BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS
FACULTY OF MECHANICAL ENGINEERING
DEPT. OF MECHATRONICS, OPTICS AND ENGINEERING INFORMATICS

TAMÁS SZEPESSY

THESIS

Optical drone control

Consultant:
József Molnár
Master teacher

Theme leader:
Dr. Péter Tamás
c. university professor

Budapest, 2019

ABSTRACT

The goal of this thesis was to establish an automatic control system for a UAV, that navigates and collects data only by using the built-in camera. Our program is written in Python, using the OpenCV package for basic computer vision algorithms and ArUco markers for the measurements.

At start, we take a glance at some possibilities on controlling a drone with optical sensors, such as Lidar or Time-of-Flight cameras. On a much more detailed note, we summarize other studies on depth-sensing and mapping with a digital camera: stereovision, neural networks and probability models. As there seems to be a need for robust position tracking of UAVs, AR markers have been chosen for further work.

We chose a DJI Tello quadcopter with Wi-Fi connectivity and a 720p camera. Separate UDP sockets are being used to command the drone, get the video stream or read sensor state values. It can be easily controlled from a computer through various SDK commands, from which the RC control is being adapted for this project.

After the first successful tests, the mathematical model behind the system must be built. With the help of coordinate transformations, one can pass on camera positions between a markers world coordinates and the chosen global coordinate system. Each marker's coordinate system must have an origin point in global coordinates and a rotation, which takes the base to the global bases. The transformations are being done with matrix operations.

With the created program, the quadcopter can navigate itself through a path set by ArUco markers. The control automatically passes on the targeted marker, to which the program's numeric PID controller must position itself. Throughout the automatic navigation, the system collects and stores position data and writes a video of the flight.

The created system has been tested for its accuracy with a motion capture system provided by the faculty. During the 5 test flights, the drone's navigation did not encounter any problem, the system successfully navigated in the environment and measured the UAV's position. The stored data of the two systems have then been compared. Although the AR measurement system has a significant bit of noise, a Kalman filter can smooth out the movement points in post-processing. The filtered dataset is then similar in its characteristics to the motion capture's reference measurements, with only a few centimetres difference.

As conclusion, we can say, that the created system achieved its goal. We adapted an autonomous navigation system for UAVs indoors. Post processed data sets are well filtered enough, to be counted as adequate measurements. In the future the program can be advanced to present real-time flight data. We can conclude, that such a system can be used for industrial drones also, where the AR markers' positions are formerly known.

Keywords: *UAV, autonomous navigation, AR marker, positioning, measurements*

TABLE OF CONTENTS

LIST OF NOMINATIONS

The table gives the *English* and Hungarian names of the repeatedly occurring notations and, for physical quantities, their units of measurement. Where possible, the notation of each quantity is the same as that adopted in national and international literature. Explanations of rarely used notations are given at their first occurrence.

Latin letters

Apply at	Description, comment, value	Unit of measurement
g	gravitational acceleration (9,81)	m/s2
M	bold capital letter indicates matrix	SI
v	bold lower case indicates vector	SI

Indexes, eaters

Apply at	Designation, interpretation
0	denotes the vector of the global coordinate system
i	overall running index (integer)
m	*m.* index of elements in an array
n	*n.* index of elements in an array
o	indicates a vector pointing to the origin

1. INTRODUCTION

1.1. Objectives

Unmanned Aerial Vehicles (UAVs), or drones as they are more commonly known, are becoming more and more widespread, one reason being their miniaturisation and the new category they have created: the MAV (Micro Aerial Vehicle). Quadcopters, which can be used indoors, can easily navigate between different types of equipment due to their small size. In a wide range of applications, such as visual observation, surveillance or even filming, the small size allows for better manoeuvrability and the ability to fly and explore smaller, inaccessible areas.

However, the small size comes at the price of having fewer sensors, as weight is also a critical factor. A single built-in camera with appropriate software can be used to replace the many sensors needed for obstacle avoidance, which is essential for autonomous operation. Nowadays, even the on-board computers of compact devices do not have such computing power, so we can use an external computer to do the image processing and send back the control data.

The aim of this thesis is to enable the chosen drone to autonomously create a flight map based on the camera image and navigate between obstacles, thus creating an indoor navigation system in which the drone can navigate.

1.2. Overview

The thesis is structured according to the points in the terms of reference. In the literature review, optical sensors and specifically drone control based on the principle of camera image processing are discussed. This is followed by a justification of the program environment and the choice of the drone, detailing the control and data extraction methods of the chosen drone.

The main part of the thesis is a detailed description of the operating principle of the image processing and control program, the mathematical methods used, the algorithms developed and the structure of the control program. Finally, by carrying out operational tests on a calibrated system and comparing the measurements of the completed program with these reference points, the paper draws a conclusion and considers possible further developments.

2. LITERATURE REVIEW /OUTCOMES

Before choosing the optical positioning and navigation method to be used, we had to explore the different options. We looked at methods that might be suitable for drone navigation. Their applicability to microdrones is investigated.

2.1. Optical principle sensors

A variety of sensors are suitable for obstacle avoidance, such as LIDAR [1, 2] (Light Detection and Ranging) laser light-based remote sensing sensors, which provide high-accuracy, rapid imaging of the environment. It can provide information on the distance of any reflecting surface. Laser light ranging can also provide information on the geometry of an object by rapidly and systematically observing the reflection of the emitted laser pulses in rapid succession, and by measuring the phase difference between the emitted and the return laser light. Such laser scanning is often used in larger drones. These are more expensive devices due to the technology and are more used in industry. In addition to obstacle avoidance and navigation, machines equipped with LIDAR sensors can also take accurate spatial measurements of large objects. The sensor's dimensions and the necessary circuitry do not allow its application in our problem, and its use is more recommended in outdoor conditions.

A cheaper solution could be ultrasonic and infrared [3] sensors that monitor the reflection time of the emitted signals. In the case of the former, the operation is slower because it works on the principle of sound propagation rather than light propagation and returns only a single depth datum, not a depth map of the object. Ultrasonic sensors are the preferred choice for obstacle avoidance due to their cost-effectiveness, but they must be mounted on the vehicle in several directions and in several numbers.

ToF (Time-of-Flight) cameras [4], which can assign depth values to images pixel by pixel from the phase difference of the emitted infrared wavelength of the auxiliary light measured pixel by pixel, are also based on the reflection principle. These sensors can operate as accurately and rapidly as LIDAR sensors under suitable environmental conditions. [5] While LIDAR only returns spatial data, a ToF camera can also store textures of surfaces (like an ordinary digital camera), which can facilitate subsequent data processing. Depth cameras can also be used indoors and their spatial resolution depends on the number of pixels. The size of the sensor is smaller than its laser counterpart and the control circuitry is simpler, but for the weight reduction reason mentioned in the introduction, we will only work with the image from the built-in camera.

2.2. Obstacle avoidance and orientation based on camera images

2.2.1. WITH AN UPGRADED NEURAL NETWORK

Machine learning and neural networks [6] were among the camera-based methods investigated, but were discarded due to the time- and hardware-intensive learning processes. The essence of the method is that a learned neural network can assign approximate depth values to each pixel based on the location of objects from just a single image. Learning patterns can be obtained from a pair of LIDAR and digital cameras placed side by side. By inputting a sufficient number of these learning pairs, the neural network can establish a relationship between the pixels in the images and the depth values obtained from the LIDAR (or stereo camera). The mesh can be learned using supervised LIDAR-based learning and unsupervised stereo pair-based learning based on a probabilistic principle. Supervised learning is often too rigorous and unsupervised learning gives inaccurate results, so it is recommended to use semi-supervised learning [7] or to compare the results obtained by the mesh with a calibrated depth detection system during operation. [8]

Thus, similar to human vision, it can retrieve depth values from the surroundings and locations of objects (pixel sets) and build a depth map similar to the one shown in *Figure 1*. Its accuracy is still experimental, such a system is still subject to many errors, but it has been used in drone control applications [9] The depth maps in Figure 1 show the difference between the different methods and the accuracy of the method.

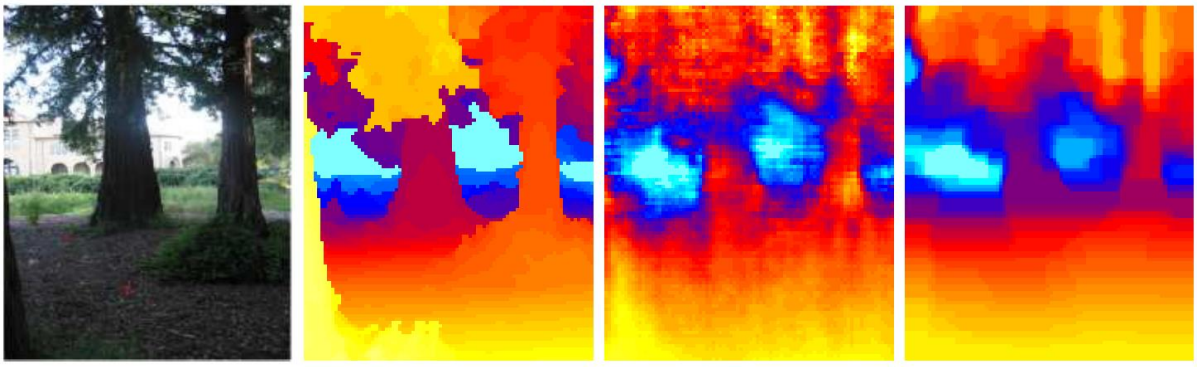


Figure 1: The loaded image (left), the real depth map based on the sensors (centre left), and finally the depth map constructed by the Gaussian model (centre right) and the Laplace model (right) [6, 1.Im]

Such a neural network takes several factors into account. [10] The explanation of the elements in *Figure 2*, i.e. the factors considered, is as follows:

- **occlusion:** an object cannot be seen over its entire extent because it is occluded by another object, the occluded object is further away.
- **relative size: the** relative size of two objects is different, with the smaller object being further away.
- **cast shadow:** The shadow cast by an object relative to the object, if it is a visible shadow.

- **shading:** the spatial extent of an object can be inferred from the shadows cast at its edges. The darker the edge, the more curved it is.
- **distance to horizon:** objects further away from the horizon are closer to the camera.
- **texture gradient:** more distant parts of an object's texture are lost. This is due to the resolution of the camera (our eyes in the case of humans), an object with a more detailed texture is likely to be closer.
- **linear perspective:** like the human perspective perception, objects are perceived as getting smaller and smaller as they move away from the human perspective. Narrower geometry indicates a more distant location.

However, teaching a neural network is time-consuming due to the large amount of image data, and would take weeks to learn on a single machine, even with a powerful graphics card. Obtaining the right amount of training samples is also resource intensive. A problem, once a trained mesh is in place, is to make it work in real time - essential for drone control - so that calibration acceleration does not cause the system to lose too much data, which can lead to collisions. During slow flight, the method is still able to avoid obstacles, but it does not treat them as obstacles with precise contours, only as a set of points closer in relative position.

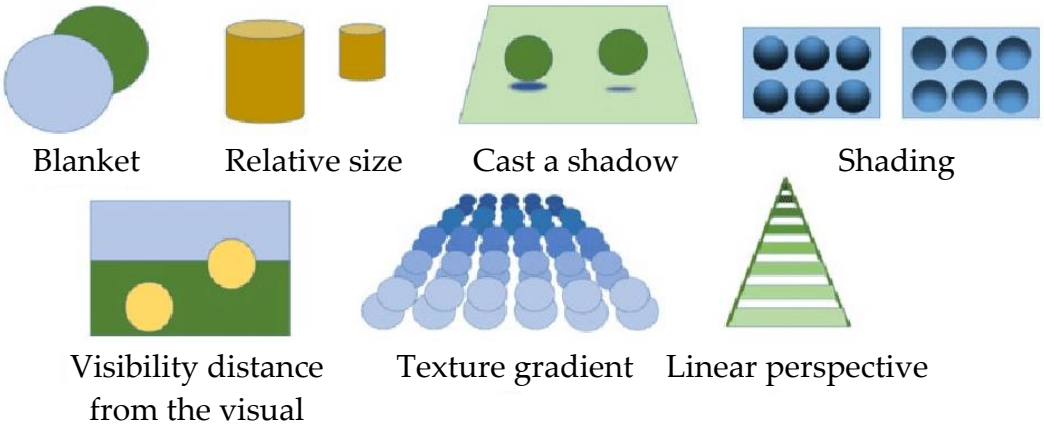


Figure 2: Factors considered by monocular depth perception [10, 2.Im]

2.2.2. STEREOLENSING PRINCIPLE

One of the oldest camera depth-sensing techniques is based on the principle of stereopairs, similar to the imaging of the human eye. [11] Two cameras fixed side by side can be calibrated simultaneously, so that the transformation matrices of the cameras can be calculated and a depth map can be constructed from the corresponding pixels identified in the two separate images using epipolar lines. The role of the epipolar line is to simplify and speed up the process of finding pairs of points between images. Using the mapping matrices, we can calculate, for example, which line a pixel in the left image will lie on in the right image after projections. This way, we don't have to scan the whole image area, just pixels along a line. To check the similarity of pixels, we can use, for example, the Fast Hessian (FH) operator, which is based on the determinant of the Hessian matrix.

The advantage of using stereolithography is that there are many free-to-use solutions available, including in the OpenCV library. It has also been used for drone control, but its reliability at high speeds is very low, as shown in the depth map in *Figure 3.* Perfect calibration and solid camera fixation are required to produce a depth map, otherwise the system will provide erroneous data. Due to the FH operator, which also uses the pixel environment, the technology can only compare surfaces with texture. However, in the case of the drone used in this thesis, only one camera is available and it is not possible to mount another one and attach it to the device.

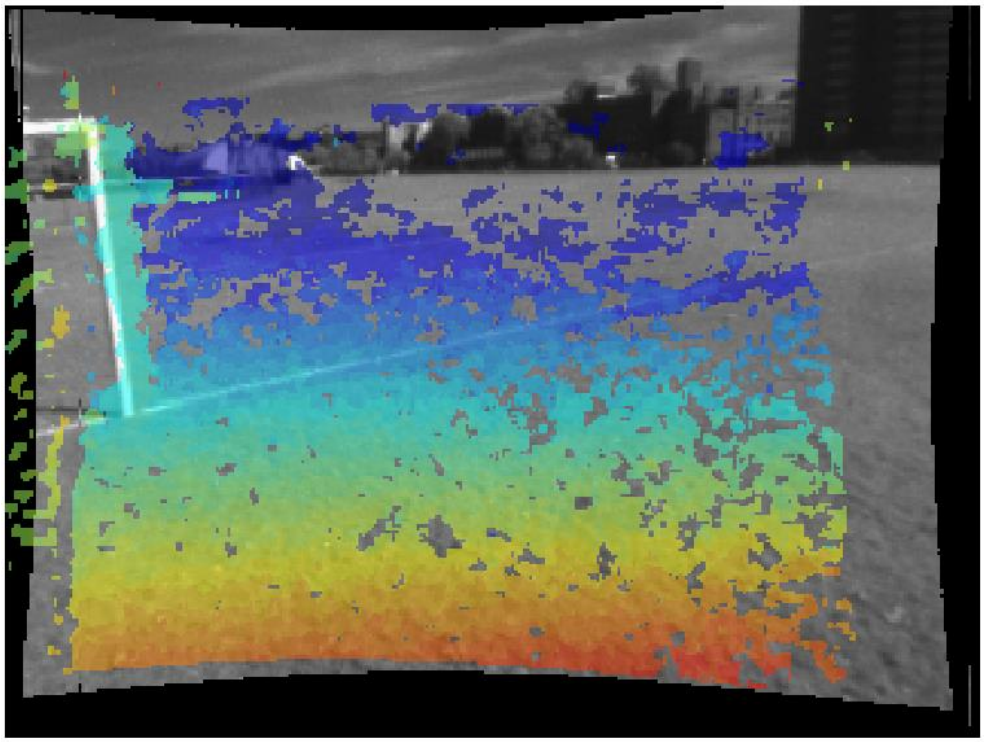


Figure 3: Depth map from a drone-mounted stereo pair (red - near / blue - far) [11, 3.Im]

A further solution could have been a single-camera, but prism-based stereo view, which uses the optics of a single camera split by the prism to achieve two views. [12] In this way, depth values could be obtained by building a disparity map on the principle of fixed stereo cameras with a single camera image divided into two virtual cameras, as shown in the drawing in *Figure 4.* Using the equations for the geometric optical refraction of the prism, a transformation matrix can be written for the right and left virtual cameras. Then, the positions of the two virtual cameras are obtained, from which a depth map can be computed from a stereo pair of calibrated cameras, based on epipolar geometries.

In our application, the problem with this method is to fix the prism firmly on the drone, as the slightest movement will completely ruin the calibration and render the stereo pairs unusable. Furthermore, the horizontal field of view is reduced, the vertical extent of the resulting image will be longer than the horizontal, which is essential for our drone when navigating.

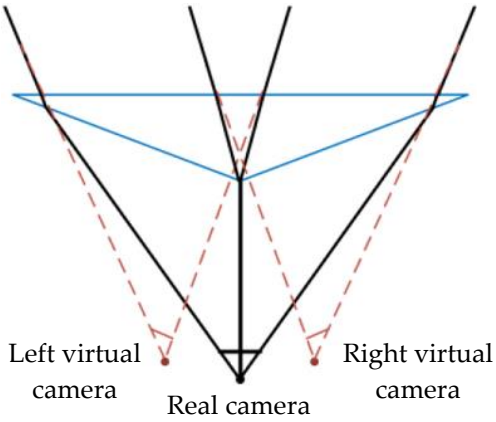


Figure 4: Two virtual camera images created by the prism splitting the image [4.Im]

2.2.3. BASED ON MOVING CAMERA IMAGE

A solution that has been considered for a long time is a depth map based on the camera positions calculated from the drone's movement and the epipolar geometries fitted to the images taken at a given moment in time. [13] The idea is that if you don't have two cameras, you can move the drone to take a picture of an object from multiple angles. Unlike stereolithography, we take tens of images rather than two and compare the pixels with each other and with the camera position at the time the image was taken. In the images, it is important to select key points - similar to the FH operators for stereo pairs - that are sure to be the same point in another image. This way, we can obtain a depth image based on multiple viewpoints and camera positions. (*Figure 5*)

What was surprising, based on the *Alvarez et al.* study [13], was that the drone's positioning errors during hovering provide enough displacement to create a depth map, and 30 images are enough to extract the necessary data. If the collected images are processed on a graphics card, the process is significantly faster, with image acquisition taking 1 second at 30 FPS and processing taking 0.5 seconds. In total, the system calculates a 64-layer depth map in 1.5 seconds. For the time being, this time makes the drone move in sections, which is acceptable for an indoor, low-speed application.

