Thomas Sundvoll

A Camera-based Perception System for Autonomous Quadcopter Landing on a Marine Vessel

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Thomas Sundvoll

A Camera-based Perception System for Autonomous Quadcopter Landing on a Marine Vessel

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of Science and Technology

Abstract

Small Unmanned Aerial Vehicles (UAVs) have attracted a lot of attention in recent years, and one of the more studied UAVs is the quadcopter. A quadcopter is also a Vertical Take-Off and Landing (VTOL) vehicle, meaning that it has an advantage when operating in tight spaces. Combined with their high maneuverability, they are a great tool for many tasks, such as inspection, transportation of small packages and surveillance for search and rescue operations.

To increase the flight time and reduce the cost of manual piloting the vehicles, a lot of research is done recently on autonomous quadcopters. Some parts of the autonomous flight, especially the landing, require a precise position estimate. This thesis investigates an area of application where the landing spot is severely restricted in terms of size, namely landing on a small marine vessel. In this case, the landing spot might be approximately of the same size as the quadcopter itself, requiring an even more precise position estimate. In this case, regular GPS measurements are not precise enough to perform autonomous landing. Therefore, this thesis investigates the use of a camera as the main sensor for estimating the position of a quadcopter, anticipating that this will give a better estimate.

A landing platform is designed and created to serve as the landing spot in the experiments. It is designed to resemble a standard landing platform typically found on marine vessels. The marine vessel that eventually will host the landing platform is the ReVolt model ship created by DNV GL, so design measures are taken to fit the landing platform to that specific ship.

A computer vision system is developed with the main purpose of estimating the quadcopter's position relative to the landing platform. The main challenge with a computer vision system at sea is the lack of fixed points to navigate by, since the sea is constantly moving. To solve this issue, traditional computer vision methods are applied, such as color segmentation, edge detection and corner detection, to extract the known features of the landing platform. From this, the position is estimated using the pinhole camera model and known measurements on the landing platform. The methods and algorithms for the position estimate are developed using the OpenCV library for Python and the computer vision system is integrated into the Robot Operating System (ROS) framework. In addition, a dead reckoning module is developed to give an estimate based on the inertial measurements of the quadcopter when no computer vision estimate is available.

The system is tested both in a simulator and with a physical quadcopter and landing platform, with accurate results in the simulator and promising, but a bit more noisy results with the physical quadcopter. Finally, some suggestions for improvements of the methods and future work on the topic are given.

Sammendrag

Små, ubemannede luftfartøyer (UAVer) har tiltrukket seg mye oppmerksomhet de siste årene, og et av de mest studerte UAVene er quadcopteret. Et quadcopter går også under kategorien VTOL-fartøy som er et begrep som brukes om fartøy som kan ta av og lande vertikalt. Dette gjør at quadcoptere har en fordel når de opererer i områder med lite plass. Sammen med deres høye manøvrerbarhet er de et godt vertøy til å utføre mange oppgaver, slik som inspeksjon, transport av små pakker og overvåkning for søk- og redningsopperasjoner.

For å øke flygetiden og redusere kostnadene ved å manuelt styre slike fartøy, er det i det siste forsket mye på autonome quadcoptre. Deler av en autonom flytur, og særlig landingen, krever et presist posisjonsestimat. Denne oppgaven undersøker et bruksområde hvor landingsplassen er betydelig begrenset når det kommer til størrelse, nemlig å lande på et lite, sjøgående fartøy. I dette tilfellet kan landingsplassen være omtrent på samme størrelse som quadcopteret selv, noe som krever et posisjonsestimat med enda høyere presisjon. I dette tilfellet vil ikke vanlige GPS-målinger være presist nok til å utføre autonom landing. Derfor undersøker denne oppgaven bruken av kamera som hovedsensor å estimere posisjonen til et quadcopter, med forventning om at dette vil gi et bedre estimat.

En landingsplattform er designet og bygget for å fungere som landingsplass i eksperimentene. Den er designet for å etterligne en standard landingsplattform som vanligvis er å finne på sjøgående fartøy og marine installasjoner. Fartøyet som til slutt vil bruke landingsplattformen er modellskipet ReVolt som er laget av DNV GL, så designet er tilpasset for at landingsplattformen skal passe til dette spesifikke skipet.

Et datasyn-system er utviklet med hovedhensikt å estimere quadcopterets posisjon relativt til landingsplattformen. Hovedutfordringen med et datasyn-system på sjøen er mangelen på faste punkter å navigere etter, siden sjøen er i konstant bevegelse. For å løse dette problemet er tradisjonelle datasyn-metoder brukt, blant annet fargesegmentering, deteksjon av kanter og deteksjon av hjørner, for å hente ut allerede kjente kjennetegn på landingsplattformen. Ut fra dette er posisjonen estimert ved bruk av hullkamera-modellen og kjente mål på landingsplattformen. Metodene og algoritmene for posisjonsestimatet er utviklet ved bruk av OpenCV-biblioteket i Python, og datasyn-systemet er integrert inn i rammeverket Robot Operating System (ROS). I tillegg er en bestikkregning-modul utviklet for å gi et estimat basert på interne målinger hos quadcopteret, for bruk når ingen datasyn-estimat er tilgjengelig.

Systemet er testet både i en simulator og med et fysisk quadcopter og landingsplattform, med nøyaktige resultat i simulatoren og lovende, men støyfulle resultat med det fysiske quadcopteret. Til slutt er det gitt noen forslag til forbedringer av metodene og fremtidig arbeid på temaet.

Preface

This master's thesis is written during the spring semester of 2020 at the Norwegian University of Science and Technology (NTNU) and concludes my Master of Science in Engineering Cybernetics. My supervisor has been Anastasios Lekkas from Department of Engineering Cybernetics, NTNU and my co-supervisor has been Tom Arne Pedersen from DNV GL.

I have always been fascinated by small air vehicles. As a kid I had some radio controlled airplanes and helicopters to play with and it was great fun to fly them. In my experience, the manual control was hard to master and it took a lot of concentration not to crash the vehicles. Therefore, it was interesting to take on this task with autonomous unmanned air vehicles and to learn more about the challenges and possibilities with autonomy for such vehicles. Furthermore, I was interested in doing some practical testing and experiments.

The main goal of this thesis is to develop a computer vision software system that can provide a pose estimate for a quadcopter based on monocular camera images. The thesis is also a part of a larger project where the goal is to achieve autonomous landing on a scaled version of the ReVolt vessel created by DNV GL. Considerations about this broader area of application is taken into account when conducting this thesis, and a landing platform is designed and built to fit on the ReVolt.

In addition, this thesis includes practical testing of the computer vision system, both in a simulator and with a physical quadcopter. One of the original objectives was to test the developed system in the drone lab at NTNU, then with the ReVolt on land and finally with the ReVolt at sea. However, due to the Covid-19 virus outbreak, and the following lockdown in Norway from the middle of March, extensive testing with the physical quadcopter became difficult. Nevertheless, some small tests in the front yard at home were possible along with increased testing in the simulator.

This thesis is a continuation of my project report on the same topic, conducted during the fall semester of 2019 [1]. The work with the project report gave me time to become familiar with the topic, learn how to code with ROS, how to set up the simulator and how to control the quadcopter, all of which was useful experiences when conducting this master's thesis. The general idea for the layout of the landing platform also came to be during the project, although refinements are done in this thesis.

The computer vision system developed in this thesis is also inspired by some of the promising findings in the project report. The good results using traditional computer vision methods for detecting the circular shapes of the landing platform led to the idea of extending the detection to ellipses. Furthermore, the project report uses a neural network for classification of the 3D position. At the same time it suggests in the future work section to investigate the possibility of doing this classification using the camera parameters and known geometry. This idea is followed up here and the neural network is set aside in favour of more focus on traditional computer vision methods.

This thesis benefits from some open-source software:

- ROS and Gazebo developed by Open Robotics.
- ROS packages, including *ardrone_autonomy*, *tum_simulator*, *uuv_simulator* [2].
- Python packages, including Numpy, Scipy, Matplotlib and OpenCV.
- The free 3D computer graphics software Blender for designing the landing platform.
- The free video editing software Kdenlive for editing the attatched videos.

The Department of Engineering Cybernetics at NTNU has provided:

- Work station computer: Dell OptiPlex 7040, with Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and 32 GB RAM.
- A Parrot AR.Drone 2.0 quadcopter with spare batteries.
- Funding to create the physical landing platform.
- The handheld controller, a PlayStation 4 controller, for controlling the quadcopter.

DNV GL has provided:

- 3D model of the ReVolt vessel.
- Access to the physical radar for test-fitting the landing platform.

All images, figures and plots are generated by the author unless otherwise stated.

This thesis would not have been possible without the help I have received from a number of wonderful people. First of all, I would like to thank my supervisor Anastasios Lekkas and co-supervisor Tom Arne Pedersen for their guidance and support during both the project report and this master's thesis. Furthermore, I would like to thank my fellow student Daniel Tavakoli, who has been working on the planning and control part of the larger project, for good cooperation and valuable exchange of views. Next, I want to thank Glenn Angell at the workshop at ITK for producing the parts I needed to build the landing platform. I also want to thank Tania Bonilla for her input on how to make the 3D model of the stands and for suggesting the use of hook-and-loop to mount the landing platform onto the radar. Also, I am grateful to my brother Elias for borrowing me his laptop and an extra computer screen to make my work more efficient and comfortable during the period with a home office. Finally, I want to thank my loving wife Siri and all my friends in Trondheim for making the past five years an unforgettable period of my life.

Thomas Sundvoll
Trondheim, 19. June, 2020

Table of Contents

Abstract i

Sammendrag ii

Preface iii

List of Tables vii

List of Figures x

Abbreviations xi

1 Introduction **1**
1.1 Background and motivation 1
1.2 Objectives 3
1.3 Contributions 3
1.4 Outline 4

2 Theory **5**
2.1 Computer vision 5
2.1.1 Traditional computer vision 5
2.1.2 Edge detection 7
2.1.3 Corner detection 8
2.1.4 Fitting an ellipse to a set of image points 9
2.1.5 Finding the center and axes from ellipse parameter 12
2.1.6 HSV color space 14
2.2 Camera pose estimation 14
2.3 Quadcopter dynamics 15

3 Design of landing platform and experimental setup **17**
3.1 The landing platform 17
3.1.1 Plan for attachment to the Revolt 18

3.1.2 Assembling the landing platform . . . 20
3.2 The quadcopter . . . 22
3.3 Software . . . 24
3.3.1 ROS . . . 24
3.3.2 The Gazebo simulator . . . 25
3.4 Handheld controller . . . 26

4 System design **27**
4.1 Pose estimation . . . 27
4.1.1 Color segmentation . . . 29
4.1.2 Edge detection . . . 31
4.1.3 Corner detection . . . 32
4.1.4 Finding higher level features . . . 34
4.1.5 Choosing which method to use . . . 38
4.1.6 Calculating position from high level features . . . 38
4.2 Filter . . . 42
4.3 Dead reckoning . . . 43
4.4 User interface . . . 44
4.5 PID controller . . . 46
4.6 Automated landing . . . 47
4.7 DDPG controller . . . 47
4.8 Connection to the quadcopter . . . 47
4.9 Running the system . . . 48

5 Results **49**
5.1 Experiments in the simulator . . . 49
5.1.1 Assessment of the computer vision system . . . 49
5.1.2 Test of all methods when flying up and down . . . 50
5.1.3 Test of all methods when hovering . . . 53
5.1.4 Test of filter . . . 60
5.1.5 Test of dead reckoning . . . 60
5.1.6 Test of yaw estimate while rotating . . . 62
5.1.7 Landing using the PID controller and the automated landing planner 63
5.1.8 Landing using external DDPG controller . . . 63
5.2 Experiment with the physical quadcopter . . . 65

6 Conclusion **69**
6.1 Future work . . . 69

Bibliography **71**

A Technical specifications of the Parrot AR.Drone 2.0 **79**

B ROS message definitions **81**

List of Tables

4.1 HSV threshold values for color segmentation 31
4.2 PID parameters 47

5.1 Accuracy and precision of the different methods for x-position estimate. . 58
5.2 Accuracy and precision of the different methods for y-position estimate. . 58
5.3 Accuracy and precision of the different methods for z-position estimate. . 59
5.4 Accuracy and precision of the different methods for yaw-rotation estimate. 59

A.1 Technical specifications of the Parrot AR.Drone 2.0 79

List of Figures

2.1 An ellipse tilted at an angle θ. 13
2.2 The hue range 14
2.3 Comparison of the color spaces RGB and HSV 14

3.1 The motive on the landing platform. 17
3.2 Dimensions of the radar 18
3.3 The 3D printed support stand resting on the radar. 19
3.4 An early sketch of how to mount the landing platform to the radar. 19
3.5 Mounting the stands to the radar. 20
3.6 Gluing the stands to the plexiglass. 21
3.7 Attaching the sticker to the plexiglass. 22
3.8 The finished landing platform attached to the radar. 23
3.9 The Parrot AR.Drone 2.0 23
3.10 The simulated environment 25
3.11 The Sony PlayStation 4 handheld controller used in this project. 26

4.1 Architecture of the ROS application 28
4.2 The different coordinate systems used with the quadcopter 28
4.3 Image segmentation in the simulator. 29
4.4 Image segmentation with images from the physical quadcopter indoors. . 30
4.5 Image segmentation with images from the physical quadcopter outdoors. . 30
4.6 Binary mask of the landing platform with edge in red. 32
4.7 Output from the Harris corner function 33
4.8 Corner selection for white segmentation 34
4.9 Geometric notation for the *corners* method 37
4.10 The pinhole camera model 39
4.11 The geometry of the pinhole camera model seen directly onto the s vector, which goes through both the center of the image, (u_0, v_0), and the center of the landing platform in the image, (u_p, v_p). 40

4.12 The geometry of the pinhole camera model seen from the side, parallel to the x-axis of the landing platform. 40
4.13 The geometry of the pinhole camera model seen from the side, parallel to the y-axis of the landing platform. 41
4.14 Block diagram of a typical PID controller 46
4.15 Control loop of the system. 46

5.1 Examples of feature detection on the landing platform 51
5.2 All variable's values and errors when flying from a hovering position at 0.2 meter above the landing platform, up to 5 meters above the landing platform and down again. 52
5.3 Examples where the arrowhead is wrongly detected. 53
5.4 Estimate error for x-position from all methods at different altitudes. . . . 54
5.5 Estimate error for y-position from all methods at different altitudes. . . . 55
5.6 Estimate error for z-position from all methods at different altitudes. . . . 56
5.7 Estimate error for yaw-position from all methods at different altitudes. . . 57
5.8 The estimates, filtered estimate and dead reckoning during a change in the z-position. 61
5.9 Test of the dead reckoning system when flying around the ship. 62
5.10 The yaw estimate when rotating 360°counterclockwise. 63
5.11 The trajectory, seen from all three sides, when landing using the automated landing planner. 64
5.12 The trajectory when landing, using the external DDPG method seen from all three sides. 66
5.13 Image from the outdoor testing. 67
5.14 Position estimates from outdoor test. 68

Abbreviations

UAV	=	Unmanned Aerial Vehicle
VTOL	=	Vertical Take-Off and Landing
ROS	=	Robot Operating System
DOF	=	Degrees Of Freedom
GPS	=	Global Positioning System
LIDAR	=	Light Detection And Ranging
DL	=	Deep Learning
DRL	=	Deep Reinforcement Learning
DDPG	=	Deep Deterministic Policy Gradient
ML	=	Machine Learning
SIFT	=	Scale Invariant Feature Transform
SURF	=	Speeded-Up Robust Features
ORB	=	Oriented Fast and Rotated Brief
NN	=	Neural Network
YOLO	=	You Only Look Once
RGB	=	Red, Green, Blue
HSV	=	Hue, Saturation, Value
PnP	=	Perspective-n-Problem
SDK	=	Software Development Kit
UUV	=	Unmanned Underwater Vehicle
IMU	=	Inertial Measurement Unit
RANSAC	=	RANdom SAmple Consensus

Chapter 1

Introduction

1.1 Background and motivation

Unmanned Aerial Vehicles (UAVs) and especially small quadcopters have become increasingly popular in recent years. One of the reasons for the popularity of quadcopters is their high maneuverability and their capability for Vertical Take-off and Landing (VTOL). This enables them to operate in areas with only a limited amount of space. Another reason is the simple mechanical structure of quadcopters, which makes them robust and requiring less maintenance [3]. Quadcopters have been studied for a long time and in recent years there has been more and more interest in autonomous quadcopters.

This master's thesis is written with focus on using an autonomous quadcopter in cooperation with the unmanned model ship ReVolt, created and operated by DNV GL. This is a 3 meter long scaled model of a 60 meter long unmanned, zero-emission concept ship bearing the same name [4]. The concept ship will be built in the future for autonomous shortsea cargo shipping. An autonomous quadcopter can supplement such a ship with many tasks, for example inspection of the ship or the cargo, smaller parcel deliveries to the shoreline or an external viewpoint for autonomous docking.

An autonomous quadcopter also has many use cases together with other manned marine vessels as well. One example is as a tool for search and rescue missions. In the case of a search and rescue mission at sea, efficiency is of extra importance. If for instance a passenger falls over board, it is critical to locate the victim as soon as possible. As time passes, the risk of hypothermia increases. Furthermore, the longer the victim has been in the water, the harder the search will be, due to sea currents and the fact that the ship may move away from the victim. It can take some time to launch a life raft, and by the time the life raft is in the water, the victim may already have drifted several hundred meters away. If an autonomous quadcopter could take off and start searching for the victim as soon as the alarm goes off, invaluable time could be saved in the rescuing process. The quadcopter could be used to locate the victim and report its position back to the rescuing team. It is also possible that the quadcopter could bring a small floating device for the victim to hold on to until the rescuing personnel arrives. The alternative to having an autonomous

quadcopter for this task might be to hire a human quadcopter pilot to control it. Hiring one extra crew member is very costly, so by making the quadcopter autonomous, such a solution might be viable for more ships.

One of the main challenges with autonomous quadcopters is perception. This concerns how to use sensory information available on the quadcopter to be aware of and interpret the environment around the quadcopter, and one important part of perception is self-localization in space. A quadcopter has six degrees of freedom (DOF), consisting of its position (x,y,z) and orientation (roll, pitch, yaw). An autonomous quadcopter must know this pose (position and orientation), relative to some reference system, in order to make any sensible actions. Information about the pose can be estimated using a variety of sensors, including Global Positioning System (GPS), Light Detection And Ranging (LIDAR), ultrasound, accelerometer, gyroscope, magnetometer and camera.

When the pose of a quadcopter is known, it can accomplish incredible tasks. For instance, balancing an inverted pendulum [5], lifting building blocks onto constructions [6] and flying in dense formations [7]. Multiple quadcopters can be even be used together to perform cooperative tasks, such as gripping and transporting items [8]. Common for all these examples is that they were used indoors with an external motion capture system providing an accurate pose estimate for each quadcopter. The nature of the application in this thesis however, implies that the quadcopter cannot use any such systems and must rely on the onboard sensors. The onboard monocular camera and the inertial sensors will be used to perform the pose estimate.

During an autonomous operation, one of the most critical tasks is the landing. In this case, the landing platform will be of approximately the same size as the quadcopter and mounted on a ship with water surrounding it. In addition, the ship may be subject to motion, either caused by itself or by the waves and current. Autonomous landing in such conditions requires a high degree of precision and reliability, both from the perception system and the controller.

The pose estimate provided by this thesis will be the input for a planning and control system developed by another thesis on the subject of autonomous quadcopter landing using reinforcement learning [9]. Reinforcement learning is a fairly simple concept that has been around for many years. It is based on the idea of letting an agent learn a desired behaviour through trial and error, by receiving rewards for actions that lead towards a certain goal and punishment for actions the lead away from that goal. The challenging part however, is to determine which actions should be rewarded and which should be punished [10]. The recent advances in deep learning (DL) have lead to deep reinforcement learning (DRL), which tries to solve this challenge by estimating the action-reward function using a deep neural network. One such DRL method is the deep deterministic policy gradient (DDPG) method [11], which is used in the other thesis.

Previously, a variety of different landing platforms has been utilized in projects on autonomous landing. Some have used LED lights on the platform to make it easier to detect [12], others have used distinctive colors and shapes, such as an 'X' shape [13], colored squares [14], an university logo [15] or a square fiducial marker [16]. However, this thesis seek to use an as naturalistic landing platform as possible, which to the best of the author's knowledge has not been done before in the literature.

Many camera-based self-localization methods exists already, such as Visual Odometry

[17] and Visual Simultaneously Localization And Mapping [18]. However, both of these methods work best when the environment is fixed. In the case of this project, most of the image will contain sea with an ever changing texture. Furthermore, the recent progress in the field of machine learning (ML) and deep learning (DL) has resulted in some astonishing results when it comes to extracting information from images, such as detecting, locating and classifying objects in images [19] [20] [21]. Nevertheless, when it comes to robotic perception there is still a need for expert knowledge to tailor a perception system to each individual application [22]. Therefore, in this thesis, traditional computer vision methods are chosen to custom make a perception system to suit the specific application in this project.

1.2 Objectives

The main goal of this thesis is to provide sufficient perception for a quadcopter to be able to land autonomously, with focus on pose estimation. The thesis asks the question: *Is it possible to use traditional computer vision methods to give a robust pose estimate for a quadcopter in a marine environment, using an onboard monocular camera?* This is to be achieved with a standard landing platform that is designed and built as part of this project. The aim is that this estimate will be precise enough to be used by an external controller for landing the quadcopter on the landing platform as well as perform a stable take-off and hover.

1.3 Contributions

The main contribution of this thesis is a computer vision system that estimates the simplified pose (position and orientation) of the quadcopter relative to a specific landing platform. The pose estimate is simplified by not estimating the roll and pitch-rotations about the x- and y- axis, only the yaw-rotation around the z-axis. This is done under the assumption that the quadcopter at all times is oriented horizontally. The reason why this is a legitimate assumption is that the quadcopter has an onboard autopilot that keeps the quadcopter hovering approximately on the same spot until any other control command is issued. However, this assumption is violated in windy conditions, as the quadcopter then has to tilt in order to hover at the same spot. Windy conditions are outside the scope of this thesis, so the assumption should hold. The computer vision system first finds the three distinct colors of the landing platform in the image; green, orange and white. Then, the characteristic shape of each colored area is used to find the center of the landing platform and the radius of the landing platform, both in pixels. Finally, this information is used together with the real radius in meters to calculate the quadcopter's position. Whenever it is sufficient information in the image to decide the rotation of the landing platform, also the yaw-rotation about the z-axis is calculated. The code for the computer vision system and for the rest of the developed system is available on GitHub [1].

The second contribution of this thesis is a full-size landing platform for a quadcopter. The landing platform is made of a plexiglass plate with a vinyl sticker on top of it. Un-

[1] `https://github.com/mrSundvoll/master_project`

derneath the landing platform there are three 3D printed stands that fit on top of a radar on the ReVolt vessel. They are attached to the radar using hook-and-loop for easy attachment and detachment. The design of the structure and the layout on top, as well as the ordering of all the necessary parts and the final assembly is done by the author.

The third contribution of this thesis is a dead reckoning module. This gives redundancy to the computer vision estimate and solves the problem of loosing the estimate for some period of time. Internal sensors on the quadcopter, such as the accelerometer, gyroscope and compass, give measurements of the velocities and accelerations along the three axes and the global orientation of the quadcopter. These measurements are used to iteratively calculate the quadcopter's position from the last available computer vision estimate.

The final contribution of this thesis is a holistic architecture for a perception, planning and control system, implemented in the Robot Operating System (ROS). In addition to the aforementioned computer vision system constituting the perception part of the architecture, a simple sequential planning module and a simple PID controller is implemented as individual nodes. There is put little emphasize on the creation of these modules as they are mostly there to help demonstrate the computer vision system in action. However, since they are created as individual nodes, they can easily be substituted by more sophisticated modules in future work.

