RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'EDUCATION SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

مخبر التحكم في العمليات التصنيعية
Laboratoire de Commande des Processus

Département d'Automatique

End of Studies Project

For the attainment of an Engineer degree in Control Engineering

---

Assembly and Tuning of a Quadcopter for Visual SLAM Applications

---

SASSI Zakaria Fakhri & DOULI Abdelhak Samir

Under the supervision of **Pr. BOUDANA Djamel** and **Mr. BAMOUNE Fayçal**

Publicly presented and discussed on (01/07/2024)

Composition of the jury:

President:	Pr. BOUDJEMA Farès	ENP
Promoter :	Pr BOUDANA Djamel	ENP
Promoter :	Mr BAMOUNE Fayçal	ENP
Examiner:	Pr. TADJINE Mohamed	ENP

ENP 2024

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'EDUCATION SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

المدرسة الوطنية المتعددة التقنيات
Ecole Nationale Polytechnique

مخبر التحكم في العمليات التصنيعية
Laboratoire de Commande des Processus

Département d'Automatique

End of Studies Project

For the attainment of an Engineer degree in Control Engineering

---

Assembly and Tuning of a Quadcopter for Visual SLAM Applications

---

SASSI Zakaria Fakhri & DOULI Abdelhak Samir

Under the supervision of **Pr. BOUDANA Djamel** and **Mr. BAMOUNE Fayçal**

Publicly presented and discussed on (01/07/2024)

Composition of the jury:

President:	Pr. BOUDJEMA Farès	ENP
Promoter :	Pr BOUDANA Djamel	ENP
Promoter :	Mr BAMOUNE Fayçal	ENP
Examiner:	Pr. TADJINE Mohamed	ENP

ENP 2024

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'EDUCATION SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

Département d'Automatique

Mémoire de projet de fin d'études

pour l'obtention du diplôme d'Ingénieur d'État en Automatique

---

Montage et Implémentation d'un Quadrirotor pour des application de SLAM visuel

---

SASSI Zakaria Fakhri & DOULI Abdelhak Samir

Sous la supervision de **Pr. BOUDANA Djamel** et **Mr. BAMOUNE Fayçal**

Présenté et soutenue publiquement le (01/07/2024)

Membres du jury:

Président:	Pr. BOUDJEMA Farès	ENP
Promoteur:	Pr. BOUDANA Djamel	ENP
Promoteur:	Mr BAMOUNE Fayçal	ENP
Examinateur:	Pr. TADJINE Mohamed	ENP

ENP 2024

Département d'Automatique / ENP, 10 Rue des Frères OUDEK, El Harrach 16200 Alger.

ملخص

يركز هذا المشروع على تطوير وتحسين طائرة بدون طيار رباعية المحركات (UAV)، مع التركيز على أنظمة التحكم والخوارزميات المكانية والخرائطية المتزامنة (SLAM). نبدأ باستكشاف وحدة التحكم في الطيران Pixhawk 2.4.8 ، حيث نطور نموذجًا رياضيًا للطائرة الرباعية ونصمم وحدات تحكم باستخدام طرق منظم تربيعي خطي LQR وتحديد الأقطاب و PID لضمان الأداء الأمثل. يتم تنفيذ الـ SLAM البصري باستخدام خوارزمية ORB-SLAM على Raspberry Pi 5 ، مع استعراض تفصيلي للأسس النظرية، والتطبيق العملي، والاختبارات، مبرزين نقاط القوة والتحديات. بالإضافة إلى ذلك، نقوم بإجراء اختيار شامل للمكونات، وتحليل التكلفة، وتحديد المعلمات، واختبارات في الظروف الواقعية للتحقق من فعالية نهجنا.

الكلمات المفتاحية : الطائرات بدون طيار، رباعية المحركات، أنظمة التحكم، SLAM ، ORB-SLAM البصري، الملاحة الذاتية، Pixhawk ، SLAM

Résumé

Ce projet se concentre sur le développement et l'optimisation d'un UAV quadrirotor, en mettant l'accent sur les systèmes de contrôle et les algorithmes de localisation et de cartographie simultanées (SLAM). Nous commençons par explorer le contrôleur de vol Pixhawk 2.4.8, en développant un modèle mathématique pour le quadrirotor et en concevant des contrôleurs utilisant les méthodes LQR, placement de pôles et PID pour assurer des performances optimales. La mise en œuvre du SLAM visuel est réalisée en utilisant l'algorithme ORB-SLAM sur un Raspberry Pi 5, avec un examen détaillé de ses fondements théoriques, de son application pratique et de ses tests, soulignant à la fois ses forces et ses défis. De plus, nous effectuons une sélection approfondie des composants, une analyse des coûts, une identification des paramètres et des tests en conditions réelles pour valider l'efficacité de nos approches.

Mots clés : UAV, Quadrirotor, Systèmes de contrôle, SLAM, ORB-SLAM, Pixhawk, Navigation autonome, SLAM visuel.

Abstract

This project focuses on the development and optimization of a quadrotor UAV, emphasizing control systems and Simultaneous Localization and Mapping (SLAM) algorithms. We begin by exploring the Pixhawk 2.4.8 flight controller, developing a mathematical model for the quadrotor, and designing controllers using LQR, pole placement, and PID methods to ensure optimal performance. The implementation of Visual SLAM is carried out using the ORB-SLAM algorithm on a Raspberry Pi 5, with a detailed examination of its theoretical foundations, practical application, and testing, highlighting both its strengths and challenges. Additionally, we conduct thorough component selection, cost analysis, parameter identification, and real-life testing to validate the effectiveness of our approaches.

