Tampere University

Arno Virtanen

MACHINE VISION BASED LANDING ZONE RECOGNITION

Faculty of Information Technology and Communication Sciences
Master's thesis
October 2019

ABSTRACT

ARNO VIRTANEN: Machine vision based landing zone recognition
Tampere University of Technology
Master of Science Thesis, 55 pages, 1 Appendix page
October 2019
Master's Degree Programme in Electrical Engineering
Major: Embedded systems
Examiner: Prof. Timo D. Hämäläinen

Keywords: Machine vision, UAV, ArUco, Gazebo, PX4, ROS, Autonomous landing

This thesis presents how machine vision can be used in the automatic landing process of an unmanned aerial vehicle. The goal of this work is to study what concepts are related to machine vision and UAVs. The goal is to implement a system that can recognise a suitable landing location from the air and land to this location without user interference to the landing procedure.

The work was implemented by using open-source tools and libraries as much as possible and only the landing logic was implemented during this work. For this work different kinds of options were used at the landing location of the UAV. The landing location must enable the UAV to define the coordinates and orientation where the UAV will land at. Also, the landing location must be uniquely identified in an environment where there are multiple landing locations. The landing is considered successful if the UAV lands within 15 cm of the landing location center and with a maximum of a 10-degree deviation. In this work a QR-code like shape was used called the ArUco marker.

The system functionality was developed with the Gazebo simulator environment. In this environment almost all of the components could be simulated and these could communicate with the developed software that was made for this work. The developed software was related to the control logic of the UAV and utilised a library specialised for robot applications called Robot operating system (ROS). The tests conducted in the simulator environment were promising and the landing accuracy was in the range of 5 cm and the deviation of the orientation was 0.5 degrees at maximum. The tests conducted in the simulator environment had an overall success rate of over 90 %.

The software processes developed with the help of the simulation environment could be transferred to actual hardware without modifications. During early test flights it was noticed that the flight should have started simultaneously with the simulator environment tests. This is because in the simulator environment the GPS-signal was nearly perfect but in reality it is not and has great errors in it. During the software development phase it wasn't understood that the used control signals were GPS based. This caused great problems during the real-world tests and it was decided to stop the experiments since the UAVs movement was erratic and unpredictable thus the UAV could not land at all.

It was concluded that applications that require accuracy cannot be based on GPS data. Though the system cannot solely rely on one single sensor but the system should use a variety of different sensors. The system developed during this work functioned as intended in an ideal simulator environment but did not function properly when used in a real-world environment. It was determined that the use of GPS data in the control scheme of the system caused the failure of the real-world experiments.

TIIVISTELMÄ

ARNO VIRTANEN: Tampereen teknillisen yliopiston opinnäytepohja
Tampereen teknillinen yliopisto
Diplomityö, 55 sivua, 1 liitesivu
Lokakuu 2019
Sähkötekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Sulautetut järjestelmät
Tarkastaja: Prof .Timo D. Hämäläinen

Avainsanat: Konenäkö, UAV, ArUco, Gazebo, PX4, ROS, Autonominen laskeutuminen

Diplomityö esittelee kuinka konenäköä voidaan hyödyntää miehittämättömän ilma-aluksen automaattisessa laskeutumisprosessissa. Työn tavoitteena oli tutkia mitä olennaisia käsitteitä liittyy konenäköön ja ilma-alusten ohjaukseen. Työn tavoitteena on toteuttaa järjestelmä, joka pystyy itsenäisesti tunnistamaan halutun laskeutumispaikan ilmasta käsin ja laskeutumaan haluttuun paikkaan ilman käyttäjän puuttumista laskeutumisprosessiin.

Työssä tutkittiin erilaisia vaihtoehtoja, joita voitaisiin käyttää ilma-aluksen laskeutumispaikassa. Laskeutumispaikan avulla ilma-aluksen pitäisi pystyä määrittämään koordinaatit ja orientaation, johon ilma-alus laskeutuu. Laskeutumispaikan pitää olla myös yksilöllisesti tunnistettavissa ympäristössä, jossa on monta samannäköista laskeutumispaikkaa. Laskeutumista pidetään onnistuneena, jos ilma-alus pystyy laskeutumaan 15 senttimetrin päähän laskeutumispaikan keskikohdasta nähden. Ilma-aluksen orientaatio saa poiketa maksimissaan myös 10 astetta laskeutumispaikan nollakohtaan nähden. Työssä päätettiin käyttää QR-koodia muistuttavaa ArUco-tunnistetta.

Järjestelmän toiminta kehitettiin Gazebo-simulaatioympäristöä hyödyntämällä. Simulaatioympäristössä pystyttiin mallintamaan lähes kaikki järjestelmässä käytetyt komponentit ja nämä pystyivät kommunikoimaan kehitettyjen ohjelmistojen kanssa. Kehitetyt ohjelmistot liittyivät ilma-aluksen ohjauslogiikkaan ja toimivat robotiikkasovelluksiin erikoistunutta ohjelmistokirjastoa nimeltä Robot Operating System (ROS). Simulaatioympäristössä tulokset olivat hyvin lupaavia, sillä laskeutumistarkkuus oli n. 5 senttimetrin luokkaa ja orientaatio laskeutumispaikan nollakohdastaan nähden oli maksimissaa 0,5 astetta. Simulaatioympäristössä suoritettujen kokeiden onnistumisprosentti oli yli 90%.

Simulaatioympäristön avulla kehitetty järjestelmä pystyttiin siirtämään sellaisenaan oikealle ilma-alukselle, eikä kehitettyyn ohjelmaan tarvinnut tehdä muutoksia. Ensimmäisten testilentojen aikana kuitenkin huomattiin, että testilennot olisi pitänyt aloittaa oikealla raudalla simulaattorin rinnalla. Simulaatioympäristössä GPS-signaali oli melkein täydellinen, mutta todellisuudessa GPS-signaali on epätarkka. Ohjelmiston kehityksen aikana myöskään ei tiedetty, että käytetty dronen ohjausmetodi perustui GPS-koordinaatteihin. Tämä aiheutti suuria ongelmia käytännön kokeiden aikana ja kokeet jouduttiin keskeyttämään, sillä ilma-alus heilui ilmassa arvaamattomasti, eikä ilma-alus kyennyt laskeutumaan laskeutumisalustalle.

Työssä todettiin, että tarkkuutta vaativissa sovelluksissa GPS-signaaliin perustuvaa ohjausta ei kannata tehdä. Myöskään laite ei saa olla vain yhden anturin varassa ja laitteen tulisi hyödyntää montaa eri sensoria. Työn aikana huomattiin, että markkinoilla on monta hyvää ja erilaista avoimeen lähdekoodiin perustuvaa kirjastoa ja työkalua.

PREFACE

I would like to thank Intel Finland Oy and especially Ville Ilvonen and Eckhart Köppen for giving me the opportunity to pursue this thesis topic. Also I would like to thank people at Intel Finland Oy who gave me invaluable help during my work.

Tampere, 31.10.2019

Arno Virtanen

CONTENTS

1. INTRODUCTION ... 1
2. BACKGROUND ... 2
2.1 Machine vision ... 2
2.2 Camera model ... 2
2.3 Distortions ... 5
2.4 Vision based object pose estimation ... 6
2.5 Methodology ... 7
2.5.1 OpenCV ... 7
2.5.2 Flight controller software ... 7
2.5.3 Robot operating system ... 8
3. SYSTEM ARCHITECTURE ... 10
3.1 Hardware ... 10
3.2 Software ... 11
3.3 Software libraries ... 12
3.3.1 cv_bridge ... 12
3.3.2 MAVROS ... 13
4. SHAPE AND MATERIAL FOR RECOGNITION ... 14
4.1 Design evaluation ... 14
4.1.1 X letter shape ... 15
4.1.2 Concentric shapes ... 16
4.1.3 Fiducial markers ... 17
4.2 Design conclusion ... 20
4.3 Display method ... 21
4.3.1 Passive material ... 22
4.3.2 Visible light ... 22
4.3.3 Invisible light ... 22
5. REALISING MACHINE VISION BASED CONTROL ... 24
5.1 Marker detection ... 24
5.2 Aruco tag decoding ... 25
5.3 Pose estimation ... 27
5.4 System implementation ... 30
5.4.1 Simulator environment ... 30
5.4.2 Drone control ... 33
6. EXPERIMENTS AND ANALYSIS ... 37
6.1 Marker detection ... 37
6.2 Simulation experiments ... 39
6.3 Flight hardware ... 46
6.4 Flight results ... 47

7. CONCLUSIONS 49
REFERENCES 52
APPENDIX A: SIMULATED CAMERA SPECIFICATIONS.. 56

LIST OF FIGURES

Figure 1. *Pinhole camera model. Adapted from [4, p. 372]* 3
Figure 2. *Alternative way of representing the pinhole camera model. Adapted from [4, p. 372]* 3
Figure 3. *Radial distortion. Image has bulging distortion when when $k_i < 0$ and pincushion distortion when $k_i > 0$.[50]* 5
Figure 4. *The PnP problem. How can the corresponding points between the 3D world and the image points be used to define the camera or UAV pose with respect to a landing location. [41]* 7
Figure 5. *PX4 flight stack. [37]* 8

Figure 6. *Hardware system architecture* 10
Figure 7. *Software system architecture* 12
Figure 8. *cv_bridge interface. [29]* 13

Figure 9. *Model to describe landing procedure. Picture adapted from [15]* 14
Figure 11. *Concentric shapes* 17
Figure 12. *Examples of fiducial markers. [14]* 18
Figure 13. *Aerial view of HiveUAV™ landing pad. Picture adapted from [18]* 19

Figure 14. *Aruco detection pipeline* 25
Figure 15. *Aruco tags* 26
Figure 16. *Corresponding values to the color of the grid element* 26
Figure 17. *Gazebo simulation setup* 30
Figure 18. *Simulator environment. Adapted from [39]* 31
Figure 19. *mRo Pixhawk Flight Controller (Pixhawk 1). [34]* 32
Figure 20. *Desired landing procedure* 35
Figure 21. *Implemented UAV landing scheme* 36

Figure 22. *Logitech C920 HD Pro Webcam. [24]* 37
Figure 23. *Initial aruco detection setup* 37
Figure 24. *Initial aruco detection setup* 38
Figure 25. *Paths used for evaluating system functionality* 39
Figure 26. *Rendering issue in Gazebo* 41
Figure 27. *Landing accuracy* 45
Figure 28. *UAV used for field experiments* 46
Figure 29. *RPi CPU utilization* 47

LIST OF SYMBOLS AND ABBREVIATIONS

R_{ij}	Rotation coefficient of extrinsic parameter of a camera
T_{ij}	Translation coefficient of extrinsic parameter of a camera
X_O	A horizontal object point
Y_O	A vertical object point
γ	Coefficient to describe how perpendicular the vertical and horizontal pixel coordinates are to each other
c_x	Horizontal offset of the center of the image sensor from the optical axis
c_y	Vertical offset of the center of the image sensor from the optical axis
f_x	Horizontal size of a pixel
f_x	Vertical size of a pixel
k_i	Radial distortion coefficient of a camera
p_i	Tangential distortion coefficient of a camera
x_p	The horizontal point of a projected image of an object on an image plane
Z	The distance from the camera aperture to an object what reflects light rays in to the camera
f	Focal length
u	Horizontal pixel coordinate on the image plane
v	Vertical pixel coordinate on the image plane
EKF	Extended kalman filter
ESC	Electronic speed controller
GNSS	Global navigation satellite system
GPS	Global positioning system
IMU	Inertial measurement unit
IR	Infrared radiation
MAVLink	Micro Air Vehicle Link
NoIR camera	Camera that doesn’t have a infrared fileter
P*n*P	Perspective-*n*-Point
PM	Power management
Pose	Objects location and rotation relative to a coordinate system
RGB camera	Sensor that is sensitive to red, green and blue bands of light
ROS	Robot operating system
RTK	Real-time kinematics
UAV	Unmanned aerial vehicle

1. INTRODUCTION

Machine vision systems are increasing in number and in a variety of different application fields, in addition to industrial and manufacturing solutions.

This thesis considers the possibility of using machine vision in an unmanned aerial vehicle (UAV). The processed vision data would be utilized to achieve a fully autonomous landing procedure where the UAV would be able to land to a specific location with good accuracy. A shape or pattern must be designed on a landing pad which a UAV can recognize from the air. The scope of this work is to define a pattern or a shape that is recognizable from a distance of 5 meters and supports the landing accuracy of 15 cm from the center of the landing location.

Previous research has been made where Brommer et al. [5] used vision-based data for precision landing. This research used markers which consist of a combination of black-and-white squares. This type of markers has the proven ability to provide a translation and rotation (pose) estimate with respect to a camera. This type of marker seems promising to be utilized in this work as well. In [5] a marker that was printed on a paper surface was used.

There are many publications [5] [40] [21] [42] related to autonomous landing based on machine vision and the scope of this work is to get familiar with all the concepts related to machine vision and UAV's. In this work it is studied how these two topics can be integrated together to achieve a fully functional application. It is also studied how the application can be tested before using actual hardware and what challenges there are when implementing an application with real hardware.

In chapter 2 the reader is introduced to the theoretical background of the concepts used in this work and to the tools that are used in this work. Chapter 3 describes the system architecture and how the different system components are connected together. Chapter 4 presents a comparison between different type of shapes and patterns. The objective is to choose one shape that is to be implemented for simulation and real-world experiments.

Chapter 5 focuses on describing how the application functionality is implemented. In Chapter 6 the system functionality is evaluated. The vision based landing is tested both in a simulator and in a real world environment. Chapter 7 summarizes the topics presented in this thesis and gives ideas how can the system be developed in the future.

2. BACKGROUND

First the theoretical background is presented about the concepts that are used in this work. After this, the tools are presented that transfer the presented theory to practice. Also, other used tools are presented which were used in this work.

2.1 Machine vision

The foundation of this work is using an imaging unit attached to an UAV in order to observe and search a specified area to locate a suitable landing location. Machine vision is the concept of giving a machine or computer the ability to visually observe and interpret a scene. The concept of machine vision has emerged in as early as the 1930's and the earliest patents were filed in the 1950's regarding optical character recognition [54, p. 7].

2.2 Camera model

Since an imaging unit provides essential information for the system in this work it is natural to describe how a camera is modelled. The most simple and a convenient way to model a camera is called the pinhole camera model. Equations derived from this model are used in the programs to estimate the UAVs translation and orientation from a suitable landing location which is described in chapter 4. The pinhole model is depicted in Figure 1. [4, p. 370]

This model is not a practical way of capturing images but is a way to describe the basics of imaging. This model is based on a plane called the pinhole or focal plane which has a infinitesimal hole referred to as an aperture. This plane blocks all the incoming rays reflected from an object or a scene (depicted as point O in Figure 1) except the ones that pass through the aperture. Only one ray reflected from an object can pass through the aperture. [4, p. 370-371]

The rays that pass the aperture are "projected" onto a plane called the image or projective plane behind the pinhole plane. This is where the image is formed or "projected" from an object or a scene. In Figure 1 the distance from an object or a scene to the pinhole plane is Z and the distance from the pinhole plane to the image plane is f which is also referred to as the focal length. The actual length of an object or scene is X_O and x_p is the "projected" image length on the projective plane. A perpendicular line that points out from the center of the image plane is called the optical axis. [4, p. 370-371]

Figure 1 contains two similar triangles with the relation $-x_p/f = x_o/Z$. From this relation

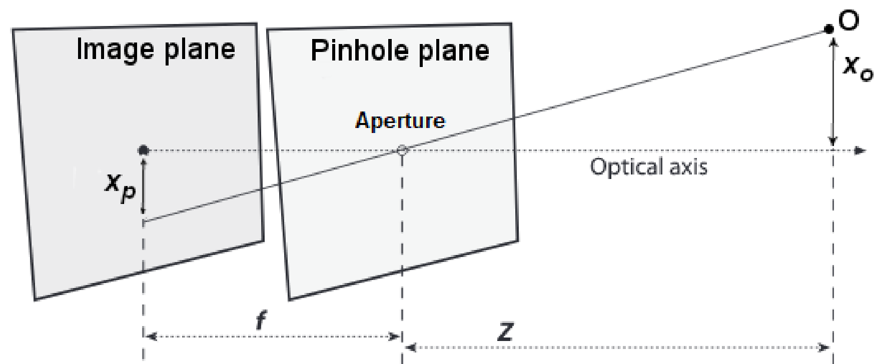


Figure 1. *Pinhole camera model. Adapted from [4, p. 372]*

the following equation can be formed: [4, p. 371]:

$$-x_p = f\frac{x_o}{Z} \tag{2.1}$$

The model depicted in Figure 1 can be rearranged to form a model where the two planes positions are swapped with each other [4, p. 371]. This model is depicted in Figure 2. This model cannot be physically built as the one depicted in Figure 1.

