POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master of Science in Computer Engineering

Master Degree Thesis

Development, integration and testing of algorithms to support autonomous flight, in the absence of GPS signal

Supervisors
prof. Alessandro Savino
prof. Stefano Di Carlo

Candidate
Matteo Accornero

Company Supervisor
Department of Exploration and Science, Thales Alenia Space Italia
ing. Carlo Paccagnini

Academic Year 2020-2021

Abstract

Modern navigational systems should function properly and dependably not just when a GPS signal is present, but even when it is absent or maliciously blocked. Traditional navigation systems fail to work in many GPS-restricted settings, such as inside, caverns, canyons, or when GPS is jammed or not even available such as an outer planet scenario. Many researchers are now proposing a variety of methods to address these constraints. Visual-Inertial Odometry (VIO) is one of several approaches for dealing with GPS-denied navigation that has piqued the scientific community's curiosity. Only a tiny portion of the offered methods can produce desirable accurate results and be considered for applications where acceptable Size, Weight, and Power (SWaP) are restricted, due to large processing needs and insufficient resilience when addressing complicated real-life scenarios.

The purpose of this work is firstly to provide a concise but complete classification of VIO algorithms and to offer a panoramic on State of the Art Techniques for UAVs Navigation in Critical environments. A deep analysis is carried out on the OpenVINS framework, published by the Robotics Group of Delaware's University with the purpose of enabling and facilitate the development and evaluation of new VIO algorithms.

The thesis focuses then to provide an embedded platform with enough computational capacity, based on Rock Pi N10 by Rockchip, to run the VIO exploiting both hardware and software needs to execute in real-time the pose estimation. Since it is crucial in order to obtain robust estimations, a complete walk through for sensor calibration with Kalibr and Kalibr_allans provided.

The system is finally evaluated inside a martian-like environment, thanks to the collaboration with Thales Alenia Space Italia, exploiting the framework capabilities and integrated analysis tools.

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Autonomous Navigation in GPS denied area

With the rapid advancement of drone technology, it is highly likely that drones will play a significant role in transportation, rescue, and other commercial or safety applications. A drone must be capable of operating in an unknown exterior environment for these applications, and its absolute position is normally determined via an external positioning system such as GPS. However, because this system relies on an external source of input, a drone hijacking is a distinct possibility. This is not only economically problematic, but it might also have significant consequences if the drones malfunction. To address this issue, a positioning system that is not reliant on external signals is recommended.
A possible list of scenarios [1] where GPS-denied drones are required could regard:

Indoor inspections : where the signal could suffer degradation due to thick walls or metallic tanks.

Mines or caves : those are similar to indoor inspection in terms of GPS challenges, or even worst, the signal just does not reach.

Bridges : flying near or under a metal bridge, drone's ability to connect to GPS may be hampered.

Search and rescue : those kind of missions often lead to environments, like searching for a missing person in a forest, where the GPS signal can weaken if the drone is required to operate under tree cover. Similar considerations apply for other natural obstructions you might encounter while on a search and rescue operation.

Space : obviously former planets doesn't have the support of a global navigation satellite system (or shortly GNSS) therefore other kind of possibility must be considered, like visual inertial navigation or deep learning.

However, the difficulty of UAV vision-assisted navigation in aerial robotics necessitates a system with various objectives. Drones are unstable in position and/or attitude, and aerial vehicles have quick dynamics and are resource restricted. As a result, precise and quick

vehicle condition estimate is required for flight control. Moreover, because cameras are frequently fixed to the vehicle, faults in the navigation algorithm output might be amplified by the vehicle's movement in reaction to the erroneous output, resulting in eventual instabilities and vehicle loss. This creates a strong, potentially destabilizing link between the navigation and control problems, where a bad navigation solution may quickly destabilize the entire system [2].

1.2 Ingenuity Example

NASA's Jet Propulsion Laboratory (JPL) developed and manufactured Ingenuity, a miniature robotic helicopter that will be used on Mars as part of NASA's Mars 2020 mission. Ingenuity was designed to fly up to five times at heights varying from 3–5 m above the ground for up to 90 seconds each during its 30-day technological demonstration. It is powered by solar-charged batteries that aliment two counter-rotating rotors piled one above the other. With such technical achievements, Ingenuity accomplished its initial goals. The missions demonstrated the helicopter's capacity to fly in the ultra-thin atmosphere of a former planet without requiring direct human control. Ingenuity is self-contained, performing movements that JPL has planned, programmed, and sent to it.
It functions under certain aspects as a typical spacecraft, with a sequencing engine on board that allows a set of commands to be uploaded in file form and then executed by the helicopter. The guidance portion of the flights is planned as a set of waypoints on the ground in simulation, and only those commands are given to the guidance program.
Flights are methodically planned, so there isn't true autonomy in the sense that there are no goals or rules, and there is no on-board learning or high-level processing. It's actually designed to fly along a pre-planned path on the ground before taking off [3, 4]. These

Figure 1.1: NASA Ingenuity Helicopter

Image Credits: NASA JPL [3]

achievements open a completely new frontier about autonomous flights on former planets. Since deep learning and artificial intelligence aren't mature yet to obtain this kind of critical qualification, visual-aided navigation seems to be the next step in the exploration of spacial skies.

1.3 Thesis Goals

The thesis is carried out with the collaboration of Thales Alenia Space Italy (TAS-I) which is one of the main player in the European and global space industry. The main goal is to analyze the research state of art about the Visual Inertial Navigation for drones having in mind a possible extraterrestrial mission, hypothesizing a step forward with respect to pre-determined Ingenuity flights.
In order to achieve this objective, the following steps take place:

1. a research as exhaustive as possible about algorithms their different characteristics, since it is a field in rapid expansion precisely because its space implications. A resume is given in Chapter 2;

2. a valid algorithm packed in a simply and extensible framework has been identified;

3. integration and calibration on the hardware available in TAS-I facility, which was the most time consuming phase and occupied the majority of the days Chapters 3 and 4;

4. an overview of the framework capabilities exploited by testing the system on the Martian field available at TAS-I Turin, Chapter 5.

This thesis has also the scope to work as a starting point for future development of the considered framework or in case of a new algorithm proposal as a guide of steps in order to obtain a correctly working system.

Chapter 2

State of art

The field of mobile robots and autonomous systems has gotten a lot of attention from researchers all around the world in recent decades, resulting in significant advancements and discoveries. Mobile robots can now execute complicated tasks independently, whereas in the past, human input and interaction were required. Mobile robots may be used in a variety of sectors, including military, medical, space, entertainment, and home appliances. Mobile robots are intended to execute difficult activities that involve navigation in complex and dynamic interior and outdoor situations without the need of humans in such applications. The robot must be able to locate itself in its surroundings in order to independently navigate, path plan, and perform these activities effectively and securely. As a result, the localization problem has been thoroughly investigated, and many approaches for solving the problem have been presented.
Wheel odometry techniques, which rely on wheel encoders to determine the amount of rotation of robot wheels, are the most basic type of localization. Wheel rotation data are utilized gradually in combination with the robot's motion model to determine the robot's present location in relation to a global reference coordinate system in those techniques. Because the localization is incremental (based on the prior estimated position), measurement errors accumulate over time, causing the estimated robot posture to wander from its real location. Wheel odometry techniques include a variety of inaccuracy causes, the most prominent of which being wheel slippage on uneven terrain or slick flooring.
Other localization procedures have been developed to overpass these constraints, including the use of inertial measuring units (IMUs), GPS, LASER odometry, and most recently Visual Odometry (VO) and Simultaneous Localization and Mapping (SLAM) approaches. VO is the technique of estimating an agent's egomotion utilizing just the input of a single or many cameras connected to it (e.g., a vehicle, a person, or a robot). SLAM, on the other hand, is a process in which a robot must simultaneously locate itself in an unfamiliar area and construct a map of that environment without any prior knowledge using external sensors (or a single sensor). Although VO does not address the drift problem, researchers have demonstrated that it outperforms wheel odometry and dead reckoning approaches, and that cameras are far less expensive than precise IMUs and LASER scanners. The primary distinction between VO and SLAM is that VO focuses primarily on local consistency and tries to progressively estimate the route of the camera/robot pose after pose, as well as perhaps conducting local optimization. SLAM, on the other hand, seeks to provide a

globally consistent estimate of the camera/robot trajectory and map. Global consistency is achieved by recognizing that a previously mapped region has been revisited (loop closure) and using this knowledge to decrease estimation drift.

2.1 Algorithms

The technique of calculating the robot's mobility (translation and rotation with respect to a reference frame) by viewing a series of pictures of its surroundings is known as VO. Structure From Motion (SFM) is a technique that addresses the challenge of 3D reconstruction of both the structure of the environment and camera positions from sequentially ordered or unordered picture data. The final refining and global optimization stage of SFM, which affects both the camera positions and the structure, is computationally intensive and is generally done offline. The estimate of the camera postures in VO, on the other hand, must be done in real-time. Many VO approaches have been developed in recent years, and they may be classified into monocular and stereo camera methods. Feature matching (matching features across several frames), feature tracking (matching features in consecutive frames), and optical flow approaches are all subsets of these methods (based on the intensity of all pixels or specific regions in sequential images).

2.2 SLAM, VIO and VINS

SLAM estimators are able to readily include loop closure restrictions by concurrently estimating the location of the sensor platform and the features in the surrounding environment, allowing for limited localization errors, which has garnered substantial research efforts in the past three decades. VINS, which includes the visual-inertial (VI)-SLAM and the visual-inertial odometry, can be thought of as a kind of SLAM, using specific visual and inertial sensors.
The former estimates feature positions and the camera/IMU pose combined to generate the state vector, whereas the latter does not include features in the state but still uses visual measurements to impose motion restrictions between camera/IMU poses. In general, the VI-SLAM achieves improved accuracy from the feature map and probable loop closures while incurring more computing cost than the VIO by performing mapping (and hence loop closure).
VIO estimators, on the other hand, are simply odometry techniques, with unbounded localization errors unless some global information (e.g., GPS or a priori map) or restrictions to past positions (e.g., loop-closures) is utilized. To reduce drift across the trajectory, several techniques use feature measurements from distinct keyframes. Most use a two-thread approach that optimizes a tiny window of "local" keyframes and features in the near term, while a background process optimizes a long-term sparse pose graph with loop-closure restrictions to enforce long-term consistency. Loop-closure restrictions, for example, are used in both the local sliding window and the global batch optimization in VINS-Mono [5]. Specifically, feature observations from keyframes give implicit loop-closure restrictions during local optimization, while the problem size is kept short by assuming ideal keyframe poses (thus removing them from optimization).

One of the fundamental distinctions between VIO and SLAM is whether or not loop closures are performed in VINS via mapping and/or location recognition. While employing loop-closure information is necessary for bounded-error VINS performance, it is difficult because to the inability to stay computationally efficient without introducing inconsistencies like considering keyframe poses as true or recycling information. To this goal, a hybrid estimator was presented in which the MSCKF was employed to do real-time local estimating and a global BA was activated on loop-closure detection. This enables for consistent relinearization and loop-closure restrictions while needing significant additional overhead time.

2.3 Filtering-based vs. Optimization-based Estimation

The multi-state constraint Kalman filter (MSCKF), one of the first effective VINS algorithms, leverages quaternion-based inertial dynamics for state propagation closely linked with an efficient EKF update. MSCKF was later used to rapid UAV autonomous flying. Instead of adding features identified and monitored over camera images to the state vector, their visual bearing measurements are projected onto the null space of the feature Jacobian matrix (i.e., linear marginalization), preserving motion constraints that only apply to probabilistically cloned camera poses in the state vector. While avoiding the requirement to co-estimate potentially hundreds of thousands of point features reduces the computing cost, it also inhibits the relinearization of the features' nonlinear measurements at a later time, resulting in approximations that degrade performance.
MSCKF has lately been expanded and enhanced in a variety of ways. Algorithms to increase filter consistency by enforcing the right observability qualities of the linearized VINS have been created by utilizing the observability-based technique. The square-root inverse sliding window filter (SR-ISWF) was created as a square-root inverse variant of the MSCKF to enhance computational efficiency and numerical stability, allowing VINS to run on mobile devices with limited resources without reducing estimate accuracy. The MSCKF-based VINS has also been developed to include rolling-shutter cameras with imperfect time synchronization, RGBD cameras, multiple cameras, and multiple IMUs. While filtering-based VINS have demonstrated high-accuracy state estimation, they do have one theoretical drawback: nonlinear data must be linearized once before being processed, which might introduce substantial linearization errors into the estimator and degrade performance.
By contrast, batch optimization approaches address a nonlinear least-squares problem over a collection of data, allowing for error reduction via relinearization but at a high computing cost. When used to VINS, a bounded-size sliding window of current states is often only regarded as active optimization variables, while older states and measurements are marginalized out. A keyframe-based optimization strategy was proposed in OKVIS, in which a set of non-sequential historical camera poses and a series of recent inertial states, linked to inertial measurements, were employed in nonlinear optimization for precise trajectory estimation.

2.4 Tightly-coupled vs. Loosely-coupled Sensor Fusion

VINS may integrate visual and inertial measurements in a variety of ways, with the loosely-coupled and tightly-coupled systems being the most common. In particular, in either filtering or optimization-based estimation, the loosely-coupled fusion processes the optical and inertial measurements individually to infer their own motion constraints, which are then fused.
Although this approach is computationally efficient, information is lost due to the decoupling of visual and inertial constraints. Tightly linked techniques, on the other hand, immediately integrate the optical and inertial data inside a single process, resulting in improved precision.

2.5 Direct vs. Indirect Visual Processing

The visual processing pipeline is one of the most important parts of any VINS, as it is capable of transforming dense imagery input into motion constraints that can be employed in the estimate problem, with techniques that are either direct or indirect depending on the visual residual models used. Indirect approaches, sometimes known as the traditional methodology, extract and track point characteristics in the environment while estimating geometric reprojection constraints. The ORB-SLAM2, which performs graph-based optimization of camera positions using information from 3D feature point correspondences, is an example of a current state-of-the-art indirect visual SLAM.
Direct approaches, on the other hand, are based on raw pixel intensities and allow for the incorporation of a higher percentage of the available picture data. LSD-SLAM is a state-of-the-art direct visual-SLAM that optimizes the transition between pairs of camera keyframes by reducing the intensity error between them. This method additionally optimizes a secondary graph incorporating keyframe restrictions, allowing extremely informative loop-closures to be used to correct drift across lengthy trajectories. Direct approaches for VINS have recently gotten a lot of attention because of their ability to track dynamic motion even in low-texture surroundings.
Because of the photometric consistency assumption, direct picture alignments necessitate a reasonable initial guess and a high frame rate, whereas indirect visual tracking necessitates additional computer resources for extracting and matching features. However, because of their maturity and robustness, indirect methods are more often employed in actual applications, while direct approaches offer potential in textureless circumstances [6].

2.6 OpenVINS

After carefully considering the different solutions with each one its drawbacks the focus moved on the OpenVINS [7] framework.
The lack of VINS codebases with comprehensive documentation and detailed derivations for which even users with little background can learn and extend a current state-of-the-art work to address their problems at a low cost has proven to be a significant hurdle for

researchers in the robotics research community.
While there are a number of open-source visual-inertial codebases available, they are not designed for extensibility and lack good documentation and assessment tools, which are essential for quick creation and thorough comprehension. Furthermore, numerous hard-coded assumptions or characteristics exist in these systems, necessitating a thorough study of the codebases in order to adapt them to the sensor systems at hand. This, along with a lack of documentation and support, limits their use in a variety of applications.
OpenVINS covers the aforementioned needs in the community by providing an expandable, open-source codebase developed specifically for researchers and practitioners with either modest or significant previous knowledge of state estimate.
Its key functionality of the different components are:

ov_core : includes 2D image sparse visual feature tracking, linear and Gauss-Newton feature triangulation techniques, a visual-inertial simulator for any number of cameras and frequencies, and basic manifold math operations and utilities;

ov_eval : includes trajectory alignment, charting tools for evaluating trajectory accuracy and consistency, Monte-Carlo evaluation of several accuracy measures, and a utility for saving ROS topics to a file;

ov_msckf :provides an expandable modular EKF-based sliding window visual-inertial estimator with an on-manifold type system for flexible state representation.

In addition these experimental features are available within OpenVINS GitHub repository [8]:

ov_secondary :, an example secondary thread enables loop closure in a loosely linked manner. This is a modified version of the code developed by the HKUST aerial robotics group, which is available in their VINS-Fusion repository;

ov_maplab : the interface wrapper for exporting visual-inertial runs from OpenVINS into the ViMap structure used by maplab is contained in this source. As OpenVINS runs through a dataset, the state estimates and raw pictures are added to the ViMap. Features are re-extracted and triangulated using maplab's feature system when the dataset is completed.

vicon2gt : this utility was intended to produce groundtruth trajectories for use in testing visual-inertial estimating systems utilizing a motion capture device (e.g. Vicon or OptiTrack).

Chapter 3

System

3.1 Hardware

In this section is presented the hardware adopted in order to run OpenVINS's algorithms. It consists in two different sensors configurations, both based on the Rock Pi N10 Model C board. The first solution, utilized in the early phase of the thesis, involves decoupled sensors (IMU and camera, presented afterwords) software-synchronized while the second one relies on a hardware-synchronized visual inertial sensor produced by Intel™. The hardware upgrade was necessary because the first configuration, coupled with the board, wasn't able to run the VIO algorithms resulting in drifting trajectories during the early seconds of the pose estimation.

3.1.1 Board

Rock Pi N10 is a single-board computer, produced by Redxa, based around the Rockchip RK3399Pro SoC [9] and featuring a co-processor offering up to 3 TOPS of deep-learning performance for compatible workloads. The Rock Pi N10 has three models:

- Model A comes with 4GB LPDDR3 (3GB for CPU, 1GB for NPU) and 16GB eMMC;
- Model B has 6GB LPDDR3 (4GB for CPU, 2GB for NPU) and 32GB eMMC;
- Model C has the highest amount of RAM of 8GB LPDDR3 (4GB for CPU, 4GB for NPU) and eMMC storage of 64GB.

The Rock Pi N10 is plenty of interfaces. Like the more known Raspberry 4B, it has rich interfaces for audio, camera, display, Ethernet, USB and I/O pins. The Ethernet interface can support PoE (Power over Ethernet) function and has a dedicated hat near the interface. Wi-Fi is not supported natively, but there is an optional module to be embedded on the board. Redxa supports Debian 9 Desktop, Ubuntu 18.04 Server, Fedora Desktop, and Android 8.1, for which provides community images with drivers already integrated. As well, all AI development tools such as Rock-X set of rapid AI components, RKNN-API C API, and RKNN-Toolkit Python API of RKNN (Rockchip Neural Network) are provided. A complete and exhaustive overview of features is given in Tab. 3.1.

