Contents

Chapter 1

Introduction

1.1 A brief history of drones

UAVs (Unmanned Aerial Vehicles), more commonly called drones, are defined as flying vehicles without human operators on board. They can be remote-controlled, or controlled by embedded computers. The earliest recorded use of UAVs dates back to 1849, when Austria launched about 200 unmanned balloons armed with bombs against the city of Venice [26]. Due to unfavorable wind conditions, this attack failed, and the experiment was not repeated. The first functional UAVs were made towards the end of World War 1 and their use was, like the Austrian balloons, military. One example is the Kettering Bug (Figure 1.1), which was a torpedo with wings and a propeller developed by the US Army in 1918 [31].

Figure 1.1: The Kettering Bug (1918)

Throughout the 20th century, UAVs become more and more sophisticated, and were used more and more, but always for military purposes. In the more recent years, civilian UAVs have started to appear on the markets and their number quickly exceeded that of military UAVs. In February 2017, the FAA (Federal Aviation Administration) of the United States estimated that around 1.1 million units were in use in the US alone, and expected that number to rise to 3.55 million by 2021 [9]. These civilian drones are very different from military drones, in both their form and their function: civilian drones are usually smaller, and use rotors to take off vertically. They are used in a wide variety of applications.

1.2 Motivation

The ability to remote-control small and agile flying objects over large distances through the air, and to bring them to previously inaccessible locations, makes many new things possible. With the increasingly lower prices and better performances of civilian UAVs, people keep finding more and more uses for these high-tech gadgets. Some examples of these applications are: crop monitoring in agriculture [3], delivery of mail or parcels, construction [5], cinematography, entertainment, or search and rescue operations. In all these applications, the more autonomous a drone is, the more efficient it will be at its task. One of the main challenges to achieve autonomy is for an UAV to be able to correctly identify its surroundings, and localize itself within them. In outdoor environments, GPS systems allow UAVs to know their position with great accuracy, but this is not possible in GPS-denied environments, like inside buildings. The main subject of this thesis will be fully autonomous navigation by a quadcopter in a GPS-denied environment.

1.2.1 Ethical considerations

The new possibilities brought by drones also pose ethical questions about security and privacy. Even though this technology can improve people's quality of life, it also has the potential to diminish it. If drones start to be widely used commercially, we could reach a point where the sound nuisances that they cause seriously impacts people who live in densely populated areas. Moreover, they can make us feel less at home, knowing that we could be observed from the sky at any moment. For this reason, it is important to adopt strict regulations regarding the use of drones in public spaces. Fortunately, many countries are already adopting legislation in this direction.

1.3 Context

This thesis is part of a project at the UCL that spans over several years and several masters theses. This project was launched by professor Julien Hendrickx in the 2012-2013 academic year, and had as long-term goal to develop a program that would enable low-cost UAVs to navigate autonomously in indoor environments. This means creating a map of their environment, localizing themselves in this map, and avoiding obstacles during exploration, using only on-board sensors. Another goal is to allow several drones to collaborate to speed up exploration. Five masters theses have already been written on this subject, each taking the work of the previous a little further.

2012-2015: First three theses In each of the three academic years (2012-2013, 2013-2014, 2014-2015), a masters thesis on the subject of indoor navigation for autonomous low-cost drones was written. These masters theses formed the base of the future work. They implemented visual SLAM methods to allow drones to build a two-dimensional map based on keypoints (first red pucks, then visual landmarks that the drone detected from a textured scene), and to localize itself within this map. During this time, inter-drone communication was also established, and was used to allow a drone to communicate the location of a target to another drone.

2015-2016: Recent work Last year, two groups of students simultaneously wrote theses on this subject. Before doing so, they joined forces to re-implement what had been done previously, but using ROS, a toolkit to develop software for robots that would make many things simpler, and allow more flexibility (see section 1.5.2). The work of the first group of students allowed a drone to search and follow a mobile target, and call a second drone to continue this task when its battery was low.
The second group of students extended the SLAM algorithm to allow to use a 3D map to localize

the drone. Unfortunately, they did not implement triangulation, so instead of projecting points into 3D space, they made the assumption that all points were located on the ground when building the map. The end result was a drone capable of using a 3D map to localize itself, but unable to build one from its observations.

1.4 Objectives

For this thesis, the goal is to continue the work of last year's second group to allow true 3-D SLAM: to build a 3D map based on observations by the monocular camera. To achieve this goal we will first follow these intermediate objectives:

- Research the current state of the art for 3D keyframe-based monocular visual SLAM
- Implement a way to triangulate points based on multiple observations
- Implement a way to use new visual information to correct previous errors and build a consistent map

1.5 Ressources

1.5.1 Hardware

For the practical implementation of this project, a Parrot AR.Drone 2.0 was used. This quadrotor was commercialized in 2012 and is an updated version of the original AR.Drone that was launched in 2010. This drone was marketed as a high tech toy, and is designed to be controlled from a smartphone application (connected to the drone via Wi-Fi). A few augmented reality games are available for the AR.Drone, in which it can visually recognize some predefined tags using its cameras, and interact with objects or other drones with a tag. To encourage the creation of more games for their drones, Parrot has released an open SDK that allows to effectively reprogram the drones. This early release of an open SDK has made it quite popular in the scientific community to do research on autonomous flight. The drone consists of 4 rotors, each with their own electric motor and microcontroller, an internal computer with a 1GHz ARM Cortex A8 processor and 1GB DDR2 RAM at 200MHz, and various sensors.

Sensors

The AR.drone has the following sensors:

- 3 axis accelerometer with $\pm$ 50 mg accuracy
- 3 axis gyroscope with $\pm$ 2000 $^\circ\,s^{-1}$ accuracy
- Pressure sensor with $\pm$ 10 Pa accuracy
- 3 axis magnetometer with $\pm$ 6° accuracy
- Ultrasound sensor (facing downwards)
- Front camera (HD 720p 30 fps)
- Bottom camera (QVGA, 60 fps)

The accelerometer and gyroscope form the IMU, or Inertial Measurement Unit. During normal (remote-controlled) operation, the IMU is used to measure the speed of the drone, and stabilize it. The pressure sensor has a precision of 10 Pa, which is equivalent to about 83 cm, so it is not precise enough to be a useful sensor indoors. Instead, the ultrasound sensor gives

a very precise measure of the altitude of the drone. The magnetometer is used along with the IMU to estimate the drone's orientations. The bottom camera is used to estimate the drone's speed, by using optical flow methods. Finally, the front camera is not used for navigation, but for the user to see through the drone, and make films or take pictures. In this work, however, we will use the front camera as the main sensor of our Simultaneous Localization and Mapping algorithm, to detect and map visual features. In indoor environments, the front camera is the drone's sensor that has the potential to capture the most information about its surroundings. Unfortunately, this information comes in the form of images, and deducing useable information from these images is not a trivial task. The main focus of this work is to transform these images into useable information.

1.5.2 Software

Closed-source internal code

The code that runs on the drone during normal execution is unfortunately closed-source. This code receives a velocity command from the user via a smartphone app, and drives the drone according to this input. When no input is given, the drone stays stationary, in "hover mode". To achieve this, it fuses sensor information from the IMU, the ultrasound, pressure sensor, magnetometer, as well as odometry from the visual flow of the bottom camera to estimate its displacement, and uses this fused pose estimate to stabilize itself.

Parrot SDK

The software development kit released by Parrot allows to send commands and receive information from the drone. It is possible to send the drone commands to take off, land, emergency stop, hover, or move in a certain direction, but not to directly control the command send to the motors, or to have access at the control law used for the hover mode. Similarly, it is possible to read the IMU, ultrasound, odometry and video data from the drone, but not to access the data fusion that it does internally.

ROS

For the practical programming of the drone, this project uses the ROS (robot operating system) toolbox since last year. This toolbox provides useful abstraction to program robots: the program is decomposed into "nodes", each node running simultaneously in a separate thread. Each node implements part of the drone's task, and the nodes can communicate between each other by sending structured messages. The objective is to make the code modular, to be able to use existing nodes uploaded by other developers as part of packages. One such package, called "ardrone autonomy" developed by the Autonomy Lab of Simon Fraser University handles communication with a physical AR.Drone, so by using this node, we can completely disregard Parrot's specific SDK, and send all commands in ROS format. This way, we hope to make it easy to reuse code in case of a change of hardware. ROS also provides a number of convenient tools for developing software to be used on robots: it provides ways to record and replay data, to easily set and change parameters, and to cherry pick which parts of the program to run when doing tests.

OpenCV

OpenCV is an open-source computer vision library. It provides functions and data structures to manipulate images, and implements many state-of-the-art computer vision algorithms. It can work with ROS messages. We will use this library for all of our computer vision implementation: keypoint detection, tracking, description, and matching.

PCL

PCL stands for Point Cloud Library and is also an open-source library for 3D point cloud processing. The map created by the drone is a point cloud, and is saved as a PCL data structure. Currently, PCL is only used to visualize this map, but in the future, it can be used for higher-level functions like combining maps created by multiple robots, or reconstructing a dense representation of the map from a point cloud.

Ceres Solver

Ceres solver [2] is an open-source library for modeling and solving non-linear least squares problems. It is developed and maintained by Google, where it is used among other applications to estimate the pose of the Google street cars. Google released it to the public as open-source software in 2012 [1]. It will be used in this thesis to solve Bundle Adjustment problems (see section).

1.6 Outline

First, the state of the art of miniature UAVs, computer vision, and simultaneous localization and mapping will be explored in chapter 2. Chapters 3 through 5 will present the main subject of this work: computer vision (Chapter 3), localization (Chapter 4), mapping (Chapter 5). The results will then be presented in chapters 6: simultaneous localization and mapping. Finally, chapter 7 will expose the main conclusions of this thesis, and describe the main challenges for further work.