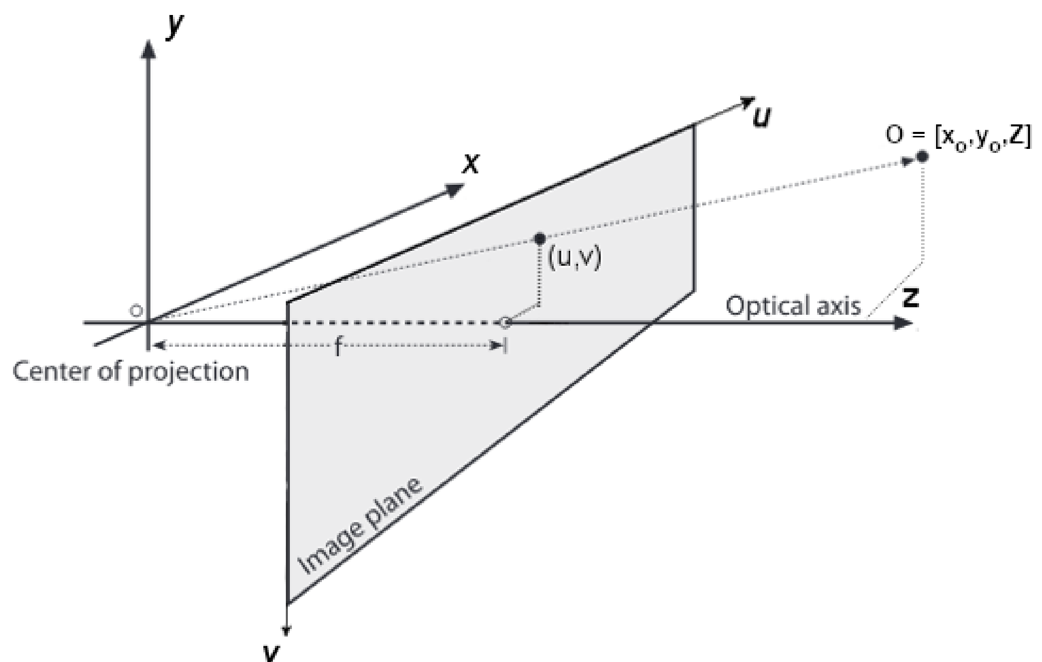


Figure 2. *Alternative way of representing the pinhole camera model. Adapted from [4, p. 372]*

In this model the aperture in Figure 1 is described as the center of projection and the rays that passed the tiny aperture in the older model are now modelled as they would be heading

towards the center of projection in Figure 2. The object points from the physical world are transformed to pixel coordinates on the image plane. These coordinates are denoted with u and v. Two similar triangles are found in this model as in the model depicted in Figure 1. The relation of the triangles in the horizontal axis is $u/f = X_O/Z$ and $v/f = Y_O/Z$ in the vertical axis. With this new model the image is no longer upside down as it was in the older model. [4, p. 371-372]

In reality the manufacturing process of an image sensor will introduce non-idealistic characteristics. These non-idealistic characteristics have an effect on the location of the image sensor and the physical shape of the pixels of the image sensor. Ideally the center of the image sensor would be on the optical axis but in reality the center is not on the optical axis. This offset from the optical axis can be denoted with c_x and c_y. [4, p. 373]

In an ideal camera the pixels are considered to be a shape of a square and the focal length could be denoted only as f. In reality the pixels are more rectangular than a square.The parameters that describe the focal length in pixels are f_x and f_y. When the non-idealistic characteristics are taken into account a point from the physical world can be projected to a pixel location with the equations 2.2. This process of transforming points from the physical world to pixel coordinates is called *projective transformation*. [4, p. 373]

$$u = f_x\left(\frac{x_o}{Z}\right) + c_x, \quad v = f_y\left(\frac{y_o}{Z}\right) + c_y \tag{2.2}$$

A convenient way to present the equations related to projective transformation is to use *homogeneous coordinates*. Homogeneous coordinates is a widespread method when operating with this type of transformations. To represent a standard Euclidean point in $\mathbb{R}^n$ a vector of n elements must be used. With homogeneous coordinates the same point is expressed with n + 1 elements. In equation 2.3 the conversion is presented between the homogeneous coordinates (left side) and Euclidean space (right side). [30]

$$\begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \\ w \end{bmatrix} = \begin{pmatrix} x_1/w \\ x_2/w \\ . \\ . \\ . \\ x_n/w \end{pmatrix} \tag{2.3}$$

With the homogeneous coordinate representation of the projective transformation can be represented as:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.4}$$

Where u and v are pixel coordinates on the image plane [30]. Expanding the equation 2.4 to incorporate the non-idealistic characteristics the equations in 2.2 the projection of points can be expressed with a 3-by-3 matrix in equation 2.5. This matrix is usually referred to as the camera matrix or intrinsic parameters of the camera. This matrix also contains a fifth coefficient γ that describes how perpendicular the pixel coordinates u and v are. Usually this is value is set to 0. [6]

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{pmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.5}$$

2.3 Distortions

The main reason that the pinhole camera model is not used in commercial cameras is that the pinhole model isn't able to gather enough light in fractions of a second when an image is taken. This is why lenses are used that more light can enter the camera when an image is being captured. Using lenses introduces distortions and this is why the camera must be calibrated to correct these distortions. Lenses usually introduce two types of distortions: radial and tangential distortion. As with the image sensors the main reason for the distortions are the errors caused by the manufacturing process. Ideally a lens would be in a parabolic shape but in practice the lenses are spherical. [4, p. 375]

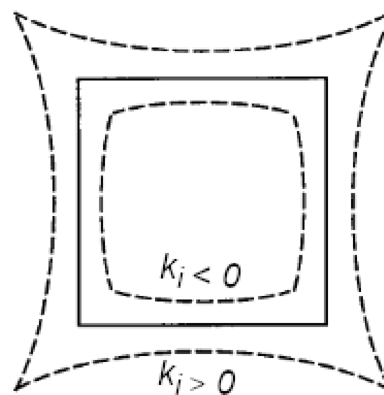


Figure 3. *Radial distortion. Image has bulging distortion when when k_i < 0 and pincushion distortion when k_i > 0.[50]*

Radial distortion is visually observed as distortion near the edges of an image. This can be seen as an bulging or pincushion distortion of an image and is caused by that rays are bent more at the edges of a lens than the center of the lens. This radial distortion is visualized in Figure 3. The radial distortion coefficients is denoted by terms k_1, k_2 and k_3. These terms are derived from a Taylor series and when the values are negative the image appears to be bulging and is known as barrel distortion and when these values are positive the image has a pincushion distortion. [4, p. 375]

The tangential distortion is caused because the lens is not perfectly parallel with the image plane. The tangential distortion coefficients is denoted by terms p_1 and p_2. The distortion coefficients can be stored in one 5-by-1 matrix in the form of k_1, k_2, p_1 and p_2 and k_3. [4, p. 375-376]

2.4 Vision based object pose estimation

Visually estimating the pose of a suitable landing location with a camera introduces a classical problem in the field of machine vision. With equation 2.5 physical 3D points can be transformed into 2D pixel coordinates when the camera coordinate system doesn't have any translation or rotation with respect to the world coordinate system [6].

As a UAV will be flying the camera unit will have a non-zero translation and rotation from the landing location. This translation and orientation of a image unit with respect to the world coordinate system (landing location) coordinate system can be described by a 3-by-4 matrix. This matrix consists of a rotation matrix R_{3x3} and translation vector T_{3x1}. This matrix is presented in equation 2.6 and is called the camera extrinsic parameters. [6]

$$\begin{bmatrix} R_{00} & R_{01} & R_{02} & T_{00} \\ R_{10} & R_{11} & R_{12} & T_{01} \\ R_{20} & R_{21} & R_{22} & T_{02} \end{bmatrix} \tag{2.6}$$

Now the full perspective model can be formed. The full perspective model is presented in equation 2.7.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R_{00} & R_{01} & R_{02} & T_{00} \\ R_{10} & R_{11} & R_{12} & T_{01} \\ R_{20} & R_{21} & R_{22} & T_{02} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.7}$$

This full perspective model can be used to calculate the values of the camera extrinsic parameters. This is possible if the camera matrix, the points of the physical world and the corresponding pixel coordinates are known.

The camera matrix can be extracted with camera calibration algorithms. In this work the camera calibration procedure [19] was used for acquiring the camera characteristics. Now the problem is to find the physical 3D points of the landing location and the corresponding 2D image points. This introduces a problem called Perspective-n-Point problem (PnP) presented in Figure 4. Many existing methods have been developed to solve this problem. [23] The method used to solve the problem in this work is presented in chapter 5. After finding the corresponding points between the physical world and the pixel coordinates the transformation which is responsible for each corresponding point can be calculated.

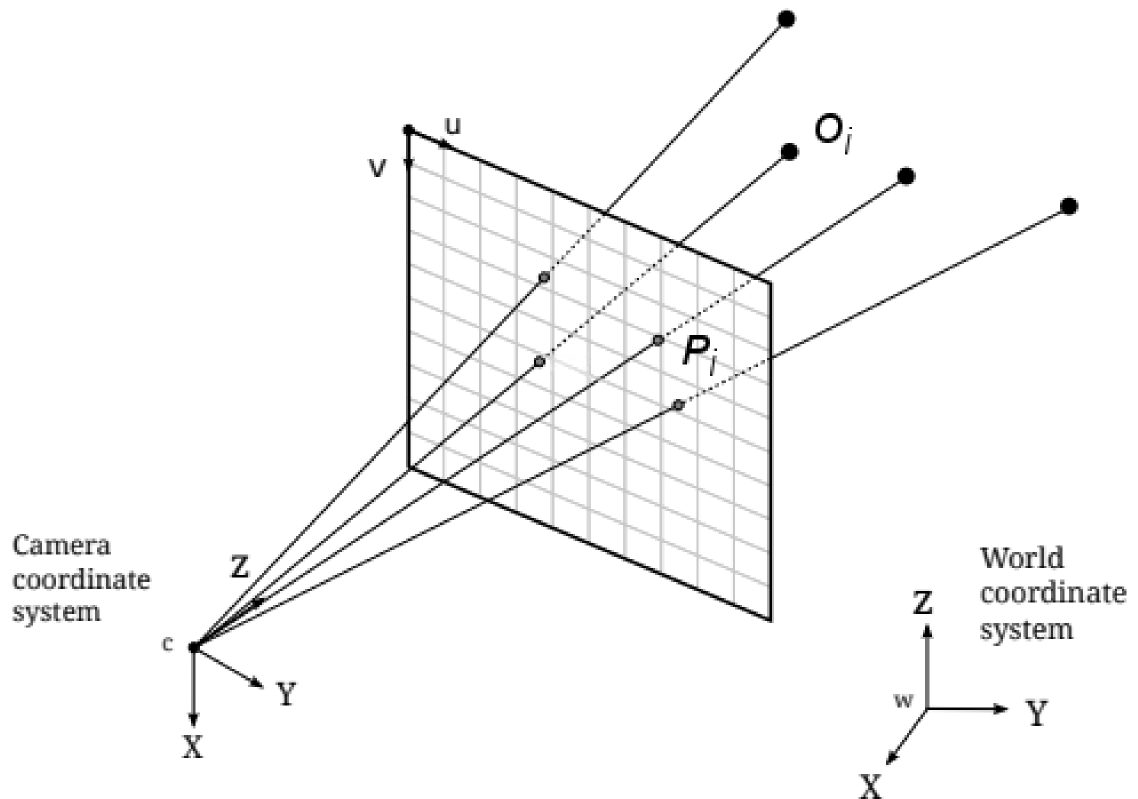


Figure 4. *The PnP problem. How can the corresponding points between the 3D world and the image points be used to define the camera or UAV pose with respect to a landing location. [41]*

2.5 Methodology

It was decided to use existing libraries and tools as much as possible in this work. For every aspect of work a existing library could be used, only the flight logic itself had to be implemented during this work. The used libraries and tools are presented in the following sections.

2.5.1 OpenCV

The system relies on image processing algorithms to locate a suitable landing location. In this work the Open source computer vision library (OpenCv) [46] was used to process image frames from an image sensor. This open-source library is a very-well known and actively maintained and also provides functions that can estimate the pose of different objects in an image relative to the image sensor. Therefore, OpenCV was decided to be used for the image processing part of this work.

2.5.2 Flight controller software

It was decided to use an existing flight control software since there are highly developed open source options available such like PX4 [36] or ardupilot [47] thus it would be out of

the scope of this work to implement a new one for this work.

There was no prior experience with either of the flight controller software but the PX4 flight controller software seemed to provide a better documentation and PX4 seemed to be used more in academic papers and hobby projects. This is why PX4 was chosen to be used in this work.

The internal part of the flight controller software that is responsible for the estimation and flight control system is called a *flight stack* [37]. In Figure 5 the high level software architecture of the PX4 flight stack is presented with only the relevant concepts regarding this work.

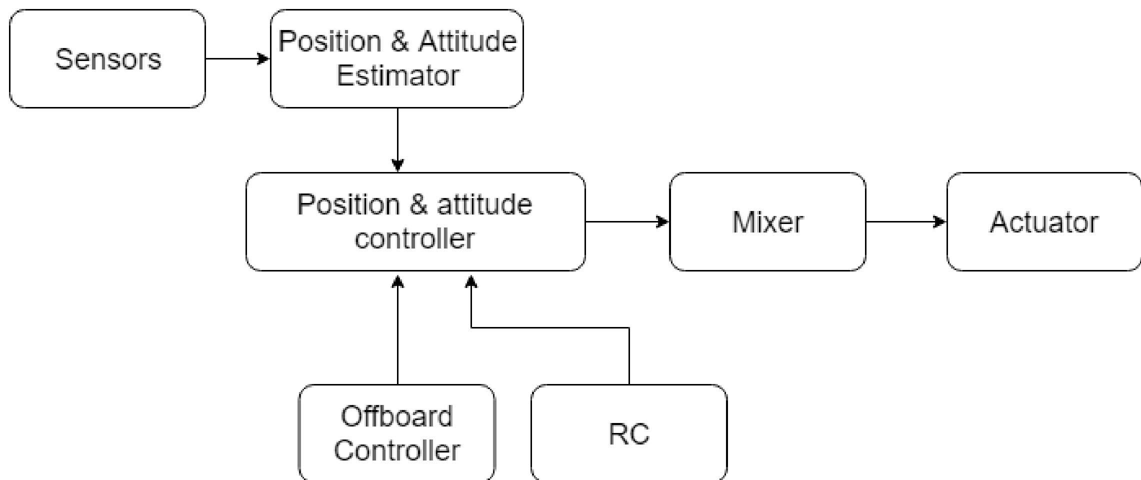


Figure 5. *PX4 flight stack. [37]*

The position and the attitude estimator are used for estimating the state i.e. position of a UAV. The position is calculated based on the raw sensor values received from different sensors such as global navigation satellite system (GNSS) or inertial measurement unit (IMU) data [37]. These sensor measurements are processed by an extended kalman filter (EKF) algorithm. [35]

This calculated value is sent to the position and attitude controller of the system. This position and attitude controller also receive a setpoint value from a radio control (RC) system or from an offboard control unit. This process outputs a thrust and an attitude setpoint that steer the UAV towards a wanted position. The mixer is responsible for translating the given setpoint values to individual motor commands. [37]

2.5.3 Robot operating system

The Robot Operating System (ROS) library is a meta-operating system for robot applications. ROS provides services such as hardware abstraction, low-level device control, communication between processes and package management. [13] In this work ROS was used for handling the communication between different processes and for system debugging purposes.

The ROS framework functions based on a peer-to-peer network of ROS processes and the ones used in this work are explained in the following [12]:

- Master: Acts a central name service in the ROS network. This name service stores information about the ROS nodes in the network and without the master process the nodes wouldn't be able to discover each other nor exchange messages.
- Parameter server: Part of the Master process and provides data storage by key.
- Nodes: ROS nodes are the units where the system computation is done and a system usually consist of many nodes each responsible for a specific task.
- Messages: The ROS nodes communicate with other ROS nodes by messages which is a data structure consisting of typed fields.
- Topics: Messages are published by a node to a topic. Nodes can subscribe to a published topic and receive data that another node is publishing. A node can publish and subscribe to multiple topics.
- Services: Services provide an alternative way to the publish and subscribe semantic. The service functionality uses the request and reply topology where a client sends a request to a providing node and waits for this servicing nodes reply.

3. SYSTEM ARCHITECTURE

The top level architecture of the system is presented next. These hardware and software components can be used both in a simulator and in a real-world environment.

3.1 Hardware

The top level hardware architecture is presented in Figure 6.