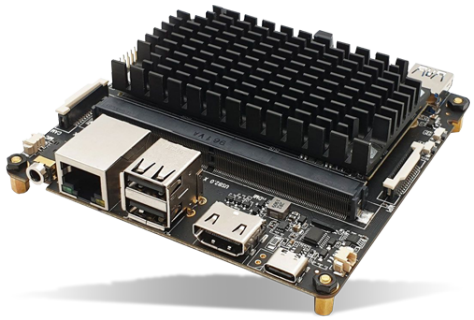


Figure 3.1: Rock Pi N10 Model C board

Image Credits: SeeedStudio [10]

SoC	RK3399Pro
CPU	Dual Cortex-A72, freqency 1.8GHz with quad Cortex-A53, frequency 1.4GHz
GPU	Mali T860MP4
NPU	up to 3.0 TOPs computing power
Memory	8GB LPDDR3 @ 1866Mb/s 4GB for CPU/GPU, 4GB for NPU
Storage	eMMC 64GB
Storage	4MB SPI flash µSD card (µSD slot supports up to 128 GB µSD card) M.2 SSD (M.2 connector supports up to 8TB M2 NVME SSD)
Display	HDMI 2.0 up to 4k*2k@60MIPI DSI 2 lanes via FPC connector 4 lanes eDP 1.3, up to 4K*2K@60
Audio	3.5mm jack with micHD codec that supports up to 24-bit/96KHz audio.
Camera	MIPI CSI 2 lanes via FPC connector, support up to 800 MP camera.
Wireless	None. Optional ROCK Pi Wireless Module
USB	1x USB 3.0 OTG, hardware switch for host/device switch, front one 2x USB 2.0 HOST
Ethernet	GbE LAN with Power over Ethernet (PoE) support additional HAT is required for powering from PoE
IO	40-pin expansion header 2 x UART, 2 x SPI bus, 3 x I2C bus, 1 x PCM/I2S, 1 x SPDIF, 2 x PWM, 1 x ADC, 6 x GPIO, 2 x 5V DC power in, 2 x 3.3V DC power in
Power	USB PD, support USB Type C PD 2.0, 9V/2A, 12V/2A, 15V/2A, 20V/2A. Qualcomm® Quick Charge™: Supports QC 3.0/2.0 adapter, 9V/2A, 12V/1.5A
Size	100mm x 100mm

Table 3.1: Rock Pi N10 Model C features

3.1.2 Camera

The EO (Edmund Optics) USB 2.0 cameras are equipped with a host of features to make machine vision easy. Each camera provides an interface that allows to set a specified Area of Interest (AOI), gain, exposure time, frame rate, trigger delay and select digital output (flash) delay and duration. There is also the possibility to set the exposure, gain and white balance to a specified level or allow the camera to adjust these parameters automatically. The camera supports mages in JPEG and Bitmap file format or video in AVI format.
A complete and exhaustive overview of features is given in Tab.3.2.

(a) EO-1312M camera body

(b) Fixed-focus 50mm lens

Figure 3.2: Camera configuration

Images Credits: Edmund Optics [11]

Manufacturer	EO
Camera Family	uEye CP
Model Number	1312M
Type	Monochrome Camera
Imaging Sensor	ON Semi MT9M001
Type of Sensor	Progressive Scan CMOS
Type of Shutter	Rolling
Camera Sensor Format	1/2"
Resolution ($MegaPixels$)	1.30
Pixels (HxV)	1,280 x 1,024
Frame Rate (fps)	25.00
Pixel Depth	8 bit
Exposure Time	37µs - 0.983s
Image Buffer	0MB
Video Output	USB 2.0
GPIOs	2 Flash Output, 2 Trigger Input, 1 USB Power Supply, 1 Ground, 1 Shield, 2 USB Data
Dimensions (mm)	34 x 32 x 41.3 (includes connectors and lens mount)
Weight (g)	58

Table 3.2: Edmund Optics Camera features

3.1.3 IMU

The MPU6050 [12,13] is an integrated 6-axis MotionTracking device that combines a 3-axis gyroscope, 3-axis accelerometer enclosed in a small package. It is also designed to interface with multiple non-inertial digital sensors, such as pressure sensors, on its auxiliary I2C port. It consists of three independent vibratory MEMS rate gyroscopes, which detect rotation about the axis. When gyros are rotated about any of the sense axes, the Coriolis Effect causes a vibration that is detected by a capacitive pickoff. The resulting signal is amplified, demodulated, and filtered to produce a voltage that is proportional to the angular rate. This voltage is digitized using individual on-chip 16-bit Analog-to-Digital Converters (ADCs) to sample each axis. The MPU's 3-axis accelerometer uses separate proof masses for each axis. Acceleration along a particular axis induces displacement on the corresponding proof mass, and capacitive sensors detect the displacement differentially. A complete and exhaustive overview of features is given in Tab.3.3.

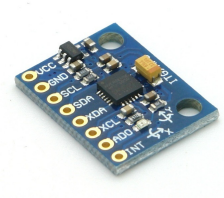


Figure 3.3: MPU6050 six axis MEMS on GY-521 board

Image Credits: Invesense [14]

Gyro Full Scale Range ($°/sec$)	$\pm$250, $\pm$500, $\pm$1000, and $\pm$2000
Gyro Sensitivity ($LSB/°/sec$)	131, 65.5, 32.8 and 16.4
Gyro Rate Noise ($mdps/rtHz$)	0.005
Accel Full Scale Range (g)	$\pm$2,$\pm$4, $\pm$8 and $\pm$16
Accel Sensitivity (LSB/g)	16384, 8192, 4096 and 2048
Digital Output	I2C
Logic Supply Voltage (V)	1.8V$\pm$5% or VDD
Operating Voltage Supply (V)	2.375–3.46
Package Size (mm)	4 x 4 x 0.9

Table 3.3: MPU6050 features

3.1.4 Intel

The choice fell on the brand new d455 sensor which fuses the best features of previous sensors (like d435i and t265, widely used for visual odometries) in a ready to use vision and motion system. It's a stereo vision global shutter camera which allows things like depth measurement in scenes within such applications. It integrates also an IMU, which can help refine the depth awareness in situations where the camera moves. The depth

sensors themselves are also mounted on the same stiffener as the RGB sensor, which helps with color and depth alignment. That RGB sensor has the same field of view as the depth sensors hat can improve the correspondence between the two data straps they generate, along with matching the field of view between those sensors

Figure 3.4: Intel™ D455 Visual Inertial Sensor

Image Credits: Intel™ [15]

Depth FOV (HxV)	86°±3° × 57°±3°
Depth resolution (HxV)	1280 × 720, Global Shutter
Depth frame rate (fps)	up to 90
RGB FOV (HxV)	86° ±3° × 57° ±3°
RGB resolution (HxV)	1280 × 800, Global Shutter
RGB frame rate (fps)	up to 90
Dimensions (mm)	124 x 26 x 29

Table 3.4: Intel™ Camera features

Gyro Full Scale Range ($°/sec$)	±125, ±250, ±500, ±1000, and ±2000
Gyro Sensitivity ($LSB/°/sec$)	262.4, 131.2, 65.6, 32.8 and 16.4
Gyro Rate Noise ($mdps/rtHz$)	0.014
Accel Full Scale Range (g)	±2,±4, ±8 and ±16
Accel Sensitivity (LSB/g)	1024, 512, 256 and 128
Digital Output	I2C, SPI
Logic Supply Voltage (V)	1.2V±5% or VDD
Operating Voltage Supply (V)	2.4-3.6
Package Size (mm)	3 x 4.5 x 0.95

Table 3.5: BMI055 features

3.2 Software

This section covers all the software aspects of the system. It is based on Ubuntu 18.04 Server, which binaries have been downloaded from the official repository of the board in order to have all drivers already working after the first installation. The OS choice has been made for two different reason:

1. incompatibilities: ROS and Ubuntu versions are strictly coupled, for each major stable release of one corresponds a stable release of the other. A tentative with Debian 9.0 has been made but, even if officially supported by the community, some dependencies problems came out.

2. resources: since the board has limited capabilities the server edition of the OS has been installed. This allowed to control strictly the packages and to choose a lighter graphical environment with respect to the default one.

3.2.1 ROS Introduction

The Robot Operating System (ROS) [16, 17] is an open source operating system that provides libraries and tools to help software developers create robot applications. It offers hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. The major objective of ROS is to promote code reuse in robotics research and development, and as a result, it has been swiftly accepted as the standard development framework by many robotics institutes and enterprises. ROS is used by a large number of robots in a wide range of fields, including aerial and ground robots, as well as humanoids and underwater vehicles. Furthermore, ROS already has support for a large number of common robot sensors. Sensors such as Inertial Measurement Units, GPS receivers, cameras, and lasers, to mention a few, can already be accessible using the ROS system's drivers. Because ROS is built on Linux, it cannot guarantee a system's hard real-time characteristics, but it does allow soft real-time applications. It's also simple to connect with other open-source software libraries like OpenCV, Point Cloud Library, and Gazebo Simulator.

Design Principles

ROS was created to address a specific set of issues that arise when creating large-scale service robots, but the resultant architecture has applications outside those. It's a set of tools, libraries, and protocols aimed at making building complicated and reliable robot behaviors easier on a range of robotic systems. As a summary, it offers:

- hardware abstraction
- low-level device control
- implementation of commonly-used functionality
- message-passing between processes and package management
- tools and libraries for obtaining, building, writing, and running code across multiple computers.

Above the host operating systems of a heterogeneous computer cluster, ROS offers a structured communications layer. It was created with a modular, tool-based software development methodology in mind. Its goal is to make available robot sensor visualizations, sensor fusion, and control algorithms as reusable as possible. It has progressively attracted a large number of developers and has evolved into a kind of standard framework for robotic

systems (mostly in research). The library is supported for a Unix-like system (Ubuntu). Support for other operating systems is considered experimental. ROS follows the following design goals:

- *Thin*: it is meant to be as light and easy to integrate as feasible with existing robot software frameworks;
- *Language independence*: the ROS framework is simple to implement in any current programming language, however Python, C++, and, experimentally, Java are the most commonly utilized;
- *Scaling*: ROS is well-suited to large-scale runtime systems and development processes.

Main Concepts

The ROS runtime "graph" is a loosely connected peer-to-peer network of processes (perhaps spread across computers) that uses the ROS communication infrastructure. This indicates that these components are working together to process data through a variety of ways. Nodes, messages, topics, and services are the key concepts of the ROS. Here is presented the graph related to IMU, camera and clock nodes discussed afterwords.

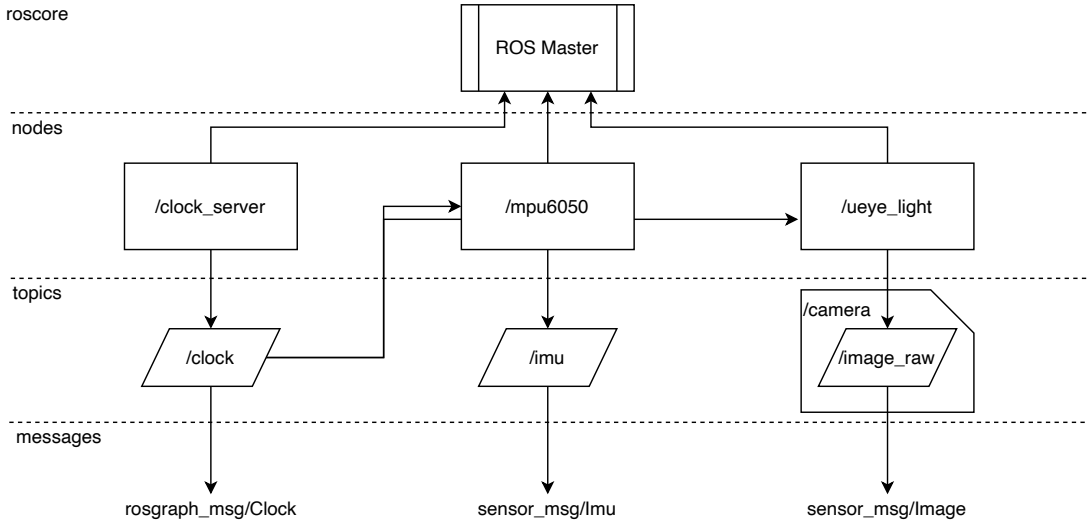


Figure 3.5: IMU, camera, clock nodes rosgraph diagram

Nodes A node is a software process that represents a module or component and conducts computation. At a fine-grained scale, ROS is meant to be modular: a system is generally made up of several nodes. Sensor and actuator drivers, state estimation methods, user interfaces, and other examples are common. The usage of nodes in ROS has a number of advantages for the overall system. It strengthens the system's modularity by requiring particular functionality to be deployed as discrete nodes, which increases fault tolerance. In comparison to monolithic systems, it facilitates the encapsulation of implementation

details. The rest of the system is exposed to a basic API, which can be based on other technologies as long as they can interact using the ROS protocols. The name of a node is the only way to identify it. This identification will be used by every tool in the ROS framework to refer to a specific node. There is a node type for each node (equivalent to their class in object oriented programming). The name of the executable and the package to which it belongs make up a node type. This places certain limitations on how you construct your packages and name your project's type.
`rosnode` is the ROS command line tool to display information about nodes.

Messages By passing messages, a node can connect with another node (peer-to-peer). A message is a data structure that is tightly typed. Standard primitive types (integer, floating point, boolean, and so on) as well as primitive type arrays and constants are supported. Messages can be made up of other messages and arrays of other messages, and they can be layered indefinitely deep. Messages are defined in message files, which are simply text files that specify the message's data structure. The following is the ROS naming standard for message types: package name + message name. ROS generates source code in both C++ and Python to use this message in a ROS node.
Nodes communicate by sending ROS messages to each other using ROS Topic. A message can be of primitive type integer, floating-point, boolean, etc. A publisher and subscriber should communicate using the same topic type. The topic type is determined by the message type.
`rosmsg` is the ROS command line tool to display information about messages.

Topics In ROS, topics are the buses used by ROS nodes to exchange messages and those are exchanged between nodes using a publisher/subscriber model. A message is sent out by a node by publishing it to a topic. It transports message between a publisher node and a subscriber node and they have anonymous publish/subscribe semantics. Nodes that generate message/data publish to a specific topic, and nodes that consume or need data subscribed to a specific topic. The relationship between publishers and subscribers is many to many. A single topic can have many publishers and subscribers at the same time, and a single node can publish and/or subscribe to numerous topics. Publishers and subscribers are generally unaware of one other's existence, which separates the creation and consumption of information.
The ROS message type types a subject. When a node subscribes to a topic, it performs a type matching check, and communication happens only if both the topic's publisher and subscriber are using the same message type. TCP-IP (TCPROS) and UDP (UDPROS) can be used to communicate ROS topics. The default transport in ROS is TCP-IP. `rostopic` is the ROS command line tool to display information about topics, including performance statistics such as:

- period of messages by all publishers (average, maximum, standard deviation);
- age of messages, based on header timestamp (average, maximum, standard deviation);
- number of dropped messages;
- traffic volume in bytes.

Services Since topics are designed for one-way, real-time conversation, ROS service is one to one two way transport and it should be used instead by nodes that need to conduct remote procedure calls, i.e. for request/reply interactions. This kind of exchanges are handled by services: clients utilize a service that the supplying node offers under a name by sending a request message and waiting for a response. A service is defined by a pair of messages, one for the request and another one for the reply. They are defined using srv files, just like msg files, except they contain two parts: a request and a response. The call from client is blocking until the reply is received. Similarly to messages, services are defined in service files, a simple text file that is compiled to automatically generate a stub implementation of the service.
`rossrv` and `rosservice` are the ROS command line tool to display information about services.

Master Provides name registration. Without the master, nodes would not be able to find each other (like a DNS). The role of the master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. It provides an XML-based API, which ROS nodes call to store and retrieve information. Primitives such as registerService, registerSubscriber and registerPublisher are available, but this is encapsulated under the client libraries roscpp and rospy. The user does not need to implement the interface at this level.
`roscore` is the command line tool to start the master.

Bags A format for saving and playing back ROS messages, it is widely used for developing and testing algorithms.
`rosbag` is the ROS command line tool to display information about services.

3.2.2 Clock Server Node

Clock node (App. C) is quite straightforward and is the only piece of software developed in python. It generates ROS `clockMessage`s to which camera and IMU nodes are subscribed in order to synchronize operations between them, since it is a fundamental condition to obtain a proper estimation.
It is based on a `timer` that triggers a callback when it expires, in which the messages are generated and then published. This approach underlines better performances with respect to `rate-sleep` one because it exploits better resistance to scheduling routines.

3.2.3 IMU Node

IMU node (App. A) is responsible of generating and publishing ROS IMU sensor messages at the desired framerate, providing also access to the offsets configuration procedure (see Sec. 4.1.1). It is structured as follow:

- wrapper class;
- node interface.

Node interface has simply the duty to setup ROS environment, to instantiate a MPUWrapper object and to startup the publication. The wrapper class includes all attributes and methods needed to properly initialize the sensor, it checks parameters validity and in accord to them setups the node. It can be launched in two modalities, through `_synchro` parameter, which controls the timing publication source:

- `_synchro:=timer`: the timer expiration generates the message and its publication;
- `_synchro:=clock`: the reception of a clock messages triggers the reading and publishing procedure;
- default (no/wrong parameter): timer source.

