Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Drone Precision Landing using Computer Vision

Semester Project

Laurin Goeller

lgoeller@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Manuel Eichelberger, Simon Tanner
Prof. Dr. Roger Wattenhofer

June 24, 2018

Acknowledgements

I am profoundly grateful to our supervisors Manuel Eichelberger and Simon Tanner for their valuable support and guidance. We always had interesting and helpful discussions in our meetings whenever assistance was needed. Furthermore, I want to give special thanks to Louis Kögel with whom I worked closely together for this semester project. He was pursuing a similar thesis and we shared a lot of insights to improve each others work and we also cooperated on Chapter 2.

Abstract

Drones have recently become very popular for many application areas in the field of automated surveillance, delivery services, agriculture and more. Unfortunately, one of the major limiting factors in drone application often comprise the insufficient duration of flighttime due to a lack of power supply after 10-20 minutes. To overcome this problematic and to implement missions beyond standard flight times, it was aimed to construct drones that were equipped with a continuous life cycle software allowing the drone to perform an automated exchange. Thus, at time points of insufficient battery, the drone would precisely land on a designated area to allow automatic charging on the ground, whereas a second drone would take off to continue the mission. The focus of this thesis was set to construct a model drone at which computer vision techniques can enhance precision landing accuracy over recent implementations. After solving unexpected issues with the flight controller software, we here report that drones equipped with computer vision achieved reproducibly precise landings on areas of interest and therefore, computer vision has evolved as a valid technique to improve automatic landing precision. As a slight limitation, these results were found to be lightning condition dependent in the testing environment, and future work should be designated to this topic.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Motivation 1

1.2 Related work 2

2 Drone Assembly **3**

2.1 Hardware 3

2.1.1 Flight Controller 3

2.1.2 Electronic Speed Controller 4

2.1.3 Motors and Propellers 4

2.1.4 Radio and Telemetry 5

2.1.5 Battery 5

2.1.6 Frame 5

2.1.7 Assembly 12

2.2 Software 16

2.2.1 PX4 16

2.2.2 MAVLink 18

2.3 PID Tuning 18

3 Computer Vision for Precision Landing **21**

3.1 Software Implementation 22

3.1.1 Computer Vision Algorithm 22

3.1.2 Position Estimation 23

3.1.3 Offboard Flight Mode 25

3.1.4 Communication with Flight Controller Software 26

4 Application **28**

4.0.1 Test 1: Computer Vision Detection Accuracy 29

4.0.2 Test 2: Computer Vision Detection Reliability 31

4.0.3 Theoretical Experiment: Computation of Velocity Components 32

5 Summary **34**

5.0.1 Future Work 34

Bibliography **36**

A Appendix Chapter **A-1**

A.1 Raspberry Pi 3 Model B Installation A-1

Chapter 1

Introduction

1.1 Motivation

In the 21st century, drones, as unmanned aerial vehicles (UAV), are becoming increasingly important in a diverse area of application and interest. Not only for hobbyists but also for many different applications in the field of surveillance, delivery services, and agriculture. In many of these cases, multicopters, a special kind of drones with four or more propellers, are being used by reason of their size, weight and high flexibility. A determining drawback of multicopters is their short fight time of around 10-20 minutes. In this paper, we want to bypass this issue by working towards a continuous life cycle of drones which includes an automatic replacement of a drone in use when the battery is low, followed by a precise landing on a landing platform where it is automatically recharged and prepared for the next mission. The possibility of non-intermittent drone missions will greatly improve their capabilities and even allow a broader range of applications, such as nonstop surveillance operations.

An accurate landing is of crucial importance for an automated charging system as the exact positioning and yaw alignment of the drone facilitates the automation drastically. The Global Positioning System (GPS) is not accurate enough for a reliable landing. Furthermore, most existing algorithms require an Infrared (IR) beacon on the platform for a precise landing. This approach depends upon additional hardware and does not control yaw orientation of the drone. To overcome this problem, we here report to construct a model drone at which computer vision techniques can enhance precision landing accuracy to improve todays drone implementations.

1.2 Related work

As drone automation nowadays is an important research area, we are not the first to contribute to this matter. Many publications about automated flying guided us during our thesis. An important paper titled ”Endless Flyer: A Continuous Flying Drone with Automatic Battery Replacement” [1] written by authors Katsuya Fujii, Keita Higuchi and Jun Rekimoto is proposing an automatic mechanism to replace a drones' battery after flight and allow the drones to fly continuously without manual battery replacement. Our thesis ties to this paper as a precise landing is desired for a reliable replacement mechanism.

Another important article that influenced our work is the already existing precision landing algorithm [2] of the PX4 flight controller software (see Chapter 2.2). This algorithm makes use of an IR beacon on the landing platform to mark its position allowing precise landings within 10 cm. In this approach, the additionally needed hardware on the ground as well as on the drone is somewhat problematic. These limitations of already established landing algorithms have inspired us to develop a vision based approach for a precise landing using the PX4 software, where only a camera on the drone is needed to recognize the desired landing position.

Chapter 2

Drone Assembly

2.1 Hardware

To begin our work with a precision landing algorithm, a drone needed to be build. For our application, the drone needs to meet the following conditions:

- Allow GPS controlled flight for autonomous missions
- Provide a telemetry link to a ground station for communication and monitoring
- External applications can gain control of the drone for precision landing
- Airtime on one battery greater than 10 minutes for hassle free testing

The above mentioned demands led us to use the following components.

2.1.1 Flight Controller

As explained below in Section 2.2.1, we used PX4 as the software to control our quadcopter. In addition to standard flight controllers we needed more computing power to perform computer vision tasks on the drone. A solution to these demands is to use a standard Pixhawk 2 flight controller [3] and add a Raspberry Pi [4] as companion computer, but instead we chose to unite these two different boards into a single one using the Emlid Navio2 shield for the Raspberry Pi. This shield provides all sensors and controllers for creating a powerful flight controller with a Raspberry Pi and PX4 software: dual IMU (inertial measurement unit) for orientation and motion sensing, a barometer for precise altitude controlling, GNSS receiver for GPS positioning and extension ports (UART) for a telemetry radio. Furthermore, this combination is compatible with Dronekit, an API to develop apps for drones (used for precision landing, for further information see Chapter 3). In addition, this hardware is the lightest and most compact stack for our application. The Navio2 mounted on a Rapsberry Pi can be seen in Figure 2.1

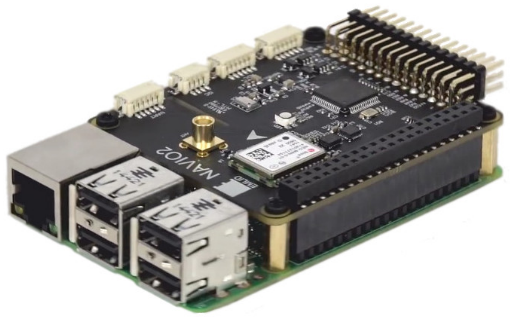


Figure 2.1: Navio2 Shield mounted on a Raspberry Pi [5]

2.1.2 Electronic Speed Controller

The flight controller sends a Pulse Width Modulated (PWM) signal to the Electronic Speed Controller (ESC) indicating how fast the motor should spin. The ESC then controls the power to the motors to reach the desired rotation speed. For our drone, we chose to use the Hobbywing xRotor 40A 4in1 [6]. These are up to date ESCs widely used in the drone industry, especially for applications where high performance and low weight might be required. In contrast to most ESCs this is a single board holding all 4 ESCs making assembly and cable management a lot easier.

2.1.3 Motors and Propellers

Our motor and propeller choice is also based on common components from todays drone industry. As the estimated drones' weight was calculated to be less than 1.4 kg, the motors of our choice are T-motors F40 1600KV [7]. According their manual, these motors can spin a 7 inch propeller [8] together with a 5S lithium polymer battery with a maximum lift of 1400 g per motor resulting in a thrust to weight ratio of around 4:1. According to several online platforms, a ratio of 2:1 is the the lower boundary for quadcopters to still allow for maneuverability and enough power to arrest a descent. Therefore, 4:1 is a powerful drone setup allowing this drone to be a platform for future projects where a carriage of more weight is required.

2.1.4 Radio and Telemetry

As a radio link to manually control the copter, we use the Frsky Taranis [9] which is one of the most famous RC transmitter for drones world wide together with a Frsky XSR receiver [9]. The telemetry connection is provided by a standard 433 Mhz module [10] for MavLINK communication (see Chapter 2.2) compatible with the PX4 flight controller software.

2.1.5 Battery

The drone is powered by a 5S Lithium Polymer (LiPo) battery with 3800mAh [11]. LiPo batteries are predominantly used in all RC hobbies based on their high power density. The voltage of our LiPo battery (5S means there are 5 cells of 3,7V in series resulting in a total Voltage of 18.5V) matches the combination of motor and propeller choice according to the motor's manual.

2.1.6 Frame

Frame Requirements and Design

The principle requirements to our frame design are simple: We were in need of a quadcopter frame which is strong enough to serve as a mounting platform for all different electronics and still light enough to keep the flight time of our drone on a satisfactory level. In detail, the frame should provide a good mounting solution for the flight controller stack as well as additional components like camera, the RC receiver and the telemetry module for the connection to our ground station. Because of our choice of flight controller, most standard frames available on the market do not offer the correct mounting holes. Instead of adapting a frame we chose to design a new frame to perfectly match the requirements mentioned above.