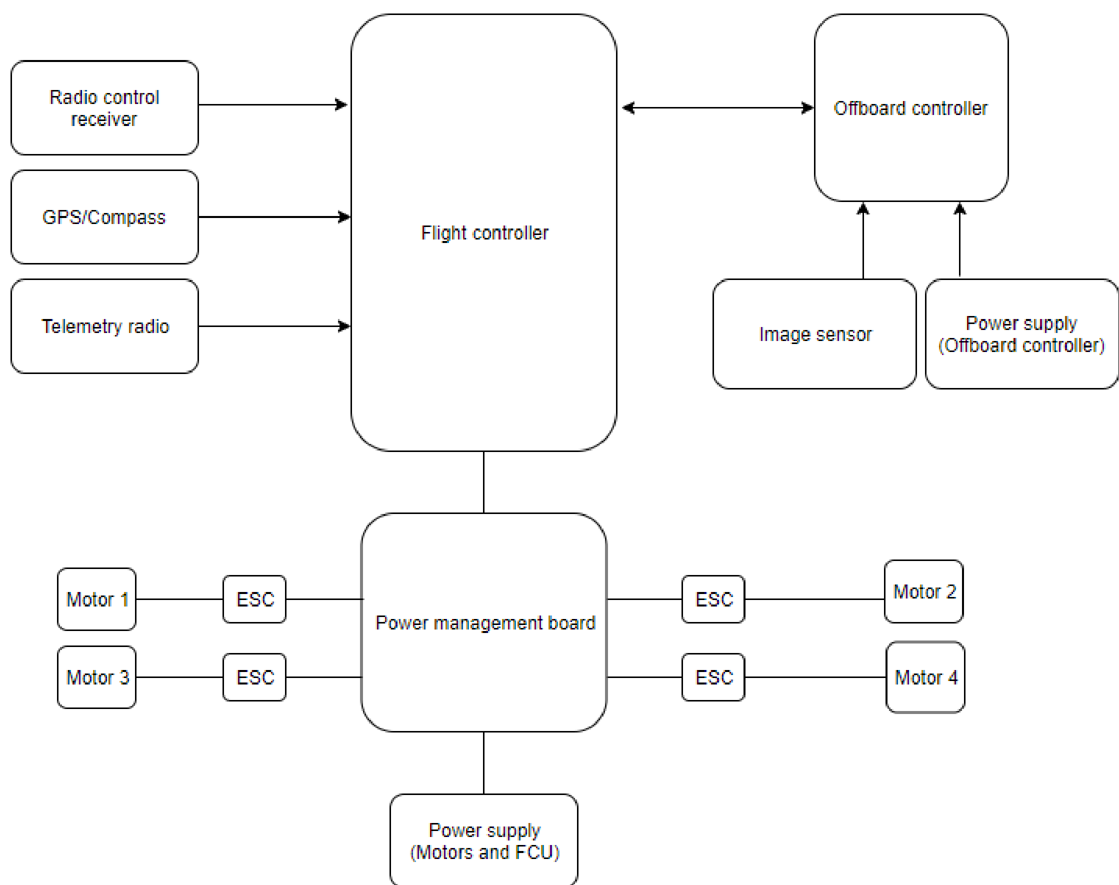


Figure 6. *Hardware system architecture*

The flight controller unit is the main control unit of the UAV and is running flight controller software presented in 2.5.2. This unit is in charge of sending control signals to the four connected motors. These motors are connected to the flight controller unit through electronic speed controllers (ESC) and a power management board (PM board). This board also distributes power from a battery between the flight controller and the connected motors. Control signals from the flight control unit to the motors are sent through the power management board.

The flight controller unit used in this work incorporates a set of different sensors and additional devices can be connected to it. In this work a GNSS receiver module, radio link, telemetry radio and a external computing unit is connected to the flight control unit.

A GNSS system provides satellite data for the UAV. This data can be used to determine the UAVs position (longitude, latitude and altitude) in an area. A GNSS system can provide position data that has a accuracy from a few centimetres to multiple meters.

A connected telemetry radio enables a pilot to remotely monitor the UAV's status. Also the UAV can be remotely controlled through a telemetry radio but in this work it was only used for monitoring purposes.

In this work the drone can be manually controlled with a radio control system through a radio module or navigation messages can be sent to the flight control unit from an external compute unit called the offboard controller. The algorithms for the autonomous landing procedure are executed on this companion computer and will only send messages that contain a location where the UAV should go. The flight controller is responsible for converting these location messages to control signals for the motors.

The offboard controller is used for providing additional computing power for the system. This computing unit is used to process the image data that is received from an image sensor and is used to send movement commands to the flight controller based on the image data. An external power supply is needed for the offboard computing unit since the power supply for the motors and flight controller unit doesn't provide suitable voltages for the offboard controller.

3.2 Software

The software architecture is presented in Figure 7. These software processes are executed on a companion computer presented in Figure 7. This companion computer is referred to as an offboard controller in this work. The software architecture is based on the ROS framework [11]. The nodes that utilise the ROS framework are marked with the keyword ROS in Figure 6. The first version of the robot operating system which was used in this work only functions on Unix-based platforms.

In this work two ROS nodes were implemented: one that handles the image processing and one that is in charge of sending movement messages to the flight controller. A more detailed explanation of the functionality of these processes are presented later on this work in chapter 5.

The system functions as follows. First image frames are acquired with an image sensor, whether it is in a simulator world or an actual camera mounted on a UAV. This is the only system component that changes between the simulator and real-world environment in this work.

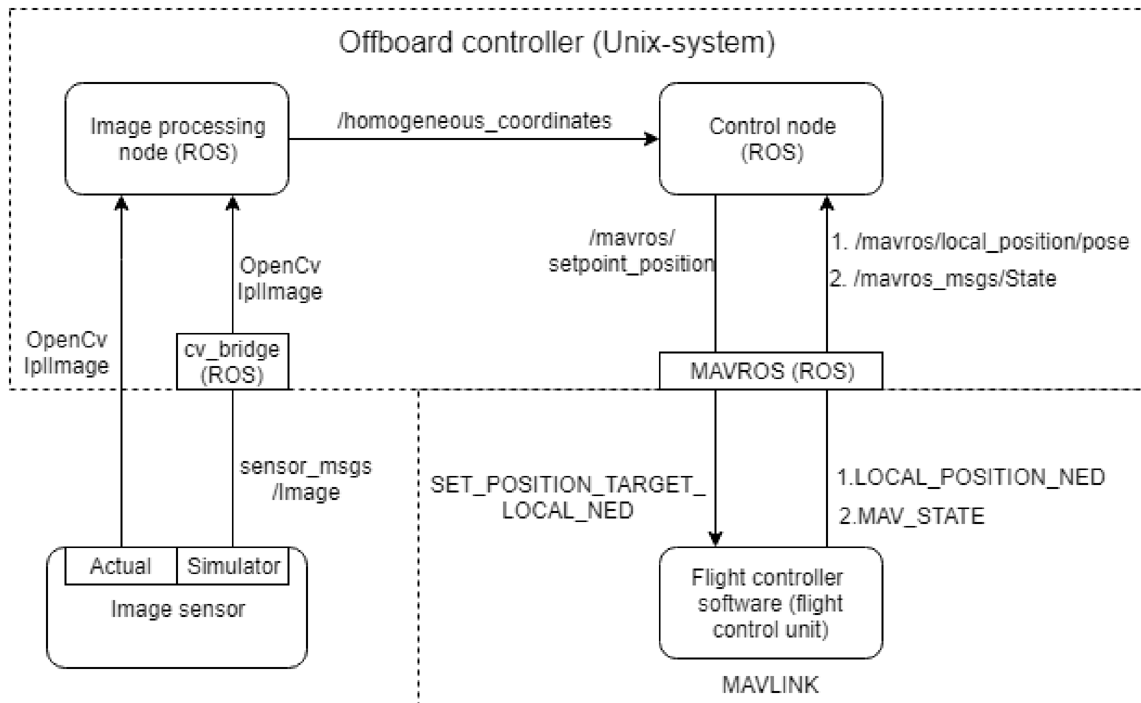


Figure 7. *Software system architecture*

The captured image frames are processed by the image processing node. This node processes the image data and calculates the translation and orientation of a detected marker with respect to the camera. After this calculation the pose information of a landing location is forwarded to the controller node. This node uses the modules provided by the OpenCV library to process the image data. The pose data of a landing location is broadcast to a topic called */homogeneous_coordinates*

The control node receives also data from the flight controller unit. The flight controller unit sends various kind of messages but the control node subscribes only to the data about the position and state information broadcast by the flight control unit. The desired setpoint of the UAV is calculated based on the position data the control node receives from the flight control unit and the data from the image processing unit.

3.3 Software libraries

3.3.1 cv_bridge

The image processing node uses the OpenCV library to process the image data it receives from a simulated or an actual image sensor. The simulator environment publishes all the simulated component data to various ROS topics including the image frames from a simulated image sensor. The simulated image frames are in the type of *sensor_msgs/Image* but the OpenCV library requires that the image frames are in the type of *IplImage*.

This conversion can be achieved with the ROS package called *cv_bridge* [29]. The functionality of the package is illustrated in Figure 8. With the non-simulated hardware

this conversion doesn't need to be made since the image frames are in a correct type so the *cv_bridge* package can be left out when using real hardware.

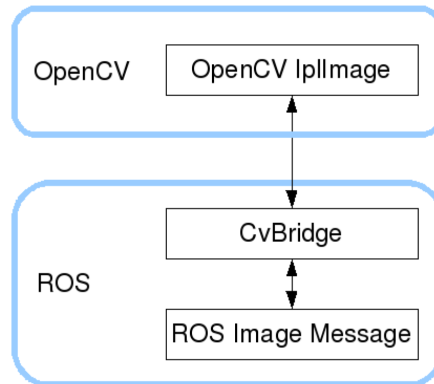


Figure 8. *cv_bridge interface. [29]*

3.3.2 MAVROS

The PX4 flight controller software and the control node use different communication protocols. By default the control node isn't able to communicate with the flight controller software. The flight control software uses the Micro Air Vehicle Link (MAVLink) communication protocol [33] and the control node operates with the ROS communication protocol. This means that the messages between these two components need to be converted that the components are able to communicate with each other. The MAVROS package [8] provides this conversion that ROS nodes can communicate with devices using the MAVLink communication protocol and vice versa.

The messages that are sent from the control process and vice versa are presented in Table 2.

Table 2. *Control node messages*

Origin	Destination	Message
Control node (ROS)	Flight control unit	
/mavros/setpoint_position/ local	SET_POSITION_TARGET _LOCAL_NED	Movement setpoint
Flight control unit	Control node (ROS)	
LOCAL_POSITION_NED MAV_STATE	/mavros/local_position/pose /mavros_msgs/State	UAV position (GPS) UAV flying mode

The control signals from the control node are broadcast to a topic called */mavros/setpoint_position/local* and are converted to the MAVLink message *SET_POSITION_TARGET _LOCAL_NED*. The flight control unit uses the *LOCAL_POSITION_NED* message to publish its current location in the UAV's local frame. This message is converted and published to the */mavros/local_position/pose* ROS topic. The drone flying mode state is published to the *mavros_msgs/State* ROS topic.

4. SHAPE AND MATERIAL FOR RECOGNITION

In this chapter a shape and its displaying method is presented what will support an autonomous landing procedure. The objective of this work is to choose a pattern that can guide the drone from a distance of five meters. It is not necessary to guide the drone with a pattern from a greater distance than five meters since other localisation mechanisms can be utilized such as GNSS data (GPS, Galileo, GLONASS). This localisation method is not adequate for precision landing since e.g. global positioning system (GPS) accuracy error is in the meter range [2].

The GPS signal accuracy can be improved by using the real-time kinematics (RTK) GPS which has a proven accuracy of centimeters thus it could be used but the signal accuracy is prone to errors. [52]

When defining the requirements for a shape and its display method, it is almost natural to think from a UAVs perspective and how it would see a shape laying on the ground. This reasoning could be done from a drone presented in Figure 9 and this point-of-view is used throughout this chapter.

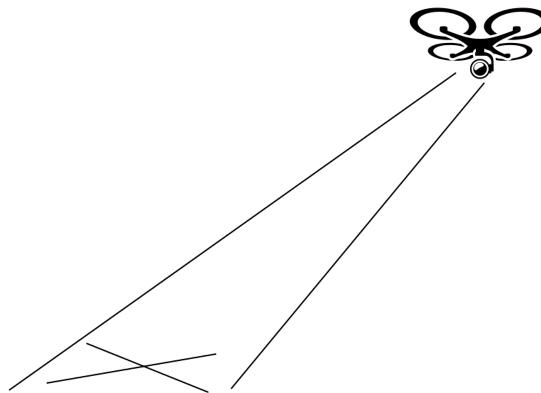


Figure 9. *Model to describe landing procedure. Picture adapted from [15]*

In the Table 3 the requirements are presented what the landing pad i.e. a shape and its displaying method must provide for the guiding process of a UAV.

4.1 Design evaluation

In this section an analysis between different shape designs is presented. Ultimately the most promising one is chosen to be implemented for simulation and field tests. The goal of this section is to define a shape that will satisfy the requirements 1, 2, 3 and 4 from Table

Table 3. Design criteria

Number	Requirement
1	The shape on the pad must be recognized from a viewing distance of 5 m
2	The shape must guide the drone throughout the landing process
3	The shape on the pad will provide information in which orientation the drone should land on the pad
4	The shape should support multiple simultaneously landing drones
5	The display method of the shape needs to work in various lighting conditions
6	The UAV should be able to land within 15 cm of the shape center

3. The item number 5 of Table 3 will be covered later in this document since it is related to the displaying method of the shape. The accuracy of the landing procedure (item number 6) is presented in chapter 6.

The evaluation of a specific shape will be divided into four parts corresponding to the items in Table 3. Each part will consist of a detailed analysis how well a shape fulfils a specific criteria. The analysis is performed as a UAV would be performing a required action, i.e. trying to define a desired orientation from a detected shape.

The first part of the evaluation will describe how the drone would see a specific shape from a distance of 5 m. The drone can guide itself to this altitude with other methods and the vision-based landing will begin after reaching this point.

The second part will analyse how can the shape guide the drone throughout the entire landing procedure. This means that the drone needs to acquire relevant information from the recognised shape such as altitude and rotation relative to the shape. Also at some point the shapes outlines might be out of the drones field of view but in some way the shape must be visible to the drone.

The third part is to analyse how can the drone adjust its orientation from the detected shape throughout its landing procedure and to land in a predefined direction. The fourth and last part will analyse how the shape would perform in a scenario where multiple drones would need to be guided simultaneously to different locations with the presented shape.

4.1.1 X letter shape

To satisfy the first requirement of recognising a shape from a distance, an intuitive approach would be to construct a large shape. For example the shape could be similar to the one in Figure 9 as in the form of the letter x. From the drone's point of view this could be seen as the image presented in Figure 10a. This would satisfy the item number 1 of the criteria list. Naturally how large the shape is directly determines how small the landing pad is at minimum.

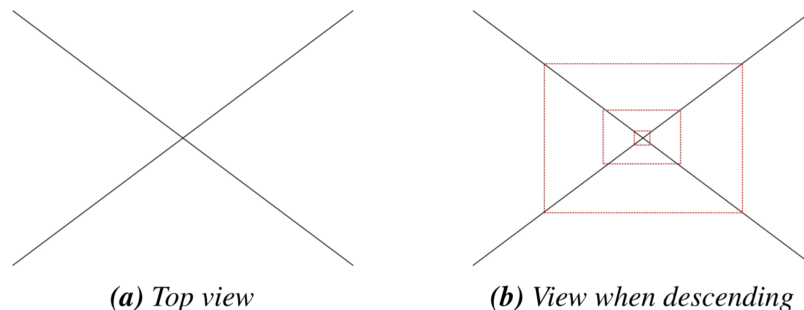
As the drone would start to descend, naturally some of the part of the shape will disappear from the camera's field of view. This situation is presented in Figure 10b. The red dashed

line indicates how the image would be seen from a drone's point of view as it descends closer to the ground and towards the shape. From this image it can be observed that shape can still be recognised when the shape gets closer to a camera thus satisfying the item number 2 of the criteria list.

Although this shape seems suitable according to the first two requirements the remaining shape related requirements would be nearly impossible to achieve with this shape under investigation. The reasoning for this is that the shape is uniform and a distinct heading cannot be interpreted from this. This means that the drone cannot rotate to a desired heading.

Also this shape cannot be used to guide simultaneously multiple drones to desired locations. This is because this shape cannot be uniquely identified if many of these shapes would be deployed in a specified area. It would be impossible to find a unique landing spot from an area. One plausible scenario would be to add an additional component which contains a unique identifier for the landing pad and would be placed beside this shape.

To conclude this shape satisfies some of the requirements but not all. Therefore the letter x shape can be discarded from further considerations.



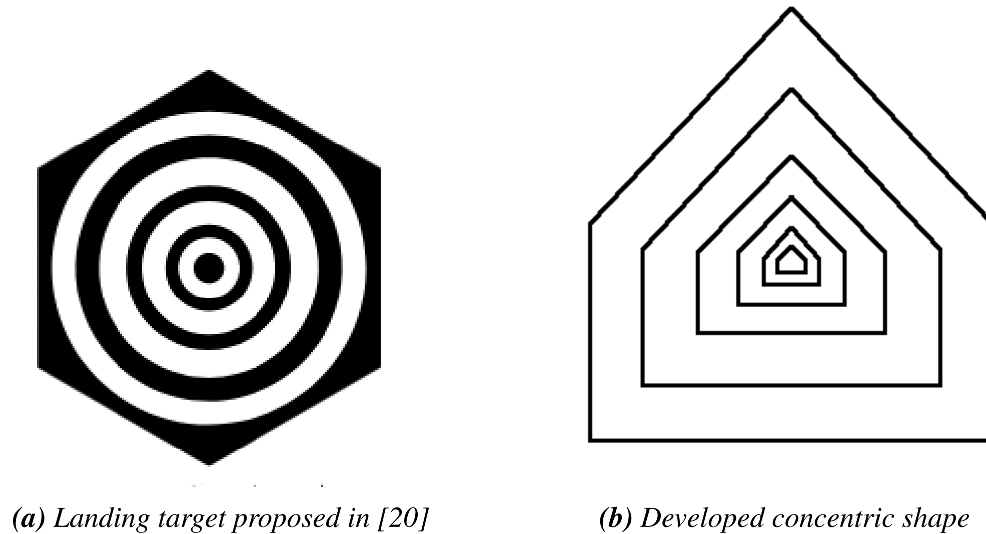
(a) *Top view* ***(b)*** *View when descending*

4.1.2 Concentric shapes

To eliminate the issues encountered with the letter x shape, a more complex shape should be utilised. One possible solution could be deploying a shape proposed in Figure 11a [20]. This work utilizes concentric shapes as providing relevant information about a drone's location. Lange et al. propose to use white rings on a black background where the white circles can be distinctly identified. The circles can be identified based on the radius ratio of the black circles what border the white circle. This ratio value can be used to deduce at which altitude the drone is. [20]

In the experiments the shape could be recognized from a distance of 2 m when the outermost circles diameter was 43 cm [20]. The proposed shape could be expanded to suit recognition from a greater distance by adding an adequate number of outer circles. If more circles need to be added, the actual landing pad area starts to increase as well.