Keywords : UAV, Quadrotor, Control Systems, SLAM, ORB-SLAM, Pixhawk, Autonomous Navigation, Visual SLAM.

Acknowledgements

First and foremost, I want to dedicate these words to my mom, who always put me at the top of her priorities. Mom, you have given me everything I needed, often at your own expense. From the bottom of my heart, thank you for everything you've done for me. I will do my best to make you even prouder, Inshallah.

To my dad, my best friend, and superhero, who shared countless laughs with me. Even your punishments were given with love, guiding me with life advice that kept me from getting lost in this vast world. Thank you, Dad, for everything you've done for our family.

To my sister Amira, the smartest girl I know. Thank you for being such a wonderful sister. Despite my initial reluctance to take advice from someone younger, you've proved me wrong time and again. I'm grateful for that, Amora.

To my little sister Malak, you taught me the meaning of having someone to protect and guide. You listen and follow my advice even when it's tough, and I'm grateful for having such a special little sister.

To my friends Aymen and Adel, who stood by me during tough moments and helped me through a lot. From both of you, I've learned the importance of honesty and loyalty. No matter what happens, you've shown me what it means to be a man. Thank you both.

To my second family, the greatest treasure of my entire five years at school. First, my brother Amine, who somehow excels in class despite appearing completely lost half the time. He often surprises us by outshining even our teachers. Ghiles, the one I admire the most. I used to think intelligence was a myth and hard work was all you needed, but Ghiles proved me wrong. He just glances at the board, and his eyes start moving—then, boom, he understands everything. Said, the most determined person I know, would send a hundred messages just to get what he wants. He never gives up, and when he says he'll do something, he does it without hesitation. He's a true source of inspiration for me. And Yasser, the kindest person on earth, always honest and ready to help. If you become his close friend, you'll be shocked at how much he's willing to do for you. He taught me that being selfish is a crime, especially among friends. Thank you all for these invaluable lessons.

Finally, to Zaki Halime. Words cannot fully express my gratitude. The countless lessons, the laughter, the debates over who can lift more weight, the dreams, the lifestyle—all those moments have taught me so much. Thank you, Zaki, for being an amazing person and an unforgettable friend.

DOULI Abdelhak Samir

Acknowledgements

You wouldn't think that five years can transform a person. I mean, what's five years in the grand scheme of life, after all? Well, today, I'm writing this teary-eyed and fully convinced that the boy who started this journey and the man who finished it have both proven that to be wrong. In five years, I gained memories that could fill my spirit for an eternity. I've done amazing things with even more amazing people. I have laughed, loved, cried, and felt every other emotion on the spectrum. So today, I'm going to mark the end of this journey by writing down a few lines that could never truly show the extent of gratitude I have. But I'll try my best.

First, to the most extraordinary woman I know, to the person who was with me from the literal start and never asked for anything in return. To the woman who showed me how to be a man, to my beautiful mother: thank you. From the depths of my heart, I thank you for the sacrifices, for the sleepless nights of worry, for always being there for me when I needed you most, even when I didn't deserve it. I love you, Ma.

To my beloved sister, who went from beating me up as a kid to becoming my biggest supporter: thank you for being the best sister I could ask for. You set the bar high. I love you sis.

A special thanks to the people who became my family during these five years. To Mehdi, thank you for showing me what a loyal friend truly means. We're finishing this just like we started—together. Thank you for everything, brother.To the five people who single-handedly defined the last three years of my life:

To Yasser, thank you for being a bundle of joy and never failing to put a smile on my face. You are one of the kindest people I know, and I'm incredibly honored to have you as a friend.

To the big friendly giant, Said, hard on the outside but adorable within, thank you for always being honest and for being the voice of reason in a sea of ignorance. Most importantly, thank you for showing me that always giving your all in everything you do is a way of life.

To Aghiles, one of the most special people I'll ever get the chance to meet, thank you for always listening and being present, and for always having the best advice for everything. Bonding over l Kouba being great is a sure sign of an everlasting friendship.

To Amine, the man who quickly became family and still one of my favorite people on the planet, thank you for always being there and for being a truly selfless person. Spending time with you randomly throughout the day will always be the highlight of my time here.

And finally, to Hakou, thank you for taking a chance on a stranger on his first day at this school. Your smile and act of kindness blossomed into a truly beautiful friendship. You taught me so many things that will stay with me forever, always know that you have a brother for life. Thank you.

And to every one that made this place feel like home, you know who you are, thank you my success is shared with you all. Alhamdoulillah I guess this is it, the end.

or is it ?

Zaki Sassi

Contents

List of Figures

List of Tables

General Introduction

The rapid advancement in technology has revolutionized various fields, and the deployment of Unmanned Aerial Vehicles (UAVs) has attracted significant attention across multiple domains, including mining, civil applications, and military operations. This underscores the need for highly accurate navigation and positioning systems that can adapt to different environments. Additionally, innovative applications have been developed to utilize autonomous drones equipped with sensors for tasks such as mapping and inspection in confined spaces where GPS access is unavailable. Indeed, GPS systems are limited in such environments due to their need for signals from at least four satellites to accurately determine coordinates. This limitation renders GPS ineffective in certain areas like buildings, underground mines, and protected outdoor environments. Additionally, Inertial Measurement Units (IMUs), while common, are not entirely reliable due to error accumulation.