The first step in design was to specify the arrangement of the motors' and the propellers' size. One could either mount the motors in a so called "+ layout" or a "x layout", whereas x is more popular since a camera mounted on the frame does have a clear view to the front. In a + configuration, the front motors can always be seen in the camera's video feed. Both motor layouts can be seen in figure 2.2

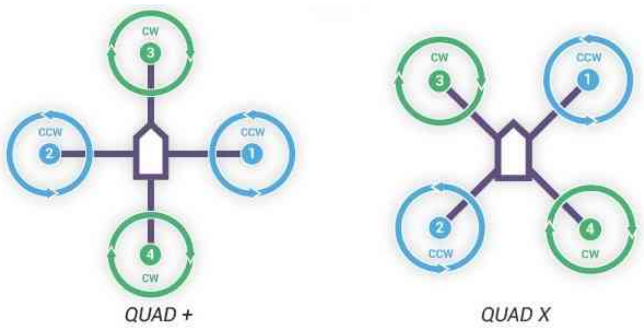


Figure 2.2: Comparison of + motor layout and x motor layout. [12]

Although we are not planning on mounting a camera pointing to the front, we still decided to use an x configuration because of usability. In a second step we identified the minimum width of the middle plate where all components are mounted on. The widest part is the Raspberry Pi with measurements of 85 mm by 49 mm, so 49 mm is the narrowest possibility for the middle part. However, by adding a little slack, we ended up with a width of 55 mm for the middle part of the frame.

Based on the propellers diameter, the x configuration and the middle part width we can now start to draw the frame. The design itself is straight forward always meeting the predefined requirements. The resulting layout can be seen in figures 2.3 and 2.4, where both the rendered frame design and a photography of the frame is shown.

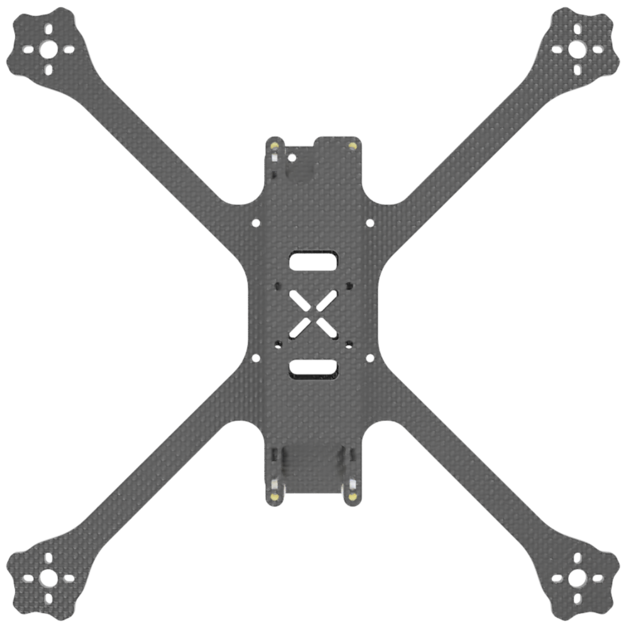


Figure 2.3: Rendering of the final frame design. This is a view from the top.

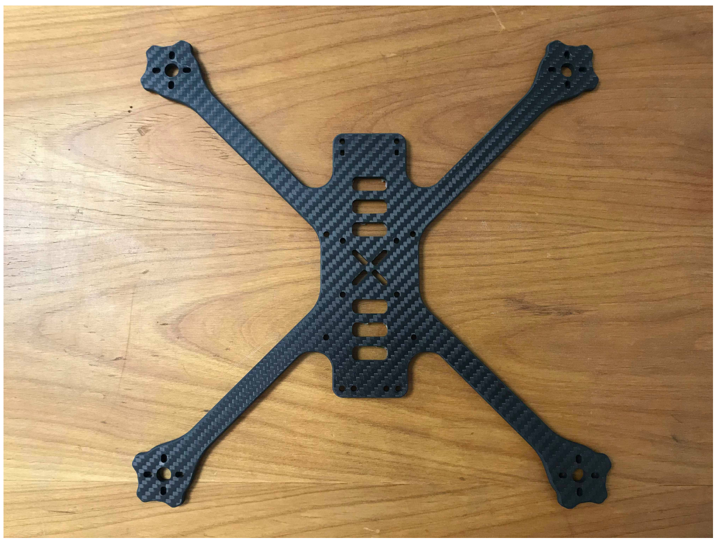


Figure 2.4: Image of the final product made out of carbon fiber composite material.

Before the production of the frames, we conducted a static stress simulation to make sure the weak point of the frame is not in its center. This way the electronics are well protected in case of a crash. Fusion 360 (see Chapter 2.1.6) allows to conduct a static stress analysis on our modeled frame. A typical impact point in a drone crash is on the end of an arm. According to this we stressed one arm with forces in several directions. The point of highest mechanical tension always occurred near the connection between arm and body which is shown in figure 2.5. Obviously, static stress testing is a preliminary technique which does not allow too reliable results due to the fact that forces in a real crash are dynamic and might differ from simulated ones.

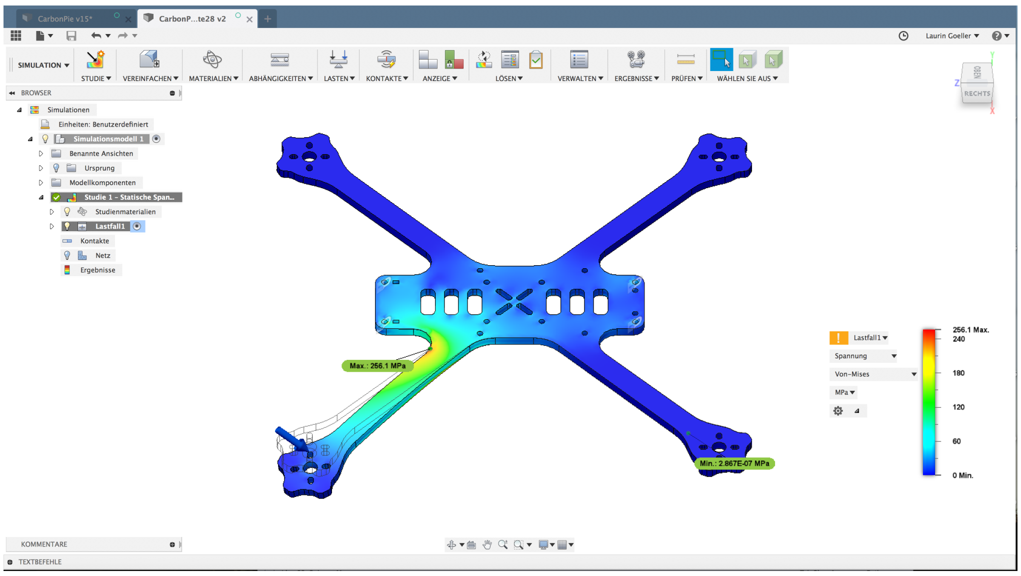


Figure 2.5: Screenshot of static stress analysis in Fusion 360. The bottom left arm is stressed with force of 200 Newton. Red areas show the region of highest mechanical tension.

Design Environment

The CAD design software of our choice is Autodesk Fusion 360 [13]. Thanks to its student license this software provides free access to many features in CAD (computer aided design), CAE (computer aided engineering) and CAM (computer aided manufacturing). Using this software was an easy decision not only because of our experience but for its great possibilities as a cloud based platform. This enabled a reliable work process and easy collaboration within our team.

Additional Parts

To attach all optional parts (such as camera, GPS antenna, RC receiver) to our frame, some additional mounts needed to be manufactured. As we have a 3D printer at our disposal, 3D printing these parts is a good choice for fabrication. Figures 2.6, 2.7 and 2.8 show the additional parts with a short description of their use.

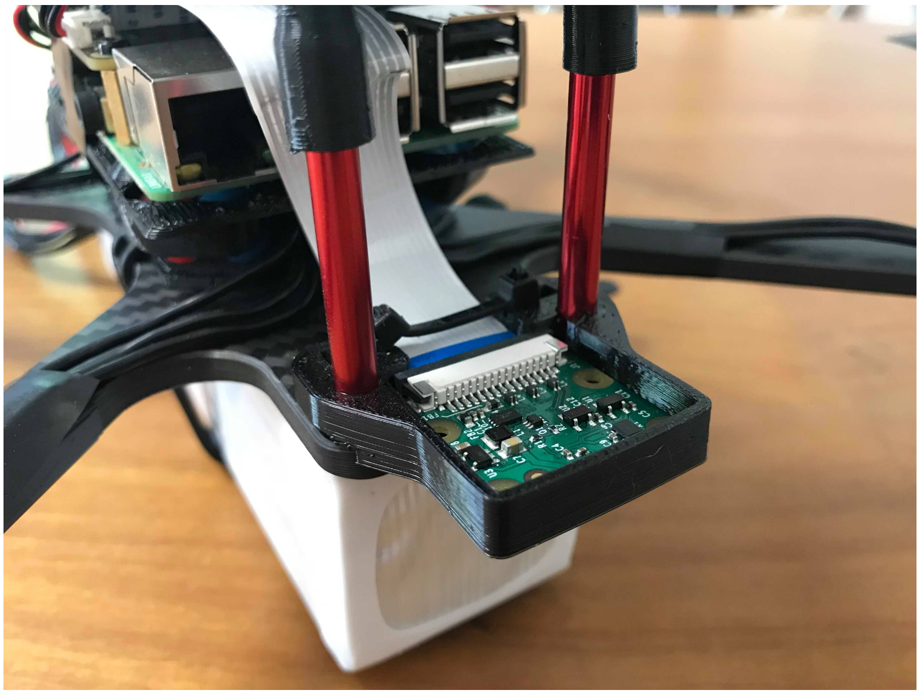


Figure 2.6: Raspberry Pi camera mount holds the camera in front of the frame.