This shape concept can be further developed to get rid some of the limitations that the letter x shape has. These limitations were that the orientation cannot be deduced from the shape. Also the shape cannot be reproduced in a unique manner to support the multi-drone environment.



(a) *Landing target proposed in [20]* ***(b)*** *Developed concentric shape*

Figure 11. *Concentric shapes*

To eliminate the restriction of deducing a heading from the previously mentioned shape, a proposal would be to use a shape shown in Figure 11b. With this kind of shape the orientation can be deduced from the shape. The orientation that the drone needs to land in can be derived from the orientation of the upper part of the shape, i.e. the triangle part of the shape. For instance if the triangle's uppermost point faces upwards as in Figure 11b, the drone would orientate the frontside of the drone in the direction where the triangle is pointing to.

This shape also cannot naturally suite a multi-drone environment and would require an additional component that this shape could be uniquely identified. This starts raising concerns that a shape that is understandable to a human-being is not suitable for vision-based precision landing.

4.1.3 Fiducial markers

A possible solution to overcome the limitations of the previously mentioned shapes could be the use of fiducial markers. A common example of a fiducial marker is a ruler. These objects are placed as a point of reference or a measure and place in the field of view of an imaging system. This type of markers have been successfully utilised in vision-based landing in [5] and [22]. Examples of fiducial markers is presented in Figure 12.

Examples of these fiducial markers are the ARToolKitPlus, ARTag and BinARyID markers displayed in Figure 12. This type of markers is referred as "tags" [27]. These tags resemble

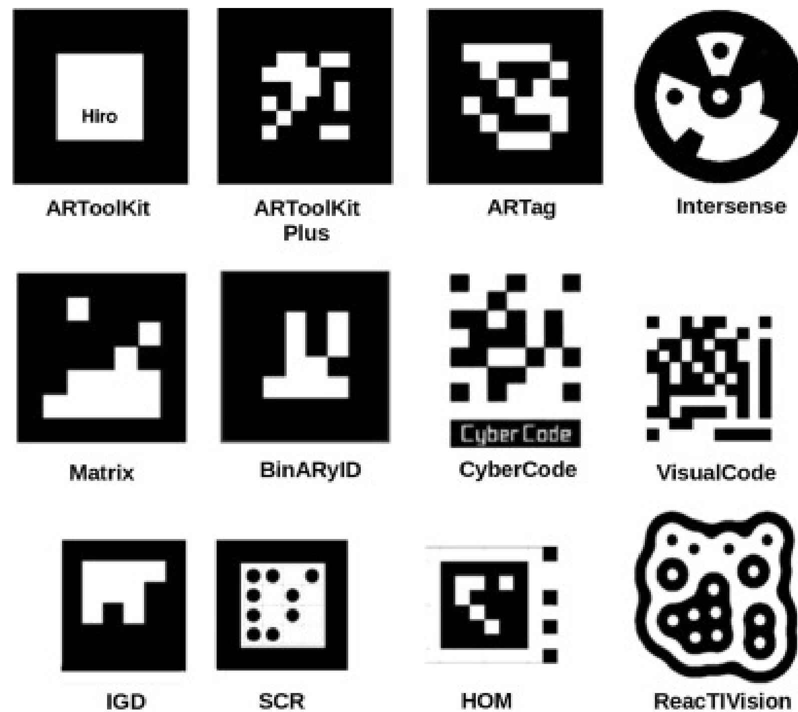


Figure 12. *Examples of fiducial markers. [14]*

2D barcodes such as QR codes. QR codes encode much more data compared to markers shown in Figure 12. QR codes can store up to 2953 bytes of data [48] but the tags used in this work store significantly less data.

The data capacity of the markers used in this work is in the range of tens of bits. The lower amount of pixels improve the registration robustness, especially increase the range of detection which is a key aspect in this work. [27]

These markers have been proven to be recognised from a distance that is suitable for this applications requirements [49]. The maximum distance a pattern must be recognised in this application is 5 m. In [49] Wang and Olson demonstrated that with a 0.167 m wide pattern, the image can be recognised from a distance of 7 m. The recognition distance obviously depends on the quality of the imaging sensor as well. This suits the need of recognising the pattern from a distance of 5 m. These markers seem promising for this work since with a small pattern a sufficient recognising distance is achieved.

The only weakness of this marker will be in the situation where the drone is under 1 m from the pattern. In this situation the pattern starts to be larger than the camera field of view and the marker cannot be recognised successfully any more. This would lead to the scenario were the landing accuracy would decrease because the UAV cannot be guided with a marker any more. For instance in [49] the minimum distance a pattern was recognised with a 0.167 m wide pattern was 0.6 m from the pattern.

Assuming that the imaging unit is placed on the drones body and not for instance on the landing gear of the drone. When a drone would contact the landing pad, the imaging

unit will be much more closer to the surface than 0.6 m. This will mean that the pattern would be unrecognisable for the last 0.6 meters of the landing procedure if a similar shape was used as in [49]. This distance would vary depending on the height of the landing gear of the drone. With shorter landing gear the imaging unit will be closer to the pattern and respectively with longer landing gear the imaging unit will be further away from the pattern.

The first method to resolve this issue is by calculating a landing point from a distance where the marker is still visible. At this point a precise calculation of the landing location can be done and then the UAV is commanded to land at this calculated position. This is how the UAV doesn't need to see the marker during the final approach and still can perform an accurate landing.

Another option is to use additional patterns on the landing pad and redefining the shape of the landing pad itself. The landing pad itself can be in a concave shape like a egg cup holder with a pattern at the bottom and additional patterns on the walls of the concave shape. An example of this is illustrated in Figure 13. This is a commercial product of the HiveUAV™ company [17].

Figure 13. *Aerial view of HiveUAV™ landing pad. Picture adapted from [18]*

Using a solution as presented in Figure 13 would require the drone to have one or more imaging units mounted perpendicular to the one facing down. The function of these additional imaging units is to recognise the markers on the walls of the concave shape.

A third option is to deploy multiple markers on the landing location with markers of different sizes. [3] The landing location would contain larger markers and smaller markers. The larger markers can be detected from a greater altitude than the smaller ones. As the UAV would descend closer to the ground the larger markers will disappear from the camera's field of view but the smaller will become visible thus the UAV can still location

the landing location. Using this kind of solution would require a large area thus increasing the overall size of the landing pad.

As the area of the marker can be a factor, one possible option could be to embed a marker within another marker. [51] This will provide the same function as using separate different size markers but in a smaller area.

A fiducial marker can provide an estimation of an altitude and a rotation of a marker relative to the imaging unit [14]. This means that the drone can rotate and descend towards the fiducial pattern when it is visible to the drone.

The fiducial pattern has the greatest advantage in the multi-drone environment compared to the rest of the presented shapes and patterns. This pattern encodes the information in the pattern itself in similar fashion than barcodes and QR codes. This enables that a marker can be uniquely identified with a unique value. This means that this kind of pattern could be utilised in a multi-drone environment.

4.2 Design conclusion

The letter x and the concentric shapes were discarded from further considerations since they do not fulfil the set requirements or extensive effort must be put in order to make a shape a viable option.

This is why the fiducial marker is used in this work. Since there are many different fiducial markers, one will be chosen for further development. In this work the used fiducial marker is a squared fiducial marker and the ArUco variant of the squared patterns.

The reason for choosing the squared fiducial marker and the ArUco tag is that these markers have been proven to be suitable in the application scenario of this work [1]. The ArUco variant is chosen since it has a small but adequate amount of unique identifiers. Smaller amount of identifiers improved recognition robustness [27]. In addition these squared fiducial markers can provide an estimate of the UAV's altitude and orientation from a marker with a little processing. [14].

A summary how the different shapes fulfil the requirements is presented in the following tables.

Table 4. *X letter shape evaluation*

X letter shape	
Recognised from a distance of 5 m	Plausible depending on the marker size
The shape must guide the drone throughout the landing process	Possible
Orientation deduced from shape	Not possible
Scalable to support multiple drone landing	Possible with additional components

Table 5. *Concentric shape evaluation*

Concentric shapes	
Recognised from a distance of 5 m	Plausible depending on the marker size
The shape must guide the drone throughout the landing process	Possible
Orientation deduced from shape	Possible
Scalable to support multiple drone landing	Possible with additional components

Table 6. *Fiducial pattern evaluation*

Fiducial marker	
Recognised from a distance of 5 m	Plausible depending on the marker size
The shape must guide the drone throughout the landing process	Possible with additional components
Orientation deduced from shape	Possible
Scalable to support multiple drone landing	Possible

4.3 Display method

As the design of the pattern is concluded, only the displaying method of the marker needs to be chosen in order to satisfy all of the requirements in Table 3. This section will provide an solution to the item number 5 in the requirement list stated in Table 3. This section will only consider options only at a theoretical level since implementing other parts of this work consumed all the time reserved for this work.

In the upcoming part a comparison between three different visualising methods is presented. As stated in the introduction chapter the goal of this work is to propose a pattern that will function in varying environmental conditions. If the pattern cannot be recognised in different conditions it will severely limit the application scenarios where this kind of vision-based landing can be used.

For example if a pattern is printed on some surface, the ability to recognise this with a RGB camera is mostly restricted by the luminance of the pattern. Before commencing a mission with a drone it would be necessary to make sure that the brightness of the area where the landing pad is located stays mostly the same. If the brightness of the landing pad area would dramatically decrease during a mission, the drone wouldn't be able to land based on visual data.

The luminance of the pattern can decrease because of numerous reasons. The amount of sun light in the area can change due to the sun setting, also clouds might form in the area and decrease the amount of light in the area. Also the area where the drone is flying might be different e.g. an well lighted open area compared to an dark indoor environment.

To use a marker in changing lighting conditions a few different display methods can be utilised. The first is to use visible lighting to display a marker. The second is to use invisible light to the human eye such as infrared radiation (IR).

4.3.1 Passive material

A straightforward approach in visualising the selected marker would be printing it on some material e.g. paper like in [49]. This way of visualising a marker would be always suitable in environments that have a adequate and constant illumination e.g. a indoor area with constant lighting. This presentation would work in an outdoor environment as long as the sun is providing light in the area but as described earlier the luminance of an area might change rapidly.

Also paper is a fragile material and could break for numerous reasons. The paper can be torn apart by some actor in the landing pad environment or even the drone itself when the drone is landing on to the landing pad. Also paper is subject to environmental conditions such like moisture and would render the pattern useless. To make paper more robust it can be laminated so the paper can withstand more mechanical wear and tear. It is believed that this type of display method would not be adequate to survive environmental conditions. Displaying these markers on a printed surface seem to produce robust recognition and tracking.

Keeping in mind the goal of this work, this type of display method won't be applicable in environments that have low luminance. This is why displaying a marker can be discarded from further considerations.

4.3.2 Visible light

To overcome the constraints using a passive material such as paper for displaying a pattern, an active light display could be used. The marker could be displayed on a typical LCD or LED screen. This will enable to display a marker in a dark environment. In a environment where there is an increased amount of ambient light i.e. in a sunny environment screens usually become unreadable thus a marker cannot be recognised from a screen.

A possible solution to display a marker in both a dark and a bright environment is to incorporate both the passive and active display technique. The idea would be to form a backlit marker where a marker is formed on a surface i.e. tinted glass and then underneath this surface is an active light source. During bright conditions the passive material is visible and during dark conditions the active light source beneath the marker surface will shine through the surface and light up the white squares of a marker.

4.3.3 Invisible light

A third option could be the use of light sources invisible to a human eye such like infrared light. IR light sources can be detected with cameras that don't have an IR light filter (NOIR camera). In dark environments the detection is easy since there are no natural sources of IR light.

IR light detection in a sunny environment is challenging since the Sun emits IR light and this radiation will reflect from objects in an environment and will be registered by an IR camera. This can cause false identifications since the camera cannot differentiate between the light emitted from a IR light source or a reflected source. If IR lighting would be used at a landing location the reflections could render the landing location undetectable.

This is why countermeasures must be employed to reduce the amount of reflected light. In [51] it was proven that sanding a surface can drastically reduce the amount of light reflected.

A landing pad presented in [51] could be used in this work as well. This is a landing pad where a marker embeds another marker and is backlit by IR LEDs.

5. REALISING MACHINE VISION BASED CONTROL

This chapter focuses on presenting how the autonomous landing procedure was implemented for this work. During this chapter it is explained how the *PnP* problem presented in chapter 2.4 is solved with the marker chosen in chapter 4. Also the control logic implemented in the ROS controller node in chapter 3.2 is presented.

5.1 Marker detection

The following sections present the concepts that the OpenCV ArUco library [28] uses to find an marker in an image.

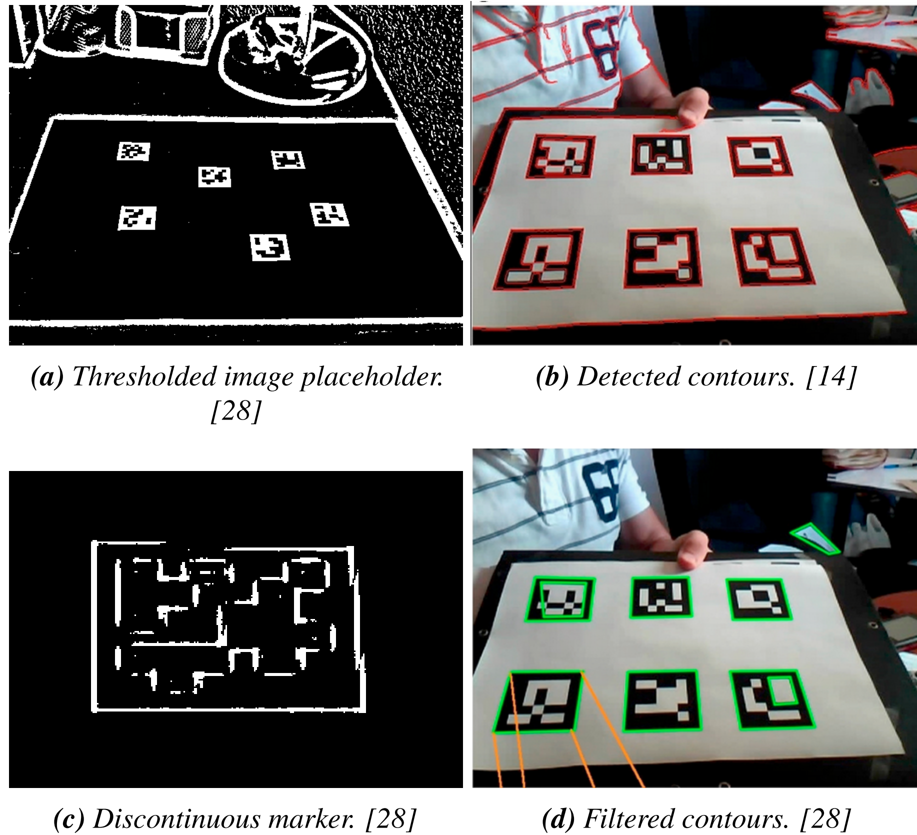
First a captured image is converted to a gray-scale image which is used in the upcoming steps of detecting a marker. The gray-scale image is processed with a adaptive threshold filter from which markers are detected in further steps of the identification process. A result of the usage of the adaptive thresholding is presented in Figure 14a. [14]

The thresholded image is processed with more filters in order to find laying ArUco markers in an image. Shape outlines also called as contours are extracted from the image with an algorithm from [44]. This algorithm extracts also many irrelevant contours from the image which aren't square markers thus further processing is required to discard the unnecessary outlines. The result of this step is presented in 14b and the detected outlines are bordered with red colour. [14]

Outline filtering should happen in early stages of processing the image and irrelevant outlines should be discarded. The irrelevant outlines will increase the computational load in latter parts of the processing tasks. There are different parameters that define how the image is filtered. Too large or too small thresholding values can cause a valid marker not to be recognised since the marker appears not to have a continuous borders as presented in Figure 14c. [28]

The AruCo library utilises the algorithm presented in [7] to detect and extract the shapes that resemble a shape that has four vertexes, e.g. a square. Extracted shapes that don't have four vertexes are discarded. There can also be squares in the image that are not AruCo markers. [14]

Squares are removed which are too big, too small or are too close to each other. This procedure should yield the most promising shapes that could possibly contain an ArUco marker. The result of this extracting phase is presented in Figure 14d. [14]



(*a*) *Thresholded image placeholder. [28]* (*b*) *Detected contours. [14]*

(*c*) *Discontinuous marker. [28]* (*d*) *Filtered contours. [28]*

Figure 14. *Aruco detection pipeline*

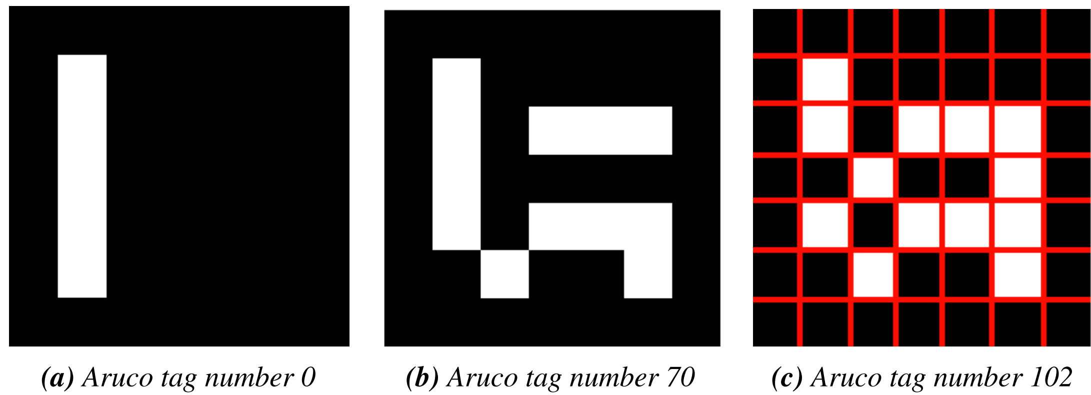
At this point a filtered image should contain only square shapes and the next phase is to determine if the shape contains an ArUco marker. First the perspective distortion is corrected and an image is processed to extract single black and white pixels. From these black and white pixels it can be deduced if the shape contains a valid marker. [28]

5.2 Aruco tag decoding

The basic ArUco library consist of tags that are divided in to a 7-by-7 matrix illustrated in Figure 15. The matrix is presented in Figure 15c. The first row and column and the last row and column form a border for all of the patterns. The data is encoded into the inner 5-by-5 matrix where the first, third and fifth columns are for parity bits and the second and fourth columns represent the data itself. A white square corresponds to a value of 1 and a black corresponds to the value 0. [9]

The maximum number we can encode with a 5-by-5 grid is 1024. This is because there is two data columns which each contain 5 data bits. This sums up to 10 data bits and the maximum number can be derived with the following equation:

$$2^{10} = 1024$$



***(a)** Aruco tag number 0* ***(b)** Aruco tag number 70* ***(c)** Aruco tag number 102*

***Figure 15.** Aruco tags*

As the white squares represent the value 1 and black squares represent the value 0 the following data representation can be formed from the aruco number 102 which is displayed in Figure 15c. In the following figure the corresponding colors have been converted to one or zero.