1.4 Outline

Chapter 2 presents some of the computer vision methods used in this thesis along with the most important quadcopter dynamics. **Chapter 3** explains the design and creation process of the landing platform, in addition to experimental setup with the quadcopter and the simulator. **Chapter 4** presents the developed ROS modules, starting with the pose estimation module and following up with the filter and the dead reckoning module. Furthermore, the smaller auxiliary modules necessary to create a holistic robotic software system are presented. **Chapter 5** presents the results from testing in the simulator and with the real quadcopter, with most emphasize on the former. Small discussions are given after each results section. **Chapter 6** gives a conclusion to the thesis and suggests future work on the topic.

Chapter 2

Theory

2.1 Computer vision

2.1.1 Traditional computer vision

Perception has been a challenging problem for many years in real-world robotics. Although a lot of research on this field has been done in recent years, the perception systems available today still require expert knowledge about the current situation in which they are applied. As there are many subareas to perception, including object recognition, object tracking, 3D environmental representation, vehicle detection and human detection, various components have to be put together and customized for the perception system to work [22]. There is in other words no universal solution to the problem of robotic perception.

There is also a fundamental difference between robotic vision and regular computer vision that makes the problem even more challenging. While the output from a regular computer vision system is commonly just used to infer information from images, the output from a robotic vision system is used to take actions in the real world. This means that robotic vision systems have higher requirements to their reliability and should preferably have an estimate of the uncertainty in the predictions [23]. In addition, many computer vision algorithms are tested in confined environments with control over all possible objects that may appear. Robotic vision on the other hand will generally be applied to open set conditions that contain unknown and unseen objects which also must be detected and handled correctly [24].

Traditional computer vision usually have a pipeline of three steps. In the first step, a feature detector is used to find points of interest, or features, in the image. These are easily distinguishable points, such as edges and corners. Examples of feature detectors are the Canny edge detector [25] and the Harris corner detector [26]. In the second step, the characteristics of each feature found in the first step are described using a feature descriptor, such as Scale Invariant Feature Transform (SIFT) [27], Speeded-Up Robust Features (SURF) [28], Oriented Fast and Rotated Brief (ORB) [29] or various Hough transforms [30]. In the third step, a classification on the described features is done. Typically tra-

ditional Machine Learning (ML) algorithms are used for this task, such as the Support Vector Machine [31] or K-Nearest Neighbours [32].

In recent years, the development of Neural Networks (NNs) and Deep Learning (DL) methods have led to many of the currently best algorithms for several computer vision tasks. DL methods embed all three steps of the traditional methods into a single NN, that performs both the feature extraction (detection and description) and classification. While traditional computer vision methods require a lot of engineering and low level construction of feature extractors, NNs and DL benefit from their ability to learn from observational data [33].

Object detection methods, such as You Only Look Once (YOLO) [19] and image segmentation methods, such as Mask R-CNN [21], are examples of methods that use DL to achieve their tasks. These methods learn from labeled data how to classify objects and locate them in the image. Image segmentation methods can even learn which pixels belong to an object and which belongs to the background. For object detection methods, the labeled input data consist of images with a bounding box around each object of interest and a label saying to which category the object belongs. Input data for image segmentation is similar, but every pixel belonging to an object has to be labeled. After the NN is trained on the input data, it can be shown a new image that is not from the training data and be able to predict the class of each object along with a prediction for a bounding box or pixel map around the object(s).

One of the reasons why DL methods have become so increasingly popular is that they do not require the expert analysis for creating hand-crafted feature extractors, which is necessary with traditional methods. Instead, the focus have shifted to choose the best DL architecture for each task. Other reasons for their popularity are that DL methods can exploit and learn from the vast amount of information available from large datasets and that they can be retrained to fit to another dataset than they were originally trained for, which gives them more flexibility [34].

Despite the almost exclusive focus on DL in the recent research, the traditional methods should not be entirely discarded. Walsh et al. argue that traditional computer vision techniques can still be useful, especially for 3D applications and when the recognition problem is sufficiently simple [34]. Among the benefits of traditional computer vision methods, they point out that they have full transparency, so it is possible to understand the reasoning behind the prediction output. This is in contradiction to DL methods where most of the reasoning happens inside a 'black box' of hidden neural layers. Furthermore, they state that it is easier to manually change parameters in a traditional computer vision method to adapt it to another environment. For a DL methods this is infeasible, due to the vast amount of parameters in such models. A final point they make is that for traditional computer vision methods, concrete knowledge about the domain, also known as priors (from prior knowledge), can be directly implemented into the algorithm to improve the classification.

Thus, it seems like the choice between DL and traditional methods has to be done individually for each perception application. For this project, the perception problem is to detect and locate a landing platform of known shape, size and layout. It is only this one object that will be detected and it will not change during operation time, although occlusions may occur and lighting conditions may alter how the colors on the landing

platform are perceived. Therefore, this problem does not need the ability of DL methods to detect many classes of objects. More valuable is the opportunity with traditional methods to include priors, such as the known shape, radius and colors of the landing platform, into the algorithm. Furthermore, human intuition can be applied in choosing manually which features to look for. The landing platform will look different from various altitudes, and for very low altitudes only parts of the landing platform will be visible. The perception system therefore has to be customized for all these different cases, and this is probably easier to do with the traditional methods. Consequently, traditional methods are chosen over DL methods in this project due to the nature of the perception problem.

In the following subsections, two different feature extraction methods from traditional computer vision is presented, namely the Canny edge detector and the Harris Corner detector. They are chosen because of their widespread use and that they are freely available in the open source library OpenCV. Furthermore, a method to fit an ellipse to a set of points is presented along with the theory for how to extract the center and the length of the axes of an ellipse. Finally, the color space used to represent the images is presented.

2.1.2 Edge detection

The edge detector chosen for this project is the Canny edge detector [25]. It is a four step algorithm and is available as one function in OpenCV [35].

The first step is to remove noise in the image using a Gaussian filter with kernel size 5x5.

The second step is to find the edge gradient and direction for each pixel. The Sobel operator, with a default kernel size of 3, is used to approximate the gradient of the image I. It works by finding the first derivative in the horizontal direction

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I \tag{2.1}$$

and the first derivative in the vertical direction

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I. \tag{2.2}$$

From this, the edge gradient, G, and the edge direction, θ, can be found for every pixel in the image:

$$G = \sqrt{G_x^2 + G_y^2}, \tag{2.3}$$

$$\theta = \arctan(\frac{G_y}{G_x}). \tag{2.4}$$

The third step is a non-maximum suppression for every pixel in the image. A pixel is kept as an edge point candidate if it is a local maximum in the direction of its gradient, or else it is discarded.

The fourth and final step is a hysteresis threshold. Edge candidates with a image intensity gradient higher than the upper threshold value are classified as edges. Similarly, edge

candidates with a gradient lower than the lower threshold value are discarded. Edge candidates with a gradient value in between those two thresholds are kept if they are connected to other pixels that are already classified as edges, if not they are discarded.

2.1.3 Corner detection

A much used corner detector and the one that will be used in this project is the Harris corner detector [26]. The detector was originally developed for motion tracking and is translation and rotation invariant, although it is not invariant to scale. There is an implementation of this method available in the OpenCV library which also presents it in a tutorial [36].

Finding corners in an image is the same as finding points (x, y) where any small perturbation (u, v) leads to a large change in image intensity. Corner candidates can be found by maximizing

$$E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2, \tag{2.5}$$

where $w(x, y)$ is a blockSize x blockSize window around the point (x, y) and $I(x, y)$ is the image intensity at the point (x, y). This equation can be approximated using the Taylor series expansion to get

$$\begin{aligned} E(u, v) &\approx \sum_{x,y} w(x, y)[I(x, y) + uI_x + vI_y - I(x, y)]^2 \\ &= \sum_{x,y} w(x, y)[uI_x + vI_y]^2 \\ &= \sum_{x,y} w(x, y)[u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2], \end{aligned} \tag{2.6}$$

using the Sobel operator to find the derivatives I_x and I_y. The approximation can be written on matrix form

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}, \tag{2.7}$$

where

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}. \tag{2.8}$$

Finally, a corner response function, R, can be defined

$$R = det(M) - k(trace(M))^2, \tag{2.9}$$

where

$$\begin{aligned} det(M) &= \lambda_1 \lambda_2 \\ trace(M) &= \lambda_1 + \lambda_2 \end{aligned} \tag{2.10}$$

and λ_1 and λ_2 are the eigenvalues of the matrix M. Any point that has a R value larger than a threshold is detected as a corner. The k parameter in Equation 2.9 is chosen empirically

in the range [0.04, 0.06]. The other parameters that can be chosen are the size of the window $w(x, y)$ and the aperture size of the Sobel operator for calculating the derivatives I_x and I_y.

2.1.4 Fitting an ellipse to a set of image points

The problem of fitting an ellipse to a set of points in an image can be solved with a non-iterative algorithm using least squares minimization proposed in a paper by Radim Halír and Jan Flusser [37]. The algorithm from this paper is presented in the following section.

Problem formulation: Given a set of N points in an image, $(x_i, y_i), i \in \{1, \ldots, N\}$, find the ellipse that best fits this set, parameterized by $\mathbf{a} = [a, b, c, d, e, f]^\mathsf{T}$.

An ellipse is a special case of a general conic that can be written on the General Cartesian form using the second order polynomial

$$F(x, y) = ax^2 + bxy + cy^2 + dx + ey + f = 0, \tag{2.11}$$

where a, b, c, d, e, f are the parameters of the ellipse, and (x, y) are the coordinates of points lying on it. For a general conic to be an ellipse, the constraint $b^2 - 4ac < 0$ must hold. The scaling of the parameters can be done so that the constraint becomes

$$4ac - b^2 = 1. \tag{2.12}$$

The second order polynomial can be written on vector form

$$F_\mathbf{a}(\mathbf{x}) = \mathbf{x} * \mathbf{a} = 0, \tag{2.13}$$

with

$$\begin{aligned} \mathbf{a} &= [a, b, c, d, e, f]^\mathsf{T} \\ \mathbf{x} &= [x^2, xy, y^2, x, y, 1]. \end{aligned} \tag{2.14}$$

The main concept of the algorithm is to choose the ellipse with parameters $\mathbf{a}$, so that the distance from each point $\mathbf{x}$ to the ellipse is minimized:

$$\min_\mathbf{a} \sum_{i=1}^{N} F(x_i, y_i)^2 = \min_\mathbf{a} \sum_{i=1}^{N} (F_\mathbf{a}(\mathbf{x_i}))^2 = \min_\mathbf{a} \sum_{i=1}^{N} (\mathbf{x}_i * \mathbf{a})^2. \tag{2.15}$$

The minimization problem, minimizing equation 2.11 with the constraint in equation 2.12, can then be stated

$$\begin{aligned} &\underset{\mathbf{a}}{\text{minimize}} && ||\mathbf{D}\mathbf{a}||^2 \\ &\text{subject to} && \mathbf{a}^\mathsf{T}\mathbf{C}\mathbf{a} = 1, \end{aligned} \tag{2.16}$$

with

$$\mathbf{D} = \begin{pmatrix} x_1^2 & x_1 y_1 & y_1^2 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_i^2 & x_i y_i & y_i^2 & x_i & y_i & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N^2 & x_N y_N & y_N^2 & x_N & y_N & 1 \end{pmatrix} \tag{2.17}$$

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.18}$$

Lagrange multipliers gives the following conditions for the optimal solution $\mathbf{a}$:

$$\begin{aligned} \mathbf{S}\mathbf{a} &= \lambda \mathbf{C}\mathbf{a} \\ \mathbf{a}^\mathsf{T}\mathbf{C}\mathbf{a} &= 1, \end{aligned} \tag{2.19}$$

with

$$\mathbf{S} = \mathbf{D}^\mathsf{T}\mathbf{D} = \begin{pmatrix} S_{x^4} & S_{x^3y} & S_{x^2y^2} & S_{x^3} & S_{x^2y} & S_{x^2} \\ S_{x^3y} & S_{x^2y^2} & S_{xy^3} & S_{x^2y} & S_{xy^2} & S_{xy} \\ S_{x^2y^2} & S_{xy^3} & S_{y^4} & S_{xy^2} & S_{y^3} & S_{y^2} \\ S_{x^3} & S_{x^2y} & S_{xy^2} & S_{x^2} & S_{xy} & S_{x} \\ S_{x^2y} & S_{xy^2} & S_{y^3} & S_{xy} & S_{y^2} & S_{y} \\ S_{x^2} & S_{xy} & S_{y^2} & S_{x} & S_{y} & S_{1} \end{pmatrix}, \tag{2.20}$$

where $S_{x^a y^b} = \sum_{i=1}^{N} x_i^a y_i^b$.

Equation 2.19 is solved by generalized eigenvectors. There exist up to six real solutions $(\lambda_j, \mathbf{a}_j)$, but since

$$||\mathbf{D}\mathbf{a}||^2 = \mathbf{a}^\mathsf{T}\mathbf{D}^\mathsf{T}\mathbf{D}\mathbf{a} = \mathbf{a}^\mathsf{T}\mathbf{S}\mathbf{a} = \lambda\mathbf{a}^\mathsf{T}\mathbf{C}\mathbf{a} = \lambda, \tag{2.21}$$

it is the eigenvector $\mathbf{a}_k$ corresponding to minimal positive eigenvalue λ_k that needs to be found.

After ensuring $\mathbf{a}_k^\mathsf{T}\mathbf{C}\mathbf{a}_k = 1$, the solution to the minimization problem is found and $\mathbf{a}$ is the parameters for an ellipse that best fits the given set of points.

There are several problems with this approach however, as stated in the paper. These includes that matrix $\mathbf{C}$ is singular and that matrix $\mathbf{S}$ is singular if all points lie exactly on an ellipse. These two facts makes the computation of the eigenvalues numerically unstable. The paper presents some adjustments and simplifications to solve these problems and make the approach numerically stable, by exploiting the special structures of the matrices $\mathbf{C}$ and $\mathbf{S}$. These adjustments are presented below:

First, the matrix $\mathbf{D}$ is split into one quadratic part and one linear part:

$$\mathbf{D} = (\mathbf{D}_1 | \mathbf{D}_2), \tag{2.22}$$

where

$$\mathbf{D}_1 = \begin{pmatrix} x_1^2 & x_1y_1 & y_1^2 \\ \vdots & \vdots & \vdots \\ x_i^2 & x_iy_i & y_i^2 \\ \vdots & \vdots & \vdots \\ x_N^2 & x_Ny_N & y_N^2 \end{pmatrix}, \mathbf{D}_2 = \begin{pmatrix} x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots \\ x_i & y_i & 1 \\ \vdots & \vdots & \vdots \\ x_N & y_N & 1 \end{pmatrix}, \tag{2.23}$$

Secondly, the $\mathbf{S}$ matrix is split:

$$\mathbf{S} = \left(\begin{array}{c|c} \mathbf{S}_1 & \mathbf{S}_2 \\ \hline \mathbf{S}_2^\mathsf{T} & \mathbf{S}_3 \end{array}\right), \tag{2.24}$$

where

$$\mathbf{S}_1 = \mathbf{D}_1^\mathsf{T}\mathbf{D}_1 \tag{2.25}$$
$$\mathbf{S}_2 = \mathbf{D}_1^\mathsf{T}\mathbf{D}_2 \tag{2.26}$$
$$\mathbf{S}_3 = \mathbf{D}_2^\mathsf{T}\mathbf{D}_2. \tag{2.27}$$

Furthermore, the $\mathbf{C}$ matrix only contains information in the upper left corner and it can be split into

$$\mathbf{C} = \left(\begin{array}{c|c} \mathbf{C}_1 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array}\right), \tag{2.28}$$

where

$$\mathbf{C}_1 = \begin{pmatrix} 0 & 0 & 2 \\ 0 & -1 & 0 \\ 2 & 0 & 0 \end{pmatrix}. \tag{2.29}$$

A final split is done with the parameter vector $\mathbf{a}$:

$$\mathbf{a} = \left(\begin{array}{c} \mathbf{a}_1 \\ \hline \mathbf{a}_2 \end{array}\right), \text{where } \mathbf{a}_1 = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, \mathbf{a}_2 = \begin{pmatrix} d \\ e \\ f \end{pmatrix}, \tag{2.30}$$

Using the new way to write the matrices $\mathbf{S}$ and $\mathbf{C}$ and vector $\mathbf{a}$, equation 2.19 can be rewritten:

$$\mathbf{Sa} = \lambda\mathbf{Ca} \implies \left(\begin{array}{c|c} \mathbf{S}_1 & \mathbf{S}_2 \\ \hline \mathbf{S}_2^\mathsf{T} & \mathbf{S}_3 \end{array}\right) * \left(\begin{array}{c} \mathbf{a}_1 \\ \hline \mathbf{a}_2 \end{array}\right) = \lambda * \left(\begin{array}{c|c} \mathbf{C}_1 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array}\right) * \left(\begin{array}{c} \mathbf{a}_1 \\ \hline \mathbf{a}_2 \end{array}\right), \tag{2.31}$$

which again can be written as the pair of equations:

$$\mathbf{S}_1\mathbf{a}_1 + \mathbf{S}_2\mathbf{a}_2 = \lambda\mathbf{C}_1\mathbf{a}_1 \tag{2.32}$$
$$\mathbf{S}_2^\mathsf{T}\mathbf{a}_1 + \mathbf{S}_3\mathbf{a}_2 = 0. \tag{2.33}$$

The matrix

$$\mathbf{S}_3 = \mathbf{D}_2^\mathsf{T}\mathbf{D}_2 = \begin{pmatrix} S_{x^2} & S_{xy} & S_x \\ S_{xy} & S_{y^2} & S_y \\ S_x & S_y & S_1 \end{pmatrix} \tag{2.34}$$

is singular only if all points lie on a line and it is regular otherwise. If all points lie on a line, then there is no solution to the ellipse-fitting problem. If there is a solution however, $\mathbf{S}_3$ is regular and therefore invertible, and equation 2.33 can be written

$$\mathbf{a}_2 = -\mathbf{S}_3^{-1}\mathbf{S}_2^{\mathsf{T}}\mathbf{a}_1. \tag{2.35}$$

This can again be inserted in equation 2.32:

$$\mathbf{S}_1\mathbf{a}_1 + \mathbf{S}_2(-\mathbf{S}_3^{-1}\mathbf{S}_2^{\mathsf{T}}\mathbf{a}_1) = \lambda\mathbf{C}_1\mathbf{a}_1 \tag{2.36}$$

$$(\mathbf{S}_1 - \mathbf{S}_2\mathbf{S}_3^{-1}\mathbf{S}_2^{\mathsf{T}})\mathbf{a}_1 = \lambda\mathbf{C}_1\mathbf{a}_1. \tag{2.37}$$

Since $\mathbf{C}_1$ is regular, it is also invertible and equation 2.37 can be written

$$\mathbf{C}_1^{-1}(\mathbf{S}_1 - \mathbf{S}_2\mathbf{S}_3^{-1}\mathbf{S}_2^{\mathsf{T}})\mathbf{a}_1 = \lambda\mathbf{a}_1 \tag{2.38}$$

The simplification of the matrix $\mathbf{C}$ gives that

$$\mathbf{a}^{\mathsf{T}}\mathbf{C}\mathbf{a} = 1 \implies \mathbf{a}_1^{\mathsf{T}}\mathbf{C}_1\mathbf{a}_1 = 1. \tag{2.39}$$

The conditions for the optimal solution in equation 2.19 can then be written:

$$\begin{aligned} \mathbf{M}\mathbf{a}_1 &= \lambda\mathbf{a}_1, \text{ with } \mathbf{M} = \mathbf{C}_1^{-1}(\mathbf{S}_1 - \mathbf{S}_2\mathbf{S}_3^{-1}\mathbf{S}_2^{\mathsf{T}}) \\ \mathbf{a}_1^{\mathsf{T}}\mathbf{C}_1\mathbf{a}_1 &= 1 \\ \mathbf{a}_2 &= -\mathbf{S}_3^{-1}\mathbf{S}_2^{\mathsf{T}}\mathbf{a}_1 \\ \mathbf{a} &= \left(\frac{\mathbf{a}_1}{\mathbf{a}_2}\right), \end{aligned} \tag{2.40}$$

The task to solve the problem then becomes to find the appropriate eigenvector $\mathbf{a}_1$ of the matrix $\mathbf{M}$.

2.1.5 Finding the center and axes from ellipse parameter

When the six ellipse parameters, $\mathbf{a} = [a, b, c, d, e, f]^{\mathsf{T}}$, are found with the method presented in the previous section, they can be used to calculate the center of the ellipse, (x_0, y_0), and the minor and major axes, l_a and l_b. First, the standard equation for an ellipse is considered:

$$\frac{x^2}{l_a^2} + \frac{y^2}{l_b^2} = 1, \tag{2.41}$$

where l_a and l_b are the lengths of the minor and major axis, parallel with the x- and y-axis, respectively. Then, the center is translated to (x_0, y_0) and the ellipse is rotated with and angle θ about the center (Figure 2.1), using the following transformation [38]:

$$\begin{aligned} x &= (x' - x_0)\cos\theta + (y' - y_0)\sin\theta \\ y &= -(x' - x_0)\sin\theta + (y' - y_0)\cos\theta \end{aligned}. \tag{2.42}$$

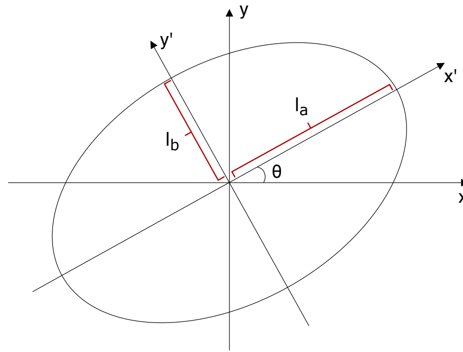


Figure 2.1: An ellipse tilted at an angle θ.

When substituting x and y in Equation 2.41 with Equation 2.42, the result can be written on the General Cartesian form in Equation 2.11 using the following relations:

$$\begin{aligned} a &= l_a^2 \sin\theta + l_b^2 \cos\theta \\ b &= 2(l_b^2 - l_a^2)\sin\theta\cos\theta \\ c &= l_a^2 \cos\theta + l_b^2 \sin\theta \\ d &= -2ax_0 - by_0 \\ e &= -bx_0 - 2cy_0 \\ f &= ax_0^2 + bx_0y_0 + cy_0^2 - l_a^2 l_b^2 \end{aligned} \quad . \tag{2.43}$$

Conversely, the center, (x_0, y_0), and the minor and major axes, l_a and l_b, can be found from the General Cartesian form with

$$\begin{aligned} x_0 &= \frac{2cd - be}{b^2 - 4ac} \\ y_0 &= \frac{2ae - bd}{b^2 - 4ac} \end{aligned} \tag{2.44}$$

and

$$l_a, l_b = \frac{-\sqrt{2(ae^2 + cd^2 - bde + (b^2 - 4ac)f)((a + c) \pm \sqrt{(a - c)^2 + b^2})}}{b^2 - 4ac}. \tag{2.45}$$

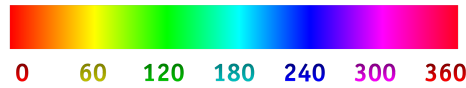


Figure 2.2: The hue range from 0 to 360. Source: [39].

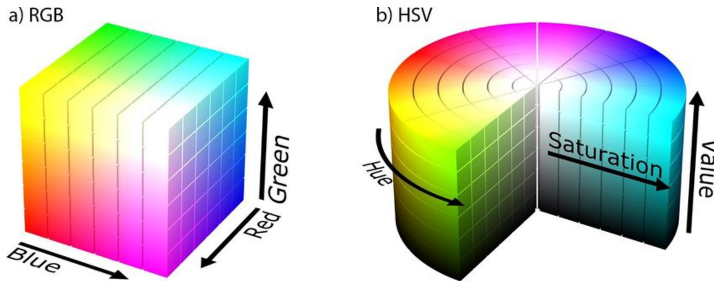


Figure 2.3: Comparison of the color spaces (a) RGB and (b) HSV. Source: [39].

2.1.6 HSV color space

Digital images can be represented and stored on a computer using different color spaces. The most common is to use the RGB (Red, Green, Blue) color space. When using RGB, each pixel is defined with three parameters, the amount of Red, Green and Blue. Every color, including black and white, can be made by combining those three parameters. One drawback with the RGB color space however, is that it is hard for a human to combine the three RGB values to create a given color.