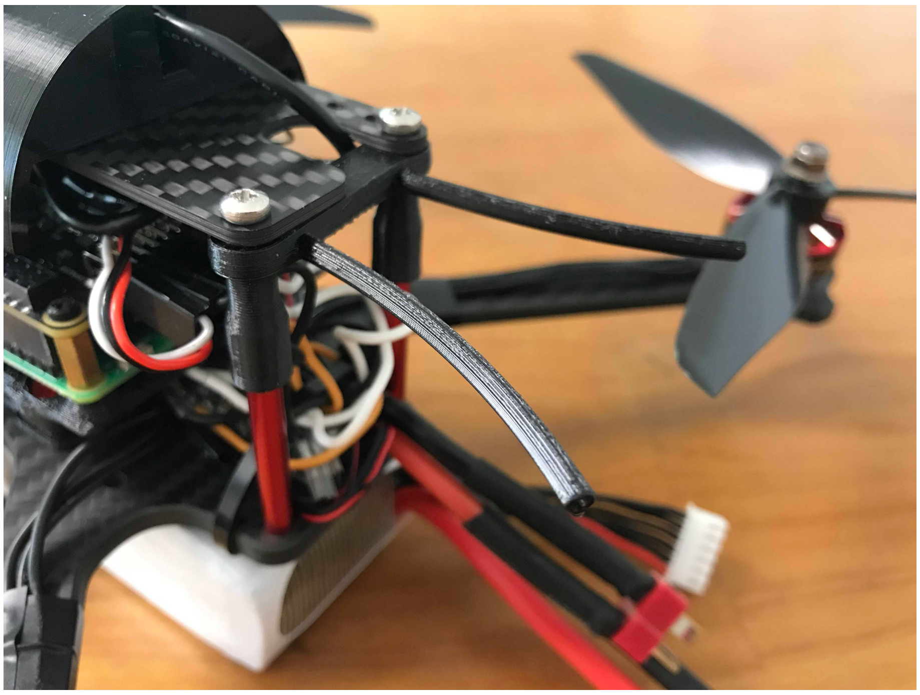


Figure 2.7: A mount for RC receiver antennas makes sure they are kept away from the propellers.

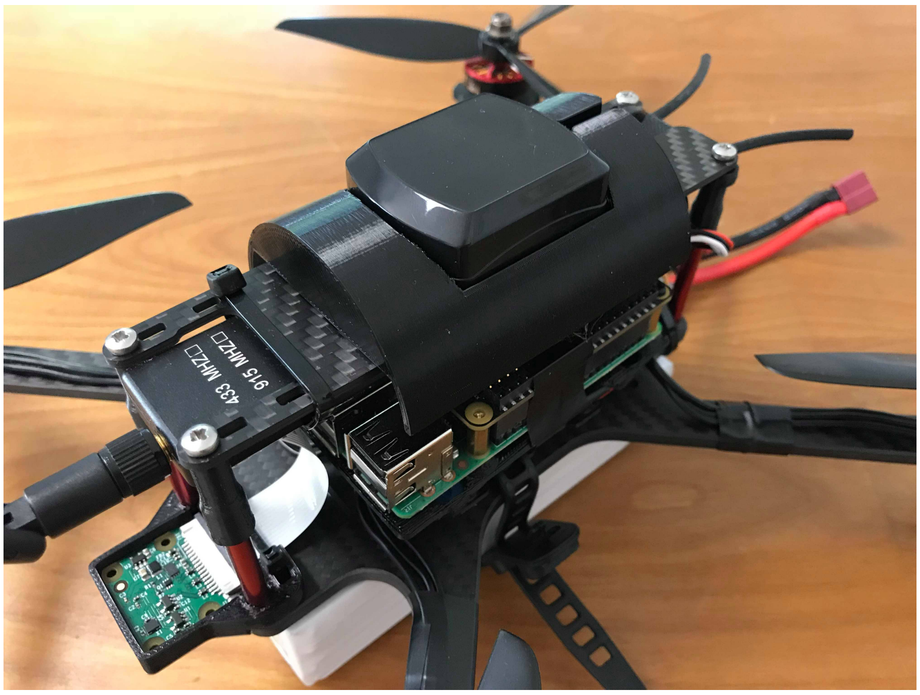


Figure 2.8: The GPS antenna mount helps to keep the GPS antenna safe on the drone. For further use an extension is recommended.

2.1.7 Assembly

The assembly of our quadcopter follows the manual for the Navio2 on Emlid's website [14]. Figure 2.9 and 2.10 show the wiring diagram and Figure 2.11 as well as Figure 2.12 present pictures of the final drone.

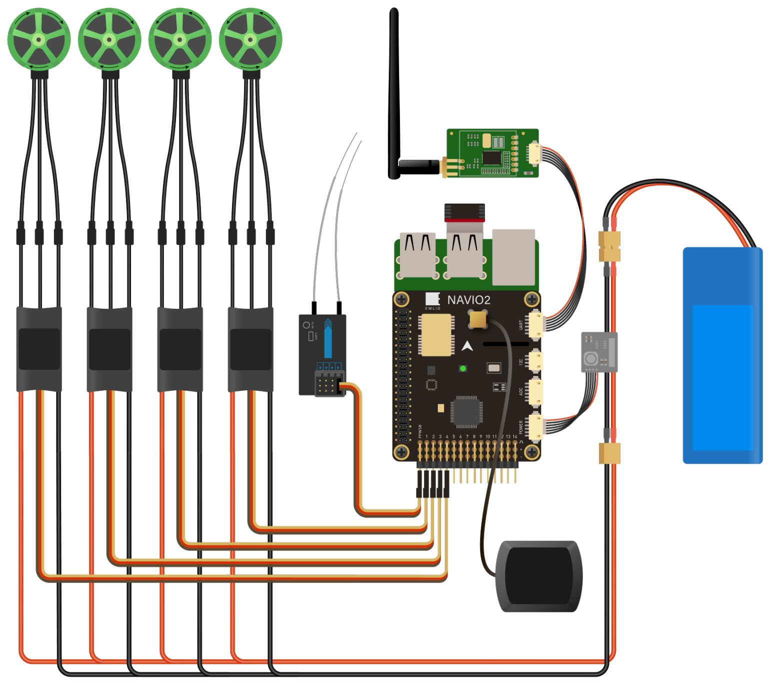


Figure 2.9: Scheme of the drone assembly and connection diagram. The only difference to our drone is that the Raspberry Pi Camera is missing and four single ESCs instead of one 4in1 ESC is used.

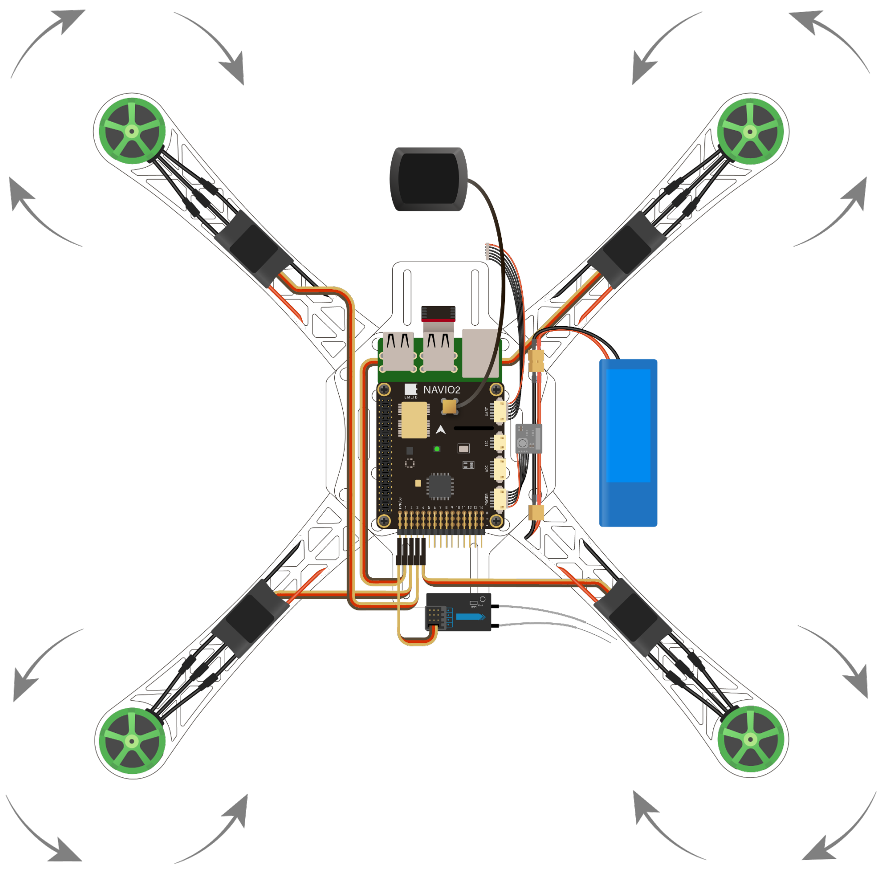


Figure 2.10: All components on a quadcopter frame. The arrows indicate the direction of rotation of the motors. In this picture the telemetry module as well as the Raspberry Pi camera are left out.

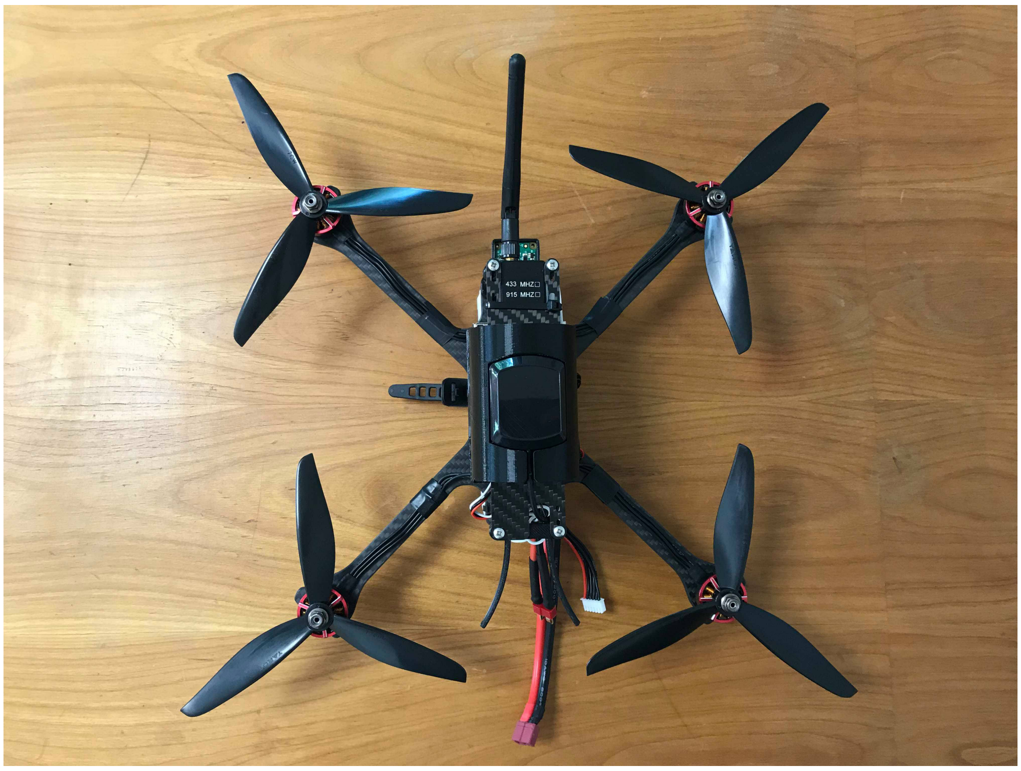


Figure 2.11: An image of the final drone from top view.