Different callbacks are needed in order to exploit these features and this is the main reason behind the wrapper approach. Since that the sensor measurements procedure is the same, independently from the time source, and obviously the messages generated with the two methods are the same, this approach permits to reuse the majority of code allowing the desired flexibility. Sensor data are read through I2C protocol with the help of mraa

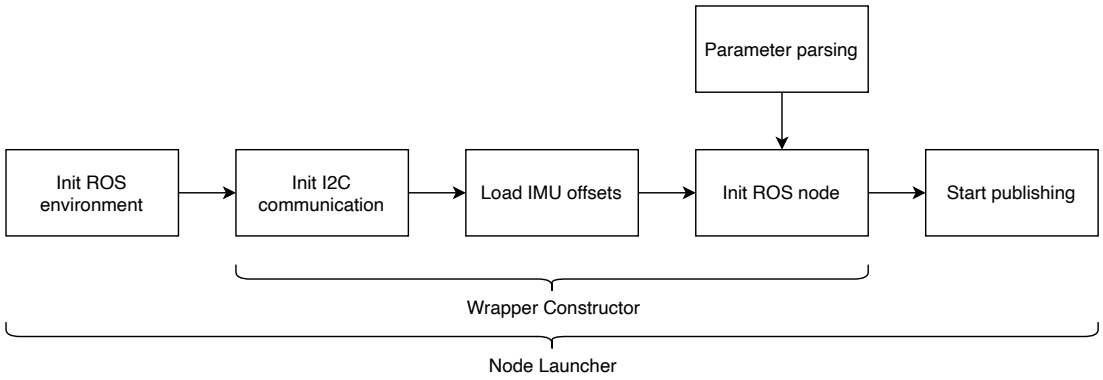


Figure 3.6: ROS IMU node flowchart

library [18]. Communication primitives are written in C and then wrapped up for several programming languages, including C++, for which everything is wrapped in a quite convenient class. The library is supported from board producers, it can be installed directly from official repository, but a bug related to onboard I2C ports has been found. Ports enumeration doesn't match the documentation, but the configuration presented in the code section dedicated to defines instead. It has been reported to producers.
Registers are read in blocks of words (16 bits at time), transformed in the correct representation (little endian to big endian conversion) and scaled in accord to the proper scale and then converted to the standard measurement units (m/s^2 for accelerometer and rad/s for gyroscope). Finally timestamp from the selected timing source and measurements are used to fill the message and then publish it. For more information about scales and conversions see Sec. 4.1.1.
In summary the available commands within IMU node are:

- running the node:

```
rosrun mpu6050 mpu6050_node _synchro:=[clock, timer]
```

- configuring offsets (mandatory before usage):

```
rosrun mpu6050 calibration
```

3.2.4 Camera Node

The design pattern (wrapper-node interface) behind the camera node (App. B) is the same of the IMU one. Observations made for `_synchro` parameter and timing sources are valid also for this node. The only difference regards the frequency: from the moment that clock messages have the rate of the fastest node (IMU in this case), a mechanism for throttling must be carried out. It is performed in a very simple way: since camera rate is ten times lower then IMU (and therefore clock) one ($20Hz$ vs $200Hz$) a counter unlocks generating and publishing the Image message once ten clock messages have been received. This is not needed in case of timer callback because it is set to expire at the correct rate.
The camera is configured at startup through a .ini file which includes all parameters like exposure, hardware framerate, pixel clock and image size. It is performed through the official SDK (installed automatically with drivers) which provide instant feedback of changes and then exported.
The sensor communicates with the board through USB 2.0 interface and proprietary uEYE APIs and primitives. Images cannot be transferred directly from the sensor to messages since camera doesn't have a dedicated memory. Therefore those must be loaded first in RAM with the appropriate function and then the pointer to the associated memory region can be passed to the function delegated to reserve the correct amount of space inside the message. The latter is created once, filled with the information related to the image (like size and bit encoding for example) which are constant during the message stream so only timestamp and image matrix are changed at each publication. From the moment that

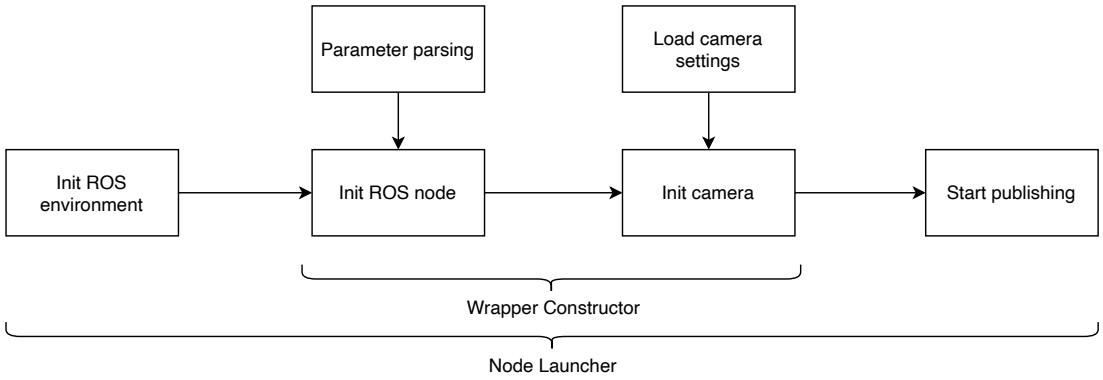


Figure 3.7: ROS camera node flowchart

transferring camera images at full resolution is quite a time consuming operation and there are strictly synchronization constraints a parameter to manage sizing has been designed.

- `_res:=full`: full resolution configuration file is loaded at startup;

- `_res:=scaled`: scaled resolution configuration file is loaded at startup;
- default (no/wrong parameter): full resolution.

In summary the available commands within camera node is:

```
rosrun ueye_light ueye_light_node _synchro:=[clock, timer] _res:=[full,
scaled]
```

3.2.5 Intel™ Sensor Node

Intel™ provides open source out-of-the-box working ROS node which benefits of direct maintenance from the company. Node is launched through a different mechanism, called `roslaunch`, which comes useful in situation where a lot of parameters must be set and passed to the node (OpenVINS itself use this approach). Since that the default launcher provides a lot of topics and data, in order to maintain simplicity, a custom one has been created. It basically publishes topics related to the two depth cameras (with the laser points projection disabled to avoid its pattern detection) and IMU. Unfortunately SDK is not available on board platform, so in order to generate the .json file it must be installed on a different computer. Once the configuration is performed it should be exported and then copied in the correct folder.
Another important aspect is IMU: from the moment that gyroscope and accelerometer has different rates (the latter is faster) in order to obtain a consistent stream between measurements the fastest rate has been selected and then gyroscope readings interpolated on a linear model. Thereby a stream of message at $200Hz$ can be reached.

3.2.6 Synchronization

To fuse data from different sensors, time instants at which measurements are recorded must be precisely known. In practice, the timestamps of each sensor typically suffer from triggering and transmission delays, leading to a temporal misalignment (time offset) between different sensor streams. Consequently, the time synchronization of sensors may cause a crucial issue to a multi-sensor system. For the visual-inertial system, the time offset between the camera and IMU dramatically affects robustness and accuracy. Most visual-inertial methods assumed measurements' timestamps are precise under a single clock. Therefore, these methods work well with a few strictly hardware-synchronized sensors. For most low-cost and self-assembled sensor sets, hardware synchronization is not available. Due to triggering and transmission delays, there always exists a temporal misalignment (time offset) between camera and IMU. The time offset usually ranges from several milliseconds to hundreds of milliseconds. Dozens of milliseconds will lead to IMU sequences totally misalignment with image stream, thus dramatically influencing the performance of a visual inertial system.

3.2.7 Workspace configuration

```
catkin_workspace
└── openvins_ws
    └── ...
```

```
├── realsense_ws
│   └── ...
└── sensors_ws
    ├── build
    ├── devel
    ├── log
    └── src
        ├── camera
        │   ├── config
        │   │   ├── full.ini
        │   │   └── scaled.ini
        │   ├── include
        │   │   └── EOWrapper.hpp
        │   ├── src
        │   │   ├── camera.cpp
        │   │   └── EOWrapper.cpp
        │   ├── CMakeLists.txt
        │   └── package.xml
        ├── clock
        │   └── clock.py
        └── imu
            ├── config
            │   ├── calibration
            │   ├── calibration.c
            │   └── offsets.yaml
            ├── include
            │   └── MPUWrapper.hpp
            ├── src
            │   ├── mpu.cpp
            │   └── MPUWrapper.cpp
            ├── CMakeLists.txt
            └── package.xml
```

Chapter 4

Calibrations

In order to obtain accurate, reliable and robust visual inertial poses estimations calibration is a crucial process to complete.
The main reason is because of the complimentary nature of the two different sensors typologies. For slow motions, the camera provides a drift-free pose estimation, while for fast motions, the camera-based localization may fail, and the IMU is able to provide valuable measurements. This kind of duality allows an effective combination of their measurements, which in turn requires accurate temporal and spatial registration among them. The camera provides high density external measurements of the environment, while the IMU measures internal ego-motion of the sensor platform. IMU is fundamental in giving robustness to the estimator while also providing system scale in the case of a mono-camera configuration, like the considered one.
In second place, there is no guarantee that a sensor is working perfectly. It is best to get it calibrated at regular intervals (recommendations vary depending on the type of sensor and on its use). Sensors may also differ just through the manufacturing process, there is no guarantee that one sensor will work as well as another so calibration is the best way to ensure that it is working correctly. Some sensors may get less responsive over time; general wear and tear can cause them to not work to the capacity that they should. Therefore calibration is important to ensure that the sensor is able to function well.
However, other factors must be taken in account. An IMU requires estimating of additional bias terms and it necessary a highly accurate calibration between the camera and IMU. Additionally small errors in the relative timestamps between the sensors can also degrade performance very quickly in dynamic trajectories.
Calibration process is completely offline, since it must be completed before running any VINS (besides OpenVINS provides small adjustments and the possibility to refine those coefficients at runtime), and it is performed combining two different tools:

- *Kalibr* [19–22], a mature and complete framework used for camera intrinsic calibration, IMU-camera transformation and time shift computing;
- *kalibr_allan* [23], an open sourced toolbox used for evaluating IMU noises through Allan distributions.

NB: All the graphs and the results are the outcome of our calibration procedures, integrated and improved with respect to what is available in literature.

4.1 IMU

IMU calibration process consists of two distinct phases:

1. offsets registers calibration;

2. noises coefficients computation.

The first is needed to remove the zero-error both from gyroscope and accelerometer readings, while the second is performed to establish how and how much those readings are affected by noise.
Both procedures must be carried out with the sensor steady positioned on a flat surface, with the package pointing upwards, avoiding any kind of movements, shocks and changes of temperatures too.

4.1.1 Offsets

As said before, this part of the calibration is meant to remove the zero-error which refers to when the sensor records small variations even though it is totally level. This error can be removed by applying an offset, through built-in registers which are automatically summed up, to the raw accelerometer and gyroscope sensor readings. The offset needs to be adjusted until the gyroscope readings are zero (no rotation, since the sensor is steady) and the accelerometer records the acceleration due to gravity pointing directly downwards (in accord to the sensor reference system and its position).
The algorithm (App.D) used for the offset estimation is quite straightforward, based on a simple iterative approach. It works as follow:

1. First of all I2C connection is turned on and the sensor is resetted, deleting possible old offsets values;

2. Full-scale range and sensitivity registers are set, power management register is configured in a manner which low power consumption related states are avoided;

3. A first round of 1000 readings (both for gyroscope and accelerometer) is performed and the mean of those values is computed. The offsets are initialized with the results of this first iteration;

   (a) The z-axis accelerometer offset is computed considering that in steady position it must return $1g$, due to gravity force. So the offset is subtracted from the correspondent scale range value, which is calculated as follow.
   Example: the scale is $\pm 2g$, since registers are on 16 bits, the scale range value should be a quarter of the full range (half because the full range includes both positive and negative values and another half for scaling down from $2g$ from $1g$, so $2^{(16-2)} = 16384$);

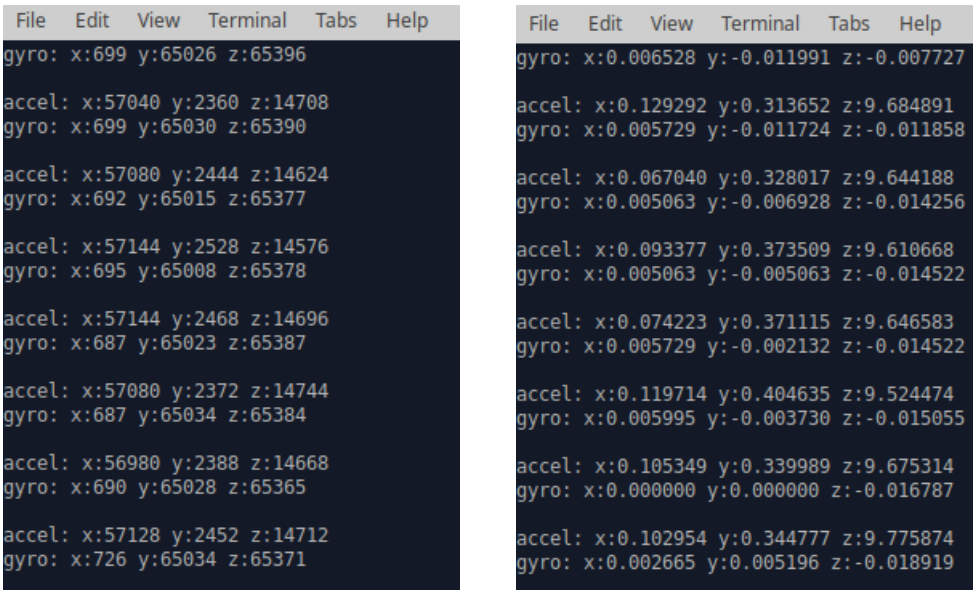
   (b) The number of readings on which the mean is determined can be changed through a `#DEFINE` (likewise all the other parameters), obviously this leads to an increasing of the time needed to the procedure to converge.

4. Then the actual iterative process starts. Each round a new mean is computed and summed up to the previous ones until a precise threshold is reached. These guard values are fixed with a `#DEFINE` and they determine the overall precision of offsets. Since the algorithm is iterative using thresholds too low could cause the procedure to not converge;

5. Finally, when the proper level is met for all registers, the loop ends. Final values are then written down in a YAML file in a such manner that the IMU node can easily import and use them at startup. This file is structured as follow:

```
#acceleration offsets
accel_offset:
    x: -727
    y: -879
    z: 2378
#gyroscope offsets
gyro_offset:
    x: -176
    y: 130
    z: 40
```

A summary of the offsets calibration algorithm (Fig.4.2) and the produced results (Fig.4.1) are presented down below:



(a) Before (b) After

Figure 4.1: IMU readings before and after offsets calibration

NB: values displayed in the second screenshot are already scaled, in accord to their gyroscope and accelerometer ranges.

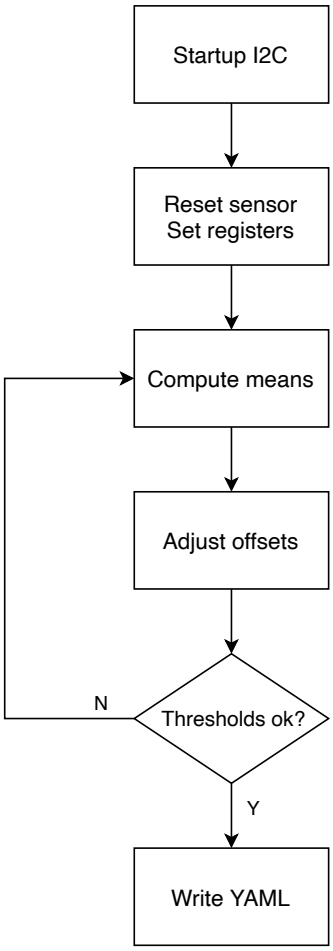


Figure 4.2: IMU offsets calibration algorithm flowchart

4.1.2 Noises

After properly estimating offsets values, there is the need to evaluate inertial sensor intrinsic noises which affect measurements. Those characteristics, are needed for the batch optimization to calibrate the camera to IMU transform and in any VINS estimator so that images and inertial readings can be correctly fused.
There are two type of coefficients necessary in order to clean properly both gyroscope and accelerometer readings:

- white noise, an additive noise term that fluctuates very rapidly;
- Random walk noise, a slowly varying sensor bias;

Some manufacturers provide inside their sensors' datasheets white noise values for fixed operating frequencies, which only needs to be scaled in accord to the actual frequency. Anyway the bias noises must be evaluated through experimental tests, such as for example the Allan deviation analysis. It works as follow:

1. First of all a rosbag is recorded with the IMU in a steady position. Since it should include enough values to estimate the noises and plot the deviations, 4 hours of measurements were taken;

2. A tool to convert a `.bag` file in a `.txt` file has been written in order to feed the Matlab script, since the provided one had dependencies problems with Ubuntu 20.04. In order to bypass this problem, the default ROS package has been modified avoiding the `.mat` conversion and the Matlab scripts changed to import the `.txt` file instead. It basically unwraps IMU messages from the bag file into tuples structured as follow:

   ```
   [timestamp accel(x, y, z) gyro(x, y, z)]
   ```

3. Before running the Matlab scripts, inside them, it must be indicated the IMU readings frequency. It is also suggested to enable the parallel toolbox since it could be a significant time consuming operation if IMU operates at high frequencies;

4. Then the scripts indicated to display the results must be executed, since the before mentioned one produced a `.mat` file with the points needed to plot the deviation;

5. As reported in Fig.4.3 [24] the two deviations plots (one for each kind of measurements) must be fitted with two lines:

   - One with a slope value of $-\frac{1}{2}$, fitted to the left side, in order to retrieve the white noise;
   - The other one with a slope of $+\frac{1}{2}$, fitted to the right side, in order to evaluate the random walk noise.

   Finally, effective values of noises are the interception at $\tau = 1$ and $\tau = 3$.

NB: Particularly when using low-cost MEMS IMUs, there could be the necessity to inflate the calculated noise model parameters by 10-20 times since the model used to evaluate the noises is optimistic and it doesn't take in account temperature variations, for example.

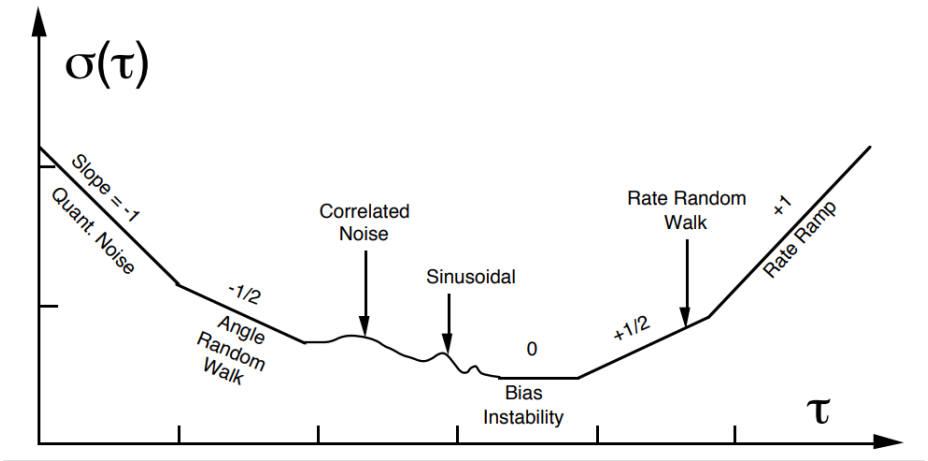


Figure 4.3: Simple plot of Allan variance analysis

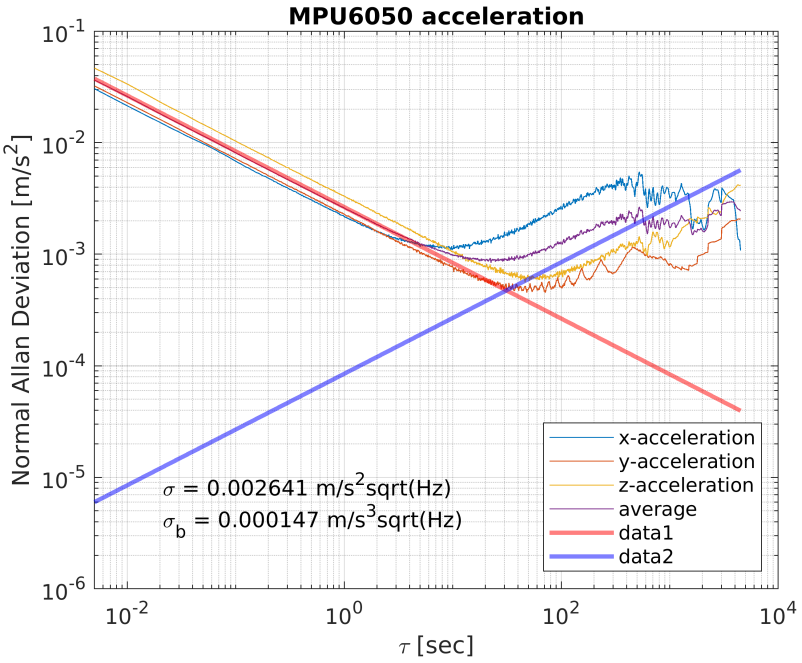


Figure 4.4: Estimated Allan deviation plot for MPU6050 accelerometer readings

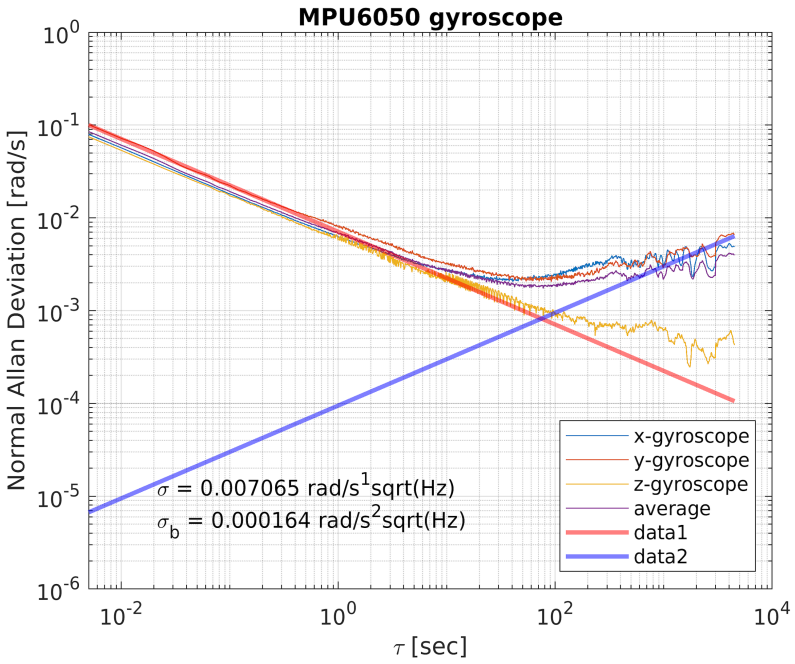


Figure 4.5: Estimated Allan deviation plot for MPU6050 gyroscope readings

Fig.4.4 and Fig.4.5 described the results for the IMU unit taken in account, those values are been increased by a factor of 15, in accord to what has been stated before.
In conclusion a `.yaml` file must be produced with the estimated noises values since it will be needed in order to complete the procedure explained in Sec.4.3:

```
#Accelerometer
accelerometer_noise_density: 0.05
accelerometer_random_walk: 0.001

#Gyroscope
gyroscope_noise_density: 0.02
gyroscope_random_walk: 4.0e-05

rostopic: /imu0 #the IMU ROS topic
update_rate: 200.0 #Hz (for discretization of noises)
```

