UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Three Dimensional Monocular SLAM for Autonomous Drone Navigation

Dissertation presented by
Boris DEHEM

for obtaining the Master's degree in
Mathematical Engineering

Supervisors
Julien HENDRICKX, François WIELANT

Reader
Christophe DE VLEESCHOUWER

Academic year 2016-2017

Abstract

This master's thesis expands on work previously done at the UCL's autonomous drone project to allow three dimensional simultaneous localization and mapping by a low-cost quadcopter. In GPS-denied environments, drones have to rely on their on-board sensors to localize themselves. We decided to use the drone's front camera to build a map of the environment and to localize the drone within that map. We take a keyframe-based approach, building a map from a small set of snapshots of the drone's camera image, and comparing keypoints in these images. We show that our method is able to build an accurate map using the drone's camera output, and a measurement of the drone's altitude (such as one from an ultrasonic sensor).

Contents

Glossary

Pose Pose is the combination of position and orientation in 3D space.

Roll The roll is the angle giving the left/right tilt of a drone.

Pitch The pitch is the angle giving the front/back tilt of a drone.

Yaw The yaw is the rotation angle of a drone around a vertical axis.

Keypoint A keypoint is a point of interest detected on an image.

Frame A frame is an image converted into a list of keypoints, with their descriptor and position in the original image.

Keyframe A keyframe is a frame saved in memory alongside with the estimated position from which the image was taken.

Landmark A landmark is an individual point in our 3D map

Acronyms

SLAM Simulataneous Localization and Mapping

UAV Unmanned Aerial Vehicle

IMU Inertial Measurement Unit

RGB-D Red - Green - Blue - Depth

SDK Software Development Kit

ROS Robot Operating System

PCL Point Cloud Library

SIFT Scale-Invariant Feature Transform

SURF Speeded-Up Robust Features

FAST Features from Accelerated Segment Test

BRIEF Binary Robust Independent Elementary Features

ORB Oriented FAST and Rotated BRIEF

RANSAC Random Sample Consensus

Chapter 1

Introduction

In this chapter, we will first briefly look at the history of drones, and their various applications today. We will put this work in the context of the larger project at the UCL that it is a part of. We will set the objectives of this master's thesis, and then list the resources that will be used to reach these objectives.

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 [28]. 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 [33].

Figure 1.1: The Kettering Bug (1918).

Throughout the 20th century, UAVs became 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 [4], 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 has 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 basis for subsequent work. They implemented visual SLAM methods to allow drones to build a two-dimensional map based on keypoints (first the keypoints were red pucks placed in advance, 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 within this project. Before doing so, they joined forces to re-implement all the existing code, 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 replace it and 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 only used the drone's bottom-facing camera, and made the assumption that all points seen were located on the ground. 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 follow these intermediate objectives:

- Research the current state of the art for 3D keyframe-based monocular visual SLAM
- Study how the drone can estimate its position by using a 3D map
- 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
- Provide improvements to the current code aiming for performance, speed, modularity, and readability

1.5 Resources

For the practical part of this project, we worked with a Parrot AR.Drone and various open-source software.

Figure 1.2: The Parrot AR.Drone

1.5.1 Hardware

The Parrot AR.Drone 2.0 that we used 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 perform 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 $^\circ\,\mathrm{s}^{-1}$ 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 estimate the speed of the drone, and stabilize it. The pressure sensor has a precision of 10 Pa, which is equivalent to about 83 cm of air, so it is not precise enough to be a useful sensor indoors. Instead, the ultrasound sensor gives a very precise measurement of the altitude of the drone, but it only measures a distance in a single direction. 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.

Front Camera

The front camera is not intended to be used as a sensor. During normal operation of the drone, its use is only for the user to be able 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 (SLAM) 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, as unlike the ultrasound sensor, it gives information in an entire area. Its field of view is also usually larger than the bottom camera's, because its scene is farther away. Because it is directed towards the front, it can see where the drone will go to, and so detect potential obstacles. Unfortunately, this information comes in the form of images, and deducing useable information from these images is not a trivial task, and requires computationally expensive algorithms. The main focus of this work will be to transform the images seen by this camera into useable information.

Computational power

The drone can communicate with an external computer through a Wi-Fi connection (the same that is used to connect it to a smartphone for a remote-controlled flight). For simplicity and to accelerate the development phase, we run our code on an external computer, communicating camera and other sensor readings from the drone to the computer, and commands from the computer to the drone. This is contradictory with the fact that the drone should be autonomous,

but we could theoretically move the code on the drone for it to be executed on-board. Even though the computer we are using in the lab has a 3.2 GHz quad-core processor, and is much faster than the embedded computer of the drone, it is realistic for our code to run in real time on an embedded computer, as today there exist cheap embedded computers that are much faster, than the ones included in the AR.Drone. Moreover, the most computationally expensive algorithms we run can be significantly sped up by using a graphics card, which we are not doing at the moment. There also exist graphics cards light enough to be embedded on a flying drone.

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 optical 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 at a certain speed in a certain direction, and it is possible to receive the drone's pose estimate. In both cases, however, the drone works as a black box: we can't directly control the command sent to the motors, and we don't have access at the control law used for the hover mode and for speed control. Similarly, we don't have access to the code estimating the drone's speed from optical flow, or to the data fusion that it does from all it's internal sensor readings.

ROS

When programming the drone for the practical part of this project, we used the ROS (robot operating system) toolbox. All the previous work was already translated last year to use ROS. 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 general and modular, both at development level, by being able to make nodes that work with different kinds of robots, and reuse existing nodes created by other developers, and at runtime level, by being able to run only certain nodes at a time to test each part of the program separately. This also allows for collaboration between ROS's users. One existing ROS package is called "ardrone autonomy" [37]. This package was developed by the Autonomy Lab of Simon Fraser University and contains a node that handles communication with a physical AR.Drone. By using this node, we can completely disregard Parrot's specific SDK, and send all commands in ROS message format. This way, we hope to make it easy to reuse code in case future contributors to this project decide to change the hardware. ROS also provides a number of convenient tools for developing software to be used on robots, for example, it provides ways to record and replay data, and to easily set and change parameters, even during operation.

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 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 store and 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 was first created to solve bundle adjustment problems, but is now a general nonlinear optimization solver. It is developed and maintained by Google, where it is used for a variety of applications, for example to estimate the pose of the Google street cars and solve bundle adjustment problems in project Tango (Google's augmented reality project). It was released to the public as open-source software in 2012 [1]. It will be used in this thesis to solve bundle adjustment problems (see section 5.3).

1.6 Constraints

For the development part of this work, we were hindered by a few constraints:

- We have no way to measure the ground truth of the drone's position. It would be greatly useful to have an external sensor able to tell the position of the drone precisely to compare it to the one the drone estimates.
- The lights in the lab where we carried out our tests are neon lights, which flicker at a high frequency. The frequency of this flickering leads to an aliasing effect on the camera's image.
- Parrot's internal software is closed-source, and we don't have access to some sensors (like the ultrasound) when the drone isn't flying.

1.7 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). Each of these sections explains the theory behind that part of the task, and ends with a small summary and a flowchart outlining the organization of the code for that part. A general outline showing how the different parts of the program work together can be found in appendix A. The main results will then be presented in chapter 6. Finally, chapter 7 will expose the main conclusions of this thesis, and describe the main challenges for further work on this project.

Chapter 2

State of the art

We begin this work by researching the state of the art in drone hardware, computer vision, and SLAM solutions. All three of these are very active topics with frequent innovations, and we will explore some of them.

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 low-cost drones available to choose from on the market. For civilian UAV's, a tradeoff has to be made between performance (speed, agility, accuracy of sensors, power of embedded computers), battery life, and price. However, it is still a relatively young market, and each year, new drones are created that improve on these quantities.

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 hexacopters, however, have tilted rotors, and are fully actuated [38].

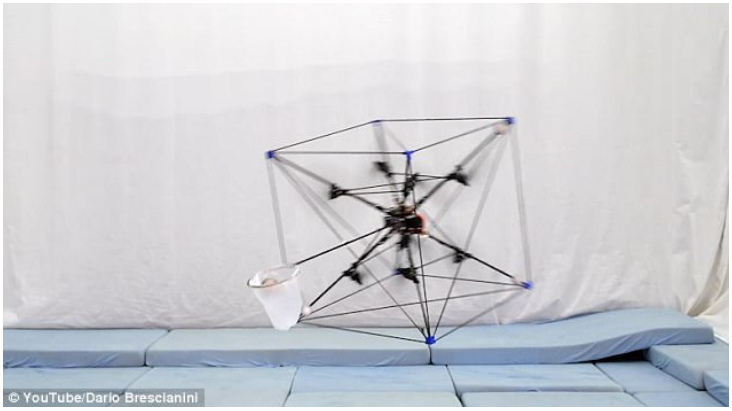


Figure 2.1: The Omnicopter.

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

Computer vision is a very active field of study. Here, we research the state of the art for the two main categories of computer vision tasks we will need to carry out in this thesis: keypoint detection, description and matching, as well as bundle adjustment.

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 University, SIFT [25] 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 [5] 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 [29] 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 [30] 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.

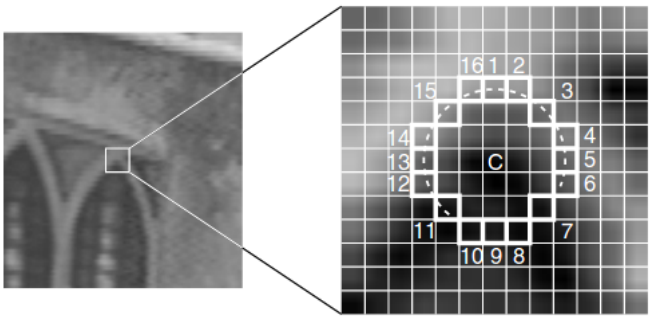


Figure 2.2: Bresenham circle used for FAST. ([29])

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. [31] 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

Bundle adjustment is the problem of using a set of images to simultaneously reconstruct a 3D structure and estimate cameras' positions and intrinsic parameters. It was first used in the 1950's in the field of photogrammetry, the science of taking measurements of photographs [36]. There exist many open-source libraries solving bundle adjustment problems. SBA (sparse bundle adjustment) [23] is a library for solving bundle adjustment problems that takes advantage of their sparse structure. Ceres [2] is another general-purpose solver that is well suited for solving bundle adjustment problems. Bundler [32] is a structure-from-motion system based on SBA that uses bundle adjustment to transform a set of unordered photographs (found on the internet for example) of a common scene to reconstruct it.

Figure 2.3: Result of bundle adjustment (taken from [32])

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 [20] 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 features). 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, their implementation builds an initial map, and then stops mapping, only using this initial map for tracking and position control, which makes it unusable as soon as the drone moves away from this initial map.

2.3.2 ORB-SLAM

ORB-SLAM [26], 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 [27], 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)

Developed 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 2.4 shows an illustration of the accumulated error of DSO after going through an entire subway station.

Figure 2.4: Drift of DSO after going through an entire subway station. We only see the beginning and end of the trajectory (in red). [13]

2.3.5 CNN-SLAM

CNN-SLAM [34] uses a deep Convolutional Neural Network (CNN) to learn the depth field from single views of a monocular camera, and then fuses this information with depth measurements obtained from a keyframe-based monocular SLAM. It also incorporates a second CNN to learn semantic labels from a single view, to learn what parts of the scene constitutes floors, walls, or other objects. Their use of a CNN allows to overcome one of monocular SLAM's main challenges: scale estimation, because it gives an absolute estimation of the scale at every frame, preventing the scale from drifting over time.

