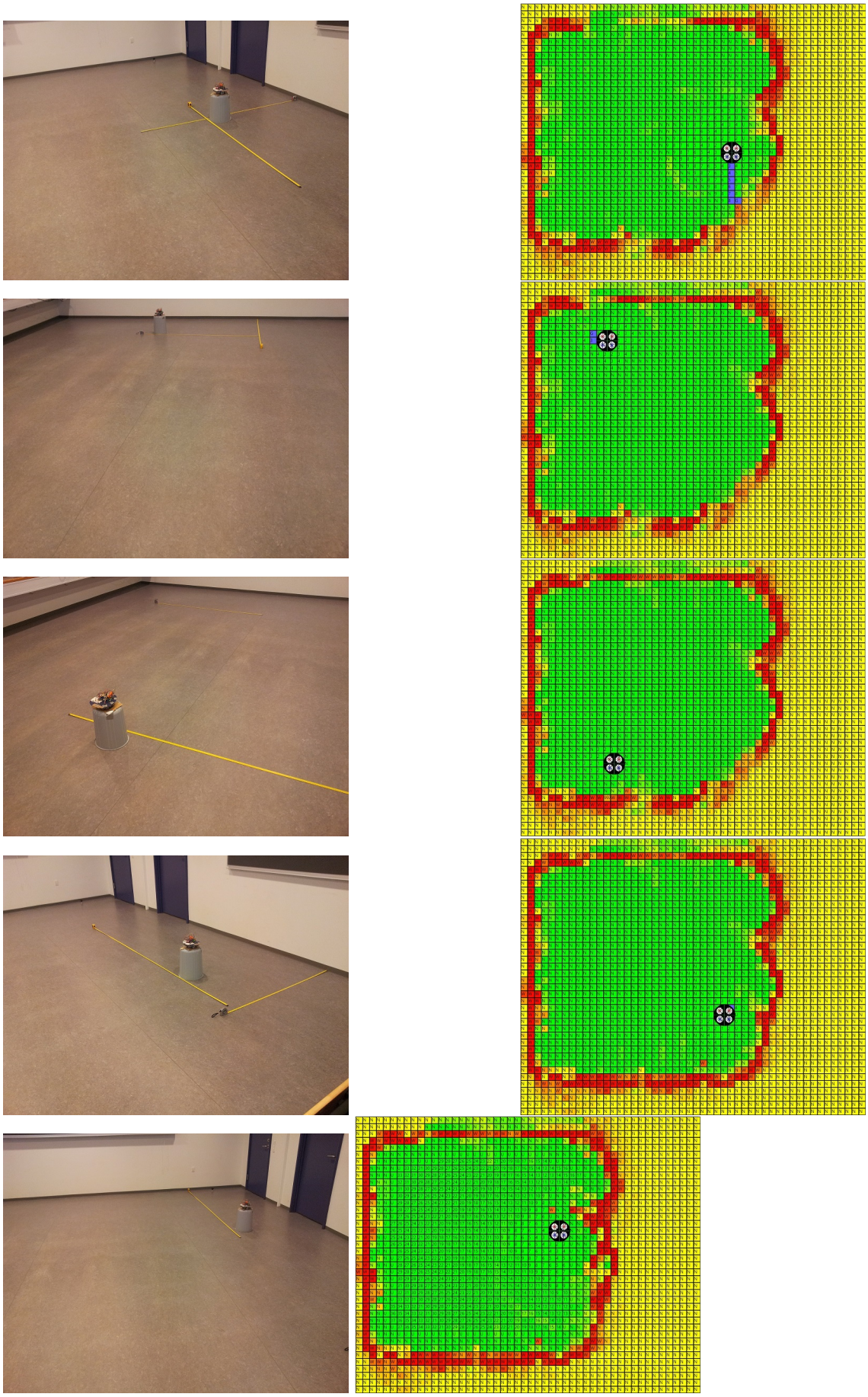
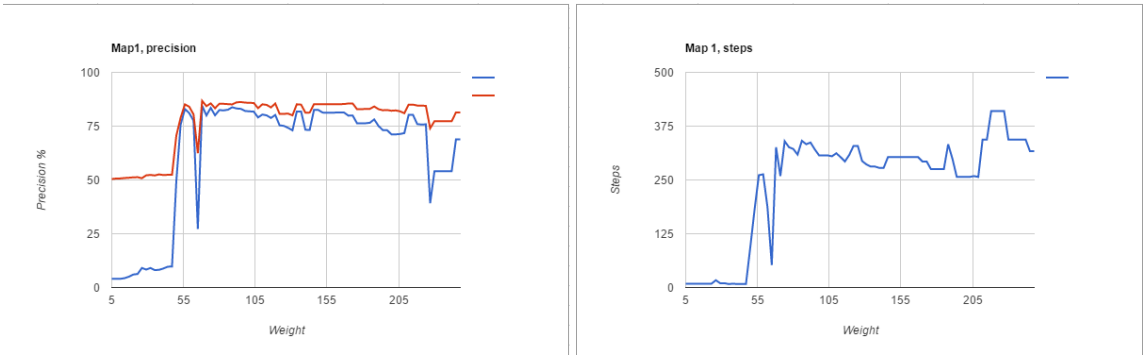


Figure D.17: Pictures of the placement of the quadcopter representation and the system, step by step