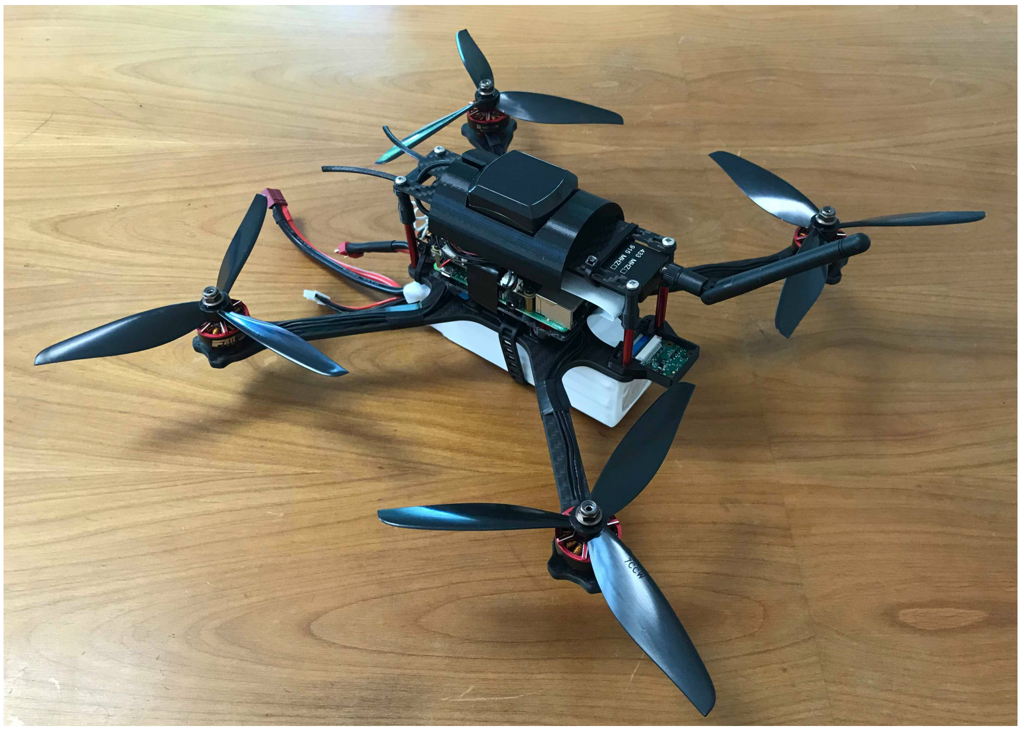


Figure 2.12: An image of the final drone from side view.

2.2 Software

So far we have elaborated the hardware components of our aircraft. Now we want to make these components work together in order to fly. Many open source autopilot software for plenty of applications and different UAV setups exist. During our semester thesis we used PX4 as our autopilot software, because it brings along a large toolset for autonomous flights. In addition, PX4 includes the ability to communicate with a Ground Station.

2.2.1 PX4

PX4 is the autopilot of choice because the PX4 project was founded at ETH Zürich and still many ETH engineers are developers for the project. This was indispensable for complications we had because we could directly address our problems to the developers.

The high level software architecture of PX4 is shown in Figure 2.13. One of the core algorithms is the State Estimator which is an Extended Kalman Filter (EKF). It fuses the IMU sensor data with the GPS sensor data to estimate the orientation and a local and global position. PX4 then uses this position estimation to hold its position and to hover in place. This is a key feature we rely on to fly autonomously. In this overview we also see PX4 features like Connectivity and Storage which we use to build and test our application. We use MAVLink as the communication protocol to communicate with the flight controller. From our ground computer we will later look into log data of our drone, which is used to tune the PID controllers.

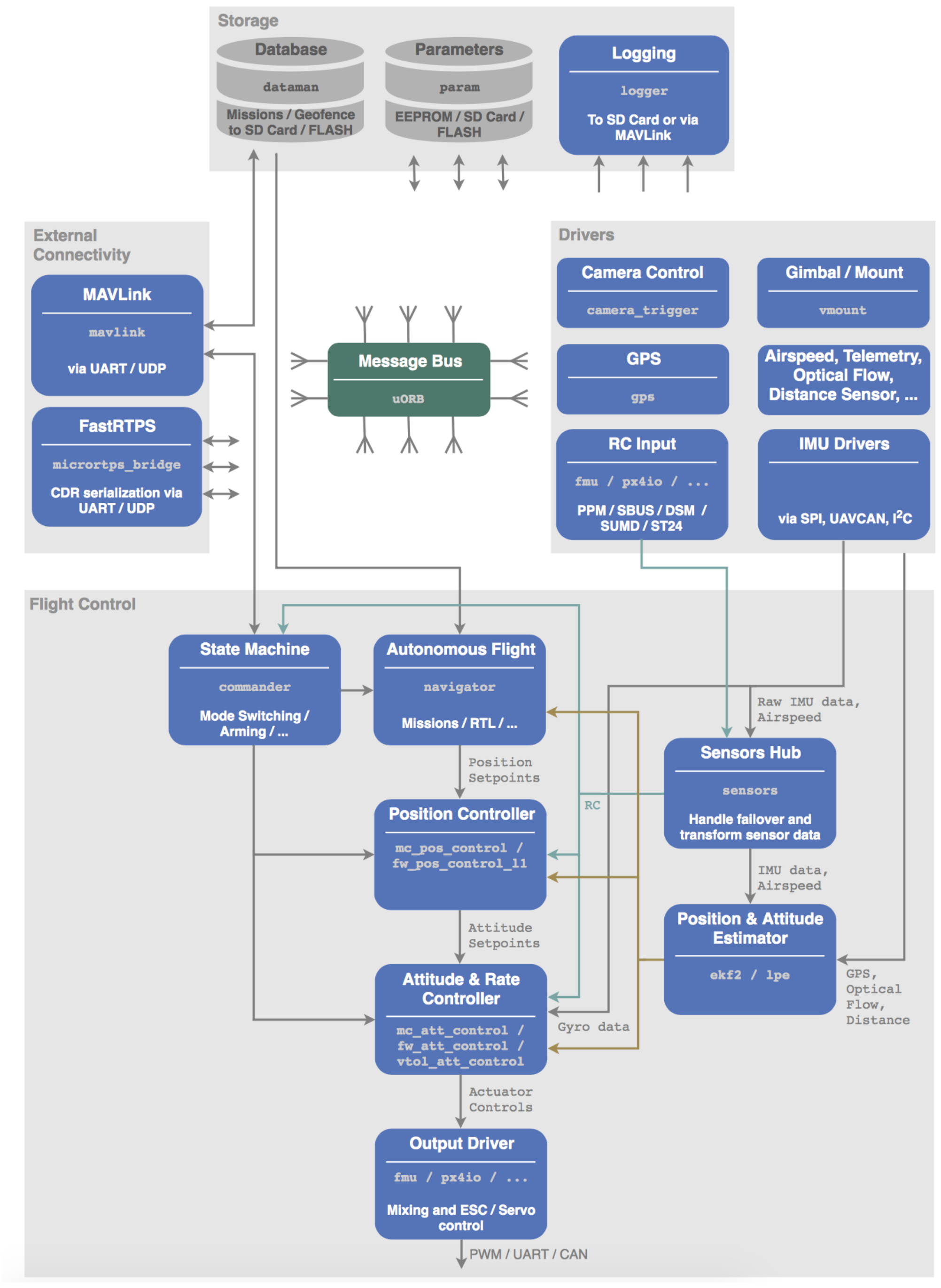


Figure 2.13: PX4 high level software architecture [15]. The Drivers collect data from all the different sensors. The Flight Control uses the data to maneuver the drone. Over the External Connectivity the drone can be connected to a Ground Computer. Data can be exchanged and commands can be send to the drone. The Storage logs the flight.

2.2.2 MAVLink

MAVLink means Micro Air Vehicle Link and is a protocol for communication. It is used by PX4 for the communication between the Flight Control Unit (FCU) and the Ground Computer. In our case it is also used for the communication between the computer vision script and the FCU.

2.3 PID Tuning

PX4 uses PID controllers, which are widely used feedback loop controllers in quadcopter control systems and many other applications. The controller continuously calculates an error value e(t) as the difference between a desired setpoint r(t) (given by the RC remote) and a measured actual system state y(t) (which is an estimate from the sensor data through the EKF). Based on the error e(t), a proportional, integral, and derivative correction term is calculated (denoted by P, I, and D respectively). Each term is multiplied by a coefficient to create the corrective control output.

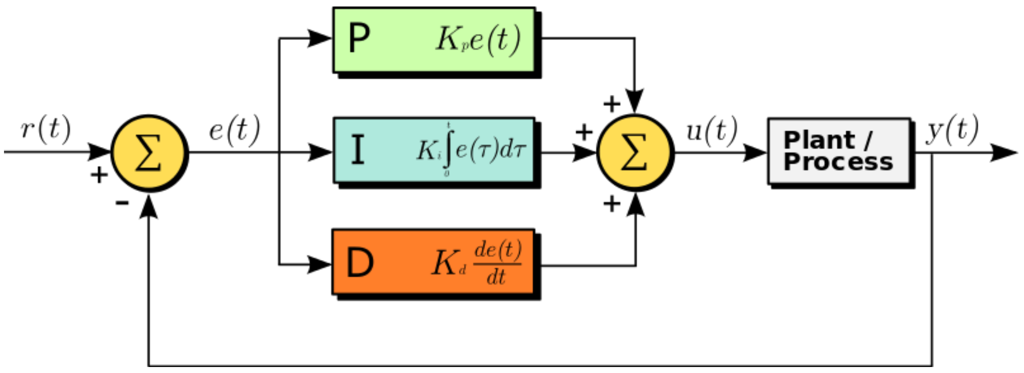


Figure 2.14: PID control loop [16].