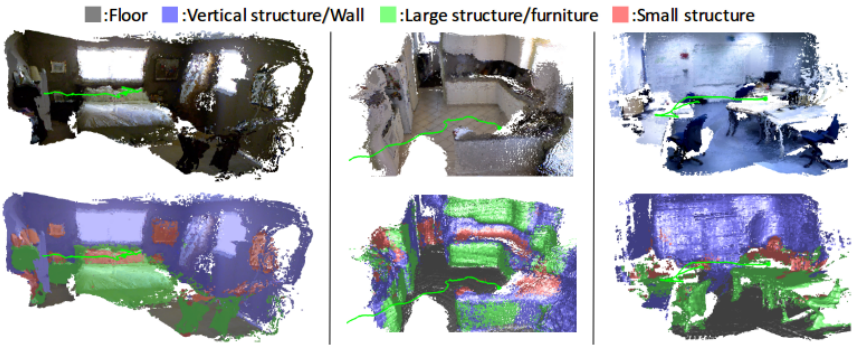


Figure 2.5: Result of CNN-SLAM, with semantic labels. [34]

Chapter 3

Computer Vision

As our visual SLAM uses a camera as its main sensor, many computer vision tasks are required to transform the images obtained from this camera into useful information. In this section, we begin by looking at how a camera perceives the world. Because we are using keyframe-based SLAM, we build a map made out of points observed from a small number of previous poses of the drone (keyframes). As we will see later, at least two observations of a same point 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 point.

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 [18]).

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 Pinhole camera model

The simplest model for a camera is the pinhole camera model. In this model, we consider that light travels in a straight line from the scene to a screen, passing through a very small hole, (a pinhole). As a result the image in the screen is rotated 180°, but if we consider a virtual screen in front of the pinhole instead of behind, we obtain an unrotated image of the scene. We will call this virtual screen the image plane. Using the pinhole camera model, we need two set of parameters to uniquely define the projection from the scene to the image plane: the extrinsic and intrinsic parameters of the camera.

Extrinsic parameters

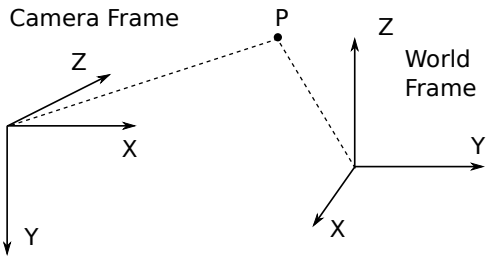


Figure 3.1: Extrinsic parameters

The camera's extrinsic parameters are its position and orientation in the 3D world. They are not constant, and change as the camera is displaced.

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 image respectively (see figure 3.1). This works nicely with image coordinates, where $(0, 0)$ is usually defined to be the top-left of the image.

However, because the camera will be moving around, points of the map have to be expressed in a reference coordinate system, constant and independent of the movement of the camera. The transformation between the reference coordinate system and the camera's coordinate system defines the camera's extrinsic parameters.

Let $p_w = (x_w, y_w, z_w)^T$ and $p_c = (x_c, y_c z_c)^T$ 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 three 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 possible 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.

Intrinsic Parameters

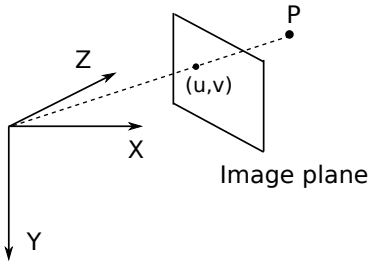


Figure 3.2: Intrinsic parameters

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 intrinsic parameters. These are the horizontal and vertical focal lengths f_x and f_y, principal point coordinates c_x and c_y and skew factor s. From these, the image coordinates u and v can be recovered by projecting the three-dimensional coordinates 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, and do not change during operation, 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 the homogenous image coordinates so that the last coordinate is 1.

3.1.2 Lens Distortion

The pinhole camera model does not perfectly describe the projection of the world onto the camera's image, however, as it neglects lens distortion. Real cameras use lenses, and these lenses slightly deflect the light rays moving through them, so it is not accurate to consider that light travels in a straight line though a pinhole. This deflection causes distortion in the image. We consider two types of distortion: radial and tangential distortion. This distortion can be modeled using the Brown-Conrady distortion model (using a third order approximation for the radial distortion, and a second order approximation for the tangential distortion):

$$x_d = x_u(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x_u y_u + p_2(r^2 + 2x_u) \tag{3.8}$$

$$y_d = y_u(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_2 x_u y_u + p_1(r^2 + 2y_u) \tag{3.9}$$

Where k_i and p_i are the radial and tangential distortion coefficients respectively, (x_d, y_d) is the distorted image point (where it is actually observed), (x_u, y_u) is the undistorted image point (where it would be observed if there was no distortion), and $r = \sqrt{(x_u - x_c)^2 + (y_u - y_c)^2}$ is the distance between the undistorted point, and the image center (x_c, y_c).
If we know $(k_1, k_2, k_3, p_1, p_2)$, we can invert equations 3.8 and 3.9 to obtain the undistorted points. [35]

3.1.3 Camera calibration

Both the distortion coefficients and the camera matrix (consisting of the focal lengths, principal point, and skew factor) are intrinsic to the camera. It would be useful to know them once and for all, as they do not change during operation.

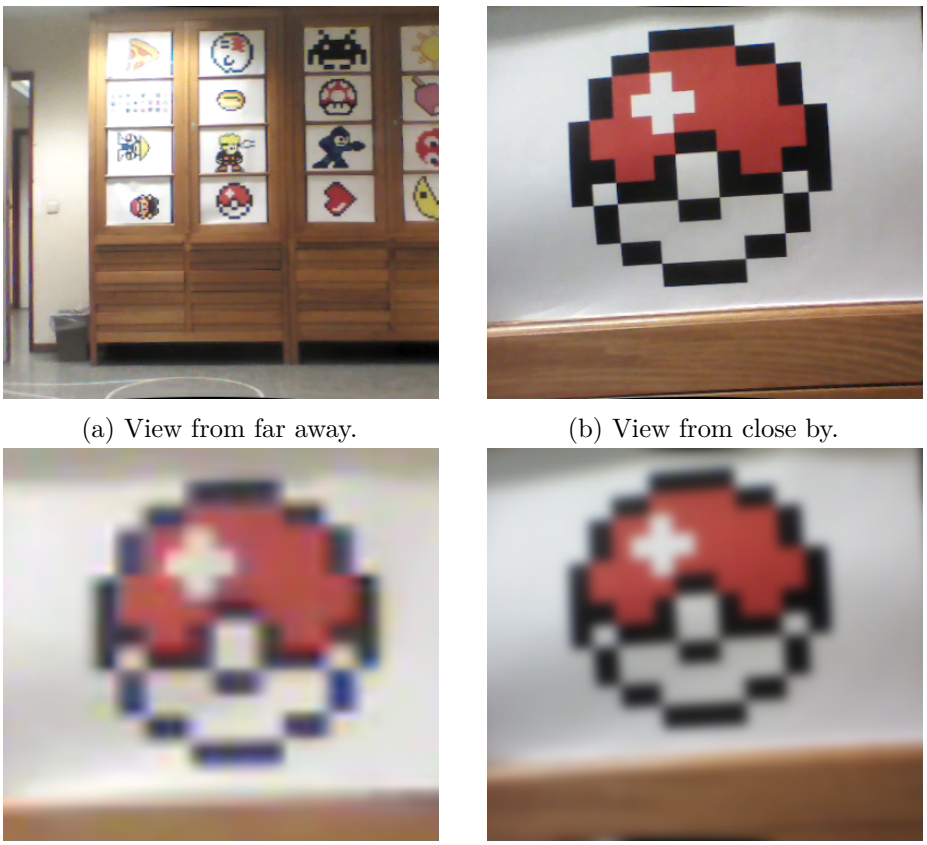
The process of camera calibration refers to the estimation of the distortion coefficients and camera matrix. It is typically done by looking at checkerboards placed at various points in the camera's field of view. The distortion coefficients are optimized such that the corners on the checkerboards appear on straight lines. The fact that the points of the checkerboard are part of the same plane, and that they form squares is used to estimate the camera's intrinsic parameters. There is a convenient ROS node called `camera_calibration` that does this.

To be able to use the pinhole camera model in all our algorithms, we will use another convenient ROS node called `image_proc` that takes a stream of distorted images as input, and uses the distortion coefficients that were previously estimated to produce a stream of undistorted images as output.

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. [22]

To illustrate this with an example, we look at figure 3.3. Images 3.3a and 3.3b have been observed and we would like to match them. Zooming in on image 3.3a gives 3.3c, but this image is quite different from 3.3b. If we look at the scale-space representation of 3.3b we might find a match at higher scales, when the image becomes blurred to look like image 3.3d. 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 3.3: 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:

$$\mathcal{H}(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix} \tag{3.10}$$

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 a 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 their response 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. [5]

3.4 SIFT Keypoint Descriptor

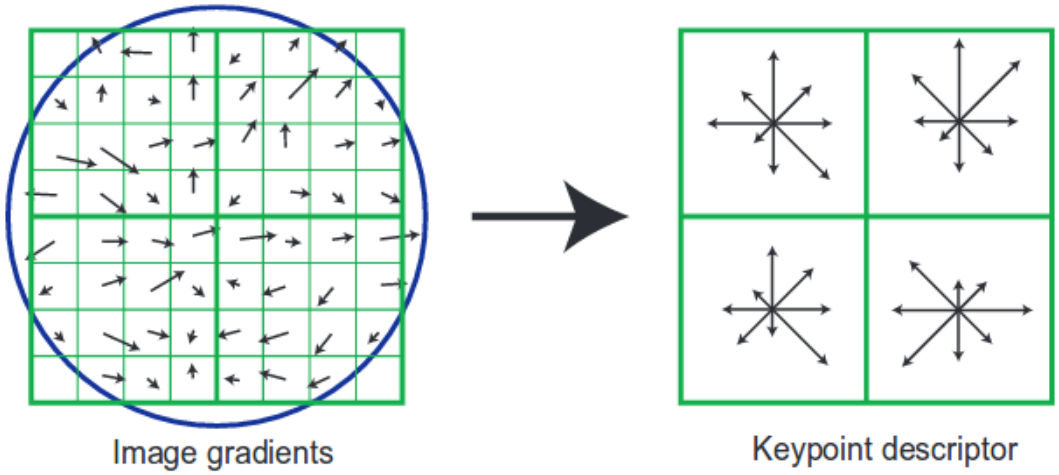


Figure 3.4: Illustration of a SIFT descriptor. Each subregion containing 4x4 points defines a histogram with 8 bins. Note that SIFT uses 4x4 subregions, not 2x2 like in this illustration. [24]

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 3.4 shows an illustration of the histograms. As there are 16 histograms of 8 bins each, the total length of the descriptor vector is 128. [25]

3.4.1 Limitations

The dimension of SIFT descriptors is 128, which is quite large. Having such a large dimension has the advantage of allowing the descriptors to be 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.

Another limitation is that the points detected are points of the image with a high intensity gradient. These points are often corners, with two distinctive regions, one inside and one outside the corner. The descriptor depends on the gradients of the image, which depend on both these regions. To be able to recognize the keypoints, the descriptors have to be similar independently of the viewpoint of the keypoint. This is the case when both regions of the corner are part of the

same plane, like in the case of a 2D drawing, but it is not the case when the inside of the corner is some object, and the outside is the background, as the background can change depending on the viewpoint. For example, figure 3.5 shows the same point seen from two different angles, where the descriptor would be quite different, because there is a different background

Figure 3.5: Two views of a same point, with a different background.