Conversely, research on aerial vehicles and technological advancements in embedded systems, such as microcomputers and onboard sensors, have significantly enhanced the performance of these systems. UAVs designed for environments without GPS access can rely on alternative localization systems using onboard sensors to achieve Simultaneous Localization and Mapping (SLAM). To address this challenge, the fusion of visual sensors and laser sensors has been proposed as a viable alternative to traditional positioning methods due to their independence from external assistance and lower susceptibility to external interferences.

This thesis explores the multiple aspects for developing and optimizing a quadrotor UAV. We aim to enhance the performance and capabilities of the UAV, making it more efficient and reliable for real-world applications.

In the first chapter, we examine the Pixhawk 2.4.8 flight controller, its operating system, and flight stack architecture, develop a mathematical model for the quadrotor, and design three types of controllers—LQR, pole placement, and PID—to ensure optimal flight performance.

The second chapter discusses implementing Visual SLAM for UAVs using the ORB-SLAM algorithm. We cover the theoretical foundations, detail the algorithm's workings, and test it using the EuRoC MAV Dataset, noting its effectiveness in controlled environments and challenges in dynamic scenarios.

In the third chapter, we select components, conduct cost analysis, identify robot parameters, and test SLAM and control systems in real-life scenarios. These tests confirm the effectiveness of our approaches, and we explore further enhancements like achieving full autonomy, implementing obstacle avoidance, and integrating additional sensors for improved performance.

Overall, this thesis aims to provide a comprehensive guide for developing and optimizing quadrotor UAV systems, offering valuable insights into the challenges and solutions encountered in this field. Our work not only contributes to the academic understanding of UAV control and SLAM algorithms but also lays the groundwork for future innovations and practical applications in autonomous aerial systems.

Chapter 1

UAV System and Control Algorithm: Overview and Implementation

1.1 UAV System Overview and Component Analysis

1.1.1 Introduction

The primary aim of this thesis is to develop an autonomous UAV capable of efficiently reaching specific locations for security purposes. The initial requirement is based on this goal, necessitating the implementation of a sophisticated flight control algorithm with significant computational power to precisely follow predefined paths.

However, in a dynamic environment with limited space, the control algorithm must be highly adaptable to rapid changes in input signals from the remote controller and to various disturbances. Additionally, for public security purposes, the drone must be capable of operating under extreme temperature conditions, as it will be deployed in hazardous areas. Finally, the cost of this machine needs to be kept as low as possible to meet market demands.

Considering all of this, our system must address the following prerequisites:

- Use commercial off-the-shelf (COTS) autopilot hardware with adequate computational capability.
- Implement a software stack that can adapt to diverse airframes. Ideally, different UAV airframes with different tasks will use the same flight controller.
- Utilize Model-Based Design with automatic code generation to simulate algorithm results before moving to the prototyping phase.

After evaluating a large number of flight controllers from both sources [15] and [10], we chose and tested two different Pixhawk boards: the Pixhawk 4 Mini and the Pixhawk 2.4.8. These boards are now supported by open-source code for PX4 autopilots, along with the new UAV Toolbox that enables the use of Simulink and MATLAB codes.

The decision was also made by analyzing different articles, especially [1] where they built their own control algorithm using[16] the Pixhawk Pilot Support Package. This package offers the possibility to design control algorithms using Simulink.

1.1.2 Pixhawk and PX4 Software

1.1.2.1 Pixhawk 2.4.8 and Pixhawk 4 mini

Pixhawk is an independent open-hardware project that provides low-cost autopilot hardware designs for academic, hobbyist, and industrial use. It began as a master's thesis project at ETH Zurich in 2008 and has since evolved into a widely-used, open-source, standardized solution with the PX4 flight stack. Pixhawk offers a practical alternative for implementing flight control algorithms in both professional and economical contexts.

This autopilot board allows users to code advanced tasks without needing in-depth knowledge of autopilot design. For automatic control solutions, an average understanding of control theory and high-level programming languages (such as C++, Java, or Python) is sufficient.

The Pixhawk project creates open hardware designs in the form of schematics, which define the components (CPU, sensors, etc.) and their connections/pin mappings. These schematics and reference designs are licensed under CC BY-SA 3.0, allowing users to use, sell, share, modify, and build on the files with proper credit and shared changes under the same open-source license.

Each design is designated as FMUvX (Flight Management Unit Version X), where higher numbers indicate more recent versions, though not necessarily greater capabilities.

The Pixhawk 2.4.8: a cutting-edge autopilot system tailored for unmanned aerial vehicles (UAVs). Powered by a robust 32-bit ARM Cortex M4 core with FPU, it offers superior control and reliability with a clock speed of 168 MHz, 256 KB RAM, and 2 MB Flash memory. Its advanced sensor suite, including MPU6000, ST Micro gyroscope, and accelerometer/compass, ensures precise navigation. With a reliable power management system and versatile connectivity options like UART serial ports, Spektrum/Futaba inputs, and USB interfaces, the Pixhawk 2.4.8 guarantees seamless integration and performance. Compact and lightweight at 38 grams, it's the ideal choice for UAV enthusiasts seeking advanced control and precision in their projects. Obtain your Pixhawk 2.4.8 from trusted manufacturers like mRo to unlock its full potential and elevate your UAV operations to new heights.