0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	1	0	1	1	1	0
0	0	1	0	0	1	0
0	1	0	1	1	1	0
0	0	1	0	0	1	0
0	0	0	0	0	0	0

***Figure 16.** Corresponding values to the color of the grid element*

From Figure 16 the value can be read. The inner 5-by-5 matrix contains the actual data, thus further explanation will regard the first element to be at the start of this 5-by-5 grid. The inner 5-by-5 matrix is denoted by the green square in Figure 16. The numeric identifier is encoded in to the second and fourth column. Starting indexing from number 1, the MSB (most-significant-bit) is located at index 1,2. The digits are read from left to right hand fashion [9]. When the digit on the fourth column has been red at a particular row, the reading is continued to the row below except when the reading is done at index 5,4.

With this information the value can be read. Starting from the MSB index 1,2. The first row produces the value of 00, the second 01, the third 10, the fourth 01 and the last and fifth row 10. When combined the 10 bit value is 0001100110. This equals to a decimal value of 102.

With the following program ArUco markers can be recognised from a video stream. [28]

```
1  # Get frames from camera
```

```
2 ret, orig_frame = self.cap.read()
3 # Image that is displayed after all processing
4 drawn_frame = orig_frame
5 # Convert image to grayscale for futher processing
6 gray = cv2.cvtColor(orig_frame, cv2.COLOR_BGR2GRAY)
7 # Detect markers from defined ArUco dictionary
8 corners, ids, _ = aruco.detectMarkers(
9         gray, self.aruco_dict, parameters=self.parameters)
```

Program 5.1. *Detection of ArUco markers*

In the program 5.1 lines from 1 to 6 are related to getting the frames from a imaging unit and converting the RGB frames to grayscale frames. The function *detectMarkers(...)* on line 8 and 9 extracts and identifies the ArUco markers from a grayscale image. The first parameter defines the image from which image an ArUco marker is searched from.

The second parameter defines which type of ArUco markers are being searched for from the input image. In this work the ArUco markers are searched from a pool called *ARUCO_ORIGINAL*. This pool contains the least amount of unique identification numbers but this amount is an adequate amount of unique ids for the scope of this work. The *ARUCO_ORIGINAL* marker pool contains 1024 unique ids. As mentioned in section 4.1.3 the lower amount of bits contained in a marker improves the registration robustness and also increases the range of recognition.

The last and third parameter defines the parameters that are used to detect a marker. These parameters contain e.g. the thresholding values that are used to extract contours from an image as mentioned previously in this chapter and presented in Figure 14. The ArUco library default values are used as parameters.

The return values are related to storing a detected marker identification value and the 2D position of each corner of a successfully identified marker. These points are used to solve the *PnP* problem. The last return value can be omitted from the program because this variable would store rejected squares that did not contain a valid binary code that is contained in the previously mentioned *ARUCO_ORIGINAL* pool. This variable is mostly used for debugging reasons and would cause additional computing costs if this variable would be present.

5.3 Pose estimation

After an ArUco marker has been successfully detected, the ArUco OpenCV module [45] can be used to determine a pose of a marker with respect to an imaging unit. The ArUco module is able to solve the P*n*P problem presented in section 2.4 in order to estimate the pose of a detected marker. In order to calculate the pose estimation the camera intrinsic parameters, distortion vector and the points in the 3D physical world and their corresponding points in pixel coordinates must be known.

The camera intrinsic parameters and the distortion coefficients could be extracted with the camera calibration tools. In this work the camera calibration algorithm [19] was used. The ArUco module utilises the four corners of the marker square as the 3D points. The 3D points of a marker can be determined by assuming that the marker is horizontally place at the origin of the world coordinate system. The physical size d of the marker must be known to define the 3D points. [45] The 3D points M_1, M_2, M_3 and M_4 of a marker are:

$$M_{1-4} = \begin{bmatrix} -d/2 & d/2 & d/2 & -d/2 \\ d/2 & d/2 & -d/2 & -d/2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The ArUco module will internally calculate the pixel coordinates on the image plane based on these 3D points. Now all the needed values have been defined in order to calculate the extrinsic camera parameters. The ArUco module will internally use the OpenCV function *SolvePnP()* to return two transformation vectors that can be used to calculate the extrinsic matrix. These transformation are 3-by-1 vectors and are called the translation and rotation transformation vectors which are presented in equation 5.1.

$$\text{translation matrix} = \begin{bmatrix} t_x & t_y & t_z \end{bmatrix}, \ \text{rotation matrix} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} \tag{5.1}$$

These transformation vectors can be acquired by calling the *estimatePoseSingleMarkers()* function from the ArUco module.

As the extrinsic parameters is a 4-by-3 matrix that contained both the translation and rotation it is obvious that operations need to be done in order to place these returned vectors into the matrix. The part that defines the rotation in the extrinsic matrix is a 3-by-3 matrix. The OpenCV library provides a function to obtain this 3-by-3 matrix by using the Rodrigues' rotation formula. This transformation formula provides an efficient way to compute a transformation matrix from a rotation vector representation. [4, p. 402].

Next the translation vector must be manipulated so it can be placed in the extrinsic matrix. This is a straightforward step since the returned vector from the ArUco library is a 3-by-1 vector and the extrinsic matrix requires a 1-by-3 vector. This transformation can be achieved by taking the transpose of the returned translation vector from the ArUco library. Finally, this 3-by-1 vector for translation can be placed in the corresponding part in the extrinsic matrix.

With the program 5.2 the marker can be identified and the pose can be estimated. In addition to this the program will publish this combined data to the ROS topic *homogeneous_coordinates*. This is the functionality of the image processing node presented in section 3.2. This data is going to be received by the control node of the system. The lines 1-13 in program 5.2 used instructions from [28] to extract the transformation vectors. Rest of the program was implemented by the writer.

```
1  def detect_marker(self):
2          try:
3              ret, orig_frame = self.cap.read()
4              if ret is True:
5                  gray = cv2.cvtColor(orig_frame, cv2.COLOR_BGR2GRAY)
6                  corners, ids, _ = aruco.detectMarkers(gray, self.
                       aruco_dict, parameters=self.parameters,
                       cameraMatrix=self.camera_matrix, distCoeff=self.
                       dist_coeffs)
7                  if ids.any() is not None and self.marker_to_detect
                       in ids:
8                          wanted_marker_index = np.where(ids == self.
                               marker_to_detect)
9                          #Get transform vectors
10                         rvec, tvec, _ = aruco.
                               estimatePoseSingleMarkers(
11                                    corners, 10, self.camera_matrix,
                                          self.dist_coeffs)
12                     #Pass trasnformation vectors for further
                           prcoessing
13                         self.update_location(str(self.
                               marker_to_detect), rvec[wanted_node_index
                               ], tvec[wanted_node_index])
14             else:
15                 pass
16 #thrown when two  same id's have been recognized
17         except AttributeError:
18             pass
19
20     def update_location(self, target_id, rvec, tvec):
21         try:
22                 #Transform 3 by 1 vector to 3 by 3 matrix with
                       Rodrigues' formula
23                 world_camera_rot, _ = cv2.Rodrigues(rvec)
24
25                 #Update homogeneous coordinates with rotation matrix
26                 self.current_location[0:3, 0:3] = world_camera_rot
27                 #Update homogeneous coordinates with transposed
                       translation transform vector
28                         self.current_location[0:3, 3:4] = tvec.T
29
30                 #Publish marker id and marker data to ROS topic
31                 msg = ros_numpy.msgify(custom_transform, self.
                       current_location)
32                 msg.aruco_id.data = target_id
33                 self.pub.publish(msg)
34
35         except ValueError:
36                 pass
```

Program 5.2. *Detection of ArUco markers*

In program 5.2 the lines from 2 to 7 identify the ArUco markers from an image. Lines 7 and 8 are used to check if a wanted marker is seen in an image and store the index of this marker in the *ids* vector to the variable *wanted_marker_index*. The transformation vectors are returned as a n-by-3 vector, where n is the amount of markers detected in an image. The transformation vectors of the wanted marker can be indexed with the value stored in variable *wanted_marker_index*.

These transformation vectors alongside the wanted marker id is passed on for further processing. With lines 23-29 the transform vectors are processed and then placed in the extrinsic matrix. The contents of the extrinsic matrix is transformed to quaternion representation and then broadcast to the *homogeneous_coordinates* topic.

5.4 System implementation

5.4.1 Simulator environment

Next the system functionality was developed and tested using a simulator environment before conducting experiments with actual UAV hardware. This is a safe and efficient way to test the system functionality before conducting real-life experiments. In this work the Gazebo simulation environment [10] is used. This simulation environment was chosen as it supports the other software components used in this work and has an active community that provided essential help.

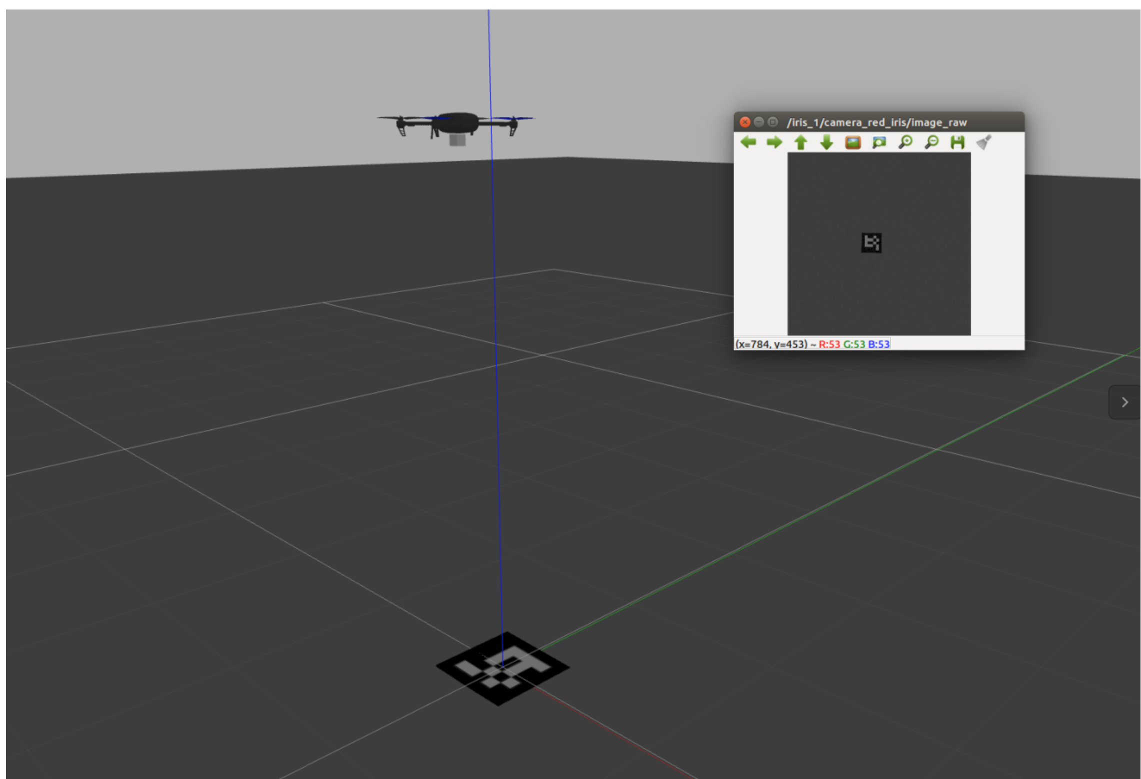


Figure 17. *Gazebo simulation setup*

The simulation setup is as follows. The UAV and an imaging unit (modelled as a gray square) attached to the UAV is spawned in a simulator world. Also an ArUco marker can be spawned in the same simulator world and the used simulator environment is presented in Figure 17. The video stream from the simulated camera unit is displayed in the smaller window of Figure 17.

Setting up the simulator environment followed the instructions presented in the PX4 documentation [39]. This documentation provided instructions how to launch the gazebo simulator environment with the ROS framework and how these components could interact with the flight controller software. How these components connect to each other is presented in the Figure 18.

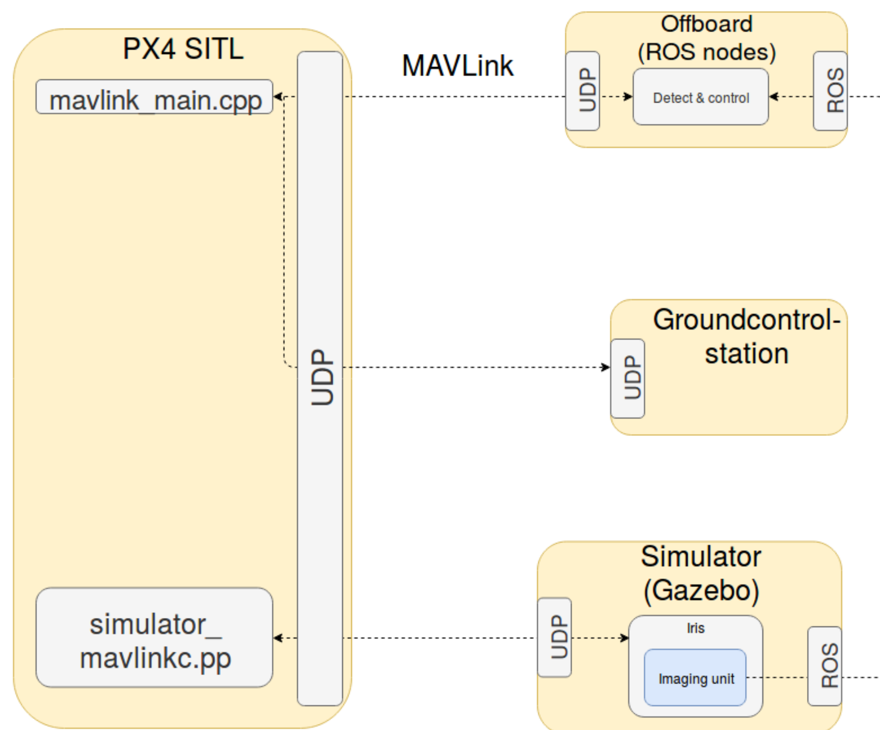


Figure 18. *Simulator environment. Adapted from [39]*

All of the required simulator software components presented in Figure 18 (PX4 flight controller software, ROS and Gazebo 7 simulator) can be installed by a setup script provided in the PX4 development environment setup documentation [31]. Once the setup script has finished installing the required software components, the actual simulator world could be set up for conducting the experiments.

After the software components have been installed the simulator can be launched. By using a launch file provided by the PX4 software all simulated components can be launched from a single file. This launch file is provided in the PX4 software [32]. The gazebo simulator provided everything else built-in except the ArUco marker models.

An existing project [25] provided ArUco models for the gazebo simulator environment presented in Figure 18. These marker models worked after adding the marker model files to the gazebo model directory. This project provides models for the AruCo marker numbers from 0 to 7 from the *ARUCO_ORIGINAL* marker pool. With little modifications every AruCo marker in the *ARUCO_ORIGINAL* can be added to the simulator world e.g. in Figure 17 the decoded marker number is 102.

The launch script [32] also enables the data flow between the different software components presented in Figure 18. The actual PX4 flight controller software is simulated in the gazebo simulator (regular desktop) and it behaves exactly the same if the same PX4 flight controller software was running on an autopilot presented in the Figure 19.

Figure 19. *mRo Pixhawk Flight Controller (Pixhawk 1). [34]*

This kind of simulation environment is called a software-in-the-loop (SITL). As displayed in Figure 18 the simulated PX4 flight control software connects to an offboard control unit, a ground control station (GCS) and the simulator software. The goal is to simulate the hardware used in the real-world tests as closely as possible to reduce the workload when switching between environments.