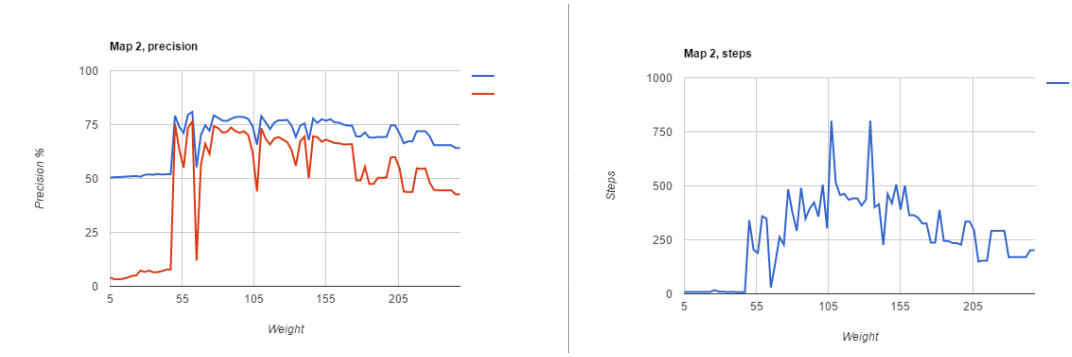
D.3 Weight test 2 data

D.3.1 Task 4, simulation test part 1



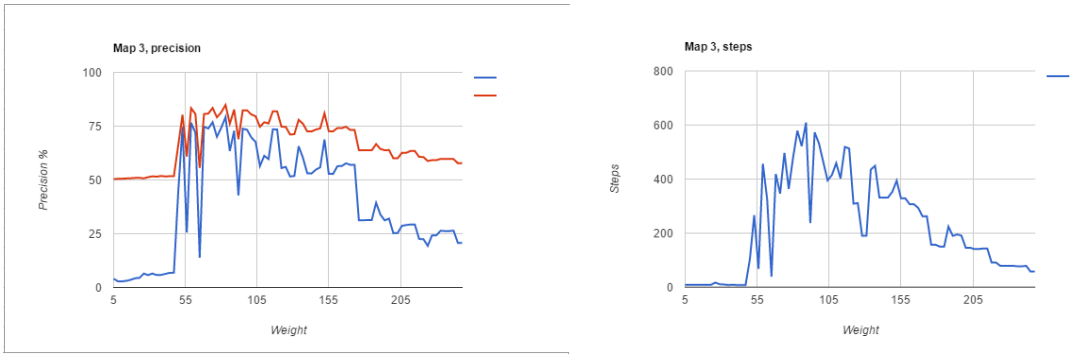
(a) Precision (b) Steps

Figure D.18: Map 1



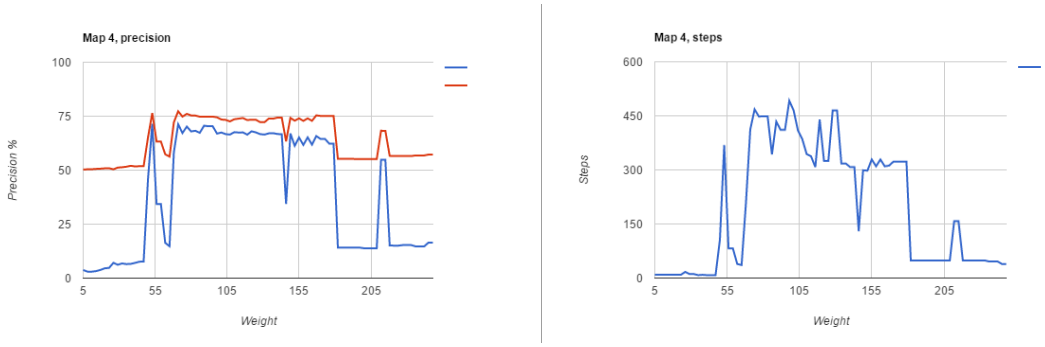
(a) Precision (b) Steps

Figure D.19: Map 2



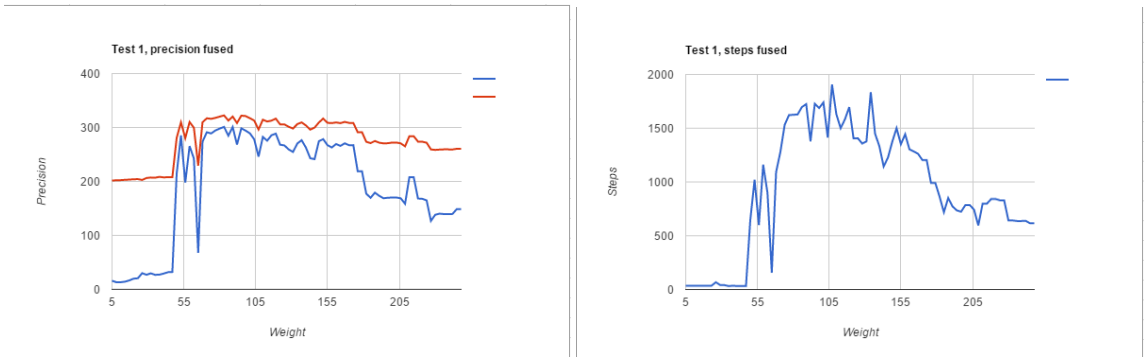
(a) Precision (b) Steps

Figure D.20: Map 3



(a) Precision (b) Steps

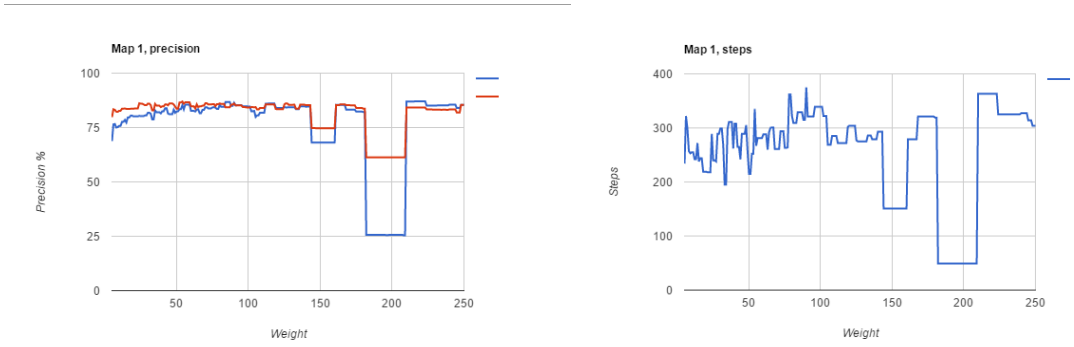
Figure D.21: Map 4



(a) Precision (b) Steps

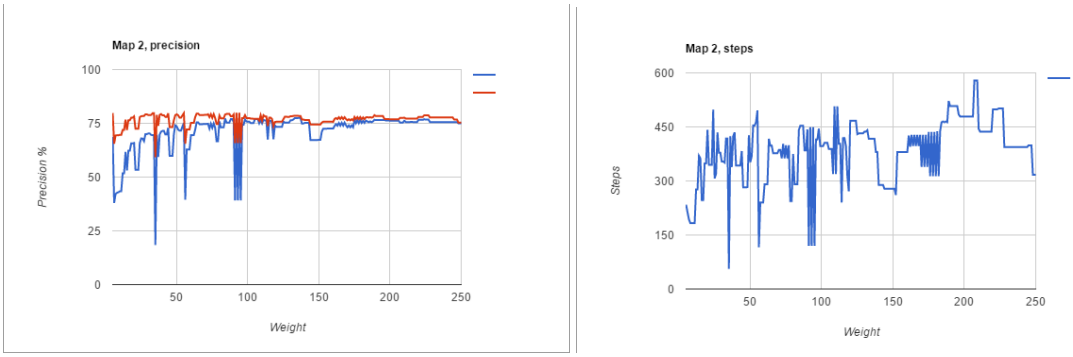
Figure D.22: Map 1, map 2, map 3 and map 4, fused together