Again, tests on the accuracy of the system show that it only gives good depth values for well-textured, separable surfaces. In a pre-set environment (as shown in Figure 5) the drone was able to navigate between obstacles with a high degree of confidence (90-100%). Flying through a doorway, with its larger, smoother surfaces, caused problems: it only passed the obstacle with 50% efficiency.

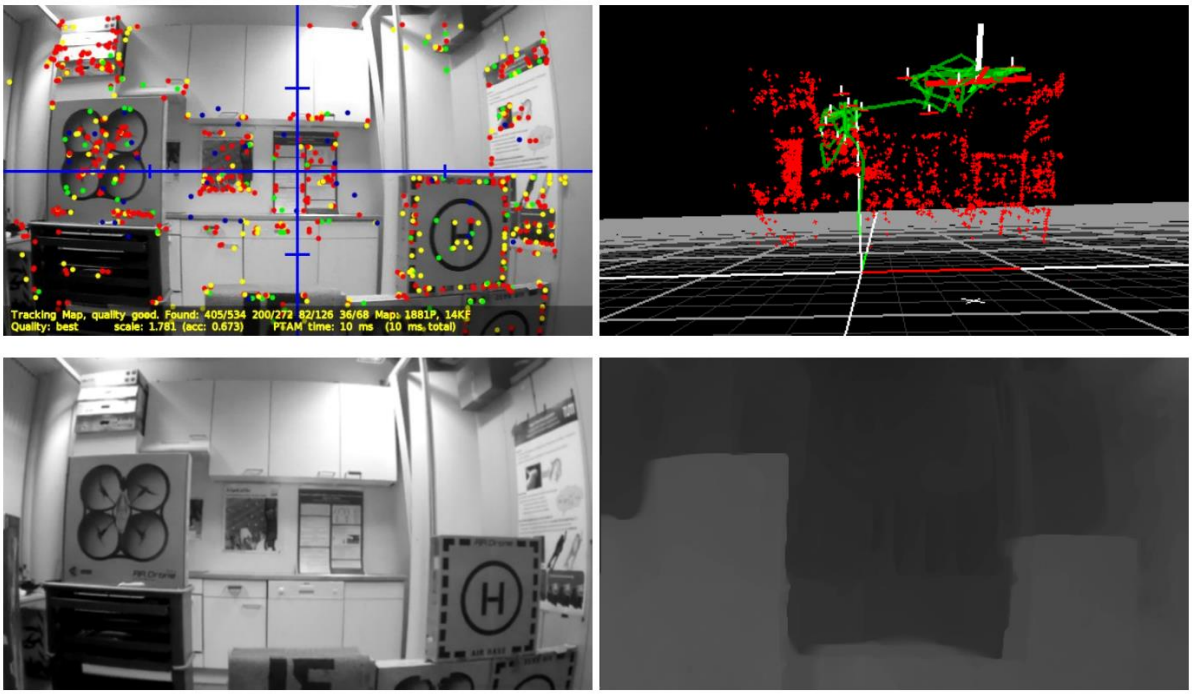


Figure 5: Top row: pixels tracked during the movement and the depth map derived from themBottom row: original camera image and depth map derived from the movement [13, 5.Im]

This, however, requires knowledge of the camera's position and orientation, attempts to determine which are discussed in some detail in *section 3.4*. In the present case, this idea has also been discarded because of the noise from the accelerometers and the errors that can be introduced.

2.2.4. POSITIONING WITH MARKERS

As discussed *in chapter 2.2.3* the objective of a low-cost camera-based positioning system was formulated. This would allow the position of the drone to be tracked, as well as the angular rotation and camera position, as these can only be obtained from accelerometer data with a very high degree of inaccuracy. This is how the ArUco marker using augmented reality from the OpenCV open source programming toolkit was finally chosen. These have been used in drone control [14, 15], to obtain position and angular rotation in the camera space, and to communicate control data to the drone using the marker sequence number, so that it can even autonomously fly in a properly mapped space.

A student colleague and I have already used ArUco markers for distance sensing in a previous university project. [16] We successfully found that the functions in the OpenCV ArUco library can be used for marker recognition under a variety of conditions, including dim, low-light conditions. Regarding the accuracy of the measured marker positions, we found that we are able to detect markers over a wide range with an average accuracy of ±0.05 m even for small side markers (0.03 m).

In our project, we also found that the specifications of the camera used can be a determining factor in positioning with markers. Thus, wide-angle cameras may result in more inaccurate measurements due to higher lens distortion, just as autofocus

cameras may corrupt the camera calibration matrix to some extent during continuous focus adjustment. In addition, a larger camera aperture, if adhered to, will obviously produce smaller errors, as a larger number of pixels then means smaller measurement units.

Markers are often used to determine the position and navigation of autonomous robots. Displacements relative to fixed markers can be determined by geometric transformations in Descartes space, and even camera rotation can be calculated from the rotation of the markers. Since the ArUco markers are detected sufficiently fast, the system can operate in real time.

Previous studies [17] have investigated the camera position of a robot capable of autonomous locomotion using markers. The robot was able to navigate in the space bounded by the markers and recalculate its position from the marker space to the global space. The accuracy of the method depended on the number of markers seen and the side length of the marker. With more markers, more accurate data could be obtained by averaging the read positions, and better measurements could be obtained by using larger markers.

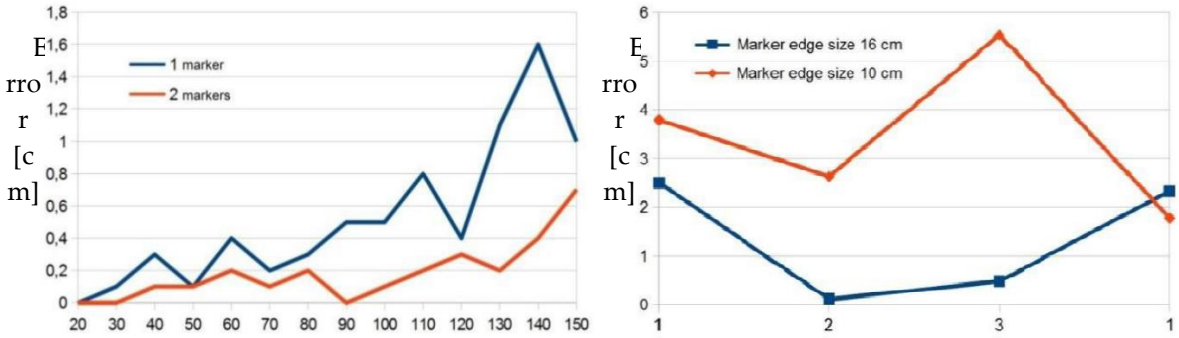


Figure 6: Accuracy of the data obtained from the markers.
a) Accuracy is measured from the camera marker(s) distance from the camera to the marker when using 1 and 2 markers. b) At each known position of the robot the measurement deviations for different marker side lengths (10 cm and 16 cm) [17, 6.Im].

Taking these results into account is necessary to create a proprietary system in which a drone can navigate itself between markers while collecting data on its movements. In designing the system, let us adopt the study by *Babinec et al* [17] and base the choice of marker properties on it.

3. EXPERIMENTAL DESIGN

3.1. Choosing a drone

After looking at the features of the possible programmable drones, the device of choice is the *Tello* quadcopter, developed by Ryze but supported by DJI. It has an Intel processor and a built-in 5 MP resolution camera, can communicate via wifi, UDP connection, has a clean flight time of about 13 minutes and is capable of accurate positioning in windless environments, stable hovering in one place. The device weighs only 89 grams with propeller guards and battery inserted, which shows that weight is a critical factor.

Position control and altitude control is done by several sensors. A built-in barometer provides altitude data to the drone, as does an infrared sensor and ToF camera mounted in the bottom of the device, which also helps to maintain in-plane position. (*Figure 7*) An IMU sensor is also essential, providing three-way acceleration and angular position data.

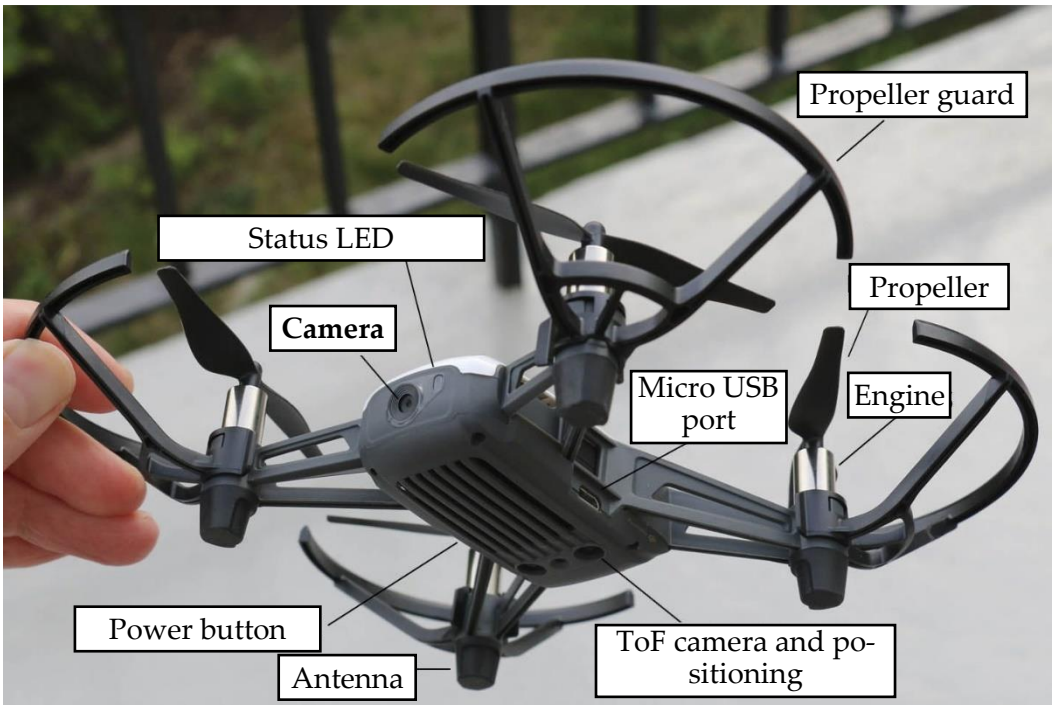


Figure 7: Image of the DJI Tello drone, showing its components [7.Im]

An SDK written in *Python is* available for the drone, which contains the details of UDP communication. By connecting to the drone's own wifi network, you can control it with various commands and read data from it. The maximum packet size of the UDP connection is smaller than the camera image transmission, so the drone transmits it in multiple packets.

3.2. Programming language and toolset

The SDK provided for the drone requires the use of the Python programming language, and beyond that the freely available GitHub library [16] of *Damiá Fuentes* was used as a starting point. Image processing is based on the open source *OpenCV* (Open Computer Vision) module package version 4.1.0. Furthermore, the user interface is *pyGame*, and the plotting of the measurement data is done using the *Matplotlib* package.

Because of the higher level programming language, the operation will unfortunately be slower than if you were working in C++. This is a critical factor for obstacle avoidance, but for now we are working in a static environment pre-set for our tests, so dynamic obstacle avoidance should not be expected.

3.3. How to control the Tello drone

The documentation provided by the manufacturer details how to control the drone using a Python program. [19] Connected to the drone's own wifi network, a UDP connection must be opened to control the drone via 192.168.10.1 and port 8889. A *"command"* command must always be sent to the drone before starting the control, which switches to a state where it is waiting for commands.

It is also possible to read the flight controller status from the drone, which is also used by the device's firmware, by opening the server 0.0.0.0 on port 8890. This was not included in the GitHub library mentioned above, so we added it to the program. We opened a new UDP socket connection with the aforementioned server and port data running on a separate thread, similar to the background reception of drone control responses.

Code snippet 1: Implementation of the status thread (*tello.py/class Tello/def __init__()*)

```
# Monitor the drone's response to control commands
thread = threading.Thread(target=self.run_udp_receiver, args=())
thread.daemon = True
thread.start()

# Thread of the server that reads the drone's status
thread_st = threading.Thread(target=self.get_read_state, args=())
thread_st.daemon = True
thread_st.start()
```

Code snippet 1 shows the thread handler contained in the Python *Threading* library, which can be assigned to a specific function that will run on a separate thread from the main program. The thread that reads the state points to the *get_read_state()* function of the Tello class, and starts it in a separate thread, so it runs in the background. You should enable the `daemon = True` option, so that the thread will start as a daemon, i.e. if the higher-level program exits, the daemon threads will exit as well. If this is not turned on, you will need to handle thread exits separately.

The states are broadcast by the drone on the aforementioned server, reading them will not slow down the camera image readout from the drone, since the drone will broadcast the string of states after entering *command* mode even if we do not read it. The official documentation [19] specifies the format in which the drone sends out the status of its sensors. It transmits a single, coherent, semicolon-delimited ASCII text file over the wireless UDP connection to the receiving server, in the following format:

```
"pitch:%d;roll:%d;yaw:%d;vgx:%d;vgy:%d;vgz:%d;templ:%d;temph:%d;tof:%d;h:%d;
bat:%d; baro:%.2f;time:%d;agx:%.2f;agy:%.2f;agz:%.2f;\r\n"
```

Where the individual state values (the interpretation of the coordinates is given in Figure *8*):

- *pitch, roll, yaw: the* intrinsic rotation of the drone about the x, y and z axes, measured as a whole angle, clockwise positive, according to the left-hand rule
- *vgx, vgy, vgz: the* speed of the drone set in x, y, z directions in cm/s (Not the actual measurable speed values, but the value attached to the drone speed variable during control.)
- *temp:* the lowest temperature in C°
- *temph:* the highest temperature in C°
- *tof:* height in cm of the Time-of-Flight camera at the bottom of the drone
- *h:* height in cm
- *bat:* remaining battery capacity of the drone in whole percentage
- *baro:* height value based on the installed barometer in cm
- *time:* time the engines are on (flight time) in seconds
- *agx, brain, agz:* acceleration of the drone in x, y, z direction from the acceleration sensor interpreted in 0.001g, where g is the acceleration of gravity

This can be split into an array along the semicolons using Python *string.split("; ")*, and then the values after the colon in the array are converted to a numeric type in the appropriate format. Only the required values are converted to a number, and then the values are loaded into a list which is passed to a line. In Python's thread handling, only the *Queue* type is capable of secure thread-to-thread communication; if this is not used to exchange data between threads in the program, the program may freeze. The status read function is shown in

Code snippet *2*. For Euler angles, a negative sign is required for the transition to the right-hand rule.

You also have the option to read the status during control by sending a query to the drone as a command, which it will respond to via UDP connection. These query words give the same sensor states as the state readout (e.g. : *"acceleration?"*, *"speed?"*, *"battery?"*, *"tof?*" ...). The use of this function was also considered, but tests show that this method gives less reliable results. This communication, unlike status readout, slows down and overloads the other functions of the drone, as it needs to provide a separate processor cortex to respond to them. Thus, since we cannot get reliable results even with thread handling, this feature is not useful for our purposes.

Code snippet 2: Function to read drone states (*tello.py/class Tello/def get_read_state()*)

```
def get_read_state(self):
    while True:
        time.sleep(1/25)
        try:
            state_temp, _ = self.stateSocket.recvfrom(1024)
            self.state = state_temp.decode('ASCII'). split(";")
            pitch = -int(self.state[0][self.state[0].index(":")+1:])
            roll = -int(self.state[1][self.state[1].index(":")+1:])
            yaw = -int(self.state[2][self.state[2].index(":")+1:])
            tof = int(self.state[10][self.state[8].index(":")+1:])
            bat = int(self.state[10][self.state[10].index(":")+1:])
            self.data_queue.put([pitch, roll, yaw, tof, bat])
        except Exception as e:
            self.LOGGER.error(e)
            break
```

The camera image, which is essential to the task, is also broadcast via UDP, opening the server 0.0.0.0 via port 11111. Since the size of the encoded data of the image is larger than the maximum payload size of a UDP transmission, the image data is transmitted by the drone in blocks of 1460 bytes, the last block of which is smaller than 1460 bytes. [20] The broadcast video can be displayed after reception using the OpenCV package. The *ffmpeg* and *libh264decoder* codec packages provided in the official DJI-SDK documentation [20] are therefore not required in this case, testing the program under *Microsoft Windows* operating system.

The SDK control instructions can be divided into three groups:

1. management instructions
    a. return "*ok*" if satisfied
    b. *"error"* or an error message if you do not
2. reading instructions
    a. returns the value of the requested parameter
3. control by setting parameters
    a. return "*ok*" if satisfied
    b. *"error"* or an error message if you do not

The drone is sent the *"takeoff"* command, takes off, lands on *"land"*, *"streamon"* and *"streamoff"* commands to switch video streaming. The drone itself can be controlled per direction and simultaneously with a *"go x y z speed"* command. In this case, the drone will move at the specified speed relative to the given position in the direction specified by the three coordinates. The accuracy of this control is equal to the dimensions of the drone, so it cannot move less than 20 centimetres.

The firmware also provides the possibility to control the drone in an arc: by specifying two more points with x, y, z coordinates relative to the drone's current position, it will describe the arc defined by them, if the radius of the arc is between 0.5 and 10 meters.

Figure 8 was created to interpret the coordinates of the drone. We will not refer to this figure in the following, when we refer to the drone's directions, we mean the directions defined by it.