The offboard control unit can be simulated by running the software processes presented in section 3.2 on the same hardware as the simulator. In the simulator environment the image processing process receives it's data from the simulated camera attached to an UAV in the simulator environment. The control node communicates with the same flight control software that would be running on an actual flight control unit.

It is also possible to communicate to the flight controller unit through a control program called ground control station. With the help of a GCS a user is able to control the UAV, read telemetry values from the UAV and set a variety of different parameters of the UAV. An example of a ground control station software is QGroundcontrol (QGC). QGroundcontrol is compatible with flight control software that use the MAVLink communication protocol such as PX4 that is used in this work and ArduPilot. [38]

The PX4 software didn't provide a suitable UAV with a imaging unit attached to it so small modification was needed to be done. To attach a camera to a UAV in the simulator environment, the contents of appendix A was needed to be added in the file that defines the simulated quadcopter model.

The technical specifications of the simulated camera can be set e.g. the camera matrix and the distortion coefficients. These parameters were set to match the specifications of a camera that is used in the real-world experiments. With similar content all the other simulated hardware specifications such as the inertial measurement unit (IMU) values can be set. In this work there was no need to define any specific hardware configuration so the values were left at default values.

To view the simulated camera feed a new window was launched as the smaller window in Figure 17. The gazebo simulator is able to to display the video feed but it was decided to view the video stream with the *image_view* package [26] which is used to display ROS image topics. The reasoning for this was to ensure that correct frames were transmitted over to the rest of the implemented system which used the ROS framework.

The launched simulator will contain a world that is presented in Figure 17 and has the communication enabled between the different components presented in Figure 18. With this setup the simulator experiments could be conducted.

5.4.2 Drone control

After initialising the simulator environment, the actual control algorithm could be implemented and verified. First it was studied how the UAV can be controlled from an external computing unit or the offboard controller. The available commands that could be sent from the offboard controller to the flight control unit is presented in [8].

During the development of the control algorithms it was noticed that some of these commands did not work. There was no apparent reason for this but it is believed that the mavros package converts the messages to a format what the flight control software doesn't understand.

According to the mavros documentation [8] there is different topics which can be used to control a UAV. During initial tests commands were published to the $/mavros/setpoint_raw$ topic to control the simulated UAV.

Sending messages to the $/mavros/setpoint_raw$ topic can be used to control a UAV position in a local or global coordinate system. Also the attitude can be controlled. It was decided to use the local coordinate system since a detected AruCo marker would produce the distance to the marker in a local coordinate system.

During experiments run in the simulator using control signals published to the */mavros/set point_raw* topic, it was noticed that the UAV would stop its motors and the UAV would fall

from the sky. Setting a desired position in a local coordinate system worked as intended and the UAV would move to a desired location. Once commands were sent that would affect the orientation of the drone, the drone would stop its motors and the UAV would fall from the sky. The desired attitude setpoint is published to the $/mavros/setpoint_raw/attitude$ topic. Different kind of messages were published to this topic but still the UAV would crash after so it was decided to use other topics to control the UAV.

The second option tested for controlling the drone was publishing control commands to the $/mavros/setpoint_position$ topic. This control command can be used to set a position and altitude setpoint in a local or global coordinate system. The local coordinate system was once again used. Using this control command seemed to work fine and the UAV responded to the given commands perfectly thus this command signal was decided to be used for controlling the UAV.

Next the control algorithm of the UAV was implemented. The control algorithm is presented in Figure 20. This functionality is implemented in the control ROS node presented in section 3.2. The implemented control logic is split into two phases presented in Figure 21. First the UAV descends to a position near the marker and after this initial descend the UAV will make its final descend on to a marker.

In the first landing phase depicted as the number 1 in Figure 21 an UAV is flying in an area and is trying to locate a valid marker from it. Once a valid marker is recognised, the UAV will stop its movement and start to hover at its current location. At this position the control process will start accepting marker pose values from the image processing node. After the process has received enough marker data from the image processing algorithm the UAV orientation is calculated from the gathered marker data values and the current orientation of the UAV.

In this work the current vehicle state (orientation and position) is defined with the sensor data from the flight controller itself and not visual data. In this work the desired orientation is in the same direction as the positive y-axis of the marker. This orientation is denoted as the zero orientation in the marker coordinate system.

If the orientation of the UAV is greater than 10 degrees based on the orientation estimate from the marker data, the UAV will calculate a new orientation setpoint that will turn the UAV ideally to the zero orientation. This value is calculated based on the current UAV orientation (based on the flight controller data) and the estimated orientation of the UAV from a marker. The orientation setpoint is calculated by adding the orientation estimate from the current UAV orientation. Then the UAV is set to turn to this calculated value.

After the UAV has turned to the calculated orientation the algorithm calculates the position of the marker. The desired position is calculated based on the current position of the UAV and the estimated position offset from a marker center. The offset value is added to the current position of the UAV. The new position setpoint is ideally perpendicular to the

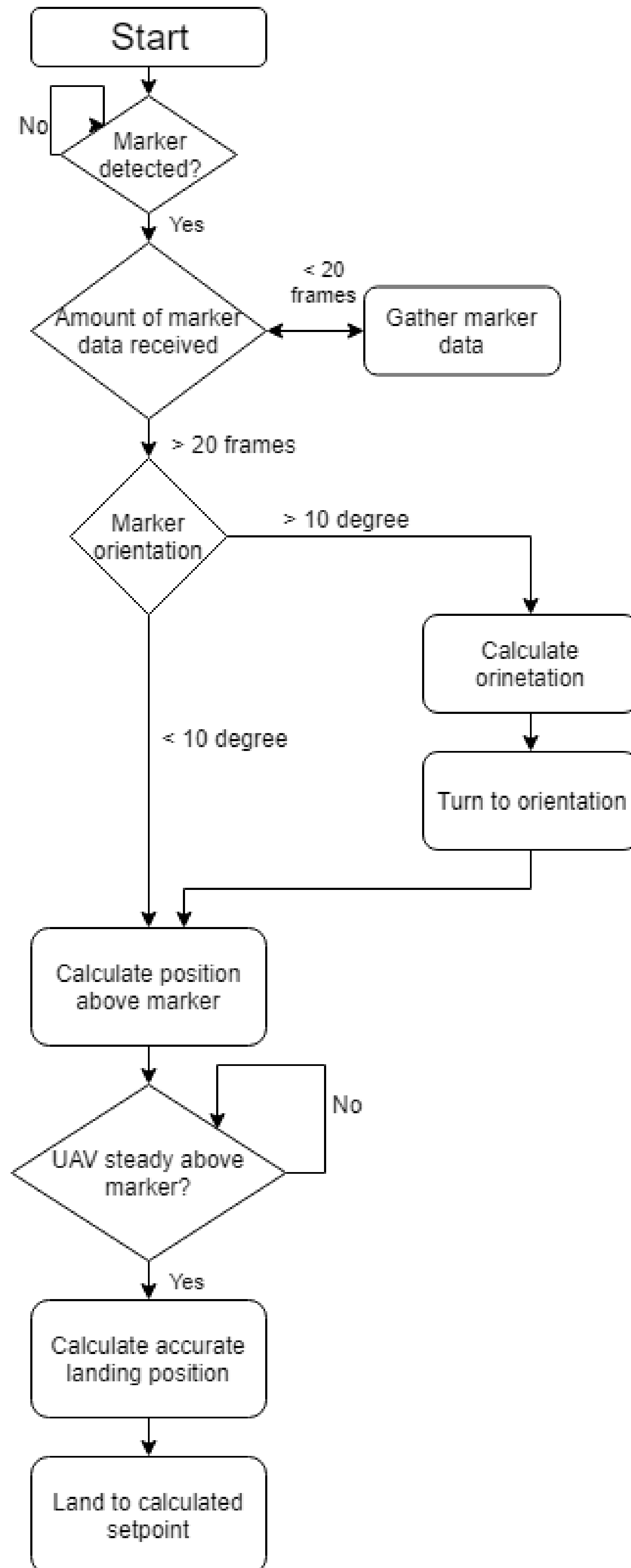


Figure 20. *Desired landing procedure*

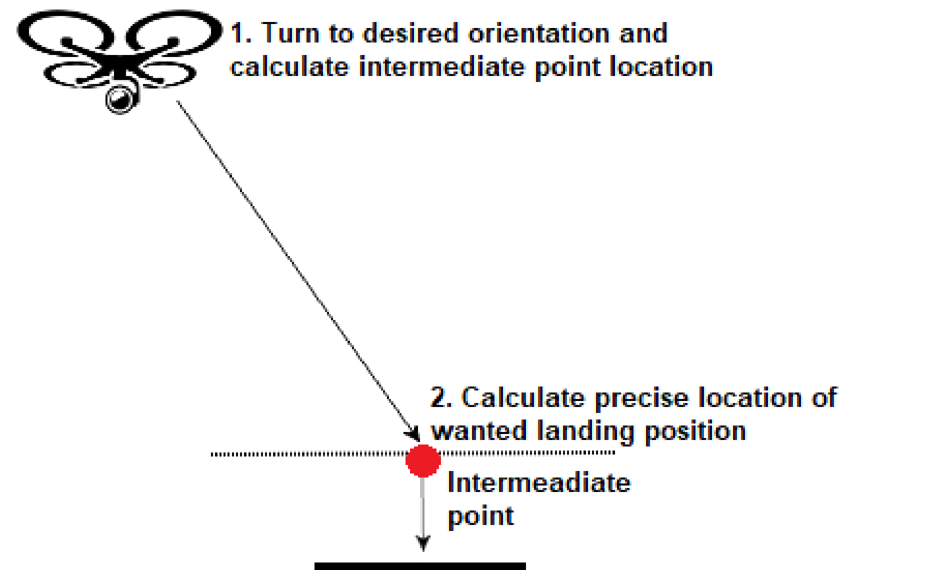


Figure 21. Implemented UAV landing scheme

marker center and 50 cm above the marker. After the position setpoint has been calculated the UAV descends to the calculated setpoint. This point is the intermediate point in Figure 21. Now the UAV has completed its first phase of the landing procedure.

In the second phase of the landing process the UAV will complete its final descend. Before beginning the final approach the UAV is wanted to be as stationary above a marker as possible and to be at a maximum distance of 50 cm from the marker. This is because it was noticed that if the UAV is in motion when the UAV starts its final descend towards a marker the landing accuracy decreases. Also at a shorter distance from the marker the estimation accuracy increases and the landing location might have some obstacles surrounding it so it would be risky to directly descend to a landing spot.

After these conditions have been met the algorithm calculates the final landing position similarly as the position in the first phase and is set to land to the calculated values.

6. EXPERIMENTS AND ANALYSIS

6.1 Marker detection

Before starting any other work the functionality of the marker detection algorithms was tested. If the used algorithms don't give a accurate pose estimation the rest of the work would be futile because the work is based on machine vision based localisation. Due to the other work conducted in this field, there was high confidence that the detection algorithms would work.

Verifying the functionality of the image processing algorithms could be easily tested. This setup only requires a printed marker, a workstation, a basic web camera such as one presented in Figure 22 connected to the workstation. This corresponds to the section presented in Figure 23 of the software architecture presented in chapter 3.2.

Figure 22. *Logitech C920 HD Pro Webcam. [24]*

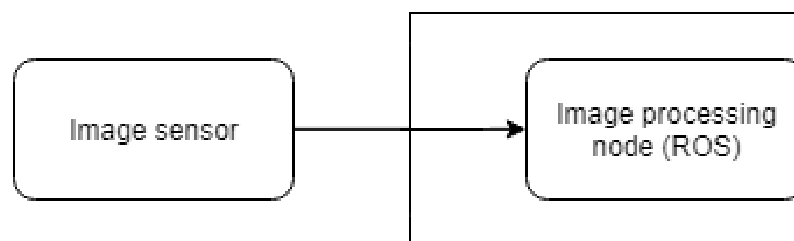


Figure 23. *Initial aruco detection setup*

The test scheme depicted by Figure 24 was used to evaluate the accuracy of the pose estimation of a detected marker. For the tests conducted the same image sensor was used that was going to be used in the real-world experiments. To estimate the pose of a marker the program 5.1 was used. This program required that the camera matrix, camera distortion coefficients and the physical marker size was known. The marker size in the experiments was 10 cm. The camera matrix and distortion coefficients were defined by using the camera calibration algorithm in [19]. The camera calibration yielded the following camera

matrix K and the distortion coefficient vector D where the values are rounded to the nearest hundredths:

$$K = \begin{bmatrix} 638.20 & 0.00 & 313.13 \\ 0.00 & 640.21 & 250.21 \\ 0.00 & 0.00 & 1.00 \end{bmatrix}, D = \begin{bmatrix} -0.13 & 1.53 & -0.01 & -0.01 & -3.65 \end{bmatrix}$$

The tests scheme depicted by Figure 24 has a total of five measuring points *P1-P5*. Points *P1-P5* were used to measure how accurately the distance and the yaw angle be estimated from a detected marker.

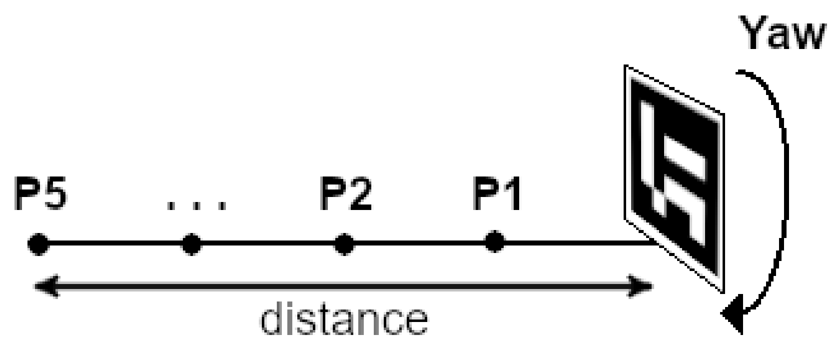


Figure 24. *Initial aruco detection setup*

The results of the conducted experiments are presented in Table 7. From the results in Table 7 it can be seen that estimated orientation matches the actual orientation of a marker very well and the error is 8 degrees at maximum.

Table 7. *Simulation results*

Measurement point	Distance (cm) and yaw (°) (estimated)	Distance (cm) and yaw (°) (measured)
1	51 0	50 0
2	105 91	100 90
3	213 172	200 180
4	327 272	300 270
5	458 0	400 0

Surprisingly this was not measured from the furthest location but from a distance of 3 m and the other estimations are accurate. This raises the thought that either the camera or

the marker had been tilted but overall the estimation is accurate. On the other hand the distance estimation error is relative to the distance between the camera and the marker. At 50 cm distance there is practically no error and the error is about 5 % until the distance grows over 2 m. At the maximum distance measured at 4,5 m the error was slightly under 15 %.

6.2 Simulation experiments

The functionality of the system was tested with the following experiment both in the simulator and a real world environment. The UAV is flown according to the trajectories presented in Figure 25 to evaluate the functionality of the developed system. This simulates the situation where the UAV is searching a marker in an area.

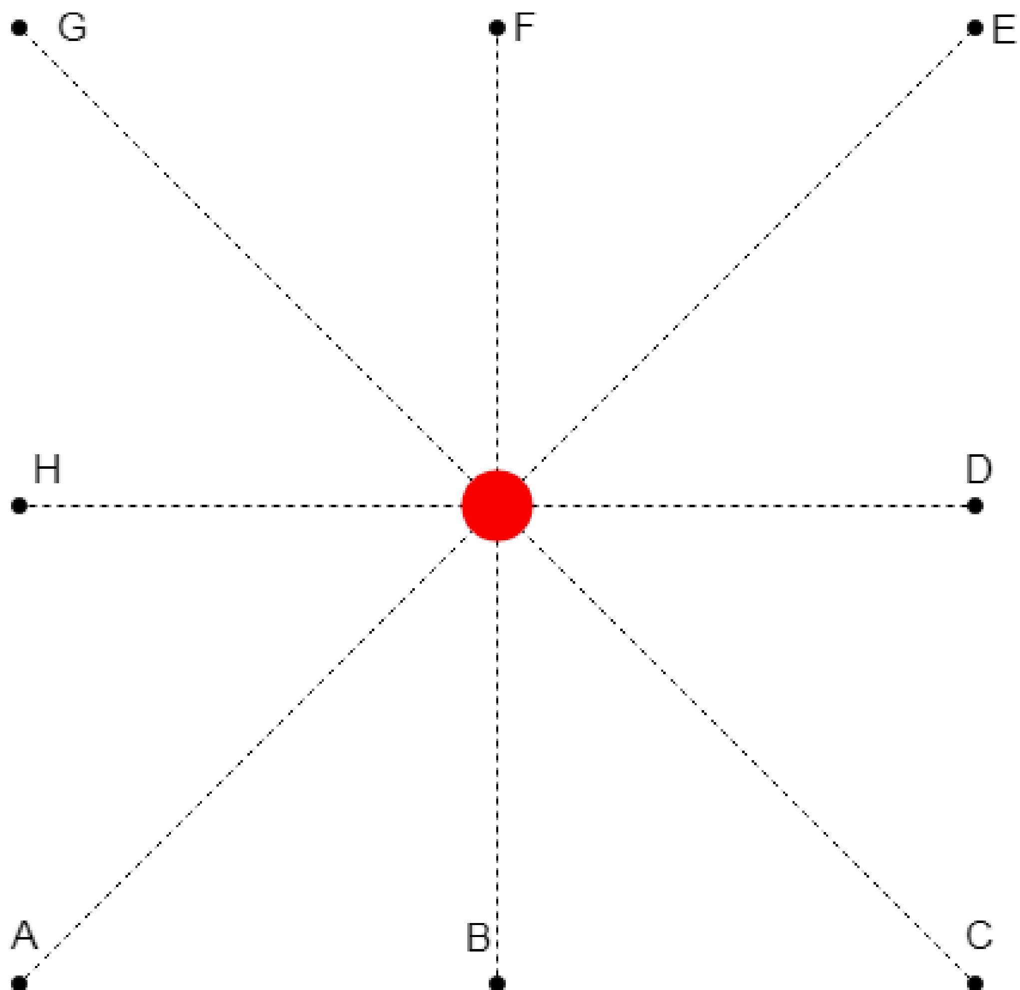


Figure 25. *Paths used for evaluating system functionality*

The used testing plan consists of six possible flight paths. The points denoted A-H in Figure 25 can be used both as a starting and an end point for a flight path. The paths used in the experiments were the paths between points AE, BF, CG, DH, EA, FB, GC, and HD. Every path is flown four times.