Therefore, when working with computer imaging, it can be beneficial to use the HSV (Hue, Saturation, Value) color space instead. In the HSV color space, the different ”colors” as a human eye sees them are defined by only one parameter, the Hue parameter. Figure 2.2 shows the hue range and indicates that one color can be found by specifying an interval on the hue range. The two other parameters in the HSV color space defines the ”brightness” of the color (value) and the ”intensity” of the color (saturation). The different structures of the two color spaces are visualized in Figure 2.3.

2.2 Camera pose estimation

The problem of estimating the pose of a camera from N image points with known corresponding 3D points is known as the Perspective-n-Problem (PnP). Most solutions that have been implemented require $N >= 4$, although there exist situations where $N = 3$ is sufficient [40]. The method used in this thesis resembles the common solutions to the PnP problem. However, there will be only one image point with a known corresponding 3D point available, namely the center of the landing platform. Nevertheless, two additional pieces of information are available that render possible a solution. First of all, there is a distance in the image corresponding to a real known distance, namely the radius of the landing platform. Secondly, the orientation of the only available 3D point is assumed

known.

No published solution for this problem was found after a reasonable amount of searching, although it is probably done before. Therefore, a solution to this problem is derived in this thesis and a detailed description of the solution is provided in Section 4.1.6. The basis of the solution uses the pinhole camera model to describe the image formation process and to map a pixel point to a 3D point.

2.3 Quadcopter dynamics

The basic dynamics for a quadcopter are available in the literature [41], and are presented below:

$$\begin{aligned} m\ddot{x} &= (\sin\psi\sin\phi + \cos\psi\cos\phi\sin\theta)u_1 \\ m\ddot{y} &= (-\cos\psi\sin\phi + \sin\theta\sin\psi\cos\phi)u_1 \\ m(\ddot{z} + g) &= \cos\theta\cos\phi u_1 \\ I_{xx}\ddot{\phi} + (I_{zz} - I_{yy})\dot{\theta}\dot{\psi} &= u_2 \\ I_{yy}\ddot{\theta} + (I_{xx} - I_{zz})\dot{\phi}\dot{\psi} &= u_3 \\ I_{zz}\ddot{\psi} &= u_4 \end{aligned}, \tag{2.46}$$

where ϕ is the roll angle, θ is the pitch angle and ψ is the yaw angle. These angles are the rotations about the x-, y- and z-axes, respectively, measured in the body coordinate system of the quadcopter. u_1 is the control input for controlling the altitude z and u_2, u_3, u_4 are the control inputs for controlling the angles ϕ, θ, ψ, respectively. m is the mass of the quadcopter and I_{xx}, I_{yy}, I_{zz} are the moments of inertia in the x-axis, y-axis and z-axis respectively.

These are the most important dynamics of a quadcopter. However, they do not include effects from aerodynamics, such as ground effects when the quadcopter is close to the ground and other disturbances such as wind gusts. These effects are not included in the simulated model of the quadcopter either and may make the transition from the simulator to the real quadcopter more difficult. The computer vision system that is developed in this thesis is not made to be universal, but is designed to be used with a quadcopter. The knowledge about how the vehicle might behave can therefore be used to make the computer vision system more robust and to be prepared for challenges caused by the quadcopter dynamics.

Chapter 3

Design of landing platform and experimental setup

This chapter presents the design and assembly process of making the landing platform. Furthermore, it presents the hardware and software used in this project.

3.1 The landing platform

The motive on the landing platform, as shown in Figure 3.1, is chosen with an eye to mimic real landing platforms typically found on large ships and maritime installations to be used by helicopters. It has the characteristic white letter 'H' in the center with an orange circle around it. An orange arrow is located on the orange circle to indicate the forward direction of the ship that the landing platform will be placed on. Those features are placed on a green background that gives good contrast to the orange circle and should in most cases also give good contrast to the ship and to the sea surrounding it. The landing platform is modeled and visualized in the simulator as well as created in a physical version. It is designed to specifically suit this project, but at the same time it is made with robust and durable materials, so that it can be used as a resource for other projects in the future. The design of the layout is an improvement from the one made in the project report [1], with the main changes being that the orange arrow is added and a fiducial marker is removed.

Figure 3.1: The motive on the landing platform.

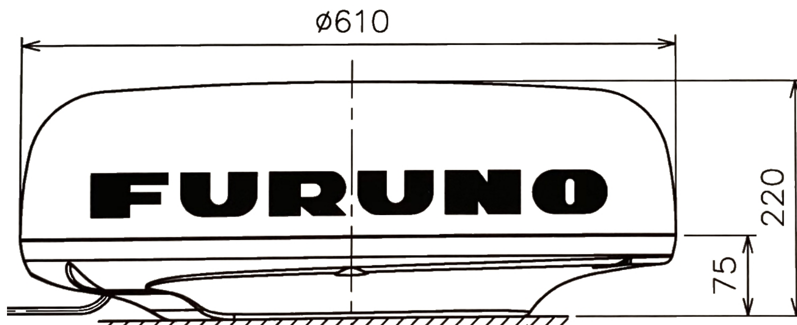


Figure 3.2: Dimensions of the radar seen from the side. Source: [42].

3.1.1 Plan for attachment to the Revolt

Since there is limited space available on the Revolt vessel, it is agreed with DNV GL that the best location to put the landing platform is on top of a radar on the ship. This is a suitable place, since the radar has a circular shape with approximately the same diameter as the quadcopter.

Because the radar itself will be mounted on a high place on the Revolt, any extra weight applied on top of that will significantly alternate the ship's center of mass. In order to make the landing platform as lightweight as possible, a minimalistic design is chosen, consisting mainly of a circular plate with a sticker on top of it. The quadcopter is 73 cm on its widest span, so a diameter of 80 cm was chosen. This is the smallest diameter possible that still leaves some margins for landing the quadcopter. The material of the plate is chosen to be 3mm plexiglass, which, with its low density of 1.19 g/cm^3, makes the plate itself weigh 1.8 kg.

Furthermore, because any material placed on or around the radar may interfere with its functioning, it is specified that it has to be easy to remove the landing platform whenever necessary. The solution chosen is to use hook-and-loop between the landing platform and the radar for easy attachment and detachment. The hook-and-loop is mounted quite firmly to the radar using its self-adhesive backside. However, it is tested with a small patch of hook-and-loop that it is removable without leaving any marks on the radar.

The main challenge with this location, from the design point of view, is the curved surface of the radar (see Figure 3.2). This means that the landing platform can not be placed directly on the top. To cope with the curvature, a stand is designed to follow this curve and make a leveled support for mounting the top plate. The stands are 3D printed by the workshop at ITK. Figure 3.3 shows how the 3D printed stand fits to the curve of the radar. The top of the stand is a bit higher than the highest point of the radar, leaving a small gap between the radar and the landing platform. This way, the weight of the top plate is equally distributed across the three stands and not directly onto the radar. Figure 3.4 shows the overall plan for mounting the landing platform to the radar.

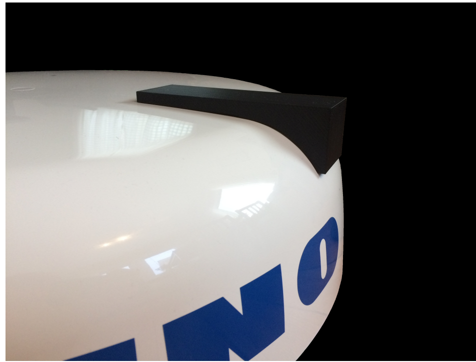


Figure 3.3: The 3D printed support stand resting on the radar.

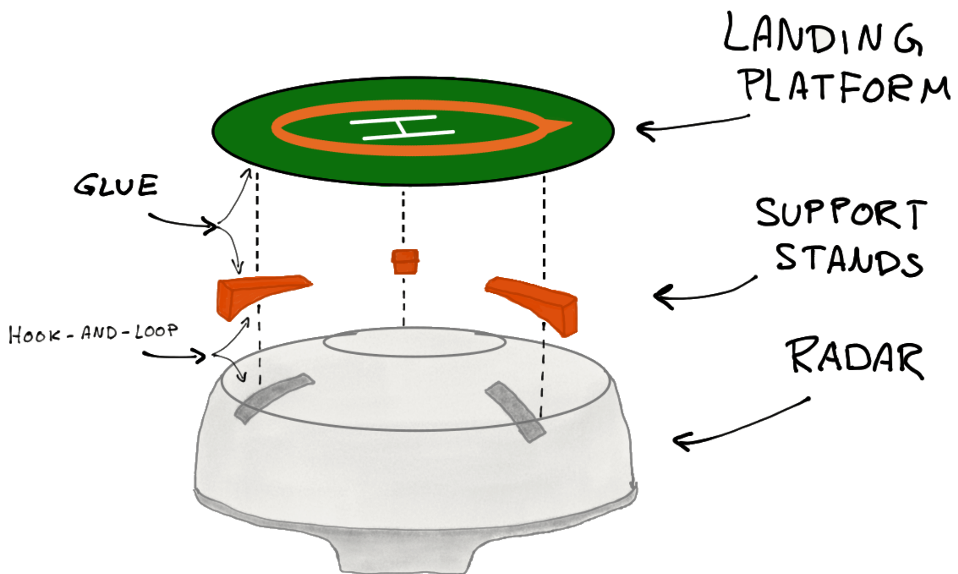
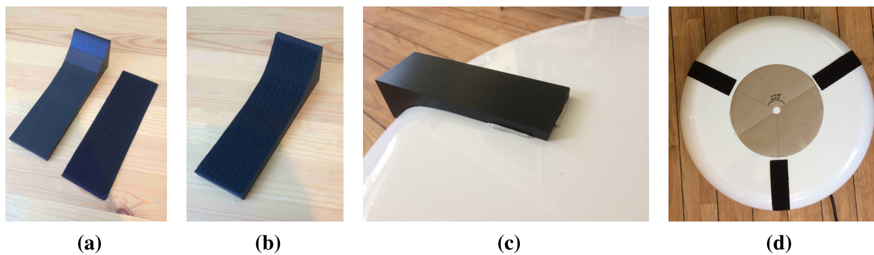


Figure 3.4: An early sketch of how to mount the landing platform to the radar.



(a) (b) (c) (d)

Figure 3.5: Mounting the stands to the radar.

3.1.2 Assembling the landing platform

Materials

- A vinyl sticker, 780 mm in diameter, ordered from NTNU Grafisk.
- A plexiglass plate, 800 mm in diameter, 3mm thick, ordered and cut to shape by the workshop.
- Three 3D printed stands, 50mm wide, 155mm long, 45mm high, printed by the workshop.
- Black TEC7 modified silan polymer glue.
- Hook and loop.

Costs

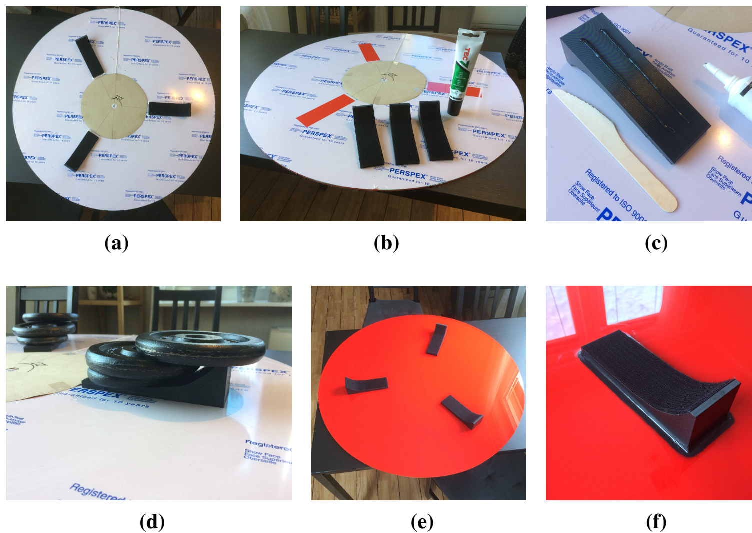
- Hook and loop from Clas Ohlson: 160,00
- Sticker foil from NTNU Grafisk: 630,00

Mounting hook-and-loop to the stands and the radar

Strips of hook-and-loop are cut out to fit the inside of the stands (Figure 3.5a). The hook-and-loop has a self-adhesive backside and is easily attached to the stands (Figure 3.5b). The stands are then test fitted on the radar and putty is used to mark their right position (Figure 3.5c). Matching strips of hook-and-loop are glued in place at the right spots around the landing platform with an angle of 120 °between each strip. A cardboard circle is used to ensure the right spacing (Figure3.5d).

Gluing the stands to the plexiglass

While the protective paper is still on, the three stands are aligned on the plexiglass and held in place with putty. The same cardboard circle is used to match the placements on the radar (Figure 3.6a). It is worth to notice that this is a mirrored configuration, since the top of the radar must match the bottom of the plexiglass, and it only works because the angles between all spots are equal. Before the cardboard is removed from the radar in the



(a) (b) (c)

(d) (e) (f)

Figure 3.6: Gluing the stands to the plexiglass.

last step, the forward direction of the radar is marked on it. When the cardboard is placed on the plexiglass, this marks the backward direction which is important to get right in the next step.

To make sure the stands are glued in the right spots, a trimming knife is used to trace around each stand. Then, the protective paper under the stands is easily removed (Figure 3.6b). Two strips of glue are applied to each stand and smoothed out to a thin layer using a wooden knife (Figure 3.6c). Then, each stand is pressed onto its spot and weighted down with weight disks for four hours (Figure 3.6d). According to the producer, the glue has a full cure time of 24 hours in 23°, so any nudges are avoided during this time. After the glue has cured, the protective paper is removed (Figure 3.6e) and an extra seam of glue is applied around each stand (Figure 3.6f).

Applying the sticker

The sticker is ordered from NTNU Grafisk and is delivered on a square paper (Figure 3.7a). The circle is cut out using a pair of scissors and taped to the plexiglass using masking tape, leaving an even space of 10 mm on every side of the sticker. A cotton thread is used to mark the front, rear and center of the landing platform (Figure 3.7b). The landing platform is put on edge (Figure 3.7c) and the tape and protective paper are removed from the lower part of the sticker (Figure 3.7d). To help smooth the sticker out, the soft part of some leftover hook-and-loop is attached to a piece of cardboard (Figure 3.7e). Then, the sticker is applied from the center towards the bottom (Figure 3.7f), before the process is repeated from the center towards the top (Figure 3.7g). Inevitably, when applying such a large sticker by an untrained hand, some bubbles are left in the sticker. However, when looking at the bubbles from a distance, they merely resembles small water droplets and will not be

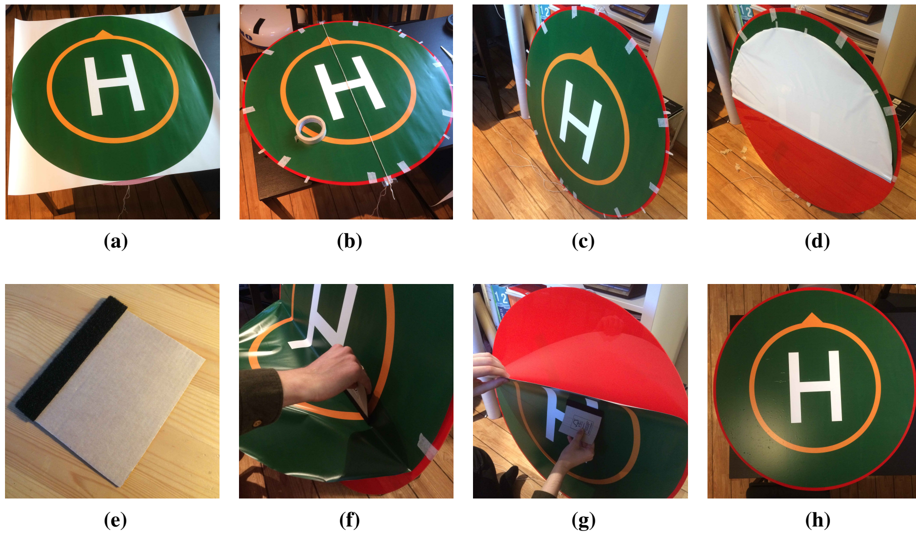


Figure 3.7: Attaching the sticker to the plexiglass.

a problem for this project (Figure 3.7h).

The finished landing platform

The finished landing platform weighs 2 kg, which is a reasonable weight to add on the ship. It is attached to the radar for a final test fit (Figure 3.8). The stands fit well on the curved surface and the top plate form a stable, level surface to land on. When removing the landing platform from the radar again, a substantial amount of force has to be applied to separate the hook-and-loop from each other. This means that the landing platform is rigidly enough attached, but still detachable when necessary, so the design specifications are met.

3.2 The quadcopter

The quadcopter used in this project is the reasonably priced AR.Drone 2.0 from the French company Parrot. The quadcopter, shown in Figure 3.9, has a simple construction with two interchangeable hulls. The hull displayed in the figure is for indoor use and another more aerodynamic hull is supplied to be used outdoors. Some of the technical specifications of the quadcopter is available in Appendix A.

The quadcopter is originally intended to be used with a simple and intuitive controller on a smartphone or tablet [43]. However, Parrot has made available a Software Development Kit (SDK) for third party developers to develop their own applications for mobile devices and personal computers [44]. Through this SDK, it is possible to communicate with the quadcopter over WiFi.

(a) **(b)**

Figure 3.8: The finished landing platform attached to the radar.

Figure 3.9: The Parrot AR.Drone 2.0. Source: [43].

Since the quadcopter has limited computational power on board, an auxiliary computer is used in cooperation with the quadcopter. The WiFi connection is used to retrieve images and IMU measurements from the quadcopter. Then, the heavier image processing is performed on the auxiliary computer, which also implements an high level pose controller and sends back control signals to the quadcopter over the WiFi connection.

The quadcopter also has a built-in autopilot. This handles the low-level control of the four motors and the mid-level altitude and attitude control for the quadcopter dynamics presented in section 2.3. External control inputs can be sent to the quadcopter through the SDK in the form of linear and angular velocity set points. The autopilot will then control the quadcopter to maintain these velocities. If the linear and angular velocity set points are set to zero, the autopilot will try to hold the quadcopter at a constant altitude and with a horizontal attitude. The SDK and the built-in autopilot makes the Parrot AR.Drone 2.0 an excellent choice for research projects, since the focus can be on higher level research questions and applications rather than low level control of the quadcopter.

3.3 Software

The software in this project is mostly developed and run on a MSI GF75 Thin laptop, with an Intel® Core™ i7-9750H CPU @ 2.60GHz × 12 and 16 GB RAM, with Ubuntu 16.04 operative system.

3.3.1 ROS

The Robot Operating System (ROS) is a software developing framework intended for robotic systems [45]. It is not actually an operating system, as the name implies, but more a set of tools and libraries making the development more convenient. Terms in this thesis related to the ROS framework are written in *italics*. ROS comes in different distributions and the distribution used in this thesis is the ROS Kinetic, because some of the packages needed was best compatible with this distribution.

ROS has several features that enables a modular structure of the code, where different modules, or *nodes*, can be developed independently of each other. The nodes can communicate with each other using message passing over different channels, called *topics*. A node can *subscribe* to or *publish* different *messages* to a topic. Different nodes can even be written in different languages. The ROS distribution used in this project, ROS Kinetic, is compatible with C++ and Python 2.7.

Another benefit of using ROS is the global community around it and that there already exist many good open-source *packages*. This means the software does not have to be built entirely from scratch, but can rely on already existing work of others. One example of an open-source package that is used in this project is the *ardrone_autonomy* package for interaction with the quadcopter. This package is based on the official SDK from Parrot and can be used as a driver for the AR.Drone 2.0 [46]. Another example is the *uuv_simulator* package [2]. It contains all the necessary nodes for simulating Unmanned Underwater Vehicles (UUVs). Only a small fraction of the package is used in this project. It includes a realistic model of an ocean that is used to get more visually correct images from the quadcopter in the simulator.

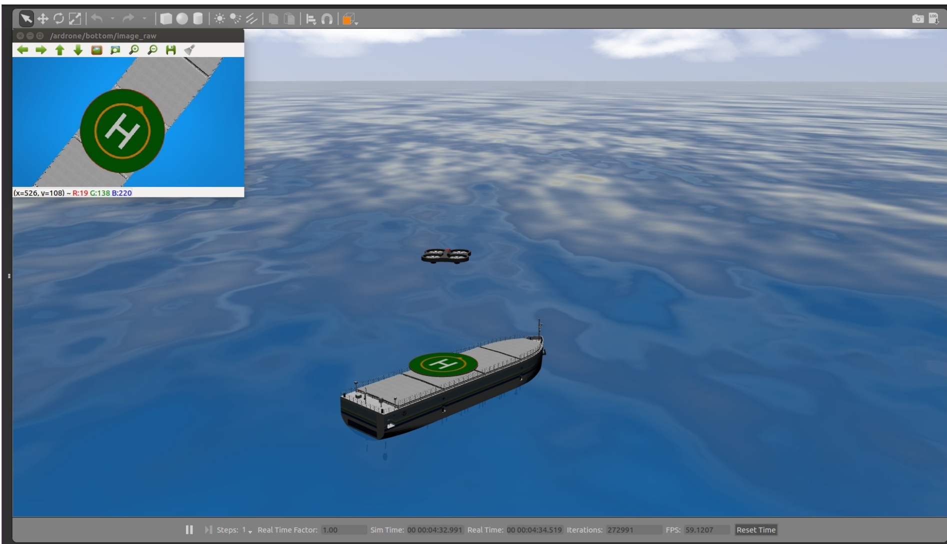


Figure 3.10: The simulated environment with the Revolt vessel, the landing platform and the quadcopter. The image in the top left corner is the image from the bottom camera.

In this project, several different built-in message types are used. These most used messages are listed below:

- Empty()
- Image()
- Twist()
- Odometry()
- Joy()

The content of each message is listed in Appendix B.

3.3.2 The Gazebo simulator

The simulator used in this project is the Gazebo simulator, version 7.0.0.

The Parrot AR.Drone also has its own simulated version, which can be launched as a ROS package and ran in the Gazebo simulator. For simulation of ocean waves, the Unmanned Underwater Vehicle Simulator is used together with Gazebo [2]. A 3D model of the Revolt vessel is provided by DNV GL and this is added to the simulator and the model of the landing platform is placed on top of it.

Figure 3.11: The Sony PlayStation 4 handheld controller used in this project.

3.4 Handheld controller

To allow for easy manual control of the quadcopter, a wireless handheld controller is added to the project. The controller of choice is a Sony PlayStation 4 DualShock wireless controller (see Figure 3.11). This particular controller is chosen because of its high quality and that it is easily connected to any computer over Bluetooth.

Chapter 4

System design

In this chapter, the software system developed in this project is presented. Figure 4.1 shows the different nodes of the system, how they are connected and the main topics over which they communicate with each other. Each node will be presented, with most emphasize on the position estimator node as this is the main contribution. All nodes are implemented in Python 2.7 and integrated with ROS. Furthermore, a theoretical analysis of how to infer 3D position from a 2D image point is presented in section 4.1.6. It is considered a contribution and is therefore included here instead of in the theory section.

4.1 Pose estimation

A pose is a combination of both position and orientation. Knowing it's pose is vital for any robot to be able to navigate itself in an environment. It is chosen in this project to estimate the quadcopter's pose relative to the landing platform, measured in the coordinate system of the landing platform (see Figure. 4.2).

Due to the specific application area, it is possible to do some assumptions prior to the pose estimation process. As mentioned in Section 2.3, the orientation of the quadcopter is stated as the roll-, pitch- and yaw-rotations about the x-, y- and z-axes of the body coordinate system, respectively (see Figure 4.2). However, since the onboard autopilot will try to keep the quadcopter hovering at the same spot, the roll and pitch angles will be controlled towards zero. Therefore, the roll- and pitch-angles are assumed zero and are not estimated. Another assumption that follows directly from this, is that the xy-plane of the quadcopter is parallel to the xy-plane of the landing platform. These assumptions are violated as soon as the quadcopter performs a movement in x or y direction or if windy conditions forces the quadcopter to tilt towards the wind to hold its position. They are also violated when the landing platform itself is exposed to a roll or pitch motion, for instance due to waves or that the ship is performing a maneuver. However, as long as the quadcopter is in hover mode and good weather conditions apply, these assumptions hold. Flight in poor weather conditions and with a moving marine vessel is beyond the scope of this thesis, so the assumptions are legitimate here.