The Pixhawk 4 Mini: Engineered for engineers and hobbyists seeking the power of Pixhawk 4 in compact drones. Derived from the Pixhawk 4, it retains FMU processor and memory resources while optimizing size for 250mm racer drones. With a 2.54mm pitch connector, connecting 8 PWM outputs to ESCs is effortless. Developed in collaboration with Holybro® and Auterion®, it adheres to Pixhawk FMUv5 standards, ensuring compatibility with PX4 flight control software. Features include an aluminum casing for superior thermal performance, Bosch® and InvenSense® sensor technology, redundant IMUs, NuttX RTOS, and pre-installed PX4 firmware. Technical specs include an STM32F765 processor, onboard sensors including ICM-20689 and BMI055, IST8310 magnetometer, MS5611 barometer, and ublox Neo-M8N GPS/GLONASS receiver. Voltage ratings range from 4.75V to 24V, with max current sensing at 120A and consumption less than 250mA at 5V. Elevate your drone's capabilities with the Pixhawk 4 Mini – the compact powerhouse for precise and reliable performance.

Figure 1.1: Pixhawk 2.4.8 autopilot board

Figure 1.2: Pixhawk 4 mini autopilot board

We began our project using the Pixhawk 4 Mini autopilot board, achieving excellent results. However, due to material constraints, we had to switch to the Pixhawk 2.4.8 to complete our final project. Despite the change, we successfully met all our objectives.

1.1.2.2 NuttX OS

NuttX is a real-time operating system (RTOS) designed to be UNIX-compatible and efficient in memory usage. It aims to provide standard operating system interfaces for a rich, multi-threaded development environment on deeply embedded processors. Figure 1.3 illustrates how NuttX interfaces with the components of an autopilot board, resembling the architecture of a personal computer where applications interact with hardware through system calls provided by the RTOS. NuttX is scalable, supporting a range of embedded platforms from tiny to moderate in size.

The documentation highlights NuttX's compliance with standards, likening it to a miniature Linux OS with fewer features but supporting the Executable and Linkable Format (ELF) for customized applications. NuttX offers robust multithreading capabilities, emulating standard Unix processes and threads, and is fully preemptible, allowing tasks or threads to be interrupted

at any time for strict priority scheduling. For I/O management, including USB serial ports, NuttX has its own implementation, such as the $stm32otgfshost.c$ source code file relevant to this work.

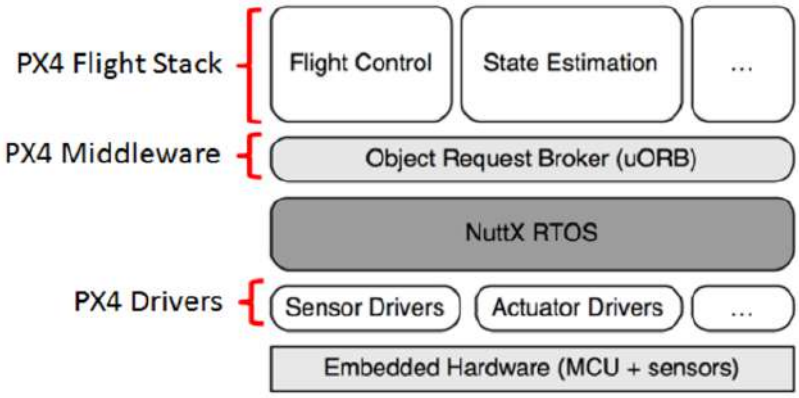


Figure 1.3: Pixhawk software layers [1].

NuttX includes a lightweight, bash-like shell called NuttShell (NSH) for basic user interaction. NSH offers a rich set of commands, scripting capabilities, and the ability to run applications as "built-in." It is implemented as part of a library called nshlib and is optional. If disabled, NuttX can directly load a specific task at startup instead of NSH, as is the case with the px4$simulinkappmodule$.

When NuttX is installed on a Pixhawk board, NuttShell can be accessed via a serial connection by setting the correct baud rate (57600) and using a terminal emulator like PuTTY, TerraTerm, or the NSH console in ground station software such as QGroundControl. Some useful PX4-related commands are documented as well.

These features make NuttX a valuable interface, and understanding its operation is a good starting point for comprehending the environment in which an autopilot software stack functions. For more information on NuttX, refer to the documentation.

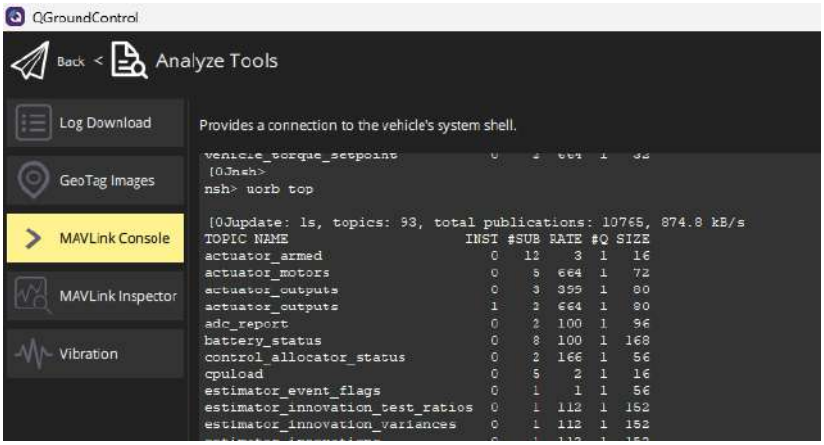


Figure 1.4: NuttShell console view opened from QGroundStation and connected to Pixhawk 2.4.8 autopilot board.

