

Title: Visual Inertial Odometry for Mobile Robotics

Author: Faust Terrado González

Director: Juan Andrade-Cetto

Department: Institut de Robòtica Industrial

Date: May 22, 2015

Visual Inertial Odometry for Mobile Robotics



Faust Terrado González

May 2015

Institut de Robòtica Industrial
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

PROJECT INFORMATION

Project title: Visual Inertial Odometry for Mobile Robotics

Author: Faust Terrado

Degree: Informatics Engineering

Credits: 37,5

Director: Juan Andrade-Cetto

Department: Institut de Robòtica Industrial

EXAMINERS

President: Josep Fernández Ruzafa

Vocal: Jose M. Cela Espin

Secretary: Manel Frigola Bourlon

GRADE

Numeric grade:

Qualitative grade:

Date:

Abstract

This work targets the development of a real-time algorithm to track the pose of a mobile robot equipped with an inertial measurement unit and a monocular camera. The method proposed is the Multi-State Constraint Kalman Filter, developed by A. Mourikis and S. Roumeliotis. Its main strength is that it uses the geometric constraints of the features observed without adding them to the state (as in SLAM). The feature processing is delayed until they are out of view, which avoids the usual problems associated to feature initialization. The cost of the algorithm is linear in the number of features (which can be controlled by the detector), which makes it suitable for real-time execution. The method follows the Kalman filtering framework in its error-state variant, which estimates the accumulated error instead of the true-state. The filter is updated with the feature positions, which are computed using Gauss-Newton optimization using the inverse-depth parameterization.

Acknowledgements

I would like to express my gratitude to my supervisor Juan Andrade-Cetto, for his support while doing this thesis. My first experiments with the integration of the IMU measurements are due to him. This document owes much to his corrections and highly regarded feedback.

A very special thank goes to professor Joan Solà. His endless patience to teach me the basic -and not so basic- concepts in SLAM, state estimation and computer vision have been invaluable. Without his experience, intuition and vision this work would have been impossible. And still, the most important skill I could learn from him is the ability to reason critically about the problems.

In addition, I also want to thank my colleagues at the *Institut de Robòtica Industrial* for their help during my time doing this project. Martí Morta helped me to record our first datasets using a mobile robot from the lab. Àngel Santamaria explained me the basics of reference frame transformations and helped me to build our own simulator using ROS and Gazebo. Gonzalo Ferrer provided the first insights on robotics visualization tools and was really helpful regarding L^AT_EX when writing this document. Joan Vallvé provided assistance with linear algebra decompositions. Finally, I also earned much knowledge from the reading group sessions we all had at the lab.

Contents

Contents	1
1 Introduction	3
1.1 Motivation	3
1.2 Solution	4
2 Objectives	7
2.1 Final objective	7
2.2 Preliminary objective	7
3 The Inertial Measurement Unit	9
3.1 Accelerometer	10
3.2 Gyroscope	11
4 The Error-State Kalman Filter	13
4.1 Kinematics in continous time	14
4.2 Kinematics in discrete time	16
4.3 Filter propagation	18
4.4 Filter correction	19
5 Fundamental principles in Computer Vision	21
5.1 Features	21
5.2 Optical Flow	24
5.3 The pinhole camera model	30
5.4 Triangulation methods	33
6 Multi-State Constraint Kalman Filter	39
6.1 Overview	39
6.2 State vector	41
6.3 State propagation	43
6.4 State augment	44
6.5 Observation model	47
6.6 Filter update	53
7 Implementation	57
7.1 External tools	57
7.2 Simulator	64
7.3 Data Structures	66
7.4 Class Structure	68

8	Results	71
8.1	IMU propagation	71
8.2	Feature tracks	72
8.3	Filter update	73
9	Project Management	75
9.1	Planning	75
9.2	Costs	76
10	Discussion	79
	Bibliography	83

1

Introduction

Truly semi-autonomous robots are starting to be possible. Examples of this include Amazon’s intention to deliver packages using autonomous drones, as well as the self-driving truck from Daimler (see Figure 1.1) and the already famous Google’s self-driving car.

Such robots mainly need to solve two problems. First, they need to know their local relative motion. Second, they need to localize themselves within the area around which they are moving.

Odometry is the solution to the first problem. It estimates the pose of the robot over time using sensor measurements. Such sensors may range from simple wheel encoders to 3D lasers, monocular cameras or inertial units, or a combination of them.

The second problem is solved using *Localization and Mapping*. It consists in building a map of the environment to be used as a reference to localize. Somehow, it is some sort of a chicken-and-egg problem: localization is needed to build a map, and a map is needed to localize. A map consists of a set of landmarks, which are static elements in the scene that must be recognizable by the robot’s sensors.

This thesis is centered in the *odometry* problem. This word comes from the Greek for *odos* (route) and *metron* (measure) and it does not apply only to robots. Animals integrate their path to be able to return to their nest, sometimes while traveling at night and hence without using vision. In odometry, the current position is estimated using the previous position, together with information coming from sensors, no matter if they are biological (e.g. vestibular organs, optical flow) or artificial (e.g. inertial units, cameras, lasers).

1.1 Motivation

Our aim is to implement an odometry system that is capable of running in real-time using commodity hardware in a mobile robot. It is assumed that GPS measurements will not be available most of the time and cannot be relied upon, which is the case for indoor areas and most urban environments. Additionally, we require the hardware to be cheap and light-weight, and we want to avoid carrying out long and tedious



Figure 1.1: **Amazon’s Prime Air autonomous delivery drone and Daimler Freightliner self-driving truck**

calibration routines.

The method should give an accurate estimate of the current position and be *consistent*, which means that the propagated uncertainty for the current estimate is correct. Ideally, the algorithm should be able to work in any situation, such as in the presence of moving objects, or under poor lighting conditions.

1.2 Solution

Although many different sensors can be used to estimate a robot’s odometry, the constraints and requirements outlined in the previous section rule out many of the possible options.

One possible option would be using a 3D laser. A laser works by sending pulses of light and computing the time it takes for them to be reflected back. This *time of flight* is proportional to the distance to the surrounding objects. 3D lasers are really useful to create maps, but this comes with drawbacks. They are quite expensive and fragile, and they are too heavy to be carried by small robots (such as quadrotors).

Radars follow a similar principle than lasers, but they use radio waves instead of light. They provide better performance in some cases, such as under the presence of rain. However, they are also expensive and their measurements are less straightforward to use.

On the contrary, single cameras are cheap, small and light-weight sensors that are integrated into many different devices, from smartphones to drones. Computer vision techniques can be used to extract geometric information of the environment, such as points in the scene that project to the image plane. When the camera moves, these points can be triangulated and its 3D position in the scene can be recovered. This gives information about the surrounding world itself as well as about the camera position.

One of the main drawbacks of monocular cameras is that they do not provide scale information. Hence, it is not possible to figure out the size of an object from a single image. For example, looking at a car in a picture, how do we know that it is a real car and not a small toy? This shortcoming makes monocular cameras alone a

somewhat poor option for odometry, where scale is important.

Nevertheless, we do not need to employ only a single camera. We can combine it with other sensors. For instance, we can use a stereo camera, which is nothing else but two cameras separated by a known distance. This distance, called *baseline*, and the association of the same points in both images provide depth information (i.e., scale), even without the need of motion. However, stereo cameras are not perfect. They are bigger and more expensive than single cameras, and on top of that their calibration processes are more problematic.

Another sensor that can be combined with a monocular camera is an inertial measurement unit (IMU), which is another small, cheap and ubiquitous sensor. It combines a gyroscope and an accelerometer and delivers measurements of the angular velocity and linear acceleration applied to its body. Integrating these measurements alone yields position and orientation magnitudes. Unfortunately, the sensor readings have noise, and integrating them produces errors that grow with time, which make the estimated magnitudes drift in the (not so) long term.

Combining the IMU with a camera we get the best of both sensors. On one hand, the geometric information obtained by triangulating the features from the camera can be used to correct the errors accumulated during the integration of the IMU measurements. On the other hand, the IMU provides scale information, thanks to the accelerometer.

This thesis proposes a *visual inertial odometry* solution using a monocular camera and an inertial measurement unit. We do not attempt to construct a map of the scene, neither do we try to localize ourselves using an existing map. Instead, we try to estimate the current robot pose and the uncertainty we have about that pose.

This document is organized as follows. Chapter 3 develops a model for the IMU sensor. Chapter 4 explains the Error-State Kalman Filter, which is the basis of most sensor fusion methods that use IMUs as one of its building blocks. Then, Chapter 5 lays out some essential computer vision techniques that are needed for our visual inertial odometry algorithm. Next, Chapter 6 explains the Multi-State Constraint Kalman Filter, which is a visual inertial odometry method based on a sliding window of old poses and is the algorithm finally implemented in this thesis. Chapter 7 lists the software libraries and data structures used for the implementation of the algorithm. Chapter 8 shows the results of the implementation. Finally, Chapter 9 presents the project's planning and costs, whereas Chapter 10 discusses the final status of the implementation and enumerates some ideas to improve the method.

2

Objectives

2.1 Final objective

The main objective of this project is to implement a visual inertial odometry (VIO) algorithm to track the pose of a moving robot using an inertial measurement unit (IMU) and a monocular camera. The method we rely on is the Multi-State Constraint Kalman Filter (MSCKF) developed by A. Mourikis and S. Roumeliotis [2]. This solution uses a sliding window of past poses to triangulate the feature positions. The features are then used to update the current pose and the sliding window. It uses the Kalman filter framework in its error-state variant, which tries to estimate the accumulated error instead of the true state (as in regular Extended Kalman Filters). The implementation is done in C++ to run in real-time, and it must allow to easily tune the parameters for testing. Additionally, a simulator is also implemented to get ideal sensor data and to help in the filter debug process.

2.2 Preliminary objective

Considerable research was done before undertaking the problem in order to get the foundations in linear algebra, statistics, computer vision, optimization, etc. Some books were carefully read, such as [24], [10] or [8]. A review on state-of-the-art solutions to the problem was also done to find a suitable method. Although the initial idea was to implement a monocular SLAM with no IMU whatsoever, the MSCKF method was finally chosen.

Before starting the implementation, some research was also done to determine the tools needed for the project. Since the initial purpose was to port the algorithm to Android, the algorithm needed to be efficiently implemented to run in real-time, so C++ was selected as the main language. After setting the language, we needed a computer vision library to provide several feature detectors and useful computer vision methods. OpenCV was the best match for this. A linear algebra engine was also required, and after some research the Eigen library was selected. The ROS framework was examined too, since it was the robotics framework used at IRI.

3

The Inertial Measurement Unit

An inertial measurement unit (IMU) is a device that measures acceleration and angular velocity using a combination of accelerometers and gyroscopes.

Long time ago, in the aerospace industry, such devices were constructed using large mechanical components and needed to be mounted on a stable platform to isolate them from the vehicle dynamics. Nowadays, so-called *strapdown* systems are used, which are much smaller and do not need any stabilizing platform to work properly. They are built using electronic components and hence are much cheaper to produce. This makes them ubiquitous and very convenient for their use in the field of mobile robotics, where they are employed to implement inertial navigation systems.

Unfortunately, real world sensors will not deliver the true values of the physic magnitudes they measure. Imperfections in the sensor components, misalignments or overall sensor quality will produce some differences between the measured and true values. Our model will take into account these imperfections in the form of:

$$\mathbf{m}_{measured} = \mathbf{m}_{true} + \mathbf{b} + \mathbf{w} \quad (3.1)$$

where \mathbf{m} accounts for the physical magnitude to be measured (i.e. acceleration or angular rate). The bias \mathbf{b} is a random offset that is added to the true value. It will be different every time the sensor is turned on, and it may change slowly during operation. The noise \mathbf{w} is also random but is sampled every time a new measurement is made. An optimal inertial system will take into account these random components to estimate the true magnitude \mathbf{m}_{true} .

An IMU consists on three accelerometers and three gyroscopes mounted in three perpendicular directions, in order to measure independently each axis of motion. The following sections explain in more detail the models used with accelerometers and gyroscopes. Next chapter explains the Error-State Kalman Filter, which is a common model to implement inertial-aided systems (i.e. systems that use external measures like GPS or images to correct for the IMU errors).

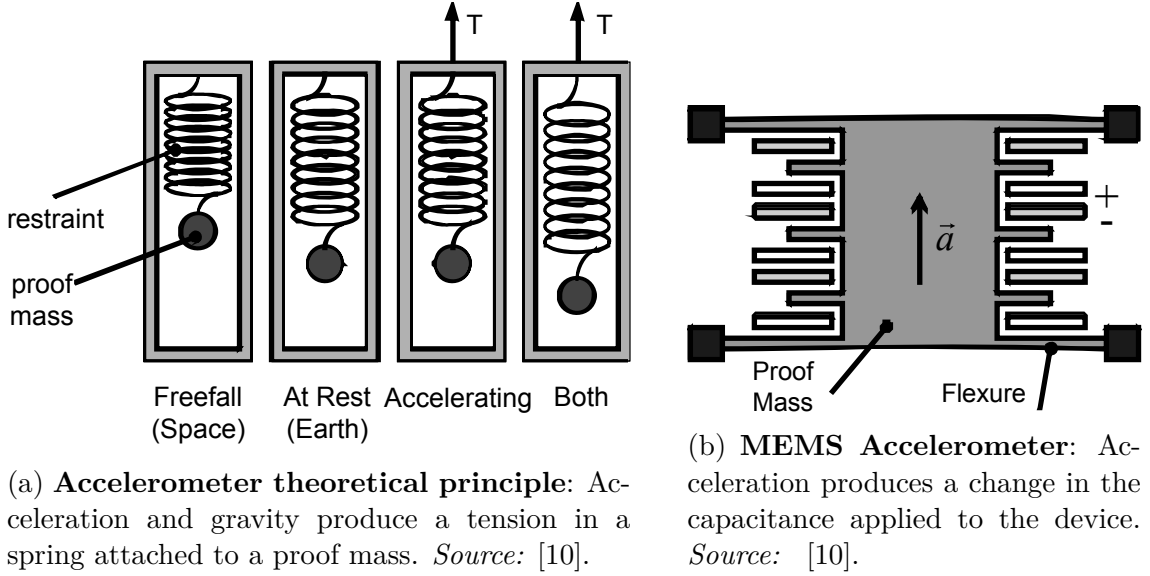


Figure 3.1: Accelerometer principle and MEMS accelerometer.

3.1 Accelerometer

Although in the introduction it was said that accelerometers measure acceleration, that is not exactly so. An accelerometer at rest located on the surface of the earth would measure an acceleration of 1 g due to the effect of gravity.

The theoretical operating principle of an accelerometer consists in measuring the displacement of a proof mass affected by the acceleration and gravitational force (see Figure 3.1a) [10]. This displacement is proportional to the tension in a spring attached to the proof mass. If we put the accelerometer in space and let it fall by means of the gravity force, the sensor will yield an acceleration of 0 g (i.e. no acceleration at all). Differently, if the accelerometer is at rest near the surface of the earth, or it is accelerating upward at 1 g, it will yield an acceleration of 1 g. Finally, if it accelerates upward at 1 g near the earth's surface, it will yield an acceleration of 2g.

In practice, though, real accelerometers are built with electronic components using *MEMS* (Micro ElectroMechanical Systems) technology. In this case, the proof mass is attached to a set of *flexures*. As can be seen in Figure 3.1b, an acceleration applied to the mass produces a displacement, changing the space between the flexures and hence affecting the capacitance between pairs of fingers [10].

Independently of the technology used, the accelerometer will deliver noisy measurements \mathbf{a}_m in *body frame*:

$$\mathbf{a}_m = \mathbf{R}^\top (\mathbf{a} - \mathbf{g}) + \mathbf{b}_a + \mathbf{w}_a \quad (3.2)$$

where \mathbf{R} is the rotation of the accelerometer in a global reference frame, \mathbf{g} is the gravity, \mathbf{b}_a is the accelerometer bias and \mathbf{w}_a its noise. Equation 3.2 subtracts gravity from the true acceleration \mathbf{a} and aligns it with the body frame, since this is the frame

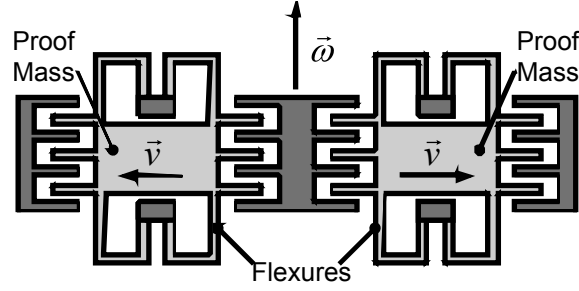


Figure 3.2: **MEMS gyroscope**. Dark shaded areas are fixed to the device case, while lighter areas oscillate left and right. *Source:* [10].

in which measurements are sensed. It is possible (and in fact necessary) to isolate the true acceleration from the measurement:

$$\mathbf{a} = \mathbf{R}(\mathbf{a}_m - \mathbf{b}_a - \mathbf{w}_a) + \mathbf{g} \quad (3.3)$$

This allows to extract the true acceleration from the measurements and will be used later to describe the kinematic system needed to implement inertial navigation.

3.2 Gyroscope

Gyroscopes measure angular velocity in body frame. Similarly to accelerometers, initial designs were built using large mechanical components, or more concretely, a spinning rotor. Angular velocity is proportional to a torque applied perpendicular to the rotation axis of the rotor.

Modern gyroscope designs also use MEMS or optical technology. Optical gyrometers work by sending beams of light through a fiber optic coil. When the device rotates, photons rotating in one direction experience a longer path than those rotating in the opposite direction [10]. Both beams are then mixed in order to measure their phase shift, which is proportional to the angular velocity.

MEMS gyroscopes work in a similar way as MEMS accelerometers, explained in the previous section. An oscillating velocity is applied to a proof mass (Figure 3.2). If the device rotates around the axis pointed by $\boldsymbol{\omega}$, the mobile mass will move in and out of the page, producing a change in the capacitance of the surrounding capacitors [10].

Similarly to the case of the accelerometer, a gyroscope will deliver noisy angular velocity readings $\boldsymbol{\omega}_m$ in body frame:

$$\boldsymbol{\omega}_m = \boldsymbol{\omega} + \mathbf{b}_\omega + \mathbf{w}_\omega \quad (3.4)$$

where $\boldsymbol{\omega}$ is the true angular velocity, \mathbf{b}_ω is the gyroscope bias and \mathbf{w}_ω is the gyroscope noise. We can isolate the true angular velocity:

$$\boldsymbol{\omega} = \boldsymbol{\omega}_m - \mathbf{b}_\omega - \mathbf{w}_\omega \quad (3.5)$$

4

The Error-State Kalman Filter

The previous chapter explained how an IMU works and presented a model to compute the physical magnitudes from the sensor readings. Our objective is to implement an inertial navigation system to estimate the pose of a moving robot. Unfortunately, the sensors measure only accelerations and angular rates, and not absolute positions and orientations. Still, we can integrate acceleration to obtain velocity and displacement, and also angular rates can be integrated to compute changes in orientation.

Although this is certainly possible, in reality is more complex than it may seem at first sight. Sensor readings are noisy, and integrating noise every time (even twice in the case of acceleration) accumulates errors that may grow huge after some time. This leads to *dead-reckoning* systems in which the computed position and orientation drift with time. Of course, using more expensive, high-quality sensors will make the errors smaller, but sometimes it is just not possible to use such sensors, and the errors will eventually get untenable anyway.

Another common solution to this consists in fusing the IMU integrated values with absolute position measurements such as GPS or (as in our case) images. These absolute measurements will render the IMU errors observable, avoiding long-term drift. Accordingly, we only need a model to fuse the IMU readings with the external absolute measurements. The Kalman filter framework is a popular tool for this task.

More concisely, we will focus on the *Error-State Kalman Filter* (ESKF). In this formulation, the state vector is divided between the true, nominal and error components, the true state being a composition of the nominal and the error states. The error state will be small and more linear, making it suitable for Gaussian filtering. Therefore, the covariance matrix will reflect the uncertainty in the error state (and not the nominal, as in the normal Kalman filter), which will be corrected by the filter.

The ESKF starts by integrating IMU measurements into the nominal state, ignoring noise and errors. Consequently, the nominal state will accumulate errors. To solve this problem, these errors are stored in the error state. The filter predicts a Gaussian estimate of this error state, which means that it has a covariance matrix that reflects its uncertainty and that it grows with time. This prediction is done for each IMU reading until a measurement of the external sensor (e.g. GPS, camera) arrives, which usually happens at a lower rate. At that point, this external information renders the

errors observable and allows to correct the error state estimate. Finally, the mean of the error state is injected into the nominal state and it is reset to zero, while the covariance is updated to reflect the new information provided by the external measurements.

The ESKF has some benefits over the regular EKF. The error state is small, which makes second order derivatives negligible. This also prevents from parameter singularities (such as the gimbal lock for Euler angles), improving linearity conditions. And since the errors grow slowly, we can perform filter corrections at a lower rate than predictions.

The following sections explain the ESKF in detail. Section 4.1 presents a continuous time model for the propagation of the true, nominal and error states. Section 4.2 integrates the continuous time equations to obtain a discrete time model, which is necessary if we want to implement the algorithm in a computer (and this is what we do). Finally, the filter prediction and correction steps are explained. Most of the ideas presented here draw from [21].

4.1 Kinematics in continous time

First we need to write the true-state kinematic differential equations to understand how the system evolves with time:

$$\dot{\mathbf{p}}_t = \mathbf{v}_t \quad (4.1a)$$

$$\dot{\mathbf{v}}_t = \mathbf{a}_t = \mathbf{R}_t(\mathbf{a}_m - \mathbf{b}_{at} - \mathbf{w}_a) + \mathbf{g} \quad (4.1b)$$

$$\dot{\mathbf{q}}_t = \frac{1}{2}\mathbf{q}_t \otimes \boldsymbol{\omega}_t = \frac{1}{2}\mathbf{q}_t \otimes (\boldsymbol{\omega}_m - \mathbf{b}_{\omega t} - \mathbf{w}_\omega) \quad (4.1c)$$

$$\dot{\mathbf{b}}_{at} = \mathbf{w}_{ba} \quad (4.1d)$$

$$\dot{\mathbf{b}}_{\omega t} = \mathbf{w}_{b\omega} \quad (4.1e)$$

where Equations 4.1b and 4.1c come from Equations 3.3 and 3.5, \mathbf{w}_{ba} and $\mathbf{w}_{b\omega}$ are the bias noises, \mathbf{R}_t is the rotation matrix associated to \mathbf{q}_t , and Equation 4.1c corresponds to the quaternion time-derivative [21].

As can be seen in Equations 4.1, this state includes the noise and error components. We can now split the true state \mathbf{x}_t into the nominal state \mathbf{x} and the error state $\delta\mathbf{x}$:

$$\mathbf{x}_t = \begin{bmatrix} \mathbf{p}_t \\ \mathbf{v}_t \\ \mathbf{q}_t \\ \mathbf{b}_{at} \\ \mathbf{b}_{\omega t} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} \mathbf{p} \\ \mathbf{v} \\ \mathbf{q} \\ \mathbf{b}_a \\ \mathbf{b}_\omega \end{bmatrix} \quad \delta\mathbf{x} = \begin{bmatrix} \delta\mathbf{p} \\ \delta\mathbf{v} \\ \delta\boldsymbol{\theta} \\ \delta\mathbf{b}_a \\ \delta\mathbf{b}_\omega \end{bmatrix}$$

the true state being a suitable composition between the nominal and the error state:

$$\mathbf{x}_t = \mathbf{x} \oplus \delta\mathbf{x}$$

Here the symbol \oplus means a suitable, component-wise composition. For all components but the orientation, this will be a regular addition (e.g. $\mathbf{p}_t = \mathbf{p} + \delta\mathbf{p}$, $\mathbf{v}_t = \mathbf{v} + \delta\mathbf{v}$). The orientation composition is a bit more complex. While the true and nominal orientations are represented by a quaternion, the error orientation is *minimally* represented by the error angles $\delta\boldsymbol{\theta}$ [21]. Since this will be small, the error quaternion $\delta\mathbf{q}$ can be approximated with:

$$\delta\mathbf{q} \simeq \begin{bmatrix} 1 \\ \frac{1}{2}\delta\boldsymbol{\theta} \end{bmatrix} \quad (4.2)$$

The orientation composition is then defined by:

$$\mathbf{q}_t = \delta\mathbf{q} \otimes \mathbf{q} \quad (4.3)$$

It is important to note that $\delta\mathbf{q}$ appears in the composition's left-hand side because it is defined in the *global* frame. Although it would be possible to define the orientation error in the local frame (and put it in the right-hand side, i.e. $\mathbf{q}_t = \mathbf{q} \otimes \delta\mathbf{q}$), we prefer to avoid it on the grounds that it adds undesirable terms in the filter's observability matrix [15]. The equivalent of Equation 4.3 for rotation matrices is:

$$\mathbf{R}_t = (\mathbf{I} + [\delta\boldsymbol{\theta}]_{\times})\mathbf{R} \quad (4.4)$$

where $[\mathbf{v}]_{\times}$ is the skew-symmetric matrix representing the cross product:

$$[\mathbf{v}]_{\times} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix} \quad [\mathbf{v}]_{\times}\mathbf{w} = \mathbf{v} \times \mathbf{w}$$

After having defined the state composition, we only need to describe the kinematic equations for the nominal and error states. The nominal state is simply obtained by removing the noises and perturbations from Equations 4.1:

$$\dot{\mathbf{p}} = \mathbf{v} \quad (4.5a)$$

$$\dot{\mathbf{v}} = \mathbf{a} = \mathbf{R}(\mathbf{a}_m - \mathbf{b}_a) + \mathbf{g} \quad (4.5b)$$

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \otimes \boldsymbol{\omega} = \frac{1}{2}\mathbf{q} \otimes (\boldsymbol{\omega}_m - \mathbf{b}_{\omega}) \quad (4.5c)$$

$$\dot{\mathbf{b}}_a = 0 \quad (4.5d)$$

$$\dot{\mathbf{b}}_{\omega} = 0 \quad (4.5e)$$

And finally, the error state is derived by isolating the error components from the true state, using the error rotation matrix from Equation 4.4 for the orientation error:

$$\delta \dot{\mathbf{p}} = \delta \mathbf{v} \quad (4.6a)$$

$$\delta \dot{\mathbf{v}} = -[\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a)]_{\times} \delta \boldsymbol{\theta} - \mathbf{R} \delta \mathbf{b}_a - \mathbf{R} \mathbf{w}_a \quad (4.6b)$$

$$\delta \dot{\mathbf{q}} = -\mathbf{R} \delta \mathbf{b}_\omega - \mathbf{R} \mathbf{w}_\omega \quad (4.6c)$$

$$\delta \dot{\mathbf{b}}_a = \mathbf{w}_{ba} \quad (4.6d)$$

$$\delta \dot{\mathbf{b}}_\omega = \mathbf{w}_{b\omega} \quad (4.6e)$$

The developments from Equations 4.6b and 4.6c are not trivial and can be seen in [21]. The skew components come from Equation 4.4 and by noting that $[\mathbf{v}]_{\times} \mathbf{w} = -[\mathbf{w}]_{\times} \mathbf{v}$.