D.3.2 Task 4, simulation test part 2



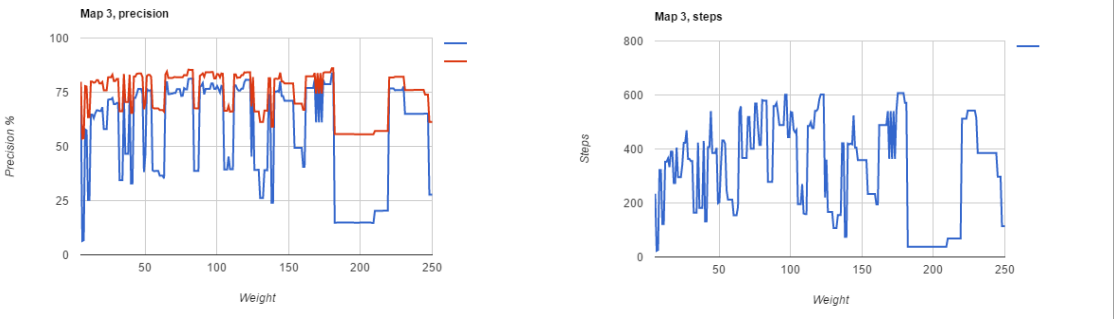
(a) Precision (b) Steps

Figure D.23: Map 1



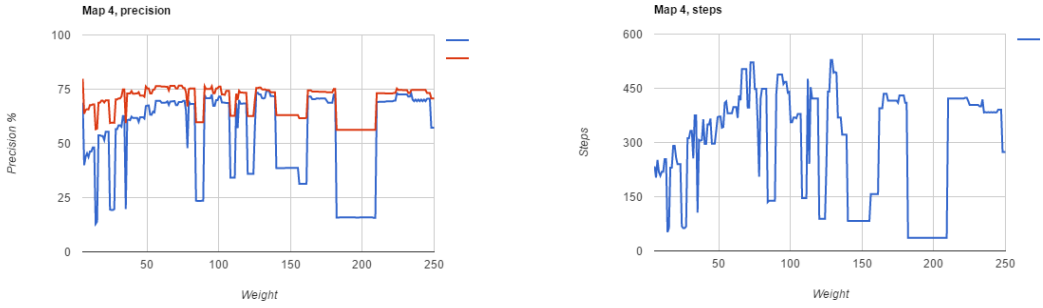
(a) Precision (b) Steps

Figure D.24: Map 2



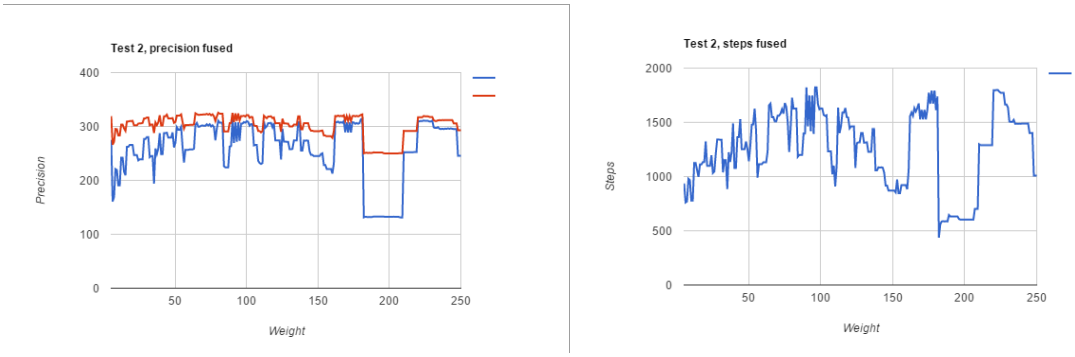
(a) Precision (b) Steps

Figure D.25: Map 3



(a) Precision (b) Steps

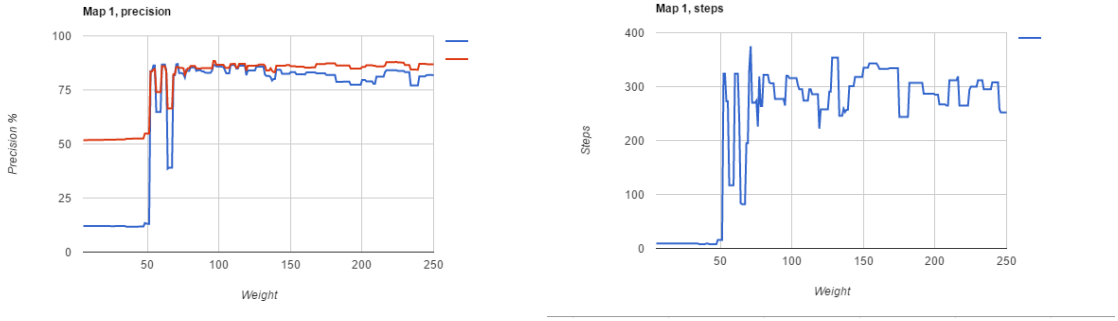
Figure D.26: Map 4



(a) Precision (b) Steps

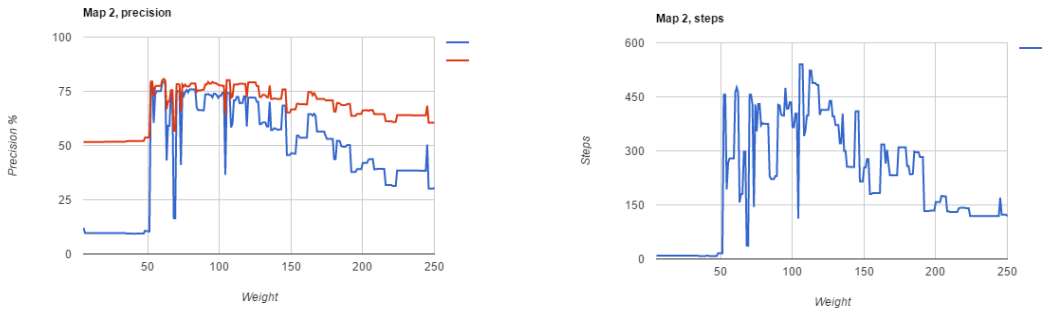
Figure D.27: Map 1, map 2, map 3 and map 4, fused together