Here in the figure 1.4, this is a simple but very useful debugging command called uorb top. It shows all the topics in our flight controller, the number of instances of each topic, the number of subscribers to each topic, the rate, the queue size, and finally, the size. For more information regarding NuttX, refer to [2].

1.1.2.3 uORB middleware

An autopilot system operates in a multi-task, multi-thread environment where applications are divided into modules that must coordinate various operations. To ensure synchronization, inter-task and inter-thread communication is essential.

In 2015, the micro Object Request Broker (uORB) was developed by the same team behind the Pixhawk design, leveraging the multithreading capabilities of the NuttX operating system. The uORB is an asynchronous messaging API based on shared memory, allowing all modules to share the same memory space.

uORB uses a publish-subscribe design pattern: communication participants are called "nodes," with "publishers" (senders) and "subscribers" (receivers). Publishers share information by advertising a "topic" and updating data at their own pace. Subscribers can subscribe to a topic and either poll for new data or be notified when new data is available.

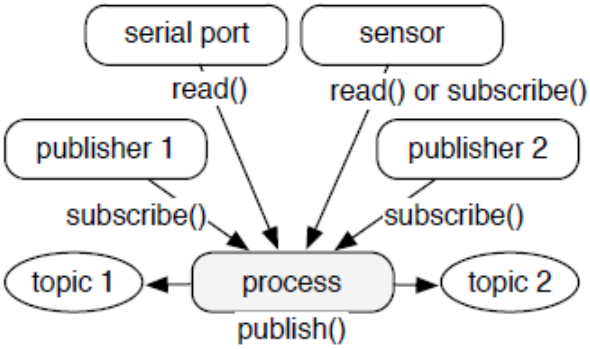


Figure 1.5: A single process can subscribe (consume) and publish multiple topics, allowing it to interface at different rates[2].

A process can act as both a publisher and a subscriber, and can handle multiple topics simultaneously. This setup is illustrated in Figure 1.5. Figure 1.6 shows the precedent output of the uorb top command in NuttShell on our PX4 autopilot, indicating that most sensor topics update at either 664 Hz (every 1.5 ms) or 111 Hz (every 9 ms). If needed, subscribers can limit the update rate.
The uORB framework, along with the task priority setup of the operating system, allows for synchronization between nodes and mixing of low- and high-priority tasks. Additionally, uORB supports multiple independent instances of the same topic, useful for systems with several similar sensors.

With uORB, senders and receivers do not need to know about each other, maintaining an unknown system topology from each module's perspective. Data publication and subscriber access are atomic operations, ensuring data consistency. When data is transmitted, the previous value is replaced, so subscribers always receive the latest data.

1.1.2.4 PX4 flight stack architecture

PX4 was initially designed for UAVs, particularly multirotors, but has evolved into a versatile and modular platform suitable for various robotic systems, using the same codebase across different vehicle types. For instance, in [17] , PX4 is used to control a rover's ground path, and in [37], and it manages an Autonomous Surface Vehicle (ASV) on water.

TOPIC NAME	INST	#SUB	RATE	#Q	SIZE
actuator_armed	0	12	2	1	16
actuator_motors	0	5	664	1	72
actuator_outputs	0	3	399	1	80
actuator_outputs	1	2	664	1	80
adc_report	0	2	101	1	96
battery_status	0	8	100	1	168
control_allocator_status	0	2	167	1	56
cpuload	0	5	2	1	16
estimator_event_flags	0	1	1	1	56
estimator_innovation_test_ratios	0	1	111	1	152
estimator_innovation_variances	0	1	111	1	152
estimator_innovations	0	1	111	1	152
estimator_sensor_bias	0	5	1	1	120
estimator_states	0	1	111	1	216
estimator_status	0	4	111	1	120
estimator_status_flags	0	2	1	1	96
failsafe_flags	0	2	2	1	88
failure_detector_status	0	2	2	1	24
magnetometer_bias_estimate	0	2	47	1	64
position_setpoint_triplet	0	5	21	1	224
px4io_status	0	1	1	1	144
rate_ctrl_status	0	1	664	1	24
rtl_time_estimate	0	4	1	1	24
sensor_accel	0	6	667	8	48
sensor_baro	0	5	71	4	32

Figure 1.6: Published topics on a PX4 autopilot. From left to right the columns represent topic name, multi-instance index, number of subscribers, publishing frequency in Hz, number of lost messages per second (for all subscribers combined), and queue size

Figure 1.7: a rover controlled by PX4.

The PX4 flight stack comprises an estimation and flight control system, leveraging the uORB middleware for internal communication between modules and hardware integration via dedicated drivers. For external communication, PX4 utilizes MAVLink, a lightweight messaging protocol tailored for the drone ecosystem. MAVLink enables communication with ground stations and integration with other components such as companion computers, cameras, proximity sensors, and spraying devices. When running on the NuttX operating system, MAVLink can connect the board with a terminal emulator using NuttShell.

The PX4 system is designed to be reactive, meaning all functionalities are divided into interchangeable and reusable components, communicating asynchronously through self-contained modules (uORB nodes) that share data via topics. This modularity allows rapid and easy replacement of modules, even at runtime. This flexibility is crucial for modifying the flight stack using the MathWorks Embedded Coder Support Package for PX4 autopilots.

