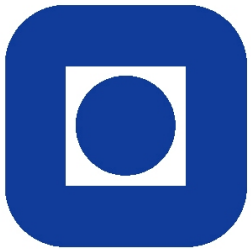
NTNU
Det skapende universitet

Assistert fjernkontroll av innendørs UAV

Anders Strand

Master i kybernetikk og robotikk
Innlevert: juni 2015
Hovedveileder: Amund Skavhaug, ITK

Norges teknisk-naturvitenskapelige universitet
Institutt for teknisk kybernetikk



Aided remote control of indoor UAV

Anders Strand

Trondheim, June 2015

Master thesis

Department of Engineering Cybernetics

Norwegian University of Science and Technology

Supervisor: Amund Skavhaug

Preface

The use Unmanned Aerial Vehicles (UAV) is steadily expanding to new areas. Especially quadrotor UAVS's are gaining popularity in both professional and hobby scenes. Controlling a quadrotor UAV is an active and demanding task, which requires two hands and complete attention from the human pilot. This paper suggests an assistant control system which handles many of the pilot's tasks, leaving a simple control interface for the human pilot to use. The pilot can simply tell the UAV where to move, and the assistant system will take care of position holding and collision avoidance. The system will be robust to latency, and is therefore applicable where control over large distances are beneficial, for example over the internet.

This is a Master thesis conducted at the department of Engineering Cybernetics at NTNU. It was carried out in the spring semester of 2015.

Trondheim, 2015-06-01

.....

Anders Strand

Acknowledgment

I would like to thank my supervisor Amund Skavhaug for continous support and guidance during my work on this paper.

A.S.

Executive Summary

This paper presents a "proof of concept" system for assisted control of indoor UAV's. The system is designed to handle the most active and demanding tasks required to fly a UAV indoors, which is attitude and position control. It provides an easy to use control interface to the remote operator. The implemented solution is based on distance measurements in six directions using ultrasonic range finders, and a camera stream. It is implemented on an embedded Linux computer and a microcontroller unit, all mounted on a quad rotor multirotor platform. The complete system was built and tested, from the low level sensors and microcontroller, trough the embedded Linux computer to a web interface presented to the remote operator. The system proposed and implemented in this paper did not provide a sufficiently accurate position controller in order to offload the remote operator during flight. The implemented system's best application is as a collision avoidance system for indoor UAV systems.

Sammendrag

Denne oppgaven presenterer et konseptsystem for assistert fjernkontroll av innendørs UAV. Systemet skal håndtere de mest aktive og krevende oppgaver knyttet til flyging av innendørs UAV, hvilket er posisjonsregulering. Det skal ha et brukergrensesnitt som er lett å bruke, slik at flyging blir en enkel affære for "piloten". Den implementerte løsningen er basert på avstandsmålinger i seks retning ved bruk av sonarer, samt et kamera for overføring av bilde. Løsningen er implementert på en kombinasjon av en minidatamaskin med Linux og en mikrokontroller. Systemet er montert på et firerotors helikopter (quadcopter). Løsningen er bygget og testet. Testene viste at systemet ikke kan levere en posisjonskontroll som er nøyaktig nok for innendørs flygning. Systemet vil passe best som et anti-kollisjonssystem for en UAV, hvor posisjonkontroll er håndtert av andre systemer.

Contents

Chapter 1

Introduction

1.1 Background

Unmanned Aerial Vehicles, or UAV for short, comes in many shapes and sizes. Fixed wing UAV's are small airplanes with good range capabilities, and are used in both military and civilian applications. They are typically used for reconnaissance and inspection purposes, equipped with cameras and other sensors. Fixed wing UAV's are not new, but their use in military campaigns have seen an increase in later years. Multirotor UAV's is a newer concept, and has a separate set of features compared to the fixed wing. It is a helicopter with four, six, or even eight rotors. It's biggest strength is the ability to take of and land vertically, as well as being able to hover stationary over time. While a fixed wing UAV creates lift with its wings, a multirotor UAV must create all lift with its upward-facing rotors. It is therefore not as energy efficient, and have significantly lower flight times compared to the fixed wing UAV. It is a mechanically simpler vehicle, and can

Figure 1.1: Fixed wing UAV(left), quad rotor UAV (right)

be built with a variety of materials and sizes. The most common variation of the multirotor UAV is the quadrotor, commonly known as the quadcopter. It has, as the name suggests, four rotors working together to control the vehicle. All multirotors are inherently unstable during flight, and require controller algorithms to fly. These algorithms typically run on a microcontroller or small embedded computer, called a *Flight Controller (FC)*. These flight controllers have traditionally been expensive, but both the the price and size required for high processing power has decreased steadily in recent years. This causes the multirotor to be much more attractive for both hobbyists and professionals, and their popularity have exploded in a couple of years.

It is safe to say that the use of UAV's gradually will spread to new areas, and with this comes new challenges when it comes to designing the UAV. They can have a significant weight and speed, as well as big rotors spinning with several thousand rounds per minute. It is clear that a flying UAV poses a safety threat for humans and animals on the ground. They must therefore be designed with fault tolerance to motor failures, as well as external factors like harsh weather and obstacles in the flight path. This is especially true if the UAV is to fly in urban or indoor environments. A collision with a power cable hanging between two houses can cause a fatal crash, potentially into the street below. The UAV must therefore be aware of its surroundings, and this is not always an easy task. Too much sensory equipment can add a lot of weight, and may require extensive computing power. This is especially true for solutions using cameras and video processing to identify and locate objects in the environment. A UAV system equipped with state of the art sensor and camera equipment, might still miss important information about the environment. Although object recognition software can be very effective, it is still far inferior to a human operator observing the same thing. It is the authors opinion that remotely operated UAV's will be the first step on the way to autonomous UAV's in indoor environments. In such a system, the UAV should assist the operator with active and demanding tasks like altitude and position holding, as well as provide useful information about both its surroundings and the status and orientation of the UAV itself. The operator should either have line of sight to the UAV, or be provided with a video stream in real time. In the latter case, information about the environment not covered by the camera is very important. A video stream enhanced by distance measurements and identification of potential obstacles will provide the operator with information needed to fly the UAV safely. The UAV can in addition have collision avoidance features,

with minimum distances to walls and objects.

Accurate position information is required in order to implement a working position controller. The most accurate method for positioning an indoor UAV, is to mount an array of cameras along the walls of a room. The camera footage is processed to estimate the UAVs position. The UAV is often carrying a beacon which is very visible to the cameras, for example a infrared radiating emitter. A similar method can be implemented with radio transmitters and receivers instead of cameras. Triangulation is then used to locate the UAV. These solutions are not as accurate, and can have issues with line of sight and objects interfering. Both solutions are only applicable where the UAV will fly in a known location, which is a big limitation in many cases. It will also be very expensive to build such a system on a large scale, for example in a factory or warehouse. One of the biggest advantage of an UAV is that is is unmanned. It can fly places where a human cannot, for example because of environmental hazards. Many of the future uses of UAV's will be in places where pre-mounted localization equipment does not exist. It is a big advantage to be able to fly in unknown areas, and still be able to do it safely. In the case of the UAV being remotely operated, position estimates relative to the starting point is sufficient. The UAV does not need to have information about where it is, just its position relative to where is was instructed to hover. The navigation is left to the remote operator. Typical use cases for such a system can be exploration of an evacuated nuclear facility, or inspection of buildings.

Problem Formulation

If the use of UAV's is to expand into indoor environments, more advanced localization and sensory system are needed compared to outdoor flying. When flying well above ground, and without obstacles, GPS will provide a sufficiently accurate localization. GPS data can in most outdoor scenarios be assumed to be available. In indoor environments, the GPS signal will suffer from multipath errors, or in the worst case not be available due to buildings blocking the signal. Even if a good GPS signal is present, the accuracy of the location data is not good enough for flying close to walls or obstacles. A system for accurate and robust indoor position holding is needed in order to safely fly an UAV indoors. The remote operator must be provided with a easy to use control interface, where movement commands can be sent to the UAV. The user interface must

contain all the information the remote operator needs to fly the UAV safely. This includes distances to walls and objects, the attitude and altitude of the UAV, and a real time video stream. The remote operator must be able to fly the UAV indoors, without any previous knowledge of the area which is flown in.

1.2 Brief survey of previous works

The future demand of localization systems for indoor UAV's is discussed in Mui and Chintalapally (2014). It discusses the current solutions for indoor localizations, like a networks of cameras. It concludes that future solutions must move from static equipment mounted in the environment, to solutions mounted on the UAV itself. Protocols for video streaming is discussed in Begen et al.. It evaluates the characteristics of various protocols, as well as the demands of various video streaming applications. It divides common scenarios into push-based and pull-based services. A typical pull-based service is a web video player, while a typical push-based service is real time streaming of video. The theory of encoding and decoding is discussed in Ozer (2014). It presents the basics of compression, codecs and media container formats for video streaming.

Sonar mapping and localization for a non-flying robot is suggested in Tardós et al. (2002). A ring of 24 sonars in a 360 degree configuration is utilized in feature recognition and mapping of surrounding objects. The article compares the sonar solution with a similar laser based solution during the experiments. The main motivation for using sonars are cost considerations, and the goal is to achieve sufficient mapping without lasers. Kim et al. (2002) explores a similar problem. In this article, a robot will stop at each point and do a 360 rotation about its own axis while measuring distances forward. The measurements are used for environment mapping, and provides accurate results in a semi-known environment. Carillo et al. (2011) proposes a combination of inertial navigation and stereo vision for UAV navigation. Position and velocity estimates were fed into the main control loop, and position control is achieved. The stereo vision was provided with two cameras, and the graphical processing was implemented on a netbook processor. The computations proved to be too heavy to achieve real-time performance.

1.3 Objectives

A complete system for remote indoor control of an UAV is to be developed. The first part is developing and building a quadrotor UAV equipped with the necessary sensors and control systems for position hold and collision avoidance. The second part is the remote control system, where the remote operator controls the UAV wirelessly. It provides a control interface displaying relevant information about the environment the UAV is in, including a real time video feed. The whole system should be robust to latency in the transmission between the remote operator and UAV, to facilitate control over slow transmission lines like the Internet or a satellite connection.

1.4 Approach

A standard hobby class quad rotor UAV is used as the platform. Its onboard flight controller will control the attitude, and receive control inputs from the main system developed in this paper. Six sonars and a IMU is mounted on the quadrotor. The six sonars measure distances in the directions; Up, down, right, left, front and backward. The IMU measures acceleration and angle velocity, and combines these into an attitude estimation. A microcontroller unit (MCU) handles real-time tasks like interfacing with sensors, as well as passing control inputs to the flight controller.

The MCU will communicate with a mini-computer running Linux. This device will receive sensor data from the MCU, and create a position estimate. It receives control inputs from the pilot through the control interface, and runs a controller algorithm which decides what control signals to send to the flight controller. In other words, the controller algorithm running on the embedded computer decides how the UAV should move to match the pilots commands. The embedded system transfers environment data and video feed up to the remote interface.

1.5 Outline of the report

The rest of the report is structured as follows. Chapter 2 gives an introduction to the theory and technologies the solution is based on. Chapter 3 presents the proposed solution. It contains the requirements for the different part of the solution, as well as an abstract overview of what mod-

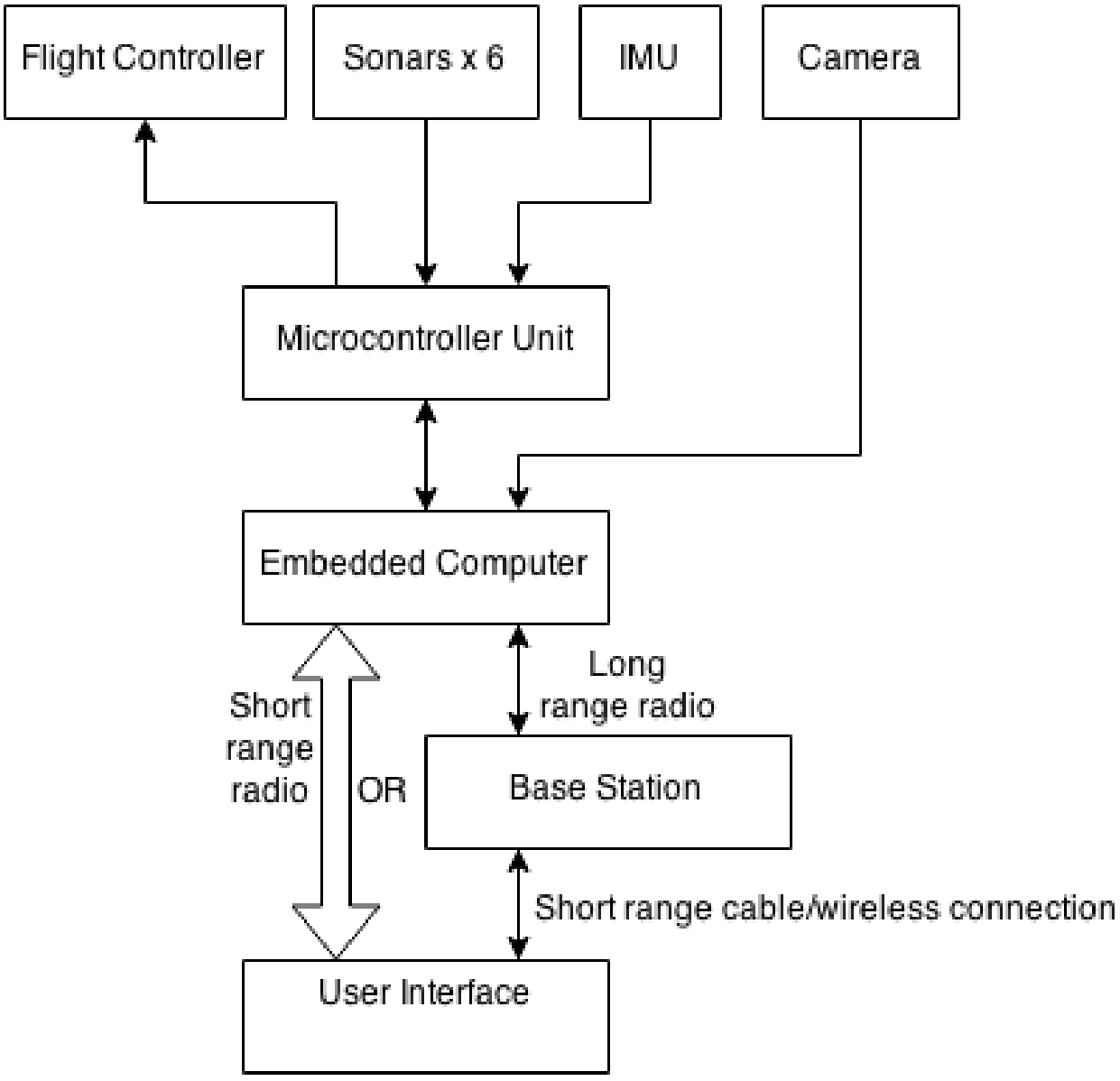


Figure 1.2: Diagram of the approach.

ules the solution must contain. Chapter 4 presents the implemented solution. It describes what equipment was used, and at what basis it was chosen. Software architecture and algorithms is described, as well as the mounting and wiring of the system. Chapter 5 presents the results achieved in testing of the solution. Three tests are presented, each with a different purpose. The first two tests are aimed at testing the sonars themselves, while the last evaluates the solutions performance in flight. Chapter 6 contains a discussion of the implemented solution and its test results. It then sums up the project, and presents the conclusions. The reader is assumed to have a basic knowledge of embedded systems, electronics and control systems. The paper also uses theory from other fields, to which the introduction in chapter two should be sufficient in order to understand.

Chapter 2

Background theory

This chapter gives a brief introduction to the subjects visited in this paper.

2.1 Ultrasonic ranging

Ultrasonic ranging utilizes ultrasonic sound waves for distance measurement. This is sound with frequencies humans cannot hear, normally from 20kHz and upwards. A more common name for ultrasonic ranging devices is SONARs (Sound Navigation and Ranging). This name will be used in the rest of the text. Sonars measure distance by sending out a specific ultrasonic signal with a transducer, and measure the time it takes before the echo returns. Using the known speed of sound in the medium measured in, one can easily calculate the relationship between echo time and distance. This is the same method bats and dolphins use to supplement their vision. Although compared to our sonar technology, these animals are able to process the echo signal at extreme accuracy and speed.

2.2 Attitude estimation

The attitude of an object describes how it is oriented in space. It is commonly expressed in euler angles, with degrees or radians as the unit. We estimate the roll, pitch and yaw of an UAV. These values represent the rotation about the Y,X and Z axes respectively. The coordinate frame used in this project is the NED-frame (North East Down). This frame is defined relative to Earths

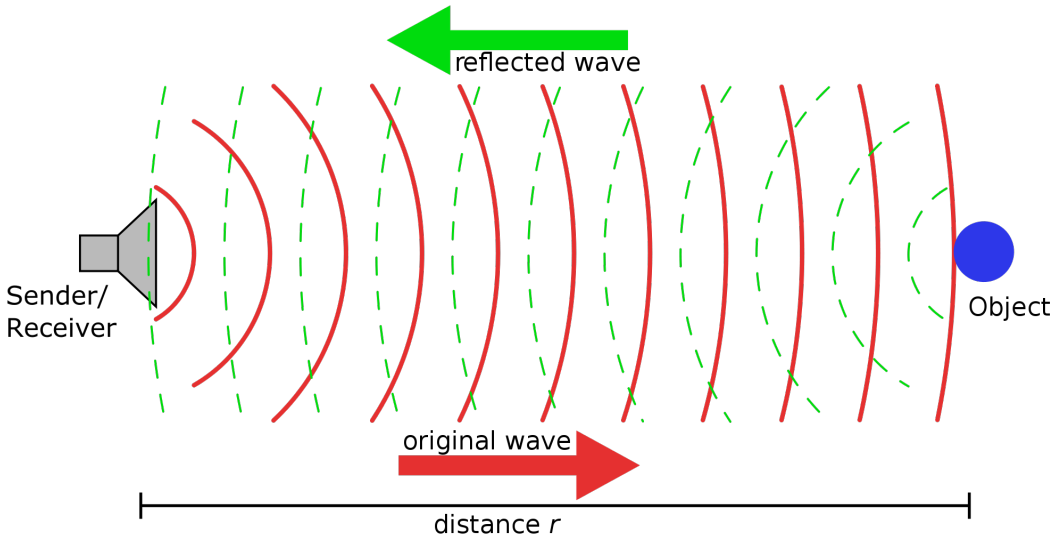


Figure 2.1: Measurement principle of Sonars. Source: http://www.kerrywong.com

reference ellipsoid, which is a mathematical approximation of the shape of the earth. The X-axis points toward true north, and the Z axis downward perpendicularly to the tangent plane of the reference ellipsoid. The Y-axis points east and completes the right hand orthogonal coordinate system. See figure 2.2 for a visual description of the NED frame. In order to understand attitude estimation, there are a couple of concepts and sensor technologies that must be introduced.