Chapter 2

State of the art

2.1 Drones

When talking about autonomous drone navigation, it is important to be aware of what the current hardware is capable of doing, and how we can expect it to evolve in the near future. There is quite a large choice of civilian drones available to choose from on the market. Here, we will look at the main differences between the available choices.

2.1.1 Number of rotors

Multirotors, or multicopters, are defined as rotorcrafts with three or more rotors. Having more rotors enables them to maneuver in 3D space with fixed-pitch rotors, unlike helicopters, which have articulations at the bases of their rotors. The most common multirotors have 3, 4, 6, or 8 rotors, and are respectively called tricopters, quadcopters (or quadrotors), hexacopters, and octocopters. Having more rotors has the advantage of giving more agility, at the cost of more energy consumption, and therefore a shorter battery life. A free solid object in 3D space, such as a multicopter, has 6 degrees of freedom: 3 for translation and 3 for rotation. To be able to directly control each of these 6 degrees of freedom, it must be possible to give 6 independent controls to the drone. This means that tricopters and quadcopters are always under-actuated: they can't directly control all 6 degrees of freedom. For example, quadrotors whose rotors are all in the same plane (as is almost always the case), can directly control all three of their rotational degrees of freedom, but only one translational degree of freedom, as they can only accelerate in the direction parallel to the rotation of their rotors. Therefore, to control their position in the plane perpendicular to the direction of gravity, they have to first adapt their roll and pitch, so that the resulting force of gravity and the thrust of their rotors points inside that plane. Most hexacopters also work this way, as their rotors are also often in the same plane. Some hexrotors, however, have tilted rotors, and are fully actuated [32].
Octocopters on the other hand, are always over-actuated. One example, the Omnicopter, developed at ETH Zurich [7] can perform a 360° rotation along any axis, and move in a straight line in any direction, which enables it to perform complex and precise maneuvers. It is able to catch thrown ping-pong balls with a little net.

2.2 Computer Vision

2.2.1 Keypoint Detection, Description, and Matching

Keypoint detection will be at the basis of our work, as we will build a map to localize the drone, and this map will be made up of landmarks, visual features observed with the camera of the drone. Good keypoints need to be recognizable from a wide variety of viewpoints, and ideally be invariant to changes of illumination, scale, rotation, or occlusion. Here is a quick overview of

some of the most important keypoint detectors and descriptors that exist in the literature.

Scale-Invariant Feature Transform (SIFT) Published in 1999 by David Lowe from Columbia Univeristy, SIFT [23] is the reference in keypoint description, because it is quite robust. It uses the maxima and minima of a difference of Gaussian function of the image, rescaled at different levels. The image is blurred by Gaussian filters at different scales, and the difference between blurred images is taken. The result is an image without its highest spacial frequencies (noise) and lowest spacial frequencies (untextured parts of the image), leaving only a certain range of frequencies, that correspond to specific detail levels. The maxima and minima of the resulting image are considered to be corners, and become keypoints. Each keypoint is then described by a vector of size 128, representing histograms of orientation in the pixels around the keypoint. SIFT keypoints are very robust to changes of viewpoint, occlusion and illumination. Their main drawback is the computational cost to find them and to extract their descriptor.

Speeded Up Robust Features (SURF) Inspired by SIFT and first published in 2006, Speeded-Up Robust Features [15] was a new kind of keypoint detector and descriptor. SURF uses square shaped filters to approximate Gaussian smoothing, which can be computed much faster, and then selects points where the determinant of the Hessian matrix is maximal. Its performance in terms of robustness to changes in viewpoint, occlusion, and illumination is similar to that of SIFT but it is computationally much faster.

Features from Accelerated Segment Test (FAST) Published in 2005, the FAST detector [28] uses a different approach than SIFT and SURF, and is even faster than SURF. FAST compares the intensity of a central pixel with the intensities of the 16 pixels forming a Bresenham circle of radius 3 around the central pixel. If the central pixel is brighter or darker by a certain threshold than N contiguous pixels in the circle, the pixel is considered a corner. This test is very fast, because it is possible to reject points that do not match the criteria without comparing the intensities of all points of the circle. The threshold and the number N are parameters that have to be fixed by the user. In the original version of FAST, $N = 12$ and the four pixels at the cardinal points of the circle (up, down, left, right) are first tested, and the point is rejected if the values of these four points make it impossible for N consecutive brighter or darker pixels to exist. Expanding on this idea, the creators of FAST proposed an upgrade to their algorithm in [29] using machine learning. Their upgrade uses the ID3 algorithm to learn a decision tree to decide whether a point is a keypoint or not using the intensities of the 16 pixels. This algorithm automatically selects to best tests to perform on these 16 pixels to quickly decide whether they are corners or not. With this detector, the natural descriptor to use are the intensities of the 16 surrounding pixels, as well as whether the keypoint is a minimum or a maximum. We can already see that the dimension of the descriptor vector of FAST is 16, where SIFT's descriptor is of dimension 128, so we can expect this descriptor to be much less robust than SIFT.

Binary Robust Independent Elementary Features (BRIEF) BRIEF [8] is a descriptor introduced in 2010 that represents keypoints with binary strings. Each bit in the string is the result of a test testing which one of two specific pixels in the neighborhood of the keypoint is brightest in the smoothed image. Different lengths can be used for the string, but the creators of this descriptor found that a size of only 256 or even 128 bits is often enough for accurate matching. Matching between BRIEF features is done using the Hamming distance, which can be computed very quickly on modern computers.

Oriented FAST and Rotated BRIEF (ORB) In 2011 Rublee et al. [30] proposed a combination of a modified FAST detector, and a modified BRIEF descriptor, to obtain the

ORB detector and descriptor. One of the main drawbacks of FAST is that is has no orientation component, so it is unable to recognize keypoints when these are rotated in the camera plane. ORB computes an orientation component to each keypoint detected by FAST by using the intensity centroid of the patch, and then computes a rotated BRIEF descriptor, so that the resulting keypoint is invariant to rotations. Because the FAST detector and BRIEF descriptor are both very fast methods, and result in very small descriptors, ORB keypoints are very efficient computationally.

2.2.2 Bundle Adjustment

2.3 Simultaneous Localization And Mapping

Simultaneous Localization and Mapping (SLAM) refers to the joint task of creating a map of a robot's surroundings, while also keeping track of the robot's location in this map. The word "robot" should be understood very broadly in this context, for example it could be a simple hand-held camera. Because there are countless different types of robots that do SLAM, SLAM is also a very diverse field, with different algorithms for different kinds of sensors.

2.3.1 Parallel Tracking and Mapping (PTAM)

PTAM [19] was one of the first applications of Bundle Adjustment in real time. It is able to track the movement of a hand-held camera by constructing a map of visual features (it uses FAST). It is called parallel tracking and mapping, because the tracking and mapping tasks are completely decoupled, and happen simultaneously in different threads. However, it is quite expensive computationally, so it only works in real time with small workspaces, and lacks loop closure (the ability to correct accumulated errors when revisiting a previously visited location). PTAM has been adapted to work with Parrot AR.Drones at the Technische Universtät München [10] with impressive results, however this solutions suffers from PTAM's problems: the size of the map is quite limited, and there is no loop closure. As a result, the implementation if [10] builds an initial map, and then stops mapping, only using this initial map for tracking, which makes it unusable for large environments.

2.3.2 ORB-SLAM

ORB-SLAM [24], created in 2015 is a keyframe-based monocular visual SLAM that builds a map of ORB features. The creators of ORB-SLAM extended it to ORB-SLAM 2 [25], which works with stereo and RGB-D cameras for better accuracy. It uses bundle adjustment to build a consistent map and for loop closure, and works in real time in large environments, by using a covisibility graph to focus the mapping and localization tasks on a small portion of the map at a time. It also uses a survival of the fittest approach to remove redundant keyframes and points.

2.3.3 Large-Scale Direct Monocular SLAM (LSD-SLAM)

Developped at the Technishe Universität München in 2014, LSD-SLAM [12] uses a semi-dense approach to SLAM. Unlike the previously mentioned methods, it does not use keypoints, but rather, the entire image. Each new image is used to estimate a similarity transform from the previous image to estimate the position of the camera, and is then used to either refine the last keyframe, or create a new one. Each keyframe consists of an image and a depth map created with per-pixel stereo comparisons between consecutive images. The result works in real time on a CPU, and is able to obtain accurate maps of large environments, and to correct accumulated errors after loop closure.

2.3.4 Direct Sparse Odometry (DSO)

Also produced at the TUM, DSO [11] was created in 2016. Like LSD-SLAM, DSO is a direct method: it works directly on the image pixels by computing an associated depth field, instead of extracting keypoints from the image. Working directly on the image allows to bypass the costly point detection and description steps, and also allows to use points that are not recognizable by themselves, making more data useable (edges, weak intensity regions) and to work in less textured environments. Unlike LSD-SLAM, DSO is a sparse method, it samples a limited number of points on each keyframe, and does not use a smoothness geometry prior. Because DSO permanently marginalizes old points, it cannot detect a loop closure and correct accumulated errors, so it is more a pure visual odometry than a complete SLAM system, and cannot be used to obtain a consistent map (if it visits the same location twice, all points will be doubled). However, it is very accurate, and even after very large loops, the accumulated error remains small. Figure 2.1 shows an illustration of the accumulated error of DSO after going through an entire subway station.