3.5 Keypoint Tracking

Because the above algorithm to extract a descriptor of a keypoint is quite slow (it takes approximately 413 ms to extract all descriptors from a new image, see table 3.1), 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 [18].

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 non-increasing, 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.

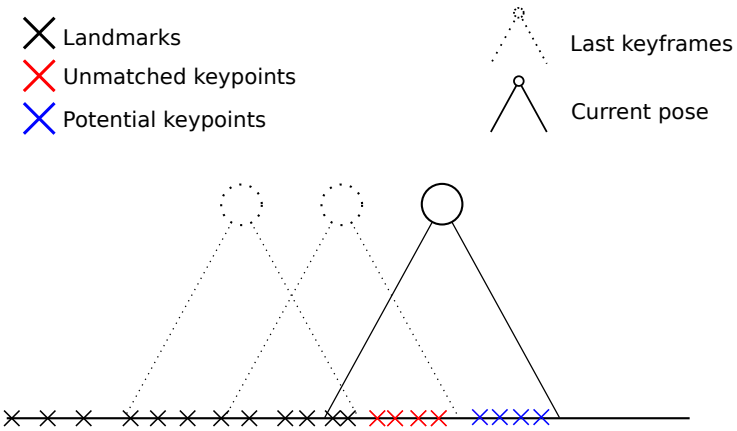


Figure 3.6: Delay between first observation of a keypoint, and mapping of this keypoint.

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 3.6 is an illustration of this problem, in a one dimensional world. 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 3.6, if the drone creates a new keyframe at its current position, the unmatched keypoints can become landmarks, and the potential keypoints (blue crosses) become new unmatched keypoints. If we had only used tracking since the last keyframe, however, we would not have detected those potential keypoints. As a result, when the drone continues advancing it would not have any unmatched keypoints from the previous keyframe, and it would 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 new 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. This is illustrated on figure 3.7.

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

Figure 3.7: Illustration of the hybrid between tracking and detection.

We can see on table 3.1 that our hybrid method takes about 55 ms per image. This is the time for both the tracking of the keypoints of the previous image, and detection and description of keypoints on the sides. Ideally, we would like this time to be below 33 ms, because our front camera has a rate of 30 FPS, so it sends an image every 33 ms. However, we do not need to use our hybrid at every new image, only when a significant part of the sides of the image contains no points from tracking. The rest of the time, we can use pure tracking, which runs in 6 ms. In the end we are alternating between tracking and our hybrid method, instead of between tracking and full detection. The end result is that we are more regular, both in the number of keypoints of any given frame, and in the computation time required per frame.

Task	Time per Frame [second[]
Keypoint detection	0.032
Keypoint description	0.413
Keypoint tracking	0.006
Hybrid	0.055

Table 3.1: Timings for the different tasks. The time for the hybrid method includes tracking, detection, and description.

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 a very close neighbor the rest of the time. 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. It is quite fast: to find amongst 666 keypoints of a keyframe the closest match for each of 502 keypoints of another keyframe, it took approximately 21 ms.

3.8 Summary

In the code, the computer vision node serves as an intermediary between the output of the camera and the input of the localization and mapping nodes. It transforms each image into a list of keypoints, each with their 2-dimension location on the image plane, and 128-dimension descriptor. To do this, it first undistorts the image, using distortion coefficients estimated in advance, then either tracks keypoints from the previous frame, and detects descriptors at the edges of the frame, or discards all keypoints of the previous frame and detects keypoints on the entire image. In both cases, descriptors for the newly detected keypoints are then extracted. Later, the localization and the mapping nodes can use keypoint matching to find keypoints in different frames that correspond to the same landmark.

In this part, we kept using SIFT descriptors and a SURF detector, as was implemented last year, but the OpenCV library contains procedures for working with many other types of descriptors and detectors, which were not tested this year. In future work on this project, it might be interesting to test using other types of keypoints, such as ORB, as the tests leading to the choice for the current SIFT/SURF combination were made before the current 3D framework existed. Figure 3.8 summarizes the computer vision part of the code in the form of a flowchart.

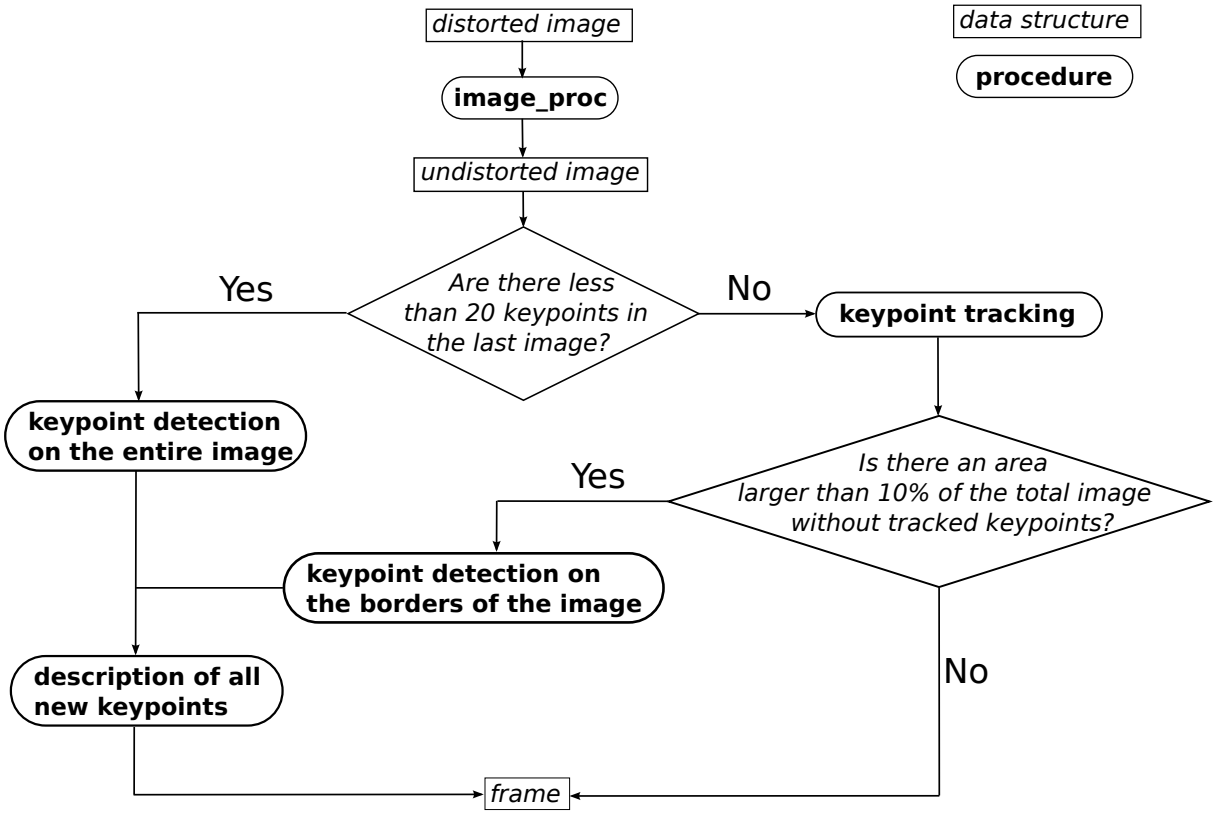


Figure 3.8: Flowchart of computer vision node

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

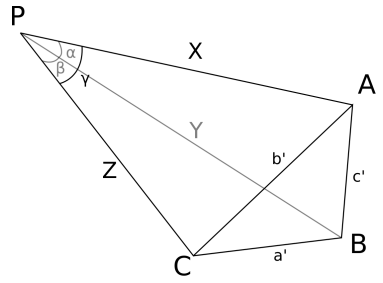


Figure 4.1: Triangles in P3P

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: $\widehat{APB} = \alpha$, $\widehat{BPC} = \beta$, $\widehat{CPA} = \gamma$ (see figure 4.1). 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 [15]. 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 if there exists a finite number of solutions, that number has to be less than or equal to 4. To eliminate the bad solutions, a fourth point correspondence is used. 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) [21] 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 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 required 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 [14].

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 Internal Pose Estimation

As we said in section 1.5.2, the Parrot's closed-source code does not give us access to the individual sensor readings, but gives us the output of its own pose fusion. The information given is:

- The linear speed in all three directions (obtained by fusing the speed estimate from visual flow of the bottom camera and from integration of the accelerometers),

- The three orientation angles (obtained by fusing the integrated rotational speed of the gyroscopes, an estimate of the gravity direction from the accelerometers, and the magnetometer)

- The altitude (from the ultrasound sensor)

Of this information, the altitude and the roll and pitch angles are very accurate and drift-free. The other measurements are either speed estimates that have to be integrated, or the result of integration themselves. As a result, they are subject to drift, so while they are useful to stabilize the drone, they can't be used as an absolute reference.

4.2.1 Visual Odometry from optical flow of the bottom camera

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 part of the closed-source firmware, so we only have access to its output and have to use it as a black box.

4.3 Pose Fusion

To obtain a final pose estimation, we will combine our estimation based on PnP and the estimation we receive from the drone. Our objective in doing so is for each source of information to compensate the shortcomings of the other ones:

- The result of PnP gives an estimate that is absolute with respect to the map, so the errors don't accumulate. However, because of the computations required to estimate the pose visually, estimates are not available frequently enough for the controller to efficiently stabilize the drone. Also, if the drone temporarily loses track of the map, this method gives no information at all

- The various sensor readings are available at a high frequency, but most of them only directly measure acceleration or rotational speed, and have to be integrated to give an estimate of the position and orientation. For this reason, errors accumulate over time, leading the drone to drift away. The exception to this is the ultrasonic sensor, which gives an absolute measurement of the altitude. Deducing the gravitational direction from the accelerometer readings also gives a good measure of the roll and pitch angles.

To use the best characteristics of both estimates, we use the simple pose fusion algorithm that was developed by last year's group:

- We use the altitude from the ultrasound sensor, and the roll and pitch angles from the IMU directly.

- Every time we receive an update of the visual pose estimation, we use it directly for the x, y and yaw estimates

- In between visual pose estimations, we use the position estimated by integrating the accelerometer and gyroscope data, integrating from the last visual pose.

- To account for the possible delay between when the drone sees an image and when we can estimate the pose using PnP, we save the displacement between each successive pose estimation along with their respective times in a queue, and add all displacements that happened since the image was seen to the visual estimation obtained from that image.

4.4 Summary

The localization task is carried out by pose estimation node, using the output of the sensors, and the result of PnP, which is computed by the mapping node. When the mapping node receives a frame (list of keypoints) from the computer vision node, it matches the descriptors of this frame with the ones of the map, and uses these correspondences to estimate the pose visually, as explained in section 4.1. The pose estimation node then uses the most recent visual pose estimate along with the speeds estimated from sensor readings, such as the speed estimated from optical flow (section 4.2.1) to obtain a fused pose estimate, as explained in section 4.3. This pose fusion is quite rudimentary, and could significantly be improved, for example by using an extended Kalman filter, such as in [10].

Figure 4.2 summarizes the localization part of the code in the form of a flowchart. It should be noted that only the pose fusion is carried out by the pose estimation node, the pose estimation using the map is carried out by the mapping node, as it is the node containing the map.