2.2.1 Accelerometer

Accelerometers enables us to measure acceleration along one axis. These sensors can utilize various methods dependant on the application, but only the piezo-electric variant will be described here. These sensor uses the piezo-electric effect to generate a voltage, where the voltage is directly related to the force acting upon the sensor. Piezo-electric accelerometers can be manufactured in small and inexpensive integrated circuits (IC's), and is therefore a good choice for this project.

2.2.2 Gyro

A gyroscope is a tool used to measure angle velocity. There exist several gyro types, based on different physical principles. A mechanical gyro has a disk spinning along an axis, and utilizes the

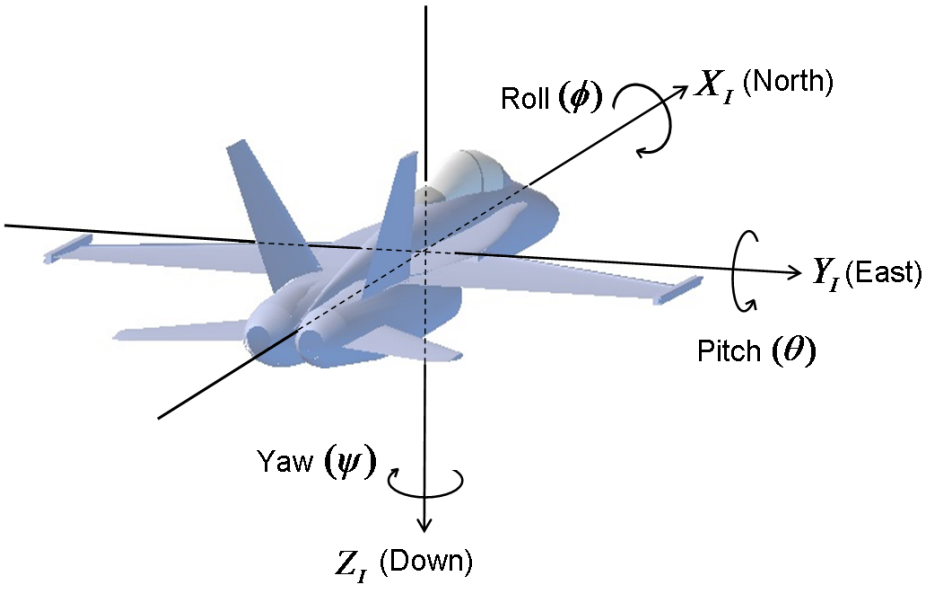


Figure 2.2: The North East Down (NED) frame.

law of preserved momentum to maintain its orientation. An optical gyro sends a pulse of light trough a circular path, where the light it split in such a way that two beams propagate around the circle in opposite ways. If the circle has an angle velocity, this will be detected by measuring the timing difference between when the two light beams arrive at their destination. The gyro technology most relevant to this paper is the MEMS (Micro-Electro-Mechanical-System) gyro. It consists of a electronically driven resonator, which is affected by the coriolis effect proportional to its angle velocity. The coriolis force is measured with a piezo-electric element.

2.2.3 Combining Accelerometer and Gyro data

The attitude of a static body can accurately be calculated using accelerometer measurements in the X,Y, and Z axis, combined with basic trigonometry. This is possible because one only has one acceleration vector to consider, which is the gravitational acceleration vector. When other acceleration vectors are introduced, these will all be measured by the accelerometers. It is no longer possible to isolate the gravitational acceleration. Well calibrated gyros can be assumed to have accurate angle velocity measurements during movement. They will on the other hand have drift while no movement in present. We can see that the accelerometers and gyros comple-

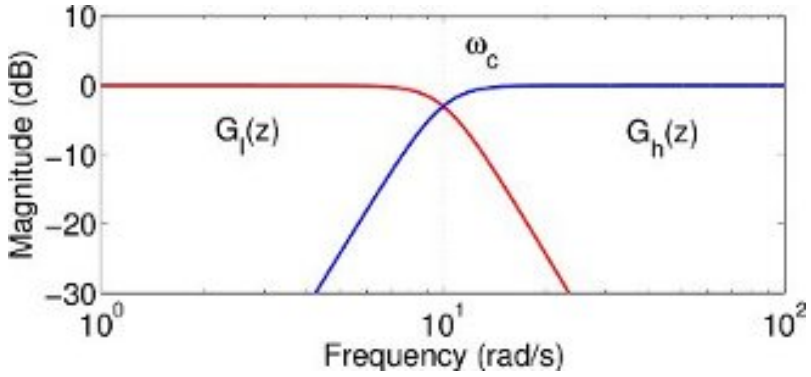


Figure 2.3: The gain of the two inputs of a complementary filter, in relation to frequency (ω_c)

ment each other well. They are therefore used together in what is called a complementary filter. The filter uses the accelerometer during small movements, and brings in the gyro when more movement is involved. The accelerometer estimate is low pass filtered, while the gyro readings are high pass filtered. The tuning of the complementary filter revolves around placing the point where the gyro takes over from the accelerometer and vice versa. This point is often called the c*ross-over frequency*. In figure 2.3, the gyro is the blue line($G_h(z)$), and the accelerometer the red line($G_l(z)$).

2.3 Quad rotor multicopter

The quad rotor multicopter, or quadcopter, can be built in a variety of materials and sizes. They all have the same basic features when it comes to control principles and mechanics.

2.3.1 Mechanics

The four motors are mounted in a square formation. Normally there is equal distances between all motors, but some models have the front motors further back to avoid a front mounted camera to see the propellers. Two motors spin in the clockwise direction, and two spin counter clockwise. This is in order to cancel out each others spin. See 2.4. The frame is either shaped like an X or an H, and materials range from light wood to carbon fiber and aluminum. A wood frame does not have the same strength to weight ratio of carbon fiber, but is both inexpensive and have very good vibration dampening properties. Plastic is also commonly used for the frame, espe-

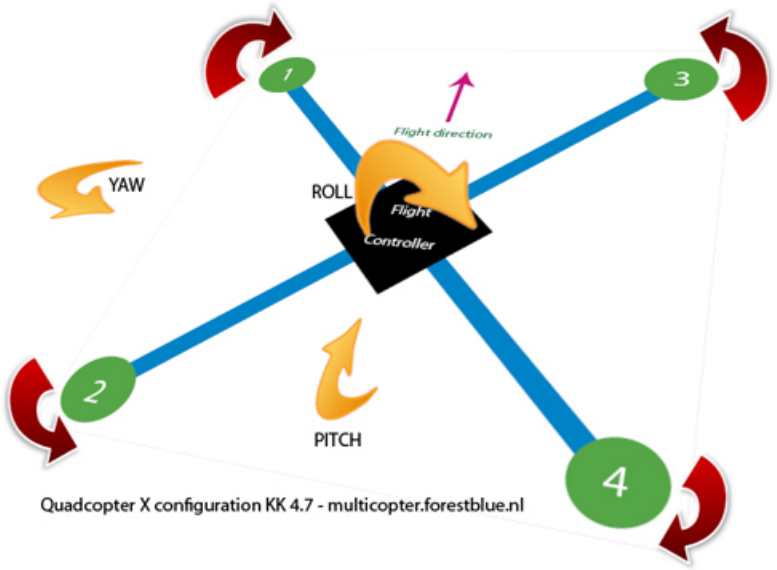


Figure 2.4: Visualization of the roll, pitch and yaw as well as motor spinning directions. Source: http://multicopter.forestblue.nl/

cially on models under 400 mm in diameter. A symmetric and stiff frame is important for the flight characteristics of the quadcopter. An X-shaped quadcopter typically consists of four rods connected on a small base in the middle. The H shaped version can have a bigger midsection, allowing for more equipment to be mounted. Figure 2.5 displays these two quadcopter types.

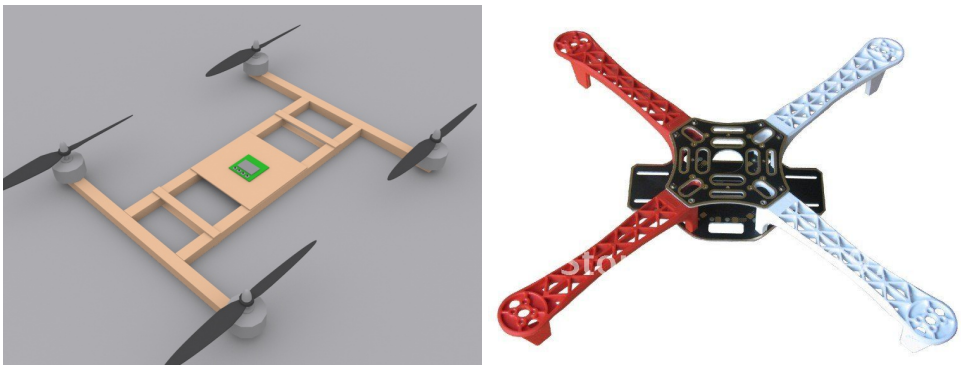


Figure 2.5: Quadcopter frames. Notice the amount of area for mounting equipment on the H frame (left). Source: http://stigern.net/

2.3.2 Electronics

The electronics of a quadcopter include the battery, the ESC (Electronic speed controller) and the motors. Most vehicles will use a litium polymer (LiPo) battery. The battery for a small quadcopter can typically have 1000 mAh of capacity, and have a C-rating of 20-30. The C-rating describes the current output the battery is capable of. The output current in mA is calculated by multiplying the C-rating with the battery capacity. The battery mentioned above will have a constant current output of 1000*20 = 20000 mA, and a peak output of 1000*30 = 30000 mA. A bigger quadcopter with a frame diameter of 500mm will typically require a battery of 2200-3000 mAh and 20-30C. Most batteries have one, two, three or four LiPo cells. This is denoted 1S, 2S, 3S or 4S. Each LiPo cell has a nominal voltage of 3,7 Volts. A 3S battery is most common, as its voltage is about 12V when fully charged. The battery of a multicopter has a significant weight compared to other equipment, and the battery size must be chosen based on the required flight time, minimum current output and weight requirement. The mounting of the battery is often the deciding factor when placing the center of gravity (CG) of a quadcopter.

The motors of most quads are brushless three-phase electronic motors. Many smaller quads use DC brush motors, but these are not as efficient on larger vehicles. The motors are denoted by their KV-rating. This is a ratio between the applied voltage and how many rounds per minute (RPM) they spin. A motor with a KV-rating 2000 will spin 2000*12 = 24000 RPM on 12 V. This is a high KV motor, and is mostly used on medium and small sized quads which cannot mount large propellers. A quad with 250 mm diameter can typically mount 5 inch propellers, and need high RPM to generate enough thrust. A more normal quad with 400-500 mm diameter will typically use ten inch propellers. They can use motors with a KV-rating all the way down to 1000. This motor/propeller combination is much more energy efficient than the high frequency ones.

The ESC takes the DC current from the battery, and generates three phase AC current for the motors. The ESC is an important part for the responsiveness and stability of a quad, and should be of good quality. It must handle currents up to 30-40 Ampere, which requires good electronic components. The quad needs one ESC per motor.

2.3.3 Flight controller

The flight controller is responsible for making the quadcopter fly. It typically takes input commands from a remote controller. These commands contains input variables for roll, pitch, yaw and throttle. The standard protocol for communication with a flight controller is pulse width modulation (PWM). The format is a pulse between 1000 and 2000 us, with a frequency of 50Hz. The flight controller uses gyro's and accelerometers to estimate attitude and measure angle velocity. The flight controller implements controller algorithms for both angle velocities and attitude. Most models have self-leveling capabilities. The algorithms typically run on a micro controller unit. The most basic ones can use 8bit 16Mhz units, while the more advanced utilize ARM cortex chips with 600-1000 mHz frequency. The output of the controller algorithms needs to be mapped to the motor setup used. If the output is to move forward (negative pitch), the back motors will speed up, and the front ones speed down. This ensures that the total thrust does not change. To adjust roll, the motors on the left and right sides speed up and down according to the wanted effect. Yaw movement is done by speeding up and down the pairs of clockwise and counterclockwise spinning motors. A dominance of the clockwise motors will move the quad clockwise, and vice versa. The motor output is sent to the ESC's using the same PWM protocol mentioned above. This signal may have a higher frequency then the input signal, ranging from 50 to 300 Hz.

2.4 Video streaming

This section will give a brief introduction to video streaming techniques.

2.4.1 Compression

Most files are originally raw data, coming from a camera or a microphone. The raw data contains all information captured, and does have a large file size. Compression is a technology used to reduce the size of such files. Common compression technologies for video are H.264,MPEG-2 and WMV. For sound, MP3 and AAC are the most common. Images are typically compressed into JPEG or PNG format. Compression can be viewed as a strategic removal of data from the

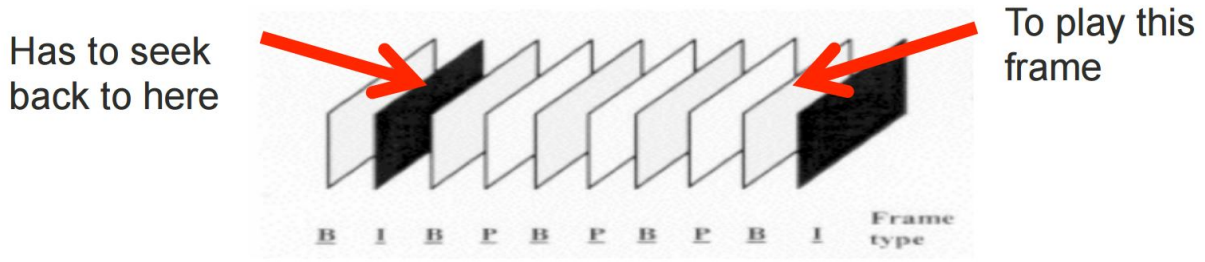


Figure 2.6: A typical encoding scheme. The key frames are denoted "I". Source: http://www.streaminglearningcenter.com/

file. The harder the compression, the more data is removed and the smaller the resulting file is. The key is to remove redundant data, so that the overall quality of the file decreases a minimal amount. The compression technologies are often referred to as "Codecs". The word is short for enCOding /DECoding. The encoding is the compression of the file, where data is removed. In order to restore the file in order to play it, it must be decoded. The decoding device/program must know the codec is was encoded with, in order to successfully restore it. The basic method of the most common video encoders is this:

1. Create a base frame
2. Find the differences between the base frame and the next frame
3. Store only the differences, not the entire next frame
4. Repeat n times before creating a new base frame

Figure 2.6 shows how the interval between key frames impact the time it takes to decode a frame. This means that both the encoder and decoder of such a video will have to look on several frames at the same time in order to process the video. This introduces a latency, which grows with the distance between each base frame. The bandwidth on the other hand, benefits from large intervals between base frames. This is because the base frames contain large amounts of data, compared to the frames in between which only contains the changes compared to the base frame.

2.4.2 Resolution and bandwidth

The resolution of a video file describes how many pixels each frame of the video contains. Each pixel takes up a certain amount of space or bandwidth, depending on the bit depth of the video. The bit depth is the number of bits used to describe the color of each pixel in a frame. The choice of resolution for a video should be based upon the screen size of the viewing device. If the resolution is too small, it will appear "pixelated" and have overall bad quality. On the other hand, too high resolution can exceed the available bandwidth, and the video stream will stutter or pause. The compression of the video is often set with a constant output bitrate. In this case, a video with small resolution will be compressed less than a video with large resolution. This means that the smaller resolution video will be of higher quality, but smaller in viewing size. Compression rate, resolution and bandwidth is three factors which impact each other, and one must find a compromise between them.

2.4.3 Delivery protocols

When streaming digital video through a network, the choice is normally between two types of protocols. The divide is whether or not the protocol features reliable packet transfer or not. In the transport layers, we choose between UDP and TCP. TCP (Transport Control Protocol) does, as the name suggest, feature transport control. This means that the protocol will keep tabs on the packets transferred, and take action upon packet loss. It implements ARQ (Automatic Resend reQuest), when packets are missing. When this occurs, the following packets will be delayed until the current packet is transferred successfully. UDP (User Datagram Protocol) is not reliable, but rather a "best effort" protocol. With UDP, the packet is sent out in the network, and then the next packet is sent straight after. UDP does not concern itself with checking whether or not the packets arrive safely on the receiver end, or if anyone receives them at all. Because of this, sending packets with UDP will give a constant packet rate with no reliability. Using TCP gives reliability, but the data flow may be interrupted by resending old packets. In systems where realtime video is required, a protocol called RTSP (Real Time Streaming Protocol) is often used. RTSP is based on UDP, but customized for video streaming. For video streams between a stored file and a web player (for example a youtube video), TCP is used. In this scenario, the player can

download a certain amount of the video over TCP before starting the playback. In other words, the quality and completeness of the video is more important than the time it takes to play it.

Chapter 3

Proposed solution

This chapter will describe proposed solutions to the problems presented in chapter 1. It will not go into detail about specific hardware or software, but will present an abstract overview of the solution.

3.1 System overview

The sensory inputs of the system includes an IMU, six sonars, and one camera. The IMU and sonars are interfaced to a microcontroller unit, while the camera is connected directly to a embedded linux (henceforth eLinux) system. The sensor data is passed on to the embedded linux computer through a serial interface. It will use this data to estimate relative position information, and use controller algorithms to compute the appropriate actions. The remote operator interface receives environment information and a video feed from the eLinux computer, and sends back control commands from the pilot. The outputs of the controller algorithms are set-points for how the vehicle should move to react to the remote operators input. These output values are sent through the serial interface the the microcontroller unit. Here it is translated into the PWM signals described in chapter 2, and sent to the flight controller board mounted on the vehicle. The UAV is equipped with a radio for communication to the remote operator.