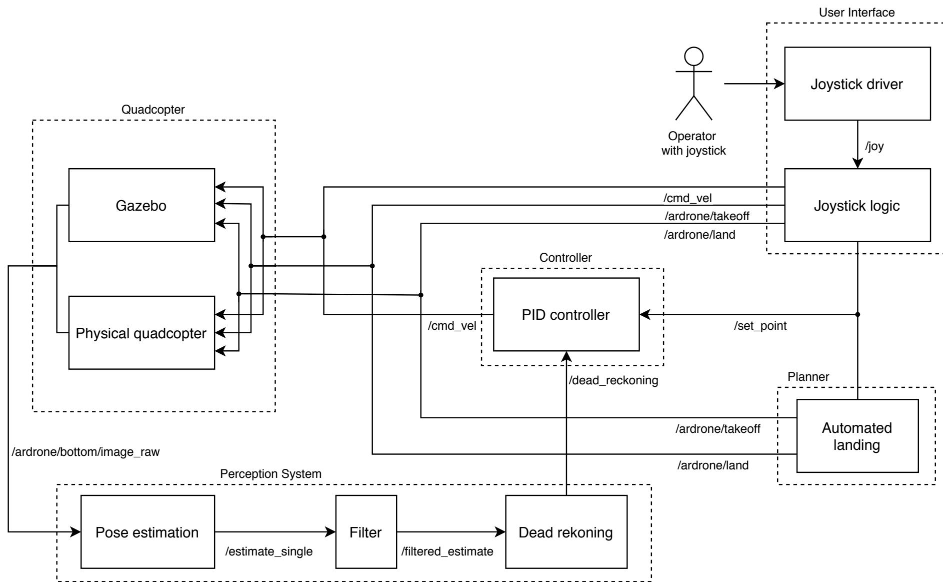


Figure 4.1: Architecture of the ROS application showing how the different nodes of the system interact. An arrow goes from a publisher to a subscriber and denotes the name of the topic on which the messages are sent.

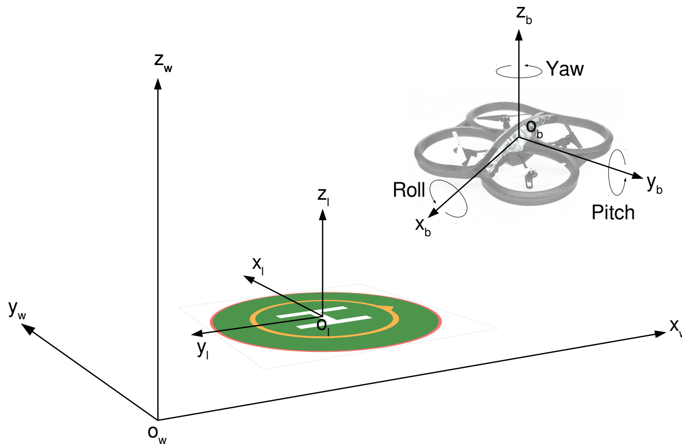
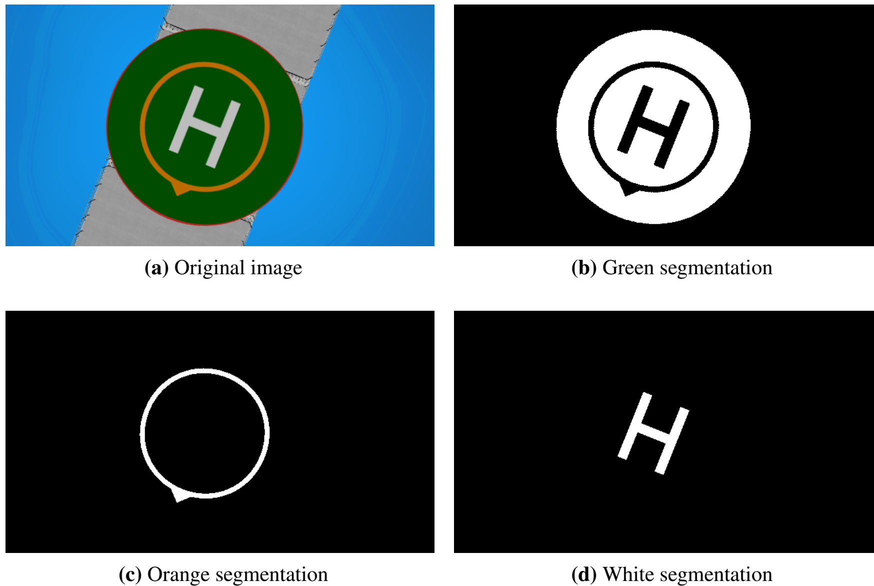


Figure 4.2: The different coordinate systems used with the quadcopter. The world coordinate system is only available in the simulator, where it is used as a ground truth position to check how well the estimate is performing.



(a) Original image (b) Green segmentation

(c) Orange segmentation (d) White segmentation

Figure 4.3: Image segmentation in the simulator.

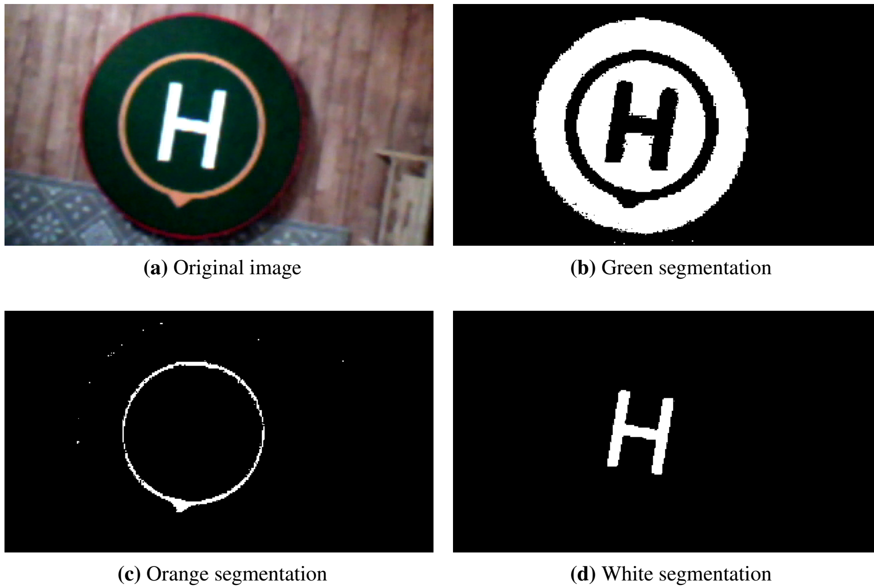
The position estimation method consists of 4 main steps:

1. Color segmentation
2. Corner and edge detection
3. Finding the center and radius of landing platform
4. Calculating the pose

In addition, if enough information is available, the yaw-rotation is calculated.

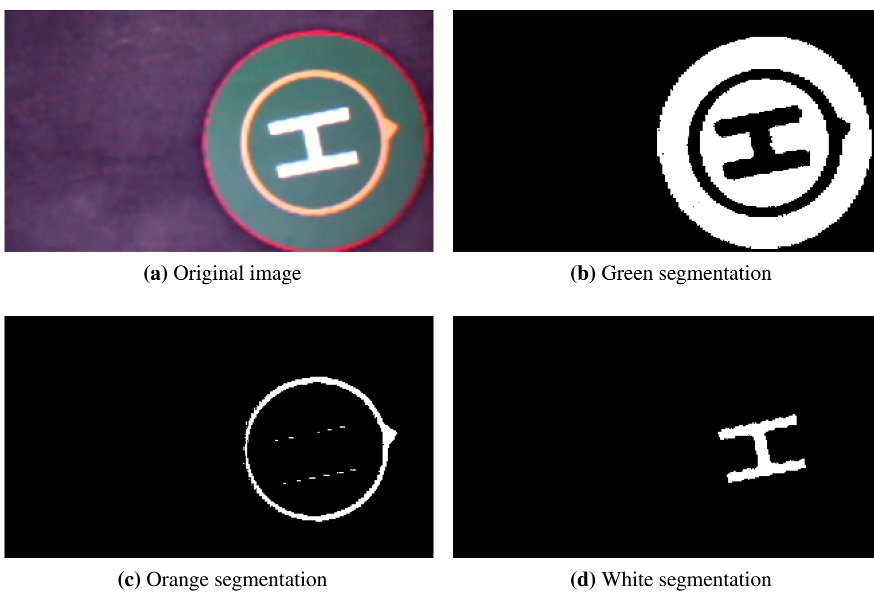
4.1.1 Color segmentation

The first step of the computer vision method is segmenting the different colors in the image. Before the segmentation, noise is removed from the image using the OpenCV function `fastNlMeansDenoisingColored()` [47] as shown in Listing 4.2. Then the OpenCV function `inRange()` is used to find the mask containing only the desired color, as shown in Listing 4.1. The low and high limit for each of the three Hue, Saturation and Value parameters are found by trial and error and the limits are listed in Table 4.1. The colors are perceived a bit differently in the simulator and with the real quadcopter and therefore the limits are adjusted for both cases. Figures 4.3, 4.4 and 4.5 shows examples of image segmentation on an image taken in the simulator, by the real quadcopter indoors and by the real quadcopter outside, respectively.



(a) Original image
(b) Green segmentation
(c) Orange segmentation
(d) White segmentation

Figure 4.4: Image segmentation with images from the physical quadcopter indoors.



(a) Original image
(b) Green segmentation
(c) Orange segmentation
(d) White segmentation

Figure 4.5: Image segmentation with images from the physical quadcopter outdoors.

Color	Parameter	Simulator		Real indoor		Real outdoor	
		Low	High	Low	High	Low	High
Green	Hue	105	135	120	200	120	220
	Saturation	85	100	60	100	0	50
	Value	15	60	0	60	0	60
Orange	Hue	21	51	10	40	10	40
	Saturation	85	100	25	100	20	70
	Value	30	100	60	100	90	100
White	Hue	0	360	0	360	0	360
	Saturation	0	15	0	30	0	7
	Value	30	100	85	100	98	100

Table 4.1: HSV threshold values for color segmentation. The limits are different for the simulator, real indoor and real outdoor use of the quadcopter. The ranges are 0-360 for Hue and 0-100 for Saturation and Value.

Listing 4.1: Color segmentation using the OpenCV library.

```
mask = cv2.inRange(image, low, high)
```

Listing 4.2: Denoising using the OpenCV library.

```
denoised = cv2.fastNlMeansDenoisingColored(image,None,10,10,7,21)
```

4.1.2 Edge detection

Edges are detected in the image by using the Canny edge detection, presented in Section 2.1.2. The edge that needs to be detected is the edge on the outer contour of the green ellipse. To prepare for the edge detection, the `floodFill()` function from OpenCV is applied to the green segmentation. This function fills all connecting pixels in the image, starting from the top left corner. The result is a solid shape with the details on the inside removed. Then the `Canny()` function from OpenCV can be applied, as shown in Listing 4.3.

Listing 4.3: Canny edge detection function.

```
mask = cv2.floodFill(green_segmentation, mask, startPoint)
edges = cv2.Canny(mask, threshold1, threshold2, apertureSize)
edge_points = np.where(edges == 255)
```

The parameters threshold1 and threshold2 are thresholds for the hysteresis procedure and are set to 100 and 200, respectively. The default aperture size for the Sobel operator apertureSize=3 is used. The result is shown in Figure 4.6, where the edge points are marked in red. They are accurately placed and covers the entire edge.

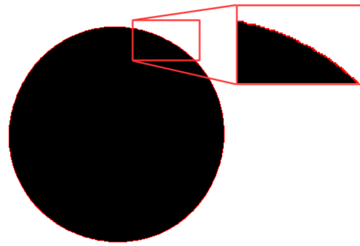


Figure 4.6: The binary mask of the landing platform. The edge points found with the Canny edge detector are marked in red.

4.1.3 Corner detection

Corners are detected in the image by using the Harris corner detector, presented in Section 2.1.3. Inspiration for this way of doing it comes from an OpenCV tutorial [36]. First, the `cornerHarris()` function from OpenCV, displayed in Listing 4.4 is used:

Listing 4.4: The Harris corner detector function from OpenCV.

```
result = cv.cornerHarris(image, blockSize, apertureParam, kParam)
```

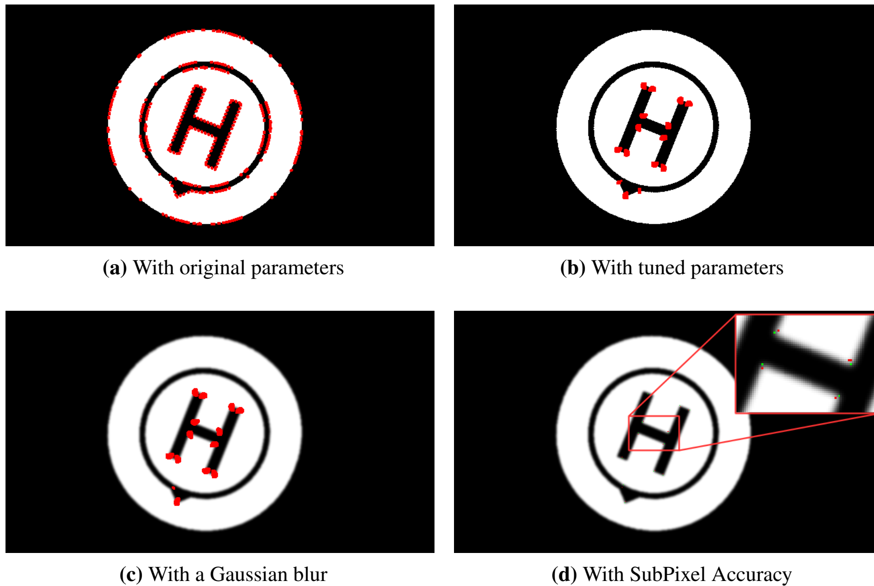
, with the suggested parameter values from the tutorial: blockSize = 2, apertureParam = 3, kParam = 0.04. The input to the function must be a grayscale image. Here, the green segmentation image from the simulator is used as a demonstration image. The result is shown in Figure 4.7a. To remove some of the false corners found on the edge of the circles, aperture parameter of the Sobel operator is increased to apertureParam = 9. Furthermore, the neighbourhood considered is increased to blockSize=7. The corner area found will then be larger and more likely cover the actual corner. The result is shown in Figure 4.7b.

Listing 4.5: The gaussian blur function from OpenCV.

```
image = cv2.GaussianBlur(image, kSize)
```

A Gaussian blur with a kernel size of 9 is applied to the image before the corner detection, using the function shown in Listing 4.5. This helps avoid false corner detections and the result is shown in Figure 4.7c. The blur will have an even larger effect on images from the real quadcopter, where there will be more noise.

Finally, a sub-pixel accuracy method [36], displayed in Listing 4.6, is used to find the best corner in each of the corner areas. This uses the result from the `cornerHarris()` function to calculate the centroids of each corner area. Then it refines the corner locations with the iterative function `cornerSubPix()`. This function considers a search window around each centroid, where win_size denotes half of the side lengths of the window. In this case win_size is set to 5 which gives a search window of 11x11 pixels. It is possible to add a dead region in the center of the search window, but his is not necessary here and it is marked with the value (-1, -1). The criteria for when to stop the iteration is set to



(a) With original parameters

(b) With tuned parameters

(c) With a Gaussian blur

(d) With SubPixel Accuracy

Figure 4.7: Output from the Harris corner function

maximum 100 iterations or when the corner is refined by less than an epsilon value of 0.001 on one iteration.

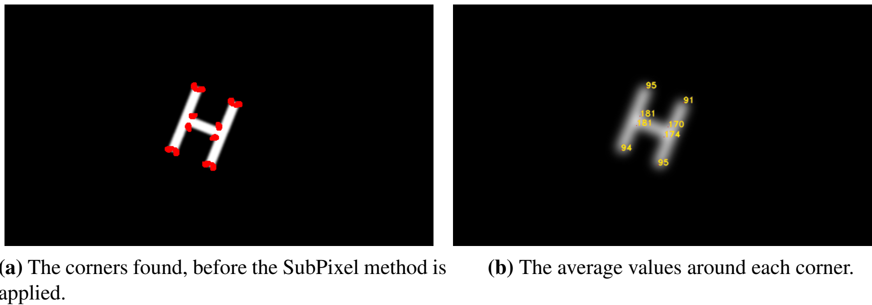
Listing 4.6: Finding corners with sub-pixel accuracy [36].

```
# find centroids
ret, labels, stats, centroids = cv2.connectedComponentsWithStats(result)

# define the criteria to stop and refine the corners
criteria = (cv2.TERM_CRITERIA_MAX_ITER + cv2.TERM_CRITERIA_EPS,
            100, 0.001)
corners = cv2.cornerSubPix(image,np.float32(centroids),
                            (win_size, win_size),(-1,-1),criteria)
```

The result is shown in Figure 4.7d, where the centroids are marked in red and the refined corners are marked in green. The resulting corners are accurately placed in the image and the sub-pixel information, meaning that the corners are denoted by a floating number instead of an integer, means that the calculations later will be even more precise.

Even though the above method gives accurate corners, there still are two refinements left to do. Firstly, any edge that meets the border of the image will appear to be a corner in that intersection. Therefore, all corners closer to the border than 7 pixels, a value found by trial and error, are discarded. Secondly, it is only the inner corners of the 'H' and the arrowhead of the orange arrow that are interesting. To illustrate this, the white segmentation is used. Figure 4.8a shows all the detected corners. To differentiate the inner corners from the rest, the average value around each corner is found, using a Gaussian blur with kernel



(a) The corners found, before the SubPixel method is applied.

(b) The average values around each corner.

Figure 4.8: Corner selection for white segmentation

size 51 on the entire image. The value of the corners location in the blurred image is now the average value. Figure 4.8b shows the blurred image with the values of each corner location denoted on it. Around an inner corner, approximately 75% of the surrounding pixels should be white. A white pixel has a value of 255, so the average value around an inner corner should be $255*75\% \approx 191$. With a margin of ± 30, only corners with a surrounding average between 160 and 210 are kept, the others are discarded.

4.1.4 Finding higher level features

After the low level features have been found, constituting of edges around the border of the green segmentation and corners found in the orange and white segmentations, these can be used to find three higher level features. The first feature is the location of the center of the landing platform, given in pixel coordinates. The second feature is the radius of the landing platform. The location of the radius is not important, only the length, measured in pixels. The third feature is any vector that indicates the orientation of the landing platform.

It is desirable that the position estimation is functioning both when the quadcopter is close to the landing platform and when it is far away. However, the landing platform will appear quite different in these two edge cases. Details on the landing platform will not be easily distinguished when the quadcopter is far away and the entire circular shape of the landing platform will not fit into the camera's field of view when the quadcopter is close to the landing platform. Therefore, there are developed three different feature extraction methods that excel at different, yet overlapping, ranges of height over the landing platform. They are named the *ellipse* method, the *arrow* method and the *corners* method, based on what they use to find the higher level features. The following subsections presents the three methods.

The *ellipse* method

When the quadcopter is far away, the details of the landing platform are not clearly visible. However, the general round shape of the landing platform is distinguishable up to 50 meters above it. The landing platform has a circular shape, but when the quadcopter is not straight above it and it is seen at an angle, it will appear like an ellipse. Therefore, the

ellipse fitting method presented in Section 2.1.4 is used to fit an ellipse to the points on the border of the landing platform, previously found with edge detection on the green segmentation. The ellipse is described with the six ellipse parameters $\mathbf{a} = [a, b, c, d, e, f]^\mathsf{T}$. From these parameters, the center of the ellipse (x_0, y_0) and the length of the axes l_a and l_b are found as described in Section 2.1.5. The center of the landing platform equals the center of the ellipse:

$$(c_x, c_y) = (x_0, y_0), \tag{4.1}$$

and the length of the radius equals the longest of the ellipse axes:

$$r = \max(l_a, l_b). \tag{4.2}$$

The low level features from the green segmentation contains no asymmetry that can be used to infer the orientation of the landing platform.

The *arrow* method

When the quadcopter is a bit closer to the landing platform, the center of the white 'H' and the arrowhead of the orange arrow are detectable and can be used. The center of the landing platform is found by looking at the white 'H' and exploiting that its shape is symmetrical both horizontally and vertically, meaning that its center of mass will be at the center of the shape. After the white is segmented from the rest, as seen in Figures 4.3b and 4.4b, the `moments()` function from OpenCV is used to calculate the spatial moments of the segmentation [48]:

$$m_{ji} = \sum_{x,y} (I(x,y) \cdot x^j \cdot y^i), \tag{4.3}$$

where $I(x, y)$ is the image intensity at pixel (x, y). Then, the center of mass $(\overline{x}, \overline{y})$, i.e. the centroid, is calculated from this matrix [48]:

$$\overline{x} = \frac{m_{10}}{m_{00}} \tag{4.4}$$

$$\overline{y} = \frac{m_{01}}{m_{00}}, \tag{4.5}$$

giving the center $(c_x, c_y) = (\overline{x}, \overline{y})$ in pixel coordinates.

The arrowhead, (a_x, a_y) of the orange arrow is found from the orange segmentation of the image, as seen in Figures 4.3c and 4.4c. The original segmentation is inverted, so that the circle and the arrowhead becomes black and the rest becomes white. This is done so that the same approach can be used when looking at the inner corners of the 'H' in the third and final higher level feature extraction method. In case several corners are found, the corner with an average value closest to the ideal value for a right-angled corner, 191, is chosen. When both the center and the arrowhead is found, the distance between those two points,

$$\texttt{len_arrow_px} = \sqrt{(a_x - c_x)^2 + (a_y - c_y)^2}, \tag{4.6}$$

is used to calculate the radius, r, using the known relation between these two lengths. The real distance between the center and the arrowhead is D_ARROW=300 mm, and the real radius of the landing platform is D_RADIUS=390 mm. Thus the radius, r, is calculated by:

$$r = \frac{\texttt{len_arrow_px} \cdot \texttt{D_RADIUS}}{\texttt{D_ARROW}} \tag{4.7}$$

Finally, the vector from the center to the arrowhead, $v_{ca} = [a_x - c_x, a_y - c_y]$ can be used to calculate the yaw orientation. Zero yaw is defined as when the arrow points to the right in the image, i.e. when it has the same direction as the reference vector $v_r = [a_x - c_x, a_y - c_y]$. The yaw angle is therefore found as the angle between those two vectors [49]:

$$\psi = \text{arctan2}(x1 \cdot y2 - y1 \cdot x2, x1 \cdot x2 + y1 \cdot y2), \tag{4.8}$$

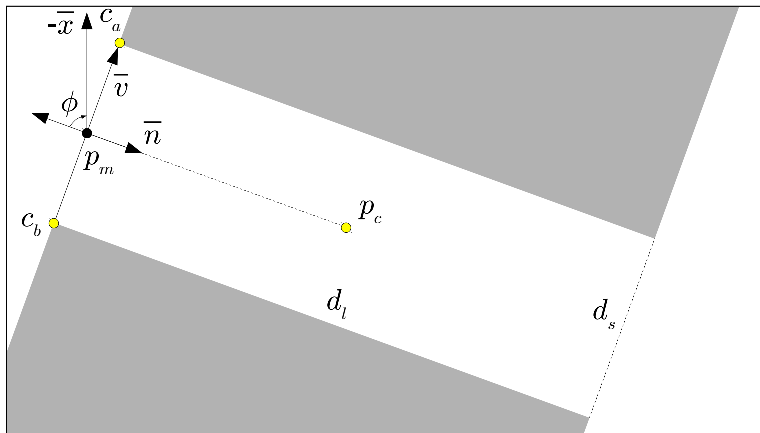
where $x1 = v_{ca}[0]$, $x2 = v_r[0]$, $y1 = v_{ca}[1]$ and $y2 = v_r[1]$.