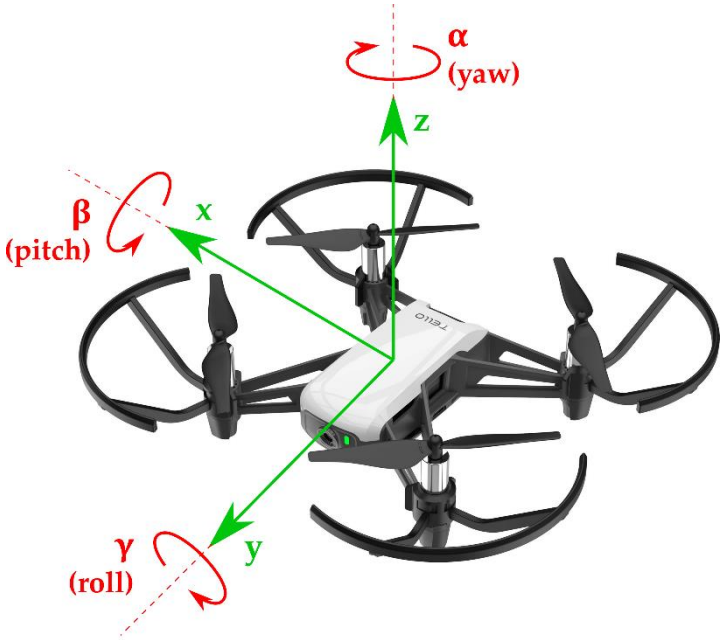


Figure 8: Directions defined when controlling the drone with the SDK

For this task, the most suitable control method based on the tests with the drone is the so-called RC control, which belongs to the group 3 of control modes. The drone's speeds in *x*, *y*, *z* and *yaw* can be set with the command *"rc x y z yaw"* with integer values between -100 and 100. (The *yaw* velocity here is also expressed in terms of the bal offset coordinate system.) Thus, you can even plot a curve by sending commands at a given frequency, provided that the velocity values of a parametric curve are entered as a control by sampling at the same frequency. With this control mode, smaller displacements can be achieved than with the *"go x y z speed"* commands already mentioned.

The 2nd group of instructions includes instructions that query sensor data. For example, the currently set speed, battery charge, flight time, altitude, acceleration and angular rate can be read from the drone's on-board controller. These are not used because of the unreliable readability mentioned earlier.

3.4. Positioning the drone

The acceleration data could be used for positioning, but the readout gives a very vague answer after sending the *"acceleration?"* command. Even after changing the *timeout* time of the connection to a request sent with 0.1 second intervals set during the tests, it was still giving a meaningful response after 0.5 seconds, but often just throwing a timeout error. Thus, the inherently noisy accelerometer data was even loaded with a significant sampling noise.

The other option is to open a server running on a separate thread to receive the previously mentioned status data. In this case, you receive data in a single string,

including accelerometer and gyroscope data. This data would first need to be run through an additional filter [22] or a Kalman filter [23] to remove sensor noise. By integrating the acceleration values twice, the drone displacement can be obtained, but the integration operation amplifies the noise, is subject to integration error and cannot be corrected for in the absence of stable position points. This option has been discarded due to the noise accumulation with integration.

This is why ArUco markers were chosen, against which the relative displacement of the camera can be measured. A problem with these is that positional data can only be obtained when a marker is clearly visible in the camera image. However, this problem can be solved by properly marking the test plot. Appropriate rules, which were formulated in *section 3.4.2* should be used to place the markers and thus the camera position can be determined.

3.4.1. THE USE OF ARUCO MARKERS

The functions needed to generate ArUco markers can be found in the *OpenCV-Contrib* package of add-on modules. Their positioning requires camera calibration, which can also be performed using the calibration algorithm in OpenCV, based on the study by *Z. Zhang* [24], by estimating the camera parameters.

To calibrate, you need a chessboard with the side lengths, row and column numbers of its squares. Based on these data, the algorithm generates a theoretical map of the interior corner points of the squares and then calculates the interior and exterior properties of the camera in transformation matrices based on the camera samples. The camera matrix contains the five *intrinsic* properties of the camera, including focal length, image sensor aspect ratio and the principal point of the pinhole camera model. The distortion matrix of the lens distortion associated with the nonlinear intrinsic properties is also obtained, which can be used to remove the barrel and cushion distortion at the edges of the image. The external (*extrinsic*) properties of the camera are also described by a transformation matrix, realizing the transfer from the coordinates of the real 3D space to the 3D coordinates of the camera. The matrix applies a homogeneous transformation: a shift to the center (origin) of the camera sensor by the vector **T** and a rotation by the rotation matrix **R.**

In theory, two samples should be enough for a correct calibration, but to achieve more accurate data, we work with 20 data sets. A photo of *Figure 9* was taken during the calibration process. The resulting matrices are stored in a file called *camcalib.npz* and read from there, so that the drone's built-in camera does not need to be calibrated before each use.

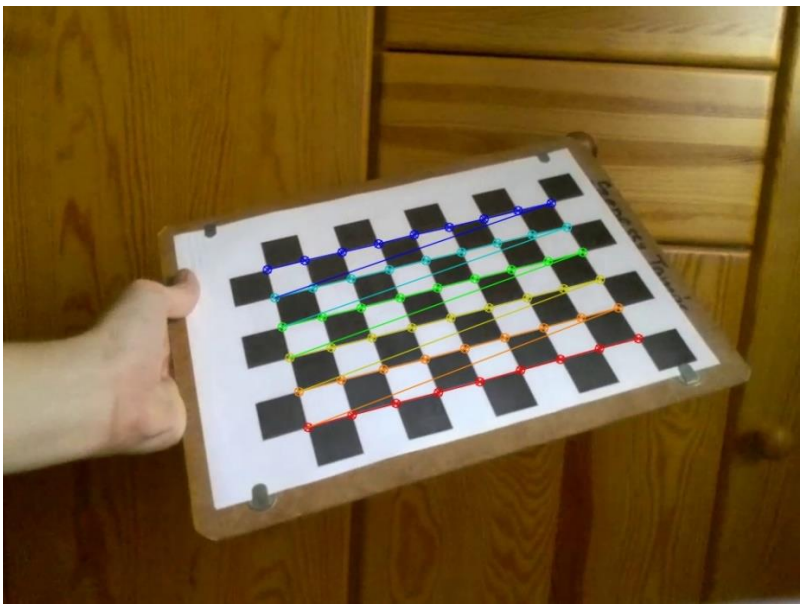


Figure 9: Photo of the camera calibration process with object points projected onto the chessboard

One of the most common sources of error in camera measurements is barrel distortion, which can be eliminated using the distortion matrix obtained during calibration using OpenCV's built-in *cv2.undistort()* function. This will reduce the fisheye effect and reduce the field of view to some extent, but will give more accurate values at the edges of the image for subsequent positioning with ArUco markers.

3.4.2. Rules for marker placement

The rules for marker placement are defined as follows:

1. After starting the program, the drone will immediately search for a marker! As soon as you find one, one of your first position data should be the origin chosen for the displacements! It may not be the first one, because markers at the very edges of the image may not give the most accurate value, despite the distortion correction.
2. There should always be at least one marker in the picture on the line of the drone's pre-selected trajectory!
3. Before a marker disappears from the view, it takes a few frames until it is visible with the next marker. Thus, with the difference in position of the two markers, the transformation of the coordinate systems can be performed and the drone positioning can be solved without interruption.
4. Each marker with an ID on the predefined track line should have a predefined meaning so that the drone knows approximately where to look for the next marker.

ArUco markers have also been used in previous studies for drone control. [25] In this, the markers were laid on the ground and the information extracted was used to compare the values with the drone's inertial sensors.

It is also important to be able to relate the orientation of the first marker seen by the drone to the orientation of the drone's camera. To do this, we assume that the

camera is positioned in the horizontal, resting flight position of the drone, detecting marker 1 in a nearly horizontal position. The drone deviates from the horizontal position during heading change and progress, but this angular deviation is negligible for now, as we only want to use the program at low speeds for the time being. The positioning of the camera results in an angle of depression of between 10° and 11°, the exact value of which is difficult to measure due to the design of the drone housing, so let's take it as 10.5°!

3.5. First tests for positioning with markers

For testing, four 7x7 ArUco markers with a side length of 0.0957 m from the DICT_7x7_100 library were used. Markers with 4x4, 5x5 and 6x6 areas are also available, but the higher resolution of the 7x7 library allows for more accurate positioning. Up to 1024 different markers are available in each library, which should be plenty enough to provide the information needed to guide the drone.

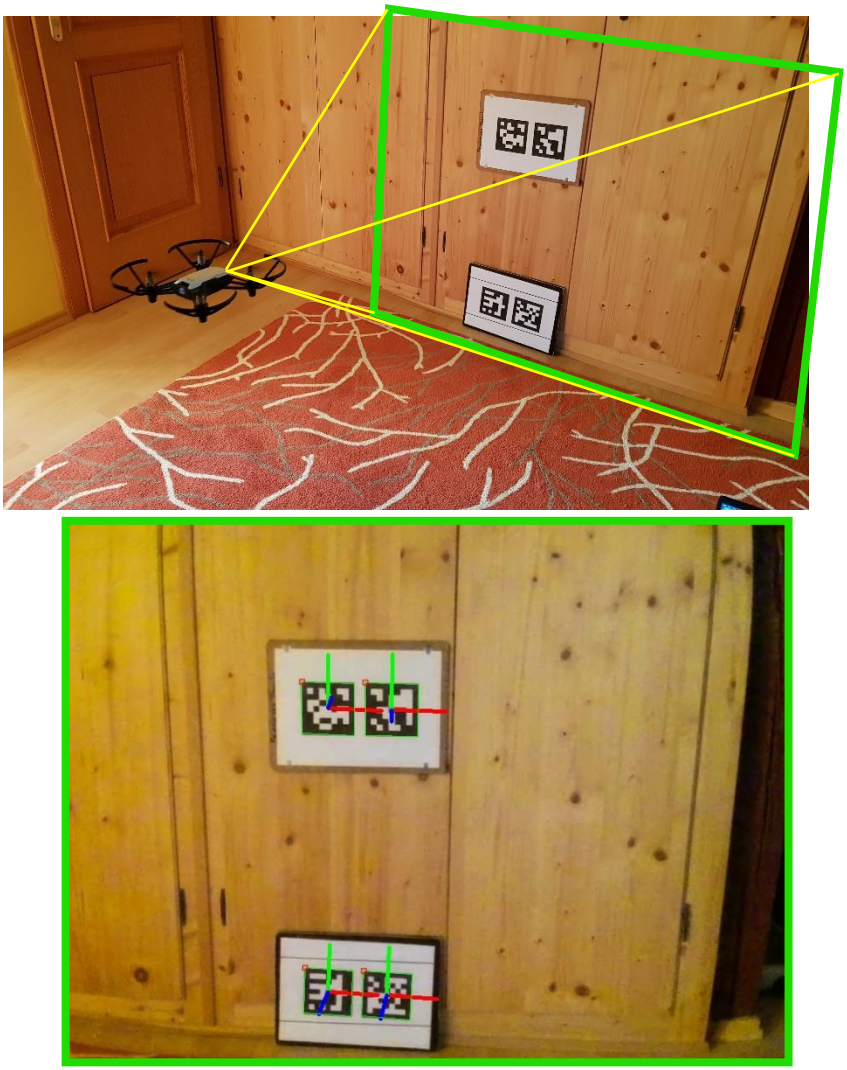


Figure 10: The experimental marker placement, showing the drone's field of view and the image seen by the drone

The program takes the first appearance of markers as a starting point and stores these points as origos, relative to which the displacement of the markers is then plotted. It is recommended to use more than one marker, because the system can average from several values, so that more accurate results can be obtained by averaging, simply by filtering out any outliers. Furthermore, if there are several markers in the drone's field of view, it is less of a problem if you lose one of them, as it can still read values from the others.

In the first experiment, the markers were placed perpendicular to the drone's camera, vertically against a wall, as shown in the photo in *Figure 10*. The purpose of the test was to see how well the software could store data that approximated reality.

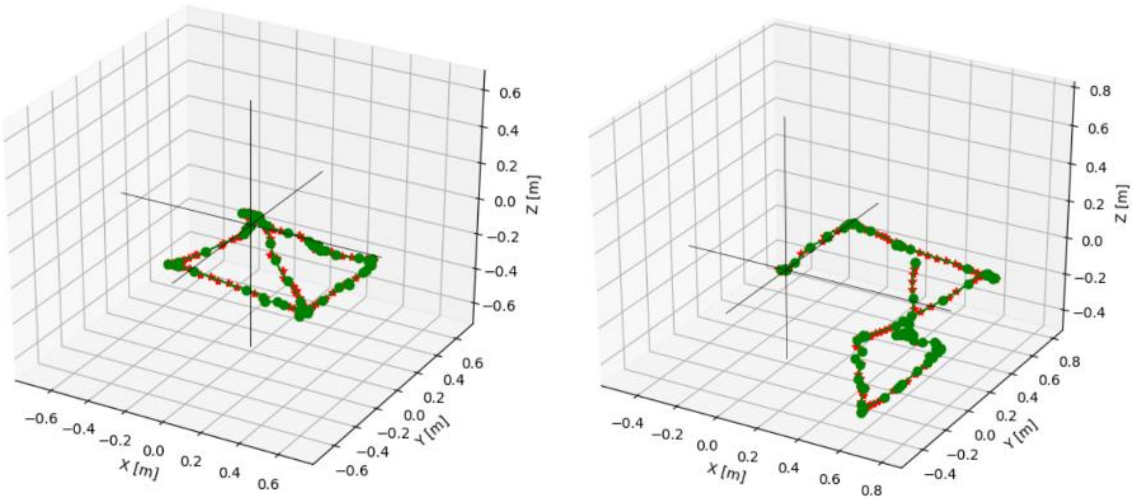


Figure 11: Two paths followed by the drone plotted in Descartes coordinate system

The point set on the left in *Figure 11* shows that the measurement point set was successfully tracked as soon as a square was described with the drone and even one of its diagonals was drawn in, and on the right it shows that the change in height was also detected with excellent accuracy by the markers. The point of intersection of the black axes is the origin of the drone's movements. The points in red in the figure are the real values, and the green curve is the B-spline curve fitted to the point set by the *splprep()* function in the *interpolate* subdirectory of the *Scipy* package available in the Python programming language. The refinement parameter of the function was chosen to be s=0.1 after a few experiments based on the documentation [26], which already sufficiently approximates the point set by filtering out the noisy points. Finally, in the final version of the program, we use a Kalman filter instead of the B-spline interpolation method to filter the data points, thinking about the future real-time implementation and the more adaptive behaviour of the Kalman filter.

The only correction to be applied to the experimental point set was to rotate the points negatively by 10.5° about the x-axis, otherwise the Z-coordinates of the points would have increased as they approached the markers, even at constant height. This is due to the camera's angle from the water level, with a higher position detected as it approached the markers. (This will be redundant later because of the correction for the angular rotation of the markers.)

In this setup, the drone will only give true displacement values if the camera plane (apart from the angle of depression) is parallel to the plane of the markers. If the position deviates from this, the drone will give incorrect values. The displacement values have to be estimated by taking the rotation angles into account, otherwise we cannot convert the values of the translation vectors of the ArUco markers into the coordinate system of the first marker seen, which we want to use as the global coordinate system. The vector values given by the OpenCV package's *cv2.aruco.estimatePoseSingleMarkers()* function are given in the camera coordinate system, which then need to be converted to the marker coordinate system.

3.5.1. EVALUATION OF THE TESTS AND EXPECTATIONS OF THE SYSTEM

Based on the first tests, it is worth further exploring this positioning option, as it can calculate the drone's position in real time with relatively few transformations. Unfortunately, we are not able to use the graphical acceleration of the *CUDA* package provided by *Nvidia*, as the *OpenCV-Python* package functions have not all been added to the *cv2.cuda* library in Python. This will slow down image processing, but will still cause a larger delay in operation due to the 1.5...2 second delay of the camera image transmitted by the drone.

The measurement error of the method is expected to be around 0.05 m with an increased marker size and an appropriate number of markers. However, due to the conversions between marker coordinate systems, we can also expect a drift error that increases with distance from the marker representing the global coordinate, due to the imprecision of the transformations between the coordinate systems.

3.6. Principle, refinement and robustness of positioning

The camera program is not tested with the drone, as it would have taken too long to charge the batteries and retry the tests. The image processing program is written using a Logitech C250 USB webcam with 640x480 pixel resolution. Although the resolution is different from the 960x720 pixel resolution of the drone, the program is designed to run with a camera independent of the drone by specifying other calibration data.

The program was written using the functions provided in the OpenCV ArUco package, such as *cv2.aruco.estimatePoseSingleMarkers()* to determine the camera position. After the calibration procedure already outlined, this is able to compute a translation and a rotation vector for each marker in the camera coordinate system using a Perspective-n-Point (PnP) transformation, the conventions of which are described in *Figure 12.* The transformation is performed by the function *cv2.aruco.solvePnP() on* the basis of the corner points in pixels of the seen markers, the camera matrix and the distortion matrix, using a pinhole camera model. [27]

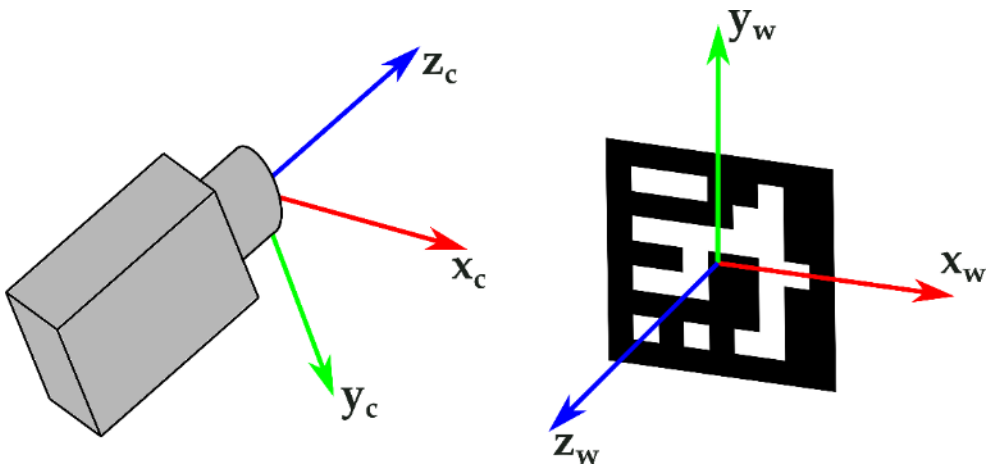


Figure 12: Camera and marker coordinate system interpreted by OpenCV

3.6.1. PERSPECTIVE PROJECTION, PNP TRANSFORMATION

The PnP transformation is a perspective projection that creates a link between the real 3D space and the 2D image plane using the calibrated camera matrices. If the relative positions of the object points are known - in this case, the corner points of the markers are at the specified side-length distance from each other - then we can convert the position of the markers from the image plane to real space.