Gazebo provides location and orientation information about all objects in the simulator environment. Gazebo is able to publish the pose information to corresponding ROS topics. A ROS package called *rqt_plot* [43] is able subscribe to these topics to which the simulator is publishing data to. The *rqt_plot* package can be used to display and log the simulator data. This package was used to convert the flight data from the simulator environment to comma-separated value (csv) files for analysing the flight data.

The accuracy of the pose estimate given by the implemented algorithm is evaluated. This pose estimation is compared to the actual position where the UAV lands. This will tell how accurately the flight controller is able to land at given coordinates. The calculated setpoint is broadcasted to a ROS topic and can be logged similarly as the data from the simulator environment with the *rqt_plot* software.

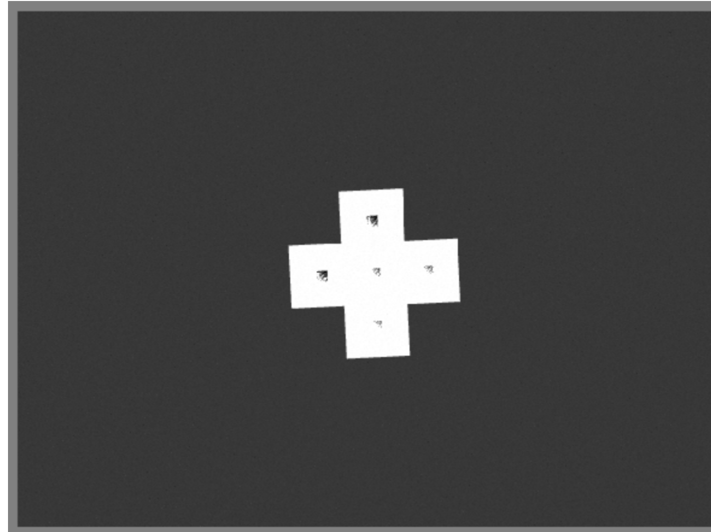
With the gazebo simulator a user is able to set the camera parameters in the simulator world. The set specifications were the image capture rate, the image size, the camera matrix and the camera distortion coefficients. The simulated camera specifications were set to match the camera used in the real world experiments. The image capture rate was set to 30 frames per second (fps). The image size was VGA resolution (680 x 480 pixels). The camera matrix and the distortion coefficient were set according to the values calculated in section 6.1.

During the simulation experiments it was noticed that the simulator environment was unable to render objects as the distance from an object grew. This meant that the simulated camera couldn't detect a marker. For this reason the simulated flights had to be done at a lower altitude than originally planned which was 5 m. Intuitively thinking increasing the marker size would improve the marker detection distance.

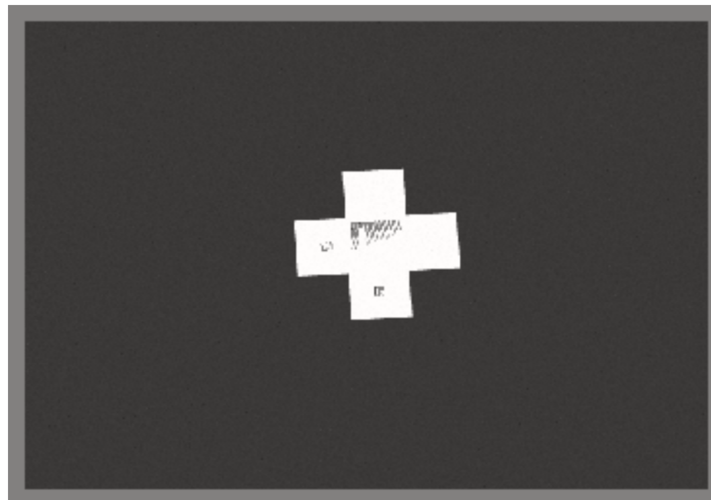
Increasing the size of the marker did not improve the marker quality as shown in Figures 26a and 26b. In Figure 26a a 10 cm marker was used and a 50 centimeter marker was used in Figure 26b. In both images the markers are partially visible thus not detectable. These result are contradictory to the results presented in [16] where the markers could be detected at a distance of 50 m in a simulator environment. This leads to thinking that this issue was caused by the hardware used for the simulations. In the simulation experiments the initial altitude was set to be 2 m from which the UAV would start its landing procedure.

To consider a landing successful the UAV must land within 15 cm from a desired markers centre. Also the UAV must land with a maximum of 10 degree deviation from the marker positive y-axis. To verify the system functionality a total of 32 test flights were concluded in the simulator environment. The success rate of these tests was ~91%. The results of the conducted experiments are presented in tables 8 and 9.

Table 8 presents the results of the accuracy of the landing procedure in the simulator environment. In Table 8 the UAV's final pose compared to the marker is presented. The position error is calculated based on the UAV positon compared to the marker center. The



(a) *10 cm marker.*



(b) *50 cm marker.*

Figure 26. *Rendering issue in Gazebo*

orientation error is calculated based on the UAV orientation compared to the positive y-axis of the marker. The z-axis distance was not shown in this table since the UAV z-coordinate is always zero when the UAV lands on the ground.

From Table 8 it can be seen that 29 out of the 32 tests stay within 15 centimetres in both x and y-axis relative to the marker center. The average deviation from the marker center in the x-axis was 4.82 cm and in the y-axis 7.45 cm. The error of the drone orientation relative to the markers y-axis was practically non-existent and in every measurement the error was smaller that 0.4 degrees.

Table 9 presents the estimated landing location by the program 5.1. From Table 9 the performance of the landing location algorithm can be analysed and as well the performance of the flight controller itself. The data from Table 9 is visualised in Figure 27. In Figure 27 the character "+" represents the actual center of the landing location. The character

"o" represents the estimated landing location and the character "x" represents the location where the UAV actually landed.

From Table 9 and Figure 27 it can be seen that the estimated position of the landing location is more accurate than the actual landing process. This kind of result was expected since it is more complicated to perform a landing process than estimating a location. The results show that the flight controller software is not able to land precisely to the coordinates that it has been given.

The algorithm that calculates the landing location has a average horizontal error of 2.52 cm and vertical error of 1.06 cm from the marker center. The average error that the flight controller has from the calculated marker center is 5.26 cm in the horizontal direction and 7.14 cm in the vertical direction. In [5] where the error in the x-axis was 0.37 m and 0.28 in the y-axis and in [42] the reported position error was 0.4 m. Compared to these results the system functioned extremely well.

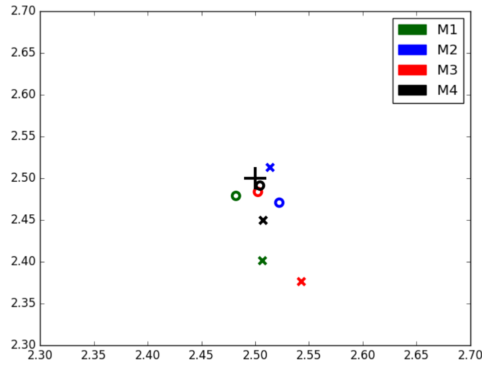
The results show that the application accomplished the set requirements and the system can be validated with real hardware.

Table 8. Simulation results

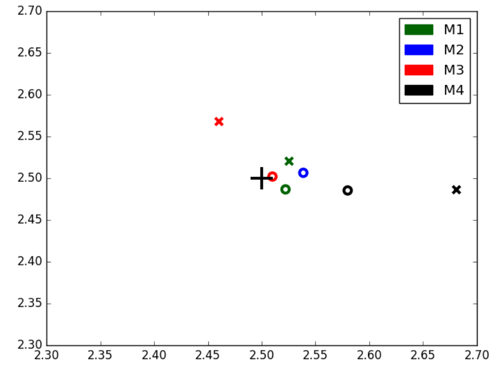
Route	Measurement	Position accuracy x axis (cm)	Position accuracy y axis (cm)	Orientation accuracy (deg)
AE				
	1	0.61	9.85	0.01
	2	1.33	1.30	0.40
	3	4.28	12.34	0.02
	4	0.70	5.03	0.02
BF				
	1	2.53	2.10	0.03
	2	14.49	12.83	0.01
	3	4.00	6.86	0.02
	4	8.05	1.34	0.01
CG				
	1	0.25	6.46	0.01
	2	7.59	0.70	0.04
	3	6.48	6.92	0.03
	4	2.51	3.17	0.01
DH				
	1	5.25	3.71	0.01
	2	8.32	11.75	0.01
	3	1.73	6.89	0.01
	4	0.45	18.26	0.03
EA				
	1	3.04	0.58	0.03
	2	3.04	0.58	0.03
	3	1.60	1.79	0.01
	4	5.46	14.03	0.05
FB				
	1	4.58	5.96	0.03
	1	1.02	14.42	0.04
	3	8.06	15.17	0.00
	4	10.65	7.80	0.01
GC				
	1	3.76	0.03	0.01
	2	7.58	0.71	0.04
	3	2.56	14.97	0.02
	4	2.99	14.63	0.04
HD				
	1	10.31	0.42	0.01
	2	0.86	8.78	0.01
	3	4.00	3.06	0.02
	4	1.57	1.70	0.01
Average		4.82	7.45	0.03

Table 9. Simulation results

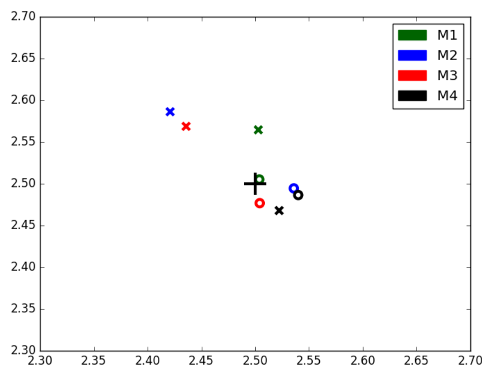
Route	Measurement	Set landing location local coordinates (m)	Actual landing location coordinates (m)	Difference (cm)
AE				
	1	2.48, 2.48	2.51, 2.40	2.40, 7.74
	2	2.52, 2.47	2.51, 2.51	0.91, 4.22
	3	2.50, 2.48	2.54, 2.38	4.03, 10.70
	4	2.50, 2.49	2.51, 2.41	0.26, 4.15
BF				
	1	2.52, 2.49	2.53, 2.52	0.32, 3.44
	2	2.54, 2.51	2.64, 2.63	10.62, 12.18
	3	2.51, 2.50	2.46, 2.59	5.00, 6.66
	4	2.58, 2.49	2.68, 2.49	10.06, 0.12
CG				
	1	2.50, 2.51	2.50, 2.56	0.13, 5.95
	2	2.54, 2.49	2.42, 2.59	11.55, 9.24
	3	2.50, 2.48	2.44, 2.57	6.90, 9.24
	4	2.54, 2.49	2.52, 2.47	1.79, 1.82
DH				
	1	2.54, 2.50	2.48, 2.46	8.94, 3.75
	2	2.51, 2.49	2.58, 2.62	7.82, 12.47
	3	2.45, 2.50	2.52, 2.57	6.58, 6.67
	4	2.47, 2.55	2.50, 2.68	2.14, 13.22
EA				
	1	2.52, 2.50	2.53, 2.51	5.78, 16.00
	2	2.52, 2.50	2.53, 2.51	1.03, 0.22
	3	2.55, 2.50	2.48, 2.52	6.75, 1.77
	4	2.50, 2.51	2.55, 2.36	5.76, 14.65
FB				
	1	2.49, 2.52	2.54, 2.56	5.46, 4.43
	1	2.51, 2.52	2.49, 2.64	1.97, 12.75
	3	2.49, 2.49	2.58, 2.65	8.60, 15.87
	4	2.51, 2.50	2.39, 2.58	11.40, 7.54
GC				
	1	2.61, 2.51	2.46, 2.50	14.84, 0.91
	2	2.53, 2.50	2.58, 2.49	4.89, 1.01
	3	2.55, 2.51	2.56, 2.65	0.88, 14.02
	4	2.52, 2.50	2.47, 2.65	4.74, 14.73
HD				
	1	2.51, 2.50	2.40, 2.50	11.19, 0.20
	2	2.51, 2.50	2.51, 2.41	0.60, 8.28
	3	2.48, 2.49	2.46, 2.53	2.37, 4.44
	4	2.45, 2.48	2.48, 2.48	3.16, 0.04
Average				5.26, 7.14



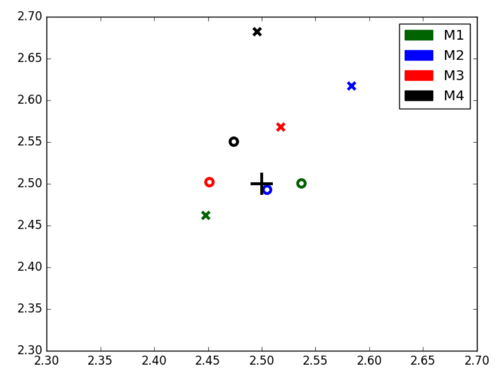
(a) Route AE



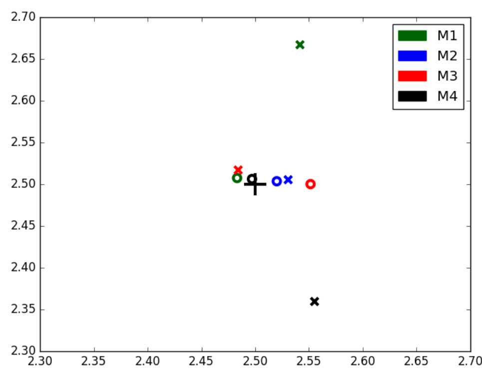
(b) Route BF



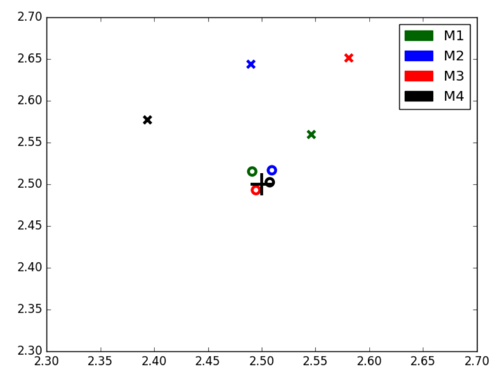
(c) Route CG



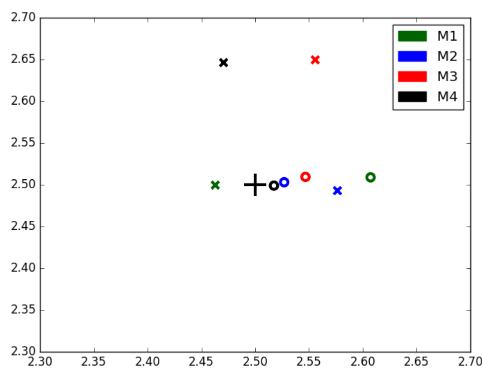
(d) Route DH



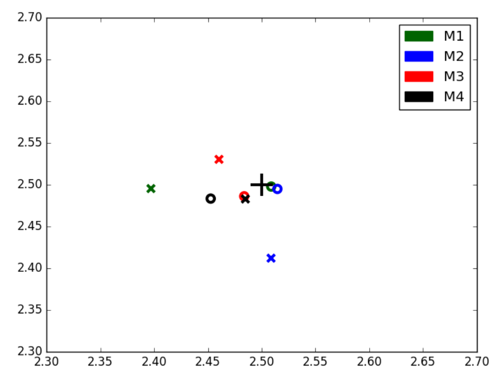
(e) Route EA



(f) Route FB



(g) Route GC



(h) Route HD

Figure 27. *Landing accuracy*

6.3 Flight hardware

The hardware setup was built according to the hardware block diagram presented in chapter 3. The corresponding components between the used hardware and the block diagram in chapter 3 is presented in Table 10. After the components were mounted and connected together the field experiments were conducted with the UAV presented in Figure 28.

Table 10. *Used hardware components*

Component	Model
Airframe	DJI F450 + Landing gear
Flight controller unit	Pixhawk 4
Flight control software	PX4
Radio link	Futaba R7008SB
Telemetry radio	Holybro Telemetry Radio
Motors	DJI 2313E
ESC	DJI 430 Lite
GPS	Holybro Pixhawk 4 GPS Module
Offboard controller	Rasberry Pi 2 B
Camera	HD Pro Webcam Logitech C920

Figure 28. *UAV used for field experiments*

The software processes presented in chapter 3 were running on the offboard controller. The offboard controller, rasberry pi (RPi) must have a unix based operating system installed because the first ROS framework version doesn't work on any other operating systems. In this work the Ubuntu Mate operating system was used because the Ubuntu MATE is an officially supported OS for the RPi and the Ubuntu operating system environment was familiar. Although the RPi doesn't provide much processing power while operating four

processing cores operating at 0.9 MHz, it was adequate for the needs of this work. With all the processes being executed the CPU utilization of the RPi was the following Figure 29.