The flight stack includes guidance, navigation, and control algorithms. It features estimators for attitude and position, controllers for various airframes, and mixers to translate outputs into motor commands, all included in the Estimation and Control Library (ECL). The Extended Kalman Filter (EKF) algorithm, used for estimation, combines sensor inputs to compute vehicle states, such as attitude from IMU data. The EKF operates in different modes based on sensor inputs, starting with a minimum viable sensor combination and incorporating additional data

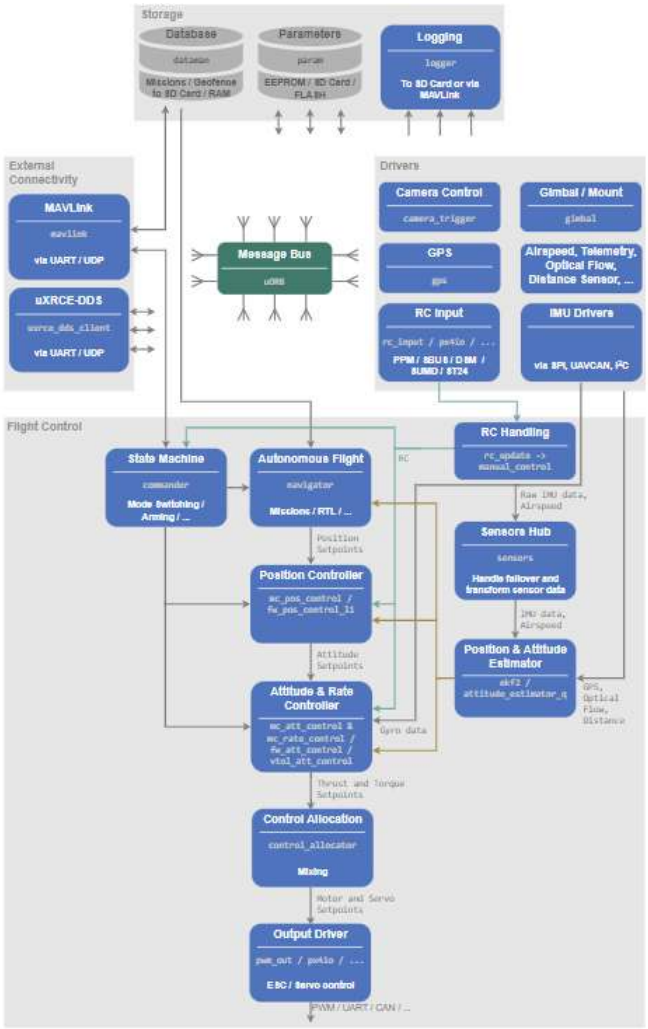


Figure 1.8: flight stack architecture

to estimate more states.

Controllers in the flight stack adjust process variables to match setpoints, using Proportional (P), Proportional-Integrative (PI), and Proportional-Integrative-Derivative (PID) controllers. These controllers are simple and adaptable but may not provide robust solutions for highly dynamic systems.

Mixers translate force commands into motor commands, specific to the vehicle type, ensuring operational limits are not exceeded. This separation of mixer logic from attitude control enhances the software stack's applicability to various robotic platforms.

Each sensor driver, estimator, controller, or mixer operates as a module, communicating through the uORB middleware by publishing or subscribing to topics. Figure 1.8 provides an overview of the flight stack pipeline, illustrating the flow from sensors, manual input (RC), and autonomous flight control (Navigator-Position Controller-Attitude Rate Controller) to motor control (Actuators).
1.9

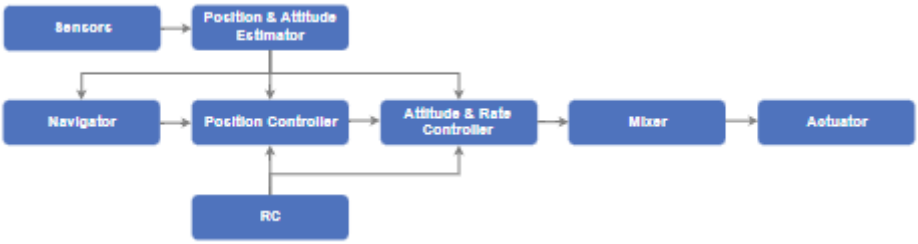


Figure 1.9: building blocks of the flight stack

1.1.2.5 Support Package for PX4 Autopilots by MathWorks Embedded Coder

In academic research, MATLAB/Simulink environment is frequently used as a tool for system modeling and control design [18]. In particular, MATLAB/Simulink is the standard tool for exploiting model-based design approach that consists in the development of embedded software, starting from block models. This approach applied to UAVs development cycle is deeply described in [19]. The most interesting steps in UAV development cycle for the purposes of this thesis are the Software-in-the-loop (SIL) and Processor-in-the-loop (PIL) simulations: during these phases, the production code dedicated to the aerial system control and derived from the model is tested on an emulated environment (SIL) or on the actual autopilot board (PIL), to check its robustness and to evaluate performances and potential optimizations, before proceeding to real flight tests.