Given a 3D space with the aforementioned marker corner points as object points, and the position in pixels of the corner points on the 2D image plane. These can be transformed into the coordinate system of the camera, obtaining the 6 degrees of freedom (6 DoF) position of the marker relative to the camera using a translation vector and a rotation vector. Based on these, we can feed the *cv2.solvePnP()* function with the side lengths of the markers to the object points, measuring their corner points in pixels as projection points, and we need the camera matrix and the distortion matrix obtained during calibration. Thus the corner points of a marker, in its own coordinate system, where *mLen* is the lateral length of the marker in metres:

$$\mathbf{x_{w_1}} = \begin{bmatrix} -mLen/2 \\ mLen/2 \\ 0 \end{bmatrix} \quad \mathbf{x_{w_2}} = \begin{bmatrix} mLen/2 \\ mLen/2 \\ 0 \end{bmatrix} \quad \mathbf{x_{w_3}} = \begin{bmatrix} mLen/2 \\ -mLen/2 \\ 0 \end{bmatrix} \quad \mathbf{x_{w_4}} = \begin{bmatrix} -mLen/2 \\ -mLen/2 \\ 0 \end{bmatrix}$$

The object points as points of world coordinates ($\mathbf{x_w}$) can be projected onto the image plane ($[u, v]$) using a perspective projection matrix ($\mathbf{\Pi}$) and the camera's intrinsic matrix ($\mathbf{A}$), as shown in *equation (3.1.)*. In addition, we need a ${}^{\mathbf{c}}\mathbf{M_w}$ matrix for the transformation from the coordinate system of the marker world to the coordinate system of the camera. Since all other parameters of the mapping are known, the unknown transformation matrix can be obtained by rearranging the matrix equation. From ${}^{\mathbf{c}}\mathbf{M_w}$ matrix performs a homogeneous transformation between coordinate systems using a rotation and an offset according to *equation (3.2.)*. By transforming the matrix, we obtain the translation and rotation vector values for each marker. [28]

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{A\Pi^c M_w} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{3.1.}$$

$$\mathbf{A^{-1}\Pi^{-1}} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{3.2.}$$

Note: The conversion of the rotation matrix into a rotation vector is explained in detail in *section 3.6.3*, following the conventions used by OpenCV.

3.6.2. THE MEASUREMENT CONVENTIONS

The plan is to store the positions and orientations of the markers to the first marker seen as the origin, so that later positions can be converted to our global coordinate system. When designing the system, assume that the drone is positioned at the first marker with a minimum angular position! Thus its rotation is also measured against it. In the drone state readout, we have the option to read the angular position of the drone, as mentioned above. As this gives a well filtered, robust angular position of the drone's orientation according to the Euler angles convention, we will use this instead. An attempt was also made to extract the angular position from markers, but the former of the two methods proved to be more appropriate.

To process and visualize the orientation of the drone in Euler angles, a rotation matrix can be computed using *formula (3.3.)*. Multiplying this by the coordinate base corresponding to the drone gives its rotation. To understand the angular conventions used for the Tello drone, see Figure 8. These are given by the balun rule, with the clockwise rotation of the drone giving a positive value.

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3.}$$

where:

R: the rotation matrix obtained from the angles

α: about the *z* axis (yaw - rotation about the vertical axis)

β: around the x-axis (pitch - "dip" around the transversal axis)

γ: around the *y* axis (roll - "decision" around the longitudinal axis)

Since there is no universal agreement on the convention for Euler angles, we always have to define the convention used. In this case, we store the angles in the order already mentioned, according to the negative values obtained from the drone SDK. These are identical to the angle convention used for aircraft according to the DIN 9300-2 flight standard (*Figure 13*). The more precise definition group for the chosen angle convention is the intrinsic convention of *Tait-Bryan angles z-y'-x"*, also known as nautical or *Cardan angles*. [29]

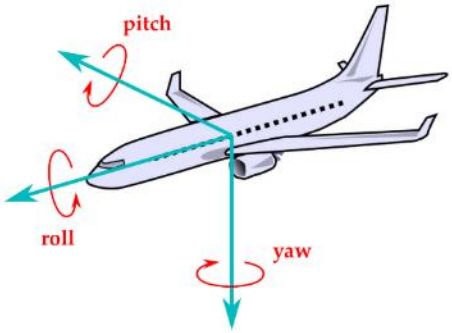


Figure 13: Main axles of an aircraft according to DIN 9300 [13.Im]

3.6.3. CONVENTIONS USED BY OPENCV

Among the values returned by *estimatePoseSingleMarkers()*, the translation vector is interpreted in meters, and the rotation vector uses an axis-angle or Rodrigues representation, which is the most compact storage form of the rotation matrix. OpenCV's *Rodrigues()* function can compute a rotation matrix from rotation vectors given as row vectors according to *equation (3.4.)*and can also determine its inverse if the input is a rotation matrix. [28]

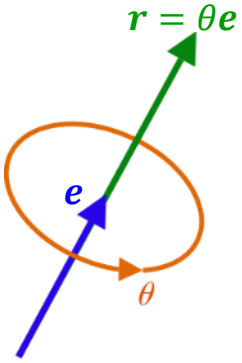


Figure 14: Interpretation of the Rodrigues axis-angle rotation vector [14.Im]

The rotation vector should be as follows (as shown in *Figure 14*):

$$\mathbf{r} = \theta\mathbf{e}$$

thus: $\theta = norm(\mathbf{r})$ and $\mathbf{e} = \mathbf{r}/\theta$

$$\mathbf{R} = cos\theta\mathbf{I} + (1 - cos\theta)\mathbf{e}\mathbf{e}^{\mathbf{T}} + sin\theta \begin{bmatrix} 0 & -e_z & e_y \\ e_z & 0 & -e_x \\ -e_y & e_x & 0 \end{bmatrix} \quad (3.4.)$$

where:

R: the Rodrigues rotation matrix

r: the Rodrigues rotation vector

e: unit direction vector of the axis of rotation

θ: norm (length) of the rotation vector, angle of rotation about the axis

3.6.4. CONVERT CAMERA POSITION TO GLOBAL COORDINATE SYSTEM

The use of homogeneous transformations has also been considered, since the conversion of coordinate systems, a rotation and an offset can be implemented simultaneously. Thus, a single matrix multiplication could be used to interleave the coordinate systems by transforming the camera position into a quaternion. This transformation is shown in *equation (3.5.)*. However, in addition to the vector conventions used by *OpenCV*, it was found to be more transparent to rotate and shift the coordinate systems of the markers separately, as shown in *equation (3.6.)*

The transformation is a point on the coordinate system of marker *n*. $\mathbf{t_{nn}}$ and the global coordinate system. This maps the point to the $\mathbf{dR_n}$ to coordinate system 0 using the rotation matrix, and shifts it to the origin of coordinate system *n*, which is defined by $\mathbf{t_{on}}$ vector (interpreted in global coordinate system).

With homogeneous transformation:

$$\mathbf{t_{n0}} = \begin{bmatrix} \mathbf{dR_n} & \mathbf{t_{on}} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{t_{nn}} \\ 1 \end{bmatrix} = \begin{bmatrix} dR_{n_{11}} & dR_{n_{12}} & dR_{n_{13}} & t_{on_x} \\ dR_{n_{21}} & dR_{n_{22}} & dR_{n_{23}} & t_{on_y} \\ dR_{n_{31}} & dR_{n_{32}} & dR_{n_{33}} & t_{on_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_{nn_x} \\ t_{nn_y} \\ t_{nn_z} \\ 1 \end{bmatrix} \tag{3.5.}$$

By shifting and rotating (this is what we do):

$$\mathbf{t_{nm}} = \mathbf{t_{on}} + \mathbf{dR_n t_{nn}} \tag{3.6.}$$

Thus, at each instant in time, the camera position interpreted in the coordinate system of the seen marker can be calculated and from there converted to the global coordinate system as illustrated in *Figure 15.* These require the determination of the offsets and rotations between the coordinate systems. The idea is that the program calculates and stores these by automatic averaging from the drone camera image for two seen markers from a given number of samples.

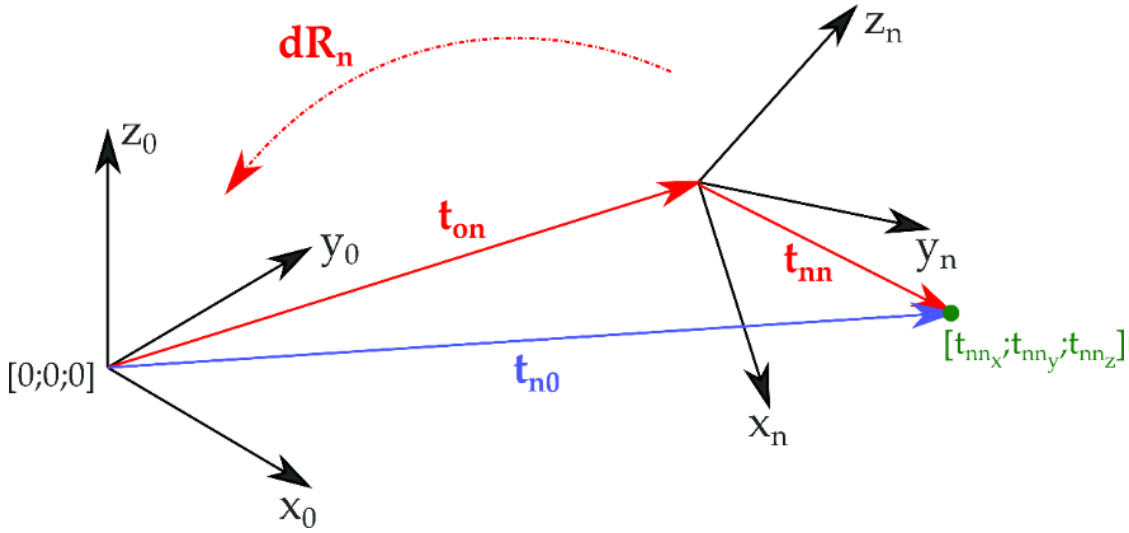


Figure 15: Illustration of the transformation to the global coordinate system

3.6.5. CALCULATING THE ORIGIN AND ROTATION OF MARKERS

For each marker, it is necessary to store an offset, which gives the position of its origin in the global coordinate system, and a rotation matrix, which rotates (projects) the marker's coordinate system into the global coordinate system 0. These are the marker *n*. already mentioned $\mathbf{t_{on}}$ translation vector and the $\mathbf{dR_n}$ rotation matrix.

They are defined as follows, if given:

- → $\mathbf{t_{om}}$: global vector pointing to the origin of the *M. C.*
- → $\mathbf{dR_m}$: the matrix rotating *m.* krs. into the global
- → $\mathbf{t_{cam_m}}$ and $\mathbf{t_{cam_n}}$: the vector of the marker *m.* and *n.* in the crs. of the camera
- → $\mathbf{R_{cam_m}}$ and $\mathbf{R_{cam_n}}$: the rotation matrix of markers *m* and *n* with respect to the camera (these can be calculated by a built-in function based on *transformation (3.4.)*)

It is known that the rotation matrix implements an orthogonal transformation between Descartes coordinates, i.e. the inverse of the rotation matrix is equal to its transpose ($\mathbf{R^{-1} = R^T}$). Thus, in general, the following transformation holds for the conversion of the camera position into the coordinate system of the marker:

$$\mathbf{t_{marker}} = -\mathbf{R}_{\mathbf{cam}}^{\mathrm{T}} \cdot \mathbf{t_{cam}} \tag{3.7.}$$

The difference between markers in the coordinate system of the camera:

$$\mathbf{dt_{cam}} = \mathbf{t_{cam_m}} - \mathbf{t_{cam_n}} \tag{3.8.}$$

Based on these, the origin of marker *n* can be calculated from *equation (3.9.)*using the known data for marker *m. The* transformations of the markers are thus cumulated starting from the global origin.

$$\mathbf{t_{on}} = \mathbf{t_{om}} + \mathbf{dR_m}\left(-\mathbf{R}_{\mathbf{cam_m}}^{\mathrm{T}} \cdot \mathbf{dt_{cam}}\right) \tag{3.9.}$$

Since the relative positions of the two markers are fixed, a constant rotation transformation can be written between them, rotating from the coordinate system of marker *n* to marker *m.* This can be proved from the properties of the rotation and the matrix equations, but will not be discussed here. From these, the rotation matrix can be obtained in general as follows:

$$\mathbf{dR} = \mathbf{R}_{\mathbf{cam_m}}^{\mathrm{T}} \cdot \mathbf{R_{cam_n}} \tag{3.10.}$$

The global coordinate system rotation matrix for marker *n*:

$$\mathbf{dR_n} = \mathbf{dR_m}\left(\mathbf{R}_{\mathbf{cam_m}}^{\mathrm{T}} \cdot \mathbf{R_{cam_n}}\right) \tag{3.11.}$$

These values are calculated by averaging over a number of samples in the actual program to filter out possible outliers. This compensates for camera mapping errors. The resulting $\mathbf{t_{on}}$ vector and the $\mathbf{dR_n}$ matrix are stored in the data of marker *n*, which can later be used to convert the camera position into the global coordinate system. The disadvantage of this method is that both markers, between which a transformation is calculated, must be correctly visible in the camera image for a given period of time.

4. IMPLEMENTATION OF THE DRONE PROJECT

4.1. Overview of the structure of the programmes

A total of 8 programs will be used to control the drone and collect data. Data processing ends in a separate script, and real-time data display is not possible due to the camera image delay. In order to understand the structure of the programs, *Figure 16* shows which Python script is related to which other script and which task it performs.

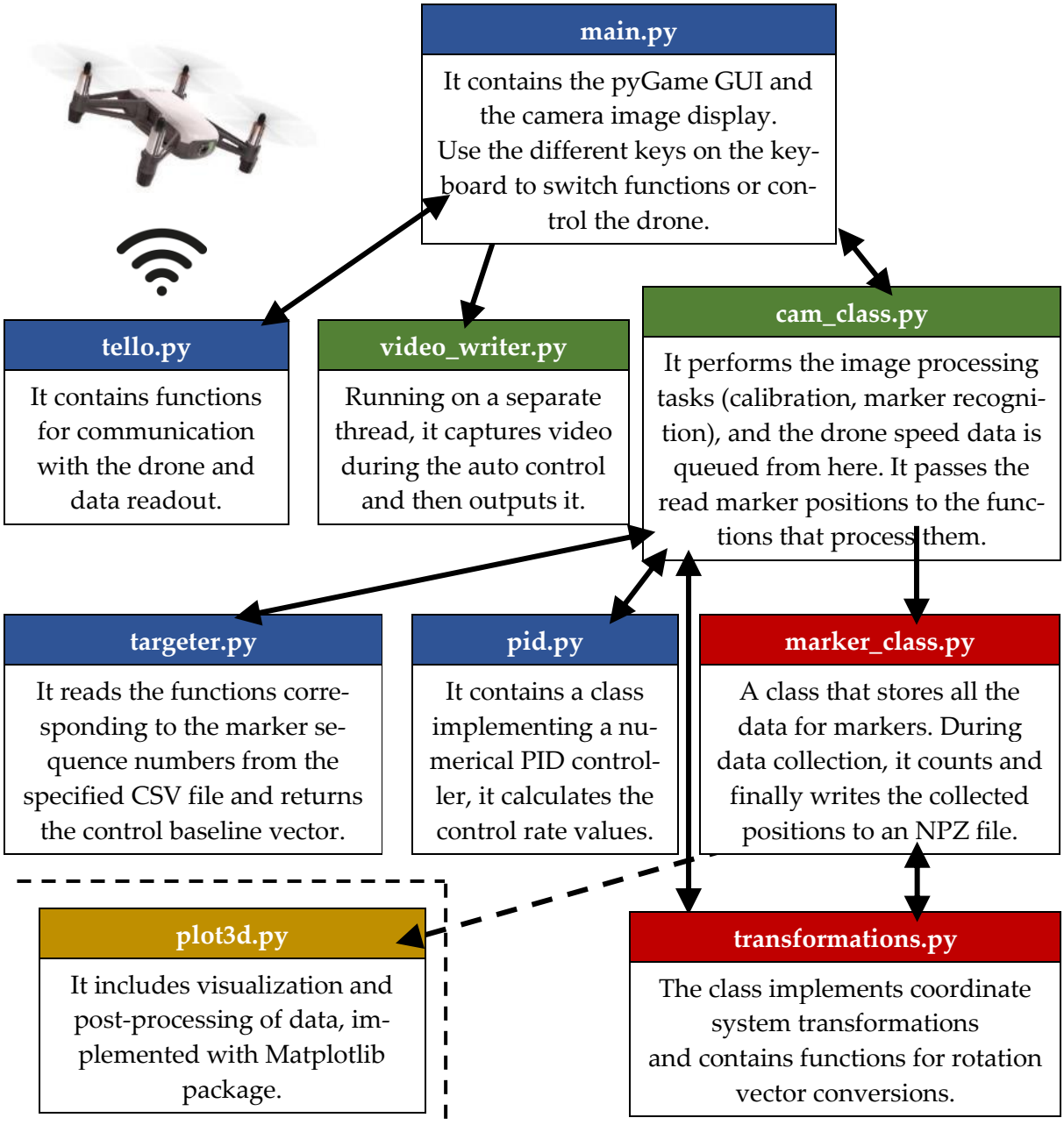


Figure 16: Structure of the drone control programs

4.2. Programs that control the drone

The communication with the drone is handled by the *tello.py* program, much of this code is based on the drone control program written by *Damià Fuentes* [18], as well as the keyboard control of *main.py* and the pyGame interface. The status readout thread, already mentioned

Code snippet 2 new feature that is important to us for data collection.

The pyGame GUI can efficiently handle keystrokes in the Python programming language, which is why it was decided to keep it. Thus, the possibility to control the *main.py* program with the keyboard was retained, in case of intervention in dangerous situations. Appropriate keys trigger different events that can be used to start certain functions of the program, such as navigation, calibration, etc. The importance and advantage of the event class in Python compared to simple truth values stored as variables is that it can communicate between threads. Thus, we can observe the occurrence of the same event in separate threads, without having to pass it as a function argument or in a queue, during initialization, it only needs to be passed once.

You can switch between the navigation types by pressing the "O" key, which activates the marker search and control program. The velocity values are calculated by the navigateToMarker*()* function of the *cam_class.py* Camera class and stored in a row type container. (This is discussed in more detail in *Section 4.3*) When using this, care must be taken to saturate the queue, because if items are loaded into the queue faster than they are read, an increasing delay is introduced during the read. Therefore, it is recommended to use the *clear()* function of the Queue type after each readout, so that the queue is cleared regularly without fear of saturation. Similarly, pressing the "K" key will recalibrate the drone's camera, pressing the "M" key will save the camera image to a JPG file, and pressing the "X" key will close the video capture. (This overrides the automatic lock.)

