University of Technology
Eindhoven

Master thesis

Visual Simultaneous Localisation and Mapping on-board an AR.drone 2.0

DC 2017.100

Department of Mechanical Engineering
Dynamics and Control Group

Student:	Joris Knol
Identity number:	0863192
Thesis supervisor:	prof.dr. H. Nijmeijer
Coach:	dr.ir. A.A.J. Lefeber
Date:	October 23, 2017

Abstract

In this project a VSLAM (Visual Simultaneous Localisation And Mapping) algorithm is proposed to operate on the AR.drone Parrot 2.0 quadrocopter. There has been research to implement VSLAM on drones. However, those drones all had large computers onboard that are capable of doing these computations live. Projects that use VSLAM on smaller drones generally run the VSLAM algorithm on an external device. The limiting factor is the shortage of computational power. Therefore, the focus lies heavily on the reduction of computations. By doing this, a faster algorithm is proposed by sacrificing some robustness and accuracy. With simulations it is shown that the algorithm is capable of computing the location of the drone.

Contents

Chapter 1

Introduction and background

Unmanned Aerial Vehicles (UAV for short) have become more popular in the past few years. Mainly quadrocopters have risen in popularity for their application in many sectors, ranging from rescue missions to household support. Quadrocopters are commonly used as a platform for UAVs. They can be freely controlled over four Degrees of Freedom (DoF), it is thus an under actuated system. Many controllers are available for quadrocopters. One such controller was created by van den Eijnden and Jeurgens [21, 50]. They created a non-linear controller and implemented this on a AR.drone Parrot 2.0 quadrocopter [1]. Their work is part of a project which aims to have the AR.drone Parrot 2.0 (from here on simply referred to as 'drone') fly autonomously without using any external devices and in any environment. To clarify, the drone has to compute everything on its CPU and cannot make use of, for instance, GPS signals to determine its position since these signals are unavailable indoors.

Van den Eijnden and Jeurgens use an observer to estimate the state of the drone. The observer uses data from the onboard IMU. However, the measurements of the IMU are biased, resulting in offsets in, for example, acceleration. By integration of the acceleration to position an ever increasing or decreasing position is computed. This problem is known as 'drift'. The main reason for drift is that the new position is based on the old one, stacking errors. To compensate for this drift the drone now uses an external camera hooked up to a PC to update the position of the drone. The camera is mounted on the ceiling and the PC computes the position of the drone from its images. The computed position is then sent to the drone. The computed position is only given at a low frequency, because computation times are large. Therefore, the drone still has to use the integrated position and update this periodically with the externally computed position. This way the drone knows its position from the odometry and the external camera compensates the drift.

The drone now operates with the use of an external camera and PC, while the goal of the overall project is to have the drone fly completely autonomous. In order to achieve this goal, an alternative for the drift compensation is needed. A possible solution is to use the onboard camera instead of an external camera and use the onboard CPU for processing the images. This method gives rise to a few complications: heavy computations need to be done on the onboard CPU, the drone needs to recognise certain landmarks and needs to know the position of these landmarks. This last problem is a typical VSLAM (Visual Simultaneous Localisation and Mapping) problem. Although many quadrocopters are using a visual positioning method, almost all use external PC and/or camera to determine its position. However, Achtelik et al. [6] already proved in 2011 that it is possible to operate a quadrocopter completely autonomous and using the internal camera and CPU to visually determine its location. In that paper they used a different quadrocopter with a more powerful CPU, however their work is a promising start of flying the AR.drone completely autonomously.

1.1 Background

1.1.1 AR.drone Parrot 2.0

The most used platform for UAVs is the quadrocopter. The quadrocopter is used often because of its simplicity in construction. This leads to a relatively cheap platform. The AR. drone Parrot 2.0, Figure 1.1, is a fairly popular platform. It is a relatively cheap, even among quadrocopters, and an easy-to-access quadrocopter [45]. Previous research that used the Parrot 2.0 are, among others: [21, 30, 44, 50].

Figure 1.1: AR.drone Parrot 2.0 power edition

The most important specifications for this research are the positioning sensors. The Parrot 2.0 has multiple sensors available (see Appendix B for more details on the specifications):

- Ultrasound altimeter;
- 3 axis gyroscopes;
- 3 axis accelerometers;
- 3 axis magnetometer;
- Front camera;
- Vertical camera.

Jeurgens [21] has already implemented the sensors, apart from the cameras, to create a state estimate of the drone, the observer. The state generated this way is subjected to drift. The drift occurs by integrating the measured data of the gyroscopes and accelerometer, the collective of these sensors is referred to as IMU: Inertial Measurement Unit. The IMU data is subjected to a bias, and by integrating this bias the drone thinks it is slowly accelerating in some direction while it is not. This causes drift that is clearly visible in experiments.

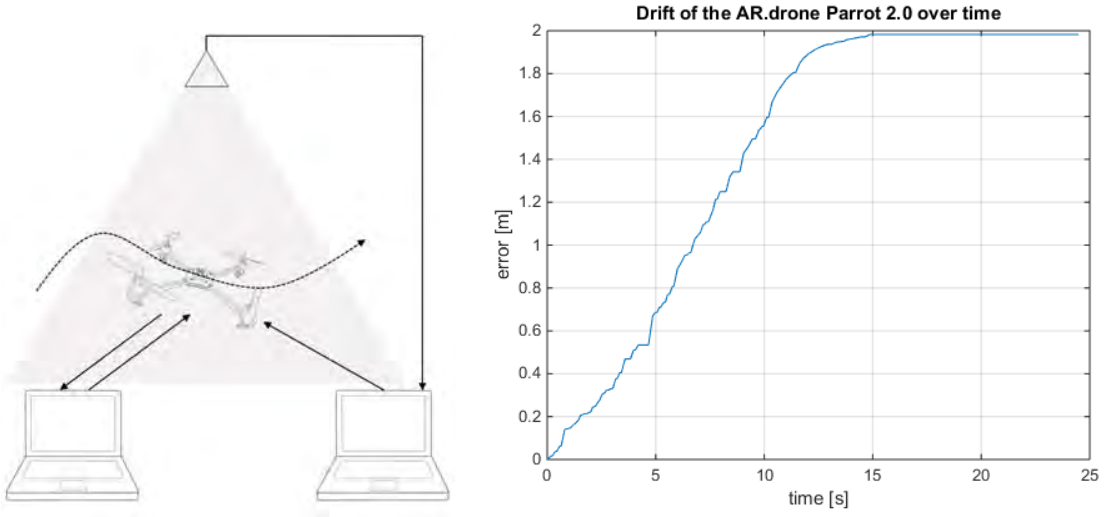
The AR.drone is equipped with a *OMAP 3630 1GHz ARM cortex A8*. The ARM cortex includes a *TI C64x DSP 800MHz*. In total the onboard computer has 128MB of RAM and 128MB of flash memory. All specifications of the drone can be found in Appendix B. The onboard computer power is a major limitation in the project. The processor pales in comparison with the computer power of the average laptop. The storage capability of the small RAM is limited. Computations with the camera images are computationally heavy by nature. That is because the images contain a lot of data/pixels and are thus large. In general, for image processing, computations are done for each pixel. In case of the bottom camera of the drone, there is a total of $320 \times 240 = 76800$ pixels, and $1280 \times 720 = 921600$ pixels for the front camera. Each pixel contains multiple channels,

further increasing the amount of data and thus computations. Moreover, image processing itself is only part of the VSLAM algorithm. This means that the computations need to be extremely efficient.

1.1.2 Experimental set-up with AR.drone

The drone drifts by only using the IMU. Therefore, [21] and [50] use a camera hanging from the ceiling that tracks the drone. Because of the drift problem the observer currently only uses the in-plane position estimate (the plane parallel to the ground) with an external camera. Without the camera input the in-plane position is set to zero [22].

The experimental set-up is illustrated in Figure 1.2a, and can be described as follows:
The drone is equipped with a LED-strip so that it is easier detected in the camera images. The camera hanging on the ceiling sends it image to a laptop that processes the images. This laptop tracks the drone and derives its position. The derived position is then sent to the drone with a WiFi connection. A second laptop sends commands to the drone such as trajectories. The controller of the drone operates onboard the drone itself.



(a) Experimental set-up *(b) The drone's error position in meters over time*

Figure 1.2: Experimental set-up and result

For the project it is important that the drone can fly drift free. An experiment to see the influence of drift is made. The drone is given the command to hover in place. The input from the external camera is not used, thus the drone determined its in-plane position to be at the origin. The result is shown in Figure 1.2b. In this figure the error is plotted over time. The error is the in-plane position error, as it is seen from the top camera. The error is computed by taking the distance of the drone position estimate (which equals zero) and the top camera. Clearly can be seen that the drone drifts away while the drone itself thinks it stays in place. In only 15 seconds it drifts 2 meters away, thereafter it lands and stops moving. This drift error needs to be removed before the drone can be controlled accurately.

1.1.3 Recommendations from previous work

Jeurgens reported in [21] some recommendations for future work on the project. Part of the recommendations involve improving the position estimate of the drone. The current set up seems to have too many flaws and limitations in practice. Therefore, this research focusses on improving this position estimation. The relevant recommendations are listed below:

- It is advised to use the DSP on the drone for computations, rather than only the CPU;
- The image processing is currently run in *Matlab*. It might work faster if it were programmed in, for instance, C-code. C-code is more efficient and thus runs faster. This way the delay as a result of image processing can be decreased.
- The internal camera should be used instead of the external one, because this is a goal of the overall project and because the external camera cannot detect the drone if it makes large angles. The large angle turns the LEDs on top of the drone out of view from the camera.

In addition to the other recommendations, [50] recommended the following:

- Instead of using the topcamera some other methods could be looked at to improve the location estimation of the drone. Other methods that could be looked at are: GPS, Vicon and optical flow.

The problem faced is the method used to give an accurate position estimation of the drone. Using the IMU as a basis, combined with GPS, Vicon or optical flow, only the drift needs to be compensated. A quick overview of the proposed methods is given here:

- GPS is a possible way of determining the position of an object, unfortunately, the GPS is too inaccurate for small manoeuvres and not available in most indoor environments.
- Vicon [5] is a commercial product that uses visual tracking to determine locations of objects. Vicon uses VSLAM, and is the most promising solution at this moment. More on this can be found in Section 2.2.
- Optical flow appears to be a good option, however as is pointed out later-on in this report, it is not suitable to compensate for drift.

1.2 Project goal

Conclusively, the problem is that the drone drifts with the use of only the observer. An algorithm is needed that can compensate for the drift. The algorithm must be very efficient to operate on the drone's small computer and be accurate enough for practical position estimations.

The goal is to derive a VSLAM algorithm that is capable of positioning the drone without the need of external equipment to assist the positioning.

The problems that are faced to reach this goal are listed below:

1. Foremost, the lack of computational power is limiting the options for VSLAM. The expected result is that the algorithm will not be able to run at the normal frame-rate of 30Hz. However, the positioning needs to be computed at a faster sample rate. In order to keep the sample rate as high as possible the VSLAM algorithm must be kept as efficient as possible. This is achieved by splitting the algorithm in steps and choosing the most efficient method for each step. Eventually the goal is to create a VSLAM algorithm that is capable to run on small devices such as the drone.
2. The VSLAM algorithm must be implemented on the drone alongside the controller. The only realistic option is to implement the algorithm on the drone's DSP. In addition, the algorithm must communicate with the controller.

In this report is a VSLAM algorithm proposed that focuses on the first problem.

This report is divided into different parts. The first part explains why VSLAM is necessary and shortly explains problems associated with VSLAM. It also enumerates the steps taken in solving a VSLAM problem. Next, these steps are described one after the other. Thereafter, simulations with the algorithm are given. The conclusions and recommendations are given in the final chapter.

Chapter 2

Optical positioning methods

In summary, the main goal is to have the drone compensate for drift. In previous work this compensation was achieved via an external camera. However, because of delays in computations and communication between drone, external camera and external computer, the drift compensation is always late. Additionally, the camera cannot see the drone when it flies under a large angle. As a solution, the onboard camera is used to compensate for the drift. By using the onboard camera the communication delay is eliminated and it becomes possible to fly outside the range of the external camera.

Compensating for drift is possible with visual compensation. The two cameras onboard the drone can be used for input. There are mainly two ways of using the visual input to compensate for drift:

1. Optical flow and
2. Visual Simultaneous Localisation and Mapping (VSLAM).

The two methods are described in the next sections, first the optical flow and thereafter VSLAM. In both sections research of different sources is used to validate the usefulness. After comparing both options the most promising one is chosen for further research. Near the end of this chapter, the processes for the visual drift compensation are split into different parts. These process parts are split in order to handle them one by one in following chapters. The basic build up of those chapters can also be found there as well. This provides an outline of the rest of this report. At the end a small summary is given of the chapter and of conclusions made.

2.1 Optical flow

Without going into too much detail, optical flow estimates the change in camera position from two subsequent images. It computes the shift in pixels and from there it computes the velocity of the camera, see Figure 2.1 for the shift in pixels. The shift in pixels is computed for every pixel. The shift is usually computed by taking the gradient of two subsequent gray-scaled images. By assuming that the shift of the pixels is small, the equations to derive the velocity are very small, thereby requiring almost no computations.

Optical flow requires little computational power because it consists of fairly simple equations. This is a really good prerequisite if computation must be done on the drone itself. Some previous studies attempted to compensate for drift using this method. Optical flow seems to work well when the drone is simply hovering in place. Marx [35] has shown this to be true. However, when the drone is flying a trajectory, the optical flow method does not give satisfying results, according to Li et al. and Briod et al.[13, 29]. Their results are given below.

Li et al. [29] show the actual computations of optical flow. Optical flow methods use odometry to compute the estimated position. It computes the distance travelled from one frame to the next and adds this to the previous estimated position, stacking the error with each new estimation. Their tests show some drift compensation but as they state themselves: *"Position drift is always a*

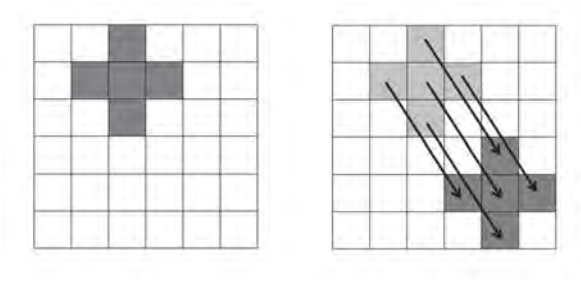


Figure 2.1: Schematic optical flow representation. Left is the first image and the right represents the shift in pixels from the first image to the next.

problem for any form of odometry measurement and from the figure can be seen that the maximum error accumulated can be up to 0.4 m and for non texturerich ground surface, the result will be worse. Thus, the method based on visual odometry can not be used as position measurement for feedback control."(Kun Li et al. 2016, [29, Chapter V. Results]). The main reason for the accumulated error is that optical flow cannot accurately estimate the distance travelled, it is accurate in estimating the direction, but not the distance.

Briod et al. [13] try to increase the accuracy of optical flow. The use of multiple optical flow sensors (up to eight) gives a more accurate position estimation. With these sensors attached to a quadrocopter, they tested and measured the position error. In their test set-up they accumulated a position error of about 0.4-1.7[m] in 2 minutes. However, this works best in large, flat environments such as large, empty rooms. Also, the AR.drone only has two cameras, and their study showed that with only two optical flow sensors the error increased dramatically to 11[m] in 2 minutes.

Overall it can be stated that optical flow is not a good solution for drift compensation, because accuracy is too low. The drone has to actively track landmarks in order to prevent a new drift due to stacking of position errors as much as possible. Thus a VSLAM system is needed. A system that recognises a landmark and can keep track of this landmark as time progresses does not stack position errors.

2.2 VSLAM

A detailed explanation of VSLAM is given below with the help of an example. But as a rough description, VSLAM works as follows: a VSLAM algorithm builds a map of the environment. The map consist of landmarks. The position of the drone is computed from landmarks it detects with the camera. When determining the position of the drone (localisation) from its camera images, one needs to know the position of the landmarks it detects. Landmarks are points in 3-D world space. The landmarks form a 3-D map (mapping) from which the drone can localise itself. This is in short what a VSLAM system does. Note that VSLAM is a specific type of SLAM, Simultaneous Localisation and Mapping, using camera images to find landmarks, while a SLAM system can use any means to find landmarks.
SLAM can be separated into three categories:

1. Landmark positions known
The camera position is derived from the position of landmarks. For instance, a moving camera could see landmarks and derive its position from them. The position of these landmarks are known and stored in a database before flight. Some papers that use this method are [12, 26, 37, 44]

2. Camera position known
The position of landmarks can be derived from the camera images, with a known camera position. However, in this study the position of the camera is what needs to be derived.

3. Camera position and landmark position unknown
Some SLAM algorithms use the camera location to determine the landmark locations and it uses the landmark locations to determine the camera location, forming a loop. Papers that use this third category are, among others, [6, 20, 24, 25, 46, 48, 49].

Since the goal of the project is to have a drone flying without the use of external equipment, the third category of SLAM is needed, even though it is the most complex one. The determination of the camera and landmark positions can be seen as separate problems. Therefore, during the derivation of the final VSLAM algorithm, these problems are solved separately and fused later-on to create the third type of VSLAM.

The only problem is that this derivation of positions forms a loop problem. The loop problem is solved by initialising around the starting position. This location can be chosen as the origin. Then by flying around the environment the drone can start creating a map which is then used to compute the position. This is possible since the drone can estimate its position apart from SLAM.

2.3 VSLAM example

First it is clarified how SLAM works when the camera position and landmarks are unknown. For simplicity consider a 2-D example, a robot driving through some unknown environment, see Figure 2.2. The situation is similar except it is in 2-D rather than 3-D. The example robot also uses one camera. In the figure the path of a robot is visualised. The robot is represented by the rectangle with the pointed tip and the landmarks are the black dots. The robot starts in a known position to the left, consider this the origin.

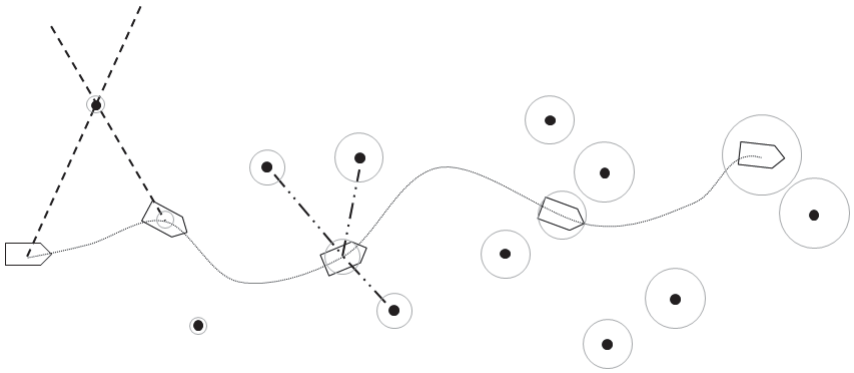


Figure 2.2: Visualised 2D path with SLAM

When the robot is moving it can map the environment. It needs at least two measurements to derive the position of a landmark, as can be seen by the first two robot positions to the left. At the first position the robot locates the landmark under some angle (the dashed line), as it also does in the second position. The crossing point of these lines is the estimated location of the landmark. Naturally, the state of the car should be known in order for this to work. So the robot must have a general idea of its state. This state derived in the observer. As the robot progresses, it computes the location of every landmark it comes across while continuing its path.

With known landmark positions the state of the robot can be derived. This is visualised on the third position. The robot detects three landmarks under certain angles (the dashed-dotted lines), with the three detected landmarks, there is only one place the robot can be located. The robot needs to detect at least three landmarks or else it can not find its unique location. This holds true because by having three landmarks (landmarks A, B and C), three sets of landmarks can be made (sets A-B, A-C and B-C). Each set provides a circle where the location of the robot

could be. By combining two sets, there are two locations the robot can be, because two circles can intersect in two places, the third set results in a circle that only crosses one of the two points. As a result, there is only one unique location the robot can be. The robot locates landmarks and its own position at every measurement.

SLAM is an estimation, it estimates the location of the landmarks and it estimates the robot's state. The accuracy of these estimations is important and is visualised by the grey circles. The starting point is set as the origin and thus is 100% accurate. Then, when the second measurement takes place after an arbitrary amount of time, the robot has moved and the location of the robot becomes an estimation. The robot is computing the location of the landmark with the estimated position, resulting in an estimation for the landmark. Even if the sensor is completely accurate the landmark position cannot be more accurate than the robot's position. The same applies when the robot estimates its position from the landmarks. As a logical consequence, the accuracy keeps decreasing while the robot continues its path. So the first estimated positions, of both robot and landmark, are the most accurate.

In a sense SLAM algorithms are unstable, the further they travel the larger the error becomes. As the travelled distance reaches infinity, so does the accuracy. However, this is only applicable for a trajectory that keeps travelling to new environments, never returning to an old environment. If the trajectories are bounded so is the accuracy. If the robot is only allowed to travel within a limited space then it will eventually returns to an old environment. With the old environment being more accurate the localisation accuracy is reset to the accuracy of that old environment. Thereby ensuring that the position error of the robot is bounded. The VSLAM map works best when it is used in a local area because of this. In a local area it detects the old landmarks more often, keeping the deviation at a minimum. Noteworthy is that the accuracy of the whole map can never be better than the accuracy of the most accurate landmark.

Over the years many different SLAM methods have been created. (V)SLAM methods like: EKF-SLAM [9], FastSLAM [10] and more recent methods like: PTAM [24] and ORB-SLAM(2)[39]. All these methods are SLAM systems, but almost all only prescribe a part of the whole (V)SLAM problem. Therefore, the different methods that are available for each part of VSLAM are compared and evaluated. VSLAM is broken up in different steps, each step is treated separately and for each of those steps the best option is chosen. These different steps are given in the next section.

2.4 Steps taken in VSLAM

The VSLAM algorithm needs to make multiple steps. These steps are visualised in Figure 2.3. Naturally these are not the only steps taken, in practice other steps need to be taken like preprocessing images, but they can be part of these steps.

1. **Visual tracking**
   First the camera images are processed. Visual tracking includes detecting landmarks in the images and describing them so they can be recognised again at a later time. The visual tracking is a major step in VSLAM. Image processing is computationally heavy and a major concern for the whole VSLAM algorithm. Visual tracking is divided into two minor steps, feature detectors (Chapter 3) and feature descriptors(Chapter 4). As the name suggests feature detectors detect features in images and feature descriptors describe the features. Choosing the appropriate detector/descriptor is focussed on efficiency and computation times.

2. **Map building**
   The map consists of all landmarks that were detected. Each landmark consist of a 3-D position, deviation factor σ and a description of the landmark so it can be recognised. With the location of the camera the positions of the landmarks are derived: the landmark positions are computed or updated or only stored, depending on how many times they were detected. The

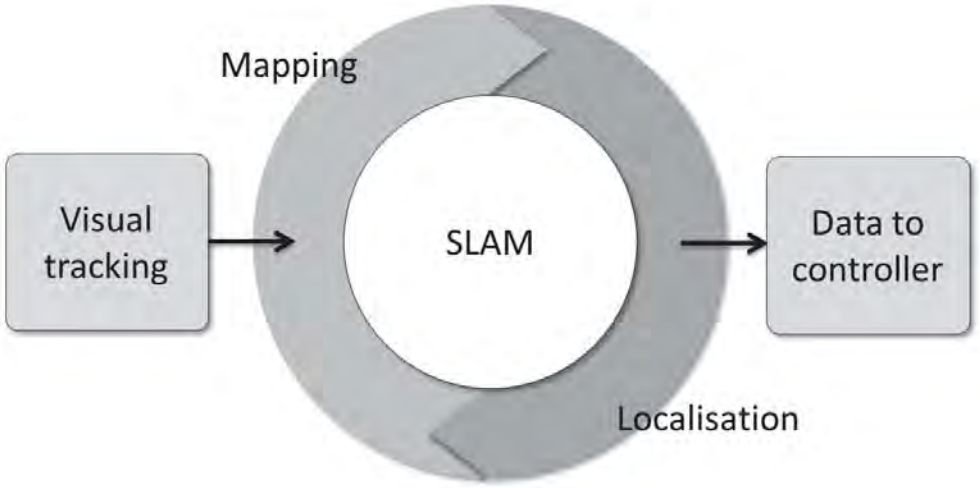


Figure 2.3: Visualised loop of VSLAM system

detected landmarks that were never detected before are only stored, the landmarks that were detected once before have their position derived and landmarks that were detected multiple times before are updated. Apart from the position computations, the map building consists also of matching the landmark descriptors.

3. **Localisation of the drone**
   The detected landmarks with known positions can be used to localise the drone. The drone location is combined with the odometry of the accelerometers to recreate the true state of the drone. This data is then sent to the controller onboard the drone. Some complications are that the odometry has a different update rate than the VSLAM part. The update rate of the IMU is roughly 200Hz and the update rate of VSLAM is 30Hz at best (the frequency of the camera) but is expected to be 1Hz or even less. The VSLAM algorithm basically gives the controller an update every once in a while saying it should redirect its course.

These steps are handled one by one in separate chapters. First the *Visual tracking* step, then a chapter for the *mapping* step and thereafter for the *Localisation*.
The chapters are constructed in the following way:

1. The chapters start with an introduction into the subject and how it is related to the main problem. The subject of the chapter is a sub-problem.
2. In the next section possible solutions for the sub problem are given and compared.
3. From the different solutions, one is chosen and further described in the next section. There the solution is described in more detail and possible adjustments are made.
4. The implementation of the solution is given the section thereafter. This section contains tests for speed and robustness.
5. At the end of each chapter a summary and conclusion is presented, containing a short overview of the choices made.

2.5 Concluding visual drift compensation

Using a camera of the drone there are two methods for visually compensating for drift: optical flow and VSLAM. From earlier research it is clear that optical flow is not suitable for localisation applications. In contrast, VSLAM is suitable and has been applied for drift compensation on drones multiple times, making it the best choice for this research.

In order to create a VSLAM algorithm that is sufficiently fast, it is broken into different steps. In each of these steps the most appropriate solution is chosen. VSLAM can be split into multiple processes and these processes are described separately in the next few chapters.

1. Visual tracking: track and trace landmarks in images using:
   (a) Feature detectors and
   (b) Feature descriptors.
2. Localisation: compute the position of the landmarks and with the position of the landmarks compute the position of the drone.
3. Mapping: storing and managing the landmarks in the drone memory.

Chapter 3

Visual tracking: Feature detectors

For the VLSAM algorithm the images taken by the camera of the drone are the main source of informational input. Getting the information from these images is done with visual tracking.

Visual tracking algorithms are increasing in popularity, especially with the increase of augmented reality. In this case visual tracking is needed as input for the VSLAM algorithm. The computations needed for visual tracking are still large. However, with the yearly increase in computational powers, real time visual tracking is available to ever shrinking applications. In the application, it is attempted to implement such a visual algorithm. The computational heavy nature of these visual algorithms is a problematic issue for the AR.drone. So the challenge is to create a very fast and efficient visual tracker.

The first step in VSLAM is gathering data from visual input. The algorithm needs landmarks to pinpoint its position with. The landmarks must be detected and recognised. Visual tracking can be split into two functions:

1. feature detection,
2. feature description.

First is the *feature detection*. In images certain features should be picked that can be detected again in the next image. After finding these features in an image, the features should be described. By describing the features the algorithm can cross-link the detected features between images. This describing of features is done by a *Feature descriptor* and cross-linking these features is commonly called *Data association*.

The fore mentioned functions: feature detection and feature description are separately handled in different chapters. In this chapter feature detection is explained and feature description is explained in Chapter 4.

The chapter has the following structure: first a list of different methods is given and one of them is chosen for the VSLAM algorithm. Thereafter, the chosen method is described in more detail and adjusted to the application. Lastly the method is implemented on the drone and tested.

3.1 Comparing different feature detectors

Feature detection is, as the name suggests, finding *features* in images. Key is to detect the features as efficiently as possible. Fortunately, much research and work is available to compare the different methods. However, all use either different testing images or different computers.

First the various most commonly used feature detectors are compared. OpenCV [4] provides an assortment of these detectors. Tables 3.1 untill 3.5 show an example of computation time of multiple detectors. The detectors listed are: FAST, SURF, STAR, MSER and GFTT. A short description of each of these detectors is given below.

1. **Features from Accelerated Segment Test: FAST [42]:**
   FAST is a corner detector that tests every pixel separately. It finds a corner by testing a circle of surrounding pixels. The surrounding pixels are tested for brightness. If, for example, $\frac{3}{4}$ of the pixels is brighter, a corner is found. The power of this method is its simplicity.

2. **Speed Up Robust Features: SURF [11]:**
   A faster implementation of SIFT [33] is SURF. SIFT [33] is known as one of the most robust detectors. It uses the Laplacian of Gaussian of the image on multiple scales and wavelet blob detection which is Hessian based. The result is a scale and rotation robust detector. SURF speeds this method up by using a block filter to approximate the Laplacian of Gaussian.

3. **Center Surround Extrema: CenSurE [7]:**
   Also known as the **STAR** feature detector in OpenCV. Similar to the SURF detector, it is a multi-scale detector. The STAR detector uses a bi-level approximation for the Laplacian of Gaussian. This gives it a higher accuracy than SURF, but reduces the number of detected features.

4. **Maximally Stable Extremal Regions: MSER [36]:**
   Maximally Stable Extremal Regions, or MSER for short, is the detector of SIFT. Just as mentioned in SURF, it computes the Laplacian of Gaussian and uses a Hessian blob detection. This is a very robust but slow method.

5. **Good Features to Track: GFTT [47]:**
   GFTT stands for Good Features to Track. It is based on the Harris corner detection, developed by Shi-Tomasi. Just like FAST it tests every pixel to test if it is a corner. For every pixel it tests the difference in brightness. It tests pixels in the surrounding area and with a slightly shifted center. If the brightness in one direction is different from the rest, the tested pixel is a corner.

For the application it is essential to have a fast algorithm. All above methods have proven themselves in practice and can all be used for stable VSLAM. During this reseach only the speed of all methods is compared. If one would like to compare the robustness/repeatablity, see [40]. Tables 3.1 untill 3.5 clearly show that the fastest method by far is the *FAST* corner detection. This conclusion is backed by practically all papers found [3, 32, 42]. Figure 3.1 shows the test images.

Being the fastest method comes at a cost, FAST is not the most robust option for feature detection. FAST is more sensitive to noise than other methods and finds the same features multiple times. A prime example of this is Table 3.1, using the mandrill image, which produces a vast amount of detections. But on average, because FAST finds more features than other methods it regains some of its robustness.

Conclusively, FAST is by far the best solution for the application, because of its incredibly short computation time. It outclasses all other methods on this front. Over time multiple branches of FAST have been made, all with there respective pros and cons. In addition, FAST requires some filtering, both of these issues are explained in more detail in the next section.