To exploit this framework, in the past, there has been a great effort to give the possibility to automatically translate algorithms developed in MATLAB/Simulink on the Pixhawk autopilot series: the original approach can be found in [20]. Nowadays, due to the advances in automated embedded coding achieved by MathWorks, PX4 development is able to support system models and control algorithms, designed with a Model-Based Design approach, without the need for the developers to be proficient in low-level programming [20]. In concrete terms, one of the practical objectives of this final project has been exploring the potentiality of the means made available by the Embedded Coder Support Package for PX4 autopilots for implementing the quad-rotor model and controller design directly on the Pixhawk 4 with automatic code generation [21].

The Embedded Coder Support Package for PX4 autopilots has been available since 2018b MATLAB/Simulink release [21]. The package is directly derived from the Simulink Pilot Support Package [22], used for the studies [19, 23] taken as a reference for this work, that in its turn, was derived from [20]. This development environment enables to access autopilot peripherals from MATLAB/Simulink environment and generate C++ code using the PX4 software stack, building and deploying algorithms while incorporating on-board sensor data. Interfaces for the PX4 architecture components are provided by Simulink blocks that work as inputs and outputs for the model [21]. Using these capabilities, position controller and attitude rate controller modules of the general PX4 architecture are replaced with user-defined algorithms: this is possible thanks to a custom startup script, which needs to be copied on the micro-SD card mounted on the Pixhawk [21]. This script, launched just after NuttX bootstrap [24], disables the default Navigator and Commander PX4 modules, substituting them with a module, called px4_simulink_app, that acts as a "wrapper" for the generated code.

The Embedded Coder [25], leveraging on CMake builder [26], generates and cross-compiles the code from the models developed in Simulink using blocks . This code is then run by the module px4_simulink_app inside the PX4 software stack. In some of the PX4 Simulink blocks, they give the possibility to subscribe or publish uORB topics to retrieve sensors read or to impose a control output (for more details about uORB middleware, refer to the uROB middleware

section). This allows building a model referencing directly to peripherals, sensors, commands of the autopilot board.

For example, the Vehicle Attitude block reads the vehicle_odometry uORB topic and outputs the attitude measurements from the Pixhawk hardware. With its own frequency, the block representing the software module checks if a new message is available on the vehicle_odometry topic. The block outputs the vehicle attitude in roll, pitch and yaw angles [21]. These information are computed into an attitude control system that, for following a reference signal, emits control outputs that, through a mixer matrix, are delivered to a Pulse Width Modulation (PWM) block (for more details about PWM, refer to [27]). Attitude control system and mixer matrix need to be selected, designed and tuned according to the particular airframe in use.The following diagram 1.10 explain in a concrete manner the precedent example.

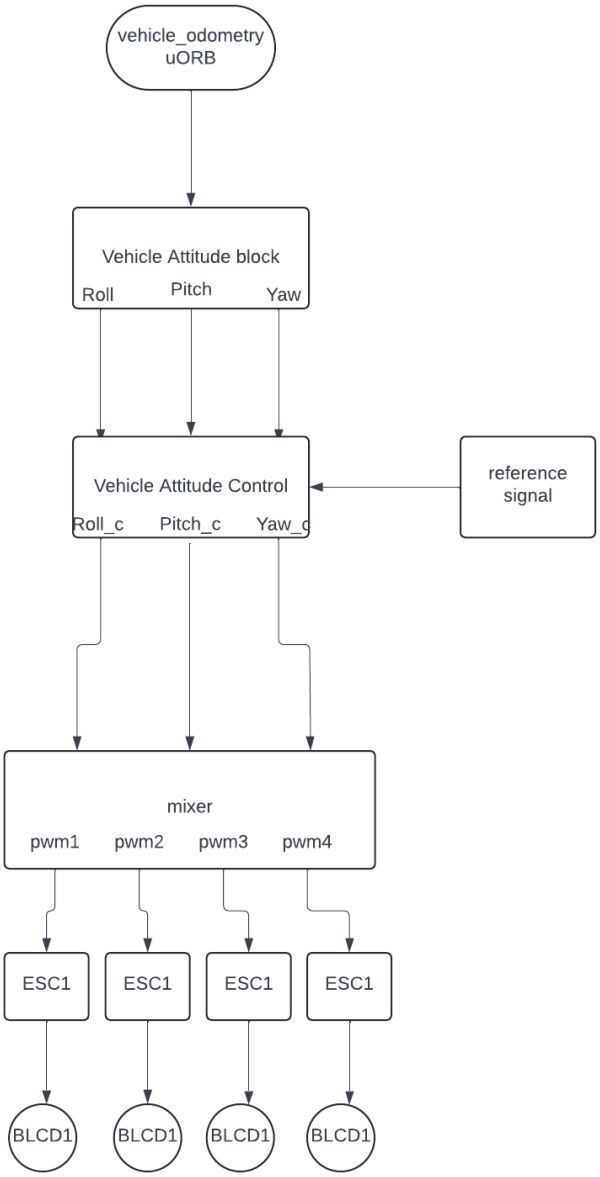


Figure 1.10: diagram of the precedent example

The PWM Block configures the PWM outputs for servo motors: the block accepts the signals from controller as input and writes those values to the selected channels, that are topics on their turn, subscribed by motor drivers modules [21]. In the interconnection between PX4 blocks and attitude controller, this model is ready for building process and deployment on the

selected Pixhawk Series flight controller, that has to be installed on the actual UAV for flight test. Finnaly the main tools that we used in the final project are:

- **PX4 Simulink Blocks & Examples**
  A library of PX4 Simulink blocks was created for the PX4 Platform Support Package to interface with the Pixhawk autopilot. In addition, examples of the PX4 Simulink model are also available in the PX4 Platform Support Package that can be used for developing the plant or controller of a vehicle.