Figure 2.1: Drift of DSO after going through an entire subway station. [17]

2.3.5 SVO

2.3.6 CNN-SLAM

Chapter 3

Computer Vision

Our approach to the SLAM problem will expand on the work done in previous years, and therefore we will continue using keyframe-based SLAM. This method builds a map made out of points observed from a small number of previous poses of the drone (keyframes). We will use the same terminology as in [10]: detected points in images will be referred to as keypoints, and keypoints whose 3D position we have an estimation of will be referred to as landmarks. As we will see later, at least two observations of a same keypoint are necessary to estimate its 3D position and use it as a landmark. Being able to recognize points in different images that correspond to the same world location lies at the basis of both the mapping and localization tasks. To do this, we will proceed in three steps. First, we need to determine which points of an image are particular enough to be recognizable from many different viewpoints. Then we need to characterize and save those points in the form of a descriptor. Finally, when we detect more points later on, we need to be able to compare these descriptors to determine which ones describe the same keypoint.
We already outlined the principal keypoint detectors and descriptors in the state of the art (see section 2.2.1). For this work, we will continue building on what was started last year, and we will use a SURF detector and a SIFT descriptor for the reasons explained in their thesis (see [4]).

3.1 Camera geometry and projection

Before we begin it is interesting to think about how cameras represent the world. Images from monocular cameras are a two dimensional representation of a three dimensional scene. One of the main challenges of this work will be to recover the 3D geometry of the scene from these 2D measurements.

3.1.1 Extrinsic parameters

First of all, a camera can only take measurements relative to itself, so it makes sense to express the coordinates of 3D points in a coordinate frame relative to the camera. The camera's coordinate frame is defined as follows: the Z direction is defined as pointing out of the camera, and the X and Y directions point towards the right and bottom of the direction of the image respectively (see figure 3.1). This concords nicely with image coordinates, where $(0, 0)$ is usually defined to be the top-left of the image.

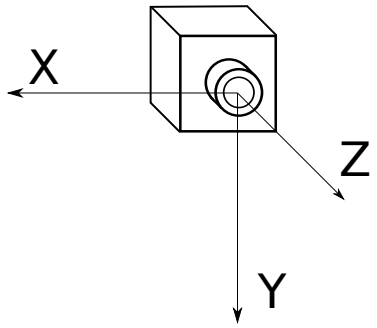


Figure 3.1: Camera coordinate system

Let p_w and p_c be the coordinates of point p in a reference coordinate system and in the camera coordinate system respectively. Knowing the camera's position and orientation in the reference system, we can deduce the transformation to go from one coordinate system to the other:

$$p_c = R \cdot p_w + t \tag{3.1}$$

where R is the rotation matrix between the two coordinate systems, and t is the origin of the reference frame expressed in the camera coordinate system. Using homogenous coordinates, equation 3.1 can be rewritten as follows:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = [R|t] \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{3.2}$$

The extrinsic parameters of the camera are its position, t, which has 3 degrees of freedom, as well as its orientation, represented by R, which also has three degrees of freedom. The orientation's degrees of freedom can be parametrized as three angles, for example the roll, pitch and yaw angles. It is posible to recover these angles from the rotation matrix, and vice versa. The task of localization will be to recover the camera's extrinsic parameters, and from these, the pose of the drone.

3.1.2 Instrinsic Parameters

Once we have the coordinates of point p in the camera's system, we can project this point onto the image plane, using the camera's instrisic matrix parameters. These are the horizontal and vertical focal lengths f_x and f_y, pricipal point coordinates c_x and c_y and skew factor s. From these, the image coordinates u and v can be recovered as follows:

$$u = \frac{f_x x_c + s y_c}{z_c} + c_x \tag{3.3}$$

$$v = \frac{f_y y_c}{z_c} + c_y \tag{3.4}$$

We can already see that our camera only gives information up to a scale factor, as multiplying all camera frame coordinates by a scalar gives the same image coordinates. The above equations can also we rewritten in matrix form, again using homogenous coordinates:

$$z_c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} \tag{3.5}$$

The matrix containing all intrinsic parameters of the camera is called the intrinsic matrix K:

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

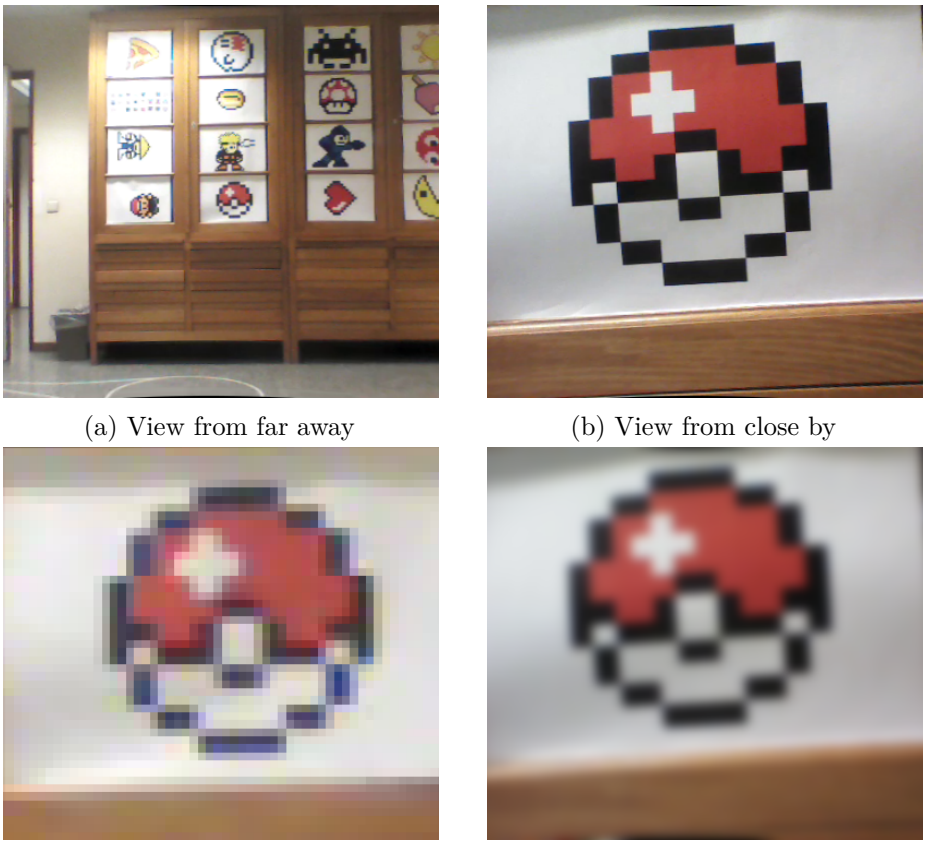
The intrinsic parameters of the camera can be calculated from the output of the camera through a process called camera calibration. As their name implies, they are characteristics intrinsic to a camera, so they only need to be calculated once to be useable as known quantities.
Finally, the full projection from world coordinates to image coordinates can be performed as follows:

$$z_c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K[R|t] \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \tag{3.7}$$

The projection matrix $P = K[R|t]$ gives the image coordinates up to a scale factor (z_c), but this can easily be resolved by normalizing so that the last coordinate is 1.

3.2 Scale-space representation

One of the problems when trying to recognize points from a variety of viewpoints, is that depending on the distance between the point and the camera, the point can have many different sizes in the image. The scale-space representation is an attempt to solve this problem. An image can be represented as a two variable function $f(x, y)$, where the value of f is the intensity of the pixel at location (x, y). The scale-space representation of f is a family of signals obtained by convoluting the original signal with a Gaussian filter g: $L(x, y; \sigma) = g(x, y; \sigma) \star f(x, y)$. For different values of the parameter σ (the scale), the result is a blurred image where finer and finer details are indistinguishable. When $\sigma = 0$, the Gaussian becomes an impulse function and the result of the convolution is the original image. The scale-space representation of an image, $L(x, y, \sigma)$ can be seen as a three dimensional image, made by stacking images obtained by blurring the source image more and more. By working on the scale-space representation of images, we ensure that our methods are scale invariant. [21]
To illustrate this with an example, we look at figure 3.2. Images 3.2a and 3.2b have been observed and we would like to match them. Zooming in on image 3.2a gives 3.2c, but this image is quite different from 3.2b. If we look at the scale-space representation of 3.2b we might find a match at higher scales, when the image becomes blurred to look like image 3.2d. Note that we are matching individual keypoints, which are usually smaller than this image (the corners of this image could each be keypoints).



(a) View from far away

(b) View from close by

(c) From far away, zoomed in

(d) From close by, filtered by a Gaussian

Figure 3.2: Usefulness of the scale-space representation

3.3 SURF Keypoint Detector

The SURF detector uses the determinant of the Hessian matrix to measure local change around points. The Hessian matrix is defined as follows:

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix} \tag{3.8}$$

where L_{xx}, L_{xy} and L_{yy} are the convolution of the image with the second order derivatives of the Gaussian $g(\mathbf{x}; \sigma)$. The creators of SURF decided to approximate the second order Gaussian derivatives with box filters. This decision stemmed from the fact that even though Gaussian filters are optimal, they are already modified quite heavily, as they have to be discretized and cropped, and the result of the convolution is then resampled. The big advantage of using square filters is that they can be evaluated very quickly by using integral images. To obtain a scale-space representation, the SURF method uses box filters of different sizes: for example, a 9x9 filter, approximates Gaussian with parameter $\sigma = 1.2$, with larger filters corresponding to bigger values of σ. Local maxima on 3x3x3 neighborhoods of the scale-space representation are then taken as points of interest.