PX4 has multiple layers of controllers and depending on the flight mode of the drone, the PID controller is responsible fore a stable flight [17]. In our first flight, we have only used the inner-most controller (rate controller by switching to the "Manual Flight Mode"). The inner-most controllers consist of three independent PID controllers responsible for regulating the change of angle in roll, pitch and yaw orientation.

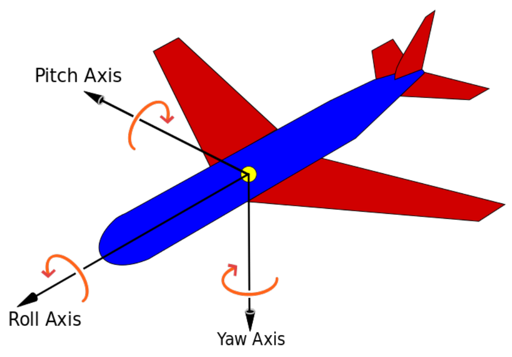


Figure 2.15: Aircraft principal axes [18].

Those controllers are not tuned appropriately as our first flight quickly proved. The drone oscillated and was hardly controllable. To tune the PID controllers we followed the steps in the PX4 PID Tuning Guide [17].

In the Figures 2.16 and 2.17 the effect of PID tuning is exemplified. As an example, the roll-rate controller output is observed (also possible are the pitch- and yaw-rate). Figure 2.16 shows a log with the default parameters whereas Figure 2.17 is a log with tuned parameters. The green line corresponds to the setpoint (or control input). The red line however is the actual roll-rate estimated by the EKF. In a well tuned controller, red line should follow the green line. As we can see in Figure 2.16, the red line oscillates around the green one with a high amplitude. This explains the strong oscillation in the drones' flight. In contrast to that, Figure 2.17 shows that the red line closely follows the green one. The result of PID tuning is a smooth flight and a controllable drone.

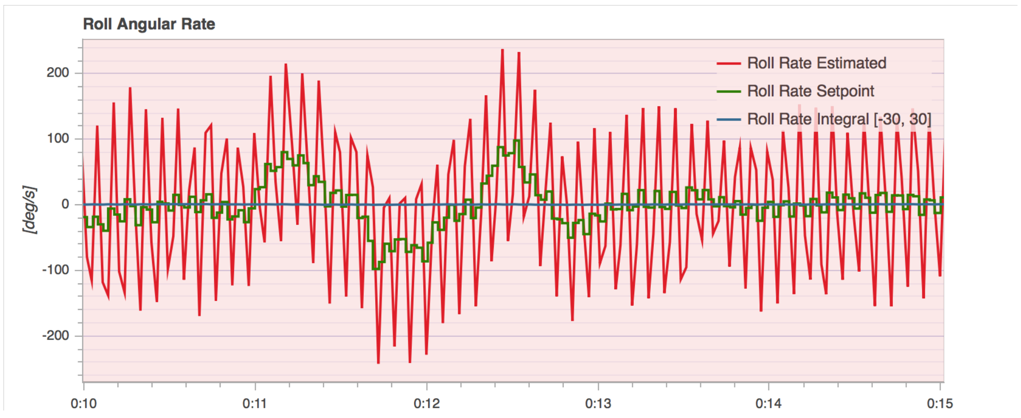


Figure 2.16: The *Roll Angular Rate* Log data from a test flight. This rate is controlled by a PID controller. The green line is the desired rate and the red line is the actual rate of the quadcopter. Clearly we can infer that the PID controller is not tuned properly. The actual rate oscillates with a high amplitude which results in an unstable flight.

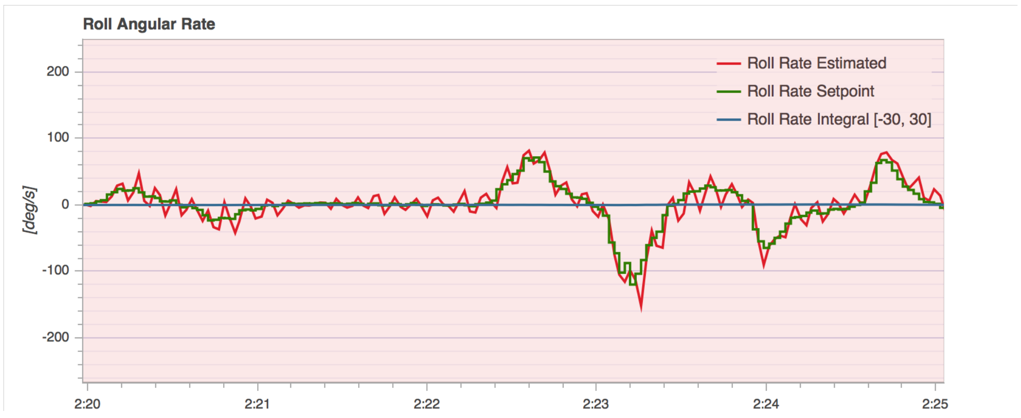


Figure 2.17: The *Roll Angular Rate* Log data as in Figure 2.16 but with tuned PID parameters. The green line is the desired rate and the red line is the actual rate of the quadcopter. With tuned PID parameters, we achieve a much more stable flight than before.

Chapter 3

Computer Vision for Precision Landing

For the purpose of automated drone missions, a precise landing is a principal requirement for many applications. Computer vision techniques allow the landing platform to be very simple: the basic idea is to mark the landing spot with a special marker which is then detected by the Raspberry Pi Camera mounted on the drone. The image is processed by an OpenCV Python script (see Chapter 3.1.1) calculating the relative position between drone and marker to gain accurate feedback for a precise landing. As a marker, we decided to use 3 circles with different colors (blue, green, pink) to enable simple detection techniques using color filters. The marker can be seen in Figure 3.1. Detailed information for the computer vision algorithm as well as the communication between image processing and flight controller with an API called Dronekit (see Chapter 3.1.4) can be found in the following section.

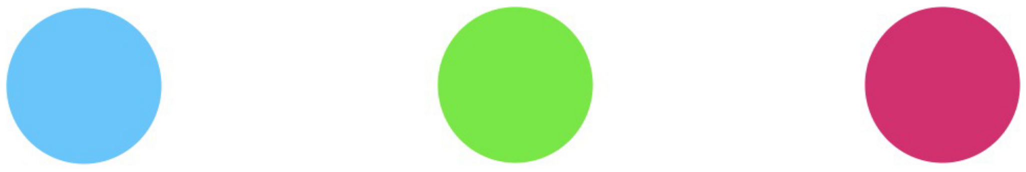


Figure 3.1: This marker is positioned on the landing platform for image recognition. It concists of three circles in the color blue, green and pink.

As other precision landing algorithms in px4 have an accuracy of 5-30cm [19], our goal was to verify whether computer vision techniques can achieve a landing with equal or improved precision.

3.1 Software Implementation

The landing tracker script consists of three different sections. The first is handling all image processing and calculating the markers coordinates in terms of pixel position in the frame. The second part uses these coordinates to calculate the desired movements of the drone and the third part handles the communication with the flight controller.

3.1.1 Computer Vision Algorithm

The computer vision part is implemented by using the OpenCV library for Python. OpenCV [20] is a free library including algorithms for image processing and computer vision. It is available for C, C++, Python and Java. In this project we decided to use Python because the second software component, Dronekit [21], is an API for Python only.

The algorithm follows the subsequent procedure:

- grab the current frame from the camera
- process each frame (gaussian blur filters, convert to hue, saturation, value (hsv) color space, resize)
- apply predefined color filters for each color
- find largest contours in the frame using the OpenCV function cv2.findContours
- compute center coordinates of largest contours and add them to a stack for further accessing

Figure 3.2 shows the processing of a camera image before and after applying the color mask for the blue marker. This is done similarly for all three colors. The code for marker detection is based on the OpenCV tutorial for ball tracking by Adrian Rosebrock [22]. His article has been of great help understanding basic OpenCV functions and implementing the marker tracking algorithm.

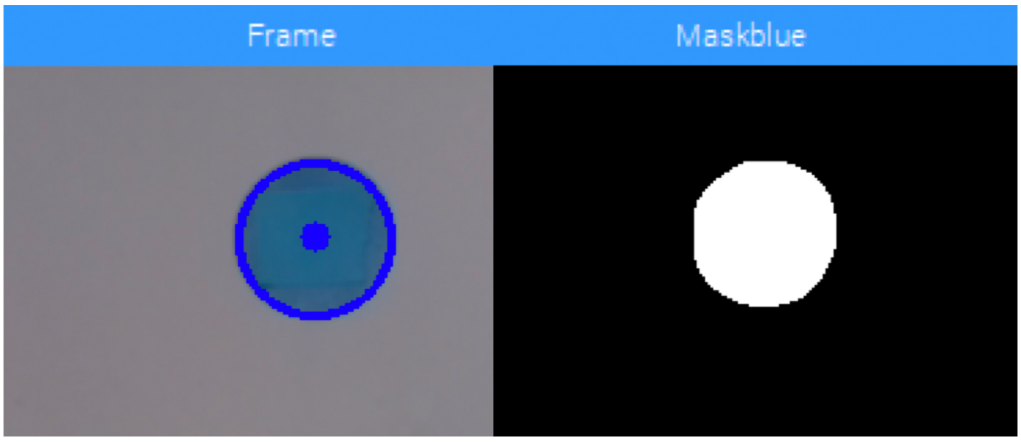


Figure 3.2: Camera image before and after color filtering for blue marker.

3.1.2 Position Estimation

This fraction of the code is for computing the drone's relative position to the marker and to calculate movement commands to hold the drone's position above the marker (in pixel coordinates). Figure 3.3 explains the coordinate system seen by the camera.