Figure 3.1: Test images from source [3]. From left to right: Mandrill, Barbara, Lena, Peppers. The test images were of size 512×512 *pixels.*

Table 3.1: Mandrill, source [3]

Method	Computation time [ms]	Detected features [-]	Time per feature [ms]
FAST	70	16947	0.004
SURF	2100	2245	0.935
STAR	600	263	2.281
MSER	1000	358	2.793
GFTT	400	1000	0.400

Table 3.2: Peppers, source [3]

Method	Computation time [ms]	Detected features [-]	Time per feature [ms]
FAST	100	2427	0.041
SURF	3400	725	4.690
STAR	600	136	4.412
MSER	900	208	4.327
GFTT	400	108	3.704

Table 3.3: Barbara, source [3]

Method	Computation time [ms]	Detected features [-]	Time per feature [ms]
FAST	70	6995	0.010
SURF	1850	985	1.878
STAR	600	131	4.580
MSER	900	268	3.358
GFTT	300	964	0.311

Table 3.4: Lena, source [3]

Method	Computation time [ms]	Detected features [-]	Time per feature [ms]
FAST	40	4743	0.008
SURF	1620	687	2.358
STAR	600	91	6.593
MSER	800	110	7.272
GFTT	320	308	1.039

Table 3.5: Average of the four images, source [3]

Method	Detected features [-]	Time per feature [ms]	Tracking error [pixels]
FAST	7778	0.016	2.7
SURF	1160.5	2.465	2.6
STAR	155.25	4.438	0.3
MSER	595	4.445	5.5
GFTT	355	1.364	4.0

3.2 FAST corner detection

In this section the FAST corner detector is explained. First the basics of FAST are described. Thereafter, one of the different methods of FAST is chosen for the VSLAM algorithm and thereafter explained in detail.

FAST corner detection is the best suitable method because of its short computation times. It has such short computation times because of its simple design. In short, FAST detects corners by comparing pixels with their surroundings. The center pixel and the surrounding pixel that are tested are best visualised in Figure 3.2. FAST tests whether the surrounding pixels are similar, darker or lighter. If a set of subsequent surrounding pixels is darker or brighter than the center pixel, this center pixel is considered a corner. FAST does this for all pixels $i \times j$ in an image.

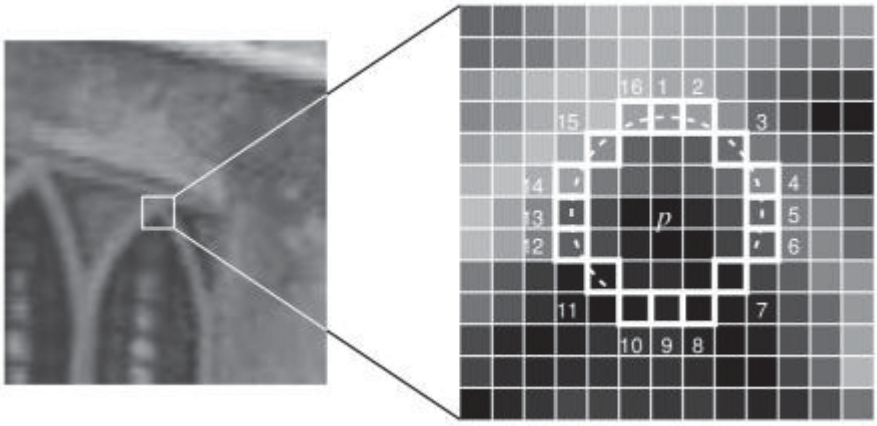


Figure 3.2: Example of a pixel test. Source [42]

A surrounding pixel p_n is tested to be either similar, darker or brighter than the center pixel $p_{i,j}$. This is tested with a threshold t. On average t is set to 35 on a scale of 0–255, this seems to give the best result. Whether the surrounding pixels are similar, brighter or darker for a specific $p_{i,j}$ is given by ${}^{i,j}S_n$

$$ {}^{i,j}S_n = \begin{cases} \text{if } p_n \text{ is brighter:} & b, \;\; p_{i,j} + t < p_n \\ \text{if } p_n \text{ is darker:} & d, \;\; p_{i,j} - t > p_n \\ \text{if } p_n \text{ is similar:} & r, \;\; p_{i,j} - t < p_n < p_{i,j} + t \end{cases} \tag{3.1} $$

If all surrounding pixels are tested and s *subsequent* pixels p_n have produced the same ${}^{i,j}S_n$, then $p_{i,j}$ is a corner. For example, $p_{i,j}$ is a corner if pixels p_n 1 through s are all darker/brighter. The number s is fixed in advance and usually ranges between 16 and 9. For clarification, $s = 12$ means that three-quarters of the circle must be darker/brighter and $s = 9$ means that half of the circle must be darker/brighter. If s is smaller than 9 FAST becomes an edge detector. Each of these numbers for s can be considered to be a different type of FAST. These types among others are compared next.

3.2.1 Different methods for FAST corner detection

In short, there are different types of FAST. FAST with different numbers s are considered different types, these are named FASTs (so FAST9, FAST10 and so forth). In addition there is *FAST pyramid* [3] and *FAST-ER* [42]. First a small description is given of these last 2 methods.

1. FAST pyramid [42] excludes itself from the rest by testing FAST at different scales, this increases the matching rate for *data association* later on.
2. FAST-ER [42] builds a search tree [41] when testing the surrounding pixels p_n. This makes the testing more efficient.

Rosten et al. [42] tested most of the different FAST methods. They tested FAST-ER, and FASTs for speed and repeatability. Repeatability implies that the same corner is found under different circumstances, for instance when a blur effect is applied to the test image. Rosten tested on repeatability for $s = [16\ 15\ ...\ 9]$ and computation times for $s = 12$ and $s = 9$. Their conclusion on repeatability is that the lower s, the better the results. The better repeatability is a result of perspective, if one looks a square from an angle, the angles of its corners become sharper on one side and blunt on the other. Thus with a wider angle the blunted angles are still detected with a lower s.

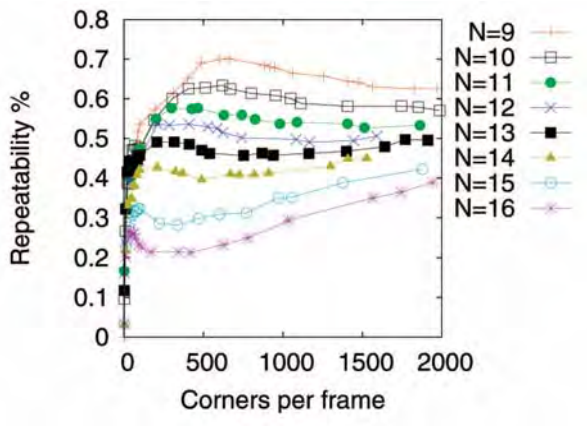


Figure 3.3: A direct copy of the repeatability test results by Rosten et al. [42]. The repeatability is tested for a Gaussian noise with $\sigma \approx 12$; the repeatability is given as a ratio, so $0.5 = 50\%$. For the test they used the "Oxford" data set. This data set is often used for computer vision to compare different methods. The data set is available on the internet: `http://www.robots.ox.ac.uk/~vgg/data/data-aff.html`.

According to Rosten et al. [42] the fastest methods are FAST9 and closely followed by FAST12. These two versions distinguish themselves from the rest by making an initial test. Without the initial test FAST tests a minimum of s surrounding pixels, this is a large sum of computations. With the initial test only 4 pixels are tested, and almost all pixels are then discarded as possible corners. Only the pixels that pass the initial test are tested with the remaining surrounding pixels. This reduces the amount of computations drastically. As can be seen in Table 3.6, the amount of pixels per second is doubled. Table 3.6 shows the result of FAST detectors on 2 sets of video images. The training set was of size 992×668 pixels and the test set of size 352×288. They used a 3.0GHz Pentium 4D processor. In the table there is an extra method *Original FAST*. This method is the original algorithm. The original method does not include the initial test.

Table 3.6: Computation time results of Rosten et al. [42]. This result includes non-maximimum suppression (see Chapter 3.2.2 for more information on this suppression).

Method	Training set Pixel rate [Mpix/s]	Test set Pixel rate [Mpix/s]
FAST9	188	179
FAST12	158	154
Original FAST ($s = 12$)	79.0	82.2
FAST-ER	75.4	67.5

The initial test can be explained as follows. Pixel $p_{i,j}$ is considered a corner if three-quarters (in case of $s = 12$) of the surrounding circle is brighter or darker. Meaning if four equally divided pixels (for instance pixel number 1, 5, 9 and 12, see Figure 3.2) are tested, three must all be either darker or brighter. By testing only these four pixels almost all pixels $p_{i,j}$ can be discarded as potential corners after this test. See Figure 3.4, where two cases are shown: a positive and negative test result. If during this initial test 2 pixels p_n are found similar, then $p_{i,j}$ also cannot be a corner anymore.

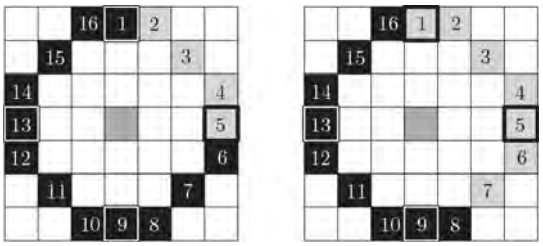


Figure 3.4: Example of Initial test for FAST12. The pixels tested for the initial test are numbers 1, 5, 9 and 13 (though any set of four opposite pixels does suffice). The left side shows a positive ttest result: there are three darker pixels. The right side shows a negative test result: there are only two darker and two brighter pixels. Therefore, it is not possible for the left situation to be a corner for FAST12

3.2.2 Non-maximal suppression

FAST detection detects a corner multiple times, because if a pixel $p_{i,j}$ is a corner the adjoining pixels are likely to be corners as well. These clusters of adjacent positive corners need to be reduced to single pixels representing one corner. An example of these clusters of the adjacent positive corners is visualised in Figure 3.5.

Figure 3.5: Example of clusters of positive corners (green pluses) using FAST

This problem is commonly solved with *non-maximal suppression* [42]. This suppression implies that the 'strongest' $p_{i,j}$ is chosen as a representative of the adjacent positive corners. The strength V of a corner is equal to the minimal threshold t needed for it to be a corner. By comparing the adjacent corner's strength V, the strongest $p_{i,j}$ is chosen to represent the cluster and the weaker pixels $p_{i,j}$ are discarded. Using non-maximal suppression adds computations, however it reduces computations for *data association*.

The strength V is found by testing the corner again with FAST. The highest difference $\delta_p = p_{i,j} - p_n$ of the subsequent set of darker/brighter pixels equals the minimal threshold required. So the strength is computed by:

$$V_{i,j} = \min |p_{i,j} - p_n| \quad \text{with} \quad n \in [1, 2, \dots, 16]. \tag{3.2}$$

This can easily be implemented with the newly proposed FAST algorithm.

Summarizing the section, the FAST methods FAST9 followed by FAST12 are the fastest and best candidates for the VSLAM application because of their initial test. After FAST detection non-maximal suppression is used to remove clusters and uses the strongest pixel to represent the respective corner. The codes tested in Table 3.6 and the non-maximal suppression algorithms used by Rosten et al. are available and are tested on the drone in the next section. From the results it is clear that these algorithms are not fast enough for the drone's small CPU.

3.3 Improving FAST algorithm for application

FAST is the fastest and therefore in this case best method for the VSLAM algorithm, however the standard FAST implementations are not sufficiently fast for the drone's small CPU. First the standard FAST implementations are tested on the drone. FAST algorithms are available in the OpenCV libraries and as open source codes. Rosten has put the code he used available on the internet `https://www.edwardrosten.com/work/fast.html`. The available algorithms of interest are FAST9 to FAST12. These algorithms are placed on the AR.drone to test their speed.

Rosten's codes were tested with matlab simulink and the toolbox of Sanne Marx [35]. The toolbox provides the ability to make computations with the drone's camera and have the drone share data with an external device over a UDP connection. Two models were created in simulink. One model runs on the external device and receives data from the drone and can send commands. The other model is placed on de drone. This second model uses functions from Marx that read camera data, resize the image and uses clock reading to compute the time it takes to compute the FAST algorithms used in function blocks. The function blocks contain the FAST code of Rosten who provides matlab codes.

By creating a code via simulink an easy test environment is created. However, this method is not optimal for computation speeds. Nonetheless, it gives a good comparison of different FAST algorithms when it comes to computation times. To improve comparability the computation times are plotted against the number of detected corners. Thereby making the data irrelevant of camera input.

Figure 3.6 shows it is possible to run FAST on the drone, however the computation time is high. For a 30Hz camera input the FAST computations should be computed within $\frac{1}{30}$[sec]. But, the computation time is practically always larger than $\frac{1}{30}$[sec]. The computation load of the FAST algorithm is improved by rewriting the FAST codes into a more suitable one. The codes provided by Rosten are a string of *if/else* statements, containing every possible combination of darker/brighter corners. This results in a long code of approximately 2000 lines. The FAST algorithm is therefore changed to a more compact code.

In order to create a faster code, the amount of computations should be minimized. The amount of computations is minimized if the needed information is gathered and used in the most efficient way. First the initial test: the initial test tests whether three out of four pixels p_n are darker/brighter. This gives us the information that the corner is darker/brighter. This means that for the remaining pixels only has to be tested if they are darker/brighter. This potentially already halves the amount of tests.

The p_n only have to be tested if they are darker or brighter, depending on the outcome of the initial test. Conventionally, the pixels are tested in order $\vec{n} = [1, 2, ... , 16]$. However, the pixel p_n that gives the most information is (in case no other information is available) the intermediate to the previously tested pixels p_n. By testing pixels p_n in a selected order the efficiency of information gathering is enhanced, exploiting FAST's power to discard pixels $p_{i,j}$ as potential corners to the fullest. In a way the FAST-ER method also used this principle, however by adding computations to determine the next best test eventually led to more computations. By pre-selecting the order there are no additional computations needed. The order is chosen as:

$$\vec{n} = [1, 9, 5, 13, 3, 11, 7, 15, 2, 10, 6, 14, 4, 12, 8, 16]. \tag{3.3}$$

Note that each set of four subsequent numbers n represents a plus shape, much like the initial test. The most informative pixel set after the pixels 1, 9, 5 and 13 would be numbers 3, 11, 7 and 15. Thereafter, the last two sets numbered 2, 10, 6 and 14 and numbered 4, 12, 8 and 16 in arbitrary order.

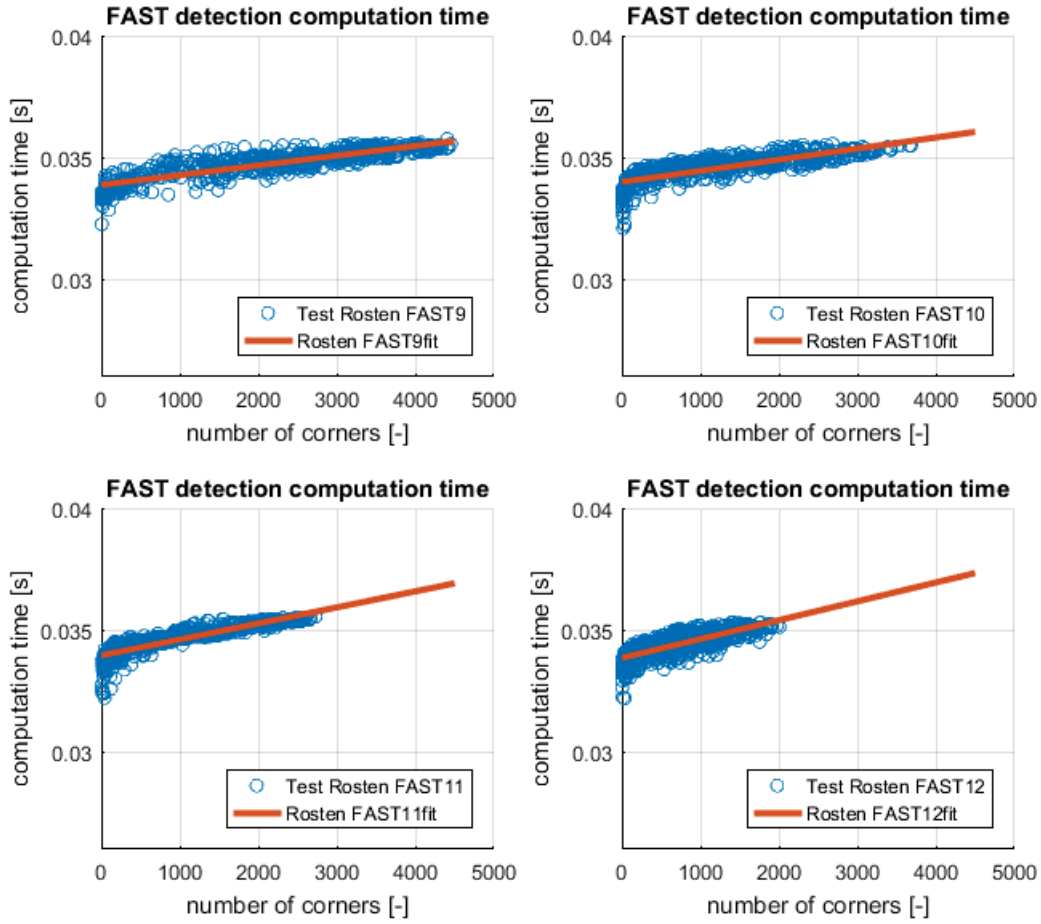


Figure 3.6: Computation time of Rosten's FAST9 to 12 versus the detected amount of corners. These graphs only contain the FAST algorithm. They do not contain the non-maximal suppression. The result hold for a 30Hz input of image size 320×240. *The code was made in Matlab simulink with the use of Marx's toolbox [35] and compiled to the drone.*

3.4 Implementing Proposed FAST in C++

The proposed FAST algorithm is created in both Matlab 2016 and in C++. The algorithm is similar for both and in this section the implementation of FAST in C++-code is described. The proposed FAST is written so that the computation time on the drone is minimized. This means that a minimal amount of computations is necessary.

The whole C++ code can be found in Appendix E, but parts of the code are discussed below. Starting with the initial test mentioned earlier. As a recap, the initial test tests four pixels and three out of four pixels must be darker or brighter than the center pixel for it to be a possible corner. After the initial test the FAST code's main test is explained in more detail. The FAST code consists of a set of statements and loops.

Equation (3.1) stated under which conditions a surrounding pixel p_n was darker, brighter or similar (${}^{i,j}S_n$). For the initial test only the amount of darker and brighter pixels are needed. The code for the initial test is given in Listing 3.1 and is part of the code *ProposedFAST.cpp* found in the Appendix E. The initial test is a **for** loop that is run four times for different p_n. In the **for** loop the difference between the center pixel $p_{i,j}$ is computed (line 120). Then in lines 121 to 126 the difference is compared with the threshold t to see if the pixel is darker or brighter.

Listing 3.1: Initial test

```
119 for (n = 4; n < 8; n++) {
120     delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
121     if (delta_p_test > threshold) {
122         c_lighter++;
123     }
124     else if (delta_p_test < -threshold) {
125         c_darker++;
126     }
127 }
```

If the initial test does not find three brighter or darker pixels it can continue with the next center pixel $p_{i,j}$. However if there are three of the same, then $p_{i.j}$ must be fully tested. Consider the case when three darker pixels were found. Then the surrounding pixels p_n must be tested to see if there are s subsequent pixels p_n that are darker, see Listing 3.2 line 158.

The main test of FAST after the initial test is as given in Listing 3.2. The test should go on until either a corner is found or until $p_{i,j}$ can no longer be a corner. Thus the test should be aborted as soon as possible to increase computation time. The test is conducted in a while loop, see line 161. This goes on until either all pixels are tested ($n = 16$) or when when the stop sign is given. The stop sign is given if the test finds that there is no corner.

Listing 3.2: FAST test for dark pixels

```
158 if (c_darker >= 3) {
159     stop = 0;
160     n = 0;
161     while ((stop == 0) && (n < 16)) {
162         delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
163         if (!(delta_p_test < -threshold)) {
164             if (firstN == 17) {
165                 firstN = 9 - pixelOrder[n];
166             }
167             pID = pixelOrder[n] + firstN;
168             if (pID > 16) {
169                 pID -= 16;
170             }
171             else if (pID < 1) {
172                 pID += 16;
173             }
174             if (pID < nmin) {
175                 nmin = pID;
176             }
177             else if (pID > nmax) {
178                 nmax = pID;
179             }
180             if (nSubPix < (nmax - nmin)) {
181                 stop = 1;
182             }
183         }
184         n++;
185     }
186 }
```

Inside the **while** loop the difference in brightness is again computed first (line 162) and compared to the threshold to see if the pixel is darker (line 163). If the pixel is darker the next pixel p_n is tested. On the other hand, if p_n is *not* darker then it must be within range of $16-s = 16-12 = 4$

steps away from the furtherest non-dark pixel. If it is further away than $p_{i,j}$ can no longer be a corner.

Checking if the newly tested non-dark p_n is 4 steps away from the previously tested non-dark p_n requires some extra procedures since $p_{n=1}$ and $p_{n=16}$ neighbour each other. If for instance $p_{n=15}$ was found non-dark and later $p_{n=1}$, then the computer must know that distance is equal to 2 and not $15 - 1 = 14$. This problem only occurs when the non-dark pixels are close to $n = 1$ or $n = 16$. The extra procedure to bypass this problem is to give the pixels a new n-like number, a new ID.

The new ID ranges from 1 to 16 and is distributed much like n is in figures 3.2 and 3.4. The only difference is that it starts with counting on a different position: a position that places the non-dark pixel away from $n = 1$ and $n = 16$. When the first non-dark p_n is found the new ID is initialised around that point. The first non-dark p_n has ID=9, the number furthest from 1 and 16. This initialisation is done in lines 164–166, the **if** statement in those lines only holds true the first time it is called in the **while** loop because firstN is initialised at 17.

After the initialisation the ID is assigned in line 167, for every non-dark pixel. The new ID should range from 1 and 16, this is assured in lines 168–173.

With the new IDs the maximum amount of steps must be computed to check if $p_{i,j}$ can still be a corner. It is not necessary to keep track of every non-dark pixel, only storing the lowest and highest ID is sufficient. The maximum number of steps is always the difference the highest and lowest number. This maximum number of steps is computed and tested in line 180. If the number of steps is within acceptable range, the test continues, otherwise the stop sign is given and the test is ended.

After the main test is finished it is known whether or not $p_{i,j}$ is a corner. After it repeats the whole process for all pixels $p_{i,j}$ that need to be tested, the FAST corner detection is finished. non-maximal suppression is excluded for the test here, the results of non-maximal suppression are stated later on in Section 3.6.
By making the proposed FAST algorithm in the above mentioned way, the following results are achieved when tested on the AR.drone: see Figure 3.7. Clearly, the results of the proposed FAST algorithms out performs Rosten's FAST in terms of speed. Both FAST algorithms result in exactly the same corners found. In other words, the output of both algorithms are equal for the same input.

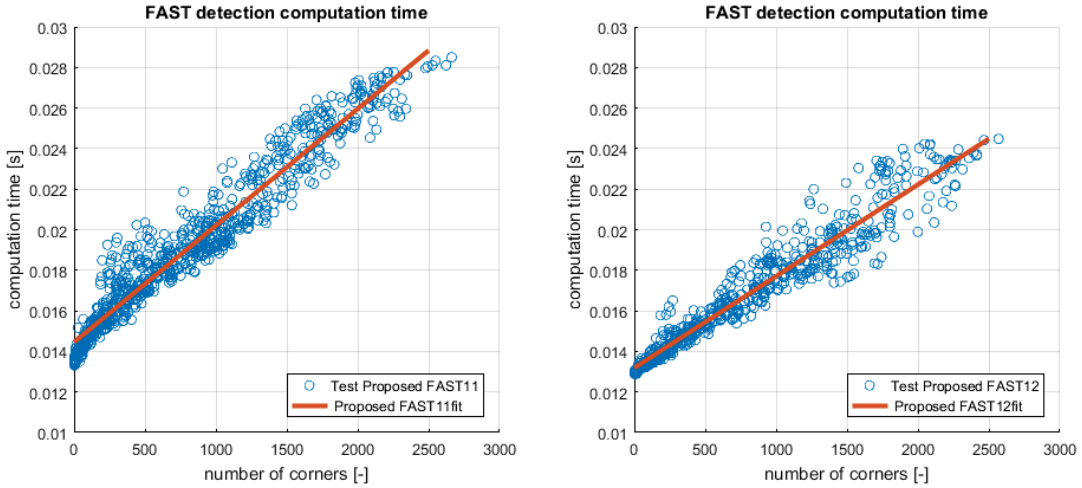


Figure 3.7: Computation time of the proposed FAST11 and 12 versus the detected amount of corners. These graphs only contain the FAST algorithm. They do not contain the non-maximal suppression. The results hold for a 30Hz input of image size 320×240*. The code was made in Matlab simulink with the use of Marx's toolbox [35] and compiled to the drone.*

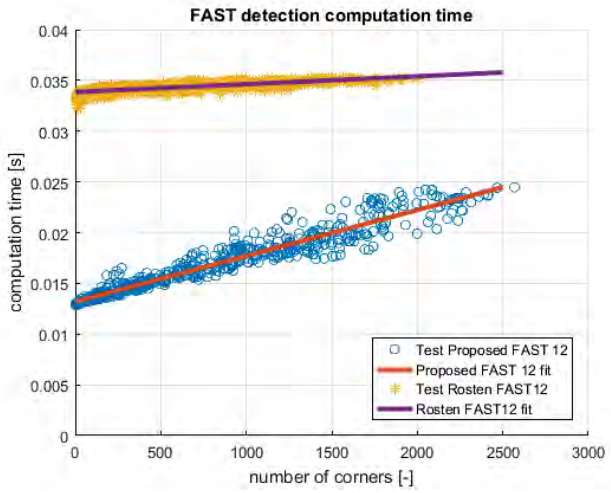


Figure 3.8: Computation time of the proposed FAST12 and Rosten's FAST12 versus the detected amount of corners. These graphs only contain the FAST algorithm. They do not contain the non-maximal suppression. The results hold for a 30Hz input of image size 320×240*. The code was made in Matlab simulink with the use of Marx's toolbox [35] and compiled to the drone.*

3.5 Minimizing computations by combining FAST and non-maximal suppression

Using the proposed FAST algorithm and non-maximal suppression seems like a possible solution. However, in this report a different way to prevent the clusters of adjacent positive corners is proposed. Eventually the method to be proposed increases the efficiency of the detector. Naturally, minimizing the computation time is done by reducing the amount of computations. The minimal amount of computations can be achieved by only computing the minimal amount of information. So, computing all positive $p_{i,j}$ in clusters, only to discard them afterwards adds unnecessary computations. The method proposed reduces the overall computation time by reducing the pixel tests.

By only partially testing the pixels of an image the strongest corners (the clusters) are most likely to be found and the amount of computations drop. For instance, by testing a random pixel, a corner with three pixels has a three times higher likelihood to be found than a corner with a single pixel. By choosing a smart order of testing pixels the likelihood of finding the clusters increases.

In order to *not* add any extra computation only the testing order is changed. The image is divided into blocks of 4×4 pixels. The pixels $p_{i,j}$ are tested one at a time and one in each block. The pixels are tested in a certain order, see Figure 3.10. Every time a new pixel is tested it is harder to find an area in the image where a cluster of pixels $p_{i,j}$ can fit. From the 8th test on there are only corners with single pixels left to be found.

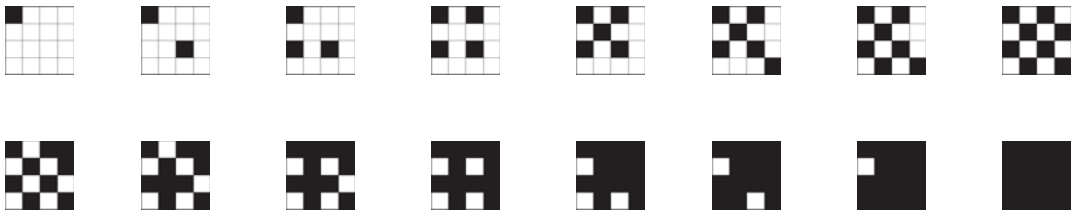


Figure 3.10: Pixel test order in 4×4 *grid*

The order of testing the pixels is looped. Every round a pixel per block is tested new corners are found. This may continue until every corner is found (and thus every pixel is tested). But it

can be limited when enough corners are found. If m is the amount corners needed for a stable position determination of the drone it can be limited when the number of found corners is larger than m.

When a cluster of corners is found, the center or strongest pixel of that cluster should again be found. This can be done by slightly changing the non-maximal suppression.
Normally the suppression checks the list of detected corners for neighbours and then tests those for their strength V. As mentioned, the strength is found by again performing the FAST algorithm to the pixel and surrounding pixels and thereby computing the minimal threshold.
Since the suppression already tests the surroundings with FAST, it is not necessary to check if the neighbour is already tested. Therefore, even if not all pixels of an image are tested before the non-maximal suppression, the suppression also tests the prior untested pixels. Thereby finding the center of clusters can still be found even if the center pixel was not tested for a corner during the FAST detection.

The proposed algorithm for FAST and non-maximal suppression is tested to verify how much of the total corners it finds. The results of this test are shown in Figure 3.11. As can be seen when only a fourth of the pixels are tested, already 45% of the total corners is found. With half of the pixels tested this becomes 75%. This number becomes even larger when the corners of different images are matched. The false positive corners are (almost) all discarded when the corners are matched. Since larger clusters have a better repeatability, the matching rate is better when relatively more clusters are found. This is further tested in Chapter 4.

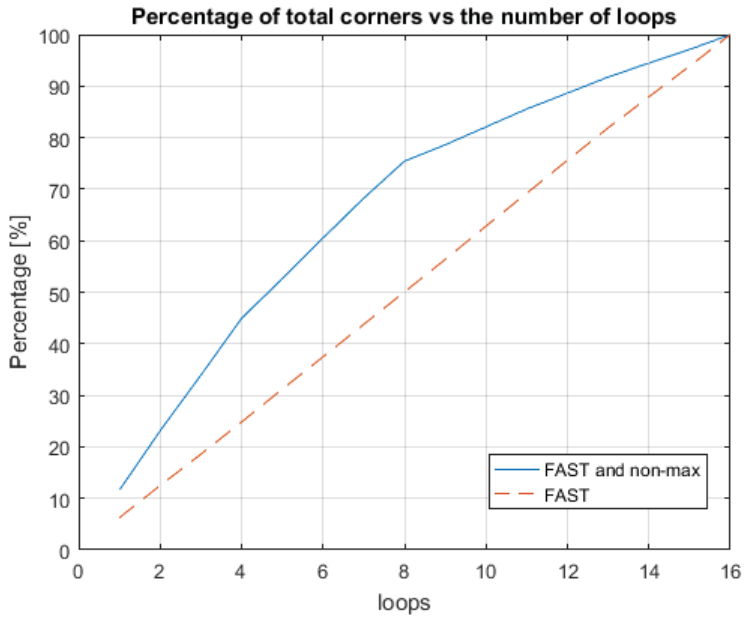


Figure 3.11: The percentage of detected corners when using the proposed algorithms. The loops indicate the number of pixels tested in the 4×4 *grid as was shown in Figure 3.10. All corners are considered detected when all pixels of the image are tested. The testing material consisted of a video image of size* 320×240 *pixels and 325 frames. The video images were subject to noise and motion blur.*

In Figure 3.12 the speed increase is shown. Much like expected this shows a straight line. The efficiency of the proposed algorithm is high because of the low computation times and increase in percentage of corners. The efficiency is better represented in Figure 3.13. In this figure the number of detected corners per second are shown. It would be most efficient to chose the number of loops to be 4. This way almost twice as many corners per second can be found. However, considering the mere 45% of detected corners, it might be better to increase this number to ensure more robustness of the whole VSLAM algorithm. This number is set after matching and mapping of the points, only then can be tested for speed and robustness.

Overall, the proposed algorithm for FAST and non-maximal suppression gives a high increase of computation speed. By rewriting the FAST algorithm the speed is increased with a factor two, and can be further increased with a factor 1 to 4, increasing the speed with an overall factor of 2-8 depending on the number of loops the algorithm ultimately runs.