Finally, so that the detected keypoints are rotationally invariant, we need to assign an orientation to each keypoint that can be recognized from subsequent viewpoints. To do this, the SURF method computes the response to vertical and horizontal Haar wavelets in a circular neighborhood of radius $6s$ around the detected points, s being the scale at which that point was detected. The Haar wavelets are rectangle shaped, so they can also be computed efficiently using integral images. The responses are plotted in a plane, where the abscissa is the horizontal response, and the ordinate is the vertical response. The responses are summed within a sliding orientation

window, giving an orientation vector for each position of the sliding window. The orientation vector with the largest norm is taken to be the orientation of the keypoint. [15]

3.4 SIFT Keypoint Descriptor

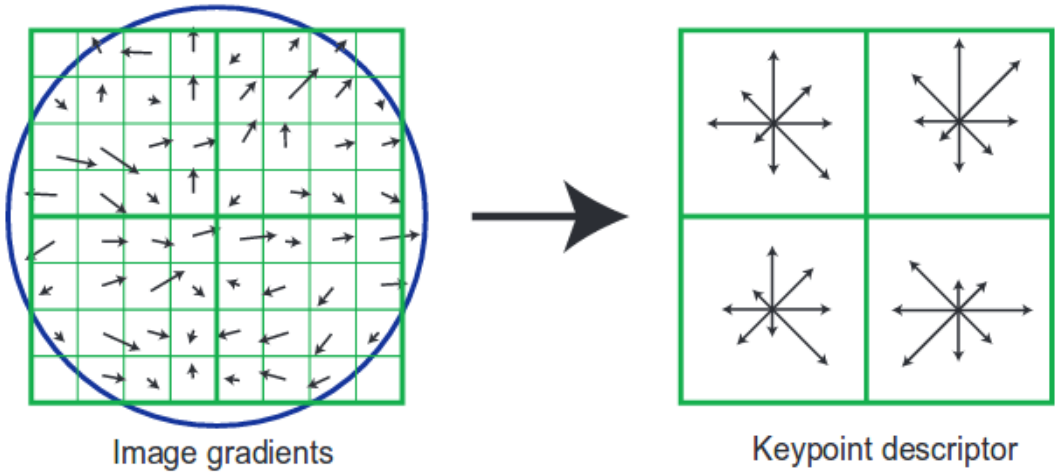


Figure 3.3: Illustration of a SIFT descriptor. Each subregion containing 4x4 points defines a histogram with 8 bins. Note that SIFT uses 4x4 subregions, not 2x2 like in this illustration. [22]

To create a SIFT descriptor, a 16x16 square region is taken around the keypoint. This region is rotated and scaled according to the orientation and scale of the detected keypoint. The 16x16 region is subdivided into 16 4x4 subregions. In each subregion, a histogram of local gradients is computed and quantized into 8 bins. The magnitudes of these histograms form the descriptor. Figure 3.3 shows an illustration of the histograms. As there are 16 histograms of 8 bins each, the total length of the descriptor vector is 128. This is quite a large dimension, which has the advantage of being highly discriminative, even with many different keypoints, but the disadvantage of requiring more space to be saved, and to slow down the matching algorithms that will be used later on. [23]

3.5 Keypoint Tracking

Because the above algorithm to extract a descriptor of a keypoint is quite slow (it takes approximately TODO to extract all descriptors from a new image, see TODO), it is not practical to run these algorithms on each new image, as this would greatly reduce the frame rate. For this reason, another, faster alternative has been found : keypoint tracking. This method finds keypoints by looking for keypoints that were observed in the previous image, and assuming that their displacement inside the image was small.
The algorithm used for this keypoint tracking is called the pyramidal Lucas-Kanade method [6]. This method assumes that the movement between successive frames is small and can be approximated by an affine transformation in local regions. It uses small dense pixel regions to estimate the movement of a keypoint between successive frames. The method was explained more in detail in last year's master thesis [4]. Keypoint tracking has the advantage of being much faster than detecting new keypoints, especially because the descriptor of each tracked keypoint is already known and does not have to be extracted again at each frame. The main drawback of this method is that it does not allow to detect any new keypoints, as all tracked keypoints have to be present in the previous frame.

3.6 Detection and Tracking hybrid

In the previous years, detection and tracking were used alternatively: each new frame was populated with keypoints using one of the two methods. As long as there were enough keypoints in the preceding frame, tracking was used on all these keypoints with no detection of new keypoints. While tracking is used, the number of keypoints in each successive frame is nonincreasing, as each keypoint can either be lost or tracked, and when the number of keypoints in the frame fell below some threshold, keypoint detection was used to populate the next frame with new keypoints. When the detection method is used, all previous keypoints are discarded, and keypoints are detected on the entire image. The reason for this choice was that detection was too slow to be used on each frame, but necessary to find new keypoints, and that if detection and tracking were used simultaneously, keypoints that are already being tracked would be re-detected, which would result in duplicate keypoints in the same frame. However, this method is suboptimal, because every time detection is used, all currently tracked keypoints are lost and have to be re-detected. It also causes quite an inconsistent frame rate.

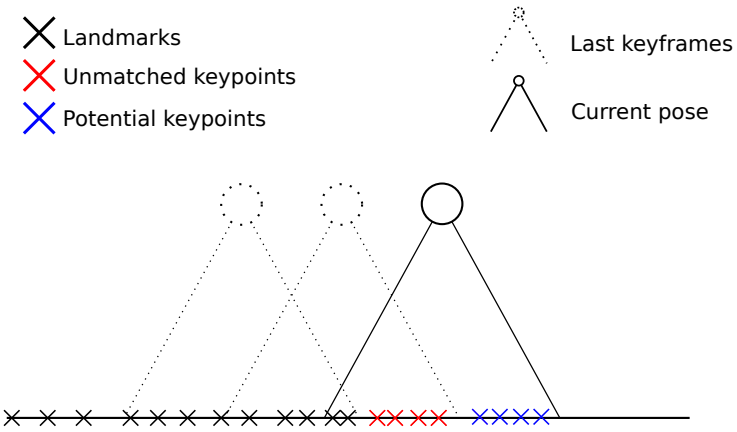


Figure 3.4: Delay between first observation of a keypoint, and mapping of this keypoint

In addition to being suboptimal, this method has a few drawbacks that make it hard to use for this project. As we will see later, at least two views of a single keypoint from different keyframes are required to estimate the 3D position of the corresponding landmark. Figure 3.4 is an illustration of this problem, in a one dimensional world for simplicity. When a keypoint is seen for the first time and saved in a keyframe, it becomes an unmatched keypoint (red cross), and its 3D location is unknown. It has to be observed from a second keyframe in order to become a landmark (black cross), with an estimated 3D position. Only then can it be used as a reference to localize the drone. On figure 3.4, if the drone creates a new keyframe at its current position, the unmatched keypoints can become landmarks, and the potential keypoints become new unmatched keypoints. If we had not detected and saved those new unmatched keypoints, when the drone continues advancing it will not be able to add any landmarks to the map. The difficulty lies in the fact that each time we create a new keyframe, we ideally need to see:

1. Enough already mapped landmarks to accurately estimate the position of the keyframe
2. Unmatched keypoints from previous keyframes so that we can put them into the map to allow further exploration
3. New, previously unobserved keypoints to ensure long-term viability (these points will be the unmatched keypoints of the next keyframe)

The need for these three ingredients means that keyframes have to be created more frequently than when we are mapping keypoints from single observations. In many cases, there are enough common keypoints between successive keyframes so that the old method did not require detection at any moment between the two keyframes. As a result, all keypoints of the most recent keyframe would already have been observed before. This means that they can be added to the map, but it seriously harms possible future exploration, as the third element of the above list is missing, so it won't be able to create any new landmarks when the next keyframe is created. Simply decreasing the threshold at which we used detection instead of tracking was not a good solution as it resulted in many more slow detection steps.

To solve this problem, a new technique had to be found. It is clear that we need new, unmatched keypoints in each keyframe, and these points cannot be tracked from previous images, so they have to be detected. However, keypoint detection takes so much time it's prohibitive to do it on each image. We noticed that the vast majority of keypoints that are lost during tracking are lost because they are moved out of the field of vision, so they leave the image from the sides. Similarly, new potential keypoints enter the field of view from the sides, so after some displacements, there will often be sections of the image with many keypoints (parts of the scene that were observed continuously since the last full keypoint detection), and sections of the image with no keypoints (parts of the scene that were not in the field of view when the last full detection took place). Our solution is to treat the part of the image with no keypoints as if it was a separate image, and do keypoint detection on that part, while keeping the part of the image with keypoints, and tracking its keypoints.

(a) Initial frame (b) After displacement and tracking (c) After detection

Figure 3.5: Illustration of the hybrid between tracking and detection

3.7 Keypoint Matching

The final computer vision task is keypoint matching. After keypoints from multiple viewpoints are saved, we need to know which observations correspond to the same point. This is needed when a keyframe is added to the map, because we need keypoint matches to place landmarks in the map, and also when trying to match an image with the map to localize the drone. We will compare the descriptors of the keypoints to determine which ones are similar enough to represent the same point. The Euclidean distance is a good metric of how different the descriptors are, because they are vectors of floating point numbers (if we were working with binary descriptors, like in the case of BRIEF, we should use the Hamming distance instead).
When matching a query image with a reference image, we find the nearest descriptor of the reference image for each descriptor of the query image. For each pair, we then determine if they are close enough to be considered a match. Finding the exact nearest neighbor would take $O(n * m)$ time for an exhaustive search, where n and m are respectively the number of keypoints in the query and in the reference images. A search with a k-d tree runs in $O(n \log(m))$ time, but suffers from the curse of dimensionality, so in high dimensional spaces it is not always faster than an exhaustive search in practice. To overcome these problems, we only look for

an approximate nearest neighbor. There exist many different approximate nearest neighbor methods, one of the most famous for high dimensional search is the best-bin first method, which was recommended by the creators of SIFT. It is based on a k-d tree and finds the nearest neighbor in most cases, and if not, it finds a close neighbor. Other, better approximate nearest neighbor methods have since been invented, so to marginalize the choice of an approximate nearest neighbor algorithm, we use the implementation of a FLANN matcher of OpenCV for the nearest neighbor computation. FLANN stands for Fast Library for Approximate Nearest Neighbors. It contains many state-of-the-art nearest neighbors algorithms and automatically chooses the most appropriate one depending on the data.