4.2 Camera

Camera calibration is an essential component of computer vision systems and, typically, those methods extract corners from a known calibration pattern, detect the pattern and solve an optimization problem in order to estimate intrinsic and extrinsic parameters of the cameras.
The intrinsic parameters, also known as internal, are the parameters intrinsic to the camera

itself, such as the focal length and lens distortion, while the extrinsic parameters, also known as external or camera poses, are the parameters used to describe the transformation between the camera and its external world.
This section regards intrinsic parameters since extrinsic ones are treated in Sec.4.3.
The first task is to calibrate the camera intrinsic values such as focal length, camera center, and distortion coefficients through following these steps:

1. First of all Kalibr toolbox must be cloned and built. Since it provides a lot of different tools meant for cameras' calibrations, it is quite heavy and slow to be compiled. If possible, it's suggested to parallelize the procedure with `catkin` parameter `-jx`, in which `x` indicates the number of jobs launched in order to complete the compilation;

2. Then a pattern must be chosen from the moment that the whole estimation (both intrinsic and extrinsic) is based on its recognition. Usually checkerboards are used due to their black and white pattern which results in associated matrix whit easy corners features extractions. Some patterns are more indicated respect to others, not because they are easier to be recognized but for the fact that their estimators are more evolved. For example the Aprilgrid pattern (Fig. 4.6) can be detected even if framed partially or completely rotated (since each square has a different pattern inside which is associated with a number) while a simple checkerboard cannot. Afterwards

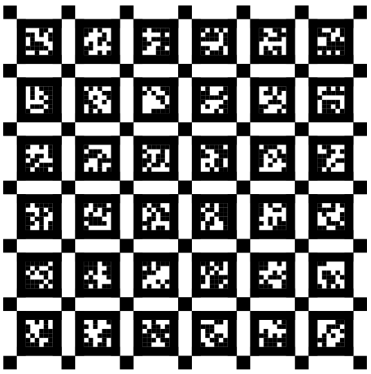


Figure 4.6: Aprilgrid calibration pattern

   the chosen pattern must be translated into a YAML file in a manner that it can be passed as parameter to the estimator, an example (for a checkerboard) could be:

```
target_type: 'checkerboard' #gridtype
targetCols: 6 #internal chessboard corners
targetRows: 9 #internal chessboard corners
rowSpacingMeters: 0.038 #size of one chessboard square [m]
colSpacingMeters: 0.038 #size of one chessboard square [m]
```

   Please note that the printed calibration target must be attached to a rigid surface in order to avoid bendings during movements which could lead to undesired and nonexistent errors estimations.

3. Next it is the turn of setting up the sensor. First of all the exposure time must be reduced as much as possible since it is the main cause of motion blur. So operating in a

good enlightened environment is vital for this and the following calibration steps. For the considered system, configuration of exposure and other sensor related parameters has been carried out thorough the camera driver interface and than exported in a `.ini` file and then imported at the startup of the ROS node.
From the moment that intrinsic parameters are being evaluated, the camera can be kept in a fixed position and could be useful ensuring that the sensor is on focus before starting the whole procedure using the Kalibr tool `kalibr_camera_focus` or manually if the lens provides a focus gear.

4. After completing the sensor parameters configuration, a ROS bag file can be recorded with a couple of precautions:

   - The ROS camera node should publish messages at a framerate between $2Hz$ and $5Hz$, in order to allow larger image variance in the dataset. If the sensor or the node don't support low framerates, ROS `message throttle` command can be used to slow down the output. It basically reduces the message output rate of a rostopic.
   - ROS messges timestamps must be consistent between each other, in accord to the chosen framerate. This can be checked automatically with the python script presented in App.E or visually with the `rqt_bag` tool.
   - The calibration target must be detected in every frame and it must cover all the sensor surface paying attention to explore the most number of possible positions in order to increase the calibration accuracy. A valid help for this process is the ROS `image_view` command for visualizing Image messages.

5. Before eventually launching calibration, the proper camera working model and its distortion model must be individuated. Kalibr supports a lot of different models, from pinhole to sphere ones, but usually most of the cameras used for this kind of application are pinhole with more or less accentuated wide-angle lenses. So in general the parameter which describes these characteristics is pinhole-equi (for lenses with more pronounced wide-angle and fisheyes) or pinhole-radtan (for low distortion lenses).

6. Calibration can be now launched, exploiting the target pattern, the working and distortion models, the bag filename and the bag camera topic name.

   ```
   kalibr_calibrate_cameras -bag [filename.bag] -topics [TOPIC] -models
   [MODEL] -target [target.yaml]
   ```

   As guessable by the command syntax is possible to calibrate more than one camera at the same time having care to indicate cameras topics and models in the correct order (TOPIC0 matches MODEL0 and so forth). Optimization can diverge right after processing the first few images due to a bad initial guess on the focal lengths. This because the estimator is based on a random selection of images, so restarting the calibration could be a solution if this kind of error occurs.
   Depending on how many frames are contained in the dataset, optimization can take on the up to few hours, this is another reason to keep the framerate low while recording the file.

7. Finally, after process termination, results must be inspected in order to grant that camera intrinsic calibration has been successfully completed. The crucial plot to analyze is the reprojection error graph which, in case of a good calibration, should highlight a final error under $0.5px$, as in Fig.4.7.
   Kalibr also provides a command tool to visually verify the calibration, it can be launched with:

   ```
   kalibr_camera_validator -cam camchain.yaml -target target.yaml
   ```

   in which the camchain file is the automatic output of the intrinsics estimation (displayed afterwords) and target is the pattern to recognize whose images are published by the topic specified in the before mentioned YAML file.
   Another way to validate the results could be re-running the calibration with a different pattern and/or dataset, since the results should be independent from the target.

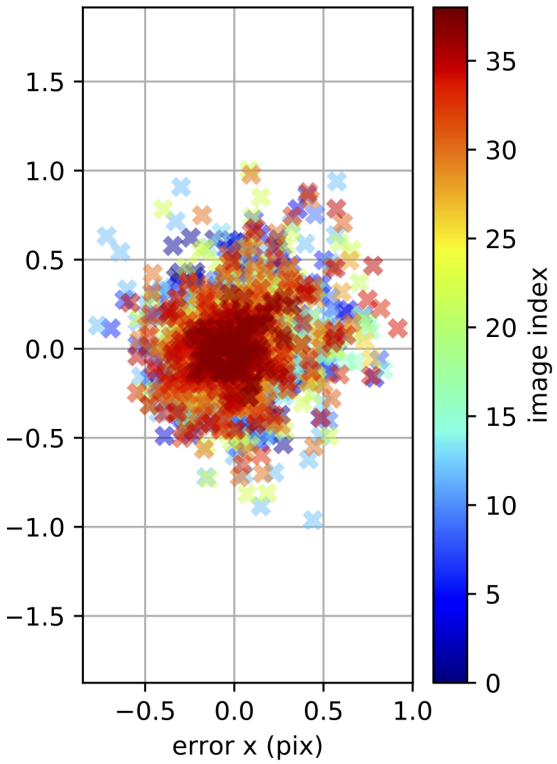


Figure 4.7: Estimated camera reprojection error plot

The output of the calibration is again a YAML file in which are exposed:

- intrinsics coefficients, the first two related to focal length while the seconds are in relation to the principal point (center of the camera);
- distortion coefficients represents radial and tangential distortion.

```
cam0:
  cam_overlaps: []
  camera_model: pinhole
  distortion_coeffs:
  - -0.18159055
  - 0.08407348
  - 0.00304879
  - 0.00361042
  distortion_model: radtan
  intrinsics:
  - 1669.25297381
  - 1675.04089023
  - 657.82144192
  - 477.38957061
  resolution:
  - 1280
  - 1024
  rostopic: /camera/image_raw
```

4.3 IMU-Camera

After obtaining the intrinsic calibration of both the camera and IMU, now dynamic calibration of the transform between the two sensors can be performed. It is called dynamic because, contrary to the previous ones which could be performed with sensors in static position, this one must be carried out by moving IMU(s) and camera(s). As with the previous procedure, since a pattern recognition must take place, it is important to minimize the motion blur in the camera while this time also ensuring that all different axes of the IMU have been excited. There is the needs to have at least one translational motion along with two degrees of orientation change for these calibration parameters to be observable, in accord to [25]. This last standing is vital in order to ensure convergence of the whole calibration.
Kalibr library simultaneously computes the homogeneous transformation between the camera and the world frame and the homogeneous transformation between IMU and the world frame. The calibration target, specified as usual through a YAML file, is used as the landmarks for the calibration procedure. Above two transformations (camera-world fame and IMU-world frame) can be used to compute the transformation between Camera and IMU. In this method the IMU is parameterized as 6x1 spline using the three degrees of freedom for translation and other three for orientation. Based on the raw sensor acceleration and angular velocity readings from the IMU, the acceleration and the angular velocity is computed in terms of IMU-world frame transformation. So, this library generates the following output:

1. transformation between the camera and the IMU;
2. offset between camera time and IMU time (synchronization time);
3. the pose of the IMU w.r.t to world frame;

4. Intrinsic camera calibration matrix, K for the camera.

A complete description of the transformation evaluation procedure can be found at [26].

In more practical terms the steps to follow are:

1. Points 1 and 2 of Sec.4.2 are taken for granted;
2. For what concern sensors setup, it is crucial that the position of IMU and camera is definitive, the one will be used during tests phase, otherwise a new extrinsics evaluation should be obviously performed. While the arguments about motion blur, focus and camera parameters are still valid, different is for sensors operating rates, since the bag file must be recorded with these modalities:
   - the camera node should be publish between $20Hz$ and $30Hz$ while the IMU between $200Hz$ and $500Hz$;
   - ROS messges timestamps must be consistent between each other, possibly strictly synchronized without any message drop, in accord to the chosen framerate. This can be checked automatically with the python script presented in App.E or visually with the `rqt_bag` tool.
   - the calibration target must be detected in every frame and all the six axis (x, y, z, pitch, roll and yaw) of the IMU must be excited in order to obtain a correct result. Fast movements and shocks must be avoided because they can produce wrong estimations and boundaries violations. As usual, a valid help for this process is the ROS `image_view` command for visualizing Image messages.
   - the bag shouldn't last more than 60-90 seconds and optionally, before stopping the bag record, some random movements could be recorded having always in mind that computation time increases rapidly with the number of frames.
3. Calibration can be now launched, exploiting the target pattern, the bag filename and the YAML files for IMU and camera intrinsics

   ```
   kalibr_calibrate_imu_camera -target target.yaml -cam camera_i.yaml -imu
   imu_i.yaml -bag rosbag.bag
   ```

   Common errors are due to bad topic synchronization, which can cause the optimization fails, and inappropriate movements at startup or at the end of the bag file. Kalibr provides two optional parameters of the command `-bag-from-to` and `-timeoffset-padding` which tackle those two problems. The first allows to cut initial and final seconds of the ROS bag file, while the second relaxes time constraint between IMU and camera timestamps. It is also available a flag (`-no-time-calibration`) that bypasses completely the synchronization check, obviously degrading the overall calibration accuracy.
   Depending on how many frames are contained in the dataset, optimization can take up to half of a day.

4. Finally, after process termination, results must be inspected in order to grant their correctness. In particular two aspects are important, both for gyroscope and accelerometer:

   - the spline fitted to the inertial readings follows correctly measurements progress;
   - estimated bias should not leave their 3-sigma boundaries.

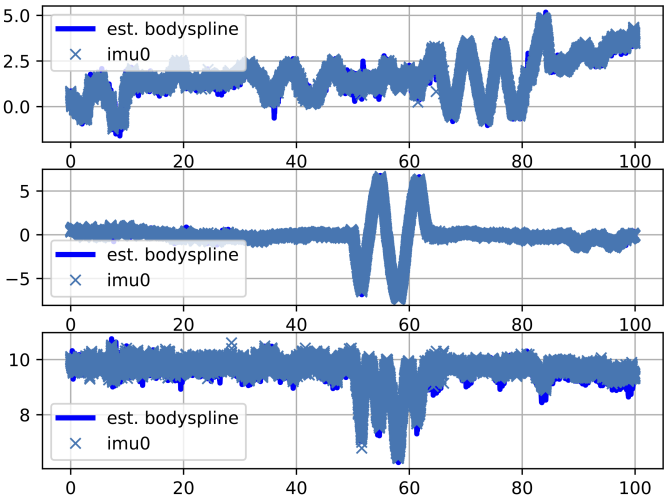


Figure 4.8: Comparison of predicted and measured accelerations

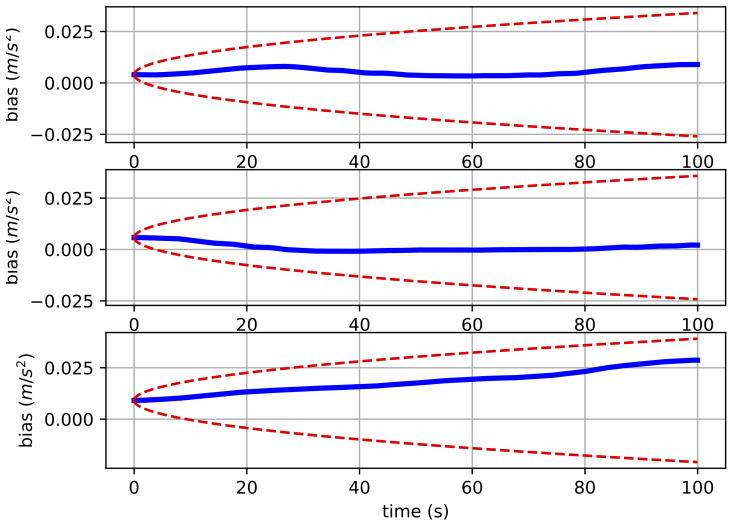


Figure 4.9: Estimated accelerometer bias

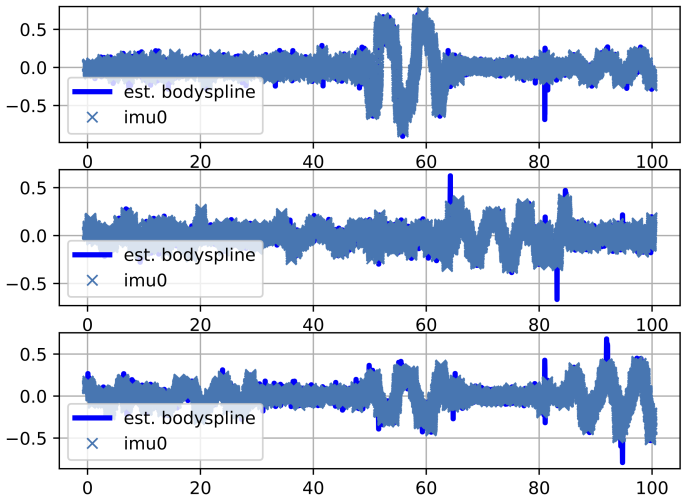


Figure 4.10: Comparison of predicted and measured angular velocities

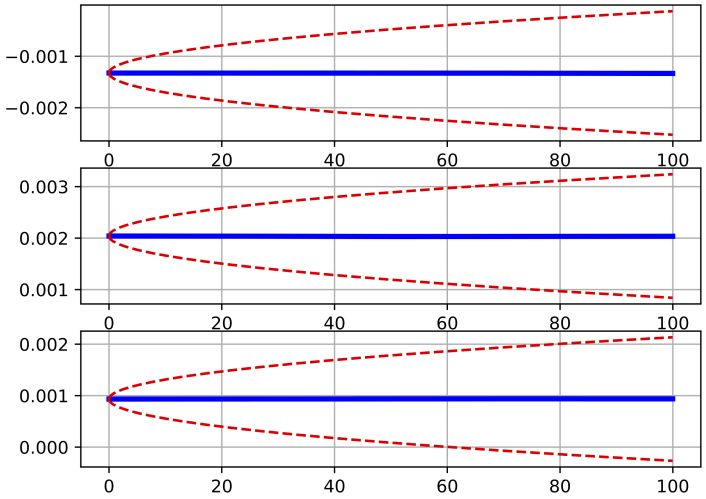


Figure 4.11: Estimated gyroscope bias

Extrinsics parameters have been retrieved directly from the calibration outputs and inserted in the associated parameters spot inside the launch file, used during performed tests, as is presented in the following Chapter.