4.2 Kinematics in discrete time

To implement the previous differential equations of the form $\dot{\mathbf{x}} = f(t, \mathbf{x})$ in a computer we need a way to discretize them by integrating over a discrete time interval Δt :

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \int_{k\Delta t}^{(k+1)\Delta t} f(\tau, \mathbf{x}(\tau)) d\tau \quad (4.7)$$

There are several ways to compute the integral in the right part of Equation 4.7: one can choose a suitable integration method among several options with varying levels of accuracy. In some cases it is also possible to find exact closed-form formulas to integrate a certain magnitude. An explanation of many of these options are found in [21], but we use one of the simplest: the one-step Euler integration. We opt for this option due to its simplicity, and because the IMU frame rate is sufficiently fast to neglect higher order terms of the integration model, though it would be easy in the future to implement different methods and choose the best one.

The Euler method assumes that $f(t, \mathbf{x}(t))$ follows a line within the interval Δt :

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t f(t_k, \mathbf{x}_k) \quad (4.8)$$

This will not be exactly true most of the times but, as said, the accumulated approximation errors will get smaller as the interval Δt gets smaller (and it would be exact in the limit to zero). Figure 4.1 shows the errors of using large and small step sizes.

In our inertial system case, we receive IMU measurements at relatively high rates (100 Hz, meaning that $\Delta t = 0.01$ sec). On top of that, we are using camera images at 30 Hz to correct for the integration errors. Altogether makes the Euler method a reasonable starting point for our algorithm.

We need to derive the discrete form of the nominal and error states explained in the previous section. Using Euler integration, we obtain the discrete form for the nominal state from Equation 4.5:

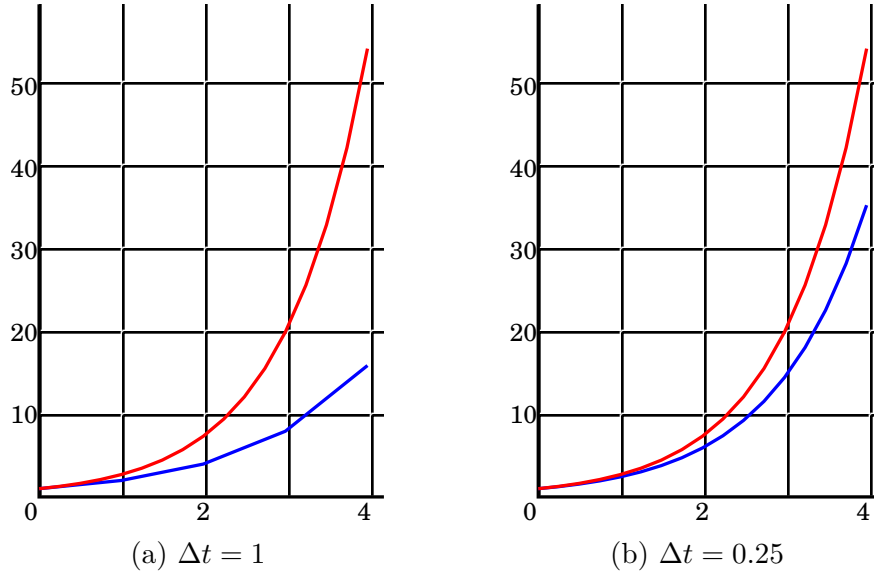


Figure 4.1: **Euler integration step sizes.** Difference between using large (a) or small (b) step sizes in the Euler integration method. The red line is the true curve, while the blue line corresponds to its Euler approximation with different time step sizes. Source: *Wikipedia*.

$$\mathbf{p}^+ \leftarrow \mathbf{p} + \mathbf{v}\Delta t + \frac{1}{2}(\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a) + \mathbf{g})\Delta t^2 \quad (4.9a)$$

$$\mathbf{v}^+ \leftarrow \mathbf{v} + (\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a) + \mathbf{g})\Delta t \quad (4.9b)$$

$$\mathbf{q}^+ \leftarrow \mathbf{q} \otimes \text{vec2q}((\boldsymbol{\omega}_m - \mathbf{b}_\omega)\Delta t) \quad (4.9c)$$

$$\mathbf{b}_a^+ \leftarrow \mathbf{b}_a \quad (4.9d)$$

$$\mathbf{b}_\omega^+ \leftarrow \mathbf{b}_\omega \quad (4.9e)$$

where $\text{vec2q}(\mathbf{v})$ converts a rotation vector to a quaternion:

$$\alpha = \|\mathbf{v}\| \quad (4.10a)$$

$$\mathbf{u} = \frac{\mathbf{v}}{\alpha} \quad (4.10b)$$

$$\mathbf{q} = \begin{bmatrix} \mathbf{u} \sin(\frac{\alpha}{2}) \\ \cos(\frac{\alpha}{2}) \end{bmatrix} \quad (4.10c)$$

The rotation vector \mathbf{v} expresses a rotation of α radians around the axis defined by the unit vector \mathbf{u} . The same rotation is expressed by the final quaternion \mathbf{q} .

Finally, we need to integrate the error state differential equations. This is a bit more complex than integrating the nominal state due to the random impulses \mathbf{i} applied to the measurements and biases. Assuming the random impulses to be zero mean white Gaussian noise, they integrate to zero, but we should integrate their covariances to

account for the uncertainty caused by them. Integrating Equations 4.6 and adding the random impulses results in:

$$\delta \mathbf{p}^+ \leftarrow \delta \mathbf{p} + \delta \mathbf{v} \Delta t \quad (4.11a)$$

$$\delta \mathbf{v}^+ \leftarrow \delta \mathbf{v} + (-[\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a)]_{\times} \delta \boldsymbol{\theta} - \mathbf{R} \delta \mathbf{b}_a) \Delta t + \mathbf{i}_v \quad (4.11b)$$

$$\delta \boldsymbol{\theta}^+ \leftarrow \delta \boldsymbol{\theta} + \mathbf{R} \delta \mathbf{b}_\omega \Delta t + \mathbf{i}_\theta \quad (4.11c)$$

$$\delta \mathbf{b}_a^+ \leftarrow \delta \mathbf{b}_a + \mathbf{i}_{ba} \quad (4.11d)$$

$$\delta \mathbf{b}_\omega^+ \leftarrow \delta \mathbf{b}_\omega + \mathbf{i}_{b\omega} \quad (4.11e)$$

where \mathbf{i} are the random impulses. Their covariances are computed by integrating the covariances of \mathbf{w}_a , \mathbf{w}_ω , \mathbf{w}_{ba} , $\mathbf{w}_{b\omega}$ (see Equations 4.6). The integration of these covariances is explained in detail in [21] and results in:

$$\mathbf{W}_v = \sigma_a^2 \Delta t^2 \mathbf{I} \quad (4.12a)$$

$$\mathbf{W}_\theta = \sigma_\omega^2 \Delta t^2 \mathbf{I} \quad (4.12b)$$

$$\mathbf{W}_{ba} = \sigma_{ba}^2 \Delta t \mathbf{I} \quad (4.12c)$$

$$\mathbf{W}_{b\omega} = \sigma_{b\omega}^2 \Delta t \mathbf{I} \quad (4.12d)$$

where $\sigma_a[m/s^2]$, $\sigma_\omega[rad/s]$, $\sigma_{ba}[m/s^2\sqrt{s}]$ and $\sigma_{b\omega}[rad/s\sqrt{s}]$ are the standard deviations of \mathbf{w}_a , \mathbf{w}_ω , \mathbf{w}_{ba} , $\mathbf{w}_{b\omega}$, and they can be extracted from the IMU data sheet or by analysing measurements.

An important difference exists between the integration of the measurement noises \mathbf{w}_v , \mathbf{w}_ω and the bias noises \mathbf{w}_{ba} , $\mathbf{w}_{b\omega}$. The former are sampled on every measurement, and they are considered constant over the integration interval Δt . The latter are never sampled and hence their integration is stochastic. The implications of this can be seen in the covariance matrices of each term: Equations 4.12a and 4.12b are quadratic with respect to Δt , while 4.12c and 4.12d are linear. Refer to [21] for more details on the noise integration.

4.3 Filter propagation

The filter propagates the nominal and error states every time a new measurement arrives. It also increases the uncertainty associated to the error state, represented by the covariance matrix \mathbf{P} .

Propagation (also called prediction) applies Equations 4.9 and 4.11 to the nominal and error state, usually in matrix form. The propagation equations can then be written as:

$$\hat{\mathbf{x}}^+ \leftarrow \mathbf{F}_x \hat{\mathbf{x}} \quad (4.13a)$$

$$\mathbf{P}^+ \leftarrow \mathbf{F}_x \mathbf{P} \mathbf{F}_x^\top + \mathbf{F}_i \mathbf{Q}_i \mathbf{F}_i^\top \quad (4.13b)$$

\mathbf{F}_x is the system transition matrix (which is the Jacobian of the system transition function with respect to the error state) and corresponds to the matrix form of Equations 4.11. Its terms depend on the form of integration used: in our case this is the Euler form. \mathbf{Q}_i is the covariance matrix of the random impulses and it contains the terms explained in Equations 4.12. \mathbf{F}_i is the Jacobian of the system transition function with respect to the random impulses, it just maps the terms in \mathbf{Q}_i to their correct places in \mathbf{P} . The content of these matrices using Euler integration and a global frame of reference for the error orientation component are:

$$\mathbf{F}_x = \begin{bmatrix} \mathbf{I} & \mathbf{I}\Delta t & 0 & 0 & 0 \\ 0 & \mathbf{I} & -[\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a)]_\times \Delta t & -\mathbf{R}\Delta t & 0 \\ 0 & 0 & \mathbf{I} & 0 & -\mathbf{R}\Delta t \\ 0 & 0 & 0 & \mathbf{I} & 0 \\ 0 & 0 & 0 & 0 & \mathbf{I} \end{bmatrix} \quad (4.14a)$$

$$\mathbf{F}_i = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \mathbf{I} & 0 & 0 & 0 \\ 0 & \mathbf{I} & 0 & 0 \\ 0 & 0 & \mathbf{I} & 0 \\ 0 & 0 & 0 & \mathbf{I} \end{bmatrix} \quad (4.14b)$$

$$\mathbf{Q}_i = \begin{bmatrix} \mathbf{W}_v & 0 & 0 & 0 \\ 0 & \mathbf{W}_\theta & 0 & 0 \\ 0 & 0 & \mathbf{W}_{ba} & 0 \\ 0 & 0 & 0 & \mathbf{W}_{b\omega} \end{bmatrix} \quad (4.14c)$$

The right component of Equation 4.13b adds the noise covariance to the covariance matrix, growing it every time, which is the normal behaviour for the state propagation step. Conversely, since the error state $\delta\mathbf{x}$ is always initialized to zero, Equation 4.13a will always return zero. The error state will be only observable during the correction step.

4.4 Filter correction

The previous section explained how the IMU measurements are used to make predictions about the error state. The way to correct these predictions is to account for the information provided by an additional sensor (a monocular camera in our case), which will render the errors observable. This additional sensor will use an observation model (represented by h) to predict the observation from the true state \mathbf{x}_t .

$$\hat{\mathbf{z}} = h(\mathbf{x}_t) + \mathbf{n} \quad (4.15)$$

where \mathbf{n} is the measurement Gaussian noise with covariance \mathbf{R} . The observation function $h(\mathbf{x}_t)$ depends on the concrete sensor used and is not defined here. The

next chapter explains the observation function for the MSCKF algorithm.

The difference between the observed and expected measurement is the residual:

$$\mathbf{r} = \mathbf{z} - \hat{\mathbf{z}} \quad (4.16)$$

With this we can apply the Kalman filter update equations, which will be used to observe the error state:

$$\mathbf{Z} = \mathbf{H}\mathbf{P}\mathbf{H}^\top + \mathbf{R} \quad (4.17a)$$

$$\mathbf{K} = \mathbf{P}\mathbf{H}^\top\mathbf{Z}^{-1} \quad (4.17b)$$

$$\hat{\delta\mathbf{x}} \leftarrow \mathbf{K}\mathbf{r} \quad (4.17c)$$

$$\mathbf{P} \leftarrow (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}(\mathbf{I} - \mathbf{K}\mathbf{H})^\top + \mathbf{K}\mathbf{R}\mathbf{K}^\top \quad (4.17d)$$

where \mathbf{H} is the Jacobian of the observation function h with respect to the error state and evaluated at the true state estimate, which at this point coincides with the nominal state since the error state is still zero. \mathbf{Z} is the innovation covariance, \mathbf{K} is the Kalman gain, and \mathbf{P} is updated using the more numerically stable Joseph form, which guarantees that \mathbf{P} remains positive and symmetric.

Once the error state is observed, it must be injected into the nominal state to correct the accumulated errors: $\mathbf{x} \leftarrow \mathbf{x} \oplus \delta\mathbf{x}$. This is done through suitable component-wise compositions, as explained in section 4.1. To sum up, the injection includes the following operations:

$$\mathbf{p}^+ \leftarrow \mathbf{p} + \hat{\delta\mathbf{p}} \quad (4.18a)$$

$$\mathbf{v}^+ \leftarrow \mathbf{v} + \hat{\delta\mathbf{v}} \quad (4.18b)$$

$$\mathbf{q}^+ \leftarrow \text{vec2q}(\hat{\delta\boldsymbol{\theta}}) \otimes \mathbf{q} \quad (4.18c)$$

$$\mathbf{b}_a^+ \leftarrow \mathbf{b}_a + \hat{\delta\mathbf{b}}_a \quad (4.18d)$$

$$\mathbf{b}_\omega^+ \leftarrow \mathbf{b}_\omega + \hat{\delta\mathbf{b}}_\omega \quad (4.18e)$$

Finally, since the errors have been observed and injected into the nominal state, the error state must be reset to zero:

$$\delta\mathbf{x} \leftarrow \mathbf{0} \quad (4.19)$$

5

Fundamental principles in Computer Vision

Digital cameras are ubiquitous nowadays. They are used in a broad range of applications, from industrial quality control to autonomous vehicle navigation. Their low cost (at least in comparison to other sensors) makes them a suitable option for odometry algorithms. The tools needed to effectively use the camera sensor for this purpose belong to the field of *Computer Vision*. Computer Vision allows the extraction of high-dimensional data from images. In our particular context of visual-inertial odometry, our ultimate objective consists in extracting the *geometrical* information of the surrounding world. This will be used to estimate the position of the camera and correct the errors generated during the IMU integration stage. This chapter explains the tools used to implement our visual-inertial odometry algorithm. First, the concept of local feature is introduced as the basic unit of information that we can obtain from images. Then, the optical flow section explains how to estimate the *motion* of such features. Lastly, the pinhole camera model is explained, which is used in the last section to triangulate the 3D position of the moving features.

5.1 Features

Local features are patterns within an image which differ from their immediate neighbourhood. They consist usually in changes in one or more image properties, such as intensity, color and texture. They represent local regions of interest of a scene or an image.

Features are used extensively in the field of Computer Vision. Object recognition, 3D scene estimation and motion tracking all rely on the use of representative features in the image. In such cases, a set of images, possibly continuous in time (i.e: a video stream), is analysed in order to detect objects in the environment and track their position with respect to the camera. In the field of *Augmented Reality*, the type of features used consists typically of predefined markers that are searched for in the environment using a hand-held camera and its pose is constantly tracked in order to superimpose virtual images over the real ones (hence the name of Augmented Reality).

Another important use for local features is in *Simultaneous Localisation and Mapping* (SLAM) and *Visual Odometry*, where they are used as landmarks. In this case, local

features consist of natural elements of the environment and are tracked without using a predefined marker. Each feature is tracked from one image in which it appears to the next and its position within the scene is estimated. This allows for instance, a mobile robot to explore its environment knowing its own location and, in the case of SLAM, creating and keeping a map of the environment at the same time.

In such applications, every new image is analysed first to find a set of relevant features. In a second step, the newly detected features are matched against a database of past features in order to find *correspondences* between them. When a feature is matched against a past feature with enough certainty, it is assumed that both of them represent the same point in space. An additional step is usually performed to reject outliers (e.g: wrong correspondences between features).

Two components related to features are needed to perform the steps described above:

Feature Detectors: This component scans an image and gives a set of locations where a local feature is present. Different detection strategies lead to different properties, the most important of which is *repeatability*. This implies that given two images of the same scene taken from different viewpoints or lighting conditions, most features should be detected in both images. Other important detector properties are the *quantity* of features detected in an image and its *efficiency*.

Feature Descriptors: After some features have been detected, a method is needed to identify them and match them with other features in order to find correspondences. This is accomplished through Descriptors, which capture the most important information in the detected interest regions. The ideal descriptor allows to recognize the same region in different images without mistaking them. The most basic descriptor consists in storing a small patch around the interest region and cross-correlate it with other patches in order to find correspondences between them.

Although there are many kinds of features, such as corners, edges, or blobs, this document considers only corners. There is a good reason for that: corners can represent points in a scene and are easily parametrized. Many Structure from Motion algorithms use corner features to estimate either the camera's pose or the location of those features in the scene.

The following section will explain the Harris detector. Although quite old, it will lay down the basics for the Lucas-Kanade optical flow method which is the one used in this project.

Harris Detector

The Harris Detector in its original form [7], although quite simple, was one of the first successful methods for feature extraction. Its basic idea consists in shifting a small window patch around an image pixel and check which directions yield a larger change in appearance:

- *Flat regions* won't change in any direction.
- *Regions with edges* won't change in the edge's direction.
- *Regions with corners* will produce a large change in all directions.

The way to measure this change is the weighted *sum of squared differences* (SSD). The term *weighted* implies that the resulting differences in intensity will be smoothed with a window function w , typically a Gaussian:

$$S(x, y) = \sum_{x, y} w(x, y) [I(x + u, y + v) - I(x, y)]^2 \quad (5.1)$$

This can be approximated by taking the first partial derivatives of the Taylor expansion:

$$S(x, y) \approx \sum_{x, y} w(x, y) [uI_x + vI_y]^2 \quad (5.2)$$

Such approximation is typically written in Matrix form:

$$S(x, y) \approx \begin{pmatrix} u & v \end{pmatrix} \mathbf{M} \begin{pmatrix} u \\ v \end{pmatrix} \quad (5.3)$$

where \mathbf{M} is a matrix obtained from the image derivatives:

$$\mathbf{M} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \quad (5.4)$$

The distribution for the x and y derivatives is different for the three types of regions (flat, edges and corners). This difference can be seen by the two eigenvalues of the M matrix, which represent the change in the directions x , y :

- λ_1, λ_2 **small**: Flat region
- λ_1, λ_2 **large**: Corner
- $\lambda_1 \gg \lambda_2, \lambda_1 \ll \lambda_2$: Edge

If one is only interested in detecting corners, it's possible to define a measure for *cornerness*:

$$r = \det(\mathbf{M}) - k \cdot \text{trace}(\mathbf{M})$$

Where k is a parameter to control the sensibility of the cornerness measure, typically in the range 0.4-0.6. Since the determinant of a matrix corresponds to the product of its eigenvalues, and the trace is their sum, high values of r correspond to large eigenvalues, which is the condition for corners.

The Harris corner detector algorithm can be described with the following steps:

1. Compute x and y derivatives of the Image (usually using the Sobel operator).

$$I_x = G_\sigma^x * I \quad I_y = G_\sigma^y * I$$

2. For every pixel, compute products of derivatives.

$$I_x^2 = I_x I_x \quad I_y^2 = I_y I_y \quad I_{xy} = I_x I_y$$

3. Smooth the product images with a Gaussian kernel.

$$S_x^2 = G_{\sigma^s} * I_x^2 \quad S_y^2 = G_{\sigma^s} * I_y^2 \quad S_{xy} = G_{\sigma^s} * I_{xy}$$

4. For every pixel (x, y) , define matrix \mathbf{M} .

$$\mathbf{M}(x, y) = \begin{bmatrix} S_x^2(x, y) & S_{xy}(x, y) \\ S_{xy}(x, y) & S_y^2(x, y) \end{bmatrix}$$

5. Compute the cornerness measure r .

$$r = \det(\mathbf{M}) - k \cdot \text{trace}(\mathbf{M})$$

6. Apply threshold on r to determine whether the pixel (x, y) is a corner.

5.2 Optical Flow

Introduction

Motion provides valuable information about the world that surround us. Humans use this information for a variety of visual tasks, such as object recognition, scene understanding, locomotion control or distance detection.

Optical flow computes an approximation of the motion of objects, edges, or corners from a sequence of images that vary in time (see Figure 5.1). Specifically, we have a camera moving in a scene and recording objects which might be moving as well. Each 3D point $\mathbf{P}_k = (X_k, Y_k, Z_k)^\top$ in the scene is projected to the image on every frame k , producing the 2D point $\mathbf{p}_k = (u_k, v_k)^\top$. Since the 3D point and the camera are moving relative to each other, one can follow the temporal path for that point: $\mathbf{p}(t) = (\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_n)$. The motion between two instances of the same point in the path is represented by the velocity $\mathbf{v}(t) = d\mathbf{p}(t)/dt$. The set of velocities for all 2D points is usually called the *motion field*, and it is what optical flow computes. The method is based on the computation of the image gradient as follows.

Optical flow methods can be either *sparse* or *dense*. *Sparse* methods compute the motion field for a relatively small feature set. In contrast, *dense* methods compute

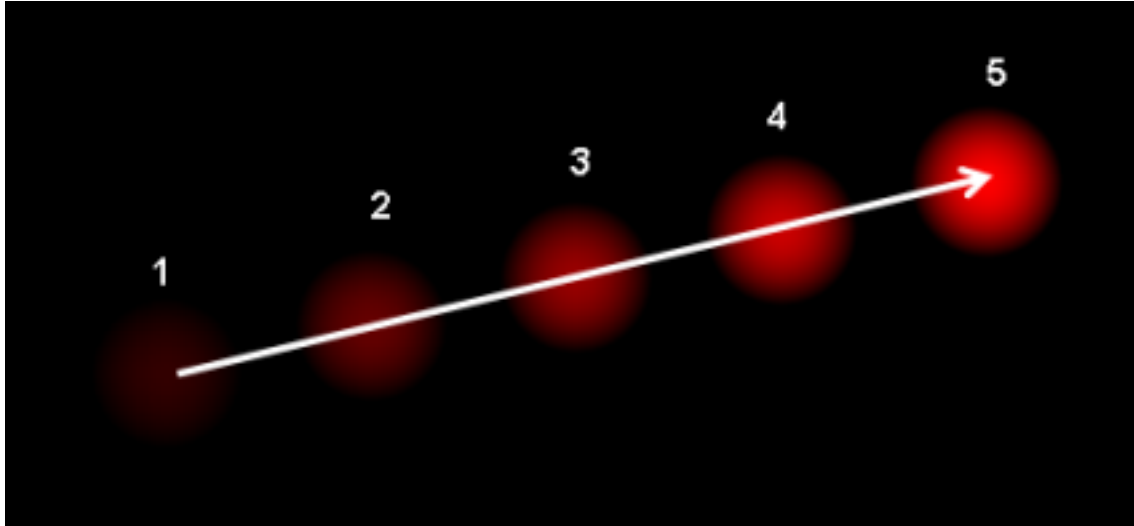


Figure 5.1: **Ball moving in 5 consecutive frames.** Optical flow computes the 2D velocity of the ball when it is projected to the image plane on each frame. Source: *Wikipedia*.

the optical flow for all visible points (i.e. pixels) in the image. Figure 5.2 shows the difference between both methods. Section 5.2 explains the *Lucas-Kanade* algorithm, which is a sparse method for computing optical flow and it is the one used in the visual inertial odometry algorithm presented in this document. The method is based on the computation of the image gradient as follows.



(a) Sparse optical flow



(b) Dense optical flow

Figure 5.2: **Difference between sparse and dense methods.** Source: *OpenCV*.

Gradient-based estimation

A common hypothesis used in optical flow is the so-called *brightness constancy constraint*:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \quad (5.5)$$

This restriction, illustrated in Equation 5.5, implies that the intensity I of the image at a certain point won't vary from one frame to the next. While this assumption

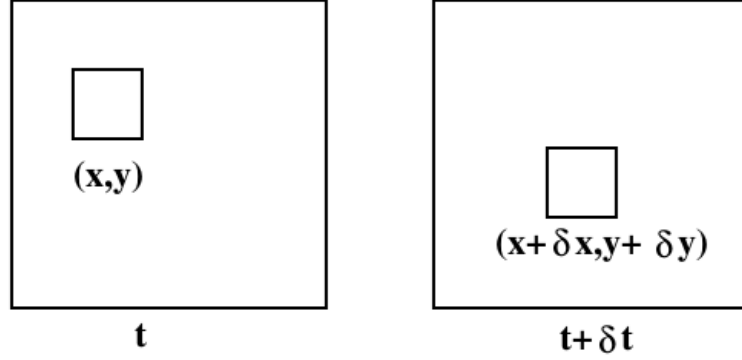


Figure 5.3: **Brightness constancy constraint:** the image intensity at (x, y, t) (left) is the same as at $(x + \delta x, y + \delta y, t + \delta t)$ (right)

won't hold exactly for most surfaces and lighting conditions. In practice, however, it is a good approximation, provided that Δx , Δy and Δt are relatively small local translations.