This is where the drone's data and the camera image are read out from the status line every cycle. As its name implies, this is the main program cycle, from here the camera image is passed to the image processing program and sent to the video writing program thread.

4.2.1. Drone position control

To position the drone according to the control, a numerical PID controller is used, in parallel P, I, D, with the time step assumed constant. This was tuned empirically by experimenting with the drone. The control is done according to the coordinate system of the camera, as it is easier to handle the adjustment to markers with different orientations.

A difficulty in the control is that the camera image arrives at the laptop server with a delay of 1.5...2 seconds. This should be taken into account in the experimental tuning. Experience has also shown that, in the direction of vertical movement (z coordinate), the gain of the controller should be increased for a more efficient adjustment. This may result from the higher motor torque required for up-down movement. A separate regulator setting has also been made to control the angular speed in the *yaw* direction. If the drone loses the marker for a full 2 seconds, it will search for the last seen x heading by turning right or left.

The target values for the setpoint are varied depending on the marker, which is the reference for the controller and is used for error calculation. The output of the controller must still be linearly amplified with the drone speed value and a constant gain. This gain can also be determined experimentally. The base velocity value used in the automatic control of the drone is 0.15 m/s, i.e. 15 cm/s. Setting a faster base speed is not possible due to the camera image delay.

Once the drone has set the desired position, it can move to the next marker, its desired position will be set on the controller's base signal and it will try to adjust to it. The adjustment to the desired position of the drone is monitored with a certain margin of error. If this condition is met, it will try to adjust to the closest of the new markers it has just seen.

The implementation of the regulator is shown in Code snippet 3, variables stored in an instance of the PID class are: K_P, K_I, K_D, previous error and error integral. The controller can be implemented in all three directions of the drone according to *equation (4.1.)*, where e_n is the error with respect to the base signal at time n, K_P is the proportional term, K_I the integrating term and K_D is the gain of the derivative term. The resulting v_{n0} value must be amplified to obtain the correct $\mathbb{Z} \in [-100; 100]$ range for the RC control of the drone. If the input velocity falls outside this range, the drone will automatically take it as maximum.

$$K_P \cdot e_n + K_I \cdot \sum_{i=0}^{n} e_i + K_D \cdot (e_n - e_{n-1}) = v_{n0} \tag{4.1.}$$

Code snippet 3: Function implementing numeric PID (*pid.py/Class PID/def control()*)

```
def control(self, error):
    self.error_int += error
    if self.error_prev is None:
        self.error_prev = error
    error_deriv = error - self.error_prev
    self.error_prev = error
    return self.kp*error + self.ki*self.error_int + self.kd*error_deriv
```

The values of the selected control members and the post-amplification (A) in each direction are given in *Table 1.* The velocity data is loaded into the Queue containing the directions and is read in the drone's *main.py* program when a value is entered in the queue. The resulting values must still be converted to integer values using Python's *int()* type conversion, since the drone's control can only interpret integer values. Now the 4 speed values can be sent to the drone using the *tello.send_rc_control()* function.

Table 1: Selected values of the members of the PID controllers by direction (S the base speed)

Speed Direction	K_P	K_I	K_D	A
x	0.3	0.00001	0.0001	$10S$
y	0.3	0.00001	0.0001	$10S$
z	0.8	0.00001	0.0001	$10S$
yaw	0.1	0.00001	0.001	$S/2$

The target location types assigned to the sequence number of each marker are read from a CSV file, so that later modification and marker expansion can be done easily without modifying the program. We created 10 marker types, each with a different basic marker vector, and the *targeter.py* script switches between them based on the CSV file and sequence number. Thus, after changing the sequence number of the target marker, the target vector must also be changed to adjust to the new position.

The basic marker types defined are:

- *Origin*: always the marker number 1, the drone must be aligned facing it to minimize the angular error between their coordinate systems.
- **straight navigation** "*Right sideways*" **and** *"Left sideways*": a drone position navigating at a given angle along a wall to see additional markers.
- **right and left rotate markers** (*"Right/Left rotate corner 1/2/3"*): more than one may be needed in a single corner, staggering is essential to see markers at the same time and avoid bumping into walls.
- **end marker** (*"End"*): currently the 50th marker, but this can be extended. The 50th marker is the end marker.

4.3. Programs that process images

Closely related to drone control, but more image processing, is the *cam_class.py* program, whose main function is responsible for ArUco marker recognition. Another image processing program is *video_writer.py*, which runs in a separate thread using OpenCV's *cv2.videoWriter()* function and automatically saves a video in AVI format from the drone's camera image. This is not discussed in detail in this paper.

ArUco markers are detected using the OpenCV built-in *cv2.aruco.detectMarkers()* function. This finds the codes in the loaded marker bundle after adaptive segmentation applied to a grayscale image and returns their corner points in pixels and their sequence number. The previously mentioned PnP transformation is used to compute the real three-dimensional position of the markers based on the camera matrices. The resulting positions with their marker sequence number can be sent for further processing (*Section 4.4*) or used for automatic navigation of the drone.

4.3.1. NAVIGATE THE DRONE USING THE MARKERS

Since we later want to display the marker in the camera image, against which the drone navigates, the function that performs the navigation is not placed in a separate class from the camera one. The target marker is indicated by the yellow line drawn in the camera image between the center of the image and the center of the marker measured in pixels. (*Figure 17*) This has no vector meaning, it is merely for illustrative purposes for the operator. In the control program descriptions of this chapter, such frames are also included, extracted from the video automatically recorded by the drone.

a) b)

Figure 17: The moment after starting the drone navigation, after positioning at marker 1 (left). In case a) the control is started with a vertical marker spool and in case b) with a horizontal marker spool. The yellow section illustrates the ongoing control to marker 2.

To start the navigation, the drone must be aligned with the required error vector to the marker number 1, which is chosen as the origin. Since the position is observed in the coordinate system of the drone's camera and not of the marker, the drone can thus place the correct coordinate system origin for two types of marker orientations: horizontal and vertical, it can interpret a correctly (not obliquely) placed marker as the origin. (Figure 17 a) and b) The orientation of the other markers is not important for the transformations. A condition for alignment is that the drone is aligned with the target marker within a certain error margin. The allowed error vector values are shown in *Table 2* further condition for set-up is that the drone must be within the error limit for 1 second. The error value can be either positive or negative and is therefore tested for absolute values.

Table 2: Error margins allowed during drone navigation

Direction	*x*	*y*	*z*	*yaw*
absolute value of the allowed error	0.15 [m]	0.15 [m]	0.15 [m]	5 [°]

As soon as it is within the specified error range, it transfers control to the nearest marker in the camera's coordinate system that has not yet been used during navigation. (*Figure 18*) This can be determined by calculating the length of the vectors and then looking for a minimum. During the transfer, the default vector of the controller,

the target vector, is also changed, this is determined by a function in *targeter.py* based on the data in the CSV file.

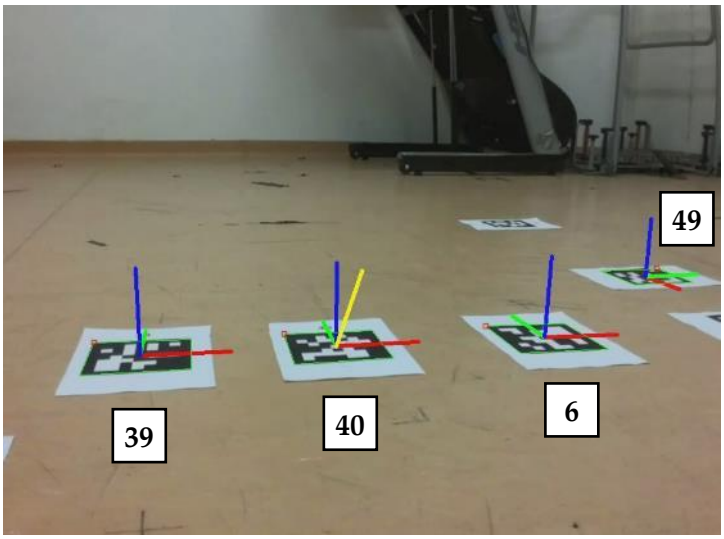
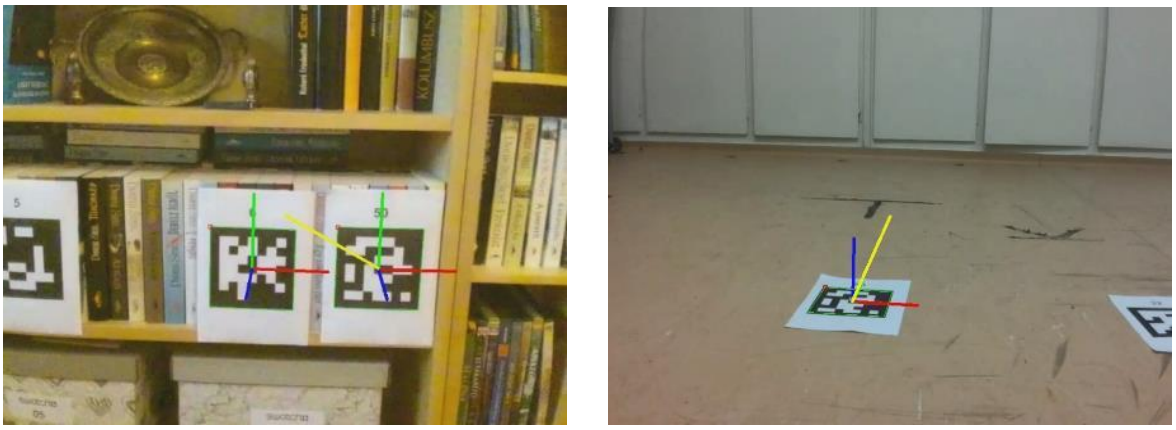


Figure 18: The drone has arrived from the left and is currently navigating to the middle marker 40. The marker 39 on the left would be closer to the camera, but it has already been there, so selects the marker closest to him from the two markers seen on the right (6 and 49), and give control to the 6.

The marker number 50 was selected as the end marker, after setting to this marker the program will save the collected data, stop navigation and output the recorded video. (*Figure 19*) If the drone fails to reach marker 50, is unable to align to it, or if a transformation between markers is missing, the half-finished data and video will not be output. It is therefore important that the markers are positioned correctly, allowing sufficient time for the transformations to be calculated. As the program runs, important messages are written to the console and the operator should monitor these to correct any missing calculations.



a) b)

Figure 19: Moment before stopping the control at marker 50 successful alignment for markers placed vertically (a) and horizontally (b).

Since the drone's camera cannot be moved, it can only fly at a low altitude of 0.3...0.4 metres when navigating with horizontal markers. This allows the camera, which is positioned at an angle of depression of 10.5°, to see in front of and below you. By selecting 0.9 m as the camera's z heading, the drone can navigate safely between horizontal markers.

4.4. Programmes that process data

The processing of the spatial points and rotations obtained from the camera image processing programs, i.e. the data collection, is done by the *marker_class.py* program. The necessary transformations, which are described in *section 3.6* are performed by the *transformations.py* functions. The filtering and display of the collected point set can be done afterwards in *plot3d.py*.

4.4.1. STORAGE OF MARKER DATA

The first task is to implement rule 1 of *chapter 3.4.2* by a filter. Only the data of markers that have none of their vertices in the pixel frame of the image can be passed on to the processing. The definition of edge is a 2-2-2-2% intercept on all four sides of the camera image. Markers located at the edges both provide more inaccurate data after the distortion is removed and can make the marking of the outer edges of the 2D marker code uncertain during marker detection, as can be seen in the image in *Figure 20*.

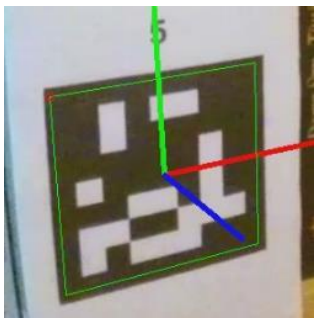


Figure 20: Cropped image from one of the recorded videos of an incorrectly detected marker

In the markers class, the properties of the markers you have seen so far are stored in different lists, such as:

- **`ids:`** number of markers already used
- **`tvec_origin:`** markerorigos in the global coordinate system
- **`rvec_origin:`** rotate marker coordinate systems relative to the global
- **`dRot:`** the rotation matrix from the given marker to the global krs.
- **`allow_use:`** an auxiliary variable to allow a given marker to be used in averaging, once enough samples have been collected, it can be counted
- **`tvec_min, tvec_max:`** auxiliary variables for filtering the average calculation

The *appendMarker()* function called during the detection of ArUco markers adds the new markers to the marker class. It takes the marker numbered 1 as the origin, determining whether it is a horizontal or vertical marker based on its Euler angular rotation about the x-axis. It also stores the base values of the angular rotations from the readout of the drone states, this becomes the origin of the Euler angles. It stores the corresponding one of the two orientations in a matrix, which will be relevant for post-processing indexing by inverting the directions of the translation vector.

Additional markers are stored by creating transformations between coordinate systems, as described *in section 3.6.5* implemented by the *transformations.py* script *getTransformations().* In operation, the system currently works from 12 valid patterns. Tests have been done with higher validation limits, but many times the system did not have time to perform the transformations until the markers were in the drone's field of view and it was unable to compute any further. Thus, it collects 12 offset vectors between the two markers, discarding the minimum and maximum normals and averaging from the 10 remaining values. Rotation matrices are also obtained by averaging 12 samples without filtering. The resulting values are stored in the object lists of the class.

If a marker already has a sample for authorisation, it can be counted by the function in *Code snippet 4*, based on the equations in *section 3.6.4.* The program counts the positions read from all the markers seen, averages them and stores this position value. From the angular rotations read from the drone's sensor, it subtracts the origin of the angles to give the orientation of the drone. The data is collected in arrays along with the time it was stored and written to an NPZ file after the navigation is completed.

Code snippet 4: Translation vectors of markers into global krs.
conversion function (*transformation.py/def calculatePos()*)

```
def calculatePos(tvec, rvec, tvec_orig, dRot):
    tvec = np.transpose(tvec)
    tvec_orig = np.transpose(tvec_orig)
    R = cv2.Rodrigues(rvec)[0]
    tvec = -R.T.dot(tvec) # camera position in the marker coordinate system
    tvec = tvec_orig + dRot.dot(tvec) # conversion to global crs.
    tvec = np.transpose(tvec)
    return tvec
```

4.4.2. Post-processing and displaying the data entered

The results of the measurements are stored in matrix form in the NPZ file, so that they can be easily read back by the names of the matrices. The *Axes3D* class of the *Matplotlib* software package is used for visualisation. A function for the stationary display of the data points and one for the animation of the drone's angular rotations were also created.

A simple, empirically calibrated Kalman filter was applied to the data collected by the drone to attenuate the measurement noise. The result of the filtering on the z-direction data of measurement 1 in *Chapter 5* is shown in *Figure 21*. The sample shows that the data values are affected by noise, which has been reduced by properly tuning the Kalman filter. In addition, the measurement is also affected by a kind of drift error due to the successive inclusion of the coordinate systems of the markers. If we know that the markers were in reality in the same plane, we can correct the results by surface fitting to the marker centres.

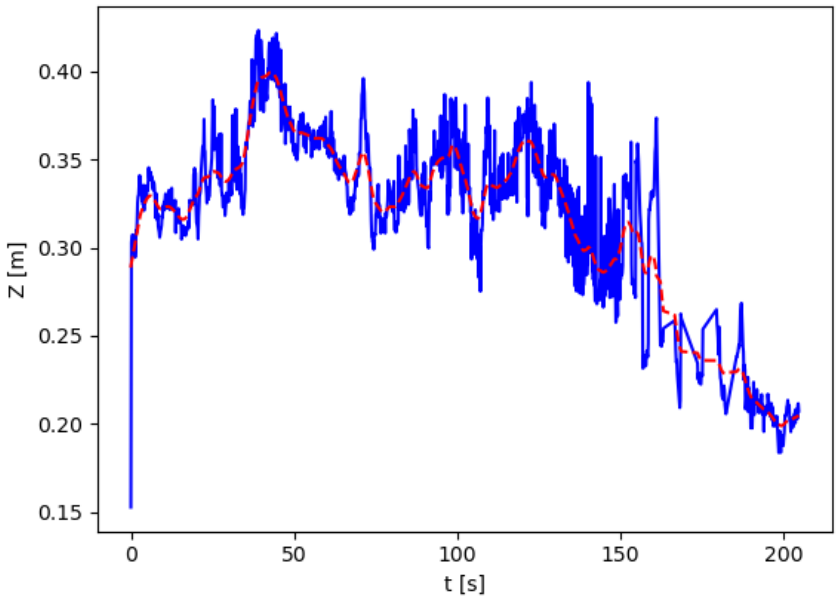
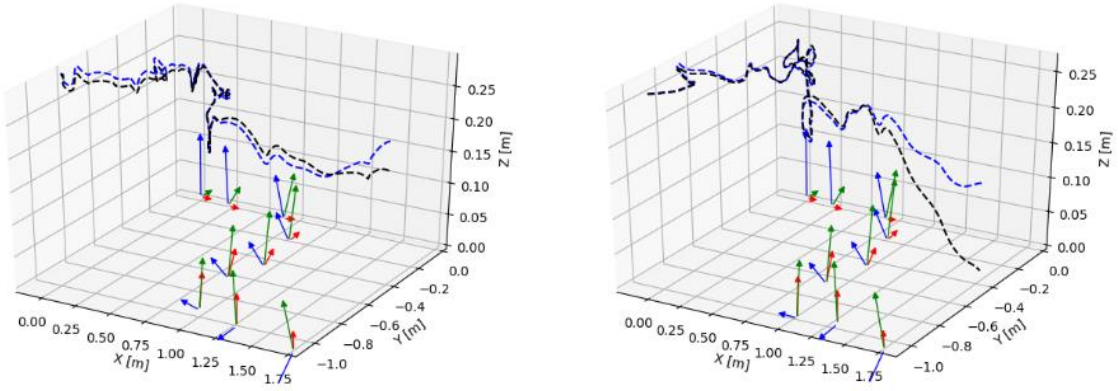


Figure 21: Unfiltered data points in the z-direction of measurement 1 and approximation by the applied Kalman filter

For the correction, we use the SciPy function *scipy.linalg.lstsq()*, which fits a surface to a set of points using the method of least squares. This has the advantage that it always gives a result, the method converges successfully. After fitting a surface of degree two, we can project the curve of the drone's trajectory in the z direction onto the surface. The z coordinate values of the projected points can be used to correct the measurement points, as shown in the curves in *Figure 22* The amount of correction increases towards the end of the data points as the measurement error of the system increases. Finally, this post-plane correction is applied to all measurement results, resulting in better height values.