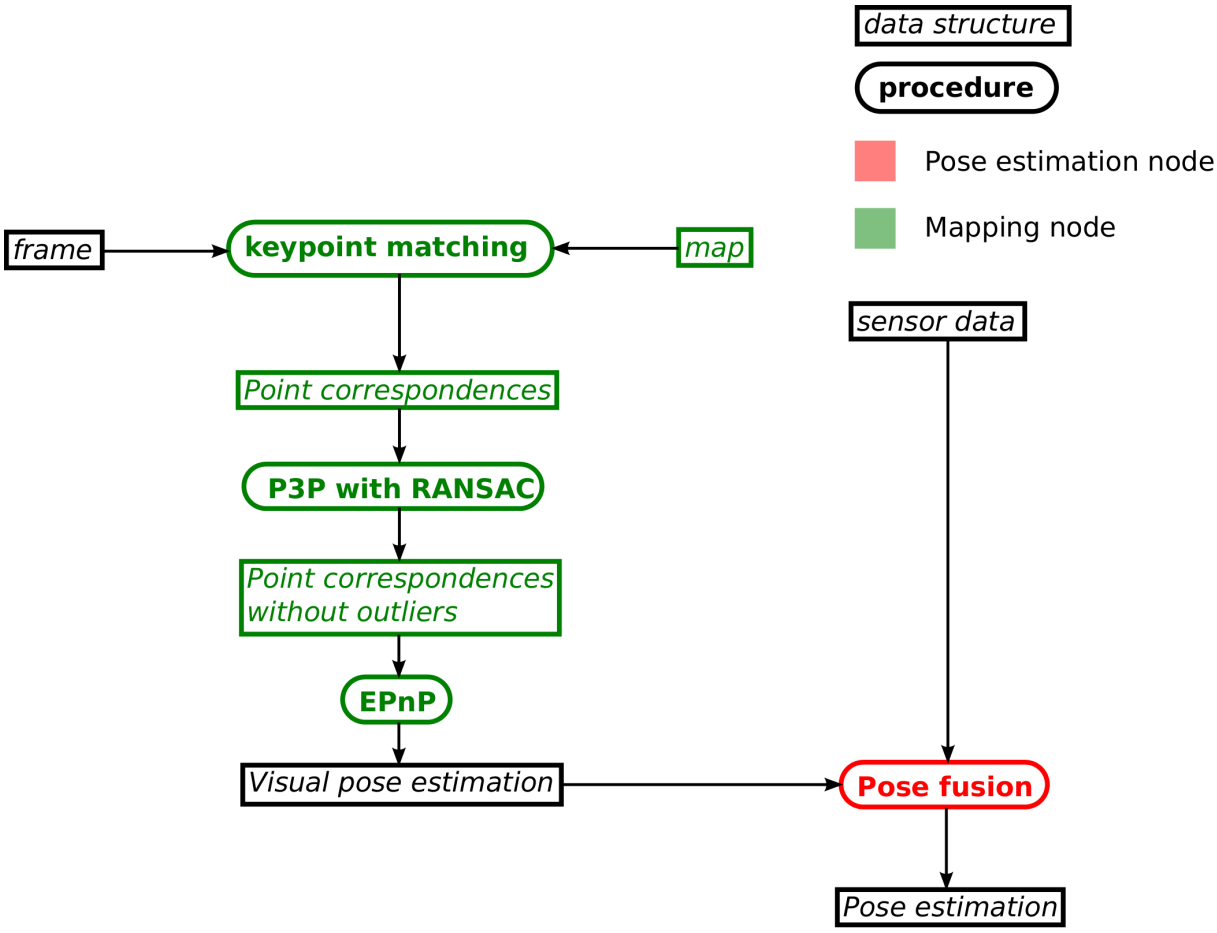


Figure 4.2: Flowchart of localization task

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 observations of any point 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.

Finally we will implement a mapping strategy, that decides when and how to grow or reduce the map, and how to initialize it.

5.1 Structure of the map

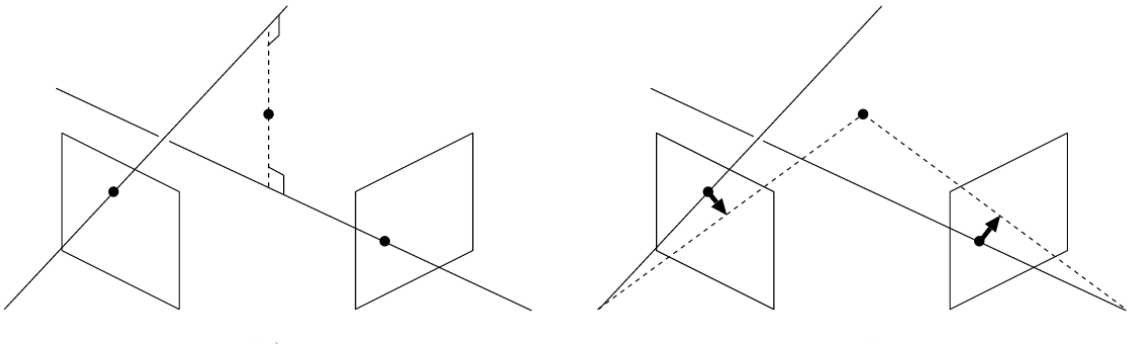
We will keep the keyframe-based structure for the map from the previous years, but this structure will need to be adapted so that each landmark does not belong to just one keyframe. The map will contain two main data structures: a list of keyframes, and a list of landmarks. Each keyframe will contain 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. For each observed keypoint of this list, we also save its descriptor, its 2D position in the image plane of the keyframe, and whether it corresponds to a landmark of the map, and which one. Each landmark will contain a descriptor, its estimated 3D coordinates, and a list of keyframes seeing it. The map can be seen as a bipartite graph, where the landmarks and the keyframes form the two sets of nodes, and where an edge is present between a landmark and a keyframe if the keyframe sees the landmark.

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.

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 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 point, 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 [17], computes the solution directly but requires finding the root of a 6th degree polynomial. Kanantani et. al.'s method [19] 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 (6.2), 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.

5.2.3 Angle between the rays

It is interesting (and will be useful) to note that the accuracy of the triangulation depends on the angle between the two rays reaching the point. In the most extreme case, one of the two points from which the landmark is seen is exactly between the landmark and the other point, like on figure 5.2a. Then the rays are parallel, the angle between them is 0° and the distance to the landmarks can't be determined. The same applies if the landmark is between the two points of view, and the angle is 180°, like on figure 5.2b. The optimal situation lies between these two extreme cases, when the two rays form an angle of 90° like on figure 5.2c. In practice, however, our scene is sufficiently far away so that this angle is always lower than 90°, so the larger the angle, the better.

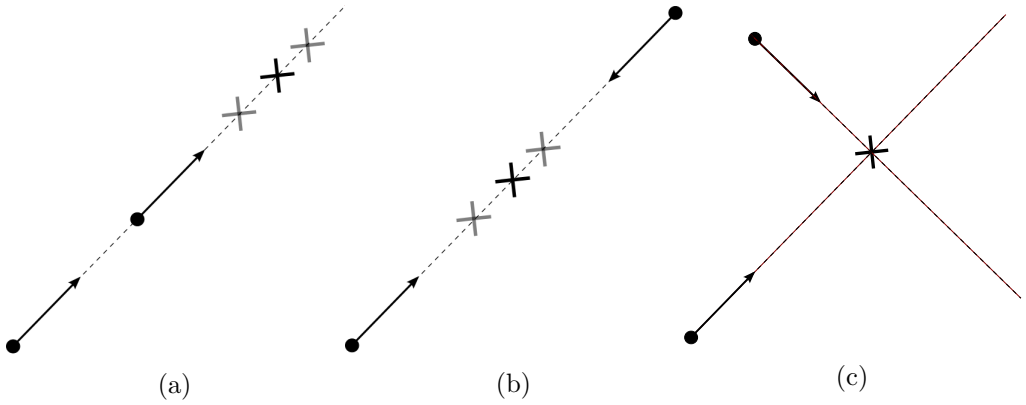


Figure 5.2: Effect of the angle between the rays

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. Because bundle adjustment uses new information to correct older parts of the map, it allows to reduce the accumulation of errors. 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 make it useable in real time.

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 $\mathbf{c}_k$ and their roll-pitch-yaw orientation angles $\mathbf{r}_k$. The 3D position of the landmarks will be called $\mathbf{p}_l$. For each observation of a landmark by a keyframe, we know at what 2D point on he image that point was observed. Let $\mathbf{i}_{lk}$ 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
$\mathbf{i}_{lk}$	$\triangleq$	Image coordinates of landmark l in keyframe k's image plane
		Decision Variables
$\mathbf{c}_k$	$\triangleq$	Position of camera center of keyframe k
$\mathbf{r}_k$	$\triangleq$	Roll-pitch-yaw angles of camera of keyframe k
$\mathbf{p}_l$	$\triangleq$	Position of landmark l

$\text{Proj}(\mathbf{p}, \mathbf{c}, \mathbf{r})$ is the projection operator. it transforms the 3D point coordinates $\mathbf{p}$ of any point into the 2D image coordinates of that point if it were perfectly observed by a keyframe located at position $\mathbf{c}$ with orientation $\mathbf{r}$. In addition of depending on the extrinsic camera parameters, ($\mathbf{c}$, and $\mathbf{r}$), this operator depends on the intrinsic (focal length, projection center, skew) 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 intrinsic 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 $\mathbf{p}_l$, the position of landmark l, we can re-project point l into the images it was observed from. For example, $\text{Proj}(\mathbf{p}_l, \mathbf{c}_k, \mathbf{r}_k)$ is the reprojection of point l into the image plane of camera k. 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. The objective then becomes:

$$\min \sum_{k \in \mathcal{K}} \sum_{l \in \mathcal{L}_k} L(\text{Proj}(\mathbf{p}_l, \mathbf{c}_k, \mathbf{r}_k) - \mathbf{i}_{lk}) \tag{5.1}$$

for some loss function $L(\cdot)$. We would like to use a squared loss function, as it has good computational properties, but it has the disadvantage of giving 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 equal to squared loss for values of x smaller than δ, and becomes 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

Bundle Adjustment is generally an unconstrained problem, but we will add some soft constraints to incorporate information we have from the sensors. We will add terms to the objective function that penalize keyframes whose altitude, roll, and pitch are far away from the ones measured during the keyframe creation. Because each of these measures is absolute, they are not subject to drift, so they should be accurate even after the drone travels large distances. Doing this will prevent the solution of the bundle adjustment taking unrealistic values.

Another advantage is that this will (softly) fix the scale. As we said before, monocular vision only gives information of the world up to a scale factor, and bundle adjustment is normally unable to recover the real scale of the map it builds. Enforcing a constraint on the altitude of the keyframes fixes this scale, on the condition that the altitudes of the different keyframes vary, because then it gives a constraint on the distance between keyframes. For this reason, making the second keyframe from a position directly above the first one is a good idea.

5.3.4 Solver

We have a least square problem of the form:

$$\min_x G(x) = ||F(x)||^2 \tag{5.3}$$

Where F is a highly non-linear, non-convex function as defined in 5.1. Because F is non-linear and non-convex, we can only hope to minimize it locally, using an iterative algorithm, so it will be very important to start from initial solutions that are close to the optimum. At each iteration, we want to find a step Δx that brings the objective $F(x + \Delta x)$ closer to zero. We use the first-order approximation of $F(x + \Delta x) \approx F(x) + J(x)\Delta x$, where $J(x) = \left[\frac{\partial F_i}{\partial x_j}\right]$ is the Jacobian of F. Deriving 5.3 with respect to Δx with this approximation and setting it to zero gives the Gauss-Newton update:

$$\left(J^T J\right) \Delta x = -J^T F \tag{5.4}$$

By adding a regularization term λI to equation 5.4, we obtain Levenberg's update:

$$\left(J^T J + \lambda I\right) \Delta x = -J^T F \tag{5.5}$$

Marquardt later improved this update by replacing λI by $\lambda \texttt{diag}(J^T J)$, for faster convergence when λ is large.