Chapter 4

Localization

The first of the two main ingredients of SLAM is localization. The task of localization is the estimation of the drone's position given a map and sensor information. In our case, the map is a map of visual features, and the main sensor used will be the camera. This part was already carried out by last year's groups, and it is already made to work in the three dimensional case. For completeness, we quickly repeat in this chapter how it works.

4.1 Perspective-n-Point

To localize the drone using visual information and a map, we need to solve the perspective-n-point (PnP) problem. PnP is the problem of estimating the six degrees of freedom 3D pose of a camera, given n observations of 3D points whose locations are known. We use two main methods to solve the PnP problem: P3P and EPnP.

4.1.1 P3P

The smallest number of point correspondences required to solve the PnP problem is 3, in which case it is called P3P. P3P can be solved with a method based on the cosine law, but it can have multiple solutions. A fourth point correspondence can be used to solve this uncertainty. Given three known points A, B and C, and the center of projection of the camera P, define the following distances: $|AB| = c'$, $|BC| = a'$, $|CA| = b'$, $|PA| = X$, $|PB| = Y$, $|PC| = Z$, and the following angles: $\widehat{APB} = \alpha$, $\widehat{BPC} = \beta$, $\widehat{CPA} = \gamma$. We can then use the cosine law on each of the triangles that has P and two of the points A, B, C as vertices to obtain the following system of equations:

$$X^2 + Y^2 - XY \cdot 2\cos(\alpha) - c'^2 = 0 \tag{4.1}$$

$$Y^2 + Z^2 - YZ \cdot 2\cos(\beta) - a'^2 = 0 \tag{4.2}$$

$$Z^2 + X^2 - ZX \cdot 2\cos(\gamma) - b'^2 = 0 \tag{4.3}$$

The only unknowns is these equations are X, Y and Z, as the distances between points A, B, C can be deduced from their known position, and the angles can be deduced from the observations of these points on the image. For the resolution of these equations, the reader is referred to the paper that established the P3P method [14]. As they show, it is possible to eliminate one of these equations and one of the variables to end up with two quadratic equations and two unknowns. This means that is there exists a finite number of solutions, that number has to be less than or equal to 4. To eliminate the bad solutions, we can use a fourth point correspondence. Once X, Y and Z are determined, the position and orientation of point P be can be deduced.

4.1.2 EPnP

EPnP (Efficient PnP) [20] is a method to solve the more general PnP problem. It uses $n \geq 4$ point correspondences to estimate the position of the camera. The central idea of this method is to express the n points as a weighted sum of four virtual points. Because it uses more points, it is be more stable and resistant to noise than P3P.

4.1.3 Random Sample Consensus

Our data is subject to noise and measurement errors, which can negatively impact our solution. Do deal with this problem, we use Random Sample Consensus (RANSAC). RANSAC is a method to estimate parameters using data that contains outliers. The general idea is to iteratively estimate the parameters using different random samples, and to use a voting scheme to select the best parameters. The algorithm works in two steps:

1. A random sample is drawn from the data with the smallest number of examples to estimate model parameters. From this random sample, we estimate the parameters.

2. We test all the data against the model estimated in step 1. All the data that whose loss according to some predefined loss function falls below a certain threshold when fitted to the model is considered part of the consensus set.

Both above steps are repeated, and the model with the largest consensus set is selected. All the data that are part of this consensus set are considered to be inliers, and the rest of the data are considered outliers. A final model can then be computed using all inliers [13].
Applying this scheme to the PnP problem, we use the P3P method to create a model from each of the random samples, and then use the EPnP method on the inliers. This way we determine what points are outliers with the fastest method (P3P), and then use EPnP so that all inliers are used for the final estimation.

4.2 Visual Odometry from optical flow

Using the output of the bottom-facing camera, it is possible to deduce the ground speed of the drone. Because monocular camera measurements always give information only up to a scale factor, this only gives the ratio between altitude and speed. However, using the ultrasound sensor, that also points downwards, we know the altitude of the drone and can deduce its absolute ground speed. This visual odometry is already implemented by the constructor of the drone, as it is used to control the velocity of the drone during normal (remote-controlled) use. Unfortunately, the code that computes this velocity from the optical flow is a black box inside the closed-source firmware, so we only have access to its output. Those speed estimations can then be integrated to obtain an estimation of the displacement.

4.3 Pose Fusion

Chapter 5

Mapping

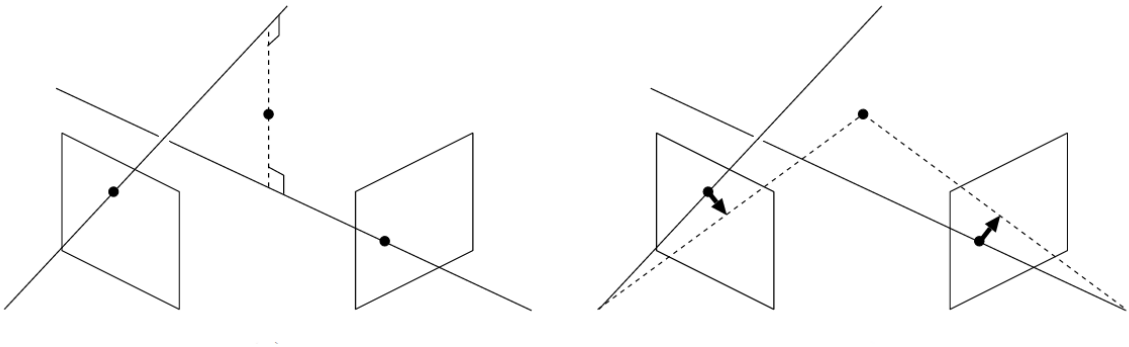
The task of mapping was the main challenge of this work. This task consists in determining the 3D coordinates of recognizable features, that can later be used as landmarks by the drone to estimate its own position. The main challenge when building a 3D map using a single monocular camera, is that a point needs to be observed from at least 2 different positions to be mapped. The simplest approach to map a point, is to simply triangulate its position from two different views. We will begin by exploring this approach. We will find that although this does work reasonably well when we are certain of the position of the cameras, it does not when this position is uncertain. In addition, this method does not allow to take more than two views into account. To remedy these problems we will implement a bundle adjustment step, that allows to build a map that is globally consistent. Throughout this section, we will have to make design choices to try to obtain a method that is both fast enough to work in real time, and accurate enough for the drone to control its position. To evaluate these performances, we will perform a standardized test.

5.1 Structure of the map

We will keep the keyframe-based structure for the map from the previous years, but this structure will need to be adapted so that each landmark does not belong to just one keyframe. The map will contain two main data structures: a list of keyframes, and a list of landmarks. Each keyframe will contain the following information: the drone's pose estimation at the moment the keyframe was created, the estimated corrected pose of the keyframe, a list of observed keypoints, meaning their descriptor and their 2D position in the image plane of the keyframe, and for each keypoint, whether it corresponds to a landmark of the map, and which one. Each landmark will contain a descriptor and estimated 3D coordinates.
The map grows every time we decide to add a keyframe. When we do, the keypoints of the current image seen by the camera, along with their 2D coordinates and descriptors, are saved in a new keyframe. The descriptors are then matched with descriptors of the other keyframes, and when there is a match between two keypoints of two different keyframes, we can triangulate a new landmark into the map.
The main challenges are :
- How to triangulate new points?
- How to take into account when more than two keyframes see a same keypoint ?
- How to prevent errors from accumulating over time?

5.2 Triangulation

Our first problem is where to put a landmark in the 3D world from two observations at two different keyframes. If all measurements were perfect, we could simply draw a ray at each keyframe that goes from the camera center and passes through the keypoint in the image plane, and those two rays would intersect at the position of the landmark.
Unfortunately, those lines never intersect in practice, due to various errors (measurement errors, errors in the model of the cameras, errors on the position estimation of the cameras), so we need to find a method to locate a 3D point as best as possible from the pair of images.



(a) Midpoint method (b) Optimal Correction

Figure 5.1: Triangulation methods

5.2.1 Midpoint Method

The simplest and most obvious solution is to take the midpoint of the common perpendicular of the two rays. This method is intuitive to understand geometrically, and is quite easy to compute. In practice, however its results are not very good, as there is no theoretical reason for this point to be the best.

5.2.2 Linear least squares

A more advanced method for finding the coordinates of a point from a correspondence is called Linear Least squares, and is described in Hartley & Zisserman's *Multiple view geometry in computer vision* [27]. This method also has the advantage that it is already implemented in the OpenCV library.

5.2.3 Optimal Correction

If we assume that the error on observed points is random and follows a Gaussian distribution with zero mean, then the optimal solution would be to displace the pixels on both images until the resulting rays meet, keeping the displacement of the pixels as small as possible in the least squared sense. Such a solution would give the maximum likelihood estimator of the position of the 3D points, under those assumptions. There are several algorithms in the literature that triangulate the position of a point using optimal correction. The most popular one, proposed by Hartley and Sturm [16], computes the solution directly but requires finding the root of a 6th degree polynomial. Kanantani et. al.'s method [18] finds a solution iteratively, but requires very few iterations to have an accurate solution, and in practice, is faster than the Hartley Sturm

method. It also has better numerical properties, as unlike the Hartley-Sturm method, it does not have singularities at the epipoles.
As we will see in the results section (6.1.1), optimal correction performs better than the midpoint method, as could be expected. The performance increase from optimal correction is not very big however, because the biggest source of error is not Gaussian noise on the measurements of the points, but errors in the position estimate of the cameras at the moment of creating the keyframes (especially in the robustness test).