Chapter 5

Integration and Tests

5.1 Framework Integration

The integration of the preovious calibration steps is quite straightforward since it is sufficient fill a `.launch` file with the intrinsics, extrinsics and noises values. It is also necessary to specify camera and IMU topics in addition to the kind of mskf node due to the fact that is possible to use the launch file on pre-recorded bag files.
Down below is listed the file used for real time tests, followed by a brief parameter description.

```
<launch>

    <!-- mono or stereo -->
    <arg name="max_cameras" default="1" />
    <arg name="use_stereo" default="false" />

    <!-- imu startup thresholds -->
    <arg name="init_window_time" default="0.75" />
    <arg name="init_imu_thresh" default="0.75" />

    <!-- saving trajectory, timing and utils information -->
    <arg name="dosave" default="true" />
    <arg name="dotime" default="true" />
    <arg name="path_est" default="/home/rock/Desktop/traj_estimate.txt" />
    <arg name="path_time" default="/home/rock/Desktop/traj_timing.txt" />
    <arg name="path_util" default="/home/rock/Desktop/psutil_log.txt" />

    <!-- MASTER NODE! -->
    <node name="run_subscribe_msckf" pkg="ov_msckf" type="
        ↪ run_subscribe_msckf" output="screen" clear_params="true"
        ↪ required="true">

        <!-- sensors topics -->
```

```
<param name="topic_imu" type="string" value="/imu" />
<param name="topic_camera0" type="string" value="/infra1/
    ↪ image_rect_raw" />
<param name="topic_camera1" type="string" value="/infra2/
    ↪ image_rect_raw" />

<!-- world/filter parameters -->
<param name="use_fej" type="bool" value="true" />
<param name="use_imuavg" type="bool" value="true" />
<param name="use_rk4int" type="bool" value="true" />
<param name="use_stereo" type="bool" value="$(arg use_stereo)" />
<param name="calib_cam_extrinsics" type="bool" value="true" />
<param name="calib_cam_intrinsics" type="bool" value="true" />
<param name="calib_cam_timeoffset" type="bool" value="true" />
<param name="calib_camimu_dt" type="double" value="0.0" />
<param name="max_clones" type="int" value="11" />
<param name="max_slam" type="int" value="25" />
<param name="max_slam_in_update" type="int" value="25" /> <!-- 25
    ↪ seems to work well -->
<param name="max_msckf_in_update" type="int" value="50" />
<param name="max_cameras" type="int" value="$(arg max_cameras)" />
<param name="dt_slam_delay" type="double" value="3" />
<param name="init_window_time" type="double" value="$(arg
    ↪ init_window_time)" />
<param name="init_imu_thresh" type="double" value="$(arg
    ↪ init_imu_thresh)" />
<rosparam param="gravity">[0.0,0.0,9.81]</rosparam>
<param name="feat_rep_msckf" type="string" value="GLOBAL_3D" />
<param name="feat_rep_slam" type="string" value="
    ↪ ANCHORED_FULL_INVERSE_DEPTH" />
<param name="feat_rep_aruco" type="string" value="
    ↪ ANCHORED_FULL_INVERSE_DEPTH" />

<!-- timing statistics recording -->
<param name="record_timing_information" type="bool" value="$(arg
    ↪ dotime)" />
<param name="record_timing_filepath" type="string" value="$(arg
    ↪ path_time)" />

<!-- tracker/extractor properties -->
<param name="use_klt" type="bool" value="true" />
<param name="num_pts" type="int" value="150" />
<param name="fast_threshold" type="int" value="15" />
<param name="grid_x" type="int" value="5" />
<param name="grid_y" type="int" value="3" />
<param name="min_px_dist" type="int" value="10" />
```

```
<param name="knn_ratio" type="double" value="0.70" />
<param name="downsample_cameras" type="bool" value="false" />
<param name="multi_threading" type="bool" value="true" />

<!-- aruco tag/mapping properties -->
<param name="use_aruco" type="bool" value="false" />
<param name="num_aruco" type="int" value="1024" />
<param name="downsize_aruco" type="bool" value="true" />

<!-- sensor noise values / update -->
<param name="up_msckf_sigma_px" type="double" value="1" />
<param name="up_msckf_chi2_multipler" type="double" value="1" />
<param name="up_slam_sigma_px" type="double" value="1" />
<param name="up_slam_chi2_multipler" type="double" value="1" />
<param name="up_aruco_sigma_px" type="double" value="1" />
<param name="up_aruco_chi2_multipler" type="double" value="1" />
<param name="accelerometer_noise_density" type="double" value
   ↪ ="1.0000e-1" />
<param name="accelerometer_random_walk" type="double" value="3.0000
   ↪ e-3" />
<param name="gyroscope_noise_density" type="double" value="1.6968e
   ↪ -02" />
<param name="gyroscope_random_walk" type="double" value="1.9393e
   ↪ -03" />


<!-- camera intrinsics -->
<!-- camera0 -->
<rosparam param="cam0_wh">[640, 480]</rosparam>
<param name="cam0_is_fisheye" type="bool" value="false" />
<rosparam param="cam0_k">[383.85249559594035, 384.3352365230173,
   ↪ 322.1176348582573, 233.01086196458934]</rosparam>
<rosparam param="cam0_d">[0.003707495533387474,
   ↪ -0.0009775064778847408, 0.00031122479401325253,
   ↪ 0.0022638424406707844]</rosparam>
<!-- camera1 -->
<rosparam param="cam1_wh">[640, 480]</rosparam>
<param name="cam1_is_fisheye" type="bool" value="false" />
<rosparam param="cam1_k">[383.85249559594035, 384.3352365230173,
   ↪ 322.1176348582573, 233.01086196458934]</rosparam>
<rosparam param="cam1_d">[0.003707495533387474,
   ↪ -0.0009775064778847408, 0.00031122479401325253,
   ↪ 0.0022638424406707844]</rosparam>

<!-- camera extrinsics -->
<!-- camera0 to IMU transformation -->
```

```
        <rosparam param="T_C0toI">
            [
             0.99991533, 0.00319211, 0.01261552, -0.02904195,
            -0.00312359, 0.99998028, -0.0054475, 0.00700187,
            -0.01263266, 0.00540764, 0.99990558, 0.01537354,
            0.0, 0.0, 0.0, 1.0
            ]
        </rosparam>
        <!-- camera1 to IMU transformation -->
        <rosparam param="T_C1toI">
            [
            0.99993621, 0.0033249, 0.01079478, 0.06487232,
            -0.00327224, 0.99998268, -0.00489186, 0.00676284,
            -0.01081086, 0.00485622, 0.99992977, 0.01486894,
            0.0, 0.0, 0.0, 1.0
            ]
        </rosparam>
    </node>

    <!-- graphical output node -->
    <node type="rviz" name="rviz" pkg="rviz" args="-d $(find ov_msckf)/
        ↪ launch/display.rviz" />

    <group if="$(arg dosave)">
        <!-- record util for post run analysis -->
        <node name="recorder_timing" pkg="ov_eval" type="pid_ros.py" output
            ↪ ="screen">
            <param name="nodes" type="str" value="/run_subscribe_msckf"/>
            <param name="output" type="str" value="$(arg path_util)"/>
        </node>

        <!-- record the trajectory for post run analysis -->
        <node name="recorder_estimate" pkg="ov_eval" type="pose_to_file"
            ↪ output="screen" required="true">
            <param name="topic" type="str" value="/ov_msckf/poseimu" />
            <param name="topic_type" type="str" value="
                ↪ PoseWithCovarianceStamped" />
            <param name="output" type="str" value="$(arg path_est)" />
        </node>
    </group>

</launch>
```

Parameter	Description
bag topics	ROS topics that we will parse the IMU and camera data from, even if it is launched in real time or in relation to a bag file. If the stereo flag is false then only the first camera topic is used.
bag params	Location of the bag we will read along with the start time, in seconds, and duration we want to run on.
world/filter params	This has most of the core parameters that can be tuned to improve filter performance including the sliding window size, representation, gravity, and number of environmental SLAM features.
tracker/extractor params	For visual front-end tracker we have a few key parameters that can tuned, most importantly is the number of features extracted.
sensor noise values	Feature measurement function is on the raw pixels, the pixel noise should be 1 pixel if the calibration was performed correctly. Continuous time white noise and random walk values for IMU must be specified in accord to the previous estimation.
camera intrinsics	Camera intrinsic values retrieved from the previous calibration.
camera extrinsics	Camera extrinsics values retrieved from the previous calibration.
post processing files	Files needed for post running analysis (carried out through `ov_eval`) utility which can be activated with `dosave` and `dotiming` parameters.

Table 5.1: OpenVINS launch file parameters overview

After creating and positioning in the appropriate folder the launch file, the system can be finally run through the following command:

```
roslaunch ov_msckf ov.launch
```

Once validated the correctness of launch file, IMU measurements are read until reaching a determined threshold, during a time window (specified in the file and empirically evaluated considering the environment, IMU and overall test). Then a rviz terminal is prompted in which pose estimation in real time is shown, presented in Fig.5.6 and Fig.5.7.
NB: it is necessary have sourced both ROS environment and `devel\`, folder similarly with what has been done for Calibration.

5.2 Onboard hardware integration

After the integration of software parameters, the board and the camera needed to be placed on the drone. Fortunately the MAVs available disposed all of a standard 3D printed payload carry system inside which dispose camera and board to run flight test (completely

independent with flight controller and all the electronics needed to operate the quad-copter except for power source). It also integrates four rubber vibration dampers in proximity of junction with drone frame. The only challenge was to fit the Rock Pi board inside the payload holder since the holes didn't match.
Down below is reported also the first configuration (the one which drifted at startup due to hardware quality issues as stated in Chapter 3, Fig.5.2 a) for completeness. It is also an opportunity to demonstrate the flexibility of payload holder systems also for evaluation of different hardware as well as different software.
For example, regarding the first configuration, a custom camera support was required in order to integrate it on payload holder. It was modeled referring to an old support, from which the idea of honeycomb structure in order to maintain low weight. But also adding some extra feature like the additional screw flap which can grant a 45deg rotation in order to avoid interferences between camera and drone's legs. So, taking advantage of the availability of a 3D printer in TAS-I laboratory and with few rudiments of modeling, I was able to create a custom support within a working day. The second configuration didn't needed any support since camera screw holes matched perfectly the preexistent holes on payload holder.

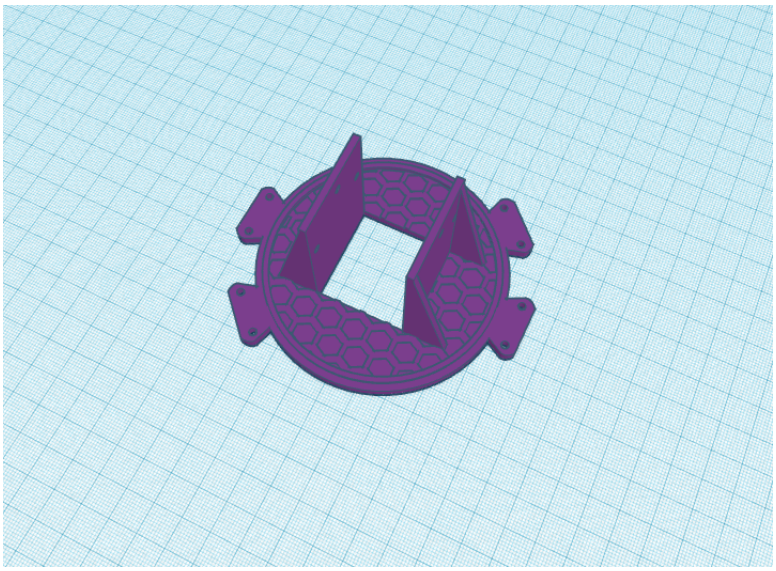
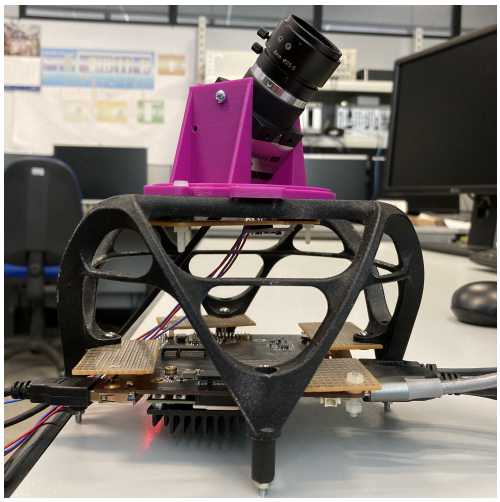


Figure 5.1: Camera support model for payload holder



(a) First payload (EO+MPU6050)

(b) Definitive payload (Realsense)

Figure 5.2: The two used different hardware configurations

Another important aspect is related to power supply, since the board and so implicitly the sensors must be alimented through drone's batteries. In our case this was possible via board's USB type-C port and a standard cable with power delivery capabilities (very important, cables meant for data transport weren't able to deliver the correct power) with one terminal modified in order to connect correctly to battery pack.
Down below is shown a brief integration schema, with a focus on payloads. Drone components, which consist in flight controller, brushless motors, radio receiver, power regulators and batteries, are not included since are canonical and not particularly relevant for this step.

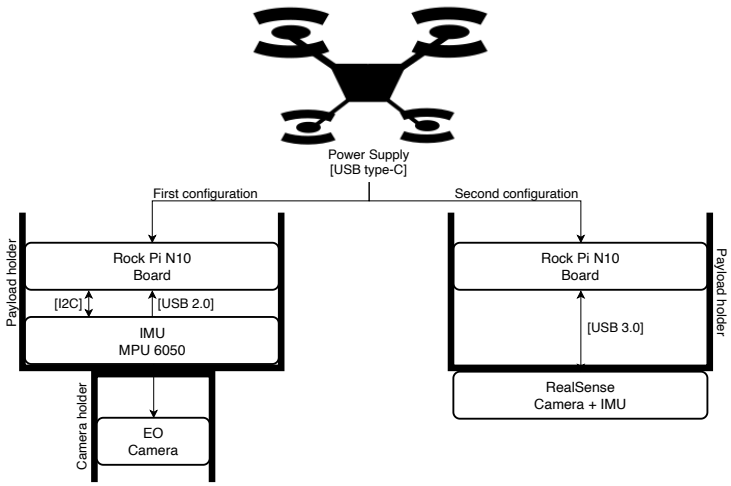


Figure 5.3: Payload integration diagram

After the completion of all integration steps, it is presented the drone inside Martian field, with payload on board, ready to be tested and fly.

Figure 5.4: Payload mounted on drone inside the Martian field

5.3 Tests Description

Tests phase can be divided in three sub-phases:

1. laboratory tests;
2. manual facility tests;
3. facility tests with the drone pilot.

The first batch of tests, the laboratory ones, were carried out in order to validate the calibration processes and results. This because we had to be sure to have a proper running environment before moving the tests on the Martian field, in order to avoid useless wastes of time. This set of tests consisted in moving around the laboratory assuring that the system correctly initiate itself and any configuration error was raised.
Successfully completed the first phase, all the devices were transported to the facility for on field tests. From the moment that the drone pilot was a person external to TAS we had to be sure that one present the system worked properly. So, before make the drone fly a set of manual test took place. For manual is intended that the drone was moved manually above the martial field in order to evaluate and adjust camera parameters like exposure and white balance. This step was crucial since it let us familiarize with the test environment and optimize the time available with the pilot.
Finally with the cooperation of a licensed drone pilot we ran the final tests, trying to cover all the Martian field in its width and depth. Firstly following the field border and then flying in circle above the zones (known thanks to the previous phase) were feature extraction worked properly. There were also area where the flatness of the ground, simulating martian deserts, led the algorithm to drift since their poor extraction points.

Figure 5.5: A moment during second batch tests

Further it is presented and analyzed one of the most representative test.
After test launch and correct initialization, the Human Computer Interface (see Fig.5.6) presents four different panels:

1. Top left: graphics user interfaces parameters;
2. Bottom left: live camera features and depths detections (Fig.5.7 a);
3. Right: pose estimation, features extraction with path tracking, in three dimensions (Fig.5.7 b).

The colors in the picture indicate whether a feature track contains stereo or monocular data. Blue indicates that it contains a stereo track in the other picture, while red indicates that it is only a monocular track. Green squares should also be present, indicating SLAM features that are continually monitored for as long as feasible.

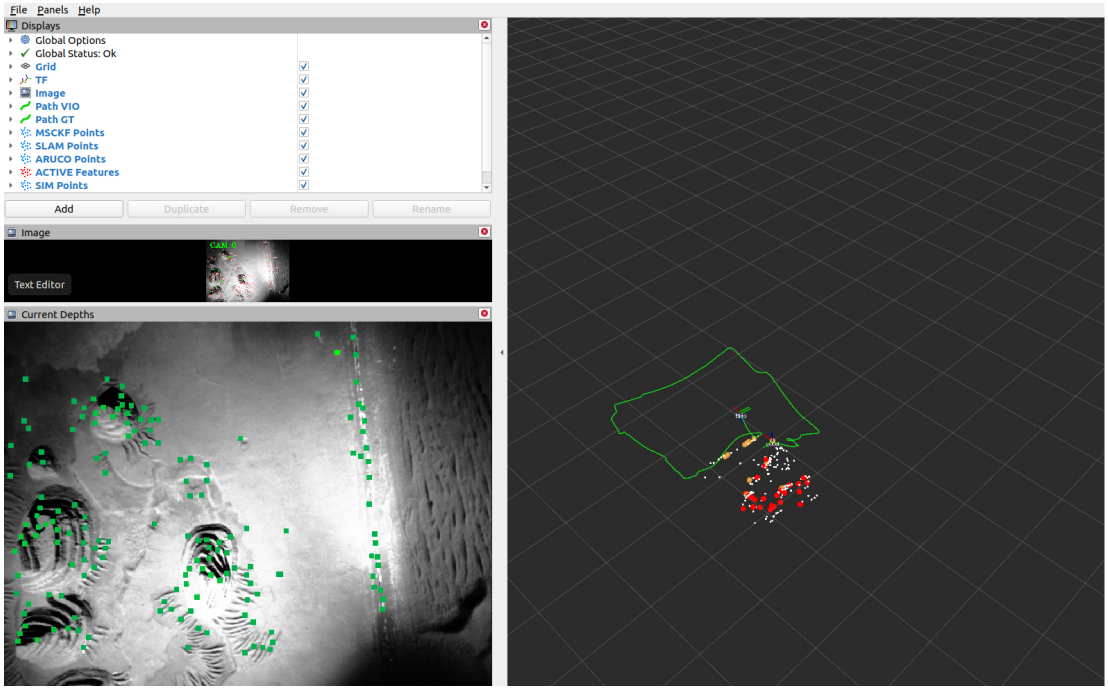
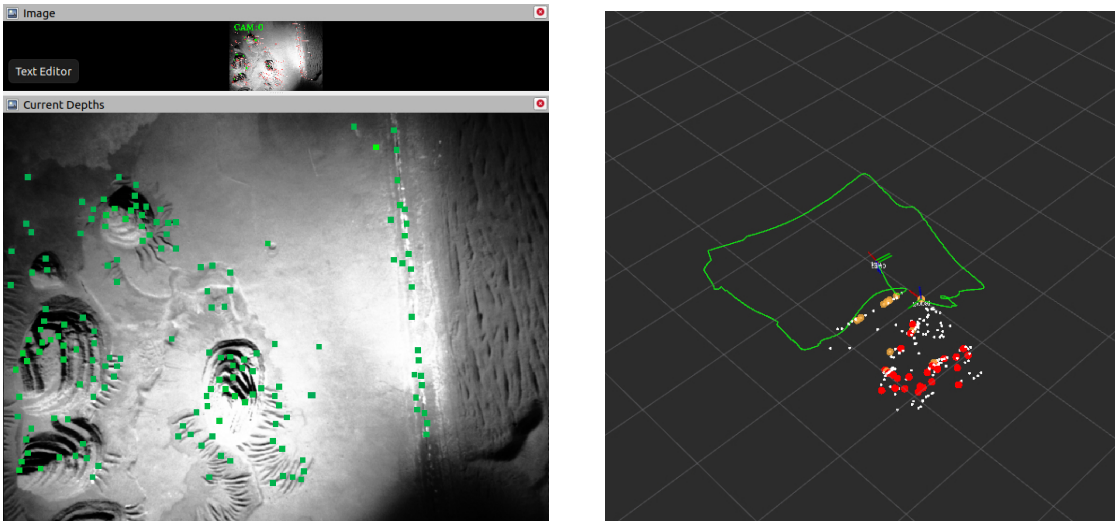


Figure 5.6: Framework interface overview



(a) Features detection over real time camera image

(b) Pose estimation with path record in 3D

Figure 5.7: Framework interface details

Unfortunately we weren't able to explore the height since facility's illumination wasn't adapt for the purpose and a modification didn't match the thesis timings. This problem was related to drone's shadow which, increasing the altitude, became more and more

dominant with respect to landscape morphology, fundamental to extract features. Possible solutions to this inconvenient could be of course change the facility illumination as stated before or perform the tests in the near martian field (used mainly for rovers) which has a higher reproduction object scale.