D.3.3 Task 4, simulation test part 3



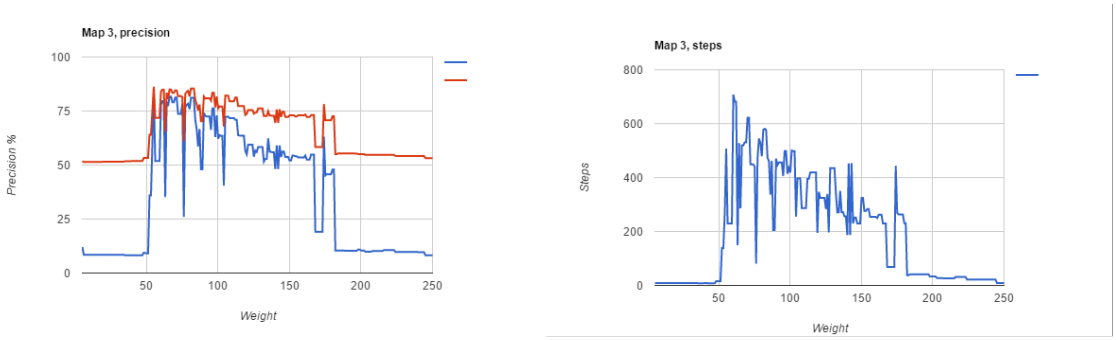
(a) Precision (b) Steps

Figure D.28: Map 1



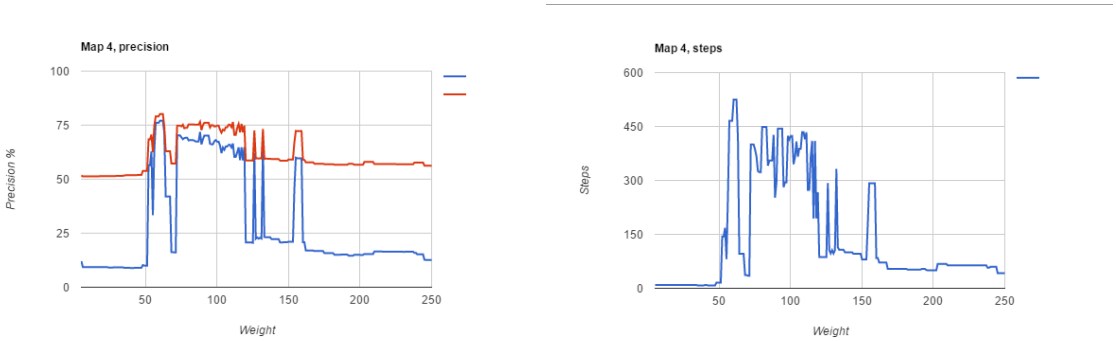
(a) Precision (b) Steps

Figure D.29: Map 2



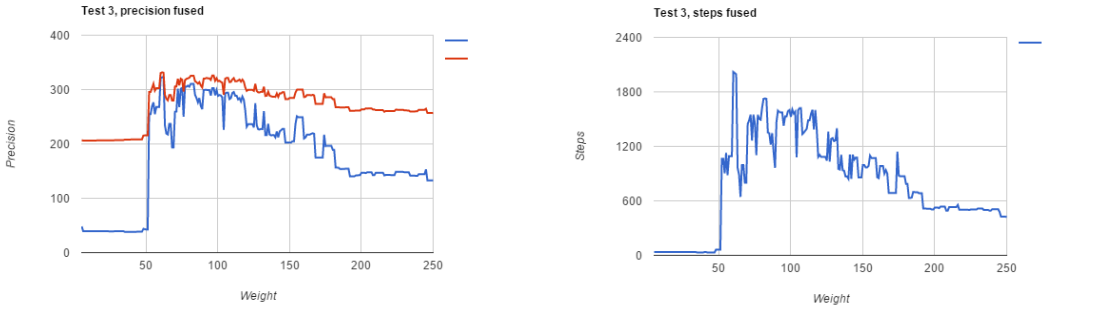
(a) Precision (b) Steps

Figure D.30: Map 3



(a) Precision (b) Steps

Figure D.31: Map 4



(a) Precision (b) Steps

Figure D.32: Map 1, map 2, map 3 and map 4, fused together

D.4 Class diagram

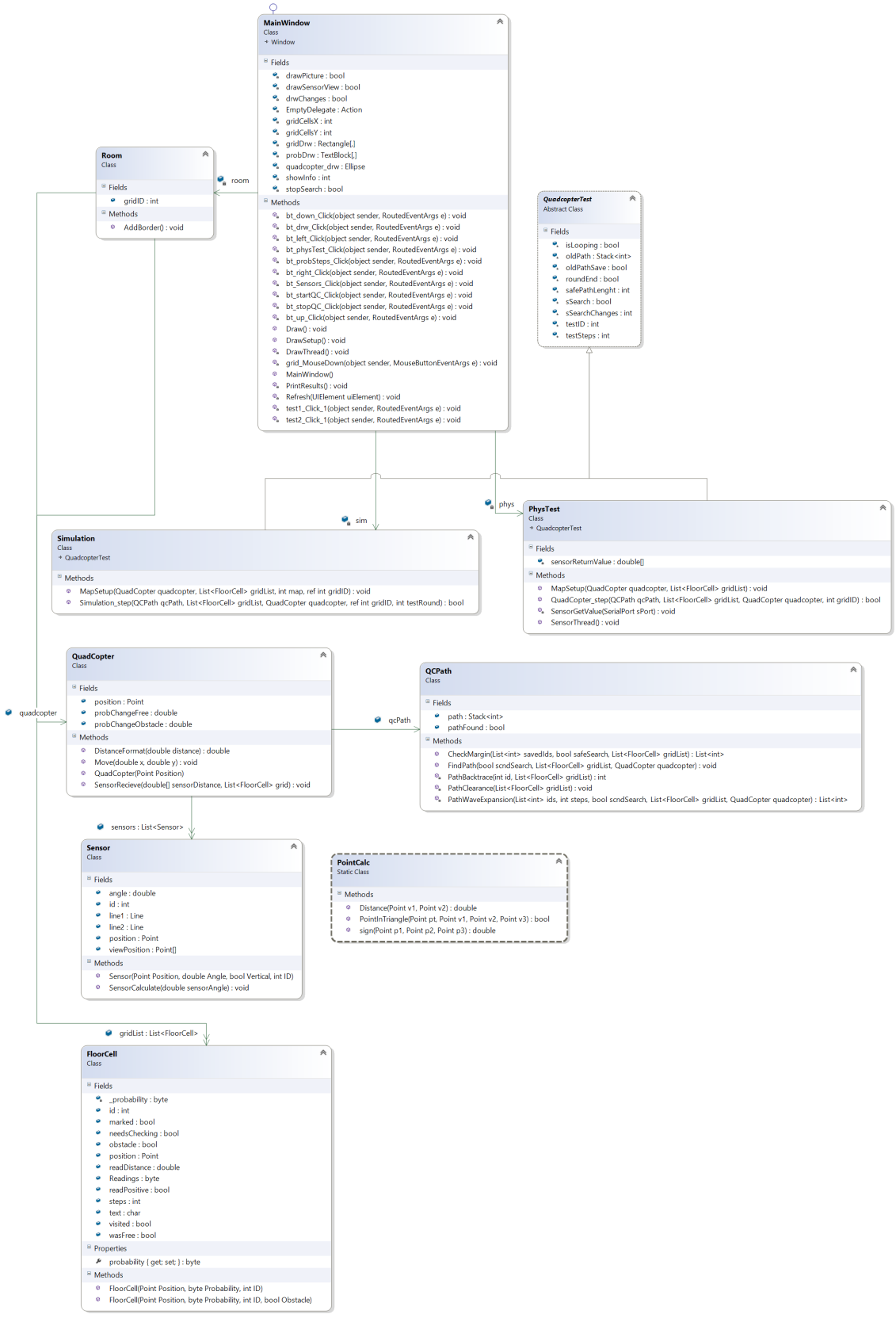


Figure D.33: Task 4 class diagram.

D.5 Weight test one analysis and result

Weight testing

The Quadcopter class contained the two variables `probChangeFree` and `probChangeObstacle` which are set to 161 and 125 respectively. These two variables represent how much probability is added or subtracted from cells, whenever an obstacle or a free cell is detected. The values of the two variables are not random, they have been found doing an extensive amount of simulations on simulated rooms. In total four rooms were created with different characteristics, in order to make the simulations as diverse as possible while still keeping the same circumstances for all input. In this case the input variables will be `probChangeFree` and `probChangeObstacle`. These simulations were done in three parts:

Part one both `probChangeFree` and `probChangeObstacle` are raised by one after each set of maps (the four maps) have been tested.

Part two `probChangeObstacle` is set to the value found in part one, and `probChangeFree` is raised by one after each set of maps.

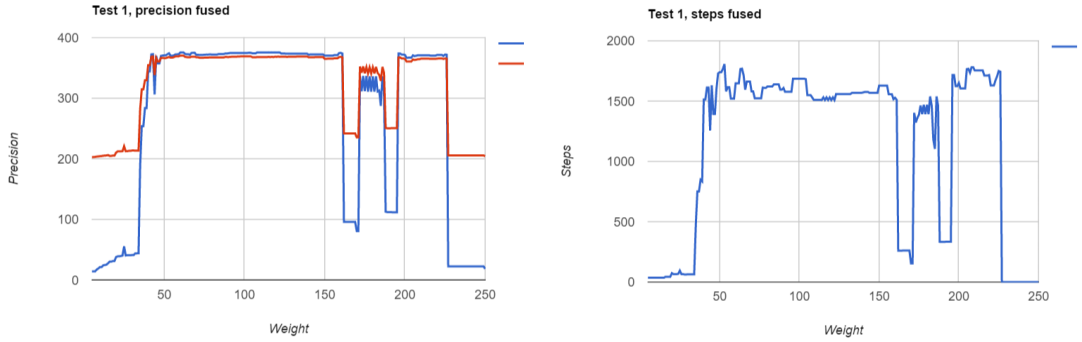
Part three `probChangeFree` is set to the value found in part two, and `probChangeObstacle` is raised by one after each set of maps.

For each part the variable(s) are tested in the range 5-250.

Part one: In figure D.34 the test results of part one has been visualized with graphs, these are the collective results of all four maps in part one. Test results for the individual maps can be found in appendix D.1.1. These figures contain two images, the left one with the caption "precision" are results of how correct the maps have been drawn in the simulation. The other figure on the right with the caption "steps", is the amount of steps the quadcopter had to make in order to complete the maps. The axis called weight refers to the variables `probChangeFree` and `probChangeObstacle`, depending on which is currently being given as input. The graph on the left with the caption "precision", contains two graphs, these are two different ways of calculating how correct a room drawing is. The first way which is what the blue line represents, is calculated by comparing how many grid cells are in the current map, with how many should be free and how many is above 75 in probability, which means that they are free. The same is done with obstacles, but instead of being above 75 they need to be below 25. In this method all cell with a probability between 25 and 75 are classified as errors. The other method which is represented by the red line is a little less harsh, this method looks at each individual cells probability. For obstacles it takes the difference between zero and the cells probability, and for free cells it takes the difference between 99 and the cells probability, these differences are the amount of error in the drawing. With this method the correctness of drawings start at 50% since all cells has a probability of 50 from the start.

When analyzing and deciding the best results, the correctness of a drawing is prioritized higher than the amount of steps the quadcopter has to use to finish the drawing, however if a significant reduction can be made to the amount of steps without loosing a lot of correctness then this will be considered.

The test results of part one revealed the preliminary optimal value for `probChangeFree` and `probChangeObstacle` to be 122. This gave a correctness of 375.33/400 using the first method to calculate correctness and 368.25/400 using the other method. This took a total of 1509 steps across all four maps.