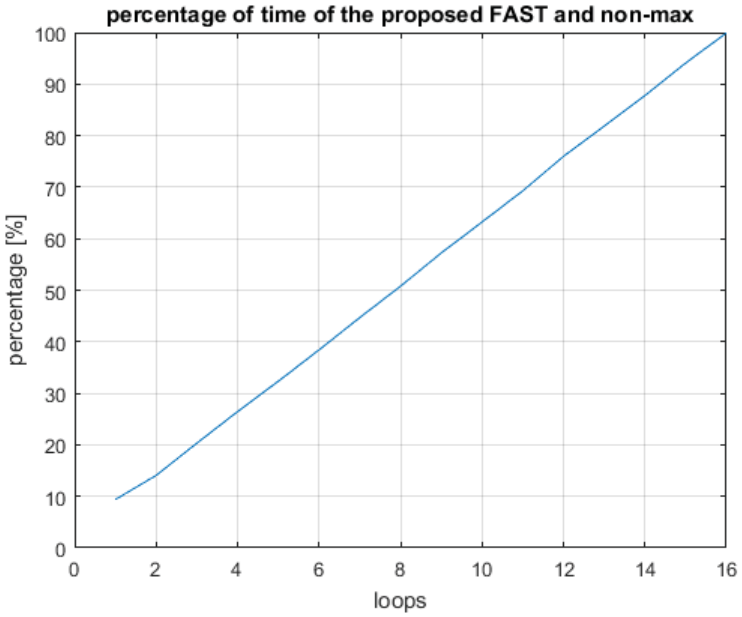


Figure 3.12: The percentage computation speed when using the proposed algorithms. The loops indicate the number of pixels tested in the 4×4 grid as was shown in Figure 3.10. The testing material consisted of a video image of size 320×240 pixels and 325 frames. The video images were subjected to noise and motion blur.

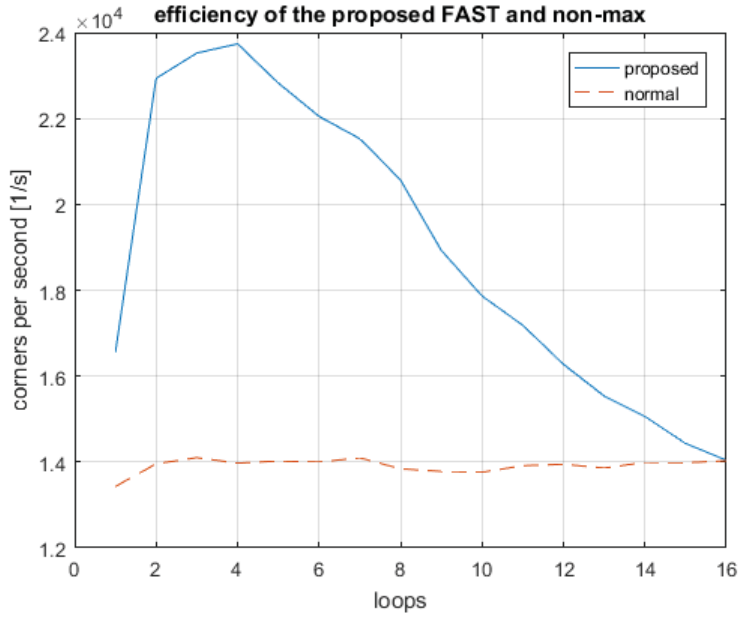


Figure 3.13: Number of corners detected per second of the proposed algorithms and normal algorithm. The loops indicate the number of pixels tested in the 4×4 grid as was shown in Figure 3.10. The testing material consisted of a video image of size 320×240 pixels and 325 frames. The video images were subject to noise and motion blur.

3.6 Implementing non-maximal suppression in C++

With non-maximal suppression the strength V of pixels is compared. Therefore, the algorithm for non-maximal suppression is build as follows, see listings 3.3 and 3.4.
The suppression is a local minimum search. Meaning that for each found corner with the proposed FAST algorithm it computes the local minimum, line 48. In order to find the local minimum the found corner pixel and its direct surroundings are tested (line 52 until 59) and the strongest pixel of this whole set of pixels is one step closer to the local minimum. Then the whole process is repeated for that pixel. This repetition continues until the pixels surrounding are all weaker, see line 50.

Listing 3.3: Nonmaximal suppression base code part

```
48 for (n = 0 ; n < numberOfCandidates ; n++) {
49     centerPixel = listOfCorners[n];
50     while (!(centerPixelOld == centerPixel))
51         savedValue = 255;
52         for (nn = 0 ; nn < 9 ; nn++){
53             ii = centerPixel + pixel[nn];
54             value = pixelTest(imageG, ii);
55             if ( value < savedValue ) {
56                 savedValue = value;
57                 cornerii = ii;
58             }
59         }
60         centerPixelOld = centerPixel;
61         centerPixel = cornerii;
62     }
63     listOfCorners[n] = cornerii;
64 }
65
66 // order vector
67 MergeSort(listOfCorners, rsize);
68
69 // remove doubles and compress
70 int nCorners = removeDoubles(listOfCorners, numberOfCandidates, rsize);
```

The surroundings of a pixel are all the adjacent pixels, eight in total. The strength of these adjacent pixels and the center pixel is compared and the strongest pixel is chosen for the next new center pixel. The strength of a pixel is computed similar to how corners are found with the FAST algorithm.

Strength V is equal to the minimal threshold t for which $p_{i,j}$ is still a corner according to the FAST algorithm. Therefore, V is equal to the lowest difference in brightness between p_n and $p_{i,j}$. But only for p_n that are brighter or darker relative to the type of corner, so only the difference from darker p_n in case of a dark corner.

The algorithm to compute V uses the FAST algorithm as a basis, see Listing 3.4. The input of this function is pixel $p_{i,j}$ and the strength of $p_{i,j}$ is the output. First the initial test is used to see if $p_{i,j}$ can be a corner, thereafter the main test starts. There is only a slight change in the code after testing pixel p_n and it is darker (in case the test involves a darker corner). The difference in brightness is saved if p_n is darker and the difference is lower than the previously saved value. Naturally the difference in brightness from the first dark p_n is saved unconditionally, in the code this is achieved by initializing the savedValue at a high number.

Listing 3.4: Nonmaximal suppression test for dark pixels

```
158 if (c_darker >= 3) {
159     stop = 0;
160     n = 0;
161     while ((stop == 0) && (n < 16)) {
162         delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
163         if (!(delta_p_test < -threshold)) {
164             if (firstN == 17) {
165                 firstN = 9 - pixelOrder[n];
166             }
167             pID = pixelOrder[n] + firstN;
168             if (pID > 16) {
169                 pID -= 16;
170             }
171             else if (pID < 1) {
172                 pID += 16;
173             }
174             if (pID < nmin) {
175                 nmin = pID;
176             }
177             else if (pID > nmax) {
178                 nmax = pID;
179             }
180             if (nSubPix < (nmax - nmin)) {
181                 stop = 1;
182             }
183         }
184         else if (savedValue > abs(delta_p_test)) {
185             savedValue = abs(delta_p_test);
186         }
187         n++;
188     }
189 }
```

The corner pixel found by FAST is overwritten with the strongest pixel found with the local minimum search. After this happened it is possible that multiple corner pixels led to the same local minimum. The pixels with the same coordinates are merged in the code. This merging is done by first sorting the list of corners in increasing value. This is done with a merge-sort function, see Listings 3.3 line 67. Merge-sort makes in total $O = n\log(n)$ computations, where n equals the number of elements in the vector it sorts. After merge-sort the doubles are all listed after each other. This order makes it easy to remove the doubles for the list of corners. For a small decrease in computations, the merge-sort and removal of doubles can be combined. However, the computation time of the two separate functions is insignificant compared to other functions. Therefore, the two functions are not combined here.

3.7 Concluding feature detectors

Features are detected from camera images to ultimately be used to compute landmarks. These features are detected with an altered FAST corner detection. Traditional FAST tests pixels $p_{i,j}$ by testing the surrounding pixels in a circular motion. By changing the order of the tests, the speed is greatly increased because pixel $p_{i,j}$ can be discarded faster as a potential corner. The order is changed so that the most informative of the surrounding pixels is tested next.

The test order of the surrounding pixels is not the only thing that has been adjusted in FAST. In addition, traditional FAST tests every pixel $p_{i,j}$ of an image. After FAST non-maximal sup-

pression is used to remove clusters except for the 'strongest' pixel in that cluster. Thus making computations that are for pixels that are discarded afterwards. The proposed FAST algorithm searches for these clusters by only testing a raster of pixels $p_{i,j}$. Thereafter, a proposed non-maximal suppression is used as a local minimum search to find the 'strongest' pixel of a cluster. From tests it is proven that if, for example only half of all pixels $p_{i,j}$ are tested, 75% of all features is found.

The proposed FAST corner detection combined with non-maximum suppression shows good results that can be combined with the overall VSLAM algorithm. The speed increase makes it more compatible for the small computer power of the AR.drone. The decrease in corners in itself lowers the robustness of the overall algorithm. However, those effects are considered to be small, since FAST collects a vast amount of positive corners. Additionally, the clusters are more likely to be found, are usually stronger corners, and are usually *not* false positives.

The detected features should be described, stored and compared as time progresses, thereby tracking the features as they pass over the image field of the drone. The description of features is done with a feature descriptor. Different feature descriptors are explored in the next chapter.

Chapter 4

Visual tracking: Feature descriptors

As is mentioned in the previous chapter, images taken by the camera of the drone are the main source of informational input for the VLSAM algorithm. Getting the information from these images is done with visual tracking.

The first step in VSLAM is gathering data from visual input. The algorithm needs landmarks to pinpoint its position with. The landmarks must be detected and recognised. Visual tracking can be split into two functions:

1. feature detection,
2. feature description.

Feature detection is described in previous chapter in detail. In summary, in images certain features should be picked that can be detected again in the next image. After finding these features in an image, the features should be described. By describing the features the algorithm can cross-link the detected features between images. This describing of features is done by a *Feature descriptor* and matching these features is commonly called *Data association*.

Matching descriptors is one of the most computationally intense methods. If newly detected features from a new camera image are matched with all stored features that would mean that there are many combinations possible. For example, if in a new camera image 200 features are detected and there is a map of 10,000 landmarks, then there are a small million matching tests needed on average to match all features. Needless to say is that an efficient matching algorithm is needed.

The fore mentioned functions: feature detection and feature description are separately handled in different chapters. In the previous chapter feature detection is explained and feature description is explained in this chapter. The chapter has the following structure: first a list of different methods is given and one of them is chosen for the VSLAM algorithm. Thereafter, the chosen method is described in more detail. Lastly the method is implemented on the drone and tested.

4.1 Texture vs. estimation vs. binary descriptors

Much like feature detection, there are numerous methods available for feature descriptors. In this report a few categories are considered

1. texture based recognition,
2. position estimation (nearest neighbour search),
3. binary descriptors.

The best solution for the VSLAM application mainly depends on the computation speed. From records [28] it is clear that *texture based recognition* is too slow; barely capable of real-time applications. In addition to its slow nature, these methods require a data-base of images, one per landmark. This data-base is too large for the drone's small memory.

That leaves only the *position estimation* and *binary descriptors*. First these two methods are shortly explained. Starting with the nearest neighbour search and thereafter the binary descriptors.

4.1.1 Position estimation: nearest neighbour search

A nearest neighbour search (NNS for short) works by estimating which of the detected corners compare based on position. This NNS is generally faster than visual tracking and uses less memory, for it only needs to store landmark positions and nothing else.

In order for NNS to work, a prediction of the drone's state must be available. For this predicted state the odometry from the accelerometers can be used. With this predicted state the position of the measured landmarks $\vec{l_i}$ can be compared with the predicted landmark $\hat{l}_j$ location. The distance between the two $\vec{l}_{i,j}$ can then be used to derive the probability $f_{i,j}$ of the compared landmarks being one and the same.

$$f_{i,j} = \frac{1}{(2\pi)^{n/2}\sqrt{|S_j|}} \exp\Big(-\frac{1}{2}\vec{l}_{i,j}^{T} S_j^{-1} \vec{l}_{i,j} \Big), \tag{4.1}$$

with S_j the covariance matrix corresponding to the predicted position v_j.
In order to find the landmark most likely to be the same, the probability of all possible landmark combinations must be computed. These are too many computations for the drone. But with smart use of the predicted state and excluding some landmarks, the amount of computations can be reduced drastically. Methods doing this are, amongst others, Sequential Compatibility Nearest Neighbour (SCNN), Joint Maximum Likelihood (JML) and Joint Compatibility (JC) [15].

Overall a NNS approach seems very applicable to the VSLAM application, however NNS is not robust enough. While the drone is flying not all images from the camera are usable. When the image is blurred too much, there are no detected features. If this continues for too long the NNS approach is no longer able to match points simply because the accumulated error is too big. In order to prevent such a scenario, a descriptor is needed that visually describes a landmark.

4.1.2 Binary descriptors

Binary descriptors are frequently used in real-time visual tracking. They are fast and fairly robust. Apart from the fast extraction of the binary descriptors, the matching speed of these descriptors is also on a whole other level. Overall binary descriptors are a prime candidate for the VSLAM algorithm.

Binary descriptors work as follows. The descriptor stores a selected few pixels/areas around a corner from one image, see Figure 4.1 for an example. This field of points describes the surroundings of the center pixel. A binary descriptor describes those areas by stating if they are brighter or darker. Each of these areas has one bit to store data. The data stored in this one bit is whether this area is brighter or darker than the center area. This way the surroundings are described only by a small set of bits (bytes). By storing the data this way, a very compact easily compared descriptor is build.

In order to check if two features of two different images are the same, the described surrounding of one feature can be compared with the surroundings of the other feature, see Figure 4.2. This comparing is a bit-wise comparison. As mentioned the descriptor is in the form of a set of bits. The *Hamming distance* between two of those strings represents the difference between the two.

The Hamming distance can best be explained with the use of an example. If there are two numbers: 10110111 and 11010011, then the Hamming distance is 3, because there are three non-matching numbers. In a binary case, the Hamming distance is a set of bit-wise XOR operations.

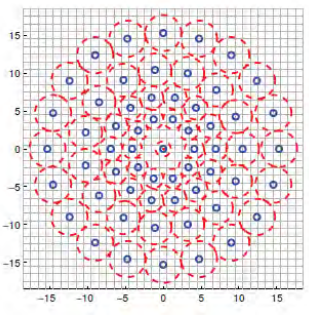


Figure 4.1: Pattern of the test areas for BRISK binary descriptors. Source [27]

That is why this works very fast on computers. A binary descriptor uses this as follows. Every comparison of the surrounding areas is one of those XOR operations. Then the Hamming distance equals the number of unmatched areas. If this number is lower than a set threshold then the two features are a match.

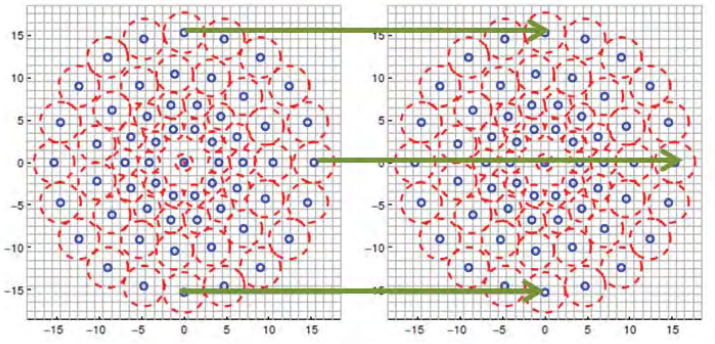


Figure 4.2: Compared areas from two corners.

Not all descriptor methods produce the same descriptor if an image is rotated. A binary descriptor can be almost rotationally invariant. Usually, if an image rotates, the descriptor changes. Some binary descriptors can compensate for the change. The surrounding field of a binary descriptor is a repeating circular pattern. As a result if the image rotates, the pattern is the same every X degrees. Meaning that the resulting descriptor is the same as the unrotated one, only shifted. The comparison of two features then becomes almost rotationally invariant by simply comparing shifted surrounding areas. Figure 4.3 shows the case when surrounding areas are tested when turned.

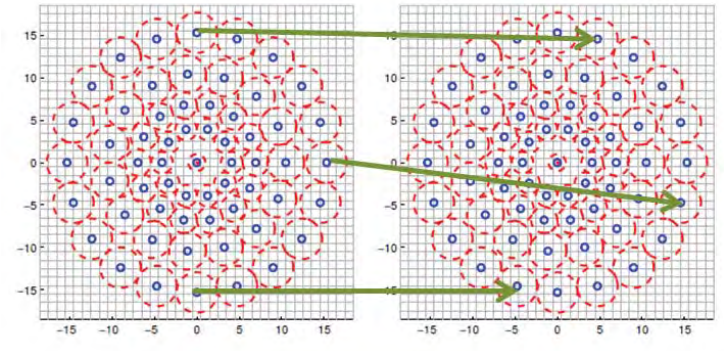


Figure 4.3: Compared areas from two corners when turned X degrees or in other words turned one segment.

Most binary descriptors are almost rotational invariant, to our knowledge only the BRIEF descriptor [14] is not, because it selects the surrounding areas in a random location and is thus not circular. The BRIEF descriptor was the first binary descriptor to be introduced in visual tracking. Because of its fast computation speed (at that time), many others followed. The most promising versions of binary descriptors are described and compared below.

4.1.3 Comparing binary descriptors

As mentioned there are multiple binary descriptor. The ones that are considered in this report are: BRISK (Binary Robust Invariant Scalable Keypoint), FREAK (Fast Retina Keypoint) and ORB (Orientated FAST and Rotated BRIEF). These are the three most commonly used descriptors. First each of these descriptors is shortly described. Thereafter, the computation speed of them is compared, combining data from different sources. Additionally the robustness of these descriptors is compared with the work other sources. The different binary descriptors can shortly be compared as follows:

1. **Binary Robust Invariant Scalable Keypoint: BRISK:**
   The BRISK descriptor is created by S. Leutenegger et al. [27]. The field of surrounding areas can be seen in Figure 4.1. Because it has a circular shape it is almost rotationally invariant. In the mentioned figure the blue dots present the location of the area. These areas are computed with a Gaussian blur. This lowers the sensitivity to noise and really describes the area if the gray value of the location is picked. The red dashed circles surrounding the blue dots represent the standard deviation σ value of the Gaussian. Every area has its own Gaussian blur with linearly growing σ as the areas stray further from the center. The figure only shows a field of 60 areas. This number may however be changed: more areas make it more robust, less areas naturally lessens the robustness.

2. **Fast Retina Keypoint: FREAK:**
   The creators of FREAK, A. Alahi et al. [8], based their field on the human eye. As a result the distribution of the areas is different from the other descriptors, see Figure 4.4.. The outer areas are larger and further away, and the inner ones are exponentially smaller and present in larger numbers. Much like BRISK it uses a Gaussian blur for the smoothing of the local areas. The only difference is the exponential growth of the areas when it is further apart from the center. This exponential growth places the areas more efficently, needing less areas and therefore a smaller descripor in bit size. The larger further apart areas are also less needed. As is stated in A. Alahi et al., after the first 16 bits of the binary descriptor have been compared, 90% of the combinations can already be discarded. The first bits correspond to the outer areas. The remaining 10% should be compared with more closely located areas.

3. **Orientated FAST and Rotated BRIEF: ORB:**
   The ORB descriptor is closely related to the BRIEF descriptor and differs from the other two. ORB stands for Oriented FAST and Rotated Brief, and was presented by E. Rublee et al. [43]. As the name implies it is based on using FAST as detector, they adjusted it so it would also give the orientation of the corner. It is possible to implement this in the proposed FAST explained in the previous chapter. The BRIEF descriptor is then changed to the orientation measured with the oriented FAST. Making the BRIEF descriptor rotational invariant.

These different descriptors all have their own pros and cons. These pros and cons are compared next and at the end is discussed which descriptor is used in the VSLAM algorithm.

First the computation speed of the different descriptors is given. Again, the computation speed is vital because of the small computational power on the drone. Therefore, the descriptor must mainly be fast and efficient. The speed of the descriptors is listed in Table 4.1. In this table

Figure 4.4: The area destribution of the FREAK descriptor. Source [8].

the computation speed per feature is given. This makes them comparable, because the number of features is equal if they are implemented in the VSLAM algorithm. After all, the features are detected by the proposed FAST. For the table three sources were picked that gave the best results and provide the information needed for the time per feature comparison. Apart from the binary descriptors also SIFT and SURF are given, providing more data for a better comparison.

Table 4.1: Computation time comparison for descriptors based on [8, 17, 27]. The time is given in time per feature.

Method	[ms] according to [8]	[ms] according to [27]	[ms] according to [17]
SIFT	2.5	5.29	0.1746
SURF	1.4	0.359	-
BRISK	0.031	0.037	0.0096
FREAK	0.018	-	0.0091
ORB	-	-	0.0146

From Table 4.1 it is clear that, from the three binary descriptors (BRISK, FREAK and ORB), Freak is the fastest closely followed by BRISK. The slowest binary feature is ORB. The results can be explained. ORB is fairly similar to BRIEF and is not focussed on decreasing the computation time. However, BRISK and FREAK were increased in speed, compared to BRIEF. While FREAK has less areas to compare, it is only slightly faster than BRISK. The reason for this is that FREAK needs to compute the Gaussian of larger outer areas, larger areas mean quadratically larger Gaussian computations. Although FREAK is only slightly faster, the lesser areas mean that its descriptor is smaller in bytes. This means that when a map is created later on, in which the descriptors also need to be present, smaller descriptors can save much ROM and RAM memory.

The robustness of the descriptors are compared next. The comparison of the binary descriptors is done with the work of an anonymous, neutral source. The results of this match up with practically every other source that was used during the project. This neutral source is chosen because it reduces the chances of biased results. The source used the OpenCV libraries to compare the robustness and shared results online. All of its results can be found on `https://docs.google.com/spreadsheets/d/1gYJsy2ROtqvIVvOKretfxQG_OOsaiFvb7uFRDu5P8hw/edit#gid=16` [2]. Some of its result are also shown here, in figures 4.5, 4.6 and 4.7 and Table 4.2. Note that the graphs also include some other methods that are not included in this report, but for *SURF BF* (SURF Brute Force matcher) see [11] and for *SURF FLANN* see [34].

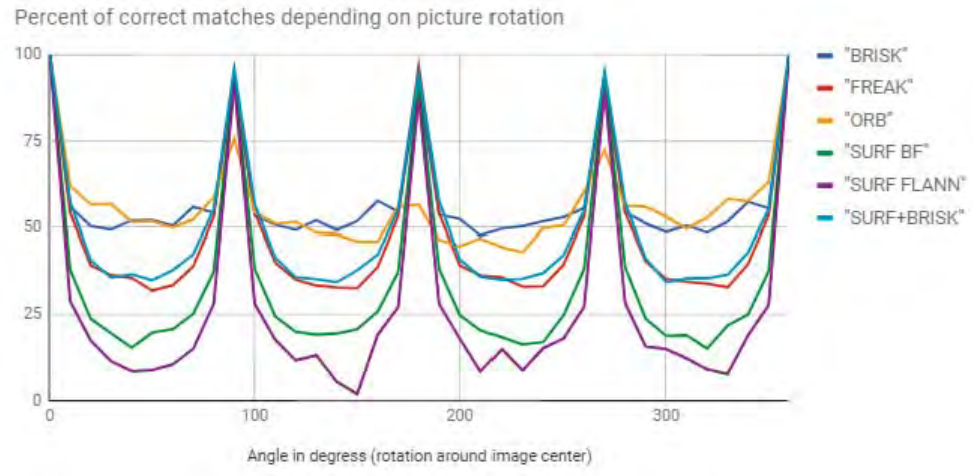


Figure 4.5: The axes of the graph represent the correct matches in percentages on the y-axis versus an rotating image in degrees on the x-axis. Source [2]

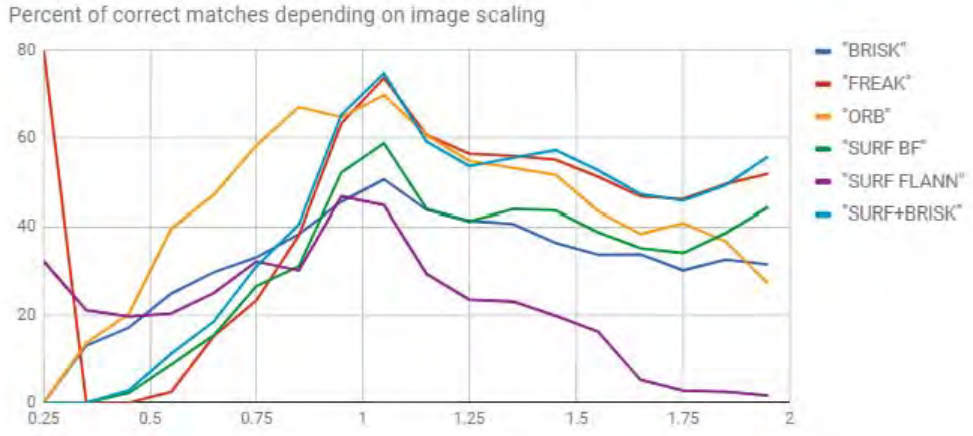


Figure 4.6: The axes of the graph represent the correct matches in percentages on the y-axis versus an image scale ratio on the x-axis. Source [2]

Table 4.2: Computation time from descriptor and detector given overall and per feature. Source [2]

Detector/Descriptor	Average time per frame [ms]	Average time per feature [ms]
SURF/BRISK	434.1	0.21
SURF/FREAK	290.6	0.19
SURF/SURF	842.4	0.34

From figures 4.5, 4.6 and 4.7 and Table 4.2 can be concluded that ORB is the most robust of the tree descriptors. It shows the best results in both blur and scale, and reaches second place in rotation robustness. Thereafter, FREAK is the most robust. FREAK beats BRISK in blur and scale robustness. Lastly is BRISK who only wins from FREAK in rotation robustness. Thus from robust to less robust is: ORB, FREAK and BRISK.

Liu et al. used FAST and Freak to create an algorithm approximately 15% faster than ORB according to their computations. Moreover, the computation time per landmark is almost 8 times faster. In theory it should be possible to filter the results from FAST to reduce the amount of corners by taking out noise and clustered corners, doing so could greatly reduce computation times. Based on this data FAST and FREAK combined is indeed faster, but more interesting is that their work showed that FAST and FREAK combined (two of the fastest methods for detection and

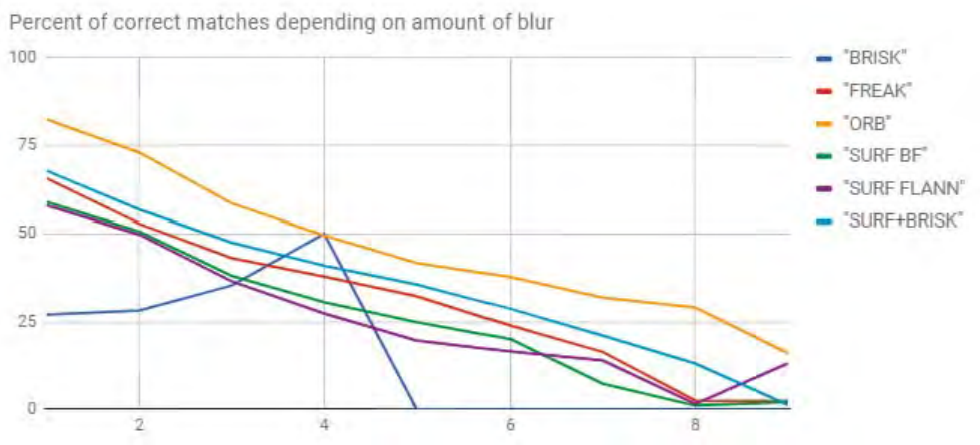


Figure 4.7: The axes of the graph represent the correct matches in percentages on the y-axis versus an increasing Gaussian blur ratio σ on the x-axis. Source [2]

description) is still robust and accurate. Their tests show that the matching rate of their algorithm is higher than those of SURF, SIFT and ORB. Although the results presented by the makers is possibly biased, the results are promising for using FAST and FREAK on the AR.drone.

Conclusively, FREAK is the best descriptor for the VSLAM algorithm. It is the fastest of the three compared binary descriptors and is second in robustness. ORB would be a good candidate as well, however the slow computation time might be a problem for the small CPU onboard the AR.drone. BRISK is comparable to FREAK, however it loses in most scenarios from FREAK, it is slightly slower and less robust in most cases. For these reasons the FREAK descriptor is chosen.

4.1.4 Matching binary descriptors

Features are matched by comparing their respective descriptors. Comparing descriptors of binary nature is very efficient, because it uses the hamming distance rather than the euclidean distance. The hamming distance is derived by comparing two bits with an XOR operator and a bit count thereafter, see previous section. Although binary features can be matched fast, if matched using a brute-force approach, comparing large quantities of features is exhaustive.

Most smart matching algorithms are vector based and do not give any real advantage for binary descriptors. Marius Maju and David Lowe [38] proposed an algorithm that is proven to work well with large sets of features. This method is also used by Matlab for matching binary features. For now this matcher is assumed to be the best option for the VSLAM algorithm, but more detailed research could potentially conclude in a different solution.

To present the performance of the descriptors and matching algorithm the following example is included. Two pictures were taken of the same wall, using the proposed FAST detector, FREAK descriptors and matcher. The result is given in Figure 4.8. In this figure the two pictures are laid over each other and the matches are visible as the yellow lines. In the figure some mismatches are visible, mainly located on the bottom of the images. Overall good matches are found and the mismatches must be removed, for they disrupt the map and localisation of the drone.

Figure 4.8: Matches using FREAK descriptors and the fast matching algorithm of Marius Maju and David Lowe [38]. There are a few clear mismatch visible, located to the bottom.

4.2 Implementing FAST, non-maximal suppression and FREAK in VSLAM algorithm

The whole algorithm of visual tracking is rather straight forward. The algorithm is schematically given in Algorithm 1, with the initialisations needed. The FAST algorithm has 2 variables that can be set to a user-defined value. The threshold of the FAST algorithm and the number of loops (see previous chapter). The dimensions of the image are set, taken from an test image. With those values the algorithm itself can start. First an image is taken from the camera. Therafter in order: FAST, non-maximal suppression and FREAK. At last the matcher can be used to match the computed FREAK descriptor with any other set of FREAK descriptors.

In the code FREAK descriptors are added with OpenCV's libraries. To verify that the combination of the proposed FAST detector and FREAK descriptors work, a test was conducted on four different video sets. These four videos are all made in the same environment but with different motions. One has pure rotational motion, the other three only translational motion: one sideways, one fort and back, and the last in a circular motion. The images in the video are of size 568×320 and the video sets consist of ≈ 300 images each.

The conducted tests are to check the influence of partially testing the image pixels for corners. As can be seen in Figure 4.9, the combination of the proposed FAST, non-maximal suppression and FREAK is approximately equal to the number of corners found. Overall the combination is curved, and at 8 loops there is a clear kink, especially in Figure 4.9b. This is the result of the dispersion of tested pixels. At 8 loops only single pixel corners can yet be found. Meaning that most of the corners are already found at this point.

4.3 Concluding feature descriptors

The FREAK descriptor is chosen to use for the VSLAM algorithm. This choice is based on research of various sources and is the fastest binary descriptor and has roughly the same robustness to blur, image rotation and scale as BRISK, which is just slightly slower than FREAK.