When the value of λ is very small, the Levenberg-Marquardt update is quite close to a Gauss-Newton update, and when it is large, it becomes close to a gradient update. The Levenberg-Marquardt algorithm can be seen as a hybrid between the two. After each iteration, λ is increased if we are getting closer to a minimum, or decreased if the iteration results in a worse value of the objective function. As a result, when the 2^{nd} order approximation is good, the method is close to a Gauss-Newton method, and very fast, and when it is bad, the method reverts to a gradient descent for stability.

To measure how good the approximation is we use ρ, a metric to compare the gain in the objective function with the one we would have gotten if the approximation was exact:

$$\rho = \frac{G(x) - G(x + \Delta x)}{G(x) - ||(F(x) + J(x)\Delta x)||^2} \tag{5.6}$$

After each iteration, if ρ is above some threshold, we perform the step : $x_{\texttt{new}} = x + \Delta x$ and λ is decreased by some factor, if not, we do not update the value of x, and λ is increased. We stop the algorithm when either the gradient, or the relative update of the parameters ($|\frac{\Delta x}{x}$) falls below some threshold. [16]

5.4 Mapping Strategy

The last remaining task of the mapping task is to put all these elements together to build the mapping algorithm. This algorithm needs to decide how to initialize the map, when to create new keyframes, when and how to run bundle adjustment, and how to limit the growth of the map.

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 place the origin of our reference frame at the pose of the drone where it creates the first keyframe, with the x direction pointing forwards, the y direction to the left, and the z direction upwards (this is the coordinate system used by `ardrone_autonomy`). The first keyframe will serve as the anchor of the map: its position will never change, and will always stay at the origin.

A single 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 create any landmarks, 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 displacement between the two first keyframes. 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.

5.4.2 Local and global bundle adjustment

As we will see in chapter 6, the time taken for bundle adjustment will be one of the major obstacles of this work. This time taken depends on the number of points and keyframes concerned, so as the map grows, it will become impractical to run bundle adjustment on all of it. For this reason, we will only solve bundle adjustment on a subset of the map, considering only the most recent keyframes, and the points seen by them. This will allow to put a cap on the size of the problem that has to be solved each time a keyframe is created. Confining bundle adjustment to a local window means that errors will still accumulate over time, but still much more slowly than with no bundle adjustment. To solve this, we can occasionally run bundle adjustment on the whole map, which would be especially useful if the drone revisits a previously seen location.

Local Bundle Adjustment

Every time a keyframe is created, we follow this creation with a round of local bundle adjustment, to correct the error due to the inaccuracy of this keyframe's position. When running local bundle adjustment, we only solve a sub-problem of the full bundle adjustment. In this sub-problem, we only consider the n last keyframes, and the landmarks seen by at least two of those n keyframes.

We will set the poses of the two oldest keyframes considered in the local bundle adjustment as constant. This is necessary to prevent drift. It there was no keyframe set as constant, the entire map could be shifted with no effect on the objective function. Likewise, if there was only one keyframe set as constant, the entire map could be scaled by a factor around the constant keyframe, with no change to the objective function. The two constant keyframes serve as anchor of the local map, fixing its position within the global map. Choosing the oldest keyframes to be set as constant ensures that the constant keyframes have already been through some $(n-2)$ rounds of local bundle adjustment, so we can expect their position to already be close to the correct one, or at least to be locally consistent with the map.

Global Bundle Adjustment

As local bundle adjustment only considers a sub-problem, its solution is not optimal with respect to the full problem. When the drone has processor time available, we can improve our solution by running the algorithm on the full problem. When doing this, we are modifying all keyframes except for the first one, the anchor of the global map. The keyframes that are normally kept constant during local bundle adjustment are also modified, correcting some of the accumulated error that was made until there. However, errors will still accumulate as the drone moves away from the global anchor. These accumulated errors can be corrected if there are loops in the drone's trajectory, but they do not prevent the drone from exploring because the map stays locally consistent.

Loop Closure

Each landmark couples the positions of all keyframes seeing them: modifying the position of one keyframe changes the optimal position of the landmark, which changes the optimal position of the other keyframes. We can consider keyframes to be neighbors if they see a common landmark. Having keyframes that are far away from each other, in the sense that many intermediate keyframes are required to go from one to the other, traveling from neighbor to neighbor, means that the two keyframes are very loosely coupled. The looser keyframes are coupled from the first keyframe, the more they are subject to drift, as the errors accumulate between them.

Using local bundle adjustment further increases this drift, as keyframes whose positions are not simultaneously optimized during bundle adjustment are only coupled through the anchors of the

local bundle adjustment, not directly though point correspondences. When running global bundle adjustment, this drift can be corrected, especially when the keyframes are strongly coupled. The best situation for this is when the drone makes a full loop, and revisits locations seen by much older keyframes. The newest keyframes then become neighbors to the oldest, and global bundle adjustment can correct all the errors accumulated during the loop, resulting in a completely consistent map.

5.4.3 Rejecting outliers

When looking more closely at the results of bundle adjustment, we find that despite the use of a robust loss function (see section 5.3.2) a large part of the errors comes from a small number of points. Figure 5.3 shows the repartition of contribution to the objective function between the landmarks (a logarithmic scale is used for the x-axis). 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 are bad matches, and 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.

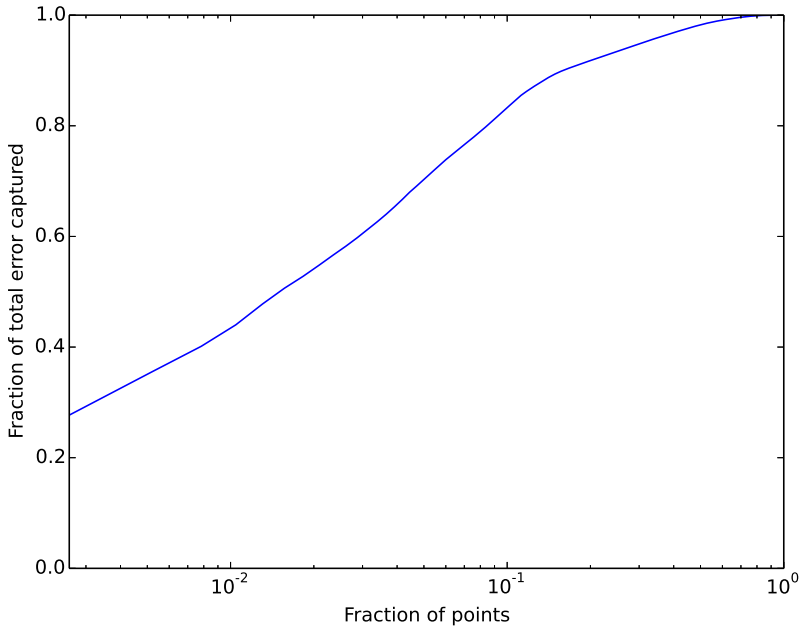


Figure 5.3: Repartition of error among points

To solve this problem, after every bundle adjustment pass, we eliminate points whose error is higher than some threshold to be defined. Because the number of keyframes seeing each individual landmark varies, we should look at the error per observation of each point. It is also normal that the error per observation becomes larger for points observed by more keyframes, because every time an observation is added, the location of the point will change a bit, and its position will become less optimal with respect to the keyframes that already saw it.

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 per observation before removing it.

As we will see in 6.3.1, removing outliers greatly improves the performance.

5.4.4 Choosing when to create keyframes

Choosing when to create keyframes was the most challenging task of this work. It is not a simple tradeoff between computation time and accuracy, as many (sometimes contradicting) points have to be kept in mind:

- Landmarks have to be observed from at least two keyframes before they can be used, so it is important to create keyframes sufficiently in advance
- If two keyframes are too close to one another, the lines going from landmarks to these two keyframes will almost be parallel, causing a large uncertainty on the landmark's position
- It is important to have a good pose estimate at the moment a keyframe is created, as this estimate will be the initial value of the keyframe's pose
- Because when we add a keyframe we run a local bundle adjustment on the last n keyframes, if the keyframes come in rapid succession, errors will accumulate faster, as there are more intermediates between the first and last keyframes.

The choice of when to create a new keyframe has to be made heuristically using any of the measures we have available. Amongst the quantities considered for this choice are:

- The time elapsed since the last keyframe was created
- The distance to the position where the last keyframe was created
- The distance to the position of other keyframes
- The number of mapped points currently in the field of view
- The number of unmapped points currently in the field of view
- The number of inliers the last times RANSAC was run
- Large areas in the field of view not containing any mapped points

After many trial-and-error tests, we came up with the following heuristic:

- If bundle adjustment is currently running, don't create a keyframe
- If the current position (disregarding orientation) is closer than 10 cm to the last keyframe, don't create a new keyframe
- The is no area larger than 20 % of the image on any of the 4 sides that contains no RANSAC inliers
- If any of the following conditions is true, create a new keyframe:
  - The current position (disregarding orientation) is farther than 70 cm from the last keyframe
  - An area larger than 33 % of the image (from any of the 4 sides) contains no RANSAC inliers
  - The current image contains less than 40 RANSAC inliers

The reason we use the distance between poses, disregarding orientation, is that if one keyframe results from the drone performing a pure rotation from another keyframe, the rays between points of the scene and the keyframes will be close to parallel in the reference coordinate system, and cause resulting landmarks to have a large uncertainty. For this reason, it is also important that the path-planning node (not worked on in this thesis) avoids pure rotations.

5.5 Summary

Every time the mapping node receives a new frame from the computer vision node, it decides whether a new keyframe is needed, using the heuristic described in section 5.4.4. If a new keyframe is needed, the frame is transformed into a keyframe, the list of keypoints, each with their descriptor and 2D position on the image is saved, and the estimated position of the drone at the moment of creating the keyframe is assigned to the keyframe. The descriptors of the new keyframe are then matched with the descriptors of already mapped landmarks and with the descriptors of currently unmatched keypoints in older keyframes. If there is a match between two unmatched keypoints in different keyframes, a new landmark is created by triangulating its position with optimal correction (see section 5.2).

For each keypoint in each keyframe, we also save whether it corresponds to any landmark of the map. After the keyframe is created, we decide whether to run local or global bundle adjustment, as explained in section 5.4.2. We then run bundle adjustment (section 5.3), either on the entire map if we decided to run global bundle adjustment, or on the last n keyframes and the landmarks seen by at least two of those if we decided to run local bundle adjustment. Running bundle adjustment allows to use the matches between keypoints to correct the estimations of the positions of those keypoints and the keyframes seeing them.

Figure 5.4 summarizes the mapping part of the code in the form of a flowchart. Bundle adjustment runs in a separate node, so that it doesn't block the entire mapping node.