5.3 Bundle Adjustment

Having a bad estimation of the keyframe's pose is problematic, as it will result in badly located landmarks, which in turn will cause a bad estimation of the camera's position when future keyframes are created. In the long term, errors will accumulate, and the map will be completely distorted. Luckily, if we have enough point correspondences between two images, it is possible to deduce the relative displacement between the two images. This means that from a set of images, we can reconstruct a scene, without even needing a prior estimation of the position of the cameras that took the images. This is good news as it means that the images can give us some absolute information about the scene, that can be used to correct the errors on the camera poses of the previous keyframes.
The problem of adjusting camera poses and 3D point locations in order to minimize the reprojection errors of the 3D points onto the image planes is known as bundle adjustment. As stated above, bundle adjustment has the advantage of being absolute with respect to the world, and so not having errors accumulate. Another advantage of bundle adjustment, is that it can easily take into account points that are seen by more than two cameras, which is not trivial for the triangulation techniques described above. The main disadvantage of bundle adjustment is that it is computationally heavy, so it is important to adapt it to be useable un real time.
To show the advantages of bundle adjustment, we compare it with triangulation using the optimal correction method. When using bundle adjustment, optimal correction triangulation is also used to obtain an initial solution, from which we optimize.

Bundle Adjustment an optimization problem of a nonlinear least-squares problem. The problem can be described as follows:

5.3.1 Variables and constants

We are simultaneously trying to determine the layout of the drone's surroundings (landmarks) and to correct the estimations of the drone's poses at the previous keyframe locations. The variables we are optimizing are the previous keyframe poses and the 3D locations of the landmarks. Let $\mathcal{K}$ be the set of keyframes, $\mathcal{L}$ be the set of landmarks and $\mathcal{L}_k$ be the set of landmarks observed by keyframe k. We will call the camera centers of the previous keyframes c_k and their roll-pitch-yaw orientation angles r_k. The 3D position of the landmarks will be called p_l. For each observation of a landmark by a keyframe, we know at what 2D point on he image that point was observed. Let i_{lk} be the image coordinates of landmark l in keyframe k.

Table 5.1: Table of notation for Bundle Adjustment

		Constants
$\mathcal{K}$	$\triangleq$	Set of keyframes
$\mathcal{L}$	$\triangleq$	Set of landmarks
$\mathcal{L}_k$	$\triangleq$	Set of landmarks seen by keyframe k
i_{lk}	$\triangleq$	Image coordinates of landmark l in keyframe k's image plane
		Decision Variables
c_k	$\triangleq$	Position of camera center of keyframe k
r_k	$\triangleq$	Roll-pitch-yaw angles of camera of keyframe k
p_l	$\triangleq$	Position of landmark l

$\text{Proj}_k(\cdot)$ is the projection operator of keyframe k, it transforms the 3D point coordinates of any point into the 2D image coordinates of that point if it were perfectly observed by keyframe k. This operator depends on the intrinsic (focal length, projection center, skew), and extrinsic (position and orientation) parameters of the camera. In general, bundle adjustment refers to the problem of adjusting both the intrinsic and extrinsic camera parameters of the different keyframes, but in our case, the same camera is used at every location, and the intrinsic parameters have been found in advance using camera calibration. Therefore, we can simplify our model by taking the instrinsic parameters as known constants, and only solving for the extrinsic parameters and the position of the landmarks.

5.3.2 Objective function

We want to minimize the reprojection error of the observed landmarks. After estimating p_l, the position of landmark l, we can reproject point l into the images it was observed from. For example, $\text{Proj}_k(p_l)$ is the reprojection of point l into the image plane of camera k. The projection operator $\text{Proj}_k(\cdot)$ is uniquely defined by the camera's position and orientation (as the intrinsic parameters were fixed in advance). We can then compare the reprojection with the actual observation of point l by camera k to find out how consistent the estimation of the position of point l is with the estimation of the position and orientation of keyframe k. Using a least squares loss function, the objective becomes:

$$\min \sum_{k \in \mathcal{K}} \sum_{l \in \mathcal{L}_k} L(\text{Proj}_k(p_l) - i_{lk}) \tag{5.1}$$

for some loss function $L(\cdot)$. We would like to use a squared loss function, as it has good computational properties, but the drawback it gives a lot of weight to outliers. If some landmarks result from bad point correspondences, their reprojection errors will be very large, so they will influence the end result quite a lot. To reduce the effect of outliers, we use a more robust loss function: the Huber loss function.

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{if} |x| \leq \delta \\ \delta|x| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{5.2}$$

This function is a parabola for values if x smaller than δ, and a straight line elsewhere.

Note that if we kept the keyframe positions and orientations constant, and if there were only 2 keyframes observing each landmark, then the triangulation obtained with optimal correction

would already give us the landmark positions that minimize this reprojection error. The interest of using bundle adjustment is that it allows to correct the positions of the camera at the keyframes, and to consider more than two observations per landmark.

5.3.3 Constraints

5.3.4 Solver

5.4 Mapping Strategy

5.4.1 Map Initialization

Because of the chicken-and-egg nature of SLAM (an estimate of the position is necessary to place landmarks, but a map is necessary to estimate the position), a special procedure is needed to initialize the map. Arbitrarily, we decide to fix the origin of the reference frame to be the at the pose of the drone where it creates first keyframe. However, this first keyframe is not enough to initialize a map, because two views of keypoints are required to triangulate them. THis means we have to fly blindly to the location of the second keyframe before we can initialize the map, and begin to estimate the drone's position using the map and PnP.
The only available sensors during this first, blind flight are the IMU, the ultrasonic sensor, and the bottom camera (for optical flow). Of these, only the ultrasonic sensor gives an absolute measurement of position, the others all measure speed, and this speed estimation has to be integrated to give a displacement estimate. For this reason, we will mostly rely on the ultrasonic sensor to estimate the relative position from where we take the second view. Because the ultrasonic sensor only gives the distance from the bottom of the drone to the ground, the drone should fly straight up from its first position (the origin) to reach its second position.
Once in its second position, the drone can match seen keypoints from both views, and from its estimated position, triangulate those points to the map. Then, using bundle adjustment, the error in the estimation of the displacement between the two first keyframes can be corrected, to give a better map.

5.4.2 Rejecting outliers

When looking closer at the results of bundle adjustment, we find that a large part of the errors comes from a small number of points. Figure 5.2 shows the repartition of contribution to the objective function between the landmarks as a logarithmic scale. We can see that 1 % of the landmarks account for more than 40 % of the total error, and that 10 % of the landmarks account for more than 80 % of the total error. One possible explanation is that these points do not correspond to real world points, so that even if we have the correct position of all cameras exactly, the rays corresponding to these points will not intersect, or even be close to intersection. For this reason, after every bundle adjustment pass, we could look at the contribution of every mapped point to the objective function, and eliminate points whose error is too high. Before doing this, there are some things that we have to take into account, because as a point is observed by an increasingly large number of cameras:

- Its total contribution to the cost function will increase, as each observation of a point adds a term to the objective function
- Its contribution per observation to the cost function will also increase, because every time an observation is added, the location of the point will change a bit, and its position will become less optimal if we only consider the cameras that already saw the point before.

For these reasons, we will put a different threshold on the points depending on the number of keyframes that see the point. If a point is seen by a large number of keyframes, we will allow it to have larger errors in the objective function before removing it.