The objective consists in computing the velocity at each point, which can be approximated using the Taylor series and ignoring higher order terms. This approximation is reasonable since it is assumed that the movement between images is small.

$$I(x + \Delta x, y + \Delta y, t + \Delta t) = I(x, y, t) + I_x \Delta x + I_y \Delta y + I_t \Delta t \quad (5.6)$$

Where I_x , I_y , and I_t are the derivatives of the image at (x, y, t) in each direction. From Eq. 5.5 it follows that:

$$I_x \Delta x + I_y \Delta y + I_t \Delta t = 0$$

$$I_x \frac{\Delta x}{\Delta t} + I_y \frac{\Delta y}{\Delta t} + I_t \frac{\Delta t}{\Delta t} = 0$$

$$I_x v_x + I_y v_y + I_t = 0$$

$$I_x v_x + I_y v_y = -I_t \quad (5.7)$$

v_x and v_y are the image velocity components in the x and y directions (i.e. the *optical flow*). Eq. 5.7 is one equation with two unknowns and hence it cannot be solved. This is the effect of the *aperture problem*. The local region doesn't provide enough information to recover the velocity direction. As a result, the motion direction of the local structure component (e.g. an edge) is ambiguous. Figure 5.4 illustrates this: a pattern of leaning edges moves in a certain direction behind a small circular aperture. It is impossible to recover the full velocity direction in this case: only the velocity normal to the edge can be computed.

Equation 5.7 can be pictured as a line in velocity space, with its parameters determined by I_x , I_y and I_t . The correct velocity is somewhere on that line, but we need additional constraints (i.e. crossing lines) to find it. The velocity normal to the edge is the velocity along the line with the smallest magnitude: it is perpendicular to the line and goes through the origin.

Different optical flow methods apply additional restrictions to compute the velocity. Section 5.2 explains the Lucas-Kanade method, which assumes constant velocity in the local neighbourhood of a pixel and applies least squares to find it.

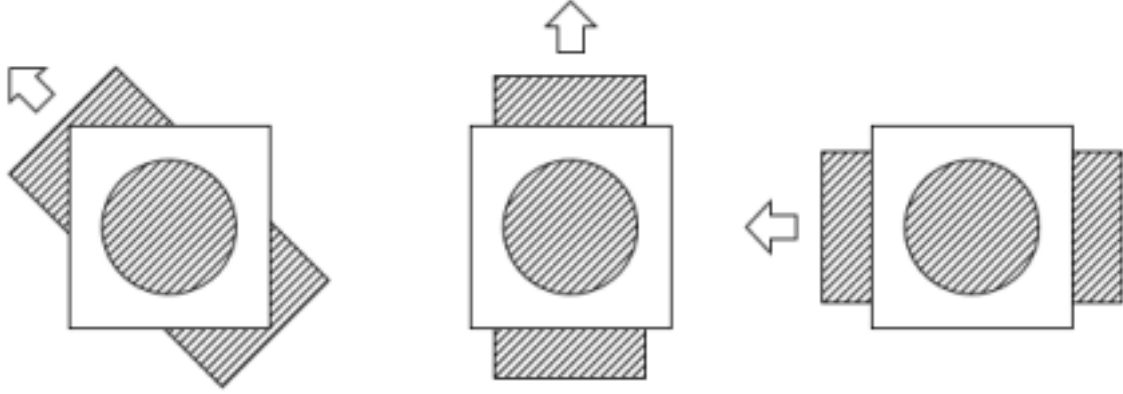


Figure 5.4: **An example of the aperture problem**

Lucas-Kanade method

As explained in the previous section, the image velocity in a local region cannot be recovered directly, due to the fact that we have one equation with two unknowns. The Lucas-Kanade method [16] adds additional equations by applying Equation 5.7 to a window centered at the pixel p under consideration. It assumes that the image velocity \mathbf{v} is small and constant within this local window.

Specifically, the Lucas-Kanade method implements a weighted least squares that minimizes the difference in the coefficients:

$$\sum_{x,y \in \Omega} w(x,y) [\nabla I(x,y,t) \cdot \mathbf{v} + I_t(x,y,t)]^2 \quad (5.8)$$

where Ω means the local neighbourhood centred at the interest point \mathbf{p} , and w is a window function to give more weight to the points nearer to \mathbf{p} (w is typically a Gaussian window).

For every pixel within the local neighbourhood, $\mathbf{p}_i = (x_i, y_i) \in \Omega$, the optical flow equations can be written in matrix form $\mathbf{A}\mathbf{v} = \mathbf{b}$:

$$\mathbf{A} = \begin{bmatrix} \nabla I(x_1, y_1) \\ \nabla I(x_2, y_2) \\ \dots \\ \nabla I(x_n, y_n) \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -I_t(x_1, y_1) \\ -I_t(x_2, y_2) \\ \dots \\ -I_t(x_n, y_n) \end{bmatrix} \quad (5.9)$$

The least-squares solution 5.8 is used to solve this over-determined system of equations:

$$\begin{aligned} SSE(\mathbf{v}) &= \min\{(\mathbf{b} - \mathbf{A}\mathbf{v})^\top \mathbf{W}(\mathbf{b} - \mathbf{A}\mathbf{v})\} \\ SSE(\mathbf{v}) &= \mathbf{b}^\top \mathbf{W}\mathbf{b} - \mathbf{v}^\top \mathbf{A}^\top \mathbf{W}\mathbf{b} - \mathbf{b}^\top \mathbf{W}\mathbf{A}\mathbf{v} + \mathbf{v}^\top \mathbf{A}^\top \mathbf{W}\mathbf{A}\mathbf{v} \end{aligned}$$

where \mathbf{W} is the matrix that contains the weights w . Noting that $(\mathbf{v}^\top \mathbf{A}^\top \mathbf{W}\mathbf{b})^\top = \mathbf{b}^\top \mathbf{W}\mathbf{A}\mathbf{v}$, this leads to:

$$SSE(\mathbf{v}) = \mathbf{b}^\top \mathbf{W}\mathbf{b} - 2\mathbf{b}^\top \mathbf{W}\mathbf{A}\mathbf{v} + \mathbf{v}^\top \mathbf{A}^\top \mathbf{W}\mathbf{A}\mathbf{v}$$

Now we take the partial derivative with respect to \mathbf{v} and set it to zero:

$$\frac{\partial SSE(\mathbf{v})}{\partial \mathbf{v}} = -2\mathbf{A}^\top \mathbf{W}\mathbf{b} + 2\mathbf{A}^\top \mathbf{W}\mathbf{A}\mathbf{v} = 0$$

Solving for \mathbf{v} results in:

$$\mathbf{v} = (\mathbf{A}^\top \mathbf{W}\mathbf{A})^{-1} \mathbf{A}^\top \mathbf{W}\mathbf{b} \quad (5.12)$$

which can be written in closed-form:

$$\begin{bmatrix} v_x \\ v_y \end{bmatrix} = \begin{bmatrix} \sum_i w_i I_x^2(x_i, y_i) & \sum_i w_i I_x(x_i, y_i) I_y(x_i, y_i) \\ \sum_i w_i I_x(x_i, y_i) I_y(x_i, y_i) & \sum_i w_i I_y^2(x_i, y_i) \end{bmatrix} \begin{bmatrix} -\sum_i w_i I_x(x_i, y_i) I_t(x_i, y_i) \\ -\sum_i w_i I_y(x_i, y_i) I_t(x_i, y_i) \end{bmatrix}$$

Finally, it's interesting to analyze the eigenvalue decomposition of the matrix $(\mathbf{A}^\top \mathbf{A})^{-1}$, which is analogous to equation 5.4 from the Harris detector. In case that the difference between both eigenvalues is too large (i.e. λ_1/λ_2 is large), it means that the window Ω contains an edge, and hence it's affected by the aperture problem (see Figure 5.4). On the other hand, if both λ_1 and λ_2 are large and have similar magnitude, the region Ω will contain a corner.

The implications of the eigenvalue analysis is clear: some points in the image are better than others for the Lucas-Kanade method. Specifically, corners with large eigenvalues (say, larger than a user-defined constant) will be tracked better.

In practice, in our feature tracker for visual-inertial odometry, we use this fact to initialize the tracking process. When the first image is received, the corner detector is used to find a number of *suitable corners*. The selected corners will have large eigenvalues and will be evenly spaced in the image. The Lucas-Kanade algorithm will be given these corners and will track them until they disappear. New suitable corners will be computed on a regular basis to keep a reasonable count of visible features.

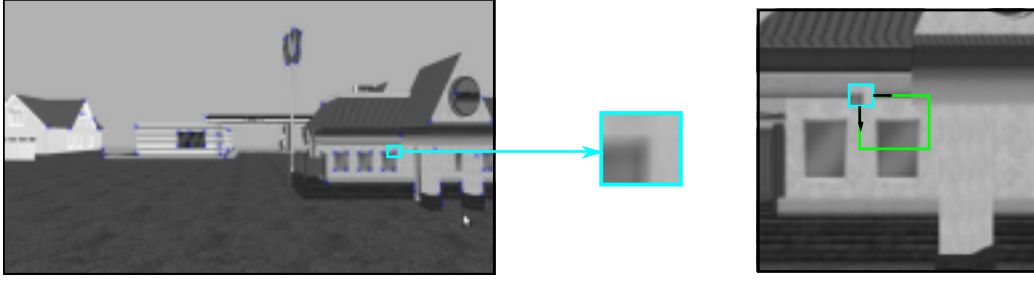


Figure 5.5: **Patch extraction and matching.** The reference patch is extracted the first time a feature is observed (left). Later it will be slid over an associated patch centered at the new feature position (right). The position that yields the highest patch similarity is considered a match.

Correlation-based feature matching correction

We use the Lucas-Kanade algorithm from the previous section to track features. The main drawback of using this method for tracking purposes is that the feature drifts with time. The reason for this is that Lucas-Kanade uses only the last frames to compute the optical flow. When a point is tracked for a long time, its appearance in the image may change due to changes in perspective or scale. This implies that, for long tracks, the point tracked at the end might not be the same as the initial one. When triangulating the feature position, this leads to errors that may impact negatively on the filter performance.

The solution to this consists basically in correcting every new observation by comparing its appearance to the first one. Two things are needed for this: a way to describe the appearance of a point, and a method to compare the appearances of different points.

The appearance of a point is described with a small rectangular patch centered at the pixel detected by the feature detector. This will be considered the *reference patch*. On the other hand, another patch will be created every time the tracker matches that point: we can call this the *associated patch*.

To compare different points, we need to *slide* the reference patch through the associated patch. Sliding means putting one patch over the other and moving it one pixel at a time (left to right, up to down). For each pixel, a *similarity metric* is computed to describe how close both patches are. This yields the *result matrix* in which each element contains the result of the similarity metric in a sliding position.

There exist several alternatives for the similarity metric, but we use the *Normalized Cross-Correlation* (NCC). Other options include the Sum of Squared Differences (SSD), simple Cross-Correlation (CC), or Zero-mean Normalized Cross-Correlation (ZNCC), but these are not described in this document. The NCC is computed according to the formula:

$$\mathbf{R}(x, y) = \frac{\sum_{x', y'} (\mathbf{I}(x', y') \cdot \mathbf{J}(x + x', y + y'))}{\sqrt{\sum_{x', y'} \mathbf{I}(x', y')^2 \cdot \sum_{x', y'} \mathbf{J}(x + x', y + y')^2}} \quad (5.13)$$

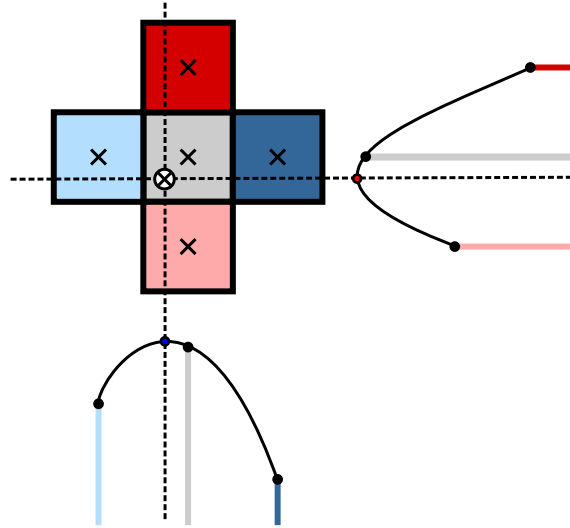


Figure 5.6: **Parabolic interpolation.** One parabolic interpolation for each direction is done in the four surrounding pixels of the result matrix to obtain a sub-pixellic result. Darker colours represent lower NCC values. The maximum in the parabolas correspond to the most similar match for both patches.

where \mathbf{I} is the reference patch defined when the feature was detected, \mathbf{J} is the associated patch that we are currently testing, and \mathbf{R} is the result matrix. Using a normalized metric leads to a measure which is invariant to contrast variations.

The computation of the NCC yields the result matrix \mathbf{R} , which contains the NCC measure at each pixel. To find the pixel that makes the two patches most similar, we just need to find the maximum value in the result matrix. The problem with this is that it will yield an integer pixel value, while in practice the feature won't typically lie exactly at the center of a pixel. To achieve a sub-pixellic feature location, we use parabolic interpolation of the result matrix around the pixel with a highest NCC measure (see Figure 5.6).

5.3 The pinhole camera model

Cameras are sensors that provide rich information about the surrounding world. A camera maps points from the 3D world to 2D images, so we can use those images to infer the location of objects in the environment. Such locations are needed to construct a map of the environment (as in SLAM methods), or, what is more relevant in our odometry case, to estimate the camera pose.

There are several camera models with different complexity and accuracy levels, but we are using one of the simplest: the *pinhole camera model*. This model assumes for instance that no lenses are used and includes no geometric distortion or blur. Although unrealistic, this model is really popular and performs quite well in practice.

In general, we want to project a 3D point $\mathbf{X}^g = (X, Y, Z)$ in *global frame* into the camera's image plane to obtain the 2D *pixel* coordinates \mathbf{x} . Such projection can be represented by the projection matrix \mathbf{P} . Using homogeneous vectors for the point

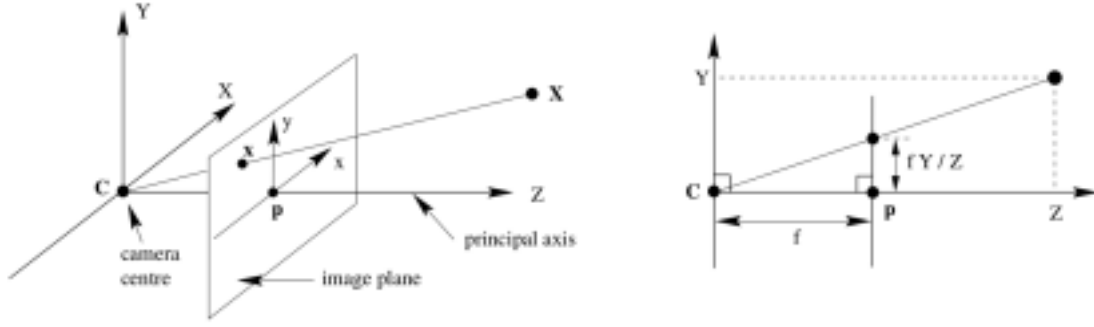


Figure 5.7: The pinhole camera model. Source: [8].

coordinates, the projection is easily expressed:

$$\mathbf{x} = \mathbf{P}\mathbf{X}^{\mathcal{G}} \quad (5.14)$$

The matrix \mathbf{P} performs two different transformations and can be separated in two parts. First, for the 3D point to be projected, its position must be expressed in relation to the camera's principal point. This is not the case: our 3D point $\mathbf{X}^{\mathcal{G}}$ is in global frame (noted by the superscript \mathcal{G}). The first transformation moves the point $\mathbf{X}^{\mathcal{G}}$ to $\mathbf{X}^{\mathcal{C}}$, which means that the same point is now expressed in the *camera frame* \mathcal{C} . It uses the *extrinsic* parameters of the camera, i.e. the position and orientation of the camera with respect to the global frame. The second transformation performed by the matrix \mathbf{P} projects the point $\mathbf{X}^{\mathcal{C}}$ to the camera's image plane to obtain the pixel coordinates (u, v) . This projection depends on the camera's *intrinsic* parameters, such as pixel size.

After splitting matrix \mathbf{P} into two different transformations, Equation 5.14 can be rewritten as:

$$\mathbf{x} = \mathbf{K} \begin{bmatrix} \mathbf{R} & | & \mathbf{t} \end{bmatrix} \mathbf{X}^{\mathcal{G}} \quad (5.15)$$

where \mathbf{K} is the *camera calibration matrix* that contains the camera's intrinsic parameters, and $\begin{bmatrix} \mathbf{R} & | & \mathbf{t} \end{bmatrix}$ is the matrix with the extrinsic parameters that relate the camera to the global frame. The following sections explain both matrices in detail.

Intrinsic parameters: \mathbf{K}

Figure 5.7 shows the ideal pinhole camera model [8]. The *camera center* is considered the center of the camera coordinate frame, in which all points must be expressed to be projected onto the *image plane*. The image plane is a plane parallel to the XY plane of the camera coordinate system and its distance from the camera center is the focal length f . The projected point \mathbf{x} is the intersection between the image plane and the line that joins the 3D point $\mathbf{X}^{\mathcal{C}}$ to the camera center. The line from the camera center perpendicular to the image plane is the *principal axis*, and the intersection of this line with the image plane is the *principal point*. This point corresponds to the image center.

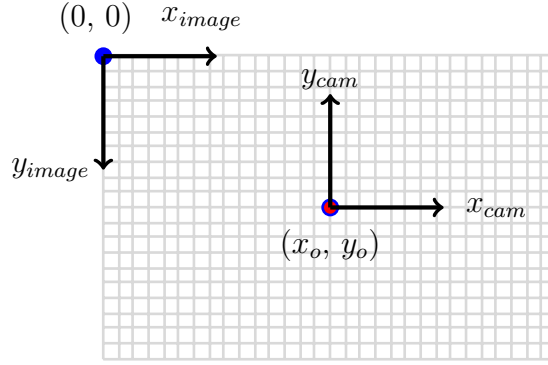


Figure 5.8: Image and camera coordinate systems

As can be seen in Figure 5.7 (right), the projected point (u, v) can be computed from the 3D point (X, Y, Z) by a simple triangulation using the focal length f . This focal length is in world units (e.g. millimeters), but we would like the point (u, v) to be in image units (pixels). The conversion between both units can be done by dividing the focal length f by the pixel width w and height h :

$$f_x = \frac{f}{w} \quad f_y = \frac{f}{h} \quad (5.16)$$

Since pixels might not be square, f_x and f_y might have different values. Now we can project the 3D point to obtain the 2D pixel coordinates;

$$u = \frac{X f_x}{Z} \quad v = \frac{Y f_y}{Z} \quad (5.17)$$

Equation 5.16 assumes that the origin of coordinates in the image plane lies on the principal point (red dot in Figure 5.8). However, in general the image coordinate center lies elsewhere, typically on the image upper-left corner (blue dot in Figure 5.8). Hence we must perform a translation to obtain the pixel coordinates. This translation transforms the *camera coordinate frame* to the *image coordinate frame*. Equation 5.17 can be rewritten to take this transformation into account:

$$u = \frac{X f}{Z} + x_o \quad v = \frac{Y f}{Z} + y_o \quad (5.18)$$

Finally, all this transformations can be represented in matrix form using homogeneous coordinates and dividing the result by Z^c to obtain the pixel coordinates (u, v) :

$$Z^c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f x & 0 & x_o & 0 \\ 0 & f_y & y_o & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X^c \\ Y^c \\ Z^c \\ 1 \end{bmatrix} \quad (5.19)$$

Defining:

$$\mathbf{K} = \begin{bmatrix} fx & 0 & x_o \\ 0 & fy & y_o \\ 0 & 0 & 1 \end{bmatrix} \quad (5.20)$$

we obtain the *camera calibration matrix* \mathbf{K} from Equation 5.15, which is part of the projection matrix \mathbf{P} .

The calibration matrix defines the geometric parameters of a camera. They can be computed by an algorithm by taking several views of a pre-defined calibration pattern (usually a chess board).

Extrinsic parameters

The previous section explained how to project a 3D point in *camera frame* to obtain its pixel coordinates. However, 3D points are usually expressed in a different coordinate frame, namely the *global coordinate frame*. Before projecting the point we must convert it to the camera frame, which can be done through a translation and a rotation:

$$\mathbf{X}^c = \mathbf{R}(\mathbf{X}^g - \mathbf{C}) \quad (5.21)$$

where the point \mathbf{C} is the position of the camera center expressed in global frame, and \mathbf{R} is the rotation matrix that expresses the orientation of the camera with respect to the global frame.

Equation 5.21 can be rewritten in homogeneous coordinates [8]:

$$\mathbf{X}^c = \begin{bmatrix} \mathbf{R} & -\mathbf{RC} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} X^g \\ Y^g \\ Z^g \\ 1 \end{bmatrix} \quad (5.22)$$

Finally, we can put all together to obtain the final projection from global to image coordinates:

$$\mathbf{x} = \mathbf{K} \begin{bmatrix} \mathbf{R} & | & \mathbf{t} \end{bmatrix} \mathbf{X}^g \quad (5.23)$$

where $\mathbf{t} = -\mathbf{RC}$.

5.4 Triangulation methods

The previous section shows how to detect and track features in a set of images, and how to model the camera geometry. In order to effectively use the information contained in the images in our algorithm, we need a way to extract the 3D position of the observed features. Consider the case where a feature located at \mathbf{p}_f^g has been

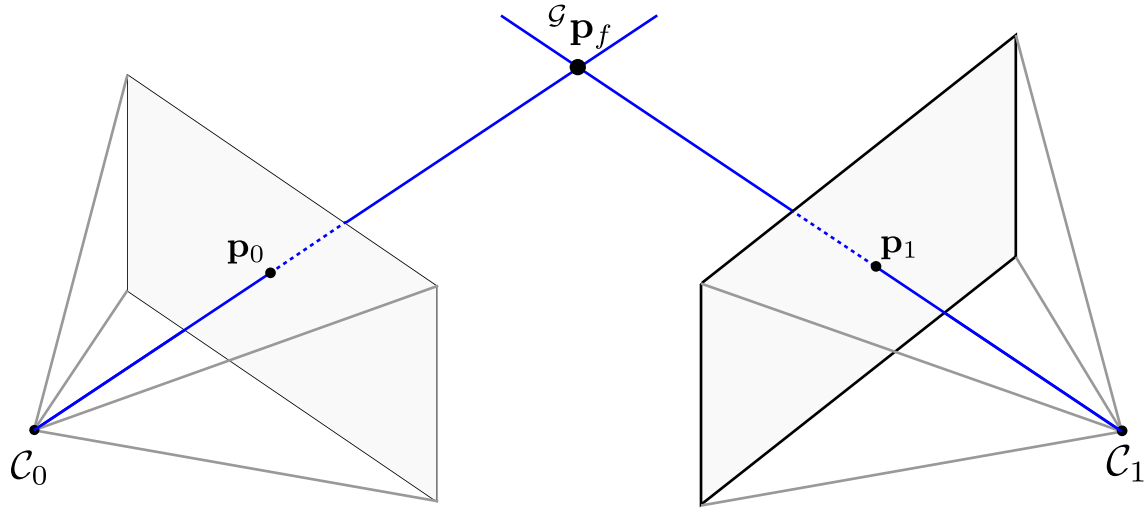


Figure 5.9: **Ideal triangulation:** both projected lines intersect at ${}^G\mathbf{p}_f$.

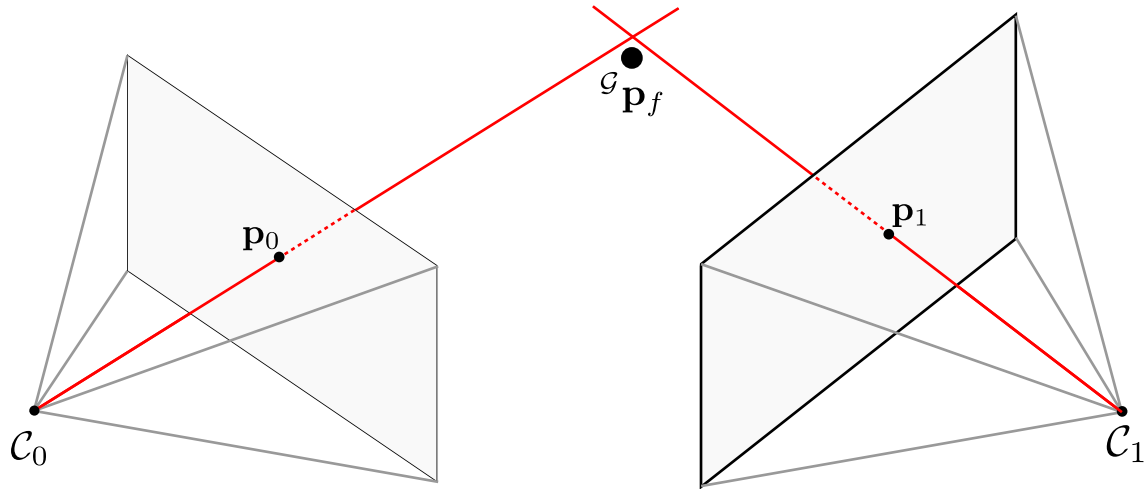


Figure 5.10: **Real triangulation:** lines don't intersect at ${}^G\mathbf{p}_f$ (or don't intersect at all).

observed from different camera locations C_0, C_1, \dots, C_n and has been projected to each image plane at $\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_n$. In an ideal world with perfect cameras, no errors and no noise, tracing a ray from each camera center to each pixel where the feature has been observed would intersect at the feature position ${}^G\mathbf{p}_f$. Figure 5.9 shows this ideal case with two camera locations. In practice, though, this perfect intersection is not possible because the measured projected points differ from the real, ideal projections for several reasons (see Figure 5.10). For instance, digital cameras represent images with discrete values (pixels), but projections won't match the pixel center perfectly. Another reason is that the pixels to be projected are found using feature detectors, which may have some errors when defining e.g. a corner. Other reasons for the deviation from the ideal cases include imperfections in the pinhole camera model (such as unmodeled distortions) or the presence of image noise.