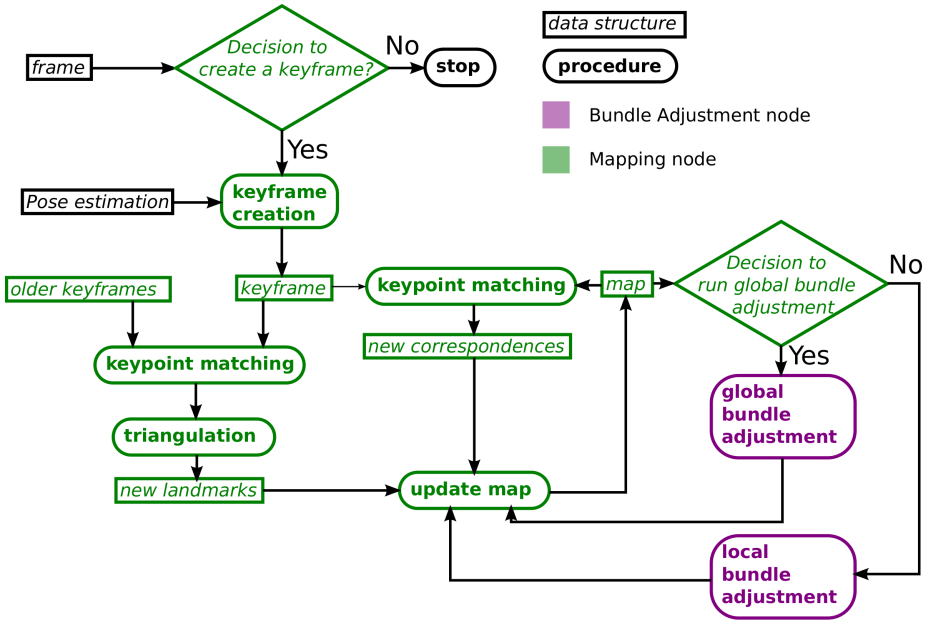


Figure 5.4: Flowchart of the mapping task

Chapter 6

Results

In this chapter we will evaluate the results of our final algorithm. To do so, we first create a test on which we can try out our algorithm, and measure its performance. We will begin by evaluating the mapping algorithm without the mapping strategy, by only initializing a map and testing it out. We will use this simple test to tune the parameters of bundle adjustment. Afterwards, we will evaluate the full mapping algorithm.

All experiments were carried out with the drone on a mobile platform. We did not work on the drone's controller or path planning, so to focus on the computer vision and mapping parts of the project we decided to move the drone manually instead of making it fly.

6.1 Evaluation procedure

We will implement two tests to evaluate the quality of the map creation, the accuracy test, and the robustness test.

Accuracy test

During the accuracy test, 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 6.1.

The evaluation is performed as follows:

1. The drone is placed in a predefined and known position on a table (position A in figure 6.1). 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 4, then back to 1 on figure 6.1). 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. While the drone is moved from point 4 to 1, its camera is obstructed, to test whether it can recover from being lost. At each point, 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.

Robustness test

In reality, the drone does not know its exact position when making keyframes at points B, C, and D (position A is defined as the origin). To emulate this, we implement a second type of test: the robustness test. The robustness test works exactly like the accuracy test, with as only difference that there is some error on the positions that are communicated to the drone at the points B, C and D. This serves to emulate the fact that the drone only has an estimate of its position when building the map and does not know it exactly.

We do not yet know what a realistic error would be on the position of the drone, so we will choose one arbitrarily, aiming to be pessimistic. At each of the three uncertain poses, we will do two of the following:

1. add 30 cm to the x-coordinate.

2. add 30 cm to the y-coordinate.

3. add 18° to the yaw.

Performance measure

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 estimated position and its real position when it is at points 1, 2, 3, 4, then back at 1. The average angle error is the average absolute value of the difference between the estimated yaw angle and the real yaw angle at each of these positions. We do not take the errors on the altitude, as well as on the roll and pitch angles into account, 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 the known positions, as we do not have equipment to measure its real position during those displacements.

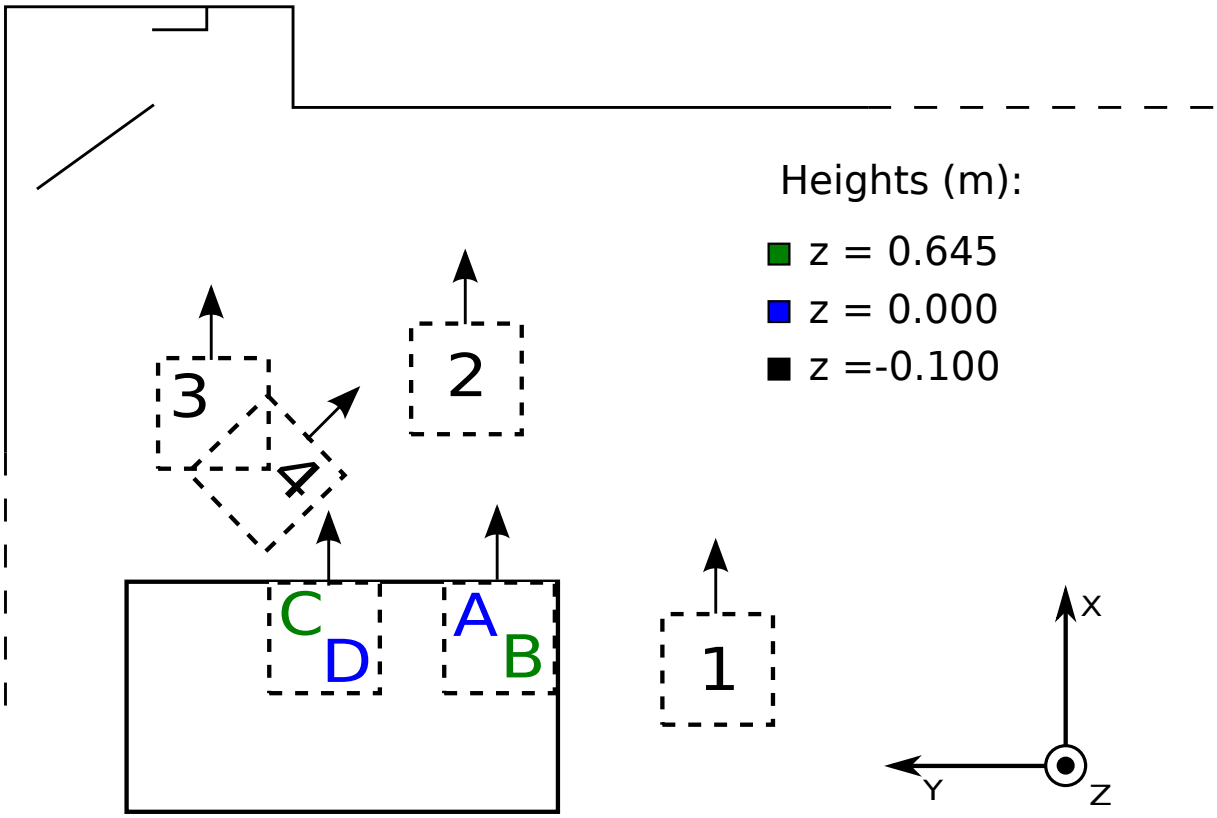
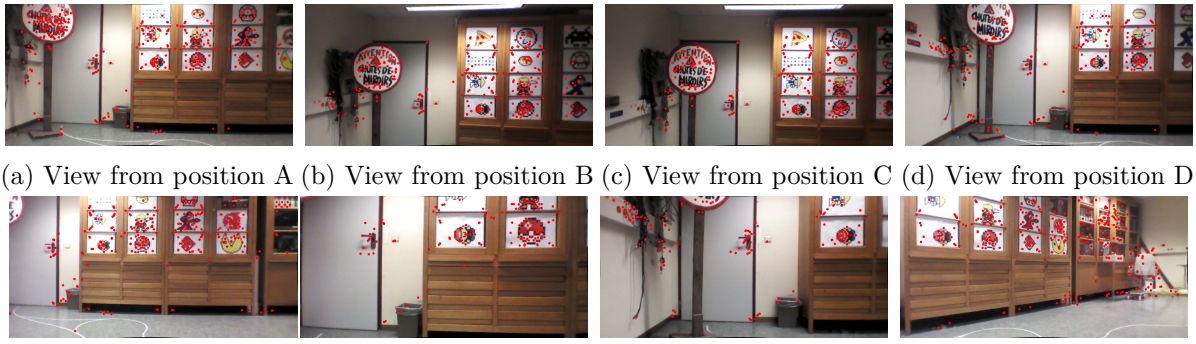


Figure 6.1: Different positions of the drone during the evaluation



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

(e) View from position 1 (f) View from position 2 (g) View from position 3 (h) View from position 4

Figure 6.2: Views during the evaluation

6.2 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 on both tests can be seen on table 6.1.

Table 6.1: Comparison between triangulation methods

	Accuracy test		Robustness test	
	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 correction 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 it gives slightly better results, 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 inexact. That is the reason why we need bundle adjustment.

6.3 Bundle Adjustment

Table 6.2: Performance of Bundle Adjustment

	Accuracy test			Robustness test		
	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	0.17	3.8°	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. 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 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 runs 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, because 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 ideas to keep this computation time low, while still having a good enough performance.

	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.

We already notice the difference in the time it takes for the robustness and accuracy tests. Bundle adjustment is an iterative algorithm that seeks to minimize a certain cost function. Because during the accuracy test we start from closer to the solution, it requires fewer iterations to correct the errors, so the better our position estimates, the faster bundle adjustment will run. This explains why bundle adjustment is takes more time for the robustness test than for the accuracy test

6.3.1 Tuning the Bundle Adjustment

There are many different parameters for bundle adjustment that can be tuned. We will try to find the best balance between performance and speed. Because there is a high variance on the performance on the tests, we will carry out each test multiple times to give an estimate of the performance of each set of parameters, and compare both the average performance, and the worst-case performance of the different sets of parameters. During this tuning phase, we will only perform the robustness test, as the performance on the accuracy test is already quite good, and does not vary much between the tests.

Outlier threshold

After each round of bundle adjustment, we consider points to be outliers if their reprojection error is above some threshold. If an outlier is seen by only two keyframes, it is removed from the map. If it is seen by more, then we only disregard the last observation of this landmark, because if it was still present in the map, that must mean that it wasn't an outlier before that last observation was taken into account. The loss function we are using (Huber's loss) is already designed so that outliers' contribution to the objective function is not too high, so it makes sense to tune its parameter δ at the same time as the threshold for removing points considered outliers.