Microcontroller subsystem The microcontroller unit is used because it can handle real time tasks like pulse width measurement and embedded communication protocols more effectively

than a higher level computer. The microcontroller will have four tasks:

1. Read six sonar values
2. Read IMU values
3. Communicate with the eLinux system
4. Generate PWM output to the flight controller

A standard 8 bit microcontroller can be used. it must have at least six GPIO (General Purpose input/output) for the sonar reading. The pwm output for the flight controller requires additional four. It must support serial communication through UART, as well as TWI (two wire interface) or SPI (Serial Peripheral Interace). Its tasks are not computationally heavy, but a tight loop timing is required. The sensors should be read every 100 ms, or with a 10Hz sample frequency.

3.2 Embedded linux subsystem

The main reason for using a linux system is the video transmission. Video processing is far to intensive for a small microcontroller. in addition, a linux system makes a variety of tasks simple. For example utilizing WiFi, or hosting a webserver for a remote interface. Two main programs will run on the eLinux system, the backend and the frontend. The backend takes care of communication with the microcontroller, position estimation and vehicle control. The frontend takes care of the remote operation communication. The frontend and backend can be two separate processes or two threads in the same program. This affects how they communicate. It is optimal to separate these two programs, so that a failure in one does not take down the other as well. The backend program will be a periodic program, while the frontend will be event-driven. The events will typically be user input from the remote operator., but can also be new data from the backend.

3.3 Radio communication

This can be a long or short range radio link, either directly to the operator, or to a base station. In the latter case, there is many possibilities to how the remote operator can connect to the

base station. It can be trough a radio system, a direct cable, a satellite connection or even the Internet. This is illustrated with the two different paths between the embedded computer and the user interface in figure 1.2.

3.4 Quadcopter platform

The quadcopter platforms main task is to carry equipment. It must therefore provide enough lifting thrust to take off and have a stable flight. A common rule of thumb is that the quadcopter should provide twice as much thrust as the gravitational force acting upon it. The ample thrust is there to ensure enough for control and stabilizing during flight. The quadcopter must also be big enough to fit the equipment needed. The size of the quad determines how big propellers that can be mounted. A bigger propeller can generally be assumed to deliver a bigger thrust, as long as the power supply chain can deliver enough current. A common size for quadcopters are a diameter of 450 or 550 mm. For indoor flight, this is in many cases too large. A quadcopter of this size is also hard to transport without disassembly, a factor that is important in many applications. Many mini-quads have a diameter of 250 mm. These are very nimble and agile platforms. but might not provide enough lifting thrust for some applications. A size between 300 and 450 mm will most likely fit an indoor flight application the best.

The quadcopter platform must provide a very stable takeoff and hover, which comes from accurate attitude control. This task is done by the flight controller, but the symmetry and stiffness of the frame is also very very important in this regard. If the propellers are close to the ground during takeoff, ground effects are significant. Is therefore preferable to have long legs for the quadcopter to stand on. The propellers will then always have a proper distance to the ground. The legs also allow for equipment to be mounted on the "belly" of the quadcopter. This is both practical for mounting, but can also allow for the center of gravity to be placed lower. The lower center of gravity will make the quadcopter inherently more stable.

Chapter 4

Implemented Solution

This chapter describes in detail how to proposed solution from chapter 3 was implemented. It includes hardware and equipment as well as software architecture and algorithms.

4.1 Quadrotor Platform

The quadrotor was built on a 330mm frame. It has a glass fiber hub and plastic arms. The frame was chosen primarily based on its convenient size for indoor flying as well as transport. The model has a simple design, and is constructed for easy mounting of equipment. It comes in many sizes, with a common center hub. Extra parts like landing gear is therefore easy to obtain. The flight controller board chosen was the KK2. This board is very popular and widely used in the hobby community. It is inexpensive (120 NOK), but provides all the features needed in this project. It features a MPU6050 IMU, and the control algorithms are run on an ATMEGA 644PA microcontroller unit. The main advantage of this flight controller is its display and buttons. While most flight controllers must be connected to a computer for configuration, this can be configured quickly and easily directly on the board. The configurations include the gains of the attitude controller, as well as propeller setup and sensor calibration. The opportunity to tune the attitude controller and self level settings between test flights is a great advantage, which reduces the setup time significantly.

The ESC used is the Hobbywing Skywalker Quattro 25A*4. As the name suggests, this ESC can deliver 25 Ampere of current to four motors. It is four ESC's combined in one small pack-

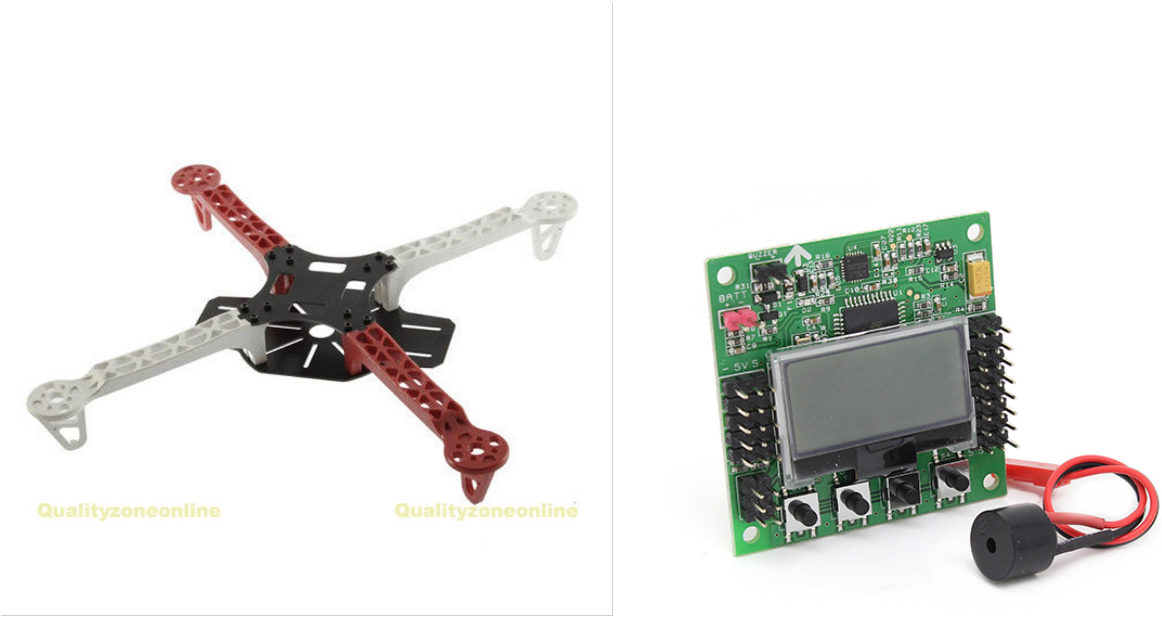


Figure 4.1: F330 frame (left), KK2 FC (right)

age. This is in contrast to the more common approach, which is to have four separate ESC's. The combined package has several advantages. One can easily calibrate all four ESC's at the same time (The calibration sets the minimum and maximum throttle levels). It eliminates the requirement of a power distribution circuit. This circuit would distribute up to 100A of current, and would require excellent soldering and cables. Otherwise it is a potential weak link in the power supply chain, and might even be a fire hazard. Instead, the wiring is cleaner and simpler. Two power wires connect to the battery, four signal wires connect to the flight controller, and three wires connect to each motor.

The four motors are of the type EMAX CF2812. These are rated at 1534 KV, or RPM per Volt. They are equipped with 8.45 propellers. These propellers are eigth inches in length, and have a pitch of 4,5. The pitch is defined by how far the propeller "moves" forward in one 360 degree rotation. It describes the angle in which the propeller blade is in relation to the spin axis. Higher pitch gives a larger thrust, but drains more power. It also requires a stronger motor. The pitch of a propeller can be compared with the gear system of a car. A low pitch (or gear) gives high torque but low top speed, while a high pitch (or gear) gives low torque but high top speed. See figure 4.3. The battery used is a Turnigy 3S 2200 mAh 25-30C rating LiPo. It has three cells (3S), and a nominal voltage of 11.7 V.

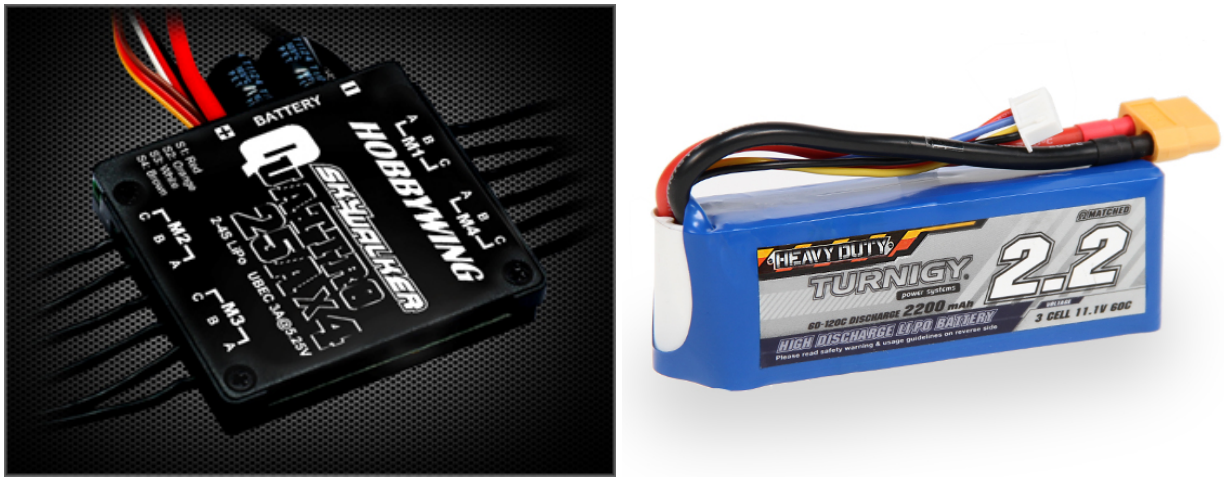


Figure 4.2: Hobbywing ESC (left), turnigy battery (right)

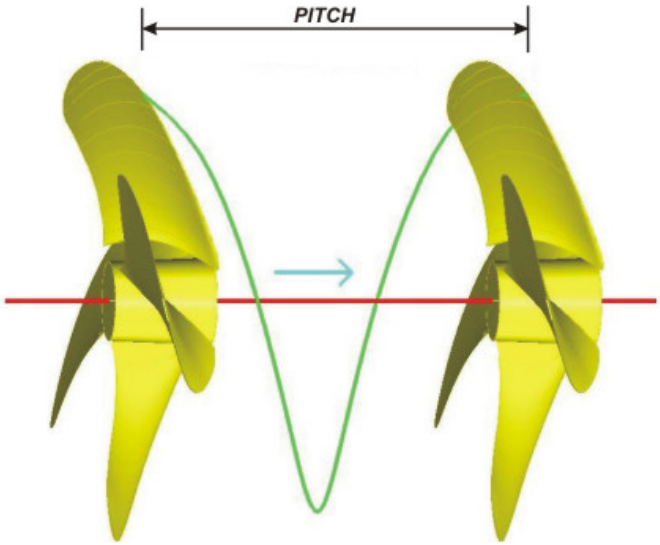


Figure 4.3: Visual explanation of propeller pitch value

Figure 4.4: Quadrotor platform

The quadcopter frame was assembled according to the provided instructions. The ESC was mounted on the top of the center hub, with the flight controller on top. Both these very mounted with velcro tape. The battery was put inside the center hub of the frame, and fastened with a strap. Each motor was mounted on the motor mounts at the end of each arm. The nuts were secured with a Locktite fluid to ensure they would not unscrew themselves during flight. The landing gear is designed to fit the frame, and was simply attached to the bottom plate of the center hub. The propellers were balanced, and then mounted with propeller adapters to the motor shafts. The balancing is done to minimize vibrations from the propellers, which can lead to bad IMU measurements as well as unnecessary tear on the construction.

Figure 4.5: The hcsr04 sonar module

4.2 Sensors

4.2.1 HCSR04 Sonar module

The HCSR04 Sonar module is inexpensive (2-5 dollars) and widely used among hobbyists. It utilizes two separate sonic transducers/receivers to measure distance. The two measurements is combined to one internally in the module. It can measure distances between 2-450 cm, and has a 20 degree spread angle on the sonar measurements. The measurement resolution is up to 30mm.

Using the HCSR04 module is simple and straightforward with a micro controller.

1. Send a minimum 10 μs pulse on the TRIG pin
2. Measure the pulse width returned on the ECHO pin
3. Calculate distance based on the pulse time

See figure 4.6 for a visual description of the hcsr04 timing sequence.

4.2.2 MaxBotix LV-MaxSonar-EZ4 1040

The Maxsonar is a more expensive sonar than the HCSR04, with a price of 30$. It is triggered by a 20 us pulse on the trigger pin. The measured distance can be transmitted using a variable voltage (analog), a serial port or pulse width modulation (PWM). The PWM signalling is identical to the HCSR04, which makes the sensors interchangeable. The MaxSonar has a maximum range

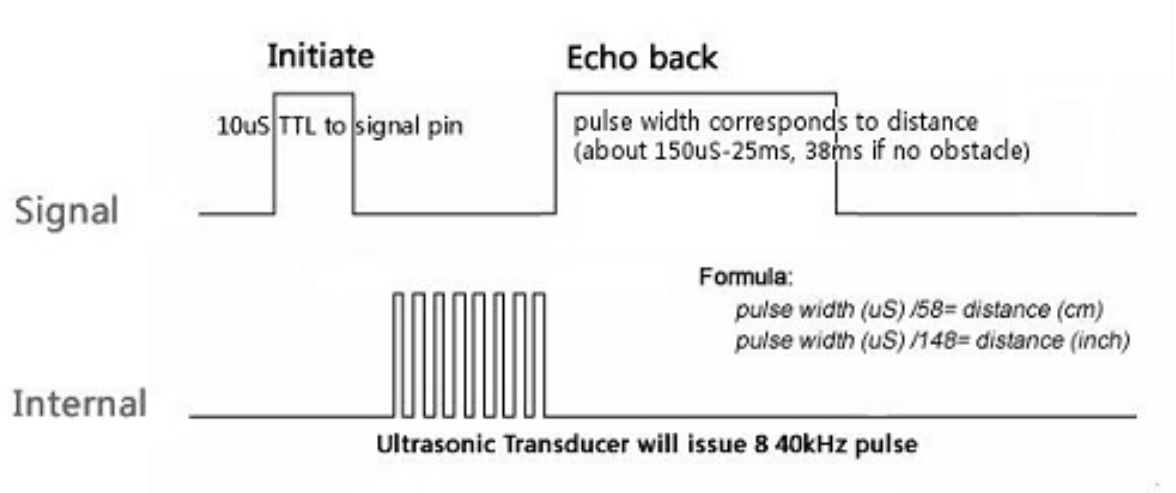


Figure 4.6: The timing sequence for interfacing a micro controller with the hcsr04 sonar module

of 6,4 meters, and an aperture angle of 30 degrees. Only one MaxSonar was used in this project, with the main purpose of comparing it to the cheaper HCSR04.

4.2.3 MPU6050 IMU

The InvenSense MPU6050 is a 6 DOF (degree of freedom) IMU (Inertial Measurement Unit). It has three MEMS gyros, which measures angle velocity about its X,Y and Z axes. It also has three MEMS accelerometers measuring acceleration along the X,Y and Z axes. Communication with the IMU is done with i2c, a very common serial interfacing protocol in embedded systems. The MPU6050 has onboard functionality for sensor fusion and motion processing. This motion processing computes an attitude estimaton. The user can choose between using this, or extracting raw data directly. The former was chosen in this project, to offload processing from the master device.

4.3 Microcontroller Unit

4.3.1 Hardware

The MCU board used was the Ardunio Nano. It is a very small MCU board at 45mm x 18mm. It features an ATMEL ATmega 328 microcontroller, which is an 8 bit MCU running at 16 MHz.

Figure 4.7: MaxBotix LV-MaxSonar-EZ4 1040

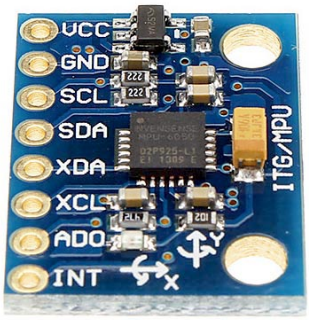


Figure 4.8: The MPU6050 breakout board

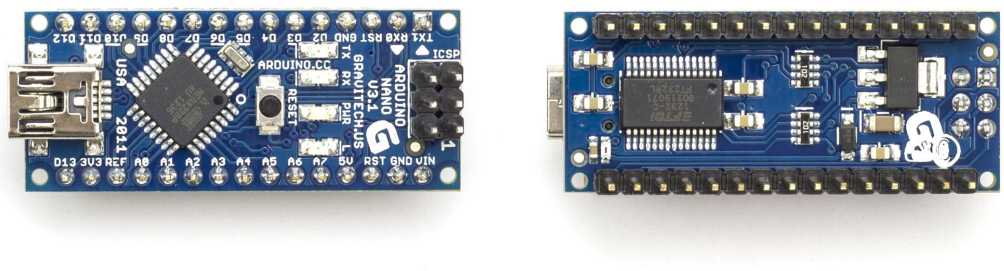


Figure 4.9: The front and back side of the Arduino Nano microcontroller board

The board was chosen based on several factors. The Arduino platform provides an easy to use interface for the proven ATmega micro controllers. They provide a hardware abstraction layer in their coding environment, which severely cuts down on development time. The Arduino boards all provide a USB to Uart interface, enabling simple connection to the PC via USB. The more cumbersome JTAG connection is also available, but generally only used for debugging. The Arduino Nano was specifically chosen for its low weight and size, in addition to a low price. The ATmega 328 satisfies the requirements stated in chapter 3, and there is no reason to opt for a more powerful micro controller.