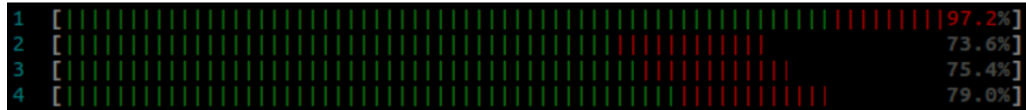


Figure 29. *RPi CPU utilization*

When assembling the UAV the problem arose how the RPi can be powered when the UAV was in flight. The RPi couldn't be connected to a normal wall socket. When all the software processes and the camera was running the measured current draw was about 700 mA. The RPi requires an operating voltage of 5 V. A commercially available power bank was used to provide power to the RPi. This power bank is able to provide a current draw 2 A so if there are current surges the power bank can still provide the required current for the RPi. This power bank had a suitable size that it could be attached under UAV the air frame.

6.4 Flight results

The same test plan was going to be used as in the simulator environment. During the flight tests it was noticed that the drone flight was much more unstable as in the simulator environment. With the same success criteria for the experiments as in the simulator environment, the overall success rate of the landing procedure was 0 %. Despite the failure rate the UAV was able to turn to a desired orientation within the given error limits. Acquiring the measurement data was difficult and only five measurements were able to be conducted. The main reason for difficulty acquiring measurement data was the unstable and unpredictable flight of the drone. It was noticed that the drone control was erroneously implemented using GPS data and according to [2] this data can fluctuate in the meter range which was noticed during the experiments.

Improvements have been made to make satellite-based positioning more accurate. In this work the real-time kinematics (RTK) positioning was used to improve the accuracy and stability of the GPS signal. The RTK positioning is based on correcting GPS data with an extra reference station [53]. This has been proven to provide position accuracy of centimetres. This sounded as a possible solution to fix the issues encountered. The pixhawk flight controller supports use of RTK devices and thus would be a quick solution as the time for implementing the project was ending.

The drone flight was much more stable with the corrected GPS data than with the uncorrected data. During new flight tests the GPS data was still inaccurate and fluctuating for a accurate landing procedure. During the tests the UAV would drift many meters in all directions although the UAV was commanded to hover in place. The main reason for this behaviour was that the GPS correction data was not the optimal one denoted as a RTK fix. The GPS correction data was limited to a RTK float.

This resulted that the drone movement was still unstable and unpredictable. The results clearly show that the system did not work with the current setup and the time for implementing the system had ran out so further improvements could not be made. At this point the tests had to be stopped prematurely. The experiment results conducted are displayed in Table 11. Only the orientation could be acquired and these results were visually estimated. The UAV could not land automatically so the UAV must be landed manually. This could introduce some error in the orientation but the UAV was landed as smoothly as possible not to turn the UAV while landing.

Table 11. *Flight test results*

Measurement	Position accuracy x axis (cm)	Position accuracy y axis (cm)	Orientation accuracy (deg)
1	Fail	Fail	5
2	Fail	Fail	5
3	Fail	Fail	10
4	Fail	Fail	10
5	Fail	Fail	5

7. CONCLUSIONS

This thesis presented one possible solution how machine vision can be used on an UAV to achieve an autonomous landing procedure. This work covered all the topics from choosing a object that is to be placed at a desired landing location, implementing a landing algorithm which was first tested in a simulator environment and then used with an actual UAV. It was decided to use existing components as much as possible and it was discovered that there are plenty of working and actively maintained open-source tools in the UAV industry. It required much work to get different systems working together but eventually a functioning system was developed.

The autonomous landing procedure relies on recognising a desired object at the UAVs landing position. From this object the UAV can deduce its distance and orientation from a desired landing location. In this work the set requirements were that this object must be recognised from a distance of 5 m, it must be visible throughout the landing process, it must provide information in which orientation the UAV is compared to the marker and the object must provide unique landing location identifiers in a situation where multiple UAVs are landing simultaneously. Display methods of this object were only considered on at theoretical level.

The relevant problem of estimating a pose of a camera respect to the desired landing location of a UAV is called the P*n*P problem. This problem stated how known 3D points of the physical world and their corresponding pixel coordinates can be used in estimating the pose of a camera with respect to a recognised object. In this work a QR code like pattern called the AruCo marker was used. The four corners of the marker were used to estimate the translation and orientation of the camera with respect to the marker. The marker encoded a 10-bit binary number which can be used to uniquely identifying a marker if multiple UAVs are used.

The open source computer vision library provided modules that could process the image data including a recognised marker. During initial experiments the translation and orientation estimate was really accurate. The orientation estimation was really accurate and did not practically have any error in the estimation although the distance was grown to 5 m. The distance estimation was very accurate when the distance from a marker is 50 cm and there is practically no error. When the distance was grown the error didn't increase dramatically and the maximum error at 4.5 m distance was 15 %.

After deciding what kind of object to use at the landing location the actual landing algorithm was implemented. During this work an open source simulation environment was used to implement and verify the system functionality. During later phases of this work it

was noticed that this was a wrong way of implementing the system functionality. With the simulator environment all the system components could be simulated which were being on an actual UAV. Only the imaging unit would change between the simulator and the real-world environment. The flight controller software (PX4) and the implemented software (image processing and control) stay the same in both environments. This is how redundant workload can be decreased.

The simulator experiments were promising as the overall success rate was 90 % of the landing experiments. To consider a landing successful it was decided that the UAV must land at a distance of 15 cm of the marker center and the orientation cannot deviate any more than 10 degrees from a desired orientation. The UAV could turn to the set orientation very well as the average error was practically 0 degrees. The algorithm that calculated the landing location was accurate and it could estimate the marker center location with a maximum of 2 cm error from the actual marker center.

It was noticed that the flight controller software was not able to land exactly at the calculated point. The average error from the marker center in the x-axis was about 5 cm and 7 cm in the y-axis. Compared to other studies in the field [6] [5] simulation results were promising. In [6] where the landing accuracy was tested only in a simulator environment the landing accuracy was in the range of 5 cm. In [5] the landing accuracy in a real-world environment was 0.37 cm in the vertical axis and 0.28 in the lateral axis.

Then the same system was implemented with actual hardware. The drone movement was unstable and unpredictable. The resulting experiment success rate with an actual UAV was 0 %. The unstable movement of the UAV was caused by the use of GPS data in the control algorithms and it was only realised after starting the flight tests that the implemented control logic was solely based on this fluctuating data. The control program implemented uses GPS data in two sections. The first use of GPS data is where marker offset is added to the current position of the UAV. The current position of the UAV is based on GPS data. The second use of GPS data is that the control signal from the program to the flight controller are coordinates in the UAVs local frame. Since the GPS data is not stable and will fluctuate during time, the position and the drone control will be wrong.

It would have required deeper studying of the drone control library to choose a proper message that was sent from the implemented control logic to the flight controller itself to be used in this kind of application. These signals were erroneously chosen to use defined points in the local frame of the UAV. This is because the writer did not have prior experience with UAV control software and the UAV seemed to function properly in the simulator with the GPS based control signals. It was realised too far in the development process that the system was implemented wrong. The messages should have been sent as acceleration or force setpoint to the UAV flight controller. This setpoint needs to be run through some control algorithm to provide a suitable acceleration or force setpoint.

Although GPS hardware was used that can achieve better accuracy the system did not

function properly. A possible reason could be that not enough satellites were not present at the area, according to [53] to acquire a RTK fix requires that 5 common antennas are connected to the extra reference station and to the antenna on the UAV. Another reason for this could be that the antenna on the UAV was placed next to other electronic components that were not insulated and could introduce interference to the GPS signal.

From this experience it was learned that system verification should start with the actual hardware simultaneously with the simulator environment. During this work it would have clearly shown that the control scheme to be implemented was doomed to fail and there would have been time to correct the mistakes. Even if the control logic would have been switched from the GPS based one, it is thought that a system will need data from different sensors to make a system robust i.e. GPS data could still be used in some parts and isn't completely discarded from the system.

REFERENCES

[1] R. Acuna, R.M. Carpio, V. Willert, Dynamic markers: Uav landing proof of concept, arXiv preprint arXiv:1709.04981, 2017.

[2] C. Adrados, I. Girard, J.P. Gendner, G. Janeau, Global positioning system (gps) location accuracy improvement due to selective availability removal, Comptes Rendus Biologies, Vol. 325, Iss. 2, 2002, pp. 165 – 170.

[3] O. Araar, N. Aouf, I. Vitanov, Vision based autonomous landing of multirotor uav on moving platform, Journal of Intelligent & Robotic Systems, Vol. 85, Iss. 2, 2017, pp. 369–384.

[4] G. Bradski, A. Kaehler, Learning OpenCV: Computer vision with the OpenCV library, " O'Reilly Media, Inc.", 2008, p. 555.

[5] C. Brommer, D. Malyuta, D. Hentzen, R. Brockers, Long-duration autonomy for small rotorcraft uas including recharging, arXiv preprint arXiv:1810.05683, 2018.

[6] T.G. Carreira, Quadcopter automatic landing on a docking station, Instituto Superior Técnico, 2013.

[7] D.H. Douglas, T.K. Peucker, Algorithms for the reduction of the number of points required to represent a digitized line or its caricature, Cartographica: The International Journal for Geographic Information and Geovisualization, Vol. 10, Iss. 2, 1973, pp. 112–122.

[8] V. Ermakov, mavros, 2018. Available (accessed on 19.2.2019): http://wiki.ros.org/mavros

[9] J. Ferrão, P. Dias, A.J. Neves, Detection of aruco markers using the quadrilateral sum conjuncture, in: International Conference Image Analysis and Recognition, Springer, 2018, pp. 363–369.

[10] O.S.R. Foundation, Gazebo, 2019. Available (accessed on 18.6.2019): http://gazebosim.org/

[11] O.S.R. Foundation, Robot operating system, 2019. Available (accessed on 28.1.2019): http://www.ros.org/

[12] O.S.R. Foundation, Robot operating system/concepts, 2019. Available (accessed on 28.1.2019): http://wiki.ros.org/ROS/Concepts

[13] O.S.R. Foundation, Robot operating system/introduction, 2019. Available (accessed on 28.1.2019): http://wiki.ros.org/ROS/Introduction

[14] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, M. Marín-Jiménez, Automatic generation and detection of highly reliable fiducial markers under occlusion, Pattern Recognition, Vol. 47, Iss. 6, 2014, pp. 2280 – 2292.

[15] J. Herzog, Aerial photography uav icon. Available (accessed on 13.11.2018): https://commons.wikimedia.org/wiki/File:Aerial_Photography_UAV_Icon.svg

[16] A. Hinas, J.M. Roberts, F. Gonzalez, Vision-based target finding and inspection of a ground target using a multirotor uav system, Sensors, Vol. 17, Iss. 12, 2017, p. 2929.

[17] HiveUAV, Hiveuav, 2018. Available (accessed on 18.10.2018): http://hiveuav.com/

[18] HiveUAV, Let's fly, 2018. Available (accessed on 14.11.2018): https://www.youtube.com/watch?v=aJNiCLrZlx4

[19] A.M..A. K, camera calibration, 2013. Available (accessed on 2.4.2019): https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html

[20] S. Lange, N. Sünderhauf, P. Protzel, Autonomous landing for a multirotor uav using vision, in: International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2008), 2008, pp. 482–491.

[21] D. Lee, T. Ryan, H.J. Kim, Autonomous landing of a vtol uav on a moving platform using image-based visual servoing, in: 2012 IEEE international conference on robotics and automation, IEEE, 2012, pp. 971–976.

[22] D. Lee, T. Ryan, H.J. Kim, Autonomous landing of a vtol uav on a moving platform using image-based visual servoing, in: 2012 IEEE International Conference on Robotics and Automation, May, 2012, pp. 971–976.

[23] V. Lepetit, F. Moreno-Noguer, P. Fua, Epnp: An accurate o (n) solution to the pnp problem, International journal of computer vision, Vol. 81, Iss. 2, 2009, p. 155.

[24] Logitech, Logitech c920 hd pro webcam, 2019. Available (accessed on 4.3.2019): https://www.logitech.com/en-ch/product/hd-pro-webcam-c920

[25] J.L.S. Lopez, *aruco_gazebo*, 2019. Available (accessed on 19.2.2019): https://github.com/joselusl/aruco_gazebo

[26] P. Mihelich, image view, 2019. Available (accessed on 23.10.2019): http://wiki.ros.org/image_view

[27] E. Olson, Apriltag: A robust and flexible visual fiducial system, in: Robotics and Automation (ICRA), 2011 IEEE International Conference on, IEEE, 2011, pp. 3400–3407.

[28] OpenCV, Detection of aruco markers, 2018. Available (accessed on 11.12.2018): https://docs.opencv.org/3.4/d5/dae/tutorial_aruco_detection.html

[29] J.B. Patrick Mihelich, cv bridge, 2019. Available (accessed on 18.2.2019): http://wiki.ros.org/cv_bridge

[30] B. Poling, A tutorial on camera models, University of Minnesota, 2015, pp. 1–10.

[31] D. Project, Development environment on linux, 2019. Available (accessed on 19.2.2019): https://dev.px4.io/en/setup/dev_env_linux.html

[32] D. Project, mavros_posix_sitl, 2019. Available (accessed on 19.6.2019): https://github.com/PX4/Firmware/blob/master/launch/mavros_posix_sitl.launch/

[33] D. Project, Micro air vehicle communication protocol, 2019. Available (accessed on 25.2.2019): https://mavlink.io/en/

[34] D. Project, mro pixhawk, 2019. Available (accessed on 21.2.2019): https://docs.px4.io/en/flight_controller/mro_pixhawk.html

[35] D. Project, Px4 controller diagrams, 2019. Available (accessed on 18.6.2019): https://dev.px4.io/en/flight_stack/controller_diagrams.html/

[36] D. Project, Px4 flight controlelr, 2019. Available (accessed on 18.6.2019): https://dev.px4.io/en/

[37] D. Project, Px4 flight stack, 2019. Available (accessed on 18.6.2019): https://dev.px4.io/en/concept/architecture.html/

[38] D. Project, qgroundcontrol, 2019. Available (accessed on 21.2.2019): http://qgroundcontrol.com/

[39] D. Project, Ros with gazebo simulation, 2019. Available (accessed on 19.2.2019): https://dev.px4.io/en/simulation/ros_interface.html

[40] Y. Qi, J. Jiang, J. Wu, J. Wang, C. Wang, J. Shan, Autonomous landing solution of low-cost quadrotor on a moving platform, Robotics and Autonomous Systems, Vol. 119, 2019, pp. 64 – 76.

[41] E. Riba Pi, Implementation of a 3d pose estimation algorithm, master's thesis, Universitat Politecnica de Catalunya, 2015.

[42] S. Saripalli, J.F. Montgomery, G.S. Sukhatme, Vision-based autonomous landing of an unmanned aerial vehicle, in: Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292), IEEE, 2002, Vol. 3, pp. 2799–2804.

[43] D. Scholz, rqt_plot, 2017. Available (accessed on 8.3.2019): http://wiki.ros.org/rqt_plot

[44] S. Suzuki *et al.*, Topological structural analysis of digitized binary images by border following, Computer vision, graphics, and image processing, Vol. 30, Iss. 1, 1985, pp. 32–46.

[45] O. team, Aruco detection module, 2019. Available (accessed on 25.6.2019): https://github.com/opencv/opencv_contrib/blob/master/modules/aruco/src/aruco.cpp/

[46] O. team, Opencv, 2019. Available (accessed on 28.1.2019): https://opencv.org

[47] A.D. Teamt, Ardupilot, 2019. Available (accessed on 18.6.2019): http://ardupilot.org/

[48] N. Victor, Enhancing the data capacity of qr codes by compressing the data before generation, 2012.

[49] J. Wang, E. Olson, Apriltag 2: Efficient and robust fiducial detection., in: IROS, 2016, pp. 4193–4198.

[50] J. Weng, P. Cohen, M. Herniou, Camera calibration with distortion models and accuracy evaluation, IEEE Transactions on Pattern Analysis & Machine Intelligence, Iss. 10, 1992, pp. 965–980.

[51] J. Wynn, T. McLain, Visual servoing for multirotor precision landing in varying light conditions, 2018.

[52] G. Xu, Y. Xu, GPS: theory, algorithms and applications, Springer, 2016.

[53] H. Xu, Application of gps-rtk technology in the land change survey, Procedia Engineering, Vol. 29, 2012, pp. 3454–3459.

[54] N. Zeuch, Understanding and applying machine vision, revised and expanded, CRC Press, 2000.

APPENDIX A: SIMULATED CAMERA SPECIFICATIONS

```
1  <xacro:camera_macro
2      namespace="${namespace}"
3      parent_link="base_link"
4      camera_suffix="red_iris"
5      frame_rate="30.0"
6      horizontal_fov="1.3962634"
7      image_width="640"
8      image_height="480"
9      image_format="R8G8B8"
10     min_distance="0.02"
11     max_distance="300"
12     noise_mean="0.0"
13     noise_stddev="0.007"
14     enable_visual="1"
15     >
16     <box size="0.05␣0.05␣0.05" />
17     <origin xyz="0␣0␣-0.07" rpy="0␣1.57079␣0"/>
18 </xacro:camera_macro>
```