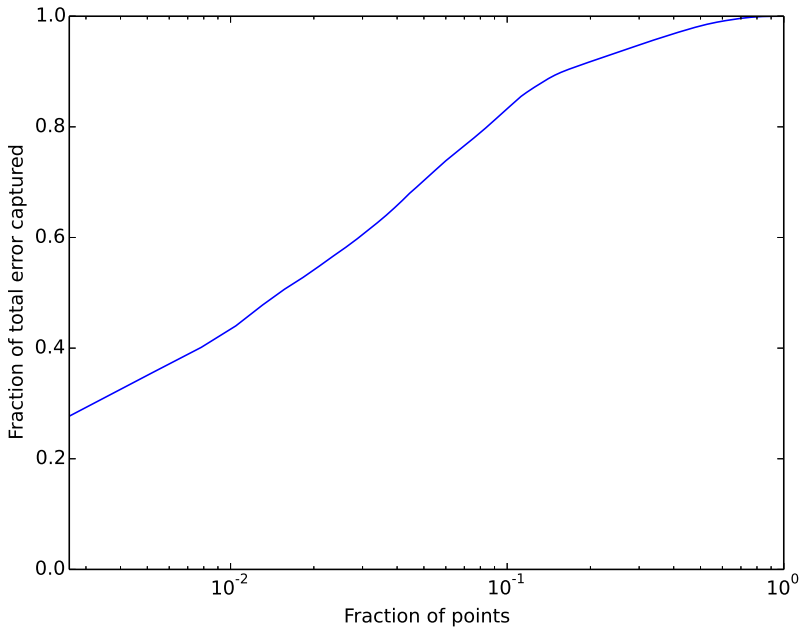


Figure 5.2: Repartition of error among points

Chapter 6

Results

6.1 Evaluation procedure

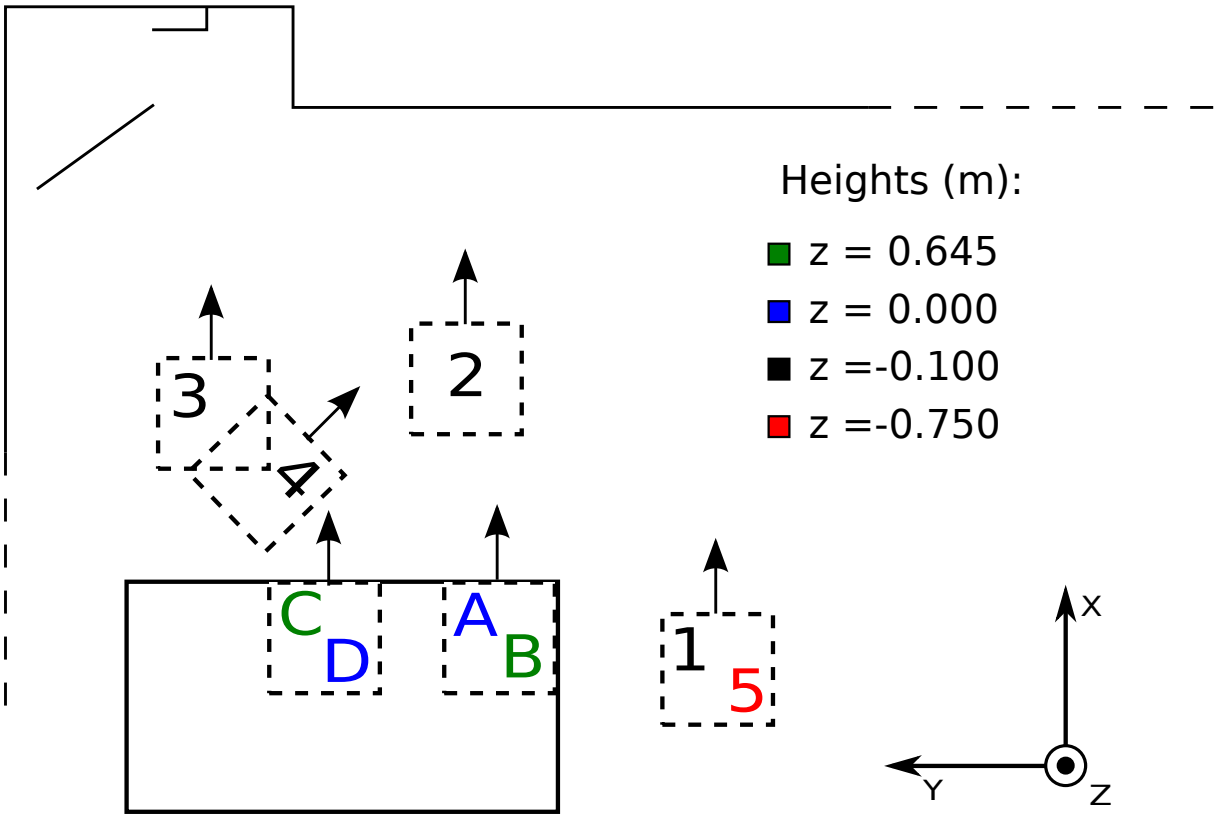
To evaluate the performance of the map we create and use a benchmark test.
During the evaluation, the drone will first build a map, and then that map will be tested by comparing the real position of the drone with the one it estimates using the map and PnP. The setup used for this evaluation is illustrated on figure 6.1a.
The evaluation is performed as follows:

1. The drone is placed in a known position on a table (position A in figure 6.1a). There, the drone is turned on, and it begins initializing its map by creating a first keyframe.

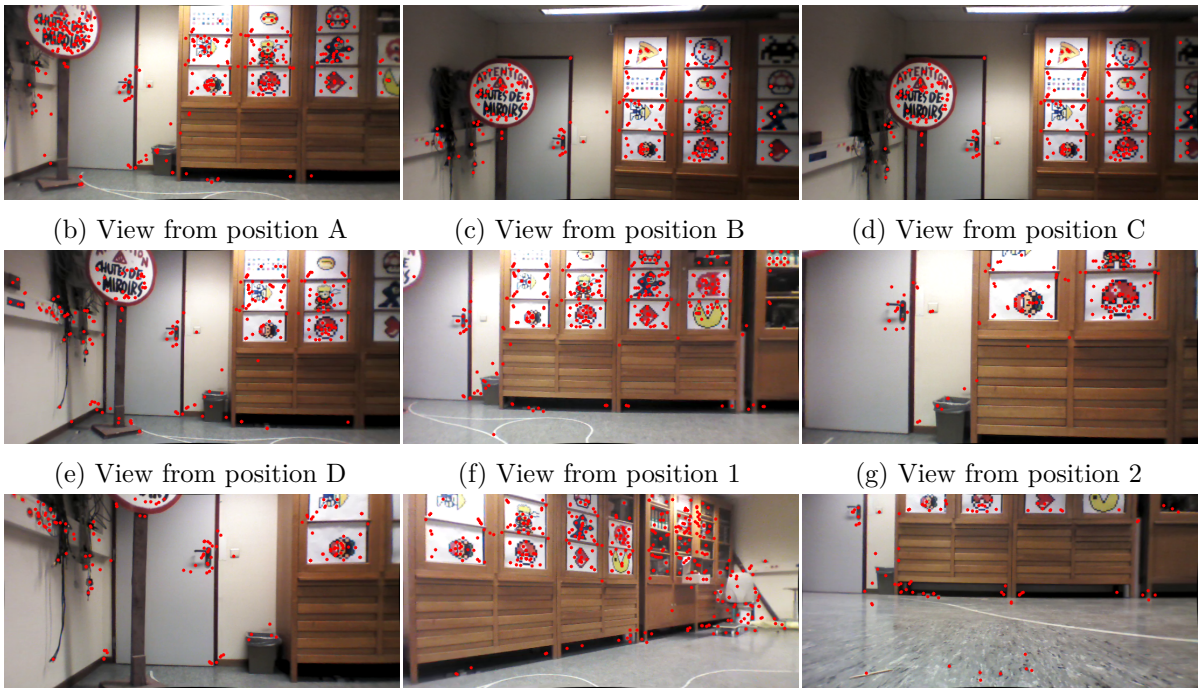
2. The drone is then successively placed in 3 other known locations (B, then C, then D), and at each of these locations, the drone manually receives its position. Every time this happens, the drone creates a keyframe, adds landmarks into the map, and optionally updates existing landmarks' positions, or even removes some landmarks.

3. Finally, the drone is again placed at different known locations (positions 1 through 5 on figure 6.1a). This time, no information is communicated to the drone from the outside, and the drone does not modify its map. The drone estimates its position using only its camera and the map that it built during step 2. We compare the drone's estimated position with its real position to evaluate the quality of the map.

The two quantities we will seek to optimize are the accuracy of the drone's visual pose estimation (as measured during step 3), and the time taken to create the map during step 2.
In reality, the drone would not know its exact position when making keyframes at points B, C, and D (position A is defined as the origin), especially at the beginning, as it would have to rely on its IMU and other internal sensors to estimate its position. To emulate this, we implement a second type of test: the robustness test, where the position that is communicated to the drone is slightly different from its real position, at each of the uncertain points (B, C and D). To differentiate it from the robustness test, we call the first test the precision test.
For both the robustness and precision tests, we measure two quantities: the average distance error and average angle error. The average distance error is the average distance in the xy-plane between the drone's posistion estimate and its real position when it is at the known points of step 3, and the average angle error is the average absolute value of the difference between the estimated yaw angle and the real yaw angle et these positions. We are not counting errors in roll, pitch, and altitude, because as explained in section 4.3, we do not use the result of the visual pose estimate for those quantities, but use the IMU and ultrasound directly. We also do not measure the error when we are moving the drone between those positions, as we do not have the equipment to measure its real position during those transitions.



(a) Different positions of the drone during benchmark test



(b) View from position A (c) View from position B (d) View from position C

(e) View from position D (f) View from position 1 (g) View from position 2

(h) View from position 3 (i) View from position 4 (j) View from position 5

Figure 6.1: Views during benchmark test

6.1.1 Comparison of triangulation methods

Using the evaluation procedure described in section 6.1, we can compare midpoint triangulation and optimal correction. A comparison of their performances can be seen on table 6.1.

Table 6.1: Comparison between triangulation methods

	Accuracy		Robustness	
	Distance (m)	Angle	Distance (m)	Angle
Midpoint Method	0.12	3.0°	0.84	15.5°
Optimal Correction	0.10	2.3°	0.78	12.5°

We can see that the optimal correction method performs somewhat better. However, this comes at the cost of computation time, because it takes approximately 12 ms to triangulate all 165 matches between two views using the midpoint method, and 24 ms to do the same with the optimal corrction method. In both cases however, this time is insignificant when compared to bundle adjustment (which is performed just after triangulation). For this reason, and the fact that optimal correction is better motivated theoretically and that is gives slightly better results on the benchmark test, we will use optimal correction.
It is interesting to see, however, that the difference in performance is much greater between the accuracy and robustness tests than between the two triangulation methods. This can be explained by the fact that optimal correction is well suited to correct measurement errors that follow a normal distribution, but not errors stemming from the fact that the position estimate of the cameras is wrong. That is the reason we need bundle adjustment.

6.1.2 Bundle Adjustment

Table 6.2: Performance of Bundle Adjustment

	Accuracy			Robustness		
	Distance (m)	Angle	Time (s)	Distance (m)	Angle	Time (s)
No B.A.	0.10	2.3°	—	0.78	12.5°	—
With B.A.	0.08	1.3°	0.775	0.14	2.8°	1.836

We can see on table 6.2 that bundle adjustment improves the results of both the accuracy and the robustness tests. The improvement is significant in the accuracy test, but it is especially large in the robustness test. The performance in the robustness test is almost as good as it was in the accuracy test with no bundle adjustment. This first little experiment convinces us that bundle adjustment really works, and is a good way to correct errors coming from imprecise pose estimation. The improvement of the performance in the accuracy test can be explained by the fact that even if we tried giving the drone its exact position when it was building the map, the information we sent was not 100 % exact, and these inaccuracies were corrected by bundle adjustment.