Convergence of the solver

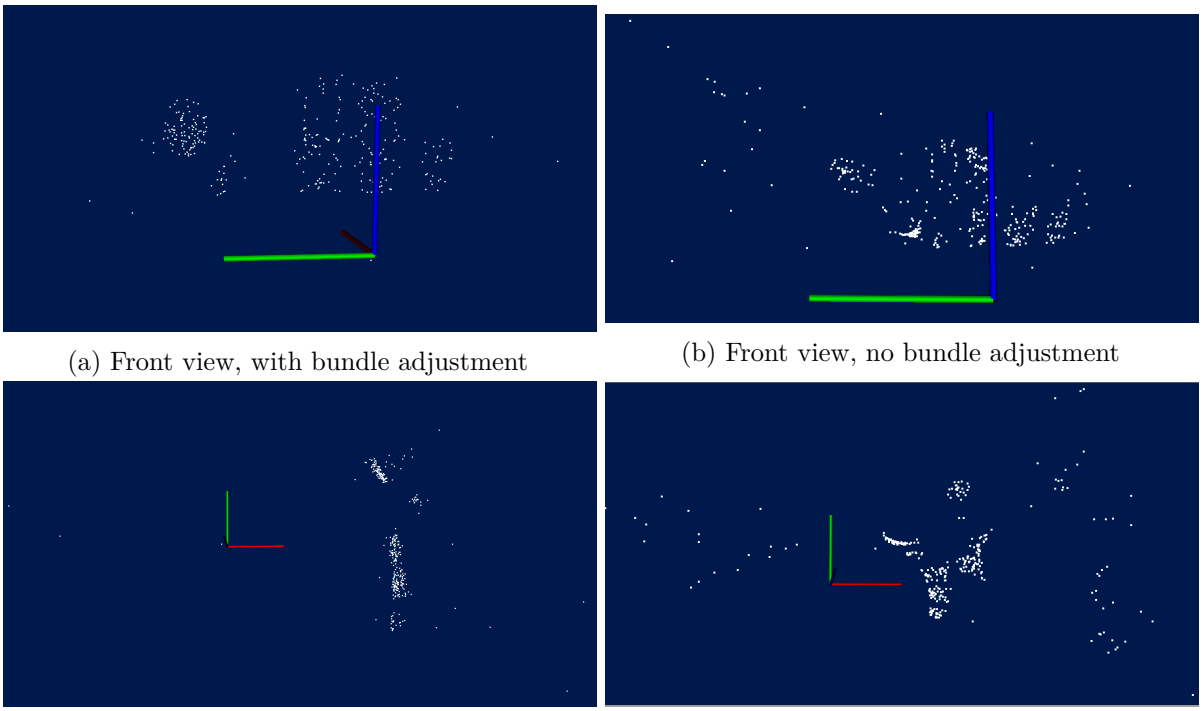
We can also tune the convergence criterion of the solver that solves the bundle adjustment problem. We will stop when the ratio of the change in the objective function to the value of this function arrives below some threshold. This ensures that the criterion scales with the problem, which is important 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 this gives very bad results, and takes quite a long time.

After this tuning, we are able to bring the average error during the robustness test down to the following values:

Table 6.4: Performance after tuning the bundle adjustment

	Distance (m)	Angle	Time (s)
Before tuning	0.17	3.8°	1.836
After tuning	0.11	2.0°	1.825

6.3.2 Final results of bundle adjustment



(a) Front view, with bundle adjustment

(b) Front view, no bundle adjustment

(c) Top View, with bundle adjustment

(d) Top view, no bundle adjustment

Figure 6.3: Point clouds after initialization during the robustness test

Figure 6.3 shows the map created by the drone during the robustness test, with and without bundle adjustment. With bundle adjustment, we can recognize the sign, the cabinet, and the door handle from figure 6.2, and from the top view we can see that these structures are quite planar. Without bundle adjustment, however, the map is unrecognizable.

Figure 6.4 compares the estimated trajectory of the drone during the robustness test, with the real reference positions. We can see that the largest error made is about 20 cm of distance in the xy-plane and 5° of yaw-error. For the applications we are considering, these errors are at the limit of what is acceptable, because the drone we are using is 51 cm wide with its indoor hull, and it could just not pass a door (90 cm wide) with 20 cm on either side.

Despite being on the limit in the worst case, we are satisfied with this result, because the robustness test only tests the general useability of bundle adjustment and not of the entire mapping algorithm, as no new keyframes are created after the initialization. On Figure 6.4, we see that the largest errors were made at position 3, more than 80 cm away from the closest keyframe, and more than 80 s later. We can also see on figure 6.2g that the overlap between its field of view and the ones at positions A, B, C and D is quite small. During real operation, the drone would have created more keyframes during its displacement towards position 3, so when arriving there, it would have more landmarks in its field of view. For this reason, we believe

that the errors made at positions 1, 2, and 4 are more indicative of the real performance of our map.

The straight lines between positions 4 and 1 correspond to the time during which the camera of the drone is blocked. During this time, the drone does not publish a visual estimation of its position. We can see that after this occlusion, the drone is capable of finding its position again.

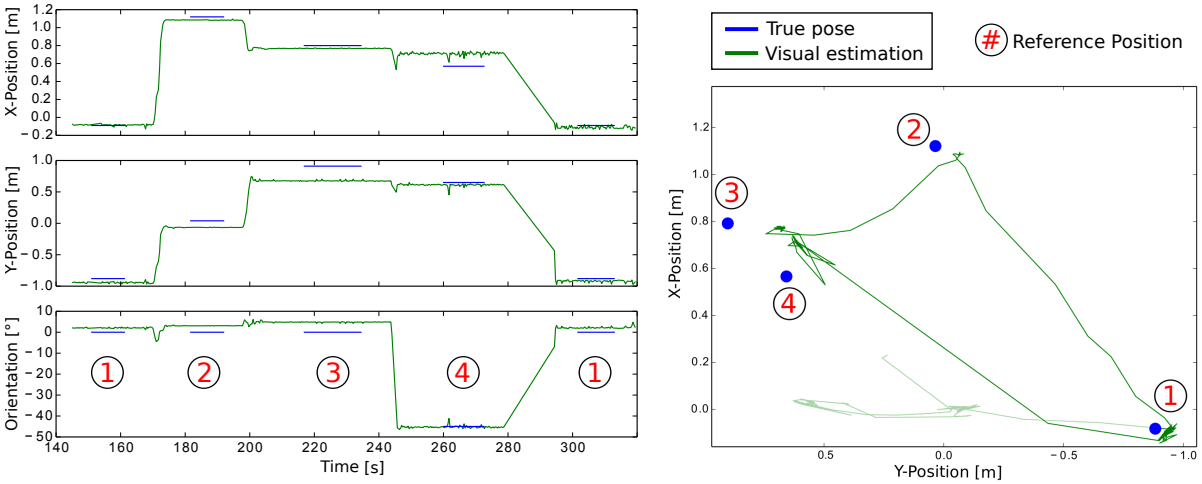


Figure 6.4: Result of robustness test with bundle adjustment. The four positions correspond to the ones of the evaluation, as shown on figure 6.1. The grayed-out part of the trajectory corresponds to the initialization of the map (positions A through D)

.

6.4 Mapping strategy

Now that we tested how well bundle adjustment works to create a map, and that we convinced ourselves that it is capable of recreating the 3D geometry of the world, we need to test the mapping algorithm itself (which was explained in section 5.4.4). Unfortunately, the limited access to the sensors when the drone isn't flying means that it is very difficult to recreate realistic conditions regarding the information the drone has. Regardless, for simple trajectories, such as pure translations, the algorithm works quite well. Unfortunately, for longer trajectories, the scale problem ends up making the tests fail.

6.4.1 Difficulties for testing the mapping strategy

It is much more difficult to test our mapping strategy than is was to test the mapping itself, because we do not know in advance when the drone will create keyframes. This poses a problem, because our drone isn't flying during the tests, so we don't have access to the ultrasound. As a result, we have to either manually communicate its altitude to the drone during the entire flight, or use the drone's height estimation from PnP as if it were the ultrasound's estimation. Both approaches have disadvantages.

If we communicate its altitude manually to the drone, that means the drone has to stay at the same altitude during the entire test, because we would not be able to give it its exact altitude during transitions from one to another. If the drone stays at the same altitude throughout the test, the soft constraint we put on the keyframes' altitudes don't fix the scale anymore, because they do not put any constraint on the distance between keyframes. In that case, the only thing fixing the scale are the two keyframes that are held constant during local bundle adjustment,

but they are also subject to drift.

If we use the drone's visual estimation of the altitude as if it were the ultrasound sensor, the scale also drifts, because the visual estimation depends on the map, so errors accumulate.

6.4.2 Short trajectories

Despite the problems explained in section 6.4.1, the heuristic developed in section 5.4.4 works quite well when the drone carries out short trajectories. On figure 6.5, we can see the resulting map from the drone moving in a straight line to the right after initialization. During initialization, only the part to the left of the black line was visible by the drone, and as it moved away, it created keyframes, and proceeded with local and global bundle adjustment as explained in section 5.4. At the end of its trajectory, none of its field of view contained parts of the initial map. We see that it worked, well, because the scene seems planar on the map, as it is in real life. However, there are still many points that seem to be outliers that weren't removed.

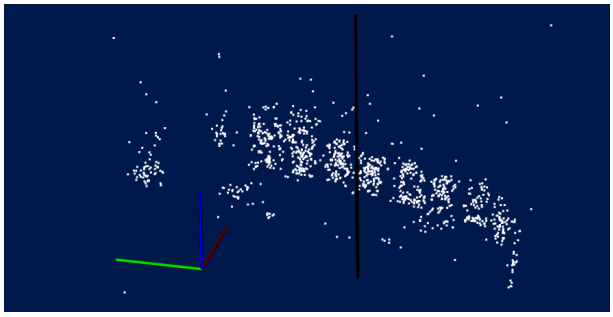


Figure 6.5: Map resulting from free exploration

6.4.3 Long trajectories

For longer trajectories, the performance quickly deteriorates. We are convinced that the reasons explained in section 6.4.1 are to blame for this deterioration. Unfortunately, with the available equipment, we were not able to devise a test that realistically gave the drone an altitude measurement that corresponds to one measured by the ultrasonic sensor, while also varying this altitude during the test. We are certain that the results would be much better on such a test.

We attempted to make the drone carry out a full loop, to be able to run global bundle adjustment at the end of the loop, correcting any accumulated errors. Unfortunately, as we moved away from the initial keyframes, the accuracy of the map became poorer, until it could no longer be used by the drone.

6.4.4 Loop closure

The attempt to carry out a full loop to test out the mapping strategy failed, but we would still like to see if our algorithm is able to successfully carry out loop closure.

To show that loop closure does work, we finally carried out a test without the mapping strategy, where we manually told the drone when to create keyframes during the entire experiment. This allowed to resolve the problems cited in section 6.4.1, because as we tell the drone when to create a keyframe, we can also give it its exact altitude, even if this altitude is not constant. By working this way, we were able to carry out a full loop, mapping an entire room in the process. The

resulting final map can be seen on figure 6.6.

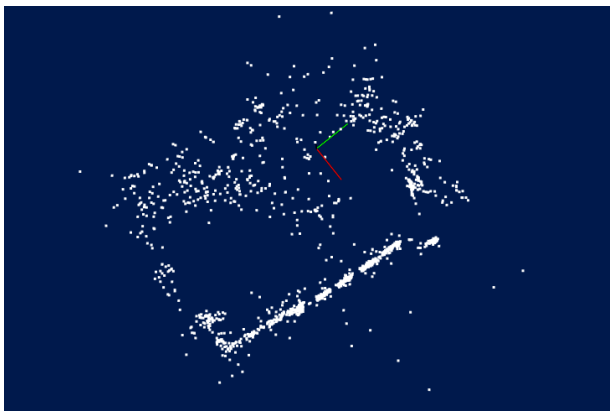


Figure 6.6: Map resulting from a full loop, where keyframes were added manually (top view)

Chapter 7

Conclusion

In this master's thesis, we achieved our goal of allowing true 3-D SLAM. We are able to build a 3D map based on observations by the monocular camera. To do so, we successfully:

- Researched the state of the art for monocular visual SLAM
- Redesigned the keypoint acquisition step to use a more effective hybrid of keypoint tracking and detection
- Implemented a state-of-the art method for optimal correction when triangulating keypoints
- Implemented a bundle adjustment step that re-optimizes the map with new information
- Implemented a mapping strategy that decides when to create keyframe, and when to re-optimize the map with global bundle adjustment

The code is now able to build a 3D map using a monocular camera, without making any assumptions regarding the location of keypoints. Using this map, we have shown that the drone can estimate its position very accurately, although at a low frequency. We are able to solve this frequency problem by fusing the visual pose estimation with the IMU, which is available at a much higher frequency, using the pose fusion algorithm created last year.