The *corners* method

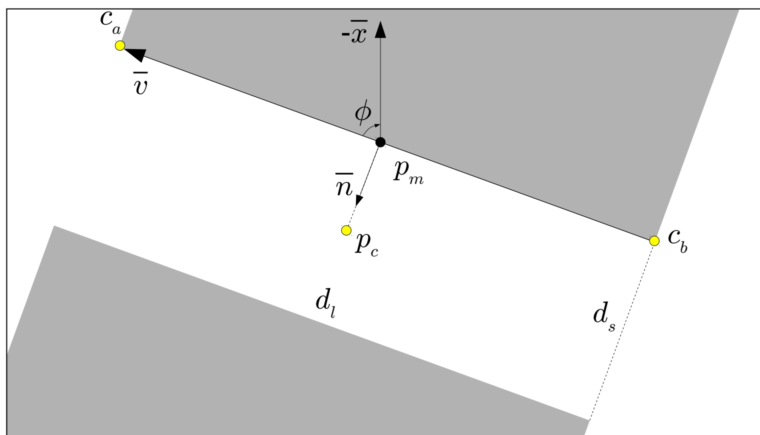
When the quadcopter is close to the landing platform, there are very few features left to navigate by. However, those that are left are sufficient to infer both the quadcopter's position and rotation. An image taken close to the landing platform will contain mostly green and secondly some part of, or the entire, white 'H'. The most distinct features of this letter are the twelve corners on the 'H', classified in this project as eight outer corners and four inner corners. There are two outer corners on each leg and in total four inner corners on the cross-bar of the letter. The four inner corners are found and distinguished from the others using the corner detection method in Section 4.1.3.

After the inner corners are found, they can be used to find the center of the landing platform, $p_c = (c_x, c_y)$ and the radius of the landing platform, r, in pixels. Also, the yaw orientation, ψ, can be found. The number of corners visible in each image may vary. When the quadcopter is far away, all four corners will be visible, but when the quadcopter is closer, some corners may be outside the field of view. The following algorithm is able to extract the necessary information as long as at least two corners are visible.

Before the algorithm starts, c_a is chosen as the uppermost corner and c_b is chosen as the corner closest to c_a. Also, there will be needed a distinction between whether a certain pixel is black or white. For this task, a threshold limit is set to be 75 % of the maximal intensity, 255: I_LIM=191. A point with an intensity above this value will be a white or almost white point. Furthermore, the real lengths of the long and short sides of the cross-bar in the 'H' are stored as D_LONG=120 mm and D_SHORT=40 mm. Since both these real lengths are known, in addition to D_RADIUS, the relative lengths between them are known. Thus, finding only one of them is sufficient to calculate the others. Figure 4.9



(a) Case when the corners are on the short side



(b) Case when the corners are on the long side

Figure 4.9: Geometric notation for the *corners* method

show the notation used. The algorithm is presented next:

Algorithm 1: Evaluating inner corners

Data: Corners c_a and c_b
Result: p_c, r, ϕ
$\vec{v} \longleftarrow$ vector from c_a to c_b
$\vec{x} \longleftarrow$ x-axis of the image (downwards)
$p_m \longleftarrow$ point in the middle of c_a and c_b
$i_{p_m} \longleftarrow$ average intensity around the point p_m
$\vec{n} \longleftarrow$ the normal to $\vec{v}$
if $i_{p_m} < \mathit{I_LIM}$ **then**
    `// Long side`
    $d_s \longleftarrow ||\vec{v}||$
    $d_l \longleftarrow d_s \cdot \texttt{D_LONG/D_SHORT}$
    $p_c \longleftarrow p_m + \frac{\vec{v}}{||\vec{v}||} \cdot (d_l/2)$
    $\phi \longleftarrow$ angle from $-\vec{n}$ to $-\vec{x}$
else
    `// Short side`
    $d_l \longleftarrow ||\vec{v}||$
    $d_s \longleftarrow d_l \cdot \texttt{D_SHORT/D_LONG}$
    $p_c \longleftarrow p_m + \frac{\vec{v}}{||\vec{v}||} \cdot (d_s/2)$
    $\phi \longleftarrow$ angle from $\vec{v}$ to $-\vec{x}$
end
$r \longleftarrow d_s \cdot \texttt{D_RADIUS/D_SHORT}$

The normal, $\vec{n}$ to the vector $\vec{v}$ can have two orientations. The correct orientation, pointing towards the center, is found by considering the pixels around the corners c_a and c_b.

4.1.5 Choosing which method to use

In one image, there might be more than one feature extraction method that is able to find the center and the radius. However, there can only be one estimate output from the computer vision module. The approach chosen in this thesis is to give the methods a priority and choose the available method with the highest priority. The *corners* method is given the highest priority, the *arrow* method the second highest priority and the *ellipse* method the lowest priority. In addition, the *corners* method is only used if the green area is touching the border of the image. In most cases, this means that the quadcopter is close to the landing platform and ensures that the method is only used for low altitudes. Similarly, the *ellipse* method is only used if the green area is not touching the border, i.e. if the entire green area is visible in the image or no green is visible at all.

4.1.6 Calculating position from high level features

The next step is to use the features found in the previous section to calculate the vector from the origin of the camera coordinate system to the landing platform coordinate system w.r.t the landing platform coordinate system, $d_c^l = (d_{c,x}^l, d_{c,y}^l, d_{c,z}^l)$ (see Figure 4.10).

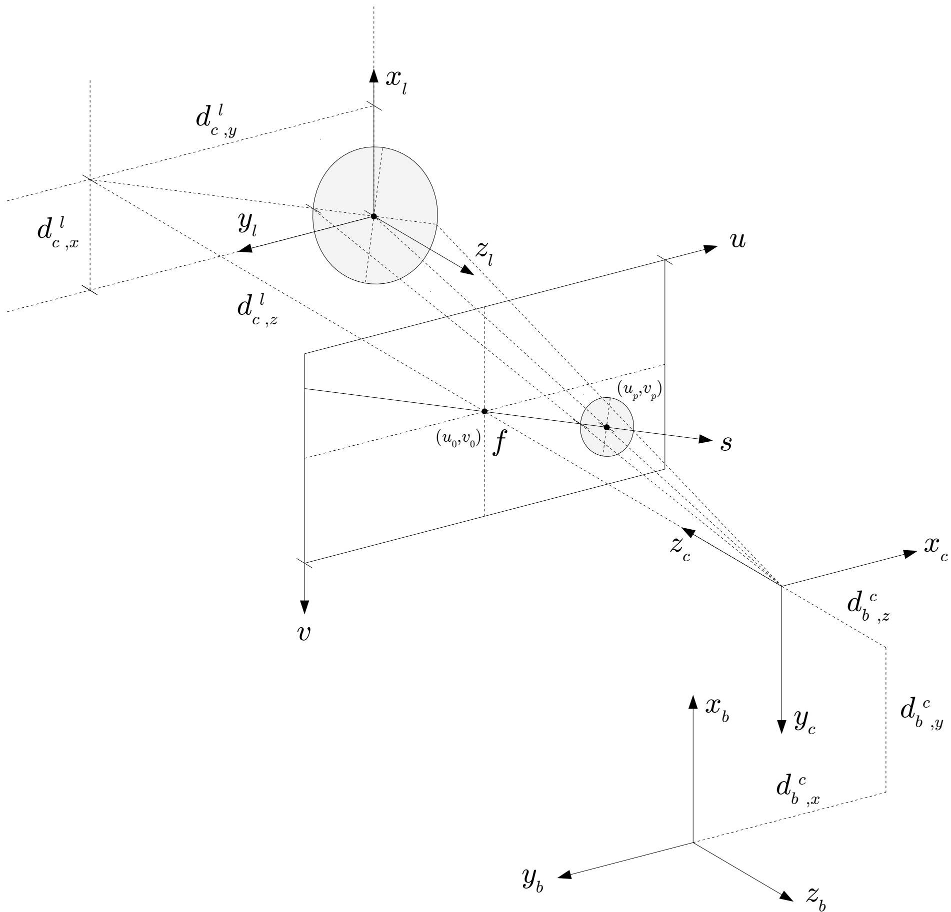


Figure 4.10: The pinhole camera model showing the projection of the landing platform onto the image plane and to the origin of the camera.

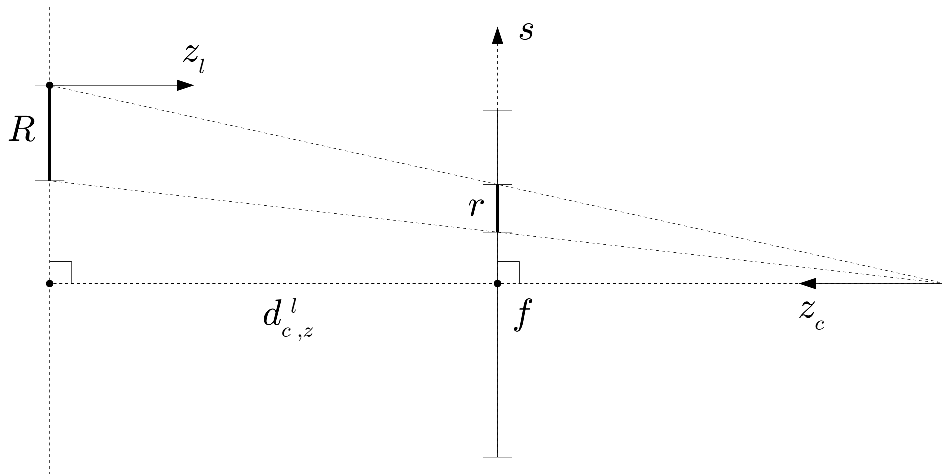


Figure 4.11: The geometry of the pinhole camera model seen directly onto the s vector, which goes through both the center of the image, (u_0, v_0), and the center of the landing platform in the image, (u_p, v_p).

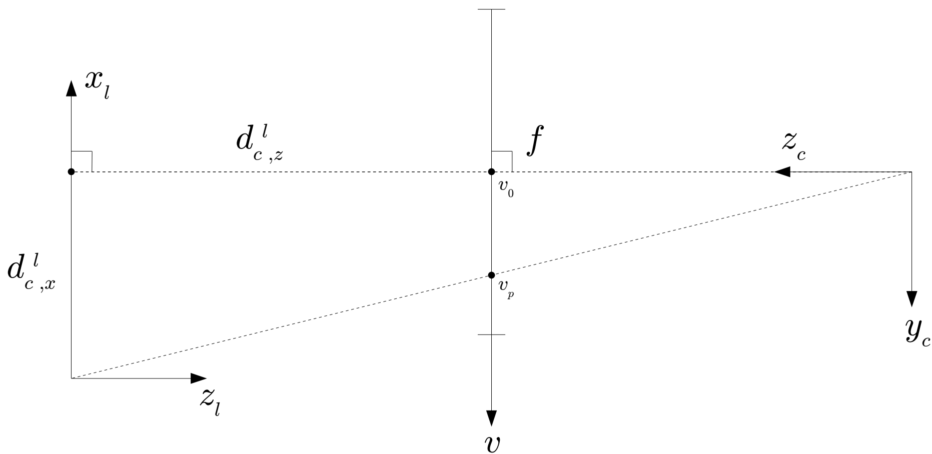


Figure 4.12: The geometry of the pinhole camera model seen from the side, parallel to the x-axis of the landing platform.

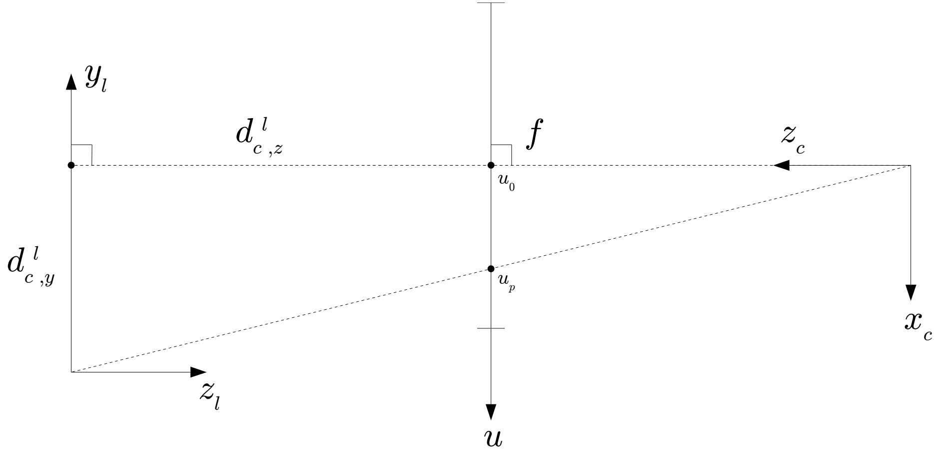


Figure 4.13: The geometry of the pinhole camera model seen from the side, parallel to the y-axis of the landing platform.

First, the z-component of the vector, $d^l_{c,z}$ is found, exploiting the assumption that the two xy-planes are parallel. It is a bit hard to see from the figure, but $d^l_{c,z}$ is measured as the distance between the two xy-planes. The information available from the computer vision system is a predicted length of the radius of the landing platform, r, in pixels. The other known information is the length of the real radius, R in meters and the focal depth, f, in pixels. Figure 4.11 shows the coordinate systems in Figure 4.10 seen down on the s vector from the center of the image frame towards the center of the landing platform in the image. The concept of similar triangles is then used to find the z-component in meters:

$$d^l_{c,z} = \frac{Rf}{r} \quad . \tag{4.9}$$

Next, the x- and y-components are found in a similar fashion. The figures 4.12 and 4.13 show the coordinate systems seen from the side and from above, respectively. The additional information from the computer vision system is the center of the landing platform (u_p, v_p), measured in pixels. The known information is the center of the image, (u_0, v_0), measured in pixels and the length, $d^l_{c,z}$, from the above calculations. Thus the x-components is found as

$$d^l_{c,x} = \frac{(v_p - v_0)R}{f} \tag{4.10}$$

and similarly, the y component is found as

$$d^l_{c,y} = \frac{(u_p - u_0)R}{f}. \tag{4.11}$$

The final step is to calculate the homogeneous transformation H^l_b from the landing plat-

form coordinate system to the body coordinate system by finding

$$\begin{aligned} H_b^l &= H_c^l H_b^c \\ &= \begin{bmatrix} R_b^l & d_b^l \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_c^l & d_c^l \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_b^c & d_b^c \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_c^l R_b^c & R_c^l d_b^c + d_c^l \\ 0 & 1 \end{bmatrix} \end{aligned} \tag{4.12}$$

with

$$R_b^l = R_c^l R_b^c = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{4.13}$$

and

$$d_b^l = R_c^l d_b^c + d_c^l, \tag{4.14}$$

where R_c^l is given in Equation 4.13, the components of vector d_c^l is given in Equations 4.9, 4.10 and 4.11 and d_b^c is the vector from the camera to the body coordinate system as seen in Figure 4.10. Thus, the position of the quadcopter relative to the landing platform is given by the vector d_b^l.

4.2 Filter

After one of the estimates is selected, it is passed onto a filter, which uses a combination of a moving median filter and a moving average filter. The moving median filter on one hand is good at suppressing impulse noise[50], which will occur due to sudden large errors in the estimate, for instance caused by false detections of the landing platform. The moving average filter on the other hand can act as a low-pass filter [51]. A low-pass filter is good at reducing high-frequency noise in the measurements, which will occur due to inaccuracies in the estimate. The filter is defined in two steps, where the output of the median filter m_i is used as the input for the average filter y_i:

$$\begin{aligned} m_i &= \texttt{median}(x_i, x_{i-1}, \ldots, x_{i-M+1}) \\ y_i &= \frac{1}{N} \sum_{j=0}^{N-1} m_{i-j}, \end{aligned} \tag{4.15}$$

where M is the moving window size of the median filter and N is the moving window size of the average filter. Both sizes are set to $M = N = 3$.

4.3 Dead reckoning

The position estimate is not always available. Factors such as noise in the image, changes in lightning conditions and a very small or very large distance between the quadcopter and the landing platform affects the estimate negatively and may lead to no estimate. The landing platform may not even be visible in the image, for instance at times when the quadcopter is performing a mission some distance away from the ship. Furthermore, there might be hidden faults in the software corrupting the estimate. When no position estimate is available, for whatever reason, the quadcopter is blind and will have a hard time finding back to the ship, unless it has some other means of navigating.

To cope with this problem, a method called dead reckoning is applied. This method calculates the current position by integrating measurements of velocity and acceleration and adding this to the last known position [52]. The quadcopter measures its linear velocity, v_t, linear acceleration, a_t, and yaw, ψ, using the Inertial Measurement Unit (IMU), which is constantly published on the topic */ardrone/navdata*.

The position is found by simply using the formulas for equations of motion with constant acceleration:

$$p_n = p_{n-1} + (t_n - p_{n-1})v_n + \frac{1}{2}(t_n - t_{n-1})^2 a_n, \tag{4.16}$$

where p_{n-1} is the last known position at time t_{n-1}, v_n and a_n are measurements of the velocity and acceleration at time t_n and p_n is the new estimated position. Similar can be done for the yaw:

$$\psi_n = \psi_{n-1} + \Delta\psi. \tag{4.17}$$

For the yaw, the angle is calculated directly and no angular velocities are available. This is solved by calculating the change between the current IMU estimate, ψ_{CURR}, and the previous one, ψ_{PREV} to get $\Delta\psi = \psi_{\text{CURR}} - \psi_{\text{PREV}}$. Since the yaw is measured in range $(-180, 180]$, the following short algorithm is used to ensure the correct $\Delta\psi$ when passing these limits:

Algorithm 2: Correction for $\Delta\psi$

```
if Δψ > 180 then
 |  Δψ -= 360
else if Δψ < −180 then
 |  Δψ += 360
```

When implementing these equations and testing it in the simulator, there were some errors that needed to be corrected. First of all, the sensor measurements does not have a mean around zero, meaning that when the quadcopter is standing still, with no velocity or acceleration, the sensors still report slight movement inn all axes. This is especially apparent with the acceleration along the z-axis. Due to the acceleration of gravity, a standstill accelerometer measures an acceleration of 9.81 m/s^2 along the z-axis. To solve this problem, a calibration phase of 10 seconds is conducted before the dead reckoning starts. The average of the measurements in the calibration phase is stored in a variable, and for every measurement after the calibration phase, this value is subtracted from the measurement. This results in a more stable dead reckoning calculation that to a smaller extent drifts from its actual value.

Secondly, it is observed that when the quadcopter is standing still, even after the calibration, there is still some small fluctuations in the measurements. This is unwanted, because it can contribute to drift in the dead reckoning even though the quadcopter is standing still. To cope with this problem, the signals are observed for a period when the quadcopter is standing still and the largest fluctuations are noted. Then, in the dead reckoning implementation, all measurements smaller than this value are set to zero. This results in less drift in the dead reckoning calculation when the quadcopter is standing still.

Finally, the rotation of the quadcopter is taken into account. The linear velocities and accelerations are measured in the quadcopter's body coordinate system, and when the quadcopter rotates, the x-axes of the landing platform coordinate system will no longer be aligned with the x-axes of the quadcopter. This will have implications when the quadcopter rotates around the z-axis while not directly above the landing platform. No linear velocities or accelerations will be registered in x-, y-, or z-direction, so if the rotation is not taken into account, the dead reckoning will output the same position. However, since the quadcopter has rotated, the change in the quadcopter's position has to be rotated around the z-axis, in the opposite way of the rotation ψ, before it is added to the previously known position. This can be generalized to

$$p_b = R_{z,-\psi} p_a, \tag{4.18}$$

where p_a is the position before the correction and p_b is the correct position when the quadcopter has rotated ϕ degrees around the z-axis and

$$R_{z,-\psi} = \begin{bmatrix} \cos -\psi & -\sin -\psi & 0 \\ \sin -\psi & \cos -\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos \psi & \sin \psi & 0 \\ -\sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.19}$$

When this is done, the dead reckoning output is published at topic */estimate/dead_reckoning*. Test results for the dead reckoning is shown in Section 4.3.

4.4 User interface

Even though the future goal of this system is to be autonomous, for testing and safety purposes an operator has to be able to interfere with the system. Therefore a simple user interface is created, where the operator gets control access to the quadcopter through the handheld controller presented in section 3.4.

The handheld controller is connected to the computer through two nodes in ROS. The first node is called *joy_node* and is available from the open source *joy* package [53]. This node takes signals from any joystick connected to the computer, in this case the PlayStation controller, and publish a message on the */joy* topic. The message contains the state of the joystick's buttons and axes. The second node is developed as a part of this thesis. It subscribes to the */joy* topic from the first node and translates signals about joystick movements and button presses to the rest of the system. The handheld controller can be set to two modes: manual control or set point control. In manual control, the movements of the joysticks are translated directly to control signals for the quadcopter and published as `Twist()` messages on topic */cmd_vel*.

The control for take off and landing is available in both modes. A take off or landing command is sent to the quadcopter by publishing an `Empty()` message on the topics */ardrone/takeoff* and */ardrone/land*, respectively. The mapping between the controller and the control signals to the quadcopter is the following:

- Left joystick up/down: Linear speed in z direction
- Left joystick left/right: Angular speed around z axis
- Right joystick up/down: Linear speed x direction
- Right joystick left/right: Linear speed y direction
- Cross button: Take off
- Circle button: Land
- R1: Emergency stop
- L1: Toggle between manual control and set point control
- Start: Initiate automated landing

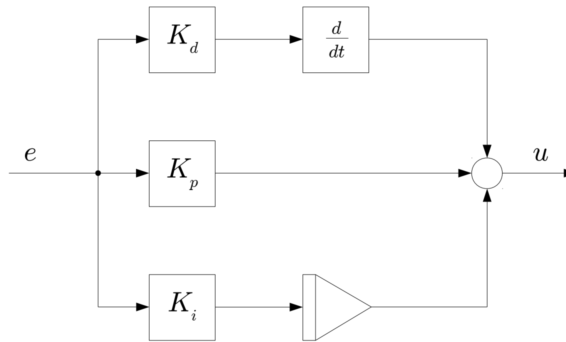


Figure 4.14: Block diagram of a typical PID controller. The controller has a proportional term (in the middle), a integral term (at the bottom) and a derivative term (at the top). Modified from [54].

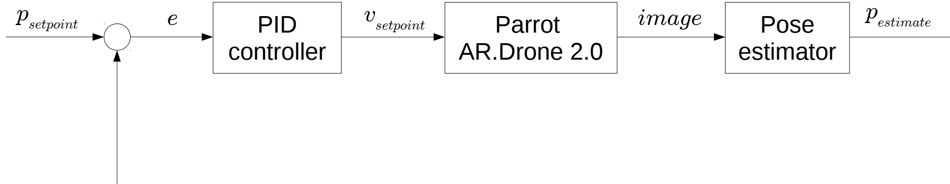


Figure 4.15: Control loop of the system.

4.5 PID controller

A simple PID controller is implemented to control the quadcopter to a given set point. Such controllers are very simple and are covered extensively in the literature [54]. Figure 4.14 shows the structure of the PID regulator and Figure 4.15 shows the PID controller in the closed control loop with feedback from the pose estimator.

Since the quadcopter has an onboard autopilot that handles the low-level control of the motors and the stabilization of the roll and pitch angles, the set points for the PID controller can be position in x-, y-, z-direction and orientation in yaw. Under the circumstances of this thesis, control of one of these variables can be done independently of the others, so there is implemented one controller for each variable. Each PID controller has three parameters, the proportional gain, the integral gain and the derivative gain. These parameters are tuned individually for each variable by trial and error until the step response of the quadcopter is reasonably quick without overshooting. Due to the latency introduced by the pose estimate, and to help maintain the assumption of a horizontal quadcopter, it is best to have a slow and steady control response. The resulting K_p, K_i and K_d parameters and are listed in Table 4.2.

Variable	Kp	Ki	Kd
x, y, z	0.7	0.001	0.5
yaw	0.01	0.0	0.0

Table 4.2: PID parameters

4.6 Automated landing

The automated landing node is implemented as a sequential planner, where a sequence of set points are published to the PID controller. The automated landing can be initiated by pressing the "Start"-button on the handheld controller. The node then generates a trajectory of set points from the current position of the quadcopter, via the way point (0.0, 0.0, 2.0), 2 m above the landing platform, and to the way point (0.0, 0.0, 0.3), 30 cm above the landing platform. The set point for the yaw is set to zero for the entire trajectory. A desired speed of 0.4 m/s is set and the node publishes intermediate set points towards the main way point at a frequency of 10 Hz. When the current intermediate set points is closer than 1 meter to the main way point, the speed is reduced proportionally to how close the main way point is. When the quadcopter is closer than 0.01 m to the current way point, the next way point is chosen. After the trajectory is executed, the node publishes the land command to the quadcopter and it lands.