5.4 Analysis

5.4.1 Estimated Trajectories

The initial stage in any evaluation is to gather the proposed systems' predicted trajectory. The goal is to capture the estimate at the current timestep since we're interested in robotic application of different estimators (as compared to a "smoothed" output or one that includes loop-closures from future timesteps). This means that it is sufficient to publish the current estimate at the current timestep in the ROS framework. For this purpose `ov_eval` is recommended: the estimator output may be recorded straight into a text file using the correct node. PoseWithCovarianceStamped, PoseStamped, TransformStamped, and Odometry topics are supported.

```
<node name="recorder\_estimate" pkg="ov\_eval" type="pose\_to\_file"
    ↪ output="screen">
<param name="topic" type="str" value="/ov\_msckf/poseimu" />
<param name="topic\_type" type="str" value="PoseWithCovarianceStamped" />
<param name="output" type="str" value="/home/user/data/traj\_log.txt" />
</node>
```

The attention now shifts to processing and visualizing the data, which has been captured and formatted correctly. The data is first transformed into a collection of output text files, which the user may then use to plot the findings in their preferred software or language. All of the following instructions can have an align mode of posyaw, posyawsingle, se3, se3single, sim3, or none. The data will be plotted as a 2d xy and z-time position plot using the follow command.

```
rosrun ov_eval plot_trajectories <align_mode> <file_gt.txt> ...
<file_est9.txt>
```

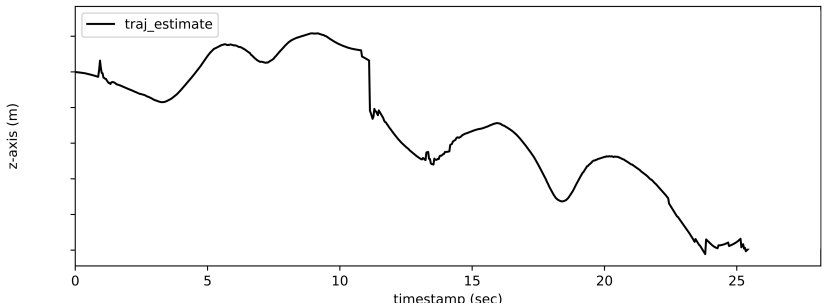


Figure 5.8: z time-positions plot

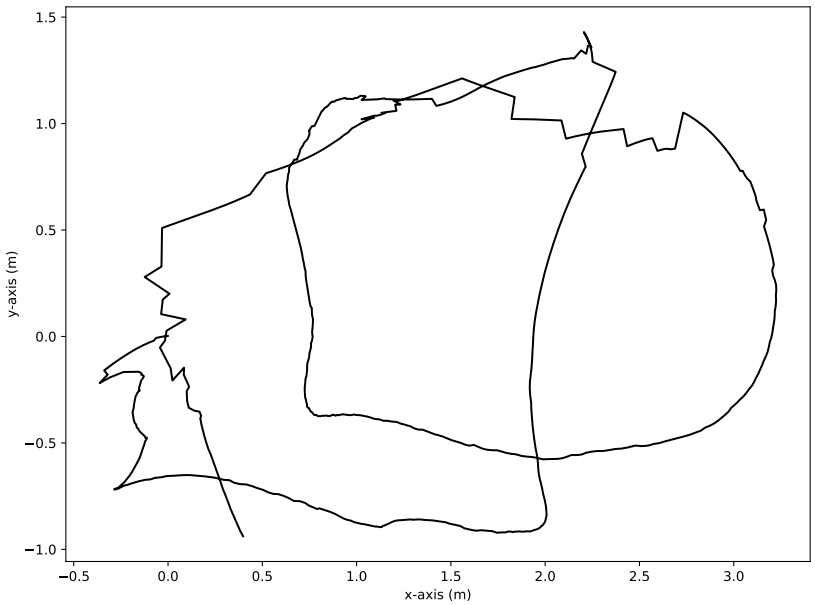


Figure 5.9: xy time-positions plot

The trajectories estimated after the considered flight were actually good and consistent with the path agreed with the drone pilot, both for xy and z plots. It is possible to notice a bit of drift, in particular during the second part of the flight since there was a moment in which the tracked features were insufficient to correctly estimate pose. For this kind of analysis which require an elevate grade of accuracy will be fundamental implement a mechanism for ground truth paths generation for trajectory comparison. The TAS facility has already a system meant for the position detection of an object, better presented in the Further Development section in Conclusion chapter.

5.4.2 Timings

To profile the different parts of the system we record the timing information from directly inside the ov_msckf. The file should be comma separated format, with the first column being the timing, and the last column being the total time (units are all in seconds). The middle columns should describe how much each component takes (whose names are extracted from the header of the csv file). You can use the bellow tools as long as you follow this format, and add or remove components as you see fit to the middle columns.
A python script that uses the psutil python package to record CPU and memory percentages uses is provided to analyze the computational load (not computation time). This may be added to the launch file as an extra node that just requires the node for which information is required. This will poll the node for memory and CPU use percentages, as well as the total number of threads. It's beneficial for comparing various approaches on

the same platform, but it's not appropriate for comparing the performance of the same or different algorithms across different hardware devices.

```
<node name="recorder\_timing" pkg="ov\_eval" type="pid\_ros.py" output="
    ↪ screen">
<param name="nodes" type="str" value="/feature\_tracker,/vins\_estimator,/
    ↪ pose\_graph" />
<param name="output" type="str" value="/tmp/psutil\_log.txt" />
</node>
```

It's also worth noting that if the estimator contains many nodes, you may subscribe to all of them by separating their names with a comma.
The flame graph script attempts to recreate a FlameGraph of the system's essential components. While not all functions are traced, the important "top level" function timings are saved to a file to provide insight into what consumes the majority of the calculation time. The file should be in a comma-separated format, with the timing in the first column and the total time in the last column. The center columns should state the amount of time each component takes (whose names are extracted from the header of the csv file).

```
rosrun ov_eval timing_flamegraph <file_times.txt>
```

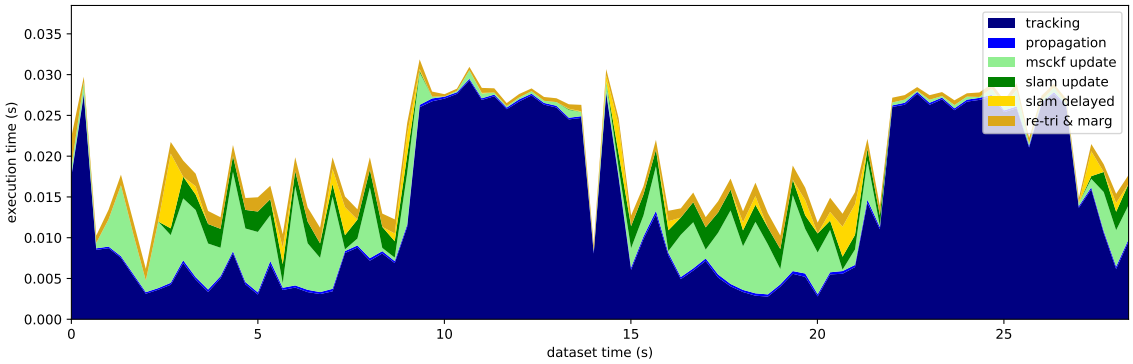


Figure 5.10: Timing flamegraph output

From the graph, relative to one of our flight, it evinces that the operations more execution time consuming are the traking and the msckf update. In order to obtain stable and precise estimations those continuous load peaks should be avoided, reducing the number of features (with the risk of lower the estimation quality) or switching to a more powerful hardware. The last option was our case, since lowering the number of features caused a degradation of estimation and a nonrecoverable drift of the system.

Chapter 6

Conclusions

6.1 Algorithm

While there have been tremendous advancements in VINS over the last decade, there are still numerous obstacles to overcome:

Distributed cooperative VINS : although collaborative VINS have been investigated in the past, developing real-time distributed VINS, such as for crowd sourcing activities, remains difficult. Recent research on cooperative mapping may provide some insight on how to address this issue, such as the autonomous flight in swarms [27];

Semantic localization and mapping : although existing VINS largely employ geometric features such as points, lines, and planes for localization, these handcrafted characteristics may not be the greatest for navigation, and it is critical to be able to discover the appropriate features for VINS using recent breakthroughs in deep learning. Furthermore, a few recent research attempts have tried to provide VINS with semantic awareness of surroundings, which is yet underdeveloped but has huge promise;

Persistent localization : while current VINS can accurately track 3D motion in small-scale contexts, they are not robust enough for long-term, large-scale, safety-critical deployments, such as autonomous driving, due to resource limits. As a result, enabling persistent VINS even in difficult settings (such as bad lighting and motions) is problematic, for example, by effectively incorporating loop closures or creating and employing innovative maps;

Extensions with other sensors : other aiding sensors may be more appropriate for certain environments and different kind of motions. Acoustic sonars are perfect for underwater areas; LiDARs may work better in environments with poor lighting conditions; and event cameras may capture dynamic motions better.

High-dimensional object tracking : in addition to increasing localization precision, it is frequently important to recognize, represent, and track moving objects that co-exist in the same place in real time while travelling in dynamic complex situations, such as 3D object tracking in autonomous navigation;

6.2 Future Developments

In relation to this specific thesis work, the usage of an open source and modular frameworks, as adopted in my thesis work, allows a lot of further improvements.
The first implementation steps should regard autonomous take off and landing procedure. This because, at the moment, the take off mechanism works with an initial guessing which is corrected during first seconds of flight. If this adjustment isn't performed correctly it can lead to drift and disruptive situations with catastrophic scenarios in case of a critical mission. A possible solution could be the integration with machine learning algorithms trained on previous starting conditions or implementing a mechanism to reversing information obtained with landing considering also atmospheric condition changes.
As well landing is a fundamental stage of the flight since drone are battery powered and those cannot be recharged during use. So the selection of a safe spot where touchdown is vital in order to refill batteries through solar panels (likewise Ingenuity) without compromising the vehicle's safeness. Possible solutions are the usage of a down pointing camera used maybe in conjunction with hazard and collision avoiding system and a laser. Top solution infrared camera with cloud points projection, extra computational power for available site detection, evaluation and selection. Quite surely both landing and take off procedure will need the activation of an extra thread because the main one will be still engaged in maintain the drone attitude increasing and decreasing altitude.
As mentioned before another important features to implement could be an hazard and collision avoiding system which relies on an active detection. This kind of step doesn't appear crucial yet in relation to current exploration missions on former planets since those doesn't have reached this level of complexity. Surely these methodologies could have an important impact on safety in relation to mission vitality and surviving. Enhancements related to extra sensors should be quite simple to include into the framework, since it has been deployed with this purpose, it has to be figured out the data fusion methodology and the weight of these new metrics with respect to the already present ones.
Finally In order to improve the development, the integration and the test of new algorithms and features it could be useful integrates the VICON [28] system (which is already present in TAS-I Turin Facility). It basically consists in a large number of infrared cameras place all around the cage which track in a very accurate way the motion of 9 reflective spheres placed on the protective foam around the drone. OpenVINS provides an experimental git repository which allows to generate the path of tracked object (in this case the autonomous drone) starting from VICON readings. The result of this process leads to the construction of ground-truth paths useful to determine algorithm's quality.

Appendix A

IMU Node

Listing A.1: main

```
/*
 * Author: Matteo Accornero
 *
 * Example usage: rosrun mpu6050 mpu6050_node _synchro:=[clock, timer]
 */

#include "MPUWrapper.hpp"


int main(int argc, char **argv)
{
    ros::init(argc, argv, "mpu6050");

    MPUWrapper* mw = new MPUWrapper();
    mw->startPublish();

    ros::spin();

    delete mw;
    ros::shutdown();

    return 0;
}
```

Listing A.2: header

```
#ifndef MPUWRAPPER_HPP
#define MPUWRAPPER_HPP

/* standard headers */
#include <iostream>
```

```
#include <csignal>
#include <math.h>
#include <chrono>
#include <thread>
#include <endian.h>

/* mraa headers */
#include "mraa/common.hpp"
#include "mraa/i2c.hpp"

/* yaml headers */
#include "yaml-cpp/yaml.h"

/* ROS headers */
#include <ros/ros.h>
#include <sensor_msgs/Imu.h>
#include <rosgraph_msgs/Clock.h>

/*
 * I2C buses definition
 * 0 = I2C7
 * 1 = I2C2
 * 2 = I2C6
 */
#define I2C_BUS 1

/* register definitions */
#define MPU6050_ADDR 0x68
#define MPU6050_REG_PWR_MGMT_1 0x6b
#define MPU6050_REG_RAW_ACCEL_EP 0x3b
#define MPU6050_REG_RAW_GYRO_EP 0x43

/* acceleration offsets registers */
#define MPU6050_REG_OFF_ACCEL_X 0x06
#define MPU6050_REG_OFF_ACCEL_Y 0x08
#define MPU6050_REG_OFF_ACCEL_Z 0x0A

/* gyroscope offsets registers */
#define MPU6050_REG_OFF_GYRO_X 0x13
#define MPU6050_REG_OFF_GYRO_Y 0x15
#define MPU6050_REG_OFF_GYRO_Z 0x17

#define MPU6050_REG_BLOCK_SIZE 6

/* bit definitions */
#define MPU6050_RESET 0x80
```

```
#define MPU6050_SLEEP (1 << 6)
#define MPU6050_PLL_GYRO_X (1 << 1)

/* accelerometer scale factor for (+/-)2g */
#define MPU6050_ACCEL_SCALE 16384.0

/* gyroscope scale factor for (+/-)250/s */
#define MPU6050_GYRO_SCALE 131.0

/* g (gravity) for Imu message's linear acceleration conversion
   (it must be in m/s^2, see documentation) */
#define G 9.807

/* conversion factor degrees to rad for Imu message's angular velocity
    ↪ conversion
   (it must be in rad/s, see documentation) */
#define DEG_TO_RAD 0.0174533

/* mpu_config.yaml offsets input file */
#define PATH_CFG_FILE "/home/rock/Documents/Workspace/catkin_ws_sensors/
    ↪ src/imu/config/offsets.yaml"

/* ROS node publishing topic name */
#define ROS_IMU_TOPIC "/imu0"

/* ROS node publishing frequency [Hz] */
#define DEFAULT_HZ 200

/* ROS imu messages frame id */
#define MPU_FRAMEID "base_imu"

/* ROS node param name */
#define NODE_PARAM_NAME "synchro"
#define NODE_PARAM_TIMER "timer"
#define NODE_PARAM_CLOCK "clock"

class MPUWrapper {

   private:
    mraa::I2c* i2c;
    ros::NodeHandle* node;
    ros::Publisher pub;
    ros::Subscriber sub;  // needed for clk mode
    ros::Timer timer;
    sensor_msgs::Imu msg;
    bool mode;    // [true: timer, 0: clock]
```

```
    void initI2C();
    void loadMPUConfig();
    void initROSNode();
    void readData();
    void timerCallback(const ros::TimerEvent& event);
    void clockCallback(const rosgraph_msgs::Clock::ConstPtr& clkMsg);

  public:
    MPUWrapper();
    ~MPUWrapper() { delete i2c; delete node; }
    void startPublish();

};

#endif // MPUWRAPPER_HPP
```