The full mapping algorithm works well for short distances, and we demonstrated that it is conceptually possible for it to work for long trajectories. Unfortunately, due to the unavailable sensors when out of flight, we were not able to show that our full mapping algorithm works during long trajectories. We did show that it could theoretically work during long trajectories by showing that it works when we replace the mapping strategy by a human decision. We are persuaded that if we had access to an ultrasonic sensor, the full mapping algorithm would work, without a human decision maker.

We will look at the two main limitations of the current code, and at some possible solutions to these limitations. Then we will look at some ideas to make the current implementation perform better. Finally, we will look at some new directions the UCL's drone project could take in the future years, in particular, what is made possible now that we implemented 3D mapping.

7.1 Limitations

The main limitations of our solution, is that with no ultrasound, the mapping algorithm fails on long trajectories, and that the speed of exploration is limited by the time required to run bundle adjustment. We will look at some ways to overcome these limitations.

7.1.1 Failure on long trajectories

In chapter 6 we were not able to show that our algorithm works during long trajectories. However, we did show that it works on short trajectories, and that if we replace the mapping strategy by a human decision of when to create keyframes (to be able to tell the drone its altitude when keyframes are created), it works quite well. For these reasons, we are convinced that with an available ultrasonic sensor, the current algorithm would work on long trajectories, and would be able to carry out loop closure.

Possible solutions

It would greatly ease development to have access to all sensor data individually, and at any time. This is impossible with the Parrot AR.Drone, because the software running inside is closed-source. Only a change in hardware can really solve this problem, but several solutions are possible:

- Use another drone, one that has open software, and that is modular so that sensors can be added and removed at will.
- Attach a new, independent, small ultrasonic sensor to the bottom of the drone, that can communicate its altitude to the drone at all times.
- Install an external sensor, such as a Kinect™, to measure the position of the drone, and use it to simulate the ultrasonic sensor. This would have the added benefit of giving a ground truth of the drone's position at all times, even during displacements.

Computation time of bundle adjustment

The time taken by bundle adjustment varies widely depending on the number of keyframes considered, the number of points considered, the quality of the initial solution, and the number of bad point matches. In the worst cases, we found that local bundle adjustment could take more than 1 s, which severely impacts the speed at which the drone can explore, because local bundle adjustment is run every time a keyframe is created, and keyframes have to be created frequently enough for all points to be seen from at least two different positions.

Possible solutions

Some of the possible ways to reduce the computation time of bundle adjustment are:

- Further reduce the map to keep only the best points.
- Reduce the number of keyframes considered during local bundle adjustment. Currently, more keyframes are required to reduce the scale drift (by setting two keyframes constant), but if we had an ultrasound available, it could be used to prevent scale drift, reducing the need for many keyframes during local bundle adjustment.
- Use a graphics card to parallelize computations.

7.2 Possible improvements of our implementation

Here are some ideas that could improve the results of our work, but that weren't tried out:

Further sanitize the map

As we could see on the various images of point clouds, many points remain that are behind the wall of the room in which the tests were carried out, which indicates that at least these points don't correspond to real world locations. One idea we did not try out is to remove landmarks that have been present in the map for some time (to be defined) but that were never (or rarely) inliers for RANSAC during localization. This would indicate that they are not useful, and can be removed.

Changing the type of keypoints

SIFT keypoints with a SURF detector were chosen last year during one of the master's theses that was written that year. The student who made this choice found that it has the best compromise between performance and complexity. Their have been some important changes to the implementation since then, so it might be interesting to do some experiments with different types of keypoints. For example, ORB keypoints seem to have worked well for other, similar projects.

A survival-of-the-fittest type strategy for keyframes

The idea for a survival-of-the-fittest type strategy for keyframes comes from ORB-SLAM [26]. With this strategy, we would create many more keyframes, and then remove all but the "best" ones. This way, we could aim for keyframes placed in space such that there is enough overlap between them, but also such that they are not too close to each other. This would make the need for deciding in advance when to create keyframes less important, moving the decision to later, when there is more information available.

A separate node for global bundle adjustment

Because bundle adjustment takes a long time, it is done in a separate node. The mapping node sends part of the map to the bundle adjustment node, which optimizes this part of the map and sends it back to the mapping node. When the mapping node decides to run global bundle adjustment, it simply sends the entire map to the bundle adjustment node. Having a separate node just for global bundle adjustment would allow to run global bundle adjustment in the background permanently.

7.3 Next steps for this project

There are many different directions that can be taken for the next steps of this project. Here, we highlight some of those that would build on what was achieved in this master's thesis.

A higher level representation of the map

Having a real 3D map can be useful for a variety of applications. The current point-cloud configuration of our map well suited for using it to localize the drone by solving the PnP problem, but less suited for other tasks. The point cloud can be transformed to higher level representations, such as a voxel grid. Voxel stands for volumetric pixel, and is a division of 3D space into small units. Such a representation would allow to be able to tell for each point in space whether there is an object there or not, which can then be used for obstacle avoidance. Alternatively, it could be interesting to use the point cloud to determine where walls are, in order to create a human-readable map of a building.

Collaborative mapping

One of the goals of this project is to enable a group of drones to collaborate. In the previous years, basic communication between drones was implemented, for example allowing them to use a map created by another drone. A more ambitious objective would be for a group of drones to simultaneously build a common map. The implementation of bundle adjustment takes a step in this direction, as we could imagine matching keypoints between keyframes created by two separate drones, resulting in a common map. This would not be trivial, however, as we would need to keep multiple separate maps until there is some overlap between them.

An Extended Kalman Filter

The pose estimation coming from PnP alone is not enough to stabilize the drone in flight, as it comes at a frequency too low for the controller. The current pose fusion scheme incorporates the IMU's pose estimation, but does so with a very basic method. An EKF would significantly improve the accuracy of the fused pose estimation, and allow for a much better controller.

Hardware change

A hardware change is always an attractive idea. One of the reasons for a change would be to use a more modular drone, to be able to freely add and remove sensors. Changing the current monocular camera to a stereo or RGB-D camera is tempting, because it lifts the scale uncertainty, and allows to map points from single views, but it should be kept in mind that a monocular camera also has some advantages:

- They are much more common, making our code useable on a wide variety of devices.
- The scale ambiguity can also be a monocular camera's strong point: it means that they can be used at any scale, using the same camera, and the same techniques. This is not the case for stereo and RGB-D cameras, which have a maximal and minimal working range.
- A considerable part of the current code was written specifically for monocular cameras, and would have to be redone in the event of a change.

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

[5] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "SURF Speeded Up Robust Features". In: *Computer Vision and Image Understanding (CVIU)* 110.3 (2008), pp. 346–359.

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

[14] M. Fischler and R. 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*. 1987, pp. 726–740. ISBN: 0-934613-33-8.

[15] 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.

[16] Henri P. Gavin. *The Levenberg-Marquardt method for nonlinear least squares curve-fitting problems*. 2017. URL: `http://people.duke.edu/~hpgavin/ce281/lm.pdf`.

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

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

[19] 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.

[20] 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.

[21] 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.

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

[23] M.I. A. Lourakis and A.A. Argyros. “SBA: A Software Package for Generic Sparse Bundle Adjustment”. In: *ACM Trans. Math. Software* 36.1 (2009), pp. 1–30. DOI: http://doi.acm.org/10.1145/1486525.1486527.

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

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

[26] 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).

[27] 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).

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

[29] 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.

[30] 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.

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

[32] Noah Snavely. *Bundler: Structure from Motion (SfM) for Unordered Image Collections*. 2008. URL: http://www.cs.cornell.edu/~snavely/bundler/.

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

[34] K. Tateno et al. “CNNSLAM : Real-time dense monocular SLAM with learned depth prediction”. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. Hawaii USA, June 2017.

[35] opencv dev team. *Camera Calibration and 3D Reconstruction*. URL: http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html.

[36] Bill Triggs et al. “Bundle Adjustment – A Modern Synthesis”. In: *VISION ALGORITHMS: THEORY AND PRACTICE, LNCS*. Springer Verlag, 2000, pp. 298–375.

[37] Simon Fraser University. *ardrone-autonomy*. software. URL: http://ardrone-autonomy.readthedocs.io.

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

Appendix A

Full code flowchart

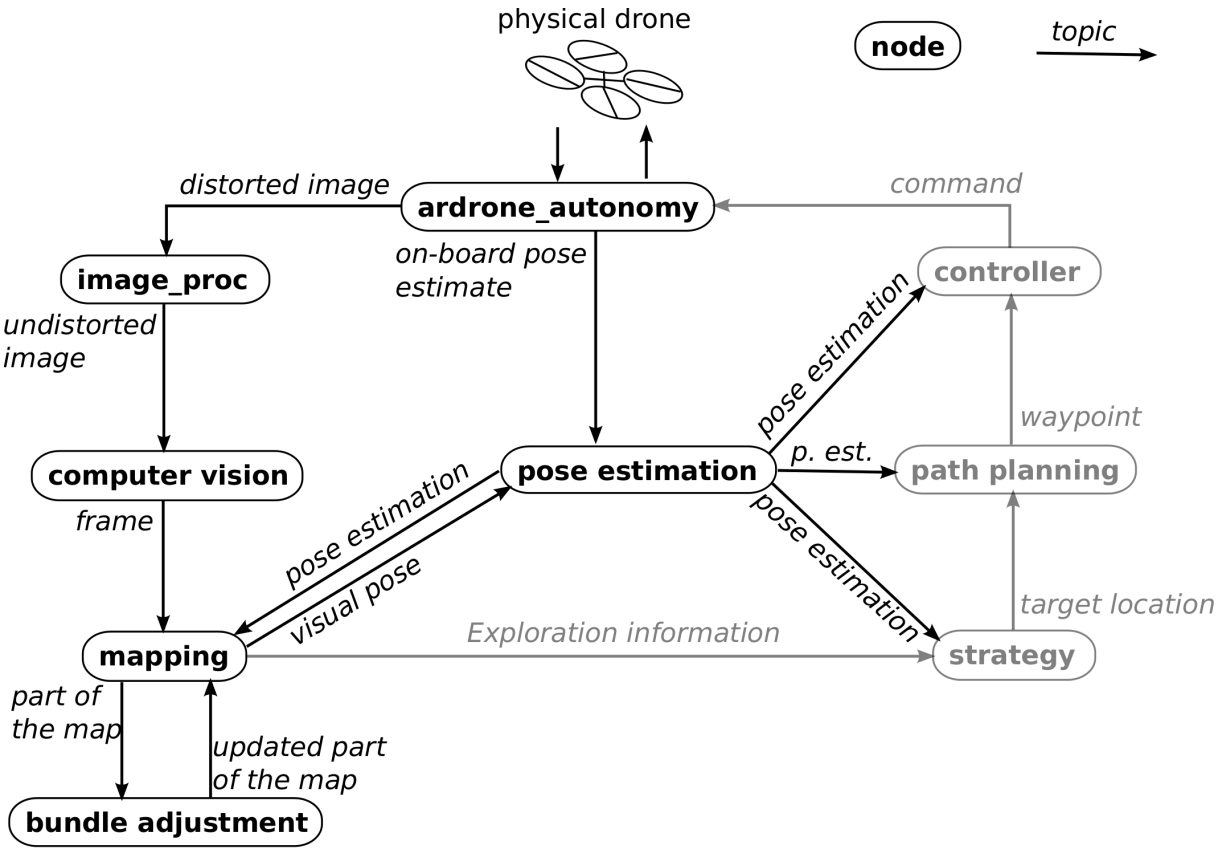


Figure A.1: Flowchart of the different nodes and the communication between them. Grayed-out parts were not worked on in this master's thesis

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