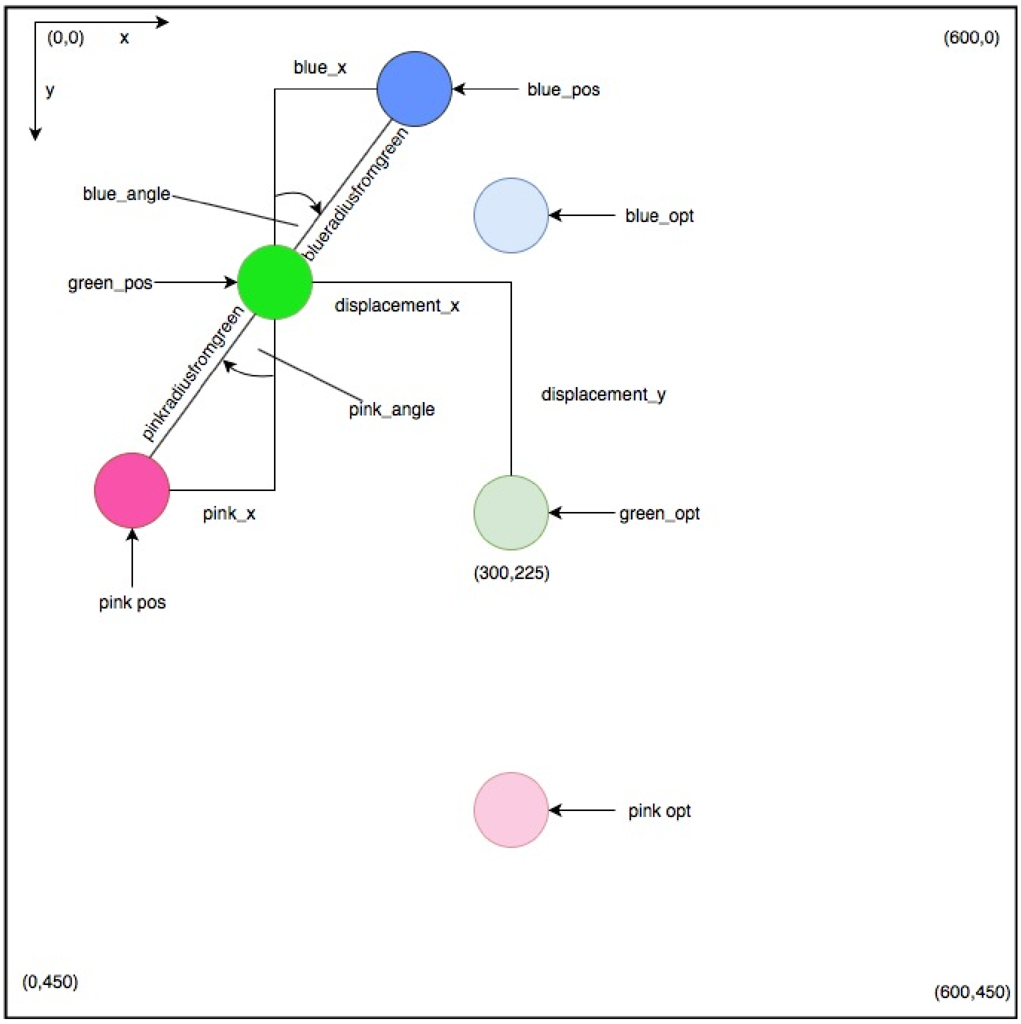


Figure 3.3: This figure shows the coordinate system from the view of the Raspberry Pi camera. The transparent markers in the middle of the frame refer to the markers position when the drone is aligned to the landing platform. The green transparent marker also coincides with the center point of the frame. The fully colored markers represent the actual positions of the markers in the camera frame. The displacement in x and y direction corresponds to the drones position relative to the landing platform in pixels. The pink angle and blue angle are used to identify the drones yaw orientation.

For the position calculation (displacement x and displacement y in Figure 3.3) the green point's coordinates relative to the center of the frame (represented by the coordinates (300, 225)) are used. The displacement vector contains the relative position between center_pos and green_pos and is calculated by subtracting the center coordinates from green_pos coordinates. In a second step the angles pink_angle and blue_angle are calculated separately and then combined into a total rotation output. The calculation follows the following schedule:

- Check whether the green point was detected successfully. If not, skip this frame.
- Calculate the displacement vector by subtracting center coordinates from green_pos coordinates.
- If x and y displacement are within a certain region of the center coordinates, position is reached. Else the desired movementis the displacement vector.
- Check wether blue and pink points have been detected successfully. If only one was found, display warning and continue with only one reference point.
- Calculate both angles with basic trigonometric functions and convert the solution from rad to degrees. For example pink_angle=asin(pink_x/pinkradiusfromgreen)
- Compare both results for pink and blue, if they are within a predefined threshold angle, the computation is accepted. If the difference between both angles is greater than this threshold, the identification of the markers must be erroneous. The current frame is skipped for yaw position alignment.

3.1.3 Offboard Flight Mode

The offboard flight mode of the PX4 flight controller software is primarily used for controlling vehicle movement and attitude with a limited set of MAVLink commands. Using this mode one can set the vehicles speed components in x, y, z direction using the Dronekit function send_ned_velocity in the NED (North, East, Down) frame. This frame is used in the MaVLINK command internally. Now the task is to compute the speeds in x and y direction (NED frame) using the position estimate from the Chapter 3.1.2. This is done by taking the current attitude information of the drone (in NED frame) together with the position estimation results from Chapter 3.1.2 to compute a velocity vector as well as a rotation around the yaw axis to maintain a stable position hold above the landing platform. The code follows this scheme:

- Get current attitude information by calling the "vehicle.attitude" attribute of the "vehicle" class from Dronekit
- Compute velocity components in NED frame from displacement vector (in pixel coordinates), height of the drone and and current yaw direction.

- Process velocity components and yaw heading with a filter to smoothen the output.
- Send movement commands to drone

3.1.4 Communication with Flight Controller Software

To pass the results of position estimating to the drone's flight controller, Dronekit is used for communication. Dronekit is a Python API that allows developers to create apps that communicate with ArduPilot [23] (alternative software for PX4) flight controller using MAVlink as transmission protocol. As PX4 is also based on MAVlink communication, Dronekit is compatible with our PX4 stack too. This API is well documented and provides programmatic access to the drones telemetry, state and parameter information as well as waypoint management and direct control over vehicle movement. In our case, vehicle movement control is better suited than waypoint missions as the required movement changes rapidly and cannot be controlled by GPS coordinates. The communication code is based on the following procedure:

- connect to the vehicle using the loopback address (127.0.0.1) for UDP protocol on port 14550
- read and print useful attributes such as vehicle.attitude, vehicle.is_armable and vehicle.system_status.state
- perform simple takeoff in offboard mode to 1 m height
- start color tracking and hold position with the script.
- slow decent onto landing platform while constantly updating position
- disarm motors when landing platform is reached.

The total flow of information within the landing algorithm can be seen in Figure 3.4

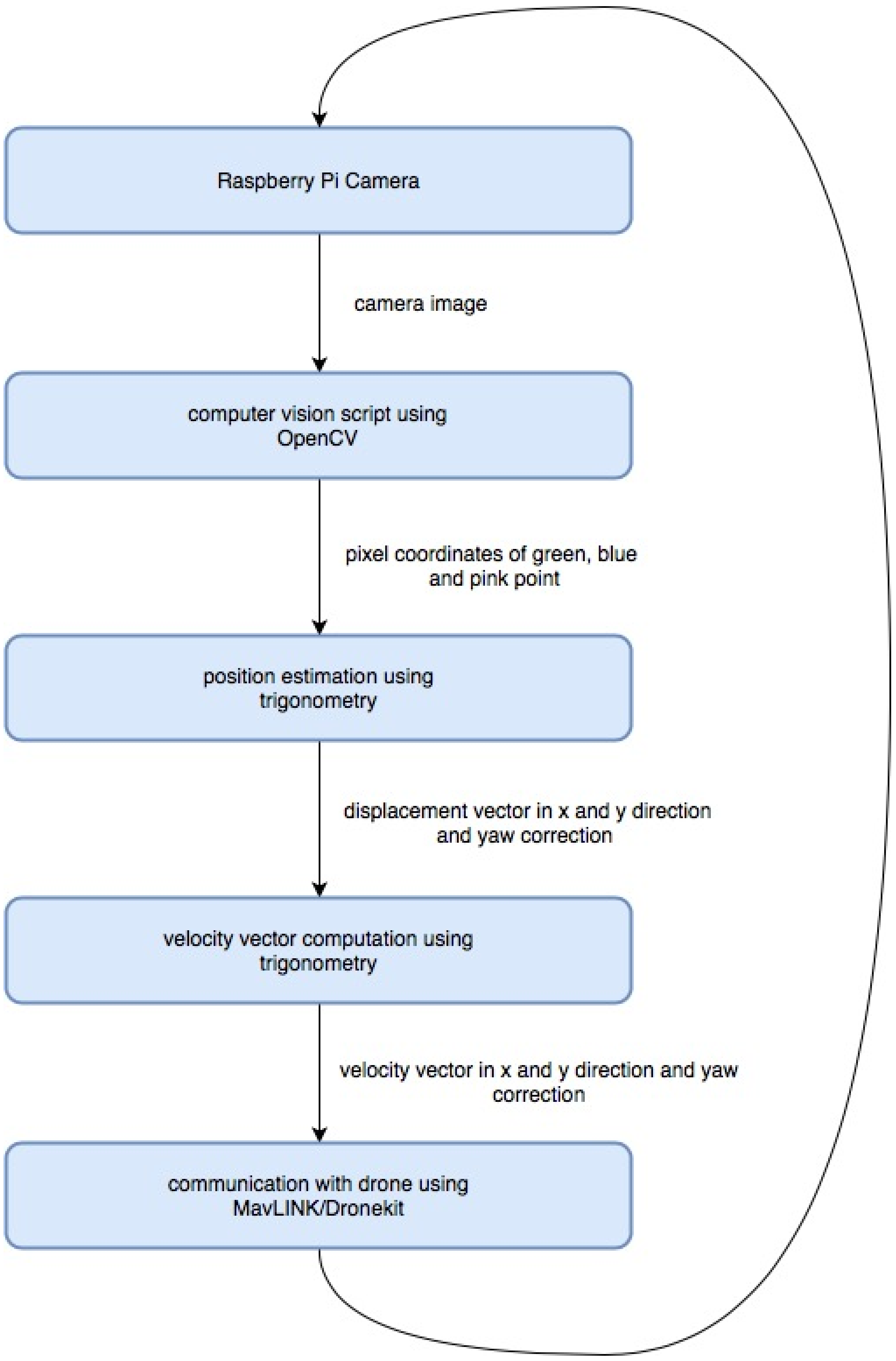

Raspberry Pi Camera
camera image
computer vision script using OpenCV
pixel coordinates of green, blue and pink point
position estimation using trigonometry
displacement vector in x and y direction and yaw correction
velocity vector computation using trigonometry
velocity vector in x and y direction and yaw correction
communication with drone using MavLINK/Dronekit