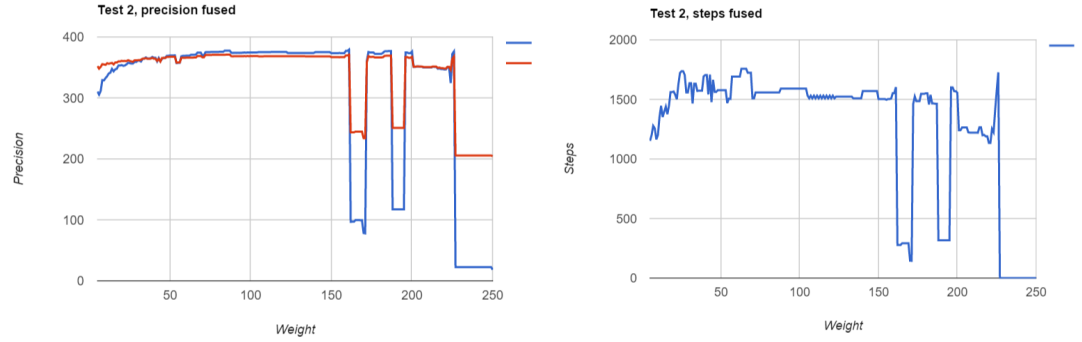


(a) Precision (b) Steps

Figure D.34: Map 1, map 2, map 3 and map 4, fused together

Part two In figure D.35 the test results of part two has been visualized with graphs, these are the collective results of all four maps in part two. Test results for the individual maps can be found in appendix D.1.2.

This test was made with `probChangeObstacle` set to a constant of 122, found in part one. Only `probChangeFree` is changed in this part. The results of part two revealed 161 to be the new optimal value for `probChangeFree`, giving a correctness of 379.29/400 and 370.85/400 using 1603 steps. This is an increase in correctness of 3.96 and 2.6.



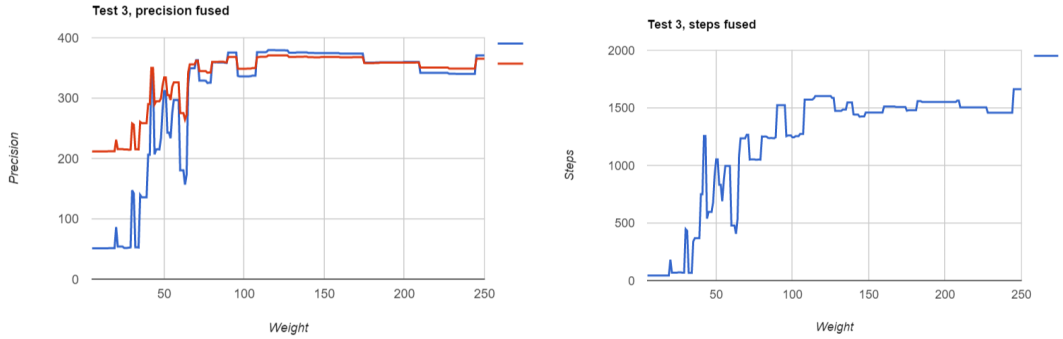
(a) Precision (b) Steps

Figure D.35: Map 1, map 2, map 3 and map 4, fused together

Part three In figure D.36 the test results of part three has been visualized with graphs, these are the collective results of all four maps in part three. Test results for the individual maps can be found in appendix D.1.3.

This test was made with `probChangeFree` set to a constant of 161, found in part two. Only `probChangeObstacle` is changed in this part. The results of part three revealed 125 to be the new optimal value for `probChangeObstacle`, giving a correctness of 379.29/400 and

370.86/400 using 1603 steps. This last test made almost no difference, it only increased the second correctness with 0.01, although without any negative effects. Another test was made to make sure that this is indeed the optimal values, that test revealed that the current values are indeed the optimal ones.



(a) Precision (b) Steps

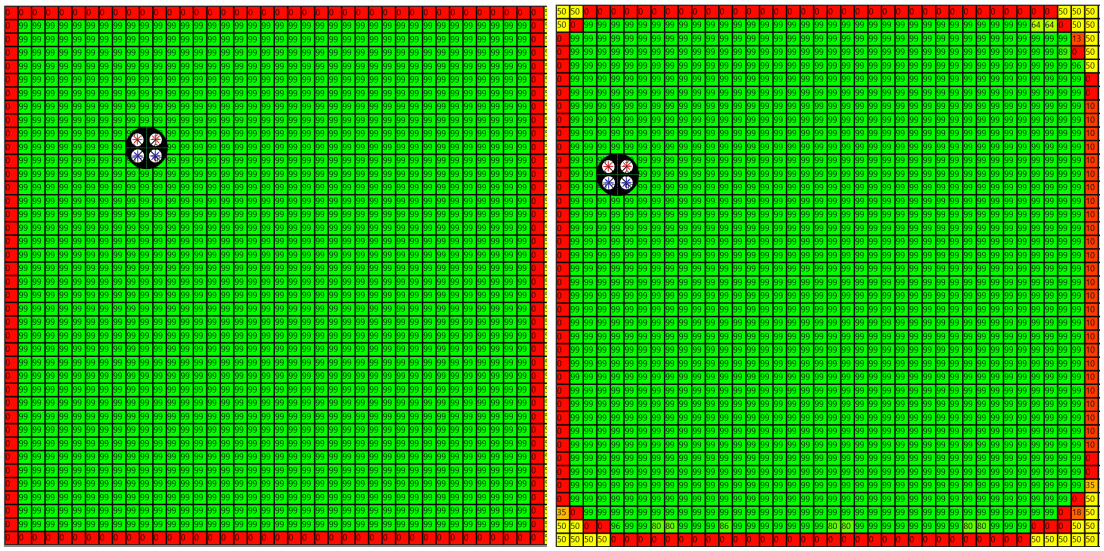
Figure D.36: Map 1, map 2, map 3 and map 4, fused together

D.6 Step one result

Step one results

The following four figures shows a representation of how the different maps actually look on the left, and how precise the system was able to draw the same room during the simulation, with the optimal values for `probChangeFree` and `probChangeObstacle`, on the right. This includes figure D.37, D.38, D.39 and D.40.

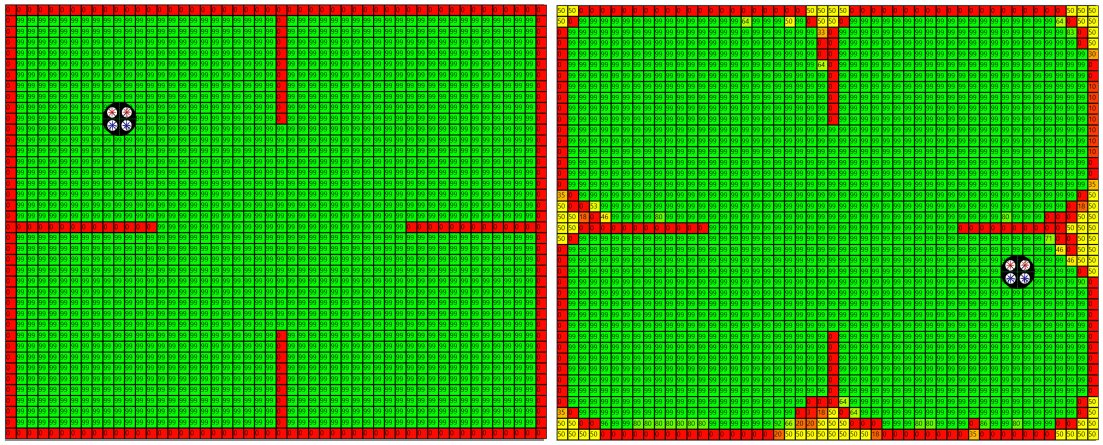
The system seems to be able to draw map 1 almost perfectly with the exception of the corners, which are hard to draw because of the inconsistencies of the sensors. This map has a correctness of 96.81/100 and 95.07/100 with 340 steps. The room itself is 8x8 meters.