In practice, then, we need to find a reliable way to triangulate the observed features to obtain their coordinates in the 3D world. Section 5.4 explains a simple method

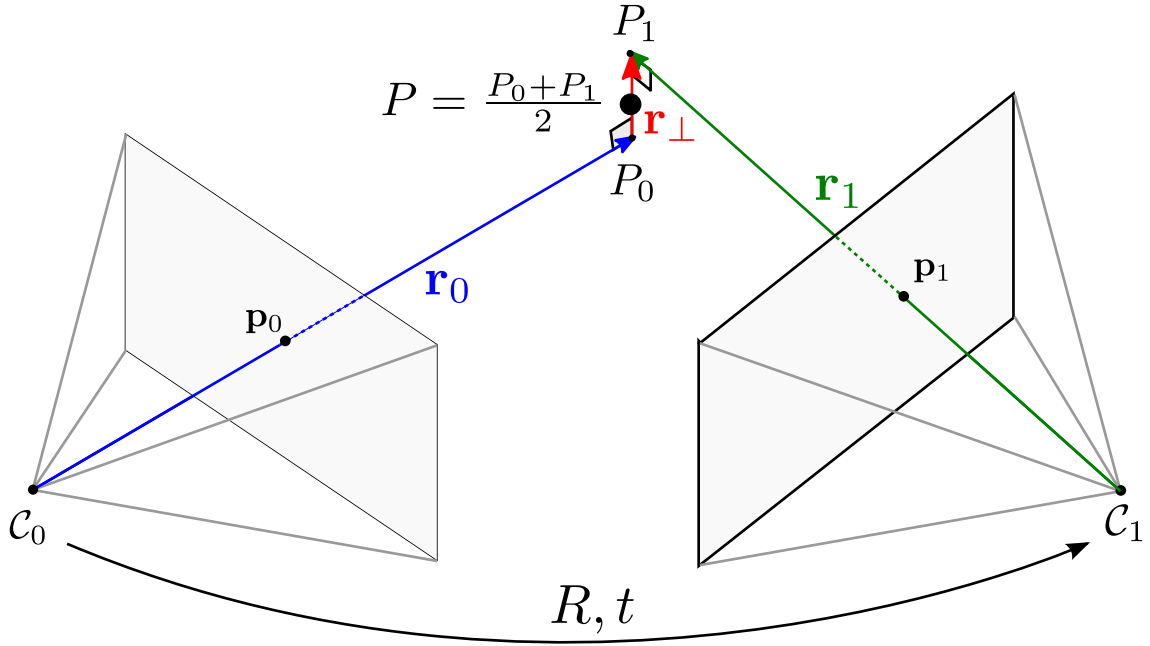


Figure 5.11: Midpoint triangulation method.

to triangulate a feature from two observations. The result of this method will be used as an initial estimate to the optimization process from section 5.4.

The midpoint intersection method

This method [9] finds the midpoint of a line perpendicular to the two rays that join the center of each camera with the observed feature. As Figure 5.11 shows, \mathbf{r}_0 and \mathbf{r}_1 are the rays corresponding to each projected point. These two rays will not intersect due to noise and model imperfections. This method computes a line \mathbf{r}_\perp , which is perpendicular to both \mathbf{r}_0 and \mathbf{r}_1 . Since there are many perpendicular lines, we need to find the points \mathbf{P}_0 and \mathbf{P}_1 that minimize the distance between both rays. If \mathbf{x}_i is the position of the i th camera center and \mathbf{R}_i is the rotation of camera i , the projected rays are:

$$\mathbf{r}_i = \mathbf{R}_i \mathbf{K}^{-1} \mathbf{u}_i \quad (5.24)$$

where \mathbf{u}_i is the measured pixel and \mathbf{K} is the camera calibration matrix from Equation 5.20. Any point in the ray can be encoded with a length α_i from the camera center:

$$\mathbf{P}_i = \mathbf{c}_i + \alpha_i \mathbf{r}_i \quad (5.25)$$

Since we have two rays, we need to find the α_0 and α_1 parameters that minimize the (squared) distance between both rays:

$$\mathbf{c}_0 + \alpha_0 \mathbf{r}_0 = \mathbf{c}_1 + \alpha_1 \mathbf{r}_1 \quad (5.26)$$

This bears three equations in two unknowns (the length parameters α_0 and α_1) and can be solved using linear least squares. The result will be the two points P_0 and P_1 . Finally, the midpoint P can be found with:

$$P = \frac{P_0 + P_1}{2} \quad (5.27)$$

This method is really easy to compute but is not optimal, because “perpendicularity is not an affine and midpoint not a projective concept” [9]. It also suffers from singularities in some circumstances (e.g. parallel rays). However, the midpoint method is still useful to find an initial estimate for the Gauss-Newton minimization process explained in the next section.

Gauss-Newton minimization with inverse-depth

As explained in the previous sections, triangulating a feature that has been observed from a set of cameras implies solving a system of equations. However, in the real world this system has no solution due to measurement errors. Even if no solution exists, it is possible to find an *optimal* solution. This means finding the 3D point that *minimizes* the error of the observations. This error is defined with the cost function \mathbf{f} :

$$\mathbf{f}(\boldsymbol{\theta}) = \mathbf{z} - \mathbf{h}(\boldsymbol{\theta}) \quad (5.28)$$

Equation 5.28 computes the difference between the *observed* value \mathbf{z} and the *expected* value, given the parameter $\boldsymbol{\theta}$. Function \mathbf{h} projects the parameter onto measurement space.

We need to find the feature position $\mathbf{p}_{f_j}^{\mathcal{G}}$ that minimizes the errors given by the cost function. This feature has been tracked in N consecutive images $\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_{N-1}$, where \mathcal{C}_i is the camera frame (i.e. rotation and translation) corresponding to the i -th observation.

In our case, each measurement \mathbf{z}_i^j corresponds to the measured pixel coordinates $(u_i^j, v_i^j)^\top$ for the feature f_j , observed from the camera frame \mathcal{C}_i .

The parameter $\boldsymbol{\theta}$ to be optimized corresponds to the 3D feature position $\mathbf{p}_{f_j}^{\mathcal{G}}$. During the optimization process, this feature position is not represented by the traditional Euclidean point $(X_j, Y_j, Z_j)^\top$. Instead, to avoid local minima, it is parametrized using the ray from the position from which it was first observed, together with the inverse depth of that ray [4]. We define the inverse-depth parameters for the feature position:

$$\mathbf{r}_j^{\mathcal{G}} = \mathbf{p}_{f_j}^{\mathcal{G}} - \mathbf{t}_0^{\mathcal{G}} = \begin{bmatrix} r_j^x \\ r_j^y \\ r_j^z \end{bmatrix} \quad (5.29)$$

$$\alpha_0 = \frac{r_j^x}{r_j^z} \quad \beta_0 = \frac{r_j^y}{r_j^z} \quad \rho_0 = \frac{1}{r_j^z} \quad (5.30)$$

where $\mathbf{r}_j^{\mathcal{G}}$ is the ray from the first camera position to the feature in *global frame*, and ρ_0 is the inverse depth. The more familiar Euclidean coordinates of the feature position can be recovered from the parameters α_0 , β_0 and ρ_0 :

$$\mathbf{p}_{f_j}^{\mathcal{G}} = \mathbf{t}_0^{\mathcal{G}} + \frac{1}{\rho_0} \begin{bmatrix} \alpha_0 \\ \beta_0 \\ 1 \end{bmatrix} \quad (5.31)$$

It is important to note that the ray $\mathbf{r}_j^{\mathcal{G}}$ is referenced to the first camera frame from which the feature was observed, as denoted by the subscripts in α_0 , β_0 and ρ_0 . This frame of reference different than the origin is known as the *anchor* [22] and improves the linearity conditions during the optimization process.

We can now use the measurements \mathbf{z}_i^j and their corresponding camera poses \mathcal{C}_i to obtain the estimates $\hat{\alpha}_0$, $\hat{\beta}_0$ and $\hat{\rho}_0$ using non-linear least squares minimization [17]. Our cost function (see Equation 5.28) represents the reprojection error of each observation to each camera pose. The observation model $\mathbf{h}(\boldsymbol{\theta})$ used there corresponds to:

$$\begin{bmatrix} \hat{x}_i^j \\ \hat{y}_i^j \\ \hat{w}_i^j \end{bmatrix} = \mathbf{K} \mathbf{R}_i^{\top} \left(\mathbf{t}_0^{\mathcal{G}} - \mathbf{t}_i^{\mathcal{G}} + \frac{1}{\hat{\rho}_0} \begin{bmatrix} \hat{\alpha}_0 \\ \hat{\beta}_0 \\ 1 \end{bmatrix} \right) \quad (5.32)$$

$$\hat{\mathbf{z}}_i^j = \begin{bmatrix} \hat{u}_i^j \\ \hat{v}_i^j \end{bmatrix} = \frac{1}{\hat{w}_i^j} \begin{bmatrix} \hat{x}_i^j \\ \hat{y}_i^j \end{bmatrix} \quad (5.33)$$

where \mathbf{K} is the camera intrinsic matrix from Equation 5.20, \mathbf{R}_i and $\mathbf{t}_i^{\mathcal{G}}$ are the extrinsic parameters for the camera \mathcal{C}_i (see section 5.3) and $\mathbf{t}_0^{\mathcal{G}}$ is the position of the first camera \mathcal{C}_0 from which the feature was observed (i.e. the anchor). Finally, the residual (or cost) is computed by subtracting the observed value from the expected one:

$$\mathbf{f}(\boldsymbol{\theta}_i^j) = \mathbf{z}_i^j - \hat{\mathbf{z}}_i^j \quad (5.34)$$

The optimization process computes the residuals for all the feature observations using the inverse-depth for higher linearity. The result in the euclidean point format can be recovered using Equation 5.31.

6

Multi-State Constraint Kalman Filter

The Multi-State Constraint Kalman Filter (MSCKF) [2] is the visual inertial odometry algorithm implemented in this work. It is similar to inertial-aided EKF SLAM, the main difference being that the MSCKF does not build a map by adding the feature positions to the state. Instead, it uses a sliding window of past poses which are used to triangulate the observed features without adding them to the state. On one hand, using a map provides valuable information by accounting for the correlations between the feature positions. However, processing that amount of information is expensive in computational terms.

The MSCKF makes optimal use of the geometric constraints that arise from observing the same feature in different images, without adding them to the state. This makes the computational cost linear in the number of features. Another asset of the MSCKF is the delayed utilization of feature information. Features are only used to correct the filter when they have moved out of view or they are too old, which makes the triangulation of the feature more accurate.

This chapter starts with a brief overview of how the MSCKF algorithm works. Next the state vector is defined. The state prediction and augment steps are then explained. Right after this the measurement model is developed, taking into account the feature information to correct the filter. Finally, the filter correction step is described.

6.1 Overview

The aim of the MSCKF consists in estimating the pose of the IMU frame \mathcal{I} with respect to the global frame \mathcal{G} . Figure 6.1 illustrates the coordinate frames used. The global frame \mathcal{G} is the reference frame to track the IMU pose. It could be placed anywhere, but we arbitrarily define it as the initial pose when the algorithm is started. It is convenient to align its orientation with the gravity vector: this will simplify the gravity correction for the accelerometer measurements. The second frame we use corresponds to the IMU and is represented by \mathcal{I} . The IMU is moving all the time and its pose with respect to \mathcal{G} , denoted in Figure 6.1 as $\mathbf{R}^{\mathcal{GI}}$ and $\mathbf{t}^{\mathcal{GI}}$, is constantly tracked by the algorithm. Finally, we have the camera frame \mathcal{C} , which is fixed with respect to the IMU and displaced of it by $\mathbf{R}^{\mathcal{IC}}$ and $\mathbf{t}^{\mathcal{IC}}$. This pose

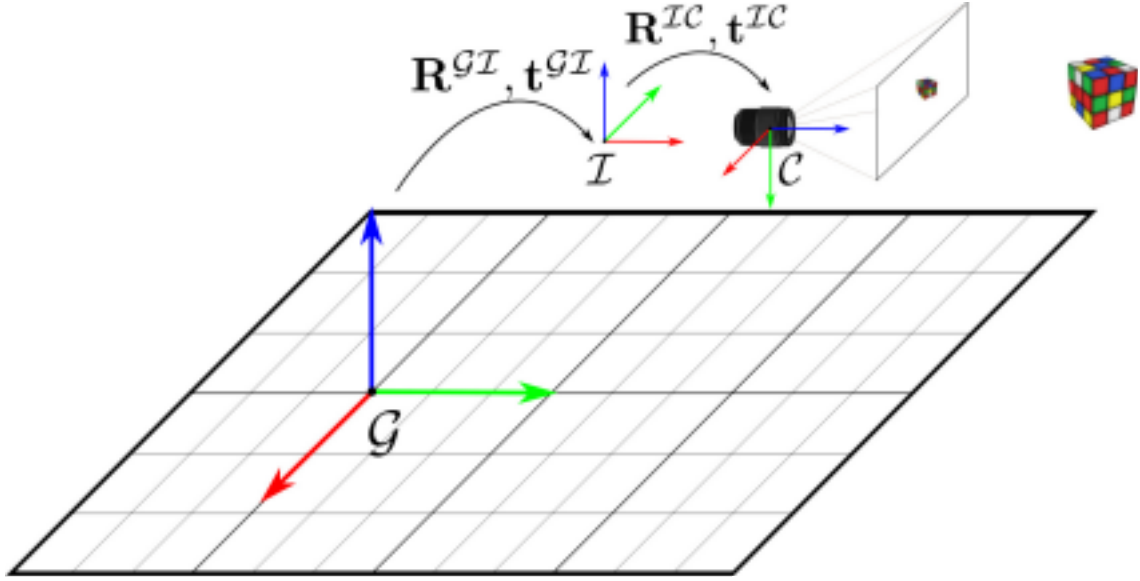


Figure 6.1: **Frame coordinates for visual inertial odometry:** the global frame \mathcal{G} , the IMU (or body) frame \mathcal{I} , and the camera frame \mathcal{C} . The transformations \mathbf{R}, \mathbf{t} between them are also displayed.

doesn't change during execution and must be manually defined at the beginning after a calibration process.

For the tracking of the IMU frame, IMU measurements sensed at \mathcal{I} and images taken at \mathcal{C} are used. Inertial measurements will usually come at a high rate (100 Hz) while camera frames come at a lower rate (30 Hz).

First, every time an IMU measurement is received, it is used to propagate the nominal and error states. This implies integrating the sensed acceleration and angular rate values to obtain the new position, velocity and orientation. The covariance is also propagated: it must grow after every measurement to reflect the increase in the uncertainty of the state. This step is really fast and is repeated for every measurement.

Second, when an image arrives, several tasks are done. On the first place, the state is augmented by pushing the current camera pose to the sliding window and updating the covariance accordingly. Figure 6.2 shows an example of a sliding window with four camera poses. After the state augment is done, the feature tracker processes the new image. It will find the features that are currently being tracked and detect the ones that disappeared, while looking for new feature tracks if needed.

When the image processing finishes, only the feature tracks that have finished or the ones that are longer than the state vector are selected for a filter update. A track is a set of observations of the same feature in different images. For the update, the feature positions are triangulated using Gauss-Newton optimization. Since all tracks will be as long as possible and their processing will be delayed, the triangulation will be quite accurate. The reprojection errors of the estimated feature positions with respect to the observed values will be used as the filter residuals. The residuals will be used to perform the filter correction, which will yield the observed error state. The error state will then be injected to the nominal state and reset to zero.

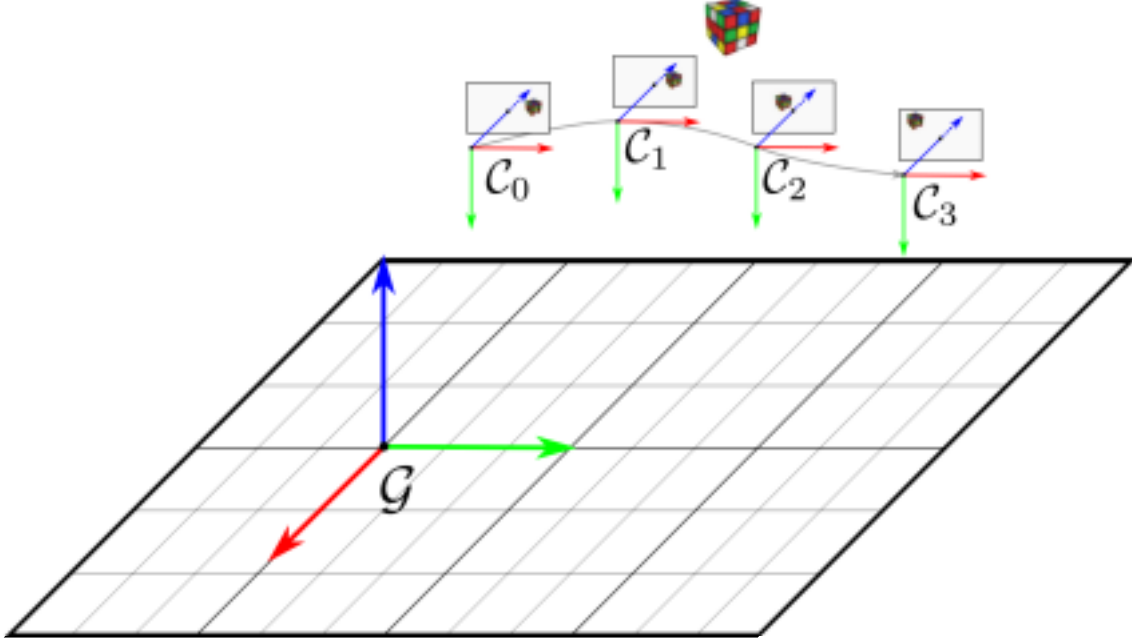


Figure 6.2: **Sliding window of past poses.** The window contains the camera poses \mathcal{C}_i every time a new image arrived. The trajectory between consecutive poses (in gray) is computed by integrating the IMU measurements.

Algorithm 1 shows the top-level structure of the MSCKF algorithm. See the next sections for details on each step.

6.2 State vector

The state vector includes the parameters that are constantly estimated by the filter. In our case, we can split the full nominal state \mathbf{x} in two main blocks: on one hand we have the IMU state \mathbf{x}_{IMU} ; on the other there is the sliding window of past camera poses. The IMU state contains the current estimated magnitudes for the IMU with respect to the global frame \mathcal{G} , together with the accelerometer and gyroscope biases:

$$\mathbf{x}_{IMU} = [\mathbf{p}^{\mathcal{GI}\top} \quad \mathbf{v}^{\mathcal{GI}\top} \quad \mathbf{q}^{\mathcal{GI}\top} \quad \mathbf{b}_a^\top \quad \mathbf{b}_\omega^\top]^\top \quad (6.1)$$

The full nominal state adds the sliding window to the IMU state. Assuming that the sliding window has room for $N - 1$ past poses:

$$\mathbf{x} = [\mathbf{x}_{IMU}^\top \quad \boldsymbol{\pi}_1^\top \quad \boldsymbol{\pi}_2^\top \quad \cdots \quad \boldsymbol{\pi}_N^\top]^\top \quad (6.2)$$

where $\boldsymbol{\pi}_i$ corresponds to the camera pose at frame i :

$$\boldsymbol{\pi}_i = [\mathbf{p}^{\mathcal{GC}_i\top} \quad \mathbf{q}^{\mathcal{GC}_i\top}]^\top \quad (6.3)$$

At this point it is important to note that the IMU state stores the magnitudes in the IMU frame \mathcal{I} while the sliding window stores camera poses in frame \mathcal{C} .

Algorithm 1 MSCKF algorithm overview

```

for new IMU measurement  $\mathbf{a}_m, \boldsymbol{\omega}_m, \Delta t$  do
  PROPAGATE STATE( $\mathbf{a}_m, \boldsymbol{\omega}_m, \Delta t$ )
end for

for new Image  $\mathbf{I}_i$  do
  TRACK FEATURES( $\mathbf{I}_i$ )
  if FINISHED TRACKS is not empty then
    UPDATE FILTER
  end if
end for

```

Since the MSCKF uses the error state formulation of the Kalman Filter, we must keep both the nominal (presented above) and the error state. In practice, though, the error state is never propagated nor stored. This is due to Equation 4.13a: since the error state is always reset to zero, its propagation returns always zero. The only moment when the error state is actually observed and computed is during the filter update. In any case, we define the error state analogously to the nominal state, starting with the IMU block:

$$\delta \mathbf{x}_{IMU} = \begin{bmatrix} \delta \mathbf{p}^{GI\top} & \delta \mathbf{v}^{GI\top} & \delta \boldsymbol{\theta}^{GI\top} & \delta \mathbf{b}_a^\top & \delta \mathbf{b}_\omega^\top \end{bmatrix}^\top \quad (6.4)$$

As explained in section 4.1, the orientation error is represented by the error angle $\delta \boldsymbol{\theta}$ in the global reference frame. This representation is minimal because it needs three parameters for three degrees of freedom (differently to the quaternion, that needs four parameters). And since the error angles are always small, the usual Gimbal lock issues associated to the Euler angles representation do not arise. The global frame means that the composition with the nominal orientation goes to the left : $\mathbf{q}_t = \delta \mathbf{q} \otimes \mathbf{q}$. The error quaternion can be recovered from the error angle through the formula $\delta \mathbf{q} = \text{vec2q}(\delta \boldsymbol{\theta})$ from Equations 4.10.

Akin to the nominal state, the full error state includes the error components for the sliding window:

$$\delta \mathbf{x} = \begin{bmatrix} \delta \mathbf{x}_{IMU}^\top & \delta \boldsymbol{\pi}_1^\top & \delta \boldsymbol{\pi}_2^\top & \dots & \delta \boldsymbol{\pi}_N^\top \end{bmatrix}^\top \quad (6.5)$$

where $\delta \boldsymbol{\pi}_i^\top$ represents the error pose of the camera at frame i , using the error angle parameterization:

$$\delta \boldsymbol{\pi}_i = \begin{bmatrix} \delta \mathbf{p}^{G\mathcal{C}_i\top} & \delta \boldsymbol{\theta}^{G\mathcal{C}_i\top} \end{bmatrix}^\top \quad (6.6)$$

Finally, we only need to describe the covariance matrix, which represents the uncertainty of the error state and the cross covariances between all the state components. For that reason it contains the same layout as the error state. Using matrix block, notation, the covariance matrix can be described by:

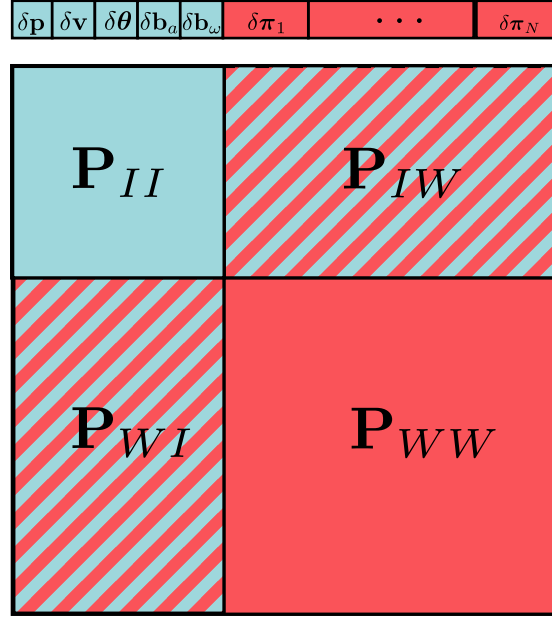


Figure 6.3: **Error state and covariance matrix.** The IMU components are shown in blue, while the sliding window is displayed in red. \mathbf{P}_{IW} corresponds to the cross-variance of the IMU with the sliding window.

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{II} & \mathbf{P}_{IW} \\ \mathbf{P}_{IW}^T & \mathbf{P}_{WW} \end{bmatrix} \quad (6.7)$$

where \mathbf{P}_{II} represents the covariance of the IMU error state, \mathbf{P}_{IW} is the cross-covariance of the IMU with the sliding window, and lastly \mathbf{P}_{WW} is the covariance of the sliding window. Figure 6.3 presents a schema of the error state and the covariance matrix.

6.3 State propagation

The propagation step is carried out every time an IMU measurement is received, and it basically consists in integrating the sensed magnitudes into the state and propagating the state covariances. The discrete propagation equations are derived in section 4.2 from the continuous-time system by using Euler integration.

The nominal state is propagated using Equations 4.9. For the covariance propagation, we need to build the system transition matrix \mathbf{F}_x and the perturbation matrix \mathbf{Q}_i (see Equations 4.14).

Algorithm 2 illustrates the propagation step. There are several ways to compose the system matrix \mathbf{F}_x . The selected one corresponds to using Euler integration with global error parameterization. The perturbation matrix \mathbf{Q}_i is described in Equations 4.12 and contains the integrated variances of the IMU noises.