4.3.2 Software

The software of the Arduino har the following tasks:

- Read IMU values
- Read sonar values
- Send sensor values out on serial
- Read flight controller setpoints from serial
- Update pwm output to flight controller

These tasks are done with a frequency of 10 Hz. Most tasks are executed fast, but the sonar reading has a considerable duty cycle. The pulse width response can be anywhere between 150 us and 38 ms, which is a large part of the total loop time of 100 ms. To ensure the sonar

readings interfere as little as possible with the other tasks, an interrupt driven model is used. An interrupt routine is attached to each sonar echo pin. This routine starts a counter on a positive flank, and stops it on a negative flank of the sonar echo signal. The measured time is stored for each sonar, and can easily be extracted by other code. All sonars are connected to the same trigger pin, which is set high for 20 us to trigger a new measurement. The IMU is read through the i2c protocol. Most of this work is done by the "MPU6050_6Axis_MotionApps20.h" Arduino library. In this setup, the IMU triggers an interrupt line when a new measurement is ready. This is convenient, but not necessary when the reading happens once every 100ms. A new value can be assumed to be ready every cycle at this frequency.

The communication with the embedded eLinux system is done via the serial interface. This connection is done trough the TTL (Transistor<->Transistor Logic) to USB chip on the bottom side of the Arduino Nano board. This chip enables the computer to register the Arduino as a standard COM port. The data is sent with 115200 baud rate, with 8 bit frames with 1 stop bit and no parity. The baud rate is the highest available, and is chosen to achieve high bandwidth. This setting works well, and the connection is very reliable. However, this high bandwidth is not required in this system and should be lowered to keep the theoretic data loss to a minimum. This has not been done in the implementation, as data loss has not been an issue. The serial messages follow a simple format: "TYPE:VALUE;" The various types correspond to the various sensors and outputs. The string "r:1450;p:1500;y:1300;t:1750;" is an example of a string that the Arduino would receive from the eLinux system. It contains the pwm output values for roll, pitch, yaw and throttle respectively, given in microseconds. The string "sb:50;su:180;sd:8;gy:40;r:10" contains values from the backward sonar, upward sonar, downward sonar,y-axis gyro value and roll value. This string is sent from the Arduino to the eLinux system. The string syntax makes is easy to extract data on the receiving end, and also for detecting syntax errors by looking for the separating characters ':' and ';'. The typical message size sent from the arduino range from 60-100 bytes, depending on the number of digits in the values. The eLinux system has ample input buffer size, such that overflow error is assumed not to be an issue. It is however, so large that it will fill up with old data very fast, if it is not read with the same frequency as it is filled. This issue is addressed by tuning the timing of each time it is read, as well as clearing the buffer after each read. This ensures that the values read from the buffer always is the newest values

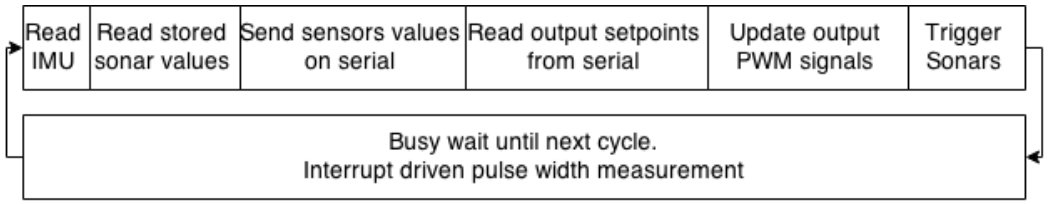


Figure 4.10: The 100 ms duty cycle of the Arduino code

available. In the other direction, the message consists of four values with four digits each. The "type" variable is one character. This leads to a message size of: 4 * type(1 byte) + 4* value(4 bytes) + 4 * ':' (1 byte) + 4* ';' (1 byte) = 28 bytes. The input buffer of the Ardunio is 64 bytes, so overflow is not an issue in this direction either. The serial buffer is read until it is empty every time, so issues with old data is not a problem here either.

The pwm output to the flight controller is generated using the "servo.h" Arduino library. The servo pwm standard of 50Hz 1000us-2000us pulses is also used in most RC components. The library uses one hardware timer unit, and can implement up to 12 pwm outputs using software timers. This is a good example of how one can save time by using the Arduino platform. In order to implement four pwm outputs using AVR code, one would have to study the datasheet extensively, and set up several registers with hexadesimal values. The AVR can generate two pwm outputs per timer unit, and two timer units would have to be used if four total pwm outputs were to be generated.

Figure 4.10 shows how the main loop of the Arduino software is executed. It is no known how long the first six tasks take to execute, but it can be assumed to be under 50 ms. This leaves at least 50 ms of busy wait where the interrupt driver pwm measurement can take place without interrupting the other tasks.

4.4 Embedded linux system

4.4.1 Hardware

The main reasons for including an embedded linux computer in the overall system are the video streaming and front end interface. These tasks require significant processing power compared to a microcontroller, but no as much as a powerful desktop computer. The tasks also benefit

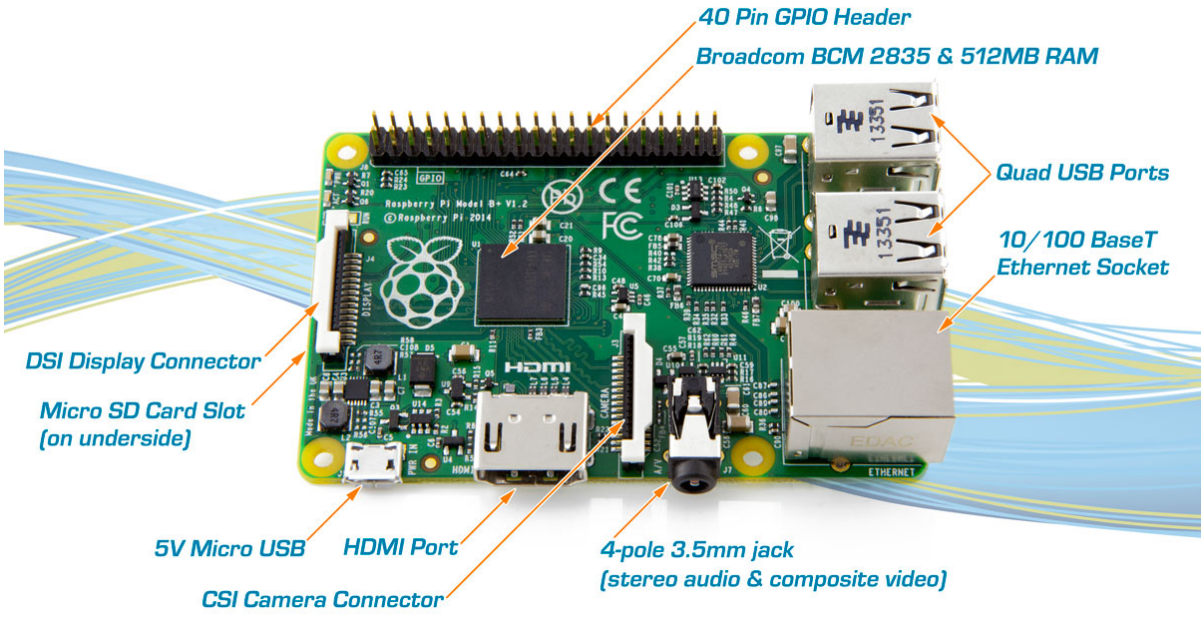


Figure 4.11: The Raspberry Pi B+

greatly from a standard operating system, with a large open source community. Based on this, the Raspberry PI B+ was chosen as the eLinux board. It features a Broadcom BCM2835 SoC (System on Chip), with a 700 Mhz ARM processor and 512 mb SDRAM. It has a small size, with dimensions of 85mm x 56mm x 17mm. This is approximately the size of a credit card. The Raspberry PI has ample connectivity, with 4 USB 2 ports, HDMI, Ethernet, audio jack and 40 GPIO pins. The main advantage of the raspberry pi is the large open source community. There is always someone else who has encountered the same problems before, and solutions are easily available online. The raspberry pi is also inexpensive, with a cost of 30-40 dollars. Some extra equipment is needed to use the raspberry pi. This includes an SD card for the operating system, a 5V micro usb power cable and a protective case. The Ethernet port is normally used to connect to a network or a host computer. In this project, a wireless solution was needed. A USB wifi dongle was therefore installed, to enable connection to the raspberry pi while flying. The dongle used is the Netgear N150 wireless usb adapter.

The camera used for the video streaming is the Playstation Eye 3 usb camera. This camera is designed for the Playstation game console, and is used for motion tracking in games. It does therefore have a very low latency, but at the cost of resolution. It has a maximum resolution of

Figure 4.12: Netgear N150 wireless usb adapter

Figure 4.13: Playstation eye 3 camera

640x480 pixels. It can deliver video of 60 fps (frames per second) at 640x480, or 120 fps at 320x240 pixels resolution. Although this camera was developed for the Playstation, it is compatible with computers as a generic usb camera.

4.4.2 Video streaming

For setting up the video stream, an open source multimedia framework called Gstreamer was utilized. Gstreamer is a powerful framework, which can be used for multimedia encoding/decoding, playback and mixing among other things. In this project, Gstreamer was used to encode the

video from a usb camera, and host a video streaming server. The video was encoded as MJPEG, which in principle is a series of JPEG pictures. The stream is transported the the UDP protocol, since the video presented to the remote operator should be as near real-time as possible. The MJPEG encoding requires little to no processing, as the images are sent out as they arrive. This is more bandwidth intensive than other protocols like h264, but latency was prioritized. In addition, a small resolution of 320x240 was used to ensure that bandwidth issues would not increase the latency between the camera and the remote operator. The video was encoded at 30 fps. The UDP protocol does not implement reliable transfer of packets. In other words, it does not check for and react to packets which does not reach the receiver. This is a very important for a real time video streaming protocol. Another popular protocol, TCP, would try to resend packets which got lost. This would take time, and further delay the following packets. These packets could be so old that their information no longer is valuable to the receiver. For a real-time video stream, latency is more important than reliability. It is not critical that all packets arrive, as a remote operator can easily overcome the occasional loss of video frames. The UDP video stream is sent to a IP and port specified in the Gstreamer server configuration. It can be played back by a range of players. For example in a web browser, smartphone application or a video player on a laptop.

4.4.3 Software

The Raspberry PI's operation system is called Raspbian. It us a Debian linux based OS optimized for the Raspberry PI. It comes with over 35000 pre-compiled software packages, which makes it easy to get started with many different tasks. Examples of such tasks from this project is implementation of a webserver, and setting up a wifi access point using a third party usb wifi adapter. The wifi AP (Access Point) is set up using a package called hostapd, and the dhcp server used is called isc-dhcp. Both these package are availiable trough the apt-get package manager bundled with Debian. A guide to setting this up will be included in appendix C.

Frontend software

The frontend part of this project was implemented as a web interface. The web server is hosted locally on the Raspberry PI, and clients can connect trough the wifi network also hosted on the PI. A web server can be implemented in a number of ways. For this project, a framework called

```
from flaskServer import app
from flask import Flask, request, redirect, url_for, abort, flash, render_template
import time, thread, os, sys, inspect
currentdir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe())))
parentdir = os.path.dirname(currentdir)
sys.path.insert(0,parentdir)
#app.debug = True
import upstream
import downstream
@app.route("/imu")
def getIMU():
        upstream.lock.acquire(True)
        returnString = str(upstream.imu.pitch)
        returnString+=" "+ str(upstream.imu.roll)
        returnString+=" "+(str(upstream.imu.yaw))
        app.logger.debug(returnString)
        upstream.lock.release()
        return returnString

@app.route('/')
def mainPage():
        app.logger.debug('A value for debugging')
        return render_template('index.html', )

@app.route('/buttons/<button>',methods=['GET','POST'])
def atButtonPress(button):
        app.logger.debug('Button %s pressed' % button)
        upstream.lock.acquire()
        upstream.yaw=button
        upstream.lock.release()

        downstream.lock.acquire()
        downstream.state = button
        downstream.lock.release()
        return redirect (url_for('mainPage'))
```

Figure 4.14: A basic webserver configuration for reacting to button presses and retrieving IMU values

Flask was chosen. It provides an easy to use python framework, where basic building blocks are used to create a web server. Because it runs on python, it is easy to connect this webserver to other python programs. The flask server is created by mapping a function to a specific url. When this url is accessed, the mapped function is run. This function can do whatever, but in most cases is will return something to the client's web-browser.

In figure 4.14 one can see a very basic flask server configuration. The upstream and downstream objects are python files containing variables, which is imported as normal objects in the code. These objects are shared between the frontend and backend threads, and protected by locks. The upstream objects contains sensor values and other information coming from the backend system. The downstream contains commands coming from the frontend web server. The locks work as binary semaphores, and are operated using the acquire() and release() func-

tions. The @app.route(url) functions map an url to the function beneath it. That is, when a web browser makes a HTTP request to this address, the function is executed. When someone enters the address of the website in their browser, the flask server routes this request to the function mainPage() on line 21 of figure 4.14. This function returns a html file to the browser. This html file is the main page of the site, can contains buttons for the user to push. These buttons are programmed to send http requests to the "/buttons/" url. A button called "down" sends requests to "/buttons/down/", and a button called "up" send a request to "/buttons/up". The last part of the url is used as a variable on the function, which updates the state variable of the downstream object with the command. This is seen on line 26.
A finished web interface was not completed in this work. Figure 4.15 displays a proposed interface, which includes the features discussed in chapter 3. A camera feed is the main element, which is augmented by colors on all sides indicating distances in those directions. The red color on the right side indicates that the distance of 42 cm is less than a set safety threshold. The lower third of the interface shows attitude and altitude, as well as information about battery capacity and distance measurements.

Backend software

The backend software includes communication with the Ardunio, position estimation, flight control and communication with the frontend. This software is also built using python. Python was chosen to enable easy communication with the front end, but also for the author to gain experience with the language and framework. It has a large community, and a plethora of available modules which are easily imported to a project. This makes python an excellent platform for rapid software development, much like the Arduino does for microcontrollers. Python is best used with an object oriented approach to software design. As with the Arduino, ease of use comes at the price of less control. Compared to using C or C++, the code writer has very little control over how various things are implemented. Python is therefore not recommended if the project has strict requirements to timing or memory use.

The backend software has a main loop which is executed every 100 ms, just like the Arduino main loop. It makes use of three modules to operate: The Serial handler, the Sonar position estimation system and the flight control system. An overview of the backend software modules

Figure 4.15: Proposed web interface for remotely piloting the UAV

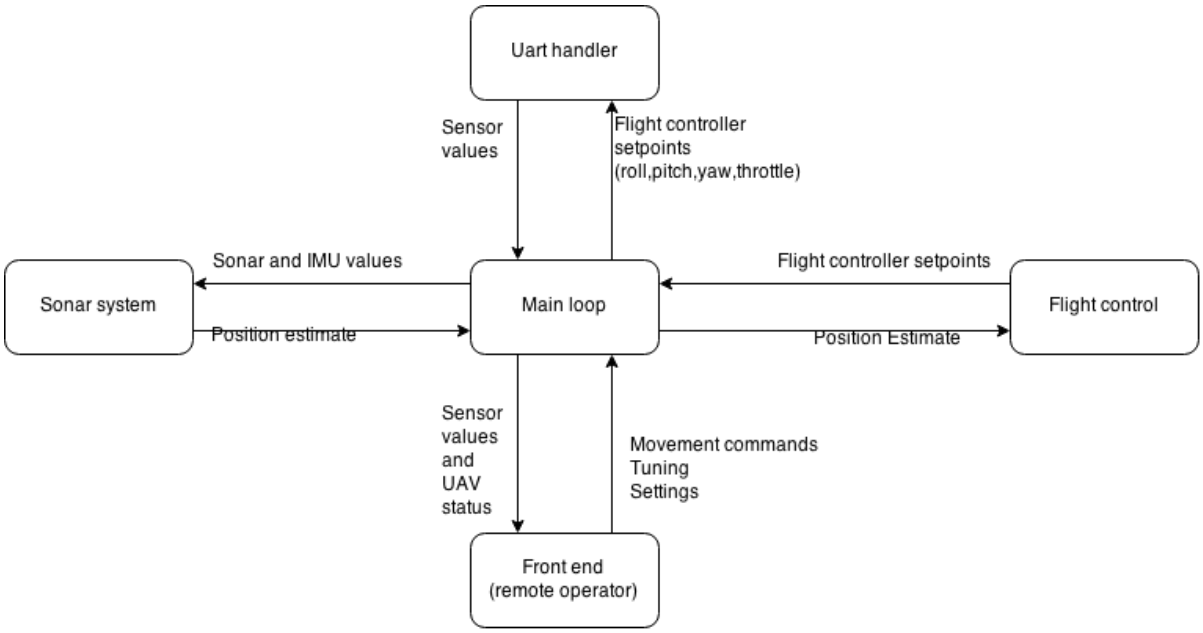


Figure 4.16: The structure of the backend software controlling the UAV

and what data is passed between them can be seen in figure 4.16.

Serial handler This module provides a class for all serial communication. It uses the "pySerial" library to access the COM port of the Raspberry PI. The class provides methods for adding commands for the Arduino, as well as reading sensor values from the Arduino. All parsing of incoming serial messages, as well as construction of outgoing messages is done within this class. The main backend loop can create an object of this class, and easily communicate with the Arduino trough the provided functions.