The reason for naming this module a planner for automated landing instead of autonomous landing, is that autonomous landing has higher requirements this simple sequential planner fulfills. For instance, it needs to make decisions during the landing process whether it is safe to land or whether the landing should be aborted.

4.7 DDPG controller

The DDPG controller developed in [9] is also mention here for completion. All this node requires is a position and yaw estimate from the perception system above, together with the IMU measurements from the quadcopter. It will then control the quadcopter from its current position to a set point straight above the landing platform. When the quadcopter is sufficiently close to this set point the controller will start to perform a descend towards the landing platform. When the quadcopter again is sufficiently close, it will land.

4.8 Connection to the quadcopter

The ROS application can either be connected to a simulated quadcopter or a physical quadcopter. When connecting to the simulator, the */tum_simulator* package is used to spawn a simulated quadcopter in the Gazebo simulator. The rest of the ROS application can then communicate with the simulated quadcopter over topics. When connecting to the physical quadcopter, the battery is first inserted into the quadcopter, which will then power itself on and perform an initial calibration. After the calibration is done, it beeps and spins all four propellers a fraction of a revolution. It is now ready to be connected to the computer over WiFi. On the computer it will be visible as a normal network. When

a WiFi connection is established, the *ardrone_driver* node from the *ardrone_autonomy* is used to connect the quadcopter to the rest of the ROS application.

4.9 Running the system

The different nodes of the system can be started individually with the `rosrun` command or a *launch file* can be written to start several nodes collectively with the `roslaunch` command. The commands for starting each node, or cluster of nodes, are given below:
Start the simulator and spawn a simulated quadcopter:

```
$ roslaunch uav_vision sim_ar2.launch
```

Connecting to the physical quadcopter:

```
$ roslaunch uav_vision real_ar2.launch
$ rosservice call /ardrone/setcamchannel 1
```

Start the entire perception system:

```
$ roslaunch uav_vision perception_system.launch
```

Start the joystick system:

```
$ roslaunch uav_vision joystick.launch
```

Start the PID controller:

```
$ rosrun uav_vision pid_controller.py
```

Start the planner for automated landing:

```
$ rosrun uav_vision automated_landing.py
```

Start the DDPG controller:

```
$ rosrun ddpg ddpg_hover_descend.py
```

View the camera stream from the bottom camera, using the *image_view* node from the *image_view* package:

```
$ rosrun image_view image_view \
image:=/ardrone/bottom/image_raw
```

Chapter 5

Results

5.1 Experiments in the simulator

The first part of the experiments is performed in the Gazebo simulator. All three computer vision methods are tested up against the ground truth for different altitudes above the landing platform. The estimates for all variables x, y, z and yaw are assessed, however most emphasize is put on the position estimates as these are the most important values to get correct in order to land safely. For brevity, the methods are referred to as the *ellipse* method, the *arrow* method and the *corners* method. Furthermore, the running median and mean filter applied to the estimate is tested. Finally, the dead reckoning module is tested. There are not implemented any wind disturbances in the simulator so the onboard autopilot is able to stabilize the quadcopter fairly well itself. Nevertheless, the quadcopter tends to slightly drift in one direction. Therefore, during the experiments, the quadcopter is controlled with the simple PID controller and the joystick set in set point mode, which makes it easy to hover in, and fly to, the desired positions for testing.

5.1.1 Assessment of the computer vision system

The computer vision system is assessed while manually flying the quadcopter above the landing platform. It is noted that the estimate is published with a rate between 5-10 Hz although it is set to be 10 Hz. Some of the features found with the different methods are marked in the image, and the marked image is published to topic */processed_image* and stored with even time steps. The marked image can be viewed with the package *image_view* available from ROS:

```
$ rosrun image_view image_view image:=/processed_image
```

Drawing on, publishing and storing images takes extra computational power and time, so this function can be turned off when not needed. Figure 5.1 shows some samples from these images. The features found with the *corners* method are marked in yellow and consist of the two inner corners used, the center of the landing platform and the direction

of the x-axis of the coordinate system of the landing platform. The features found with the *arrow* method are marked in red and consist of the arrowhead and the center of the landing platform. The feature found with the *ellipse* method is the center of the landing platform and this is marked in blue.

Figure 5.1a shows an image where all methods give an estimate. Figures 5.1b, 5.1c and 5.1d show clearly that the *corners* method can work with 4, 3 or 2 visible inner corners. Figures 5.1e and 5.1f show examples of two methods available at the same time. Figures 5.1g and 5.1h show the *ellipse estimate* working for middle to high altitudes.

Discussion

For all methods the radius is also found as a feature, but this is not marked in the image here. It would have been possible to use the center and the radius together to draw a bounding box or even a bounding circle around the detected landing platform. Nevertheless, the correctly detected center indicates which method is working, and the accurate estimates presented in the following sections show that the radius is also found correctly.

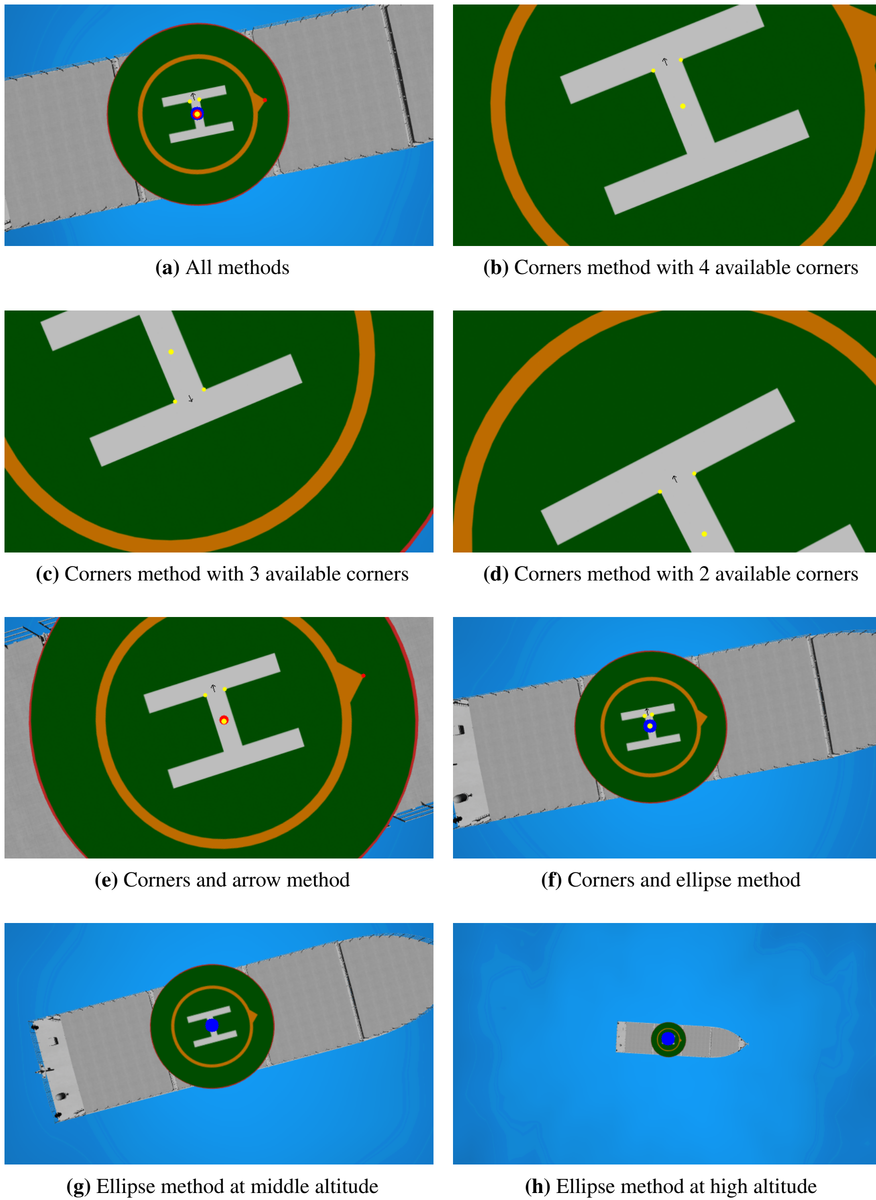
The publishing rate of the computer vision system is a bit lower than the 10 hz intended. This rate was chosen as a reasonable frequent estimate update for the controller to be able to stabilize the quadcopter for low velocity flight. The lower actual rate indicates that either the computer vision methods are too complex or that the implementation should be optimized for better running time.

5.1.2 Test of all methods when flying up and down

Next, the quadcopter is flown from position (-0.1, 0.0, 0.2) up to (-0.1, 0.0, 5.0) and back again. The reason why the x-position is set to -0.1, is that it is the position where the 'H' is best aligned in the center of the camera, and thus gives the most stable estimate for lower altitudes. Figure 5.2 shows the resulting estimates in the left column and the corresponding estimate errors in the right column. The ground truth z-position is shown in grey color in the background. The overlapping ranges of the methods can be observed, where the *corners* method has a range from around 0.2-1.0 meters, the *arrow* from 0.5-1.2 meters and the ellipse method from around 1.0 meters and upwards.

Discussion

The z-position plots on the third row show some outliers from the *arrow* method around 2-2.5 meters and above 4.5 meters. This is mainly caused because the arrowhead is detected too close to the center of the landing platform. Around 2-2.5 meters, it is detected on the center of the arrow instead of at the tip of the arrow (see Figure 5.3a), giving a shorter radius, but the right angle, as observed in the yaw-estimates in the fourth row. Above 4.5 meters however, the angle is also wrong, which indicates that points along the orange circle is detected as the arrowhead (see Figure 5.3b), thus giving a too small radius and a wrong angle. Since the arrow on the orange circle caused that much problems, it might have been better to not include the arrow and just pick any point at the circle to calculate the radius. This would however mean that the yaw estimate would not be available from the *arrow* method, since the direction of the landing platform would be hidden.



(a) All methods

(b) Corners method with 4 available corners

(c) Corners method with 3 available corners

(d) Corners method with 2 available corners

(e) Corners and arrow method

(f) Corners and ellipse method

(g) Ellipse method at middle altitude

(h) Ellipse method at high altitude

Figure 5.1: Examples of feature detection on the landing platform

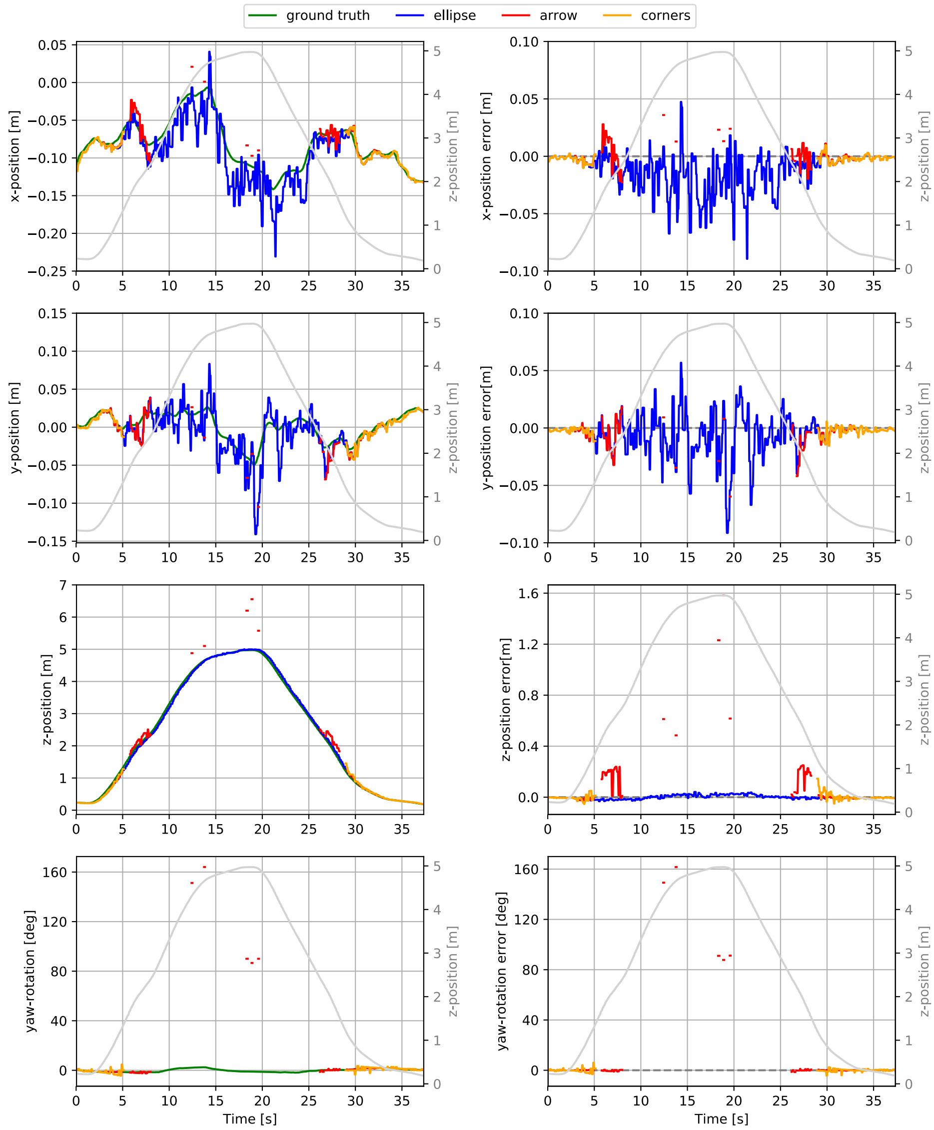
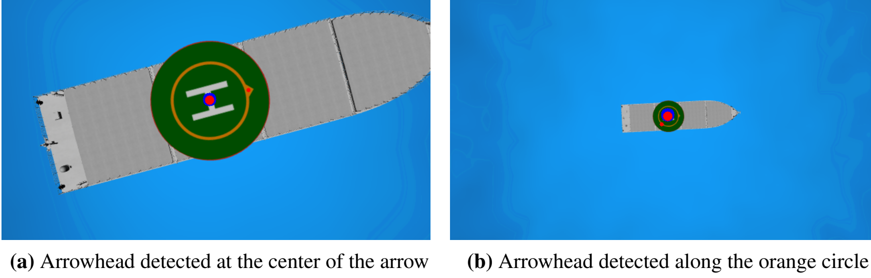


Figure 5.2: All variable's values and errors when flying from a hovering position at 0.2 meter above the landing platform, up to 5 meters above the landing platform and down again.



(a) Arrowhead detected at the center of the arrow (b) Arrowhead detected along the orange circle

Figure 5.3: Examples where the arrowhead is wrongly detected.

5.1.3 Test of all methods when hovering

For testing the methods at different altitudes, the quadcopter is set to hover for 15 seconds at positions $(0, 0, z)$ for $z \in [0.5, 1.0, 2.0, 3.0, 5.0, 10.0]$ meters. The estimate errors for the x,y and z-position and the yaw-rotation are shown in Figures 5.4, 5.5, 5.6 and 5.7, respectively. Here, the different ranges of the methods are apparent, where the *corners* method works for low altitudes, the *arrow* method for middle altitudes and the *ellipse* method for high altitudes. Also, the outliers from the *arrow* method discussed in the previous experiment are visible. The z-position estimate is affected by these outliers at an altitude above 2 meter, while the yaw estimate only is affected at an altitude above 5 meter. Furthermore, since the ellipse method does not give a yaw-estimate, there are no good yaw-estimates above 2 meter.

Furthermore, the accuracy and precision of all methods and all variables are studied for each altitude during the hovering. The accuracy is represented by the mean error

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i, \tag{5.1}$$

where x_i is the i-th error between the estimate and the ground truth in the time series of N data points. The errors are sampled at a rate of 20Hz for around 15 seconds and $N = 300$. Furthermore, the precision is represented by the standard deviation

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}. \tag{5.2}$$

For both the mean error and the standard deviation, only data points that contains an estimate are considered, i.e. data points where the method is unavailable are not considered. The availability for an estimate during the time series are found by counting the data points available and dividing by N. The mean error, standard deviation and the availability for each method are presented in the Tables 5.1, 5.2, 5.3, 5.4 along with the minimum and maximum estimate error during the time series. The best method for each variable and altitude, considering both availability, mean and standard deviation, is marked in green.

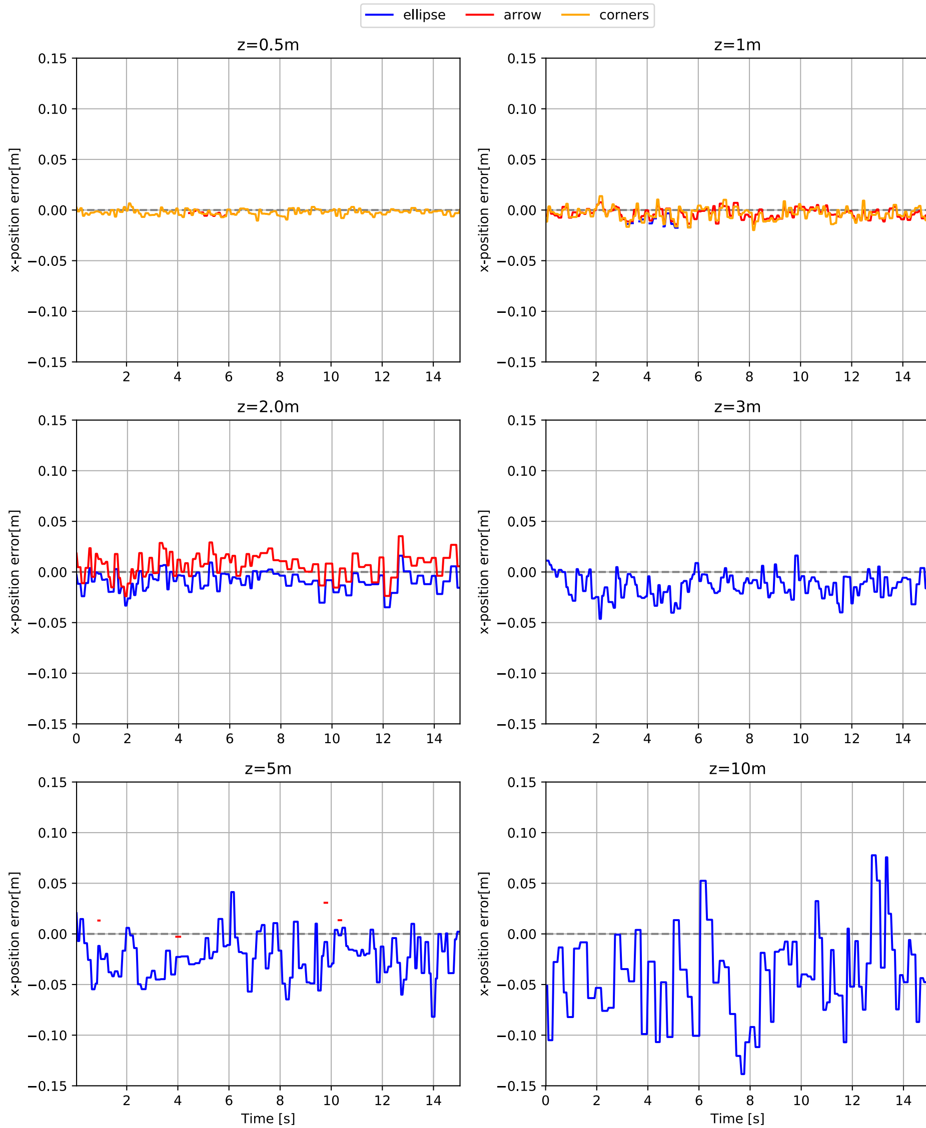


Figure 5.4: Estimate error for x-position from all methods at different altitudes.

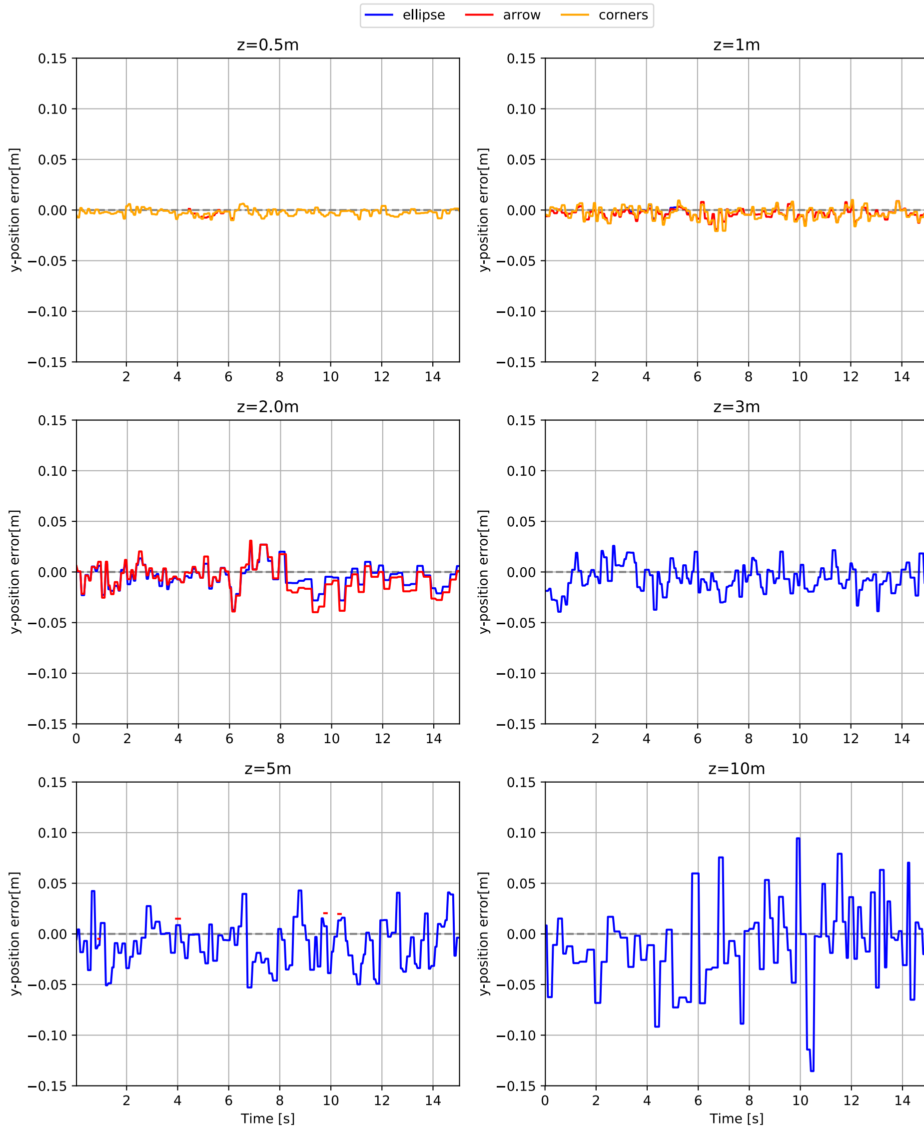


Figure 5.5: Estimate error for y-position from all methods at different altitudes.