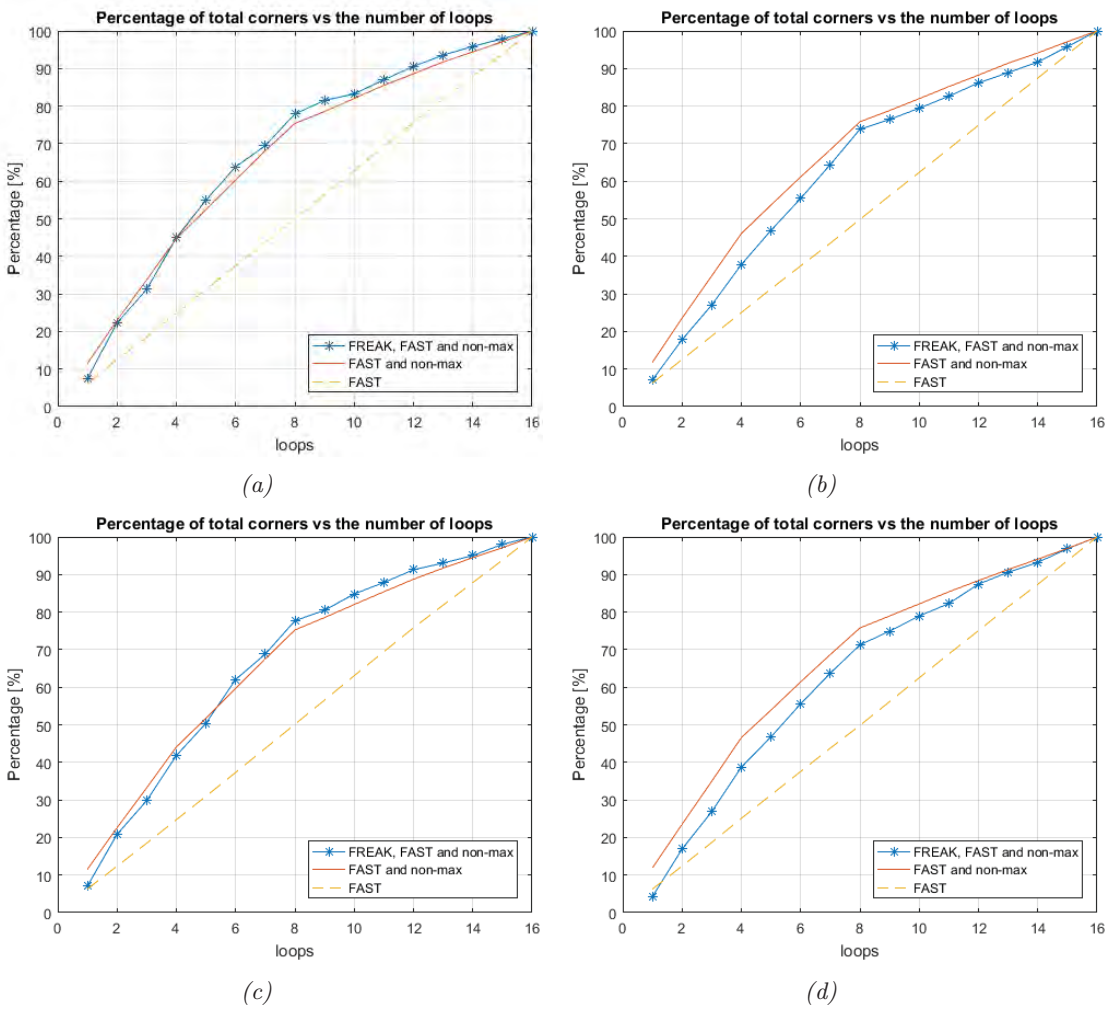


Figure 4.9: Precentage of matches versus the number of loops. The tests were conducted with four different videos. Graph (a) shows the result of the video with no rotation and where was moved in a circular motion. (b) was tested with a motion purely moving back and fort. (c) is the result of sideways motion of the camera. (d) is the result of a rotational motion. These graphs are extensions of Figure 3.12.

FREAK is a binary descriptor, it stores the descriptor in a way that it can be bit-wise compared to other FREAK descriptors. The Hamming distance then represents the error between the two compared descriptors. Overall, this way of matching is very efficient. Although the matching itself is efficient, if done in large enough quantities the computation time rises fast. An efficient matching strategy can improve computation times greatly. The matcher used was created by Maju and Lowe [38].

The chosen and altered detector and descriptor combined results in a fast algorithm that is very useful for the VSLAM algorithm. The visual part of VSLAM is generally the most computer intense one. Tests with the descriptors and binary matcher were conducted in Matlab. From simulation it was shown that FREAK descriptors combined with the matcher of Maju and Lowe give a large and accurate tracing method for the detected features. Still, some filtering to remove mismatches is needed. The removal of mismatches is discussed in the next chapter.

With features detected and described, a map of the environment can be made. The position of the features should be computed and stored so that the drone can use them to locate itself.

Algorithm 1: VSLAM algorithm: visual tracking

Input: set of descriptors for the matching function $f(descriptor1, descriptor2)$
Output: set detected features D with descriptors stored in $D_d \in D$ and pixel coordinates in $D_p \in D$, the state of the drone S at the time the image was taken and matches M
Parameters: the number of loops n and threshold t

```
1  Start camera;
2  I ← get test image from camera;
3  w ← width of I;
4  h ← height of I;
5  while run algorithm do
6  |   I ← get new image;
7  |   S ← get drone state from observer;
8  |   call C = ProposedFAST(I, n, t, w, h);
9  |   call C = NonMaximalSuppression(I, C, t, w, h);
10 |   call D = FREAK(I, C) ;                    // FREAK functions from OpenCV
11 |   M ← f(D_d, F_d) match new descriptors D with old stored descriptors F_d;
   |   /* The rest of the VSLAM algorithm is placed after these functions   */
12 end
```

Chapter 5

Map building

Map building is a key part of all (V)SLAM algorithms. The map stores the feature descriptors and their 3-D world coordinates. The descriptor combined with the 3-D position is called a landmark. The map is basically a long list of landmarks. The AR.drone's position is derived from this map and newly detected features must be matched with the descriptors listed in the map. Generally, a map can easily consist of tens of thousands of landmarks. This means that efficient matching is key. In addition, a map large in size conflicts with the ROM and RAM memory of the drone.

Computing the 3-D world coordinates of the features is an important part of mapping. The 3-D world position is derived as follows. Transforming the coordinates from 2-D pixel space to 3-D world space is possible. The pixel coordinates describe a direction, or to put it in a mathematical expression: a normalised vector $\vec{s}$. The world coordinate is then equal to the cross-section of the two vectors. With the specifications of the camera this vector can be computed. The vector can then be used to transform the position to the world coordinates.

The structure of this chapter is as follows: it is first shown what data is stored inside the map. Thereafter, the matching of descriptors and coordinate transformation are examined. The transformation of 2-D to 3-D world space are discussed, thereafter the matching algorithm. Lastly the implementation of map building is explained.

5.1 Map storage

The map contains data on features (items detected and described in 2-D) and landmarks (features that were detected multiple times and have a 3-D position). The map can thus be split in two. A map for features and a map for landmarks. The map containing the features stores the following list of data for each individual feature:

1. pixel coordinates of detected feature;
2. the feature's descriptor;
3. state of the drone containing both position and orientation;
4. accuracy of the drone state at that time.

Every time after features are detected and matched, the unmatched features are stored in this map. These stored features can then be compared and matched with newly detected features during each loop of the VSLAM algorithm, after which the matched features are removed and unmatched features are added again to this map. Because there are in general more unmatched features than matched features, this map keeps on growing. To keep this from happening the map size is capped. If the map grows too large the oldest features are removed.

The map of landmarks contains another list of data for the individual landmarks:

1. world coordinates of the landmark;

2. descriptor of the landmark;

3. the number of times the landmark is detected;

4. the accuracy of the landmark position.

Every loop of the algorithm, the newly detected features are matched with this map. The matches are used to compute the position of the drone and the information in the map is updated. Thereafter, the remaining new features are matched with the map of features. These matches are added to the map of landmarks. Generally, as this map of landmarks grows so does the accuracy of the position estimate of the drone; as there are more matched landmarks there is more data to compute the drone's position with. This map is currently not capped in size. This is sufficient for current test, however as longer flights get longer, it might be necessary to remove some of the map elements. In this case it is advised to carefully choose the elements that should be removed, based on age, number of times it was used and accuracy.

The elements of the map are constantly changed and updated. The equations needed to do this are found in the rest of this chapter. First and foremost computing the landmark position is explained.

5.2 Transforming pixel coordinates to 3-D world space

As mentioned, the direction of a detected feature can be described with a normalised vector $\vec{s}$. With a set of $\vec{s}$ pointing to the same feature from different camera positions, the 3-D position of that feature can be computed. This is done by finding the cross section of the set of vectors. All vectors should pass through the 3-D world position. First a situation is considered where the camera is ideal. A camera is considered ideal if there is no image distortion.

An ideal camera works as follows. The image plane I of the camera, see Figure 5.1, is located with its center perpendicular to the camera's *focal length* (f). Let $\vec{p}_{i,j} \in \mathbb{R}^{2\times 1}$ be an arbitrary pixel that detects some landmark in 3-D space. Then the line from the focal point to the landmark runs through $\vec{p}_{i,j}$. The earlier mentioned vector $\vec{s}$ points from the focal point to the point in 3-D space.

Deriving $\vec{s}$ from a 2-D pixel coordinate vector is only possible if it is defined as a normal vector: it represents direction without distance. The distance between the camera and an arbitrary point is simply not included in a 2-D image. The direction $\vec{s}$ can be derived with the pixel coordinates $p_{i,j}$ and camera specifications. The subscript of $p_{i,j}$ stand for the row i and column j. This row-column notation is not useful for computations. Therefore, the pixel coordinates $p_{i,j}$ are shifted so that the origin lies in the center. The resulting pixel coordinates ${}^{I}\vec{p}$ in the image-frame are denoted by ${}^{I}X$.

$$ {}^{I}\vec{p} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} p_j - \frac{w}{2} \\ p_i - \frac{h}{2} \end{bmatrix} \tag{5.1} $$

In the above equation w and h stand respectively for the width and height in pixels. The vector $\vec{s}$ can then be computed as follows:

$$ {}^{B}\vec{s} = \begin{bmatrix} f \\ -x \\ -y \end{bmatrix}_{\text{norm}} = \sqrt{f^2 + x^2 + y^2} \begin{bmatrix} f \\ -x \\ -y \end{bmatrix}, \tag{5.2} $$

with the subscript *norm* determining that the vector should be normalised and ${}^{B}X$ stating that the vector is given in the body fixed frame. The coordinate frame of the vector is relative to the camera, with x positive left, y represents up and z stands for the depth. With the equation $\vec{s}$ can be computed for an ideal camera with a known focal length.

The camera of the AR.drone is not ideal. In order to transform the drone camera to ideal some computations and estimations can be used. This transformation is commonly used for practically

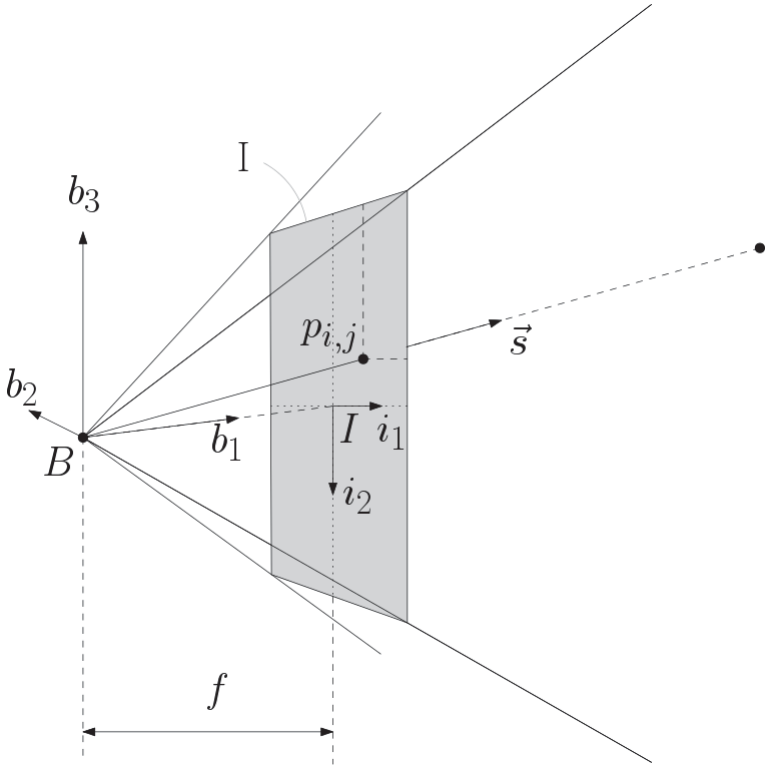


Figure 5.1: A schematic representation of an ideal camera.

all cameras. The main transformation lies in the distortion of the image as a result of the camera lens. To compensate for this lens distortion a lens correction algorithm is needed.

5.2.1 Lens correction

The camera of the drone is distorted by the lens. This lens distortion is compensated with a lens correction algorithm. Applying lens correction to an entire image is too computationally heavy for the drone, but applying the lens correction only to the selective few pixels that represent features is possible.

The most common model for lens correction is Brown's model. Brown's model uses a polynomial to estimate the distortion. It transforms an ideal model into a distorted one. This is the other way around as is needed but is often used. As a result, Browns model makes it easier to determine the distortion of an image. With the distortion known an image be undistorted.

Brown's model can only compensate for barrel and pincushion distortion. A barrel distortion is equivalent to the fish eye effect. The pincushion distortion is the opposite of the barrel distortion, having the corners dragged outwards. See Figure 5.2 for the distortion shapes. Over the years Brown's model has been adapted to cancel more types of distortions. The adaptations of Brown's model might be more accurate, however when calibrating the AR.drone's camera it was found that the camera only suffers from barrel distortion.

Brown models transforms the pixels coordinates with a polynomial fit. Let x_{ideal} and y_{ideal} stand for the pixel coordinates as they ideally should be, and x_{dis} and y_{dis} the distorted coordinates. The polynomial to estimate the x_{dis} and y_{dis} from x_{ideal} and y_{ideal} is given below. Usually a polynomial of order 2 or 3 is sufficient. Brown's polynomial then becomes:

$$x_{dis} = x_{ideal}(1 + k_1 r_{ideal} + k_2 r_{ideal}^2 + k_3 r_{ideal}^3), \tag{5.3}$$

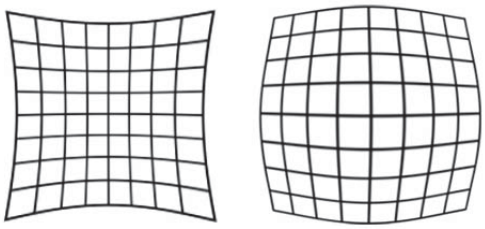


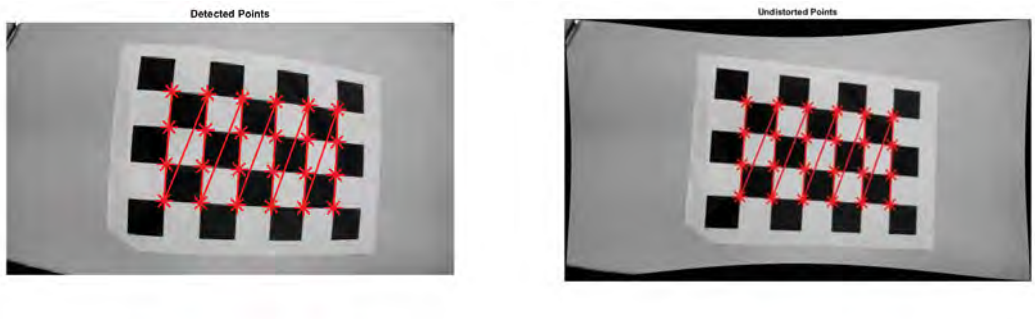
Figure 5.2: A schematic representation pincushion (left) and barrel (right) image distortion.

$$y_{dis} = y_{ideal}(1 + k_1 r_{ideal} + k_2 r_{ideal}^2 + k_3 r_{ideal}^3), \tag{5.4}$$

with r_{ideal} the squared distance of x_{ideal} and y_{ideal} from the center of the image normalised with the camera's focal length:

$$r_{ideal} = \left(\frac{x_{ideal}}{f}\right)^2 + \left(\frac{y_{ideal}}{f}\right)^2. \tag{5.5}$$

The factors k_1, k_2 and k_3 are found by fitting this polynomial onto straight lines in the distorted images. In essence this model fits a polynomial on a straight line bend by lens distortion. A checker board provides a good set of straight lines, see Figure 5.3a. After fitting the polynomial, the image can be undistorted. Only this function goes from an ideal image to a distorted one, and does not work vice versa. This problem is commonly solved by using an iterative approach, a result of an undistorted image is seen in Figure 5.3b. Although this approach is very accurate, the computation times are again too high.



(a) *(b)*

Figure 5.3: Calibrating camera using Matlab based on Brown's model

Drap and Lefèvre [16] derived an easy way to rewrite factor k_1, k_2 and k_3 into new factors (K_1, K_2, K_3). These factors approximate the inverse of Brown's model if placed in Brown's model. This results in an estimation of the inverse of (5.3) and (5.4). The new factors as a function of themselves $(K_1, K_2, K_3) = f(k_1,\ k_2,\ k_3)$. For the derivation of the function see [16], the results are:

$$K_1 = -k1, \tag{5.6}$$

$$K_2 = 3k_1^2 - k_2, \tag{5.7}$$

and

$$K_3 = -12k_1^3 + 8k_1 k_2 - k_3, \tag{5.8}$$

for the reversed estimation of Brown's polynomial

$$x_{ideal} = x_{dis}(1 + K_1 r_{dis} + K_2 r_{dis}^2 + K_3 r_{dis}^3), \tag{5.9}$$

$$y_{ideal} = y_{dis}(1 + K_1 r_{dis} + K_2 r_{dis}^2 + K_3 r_{dis}^3). \tag{5.10}$$

With the r transformed to

$$r_{dis} = \left(\frac{x_{dis}}{f}\right)^2 + \left(\frac{y_{dis}}{f}\right)^2. \tag{5.11}$$

If the iterative approach is used as a reference then the accuracy of the new factors is visible in Figure 5.4. The mean error is 2.5 pixels on an image of size 640×360 pixels. This error mainly accumulates around the edges, see Figure 5.5. Although the error grows exponentially as further from the center, the center itself is nicely estimated.

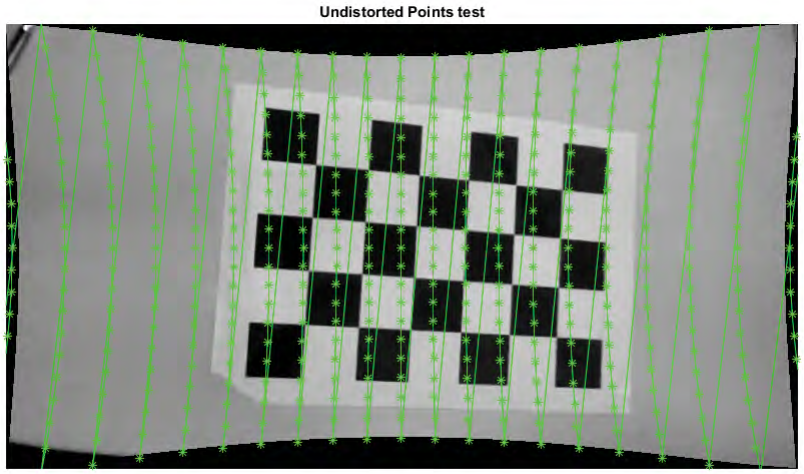


Figure 5.4: Using Drap and Lefèvre new factors in Brown's equation (green lines and pluses) laid over an image undistorted using the iterative approach

Conclusively, the solution is not ideal, however the method by Drap and Lefèvre is very fast to compute. When both approaches are tested in Matlab 2016 on a grid of points (21×21), the computation time of Drap and Lefèvre approach compared to the iterative approach is 22.6 times faster. This speed increase is most valuable for the project. With the lens correction the landmark position can be computed more accurately.

As a summary, the computation of $\vec{s}$ is done as follows:

1. The distorted image coordinates of the detected features are first shifted to the center of the image. This is done with (5.1) resulting in x_{dis} and y_{dis}.
2. The new coordinates x_{dis} and y_{dis} are rewritten to an ideal situation using (5.9) and (5.10). This results in coordinates x_{ideal} and y_{ideal}.
3. The estimated ideal coordinates are then used to compute the direction vector $\vec{s}$ according to (5.2).

With the direction of the detected features the landmarks position can be computed.

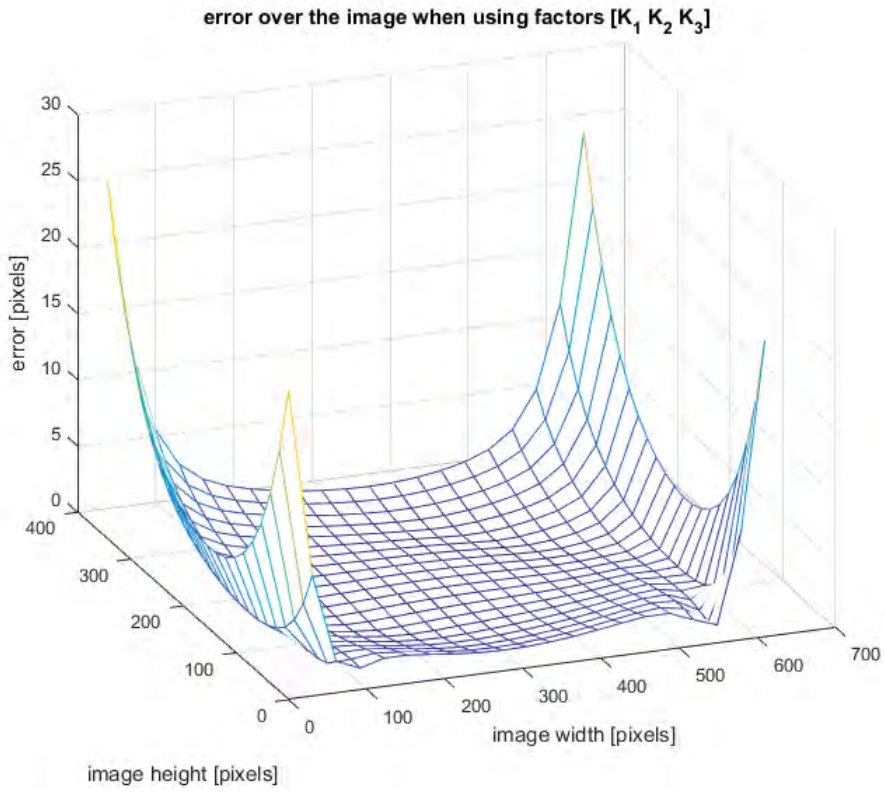


Figure 5.5: The resulting error when using Drap and Lefèvre their factors in Brown's equation positioned over the image in pixels. The error is the difference between Drap and Lefèvre approach and the iterative approach.

5.2.2 Landmark position using epipolar geometry

Using a monocular camera with visual tracking is a fundamental part of VSLAM algorithms. Monocular cameras are used often for VSLAM, however they have a downside: they give no information of depth. The depth of landmarks can be computed from two images taken from two different positions. Therefore, as the drone moves around the depth can be computed by comparing images over time. Computing the position of a landmark from two such images is called epipolar geometry.

Epipolar geometry computes the intersection of two lines originating from different points in space. It is therefore trivial in 2-D, simply finding the intersection of two given lines. Even with an error in the lines an intersection can be found. In 3-D this becomes harder, lines practically never intersect because the lines contain a small error. These 3-D lines shall narrowly pass each other if they are not perfectly aligned. In practice sensor noise gives an error in the direction of the line, and an error in the estimated drone state gives an error in the origin of the line. Because of this, these lines will most definitely not intersect. In order to explain the process of bypassing this problem first an ideal 3-D situation is considered. In the ideal situation two intersecting lines have one plane on which they both lay, thereby always intersecting.

In Figure 5.6 the geometry of an arbitrary situation is given. Two arbitrary drone positions are given as $\vec{p}_{k-1}$ and $\vec{p}_k$. The landmark's unknown position is $\vec{l}_m$. The directions in which the drone detects the landmarks are given by $\vec{s}_{k-1}$ and $\vec{s}_k$, these two vectors are normalised. All the vectors containing the prefix ^{w} have their coordinate frame fixed to the world. The three vectors ${}^w\vec{p}_{k-1}$, ${}^w\vec{p}_k$ and ${}^w\vec{l}_m$ form a triangle. The vectors ${}^w\vec{p}_{k-1}$, ${}^w\vec{p}_k$, ${}^w\vec{s}_{k-1}$ and ${}^w\vec{s}_k$ are known. The vector ${}^w\vec{l}_m$ is what needs to be derived from the known data. In order to do so the distance d of the landmark from the drone position is needed. The distance is computed with the help of geometry.

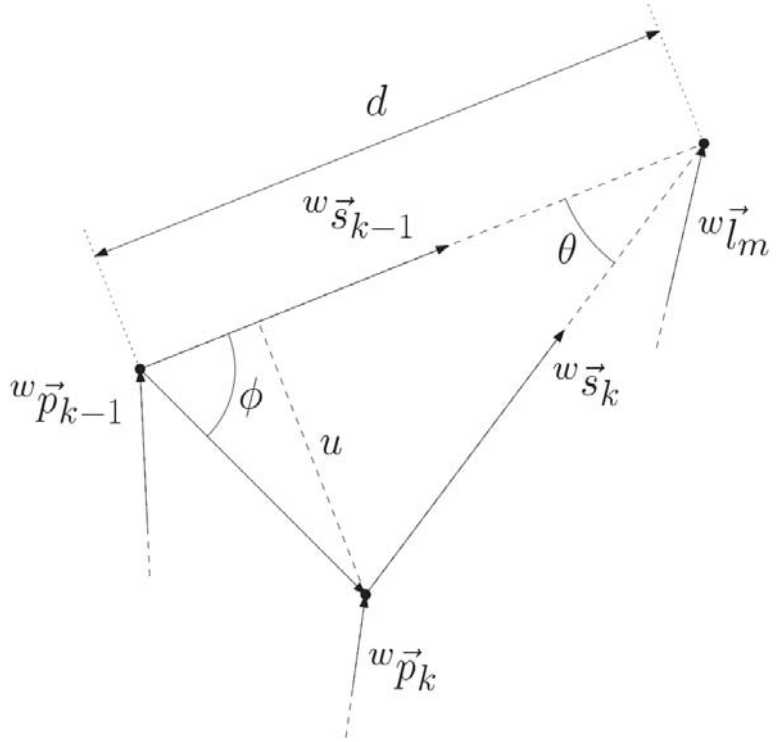


Figure 5.6: Epipolar geometry used for approximating landmark position

The angles, identifying the shape of the triangle, are computed as:

$$\phi = \arccos\Big(({}^{w}\vec{p}_k - {}^{w}\vec{p}_{k-1})_{\text{norm}} \cdot {}^{w}\vec{s}_{k-1}\Big), \tag{5.12}$$

where the subscript X_{norm} indicates a normalised vector, and

$$\theta = \arccos\Big({}^{w}\vec{s}_k \cdot {}^{w}\vec{s}_{k-1}\Big). \tag{5.13}$$

With the shape of the triangle known, only the scale needs to be computed. The only known length is the difference between the two drone positions ${}^{w}\vec{p}_{k-1}$ and ${}^{w}\vec{p}_k$. Using geometry, first the triangle is split in two, since *right triangles* are needed for basic geometry to work. The line splitting the triangle in two has length u and can also be seen in Figure 5.6. The length u is computed with

$$u = ||{}^{w}\vec{p}_k - \vec{p}_{k-1}|| \sin(\phi). \tag{5.14}$$

This length combined with angle θ is used to compute the distance between ${}^{w}\vec{p}_k$ and ${}^{w}\vec{l}_m$.

$$d = \frac{u}{\sin(\theta)}. \tag{5.15}$$

Lastly the landmark position is computed as

$${}^{w}\vec{l}_m = {}^{w}\vec{p}_k + d\,{}^{w}\vec{s}_k. \tag{5.16}$$

The list of given equations can be used to compute the landmark location ${}^{w}\vec{l}_m$ from vectors ${}^{w}\vec{p}_{k-1}$, ${}^{w}\vec{p}_k$, ${}^{w}\vec{s}_{k-1}$ and ${}^{w}\vec{s}_k$ under ideal circumstances.

The equations made for the ideal situation are only based on one assumption, that the lines described by directions ${}^{w}\vec{s}_{k-1}$ and ${}^{w}\vec{s}_k$ cross. Practically this does not happen, the lines narrowly pass each other. The situation can be altered so that a situation is created where the equations

can be used. This situation is created when the vectors ${}^w\vec{s}_{k-1}$ and ${}^w\vec{s}_k$ are projected on the same plane. Projecting a vector $\vec{v}$ onto a plane with normal $\vec{n}$ is done with the equation

$$\vec{v}_{\text{projected}} = \vec{v} - (\vec{v} \cdot \vec{n})\vec{n}. \tag{5.17}$$

It is suggested to project the vectors on the plane spanned by the vectors $\vec{\delta}_p = {}^w\vec{p}_k - {}^w\vec{p}_{k-1}$ and ${}^w\vec{s}_{k-1}$. This plane is chosen because in general circumstances the first position estimation is more accurate. The earlier position estimates are more accurate because the position estimates following afterwards are based on the previous accuracy. Therefore the first position ${}^w\vec{p}_{k-1}$ is chosen as a basis. The vectors $\vec{\delta}_p$ and ${}^w\vec{s}_{k-1}$ originate from this position. Thus the projected vector of ${}^w\vec{s}_{k-1}$ is computed by

$${}^w\vec{s}_{\text{projected}} = {}^w\vec{s}_k - ({}^w\vec{s}_k \cdot (\vec{\delta}_p \times {}^w\vec{s}_{k-1})\text{norm})(\vec{\delta}_p \times {}^w\vec{s}_{k-1})\text{norm}. \tag{5.18}$$

The landmark location $\vec{l}_m$ is computed with only a minimal error by using (5.18) with ${}^w\vec{s}_k$.

The landmark position is an estimation. The standard deviation of this estimation depends on the accuracy of the drone's position ${}^w\vec{p}_k$ and the accuracy of the detected features direction ${}^w\vec{s}_k$. The accuracy of the position of the drone is given as σ_p, and σ_a is the standard deviation of the angle error of direction $\vec{s}$. The standard deviation of the landmark now becomes:

$$\sigma_{l,k} = \sigma_{p,k} + d\tan(\sigma_a) \quad \text{for small } \sigma_a \quad \sigma_{l,k} = \sigma_{p,k} + d\sigma_a. \tag{5.19}$$

In this equation $\sigma_{p,k}$ is the standard deviation of the drone's position estimate at instance k. The derivation of this equation can be found in Appendix D. By using epipolar geometry and projecting the vectors ${}^w\vec{s}_{k-1}$ and ${}^w\vec{s}_k$ on the same plane, the position of the landmark ${}^w\vec{l}_m$ is computed with minimal computational effort. This position can then be stored into the map of landmarks, along with the other needed data.
The computations become different if the position of the landmark is already known. In this case the position of the landmark is only updated. Less computations are needed to update the position of a landmark than it is to estimate the position as was done above.