(a) 100% correct (b) Simulation

Figure D.37: Map 1

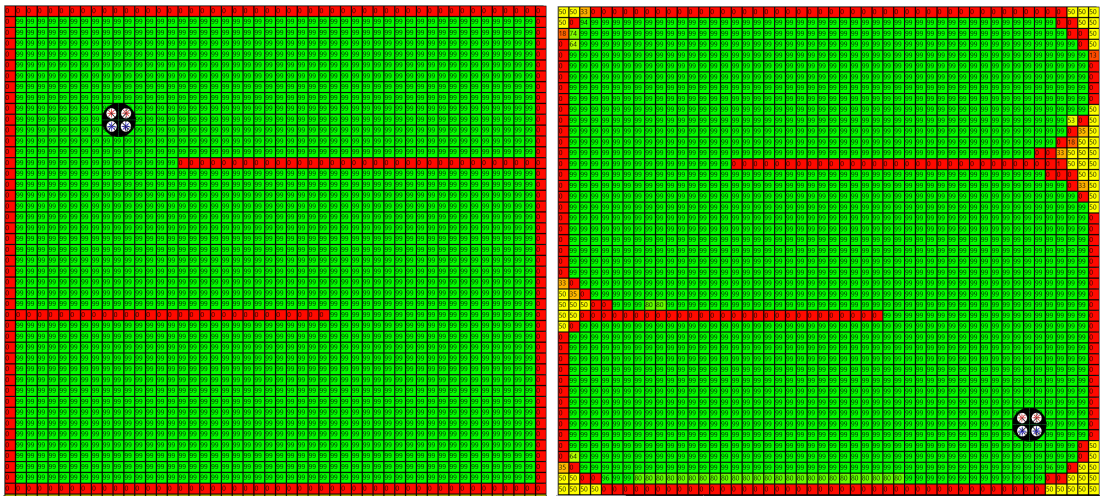
The simulation of the second map gives a very good idea of what the room actually looks like, but because the room has a lot more corners it also has a lower correctness percentage. Here the correctness is 94.7/100 and 92.67/100 using 428 steps. This room is 10x8 meters.



(a) 100% correct (b) Simulation

Figure D.38: Map 2

The third map is also drawn very well and again only corners seems to be causing problems. For this map the correctness is 96.57/100 and 94.66/100 using 429 steps. This room is 10x9 meters.

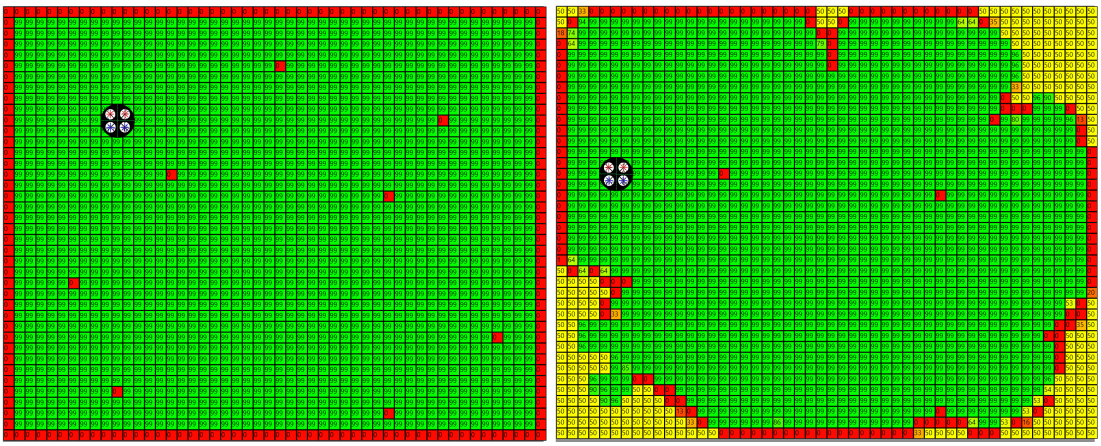


(a) 100% correct (b) Simulation

Figure D.39: Map 3

Map 4 which is the last map tested has more problems and is not drawn nearly as well as the other. This map was meant to represent a hard case for the quadcopter to draw, however it does not pose a risk for the quadcopter since it does not claim that any cell of the obstacles is a free spot. But because the sensors are too imprecise the quadcopter will see some of the obstacles in the middle as a continuous wall. In order to remove these errors the quadcopter will need to be able to move around the obstacles, as it has with two

of them. The correctness of this room is 91.2/100 and 88.45/100 using 406 steps. This room is 10x8 meters.



(a) 100% correct (b) Simulation

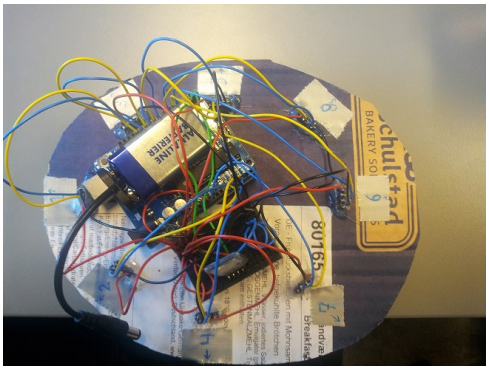
Figure D.40: Map 4

Before testing the system in a physical test one more simulation test was performed, in this test the quadcopter went through 500 randomly generated 10x10 m rooms the purpose of this test was to make sure that the quadcopter would never move over an obstacle in the simulation no matter what. The result of this test was that the quadcopter indeed would not move over any obstacles during the 500 simulations.

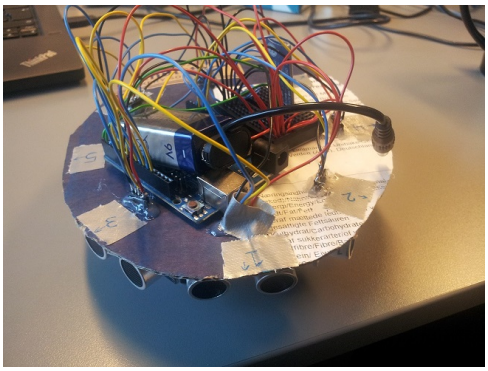
Based on the results found during the simulations, we believe that the system is ready for its first physical test using the actual sensors, which can be read more about in the next section.

D.7 Step two physical test

In this section step two, the physical test, will be explained, with a short description of the representation of the quadcopter made, to make this test possible, and the results of doing this test.



(a) Seen from above.



(b) Seen more from the side, the HC-SR04 distance sensors can be seen underneath the cardboard.

Figure D.41: Pictures of the quadcopter representation.

To do this test a rough copy of the quadcopter's size and shape were build using cardboard, placing the HC-SR04 distance sensors on the approximate location we would want them on the real quadcopter, see figure D.41. The method used in part one was then applied to the cardboard representation of the quadcopter, with the input now coming from the physical representation instead of simulating the input. The cardboard representation of the quadcopter were then moved by hand, according to the commands the system gave the quadcopter.