Algorithm 2 State propagation

function PROPAGATE STATE($\mathbf{a}_m, \boldsymbol{\omega}_m, \Delta t$)

 \triangleright 1. Propagate nominal state.

$$\mathbf{v} \leftarrow \mathbf{v} + (\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a) + \mathbf{g})\Delta t$$

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}\Delta t + \frac{1}{2}(\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a) + \mathbf{g})\Delta t^2$$

$$\mathbf{q} \leftarrow \mathbf{q} \otimes \text{vec2q}((\boldsymbol{\omega}_m - \mathbf{b}_\omega)\Delta t)$$

 \triangleright 2. Propagate state covariance.

$$\mathbf{F}_x \leftarrow \begin{bmatrix} \mathbf{I} & \mathbf{I}\Delta t & 0 & 0 & 0 \\ 0 & \mathbf{I} & -[\mathbf{R}(\mathbf{a}_m - \mathbf{b}_a)]_\times \Delta t & -\mathbf{R}\Delta t & 0 \\ 0 & 0 & \mathbf{I} & 0 & -\mathbf{R}\Delta t \\ 0 & 0 & 0 & \mathbf{I} & 0 \\ 0 & 0 & 0 & 0 & \mathbf{I} \end{bmatrix}$$

$$\mathbf{Q}_i \leftarrow \begin{bmatrix} \mathbf{W}_v & 0 & 0 & 0 \\ 0 & \mathbf{W}_\theta & 0 & 0 \\ 0 & 0 & \mathbf{W}_{ba} & 0 \\ 0 & 0 & 0 & \mathbf{W}_{b\omega} \end{bmatrix}$$

$$\mathbf{P}_{IMU} \leftarrow \mathbf{F}_x \mathbf{P}_{IMU} \mathbf{F}_x^\top + \mathbf{F}_i \mathbf{Q}_i \mathbf{F}_i^\top$$

end function

6.4 State augment

Every time an image arrives, the sliding window is augmented with the current *camera* pose. This implies updating the nominal state and the covariance as well. Algorithm 3 schematically shows how to augment the state and the covariance matrix.

Augmenting the nominal state

This is the simple part of the augment step, we just need to get the current IMU pose from the nominal state \mathbf{x}_{IMU} and transform it to the camera frame:

$$\mathbf{p}^{\mathcal{G}C_N} = \mathbf{p}^{\mathcal{G}I_N} + \mathbf{R}^{\mathcal{G}I} \mathbf{p}^{\mathcal{I}C} \quad (6.8a)$$

$$\mathbf{q}^{\mathcal{G}C_N} = \mathbf{q}^{\mathcal{G}I_N} \otimes \mathbf{q}^{\mathcal{I}C} \quad (6.8b)$$

The transformation from IMU to camera ($\mathbf{p}^{\mathcal{I}C}$ and $\mathbf{q}^{\mathcal{I}C}$) is assumed fixed and is obtained through the simulation parameters or by measuring the displacement between the IMU and the camera. As shown in [14], these parameters can also be calibrated during filter operation.

This pose is then added to the pertinent sliding window block of the nominal state:

$$\boldsymbol{\pi}_N = [\mathbf{p}^{\mathcal{G}C_N^\top} \quad \mathbf{q}^{\mathcal{G}C_N^\top}]^\top \quad (6.9)$$

Algorithm 3 State augment**function** AUGMENT STATE

▷ 1. Augment nominal state.

$$\mathbf{p}^{\mathcal{GC}} \leftarrow \mathbf{p}^{\mathcal{GI}} + \mathbf{R}^{\mathcal{GI}} \mathbf{p}^{\mathcal{IC}}$$

$$\mathbf{q}^{\mathcal{GC}} \leftarrow \mathbf{q}^{\mathcal{GI}} \otimes \mathbf{q}^{\mathcal{IC}}$$

$$\boldsymbol{\pi} \leftarrow \begin{bmatrix} \mathbf{p}^{\mathcal{GC}\top} & \mathbf{q}^{\mathcal{GC}\top} \end{bmatrix}^\top$$

ADD POSE TO STATE(\mathbf{x} , $\boldsymbol{\pi}$)

▷ 2. Augment state covariance.

$$\mathbf{J}_{\pi_N} \leftarrow \frac{\partial \delta \pi_N}{\partial \delta \mathbf{x}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & -[\mathbf{R}^{\mathcal{GI}} \mathbf{p}^{\mathcal{IC}_N}]_{\times} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \cdots & \mathbf{0} \end{bmatrix}$$

AUGMENT COVARIANCE(\mathbf{P} , \mathbf{J}_{π_N})

▷ See Figure 6.4

end function**Augmenting the covariance matrix**

When the new pose is added to the sliding window, the covariance must be updated to account for the uncertainty in that pose. The affected blocks of the covariance matrix correspond to the location of the new pose within the state vector.

Since the covariance matrix is used to estimate the error state (and not the nominal state), the magnitude of the covariance depends on the Jacobian of the *error* of the new pose $\delta \mathbf{p}^{\mathcal{GC}_N}$ with respect to the error state $\delta \mathbf{x}$. Since the only operation applied for the augment step is the frame transformation from Equations 6.8, we need to compute the Jacobian of that transformation with respect to the error state. Only the blocks related to the position and orientation are affected:

$$\mathbf{J}_{\pi_N} = \frac{\partial \delta \pi_N}{\partial \delta \mathbf{x}} = \begin{bmatrix} \frac{\partial \delta \mathbf{p}^{\mathcal{GC}_N}}{\partial \delta \mathbf{p}^{\mathcal{GI}}} & \mathbf{0} & \frac{\partial \delta \mathbf{p}^{\mathcal{GC}_N}}{\partial \delta \boldsymbol{\theta}^{\mathcal{GI}}} & \cdots & \mathbf{0} \\ \frac{\partial \delta \boldsymbol{\theta}^{\mathcal{GC}_N}}{\partial \delta \mathbf{p}^{\mathcal{GI}}} & \mathbf{0} & \frac{\partial \delta \boldsymbol{\theta}^{\mathcal{GC}_N}}{\partial \delta \boldsymbol{\theta}^{\mathcal{GI}}} & \cdots & \mathbf{0} \end{bmatrix} \quad (6.10)$$

Augment Jacobian with respect to the error position

Computing Jacobians with respect to error states is a bit more complex than usual. A good starting point is to consider that Equation 6.8a is using the true state and rewrite it to decompose it between the nominal and error components:

$$\mathbf{p}_t^{\mathcal{GC}_N} = \mathbf{p}_t^{\mathcal{GI}_N} + \mathbf{R}_t^{\mathcal{GI}_N} \mathbf{p}^{\mathcal{IC}}$$

which from the composition Equations 4.18, it leads to:

$$\mathbf{p}^{\mathcal{GC}_N} + \delta \mathbf{p}^{\mathcal{GC}_N} = \mathbf{p}^{\mathcal{GI}} + \delta \mathbf{p}^{\mathcal{GI}} + (\mathbf{I} + [\delta \boldsymbol{\theta}^{\mathcal{GI}}]_{\times}) \mathbf{R}^{\mathcal{GI}_N} \mathbf{p}^{\mathcal{IC}} \quad (6.11)$$

Note that since $\mathbf{p}^{\mathcal{IC}}$ is not estimated by the filter, it is not affected by any error component. This can be further derived:

$$\begin{aligned}\mathbf{p}^{\mathcal{GC}_N} + \delta\mathbf{p}^{\mathcal{GC}_N} &= \mathbf{p}^{\mathcal{GI}} + \delta\mathbf{p}^{\mathcal{GI}} + (\mathbf{R}^{\mathcal{GI}} + [\delta\boldsymbol{\theta}^{\mathcal{GI}}]_{\times} \mathbf{R}^{\mathcal{GI}}) \mathbf{p}^{\mathcal{IC}_N} \\ \mathbf{p}^{\mathcal{GC}_N} + \delta\mathbf{p}^{\mathcal{GC}_N} &= \mathbf{p}^{\mathcal{GI}} + \delta\mathbf{p}^{\mathcal{GI}} + \mathbf{R}^{\mathcal{GI}} \mathbf{p}^{\mathcal{IC}_N} + [\delta\boldsymbol{\theta}^{\mathcal{GI}}]_{\times} \mathbf{R}^{\mathcal{GI}} \mathbf{p}^{\mathcal{IC}_N}\end{aligned}$$

Isolating the error state components we get to:

$$\delta\mathbf{p}^{\mathcal{GC}_N} = \delta\mathbf{p}^{\mathcal{GI}} + [\delta\boldsymbol{\theta}^{\mathcal{GI}}]_{\times} \mathbf{R}^{\mathcal{GI}} \mathbf{p}^{\mathcal{IC}_N}$$

And noting that $[\mathbf{v}]_{\times} \mathbf{w} = -[\mathbf{w}]_{\times} \mathbf{v}$:

$$\delta\mathbf{p}^{\mathcal{GC}_N} = \delta\mathbf{p}^{\mathcal{GI}} - [\mathbf{R}^{\mathcal{GI}} \mathbf{p}^{\mathcal{IC}_N}]_{\times} \delta\boldsymbol{\theta}^{\mathcal{GI}} \quad (6.13)$$

From this we can finally compute the partial derivatives:

$$\frac{\partial \delta\mathbf{p}^{\mathcal{GC}_N}}{\partial \delta\mathbf{p}^{\mathcal{GI}}} = \mathbf{I} \quad \frac{\partial \delta\mathbf{p}^{\mathcal{GC}_N}}{\partial \delta\boldsymbol{\theta}^{\mathcal{GI}}} = -[\mathbf{R}^{\mathcal{GI}} \mathbf{p}^{\mathcal{IC}_N}]_{\times} \quad (6.14)$$

Augment Jacobian with respect to the error orientation

The error orientation jacobian is computed analogously as in the previous section. First we rewrite equation 6.8b in true state form:

$$\mathbf{q}_t^{\mathcal{GC}_N} = \mathbf{q}_t^{\mathcal{GI}_N} \otimes \mathbf{q}^{\mathcal{IC}} \quad (6.15)$$

And recalling the composition between the (global) error quaternion with the nominal quaternion:

$$\mathbf{q}_t = \delta\mathbf{q} \otimes \mathbf{q}$$

We can now split equation 6.15 into the error and nominal components:

$$\delta\mathbf{q}^{\mathcal{GC}_N} \otimes \mathbf{q}^{\mathcal{GC}_N} = \delta\mathbf{q}^{\mathcal{GI}} \otimes \mathbf{q}^{\mathcal{GI}} \otimes \mathbf{q}^{\mathcal{IC}_N} \quad (6.16)$$

Taking only the error terms:

$$\delta\mathbf{q}^{\mathcal{GC}_N} = \delta\mathbf{q}^{\mathcal{GI}} \quad (6.17)$$

From this we can finally compute the partial derivatives:

$$\frac{\partial \delta\boldsymbol{\theta}^{\mathcal{GC}_N}}{\partial \delta\mathbf{p}^{\mathcal{GI}}} = \mathbf{0} \quad \frac{\partial \delta\boldsymbol{\theta}^{\mathcal{GC}_N}}{\partial \delta\boldsymbol{\theta}^{\mathcal{GI}}} = \mathbf{I} \quad (6.18)$$

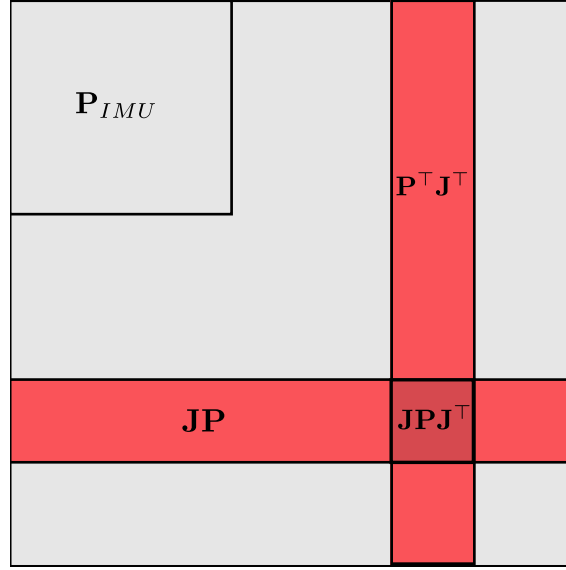


Figure 6.4: **Augment covariance.** Only the blocks corresponding to the position where the new pose was appended are affected by the augment process. Gray areas are not affected by the augment step.

Augment covariance

Finally, from Equations 6.10, 6.14 and 6.18, we can finally write the complete augment Jacobian (with the affected terms in red):

$$\mathbf{J}_{\pi_N} = \frac{\partial \delta \pi_N}{\partial \delta \mathbf{x}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & -[\mathbf{R}^{g\mathcal{I}} \mathbf{p}^{i\mathcal{C}_N}]_{\times} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \cdots & \mathbf{0} \end{bmatrix} \quad (6.19)$$

Since in practice the new pose can be added in any position, the Jacobian should only affect the blocks in the covariance matrix that correspond to that position. Figure 6.4 illustrates the update of the covariance matrix using the Jacobian \mathbf{J}_{π_N} .

6.5 Observation model

The state propagation accumulates errors with time. We correct these errors by using external measurements from a monocular camera. The observation model describes how to use the rich information present in the images to update the filter and observe the accumulated IMU errors.


Before diving into the mathematical details of the observation model, we need to explain which kind of data we need to extract from the images. Although images provide many different ways to obtain information of the surrounding world, we focus on feature corners due to their simplicity as they can represent points in space and are easily parametrized. In order to use the information provided by feature corners, two tasks need to be done. First, we need to detect corners in the images—and properly differentiate them from other kinds of features (such as edges)—.

Second, once we have detected some corners, we need to track them in a sequence of images.

Independently of the methods used to detect and track features (see next Chapter for details on our implementation), our aim consists in building long lasting, reliable *feature tracks*. A feature track is a sequence of corner detections which has been successfully tracked for several consecutive frames, without being occluded or moved out of view.

Observing a feature from different camera locations imposes constraints on these camera poses. Our observation model defines a constraint equation using all the observations of the same feature within a track to improve the estimation of such poses.

Consider a feature f_j which has been observed in K images. Each measurement \mathbf{z}_i^j has been observed from the camera pose \mathcal{C}_i and contains the pixel coordinates $(u_i^j, v_i^j)^\top$.

The first step consists in triangulating the feature position $\hat{\mathbf{p}}_{f_j}^{\mathcal{G}}$ using all the measurements from the track. This is done using Gauss-Newton optimization, as explained in Section 5.4. Only the feature position is optimized; the camera poses are considered fixed. To help with this process, an initial solution is computed first and fed into the optimizer using the mid-point intersection method, which is a simple method to triangulate a point using the first and last camera poses of the track (see Section 5.4).  **some tricks here**

Once we have obtained the feature position, we need to project it to every camera pose \mathcal{C}_i from which the feature has been observed. This is done by means of the observation function: $\hat{\mathbf{z}}_i^j = h(\hat{\mathbf{x}}) + \mathbf{v}$. This function transforms first the feature position from the global reference frame \mathcal{G} to the corresponding camera frame \mathcal{C}_i :

$$\hat{\mathbf{p}}_{f_j}^{\mathcal{C}_i} = \mathbf{R}^{\mathcal{G}\mathcal{C}_i\top} (\hat{\mathbf{p}}_{f_j}^{\mathcal{G}} - \mathbf{p}^{\mathcal{G}\mathcal{C}_i}) \quad (6.20)$$

And then, project the 3D point in the camera frame to obtain the estimated measurement $\hat{\mathbf{z}}_i^j = (\hat{u}_i^j, \hat{v}_i^j)^\top$, relying on Equation 5.18:

$$\hat{u}_i^j = \frac{x_j^i f_x}{z_j^i} + x_o \quad \hat{v}_i^j = \frac{y_j^i f_y}{z_j^i} + y_o \quad (6.21)$$

where f_x , f_y , x_o and y_o are the intrinsic parameters of the camera (see Section 5.3).

Now we can compute the *residual*, which is nothing else than the difference between the observed and the expected measurement:

$$\mathbf{r}_i^j = \mathbf{z}_i^j - \hat{\mathbf{z}}_i^j \quad (6.22)$$

To update the covariance with the residual, we need to linearize the observation function h around the current state estimate. This results in:

$$\mathbf{r}_i^j \simeq \mathbf{H}_{\delta\mathbf{x}_i}^j \delta\mathbf{x} + \mathbf{H}_{f_i}^j \delta\mathbf{p}_{f_j} + \mathbf{n} \quad (6.23)$$

where $\mathbf{H}_{\delta\mathbf{x}}$ and \mathbf{H}_{f_j} are the jacobians of the measurement with respect to the error-state and the feature position error, respectively, and \mathbf{n} is the image noise, which must be zero-mean, white and uncorrelated to the error state $\delta\mathbf{x}$.

Error state Jacobian $\mathbf{H}_{\delta\mathbf{x}}$

$\mathbf{H}_{\delta\mathbf{x}}$ is the Jacobian of the measurement \mathbf{z}_i^j with respect to the error state $\delta\mathbf{x}$:

$$\mathbf{H}_{\delta\mathbf{x}_i}^j = \frac{\partial \mathbf{z}_i^j}{\partial \delta\mathbf{x}} = \frac{\partial \mathbf{z}_i^j}{\partial \mathbf{p}_{f_j}^{C_i}} \frac{\partial \mathbf{p}_{f_j}^{C_i}}{\partial \delta\mathbf{x}} \quad (6.24)$$

where we have split the Jacobian into two parts by means of the chain rule.

The first term corresponds to the projection of the feature position $\hat{\mathbf{p}}_{f_j}^{C_i}$ into the image plane. Applying the partial derivatives to Equation 6.21 we obtain:

$$\mathbf{H}_{z_i}^j = \frac{\partial \mathbf{z}_i^j}{\partial \mathbf{p}_{f_j}^{C_i}} = \begin{bmatrix} \frac{f_x}{z_i^j} & 0 & -\frac{f_x \cdot x_i^j}{(z_i^j)^2} \\ 0 & \frac{f_y}{z_i^j} & -\frac{f_y \cdot y_i^j}{(z_i^j)^2} \end{bmatrix} \quad (6.25)$$

The second component of $\mathbf{H}_{\delta\mathbf{x}}$ is the Jacobian of the feature position with respect to the error state. Similarly to what we have done in Section 6.4, we need to rewrite Equation 6.20 including the error state components:

$$\mathbf{p}^{C\mathcal{F}} = \mathbf{R}_t^{G\mathcal{C}\top} (\mathbf{p}^{G\mathcal{F}} - \mathbf{p}_t^{G\mathcal{C}}) \quad (6.26)$$

where we use the true state and the notation $\mathbf{p}_{f_j}^{C_i}$ has been simplified into $\mathbf{p}^{C\mathcal{F}}$ for clarity. Now we can decompose the true states in Equation 6.26 into the error and nominal components:

$$\mathbf{p}^{C\mathcal{F}} = [(\mathbf{I} + [\delta\boldsymbol{\theta}]_{\times})\mathbf{R}^{G\mathcal{C}}]^\top \mathbf{p}^{G\mathcal{F}} - [(\mathbf{I} + [\delta\boldsymbol{\theta}]_{\times})\mathbf{R}^{G\mathcal{C}}]^\top (\mathbf{p}^{G\mathcal{C}} + \delta\mathbf{p}^{G\mathcal{C}}) \quad (6.27)$$

Noting that $[(\mathbf{I} + [\delta\boldsymbol{\theta}]_{\times})\mathbf{R}]^\top = \mathbf{R}^\top (\mathbf{I} - [\delta\boldsymbol{\theta}]_{\times})$, this leads to:

$$\begin{aligned} \mathbf{p}^{C\mathcal{F}} &= [\mathbf{R}^{G\mathcal{C}\top} (\mathbf{I} - [\delta\boldsymbol{\theta}]_{\times})] \mathbf{p}^{G\mathcal{F}} - [\mathbf{R}^{G\mathcal{C}\top} (\mathbf{I} - [\delta\boldsymbol{\theta}]_{\times})] (\mathbf{p}^{G\mathcal{C}} + \delta\mathbf{p}^{G\mathcal{C}}) \\ \mathbf{p}^{C\mathcal{F}} &= (\mathbf{R}^{G\mathcal{C}\top} - \mathbf{R}^{G\mathcal{C}} [\delta\boldsymbol{\theta}]_{\times}) \mathbf{p}^{G\mathcal{F}} - (\mathbf{R}^{G\mathcal{C}\top} - \mathbf{R}^{G\mathcal{C}} [\delta\boldsymbol{\theta}]_{\times}) (\mathbf{p}^{G\mathcal{C}} + \delta\mathbf{p}^{G\mathcal{C}}) \\ \mathbf{p}^{C\mathcal{F}} &= \mathbf{R}^{G\mathcal{C}\top} \mathbf{p}^{G\mathcal{F}} - \mathbf{R}^{G\mathcal{C}\top} [\delta\boldsymbol{\theta}]_{\times} \mathbf{p}^{G\mathcal{F}} - \mathbf{R}^{G\mathcal{C}\top} \mathbf{p}^{G\mathcal{C}} + \mathbf{R}^{G\mathcal{C}\top} [\delta\boldsymbol{\theta}]_{\times} \mathbf{p}^{G\mathcal{C}} \\ &\quad - \mathbf{R}^{G\mathcal{C}\top} \delta\mathbf{p}^{G\mathcal{C}} - \mathbf{R}^{G\mathcal{C}\top} [\delta\boldsymbol{\theta}]_{\times} \delta\mathbf{p}^{G\mathcal{C}} \end{aligned}$$

Taking only the error state components and ignoring high order terms we have:

$$\begin{aligned}
\mathbf{0} &= -\mathbf{R}^{\mathcal{GC}\top} [\delta\boldsymbol{\theta}]_{\times} \mathbf{p}^{\mathcal{GF}} + \mathbf{R}^{\mathcal{GC}\top} [\delta\boldsymbol{\theta}]_{\times} \mathbf{p}^{\mathcal{GC}} - \mathbf{R}^{\mathcal{GC}\top} \delta\mathbf{p}^{\mathcal{GC}} \\
\mathbf{0} &= -\mathbf{R}^{\mathcal{GC}\top} ([\delta\boldsymbol{\theta}]_{\times} \mathbf{p}^{\mathcal{GF}} + [\delta\boldsymbol{\theta}]_{\times} \mathbf{p}^{\mathcal{GC}} - \delta\mathbf{p}^{\mathcal{GC}}) \\
\mathbf{0} &= -\mathbf{R}^{\mathcal{GC}\top} ([\delta\boldsymbol{\theta}]_{\times} (\mathbf{p}^{\mathcal{GF}} + \mathbf{p}^{\mathcal{GC}}) - \delta\mathbf{p}^{\mathcal{GC}})
\end{aligned}$$

From this we obtain the Jacobian of the feature position with respect to the error position and orientation:

$$\frac{\partial \mathbf{p}_{f_j}^{\mathcal{C}_i}}{\partial \delta \mathbf{p}^{\mathcal{GC}_i}} = -\mathbf{R}^{\mathcal{GC}\top} \quad \frac{\partial \mathbf{p}_{f_j}^{\mathcal{C}_i}}{\partial \delta \boldsymbol{\theta}^{\mathcal{GC}_i}} = -\mathbf{R}^{\mathcal{GC}\top} [(\mathbf{p}^{\mathcal{GC}} - \mathbf{p}^{\mathcal{GF}})]_{\times} \quad (6.30)$$

The only relevant components for the Jacobian in the error state are the error position and orientation of the camera pose \mathcal{C}_i in the sliding window from which the feature f_j has been observed. All the other components in the state are zero. We can then group the partial derivatives from Equation 6.30 to define the Jacobian with respect of the sliding window's pose:

$$\mathbf{H}_{\pi_i}^j = \frac{\partial \mathbf{p}_{f_j}^{\mathcal{C}_i}}{\partial \pi_i} = \begin{bmatrix} -\mathbf{H}_{z_i}^j \mathbf{R}^{\mathcal{GC}\top} & -\mathbf{H}_{z_i}^j \mathbf{R}^{\mathcal{GC}\top} [(\mathbf{p}^{\mathcal{GC}} - \mathbf{p}^{\mathcal{GF}})]_{\times} \end{bmatrix} \quad (6.31)$$

Finally, the Jacobian with respect to the complete error state contains $\mathbf{H}_{\pi_i}^j$ and a lot of zeros:

$$\mathbf{H}_{\delta \mathbf{x}_i}^j = \begin{bmatrix} \mathbf{0}_{2 \times 15} & \mathbf{0}_{2 \times 6} & \cdots & \underbrace{\mathbf{H}_{\pi_i}^j}_{\text{i-th pose}} & \cdots \end{bmatrix} \quad (6.32)$$

Feature position error Jacobian \mathbf{H}_{f_j}

The residual from Equation 6.23 also includes the observation Jacobian with respect to the feature position:

$$\mathbf{H}_{f_i}^j = \frac{\partial \mathbf{z}_i^j}{\partial \mathbf{p}^{\mathcal{GF}}} = \frac{\partial \mathbf{z}_i^j}{\partial \mathbf{p}^{\mathcal{CF}}} \cdot \frac{\partial \mathbf{p}^{\mathcal{CF}}}{\partial \delta \mathbf{p}^{\mathcal{GF}}} \quad (6.33)$$

where the chain rule has been used one more time to decompose the Jacobian into the product of two (simpler) partial derivatives. The first partial derivative corresponds to the feature projection and is the same as in the previous section (see Equation 6.25). The second term can be computed by rewriting Equation 6.20 and using the true state as in Equation 6.26. This time though, the true state applies to the feature position on the grounds that this time the error to be estimated is that of the feature position (and not the error state as in the previous case). This leads to:

$$\mathbf{p}^{\mathcal{CF}} = \mathbf{R}^{\mathcal{GC}\top} (\mathbf{p}_t^{\mathcal{GF}} - \mathbf{p}^{\mathcal{GC}}) \quad (6.34)$$

From this we can decompose the true state $\mathbf{p}_t^{\mathcal{GF}}$ into the nominal and error components:

$$\mathbf{p}^{\mathcal{CF}} = \mathbf{R}^{\mathcal{GC}\top} (\mathbf{p}^{\mathcal{GF}} + \delta\mathbf{p}^{\mathcal{GF}}) - \mathbf{R}^{\mathcal{GC}\top} \mathbf{p}^{\mathcal{GC}} \quad (6.35)$$

It is easy to see that the partial derivative is:

$$\frac{\partial \mathbf{p}^{\mathcal{CF}}}{\partial \delta\mathbf{p}^{\mathcal{GF}}} = \mathbf{R}^{\mathcal{GC}\top} \quad (6.36)$$

Finally, from 6.33 it follows that the Jacobian of the measurement with respect to the feature position error is described by:

$$\mathbf{H}_{f_i}^j = \mathbf{H}_{z_i}^j \mathbf{R}^{\mathcal{GC}\top} \quad (6.37)$$

Feature track Jacobian

The previous two sections derived the Jacobians of a single observation of a feature with respect to the error state and feature position error, respectively. We now need a way to define the Jacobian of a full track to effectively use the information of all observations of a feature. Since a track is a set of single measurements, we can stack all the measurement Jacobians to compute the track's Jacobian:

$$\mathbf{H}_{\delta\mathbf{x}}^j = \begin{bmatrix} 0_{2 \times 15} & \mathbf{H}_{\pi_0}^j & 0_{2 \times 6} & \cdots & 0_{2 \times 6} \\ 0_{2 \times 15} & 0_{2 \times 6} & \mathbf{H}_{\pi_1}^j & \cdots & 0_{2 \times 6} \\ & & \vdots & & \\ 0_{2 \times 15} & 0_{2 \times 6} & 0_{2 \times 6} & \cdots & \mathbf{H}_{\pi_M}^j \end{bmatrix} \quad (6.38)$$

The track feature position error Jacobian can be computed in a similar way:

$$\mathbf{H}_f^j = \begin{bmatrix} \mathbf{H}_{f_0}^j \\ \mathbf{H}_{f_1}^j \\ \vdots \\ \mathbf{H}_{f_M}^j \end{bmatrix} \quad (6.39)$$

To compute the track's residual we can rewrite Equation 6.23 with the new stacked set of Jacobians:

$$\mathbf{r}^j \simeq \mathbf{H}_{\delta\mathbf{x}}^j \delta\mathbf{x} + \mathbf{H}_f^j \delta\mathbf{p}_{f_j} + \mathbf{n} \quad (6.40)$$

where it is easy to see that the track residual \mathbf{r}^j is also constructed by stacking the single residuals:

$$\mathbf{r}^j = \begin{bmatrix} \mathbf{r}_0^j \\ \mathbf{r}_1^j \\ \vdots \\ \mathbf{r}_M^j \end{bmatrix} \quad (6.41)$$

Marginalization of the feature position error

In general, the Extended Kalman Filter computes the expected measurement using the observation function h :

$$\hat{\mathbf{z}} = h(\mathbf{x}) + \mathbf{n}$$

The residual is the difference between the expected and the observed measurement:

$$\mathbf{r} = \mathbf{z} - \hat{\mathbf{z}}$$

For the covariance to be updated with the measurement, the residual is linearized leading to the general form:

$$\mathbf{r} \simeq \mathbf{H}\delta\mathbf{x} + \mathbf{n} \quad (6.42)$$

where \mathbf{H} is the Jacobian of the observation function h with respect to the error state.