5.2.3 Landmark position update

When the drone detects a landmark with a known position, then the position only needs to be updated. For this the location of the landmark must again be computed to determine a new position estimate. The new estimate computation is shorter and easier because more data is available. The landmark position is computed again by finding the distance between the drone's and landmark's position, distance d. This is computed by projecting $({}^w\vec{l}_{k-1} - {}^w\vec{p}_k)$ onto ${}^w\vec{s}_k$. This way the missing depth is taken from the old landmarks position and the position parallel to the image is taken from the new data. So the equations become:

$$d = {}^w\vec{s}_k \cdot ({}^w\vec{l}_{k-1} - {}^w\vec{p}_k) \tag{5.20}$$

and

$${}^w\vec{l}_k = {}^w\vec{p}_k + d{}^w\vec{s}_k. \tag{5.21}$$

By computing the distance this way the position on the line described by ${}^w\vec{s}_2$ is the closest to the original landmark position ${}^w\vec{l}_{k-1}$.
The accuracy of this position estimate is derived with the following equation:

$$\sigma_{l,k} = \sigma_{p,k} + d\sin(\sigma_a) \quad \Rightarrow \quad \sigma_{l,k} = \sigma_{p,k} + d\sigma_a \text{ for small } \sigma_a. \tag{5.22}$$

With the new position estimate the old landmark position in the map should be updated. This update is done with a Kalman filter.

Kalman filter

The Kalman filter is effective for smoothening measurement noise. The Kalman filter roughly works by estimating the expected data values and then combining the expected data with the measured data based on their accuracy. The standard equations for the Kalman filter are usually split in two sets. The first is the computation of the expected data and its accuracy called the *priori*. The measured data update with the expected data is the *posteriori*. Since the expected position and appropriate accuracy σ_l is known the priori simply becomes:

$$\text{Prior position estimate: } \hat{l}_{k|k-1} = {}^w\vec{l}_{k-1}, \tag{5.23}$$

$$\text{Prior position covariance: } P_{k|k-1} = \sigma^2_{l,k-1}. \tag{5.24}$$

The posteriori is equal to:

$$\text{Position difference measurement and priori: } \vec{e}_k = {}^w\vec{l}_k - H_k\hat{l}_{k|k-1}, \tag{5.25}$$

$$\text{Position difference's covariance: } S_k = \sigma^2_{l,k} + H_k P_{k|k-1} H_k^T, \tag{5.26}$$

$$\text{Kalman gain: } K_k = P_{k|k-1} H_k^T S_k^{-1}, \tag{5.27}$$

$$\text{Posterior position estimate: } {}^w\vec{l}_{k|k} = \hat{l}_{k|k-1} + K_k\vec{e}_k, \tag{5.28}$$

$$\text{Posterior position covariance: } P_{k|k} = (I - K_k H_k) P_{k|k-1} \tag{5.29}$$

In the case of combining the two landmark positions matrix H_k is equal to the identity, because the measurement data and expected data are of the same kind and no transformation is necessary. Also since the matrices are of size $\mathbb{R}^{1\times1}$ the matrices become scalars. This leads to the following posteriori with priori included equations:

$$\vec{e}_k = {}^w\vec{l}_k - {}^w\vec{l}_{k-1}, \tag{5.30}$$

$$s_k = \sigma^2_{l,k} + \sigma^2_{l,k-1}, \tag{5.31}$$

$$k_k = \sigma^2_{l,k-1} s_k^{-1}, \tag{5.32}$$

$${}^w\vec{l}_{k|k} = {}^w\vec{l}_{k-1} + k_k\vec{e}_k, \tag{5.33}$$

$$p_{k|k} = (1 - k_k)\sigma^2_{l,k-1} \quad \Rightarrow \quad \sigma_{l,k|k} = \sqrt{(1 - k_k)\sigma^2_{l,k-1}}. \tag{5.34}$$

The old and measurement data is discarded and the resulting landmark position ${}^w\vec{l}_{k|k}$ and standard deviation $\sigma_{l,k|k}$ are stored in the map instead.

5.3 Removing mismatches

With the geometrical data of the landmarks the mismatches can be located. Mismatches must be removed to maintain robust estimations of both landmark and drone positions. For the human eye it is clear which of the matches are false positives, some of the matches point into an entirely different direction from the rest. Considering that, there are two types of matches: inliers and outliers. Naturally, these outliers need be removed. RANSAC (Random Sample Consensus) [19] can be used to estimate the transformation to go from one image to the other, thereafter it can filter out the outliers. It uses an iterative process to determine the transformation. Each iteration it edits the transformation by reducing the RMSE (root-mean-squared error) between the detected feature's position and the expected feature position according to the transformation. Features with a RMSE larger than a set threshold are removed as outliers.

There are a few limitations with using RANSAC. First there needs to be at least eight positive matches, second these matches need to be from the same image. Otherwise RANSAC cannot compute the transformation. These two limitations are a problem for the VSLAM algorithm, there is no guarantee that there are at least eight positive matches and the matches are generally from different images. As a result it is often not possible to compute the transformation.

The transformation is needed to compute the estimated pixel position of the match. There is a way to find these estimated pixel positions for each individual match without the need to compute the transform. This estimation is computed differently for two different situations.

5.3.1 Removing mismatches with landmarks

First the situation is considered where a feature is matched with a landmark. In this case, the vector that points to the landmark ${}^{w}\vec{s}$ (see (5.2)) is expected to be

$$ {}^{w}\hat{s} = ({}^{w}\vec{l} - {}^{w}\vec{p})\text{norm}. \tag{5.35} $$

The criteria for a positive match can then be taken as the angle between the expected and measured vector.

$$ \arccos({}^{w}\hat{s} \cdot {}^{w}\vec{s}) < t \quad \Rightarrow \quad {}^{w}\hat{s} \cdot {}^{w}\vec{s} < t^{*} \tag{5.36} $$

where t^{*} is the set angle threshold, the threshold can also be given as function with a threshold in pixels as input $t^{*} = \cos\left(tf^{-1}\right)$ with f the focal length and t the threshold in pixels. The procedure to remove matches is visualised in Figure 5.7. Inhere the expected data is plotted with the measured data. The difference, denoted by the dashed line is the error distance. This error distance is proportional to (5.36).

For the removal of mismatches between features another set of equations is needed. The features do not have a 3-D position to estimate ${}^{w}\hat{s}$ with.

5.3.2 Removing mismatches with features

The first time a feature is detected the state of the drone and the direction of the feature is stored in the appropriate map. This information describes on which 3-D line the landmark corresponding to this feature lies. With this information the expected pixel position is on line called the *epipolar line*. The matched feature may only be several pixels away from this line.

The 3-D line, or rather the vector, is defined by the old position of the drone ${}^{w}\vec{p}_{k-1}$ and ${}^{w}\vec{s}^{*}_{k-1} = {}^{w}\vec{p}_{k-1} + {}^{w}\vec{s}_{k-1}$. The resulting two vectors are first moved to the body-fixed frame

$$ {}^{B}\vec{p}_{k-1} = R_{g2d}({}^{w}\vec{p}_{k-1} - {}^{w}\vec{p}_{k}) \tag{5.37} $$

$$ {}^{B}\vec{s}^{*}_{k-1} = R_{g2d}({}^{w}\vec{p}_{k-1} + {}^{w}\vec{s}_{k-1} - {}^{w}\vec{p}_{k}) \tag{5.38} $$

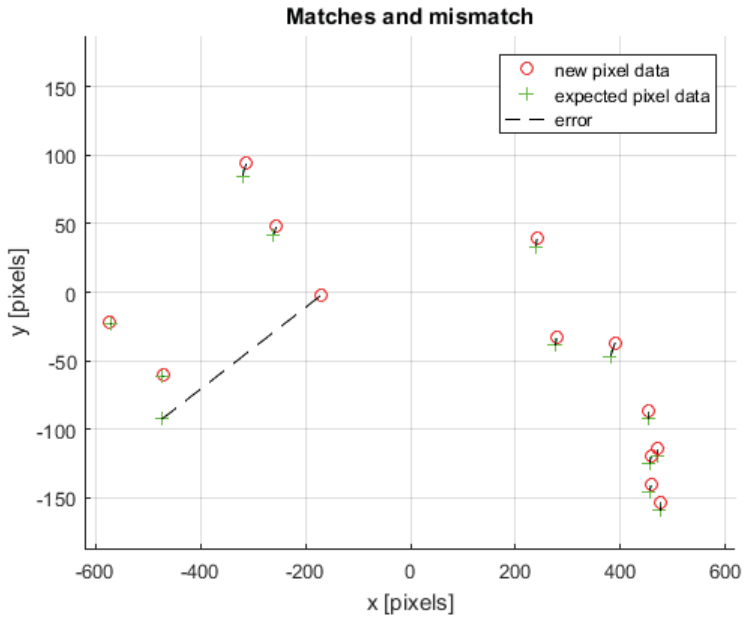


Figure 5.7: Filtering the mismatches from data is visualised here. The newly detected feature's pixel coordinates are expected to be near expected positions. This graph shows a few matches and a clear mismatch.

where X^B represents the body fixed frame and R_{g2d} is the rotation matrix from the world-fixed frame to the body-fixed frame. The 3-D line defined by these two points is projected onto the image plane. The pixels coordinates than become

$$ {}^I\vec{p}_p = fT|R_{B2I}{}^B\vec{p}_{k-1}|_{\text{norm,z}} \quad \text{with} \quad {}^I\vec{p}_p \in \mathbb{R}^{2\times 1} \tag{5.39} $$

$$ {}^I\vec{p}_s = fT|R_{B2I}{}^B\vec{s}^*_{k-1}|_{\text{norm,z}} \quad \text{with} \quad {}^I\vec{p}_s \in \mathbb{R}^{2\times 1} \tag{5.40} $$

with

$$ T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad R_{B2I} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{bmatrix}, \tag{5.41} $$

and f representing the focal length of the camera. Notation $|\vec{X}|_{\text{norm,z}}$ stands for the normalisation of vector $\vec{X}$ over the third vector element.

With the above equations two pixel coordinates are known, and thus the 2-D line. The three points ${}^I\vec{p}$, ${}^I\vec{p}_p$ and ${}^I\vec{p}_s$ form a triangle. The distance d between the line spanned by ${}^I\vec{p}_p$ and ${}^I\vec{p}_s$ to arbitrary point ${}^I\vec{p}$ is the same as dividing double the area of the triangle by length $||{}^I\vec{p}_p - {}^I\vec{p}_s||$. The area of a triangle is the same as half the area of a parallelogram spanned by two border vectors. Half the area of a parallelogram is computed with

$$ A = \frac{1}{2}({}^I\vec{p}_p - {}^I\vec{p}_s) \times ({}^I\vec{p}_p - {}^I\vec{p}), \tag{5.42} $$

hereafter distance d becomes

$$ d = \frac{2A}{||{}^I\vec{p}_p - {}^I\vec{p}_s||} \quad \Rightarrow \quad d = \frac{({}^I\vec{p}_p - {}^I\vec{p}_s) \times ({}^I\vec{p}_p - {}^I\vec{p})}{||{}^I\vec{p}_p - {}^I\vec{p}_s||}\,. \tag{5.43} $$

If the distance is adequate then

$$ d < t \tag{5.44} $$

holds true and the match is positive. The threshold t is given in pixels, the same threshold as mentioned in previous section.

Concluding the removal of mismatches, the matches that fail tests (5.36) or (5.44) are considered mismatches. Whether it takes the first or the second test depends on the type of match: a match between features or between a feature and a landmark.

Both methods of removing mismatches are based on the assumption that the error of the drone state estimate is small. Especially a large error in the angles can result in larger values for d. Therefore, the estimation of the drone state should be sufficiently accurate. As an option, it is possible to replace the different thresholds t with $t(\sigma)$ where the threshold depends on the accuracy of the different position estimates.

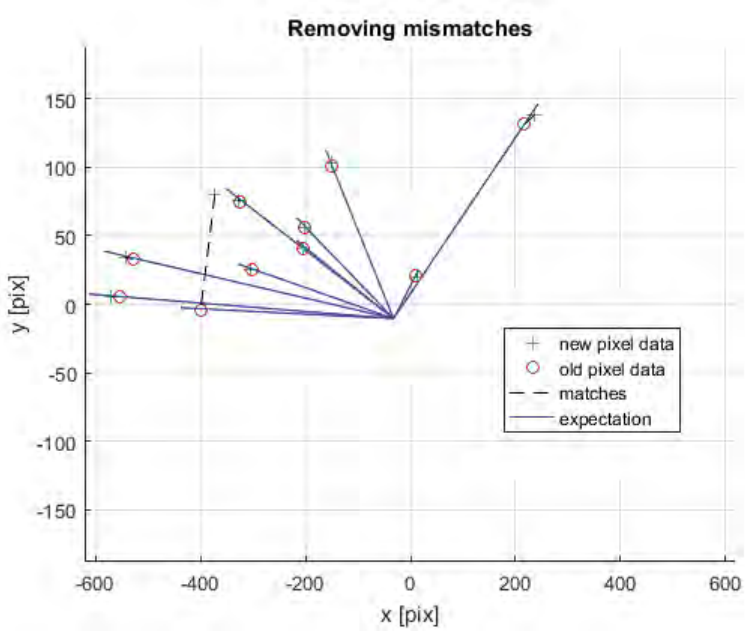


Figure 5.8: Filtering the mismatches from data is visualised here. The newly detected feature's pixel coordinates are expected to be near the lines. This graph shows a few matches and a clear mismatch. The matches are a selection of the matches seen in Figure 4.8.

5.4 Implementing map building

The algorithm part for map building is the largest of all different VSLAM steps. This is due to the many functions involved in map building. Map building contains the following functions:

1. matching features,
2. removal of mismatches,
3. lens correction,
4. position computations of landmarks,
   (a) first position computation with epipolar geometry,
   (b) position update,
5. storage of features,
6. storage of landmarks.

Localisation of the drone is also entangled in the algorithm for map building. The localisation and map building depend on each other. However, the localisation is not yet implemented in this chapter. The implementation of the localisation is edited near the end of the next chapter. For now it is assumed the true position is known.

All the above functions can be found in chronological order in Algorithm 3. The algorithm is quite large: a result of the many steps and functions needed. The algorithm has the state of the drone, the two maps that have been created over time (at time equals zero the empty map as was initialised in Algorithm 2) and the newly detected features from Algorithm 1 as input.

See Algorithm 2, herein the two maps are initialised. The first map contains the landmarks, points in 3-D space, a result of matched features, and the second map contains features, points in 2-D space which are the unmatched features. The map each contains the descriptor and position of each point, in addition the landmark map stores the standard deviation σ_l and the feature map stores the drone's state at the time it was detected.

First the map of landmarks is matched with the newly detected features, see line 1 in Algorithm 3. Thereafter, the matches are one by one put through the functions listed above (lines 2–13): first is corrected for lens distortions (line 4) then the mismatches are removed (lines 6–10). The successful matched landmarks are thereafter updated: including the landmarks position and standard deviation (line 11). Lastly the match is removed from the newly detected features. After all matches are put through this list of functions the remaining detected features are matched with the map of features.

The remaining detected features are matched with the map of features in line 14, and roughly the same routine follows. First is corrected for the lens distortion (lines 17 and 18), then the mismatches are removed in lines 19–23. Thereafter, the position of the landmark (line 24) together with the appropriate standard deviation are computed and both are stored with the descriptor in the map of landmarks (lines 23–25). Then the matches are removed from the map of features and of the detected features in lines 27 and 28.

The remaining detected features are stored in the map of features (lines 30–32). The map of features contains all unmatched features, as was stated before. Therefore, as time progresses the unmatched features become irrelevant. To keep the map from growing too large and to prevent mismatches at later times, the oldest features are removed from the map, see lines 34–37.

With that the mapping of the environment is finished. Only the localisation of the drone is missing. The localisation is added to this algorithm in the next chapter.

Algorithm 2: Map initialisation

Input:
Output: empty map of features F and map of landmarks L
Parameters:
`/* Initialisation of the maps is done prior to the start of the VSLAM algorithm. Each element in the map contains the following: */`
1 $L_p \leftarrow \emptyset$ for landmark positions;
2 $L_d \leftarrow \emptyset$ for descriptors;
3 $L_\sigma \leftarrow \emptyset$ for σ_l;
4 $F_p \leftarrow \emptyset$ for feature image coordinates ${}^I\vec{p}$;
5 $F_d \leftarrow \emptyset$ for descriptors;
6 $F_S \leftarrow \emptyset$ for drone state containing $[w\vec{p}_d,\ R_{d2w},\ \sigma_d]$;

Algorithm 3: Map building

Input: drone state S, map of features F, map of landmarks L and newly detected features D containing descriptors $D_d \in D$ and the corresponding pixel coordinates $D_p \in D$.
Output: updated maps F and L
Parameters: matches M, matching function $f(\text{descriptor1}, \text{descriptor2})$

$M \leftarrow f(L_d, D_d)$ match L with D;
foreach *element* m_n *of* M **do**
  $[i,\ j] \leftarrow m_i$ get indexes of matches;
  $d \leftarrow D_i$ get matched feature element and correct the lens distortion;
  $l \leftarrow L_j$ get matched map element;
  $p \leftarrow Tl_p$ projection on image plane with matrix T;
  if $||p - d_p|| >$ *threshold* **then**
    remove mismatch m_n from m;
    continue
  end
  $L_j \leftarrow$ update landmark with new data;
  remove d from D;
end
$M \leftarrow f(F_d, D_d)$ match L with D;
foreach *element* m_n *of* M **do**
  $[i,\ j] \leftarrow m_i$ get indexes of matches;
  $d \leftarrow D_i$ get matched feature element and correct the lens distortion;
  $f \leftarrow F_j$ get matched map element and correct the lens distortion;
  project the line onto the image plane on which f_p is expected to be;
  if f_d *is too far from the line* **then**
    remove mismatch m_n from m;
    continue
  end
  $L_p \leftarrow$ add compute landmark position;
  $L_d \leftarrow f_d$ add descriptor;
  $L_\sigma \leftarrow$ add compute landmark standard deviation;
  remove f from F;
  remove d from D;
end
$F_p \leftarrow D_p$ add feature pixel coordinates;
$F_d \leftarrow D_d$ add descriptor;
$F_S \leftarrow S$ add drone state;
if *size* $F >$ *maximal map size* **then**
  $r \leftarrow$ size F - maximal map size;
  remove r of the oldest elements in F;
end

5.5 Concluding map building

Two maps are created, one map holds the landmarks and is used for localisation, the other is used to store a list of detected features that do not yet have a position in 3-D.

In summary, to create a map of the environment the 2-D position of detected features in the image space must be transformed to a 3-D position in world space. In order to do this first the image must be corrected for distortion effects. The image is mainly prone to lens distortion. By fitting a polynomial over a distorted image the shape of the distortion is discovered. By taking the inverse function of the polynomial the distortion can be corrected. Darp et al. [16] described an easy way of estimating the inverse of the polynomial.

The directions of the feature seen from the camera can be computed with the undistorted pixel coordinates. This direction is in the form of a 3-D normalised vector and this vector depends on the

camera specifications. With this vector the position of matched features can be either computed or updated. The first time a feature is matched its position is computed with epipolar geometry. The other times the position of the landmark is only updated, requiring a different set of equations. In both cases the accuracy of the landmark's position is also stored in the form of a standard deviation.

Some of the matches are false positives: mismatches. False matches with landmarks are quite easily found, because the landmark position is known. By projecting the landmark onto the image an estimated feature location in image space is made. If the distance between the location of the detected feature and the estimate location is larger than a set value, then the match is a false positive.

To discover false matches between two features the estimated location becomes a line rather than a point. The line is defined by the previous drone state (the state in which the feature was detected) and the direction in which the feature was detected. This line is projected onto the image plane. If the detected feature is far from the line the match is again a false positive.

With the lens correction, landmarks, and mismatch filtering the map for the VSLAM algorithm is made. The map of landmarks in turn is used to compute the location of the drone, see next chapter.

Chapter 6

Localisation of the drone

All in all, the whole VSLAM algorithm is meant to compute the position of the drone. With the map of the environment build of landmarks the drone is capable of computing just that. The observer is capable of computing the orientation of the drone correctly and drift free. Therefore, the only thing that needs to be computed is the position of the drone. The orientation is thus considered to be perfect in the localisation step.

The matches with the landmarks are used for computing the position estimate. The position computed from the map is combined with the drone's position according to the observer. This returns the final position estimation of the VSLAM algorithm. The localisation should again be fast to compute. As a basis FastSLAM was examined first, a method that uses a particle filter to compute the position. This method is then edited to fit the goals of the VSLAM algorithm.

6.1 Creating a positioning scheme

FastSLAM [10] is a method to shorten the computation time, compared to EKF-SLAM [9]. It is based on the particle filter, and works similar to EKF-SLAM. Only, instead of the predicted state, it 'predicts' the position of the drone with possible locations. These locations are called particles. In Figure 6.1 the particle filter in FastSLAM is explained step by step. First is the most likely position of the robot taken, which would be the state from the observer. This position must have a known standard deviation. The expected location with deviation is indicated by the grey circle. Then particles are distributed according to the normal distribution, so the particles are clustered around the expected robot position. The particles are visible as grey to black dots. Each particle is given a weight value, this weight determines how close the particle is to the expected position. The darker the dots, the higher this weighted value is. Thereafter, for each particle is derived how likely that particle is the correct position based on the landmarks. The weighted value of the particles is then updated according to the landmarks. Notice the change in grey in the figure at the fourth step along with the shift in the grey circle. There where the most particles with the highest weight value are clustered, is the final position the robot.

The main benefit of this filter is that it changes the computation time from $m \times m$ to $m \times n$, where m is the number of landmarks and n the number of particles, compared to EKF-SLAM. Generally $m > n$, reducing the computation time. The drawback is that, as the state of the drone increases it needs more and more particles to deliver a satisfying result. Multiple papers [10, 23, 31] state that the use of FastSLAM is limited.

The amount of computations still seems excessive. Dropping the amount of computations is still one of the most important goals of this project. Therefore, another method is proposed by reversing the process. The particle filter estimates the position from the drone, however in the proposed method the position is estimated from the landmarks. See Figure 6.2, the drone has seen different landmarks in different directions. If a detected landmark is located somewhere then the position of the drone would be directly on the line defined by the direction it was detected in. A set of these lines forms a cluster. The center of this cluster is where the drone should be located.

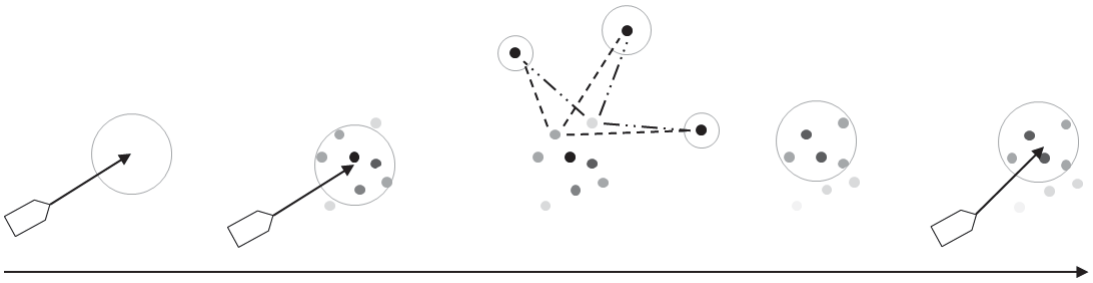


Figure 6.1: The use of a particle filter in locating landmark positions is explained in steps from left to right. On the left the drone moves to the indicated area. The position of the drone is a normally distributed in this area. The particle filter scatters possible positions according to the normal distribution in the second step. The scattered positions closer to the center have a higher weight than the ones further away from the center. Then, in the third step the weights of the scattered positions updated based on the detected landmarks. In the fourth step a new updated normally distributed position estimate is created. Lastly the center of this new normal distribution is the drone's position according to the particle filter.

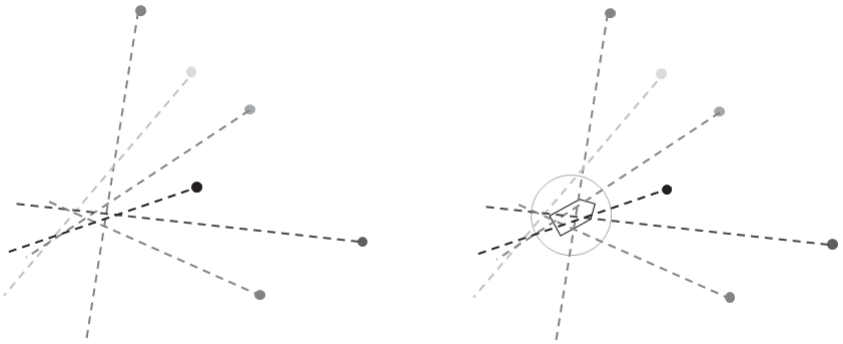


Figure 6.2: Determine the position directly from landmarks

This results however in $m \times m$ computations because to combine the lines the minimal distance has the be computed for each combination. To reduce the computations to m computations every line must be replaced by a 3-D point. The line does not give any information about the distance between the drone and the landmark. Therefore, the 3-D point is chosen so that the position change of the drone is perpendicular to the line. In other words, the position estimate gives no false information about the distance between drone and landmark, see Figure 6.3.

6.1.1 Localisation from map

The localisation is done with the matched landmarks. Let i be the indexes of the matched landmarks and ${}^{w}\hat{p}_k$ the current position estimate from the observer. Then the position of the drone is given as

$$ {}^{w}\vec{p}_{k,i} = {}^{w}\vec{l_i} - d_i \, {}^{w}\vec{s}_i \quad \text{with} \tag{6.1} $$

$$ d_i = ({}^{w}\vec{l_i} - {}^{w}\hat{p}_k) \cdot {}^{w}\vec{s}_i \tag{6.2} $$

for each individual matched landmark i. The accuracy of these estimations is given as

$$ \sigma_i = \sigma_{l,i} + d_i \sigma_a. \tag{6.3} $$

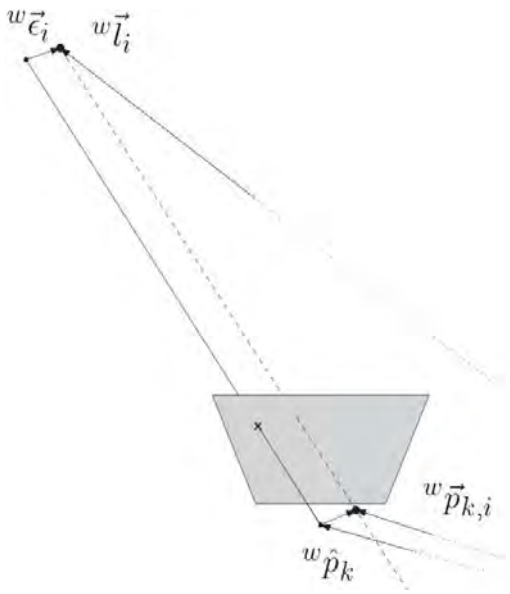


Figure 6.3: This figure shows the camera looking from the expected drone location ${}^w\hat{p}_k$. It detects the landmark i on the image plane at the location indicated by the cross ($\times$). The line on which the landmark is detected moves past the landmarks location ${}^w\vec{l}$. The error perpendicular to the detection lines and the landmarks is indicated with vector ${}^w\vec{\epsilon}$. The estimated position according the landmark is indicated as ${}^w\vec{p}_{k,i}$.

All the position estimations are then combined as a weighted mean. The weight in this case is $w_i = \sigma_i^{-1}$. This leads to the following equation:

$$ {}^w\bar{p}_k = \frac{\sum(w_i \, {}^w\vec{p}_{k,i})}{\sum w_i}, \tag{6.4} $$

Deriving the corresponding weighted mean standard deviation requires some extra attention. In order to derive the standard deviation a Monte-Carlo simulation can be used, this would however burden the drone's computer too much. Therefore, instead of using Monte-Carlo simulation on the drone, the simulation is used to find an appropriate equation for computing the appropriate standard deviation. This derived equation can then be computed onboard the drone. The details on the Monte-Carlo simulation and the resulting equation can be found in Appendix C. The resulting equation is:

$$ \bar{\sigma}_k^2 = \sum_{i=1}^{N} \frac{(w_i^*)^2\sigma_i^2}{W} \quad \text{with} \quad W = \sum_{i=1}^{N}(w_i^*)^2 \quad \text{and} \quad w_i^* = 0.2613\sigma_i^{-0.7642}. \tag{6.5} $$

The equations (6.4) and (6.5) are used to compute the position of the drone and estimate the accuracy of the computed position. By requiring only these equations the amount of computations scales only linear with the amount of detected landmarks.

6.2 Combining initial position estimate and VSLAM

The weighted mean ${}^w\bar{p}_k$ is combined with the initial expected position ${}^w\hat{p}_k$ by using a Kalman filter. These two estimates are combined to reduce the impact of erroneous values. The combining data with the Kalman filter is similar to the combination of landmarks in section 5.2.3. In this case the priori become:

$$ \hat{p}_{k|k-1} = {}^w\hat{p}_k \quad \text{and} \tag{6.6} $$

$$ P_{k|k-1} = \hat{\sigma}^2, \tag{6.7} $$

with $\hat{\sigma}$ the standard deviation of the initial position estimate. With these priori needing no computations they are directly implemented into the posteriori. The combined and final position estimate then becomes:

$$\vec{e}_k = {}^w\bar{p}_k - {}^w\hat{p}_k, \tag{6.8}$$

$$s_k = \bar{\sigma}_k^2 + \hat{\sigma}^2, \tag{6.9}$$

$$k_k = \hat{\sigma}^2 s_k^{-1}, \tag{6.10}$$

$${}^w\vec{p}_{k|k} = {}^w\hat{p}_k + k_k\vec{e}_k, \tag{6.11}$$

$$\sigma_{k|k} = \sqrt{(1-k_k)\hat{\sigma}^2}. \tag{6.12}$$

The resulting ${}^w\vec{p}_{k|k}$ and $\sigma_{k|k}$ are the output of the VSLAM algorithm. This data is send to the observer of the drone together with the initial estimate position estimate of the drone and the time at which the image was taken from the camera. The data from the VSLAM algorithm is always overdue, because of the slow computation rate expected on the drone. The observer needs to know what the position error was at the previous point in time, only then can this data be used. The combination of the data is not be discussed in this report, there has not been sufficient time to fully investigate the options, but some recommendations about this topic are stated at the end of the report in Chapter 8: *Conclusions and recommendations*.

6.3 Implement localisation into the VSLAM algorithm

Overall, only a few equations have to be implemented into the VSLAM algorithm, see Algorithm 5. The most efficient way to do this is by adding them to the algorithm for mapping, Algorithm 3. The equations to compute the position of the drone for each matched landmark can be added in the loop at line 2 to 14, at line 12. There is a list of all the estimates stored and combined as a weighted mean at line 15 for both combining position and standard deviation. Thereafter, the estimated drone position is combined with the initial position estimate using a Kalman filter at line 16 and stored as the current drone state. All the equations following after line 16 benefit from the updated drone state.

The initialisation of the drone's state is given in Algorithm 2. This initialisation states that at the start of the algorithm there is no rotation and the start position of the drone is considered the origin of the world space.

Algorithm 4: Localisation initialisation

Input:
Output: zero-set drone state
Parameters:
`/* Initialisation of the initial drone state is done prior to the start of the VSLAM algorithm. */`

1 $S_p \leftarrow \vec{0}$ with $\vec{0} \in \mathbb{R}^{3\times1}$ a zero-set vector;
2 $S_R \leftarrow I$ with $I \in \mathbb{R}^{3\times3}$ as the identity matrix;

6.4 Concluding localisation

The drone is localised with the use of one position estimate per landmark that assume that the orientation of the drone is correct. This assumption can be made because the orientation of the drone can be derived with the IMU drift free.