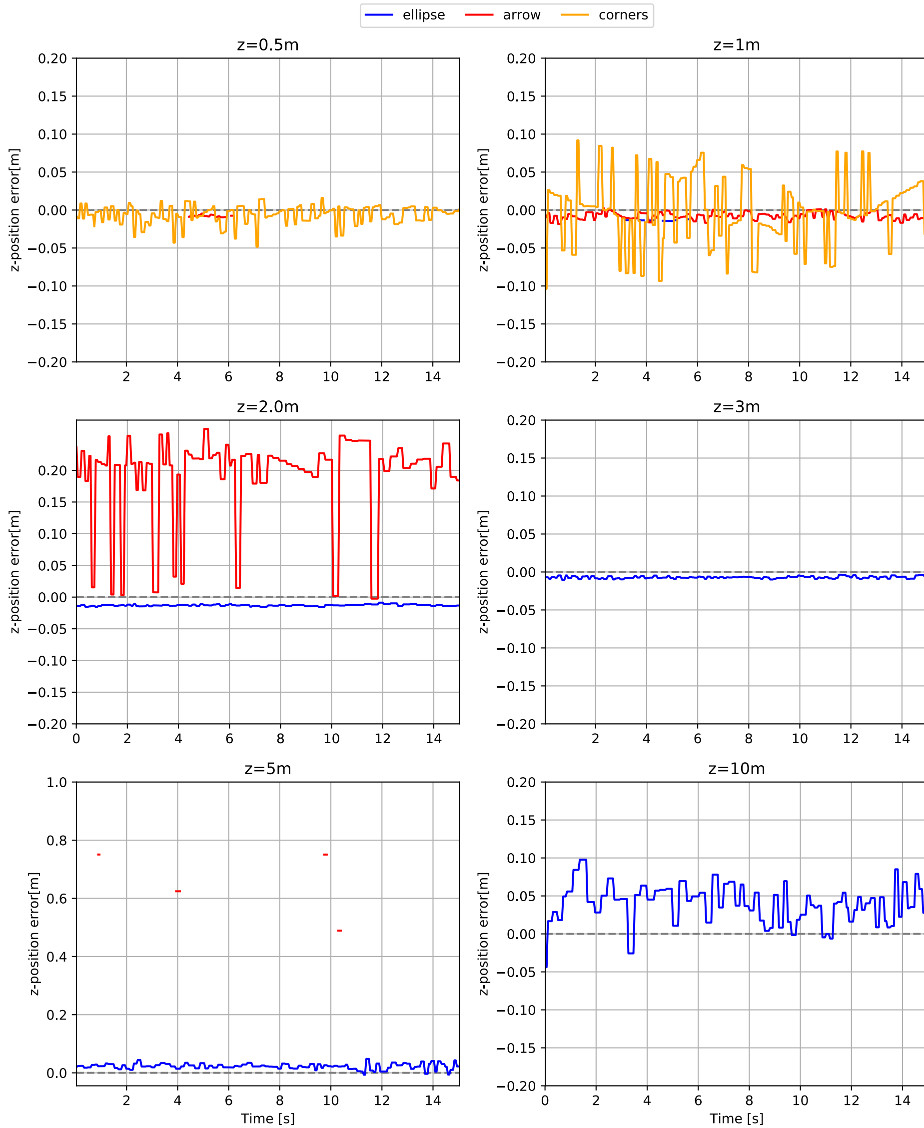


Figure 5.6: Estimate error for z-position from all methods at different altitudes.

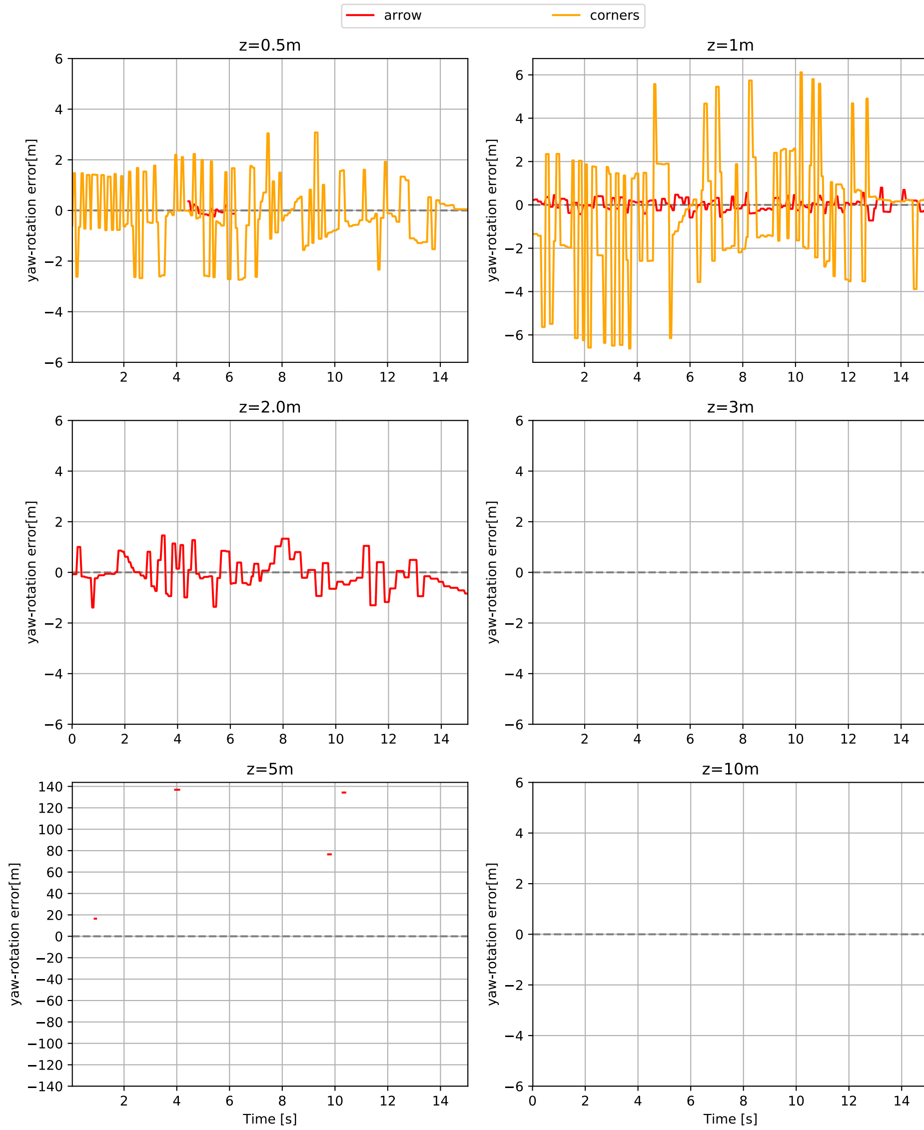


Figure 5.7: Estimate error for yaw-position from all methods at different altitudes, except from the *ellipse* method which does not give an estimate for the yaw-rotation.

Height	Method	Min [mm]	Max [mm]	Mean [mm]	Std [mm]	Avbl.
0.5m	ellipse	-	-	-	-	0%
	arrow	-6.62	0.34	-3.57	1.91	11%
	corners	-10.44	6.61	-2.15	2.85	100%
1.0m	ellipse	-17.46	-3.24	-12.01	4.31	11%
	arrow	-14.04	7.66	-3.41	5.20	100%
	corners	-19.79	13.75	-3.72	6.24	100%
2.0m	ellipse	-34.88	16.10	-9.17	9.54	100%
	arrow	-24.00	35.34	6.55	11.95	100%
	corners	-	-	-	-	0%
3.0m	ellipse	-46.30	16.24	-13.48	12.01	100%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%
5.0m	ellipse	-81.84	41.31	-22.41	21.33	100%
	arrow	-2.79	30.75	12.34	12.71	4%
	corners	-	-	-	-	0%
10.0m	ellipse	-138.44	77.64	-42.30	42.74	100%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%

Table 5.1: Accuracy (mean) and precision (std) of the different methods for x-position estimate.

Height	Method	Min [mm]	Max [mm]	Mean [mm]	Std [mm]	Avbl.
0.5m	ellipse	-	-	-	-	0%
	arrow	-8.21	1.15	-4.45	2.76	11%
	corners	-10.00	6.26	-2.30	3.22	100%
1.0m	ellipse	-9.37	4.62	-1.40	4.16	11%
	arrow	-18.84	8.83	-3.56	4.88	100%
	corners	-20.74	9.97	-3.09	6.20	100%
2.0m	ellipse	-37.47	27.08	-4.91	11.70	100%
	arrow	-39.71	30.93	-7.00	14.29	100%
	corners	-	-	-	-	0%
3.0m	ellipse	-39.29	25.80	-6.50	13.91	1000%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%
5.0m	ellipse	-52.88	42.90	-9.23	22.67	100%
	arrow	-4.87	20.46	14.22	8.84	4%
	corners	-	-	-	-	0%
10.0m	ellipse	-135.52	94.41	-11.64	41.28	100%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%

Table 5.2: Accuracy (mean) and precision (std) of the different methods for y-position estimate.

Height	Method	Min [mm]	Max [mm]	Mean [mm]	Std [mm]	Avbl.
0.5m	ellipse	-	-	-	-	0%
	arrow	-9.56	-4.65	-7.66	1.45	11%
	corners	-48.38	16.06	-6.13	12.03	100%
1.0m	ellipse	-14.46	-12.23	-13.56	0.61	11%
	arrow	-18.42	2.42	-7.65	4.64	100%
	corners	-103.75	91.83	-0.76	43.86	100%
2.0m	ellipse	-15.69	-8.74	-12.96	1.33	100%
	arrow	-2.44	265.20	191.92	69.48	100%
	corners	-	-	-	-	0%
3.0m	ellipse	-10.23	-3.57	-7.08	1.53	100%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%
5.0m	ellipse	-6.66	47.89	22.52	9.79	100%
	arrow	489.13	750.69	643.13	104.21	4%
	corners	-	-	-	-	0%
10.0m	ellipse	-43.81	97.76	40.07	25.45	100%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%

Table 5.3: Accuracy (mean) and precision (std) of the different methods for z-position estimate.

Height	Method	Min [deg]	Max [deg]	Mean [deg]	Std [deg]	Avbl.
0.5m	ellipse	-	-	-	-	0%
	arrow	-0.24	0.36	-0.01	0.16	11%
	corners	-2.74	3.08	-0.12	1.33	100%
1.0m	ellipse	-	-	-	-	0%
	arrow	-0.72	0.80	0.02	0.28	100%
	corners	-6.63	6.12	-0.42	2.83	100%
2.0m	ellipse	-	-	-	-	0%
	arrow	-1.39	1.46	-0.04	0.65	100%
	corners	-	-	-	-	0%
3.0m	ellipse	-	-	-	-	0%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%
5.0m	ellipse	-	-	-	-	0%
	arrow	16.49	136.91	101.09	45.24	4%
	corners	-	-	-	-	0%
10.0m	ellipse	-	-	-	-	0%
	arrow	-	-	-	-	0%
	corners	-	-	-	-	0%

Table 5.4: Accuracy (mean) and precision (std) of the different methods for yaw-rotation estimate.

Discussion

It is a general trend that the deviations from the ground truth increase with the altitude. This is partially explained by the fact that a small tilt of the quadcopter alters the camera's field of view to a larger degree for higher altitudes. Since the tilt is not taken into account when performing the estimate, this effect becomes larger for higher altitudes.

The Tables 5.1, 5.2, 5.3 and 5.4 supports the approach for selecting which method to use in case of more methods available. At most of the altitudes, the *corners* method works best when that is available and if is not, the *arrow* method works best, and the *ellipse* method can be chosen if none of the other two is available.

5.1.4 Test of filter

Even though the mean errors of the estimates are quite accurate, there is still some noise in the estimates. Therefore, in the third experiment, the running mean filter implemented in Section 4.2 is tested with a change in the z-direction. The quadcopter is flown from a hovering position at (0.0, 0.0, 1.0) to a hovering position at (0.0, 0.0, 3.0). The step response is shown in Figure 5.8, with the filtered estimate in grey.

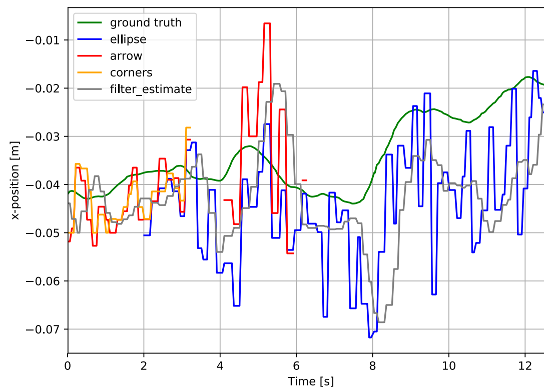
The switching between which method is used can be seen with the x-position estimate in Figure 5.8a. At t=4s, the *ellipse* method is used, but from t=4.1s, the *arrow* method becomes available and is therefore used instead. Since this estimate is higher than the one from the *ellipse* method, the filtered estimate value goes up. When the *arrow* method is no longer available at t=6s, the *ellipse method* is used again.

Discussion

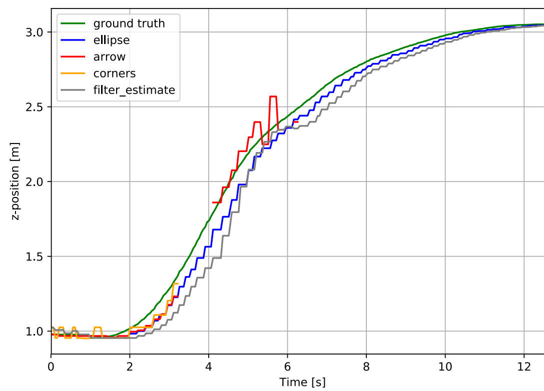
The time delay introduced by the computer vision method and the running mean filter is apparent in Figure 5.8b, where the filtered estimate is following the ground truth value, but not quite able to keep track until it stabilizes at a stationary value. The computer vision module runs at approximately 10 Hz which gives a time delay of 0.1s. The filter has a running window of 5 measurements which then adds another 0.5s time delay. This time delay must be considered when designing a controller.

5.1.5 Test of dead reckoning

Next, the dead reckoning module is tested. Since it is designed to work even tough there is no estimate from the computer vision module, the quadcopter is flown forward, from a hovering position 1 meter directly above the landing platform, around the ship, where the landing platform is not visible. Then it is flown back again to see how much the dead reckoning has drifted. The results are shown in Figure 5.9. The estimate, represented by the filtered estimate in gray, is present in the beginning and the end, but disappears when the quadcopter moves away from the landing platform. The dead reckoning is nevertheless able to calculate the position quite accurately until the quadcopter returns to the landing platform.



(a)



(b)

Figure 5.8: The estimates, filtered estimate and dead reckoning during a change in the z-position.

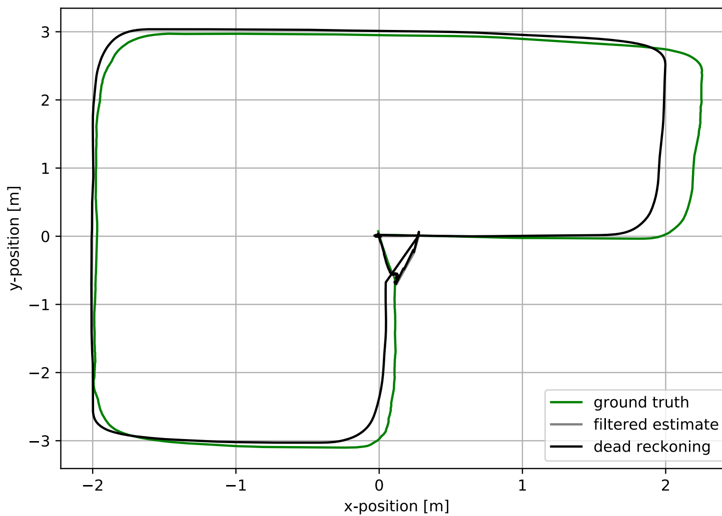


Figure 5.9: Test of the dead reckoning system when flying around the ship.

Discussion

Even though the dead reckoning is able to follow the ground truth very well in this experiment, it does not mean it always will be the case. In the simulator, there are no external disturbances. With the real quadcopter however, wind and other disturbances such as airflows from the ground effect may be present. Since the inertial measurements used for the dead reckoning of the x-, y- and z-position are only velocity and acceleration measurements and not a direct position measurement, these external disturbances may cause the dead reckoning calculation to drift more.

5.1.6 Test of yaw estimate while rotating

The next experiment examines the yaw estimate. The quadcopter is controlled manually to hover around (0.0, 0.0, 1.0) and rotate counterclockwise 360 °. Figure 5.10 shows that the *arrow* and the *corners* methods gives quite accurate estimates of the yaw-position. The *corners* method can only estimate the yaw in the range (-90, 90] and has to assume that the forward direction visible in the image is the correct. Therefore, when reaching 90 °, it steps down to -90 °and continues from there.

Discussion

This way of estimating the yaw means that controlling the yaw to around ± 90 °or ± 180 °becomes a challenge, since a small deviation in the actual yaw will result in a large deviation in the estimated yaw. The control around these angles is not considered in this thesis however, so the yaw is controlled to zero in most of the experiments, where this is not a problem.

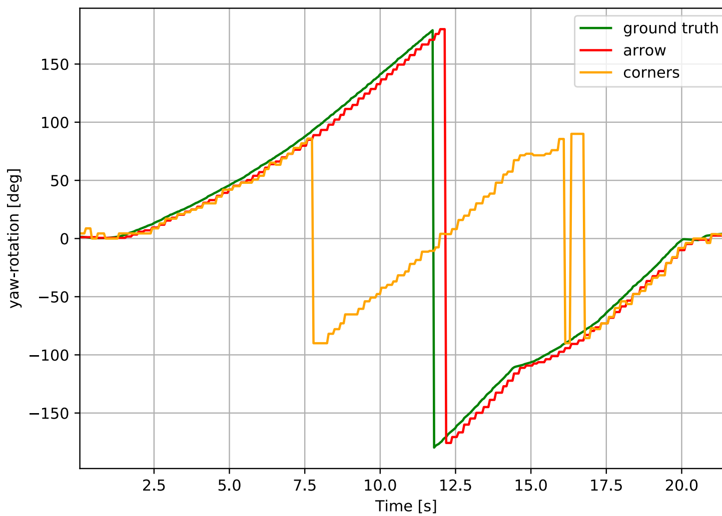


Figure 5.10: The yaw estimate when rotating 360°counterclockwise.

5.1.7 Landing using the PID controller and the automated landing planner

In the next experiment, the sequential planner and the PID controller is used to land the quadcopter. The automated landing is initiated when the quadcopter is at pose $x = 1.0$ m, $y = 2.0$ m, $z = 3.5$ m and $\psi = -90$ °, where the landing platform is only barely visible in the lower right corner of the image. The resulting actual and estimated trajectory is shown in Figure 5.11.

Discussion

These results show that the methods implemented in this thesis can be used to land a quadcopter using only an onboard monocular camera and inertial measurements as feedback to the controller. The feasibility of the estimate is demonstrated here with a simple PID controller, however the estimate works independently of the controller and can therefore be used by other control strategies as well.

5.1.8 Landing using external DDPG controller

In this final experiment in the simulator, the external DDPG controller made by Daniel Tavakoli [9] is tested using the pose estimate from this project as input. As in the previous experiment, the planner and controller is started when the quadcopter is at pose $x = 1.0$ m, $y = 2.0$ m, $z = 3.5$ m and $\psi = -90$ °. The quadcopter fluctuates a bit more than when controlled with the automated planner and the PID controller, but the DDPG controller is able to bring the quadcopter towards the landing platform, then descend and then land. The resulting actual and estimated trajectory is shown in Figure 5.11.

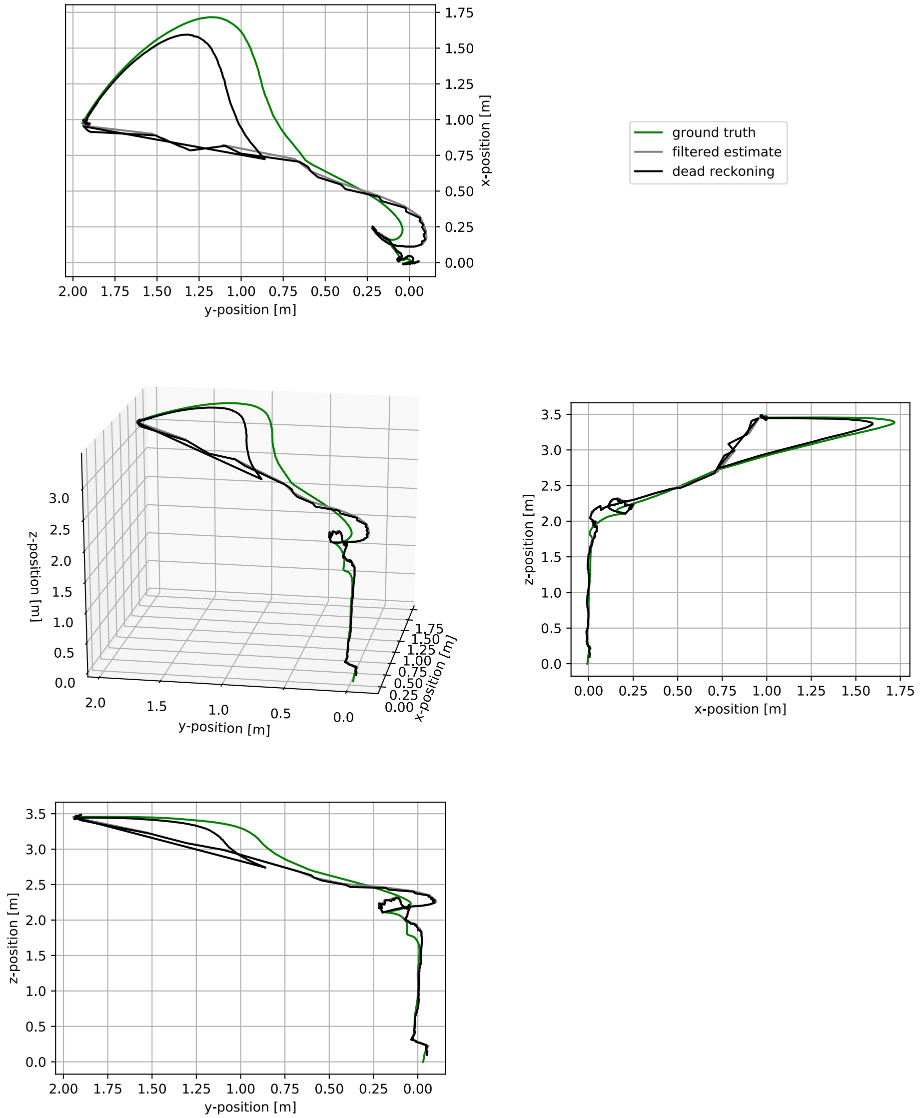


Figure 5.11: The trajectory, seen from all three sides, when landing using the automated landing planner.

Discussion

These results indicates that the pose estimation method developed in this thesis is accurate enough to be used with an external controller that is created knowing nothing about how the pose estimation method works, except that it outputs an estimate at around 10 Hz and that it works best when the yaw is controlled towards zero. This is promising results, both for traditional computer vision methods to be used in such applications and for the DDPG method.

5.2 Experiment with the physical quadcopter

An outdoor flight test with the physical quadcopter is also conducted. The landing platform is placed on the grey asphalt in the front yard. Then, the quadcopter is controlled manually to take off, hover above the landing platform and land. An image from the flight is shown in Figure 5.13. The perception system is running the entire flight and the estimate output is shown in Figure 5.14. The orange segmentation does not work well enough for the *arrow* method to be used, so no estimate is available from this method. There are also times the two other methods does not work, mostly because it is hard to control the quadcopter manually so that the landing platform is visible in the image.

Discussion

The quality of the position estimate is not good enough to be used with the PID controller, which is why the manual control had to be used. There was only a small amount of wind present during the experiment, however even some small wind gusts makes it quite hard to control the quadcopter and especially to land it. This emphasizes how useful it will be to have robust autonomous flight and landing capabilities on a quadcopter. For the proposed methods in this thesis to be used with a physical quadcopter for autonomous flight, the robustness has to be improved.