Position estimation system The basic class for the position estimation system is the sonar class. Each object of this class represents one sonar mounted on the UAV. The class keeps track of the sonars value, average value, standard deviation and validity value. Functions are provided for updating the sonars value, as well as evaluate its parameters after each update. The sonars are pairwise created in objects of the sonarPair class. Three objects of the sonarPair class exist, one for each axis X,Y and Z. This class evaluates the two sonars measuring along the same axis, and combines the two values to estimate the position along that axis. The three sonarPair objects are created in the SonarSys class. This class provides a clean interface for using the position

estimation system. It has one function, where all sonar values are updated with new values, and a new position estimate is calculated. The position estimate is extracted by simply accessing the variables, which are public in the class. See figure 4.17 for an overview of the classes of the back-end software. Race conditions are not an issue, as the functions reading and writing to these variables are executed sequentially. The position estimation algorithm is discussed in further detail further down in section 4.5

Position control The position control modules states can be divided in two main types, movement and hovering. During hover, the UAV should not move in any direction, or rotate about any axis. This is often called position hold, which is implemented with a position controller. The movement states include movement forward, backward, up,down,left,right as well turning the UAV left or right. The position controller contains one PID controller for each axis. The altitude controller is running in all states, in order to maintain the correct altitude. Movement in the Z axis (altitude) is done by setting the setpoint for the altitude controller either above or below the current position. The output of the Z axis controller is the throttle setpoint output to the flight controller. For other movement, a pitch,roll or yaw setpoints is sent to the UAV's flight controller. The setpoints are set a static value away from the center, this static value is the movement speed and can be tuned. If one want to move left, a roll value of 1400 can be sent to the flight controller. This will create a 1400 us pwm signal at the Arduino, which is connected to the flight controller mounted on the UAV. The middle pwm value is 1500us, which tell the flight controller to stay level at 0 degree roll. If a value of 1400 moves the UAV left, a value of 1600 moves it with the same velocity to the right. The same principle applies on pitch and yaw. In the hover state, the position controller decides all output. In X and Y directions, the PID output will be used as long as the position estimate is deemed valid. If not, the middle value of 1500 us is used as output. In the Z axis, the PID controller is also only used when the Z position estimate is valid. In the other case, the throttle output is either incremented or decremented with a static value each iteration. This feeds forward the users control input while the position controller does not have a valid position estimate to base its calculations on. The method is described in figure 4.18.

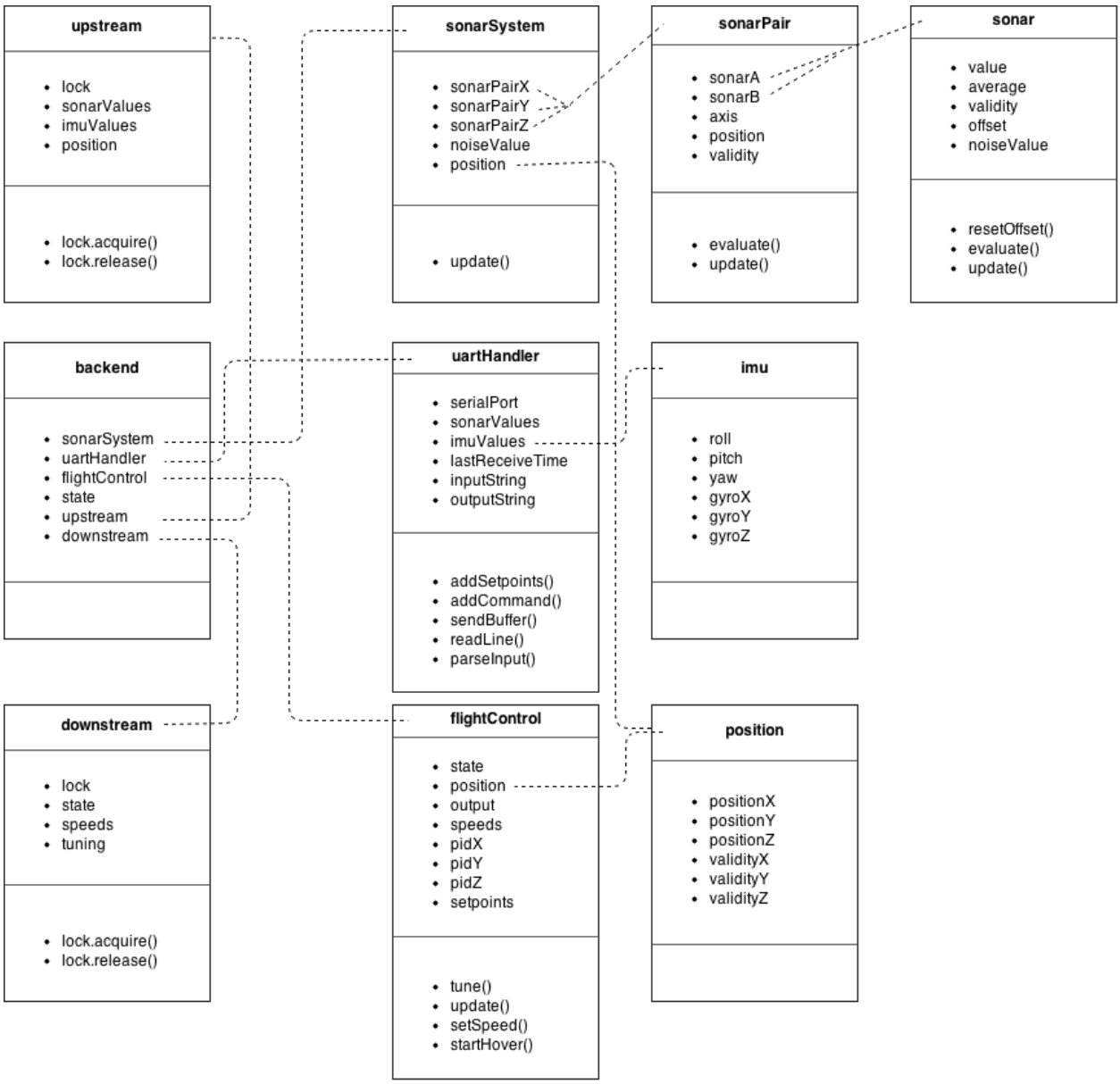


Figure 4.17: Class diagram for the backend software

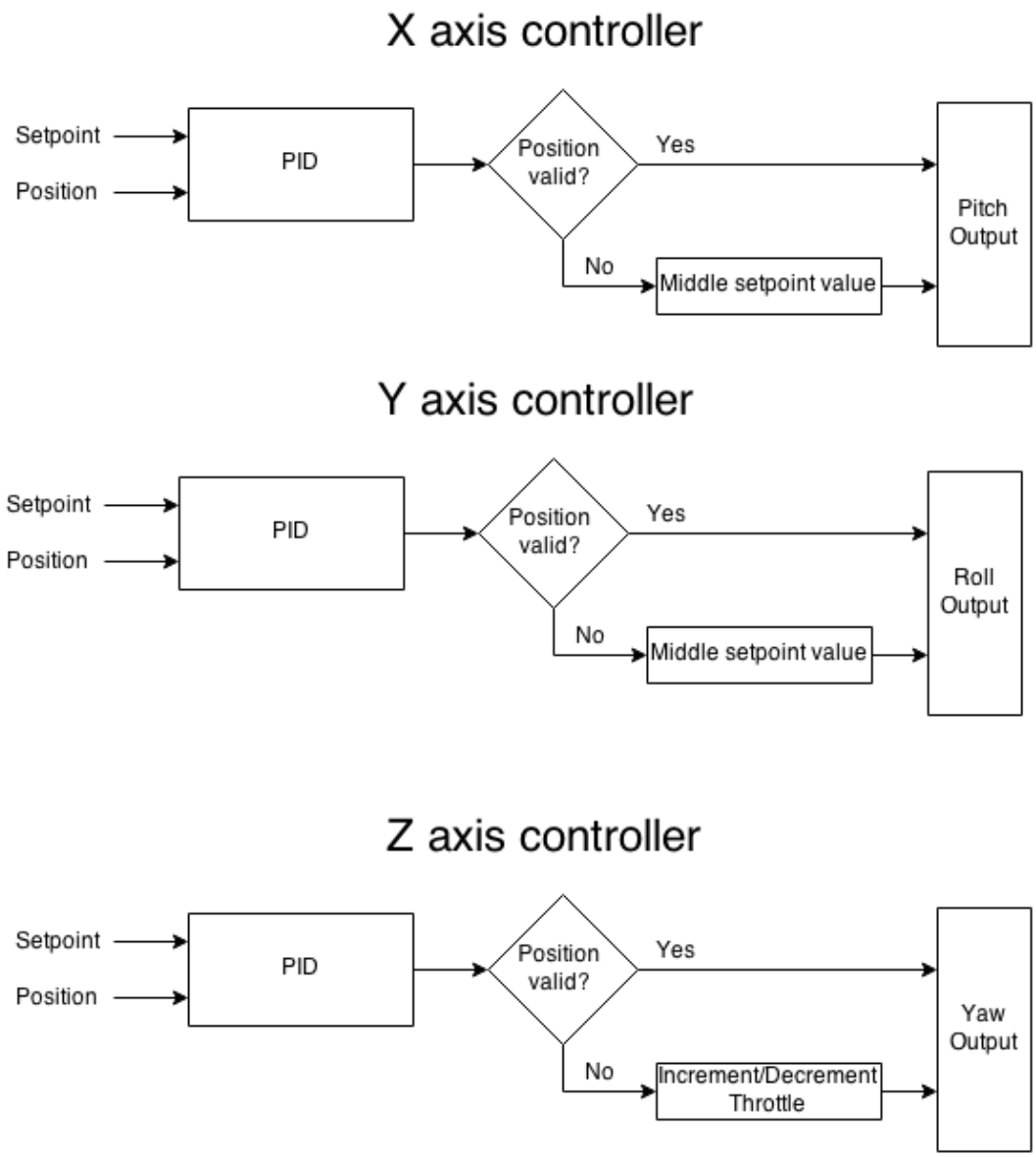


Figure 4.18: Flow of the position controller

4.5 Position estimation

The position estimation is based on six sonars, and a processing algorithm. It is built upon the work presented in Strand (2014), which is a previous work by the author. The system is developed further in this work, based on testing on a flying platform.

As described in the theory section, the sonars measure distances in a cone in front of it. The sonars used in this project has a cone of 20 degrees aperture. This means that a single sound burst from the sonar can hit and reflect off several objects. The sonar return the distance to the closest object. Based on this, they can be assumed to report the same distance continuously while pointing in the same direction. When the UAV moves, the sonar measurement will steadily increase or decrease if the sonar is measuring the distance to the same object. If new objects become the closest, the sonar measurement will see a sudden change in value. It is therefore important to detect these sudden changes. In the algorithm described below, these changes are detected by checking the difference between the newest value and the average value. Since an exponential continuous averaging method is used, the average value will transition smoothly between two values in a sudden change. If the measured value is sufficiently near the average value, is means that the measured value has been fairly stable for set amount of time. If it is not, the value is deemed invalid until it is stabilized again. The method is visualized in figure 4.19, where the green area represent the accepted interval about the average value. In the below algorithm, the averaging is tuned such that the average will use roughly 300 ms to stabilize on a new value, depending on the numeric difference to the previous value.

4.5.1 Physical setup

The six sonars are mounted so that two sonars measure distance in each axis. It is vital that all the sonars are mounted precisely according to the X,Y and Z axes. The up/down facing sonars in the Z axis, the front/back facing sonars in the X axis and the right/left facing sonars in the Y axis. Exceptions to this will not only make the calculations inaccurate, but will also introduce interference between the sonars. This interference appears when a pulse from one sonar is picked up by another. Since this requirement is fulfilled, all sonars can be triggered at the same

Figure 4.19: The green zone represent the accepted interval about the average value, the red line is the measurements.

time. In other applications, one can trigger one sonar after the other. This will yield a far inferior measurement rate, since each sonar will use about 100 ms for each measurement, which gives 10 measurements/second in total. By triggering all sonars simultaneously, one achieves 10 hz * 6 sonars = 60 measurements each second. One must also consider physical vibrations and electronic noise when mounting the sonars. Both these phenomena can interfere greatly with the distance readings. In this project, the ESC's and motors are the greatest source of electronic noise. The sonars and their wires have been placed as far away from these components as possible. All sonars are mounted on a balsa wood frame, which has good vibration dampening properties.

4.5.2 Distance measurement analysis

All the sonars are triggered to make a measurement every 100 ms, which makes the measurement frequency 10 Hz. The choice of update frequency is based on the maximum measuring frequency of the sonars. In the code, each sonar has an object assigned to it. The object contains the distance measurement, the average distance for that sonar, the calibrated sonar offset, validity of that sonar and the noise level. The sonar's offset is set to the difference between the current position and the sonar reading. This offset is reset each time the sonar transitions from being invalid to valid. The offset is later used to estimate the position from the position it was last reset. The validity value is vital to the total algorithm of the sonar system. It simply shows if the last measurement of the sonar is valid, and therefore should be used in further calculations. The noise level describes how stable the values of a sonar is. If the value has large variations, this value is large. The noise level is calculated by taking the difference between the previous and current value, and average this over time using an exponential average.

$$NoiseValue_k = NoiseValue_{k-1} + \alpha * ((value_k - value_{k-1}) - NoiseValue_{k-1})$$

When a reading cycle is complete, the following is done for all sonars:

1. Check if the reading is inside the physically possible max/min values.

2. Check if the reading is an outlier value.

3. Update the average distance for the sonar. An exponential average function is used.
4. Update the noise level of the sonar.
5. Check if the reading is within the accepted interval about the average.
6. Detect transitions from invalid to valid. Reset sonar offset to current position if a transition is detected.

The validity of the sonar value is set to true or false based on these tests. An outlier value is a value that is within the max/min limits, but makes a jump greater than 50% of the average compared to the previous value. Outliers are ignored for the rest of the cycle. The algorithm ignores maximum one outlier, before starting to use the following values. The accepted interval about the average is 30% of the average value. This number is based on testing of the system, which have shown that this interval must be larger when a sonar measures a larger distance. After this algorithm is run, we are left with six distance measurements. We have also evaluated which of these readings that are "trusted" for use in further calculations. In the original algorithm presented in Strand (2014), attitude data from the IMU was used to filter out sonar values. Thresholds were introduced in both attitude angles and angle velocities. This is why the implemented system has an IMU. During tuning of the positioning algorithm, it was deemed that the filtering based on attitude was not necessary. It is more likely that it would remove valuable readings more often than it would filter out bad ones. As the sonar readings would vary greatly during fast angle movement, the sonar values themselves provide enough basis for determining their validity. The attitude of the UAV will not impact the sonar readings greatly as long as the attitude is smaller than the aperture angle of the sonars, which is 20 degrees. This is because the sonars detect objects in a cone in front of them, and one path in the cone will be horizontal as long as the attitude is less than ± 20 degrees. Any larger attitude variations will impact the sonar reading so much that their invalidity will be detected that way.

4.5.3 Estimating position

We evaluate the three axes separately, by combining the pair of sonars along each axis. An algorithm is run for each sonar pair. It evaluates the sonars value, validity and noise level, and

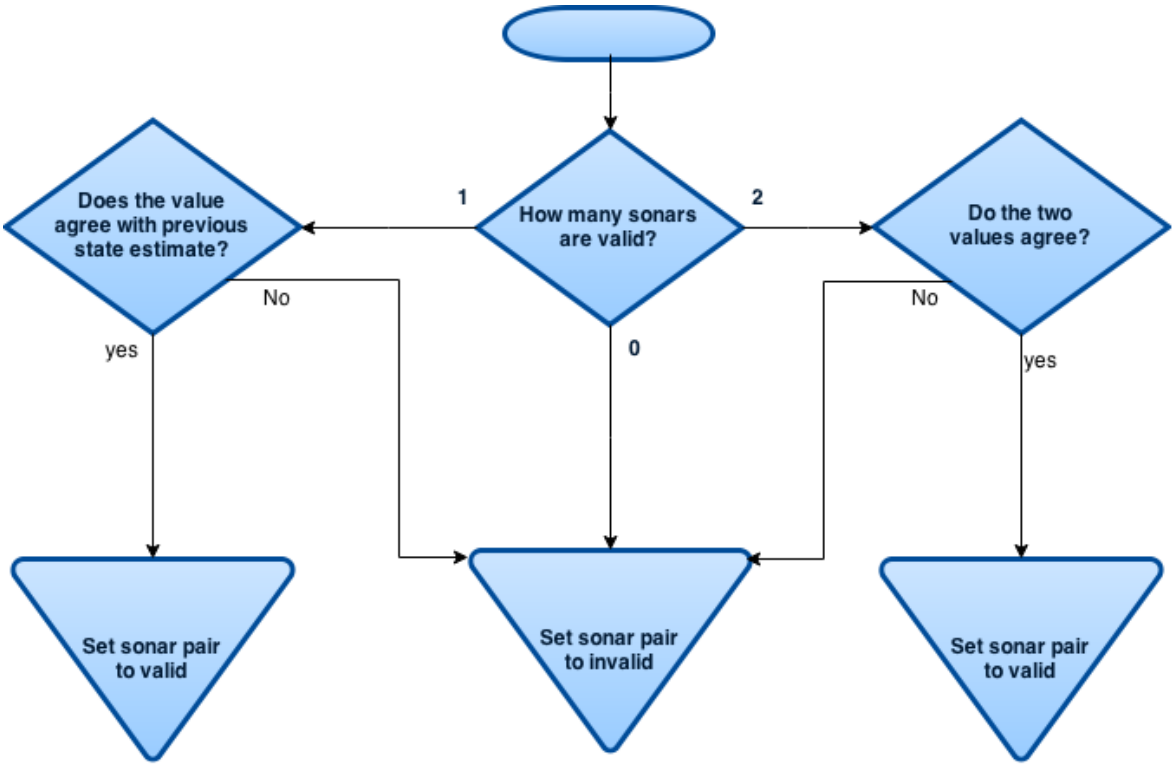


Figure 4.20: Algorithm for evaluation of sonar pair validity

combines the two sonars in the pair. The algorithm executes different actions, based on how many of the sonars in the pair is valid. It can be summarized to three cases, displayed in 4.20. If only one sonar is valid, that value is used for the position estimate. In the case where both sonars are valid, their values are merged based on their noise values. If none of the sonars are valid, the position estimate is not updated and also set to invalid. Let us denote the two sonars A and B. If sonar A has twice the noise value of sonar B, it will be given half the weight when merging the two values.

$$weight_A = \frac{noiseLevel_B}{noiseLevel_B + noiseLevel_A}$$
$$weight_B = \frac{noiseLevel_A}{noiseLevel_B + noiseLevel_A}$$
$$Position = weight_A * (value_A - offset_A) + weight_B * (value_B - offset_B)$$

The noise in the sonar reading has shown to increase with the distance measured. By using this merging method, the system will be able to use the sonar which is showing the most stable readings. For example, when landing, it will use the more accurate reading of the sonar facing

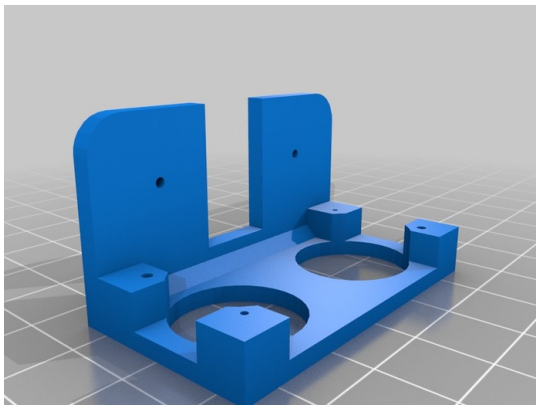


Figure 4.21: 3d printed mounting bracket for the hcsr04 sonar

downwards.