Using geometry the position of the drone is estimated for each matched landmark separately. By doing so the drone can estimate its position from even one landmark and the amount of computation is linear to amount of matched landmarks. All position estimates are combined with a weighted mean. The weights of this mean are the standard deviation of each position estimate, thereby ensuring that the most accurate estimates are most influential and the least accurate ones is not.

The weighted mean estimate is lastly combined with the initial position estimate with the use of a Kalman filter. The result is the final position estimate, along with its standard deviation, that is the output of the VSLAM algorithm. This data is send to the observer together with the starting state and time when the image was taken.

Algorithm 5: Map building and localisation

Input: drone state S, map of features F, map of landmarks L and newly detected features D containing descriptors $D_d \in D$ and the corresponding pixel coordinates $D_p \in D$.
Output: updated drone state S and updated maps F and L
Parameters: matches M, matching function $f(\text{descriptor1}, \text{descriptor2})$

$M \leftarrow f(L_d, D_d)$ match L with D;
foreach *element* m_n *of* M **do**
    $[i,\ j] \leftarrow m_i$ get indexes of matches;
    $d \leftarrow D_i$ get matched feature element;
    $l \leftarrow L_j$ get matched map element;
    $p \leftarrow Tl_p$ projection on image plane with matrix T;
    if $||p - d_p|| >$ *threshold* **then**
        remove mismatch m_n from m;
        continue
    end
    $L_j \leftarrow$ update landmark with new data;
    $\hat{S} \leftarrow$ add estimate drone position and appropriate standard deviation;
    remove d from D;
end
$\tilde{S} \leftarrow$ weighted mean of $\hat{S}$;
$S \leftarrow$ use Kalman filter to combine $\tilde{S}$ and S;
$M \leftarrow f(F_d, D_d)$ match L with D;
foreach *element* m_n *of* M **do**
    $[i,\ j] \leftarrow m_i$ get indexes of matches;
    $d \leftarrow D_i$ get matched feature element;
    $f \leftarrow F_j$ get matched map element;
    project the line onto the image plane on which f_p is expected to be;
    if f_d *is too far from the line* **then**
        remove d from D;
        remove mismatch m_n from m;
        continue
    end
    $L_p \leftarrow$ add compute landmark position;
    $L_d \leftarrow f_d$;
    add descriptor;
    $L_\sigma \leftarrow$ add compute landmark standard deviation;
    remove f from F;
end
$F_p \leftarrow D_p$ add feature pixel coordinates;
$F_d \leftarrow D_d$;
add descriptor;
$F_S \leftarrow S$ add drone state;
if *size* $F >$ *maximal map size* **then**
    $r \leftarrow$ size F - maximal map size;
    remove r of the oldest elements in F;
end

Chapter 7

Simulations of VSLAM algorithm

The algorithm is tested for results on accuracy and behaviour in simulation. The observer of the drone provides all necessary data but its accuracy remains untested. In order to test and verify the algorithm a dataset is required with accurate data. Thereafter, the algorithm is also tested with a test set made with the drone. All test are conducted on a laptop in Matlab.

To verify the VSLAM algorithm simulations where conducted on two different datasets. First the algorithm was tested with a dataset used by multiple SLAM algorithms, coded *KITTI* [18]. This data set is of a car driving around in a city. It is equipped with cameras, GPS, IMU and more sensors that are not used here, see Figure 7.1. The GPS and IMU give a qualitative position and orientation estimate, the camera returns grayscaled, undistorted images of size 1241×367. The data set provides images taken at 10Hz with corresponding rotation matrix R and position ${}^{w}\hat{p}$ which can be considered the ground truth.

Figure 7.1: The car used to make the test dataset called KITTI, source KITTI.

The second data set that the algorithm is tested with, is made with the AR.drone. The images of the front camera are stored in raw quality and the position and orientation are taken from the observer made by Niels Jeurgens [21, 22], and the same experimental set-up is used as was explained in Chapter 1.1. The top camera provides position data ${}^{w}\hat{p}$ and the IMU provides data for the rotation R. The data set has a sample rate of 3Hz.

7.1 Simulations with the KITTI dataset

As mentioned the KITTI dataset is made with a car driving around inside a city. The car drives over roads, with little to no traffic. The dataset consists of a total of 4541 measuring points k. This means the car is travelling with different speeds up to 50 [km/h]. The output of the VSLAM algorithm, the position estimate of the car, is plotted in Figure 7.3 with the ground truth. The ground truth is also used for the expected position ${}^{w}\hat{p}$ in the algorithm. The map that is created in the simulation is given in Figure 7.4.

The position estimate of the car has some error spikes, see Figure 7.3. These spikes vary in size with the largest spike having an error of 26 [m]. These spikes are a result of false matches. Even though most mismatches get filtered out, some do remain. Mismatches are removed by comparing feature pixel coordinates with expected coordinates. If there is a mismatch the expected pixel coordinates can be anywhere, although the chance is small it is possible that the expected coordinate is close to the measured one. There is a higher chance this happens between matches with features compared to matches with landmarks. As a result the mismatch is still taken into account. These mismatches can contribute to the position estimate of the car. As mentioned the chance of this happening is small, but, in case of this data set, there are 4541 images in which each 1000 features are detected and these features are compared with a map size up to $\approx 90,000$ features and landmarks. These quantities make the chance of mismatches likely to happen at some times.

In the map some erroneous landmarks are visible (Figure 7.4). A cloud of error landmarks is visible around the road. These are the result of the above mentioned mismatches. On average all landmarks are clustered around the road, which is to be expected of an urban area where everything is located close to the streets.

When looking closer, one can see that the landmarks are clustered on sides of the road, see Figure 7.4. This is a result of two things. First of all, the fact that the road itself does not provide many features to track, the sides do. The parked cars, houses, trees and more provide a large source of features and landmarks. The second is that, as the car moves forward, it cannot compute the position of the features in front of it, it needs to detect a feature under two different angles before a position can be computed.

In the zoomed image the middle street consists of 3 vertical lines. From the right, the first thin line is the houses on one side of the street, then the second line are parked cars and the left most line are the houses on the right side of the street. Figure 7.2 shows an image at $k = 3600$ which is located in this street. In the image the parked car and buildings on the left side are clearly visible and are located further from the road, which is also visible in Figure 7.4. There is a gap in parked cars that is visible in both figures, in the zoomed part this gap is located roughly half way along the street. Although the left side of the road gives a high quantity of landmarks, the right side does not. The reason for this is that the right side of the street is covered in shadow. The shadow results in a low contrast. The FAST feature detector measures the difference in contrast around a point, naturally there are less features detected on the right than on the left.

Figure 7.2: Camera image of the KITTI data set taken at location ${}^{w}\vec{p}_{3600} = [274,\ 16,\ -7]^{T}$ orientated along the x-axis in positive direction.

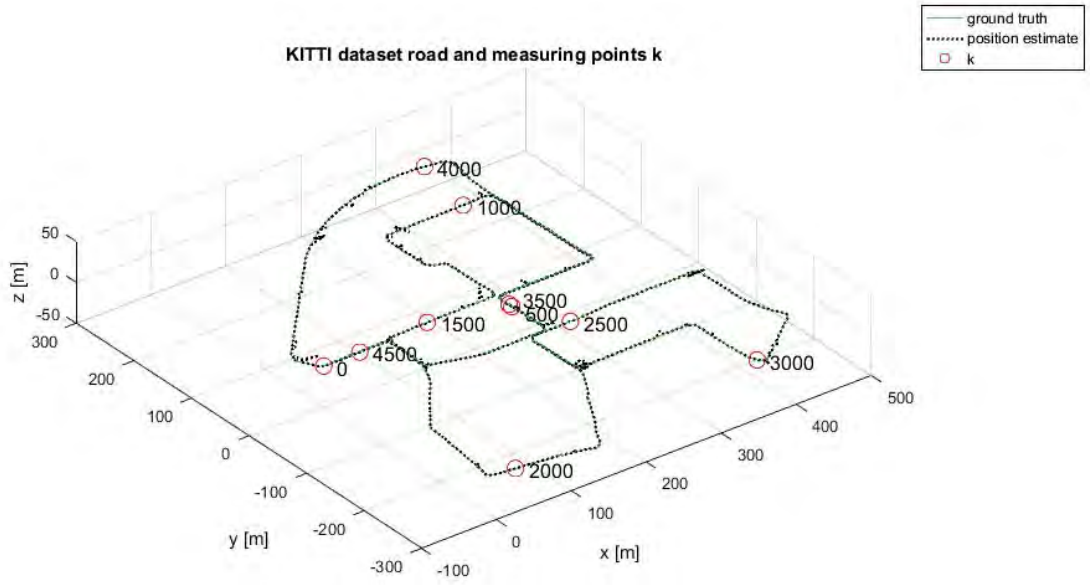


Figure 7.3: Simulation with KITTI: the position estimate according to the VSLAM algorithm.

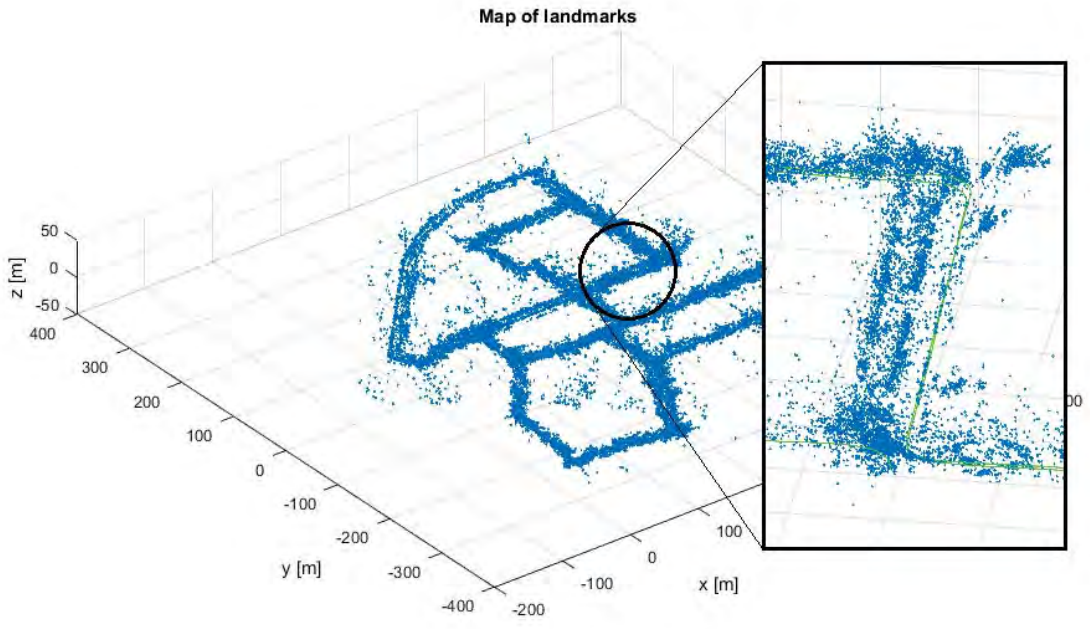


Figure 7.4: Simulation with KITTI: the resulting map. In total this map is filled with 86358 landmarks.

The position estimate and the ground truth are more or less equal, but the error is of interest. The error between the ground truth and the estimation is given as:

$$\epsilon_k = ||{}^w\vec{p}_k - {}^w\vec{p}_k{}^{\text{truth}}||. \tag{7.1}$$

The error is thus defined as the absolute error between the estimated position ${}^w\vec{p}_k$ and the ground truth ${}^w\vec{p}_k{}^{\text{truth}}$.

The absolute error is plotted against measuring instances k in Figure 7.5 at a logarithmic scale. According to this figure the error varies between 0.0002 [m] and 26[m] and the mean error is computed as 0.158[m]. But errors are not equally distributed around this mean. The red line running through the errors is the mean error taken from 40 subsequent instances $\bar{\epsilon}_k = 40^{-1}\sum_{i=k}^{k+40}\epsilon_i$ and is added for a clearer figure. There is a clear drop in the error between $k = 3300$ and $k = 3900$. In this range the car is moving along a street it already passed through before in the same direction, this is clear from Figure 7.3. Here can be seen that from just before $k = 3500$ and before $k = 4000$ the car is moving the same trajectory as $k = 500$ to $k = 1000$. This means that the car cannot only locate itself from newly derived landmarks but also has access to the old map made when the car first drove there and uses it. This strengthens the statement made that the algorithm does not drift over time but over distance; as soon as the algorithm detects old landmarks it corrects the position with this data. The result is a far better position estimate with an average of 0.01 to 0.05[m] and the maximum and minimum error being 0.0002 and 0.2[m] respectively.

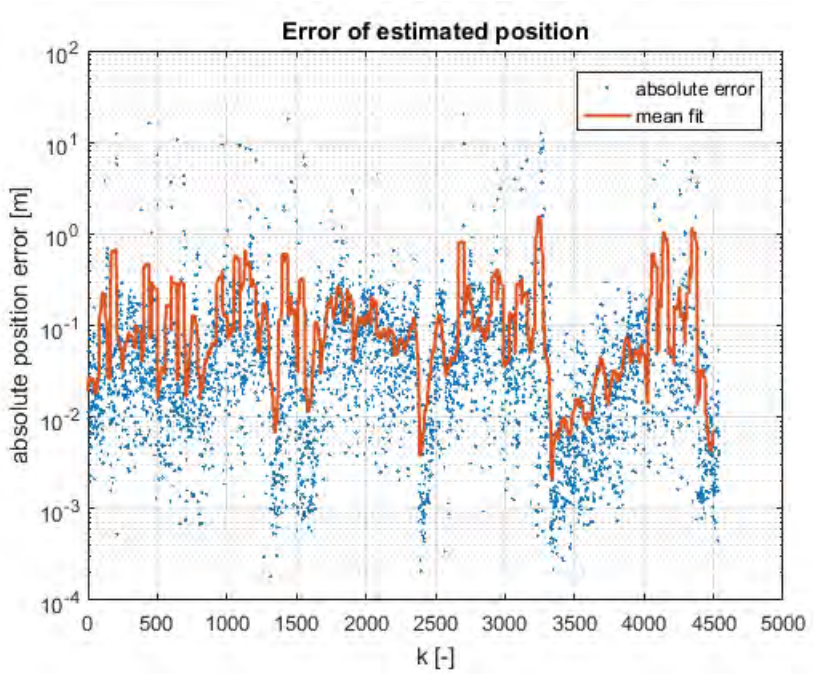


Figure 7.5: The absolute error of all instances k in logarithmic scale. The thick red line is the mean error of 40 combined instances.

More drops in the error are visible in Figure 7.5. There are drops around $k = 1300$, $k = 1600$, $k = 2400$ and $k = 4500$. At every single one of these instance the algorithm had access to an old set of landmarks in the map:

1. The instance $k = 1300$ is located at an intersection were the car drove before at $k = 550$;
2. Instance $k = 1600$ passes the same street as instance $k = 150$;
3. Instance $k = 2400$ passes the same intersection as instance $k = 400$;
4. The last instance $k = 4500$ is when the car returns to its start position $k = 0$.

When comparing these instances, the number of matches also increases. In Figure 7.6 the number of matches are showed, both the matches with landmarks and features and for each both with and without mismatches. It is most clear that around $k = 3300$ to $k = 4900$ the number of matches with landmarks increases. A natural result of the extra matches with the old landmarks. The number of matches with features shows also an increase in filtered matches. The other instance that showed an increased precision do not all show this clear increase in matches. Only $k = 1600$ and $k = 4500$ show the same behaviour, because here the car follows the same trajectory for a short time, the others merely cross roads.

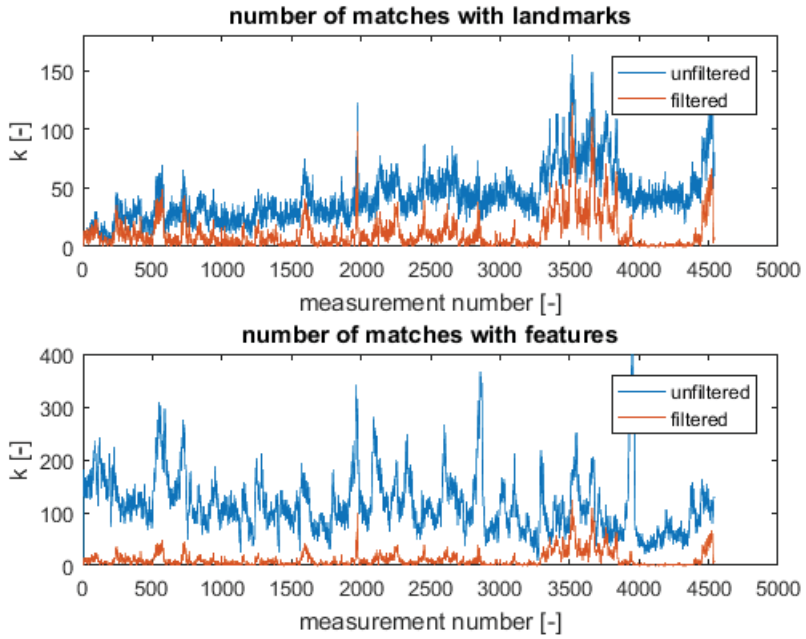


Figure 7.6: The number of filtered and unfiltered matches with landmarks and features for for each instance k.

The error is correlated with the number of landmark matches. Logically since each landmark match results in one position estimate, the more estimates the more accurate the final combined position estimation becomes. The correlation is shown in Figure 7.7. The top left plot shows the number of matches versus the error. Immediately can be seen that the large errors are a result of few matches. This holds true because if there is still an unfiltered mismatch the position estimate shall be wrong, and if there are no or few rightly derived estimates then the mean is strongly influenced. As a test the position estimates with no, one or two matches are removed. The result is plotted in the top right plot in Figure 7.7. The largest error remaining is ≈ 0.65[m]. The error is inversely exponentially proportional to the number of matches.

To reduce the error the algorithm should find as much correct matches as possible. The number of matches that the algorithm found during this test is given in the lower plots in Figure 7.7. The number of matches, both filtered and unfiltered show a Rayleigh probability distribution, the algorithm successfully matches 6 landmarks most of the times (the median) which it can use for the position estimate. On average the algorithm successfully matches 11 landmarks.

From this it can be concluded that the filtering of landmark matches might be to aggressive at the moment. By setting the threshold for the removal mismatch higher, there will be more successful matches and more data to compute the position estimate with. This is not guaranteed to improve the accuracy because of more remaining mismatches but it will be an interesting test for further research.

The algorithm works properly with the KITTI dataset, the dataset made with the AR.drone behaves differently. The error is expected to change. The accuracy increases as a result of the test environment. The drone is tested in a single room, so the algorithm will detect the same landmarks more frequently. However, the accuracy decreases because of a lower sampling rate and a poorer position/orientation estimate from the observer.

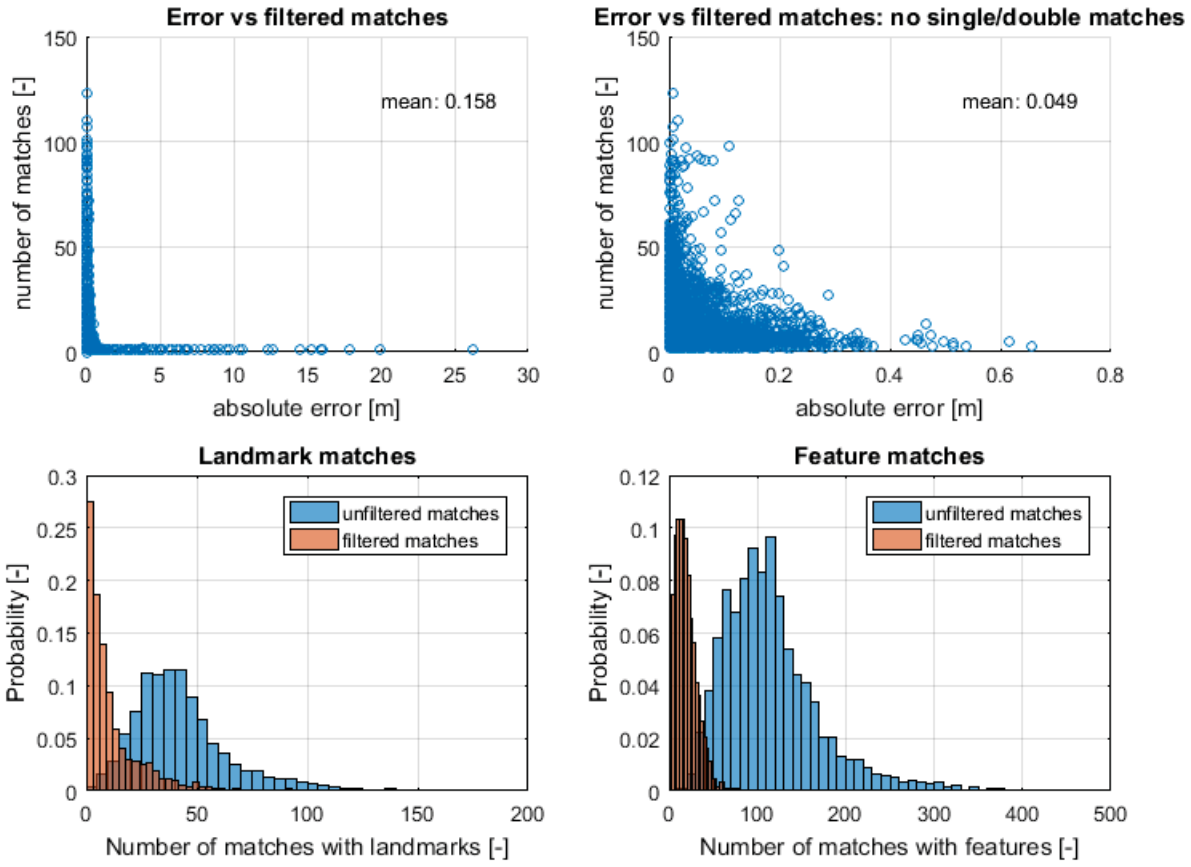


Figure 7.7: Simulation with KITTI: correlation between matches and absolute error.

7.2 Simulations with the AR.drone

The dataset made with the drone is used for further testing. The data set consists of 161 measurements, located ≈ 0.3[s] apart on the time line. The resolution of the camera is 720p, or 1280×720 pixels. The images are distorted and are corrected with the polynomial fit for lens correction.

The dataset however does not have all the appropriate data available. The observer is not capable of providing a position estimate, in addition the orientation of the data set is questionable. Because of these shortcomings the test set-up has been changed. The data that is available is the in-plane x and y position of the drone, and the yaw angle measured with the magnetometer. Because these variables are recovered, the VSLAM algorithm is converted to 2-D space. The changes involve converting all z elements of vectors ${}^w\vec{p}_k$, ${}^w\hat{p}_k$, ${}^w\vec{s}_k$ and ${}^w\vec{l}$ to zero. Thereby turning all vector computations from 3-D world-space to 2-D world space. This conversion is possible because the z translation and angles except yaw are roughly unchanged during the whole test.

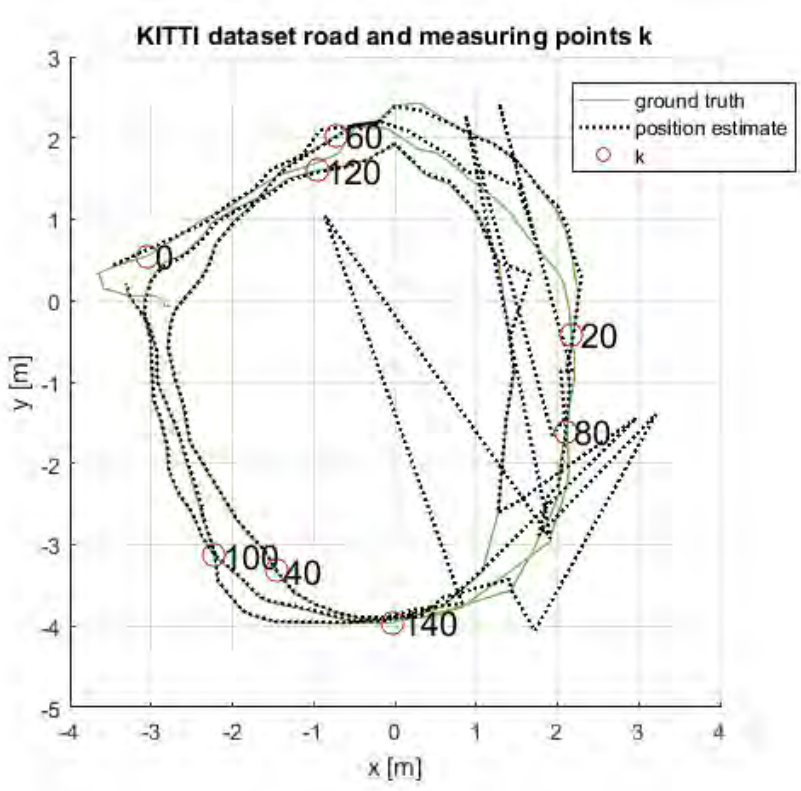


Figure 7.8: Simulation with AR.drone: the position estimate according to the VSLAM algorithm.

The results of this simulation are plotted in the same graphs as was done for the KITTI dataset. Figure 7.8 provides the ground truth and position estimates. The estimates show a relative good result on the left side, and all the biggest errors are located on the lower right. When this is compared to Figure 7.10, some conclusions can be made on the reason. In the first part of Figure 7.10 the error is relatively big, up to $k = 40$, which is roughly one complete cycle. The error here is big because the drone has a low sampling rate. The environment changes quickly and it needs some time to build a map before it can accurately create a map. After $k = 40$ the error is low because the algorithm recognises the different landmarks again. At $k = 80$ to $k = 100$ the error increases several orders of magnitude. This is a result of the orientation. In mentioned instances the orientation is different compared to the previous cycle and result in a different image, and thus different features. This reasoning is backed by Figure 7.9, herein the number of matches with both features and landmarks is low at the instances.

Noteworthy is that between $k = 0$ and $k = 40$ the number of landmark matches is extremely low, this is a result of the map being made during the first loop. Then in the second loop ($k = 55$ to $k = 115$) and third loop ($k = 115$ to $k = 161$) the number of matches clearly increases from 5 in the first loop, 45 the second loop, and 80 the third loop. From this it can be concluded that map building is the limiting factor when using a low sampling rate. In the case of a low sampling rate the algorithm should first be given time to build a map of the environment, this can be done by, for instance, flying a very slow circle while turning 360^o. By doing so a map can be made of the environment and when the drone is flying at higher speeds, the algorithm can still locate its position as long as the images remain clear of blur effects. By doing so the algorithm is in essence not dependent on the previous sampled state (although the observer's position estimate still is).

The statistical distributions of the number of matches and the correlation between the number of matches and the error, the same results are seen as in Figure 7.7. The largest errors are removed when the instances with one or two matches are taken out of consideration. The mean absolute error is more or less the same as in the tests with the KITTI dataset. The clear Rayleigh distribution that is visible in Figure 7.7 is not seen here, since the number of measurements is too low.

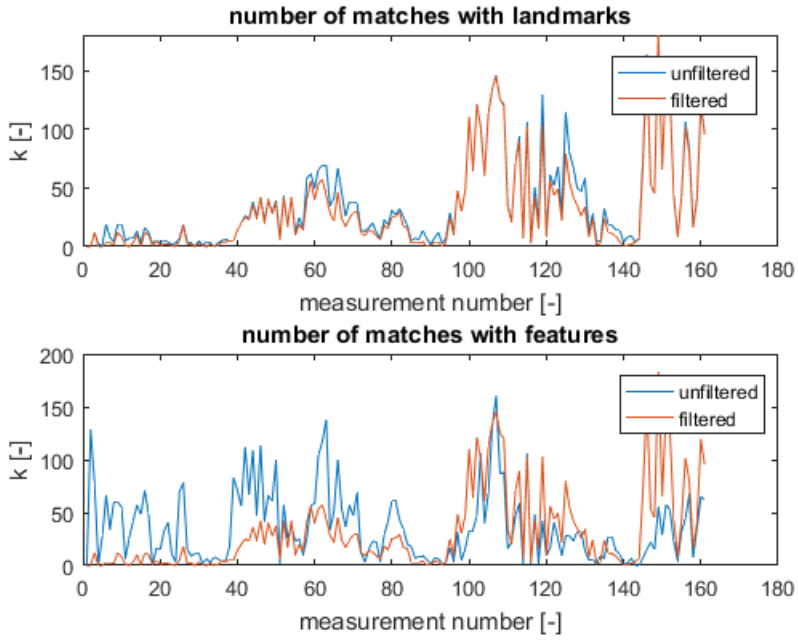


Figure 7.9: The number of filtered and unfiltered matches with landmarks and features for each instance k.

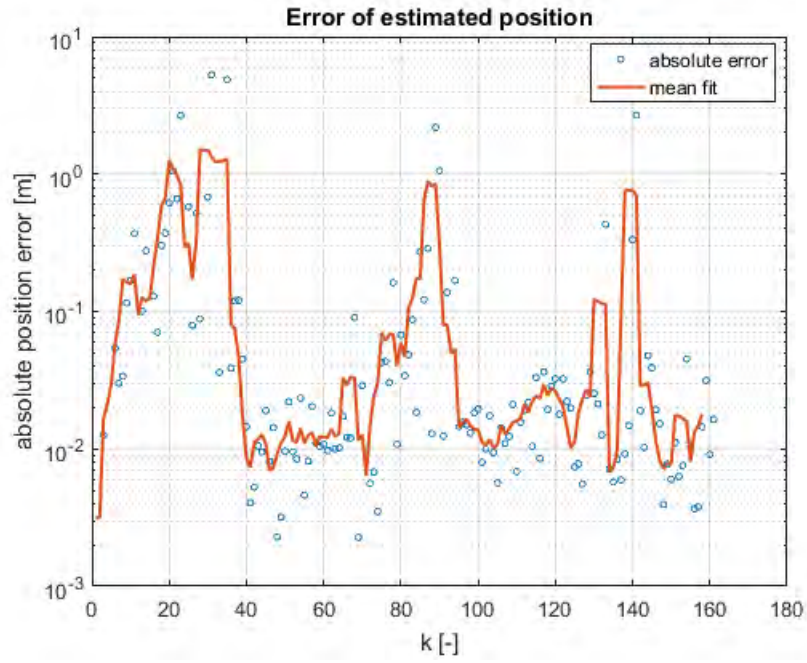


Figure 7.10: The absolute error of all instances k in logarithmic scale. The thick red line is the mean error of 3 combined instances.