Note that this is different from our linearized residual from Equation 6.40, which contains the feature position error. The reason of this term is that the feature position is computed using the state information, even if it is not included in the state vector. Consequently, the errors in the computed feature position correlate with the errors in the state.

We need the residual equation to be in the form of Equation 6.42 to be applied for the filter update. We marginalize the feature position error by multiplying the left nullspace \mathbf{L}_{f_j} of \mathbf{H}_f^j and multiply it in both sides of Equation 6.40:¹

$$\begin{aligned} \mathbf{L}_{f_j}^\top(\mathbf{r}^j) &\simeq \mathbf{L}_{f_j}^\top(\mathbf{H}_{\delta\mathbf{x}}^j\delta\mathbf{x} + \mathbf{H}_f^j\delta\mathbf{p}_{f_j} + \mathbf{n}) \\ \mathbf{L}_{f_j}^\top\mathbf{r}^j &\simeq \mathbf{L}_{f_j}^\top\mathbf{H}_{\delta\mathbf{x}}^j\delta\mathbf{x} + \mathbf{L}_{f_j}^\top\mathbf{n} \end{aligned}$$

which leads to:

$$\mathbf{r}_L^j \simeq \mathbf{H}_L^j\delta\mathbf{x} + \mathbf{n}_L \quad (6.44)$$

¹The left nullspace of a matrix \mathbf{M} corresponds to the set of vectors \mathbf{x} so that $\mathbf{x}^\top\mathbf{M} = \mathbf{0}$, and it is the same as the (right) nullspace of \mathbf{M}^\top , so that $LN(\mathbf{M}) = N(\mathbf{M}^\top)$. Consequently, multiplying a matrix by its left nullspace transposed yields zero: $LN(\mathbf{M})^\top\mathbf{M} = \mathbf{0}$.

Algorithm 4 Measurement model

```

function PROCESS TRACK( $\mathbf{t}_j$ )
   $\hat{\mathbf{p}}_{f_j}^{\mathcal{G}} \leftarrow \text{TRIANGULATE FEATURE POSITION}(\mathbf{t}_j)$  ▷ Using Gauss-Newton
  for all observation  $\mathbf{z}_i^j$  in  $\mathbf{t}_j$  do
     $\hat{\mathbf{z}}_i^j \leftarrow \text{PROJECT TRACK}(\hat{\mathbf{p}}_{f_j}^{\mathcal{G}})$  ▷ See Equation 6.21
     $\mathbf{r}_i^j \leftarrow \mathbf{z}_i^j - \hat{\mathbf{z}}_i^j$ 

    ▷ Compute state and feature Jacobians for the measurement

    
$$\mathbf{H}_{z_i}^j \leftarrow \begin{bmatrix} \frac{f_x}{z_i^j} & 0 & -\frac{f_x \cdot x_i^j}{(z_i^j)^2} \\ 0 & \frac{f_y}{z_i^j} & -\frac{f_y \cdot y_i^j}{(z_i^j)^2} \end{bmatrix}$$

    
$$\mathbf{H}_{\pi_i}^j \leftarrow \begin{bmatrix} -\mathbf{H}_{z_i}^j \mathbf{R}^{\mathcal{GC}\top} & -\mathbf{H}_{z_i}^j \mathbf{R}^{\mathcal{GC}\top} [(\mathbf{p}^{\mathcal{GC}} - \mathbf{p}^{\mathcal{GF}})]_{\times} \end{bmatrix}$$

    
$$\mathbf{H}_{\delta \mathbf{x}_i}^j \leftarrow \begin{bmatrix} \mathbf{0}_{2 \times 15} & \mathbf{0}_{2 \times 6} & \cdots & \mathbf{H}_{\pi_i}^j & \cdots \end{bmatrix}$$

    
$$\mathbf{H}_{f_i}^j \leftarrow \mathbf{H}_{z_i}^j \mathbf{R}^{\mathcal{GC}\top}$$


    ▷ Stack measurement Jacobians
     $\mathbf{H}_{\delta \mathbf{x}}^j \leftarrow \text{APPEND}(\mathbf{H}_{\delta \mathbf{x}_i}^j)$ 
     $\mathbf{H}_f^j \leftarrow \text{APPEND}(\mathbf{H}_{f_i}^j)$ 
     $\mathbf{r}^j \leftarrow \text{APPEND}(\mathbf{r}_i^j)$ 
  end for

  ▷ Marginalize feature Jacobian
   $\mathbf{L}_{f_j} \leftarrow \text{LEFT NULLSPACE}(\mathbf{H}_f^j)$ 
   $\mathbf{H}_L^j \leftarrow \mathbf{L}_{f_j}^{\top} \mathbf{H}_{\delta \mathbf{x}}^j$ 
   $\mathbf{r}_L^j \leftarrow \mathbf{L}_{f_j}^{\top} \mathbf{r}^j$ 
  return  $\mathbf{H}_L^j, \mathbf{r}_L^j$ 
end function

```

This last equation can now be used to perform filter updates. The covariance of the noise vector \mathbf{n}_L is $\mathbf{R}_L = \sigma_{im}^2 \mathbf{I}$, where σ_{im} is the standard deviation of the image noise.

Algorithm 4 illustrates all the steps performed in the measurement model for one track.

6.6 Filter update

The filter update is the final step performed by the MSCKF algorithm. It uses the geometric constraints of the feature tracks to observe and correct the errors. Two situations will trigger the update step:

- *When a feature is no longer detected*, meaning that the track has finished.
 This occurs for instance when the feature moves out of view or it is occluded.

Algorithm 5 Update filter**function** UPDATE_FILTER **for all** track \mathbf{t}_j in FINISHED TRACKS **do** $\mathbf{H}_L^j, \mathbf{r}_L^j \leftarrow \text{PROCESS_TRACK}(\mathbf{t}_j)$ $\mathbf{H}_L \leftarrow \text{APPEND}(\mathbf{H}_L^j)$ $\mathbf{r}_L \leftarrow \text{APPEND}(\mathbf{r}_L^j)$ **end for** $\mathbf{Z} \leftarrow \mathbf{H}_L \mathbf{P} \mathbf{H}_L^\top + \mathbf{R}_L$ $\mathbf{K} \leftarrow \mathbf{P} \mathbf{H}_L^\top \mathbf{Z}^{-1}$ $\delta \mathbf{x} \leftarrow \mathbf{K} \mathbf{r}_L$ $\mathbf{x} \leftarrow \mathbf{x} \oplus \delta \mathbf{x}$ **end function**

1. 当一个feature丢失了, 会对滤波器进行更新

2.

- *When the state is full.* Every time a new image arrives, the sliding window is augmented with a copy of the current camera pose. Since the window has a fixed size N_{max} , we must get rid of the old window poses to make place for the newer ones. Every time this happens, we select a set of evenly spaced window poses and perform an update with their feature observations, before discarding them. The oldest pose is always left in the window since it is convenient to keep a large baseline for the feature triangulation.

No matter which event triggered the update, at this point we need to process a set of k feature tracks, each one with its observations. For each track f_j , we compute the residual \mathbf{r}_L^j and the observation Jacobian \mathbf{H}_L^j projecting the left nullspace of the feature position error Jacobian (see 6.44). Then we stack the residuals and Jacobians of all tracks:

$$\mathbf{H}_L = \begin{bmatrix} \mathbf{H}_L^0 \\ \mathbf{H}_L^1 \\ \vdots \\ \mathbf{H}_L^K \end{bmatrix} \quad \mathbf{r}_L = \begin{bmatrix} \mathbf{r}_L^0 \\ \mathbf{r}_L^1 \\ \vdots \\ \mathbf{r}_L^K \end{bmatrix} \quad (6.45)$$

which leads to the final residual equation:

$$\mathbf{r}_L = \mathbf{H}_L \delta \mathbf{x} + \mathbf{n}_L \quad (6.46)$$

With this we can apply now the usual equations for the Kalman Filter. First the residual covariance is computed:

$$\mathbf{Z} = \mathbf{H}_L \mathbf{P} \mathbf{H}_L^\top + \mathbf{R}_L \quad (6.47)$$

And then the Kalman gain:

$$\mathbf{K} = \mathbf{P} \mathbf{H}_L^\top \mathbf{Z}^{-1} \quad (6.48)$$

We can now use the Kalman gain to finally observe the error state:

$$\delta \mathbf{x} = \mathbf{K} \mathbf{r}_L \quad (6.49)$$

This error state has the usual form:

$$\delta \mathbf{x} = \left[\underbrace{\delta \mathbf{p}^{\mathcal{GI}\top} \quad \delta \mathbf{v}^{\mathcal{GI}\top} \quad \delta \boldsymbol{\theta}^{\mathcal{GI}\top} \quad \delta \mathbf{b}_a^\top \quad \delta \mathbf{b}_\omega^\top}_{\delta \mathbf{x}_{IMU}} \mid \underbrace{\delta \boldsymbol{\pi}_0^\top \quad \cdots \quad \delta \boldsymbol{\pi}_N^\top}_{\delta \mathbf{x}_\pi} \right]^\top$$

with the error window poses having an error position and orientation:

$$\delta \boldsymbol{\pi}_i = \left[\delta \mathbf{p}^{\mathcal{GC}_i\top} \quad \delta \boldsymbol{\theta}^{\mathcal{GC}_i\top} \right]^\top$$

The error state is then injected into the nominal state to correct for the accumulated errors:

$$\mathbf{x} \leftarrow \mathbf{x} \oplus \delta \mathbf{x}$$

where the operator \oplus means a component-wise composition including the sliding window poses:

$$\begin{aligned} \mathbf{p}^{\mathcal{GI}} &\leftarrow \mathbf{p}^{\mathcal{GI}} + \delta \mathbf{p}^{\mathcal{GI}} \\ \mathbf{v}^{\mathcal{GI}} &\leftarrow \mathbf{v}^{\mathcal{GI}} + \delta \mathbf{v}^{\mathcal{GI}} \\ \mathbf{q}^{\mathcal{GI}} &\leftarrow \text{vec2q}(\delta \boldsymbol{\theta}^{\mathcal{GI}}) \otimes \mathbf{q}^{\mathcal{GI}} \\ \mathbf{b}_a &\leftarrow \mathbf{b}_a + \delta \mathbf{b}_a \\ \mathbf{b}_\omega &\leftarrow \mathbf{b}_\omega + \delta \mathbf{b}_\omega \\ &\vdots \\ \mathbf{p}^{\mathcal{GC}_i} &\leftarrow \mathbf{p}^{\mathcal{GC}_i} + \delta \mathbf{p}^{\mathcal{GC}_i} \\ \mathbf{q}^{\mathcal{GC}_i} &\leftarrow \text{vec2q}(\delta \boldsymbol{\theta}^{\mathcal{GC}_i}) \otimes \mathbf{q}^{\mathcal{GC}_i} \end{aligned}$$

Finally, we need to update the covariance matrix to reflect the reduction in uncertainty due to the observations ²

$$\mathbf{P} \leftarrow (\mathbf{I} - \mathbf{K} \mathbf{H}_L) \mathbf{P} (\mathbf{I} - \mathbf{K} \mathbf{H}_L)^\top + \mathbf{K} \mathbf{R}_L \mathbf{K}^\top \quad (6.51)$$

Algorithm 5 shows a schema of the update process.

²The computationally expensive Joseph form is used here to numerically guarantee a positive and symmetric covariance matrix. Other cheaper forms exist.

7

Implementation

This chapter explains our implementation details of the MSCKF algorithm. The development of this project has been mainly done in C++. Some of the reasons for this decision are:

- Fast execution time. This is crucial for an algorithm that must run in real-time.
- Large number of efficient external tools and libraries.
- Possibility to port the code to other platforms (such as Android or iOS).
- Large community to get support.

One of the main drawbacks of the C++ implementation is the slower development time due to the complexity of the language in comparison to other popular options such as Matlab or Python. These languages have been used in this project for small prototypes and tests, though. For instance, a first version of the feature tracker was implemented in Python, as well as a plotting tool to check the algorithm's behaviour in real-time. A good option could have been implementing a prototype of the full algorithm in Python or Matlab to test it, and then port it to C++ for a real-time implementation. Unfortunately this could not be done due to time restrictions.

The following section specifies some of the external libraries and tools that have been used for the implementation. Then we explain the simulator that we have built to test the algorithm with ideal conditions and ground-truth knowledge. The custom data structures designed for the algorithm are then described. Finally, the class structure of the project is shown.

7.1 External tools

Eigen

“*Eigen* [5] is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.” It can store matrices of any size and type, it

supports both dense and sparse representations, and it includes most basic matrix operations. Eigen is also very efficient due to its smart use of template metaprogramming which allows lazy evaluations where needed. It also supports vectorization for certain architectures, making it very fast. Additionally, it also contains a Geometry module to deal with rotations and translations, including rotation matrices and quaternions.

The most relevant Eigen's data structure for our project is the `Matrix` class, which is used to represent real matrices and has several variants depending on its size, such as `Matrix3d` (3×3), `Matrix4d` (4×4) or the generic `MatrixXd` ($n \times n$). There also exist their `Vector` counterparts: `Vector3d`, `Vector4d`, `VectorXd`, and so on, which are nothing else than $n \times 1$ matrices.

The following example illustrates how to create a matrix and a vector and multiply them:

```
Matrix2d mat;
mat << 1, 2,
      3, 4;
Vector2d u(-1,1);
cout << "mat*mat:" << endl << mat*mat << endl;
// mat*mat:
// 7 10
// 15 22
cout << "mat*u:" << endl << mat*u << endl;
// mat*u:
// 1
// 1
```

Another important functionality when working with matrices are block operations. They are used to access and operate with matrix parts. Since we use Eigen to store our state vector and covariance matrix, this is really useful to access each state component individually. Block operations are used in Eigen with `matrix.block(i,j,p,q)`, where `i` and `j` are the block's starting indices, while `p` and `q` are the size in each dimension. This snippet shows the use of the block operations within Eigen:

```
MatrixXd m(4,4);
m << 1, 2, 3, 4,
     5, 6, 7, 8,
     9,10,11,12,
    13,14,15,16;
cout << "Middle block:" << endl << m.block(1,1,2,2) << endl;
// Middle block:
// 6 7
// 10 11
```

OpenCV

“*OpenCV* [18] (Open Source Computer Vision Library) is an open source computer vision and machine learning software library”. It contains many popular vision algorithms for several tasks, such as feature detection, tracking, 3D vision, image stitching, etc.

OpenCV stores images in a data structure called `Mat`. This container acts as a shared pointer that points to the allocated memory for the image and allows to share the image data efficiently. It is really easy to read an image from the file system and store it in a `Mat` object:

```
Mat img;
img = imread("lena.jpg", CV_LOAD_IMAGE_COLOR);
```

Although there are many algorithms in OpenCV, we only use a small subset of them. The first application of OpenCV in our project is the detection of features. It has support for many different detectors such as Harris [7], BRISK [12], FAST [20], SURF [3], etc. After trying some of them we decided to use the old Harris detector due its simplicity and good results (see Section 5.1).

In our project, we need to detect features and track them as long as possible. Every time the number of tracks decreases to a certain number, new features are detected and start to get tracked. We use the Harris detector to detect new features periodically. This is really straightforward in OpenCV using the method `goodFeaturesToTrack`:

```
vector<Point2f> corners;
double qualityLevel = 0.01;
double minDistance = 10;
int blockSize = 3;
bool useHarrisDetector = true;
double k = 0.04;

goodFeaturesToTrack(image,
                    corners,
                    maxCorners,
                    qualityLevel,
                    minDistance,
                    mask,
                    blockSize,
                    useHarrisDetector,
                    k);
```

This method stores the detected corners in the vector `corners`, using the given parameters. `qualityLevel` is used to reject corners with a bad measure of *cornerness*, using the metric in 5.1 and the parameter `k`. `maxCorners` defines the maximum number of corners detected. Since the detections are sorted by quality, only the best corners will be returned by the method. `minDistance` defines the minimum (Euclidean) distance between features. In our case, since we are already tracking some features, we want to avoid duplicates when triggering the detector. For this, we use a `mask` with black circles around the existing features, so that new corners will not be tracked there.

After some corners have been detected, we need to track them until they disappear. We use the Lucas-Kanade tracker for this, explained in Section 5.2. The method `calcOpticalFlowPyrLK` implements the Lucas-Kanade optical flow algorithm:

```
void calcOpticalFlowPyrLK(
    InputArray previousImg,
    InputArray nextImg,
    InputArray previousPoints,
```

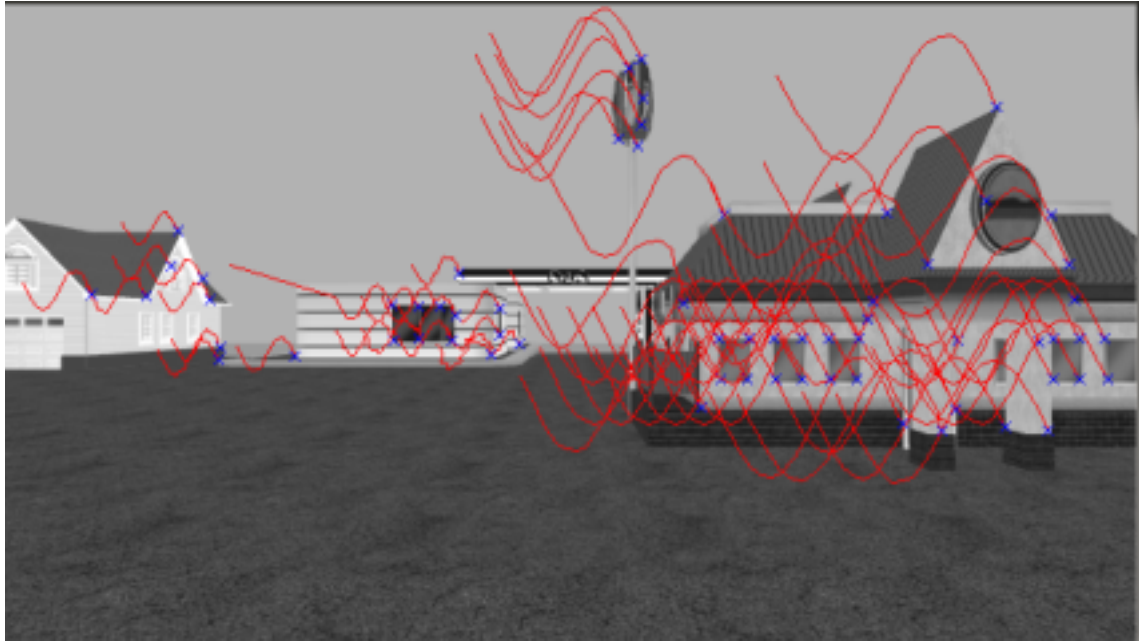



Figure 7.1: **Feature detection and tracking using OpenCV in a simulated environment.** Blue crosses show the tracked corners in the current frame. Red lines represent each track and follow the path created by all observations of the same feature. In this case, the robot follows an horizontal sinusoidal movement.

```

    InputOutputArray nextPoints,
    OutputArray status,
    OutputArray error)

```

This will be typically called every time a new image is received. `previousImg` and `nextImg` are the consecutive images for which the optical flow needs to be computed. `previousPoints` are the corner locations for the previous frame, while the new corners will be stored in `nextPoints`. These corner vectors will be swapped every frame. The `status` vector is used to indicate which features have been lost in the current frame. We use this flag to trigger the filter update with the finished tracks. The `error` vector contains a measure of the tracking error for each point. There are some other parameters but they are not so relevant in our case.

A double pass of the algorithm is performed to increase robustness. First, the points in the old frame are tracked in the new frame using `calcOpticalFlowPyrLK`. Second, the new points are re-matched against the points in the old image. The result of this must coincide with the points in the previous frame. For every corner, the distance between the original old detection and the recomputed old detection is measured. If this distance is not small enough, the detection is rejected and the track is finished. This helps to detect wrong correspondences that could ruin the filter update step.

Finally, a last action is performed to avoid drift in the tracking process. Since the Lucas-Kanade algorithm only uses the previous and current images every time, it is likely that small errors in every frame tend to accumulate, leading to large errors after some time. We solve this problem by storing a small patch around a corner the first time it is detected. On every frame, we perform a normalized cross

correlation test between the original patch and a new patch centered at the new point. The point with the highest *score* is selected as the new match. OpenCV’s function `matchTemplate` is used to compute the normalized cross correlation:

```
void matchTemplate(
    InputArray image,
    InputArray patch,
    OutputArray result,
    int method)
```

Here `image` is a patch of the new image, centered at the corner. `patch` is the original patch of the corner, extracted when it was initially detected. `result` contains the *score* for each pixel. `method` is the method used to compute the score, in our case this is the normalized cross-correlation (NCC).

To obtain subpixel accuracy when matching a patch, parabolic interpolation is carried out, using the pixels that surround the best match, which in our case corresponds to the location with the highest NCC score.

Figure 7.1 shows the result of the feature detection and tracking using OpenCV on a simulated environment.

Ceres Solver

“Ceres Solver [1] is an open source C++ library for modeling and solving large, complicated optimization problems. It is a feature rich, mature and performant library which has been used in production at Google since 2010”.

Ceres is used to solve non-linear least squares problems in the general form

$$\min_{\mathbf{x}} \quad \frac{1}{2} \sum_i \rho_i \left(\|f_i(x_{i_1}, \dots, x_{i_k})\|^2 \right)$$

where f_i is the cost function that must be minimized, and ρ_i is a loss function, used to reduce the influence of outliers when performing the minimization.

We use Ceres to perform bundle adjustment to triangulate the position of a feature that has been observed in multiple poses (see Section 5.4).

The way Ceres specifies a cost function is by implementing a templated functor (i.e. a class that overrides the `()` operator) that computes the residual (cost) with the given parameters. A Ceres `Problem` is then constructed, where we add one residual block for each measurement. In order to solve the `Problem`, the optimization engine will evaluate every residual block using our cost function, and will try to find a minimum by computing the Jacobians using automatic differentiation. A (simplified) example of our cost function is shown below. It computes the reprojection error for each feature observation (see Section 6.5):

```

struct VIOReprojectionError {

    template<typename T> bool operator()(
        const T* const camera_rotation_G,
        const T* const camera_translation_G,
        const T* const inverseDepthPoint,
        T* residual) const {

        // compute residual and store it in <residual>:
        // z_predicted = project_point(camera_rotation_G,
        //                               camera_translation_G,
        //                               inverseDepthPoint);
        // residual = z_predicted - z_observed;

        return true;
    }
};

```

The actual code is a bit longer, only the main idea is shown here in the comments. This functor is templated (see the type `T`) to automatically evaluate the Jacobians, although this is transparent to the user. The type `T` is used as a scalar but it is also used internally to propagate the Jacobians using the chain rule.