The performance increase gained from bundle adjustment comes at a price, however, in the form of computation time. Table 6.2 shows that bundle adjustment took a total of 0.775 s during the accuracy test, and 1.836 s during the robustness test. This time is the total time for all three rounds of bundle adjustment run during the initialization phase (it is run every time a keyframe is created, except for the first keyframe). As we can see on table 6.3, the time taken by the algorithm increases each time it is ran as the map grows. Because bundle adjustment is ran every

time a keyframe is created but not at every new image, relatively large times are acceptable. However, long times for bundle adjustment limit the speed at which the drone can explore, so we want to keep it under control. We will explore some paths to keep this computation time low, while still having a good enough performance.
We can already note the difference in the time it takes for the robustness and accuracy tests. Bundle adjustment is an iterative algorithm that seeks to minize a certain cost function. Because during the accuracy test we start from closer to the solution, it requires fewer iterations to correct the errors. This means that the better our position estimate, the faster bundle adjustment will run. Because the robustness test is quite pessimistic (it gives a larger error on the position than what the what the drone normally makes), we can expect the bundle adjustment to be faster in practice. Because we now have an estimate of the error the drone makes (from table 6.2), we can implement a more realistic robustness test, where the error given to the drone is closer to the real one. To still be robust, we will still consider a slightly larger error than what we measured to account for worst-case scenarios. From now on in the robustness test we will give the drone a position 20 cm and 5° off.

	Accuracy Test			Robustness Test		
	N	α	Time [s]	N	α	Time [s]
Bundle Adjustment 1	112	2	0.086	112	2	0.213
Bundle Adjustment 2	277	2.40	0.370	277	2.40	0.637
Bundle Adjustment 3	340	2.56	0.320	339	2.56	0.986
Total			0.775			1.836

Table 6.3: Bundle Adjustment Timings during the initialization step of a benchmark test. N is the number of points being adjusted, α is the average number of times each point is observed

This is surprising, as it is just in the case where the pose of the camera is uncertain that bundle adjustment should improve the result, by adjusting the pose of the camera. But because the problem is quite non linear, it is possible to fall in local minima.

6.1.3 Tuning the Bundle Adjustment

The main drawback of Bundle Adjustment is that it requires an iterative method to be solved and can take a lot of time, which of course is a limiting factor for a robot that builds a map in real time. Therefore it is important to optimize both the speed of the computations, and their accuracy. As is often the case, there will have to be a tradeoff between these two goals. To measure both the speed of the Bundle Adjustment and the accuracy of the map obtained, we will conduct some experiments using different parameters to initialize the map.

Convergence of the solver

The first element we can tune is the convergence criterion of the solver that solves the bundle adjustment problem. We will stop when the ratio of the change in the objective function to the value of this function arrives below some threshold. This ensures that the criterion scales with the problem, which is important as the size of the problem can vary during operation (as the map grows, for example). The default value of the Ceres solver is 10^{-6}, but experimentally, we find that we can use a softer threshold to significantly speed up the computations, without impacting the quality of the results too much (and even improving them in the robustness test).

Table 6.4: Effect of convergence criterion on Bundle Adjustment speed and performance

Convergence	Accuracy			Robustness		
Threshold	Distance (m)	Angle (rad)	Time (s)	Distance (m)	Angle (rad)	Time (s)
1.000×10^{1}	0.099	0.041	0.529	0.776	0.219	0.527
5	0.099	0.041	0.533	0.776	0.219	0.560
1	0.099	0.041	0.537	0.776	0.219	0.529
5×10^{-1}	0.099	0.041	0.532	0.284	0.086	0.615
1×10^{-1}	0.116	0.037	0.671	0.244	0.098	1.176
5×10^{-2}	0.068	0.016	0.743	0.291	0.113	2.117
1×10^{-2}	0.090	0.032	1.421	0.823	0.316	2.770
5×10^{-3}	0.091	0.031	1.533	0.142	0.050	4.186
1×10^{-3}	0.085	0.023	3.620	1.035	0.397	4.907
1×10^{-4}	0.084	0.026	5.058	1.362	0.480	10.484
1×10^{-5}	0.090	0.030	12.453	1.483	0.528	13.206
1×10^{-6}	0.090	0.030	15.703	1.374	0.492	13.802
1×10^{-7}	0.088	0.028	15.795	1.559	0.567	12.935

As we can observe on table 6.4, the more severe the convergence threshold, the better the performance on the accuracy test. On the robustness test, however, this is true for high convergence thresholds, but when the convergence threshold is lower than 0.01, the performance starts to degrade as the convergence threshold becomes lower. In both the accuracy and robustness tests, the lower the threshold, the longer it takes. We will chose a threshold of 0.05, as it is a nice compromise of performance on the accuracy test, performance on the robustness test, and speed.

Bibliography

[1] Sameer Agarwal and Keir Mierle. "Introducing Ceres Solver - A Nonlinear Least Squares Solver". In: *Google Open Source Blog* (2012).

[2] Sameer Agarwal, Keir Mierle, et al. *Ceres Solver*. `http://ceres-solver.org`.

[3] Chris Anderson. "Agricultural Drones". In: *MIT Technology Review* vol. 117 | no. 3 (2014).

[4] Alexandre Leclere Arnaud Jacques. "Towards 3D Visual SLAM for an Autonomous Quadcopter Running ROS". MA thesis. Université Catholique de Louvain, 2016.

[5] "Bâtir avec des drones ?" In: *https://uclouvain.be/fr/sciencetoday/actualites/batir-avec-des-drones.html* (2016).

[6] Jean-Yves Bouguet. "Pyramidal Implementation of the Lucas Kanade Feature Tracker". In: *Intel Corporation* ().

[7] Dario Brescianini and Raffaello D'Andrea. "Design, Modeling and Control of an Omni-Directional Aerial Vehicle". In: *IEEE International Conference on Robotics and Automation* (2016).

[8] Michael Calonder, Vincent Lepetit, and Pascal Fua. "BRIEF: Binary robust independent elementary features". In: *European Conference on Computer Vision*. 2010.

[9] Marcus Chavers. "Consumer Drones By the Numbers in 2017 and Beyond". In: *newsledge.com* (2017).

[10] J. Engel. "Autonomous Camera-Based Navigation of a Quadrocopter". MA thesis. Germany: Technical University Munich, Dec. 2011.

[11] J. Engel, V. Koltun, and D. Cremers. "Direct Sparse Odometry". In: *arXiv:1607.02565*. July 2016.

[12] J. Engel, T. Schöps, and D. Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM". In: *European Conference on Computer Vision (ECCV)*. Sept. 2014.

[13] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*. Ed. by Martin A. Fischler and Oscar Firschein. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 726–740. ISBN: 0-934613-33-8.

[14] Xiao-Shan Gao et al. "Complete Solution Classification for the Perspective-Three-Point Problem". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 25.8 (Aug. 2003), pp. 930–943. ISSN: 0162-8828.

[15] T. Tuytelaars H. Bay A. Ess and L. Van Gool. "SURF Speeded Up Robust Features". In: *Computer Vision and Image Understanding (CVIU)* 110.3 (2008), pp. 346–359.

[16] Richard I. Hartley and Peter Sturm. "Triangulation". In: *CVIU - Computer Vision and Image Understanding* Vol. 68, No. 2 (1997).

[17] Daniel Cremers Jakob Engel Vladlen Koltun. *DSO: Direct Sparse Odometry*. Youtube video. July 2016. URL: `https://www.youtube.com/watch?v=C6-xwSOOdqQ`.

[18] Kenichi Kanatani, Yasuyuki Sugaya, and Hirotaka Niitsuma. “Triangulation from Two Views Revisited: Hartley-Sturm vs. Optimal Correction”. In: *BMVC 2008 - Proceedings of the British Machine Vision Conference 2008*. British Machine Vision Association, BMVA, 2008.

[19] Georg Klein and David Murray. “Parallel Tracking and Mapping for Small AR Workspaces”. In: *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*. Nara, Japan, Nov. 2007.

[20] V. Lepetit, F. Moreno-Noguer, and P. Fua. “EPnP: An Accurate O(n) Solution to the PnP Problem”. In: *International Journal of Computer Vision* 81 (2008), pp. 155–166.

[21] Tony Lindeberg. “Scale-space theory: A basic tool for analysing structures at different scales”. In: *Journal of Applied Statistics* 21 (1994), pp. 224–270.

[22] David G. Lowe. *Distinctive Image Features from Scale-Invariant Keypoints*. 2004.

[23] David G. Lowe. “Object recognition from local scale-invariant features”. In: *Proceedings of the International Conference on Computer Vision*. 1999.

[24] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. “ORB-SLAM: a Versatile and Accurate Monocular SLAM System”. In: *CoRR* abs/1502.00956 (2015).

[25] Raul Mur-Artal and Juan D. Tardós. “ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras”. In: *CoRR* abs/1610.06475 (2016).

[26] Russell Naughton. *Remote Piloted Aerial Vehicles : An Anthology*. Tech. rep. Monash University, 2003.

[27] Andrew Zisserman Richard Hartley. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[28] Edward Rosten and Tom Drummond. “Fusing points and lines for high performance tracking.” In: *IEEE International Conference on Computer Vision*. Vol. 2. Oct. 2005, pp. 1508–1511. DOI: 10.1109/ICCV.2005.104. URL: http://www.edwardrosten.com/work/rosten_2005_tracking.pdf.

[29] Edward Rosten and Tom Drummond. “Machine learning for high-speed corner detection”. In: *European Conference on Computer Vision*. Vol. 1. May 2006, pp. 430–443. DOI: 10.1007/11744023_34. URL: http://www.edwardrosten.com/work/rosten_2006_machine.pdf.

[30] Ethan Rublee et al. “ORB: an efficient alternative to SIFT or SURF”. In: *In ICCV*. 2011.

[31] Jimmy Stamp. “Unmanned Drones Have Been Around Since World War I”. In: *smithsonian.com* (2013).

[32] Richard Voyles. *Dexterous Hexrotor Research*. http://web.ics.purdue.edu/ rvoyles/research.Hexrotor.html.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**