Listing A.3: functions

```
#include "MPUWrapper.hpp"

MPUWrapper::MPUWrapper()
{
    initI2C();
    loadMPUConfig();
    initROSNode();
}

void MPUWrapper::initI2C()
{
    //mraa::Result status = mraa::SUCCESS;
    uint8_t data;
    int ret, status = 0;

    i2c = new mraa::I2c(I2C_BUS);

    /* set slave address */
    status |= i2c->address(MPU6050_ADDR);

    /* reset the sensor */
    status |= i2c->writeReg(MPU6050_REG_PWR_MGMT_1, MPU6050_RESET);

    /* configure power management register */
    ret = i2c->readReg(MPU6050_REG_PWR_MGMT_1);

    data = ret;
    data |= MPU6050_PLL_GYRO_X;
```

```
    data &= ~(MPU6050_SLEEP);

    status |= i2c->writeReg(MPU6050_REG_PWR_MGMT_1, data);

    if (status != 0) {
        // dummy, TODO
        std::cout << "[ERROR] Something went wrong while setting up I2C and
            ↪  MPU" << std::endl;
    }

    std::cout << "I2C MPU connection successfully created!" << std::endl;
}

void MPUWrapper::loadMPUConfig()
{
    /* offset calibration */
    try {
        YAML::Node mpu_cfg = YAML::LoadFile(PATH_CFG_FILE);
        i2c->writeWordReg(MPU6050_REG_OFF_ACCEL_X, (uint16_t)htobe16(
            ↪ mpu_cfg["accel_offset"]["x"].as<int>()));
        i2c->writeWordReg(MPU6050_REG_OFF_ACCEL_Y, (uint16_t)htobe16(
            ↪ mpu_cfg["accel_offset"]["y"].as<int>()));
        i2c->writeWordReg(MPU6050_REG_OFF_ACCEL_Z, (uint16_t)htobe16(
            ↪ mpu_cfg["accel_offset"]["z"].as<int>()));

        i2c->writeWordReg(MPU6050_REG_OFF_GYRO_X, (uint16_t)htobe16(mpu_cfg
            ↪ ["gyro_offset"]["x"].as<int>()));
        i2c->writeWordReg(MPU6050_REG_OFF_GYRO_Y, (uint16_t)htobe16(mpu_cfg
            ↪ ["gyro_offset"]["y"].as<int>()));
        i2c->writeWordReg(MPU6050_REG_OFF_GYRO_Z, (uint16_t)htobe16(mpu_cfg
            ↪ ["gyro_offset"]["z"].as<int>()));

        std::cout << "MPU offsets succesfully setted!" << std::endl;
    }
    catch(std::exception &e) {
        std::cout << "[ERROR] Failed to load yaml file" << std::endl;
        // dummy, TODO -> call configuration and generate config file
    }
}

void MPUWrapper::initROSNode()
{
    std::string syn;

    node = new ros::NodeHandle ("~");
```

```
    pub = node->advertise<sensor_msgs::Imu>(ROS_IMU_TOPIC, DEFAULT_HZ / 10)
        ↪ ; // 2 * DEFAULT_HZ

    /* parsing synchro param */
    if(!(node->hasParam(NODE_PARAM_NAME))) {
        std::cout <<"[INFO] Wrong parameter format\nUsage: rosrun mpu6050
            ↪ mpu6050_node _synchro:=[clock, timer]\nStarting node in
            ↪ timer mode"<<std::endl;
        mode = true;
    }
    else {
        node->getParam(NODE_PARAM_NAME, syn);
        if(syn.compare(NODE_PARAM_TIMER) == 0) {
                std::cout<<"Starting node in timer mode"<<std::endl;
                mode = true;
        }
        else if(syn.compare(NODE_PARAM_CLOCK) == 0) {
                std::cout<<"Starting node in clock mode"<<std::endl;
                mode = false;
        }
        else {
                std::cout <<"[INFO] Wrong parameter format\n_synchro:=[clock
                    ↪ , timer]\nStarting node in timer mode"<<std::endl;
                mode = true;
        }
        /* cleaning cached parameters */
        node->deleteParam(NODE_PARAM_NAME);
    }

    msg.header.frame_id = MPU_FRAMEID;
}

void MPUWrapper::startPublish()
{
    if(mode)
        timer = node->createTimer(ros::Duration(1.0 / DEFAULT_HZ), &
            ↪ MPUWrapper::timerCallback, this, false);
    else
        sub = node->subscribe("/clock", 3, &MPUWrapper::clockCallback, this
            ↪ );
    std::cout << "Starting to publish at " << ROS_IMU_TOPIC << "!" << std::
        ↪ endl;
}

void MPUWrapper::timerCallback(const ros::TimerEvent& event)
{
```

```
   sensor_msgs::Imu* pubMsg = new sensor_msgs::Imu(msg);

   /* filling message header {seq (auto?), stamp, frame_id} */
   pubMsg->header.stamp = event.current_real;
   //msg.header.stamp = event.current_expected;

   readData();

   /* publish message and sleep in accord to RATE_HZ */
   pub.publish(*pubMsg);
   delete pubMsg;
}

void MPUWrapper::clockCallback(const rosgraph_msgs::Clock::ConstPtr&
    ↪ clkMsg)
{
   sensor_msgs::Imu* pubMsg = new sensor_msgs::Imu(msg);
   /* filling message header {seq (auto?), stamp, frame_id} */
   pubMsg->header.stamp = clkMsg->clock;

   readData();

   /* publish message and sleep in accord to RATE_HZ */
   pub.publish(*pubMsg);
   delete pubMsg;
}

void MPUWrapper::readData()
{
   uint8_t raw_accel_data[MPU6050_REG_BLOCK_SIZE];
   uint8_t raw_gyro_data[MPU6050_REG_BLOCK_SIZE];

   /* read raw accel data */
   i2c->readBytesReg(MPU6050_REG_RAW_ACCEL_EP, raw_accel_data,
      ↪ MPU6050_REG_BLOCK_SIZE);

   /* read raw gyro data */
   i2c->readBytesReg(MPU6050_REG_RAW_GYRO_EP, raw_gyro_data,
      ↪ MPU6050_REG_BLOCK_SIZE);


   /* NB -> RockPi is LittleEndian, MPU is BigEndian -> lower address
      ↪ register first */

   /* accel normalization and imu message filling */
```

```
    msg.linear_acceleration.x = (((int16_t)(raw_accel_data[0] << 8) |
        ↪ raw_accel_data[1]) / MPU6050_ACCEL_SCALE) * G;
    msg.linear_acceleration.y = (((int16_t)(raw_accel_data[2] << 8) |
        ↪ raw_accel_data[3]) / MPU6050_ACCEL_SCALE) * G;
    msg.linear_acceleration.z = (((int16_t)(raw_accel_data[4] << 8) |
        ↪ raw_accel_data[5]) / MPU6050_ACCEL_SCALE) * G;

    /* gyro normalization and imu message filling */
    msg.angular_velocity.x = ((int16_t)(raw_gyro_data[0] << 8) |
        ↪ raw_gyro_data[1]) / MPU6050_GYRO_SCALE * DEG_TO_RAD;
    msg.angular_velocity.y = ((int16_t)(raw_gyro_data[2] << 8) |
        ↪ raw_gyro_data[3]) / MPU6050_GYRO_SCALE * DEG_TO_RAD;
    msg.angular_velocity.z = ((int16_t)(raw_gyro_data[4] << 8) |
        ↪ raw_gyro_data[5]) / MPU6050_GYRO_SCALE * DEG_TO_RAD;
}
```

Appendix B

Camera Node

Listing B.1: main

```
/*
 * Author: Matteo Accornero
 *
 * Example usage: rosrun ueye_light ueye_light_node _synchro:=[clock,
    ↪ timer] _res:=[full, scaled]
 */

#include "EOWrapper.hpp"


int main(int argc, char **argv)
{
    ros::init(argc, argv, "ueye_light");

    EOWrapper* ew = new EOWrapper();
    ew->startPublish();

    ros::spin();

    delete ew;
    ros::shutdown();

    return 0;
}
```

Listing B.2: header

```
#ifndef EOWRAPPER_HPP
#define EOWRAPPER_HPP

/* standard headers */
```

```
#include <iostream>
#include <stdio.h>
#include <stddef.h>

/* uEYE header */
#include "ueye.h"

/* ROS headers */
#include <ros/ros.h>
#include <sensor_msgs/Image.h>
#include <rosgraph_msgs/Clock.h>

#define DEFAULT_HZ 20
#define THROTTLER 10   // CLOCK_HZ / DEFAULT_HZ
#define CAMERA_TOPIC_NAME "/camera/image_raw"
#define CONFIG_PATH_NAME "/home/rock/Documents/Workspace/catkin_ws_sensors
    ↪ /src/camera/config/"

/* image_msg defines */
#define CAMERA_ENCODING "mono8"
#define CAMERA_FRAMEID "camera"
#define CAMERA_IS_BIGENDIAN 0x00
#define CAMERA_WIDTH 1280
#define CAMERA_HEIGHT 1024

/* ROS node param name */
#define NODE_PARAM_SYNCHRO_NAME "synchro"
#define NODE_PARAM_TIMER "timer"
#define NODE_PARAM_CLOCK "clock"

#define NODE_PARAM_RESOLUTION_NAME "res"
#define NODE_PARAM_FULL "full"
#define NODE_PARAM_SCALED "scaled"

class EOWrapper {

   private:
    HIDS hCam;
    char* pMem;
    int memID;
    ros::NodeHandle* node;
    ros::Publisher pub;
    ros::Subscriber sub;  // needed for clk mode
    ros::Timer timer;
    sensor_msgs::Image msg;
    bool time;   // [1: timer, 0: clock]
```

```
    bool scale;   // [1: 640, 0: 1280]
    int imgDimension;
    int thr;

    void initCamera();
    void initROSNode();
    void transferImage();
    void timerCallback(const ros::TimerEvent& event);
    void clockCallback(const rosgraph_msgs::Clock::ConstPtr& clkMsg);

  public:
    EOWrapper();
    ~EOWrapper();
    void startPublish();

};

#endif // EOWRAPPER_HPP
```

Listing B.3: functions

```
#include "EOWrapper.hpp"

EOWrapper::EOWrapper()
{
    thr = 0;
    initROSNode();
    initCamera();
}

EOWrapper::~EOWrapper()
{
    delete node;
    is_FreeImageMem(hCam, pMem, memID);
    is_ExitCamera(hCam);
}

void EOWrapper::initCamera()
{
    hCam = 0;
    // Open cam and see if it was succesfull
    INT nRet = is_InitCamera(&hCam, NULL);
    if(nRet == IS_SUCCESS) {
        std::cout << "Camera initialized!" << std::endl;
    }
```

```
/* Needed because the library API needs a wide-string (wstring) as
    ↪ paramter */
std::string path(CONFIG_PATH_NAME);
path = path + ((!scale) ? NODE_PARAM_FULL : NODE_PARAM_SCALED) + ".ini"
    ↪ ;

const std::wstring filenameU(path.begin(), path.end());

nRet = is_ParameterSet(hCam, IS_PARAMETERSET_CMD_LOAD_FILE, (void*)
    ↪ filenameU.c_str(), 0);
if(nRet == IS_SUCCESS) {
    std::cout << "Configuration loaded!" << std::endl;
    INT colorMode;
    double fps, exposure, pixelClock;
    SENSORINFO sInfo;
    nRet |= is_PixelClock(hCam, IS_PIXELCLOCK_CMD_GET, (void*) &
        ↪ pixelClock, sizeof(double));
    nRet |= is_SetFrameRate(hCam, IS_GET_FRAMERATE, &fps);
    nRet |= is_Exposure(hCam, IS_EXPOSURE_CMD_GET_EXPOSURE, (void*) &
        ↪ exposure, sizeof(double));
    nRet |= is_GetSensorInfo (hCam, &sInfo);
    colorMode = is_SetColorMode (hCam, IS_GET_COLOR_MODE);

    std::cout << "Pixel Clock: " << pixelClock << std::endl;
    std::cout << "FPS: " << fps << std::endl;
    std::cout << "Exposure: " << exposure << std::endl;
    std::cout << "ColorMode: " << colorMode << std::endl;
    std::cout << "Width: " << sInfo.nMaxWidth << std::endl;
    std::cout << "Height: " << sInfo.nMaxHeight << std::endl;
    std::cout << "Golbal Shutter: " << (sInfo.bGlobShutter ? "true" : "
        ↪ false") << std::endl;
}
else {
    std::cout << "Failed to load .ini! ->" << nRet << std::endl;
    //TODO
}

/* reserving memory for images */
pMem = NULL;
memID = 0;
is_AllocImageMem(hCam, (!scale) ? CAMERA_WIDTH : CAMERA_WIDTH / 2, (!
    ↪ scale) ? CAMERA_HEIGHT : CAMERA_HEIGHT / 2, 8, &pMem, &memID);
is_SetImageMem(hCam, pMem, memID);

nRet = is_CaptureVideo(hCam, IS_WAIT);
if (nRet != IS_SUCCESS) {
```

```
        std::cout << "Image data could not be written in memory!" << nRet
            ↪ << std::endl;
    }

/*
    INT displayMode = IS_SET_DM_DIB;
    nRet = is_SetDisplayMode (*hCam_internal, displayMode);
    if(nRet == IS_SUCCESS)
        std::cout << "Status displayMode " << displayMode << " -> " <<
            ↪ nRet << std::endl;

    INT colorMode = IS_CM_MONO8; //IS_COLORMODE_MONOCHROME;
    nRet = is_SetColorMode(*hCam_internal, colorMode);
    if(nRet == IS_SUCCESS)
        std::cout << "Status colorMode " << colorMode << " -> " << nRet <<
            ↪  std::endl;
*/

}

void EOWrapper::initROSNode()
{
    std::string param;

    node = new ros::NodeHandle ("~");
    pub = node->advertise<sensor_msgs::Image>(CAMERA_TOPIC_NAME, 1);  //
        ↪ DEFAULT_HZ / 2

    /* parsing synchro param */
    if(!(node->hasParam(NODE_PARAM_SYNCHRO_NAME))) {
        std::cout<<"[INFO] Wrong parameter format\nUsage: rosrun ueye_light
            ↪  ueye_light_node [...] _synchro:=[clock, timer]\nStarting
            ↪ node in timer mode"<<std::endl;
        time = true;
    }
    else {
        node->getParam(NODE_PARAM_SYNCHRO_NAME, param);
        if(param.compare(NODE_PARAM_TIMER) == 0)
                time = true;
        else if(param.compare(NODE_PARAM_CLOCK) == 0)
                time = false;
        else {
                std::cout<<"[INFO] Wrong parameter format\n_synchro:=[clock,
                    ↪  timer]\nStarting node in timer mode"<<std::endl;
                time = true;
        }
```

```
        /* cleaning cached parameters */
        node->deleteParam(NODE_PARAM_SYNCHRO_NAME);
    }

    /* parsing resolution param */
    if(!(node->hasParam(NODE_PARAM_RESOLUTION_NAME))) {
        std::cout<<"[INFO] Wrong parameter format\nUsage: rosrun ueye_light
            ↪  ueye_light_node [...] _res:=[full, scaled]\nStarting node
            ↪ in full resolution mode"<<std::endl;
        time = true;
    }
    else {
        node->getParam(NODE_PARAM_RESOLUTION_NAME, param);
        if(param.compare(NODE_PARAM_FULL) == 0)
               scale = false;
        else if(param.compare(NODE_PARAM_SCALED) == 0)
               scale = true;
        else {
               std::cout<<"[INFO] Wrong parameter format\n_res:=[full,
                   ↪ scaled]\nStarting node in full resolution mode"<<std
                   ↪ ::endl;
               scale = false;
        }
        node->deleteParam(NODE_PARAM_RESOLUTION_NAME);
    }

    msg.header.frame_id = CAMERA_FRAMEID;
    msg.width = (!scale) ? CAMERA_WIDTH : CAMERA_WIDTH / 2;
    msg.height = (!scale) ? CAMERA_HEIGHT : CAMERA_HEIGHT / 2;
    msg.encoding = CAMERA_ENCODING;
    msg.is_bigendian = CAMERA_IS_BIGENDIAN;
    msg.step = (!scale) ? CAMERA_WIDTH : CAMERA_WIDTH / 2;

    imgDimension = msg.height * msg.step;
    std::cout << "ROS node successfully created!" << std::endl;
}

void EOWrapper::startPublish()
{
    if(time)
        timer = node->createTimer(ros::Duration(1.0 / DEFAULT_HZ), &
            ↪ EOWrapper::timerCallback, this, false);
    else
        sub = node->subscribe("/clock", DEFAULT_HZ, &EOWrapper::
            ↪ clockCallback, this);
```

```
    std::cout << "Starting to publish at " << CAMERA_TOPIC_NAME << "!" <<
        ↪ std::endl;
}

void EOWrapper::transferImage()
{
    VOID* pMem_b;
    /*
    int retInt = is_FreezeVideo(hCam, IS_WAIT);
    if (retInt != IS_SUCCESS) {
        std::cout << "Image data could not be written in memory!" <<
            ↪ retInt << std::endl;
    }*/
    int retInt = is_GetImageMem(hCam, &pMem_b);
    /*if (retInt != IS_SUCCESS) {
        std::cout << "Image data could not be read from memory!" << retInt
            ↪  << std::endl;
    }*/

    msg.data.reserve(imgDimension);
    memcpy(msg.data.data(), pMem_b, imgDimension);
    msg.data.resize(imgDimension); // height * step
}

void EOWrapper::timerCallback(const ros::TimerEvent& event)
{
    transferImage();
    msg.header.stamp = event.current_real;
    //msg.header.stamp = event.current_expected;
    pub.publish(msg);
}

void EOWrapper::clockCallback(const rosgraph_msgs::Clock::ConstPtr& clkMsg
    ↪ )
{
    if(thr < THROTTLER - 1) {
        thr++;
        return;
    }
    thr = 0;
    transferImage();
    msg.header.stamp = clkMsg->clock;
    pub.publish(msg);

}
```

Appendix C

Clock Server Node

Listing C.1: Clock Server Node

```
import rospy
from rosgraph_msgs.msg import Clock

class ClockGenerator :
        def __init__(self):
                # Create a ROS publisher
                self.clock_publisher = rospy.Publisher('clock', Clock,
                    ↪ queue_size = 0)
                self.clock_msg = Clock()

        def publish_clock(self, event):
                #self.clock_msg.clock = event.current_real
                self.clock_msg.clock = event.current_expected
                self.clock_publisher.publish(self.clock_msg)

if __name__ == '__main__':
        try:
                rospy.init_node("clock_server")

                cg = ClockGenerator()

                # Create another ROS Timer for publishing data
                rospy.Timer(rospy.Duration(1.0/200.0), cg.publish_clock)
                print("Starting publishing at /clock")
                # Don't forget this or else the program will exit
                rospy.spin()

        except rospy.ROSInterruptException:
                rospy.shutdown()
```