The input from the cardboard representation came from the distance sensors connected to the Arduino, which uses the JY-MCU bluetooth module, see section 5.3, to send it to the system. To do this the code in listing D.1 is used:

```
#include <NewPing.h>

#define SONAR_NUM 8
#define MAX_DIST 200
#define PING_INTERVAL 3600

unsigned int cm[SONAR_NUM];

String dataSend = "";

NewPing sonar[SONAR_NUM] = {
  NewPing(13, 12, MAX_DIST),
  NewPing(A0, A1, MAX_DIST),
  NewPing(11, 10, MAX_DIST),
  NewPing(A2, A3, MAX_DIST),
  NewPing(9, 8, MAX_DIST),
  NewPing(5, 4, MAX_DIST),
  NewPing(3, 2, MAX_DIST),
  NewPing(7, 6, MAX_DIST)
};

void setup() {
    Serial.begin(9600); // Default connection rate for my BT module
}

void loop() {
  dataSend = "";
  for(uint8_t i = 0; i < SONAR_NUM; i++){
```

```
29      cm[i] = (sonar[i].ping() / US_ROUNDTRIP_CM);
30
31      dataSend += String(cm[i]);
32      dataSend += ", ";
33      delayMicroseconds(PING_INTERVAL);
34    }
35    Serial.print(dataSend);
36    Serial.println();
37  }
```

Example code D.1: Arduino code used for the physical test

As can be seen from example code D.1, the Arduino got the different sensors initialized in an array, where every sensor is pinged in the `for`-loop, which is then converted to centimeters. Each results of one round of measuring, meaning pinging each sensor once, is then appended in the `dataSend` string. The string is then printed to the serial port of the Arduino, with the line `Serial.print(dataSend)`, where the bluetooth module JY-MCU is connected. After `dataSend` is printed, the code will go to the top of the `void loop()`, and `dataSend` will be cleared and ready to start the process again.

In order to receive sensor information and always having the newest data available, a new thread responsible for handling this was created. This thread makes sure to handle the data received at all times and assigning a variable with the newest data from the sensors. If this is not done continuously, then whenever new information is asked for it will be the data next in line since the last data retrieval. For the test the same method as in the simulation was used for controlling the quadcopter, except now the system use the data from the sensors.

D.8 Weight test two analysis and result

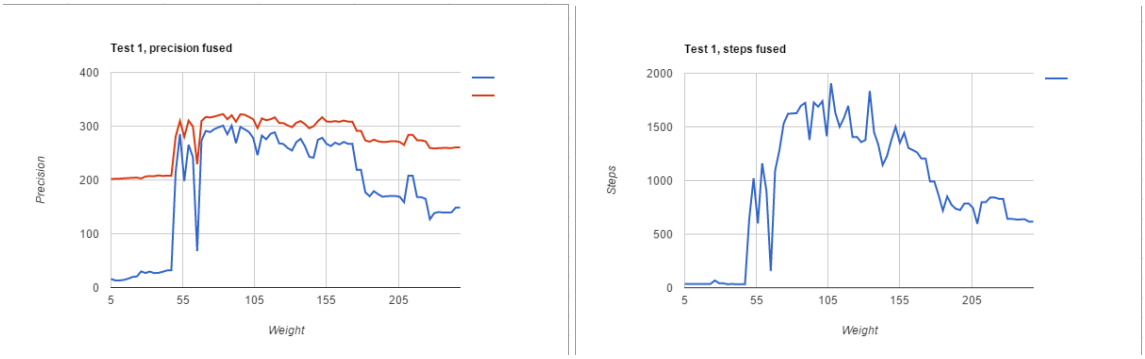
Weight testing

After tweaking the system a bit, it was again able to somewhat draw rooms, however the results are not nearly as good. Another weight test was made to find new values for the `probChangFree` and `probChangeObstacle` variables.

This test was performed the same way as weight testing for the first simulation, split into three parts and first testing both variables, then `probChangeFree` and then `probChangeObstacle`.

Part one In figure D.42 the test results of part one has been visualized with graphs, these are the collective results of all four maps in part one. Test results for the individual maps can be found in appendix D.1.2.

The test results of part one revealed the preliminary optimal value for `probChangeFree` and `probChangeObstacle` to be 83, giving a correctness of 301.08/400 and 322.39/400 using 1628 steps.

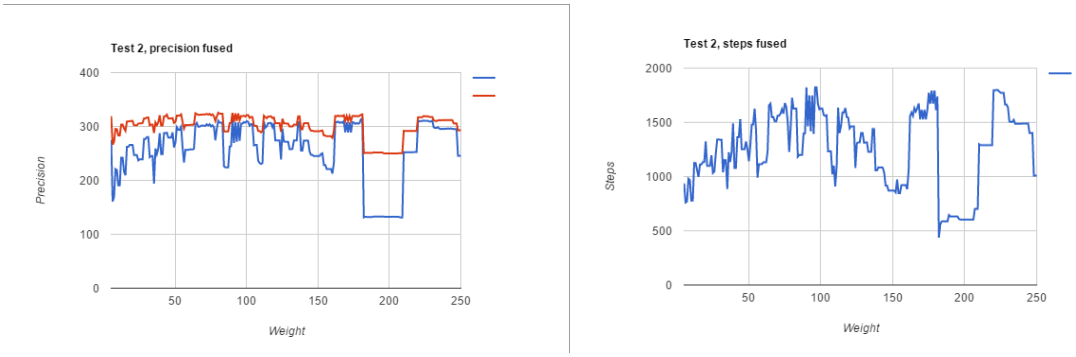


(a) Precision (b) Steps

Figure D.42: Map 1, map 2, map 3 and map 4, fused together

Part two In figure D.43 the test results of part two has been visualized with graphs, these are the collective results of all four maps in part two. Test results for the individual maps can be found in appendix D.1.2.

This test was made with `probChangeObstacle` set to a constant of 83, found in part one. Only `probChangeFree` is changed in this step. The results of part two revealed 80 to be the new optimal value for `probChangeFree`, giving a correctness of 310.54/400 and 325.52/400 using 1727 steps. This is an increase in correctness of 9.46 and 3.13.

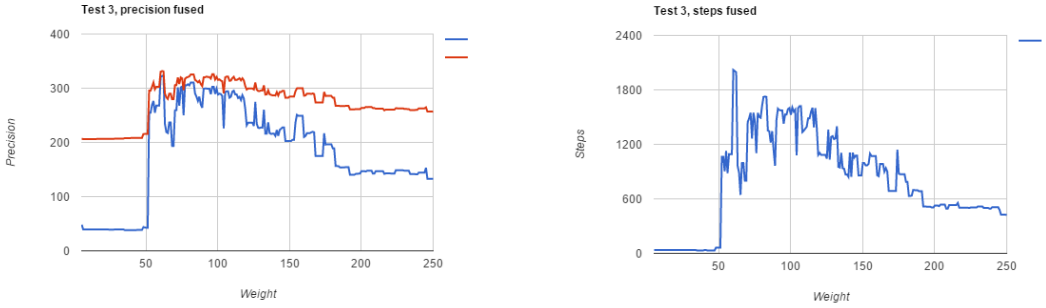


(a) Precision (b) Steps

Figure D.43: Map 1, map 2, map 3 and map 4, fused together

Part three In figure D.44 the test results of part three has been visualized with graphs, these are the collective results of all four maps in part three. Test results for the individual maps can be found in appendix D.1.2.

This test was made with `probChangeFree` set to a constant of 80, found in step one. Only `probChangeObstacle` is changed in this part. The results of part three revealed 61 to be the new optimal value for `probChangeObstacle`, giving a correctness of 322.65/400 and 331.67/400 using 2007 steps. This is an increase in correctness of 12.11 and 6.15.