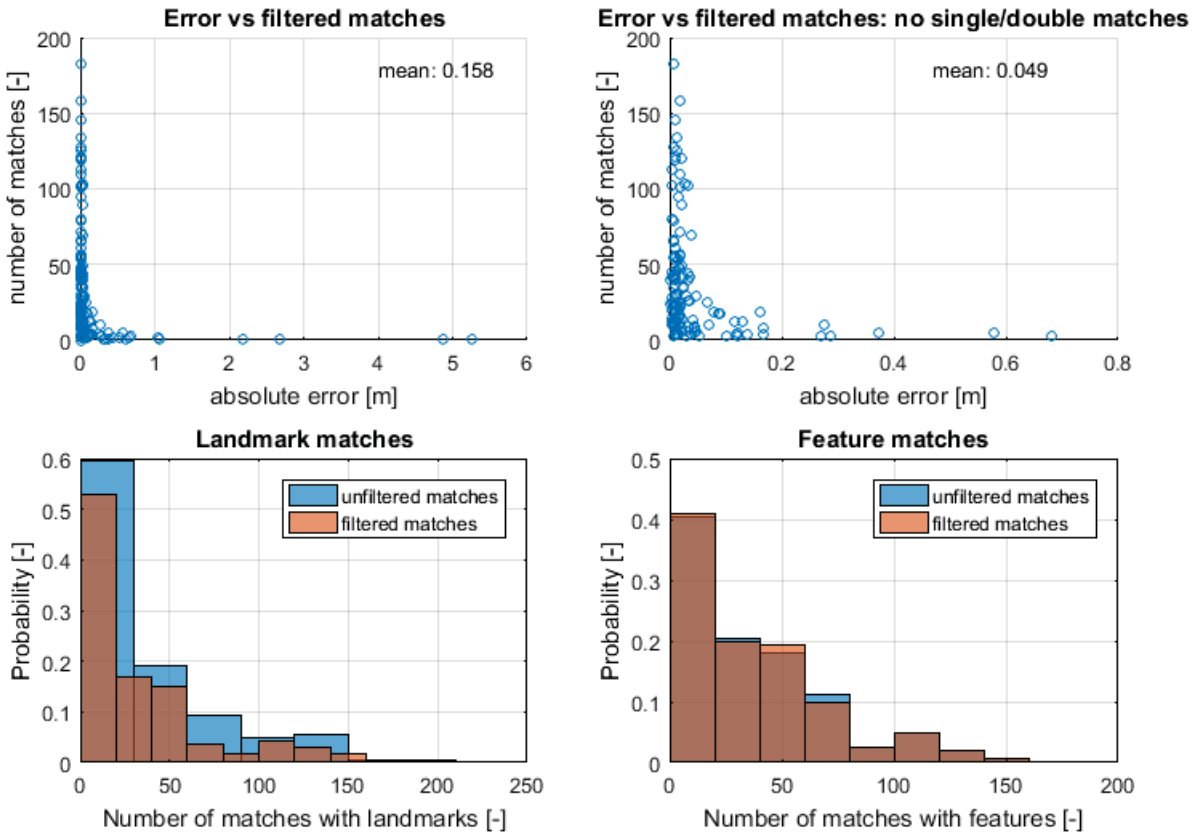


Figure 7.11: Simulation with AR.drone: correlation between matches and absolute error.

7.3 Concluding simulations

The algorithm is tested with the KITTI dataset [18] and a dataset made with the AR.drone. The KITTI dataset uses a car that drives around in an urban area. This dataset has the benefit of having a large data set and having a highly accurate ground truth for verifications. The dataset of the drone has a low sample rate and lower accuracy in position and orientation. In addition, because of unusable data, this dataset is tested in 2-D. The simulations were conducted to verify the accuracy and behaviour of the algorithm.

From the KITTI dataset it can be concluded that the algorithm works relatively well with an ever changing environment and becomes highly accurate (10× more than average) when the car drives through the same street again. The highest errors in the position estimate are a result of mismatches that are not compromised with good data because there are only one or two matches.

By simulating the algorithm with the dataset from the drone it becomes clear that the low sample rate interferes with the building of the map. There is some time needed for the map to be created but after this happens the accuracy greatly increased.

Overall can be concluded that the VSLAM algorithm can successfully map the environment and can derive the drone's/car's position based on the map. When the algorithm is implemented on the drone it is recommended to have the algorithm first examine the surroundings, this way a map can be initialised and the drone will be able to track it's position without the initial high errors.

Chapter 8

Conclusions and recommendations

The position drift of the drone was suppressed by the use of an external camera and pc. This external camera and pc are replaced by a drift compensation done solely with instruments onboard the AR.drone. Using a camera onboard the drone there are two methods of visually compensating for drift: optical flow and VSLAM. From earlier research it is clear that optical flow is not suitable for the application. In contrast, VSLAM is suitable and has been applied for drift compensation on drones multiple times. Making it the best choice.

The goal is to derive a VSLAM algorithm that is capable of positioning the drone without the need of external equipment to assist the positioning. The focus on this VSLAM algorithm lies on the increase in computational speed so it can be used on small devices such as the drone. In order to achieve a fast algorithm, it is broken down into different steps and for each step the most efficient solution is chosen.

As a rough description, VSLAM works as follows: a VSLAM algortihm builds a map of the environment, the map consist of landmarks. The position of the drone would be computed from landmarks it detects with the camera. When determining the position of the drone (localisation) from its camera images, one needs to know the position of the landmarks it detects. Landmarks are points in 3-D world space. The landmarks form a 3-D map (mapping) from which the drone can localise itself. This is in short what a VSLAM system does. Note that VSLAM is a specific type of SLAM, Simultaneous Localisation and Mapping, using camera images to find landmarks, while a SLAM system can use any means to find landmarks.

In order to do all this, VSLAM needs to detect features in an image, recognise them in other images, and compute their 3-D position. The collection of features with 3-D position, called landmarks, are stored in the map. With this map the algorithm computes the position of the drone. All these steps are created separately.

For detecting features the FAST detector [42] was chosen to be the best solution. It is by far the fastest detector. In addition the speed of this detector has been increased in two ways. Traditional FAST tests pixels by testing the surrounding pixels in a circular motion. By changing the order of the tests, the speed is greatly increased. The order is changed so that the most informative of the surrounding pixels is tested next. In addition, traditional FAST tests every pixel of an image in search for features. By testing only a fraction of the pixels and applying a local search in the non-maximal suppression (a filter applied after FAST) the FAST algorithm can detect features more efficiently, resulting in more feature detections per second. From tests it is proven that if, for example only half of all pixels are tested, 75% of all features is found in roughly the same time as traditional FAST would need to detect 50% of all features.

Recognising features is done with a *descriptor*. A descriptor describes the feature and/or its surrounds so it can be recognised again later. The FREAK descriptor [8] is chosen to use for the VSLAM algorithm. This choice is based on research of various sources as it is the fastest binary descriptor and has roughly the same robustness to blur, image rotation and scale as BRISK, which is just slightly slower than FREAK. FREAK is a binary descriptor, it stores the descriptor in a way that it can be bit-wise compared to other FREAK descriptors. The Hamming distance then represents the error between the two compared descriptors. Overall, this way of matching is very

efficient. Although the matching itself is efficient, if done in larger quantities the computation time rises fast. An efficient matching strategy improves computation times greatly. The matcher that is used was created by Maju and Lowe [38] and focuses on a decrease of computations for large matching sets.

Some of the matches between the detected features and the map are false positives. With the landmark positions it is possible to estimate if a matched feature is false or not. By projecting the landmark onto the image the expected feature location in image space is made. If the location of the detected feature in the image plane is far from an expected location then the match is a false positive.

To create a map of the environment, the 2-D image position must be transformed to a 3-D world position. In order to do this the image must first be corrected for distortion effects. The image is mainly prone to lens distortion. By fitting a polynomial over a distorted image the shape of the distortion is discovered. By taking the inverse function of the polynomial the distortion can be corrected. Darp et al. [16] described an easy way of estimating the inverse of the polynomial.

With an undistorted image the directions of the feature originating from the camera can be computed. With this vector the position of matched features can be either computed or updated. The first time a feature is matched its position is computed with epipolar geometry. The other times the position of the landmark is only updated, requiring a different set of equations with a shorter computation time. In both cases the accuracy of the landmarks position is also stored in the form of a standard deviation.

The drone is localised with the use of one position estimate per landmark that assume that the orientation of the drone is correct. This assumption can be made because the orientation of the drone can be derived from the IMU drift free.

Using geometry the position of the drone is estimated for each matched landmark separately. By doing so the drone can estimate its position from even one landmark and the amount of computation is linear to amount of matched landmarks. All position estimates are combined with a weighted mean. The weights of this mean are the standard deviation of each position estimate, thereby ensuring that the most accurate estimate is most influential and the least accurate one is not. The weighted mean estimate is lastly combined with the observer's position estimate using a Kalman filter. The result is the final position estimate, along with its standard deviation, that is the output of the VSLAM algorithm. This data is send to the observer together with the the starting state and time when the image was taken.

The algorithm is tested with the KITTI dataset [18] and a dataset made with the AR.drone. Overall can be concluded that the VSLAM algorithm can successfully map the environment and can derive the drone's/car's position based on the map. The accuracy of the localisation increases as the drone/car passes by the same landmarks multiple times. The tests with the drone conclude that the low sample rate affects the building of the map. This means that it takes longer for a map to form and for the position estimate to become accurate.

In summary, this report proposes a fast VSLAM algorithm. The increase in computation speed is mainly achieved in the following steps. The detection of features is increased in speed by combining FAST and non-maximal suppression with a local minimum search. The features descriptor is chosen to be the fastest among binary descriptors: FREAK. The filtering of mismatches is reduced to simple equations, reducing their computation time. The same holds true for the position estimate of the drone: by using simple equations it is possible to reduce the computations from $m \times m$ for EKF-SLAM and $m \times n$ for FastSLAM to only m computations (here m stands for the number of landmarks detected and n the number of particles in FastSLAM).

8.1 Recommendations

The project to implement VSLAM on the AR.drone is not yet finished. Some simulations were run to give a proof of concept but the algorithm is not fully implemented into C++ and not all of the codes have been compiled to the drone yet. This is one of the main recommendations, but there needs to be some extra work done before the implementation can be completed. As a short list the recommendations are divided in:

1. **Creating an observer for the missing position estimates**
   The observer of the drone needs to compute the two missing positions x and y as accurately as possible. It is advised to use not only the IMU for this but also the drone dynamics. By using the controller output and drone dynamics to compute an expected position a more accurate position estimate can be reached. In addition, the VSLAM algorithm requires the accuracy of this position, this needs to be tested and implemented into the observer as well.

2. **Testing the algorithm for a new dataset with the new observer**
   It is advised to further test the VSLAM algorithm with data of the new observer before it is implemented and compiled for the AR.drone. This mainly prevents unnecessary set-backs.

3. **Implementation of the full algorithm in C++ and compilation of C++ for the AR.drone**
   Only a part of the VSLAM algorithm has been implemented in C++ and is not fully up-to-date. The algorithm must thus be fully written in C++. The compiler for the AR.drone is already made available, see Appendix A, but caution is advised with the compilations. The AR.drone does not always respond the same as a PC/laptop does. Especially with the OpenCV libraries.

4. **Implement the algorithm for the DSP**
   The hardest part of implementing the algorithm is compiling the algorithm for the drone's DSP chip. This is a requirement if the VSLAM algorithm is to run along side the controller. This is expected to be a major obstacle. The documentation of the DSP dates back to 2003 and most of the programs used in the documentation is not available anymore.

5. **Communication between controller and algorithm**
   The VSLAM algorithm and the controller, or rather the observer of the controller, will need to communicate with each other. The controller is currently made in Simulink and the VSLAM algorithm should be running on the drone's DSP. It is likely needed to convert the controller to C++ as well. C++ scripts should be able to share data using global variables.

6. **Improve the computation speed of the algorithm**
   The most computer intensive steps in the algorithm is the binary matcher. There are a few ways the computation way of these method can be improved. First of all the matcher considers every every possible combination. The features can however be filtered reducing the amount of possible matches. The map could, for instance, be split in sections each section would then correspond to a range of direction in which the stored descriptors were detected. The matcher than only has to check the section which corresponds to the direction the camera is pointing.

Much work is still needed for VSLAM to be operational on the AR.drone. As a last recommendation, it might be wise to implement the VSLAM algorithm on a drone with a stronger onboard computer, or add a Raspberry pi to the drone for extra power. After successfully flying with this, the next step could be made to implement the algorithms on the AR.drone. This suggestion holds also true because the non-linear controller running on the drone nearly overloads the CPU and has little room for extensions.

References

[1] Ar.drone parrot 2.0 power edition, www.parrot.com.

[2] A battle of three descriptors: Surf, freak and brisk, https://computer-vision-talks.com/2012-08-18-a-battle-of-three-descriptors-surf-freak-and-brisk/.

[3] Comparison of the opencv feature detection algorithms, https://computer-vision-talks.com/2011-01-04-comparison-of-the-opencv-feature-detection-algorithms/.

[4] Opencv (open source computer vision library), www.opencv.org.

[5] Vicon motion capture, www.vicon.com.

[6] M. Achtelik, M. Achtelik, S. Weiss, and R. Siegwart. Onboard IMU and monocular vision based control for MAVs in unknown in- and outdoor environments. *IEEE International Conference on Robotics and Automation*, pages 3056–3063, 2011.

[7] M. Agrawal, K. Konolige, and M. Blas. CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching. *Computer Vision ECCV 2008*, pages 102–115, 2008.

[8] A. Alahi, R. Ortiz, and P. Vandergheynst. FREAK: Fast Retina Keypoint. *IEEE International Conference on Computer Vision*, pages 510–517, 2012.

[9] T. Bailey, J. Nieto, J. Guivant, M. Stevens, and E. Nebot. Consistency of the EKF-SLAM Algorithm. pages 3562–3568, 2006.

[10] T. Bailey, J. Nieto, and E. Nebot. Consistency of the FastSLAM Algorithm. *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, (May):424–429, 2006.

[11] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *ECCV 2006, Part I. LNCS, vol. 3951*, pages 404–417, 2006.

[12] M. Bibuli, M. Caccia, and L. Lapierre. Path-following algorithms and experiments for an autonomous surface vehicle. *IFAC Proceedings Volumes (IFAC-PapersOnline)*, 7(Part 1):81–86, 2007.

[13] A. Briod, J. Zufferey, and D. Floreano. A method for ego-motion estimation in micro-hovering platforms flying in very cluttered environments. *Autonomous Robots*, 40(5):789–803, 2016.

[14] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF: Binary Robust Independent Elementary Features. *ECCV 2010, Part IV. LNCS, vol. 6314*, pages 778–792, 2010.

[15] A. Cooper. A comparison of data association techniques for simultaneous localization and mapping. *Technical report, Department of Aeronautics and Astronautics, Massachusetts Institute of technology*, 2005.

[16] P. Drap and J. Lefèvre. An Exact Formula for Calculating Inverse Radial Lens Distortions. *Sensors*, 16, 2016.

[17] J. Figat, T. Kornuta, and W. Kasprzak. Performance evaluation of binary descriptors of local features. *Computer Vision and Graphics*, pages 187–194, 2014.

[18] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets robotics : The KITTI dataset. *The International Journal of Robotics Research*, 2013.

[19] R. Holies and M. Fischler. A RANSAC-based approach to model fitting and its application to finding cylinders in range data. *IJCAI*, pages 637–643.

[20] G. Huang, A. Mourikis, and S. Roumeliotis. Analysis and improvement of the consistency of extended Kalman filter based SLAM. *2008 IEEE International Conference on Robotics and Automation*, pages 473–479, 2008.

[21] N. Jeurgens. Cascade based tracking control of quadrotors, DC 2017.013. *Eindhoven Univeristy of Technical, Dynamics and Control Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, MSc Thesis*, 2017.

[22] N. Jeurgens. Implementing a SIMULINK controller in an AR.drone 2.0. *Eindhoven Univeristy of Technical, Dynamics and Control Group, Department of Mechanical Engineering, Eindhoven, The Netherlands*, 2017.

[23] I. Kim, N. Kwak, H.l Lee, and B. Lee. Improved particle fusing geometric relation between particles in FastSLAM. *Robotica*, 27(2009):853–859, 2016.

[24] G. Klein and D. Murray. Parallel tracking and mapping for small AR workspaces. *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR*, 2007.

[25] T. Krajník, M. Nitsche, S. Pedre, L. Přeučil, and M. Mejail. A simple visual navigation system for an UAV. *International Multi-Conference on Systems, Signals and Devices, SSD 2012 - Summary Proceedings*, 2012.

[26] S. Lange, N. Sunderhauf, and P. Protzel. A vision based onboard approach for landing and position control of an autonomous multirotor UAV in GPS-denied environments. *Proceedings of the International Conference on Advanced Robotics*, pages 1–6, 2009.

[27] S. Leutenegger, M. Chli, and R. Siegwart. Brisk: Binary robust invariant scalable keypoints. *IEEE International Conference on Computer Vision*, pages 2548–2555, 2011.

[28] K. Li, F. He, and X. Chen. Real-time object tracking via compressive feature selection. *Frontiers of Computer Science*, 10(4):1–13, 2016.

[29] K. Li, K. Zhang, and B. Chen. On-board Visual Odometry and Autonomous Control of a Quadrotor Micro Aerial Vehicle. *12th IEEE International Conference on Control and Automation (ICCA)*, pages 68–73, 2016.

[30] X. Li, Y. Zhang, F. Yang, and B. Wu. A Rectification Strategy for State Estimation of Qudarotors Based on Parallel Tracking and Mapping. *Proceeding of the 2015 IEEE International Conference on Information and Automation*, (August):3046–3050, 2015.

[31] D. Liu, J. Duan, and H. Shi. A Strong Tracking Square Root Central Difference FastSLAM for Unmanned Intelligent Vehicle With Adaptive Partial Systematic Resampling. *IEEE Transaction on Intelligent Transportation Systems*, 17(11):3110–3120, 2016.

[32] F. Liu, Q. Lv, H. Lin, Y. Zhang, and K. Qi. An image registration algorithm based on FREAK-FAST for visual SLAM. *Proceedings of the 35th Chinese Control Conference*, (1):6222–6226, 2016.

[33] D. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision 60*, pages 91–110, 2004.

[34] M. Maju and Lowe. D. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. *International Conference on Computer Vision Theory and Applications*, 2009.

[35] S. Marx. High Performace Drone Target Pursuit, CST 2017.017. *Eindhoven Univeristy of Technical, Control Systems Technology Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, MSc Thesis*, 2017.

[36] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. *Image and Vision Computing (2004)*, 22:761–767, 2002.

[37] M. Mueller, M. Hamer, and R. Andrea. Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation. *International Conference on Robotics and Automation*, pages 1730–1736, 2015.

[38] M. Muja and D. Lowe. Fast Matching of Binary Features. *IEEE Conference on Computer and Robot Vision*, 2012.

[39] R. Mur-Artal and J. Tardós. ORB-SLAM2 : an Open-Source SLAM System for Monocular , Stereo and RGB-D Cameras. *IEEE Transactions on Robotics*, 33:1255–1262, 2016.

[40] A. Patel, D. Kasat, S. Jain, and Thakare V. Performance analysis of various feature detector and descriptor for real-time video based face tracking. *International Journal of Computer Applications*, 93:37–41, 2014.

[41] J. Quinlan. Induction of decision trees. *Machine learing*, 1:81–106, 1985.

[42] E. Rosten, R. Porter, and T. Drummond. Faster and Better : A Machine Learning Approach to Corner Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(1):105–119, 2010.

[43] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: an efficient alternative to sift or surf. *IEEE International Conference on Computer Vision*, pages 2564–2571, 2011.

[44] L. Santana, A. Brandao, M. Sarcinelli-Filho, and R. Carelli. A Trajectory Tracking and 3D Positioning Controller for the AR . Drone Quadrotor. *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 118–123, 2014.

[45] M. Saska, T. Krajnik, J. Faigl, V. Vonasek, and L. Preucil. Low cost MAV platform ARdrone in experimental verifications of methods for vision based autonomous navigation. *IEEE International Conference on Intelligent Robots and Systems*, pages 4808–4809, 2012.

[46] T. Schops, J. Enge, and D. Cremers. Semi-dense visual odometry for AR on a smartphone. *ISMAR 2014 IEEE International Symposium on Mixed and Augmented Reality — Science and Technology 2014, Proceedings*, pages 145–150, 2014.

[47] J. Shi and C. Tomasi. Good features to track. *Computer Vision and Pattern Recognition, Proceedings CVPR '94*, pages 593–600, Jun 1994.

[48] C. Teuliere, E. Marchand, and L. Eck. 3-D model-based tracking for UAV indoor localization. *IEEE Transactions on Cybernetics*, 45(5):869–879, 2015.

[49] L. Vacchetti, V. Lepetit, and P. Fua. Combining edge and texture information for real-time accurate 3D camera tracking. *ISMAR 2004: Proceedings of the Third IEEE and ACM International Symposium on Mixed and Augmented Reality*, (ISMAR):48–57, 2004.

[50] S. Van Den Eijnden. Identification and control implementation of an AR.Drone 2.0, DC 2017.012. *Eindhoven Univeristy of Technical, Dynamics and Control Group, Department of Mechanical Engineering, Eindhoven, The Netherlands, MSc Thesis*, 2017.

Appendix A

Compiling C++ to AR.drone

Compiling C++ requires a Linux operating system. During the compilations in this project the following operating system was used: Ubuntu 16.04.3.

The VSLAM algorithm requires the OpenCV toolbox. This toolbox can be download for free from their website [4]. The version that is required is 2.4.10. Assuming the VSLAM algorithm is written in C++, the follow instructions should provide a compiler that makes it possible to compile C++ to the drone as well as the OpenCV libraries.

`http://hassannadeem.com/blog/2015/05/01/how-to-run-opencv-project-ar-drone/`

A noteworthy addition to the instructions is that at some point it is required to select the different libraries of OpenCV that should be compiled and with what tools. In the instructions there is one point where the GUI of cmake is called with commando cmake-gui. The options that were selected in the GUI are seen in Figure A.1.

With the compiled libraries there is one more thing that needs to be done before they can be used. The compiled libraries need to be renamed. The OpenCV files with a name ending in **.so.2.4.10* must be named **.so.2.4* this replaces another file in the same folder. These renamed libraries can be used on the drone. In order to use them simply add the required #**include** of the opencv headers in the C++ codes of the algorithm.

It might also be possible to create a soft-link to the files using the command *ln -s *.so.2.4.10 *.so.2.4*, with * the name of the libraries. This would be preferable because changing the names of libraries could lead to incorrect links within libraries. However, using soft-links has not been field tested.

After following all instructions of Nadeem and the above adjustments, one can compile a C++ code for the drone and run it on the drone. In order the following command are needed to do so. First move the folder contain the C++ codes that need to be compiled:

cd ./YourDirectory/

The compiling is done with the commando:

arm-none-linux-gnueabi-g++ -I$ARMPREFIX/include -I$ARMPREFIX/include/opencv -I$ARMPREFIX/include/opencv2/imgproc -I$ARMPREFIX/include/opencv2 -I /Desktop/VSLAM-Codes -L$ARMPREFIX/lib -g -o **testCam_AR getdataCPP.cpp ProposedFastCPP.cpp ProposedNonMax.cpp mergSort.cpp pixelTest.cpp videoMarx.cpp videoMarx.hpp VSLAMCPP.hpp** *$ARMPREFIX/include/opencv2/opencv.hpp -lopencv_core -lopencv_imgproc -lopencv_highgui -lopencv_flann -lopencv_features2d -lxvidcore -lx264 -lswscale -lavformat -lavutil -lswresample -lavcodec*

where the bold text starts with the name of the compiled binary (**testCam_AR**) followed by the names of the C++ codes and headers. Hereafter, the binary can be copied to the drone. This can be done in two ways. The first option is to use a usb-stick, simply copy the compiled filed to

the usb-stick and plug it in the drone. On a side note, the drone is sometimes not able to find the usb-stick, restarting the drone with the usb-stick in it should solve the problem.

The other method to copy the compiled file to the drone is by transferring over a WiFi connection. To do this first connect to the drone and then use the following command:

ftp 192.168.1.1 5551

The drone will ask for a ID, any name or simply pressing enter works. Then copy the file to the drone and exit the connection by typing:

put YourFile
exit

In order to use the binary file open a telnet connection.

telnet 192.168.1.1

The file that was copied to the drone with the WiFi connection is found in the update folder (*cd ./update*) and the USB-stick is located in /data/video/usb0/ (*cd /data/video/usb0/*).

The official controller of the drone as was made by the producers is automatically run when the drone is turn on. This controller should be stopped. To see the programs running on the drone type *top*. First stop the respawner (named: *bin/sh /bin/program.elf.respawner.sh -360p*) of the controller with:

kill xxxx

where *xxxx* should be the process number, PID. Then stop the controller (named: */bin/program.elf -360.slices*) with a forceful stop:

kill -9 xxxx

With *xxxx* the PID of the controller, the controller only has to be stopped every time the drone restarts.

Only two more things are needed before the compiled file can be run. First the drone needs to known the directory of the compiled OpenCV libraries. The command for this action is:

export LD_LIBRARY_PATH=/data/video/usb0/AR_drone/lib/

in case the libraries are located in folder */data/video/usb0/AR_drone/lib/*. This action needs to be repeated on every new telnet session. The last thing that is needed, is to give the compiled file execution rights. This is only necessary if the file was copied over a WiFi connection. The command to give a file execution rights is:

chmod +x YourFile

Finally, the file can be run by typing the file's name:

./YourFile

With these steps, a C++ file with OpenCV libraries can be run on the AR.drone.

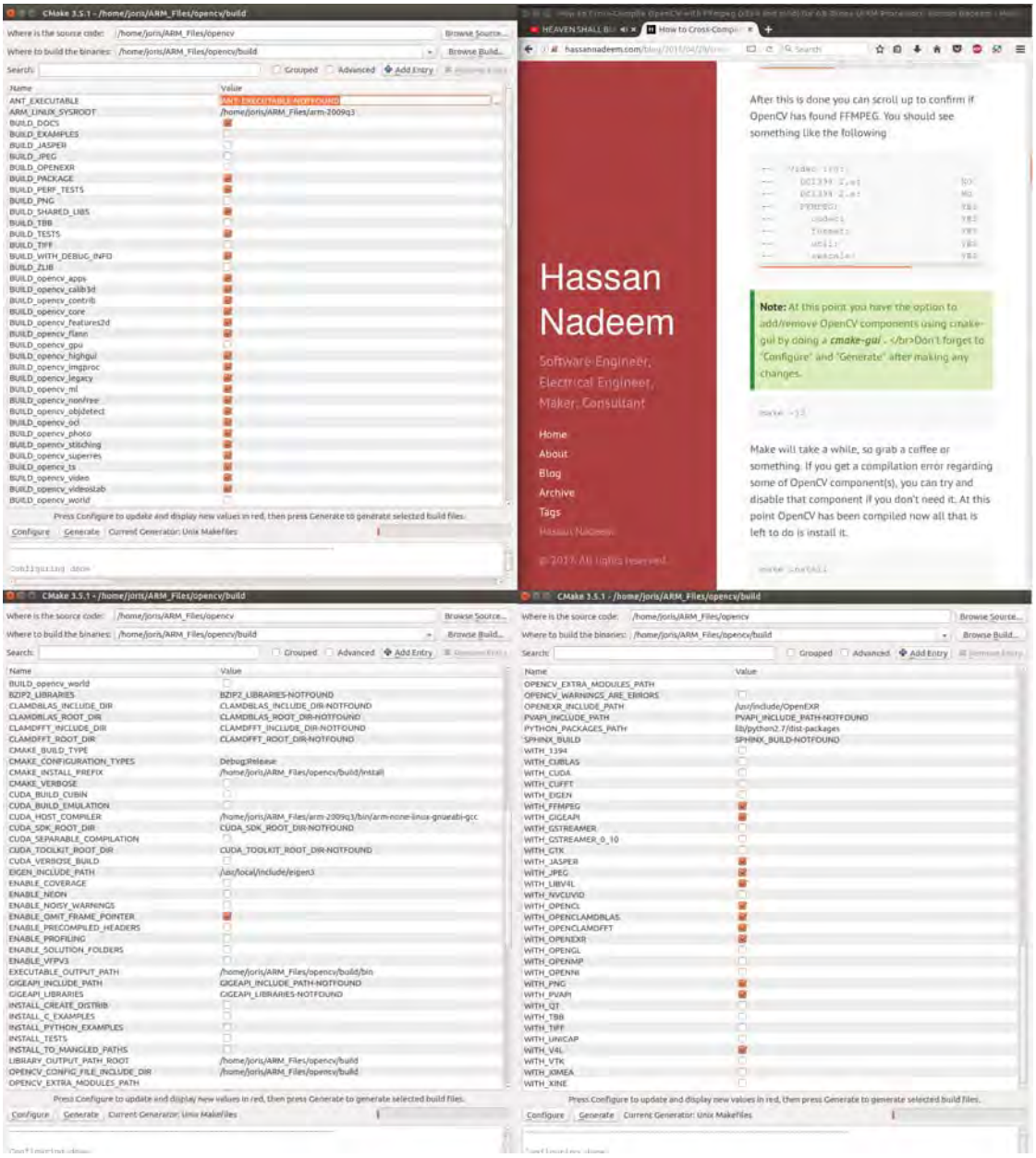


Figure A.1: Selected options in cmake-gui. The step in which the instruction require the cmake-gui is given in the top-left corner.

Appendix B

Parrot AR.drone2.0 specifications

A selection of the Parrot AR.drone2.0 specifications is listed below. First are the specification of the onboard computer system given, thereafter are the different motion sensors given. More specification are available on the official website [1].

Embedded computer systems

- CPU OMAP 3630 1GHz ARM cortex A8
    - Includes TI C64x DSP 800MHz
- DDR SDRAM 128MB
- NAND flash memory 128MB
- Wifi
- Linux OS

Sensors

- Ultrasound altimeter
    - 40kHz
    - 6m range
- Accelerometer
- Gyroscope
- Magnetometer
- Front camera
    - camera angle: 90°
    - 30 fps
    - 1280 × 720 pixels
- Vertical camera
    - camera angle: 64°
    - 60 fps
    - 320 × 240 pixels

Appendix C

Monte-Carlo simulation to determine the drone's position accuracy

The position of the drone is computed with a weighted mean. The appropriate accuracy of the resulting position is assumed to be normally distributed and is thus represented with a standard deviation. Computing the standard deviation of the localisation though the most common equation

$$\bar{\sigma}_k^2 = \sum_{i=1}^{N} \frac{w_i^2 \sigma_i^2}{W} \quad \text{with} \quad W = \sum_{i=1}^{N} w_i^2 \tag{C.1}$$

does not give satisfying results, as can be seen later on in this appendix. The weight factor $w_i = \frac{1}{\sigma_i}$ results in a formula that increases the deviation as the number of data increases, while the reverse is the truth. The resulting formula is equal to

$$\bar{\sigma}_k^2 = \frac{N}{\sum_{i=1}^{N} w_i}. \tag{C.2}$$

Because this formula is not satisfactory, the standard deviation is estimated with another weight factor w_i^* for equation (C.1).

The new weight is estimated with the use of a Monte-Carlo simulation. The Monte-Carlo simulation is the to much of a burden for the drone's small computer, therefore the equation and only the equation of the new weight is derived and implemented on the drone. The Monte-Carlo simulation involves running function an large number of times with a normal distributed input and creates a normal distribution over the output. This method is used to compute the true standard deviation σ_t for a fixed input. Then weights are chosen so that $\sigma_m \approx \sigma_t$.

The function for the simulation is (6.4). The inputs are various position estimates ${}^w\vec{p}_{k,i}$ with set σ_i. The output for every run is a the weighted position means ${}^w\vec{p}_m$. The standard deviation of a large set of this output is computed and approaches the σ_t for an increasing set size. The simulation is therefore run a large number of times to reduce the error of σ_t as much as possible. Thereafter, a function for the new weights w_i^* must be found: $w_i^* = f(\sigma_i)$.

The function $w_i^* = f(\sigma_i)$ is of the form $w_i^* = a\sigma_i^{-b}$, with constant values a and b. These constants are estimated using an iterative learning process. The learning process computes (6.5) with $w_i = w_i^*$, herein are a and b one by one introduced with an numerical error. If the resulting σ_m is closer to σ_t the erroneous value is kept else it is otherwise discarded. This process continues until the error between σ_m and σ_t is acceptable.