Data set of measurement 3 **Data set of measurement 4**

Figure 22: Measurement values corrected for z-direction (blue) and the Kalman filtered data series (black) for the example of measurement 3 and 4

From the filtered values, you can also output videos of your measurement results using the *FuncAnimation()* function of the *matplotlib.animation* package, which can also display the drone's rotation. The video is saved in MP4 format. During the animation, the coordinate system corresponding to the drone leaves behind averaged "breadcrumb" points, showing the trajectory it has travelled.

5. MEASUREMENTS WITH MOTION CAPTURE

To evaluate the system, control data are needed to get an idea of the usability and accuracy of the results. As a reference, we chose the motion lab of the MOGI department, whose *OptiTrack* [30] motion capture system (hereafter *MoCap*) provides calibrated, accurate data from the drone.

5.1. Preliminary tests in a home environment

Before measurements could be made with the departmental Motion Capture system, it was necessary to test the functionality and reliability of the program. To this end, we repaired the device in a home environment using two types of markers: vertical (wall) and horizontal (floor). During these tests, navigation of the drone was successful, as was data collection. The measurement set-up and results are shown in *Figure 23.* Note that the drone is slower to align with the target marker when the markers are placed horizontally. This is because the markers are seen at a larger angle of view, so the PnP transformation gives more inaccurate results.

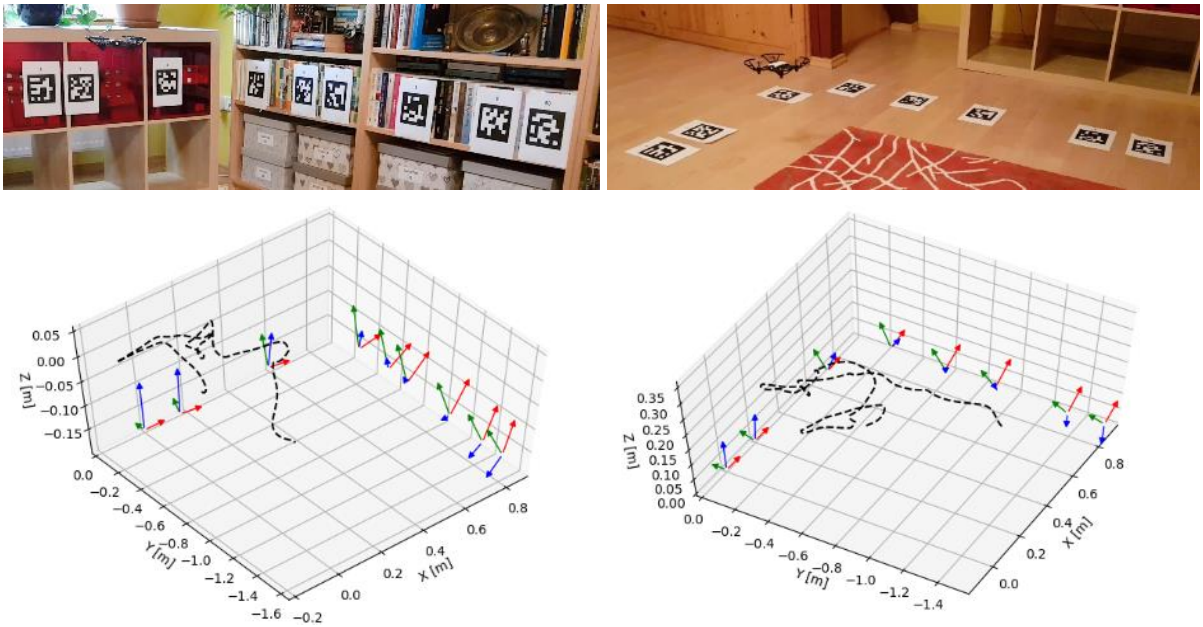


Figure 23: Results of measurements in vertical (left) and horizontal (right) marker placement

5.2. Alignment of systems, calibration

The data collected from the markers are interpreted in the coordinate system of the marker number 1, so the MoCap coordinate system must also be recalculated according to the marker. The simplest way would be to superimpose the coordinate system of the ArUco marker directly on the origin of the MoCap coordinate system, but this was not possible due to the design of the additional control paths. Marker 1, which is the origin, had to be movable.

To do this, we placed the 3 retroreflective markers of MoCap in the infrared region on ArUco marker 1, which can be seen in the right image of *Figure 24* three corners of the paper sheet. From this, we created a Rigid Body in the software, with the coordinate system center (Pivot Point) offset by a spherical marker. We even had to rotate the coordinate base, first around the y-axis by -90 degrees, then around the x-axis by -90 degrees, to get the coordinate system used by the drone software. The two coordinate systems are shown in the images in Figure 24.

Figure 24: MoCap on the left, marker coordinate system on the right
(red: *x* axis, green: *y* axis, blue: *z* axis)

After setting the coordinate systems, the drone was marked with 6 stickers. To ensure axis parallelism, the drone marked with the MoCap markers was also adjusted using the two wooden feet shown in the images in Figure 24. In the MoCap system, the Pivot Point of the drone's coordinate system was shifted to the center of the instrument's camera. (*Figure 25*) Due to the design of the drone, it is not possible to measure this precisely, and only estimated values were used. Thus, by treating the drone as a Rigid Body, a mapping was also created in the MoCap, whose data can be measured.

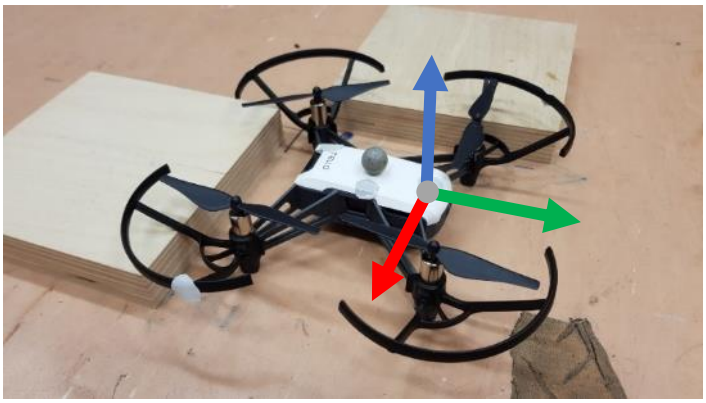


Figure 25: Drone mapped for the MoCap system

The calibrated coordinate systems were displayed in the OptiTrack system program as shown in the screenshot in *Figure 26* after calibration. The measurements of the MoCap system have to be converted into the coordinate system *"Drone_marker_cs"*, i.e. the coordinate system of the 1st ArUco marker. This can be achieved by homogeneous transformation on the data points according to *equation (3.5.)* The measurement points are rotated and shifted into the *"Drone_marker_cs"* system so that they coincide with the measurements of the drone according to the AR-marker.

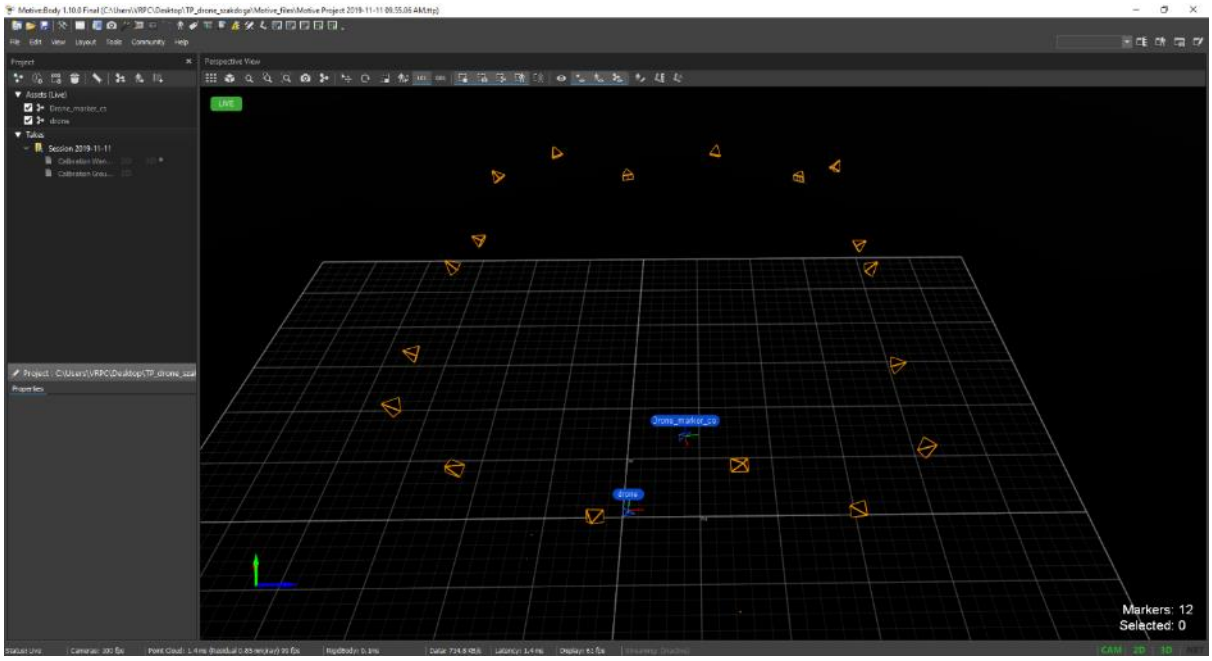


Figure 26: Calibrated coordinate systems in the OptiTrack software

The measurement data were saved to a CSV file. Since the MoCap system can sample at a maximum of 120 FPS, but the drone camera can only sample at 25 FPS at best, the MoCap sampling was set to 100 FPS, resulting in four times as much measured data. Synchronisation of the data points was also only possible afterwards, as the drone's software no longer included control of the OptiTrack software, so the two measurements did not start at exactly the same moment.

5.3. Comparison of the measurements, ArUco marker assignments

In parallel measurements, the drone was flown six times along two different trajectories. Only five of the measurements were used because a marker was misplaced during one of the measurements. From that point on, the drone turned in the wrong direction and did not reach the ArUco end marker, so the program did not return any measurement results. We would also note that the lighting in the motion lab probably did not favour the camera image of the drone, as it lost the ArUco markers it had placed much more often than in the preliminary home test environment. As a result, the drone's attitudes were slower and measurements were incomplete in several places.

5.3.1. FIRST MARKER PATH MEASUREMENTS

The ArUco markers were placed according to their type, the distances between them being established during preliminary experimentation. It is necessary to fix them to the floor, otherwise they would be blown away by the airflow generated by the drone rotors. The photo in *Figure 27* shows the path layout. The markers used in the layout, together with their meanings, are shown in *Table 3.*

Table 3: Markers used in the first measurement set-up and their meanings

Marker serial number	Report from	Target vector ([m] [m] [m] [°])
1	Origin	[0 0 0,9 0]
2	Right straight	[0 0 0,9 −45]
3	Right straight	[0 0 0,9 −45]
39	Right turn 1	[0 0 0,9 5]
40	Right turn 3	[0 0 0,9 −20]
4	Right straight	[0 0 0,9 −45]
13	Left straight	[0 0 0,9 45]
32	Left turn 3	[0 0 0,9 20]
50	End marker	[0 0 0,9 0]

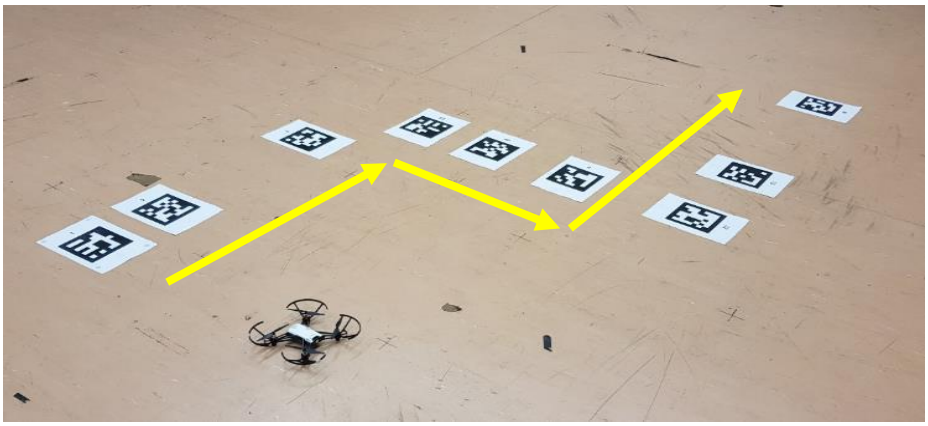


Figure 27: The first drone trail of ArUco markers for measurement

The drone successfully followed the established route three times. Unfortunately, its measurements were noisier than in the preliminary tests, due to the aforementioned poorer visibility of horizontal markers, the lighting conditions in the lab and the signal-to-noise ratio of the wifi connection. One drawback of the perspective PnP transform, for AR markers seen at an acute angle, is that the orientation can flip and then give incorrect data. An example of this is the marked marker in *Figure 28* with its axes in the wrong orientation. In the present case, this has just been filtered out by filtering for corner points, as one of its corner points falls on the edge of the camera image.

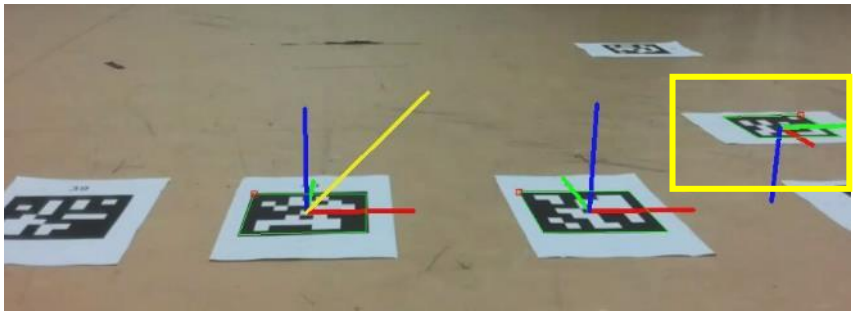


Figure 28: The z-axis of the candidate marker is not upwards but out of the image and downwards, as if the marker were on the ceiling.

The MoCap system data has not been filtered, consider it an accurate reference! To make the frequency of the measurements nearly uniform, we only work with every 4th element of the MoCap data set sampled at 100 Hz, although measuring from a drone camera only theoretically gives a value with 25 Hz sampling. Thus, the data points are already at nearly the same density. The filtered values of the drone-measured data are inaccurate compared to the MoCap data, but are correct in character. The drift error correction in the z direction, as shown in the data series in Figure 22 was applied to the data series. The two data series shown in *Figure 29* show that the range of axes is approximately the same, as is the nature of the path followed by the drone. The display of the drone's angular positions is shown in *Figure 30* on a cutout of the drone path. It can be clearly seen that the angular positions stored by our system are the same as those measured by MoCap.

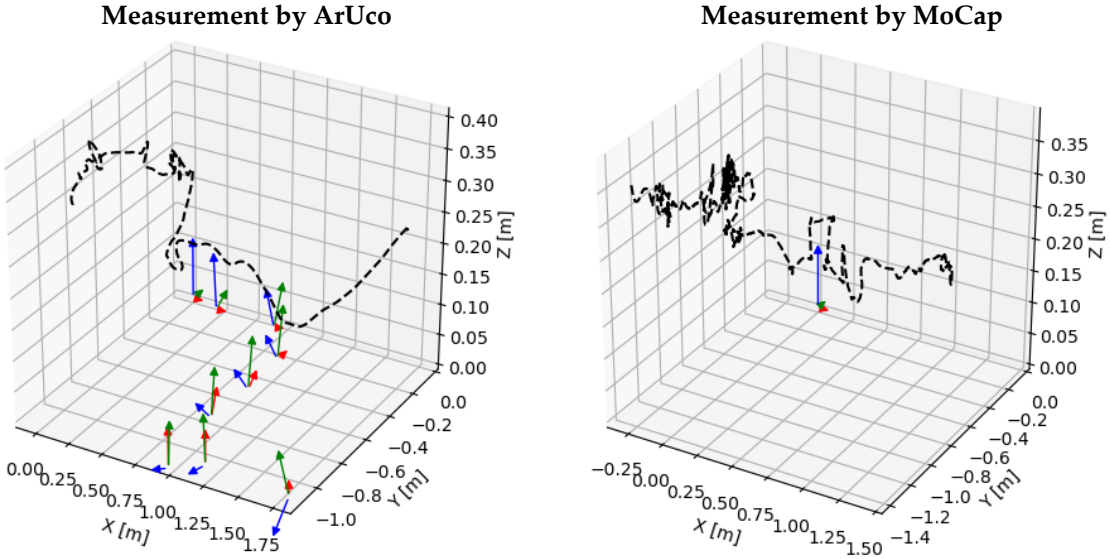


Figure 29: Results of measurement 1 based on the drone and MoCap system

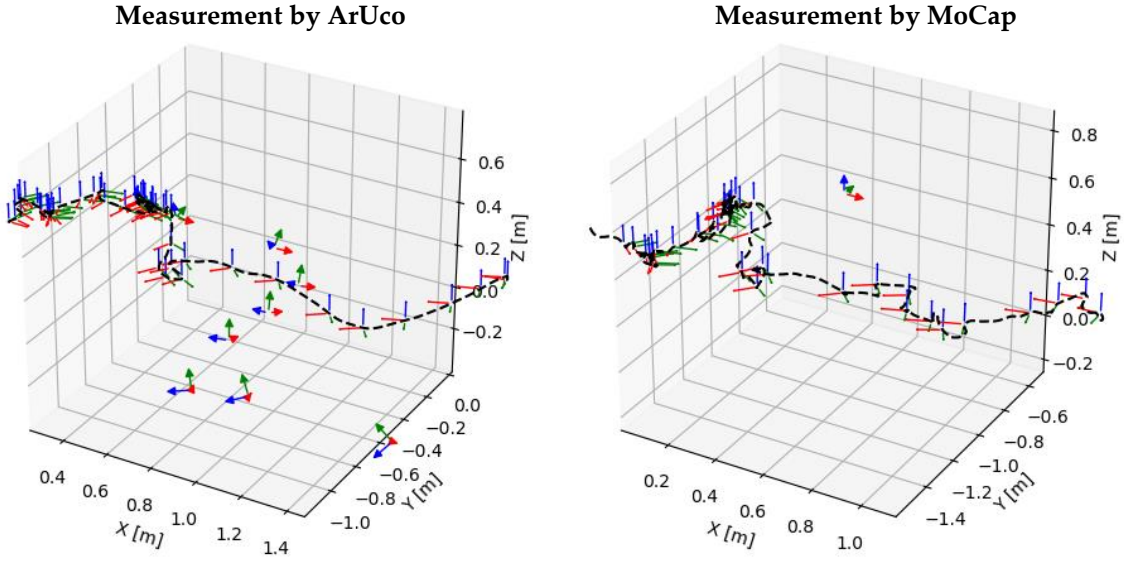


Figure 30: A detail of the data points of measurement 1,
coordinate bases with the corresponding orientation of the drone

The data of the 2nd measurement was not recorded, due to an incorrectly placed marker, but the 3rd measurement went smoothly. The only error was a storage error in the z-direction of the coordinate systems, which was successfully corrected by the aforementioned forced correction. Similar action can be taken for the 4th measurement to ensure that the drone's known altitude attitude is met for the data points. The results in *Figure 31* and *Figure 32* show that the nature of the corrected AR marker measurement points is the same as the MoCap result. It is true that there is still a difference in altitude, but it is no longer significant.