4.6 Mounting and Wiring

A rectangular frame made of balsa wood was fastened to the X frame. The six sonars were mounted to this frame using 3d printed brackets. These brackets were found on the open source 3d printing resource site www.thingverse.com, and can be seen in figure 4.21. A small electronic circuit was created for the Arduino and IMU. The ciruit was made from "proto board", which is a board with pre-drilled holes and some premade circuit paths. This circuit provided the physical connectors for everything connected to the Arduino, as well as a small power distribution circuit for the Raspberry PI and USB Hub. See figure 4.22. The USB Hub, Raspberry PI and proto board with the Arduino was put together in a "stack" using velcro tape. This "stack" was mounted on the underbelly of the quadcopter using velcro and a strap. See figure 4.23 for the complete build.

The wiring of the system is represented in figure 4.24. Note: This diagram does show which wire connect to which pin, but shows all wires and which devices they connect. The black wires are ground wires, and the red wires are 5 Volt supply voltage. The double black lines represent USB cables. Wires of other colour than black and red represent signal wires. The power supply chain starts at the battery, which supplies 12V to the ESC. The ESC has an internal 5V voltage regulator, which supplies the KK2 flight controller, the Raspberry PI and the USB hub. The USB hub was added because the USB ports of the Raspberry PI cannot provide enough supply cur-

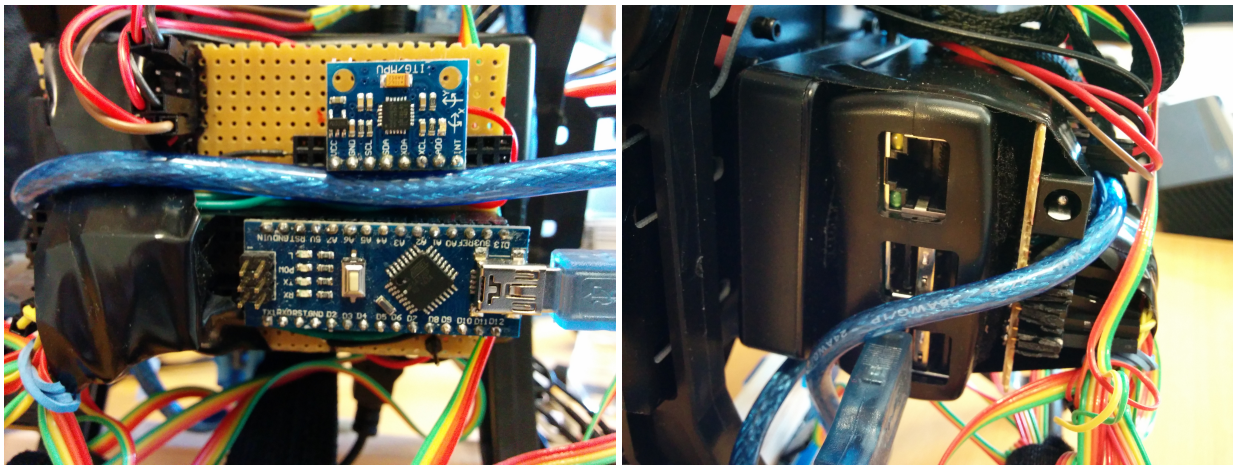


Figure 4.22: Breakout board for Arduino and IMU based on a proto board(left). Equipment "stack" (right)

rent for the Arduino, wifi dongle and camera.

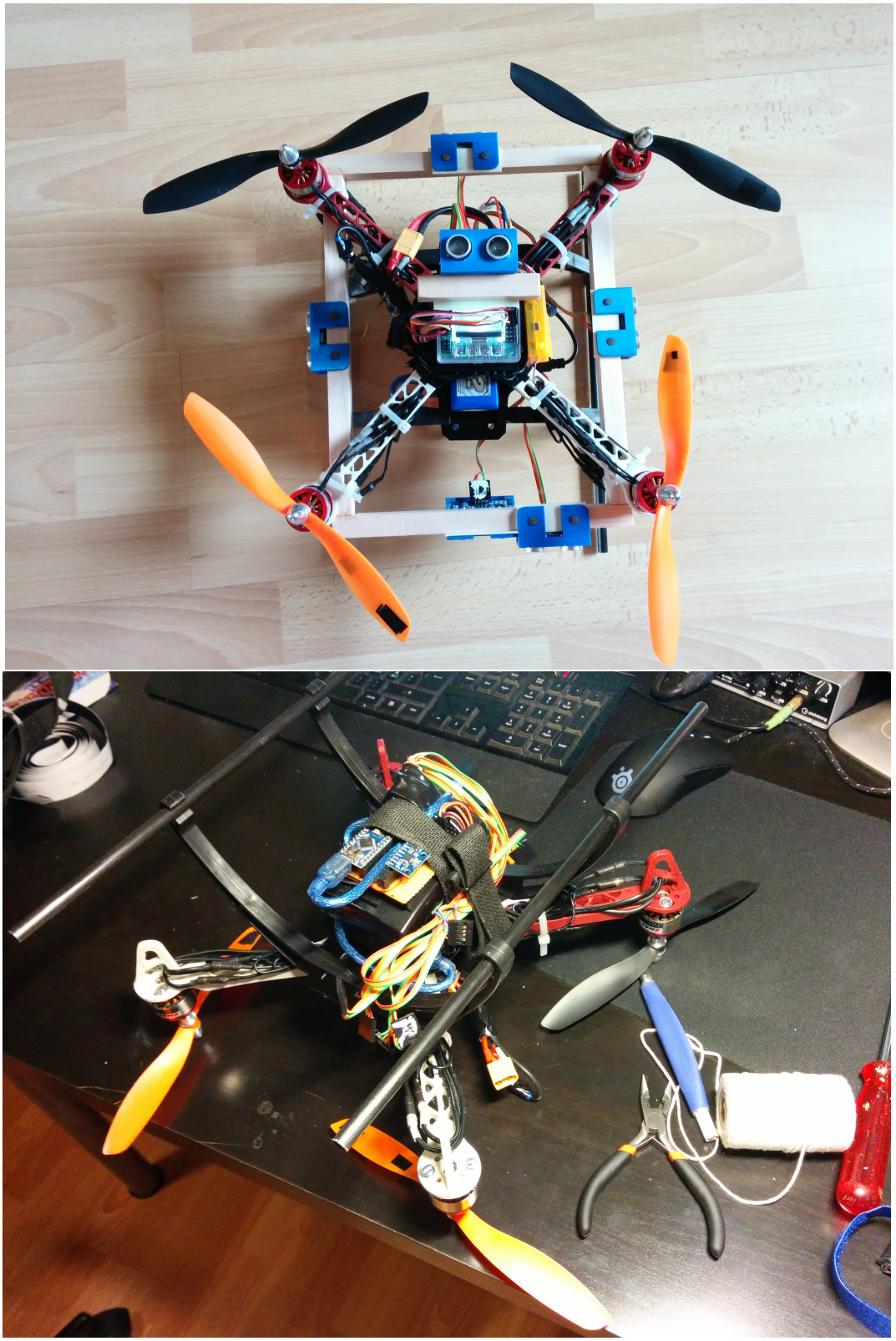


Figure 4.23: The UAV with the complete system mounted

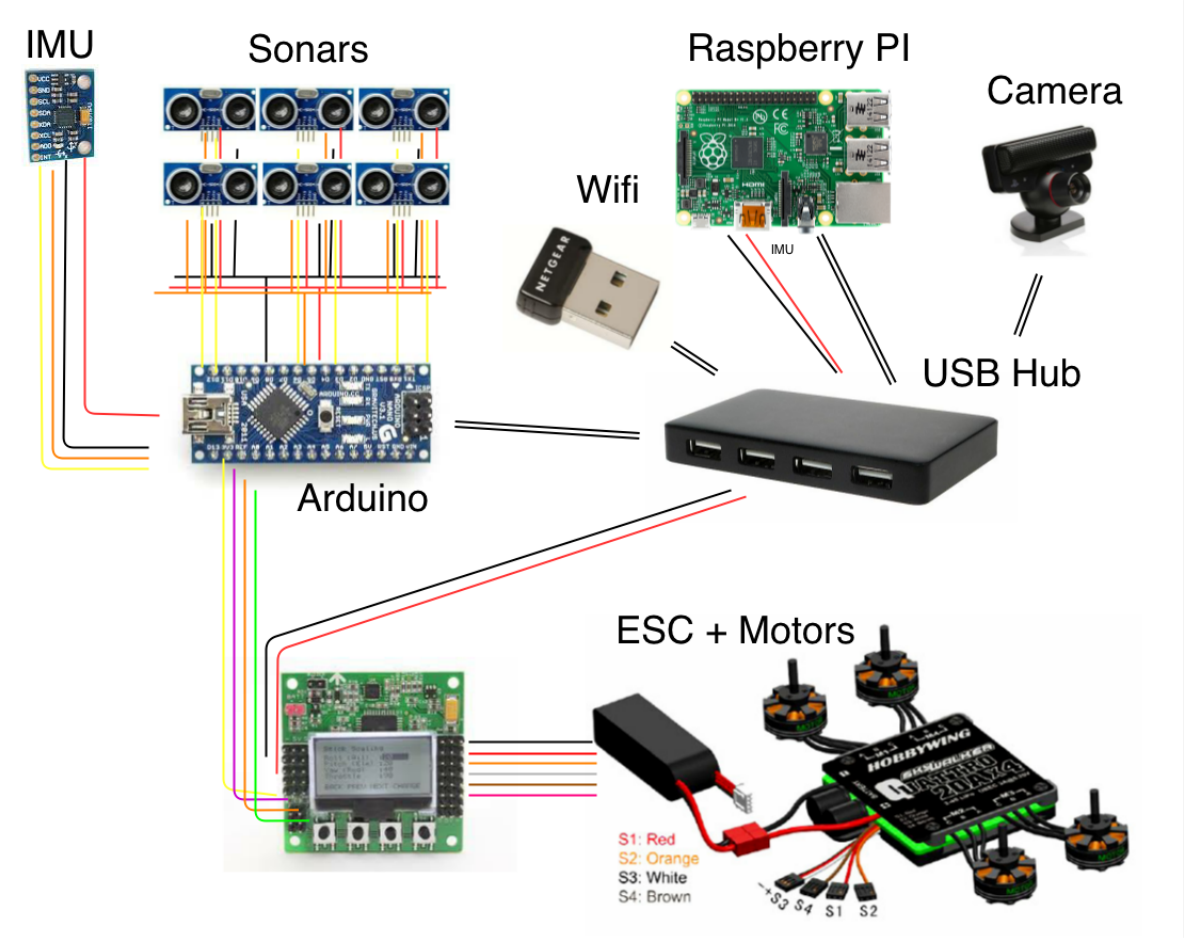


Figure 4.24: An overview of the wiring of the system

Chapter 5

System testing

The testing of the system started with evaluation of the sonars and the position estimation. Further testing would be a complete system including control via the frontend system and flight control algorithms. As seen in this chapter, the results from the position estimation system are not sufficient to use for testing of the position control algorithms. Therefore, the complete system was not tested as intended.

5.1 Test 1 - Multiple objects

In this test, multiple objects was placed in front of a sonar, at different distances. Figure 5.1 shows a diagram of the test setup. The sonar sits on a desk, 100 cm above the floor. It does not move during the test. Object 1 and 2 are chairs, placed with their backs flat towards the sonar. The chair backs are covered in cloth, with a soft inside. The test setup mimics a common indoor environments, where the rooms are populated by various objects.

In figure 5.2 we see the measured distances from the sonar. Although the sonar does not move, it's measured distance switched between the two objects in its measuring cone. In addition, there is a lot of noise in general. This might be due to the objects having a relatively small surface area compared to the size of the sonars measuring cone at that distance. A measuring series like this is not applicable in the position estimation algorithm suggested in chapter 4.

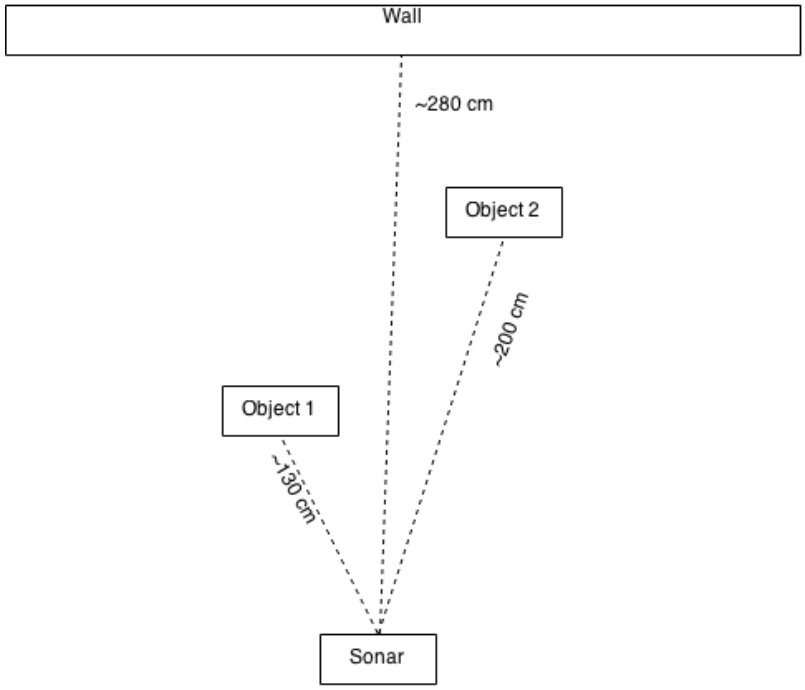


Figure 5.1: Test setup of test 1 - multiple objects

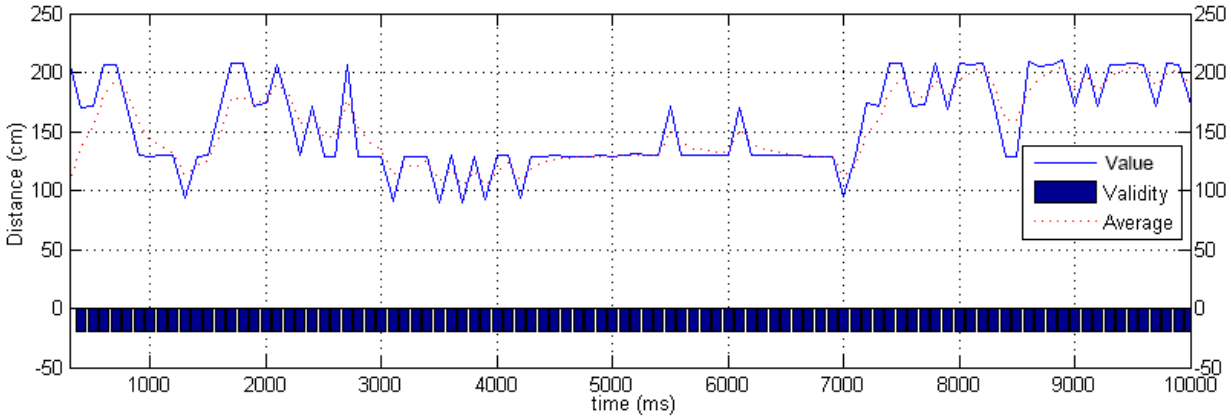


Figure 5.2: Test results with multiple objects

5.2 Test 2 - Comparison of two sonar models

This test evaluates the differences between the accuracy of the inexpensive HCSR04 sonar, and the more costly MaxBotix sonar. Two measuring series was done for each sensor, one where the distance is short (50cm) and one long range (255cm) measurement. The two sonars are mounted side by side, but does not measure at the same time as this would cause interference between the two. The sonars did not move during the measuring series.