From the simulation and learning process the values $a = 0.2613$ and $b = 0.7642$ were found. These values are then verified with a second Monte-Carlo simulation. Here the error between $\bar{\sigma}$ and σ_t is given for different inputs ${}^w\vec{p}_{k,i}$ and σ_i with random values and increasing quantities i. The average error $\sigma_e = \sum^N ||\sigma_m - \sigma_t|| N^{-1}$ is equal to 0.0921 with its own standard deviation of 0.0464. In Figure C.1 the error is compared with the error of the conventional method. A big increase in precision is visible, the new weight improves the result ten-fold. The resulting equation

can still be much improved, another function might provide a better solution.

The result is a equation combined of C.1 and $w_i^* = 0.2613\sigma_i^{-0.7642}$. This equation is used in the VSLAM algorithm to derive the standard deviation of the weighted mean position of the drone.

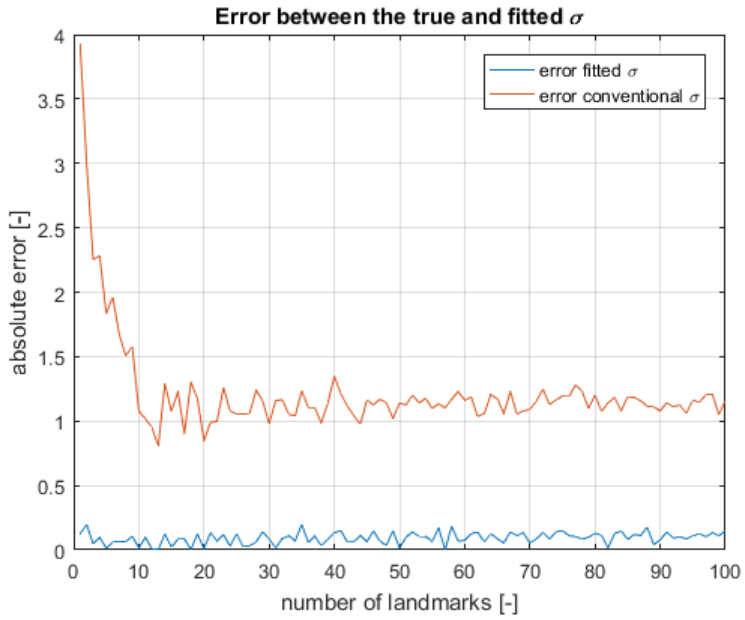


Figure C.1: The error in the fitted weighted mean $\bar{\sigma}$ and the error of the conventional weighted mean according to equation (6.5) with $w_i = \frac{1}{\sigma_i}$. The error is plotted against the number of landmarks that were used as an input.

Appendix D

Derivation of $\sigma_{l,k}$

The standard deviation of a landmark's position at instance k is given as the equation:

$$\sigma_{l,k} = \sigma_{p,k} + d\tan(\sigma_a) \quad \text{for small } \sigma_a \quad \sigma_{l,k} = \sigma_{p,k} + d\sigma_a. \tag{D.1}$$

This equation is chosen to be of this form. Before the reasoning for this choice is given, the deviation of the landmark position described.

The deviation is not normally distributed, see Figure D.1. From two positions ${}^w\vec{p}_k$ and ${}^w\vec{p}_{k-1}$ the landmark position ${}^w\vec{l}_i$ is computed using epipolar geometry. The positions and the directions from the positions to the landmark are normally distributed. The result is that the distribution of the landmark position is shaped like an ellipsoid. In the mentioned figure the elliptic shape of the distribution is visible.

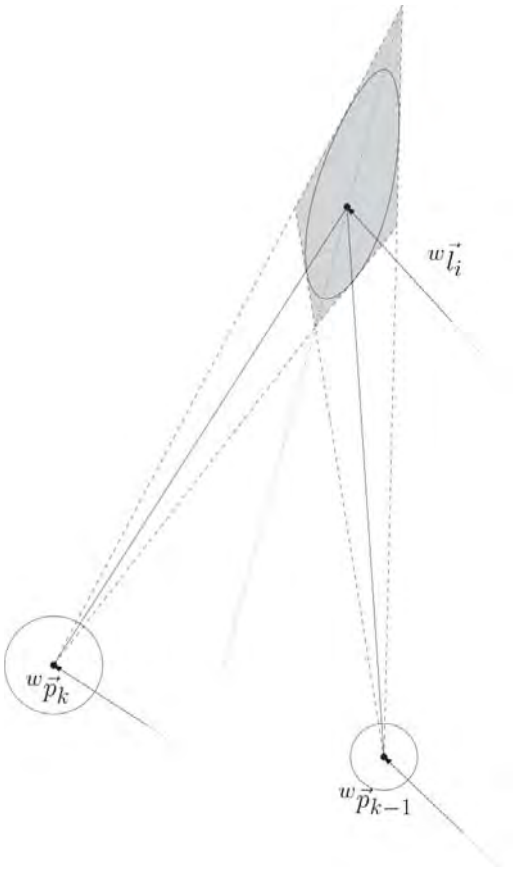


Figure D.1: The distribution of probability of the landmark takes the shape of an ellipse in 2-D. The solid lines represent the direction from positions ${}^w\vec{p}_k$ and ${}^w\vec{p}_{k-1}$ to the landmark ${}^w\vec{l}_i$. The deviation of the direction is visualised by the dashed lines and the deviation of the positions by the circles surrounding them.

The equations of the VSLAM algorithm require the deviation of the landmarks to be normally distributed, in which case the distribution should be shaped as a sphere. To discover the true distribution a Monte-Carlo simulation was used. The Monte-Carlo simulation involves running function a large number of times with a normal distributed input. The output can then be used to explore the effects of the distributed inputs with the distribution of the output. The function in the Monte-Carlo simulation use epipolar geometry. The variables ${}^w\vec{p}_k$, ${}^w\vec{p}_{k-1}$ and their appropriate directions to the landmark ${}^w\vec{s}_k$ and ${}^w\vec{s}_{k-1}$ are applied with a normal distribution. The standard deviation of ${}^w\vec{s}_k$, ${}^w\vec{s}_{k-1}$ apply to their angles, since the vector must be normalised. The values of the variables are shown in Table D.1. In this table R_θ stands for the rotation matrix over the z-axis with angle θ.

Table D.1: Values of variables in the Monte-Carlo simulation

Variable	Mean	Standard deviation
${}^w\vec{p}_k$	$[1,\ 0,\ 0]^T$ [m]	$\sigma_{p,k} = 0.01$ [m]
${}^w\vec{p}_{k-1}$	$[0,\ 0,\ 0]^T$ [m]	$\sigma_{p,k-1} = 0.01$ [m]
${}^w\vec{s}_k$	$R_\theta[0,\ 1,\ 0]^T$ [-]	$\sigma_{\theta,k} = 0.001$ [rad]
${}^w\vec{s}_{k-1}$	$R_\theta^T[0,\ 1,\ 0]^T$ [-]	$\sigma_{\theta,k-1} = 0.001$ [rad]

In Figure D.2 the results of the Monte-Carlo simulation are shown as the black cloud. The cloud of landmark positions shows the elliptic shape that was expected. The average standard deviation of all three axes x, y and z is given in the figure as the dotted sphere with radius $3 \times \bar{\sigma}_l$. There are some other normal distributions visualised in the plots as well. The smallest red sphere represents the normal distribution of the landmarks averaged only over the x and z-axis ($\bar{\sigma}_{l,xz}$). The green sphere is defined by the chosen equation: D.1.

In the mentioned figure are the landmarks more stretched as angle θ decreases, this is a result of σ_a. The normal distribution fits less as the landmarks become more stretched.

From here the reasoning for a well fitting normal distribution is discussed. Hypothetically, when a drone detects a landmark it is more likely to detect it again from roughly the same area. This is a result of the feature descriptor only working in that area; the descriptor depends on both the distance to a landmark and the angle under which it sees the landmark. Because of this the landmark is practically never detected outside a local area. The average standard deviation $\bar{\sigma}_l$ is not well suited to represent the distribution of the landmark's world position because the influence of σ_z is not that high in practice.

The drone detects landmarks in the 2-D image plane. The distribution of the landmark's position projected on the image plane at instance k (and $k-1$) is circular (normally distributed), as a result of the normally distributed ${}^w\vec{p}_k$ and ${}^w\vec{s}_k$, see Figure D.3. So the landmark's position seen from the drone is normally distributed. Since the drone can only detect the feature in roughly the same area, the projected landmark distribution is approximately the same as the distribution at instance k. The landmark's distribution can then be estimated with geometry resulting in D.1.

The equation D.1 has the property of being bounded when it is derived at instance k. Because σ_l comes from a projection of a 2-D plane, the distribution is bounded by the largest and smallest possible projection. From this follows that σ_l is bounded by $\bar{\sigma}_{l,xz}$ (the smallest projection) and $\bar{\sigma}_{l,yz}$ (the largest projection) with the assumption that $\sigma_y \geq \sigma_z \geq \sigma_x$.

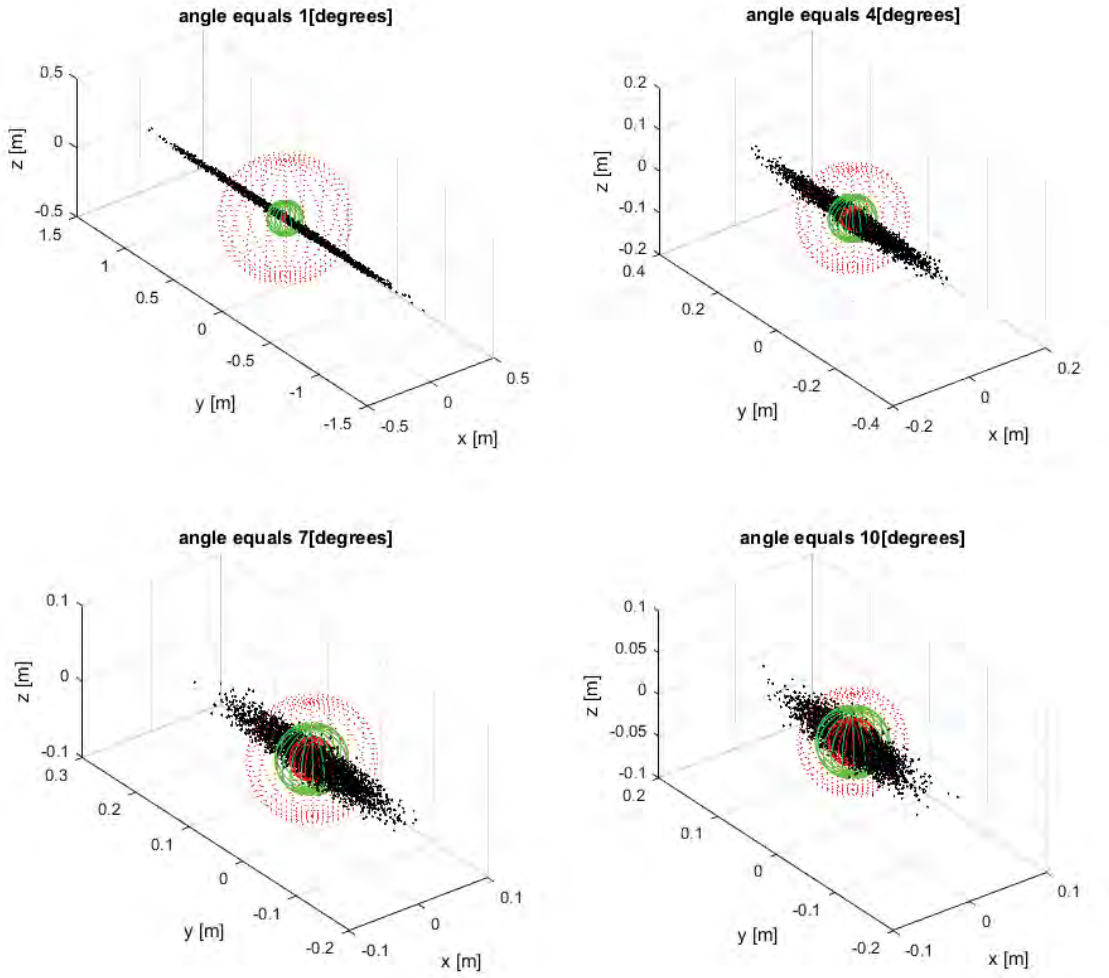


Figure D.2: Results from the Monte-Carlo simulation given for four different angles θ. The graphs show the landmark positions as a cloud of black dots. In addition, the spheres represent different different normal distribution for different standard deviations with a radius of $3 \times \sigma$. (Colour is required for these plots.)

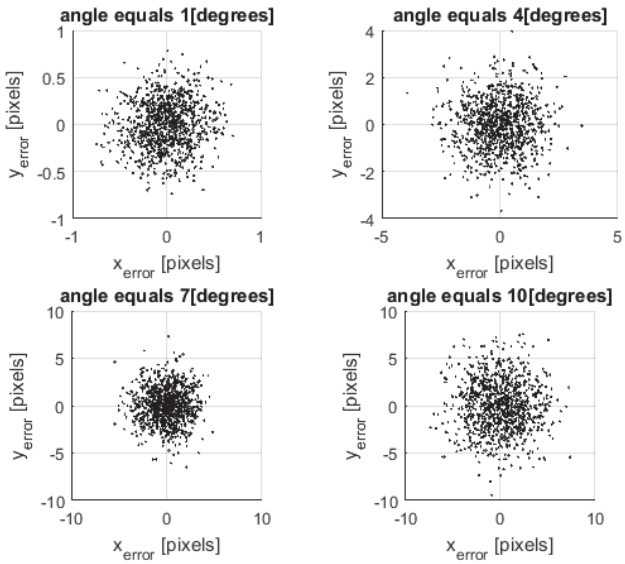


Figure D.3: Results from the Monte-Carlo simulation given for four different angles θ. The graphs show the error of the landmark positions projected onto the image plane. The considered image plane is perpendicular to the y-axis and with the focal point at position ${}^{w}\vec{p}_k$.

Appendix E

VSLAM algorithm C++ codes

Below are some of the codes and headers listed that are part of the VSLAM algorithm. A list the codes is given first on the next page. After that, the codes are given one by one. Each code has a short description of what it is and does in the first comment in the code. The file name of the codes is written down in the caption above the respective codes.

The codes were made in Microsoft Visual Studio 2017. Note that when the codes are copied from here that the symbols and lines remain correct and that the files are placed in the same folder. Additionally, when the codes are used, make sure the link to the OpenCV libraries is found. The libraries for OpenCV can be downloaded and installed from the website [4].

Listings

Listing E.1: ProposedFAST.cpp

```
// FAST corner detection for high speed computations
#include <cstdlib>
#include "opencv2/opencv.hpp"
#include "VSLAM.hpp"

numberOfCorners ProposedFast(Mat imageG, int numberOfLoops, ...
        ... int* listOfCorners) {

const int channels = imageG.channels();
const char* imData = imageG.data;
const int width = imageG.cols;
const int height = imageG.rows;
const int widthSt = width * channels;

const int borders = 4, nSubPix = 3, initialSize=512, incrementSize=128;
int rsize = initialSize;

long pixel[16];
    pixel[0]  = (-3 * channels) + ( 0 * widthSt);
    pixel[1]  = ( 3 * channels) + ( 0 * widthSt);
    pixel[2]  = ( 0 * channels) + ( 3 * widthSt);
    pixel[3]  = ( 0 * channels) + (-3 * widthSt);
    pixel[4]  = (-2 * channels) + ( 2 * widthSt);
    pixel[5]  = ( 2 * channels) + (-2 * widthSt);
    pixel[6]  = ( 2 * channels) + ( 2 * widthSt);
    pixel[7]  = (-2 * channels) + (-2 * widthSt);
    pixel[8]  = (-3 * channels) + ( 1 * widthSt);
    pixel[9]  = (-3 * channels) + (-1 * widthSt);
    pixel[10] = ( 1 * channels) + ( 3 * widthSt);
```

```
30      pixel[11] = (-1 * channels) + (-3 * widthSt);
31      pixel[12] = (-1 * channels) + ( 3 * widthSt);
32      pixel[13] = ( 1 * channels) + (-3 * widthSt);
33      pixel[14] = ( 3 * channels) + ( 1 * widthSt);
34      pixel[15] = (-3 * channels) + (-1 * widthSt);
35  int pixelOrder[16];
36      pixelOrder[0]  =  1;
37      pixelOrder[1]  =  9;
38      pixelOrder[2]  =  5;
39      pixelOrder[3]  = 13;
40      pixelOrder[4]  =  3;
41      pixelOrder[5]  = 11;
42      pixelOrder[6]  =  7;
43      pixelOrder[7]  = 15;
44      pixelOrder[8]  =  2;
45      pixelOrder[9]  = 10;
46      pixelOrder[10] =  6;
47      pixelOrder[11] = 14;
48      pixelOrder[12] =  4;
49      pixelOrder[13] = 12;
50      pixelOrder[14] =  8;
51      pixelOrder[15] = 16;
52  int startvecx[16];
53      startvecx[0]  = 4;
54      startvecx[1]  = 6;
55      startvecx[2]  = 6;
56      startvecx[3]  = 4;
57      startvecx[4]  = 5;
58      startvecx[5]  = 7;
59      startvecx[6]  = 7;
60      startvecx[7]  = 5;
61      startvecx[8]  = 5;
62      startvecx[9]  = 7;
63      startvecx[10] = 7;
64      startvecx[11] = 5;
65      startvecx[12] = 4;
66      startvecx[13] = 6;
67      startvecx[14] = 6;
68      startvecx[15] = 4;
69  int startvecy[16];
70      startvecy[0] =  4;
71      startvecy[1] =  6;
72      startvecy[2] =  4;
73      startvecy[3] =  6;
74      startvecy[4] =  5;
75      startvecy[5] =  7;
76      startvecy[6] =  5;
77      startvecy[7] =  7;
78      startvecy[8] =  4;
79      startvecy[9] =  6;
80      startvecy[10] = 4;
81      startvecy[11] = 6;
82      startvecy[12] = 5;
83      startvecy[13] = 7;
84      startvecy[14] = 5;
85      startvecy[15] = 7;
86  int threshold = 35;
```

```
87 int nCorners = 0;
88 long ii;
89 unsigned char n;
90 int p_Center;
91 int delta_p_test;
92 unsigned char stop;
93 unsigned char c_lighter;
94 unsigned char c_darker;
95 int nmin;
96 int nmax;
97 int firstN;
98 int pID;
99
100 for ( int loopCount = 0; loopCount < numberOfLoops; loopCount++ ) {
101
102     for (int x = startvecx[loopCount]; x < (width - borders); ...
103             ... x = (x + 4)) {
104         for (int y = startvecy[loopCount]; y < (height - borders); ...
105                 ... y = (y + 4)) {
106
107             ii = (int)((widthSt * y) + (x*channels));
108
109             p_Center = (int)imData[ii];
110             stop = 1;
111             c_lighter = 0;
112             c_darker = 0;
113             nmin = 9;
114             nmax = 9;
115             firstN = 17;
116
117             // Initial test
118             for (n = 4; n < 8; n++) {
119                 delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
120                 if (delta_p_test > threshold) {
121                     c_lighter++;
122                 }
123                 else if (delta_p_test < -threshold) {
124                     c_darker++;
125                 }
126             }
127             // In case the initial test was positive
128             if (c_lighter >= 3) {
129                 stop = 0;
130                 n = 0;
131                 while ((stop == 0) && (n < 16)) {
132                     delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
133                     if (!(delta_p_test > threshold)) {
134                         if (firstN == 17) {
135                             firstN = 9 - pixelOrder[n];
136                         }
137                         pID = pixelOrder[n] + firstN;
138                         if (pID > 16) {
139                             pID -= 16;
140                         }
141                         else if (pID < 0) {
142                             pID += 16;
143                         }
```

```
144                         if (pID < nmin) {
145                             nmin = pID;
146                         }
147                         else if (pID > nmax) {
148                             nmax = pID;
149                         }
150                         if (nSubPix < (nmax - nmin)) {
151                             stop = 1;
152                         }
153                     }
154                     n++;
155                 }
156             }
157             if (c_darker >= 3) {
158                 stop = 0;
159                 n = 0;
160                 while ((stop == 0) && (n < 16)) {
161                     delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
162                     if (!(delta_p_test < -threshold)) {
163                         if (firstN == 17) {
164                             firstN = 9 - pixelOrder[n];
165                         }
166                         pID = pixelOrder[n] + firstN;
167                         if (pID > 16) {
168                             pID -= 16;
169                         }
170                         else if (pID < 0) {
171                             pID += 16;
172                         }
173                         if (pID < nmin) {
174                             nmin = pID;
175                         }
176                         else if (pID > nmax) {
177                             nmax = pID;
178                         }
179                         if (nSubPix < (nmax - nmin)) {
180                             stop = 1;
181                         }
182                     }
183                     n++;
184                 }
185             }
186             if (stop == 0) {
187                 if (nCorners > rsize) {
188                     rsize += incrementSize;
189                     listOfCorners = (int*) realloc(listOfCorners, ...
190                             ... rsize * sizeof(0));
191                 }
192                 listOfCorners[nCorners] = ii;
193                 nCorners++;
194             }
195         }
196     }
197 }
198 return nCorners;
199 }
```

Listing E.2: mergeSort.cpp

```
//Merge sort rearranges an array from small to large values
//based on:
//https://www.tutorialspoint.com/ ...
//       ... data_structures_algorithms/merge_sort_program_in_c.htm
#include <stdio.h>
#include "VSLAMCPP.h"

void merging(int low, int mid, int high, int* a, int* b) {
   int l1, l2, i;

   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
      if(a[l1] <= a[l2])
         b[i] = a[l1++];
      else
         b[i] = a[l2++];
   }

   while(l1 <= mid)
      b[i++] = a[l1++];

   while(l2 <= high)
      b[i++] = a[l2++];

   for(i = low; i <= high; i++)
      a[i] = b[i];
}

void sort(int low, int high, int* a, int* b) {
    int mid;

    if (low < high) {
        mid = (low + high) / 2;
        sort(low, mid, a, b);
        sort(mid + 1, high, a, b);
        merging(low, mid, high, a, b);
    }
    else {
        return;
    }
}

void mergeSort(int* a, int rsize) {

    int* b = new int[rsize];
    sort(0, rsize, a, b);

    delete[] b;
}
```

Listing E.3: pixelTest.cpp

```
// Test a pixel for the minimum threshold
#include <stdlib.h>
#include <cstdlib>
#include "opencv2/opencv.hpp"
#include "VSLAMCPP.h"
```

```
6
7  value pixelTest(cv::Mat* imageG, int ii) {
8
9      const int channels = imageG->channels();
10     uint8_t* imData = imageG->data;
11     //const int width = imageG->cols;
12     const int widthSt = imageG->cols * channels;
13
14     const int borders = 4, nSubPix = 3;
15     int pixel[16];
16     pixel[0] = (-3 * channels) + (0 * widthSt);
17     pixel[1] = (3 * channels) + (0 * widthSt);
18     pixel[2] = (0 * channels) + (3 * widthSt);
19     pixel[3] = (0 * channels) + (-3 * widthSt);
20     pixel[4] = (-2 * channels) + (2 * widthSt);
21     pixel[5] = (2 * channels) + (-2 * widthSt);
22     pixel[6] = (2 * channels) + (2 * widthSt);
23     pixel[7] = (-2 * channels) + (-2 * widthSt);
24     pixel[8] = (-3 * channels) + (1 * widthSt);
25     pixel[9] = (-3 * channels) + (-1 * widthSt);
26     pixel[10] = (1 * channels) + (3 * widthSt);
27     pixel[11] = (-1 * channels) + (-3 * widthSt);
28     pixel[12] = (-1 * channels) + (3 * widthSt);
29     pixel[13] = (1 * channels) + (-3 * widthSt);
30     pixel[14] = (3 * channels) + (1 * widthSt);
31     pixel[15] = (-3 * channels) + (-1 * widthSt);
32     int pixelOrder[16];
33     pixelOrder[0] = 1;
34     pixelOrder[1] = 9;
35     pixelOrder[2] = 5;
36     pixelOrder[3] = 13;
37     pixelOrder[4] = 3;
38     pixelOrder[5] = 11;
39     pixelOrder[6] = 7;
40     pixelOrder[7] = 15;
41     pixelOrder[8] = 2;
42     pixelOrder[9] = 10;
43     pixelOrder[10] = 6;
44     pixelOrder[11] = 14;
45     pixelOrder[12] = 4;
46     pixelOrder[13] = 12;
47     pixelOrder[14] = 8;
48     pixelOrder[15] = 16;
49     int threshold = 35;
50     unsigned char n;
51     int p_Center;
52     int delta_p_test;
53     unsigned char stop;
54     unsigned char c_lighter;
55     unsigned char c_darker;
56     int nmin;
57     int nmax;
58     int firstN;
59     int pID;
60
61     int valueTemp = 255;
62     int savedValue = 255;
```

```
63
64      p_Center = (int)imData[ii];
65      stop = 1;
66      c_lighter = 0;
67      c_darker = 0;
68      nmin = 9;
69      nmax = 9;
70      firstN = 17;
71
72      // Initial test
73      for (n = 4; n < 8; n++) {
74          delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
75          if (delta_p_test > threshold) {
76              c_lighter++;
77          }
78          else if (delta_p_test < -threshold) {
79              c_darker++;
80          }
81      }
82      // In case the initial test was positive
83      if (c_lighter >= 3) {
84          stop = 0;
85          n = 0;
86          while ((stop == 0) && (n < 16)) {
87              delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
88              if (!(delta_p_test > threshold)) {
89                  if (firstN == 17) {
90                      firstN = 9 - pixelOrder[n];
91                  }
92                  pID = pixelOrder[n] + firstN;
93                  if (pID > 16) {
94                      pID -= 16;
95                  }
96                  else if (pID < 0) {
97                      pID += 16;
98                  }
99                  if (pID < nmin) {
100                     nmin = pID;
101                 }
102                 else if (pID > nmax) {
103                     nmax = pID;
104                 }
105                 if (nSubPix < (nmax - nmin)) {
106                     stop = 1;
107                 }
108             }
109             else if (savedValue > abs(delta_p_test)) {
110                 savedValue = abs(delta_p_test);
111             }
112             n++;
113         }
114     }
115     if (c_darker >= 3) {
116         stop = 0;
117         n = 0;
118         while ((stop == 0) && (n < 16)) {
119             delta_p_test = p_Center - (int)imData[(ii + pixel[n])];
```

```
            if (!(delta_p_test < -threshold)) {
                if (firstN == 17) {
                    firstN = 9 - pixelOrder[n];
                }
                pID = pixelOrder[n] + firstN;
                if (pID > 16) {
                    pID -= 16;
                }
                else if (pID < 0) {
                    pID += 16;
                }
                if (pID < nmin) {
                    nmin = pID;
                }
                else if (pID > nmax) {
                    nmax = pID;
                }
                if (nSubPix < (nmax - nmin)) {
                    stop = 1;
                }
            }
            else if (savedValue > abs(delta_p_test)) {
                savedValue = abs(delta_p_test);
            }
            n++;
        }
    }
    if (stop == 0) {
        valueTemp = savedValue;
    }
    return valueTemp;
}
```

Listing E.4: ProposedNonMax.cpp

```
// Nonmaximal suppression
#include <stdlib.h>
#include "opencv2/opencv.hpp"
#include "VSLAMCPP.h"


// rewrite removeDoubles so it also sees triples ed
numberOfCorners removeDoubles(int *listOfCorners, int nCandidates, ...
        ... int rsize) {
    int j = 0;
    int rsizeMin = rsize - 1;
    for (int i = (rsize - nCandidates); i<(rsizeMin); i++) {
        if (!(listOfCorners[i] == listOfCorners[i + 1])) {
            listOfCorners[j] = listOfCorners[i];
            j++;
        }
    }
    listOfCorners[j] = listOfCorners[rsize];
    j++;
    return j;
}

```

```
numberOfCorners ProposedNonMax(cv::Mat* imageG, int* listOfCorners, ...
    ... int rsize, int numberOfCandidates) {

    const int channels = imageG->channels();
    uint8_t* imData = imageG->data;
    const int width = imageG->cols;
    const int height = imageG->rows;
    const int widthSt = width * channels;

    int pixel[9];
        pixel[0] = 0;
        pixel[1] = ( 1 * channels) + (-1 * widthSt);
        pixel[2] = ( 1 * channels) + ( 0 * widthSt);
        pixel[3] = ( 1 * channels) + ( 1 * widthSt);
        pixel[4] = ( 0 * channels) + ( 1 * widthSt);
        pixel[5] = (-1 * channels) + ( 1 * widthSt);
        pixel[6] = (-1 * channels) + ( 0 * widthSt);
        pixel[7] = (-1 * channels) + (-1 * widthSt);
        pixel[8] = ( 0 * channels) + (-1 * widthSt);

    int n;
    int savedValue;
    int value;
    int cornerii;
    int nn;
    int ii;

    for (n = 0 ; n < numberOfCandidates ; n++) {
        centerPixel = listOfCorners[n];
        while (!(centerPixelOld == centerPixel))
            savedValue = 255;
            for (nn = 0 ; nn < 9 ; nn++){
                ii = centerPixel + pixel[nn];
                value = pixelTest(imageG, ii);
                if ( value < savedValue ) {
                    savedValue = value;
                    cornerii = ii;
                }
            }
            centerPixelOld = centerPixel;
            centerPixel = cornerii;
        }
        listOfCorners[n] = cornerii;
    }

    // order vector
    MergeSort(listOfCorners, rsize);

    // remove doubles and compress
    int nCorners = removeDoubles(listOfCorners, numberOfCandidates, rsize);

    return nCorners;
}
```

Listing E.5: VSLAMCPP.hpp

```
#ifndef VSLAM_H
#define VSLAM_H

#include "opencv2/opencv.hpp"

//declare maps
typedef struct {
    std::vector< float > sigma;
    std::vector< int > count; //times spotted
    std::vector< int > frameIndex; //last time spotted
    std::vector< cv::Mat > position;
    cv::Mat descriptor; //rows are the binairy descriptors, ...
                                        ... cols are the landmarks
}mapStruct;

typedef struct {
    std::vector< float > sigma;
    std::vector< cv::Mat > dronePosition;
    std::vector< cv::Mat > direction;
    cv::Mat descriptor;// CV_8UC1 IPL_DEPTH_1U
}mapCanStruct;


typedef int numberOfCandidates;
typedef int numberOfCorners;
typedef int value;
typedef cv::Mat positionVec;


numberOfCandidates ProposedFastCPP(cv::Mat* imageG, int numberOfLoops, ...
        ... int** listOfCorners, int* rsize);
numberOfCorners ProposedNonMax(cv::Mat* imageG, int** listOfCorners, ...
        ... int rsize, int numberOfCandidates);
value pixelTest(cv::Mat* imageG, int ii);
void mergeSort(int* a, int rsize);
void matchingDescriptors(cv::Mat descriptors_1, cv::Mat descriptors_2, ...
        ... std::vector< cv::DMatch >* good_matches);
positionVec epipolarGeometry(cv::KeyPoint* keypointCoor, cv::Mat* Rd2g, ...
        ... cv::Mat* dronePositionIMU, cv::Mat* sVecOld, ...
        ... cv::Mat* dronePositionOld);
positionVec pixel2Direction(cv::KeyPoint* keypointCoor, cv::Mat* Rd2g);
positionVec updateLandmarkPosition(cv::KeyPoint* keypointCoor, ...
        ... cv::Mat* Rd2g, cv::Mat* dronePositionIMU, cv::Mat* lVecOld);
void cleanUpMap(mapStruct* theMap, int frameIndex, int maxCandidates, ...
        ... int *mapIndex);

#endif
```