Figure 7.2 shows the result of the feature optimization process. The yellow points show the optimized feature position. As it can be seen in the picture, the points resemble the building in front of the camera. Errors seem to grow with the distance due to a smaller baseline.

ROS

“The Robot Operating System (ROS) [19] is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.”

ROS offers a distributed system for robot communication by using *nodes*. A node is a process that performs a certain function, such as building a map of the environment or interfacing with a sensor. Different nodes can communicate with each other by means of *messages*, which are representations of data that can be sent through the network. Messages are published in a *topic*, which basically represents a channel to which messages are sent. Other nodes can subscribe to a topic to receive the messages published on them. In this way, complex systems can be constructed in a decentralized fashion, simplifying implementations and making possible the collaboration between many people working on different problems.

Our VIO algorithm runs within a ROS node. The node receives data and passes it to our algorithm. Data may be obtained from different sources, such as a simulator, real sensors or a previously stored dataset. Our data includes IMU measurements, camera images, and ground truth poses.

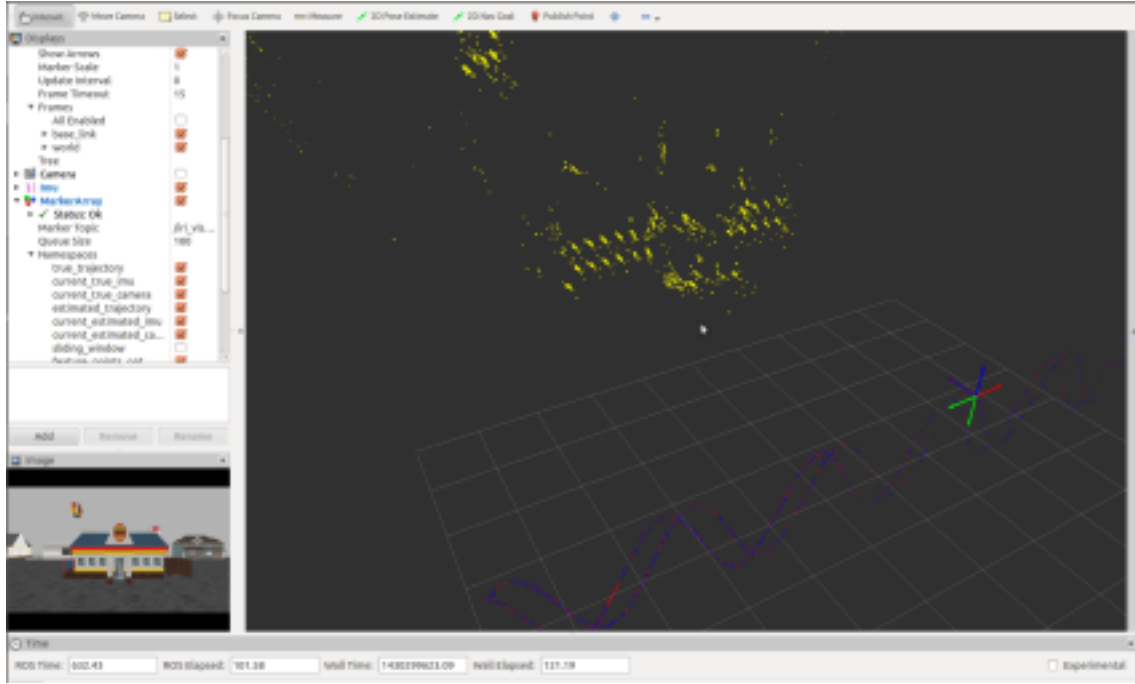


Figure 7.2: **Visualization of our MSCKF implementation.** Triangulated features are shown as yellow points. The current IMU and camera coordinates are drawn with colored arrows. The trajectory is drawn using lines. The current image from the camera can be seen at the bottom left: some buildings can be seen in front of the camera.

Our algorithm then communicates with the ROS node in order to publish the estimated odometry. Other nodes can connect to the topic to which we are publishing to use the odometry data.

One important ROS tool is *rviz*, which is a 3D visualization tool. It allows to draw figures such as cubes, spheres, arrows or generic 3D meshes and visualize them in a user interface. This has been really useful to debug our algorithm. We draw all coordinate axes present in the state vector (IMU, camera, and sliding window poses), together with an ellipsoid that represents their covariances. We also draw the feature points when they are triangulated, and the accumulated trajectory of the robot. Figure 7.2 shows the visualization of our algorithm using *rviz*.

Gazebo

“Gazebo [6] offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments”. Gazebo is a realistic robotics simulator: it allows to model robots, simulate their physical behaviour and render it in 3D and publish their sensor data. It integrates itself really easily in the ROS ecosystem and it is really popular among people working in robotics.

Gazebo works with several components to simulate environments. First, robots are defined by means of *model files*. Although there are several options, in our case this consists in a *URDF* file, which is an XML file that defines the hierarchy of the

robot's components (joints, links and sensors).

A robot alone is not realistic, we need a way for it to interact with its external environment. Gazebo provides *world files* for this purpose. These are XML files that represent an external world. One can place buildings, rooms, or put several robots that interact with each other. Sensors will also react to those elements, for instance a camera will record the surrounding buildings, or a laser will detect walls when placed inside a room.

Another important feature in Gazebo is the possibility to create plugins that perform custom simulations. A plugin is a piece of code that alters a simulation, for example by moving a robot, adding new models to the simulation, or publishing the data of a custom sensor. A plugin is compiled as a shared library and then can be inserted in a simulation when desired.

Finally, it is worth to mention the collection of sensors available in Gazebo: cameras, lasers or IMUs are a few of them. It is possible to control the noise in each sensor (or even remove it), which was really useful to debug our algorithm.

7.2 Simulator

In order to test the algorithm, a way to control the conditions of the experiments is needed. We need a way to control the IMU noise, or even to remove it completely. This is not possible in the real world, since no sensors are perfect. We also want to control the simulation parameters perfectly. For instance, calibrating the displacement between the IMU and the camera with accuracy can be difficult in a real robot.

Using a simulator helps to avoid these problems, which is really useful at the beginning of the algorithm's implementation. We use a Gazebo plugin to implement a simulation suitable for us. Our simulator consists in a flying robot, simply represented with a box, with an IMU and a camera attached. The default movement we give to the robot is a sinusoidal trajectory along its y axis, although any trajectory could be implemented if needed. We place some buildings in front of the robot, which will be recorded by the camera and used for feature tracking in the algorithm. The robot has a periodic movement side-wards and observes the buildings from different spots. Figure 7.3 shows the simulated world with our robot moving in it.

Several elements were needed to implement our simulator. First, we need to define the model for the robot. This is done by means of an URDF file that specifies all the components in XML format. We define three links: the base link and two additional links for the IMU and the camera. This allows us to specify the coordinate transformation between all components. We also add a *visual* to draw a box in the robot's location. A visual is just an XML tag that defines how a robot component is rendered. It is possible to use arbitrarily complex meshes, but a simple box is enough for our purposes. The visual helps us to locate the robot in the simulation interface. We finally add the two sensors needed for our algorithm: an IMU and a monocular camera. We can control the parameters of the sensors to match our

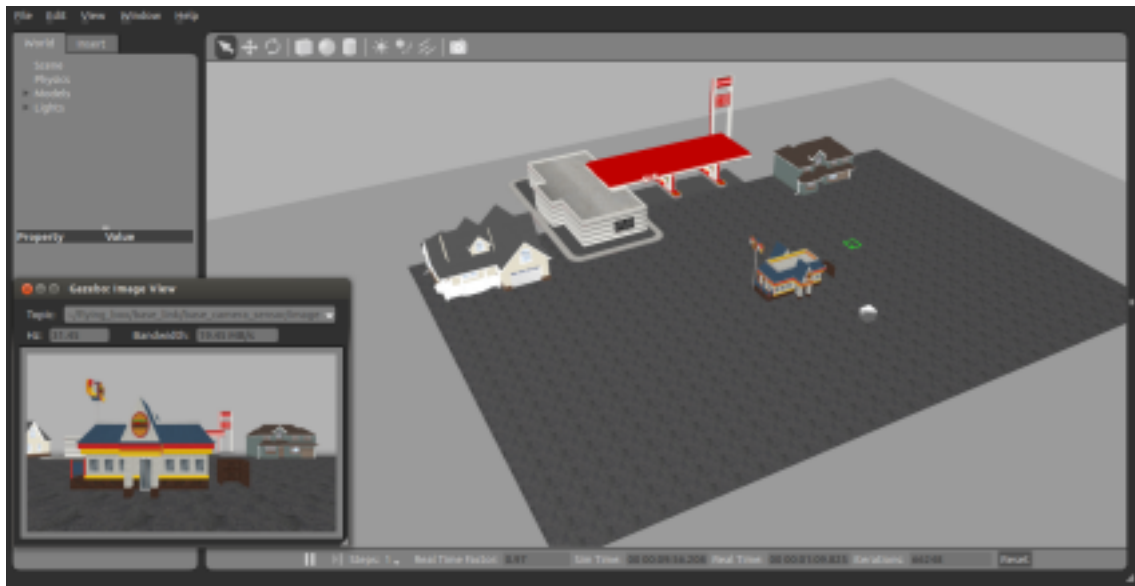


Figure 7.3: **Flying box simulator.** The white cube represents our robot, which is moving side-wards in a sinusoidal trajectory. The picture on the left shows the images taken by the robot’s camera. We placed some buildings in front of the robot, which can be seen in the images too.

needs (e.g. the noise levels or the camera intrinsic parameters).

Our simulated robot has a model now, but it can’t do anything yet. If we add it to a simulation, it will just fall to the floor and lay there. We need to implement a Gazebo plugin to define the behaviour of our robot. A plugin is just a C++ class that inherits from Gazebo’s `ModelPlugin` class. This allows us to access all the simulation states, such as the robot’s pose and velocity. Our class overrides the `onUpdate` callback, which is called at every simulation’s cycle. Inside that callback, we first add an upwards force to the robot to compensate for gravity. Otherwise the robot will fall to the floor due to the influence of gravity. We then apply a sinusoidal velocity to the base link. This will cause the robot to move in the YZ plane, following a sinusoidal trajectory. Finally, the last task our plugin does is publishing its own state. This can be used outside to obtain the ground truth of the robot’s state.

After the robot’s model and behaviour is defined, we need to create a world in which the robot can move. Using the Gazebo UI, we place some buildings in front of the robot, and put some asphalt tiles in the floor. These models come by default in Gazebo, so we take them for simplicity, although it is possible to import custom 3d models. After our world has some elements in it, we save it into a *.world* file to be used in the simulations later.

Finally, we need to wrap our simulator inside a ROS node. This will allow other nodes to use it. We create a *launch* file to start our simulator from outside. A launch file is an XML file interpreted by ROS that groups all the tasks needed to launch a node. Other nodes will use our launch file to start our simulator, overriding the needed parameters (such as the noise values or the camera intrinsic parameters).

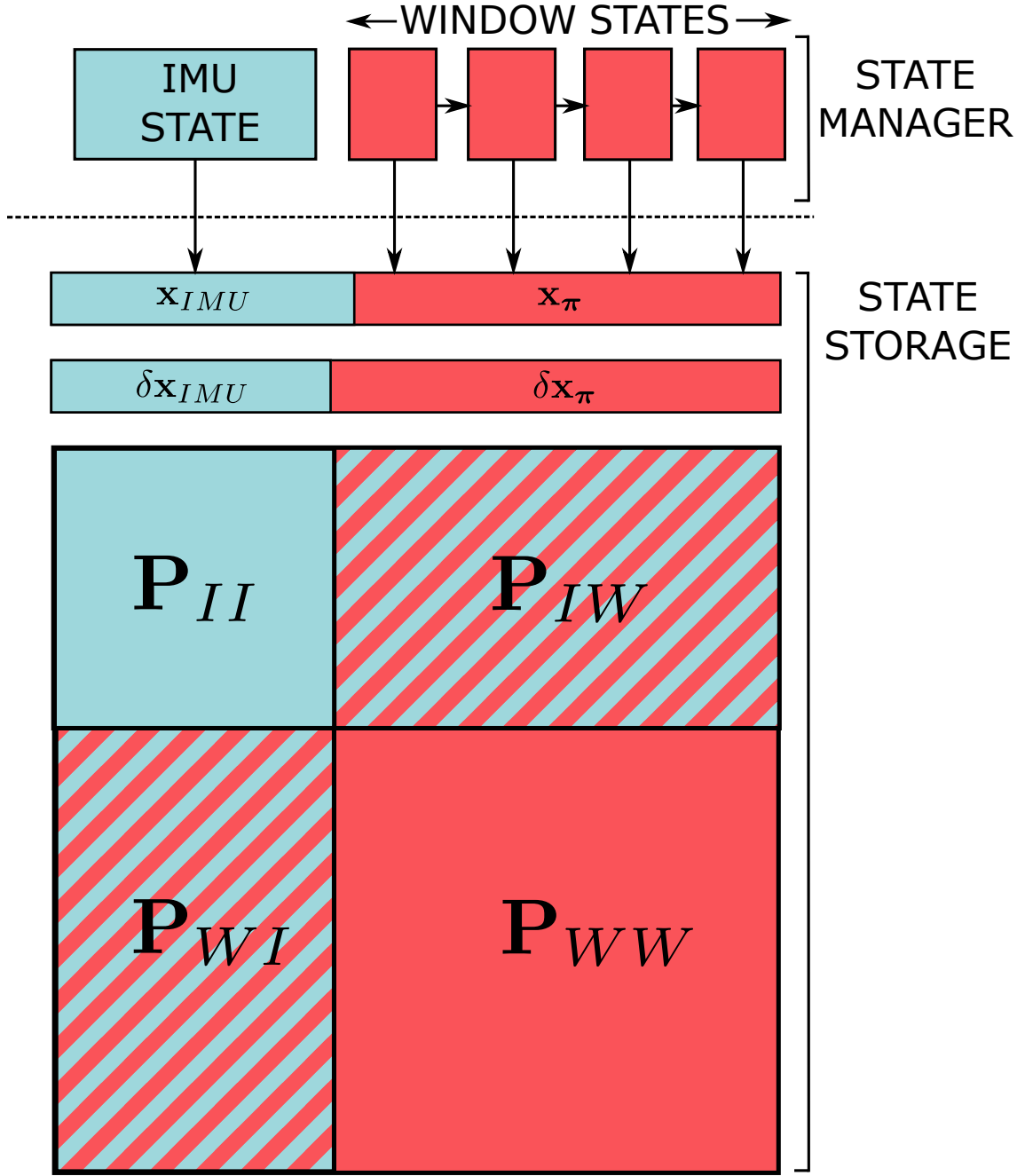


Figure 7.4: **Separation between the State Storage and the State Manager.** The manager has access to the storage and allows to read and write its blocks individually. It also manages the sliding window by adding new poses or removing the old ones when needed.

7.3 Data Structures

We need abstractions to properly work with the data we work with. Such abstractions help us to model the entities that form the problem we try to solve. By isolating the responsibility of every entity in our problem, we improve code reuse

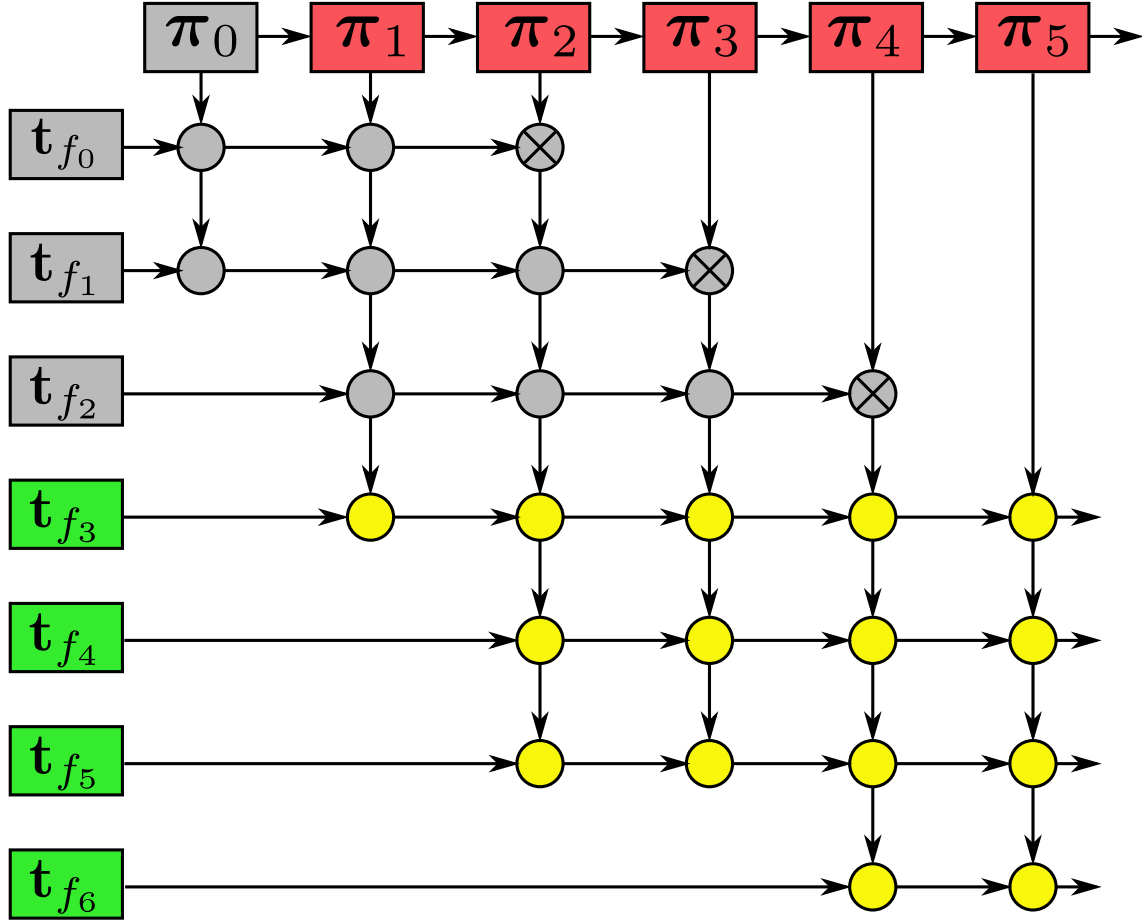


Figure 7.5: **Feature track management.** Tracks t_{f_j} (in green) contain a set of feature observations z_i^j (in yellow). Window state poses π_i (in red) also store the observations for that frame. A track is finished (in gray) when its feature is not observed anymore. A window pose is removed when it has no associated observations (π_0 in this example).

and testability.

The first difficulty we find in the implementation of our algorithm is how to properly define the state vector. In practice, the state is nothing else than a vector, and ultimately this is how it will be represented. However, the state is composed of several blocks, each one having a different meaning and associated behaviour. We need an abstraction to operate with such blocks, independently of how they are stored.

In the end, we decide to base our design in the separation of responsibility between the state storage and the state management. This leads to the creation of two main entities. The first of them will be the `StateStorage`. It just allocates the memory for the state vector and the covariance matrix, and provides access to them. The second entity is the `StateManager`. It provides an abstraction to properly access each block in the `StateStorage`. It makes possible to access the components from the IMU state and the sliding window poses, and also deals with the state when the sliding window is full, or when a pose needs to be removed. The `StateManager` contains several sub-entities: one to access the current IMU state, and a collection

of elements representing the poses in the sliding window. Figure 7.4 illustrates the separation between the `StateStorage` and the `StateManager`.

Another issue that arises in our algorithm has to do with the feature track management. A track is a set of observations from different poses. We need a way to associate the track observations with their respective window states that correspond to the camera poses. We add several entities to solve this problem. The first one is the `Track`, which contains a set of `Feature Observations`. Additionally, each sliding window pose also holds the list of the feature observations that have been observed in that frame. Figure 7.5 shows an example of the feature tracks and the sliding window.

7.4 Class Structure

In the object oriented paradigm, the data structures presented in the previous section are ultimately implemented using classes. Figure 7.6 shows the UML diagram for our algorithm's implementation.

The main class in our algorithm is the `VIOAlgorithm`. It receives the IMU measurements and the images, and is responsible to propagate the state and update the filter. It also publishes the estimated state and sends drawing instructions to the renderer by means of the `VIODataPublisher` interface from which inherits.

`VioAlgorithm` uses two classes to do its work: on one hand the `StateStorage` just allocates the state vector in memory; on the other hand the `StateManager` is used as an abstraction to access the state. It adds or removes tracks and window state poses when needed.

Two more classes form the main state parts. First, `IMUState` models the current IMU state \mathbf{x}_{IMU} . It is used to read and write the IMU state components such as the current position, orientation, etc. The other class is the `WindowPoseState`, which is stored as a collection in the `StateManager`. Every instance of this class represents a pose π_i in the sliding window. It is indexed by `id` and also allows to access the window state components such as position and orientation.

The `FeatureTrackManager` acts as a feature tracker. It detects new features when needed and tracks the existing ones, notifying `StateManager` when some tracks finish. Internally, it uses the Harris detector and the Lucas-Kanade tracker, although it could be easily extended to an interface to implement any type of detector and tracker.

The tracker also creates a `FeatureObservation` every time an existing track is detected in a new frame, and adds it to its `FeatureTrack`, which keeps a collection of all its observations. Observations are also added to their `WindowPoseState`.

When the tracker detects a new feature, it creates an `ImagePatch` centered in the feature coordinates. It will be used on every frame to correct the tracker and avoid drift.

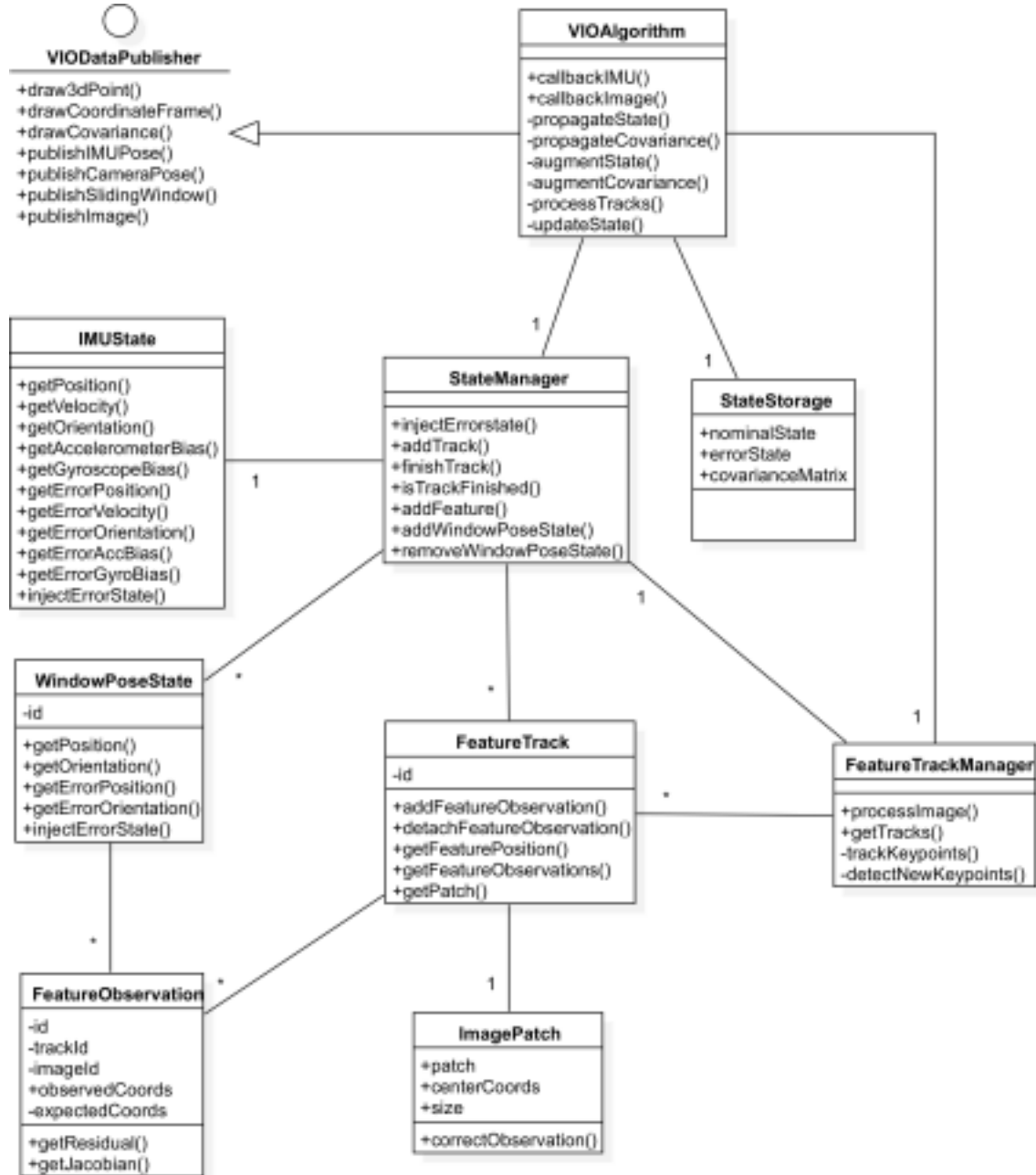


Figure 7.6: UML diagram for our MSCKF implementation. Only the most relevant classes, operations and attributes are presented.

8

Results

This chapter explains the (partial) results that have been obtained so far. At the time of this writing, the update step of the filter doesn't work properly and suddenly applies a wrong correction to the state after some time. See Chapter 10 for more details on the possible causes of this problem. In any case, all the other steps work correctly and its results are illustrated here.

8.1 IMU propagation

The first test we present is the state propagation, without performing the update step. This implies that the accumulated IMU error won't be corrected, so that the state will drift with time. Errors will mainly come from two sources: on one hand the sensor noise; on the other the approximations made in the propagation step (i.e. using Euler integration instead of more sophisticated integration forms).