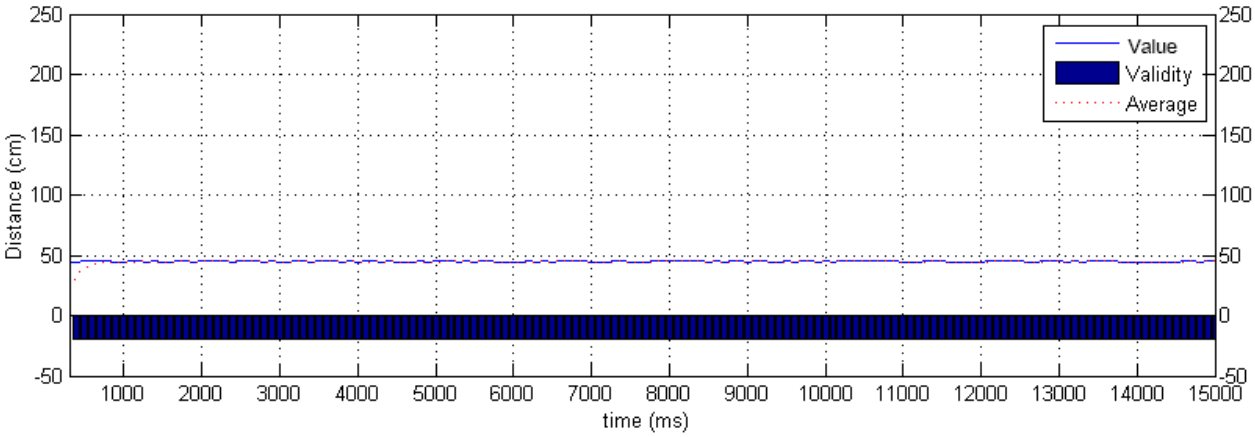


Figure 5.3: Short range test of the inexpensive hcsr04 sonar

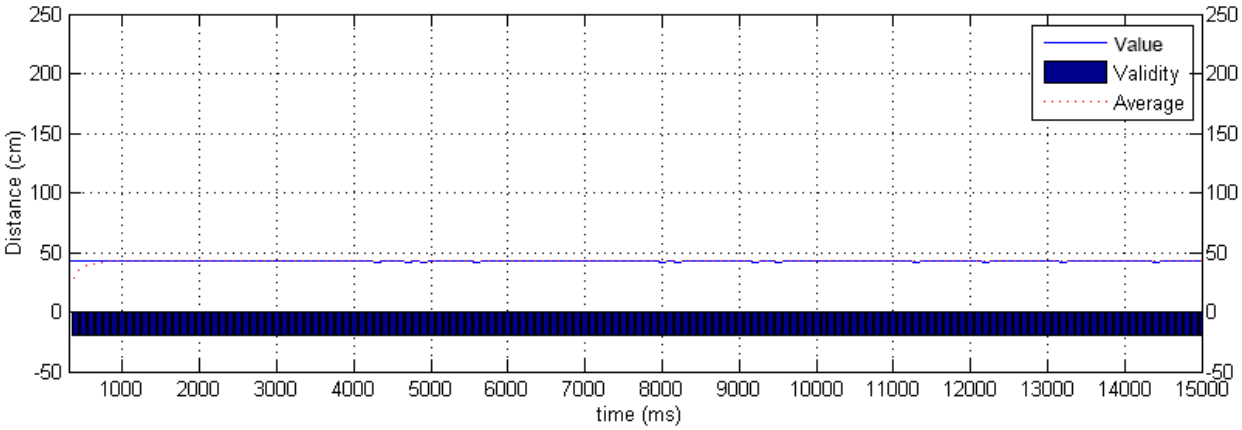


Figure 5.4: Short range test of the more expensive MaxBotix sonar

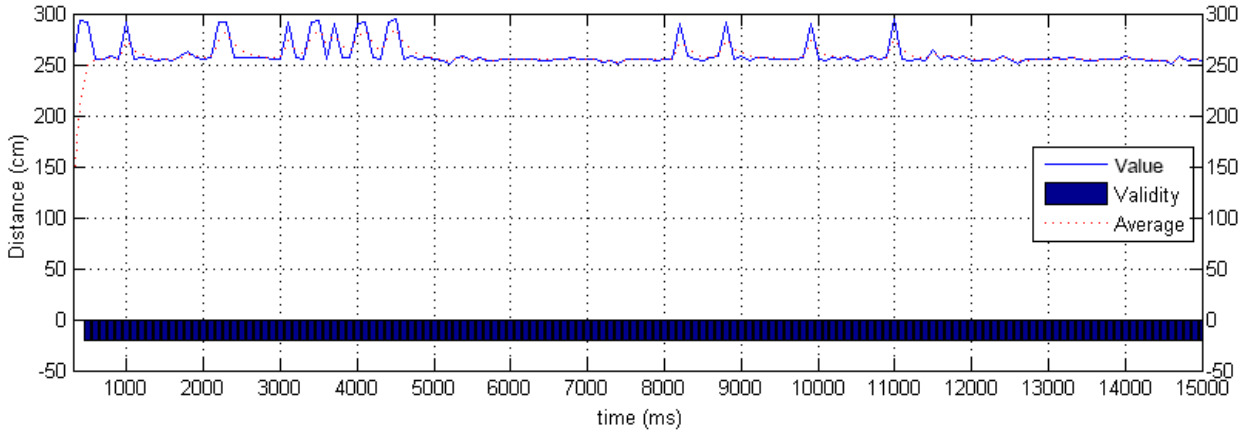


Figure 5.5: Long range test of the inexpensive hcsr04 sonar

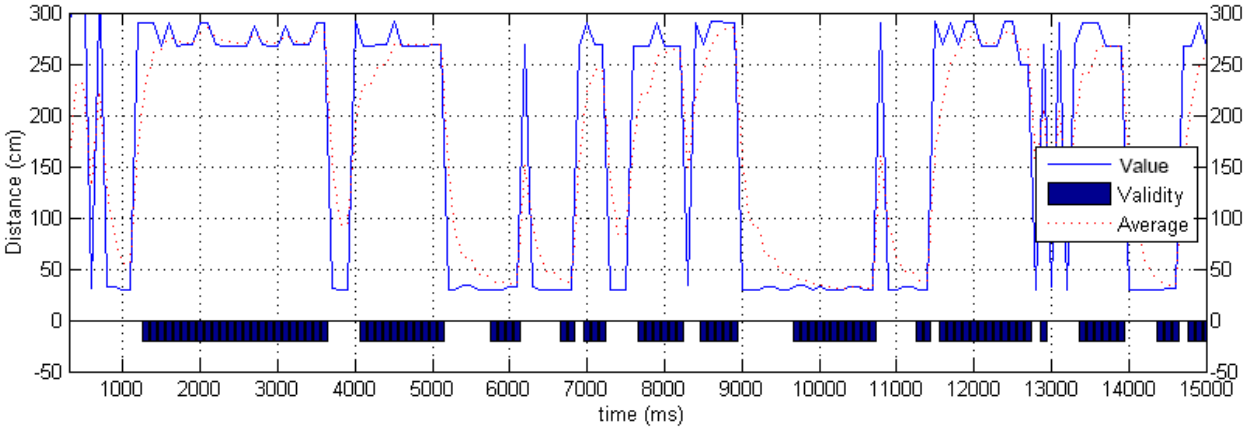


Figure 5.6: Long range test of the more expensive MaxBotix sonar

Figure 5.3 and 5.4 show that both sonars are able to provide a stable measurement with low noise at a distance of 50 cm.

In figure 5.5 we see the hcsr04 measurement on 255 cm. The series has a large noise component of about 45 cm. Figure 5.6 shows the corresponding 255 measurement from the MaxSonar. The values oscillate between 260 cm and 33 cm. There was no object present at 33 cm, so this measurement is an error. This error was not investigated further, as the intent behind this test was to determine if an investment in six more expensive sonars would have been beneficial to the system.

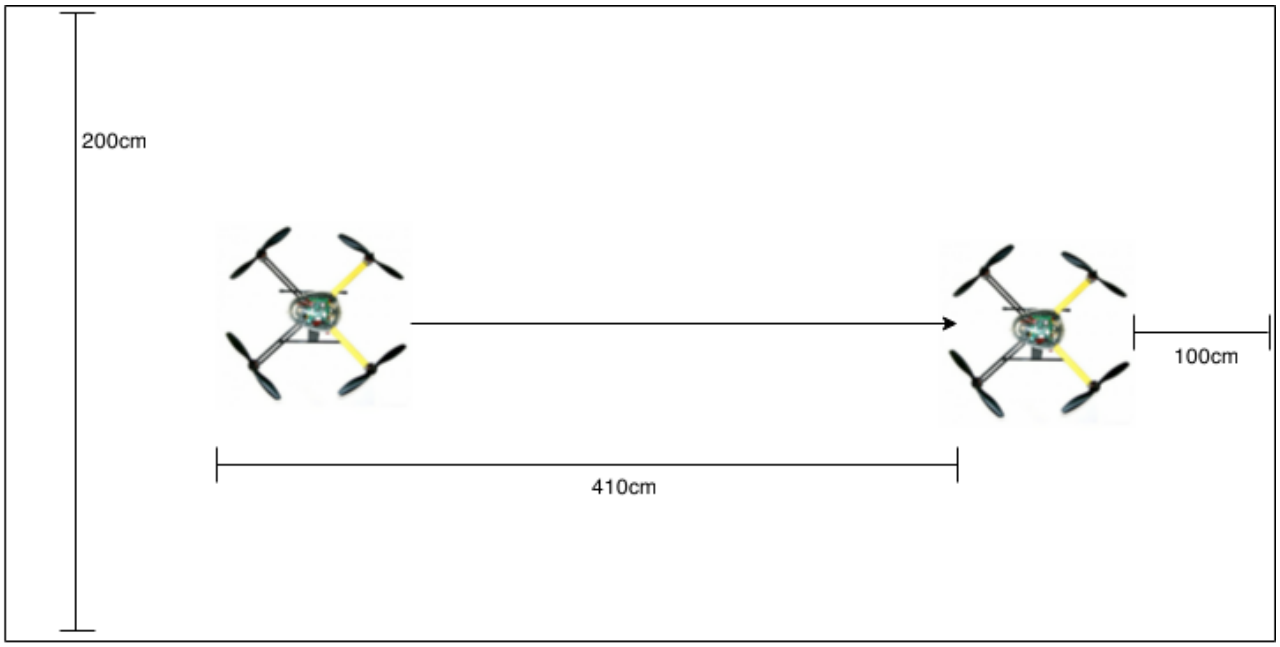


Figure 5.7: Diagram of the flight in Test 3

5.3 Test 3 - Flying in a hallway

During this test, the UAV was controlled with a remote control. The output from the position controller was not connected to the flight controller, but its outputs were logged. The purpose of the test was to evaluate the performance of the sonar measurements, the position estimates and the position controller. It was intended to establish if the system could estimate an accurate position along the Y and Z axis, as well as detect the wall in front when it came close to it. The UAV was flown in a hallway with a width of 200cm, and straight walls on all sides except behind the UAV. These conditions can be considered optimal for sonars, because of the large flat surface areas of the walls. It took off 510 cm from the wall, and flew 410 cm forward before landing. During the flight, there was movement in the Y axis (right/left) as well as some yaw. The landing occurred roughly 50cm to the left of the starting point along the Y axis.

Figure 5.8 shows the test results for the front and back sonars, as well as the position estimation in the X axis. As expected, the measurement of the front sonars is very noise initially, as it will measure the distance to the side walls as often as the wall in front. It continue to vary greatly throughout the flight, but stabilizes at 100cm when the front wall is reached. The sonar measurements in the backwards direction is also very unstable. It measures distances first to the

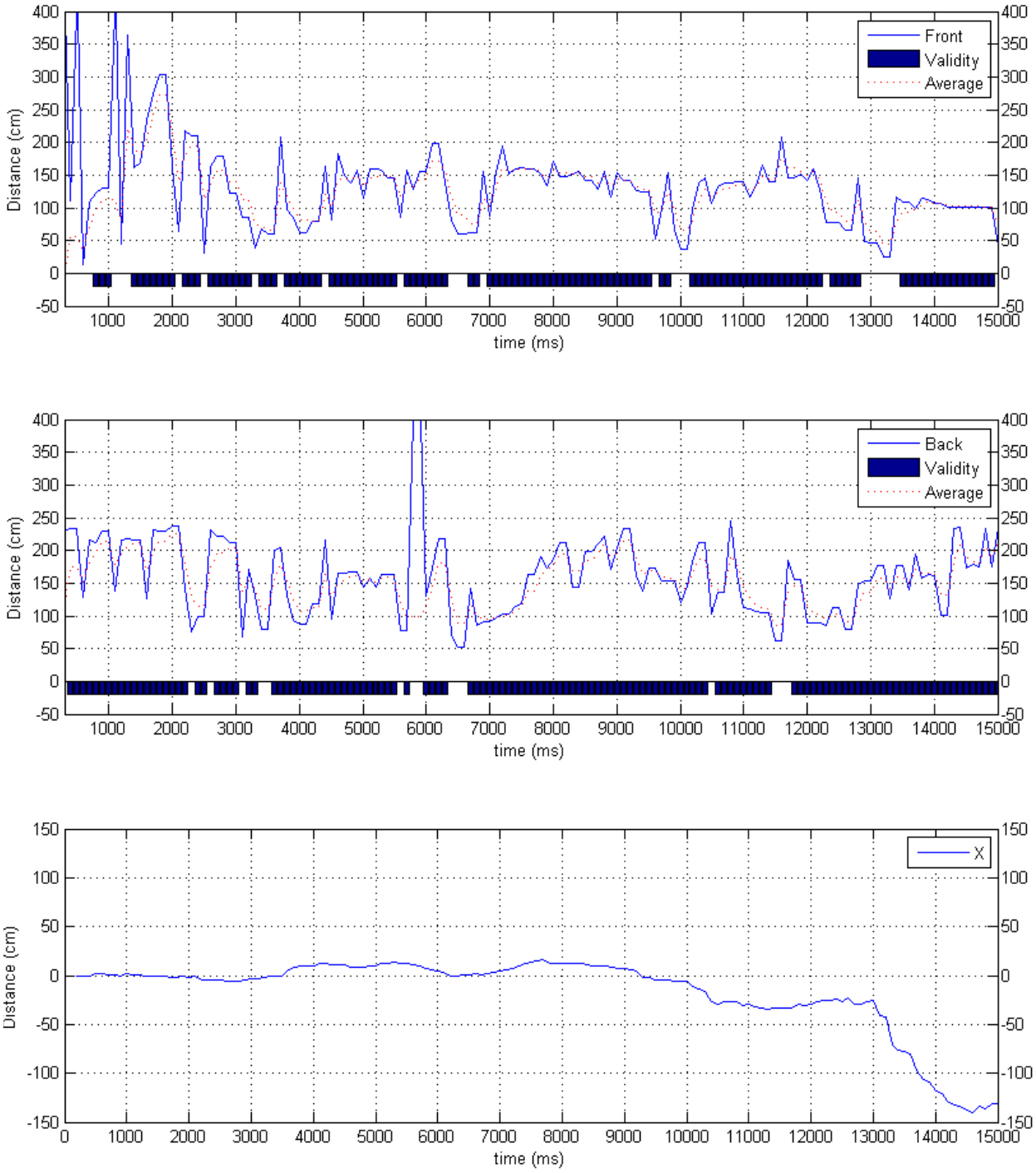


Figure 5.8: X axis result of test 3

pilot standing behind it, and then also the the side walls as it flies forward. The X axis position estimate is based on bad measurements, and is very inaccurate.

Figure 5.9 shows the test results for the left and right sonars, as well as the position estimation in the Y axis. The right facing sonar provides a stable measurement series, with a rise in value towards the end of the flight. This rise corresponds to the actual movement, as the UAV was moving to the left during landing. The sudden fall at 14500 ms however, does not represent the distance to the wall. It is most likely caused by the sonar beam hitting the floor, and reporting back that distance instead. The Y position estimate is noisy, but overall describes well the actual flight of the UAV. The resulting position is however not correct due to the sudden fall in the "right" value.

Figure 5.10 shows the test results for the up and down sonars, as well as the position estimation in the Z axis. The downward facing sonar provides a stable measuring series, which corresponds to the the UAV's flight. The sonar facing up provides a very noisy measuring series, with noise amplitudes up to 50cm. This data is almost ignored by the position estimation algorithm as long as the downward facing sonar is deemed valid. If the downward measurement was at any point deemed invalid, the estimation would have to be based on the "up" value and become very inaccurate. The Z estimation is almost purely based on the downward sonar due to the noise value weighting. It is therefore fairly accurate, and corresponds well with the actual flight pattern.