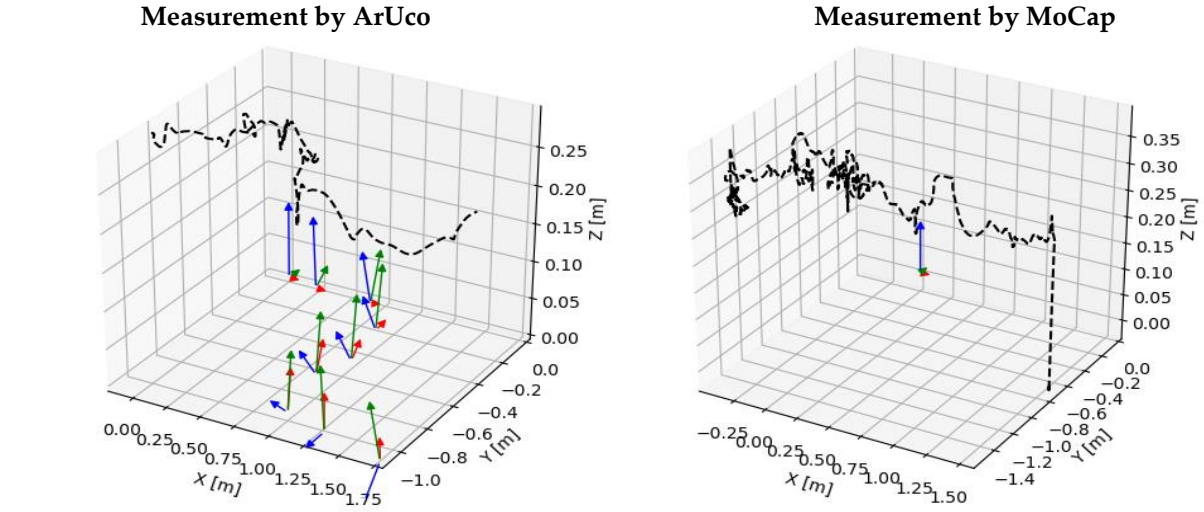


Figure 31: Data series of the 3rd measurement (the drone landing was also recorded in the MoCap)

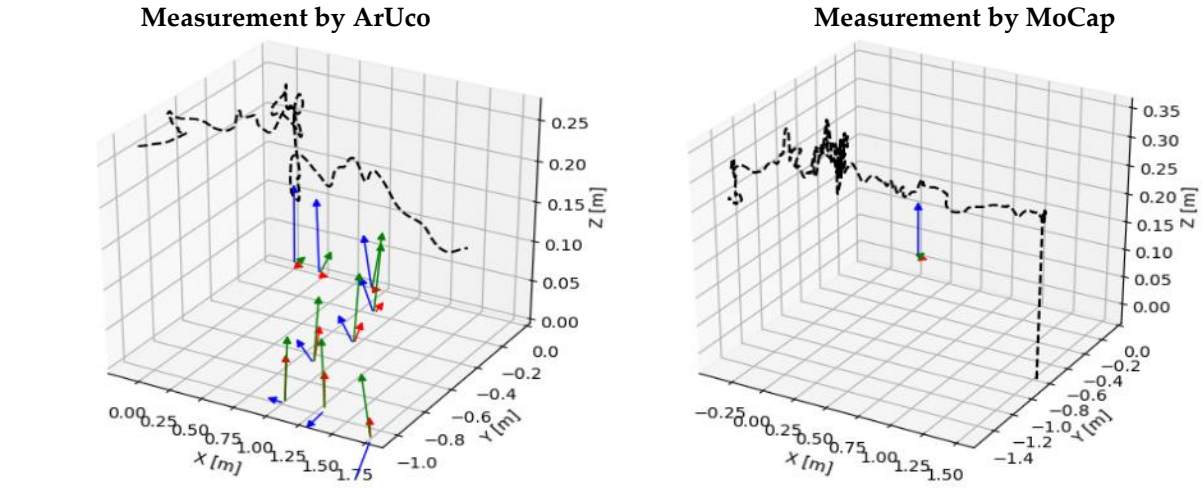


Figure 32: Data series of the 4th measurement (the drone landing was also recorded in the MoCap)

The drift defects can be successfully corrected by the surface fitting method, thus obtaining a satisfactory result in road processing. This problem was also included in our expectations *in section 3.5.1.* A real-time solution could be found by using better hardware, improving the ArUco marker recognition algorithm and better environmental conditions.

The orientation of the coordinate systems of the markers also appears to be subject to significant error, but these are distorted by the non-uniform axis representation of Matplotlib. This is why, for example, the arrows in the coordinate system in Figure 32 appear longer in the vertical direction.

5.3.2. SECOND MARKER ROUTE MEASUREMENTS

In the second measurement setup, 13 AR markers are used, their meanings are shown in *Table 4*, while their placement is shown in the photo in *Figure 33*. Here, the larger number of markers used is likely to have resulted in a larger drift error for the end markers as discussed in the previous paragraphs, but the post-plane correction can be applied to these as well.

Table 4: Markers used in the second measurement design and their meanings

Marker serial number	Report from	Target vector ([m] [m] [m] [°])
1	Origin	[0 0 0,9 0]
2	Right straight	[0 0 0,9 −45]
3	Right straight	[0 0 0,9 −45]
4	Right straight	[0 0 0,9 −45]
5	Right straight	[0 0 0,9 −45]
39	Right turn 1	[0 0 0,9 5]
40	Right turn 3	[0 0 0,9 −20]
6	Right straight	[0 0 0,9 −45]
41	Right turn 1	[0 0 0,9 5]
42	Right turn 3	[0 0 0,9 −20]
13	Left straight	[0 0 0,9 45]
32	Left turn 3	[0 0 0,9 20]
50	End marker	[0 0 0,9 0]

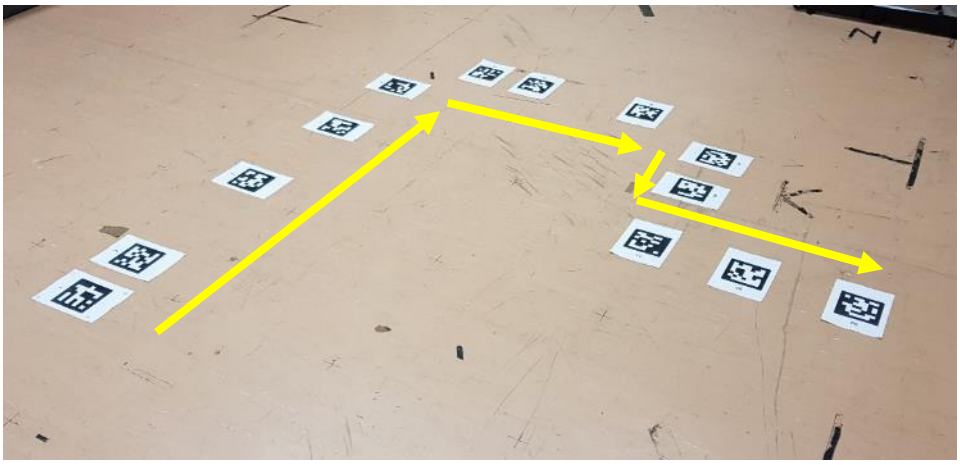


Figure 33: The second drone trail of ArUco markers for measurement

The three-dimensional data from measurement 5 perhaps best illustrate the similarity in the flight curve obtained by the two methods. The almost stationary movement of the first marker and the first left turn can be observed by comparing the two data series in *Figure 34.* The display range of the axes is also approximately the same, with some outliers in the MoCap case increasing the range in the z direction, for example. The position of the marker centres relative to each other is also very similar to the photo in Figure 33.

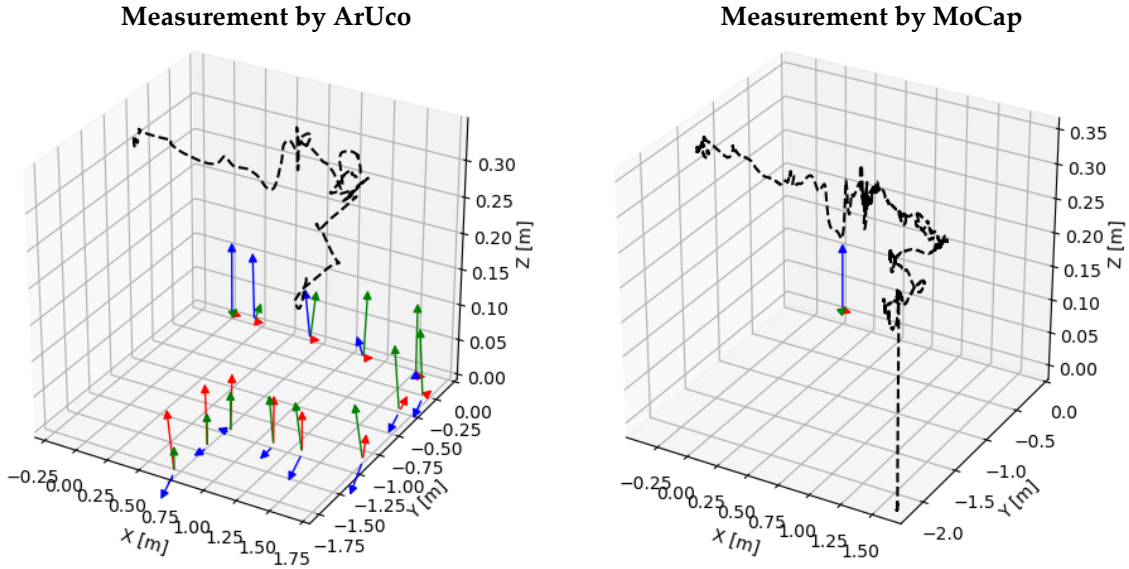


Figure 34: Data series from measurement 5 (the drone landing was also recorded in MoCap)

In measurement 6, the drift error was much smaller, and the advantage of the post-scan correction is that it corrects the data series to a lesser extent or almost not at all. The origins of the markers lie approximately in the same plane, so the measurement points were correct by default. As can be seen from the two data series in *Figure 35*, the nature of the measured trajectories is very similar in this measurement: the first left turn in both cases shows the same stuckness as before the end of the measurement. Comparing the data from measurement 6 and 5, it is clear that the drone followed almost the same path. Perhaps the AR data from measurement 5 has a slightly higher uncertainty, but the critical navigation points where the drone turns are clearly visible on both measurement lines. The accuracy of the values is greatly improved by averaging the values measured from the camera while seeing multiple markers at the same time, as previously conceived in the *Babinec et al.* [17] study.

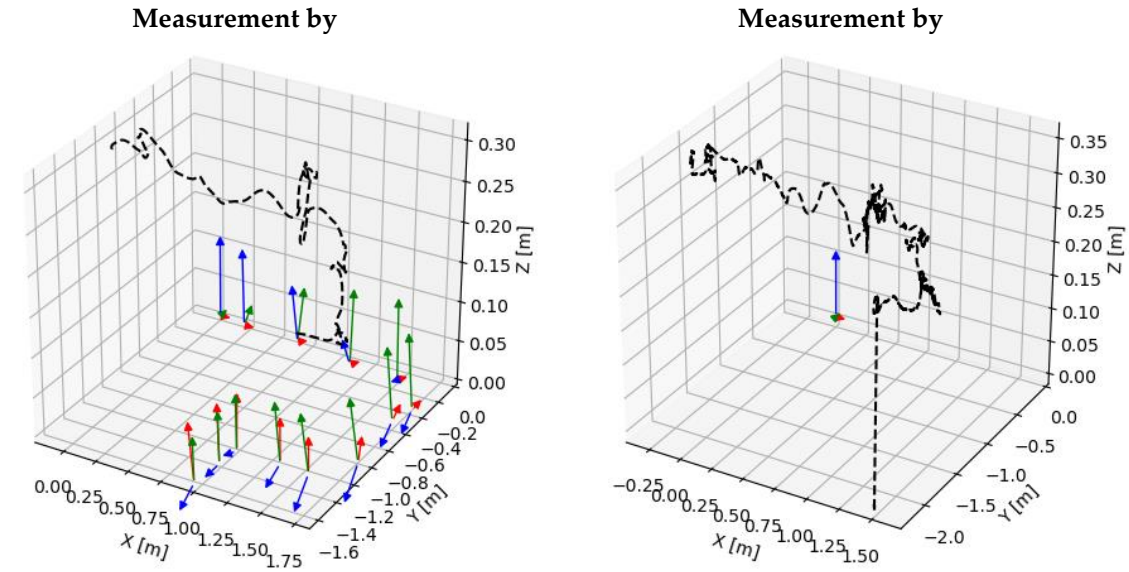


Figure 35: Data series from measurement 6 (the drone landing was also recorded in MoCap)

5.4. Video evaluation of the measurements

As already mentioned in the details of the programme developed, the image seen by the drone is automatically recorded by the system. In retrospect, the collected data points and angular rotations can be displayed in an animation using equidistant axes. A video of the measurements is included in *Appendix 2 of the* DVD accompanying the thesis.

A parallel frame from the video result of the first measurement is shown in *Figure 36*. The drone is marked by a coordinate base, with directions as described at the beginning of this paper, so the green y-axis points outwards from the nose of the drone. You can see how the drone is steering to the middle AR marker in the video, but it is also collecting data from the other two markers seen and averaging the measurements of three markers. The drone's orientation storage is fully consistent with what is seen in the recorded video, with possible slippage due to the animation's approximation of the refresh rate and poorly edited video. In general, the Tello drone's sensor provides good data even for unfiltered angular positions.

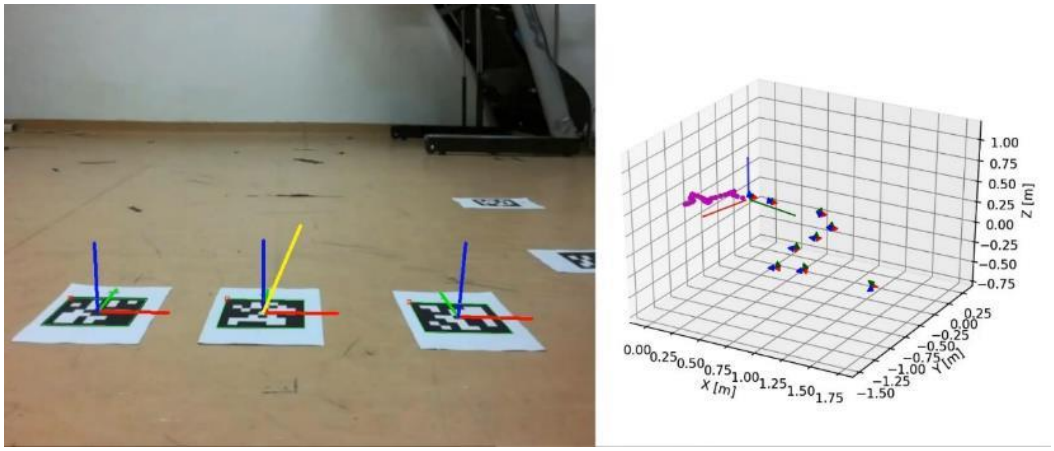


Figure 36: Video recorded during measurement 1 and animated flight path frames captured side by side at the same moment in time

The animation also uses an array of save time snapshots. Although it is not possible to load a time update into the Matplotlib animation, it is possible to display some frames for a longer time at the known FPS. From the time vector data stored during the measurements, it can be seen that the system stored the measurement data points in 0.07 seconds on average. This is slower than the 0.04 seconds required at 25 FPS, but this theoretical value was not met during imaging due to the drone's performance. Another reason for the slowdown could be the number of matrix operations performed per cycle. For example, when storing marker positions, the program performs many more calculations than when working from stored marker data only.

The drone path is displayed from the Kalman-filtered data, with "breadcrumb" points added every 12 data points. Theoretically, a waypoint appears in the animation at a measurement interval of 0.48 seconds, but this is not true as explained in the previous paragraph, the intervals are longer.

Figure 37: Image noise caused by decoding error during video transmission

During the 6th measurement, the messages on the console and the drone's flight were recorded from an external perspective. These are shown side by side in *Figure 38*. Some video transmission decoding errors are visible between the console messages. An example of this error is shown in the frame in *Figure 37*, where no marker can be recognized due to the noise in the image, the loop runs empty. This is caused by the degradation of the signal-to-noise ratio of the wifi connection of the device, whose design and performance only allows for such image transmission reliability. The data acquisition algorithm successfully overcomes these errors and, apart from a few missing data points, no error is introduced in the measurement.