Appendix D

IMU offsets calibrator

Listing D.1: IMU offset calibrator

```
/*
 * Author: Accornero Matteo
 *
 * Example usage: RockPi N10 mraa MPU6050 offset calibration
 *
 */

/* standard headers */
#include <endian.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

/* mraa header */
#include "mraa/i2c.h"

/*
 * 0 I2C-6
 * 1 I2C-2
 * 2 I2C-7
 */
#define I2C_BUS 1

/* register definitions */
#define MPU6050_ADDR 0x68
#define MPU6050_REG_PWR_MGMT_1 0x6b

/* accelerometer raw measure registers entry point address */
#define MPU6050_REG_RAW_ACCEL_EP 0x3b
```

```
/* gyroscope raw measure registers entry point address */
#define MPU6050_REG_RAW_GYRO_EP 0x43

/* accelerometer offset registers entry point address */
#define MPU6050_REG_OFF_ACCEL_EP 0x06

/* gyroscope offset registers entry point address */
#define MPU6050_REG_OFF_GYRO_EP 0x13

/* registers displacements to read coordinates words */
#define MPU6050_REG_WORD_X_DISPL 0x00
#define MPU6050_REG_WORD_Y_DISPL 0x02
#define MPU6050_REG_WORD_Z_DISPL 0x04

/* bit definitions */
#define MPU6050_RESET 0x80
#define MPU6050_SLEEP (1 << 6)
#define MPU6050_PLL_GYRO_X (1 << 1)

/*
 * accelerometer scale factors, set AFS_SEL register accordingly
 * (+/-)[Full Scale Range] [LSB Sensitivity - refactor]
 * (+/-)[2g] [16384.0 - 8] (*)
 * (+/-)[4g] [8192.0 - 4]
 * (+/-)[8g] [4092.0 - 2]
 * (+/-)[16g] [2048.0 - 1]
 */
#define MPU6050_ACCEL_SCALE 16384.0
#define ACCEL_MEAN_REFACTOR 8
#define ACCEL_MEAN_DEADZONE 8

/*
 * gyroscope scale factor, set FS_SEL register accordingly
 * (+/-)[Full Scale Range] [LSB Sensitivity - refactor]
 * (+/-)[250 deg/s] [131.0 - 4] (*)
 * (+/-)[500 deg/s] [65.5 - 2]
 * (+/-)[1000 deg/s] [32.8 - 1]
 * (+/-)[2000 deg/s] [16.4 - 0.5]
 */
#define MPU6050_GYRO_SCALE 131.0
#define GYRO_MEAN_REFACTOR 4
#define GYRO_MEAN_DEADZONE 1

/* samples number for mean calculus */
#define SAMPLES 1000
```

```
/* sleep time (s) between two accel/gyro measurements */
#define SAMPLE_FREQUENCY 0.5

/* mpu_config.yaml output file */
#define PATH_CFG_FILE "/home/rock/Documents/Workspace/catkin_ws_sensors/
    ↪ src/imu/config/offsets.yaml"

/*
 * debug:
 *  - verbose: gcc -o calibration calibration.c -lmraa -lyaml-cpp -DDEBUG
 *  - silent: gcc -o calibration calibration.c -lmraa -lyaml-cpp
 */


/* struct definitions */
typedef struct {
    int16_t x;
    int16_t y;
    int16_t z;
} coord16_t;

typedef struct {
    int32_t x;
    int32_t y;
    int32_t z;
} coord32_t;

/* functions prototypes */
int16_t i2c_read_word(mraa_i2c_context, uint8_t);
mraa_result_t i2c_read_coord16_t(mraa_i2c_context, coord16_t*, uint8_t);
mraa_result_t i2c_write_word(mraa_i2c_context, int16_t, uint8_t);
mraa_result_t i2c_write_coord16_t(mraa_i2c_context, coord16_t*, uint8_t);
void get_ag_means(mraa_i2c_context, coord16_t*, coord16_t*);
void yaml_output(coord16_t*, coord16_t*);

int iter = 1;

int main(void)
{
    mraa_result_t status = MRAA_SUCCESS;
    mraa_i2c_context i2c;
    uint8_t data;
    coord16_t accel_offset = {0}, accel_mean;
    coord16_t gyro_offset = {0}, gyro_mean;
    int ret, ready;
```

```
/* initialize mraa for the platform (not needed most of the times) */
mraa_init();

/* initialize I2C bus */
i2c = mraa_i2c_init(I2C_BUS);
if (i2c == NULL) {
    fprintf(stderr, "Failed to initialize I2C\n");
    mraa_deinit();
    return EXIT_FAILURE;
}

/* set slave address */
status = mraa_i2c_address(i2c, MPU6050_ADDR);
if (status != MRAA_SUCCESS) {
    goto err_exit;
}

/* reset the sensor */
status = mraa_i2c_write_byte_data(i2c, MPU6050_RESET,
    ↪ MPU6050_REG_PWR_MGMT_1);
if (status != MRAA_SUCCESS) {
    goto err_exit;
}

/* configure power management register */
ret = mraa_i2c_read_byte_data(i2c, MPU6050_REG_PWR_MGMT_1);
if (ret == -1) {
    return EXIT_FAILURE;
}

data = ret;
data |= MPU6050_PLL_GYRO_X;
data &= ~(MPU6050_SLEEP);

status = mraa_i2c_write_byte_data(i2c, data, MPU6050_REG_PWR_MGMT_1);
if (status != MRAA_SUCCESS) {
    goto err_exit;
}

/* reset acceleration offset registers */
i2c_write_coord16_t(i2c, &accel_offset, MPU6050_REG_OFF_ACCEL_EP);

/* reset gyroscope offset registers */
i2c_write_coord16_t(i2c, &gyro_offset, MPU6050_REG_OFF_GYRO_EP);
```

```
#ifdef DEBUG
/* check if reset succeded */
i2c_read_coord16_t(i2c, &accel_offset, MPU6050_REG_OFF_ACCEL_EP);
i2c_read_coord16_t(i2c, &gyro_offset, MPU6050_REG_OFF_GYRO_EP);
fprintf(stdout, "offsa: x:%d y:%d z:%d\n", accel_offset.x, accel_offset
    ↪ .y, accel_offset.z);
fprintf(stdout, "offsg: x:%d y:%d z:%d\n", gyro_offset.x, gyro_offset.y
    ↪ , gyro_offset.z);
#endif

/* setup time */
sleep(5);

/* compute accel and gyro mean */
get_ag_means(i2c, &accel_mean, &gyro_mean);

/* init offsets:
 *  - negative because they are summed up to registers values
 *  - z accel offset is computed having in mind that the sensor
     ↪ observs anyway g
 *  thus stationary measurements must be around 1 in accord to the
     ↪ sensor position
 */
accel_offset.x -= accel_mean.x / ACCEL_MEAN_REFACTOR;
accel_offset.y -= accel_mean.y / ACCEL_MEAN_REFACTOR;
accel_offset.z = (MPU6050_ACCEL_SCALE - accel_mean.z) /
    ↪ ACCEL_MEAN_REFACTOR;
gyro_offset.x -= gyro_mean.x / GYRO_MEAN_REFACTOR;
gyro_offset.y -= gyro_mean.y / GYRO_MEAN_REFACTOR;
gyro_offset.z -= gyro_mean.z / GYRO_MEAN_REFACTOR;

do {

    /* exit condition */
    ready = 0;

    /* writing current offsets */
    i2c_write_coord16_t(i2c, &accel_offset, MPU6050_REG_OFF_ACCEL_EP);
    i2c_write_coord16_t(i2c, &gyro_offset, MPU6050_REG_OFF_GYRO_EP);

    /* compute next round accel and gyro means */
    get_ag_means(i2c, &accel_mean, &gyro_mean);

    /* tune offsets:
     *  - if mean is under threshold -> correct offset point
     *  - otherwise subtract and repeat
```

```
     */
    if (abs(accel_mean.x) <= ACCEL_MEAN_DEADZONE) ready++;
    else accel_offset.x = accel_offset.x - accel_mean.x /
        ↪ ACCEL_MEAN_REFACTOR;

    if (abs(accel_mean.y) <= ACCEL_MEAN_DEADZONE) ready++;
    else accel_offset.y = accel_offset.y - accel_mean.y /
        ↪ ACCEL_MEAN_REFACTOR;

    if (abs(MPU6050_ACCEL_SCALE - accel_mean.z) <= ACCEL_MEAN_DEADZONE)
        ↪  ready++;
    else accel_offset.z = accel_offset.z + (MPU6050_ACCEL_SCALE -
        ↪ accel_mean.z) / ACCEL_MEAN_REFACTOR;

    if (abs(gyro_mean.x) <= GYRO_MEAN_DEADZONE) ready++;
    else gyro_offset.x = gyro_offset.x - gyro_mean.x /
        ↪ GYRO_MEAN_REFACTOR;

    if (abs(gyro_mean.y) <= GYRO_MEAN_DEADZONE) ready++;
    else gyro_offset.y = gyro_offset.y - gyro_mean.y /
        ↪ GYRO_MEAN_REFACTOR;

    if (abs(gyro_mean.z) <= GYRO_MEAN_DEADZONE) ready++;
    else gyro_offset.z = gyro_offset.z - gyro_mean.z /
        ↪ GYRO_MEAN_REFACTOR;

}
while (ready != 6);

//#ifdef DEBUG
//fprintf(stdout, "\n(mean int) accel: x:%d y:%d z:%d", accel_mean.x,
    ↪  accel_mean.y, accel_mean.z);
//fprintf(stdout, "\n(mean int) gyro : x:%d y:%d z:%d\n", gyro_mean.x
    ↪ , gyro_mean.y, gyro_mean.z);
//fprintf(stdout, "\n(mean uint) accel: x:%d y:%d z:%d", (uint16_t)
    ↪ accel_mean.x, (uint16_t) accel_mean.y, (uint16_t) accel_mean.z)
    ↪ ;
//fprintf(stdout, "\n(mean uint) gyro : x:%d y:%d z:%d\n", (uint16_t)
    ↪  gyro_mean.x, (uint16_t) gyro_mean.y, (uint16_t) gyro_mean.z);
//#endif

#ifdef DEBUG
/* check final offsets */
i2c_read_coord16_t(i2c, &accel_offset, MPU6050_REG_OFF_ACCEL_EP);
i2c_read_coord16_t(i2c, &gyro_offset, MPU6050_REG_OFF_GYRO_EP);
```

```
    fprintf(stdout, "\noffsa: x:%d y:%d z:%d", accel_offset.x, accel_offset
        ↪ .y, accel_offset.z);
    fprintf(stdout, "\noffsg: x:%d y:%d z:%d\n", gyro_offset.x, gyro_offset
        ↪ .y, gyro_offset.z);
    #endif

    /* stop i2c */
    mraa_i2c_stop(i2c);

    /* deinitialize mraa for the platform (not needed most of the times)
        ↪ */
    mraa_deinit();

    /* build yaml file */
    yaml_output(&accel_offset, &gyro_offset);

    return EXIT_SUCCESS;

err_exit:
    mraa_result_print(status);

    /* stop i2c */
    mraa_i2c_stop(i2c);

    /* deinitialize mraa for the platform (not needed most of the times)
        ↪ */
    mraa_deinit();

    return EXIT_FAILURE;
}

int16_t i2c_read_word(mraa_i2c_context dev, uint8_t command)
{
    /* Rock PI N10 Little Endian, MPU 6050 Big Endian -> conversion */
    return (int16_t)be16toh(mraa_i2c_read_word_data(dev, command));
}

mraa_result_t i2c_read_coord16_t(mraa_i2c_context dev, coord16_t* data,
    ↪ uint8_t command_ep)
{
    data->x = i2c_read_word(dev, command_ep + MPU6050_REG_WORD_X_DISPL);
    data->y = i2c_read_word(dev, command_ep + MPU6050_REG_WORD_Y_DISPL);
    data->z = i2c_read_word(dev, command_ep + MPU6050_REG_WORD_Z_DISPL);

    /* dummy */
    return MRAA_SUCCESS;
```

```
}

mraa_result_t i2c_write_word(mraa_i2c_context dev, int16_t data, uint8_t
    ↪ command)
{
    /* Rock PI N10 Little Endian, MPU 6050 Big Endian -> conversion */
    return mraa_i2c_write_word_data(dev, htobe16((uint16_t) data), command
        ↪ );
}

mraa_result_t i2c_write_coord16_t(mraa_i2c_context dev, coord16_t* data,
    ↪ uint8_t command_ep)
{
    i2c_write_word(dev, data->x, command_ep + MPU6050_REG_WORD_X_DISPL);
    i2c_write_word(dev, data->y, command_ep + MPU6050_REG_WORD_Y_DISPL);
    i2c_write_word(dev, data->z, command_ep + MPU6050_REG_WORD_Z_DISPL);

    /* dummy */
    return MRAA_SUCCESS;
}

void get_ag_means(mraa_i2c_context dev, coord16_t* accel_mean, coord16_t*
    ↪ gyro_mean)
{
    /* buffers on 32 bit in order to avoid overflow */
    coord32_t accel_buff = {0}, gyro_buff = {0};
    coord16_t accel_data, gyro_data;

    for(int i=0; i<SAMPLES; i++) {

        /* read raw accel data */
        i2c_read_coord16_t(dev, &accel_data, MPU6050_REG_RAW_ACCEL_EP);

        /* read raw gyro data */
        i2c_read_coord16_t(dev, &gyro_data, MPU6050_REG_RAW_GYRO_EP);

        #ifdef DEBUG
        /* check imu readings */
        fprintf(stdout, "(%d - %d/%d)
            ↪ ----------------------------------------\n", iter, i+1,
            ↪ SAMPLES);
        fprintf(stdout, "accel: x:%d y:%d z:%d\n", accel_data.x, accel_data
            ↪ .y, accel_data.z);
        fprintf(stdout, "gyro : x:%d y:%d z:%d\n", gyro_data.x, gyro_data.y
            ↪ , gyro_data.z);
        #endif
```

```
        /* accummulate accel measurements */
        accel_buff.x += accel_data.x;
        accel_buff.y += accel_data.y;
        accel_buff.z += accel_data.z;

        /* accumulate gyro measurements */
        gyro_buff.x += gyro_data.x;
        gyro_buff.y += gyro_data.y;
        gyro_buff.z += gyro_data.z;

        sleep(SAMPLE_FREQUENCY);
    }

    /* compute means and cast them back to int16_t (safe) */
    accel_mean->x = (int16_t)(accel_buff.x / SAMPLES);
    accel_mean->y = (int16_t)(accel_buff.y / SAMPLES);
    accel_mean->z = (int16_t)(accel_buff.z / SAMPLES);

    gyro_mean->x = (int16_t)(gyro_buff.x / SAMPLES);
    gyro_mean->y = (int16_t)(gyro_buff.y / SAMPLES);
    gyro_mean->z = (int16_t)(gyro_buff.z / SAMPLES);

    iter++;
}

void yaml_output(coord16_t* accel_offset, coord16_t* gyro_offset)
{
    FILE *fptr;

    /* open file */
    fptr = fopen(PATH_CFG_FILE, "w");

    /* write struct manually (few fields), don't use \t yaml parser doesn
        ↪ 't support them */
    fprintf(fptr, "accel_offset:\n x: %d\n y: %d\n z: %d\n", accel_offset->
        ↪ x, accel_offset->y, accel_offset->z);
    fprintf(fptr, "gyro_offset:\n x: %d\n y: %d\n z: %d\n", gyro_offset->x,
        ↪  gyro_offset->y, gyro_offset->z);

    /* close file */
    fclose(fptr);

    fprintf(stdout, "Offset config file generated!\n");
}
```

Appendix E

ROS bag validator

Listing E.1: ROS bag validator

```
import rosbag
import rospy

clk_hz = 200.0
cam_hz = 20.0

msgs_imu = 0
first_imu = 1
count_imu = 0
msgs_clk = 0
first_clk = 1
count_clk = 0
msgs_cam = 0
first_cam = 1
count_cam = 0

with rosbag.Bag('output.bag', 'w') as outbag:
    for topic, msg, t in rosbag.Bag('sync.bag').read_messages():
        if topic == "/imu0" :
            msgs_imu += 1
            t_imu = rospy.Time(msg.header.stamp.secs, msg.header.stamp.
                ↪ nsecs)
            outbag.write(topic, msg, msg.header.stamp)
            if first_imu :
                first_imu = 0
                t_prev_imu = t_imu
                continue
            t_diff = (t_imu-t_prev_imu).to_sec()
            if t_diff <= (1 / clk_hz) :
                print("[IMU_OK]: ", t_imu, " {", t_diff, "}")
```

```
            else :
                print("[IMU_ERR]: ", t_imu, " {", t_diff, "}")
                count_imu += 1
                #tmp = raw_input()
            t_prev_imu = t_imu
        elif topic == "/camera/image_raw" :
            msgs_cam += 1
            t_cam = rospy.Time(msg.header.stamp.secs, msg.header.stamp.
                ↪ nsecs)
            outbag.write(topic, msg, msg.header.stamp)
            if first_cam :
                first_cam = 0
                t_prev_cam = t_cam
                continue
            t_diff = (t_cam-t_prev_cam).to_sec()
            if t_diff <= (1 / cam_hz) :
                print("[CAM_OK]: ", t_cam, " {", t_diff, "}")
            else :
                print("[CAM_ERR]: ", t_cam, " {", t_diff, "}")
                count_cam += 1
                #tmp = raw_input()
            t_prev_cam = t_cam
        elif topic == "/clock" :
            msgs_clk += 1
            t_clk = rospy.Time(msg.clock.secs, msg.clock.nsecs)
            outbag.write(topic, msg, msg.clock)
            if first_clk:
                first_clk = 0
                t_prev_clk = t_clk
                continue
            t_diff = (t_clk-t_prev_clk).to_sec()
            if t_diff <= (1 / clk_hz):
                print("[CLK_OK]: ", t_clk, " {", t_diff, "}")
            else:
                print("[CLK_ERR]: ", t_clk, " {", t_diff, "}")
                count_clk += 1
                #tmp = raw_input()
            t_prev_clk = t_clk

if msgs_imu != 0 :
    perc_imu = count_imu / msgs_imu * 100
else:
    perc_imu = 0
if msgs_cam != 0 :
    perc_cam = count_cam / msgs_cam * 100
else:
```

```
    perc_cam = 0
if msgs_clk != 0 :
    perc_clk = count_clk / msgs_clk * 100
else:
    perc_clk = 0
print("_____________________________________________")
print("#IMU_MSG: ", msgs_imu)
print("#IMU_ERR: ", count_imu)
print("%IMU_ERR: ", perc_imu)
print("_____________________________________________")
print("#CAM_MSG: ", msgs_cam)
print("#CAM_ERR: ", count_cam)
print("%CAM_ERR: ", perc_cam)
print("_____________________________________________")
print("#CLK_MSG: ", msgs_clk)
print("#CLK_ERR: ", count_clk)
print("%CLK_ERR: ", perc_clk)
print("_____________________________________________")
```

Bibliography

[1] "Federal aviation administration." `https://www.faa.gov`, 2021.

[2] G. Chowdhary, J. Eric N., and M. Daniel, "Gps-denied indoor and outdoor monocular vision aided navigation and control of unmanned aircraft," *Journal of Field Robotics*, 2015.

[3] "Nasa jpl." `https://mars.nasa.gov/technology/helicopter/`, 2021.

[4] N. JPL, "*NASA JPL F Prime* [source code]." `https://github.com/nasa/fprime`, 2021.

[5] T. Qin, P. Li, and S. Shen, "Vins-mono: A robust and versatile monocular visual-inertial state estimator," *IEEE Transactions on Robotics*, 2018.

[6] G. Huang, "Visual-inertial navigation: A concise review," 2019.

[7] Robot Perception & Navigation Group, University of Delaware, "*OpenVINS* [documentation]." `https://docs.openvins.com/index.html`, 2021.

[8] Robot Perception & Navigation Group, University of Delaware2, "*OpenVINS* [source code]." `https://github.com/rpng/open_vins`, 2021.

[9] Rockchip, *RK3399Pro*, 1 2019. Rev. 1.0.

[10] "Seeedstudio." `https://www.seeedstudio.com/`, 2021.

[11] "Edmund optics." `https://www.edmundoptics.com`, 2021.

[12] Invesense, *MPU-6000 and MPU-6050 Product Specification*, 8 2013. Rev. 3.4.

[13] Invesense, *MPU-6000 and MPU-6050 Register Map and Descriptions*, 8 2013. Rev. 4.2.

[14] "Invesense." `https://invensense.tdk.com/`, 2021.

[15] "Intel™." `https://www.intelrealsense.com/depth-camera-d455/`, 2021.

[16] "Ros documentation wiki." `http://wiki.ros.org/Documentation`, 2021.

[17] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[18] "mraa library documentation." `https://iotdk.intel.com/docs/master/mraa/`, 2021.

[19] J. Rehder and R. Siegwart, "Unified temporal and spatial calibration for multi-sensor systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (Tokyo, Japan), 2013.

[20] P. Furgale, T. D. Barfoot, and G. Sibley, "Continuous-time batch estimation using temporal basis functions," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (St. Paul, MN), p. 2088–2095, 2012.

[21] L. Oth, P. Furgale, L. Kneip, and R. Siegwart, "Rolling shutter camera calibration," in *Proceedings of the IEEE Computer Vision and Pattern Recognition (CVPR)*, 2013.

[22] Autonomous Systems Lab, ETH Zürich, "Kalibr [source code]." `https://github.com/ethz-asl/kalibr`, 2021.

[23] P. Geneva and Robot Perception & Navigation Group, University of Delaware, "*kalibr_allan* [source code]." `https://github.com/rpng/kalibr_allan`, 2021.

[24] "Ieee standard specification format guide and test procedure for single-axis interferometric fiber optic gyros," *IEEE Std 952-1997*, pp. 1–84, 1998.

[25] Y. Yang, P. Geneva, K. Eckenhoff, and G. Huang, "Degenerate motion analysis for aided ins with online spatial and temporal sensor calibration," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 2070–2077, 2019.

[26] P. Furgale, J. Rehder, and R. Siegwart, "Unified temporal and spatial calibration for multi-sensor systems," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1280–1286, 2013.

[27] W. Power, M. Pavlovski, D. Saranovic, I. Stojkovic, and Z. Obradovic, "Autonomous navigation for drone swarms in gps-denied environments using structured learning," pp. 219–231, Springer International Publishing, 2020.

[28] "Vicon." `https://www.vicon.com`, 2021.