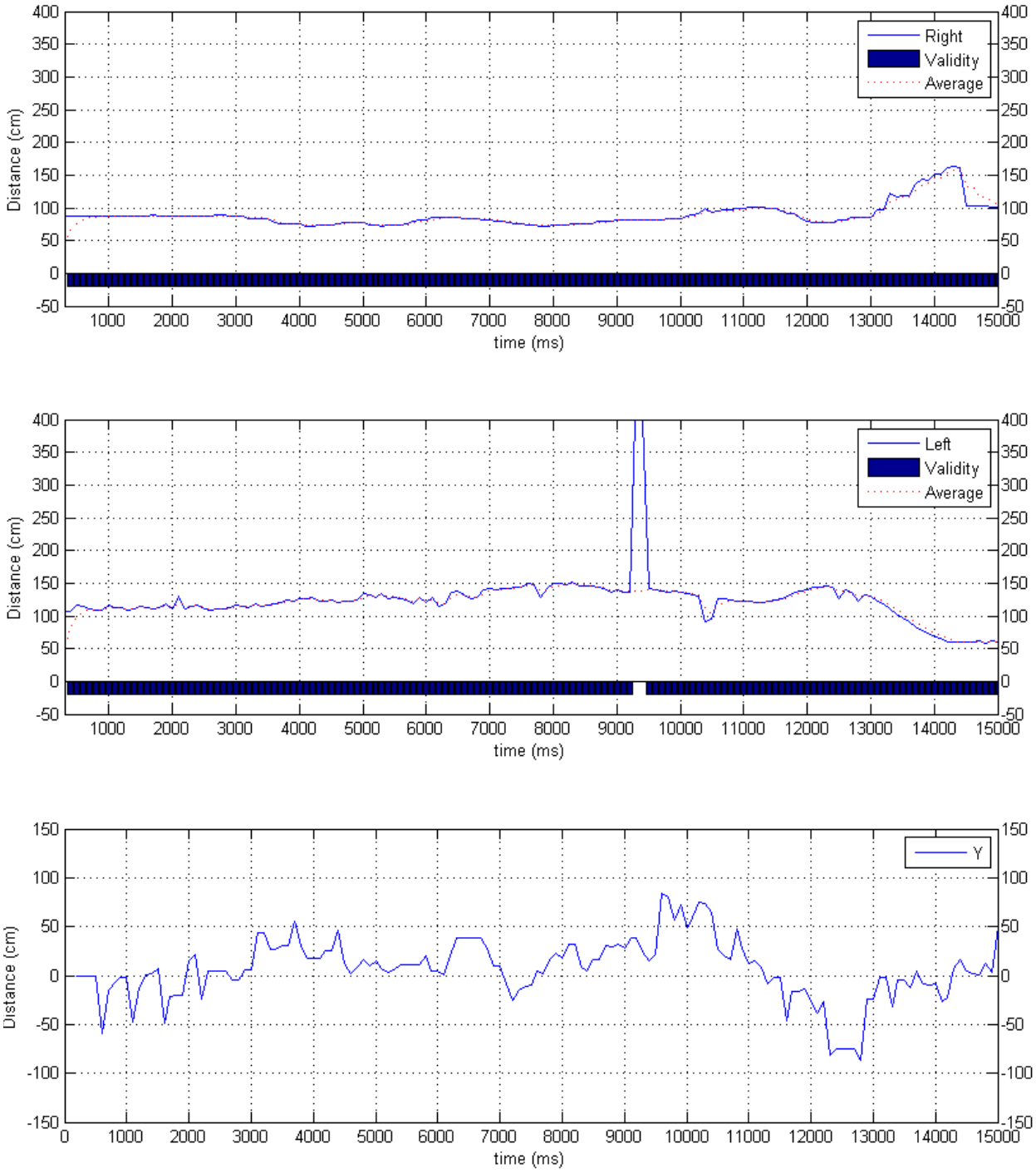


Figure 5.9: Y axis result of test 3

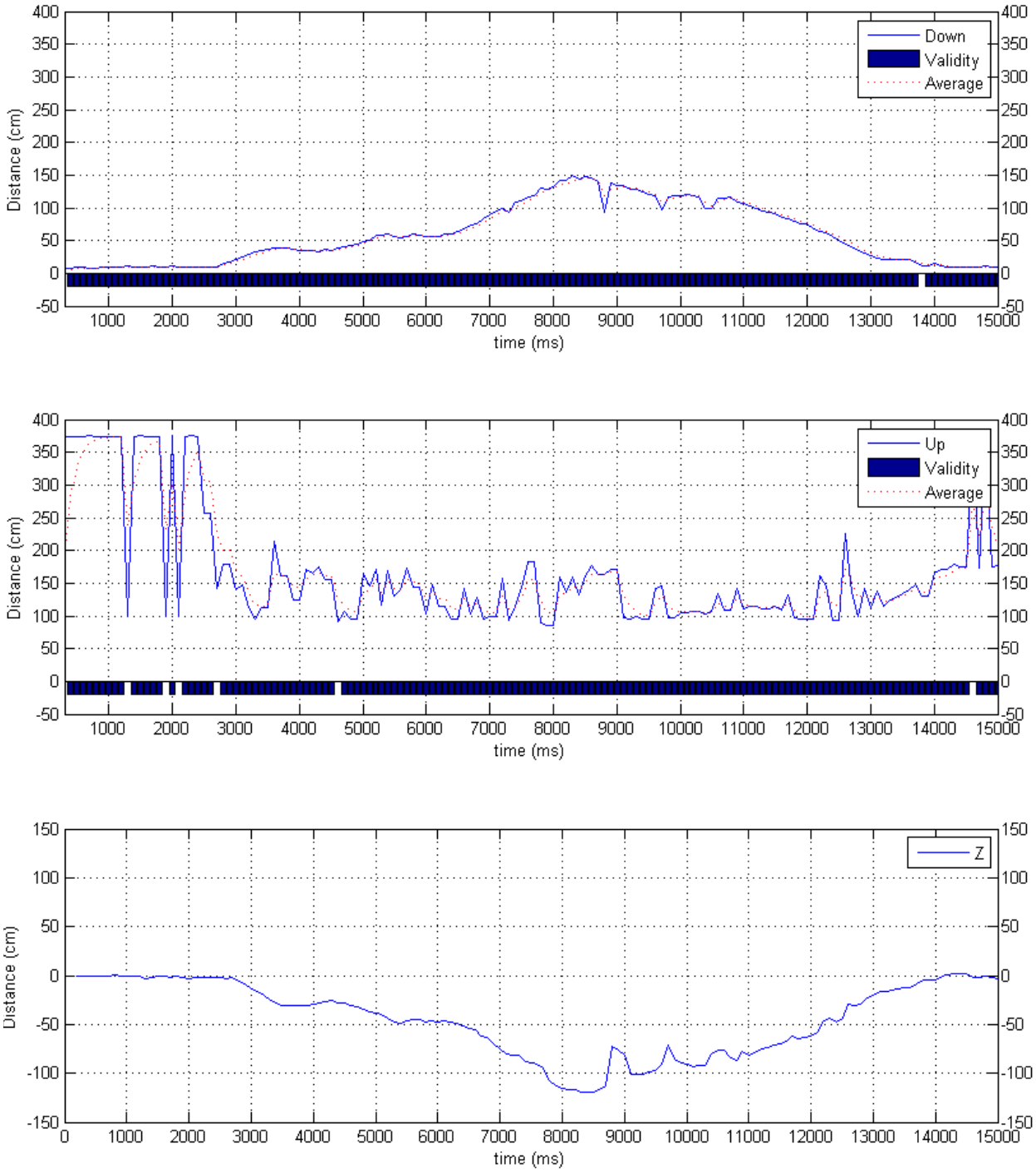


Figure 5.10: Z axis result of test 3

Chapter 6

Discussion, summary and conclusions

This chapter will start with a discussion of the equipment and solutions chosen in this paper. It will then present recommendations for further work followed up by a summary of the work presented, and conclusions.

6.1 Discussion

6.1.1 Sensors

The Playstation 3 Eye camera delivers video with low latency, but width a limited resolution. These features fits the application well, and it has proven to be bo reliable and efficient. It is also inexpensive, with a price of 120 NOK. Interfacing the camera to the Raspberry PI trough USB was done in a plug and play fashion. The camera is supported by the V4L (Video for Linux) package, which makes extraction of video easy.

The MPU6050 IMU ended up not being used in the finished solution. It is, however, a good low cost IMU for most applications where some noise can be tolerated. The i2c interface is well documented, and there exist drivers for many platforms online. With the Arduino platform, all one have to do is download a library and use its function calls to extract IMU data.

The HCSR04 sonars Are very inexpensive, and can be accurate under the right conditions. On flat surfaces, and especially with distances lower than 100 cm, their performance is very good,

with noise levels under a few centimeters. For this project, their noise levels at distances over 100cm are too high. The measured distances when multiple objects are present in the measuring cone is also problematic. Many things suggest that this is a trait of sonars in general, and not the hcsr04 model. Section 5.2 shows a comparison of the inexpensive hcsr04 and the more expensive Maxbotix Maxsonar 1040. The results give no indication that the more expensive sonar have better performance. It would be beneficial to this system to have sonars with a more narrow measuring cone, and a higher measurement rate. If on had more measurements per second, better filters could be implemented on the measurements. Sonars with 100 measurements per second exist, but at a considerable higher price range. This will be discussed further during the discussion of the position controller.

6.1.2 Microcontroller Unit subsystem

The Arduino Nano board combines easy development with solid performance. Having a USB connection makes it a perfect match for the Raspberry PI, as it can take care of small real time tasks while the PI can run a more complex program. The Arduino hardware abstraction layer speeds up development, but hides many important aspects of microcontroller programming. The Atmega 328 features three timer units, which are needed for many tasks. In this project, they are used for general stopwatch usage as well as both PWM generation and pulse width measurements. The setup of these timers are hidden under the Arduino libraries. In for example the Servo.h library, which is used for PWM generation, one simply creates a servo object and sets the pwm duty cycle. It is therefore important to keep tabs of which timers the different libraries use. When using such a library, one can investigate this by looking at the code of the library. in addition, most libraries have this information available in their header files. Other resources libraries might use is the i2c bus, serial port, SPI bus in addition to program memory. The Atmega 328 itself provided enough processing power and connectivity for this project.

6.1.3 Embedded Linux system

Hardware

The Raspberry PI platform provides a very compact linux system, supporting USB and networking trough ethernet or wifi. In addition, it has ample interfacing to more low level systems trough GPIO, Serial (UART), SPI and i2c. Its CPU is not as powerful as a typical desktop computer, but its 700 MHz processor is capable of executing most tasks without problems. The Raspberry PI can be powered trough a 5V micro USB, or trough the GPIO pins. The latter was used in this project, which was convenient as the power source was a battery.

Software

The Debian linux operating system running on the Raspberry PI is highly configurable, and has a large community around it. Implementing things like a camera or wifi usb dongle is done easily, because many has done it before and posted guides about it online. Both the backend and frontend code was written in python and run trough the python interpreter. Python is a straight forward development environment, with several "plug-and-play" libraries available for all kinds of purposes. This leads to a rapid development cycle, because one can easily utilize the work of others and not reinvent the wheel. It is important to note that neither Debian Linux nor Python has any real-time properties. One can not assume that timed tasks execute when expected. The OS runs its own scheduler, and will pause your program when it best see fit. Similarly, the Python runtime can at any time decide to do garbage collection (clear redundant variables/data from memory). In this project, the programs running on the rPI did not have hard real-time requirements, but some tasks can considered firm real-time. This means that the task can miss its deadline, but its result still has value to the overall system. One such task is the position controller, which should execute every 100 ms. Since the UAV will float around its position somewhat anyway, it is not critical that the controller algorithm is run precisely at this interval. Some jitter in the timing is tolerable. This is especially true if the PID-regulator does not use the I and D components. These components use the time since last cycle in calculating their output, and jitter in timing will then cause inaccuracies.

6.1.4 Frontend System

Controlling a quad rotor UAV is normally done with a standard rc remote controller. With these remote controllers one use two joysticks to control throttle, yaw, pitch, roll. Both hands are required to fly the UAV with this control method. This projects goal was to make a UAV with simplified controls, where most active and demanding tasks were handled by the UAV itself. The control interface was to be latency tolerant, and the remote operator should be able to attend other tasks at the same time as controller the UAV's movements. A web interface is a good model for such a control interface. Latency can be simulated, and the actions performed to control the UAV is button presses, rather than a continuous movement of a stick or lever. Many types of radio connections can be used, but video streaming limits the options. As this project was to show a proof of concept, wifi was chosen. The UAV hosts its own wifi network and web server, which can be connected to from many devices. Wifi was also used for easy control of the internal programs of the eLinux system trough a SSH connection.

6.1.5 Position Control

All position controllers depends on accurate position estimation or measurement as feedback in order to work. The position estimation algorithm proposed in this work was based on certain assumptions about the sonar measurements, which turned out to be false. The most important assumption was that the measured distance from a sonar would stay steady as long as the sonar was in the same position. We have seen in section 5.1 that this is not the case. The distance measured by the sonars used can vary from one measurement to the other. In addition, the testing has shown that the measurements often have a noise component of 50cm or more when measuring a distance over 150cm. For example when measuring along the Z axis, this is typically the upward facing sonar. While the downward facing sonar often has stable and accurate measurements because of the short distance and large flat surface (the floor), the estimate loses accuracy the moment the downward facing sonar is deemed invalid. In that iteration of the position estimator, only the upward sonar is used in the estimation, which leads to a 50cm noise component in the position estimate. When the downward sonar stabilizes and becomes valid again, it is reset to the an arbitrary position within the oscillating position estimate. The

algorithm was designed to have a "one-out-of-two" (1oo2) redundancy scheme, where the two sonars along one axis would support and compliment each other. In practice, only one of the sonars provides valuable measurements. When this sonar becomes invalid, the position estimate is in practice unusable.

The accepted interval about the mean (described in figure 4.19 was initially very small, filtering out all large deviations. Testing revealed that this also filtered out actual movement, as this also can cause large deviations in the measurements. It also nullified most measurements at longer distances, since these measurements typically were noisy. Since the position estimate is just based on distance measurements, it was clear that a noisy measurement is better than nothing. The noisy measurement would at least give rough information about the position, although oscillation about the true value. Based on these two factors, the accepted interval was widened and its size was based on the measured distance. This made the system able to detect movement better, and also make use of the the noisy measurements. On the other hand, a larger deviation was required before the algorithm assumed that the sonar was hitting a new object. This deviation must be bigger than what a movement would create, as well as bigger than the noise component. As no a priory knowledge about the surrounding of the UAV is available, it is very hard or even impossible to determine whether a deviation within the accepted interval was caused by movement, noise or a new object in the measurement cone. The noisy values is the main problem. If the measurements were stable, one could have a narrow accepted interval, and easily reset the sonar offset when stabilizing on a new object. This is true even if the measured distance switches between two objects in its cone, as long as the frequency is not so high that the measured value never stabilizes. If the measured values on both sonars are stable, the algorithm can utilize the opposite facing sonar while stabilizing on the new object on the other.

6.2 Summary and conclusions

This paper presents a "proof of concept" system for assisted control of indoor UAV's. The requirements of such a system is evaluated, and a proposed solution is presented. The proposed solution is based on distance measurements in six directions using ultrasonic range finders, a camera stream, an embedded Linux computer and a microcontroller unit, all mounted on a

quad rotor multicopter platform. The system were to handle the most active and demanding tasks required to fly a UAV indoors, which is attitude and position control. The attitude control was handled by a off the shelf flight controller unit. The position control was based on a position estimate generated from the distance measurements from the six ultrasonic range finders (sonars). The complete system was built and tested, from the low level sensors and microcontroller, trough the embedded Linux computer, to the web interface. Test flights were done by flying manually with a radio controller. These tests showed that the position estimates were not sufficiently accurate to be used in a position controller. The position controller's outputs were therefore never connected to the flight controller in order to control the UAV. For the same reason, the web user interface was not developed further than a basic functional solution. As the position estimation were a key part in the proposed solution, its failure caused the total system to not work as intended. However, all other individual modules of the system did. A key press in the web browser (on a smartphone or laptop) changed the setpoints of the position controller trough the web server and backend code running on the eLinux computer. The position controller generated the correct output based on its state and position estimate. This output was transmitted trough the serial port to the microcontroller unit where it was translated to a PWM pulse connected to the flight controller. Similarly, the sonar distance measurements were gathered by the microcontroller and traveled the same way back up to the position estimation algorithms and the web interface to be displayed to the remote operator.

The position estimation algorithm has been shown to work well when provided with accurate and stable distance measurements. If such measurements could be provided at distances over 100cm, the approach could be worth investigating further. Most sonars have similar characteristics to the ones used in this project. Such sonars are not usable in accurate position estimation because of noise at long range, and issues occurring when several objects are present in the measuring cone. They can, however, with proper filtering be used to provide absolute distances between the UAV and surrounding objects. All indoor UAV systems which are not flown by line of the sight should use this information in their control interfaces to give the remote operator a sense of location in the room. This concept is shown in figure 4.15. The distance measurements are also valuable for implementing anti-collision features. As seen in figure 5.8, the wall is detected by the forward facing sonar when the UAV is about 100cm away. If the re-

mote operator is flying using a video stream, like proposed in this paper, collision avoidance could be implemented using these measurements. This applies even though a position control is not implemented on the UAV, but handled by the remote operator instead. A simple algorithm could apply a pitch, roll or throttle output when the UAV got too close to either a side wall, a front/back wall or the ground/roof respectively. Such an algorithm was not implemented in this project, but would be a natural next step after completing a working position controller. if used purely for environmental information and collision avoidance, the sonar readings could be filtered more aggressively in order to find the true values. Aggressive low pass filtering and outlier removal creates a delay between measurement and processed value. The position controller in this paper requires real time position data, which does not allow such filtering. In a collision avoidance algorithm however, accurate values would be more valuable, even if they had a small time delay. When large variations is present in the sonar measurements, the erroneous measurements tend to show a shorter range than the true range. In a collision avoidance scheme, this would cause unwanted stops. In an indoor UAV system, these stops would only cost time. In a anti-collision system mounted in a car, such stops could cause a traffic accident. In the UAV system, an erroneous measurement which was longer than the true value would be far worse, as this could lead to collisions.

The system proposed and implemented in this paper did not provide a sufficiently accurate position controller in order to offload the remote operator during flight. The position control must either be done by the remote operator, or a more advanced and accurate positioning system. The position estimation scheme can be utilized, either with more accurate distance measurements or more aggressive filtering of values. It would in either case only serve as a helper system to the main positioning system, and provide a relative position estimate with a time delay due to filtering. The implemented system's best application is as a collision avoidance system for indoor UAV systems.

Appendix A

Acronyms

UAV Unmanned Aerial Vehicle

rPI Raspberry PI

SPI Serial Peripheral Interface

I2C Inter-Integrated Circuit

UART Universal asynchronous receiver/transmitter

NED North East Down

SONAR Sound Navigation and Ranging

IMU Inertial Measurement Unit

USD United States Dollars

DOF Degree of Freedom

MEMS Micro Electro-Mechanical System

SoC System on Chip

MCU MicroController Unit

FC Flight Controller

ESC Electronic Speed Controller

RPM Rounds Per Minute

FPS Frames Per Second

TCP Transport Control Protocol

UDP User Datagram Protocol

RTSP Real Time Streaming Protocol

GPIO General Purpose Input/Output

Appendix B

Equipment list

B.1 Main system

- HCSR04 Sonar
- Maxbotix Maxsonar 1040
- MPU6050 IMU
- Ps3 eye Camera
- Arduino Nano
- Raspberry PI model B+
- Netgear N150 USB wifi adapter

B.2 Quadrotor platform

- KK2 flight controller board
- Turnigy 2200 mAh 20C battery
- Hobbywing SkyWalker-Quattro-25A*4 ESC
- Emax 2812 1540KV motor

- F330 quadrotor frame
- 8x45 propellers

Appendix C

Guides and Tutorials

- Wifi Access Point setup for Raspberry PI
- Web Cam streaming from Raspberry Pi to Android using Gstreamer
- Flask webserver quickstart

Bibliography

Begen, A. C., Akgul, T., and Baugher, M.

Carillo, L. R. G., López, A. E. D., Lozano, R., and Pégard, C. (2011). Combining stereo vision and inertial navigation systems for a quad-rotor uav. *Robotics Journal of Intelligent and Robotic Systems*.

Kim, J., Pearce, R. A., and Amato, N. M. (2002). Robust geometric-based localization in indoor environments using sonar range sensors.

Mui, M. and Chintalapally, A. (2014). The rising demand for indoor localization of uavs.

Ozer, J. (2014). Mastering the adobe media encoder.

Strand, A. (2014). Indoor positioning or uav using inertial navigaton and ultrasonic ranging.

Tardós, J. D., Neira, J., Newman, P. M., and Leonard, J. J. (2002). Robust mapping and localization in indoor environments using sonar data. *The International Journal of Robotics Research*.