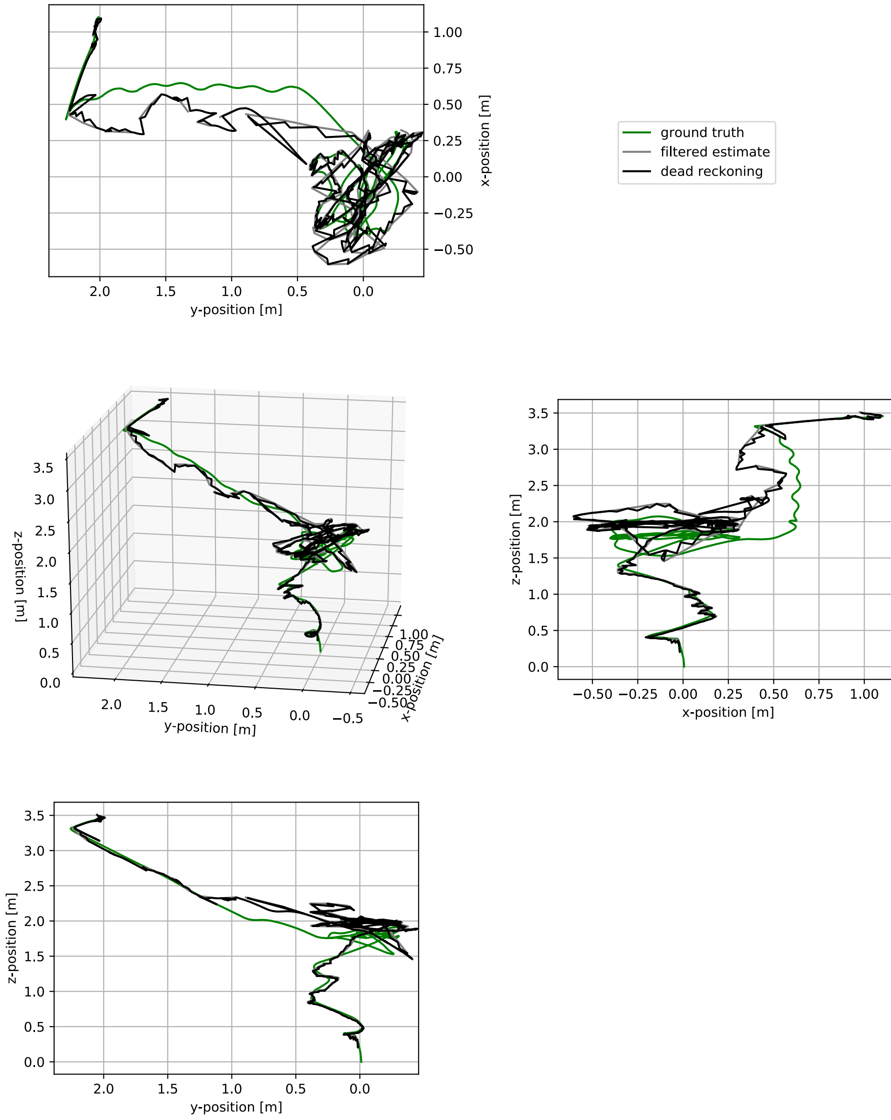


Figure 5.12: The trajectory when landing, using the external DDPG method seen from all three sides.

Figure 5.13: Image from the outdoor testing.

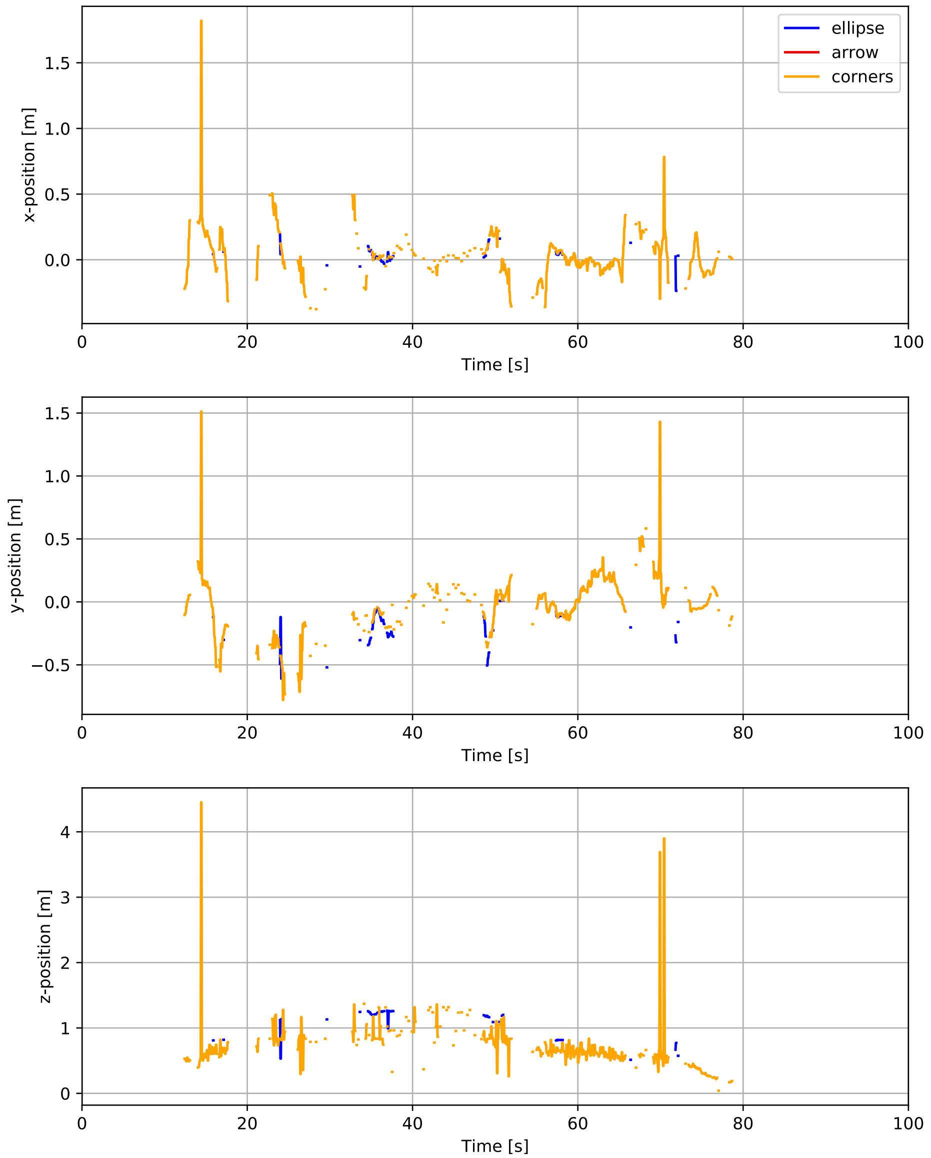


Figure 5.14: Plots showing the position estimates from the outdoor test. The gaps between the lines are due to missing estimates for that time step.

Chapter 6

Conclusion

The question posed in the beginning of this thesis was: *Is it possible to use traditional computer vision methods to give a robust position estimate for a quadcopter in a marine environment, using an onboard monocular camera?* To answer this, this thesis has presented the design of a landing platform that can be utilized for this purpose and a perception system that is able to estimate the quadcopter's simplified pose relative to this landing platform. In order to make the pose estimate more robust, there have been developed three different feature extractors that function in different altitudes above the landing platform. In addition, a simple filter has been applied to smooth the estimate and mitigate outliers. Finally, a dead reckoning module has been added to cope with the cases when no pose estimate is available, by using inertial measurements to incrementally calculate the current pose of the quadcopter.

The perception system has then been tested extensively in a simulator and has been giving good, robust pose estimates here. It has also been tested to a smaller extent with a physical version of the quadcopter and the landing platform, where changes in lightning conditions, the presence of wind, and a bit more noise in the images lowered the quality of the estimates. Thus, it can be said that the traditional methods show potential, but the lack of robustness during physical testing indicates that there has to be done more to ensure the validity of the pose estimate before it can be taken into use with a vessel, such as the ReVolt, at sea. Some suggestions for interesting and promising ideas to investigate in future work are presented in the next section.

6.1 Future work

This thesis presents three different feature extraction method. However, the selection strategy presented in Section 4.1.5 does not guarantee it is the best method that is chosen. Only one method can currently be used at a time and the main criteria for a method to be chosen is that it is available. This means one method can be chosen and yet give an estimate with a large error. One way to solve this problem can be to use another selection strategy. For instance, the average of all the available methods can be used or the average of the two

best methods. This will add some redundancy in case one of the methods gives a somewhat wrong estimate. Another way to mitigate this problem is to use an outlier detection method, such as RANdom SAmple Consensus (RANSAC). RANSAC was originally developed to be applied to just this problem of estimating the location of a camera based on some points in the image with corresponding known 3D positions [55], and should therefore give good results.

The main issue when testing outdoor seemed to be that the colors of the landing platform were perceived significantly different than when testing inside and in the simulator. Therefore, some sort of adaptive color segmentation might be interesting to look into [56]. One way of doing this could be to let the system study an image with a high certainty in the pose estimate, for instance an image taken when the pose estimate has been stable for some time. The system could then learn the colors in this image, to be used as thresholds in the segmentation later.

Another issue that was experienced to a larger extent when flying with the physical quadcopter compared to the simulator was the delay from the quadcopter moved and until the movement showed in the camera. This made it hard to control the quadcopter manually, by using just the camera, which will be mainly what the autonomous quadcopter will have to localize itself with. In this work, all the developed computer systems are deployed on an auxiliary computer. This introduces some extra lag due to the WiFi connection and sending data back and forth to quadcopter. In the future, as better and better quadcopters become available with more and more processing power, the entire perception, planning and control system should be moved on-board the quadcopter. This will also make the quadcopter have a higher level of autonomy as it would not be dependent on a ground-station, except for perhaps higher level control signals from an operator in case of emergency.

Another way of making the estimate more robust can be to apply a more sophisticated filter, such as the Extended Kalman filter [57]. With such a filter, signals from the pose estimator can be fused with signals from the IMU and possibly other sensors, such as GPS. One of the challenges here is that the covariance of the estimate noise should be known. Further investigations should be done on how such a covariance of the noise can be found with estimates that originates from computer vision.

In this study, the quadcopter has no contact with the ship apart from the visual contact with the landing platform. For larger use cases, when the quadcopter is sent on small missions away from the ship, communication between the ship and the quadcopter can be incorporated. For instance, the ship can report its own GPS location and the quadcopter, given that it is supplied with a GPS, could use this information to get close enough to get visual contact with the ship and then use the camera for more precise localization during landing.

Some assumptions have been made in this thesis, for example that the quadcopter and the landing platform are parallel to each other. However, this is not always the case. If the ship is tilted because of the waves or if the quadcopter is tilted because it moves, this assumption does no longer hold. In future works, the motion of the quadcopter and an estimate of the motion of the landing platform could be incorporated into the computer vision system to make the estimate more accurate. Furthermore, the problem of the quadcopter tilting in the wind can be mitigated by using a gimballed camera that can point downwards

regardless of the pose of the quadcopter.

The perception system in this thesis is limited to estimating the simplified pose of the quadcopter. In future work however, there is a lot more possible features of the world that the quadcopter can be taught to perceive. For instance, this could be to detect obstacles around the quadcopter or detect objects of interest, such as people in need of help in the case of a search and rescue mission. It could also be different features relevant to an inspection task, such as looking for rust on constructions and ships. Furthermore, it could be to perceive the state of an object, such as estimate its temperature, color or other conditions the object might be in. All these features and more, would increase the quadcopter's understanding of its environment and thus make it able to make better choices and take better actions.

Although traditional computer vision methods were chosen in this thesis, there should be done more work on using DL methods on such a task as well. One of the reasons is that the methods proposed in this thesis are prone to occlusions and other objects in the image with similar colors to the landing platform. Since the color segmentation alone is the only thing that is used during the first step of the method, the rest of the method is dependent on a good color segmentation. DL methods generally have the ability to look for many more features than this at the same time and can possibly utilize more of the information available in an image than a human can manage to engineer using traditional methods. Although there are some challenges when the entire landing platform is not visible in the image and so on, with the recent progress on the DL field there will probably be found a solution to this. An important success factor for such a project will be to gather a large enough dataset that is representative for different image views, different lighting conditions and possible containing occlusions on some of the images. Another way to utilize DL methods can be as a verification that there actually is a landing platform in the image, before the traditional methods are used. This could remove some false detections and therefore lead to a more robust and reliable estimate.

Bibliography

[1] T. Sundvoll, “A camera-based perception system for autonomous landing on a fixed platform,” Project report in TTK4550, Department of Engineering Cybernetics, NTNU - Norwegian University of Science and Technology, 2019.

[2] M. M. M. Manhães, S. A. Scherer, M. Voss, L. R. Douat, and T. Rauschenbach, “UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation,” in *OCEANS 2016 MTS/IEEE Monterey*, pp. 1–8, IEEE, sep 2016.

[3] P. Pounds, R. Mahony, and P. Corke, “Modelling and control of a quad-rotor robot,” in *Proceedings Australasian Conference on Robotics and Automation 2006*, Australian Robotics and Automation Association Inc., 2006.

[4] “The ReVolt - a new inspirational ship concept.” `https://www.dnvgl.com/technology-innovation/revolt/index.html`. Accessed: 17.06.2020.

[5] M. Hehn and R. D’Andrea, “A flying inverted pendulum,” in *2011 IEEE International Conference on Robotics and Automation*, pp. 763–770, IEEE, 2011.

[6] J. Willmann, F. Augugliaro, T. Cadalbert, R. D’Andrea, F. Gramazio, and M. Kohler, “Aerial robotic construction towards a new field of architectural research,” *International journal of architectural computing*, vol. 10, no. 3, pp. 439–459, 2012.

[7] J. A. Preiss, W. Honig, G. S. Sukhatme, and N. Ayanian, “Crazyswarm: A large nano-quadcopter swarm,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3299–3304, IEEE, 2017.

[8] D. Mellinger, M. Shomin, N. Michael, and V. Kumar, “Cooperative grasping and transport using multiple quadrotors,” in *Distributed autonomous robotic systems*, pp. 545–558, Springer, 2013.

[9] D. Tavakoli, “Autonomous drone landing using deep reinforcement learning,” Master’s thesis, NTNU, 2020.

[10] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. M. O. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[12] L. Wang and X. Bai, "Quadrotor autonomous approaching and landing on a vessel deck," *Journal of Intelligent & Robotic Systems*, vol. 92, no. 1, pp. 125–143, 2018.

[13] D. Falanga, A. Zanchettin, A. Simovic, J. Delmerico, and D. Scaramuzza, "Vision-based autonomous quadrotor landing on a moving platform," in *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pp. 200–207, IEEE, 2017.

[14] T. Venugopalan, T. Taher, and G. Barbastathis, "Autonomous landing of an unmanned aerial vehicle on an autonomous marine vehicle," in *2012 Oceans*, pp. 1–9, IEEE, 2012.

[15] P. M. Møst, "Visual navigation of an autonomous drone," Master's thesis, NTNU, 2014.

[16] A. Borowczyk, D.-T. Nguyen, A. P.-V. Nguyen, D. Q. Nguyen, D. Saussié, and J. Le Ny, "Autonomous landing of a quadcopter on a high-speed ground vehicle," *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 9, pp. 2378–2385, 2017.

[17] F. Fraundorfer and D. Scaramuzza, "Visual odometry: Part ii: Matching, robustness, optimization, and applications," *IEEE Robotics & Automation Magazine*, vol. 19, no. 2, pp. 78–90, 2012.

[18] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[19] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[20] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, pp. 91–99, 2015.

[21] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," in *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969, 2017.

[22] C. Premebida, R. Ambrus, and Z.-C. Marton, "Intelligent robotic perception systems," in *Applications of Mobile Robots*, IntechOpen, 2018.

[23] N. Sünderhauf, O. Brock, W. Scheirer, R. Hadsell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford, *et al.*, "The limits and potentials of deep learning for robotics," *The International Journal of Robotics Research*, vol. 37, no. 4-5, pp. 405–420, 2018.

[24] A. Bendale and T. Boult, "Towards open world recognition," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1893–1902, 2015.

[25] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[26] C. G. Harris, M. Stephens, *et al.*, "A combined corner and edge detector.," in *Alvey vision conference*, vol. 15, Citeseer, 1988.

[27] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[28] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (SURF)," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346–359, 2008.

[29] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *2011 International Conference on Computer Vision*, pp. 2564–2571, Ieee, 2011.

[30] J. Illingworth and J. Kittler, "A survey of the hough transform," *Computer vision, graphics, and image processing*, vol. 44, no. 1, pp. 87–116, 1988.

[31] J. A. Suykens and J. Vandewalle, "Least squares support vector machine classifiers," *Neural processing letters*, vol. 9, no. 3, pp. 293–300, 1999.

[32] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," in *Machine learning techniques for multimedia*, pp. 21–49, Springer, 2008.

[33] M. A. Nielsen, *Neural networks and deep learning*, vol. 2018. Determination press San Francisco, CA, USA:, 2015.

[34] J. Walsh, N. O' Mahony, S. Campbell, A. Carvalho, L. Krpalkova, G. Velasco-Hernandez, S. Harapanahalli, and D. Riordan, "Deep learning vs. traditional computer vision," in *Computer Vision Conference (CVC)*, pp. 128–144, Springer, 2019.

[35] "Opencv-python tutorials - canny edge detection." `https://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html`. Accessed: 09.06.2020.

[36] "Opencv-python tutorials - harris corner detection." `https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html`. Accessed: 09.06.2020.

[37] R. Halır and J. Flusser, "Numerically stable direct least squares fitting of ellipses," in *Proc. 6th International Conference in Central Europe on Computer Graphics and Visualization. WSCG*, vol. 98, pp. 125–132, Citeseer, 1998.

[38] "Wikipedia - ellipse." `https://en.wikipedia.org/wiki/Ellipse`. Accessed: 19.06.2020.

[39] "HSV color model." https://medium.com/neurosapiens/segmentation-and-classification-with-hsv-8f2406c62b39. Accessed: 04.02.2020.

[40] X. X. Lu, "A review of solutions for perspective-n-point problem in camera pose estimation," in *Journal of Physics: Conference Series*, vol. 1087, IOP Publishing, 2018.

[41] J. Kim, M.-S. Kang, and S. Park, "Accurate modeling and robust hovering control for a quad-rotor vtol aircraft," in *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*, pp. 9–26, Springer, 2009.

[42] "Furuno solid state doppler radar." https://www.furuno.no/Userfiles/Sites/files/DRS4D-NXT_E.pdf. Accessed: 04.06.2020.

[43] "Parrot ar.drone 2.0." https://www.parrot.com/global/drones/parrot-ardrone-20-elite-edition. Accessed: 07.06.2020.

[44] "Parrot ar.drone 2.0 - developer guide sdk 2.0." https://jpchanson.github.io/ARdrone/ParrotDevGuide.pdf. Accessed: 07.06.2020.

[45] "About ROS." https://www.ros.org/about-ros/. Accessed: 23.05.2020.

[46] "ROS ardrone_autonomy." http://wiki.ros.org/ardrone_autonomy. Accessed: 23.05.2020.

[47] "Opencv-python tutorials - image denoising." https://docs.opencv.org/3.4/d5/d69/tutorial_py_non_local_means.html. Accessed: 12.06.2020.

[48] "OpenCV moments." https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=moments#moments. Accessed: 30.03.2020.

[49] "Matlab answers - angle between vectors." https://se.mathworks.com/matlabcentral/answers/180131-how-can-i-find-the-angle-between-two-vectors-... ...including-directional-information. Accessed: 09.06.2020.

[50] T. Nodes and N. Gallagher, "Median filters: Some modifications and their properties," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 30, no. 5, pp. 739–746, 1982.

[51] S. Golestan, M. Ramezani, J. M. Guerrero, F. D. Freijedo, and M. Monfared, "Moving average filter based phase-locked loops: Performance analysis and design guidelines," *IEEE Transactions on Power Electronics*, vol. 29, no. 6, pp. 2750–2763, 2013.

[52] L. Fusini, T. A. Johansen, and T. I. Fossen, "A globally exponentially stable non-linear velocity observer for vision-aided uav dead reckoning," in *2016 IEEE Aerospace Conference*, pp. 1–9, IEEE, 2016.

[53] "ROS joy." `http://wiki.ros.org/joy`. Accessed: 24.04.2020.

[54] J. G. Balchen, T. Andresen, and B. A. Foss, *Reguleringsteknikk*. NTNU, Institutt for teknisk kybernetikk, 2016.

[55] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[56] E. Littmann and H. Ritter, "Adaptive color segmentation-a comparison of neural and statistical methods," *IEEE Transactions on neural networks*, vol. 8, no. 1, pp. 175–185, 1997.

[57] J. Sasiadek and P. Hartana, "Sensor data fusion using kalman filter," in *Proceedings of the Third International Conference on Information Fusion*, vol. 2, pp. WED5–19, IEEE, 2000.

Appendix A

Technical specifications of the Parrot AR.Drone 2.0

Table A.1: Technical specifications of the Parrot AR.Drone 2.0. Excerpt from [43].

Weight and dimensions	
without the hull	366g, 45 x 29 cm
with indoor hull	436g, 51,5 x 51,5 cm (72,8 cm on the diagonal)
with outdoor hull	400g, 45,2 x 45,2 cm
Battery	
Type	Lithium polymer battery (3 cells, 11.1V, 1000mAh)
Charging time	1h30
Running time	12 min
Embedded computer system	
Processor	OMAP 3630 1GHz ARM cortex A8
Wi-Fi	b/g/n
OS	Linux 2.6.32
Motors	
Type	Inrunner, brushless; 14.5 watts; 28,500 RPM
Vertical camera	
Type	90° wide-angle diagonal lens camera, CMOS sensor
Video frequency	60fps
Resolution	320x240 pixels (QVGA)
Other sensors	
Accelerometer	3 axis, +/- 50 mg precision
Gyroscope	3 axis, 2000°/second precision
Magnetometer	3 axis, 6° precision
Pressure sensor	+/- 10 Pa precision

Appendix B

ROS message definitions

std_msgs/Empty

sensor_msgs/Image

```
std_msgs/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

geometry_msgs/Twist

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

nav_msgs/Odometry

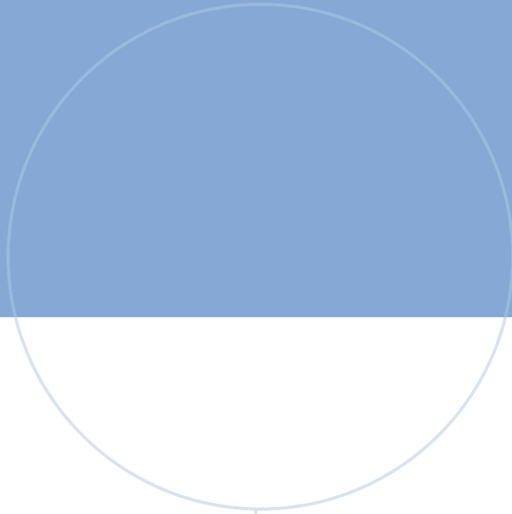
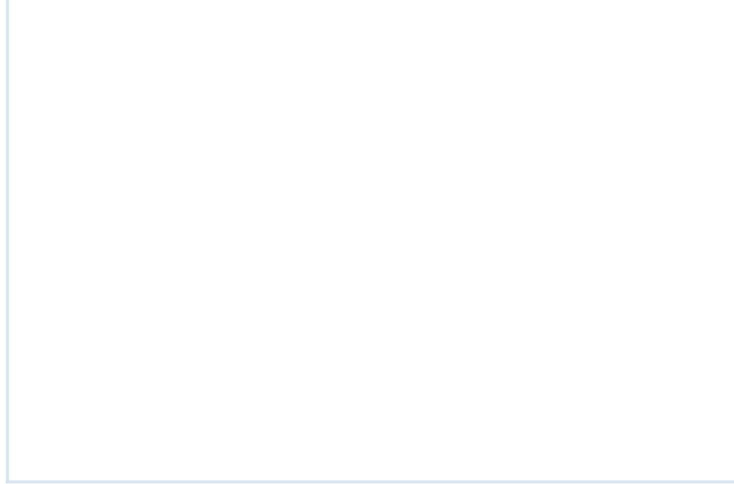
```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

sensor_msgs/Joy

```
std_msgs/Header header
float32[] axes
int32[] buttons
```

ardrone_autonomy/Navdata

```
std_msgs/Header header
float32 batteryPercent
uint32 state
int32 magX
int32 magY
int32 magZ
int32 pressure
int32 temp
float32 wind_speed
float32 wind_angle
float32 wind_comp_angle
float32 rotX
float32 rotY
float32 rotZ
int32 altd
float32 vx
float32 vy
float32 vz
float32 ax
float32 ay
float32 az
uint8 motor1
uint8 motor2
uint8 motor3
uint8 motor4
uint32 tags_count
uint32[] tags_type
uint32[] tags_xc
uint32[] tags_yc
uint32[] tags_width
uint32[] tags_height
float32[] tags_orientation
float32[] tags_distance
float32 tm
```

NTNU
Norwegian University of Science and Technology