(a) Precision (b) Steps

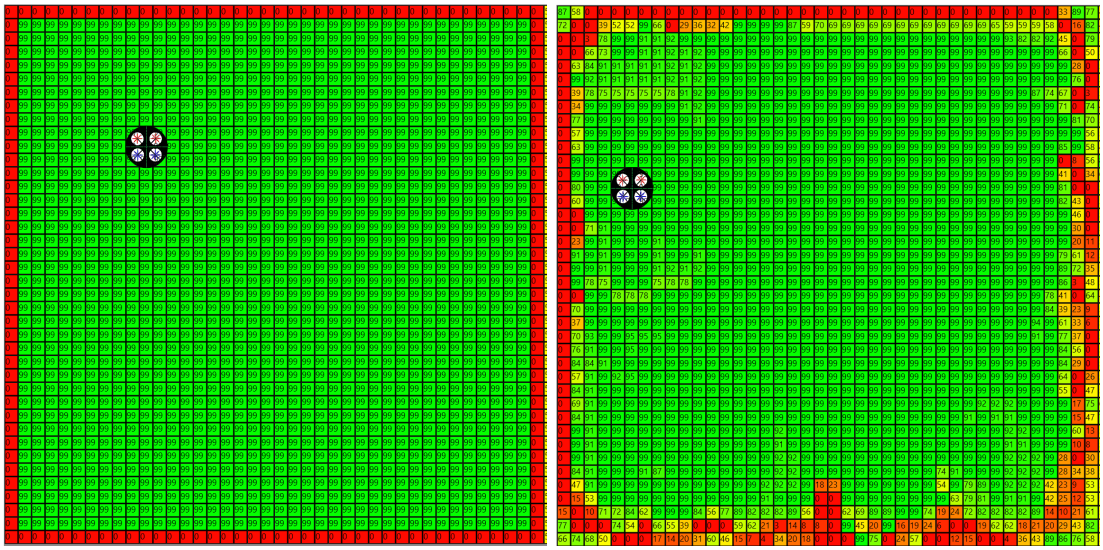
Figure D.44: Map 1, map 2, map 3 and map 4, fused together

D.9 Step three result

Step three results

The following four figures shows a representation of how the different maps actually look on the left, and how precise the system was able to draw the same room during the updated simulation, with the optimal values for `probChangeFree` and `probChangeObstacle`, on the right. The maps used for testing are the same maps as in the first simulation test. This includes figure D.45, D.46, D.47 and D.48.

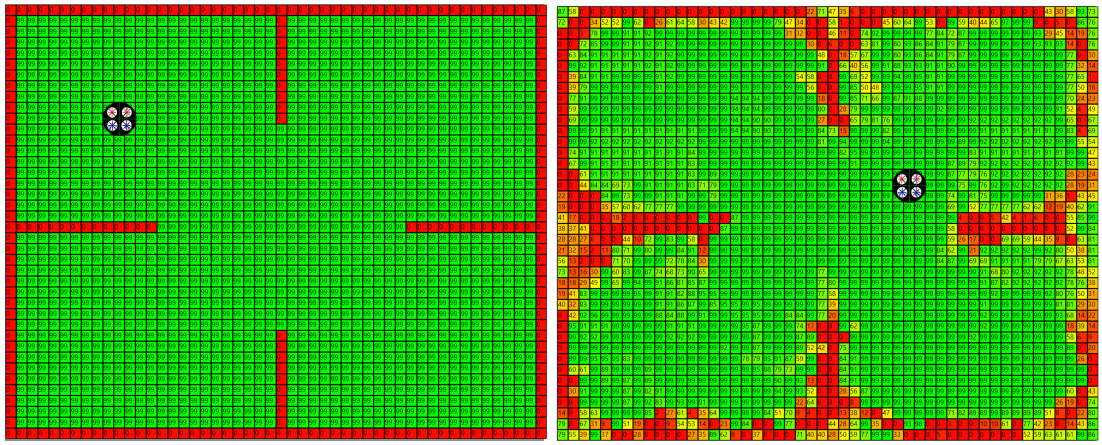
It is very clear that the quality of the drawings have been reduced substantially since the first simulation. However looking at map one it still gives a rough idea of what the room looks like, at least it is still mostly able to show where it is possible to be an the shape of the room. Making even walls seems to have become a problem though. Map 1 has a correctness of 86.75/100 and 85.93/100 with 324 steps.



(a) 100% correct (b) Simulation

Figure D.45: Map 1

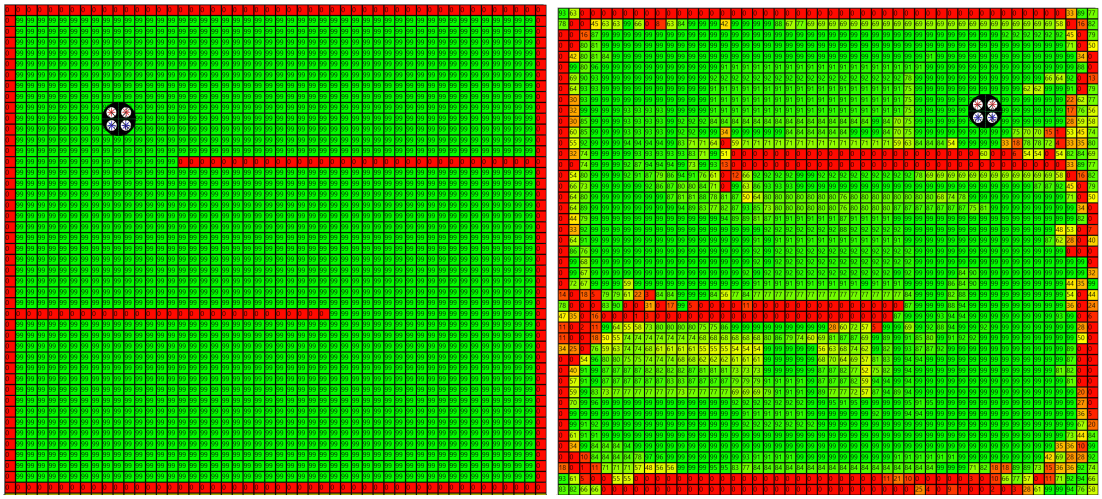
Just like with map 1, the second map has reduced quality, and yet it is still possible to get an idea of the room. The correctness of this map is 79.25/100 and 80.70/100 using 476 steps.



(a) 100% correct (b) Simulation

Figure D.46: Map 2

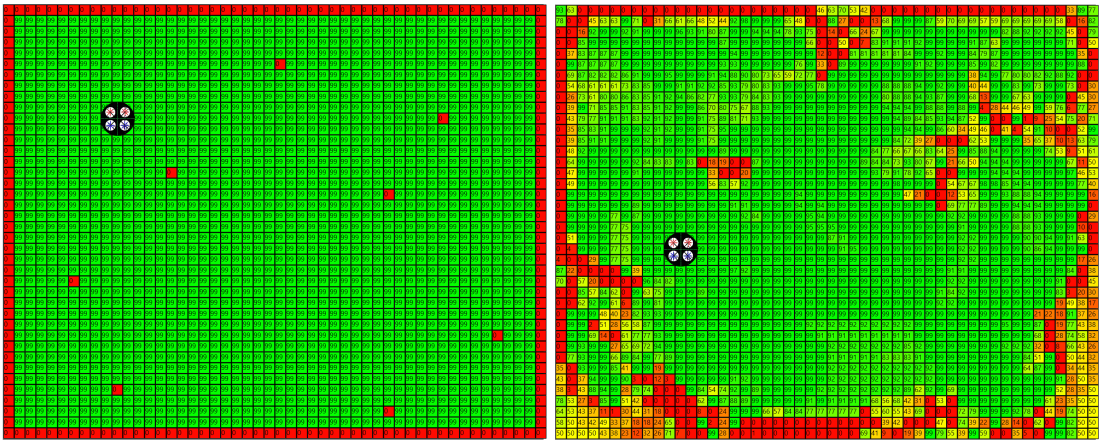
Map 3's correctness with the updated simulation has been reduced to 79.6/100 and 84.86/100 using 682 steps.



(a) 100% correct (b) Simulation

Figure D.47: Map 3

Map 4 is the only one that differs from the other three maps compared to the first simulation, its correctness has been reduced like the other maps, but it did however manage to explorer some parts of the map which wasn't explored in the first simulation. In the new simulation the quadcopter was able to explore parts of the bottom left corner and top right corner which wasn't explored in the first simulation. The correctness of this map is 77.05/100 and 80.17/100 using 525 steps.



(a) 100% correct (b) Simulation

Figure D.48: Map 4

Just like in step one this test was also finished by making sure the quadcopter would never hit a wall, by running 500 simulations in randomly generated rooms, which again was completed successfully. After this we decided to do another physical test, which is continued in the next section.