Since we can control the sensor noise in the simulator, we first perform a test with no noise. The simulated robot performs a sinusoidal movement on the ZY plane. Figure 8.1 shows the result of this. As can be seen in the upper row, the sensor readings are *ideal* and present no noise. The acceleration in the z axis shows the sinusoidal effect of the movement. The peaks in the y axis correspond to a brake to change direction smoothly. As it can be seen in the figure, no angular velocity is applied to the robot. The state propagation is presented in the lower row of Figure 8.1. Since no noise is applied, the ground truth pretty much coincides with the estimated magnitudes. Since no update is done, the uncertainty (represented by the $\pm 3\sigma$ region in red) grows constantly over time. The smooth change of direction is reflected in a gradual change in velocity and position. The z position drifts slowly due to the Euler integration errors, even if no noise is applied.

We repeat the same experiment, this time adding noise to the sensor. We apply a white, Gaussian noise of 0.02 m/s^2 to the accelerometer readings (i.e. one thousandth of 2 g, which is a reasonable value for an accelerometer range) and 0.00628 rad/s to the gyroscope (one thousandth of 2π). The results are presented in Figure 8.2. The noise in the IMU readings can be clearly seen. The first thing to note is that, although no angular velocity is applied to the robot, the noise is integrated and produces wrong orientation values. Since orientation is needed for gravity cor-

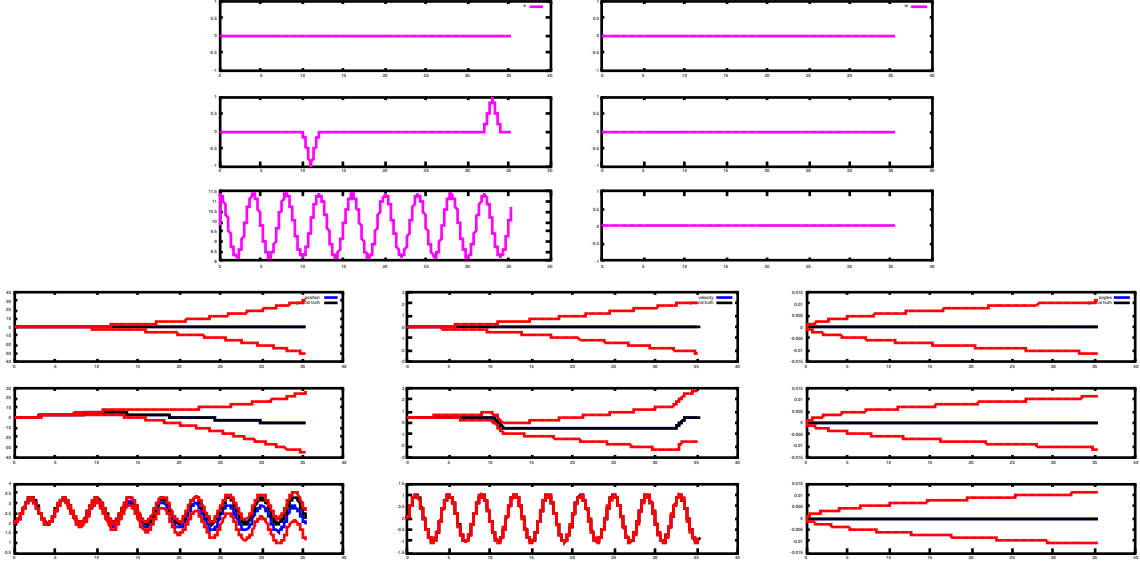


Figure 8.1: **Sensor readings and state propagation over time without noise.** Above: accelerometer (left) and gyroscope readings (right). Below, state propagation with position (left), velocity (middle) and orientation (right). Blue lines represent the magnitude computed by the algorithm, red lines are the $\pm 3\sigma$ region that represent the propagated uncertainty, and black lines are the simulator's ground truth. In all cases, x , y and z axes are presented from top to bottom, and the horizontal axes represent the time in seconds.

rection, orientation errors also contribute to the velocity and position errors. As can be seen in the figure, all estimated values slowly drift from the ground truth. Since the selected error values are relatively small, the drift is not that large.

8.2 Feature tracks

As the robot moves, its camera records images. Features are detected and tracked in every image. When a feature is no longer visible (or when the sliding window is full), it is triangulated using all the observations and camera poses from which it has been observed.

The result can be seen in Figure 8.3. The small picture on the bottom-left corner show the current tracks: the sinusoidal movement of the robot can be perceived in the shape of the tracks. Most detected features correspond to good corners, as can be seen for instance in the front building's windows.

The main picture shows the triangulated features in yellow. Feature positions have been computed using Gauss-Newton optimization, with the initial solution to the optimizer provided by means of the mid-point intersection method. The final result is quite good: the yellow points in the figure clearly resemble the front building. Farther buildings are not defined as clearly, partly because less features are detected in them due to the higher distance to the camera, and as a result of the triangulation errors that result from having a shorter baseline with respect to the feature's depth.

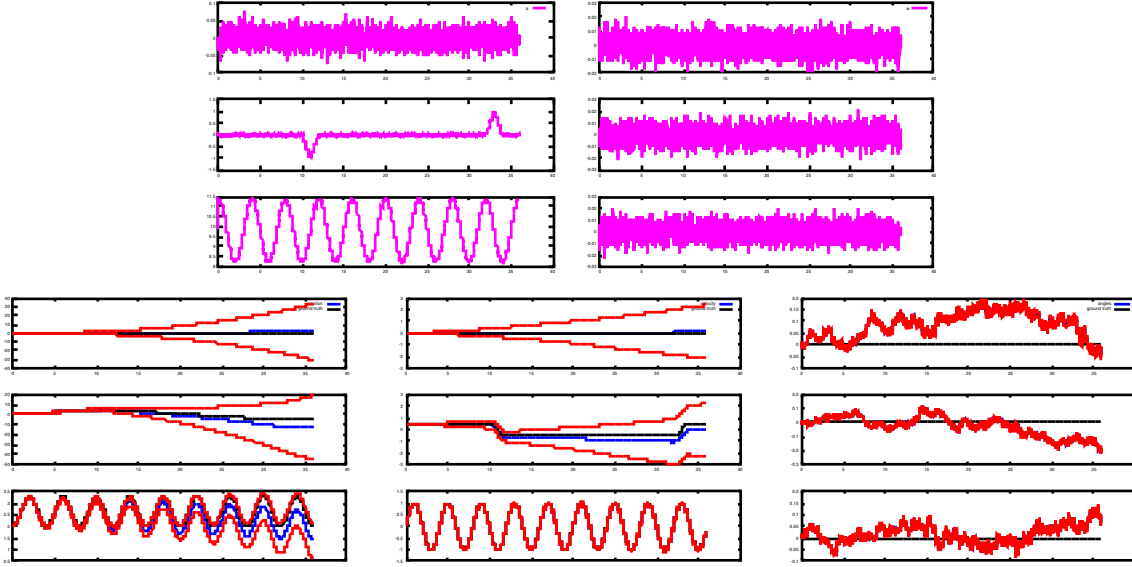


Figure 8.2: **Sensor readings and state propagation over time with noise.** Orientation errors and noise cause position and velocity errors to appear due to the gravity correction.

It is worth noting that the yellow points in Figure 8.3 are not a map in the SLAM sense. When a feature is triangulated, its position is rendered in the visualizer, and it is never removed from there. This implies that features are accumulated in the visualizer as time passes, although they are not stored anywhere after being processed.

Finally, since the objective of this experiment is to validate the feature triangulation process, no noise has been added to the IMU readings (as in the first experiment), so the camera positions used to triangulate features are pretty close to the ground truth.

8.3 Filter update

The steps in the two previous sections are combined for the filter update. First, the state and covariance are propagated using (noisy) IMU readings. Then, the observation model uses the feature positions to correct for state errors. Figure 8.4 shows the results of update step.

As it can be seen in the Figure 8.4, the drift is periodically corrected and it does not affect the estimate in the long term.

Additionally, the covariance is bound within a certain reasonable limit. It grows when the IMU measurements are propagated and it is reduced when an image is processed and the update step is performed to correct the estimate. This is the reason of the *jigsaw* shape of the covariance in Figure 8.4, specially in the x axis.

Unfortunately, after some time one update step suddenly applies a huge correction to the state. This makes the estimate to get lost without being able to recover. We

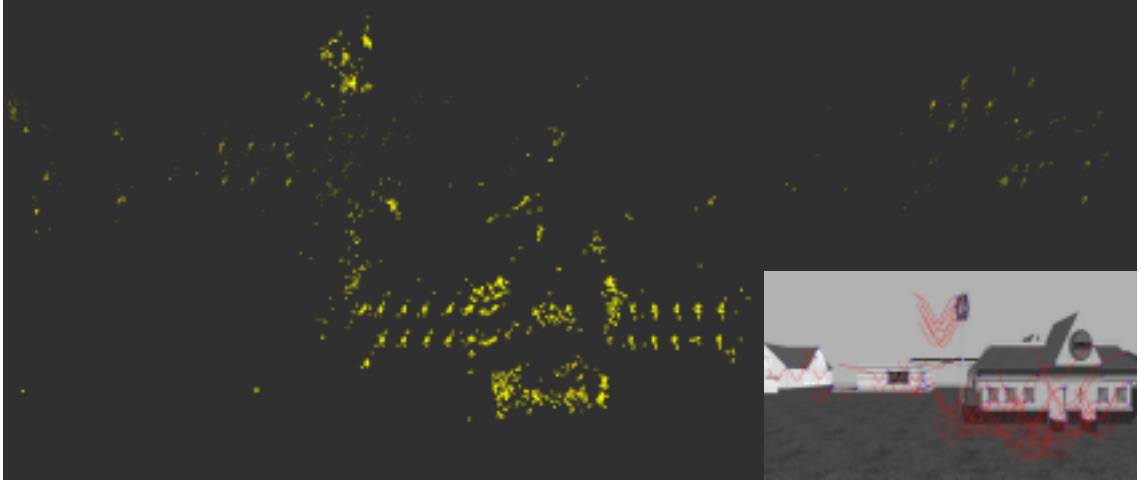


Figure 8.3: **Feature tracking and triangulation.** Bottom-right picture shows a camera image with feature tracks. Blue crosses mark the position of a feature in the current image. Red lines represent the tracks for each feature (i.e. the position of the feature in the previous images). The main image shows the triangulation of these features in yellow. The shape of the near building can be clearly seen in the picture.

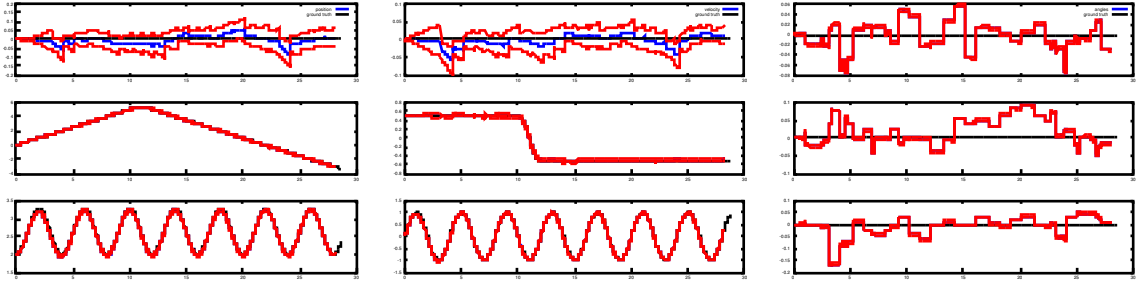


Figure 8.4: **Filter update with noise.**

could not find the reason of this problem yet.

9

Project Management

9.1 Planning

This project has three main parts: Research, Implementation and Documentation. The Research phase deals mainly with information gathering, the Implementation includes everything that has to do with code development, and the Documentation includes the writing of this document. Figure 9.1 shows a top-level overview of the project planning. It's important to note the long duration of the Implementation stage (330 days). The motive for this is a one year gap where the project was almost paused for work reasons.

Name	Begin date	End date	Duration
• Research	14/10/2013	13/12/2013	45
• Implementation	16/12/2013	20/3/2015	330
• Documentation	23/3/2015	6/5/2015	33

Figure 9.1: **Project planning overview**

The research phase includes several tasks and can be seen in detail in Figure 9.2. The first task consisted in reading articles about modern feature detectors and descriptors, such as BRISK, SURF, FAST, etc. At the same time while doing this, a review of the relevant mathematics had to be done. This mainly implied reading about linear algebra and probability theory and watching Khan Academy [11] lessons about these subjects. This helped with the next task, which consisted in reviewing works about Visual Inertial Odometry and SLAM, such as [23]. After this, A thorough reading of books and articles on Inertial systems and Kalman filtering was done [10]. Besides all the theory to be reviewed, some practical research was also done to choose the tools needed for the implementation phase. An evaluation of feature detectors and descriptors was done using OpenCV. Tutorials about ROS and Eigen were read, and some simple tests were implemented to learn how they worked. Although in practice most of these research tasks had to be constantly done during the whole project, the basis was laid down to start with the implementation stage.

The implementation stage is the most complex one, as it can be seen in Figure 9.3. The first task comprised the design and implementation of the algorithm's

Name	Begin date	End date	Duration
• Research	14/10/2013	24/1/2014	75
• Feature detectors and descriptors	14/10/2013	25/10/2013	10
• Review of linear algebra and probability	14/10/2013	29/11/2013	35
• State of the art	2/12/2013	27/12/2013	20
• Inertial Systems	30/12/2013	10/1/2014	10
• Kalman Filters and sensor fusion	13/1/2014	24/1/2014	10
• C++ Libraries and Frameworks (ROS, Ope...	2/12/2013	24/1/2014	40
• Implementation	27/1/2014	24/3/2015	302
• Documentation	25/3/2015	4/5/2015	29

Figure 9.2: **Research stage**

data structures, although in practice this required several iterations to do it right. After this was done, the first attempts to integrate the IMU measurements were performed. This was tough before using a simulator, since it was difficult to evaluate the integration due to the noise. After this, the augment step was implemented. The state part was easy with the correct data structures previously implemented. The augment of the covariance took some more time. An evaluation of the feature detectors and descriptors was done, this time adapting it to the custom data structures. At the end this didn't work properly, so the alternative was to implement a tracker using the Lucas-Kanade method. Due to the long-term drift of the features, the patch correction had to be implemented afterwards. After the tracker worked properly, the triangulation process was carried out. First the computation of the initial solution was done, and then used as initial estimate for the Gauss-Newton optimization process. In the meantime, visualization utilities were also implemented when needed for each task. For instance, the drawing of the coordinate systems used or the triangulated features. This was the only possible way to debug problems and make sure that everything was working right.

Up to this point, a quadrotor simulator was used to test the implementation. Unfortunately, this simulator gave some problems due to the vibration of the body and due to the fact that the sensor noise could not be properly controlled. The solution to this was to implement our custom simulator, much simpler than the quadrotor one. Once this was finished, the last part of the algorithm could be implemented. This included the observation model (including the Jacobians) and the filter update.

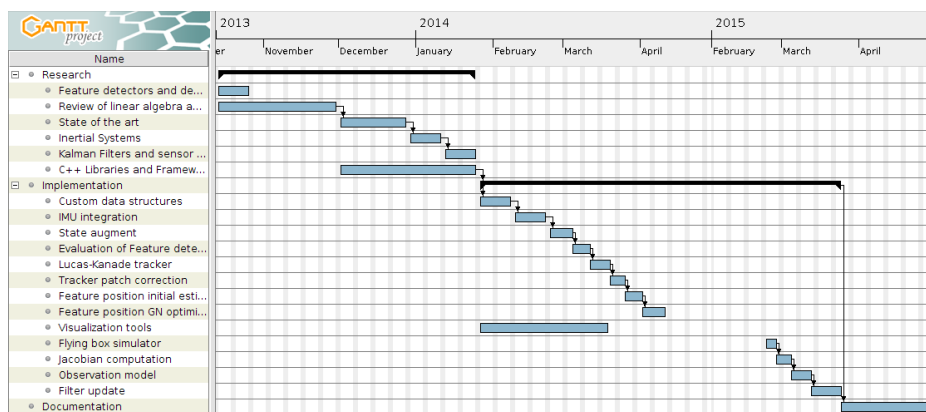
Finally, the last stage is the Documentation phase. This consisted in writing this document, and it took quite long (29 days). Figure 9.4 shows the total project's schedule.

9.2 Costs

This section calculates the economic costs of the project, which basically boil down to human resources and hardware resources. No software costs were incurred since all the tools and libraries are Open Source.

Name	Begin date	End date	Duration
• Research	14/10/2013	24/1/2014	75
• Implementation	27/1/2014	24/3/2015	302
• Custom data structures	27/1/2014	7/2/2014	10
• IMU integration	10/2/2014	21/2/2014	10
• State augment	24/2/2014	4/3/2014	7
• Evaluation of Feature detectors and descriptors	5/3/2014	11/3/2014	5
• Lucas-Kanade tracker	12/3/2014	19/3/2014	6
• Tracker patch correction	20/3/2014	25/3/2014	4
• Feature position initial estimation	26/3/2014	1/4/2014	5
• Feature position GN optimization	2/4/2014	10/4/2014	7
• Visualization tools	27/1/2014	18/3/2014	37
• Flying box simulator	23/2/2015	26/2/2015	4
• Jacobian computation	27/2/2015	4/3/2015	4
• Observation model	5/3/2015	12/3/2015	6
• Filter update	13/3/2015	24/3/2015	8
• Documentation	25/3/2015	4/5/2015	29

Figure 9.3: Implementation stage

Figure 9.4: **Complete planning.** Note that the one-year gap in the middle have been shortened to fit the picture in the page.

Human resources

We assume 8 hours of work per day in the estimation. Since different tasks need different skills and effort, we apply a different cost per hour in each stage. Table 9.1 shows the breakdown of all human resources costs:

Hardware resources

Two computers were used for this project: a personal laptop and a desktop computer from IRI:

Stage	Days	Hours	Price	Cost
Research	75	600	15 €	9000 €
Implementation	113	904	20 €	18080 €
Documentation	29	232	12 €	2784 €
Total	217	1736		29864 €

Table 9.1: **Human resource costs breakdown.**

- Packard-Bell laptop with Pentium processor P6100 dual core at 2GHz with 4Gb RAM memory (≈ 400 €).
- Dell desktop computer with Intel Xeon E31225 quad core processor at 3.10 GHz with 8Gb RAM memory (≈ 500 €).

Since both computers are not only used for this project, we need to take into account the hardware amortization. Assuming a three years life for a computer:

$$\text{Hardware cost} = \frac{400\text{€} + 500\text{€}}{3 \text{ years} \cdot 12 \text{ months/year} \cdot 20 \text{ days/month} \cdot 8 \text{ hours/day}} \simeq 0.15\text{€}$$

Multiplying by the total number of hours we get the total hardware cost:

$$\text{Total hardware cost} = 1736 \text{ hours} \cdot 0.15\text{€/hour} = 260.4\text{€}$$

Finally, the total cost of the project can be summarised in Table 9.2.

Stage	Cost
Human resources	29864
Hardware resources	260.4
Total	30124.4

Table 9.2: **Total project cost.**

10

Discussion

Unfortunately, at the time of this writing, the implementation of the algorithm could not be finished completely yet. Although the code development is finished, something seems to be wrong with the Update step. The filter works correctly for some time, but then the covariance suddenly starts to grow really fast and the estimate gets lost and cannot recover any more.

It is difficult to find the reason for this problem, although there are likely to be several causes. For instance, all Jacobians have been computed analytically, which may have caused careless mistakes to appear, even though the computations were thoroughly verified several times. Just a wrong symbol in a Jacobian could ruin the filter behaviour.

Another possible reason is a wrong tuning of the noise covariances. The covariance might be over- or underestimated, leading to wrong filter corrections.

The presence of outliers could certainly cause issues too, although it's unlikely since they are filtered using the Mahalanobis distance of a track. The tracks that are considered outliers are not used for the filter update. Nevertheless, it could be that the probability threshold to accept (or reject) is not suitable and accepts tracks that should be outliers.

Ultimately, the main problem is the difficulty to implement methods to debug and find computation errors. Such bugs come in subtle, delicate ways and are difficult to visualize. How do we know that the values in the covariance matrix are correct? We currently draw the ellipsoid for the position component of the state, but it is really difficult to visualize ellipsoids for the orientation or bias components. How do we know that the Jacobians are correct? Of course, common sense must be used to discard absurd values, but the process to find such errors can be really time-consuming. This is worsened by the decision to directly start with a C++ implementation, which makes the process of testing and debugging much slower. A python or matlab prototype would have been really valuable to make everything work. Then it could have been easily ported to C++.



Figure 10.1: **Rolling-shutter effect.** Different pixel rows of this image have been taken at different time instants. This produces a distortion due to the fast speed of the train.

Potential improvements

Besides fixing the update step, there are some improvements that could be made. Some of them are mandatory for the correct operation of the filter in the real world (i.e. outside the simulator), while others are optional but will improve the algorithm's accuracy:

- *Add the camera-IMU calibration parameters to the state vector* [14]. The translation and orientation of the camera with respect to the IMU is difficult to obtain exactly in the real world. Incorrect calibration of these parameters will accumulate errors, since a misalignment in the orientation will incorrectly integrate the gravity component into the state. We have skipped this problem by using a simulator where we specify the transformation perfectly, but that is not the case in the real world.
- *Add the IMU misalignments to the state vector.* MEMS accelerometers and gyroscopes are not perfect and may have misalignments. A misalignment in an accelerometer will propagate the gravity into other axes, leading to errors when integrating. Such misalignments can be modelled using a shape matrix, whose values can be calibrated during filter operation.
- *Add the camera intrinsic parameters to the state vector.* We assumed known parameters for the camera, since we can obtain them from the simulator. In the real world this is not possible though, so a calibration process must take place. Estimating these parameters in the filter avoid the necessity to previously calibrate the camera.
- *Use original estimates for the Jacobian computations.* As proved in [14], computing Jacobians with corrected estimates for poses in the sliding window introduces undesirable terms in the observability matrix. Using the original estimates for the Jacobians fixes this problem.

- *Take into account the rolling-shutter effect of the camera [13].* Rolling-shutter cameras capture images by rapidly scanning the scene, as opposed to global-shutter cameras that capture the whole scene at once. This implies that different parts (pixel rows or columns) of the image are recorded at different moments. This causes distortions when the camera or the objects are moving. Since most hand-held and smartphone cameras have a rolling shutter, this improvement would be valuable for such devices.

Bibliography

- [1] Sameer Agarwal, Keir Mierle, and Others. *Ceres Solver*. <http://ceres-solver.org>. Accessed: 2015-04-27.
- [2] Mourikis A.I. and Roumeliotis S.I. “A Multi-State Constraint Kalman Filter for Vision-aided Inertial Navigation”. In: *Robotics and Automation, 2007 IEEE International Conference on* (2007).
- [3] Herbert Bay et al. “Speeded-up robust features (SURF)”. In: *Computer vision and image understanding* 110.3 (2008), pp. 346–359.
- [4] Javier Civera, Andrew J Davison, and J Montiel. “Inverse depth parametrization for monocular SLAM”. In: *Robotics, IEEE Transactions on* 24.5 (2008), pp. 932–945.
- [5] *Eigen linear algebra library*. http://eigen.tuxfamily.org/index.php?title=Main_Page. Accessed: 2015-04-27.
- [6] *Gazebo*. <http://gazebo-sim.org/>. Accessed: 2015-04-27.
- [7] Chris Harris and Mike Stephens. “A combined corner and edge detector”. In: (1988), pp. 147–151.
- [8] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004.
- [9] Richard I. Hartley and Peter Sturm. *Triangulation*. 1994.
- [10] Alonzo Kelly. *Mobile robotics : mathematics, models and methods*. New York, NY, USA: Cambridge University Press, 2013. ISBN: 110703115X.
- [11] *Khan Academy*. <https://www.khanacademy.org/>. Accessed: 2015-04-27.
- [12] Stefan Leutenegger, Margarita Chli, and Roland Yves Siegwart. “BRISK: Binary robust invariant scalable keypoints”. In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2548–2555.
- [13] Mingyang Li, Byung Hyung Kim, and Anastasios I Mourikis. “Real-time motion tracking on a cellphone using inertial sensing and a rolling-shutter camera”. In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 4712–4719.
- [14] Mingyang Li and Anastasios I Mourikis. “High-precision, consistent EKF-based visual-inertial odometry”. In: *The International Journal of Robotics Research* 32.6 (2013), pp. 690–711.
- [15] Mingyang Li and Anastasios I Mourikis. “Improving the accuracy of EKF-based visual-inertial odometry”. In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE. 2012, pp. 828–835.

- [16] Bruce D. Lucas and Takeo Kanade. “An Iterative Image Registration Technique with an Application to Stereo Vision”. In: *Proceedings of Imaging Understanding* (1981).
- [17] K. Madsen et al. *IMM Methods for non-linear least squares problems*. 2004.
- [18] *OpenCV Open Source Computer Vision Library*. <http://opencv.org/>. Accessed: 2015-04-27.
- [19] *ROS: The Robot Operating System*. <http://www.ros.org/>. Accessed: 2015-04-27.
- [20] Edward Rosten and Tom Drummond. “Machine learning for high-speed corner detection”. In: *Computer Vision—ECCV 2006*. Springer, 2006, pp. 430–443.
- [21] Joan Solà. “Quaternion kinematics for the error-state KF”. Mar. 2015. URL: <https://hal.archives-ouvertes.fr/hal-01122406>.
- [22] Joan Sola et al. “Impact of landmark parametrization on monocular EKF-SLAM with points and lines”. In: *International journal of computer vision* 97.3 (2012), pp. 339–368.
- [23] Joan Solà. *Interactive course on EKF and SLAM*. <http://www.iri.upc.edu/people/jsola/JoanSola/eng/course.html>. Accessed: 2015-04-27.
- [24] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. 2005.