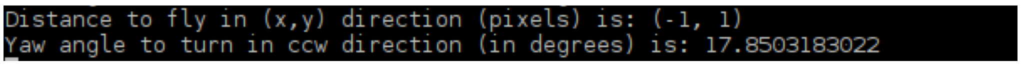
Figure 3.4: This chart shows the flow of information in the precision landing script. Each of the blue boxes represent computational parts and the arrows in between correspond to the variables being passed between these calculations. The loop is closed which indicates a loop behavior for the whole script.

Chapter 4

Application

By virtue of unexpectedly many issues with the flight controller software, we never have tested this precision landing algorithm in the drones flight. Since our drones' PX4 software had a bug with the barometer which kept us from performing GPS controlled flights, we asked a PX4 developer for help. He found out that there was a bug in the barometer driver for the Navio2 shield. He also was able to fix the bug but unfortunately only four days before our submission date [24].

Nevertheless we conducted several experiments to test our implementations reliability and accuracy. For the first two tests, the computer vision algorithm was slightly adjusted to output the desired yaw rotation and movement (in pixel coordinates according to Figure 3.3) in x and y direction. An example output can be seen in Figure 4.1 whilst Figure 4.2 shows the according camera image to this output.



```
Distance to fly in (x,y) direction (pixels) is: (-1, 1)
Yaw angle to turn in ccw direction (in degrees) is: 17.8503183022
```

Figure 4.1: The console output for the adapted testing script. The output corresponds to the situation in Figure 4.2.

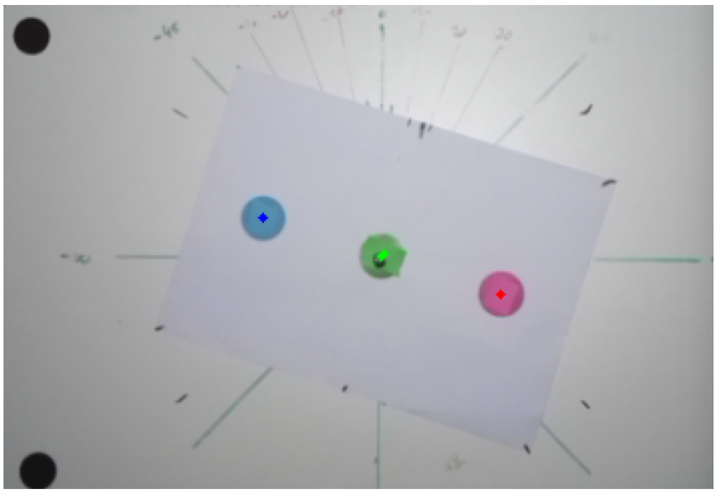


Figure 4.2: the landing pad situation seen from the Raspberry Pi camera corresponding to the console output in Figure 4.1.

4.0.1 Test 1: Computer Vision Detection Accuracy

Our setup for testing the accuracy of the position estimation algorithm of Chapter 3.1.2 can be seen in Figure 4.3. The Drone is fixed to a point about 70 cm above the landing platform. The landing pad is laying on a fixed board on which a coordinate system is drawn. The origin of this coordinate system corresponds to the center of the image according to Figure 3.1.2.

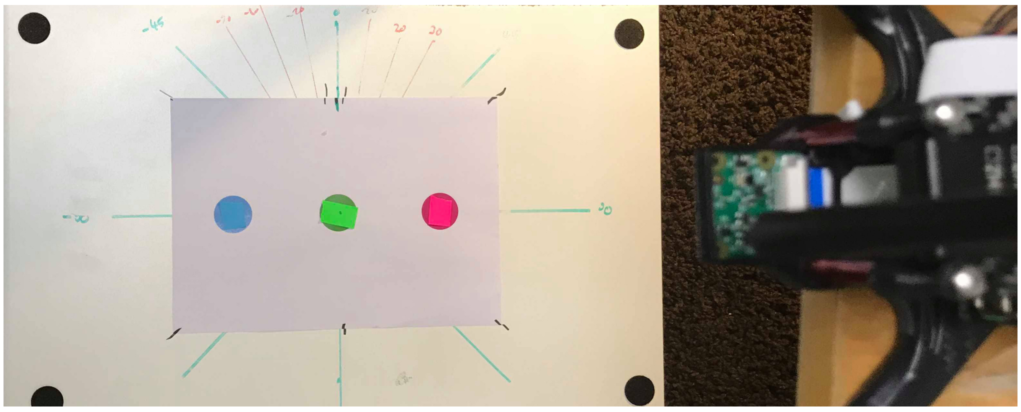


Figure 4.3: Test setup for Test 4.0.1 and Test 4.0.2. The drone is fixed to a height about 70 cm above the landing platform. The marker sheet is laying of a fixed board and can be moved around. The center coordinate of the camera frame according to figure 3.1.2 is aligned with the center of the board.

To test the yaw accuracy, the landing pad is rotated in 5 degree steps towards both directions. The output of the computer vision script is then compared to the actual rotation which is graphically determined by using a screenshot of the Raspberry Pi Camera and a photo editing software. The result can be seen in the Table 4.1

	angle from script in degrees	angle graphically determined in degrees
position 1	0.9	0
position 2	10.2	9.7
position 3	29.8	29.2
position 4	58.3	58.6
position 5	-29.8	29.8

Table 4.1: Yaw accuracy result of the computer vision algorithm. The landing pad is rotated in both directions and the output of the script is compared to the actual value. This value is computed graphically by measuring the rotation of the landing pad with the help of a screenshot from the Raspberry Pie Camera.

The key observation of Table 4.1 is that the yaw rotation computed by the script matches the actual rotation with an error always smaller than 1 degree in both directions. This fact indicates that the computer vision yaw accuracy in our controlled environment is accurate enough for a precise landing.

For testing the x and y position correctness, the same measurements are repeated except for replacing the rotation with an actual displacement in x and y direction of the frame. This way we can verify whether a movement in one direction is generating the same output as the same movement in the other direction, except for the sign. This way we can verify if our script computes feasible results for movement commands. Table 4.2 shows the result of this measurement.

	position from script in pixels	position graphically determined in cm
position 1	(0,0)	(0,0)
position 2	(0,7)	(0,1)
position 3	(0,20)	(0,4)
position 4	(7,0)	(1,0)
position 5	(-7, -28)	(-1,-4)

Table 4.2: position accuracy result of the computer vision algorithm. The landing pad is moved in x and y direction and the output of the script is compared to the actual displacement. This value is computed graphically by measuring the distance between center position and the green center coordinates of the landing pad with the help of a screenshot from the Raspberry Pi camera.

When looking at Table 4.2, it is noticeable that 1 cm of movement in either x and y direction correspond a movement of around 7 pixels according to the script output. All five positions of the markers show a maximum aberrance of one pixel, which is accurate enough to achieve the desired precision landing performance of the drone of less than 5 cm.

4.0.2 Test 2: Computer Vision Detection Reliability

The setup of this Test is the same as in Test 4.0.1. Instead of changing the position and rotation of the marker sheet, we now observe the output of the script in different lightning conditions. On Figure 4.4 one can see that a simple shadow on the blue marker results in failed recognition by the camera. The same happens when covering the green mark with a clear tape to increase the reflectivity of the green mark which can be seen in Figure 4.5

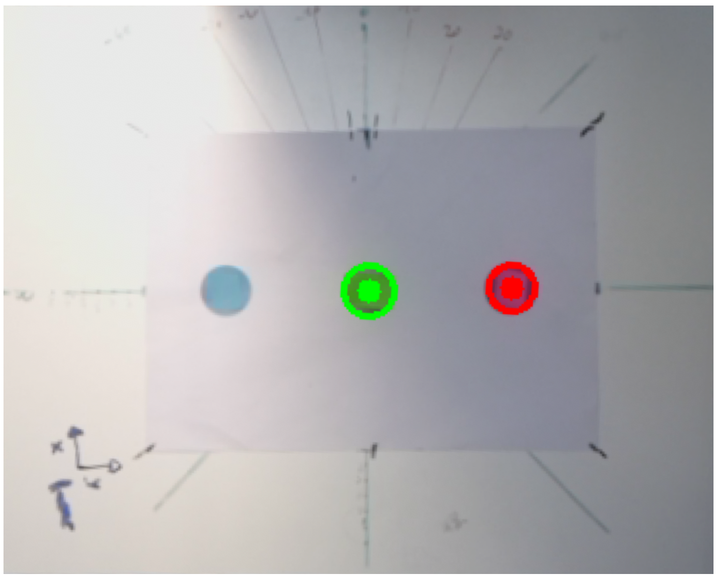


Figure 4.4: The result of placing a shadow on the landing pad. The computer vision based detection doesn't detect the blue point because of the shadow. This is a limitation of computer vision based techniques.