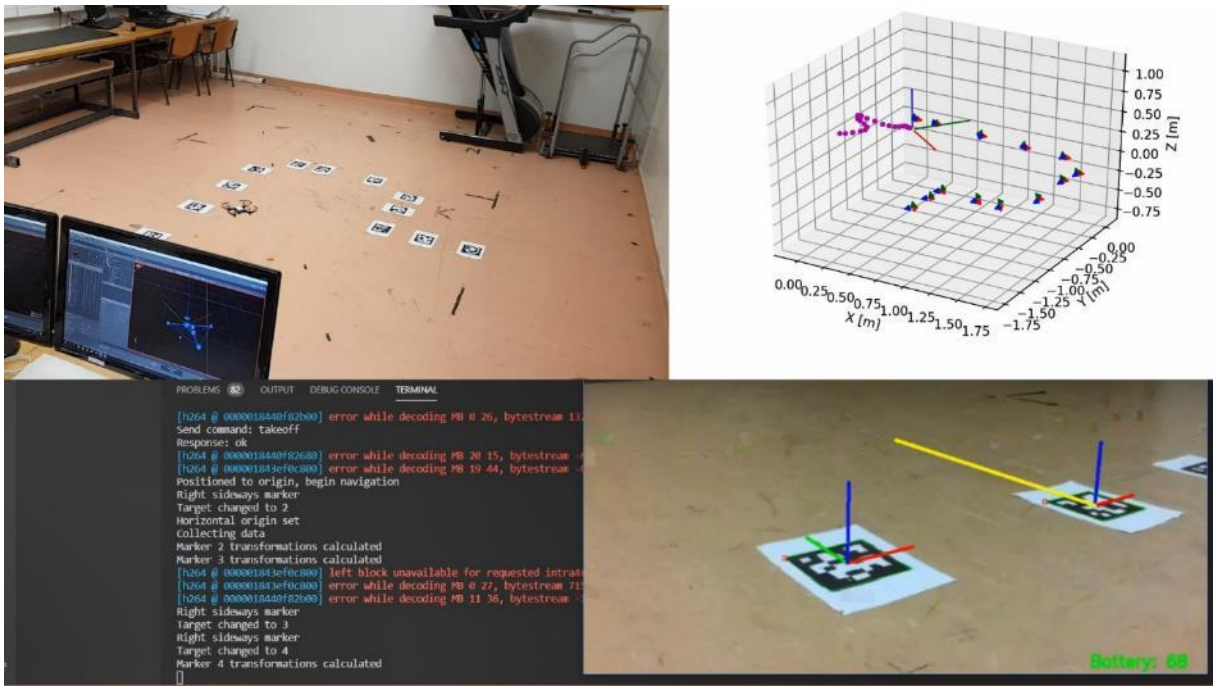


Figure 38: Screen shot, video and measurement points recorded in parallel during measurement 6

6. SUMMARY

6.1. Evaluation of results

The system developed, based on its test results, is capable of controlling the drone in a properly designed environment. However, the drone position measurements are uncertain, in some cases subject to large errors, and in other cases provide reasonably accurate values. Since the system works on the camera principle, ambient light conditions also affect its operation.

When writing the program, the aim was to make the system as general and adaptive as possible. Of course, this requires the development of a properly validated flight environment. Provided the right markers were placed, the drone was able to navigate along the established trajectory five times out of five measurements. On the other hand, an incorrect marker placement, as was the case in the 2nd measurement, already causes the navigation and the measurement to stop. Currently, it may be a problem that no backup of the half-finished data is obtained, however, one of the goals of this thesis was to make the drone capable of automatic operation, where data collection is conditional on the drone moving along the trajectory.

The computation of the transformations between the coordinate systems of the markers introduces errors that have approximately equal chance of correcting or increasing each other's errors. This part of the system is still uncertain and its appropriate correction needs to be developed. Once the transformations are established, the two resulting camera positions can be converted from the marker to the global coordinate system. These results show that the deviation of the obtained positions is in the range of ±0.05 m, as can be seen from the 9 calculation results collected in *Table 5*. The discrepancies are approximately coincident with the measurement errors of the ArUco markers, but such discrepancies accumulate during the recalculations. It is understandable that larger errors can be observed at the end of the measurements, when the global coordinate is calculated over up to 12 marker values.

Table 5: Global coordinate vectors calculated from the values of two adjacent markers
(The values of the vectors must be equal in pairs.)

$\mathbf{v_{1_{m0}}}$[m]	$\mathbf{v_{1_{n0}}}$[m]	$\mathbf{v_{2_{m0}}}$[m]	$\mathbf{v_{2_{n0}}}$[m]	$\mathbf{v_{3_{m0}}}$[m]	$\mathbf{v_{3_{n0}}}$[m]
$\begin{bmatrix}-0{,}3736\\0{,}2307\\0{,}7553\end{bmatrix}$	$\begin{bmatrix}-0{,}3816\\0{,}2160\\0{,}7580\end{bmatrix}$	$\begin{bmatrix}-0{,}4002\\0{,}2180\\0{,}7498\end{bmatrix}$	$\begin{bmatrix}-0{,}4004\\0{,}2207\\0{,}7444\end{bmatrix}$	$\begin{bmatrix}-0{,}4510\\0{,}2209\\0{,}7415\end{bmatrix}$	$\begin{bmatrix}-0{,}4690\\0{,}2141\\0{,}7164\end{bmatrix}$
$\mathbf{v_{4_{m0}}}$[m]	$\mathbf{v_{4_{n0}}}$[m]	$\mathbf{v_{5_{m0}}}$[m]	$\mathbf{v_{5_{n0}}}$[m]	$\mathbf{v_{6_{m0}}}$[m]	$\mathbf{v_{6_{n0}}}$[m]
$\begin{bmatrix}-0{,}0762\\0{,}0986\\1{,}0246\end{bmatrix}$	$\begin{bmatrix}-0{,}0865\\0{,}0443\\1{,}0374\end{bmatrix}$	$\begin{bmatrix}-0{,}0138\\0{,}0464\\0{,}8750\end{bmatrix}$	$\begin{bmatrix}0{,}0023\\0{,}0304\\0{,}8749\end{bmatrix}$	$\begin{bmatrix}0{,}0165\\0{,}0344\\0{,}4578\end{bmatrix}$	$\begin{bmatrix}0{,}0172\\0{,}0325\\0{,}4528\end{bmatrix}$
$\mathbf{v_{7_{m0}}}$[m]	$\mathbf{v_{7_{n0}}}$[m]	$\mathbf{v_{8_{m0}}}$[m]	$\mathbf{v_{8_{n0}}}$[m]	$\mathbf{v_{9_{m0}}}$[m]	$\mathbf{v_{9_{n0}}}$[m]
$\begin{bmatrix}0{,}1657\\0{,}0114\\0{,}5741\end{bmatrix}$	$\begin{bmatrix}0{,}1740\\0{,}0320\\0{,}5812\end{bmatrix}$	$\begin{bmatrix}0{,}1098\\-0{,}0285\\0{,}5635\end{bmatrix}$	$\begin{bmatrix}0{,}1131\\-0{,}0289\\0{,}5532\end{bmatrix}$	$\begin{bmatrix}0{,}1547\\-0{,}0084\\0{,}3571\end{bmatrix}$	$\begin{bmatrix}0{,}1452\\-0{,}0540\\0{,}3892\end{bmatrix}$

The solution to possible erroneous measurements is to optimise and improve ArUco marker recognition. This includes better detection of the marker edges (Figure 20), which can also cause measurement errors, and filtering of the PnP transform to eliminate axis reversals (Figure 28). Errors like these result in incorrect measurement points, which could be eliminated by improving the algorithm. There are already improved algorithms that promise better filtering under certain program environments (e.g. ROS). [31]

The determination of angular positions and their conversion into Euler angles also needs further development and filtering. Unfortunately, the 180 degree periodicity of the *atan2()* function used in the transformation from the rotation matrix [32] rendered the results unusable without filtering. Transformations with unfiltered rotation matrices can also cause measurement errors that cannot be corrected by the averaging used.

The system, despite its flaws, can to some extent correctly calculate the drone's trajectory, automatically steer the drone between markers and aggregate the data collected. The data read out from the drone via the UDP server is already well filtered, with values of angular rotations that can be used to measure, for example, the orientation of the drone. The readout program written for this purpose runs on a separate thread, does not load the main control system, but only collects data in the background.

The animation display showed that the per-cycle counting performed the data storage in 0.07 seconds on average, which is slower than the video playback at 25 FPS. This slower data acquisition may also cause errors, if not elsewhere, in the visualization. The included videos display measurement data with a frame refresh rate of 0.04 seconds, making it necessary to show some data points for longer.

The Tello drone used is not a professional device due to its size, the wifi connection is often noisy and the biggest problem is the camera image latency of almost 2 seconds. Despite this, the status readout is smooth. Although some sensors have been shown to give sometimes incorrect readings - such as the ToF camera altitude reading - the sensors are working well for the drone design. Of the control modes, the RC control used is able to position correctly with the speed values, with the larger error caused by the aforementioned latency.

The program is suitable for controlling an indoor micro-drive and has been able to successfully achieve its objectives. Its accuracy is not always adequate, both because of the lack of real-time filtering of the program, whether from image or position data, and because of hardware deficiencies. However, it can be concluded that the drone completed the course with a high degree of safety, using fully automatic navigation. The designed system could be suitable for navigation even in larger scale industrial environments. For example, in the case of a drone control within a hall, where the relative positions of the markers are known and the drone navigates and collects data based on the markers seen within the hall.

6.2. Proposals for further development

As already stated in the previous paragraph, real-time filtering is essential. Post-filtering of the measurement data points is not sufficient if there is an error in the storage of the coordinate systems of the markers. If a real-time Kalman filter could be set up, it would even be possible to display the data points immediately afterwards, so that the drone trajectory could be displayed in a near real-time solution.

Real-time operation also requires Nvidia's CUDA software package, which can be used to load matrices onto the computer's graphics card and perform matrix and image processing operations faster. This is currently not yet developed in Python, but linking libraries for the Python version of OpenCV may be available in the near future. Alternatively, the whole program could be rewritten in C++ or C#, where CUDA interfaces with OpenCV already exist.

The thesis uses only relatively cheap drones, even for home use. In the same way that positioning from ArUco markers can be considered a "low-cost" method. Of course, with better hardware, better operation can be achieved, but this requires an industrial or home-built drone. This would eliminate camera image latency for proper obstacle avoidance. With a faster wireless connection, the broadcast could arrive almost instantaneously and the drone control could operate at higher speeds. For the same reason, the controller setup error could be significantly reduced.

In a larger application, the system could be supported by external data. For example, to create a known map of the relative positions of the markers, so that the drone knows them as reference points. This method requires longer preparation, but may be more suitable for navigation in the longer term, as the positions of the markers are known and not subject to measurement error. It would be suitable for use in an industrial environment.

7. RESOURCES USED

1. ROBERTO SABATINI, SUBRAMANIAN RAMASAMY & ALESSANDRO GARDI (2015) LIDAR *Sensor Based Obstacle Avoidance System for Manned and Unmanned Aircraft.* Journal of Science and Engineering. 4. 1-13.
2. MALGORZATA VERŐNÉ WOJTASZEK (2010):*Photo-interpretation and remote sensing 3., Laser-based remote sensing,* University of West Hungary, Székesfehérvár
3. NILS GAGEIK, PAUL BENZ & SERGIO MONTENEGRO (2015) Obstacle *Detection and Collision Avoidance for a UAV With Complementary Low-Cost Sensors.* IEEE Access. 3. 1-1. 10.1109/ACCESS.2015.2432455.
4. SEBASTIAN SCHUON, CHRISTIAN THEOBALT, JAMES DAVIS & SEBASTIAN THRUN (2008): High-quality *scanning using time-of-flight depth superresolution. CVPR* Workshop on Time-of-Flight Computer Vision. 1 - 7. 10.1109/CVPRW.2008.4563171.
5. REFAEL WHYTE, LEE STREETER, MICHAEL CREE & ADRIAN DORRINGTON (2015) *Application of lidar techniques to time-of-flight range imaging.* Applied Optics. 54. 9654. 10.1364/AO.54.009654.
6. ASHUTOSH SAXENA, SUNG HEE CHUNG & ANDREW NG (2008): 3-D *Depth Reconstruction from a Single Still Image.* International Journal of Computer Vision. 76. 53-69. 10.1007/s11263-007-0071-y.
7. ALI AMIRI, SHING YAN LOO & HONG ZHANG (2019): SEMI-SUPERVISED *Monocular Depth Estimation with Left-Right Consistency Using Deep Neural Network.* 1905.07542v1
8. ASHUTOSH SAXENA, JAMIE SCHULTE & ANDREW NG (2007) DEPTH *Estimation Using Monocular and Stereo Cues, IJCAI* International Joint Conference on Artificial Intelligence, 2197-2203.
9. CLÉMENT PINARD (2019): *Robust learning of a depth map for obstacle avoidance with a monocular stabilized flying camera*. NNT: 2019SACLY003, Université Paris-Saclay, Saint-Aubin, France
10. ROSITSA BOGDANOVA, PIERRE BOULANGER & BIN ZHENG (2016) DEPTH *Perception of Surgeons in Minimally Invasive Surgery.* Surgical Innovation. 23. 10.1177/1553350616639141.
11. ANDREW J. BARRY, HELEN OLEYNIKOVA, DOMINIK HONEGGER, MARC POLLEFEYS & RUSS TEDRAKE (2015): Fast *Onboard Stereo Vision for UAVs,* MIT
12. XIAOYU CUI, KAH BIN LIM, YUE ZHAO, AND WEI LOON KEE (2014): *Single-lens stereovision system using a prism: position estimation of a multi-ocular prism*. Soc. Am. A 31, 1074-1082
13. H. ALVAREZ, L.M. PAZ, JÜRGEN STURM & D. CREMERS (2016) Collision *Avoidance for Quadrotors with a Monocular Camera.* 10.1007/978-3-319-23778-7_14.

14. MOHAMMAD FATTAHI SANI & GHADER KARIMIAN (2017) *Automatic navigation and landing of an indoor AR. drone quadrotor using ArUco marker and inertial sensors.* 102-107. 10.1109/ICONDA.2017.8270408.
15. KONSTANTIN YAKOVLEV, VSEVOLOD KHITHOV, MAXIM LOGINOV & ALEXANDER PETROV (2015): *Distributed Control and Navigation System for Quadrotor UAVs in GPS-Denied Environments. 10*.1007/978-3-319-11310-4_5.
16. LEVENTE RÉVÉSZ, TAMÁS SZEPESSY (2019): *Application of machine vision in greenhouse tomato cultivation*, Mechatronics project task, Department of Mechatronics, Optics and Mechanical Informatics, Budapest University of Technology and Economics
17. ANDREJ BABINEC, LADISLAV JURIŠICA, PETER HUBINSKÝ & FRANTIŠEK DUCHOŇ (2014): *Visual Localization of Mobile Robot Using Artificial Markers.* Procedia Engineering. 96. 10.1016/j.proeng.2014.12.091.
18. DAMIÀ FUENTES ESCOTÉ (2018): *DJITelloPy*, https://github.com/damiafuentes/DJITelloPy 20.VIII.2019 10.25 h. https://github.com/damiafuentes/DJITelloPy
19. RYZE ROBOTICS (2018): Tello SDK 1.3.0.0, https://dl-cdn.ryzerobotics.com/downloads/tello/20180910/Tello%20SDK%20Documentation%20EN_1.3.pdf/ 29 VIII 2019 16.30 h. https://dl-cdn.ryzerobotics.com/downloads/tello/20180910/Tello_SDK_Documentation EN_1.3.pdf/
20. DJI (2018): DJI-SDK/Tello-Python/doc/readme.pdf, https://github.com/dji-sdk/Tello-Python/blob/master/doc/readme.pdf https://github.com/dji-sdk/Tello-Python/blob/master/doc/readme.pdf8 VIII 2019 13.04 h.
21. FATTAHI SANI, MOHAMMAD & KARIMIAN, GHADER (2017) *Automatic navigation and landing of an indoor AR. drone quadrotor using ArUco marker and inertial sensors.* 102-107. 10.1109/ICONDA.2017.8270408.
22. MARK EUSTON, PAUL COOTE, ROBERT MAHONY, JONGHYUK KIM & T. HAMEL (2008) A *Complementary Filter for Attitude Estimation of a Fixed-Wing UAV.* 340 - 345. 10.1109/IROS.2008.4650766.
23. MATIAS TAILANIAN, SANTIAGO PATERNAIN, RODRIGO ROSA & RAFAEL CANETTI (2014) *Design and implementation of sensor data fusion for an autonomous quadrotor.* Conference Record - IEEE Instrumentation and Measurement Technology Conference. 1431-1436. 10.1109/I2MTC.2014.6860982.
24. ZHENGYOU ZHANG (2000) *A Flexible New Technique for Camera Calibration*, Pattern Analysis and Machine Intelligence, IEEE Transactions on. 22. 1330 - 1334. 10.1109/34.888718.
25. MOHAMMAD FATTAHI SANI & GHADER KARIMIAN (2017) *Automatic navigation and landing of an indoor AR. drone quadrotor using ArUco marker and inertial sensors.* 102-107. 10.1109/ICONDA.2017.8270408.
26. SCIPY.ORG (2014): *scipy.org/Docs/SciPy v0.14.0 Reference Guide/Interpolation (scipy. interpolate)/scipy.interpolate.splprep*, https://docs.scipy.org/doc/scipy-

0.14.0/reference/generated/scipy.interpolate.splprep.html 20.IX.2019 15.03h https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.interpolate.splprep.html

27. XIAO LU (2018) *A Review of Solutions for Perspective-n-Point Problem in Camera Pose Estimation,* Journal of Physics: Conference Series. 1087. 052009. 10.1088/1742-6596/1087/5/052009.
28. OPEN SOURCE COMPUTER VISION (2019): *OpenCV Docs, Camera Calibration and 3D Reconstruction,* https://docs.opencv.org/4.1.0/d9/d0c/group__calib3d.html#ga61585db663d9da06b68e70cfbf6a1eac 08.XI.2019 15.33h https://docs.opencv.org/4.1.0/d9/d0c/group__calib3d.html#ga61585db663d9da06b68e70cfbf6a1eac
29. DIN (1990): *DIN 9300-2 - Aerospace; terms, quantities and symbols of flight mechanics; movements of the aircraft and the atmosphere relative to the earth,* Deutsches Institut für Normung, Berlin
30. BME MOGI: *Movement Lab Wiki,* https://sites.google.com/mogi.bme.hu/mozgaslabor-wiki/mocap-rendszer?authuser=0 17.XI.2019 10.31h https://sites.google.com/mogi.bme.hu/mozgaslabor-wiki/mocap-rendszer?authuser=0
31. SMART ROBOTIC SYSTEMS (2015): *aruco_mapping_filter*, Image filter for better performance of ArUco detector, https://github.com/SmartRoboticSystems/aruco_mapping_filter 17 XI 2019 11.41h https://github.com/SmartRoboticSystems/aruco_mapping_filter
32. GREG SLABAUGH (1999): *Computing Euler angles from a rotation matrix*

8. LIST OF ILLUSTRATIONS