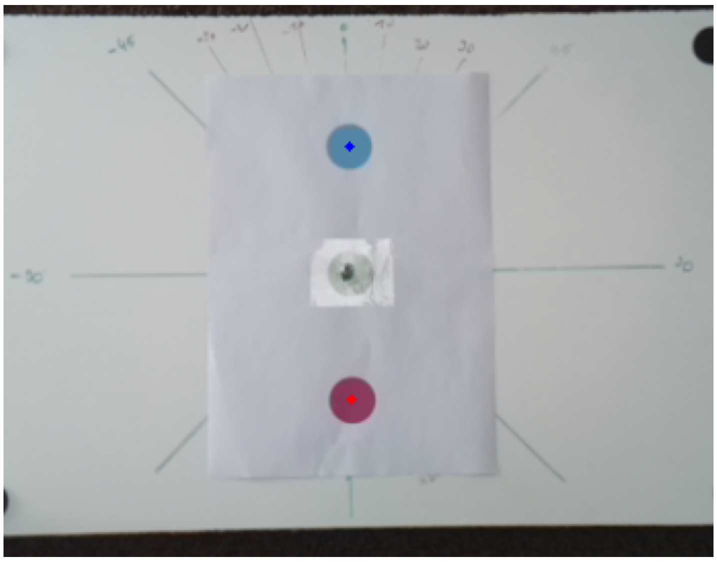


Figure 4.5: The result of putting a clear tape over the green point to increase its reflectivity. The tape is enough to introduce a failure in the detection algorithm.

Given the circumstance that a shadow on the landing platform can introduce a failure and lead to an inaccurate or aborted landing, this system seems to be prone to bad lightning conditions. When thinking about the final use case, even the drone itself could introduce a shadow on the landing platform while performing the approach so that the marker detection might fail. This is a major drawback since a failed detection because of the drones own shadow means bad reliability for the whole system and of course unacceptable landing accuracy.

4.0.3 Theoretical Experiment: Computation of Velocity Components

Our script computes the velocity vector by allocating a velocity to a certain displacement. For example, a displacement in x direction of 3 m (see figure 3.1.2) corresponds to a velocity component of 2 m/s in x direction, which is also defined as the maximum velocity during the landing. This velocity is now mapped in a linear matter so that the displacement of 0 cm corresponds to a velocity of 0 m/s and a displacement of 1.5 m corresponds to a velocity of 1 m/s.

The initial position of the drone for this experiment is a displacement of 1m in x direction. When the algorithm starts, this displacement is converted into a velocity command of 0.66 m/s in x direction so the drone will tilt towards the center of the landing platform to achieve the desired velocity. This rotation around the roll axis of the drone leads to a different camera view on the landing pad, in such the position of the green marker increases its distance from the center of the frame. This in turn leads to a higher desired velocity command in x direction to compensate fort the additional displacement (which is based on the roll angle instead of an actual displacement). Of course this only builds up until the maximum allowed velocity is reached but it might still be a reason why this script has to be adapted to be functional on a flying drone.

Chapter 5

Summary

Overall, our contribution consists of the production of a drone platform with PX4 flight controller software which is able to perform automated flight missions. Moreover we wrote a script which uses computer vision techniques on a downwards facing camera's image to recognize a landing pad. This script also computes velocity vectors which are sent to the flight controller to align the drone to the landing platform.

Through our experiments we can conclude that the landing pad detection is accurate enough for a landing within 5cm radius. The yaw alignment of the drone is also accomplishabel with a computer vision based code. However, the accuracy of the landing is heavily dependent on the lightning conditions. For example shadows or darker surroundings lead to marker failures in the detection algorithm.

This leads us to the conclusion that the system we designed for landing platform detection is a good solution for accurate results but it is not reliable enough for save application. By virtue of issues with the PX4 flight controller software, we didn't conduct any landing tests in a real flight, nonetheless our drone is a solid starting point for future projects in the area of drone automation and computer vision based tasks.

5.0.1 Future Work

The biggest drawback of our implementation is the lack of trustworthiness in the detection algorithm. To overcome this, one could improve the marker layout, for example by using a black marker on white background instead of different colors to enable better performance in suboptimal lightning conditions. Another possibility to increase the reliability is to change the marker design and the detection algorithm such that partial detection of the marker is enough for accurate positioning.

Another approach for increasing the reliability and exactness of current implementation is to fuse the IR beacon detection together with a marker for computer vision detection to achieve maximum accuracy and dependability.

Furthermore one may adjust the computation of the velocity vector of the markers' pixel coordinates. As illustrated in Chapter 4.0.3, the computation of the velocity needs some additional code to work properly. Mounting the camera onto a gimbal to make sure the camera is always aligned with the ground plane is another workaround for this issue. Additionally, instead of using a filter to smoothen the desired movements, introducing a PID controller to integrate feedback loop controlling will help increasing the system performance too.

Bibliography

[1] Katsuya Fujii, Keita Higuchi and Jun Rekimoto: Endless flyer: A continuous flying drone with automatic battery replacement. IEEE Xplore (Jan 30 2014)

[2] Developer Team, PX4: Px4 precision landing with ir beacon. https://docs.px4.io/en/advanced_features/precland.html

[3] Pixhawk Project Developers: Pixhawk 2 Flight Controller Hardware. https://pixhawk.org/modules/pixhawk2

[4] Raspberry Pi Organisation: Raspberry Pi Minicomputer. https://www.raspberrypi.org

[5] Developer Team, Emlid: Picture of Navio2 mounted to a Raspberry Pi. http://linuxgizmos.com/files/emlid_navio2_pi_angle1.jpg

[6] Hobbywing: Website of the Hobbywing ESC. http://www.hobbywing.com/goods.php?id=588

[7] Tiger Motors: Website of the tiger motors. http://store-en.tmotor.com/goods.php?id=718

[8] FPVRacing.ch: 7 inch Propellers. https://fpvracing.ch/de/propeller/2109-hqprop-dp-7x35x3-v1s-durable-pc-propeller-schwarz.html

[9] Frsky RC: Website of frsky xsr receiver. https://www.frsky-rc.com/product/xsr/

[10] Synosytems: Website of the telemetry module. https://synosystems.de/de/px4-open-hardware-kompatible/278-433-mhz-telemetrie-kit-mavlink-fur-pixhawk-px4-kompatible.html

[11] Conrad Schweiz: Shoppage of the lipo battery. https://www.conrad.ch/de/swaytronic-modellbau-akkupack-lipo-185-v-3800-mah-zellen-zahl-5-35-c-softcase-html

[12] Developer Team, APM: Comparison of + and x Motor Layout. http://ardupilot.org/copter/_images/MOTORS_QuadX_QuadPlus.jpg

[13] : Autodesk fusion 360 as design environment. https://www.autodesk.com/products/fusion-360/overview Accessed: 2018-06-22.

[14] Developer Team, Emlid: Docomentation of Navio 2 Autopilot. https://docs.emlid.com/navio2/

[15] Developer Team, PX4: PX4 Architecture. https://dev.px4.io/en/concept/architecture.html

[16] Developer Team, PX4: PID Controller. https://commons.wikimedia.org/wiki/File:PID.svg

[17] Developer Team, PX4: PX4 Controller Tuning Guide. https://docs.px4.io/en/advanced_config/pid_tuning_guide_multicopter.html

[18] Wikipedia: Picture of Yaw, Pitch and Roll in an Aircraft. https://en.wikipedia.org/wiki/Aircraft_principal_axes#/media/File:Yaw_Axis_Corrected.svg

[19] Developer Team, PX4: Docomentation of Beacon Precision Landing with PX4. http://docs.irlock.com

[20] Developer Team, OpenCV: Docomentation of OpenCV Library. https://opencv.org

[21] Developer Team, 3DR: Docomentation of Dronekit. http://dronekit.io

[22] Adrian Rosebrock: Ball tracking tutorial for opencv on which the marker detection code is based. https://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/

[23] Ardupilot Project Developers: Ardupilot Flight Controller Software. https://http://ardupilot.org

[24] Developer Team, PX4: Barometer Bug Fix. https://github.com/PX4/Firmware/pull/9721

[25] Raspberry Pi Organisation: Tutorial to expanding the filesystem using raspi-config. https://www.raspberrypi.org/documentation/configuration/raspi-config.md Accessed: 2018-06-21.

[26] Developers Team, PX4: Setting up the raspberry pi. https://docs.px4.io/en/flight_controller/raspberry_pi_navio2.html Accessed: 2018-06-21.

[27] Developers Team, PX4: Setting up the linux toolchain. https://dev.px4.io/en/setup/dev_env_linux.html Accessed: 2018-06-21.

[28] Developers Team, PX4: Cross-compiling px4 firmware. https://dev.px4.io/en/setup/building_px4.html Accessed: 2018-06-21.

Appendix A

Appendix Chapter

A.1 Raspberry Pi 3 Model B Installation

This is a guide to installing Debian (Jessie) with the PX4 Firmware on the Raspberry Pi by cross-compiling from another Linux machine.

- Flash the OS Image *Jessie* to the SD card of the *Raspberry Pi* (Using Etcher for example)
- Make sure that connecting over ssh to the Raspberry Pi is possible.
  $ ssh pi@navio
- Make sure to expand the file system of the Raspberry PI with raspi-config [25].
- Edit */boot/config.txt* by commenting the line enabling the navio-rgb overlay [26].
- Install the development toolchain for Linux to cross-compile for the Raspberry Pi [27]
- Download the PX4 Firmware on the Linux machine.
  $ git clone https://github.com/PX4/Firmware.git
- Compile the PX4 Firmware for the Raspberry Pi on the Linux machine [28].
  $ cd Firmware
  $ make posix_rpi_cross
- Get the IPv4 address of the Raspberry Pi.
  $ ifconfig
- Set the IP of the Raspberry Pi on the Linux machine.
  $ export AUTOPILOT_HOST=192.168.X.X
- Upload the PX4 Firmware from the Linux machine [28].
  $ cd Firmware
  $ make posix_rpi_cross upload

- Run PX4 in terminal of the Raspberry Pi.
  ```
  $ sudo ./px4 px4.config
  ```

- PX4 can run on autostart by adding the following line to */etc/rc.local* just above *exit 0*.
  ```
  $ cd /home/pi && ./px4 -d px4.config > px4.log
  ```

More details can be found on the *dev.px4.io* homepage.