- **PX4 Eclipse**
  The PX4 Eclipse environment provides the platform to build the application from the generated C/C++ codes of the Simulink model and download it to the Pixhawk autopilot.

1.2 Quadrotor mathematical model

1.2.1 Identification of quadrotor configuration

A quadrotor consists of four extended arms, each equipped with a BLDC motor and a fixed-pitch propeller. The propellers are labeled 1 to 4 in a clockwise sequence. The motors are configured such that one pair of propellers rotates counter-clockwise while the other pair rotates clockwise. There are two primary flight configurations for a quadrotor, known as the plus configuration and the cross configuration, as illustrated in Figure 1.11.

In the plus configuration, the quadrotor changes its attitude (roll, pitch, or yaw) by adjusting the rotational speeds of two propellers. Conversely, in the cross configuration, the quadrotor alters its attitude by varying the rotational speeds of all four propellers. This arrangement provides quadrotors in the cross configuration with greater momentum and improved maneuverability compared to those in the plus configuration.

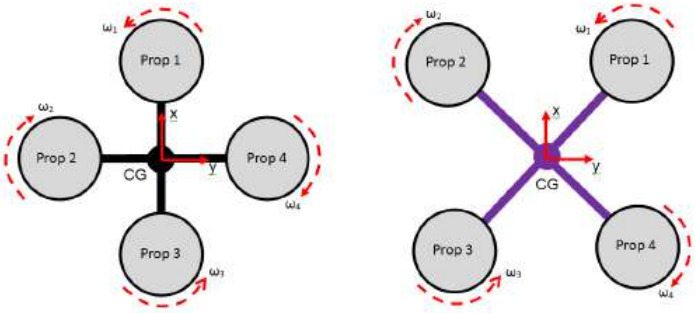


Figure 1.11: Quadrotor in Plus (+) and Cross (X) Configurations

1.2.2 Plus Configuration Flight Mechanism

For a quadrotor to adopt a plus configuration, its arms are aligned with the quadrotor's body x-axis and y-axis (following the right-hand rule orientation). In this configuration, the quadrotor changes the speed of its DC motors to perform translational or rotational maneuvers, as illustrated in Figure 8.
To generate thrust (T) and accelerate the quadrotor along the vertical z-axis, the rotational

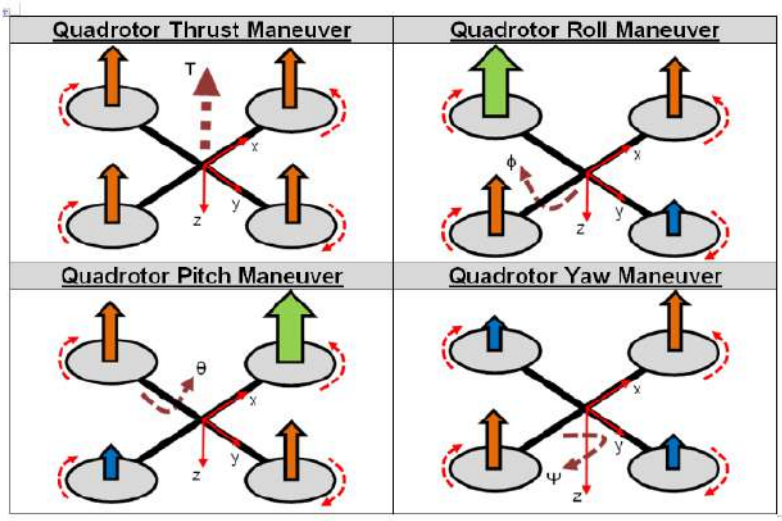


Figure 1.12: Flight Mechanisms for Quadrotor in Plus Configuration

speed of all four propellers is increased or decreased by the same amount. For a roll maneuver, the speed of propeller 2 is increased while the speed of propeller 4 is decreased, producing a torque along the x-axis (). Similarly, for a pitch maneuver, increasing the speed of propeller 1 and decreasing the speed of propeller 3 generates a torque along the y-axis (). Lastly, to perform a yaw maneuver, different speeds are applied to each pair of propellers rotating in the same direction, creating a torque along the z-axis ().

1.2.3 Cross Configuration Flight Mechanism

In the cross configuration, the quadrotor's body x-axis and y-axis are tilted 45 degrees with respect to the quadrotor arms. The quadrotor in cross configuration adjusts the speed of its DC motors to execute translational or rotational maneuvers, as shown in Figure 1.13.

The quadrotor in cross configuration increases the rotational speed of all four propellers uniformly to generate thrust (T) and accelerate along the vertical z-axis. For a roll maneuver, the rotational speed of propellers 3 and 4 is increased, while the speed of propellers 1 and 2 is reduced, creating a torque along the x-axis (τ_ϕ). For a pitch maneuver, increasing the rotational speed of propellers 1 and 4 while reducing the speed of propellers 2 and 3 generates a torque along the y-axis (τ_θ). Finally, by varying the rotational speeds of the counter-rotating pairs of propellers, a torque along the z-axis (τ_ψ) is generated to perform a yaw maneuver.

1.2.4 Notations for Quadrotor Mathematical Model

The notations for the quadrotor's translational and rotational motions are summarized in Table 1.1.

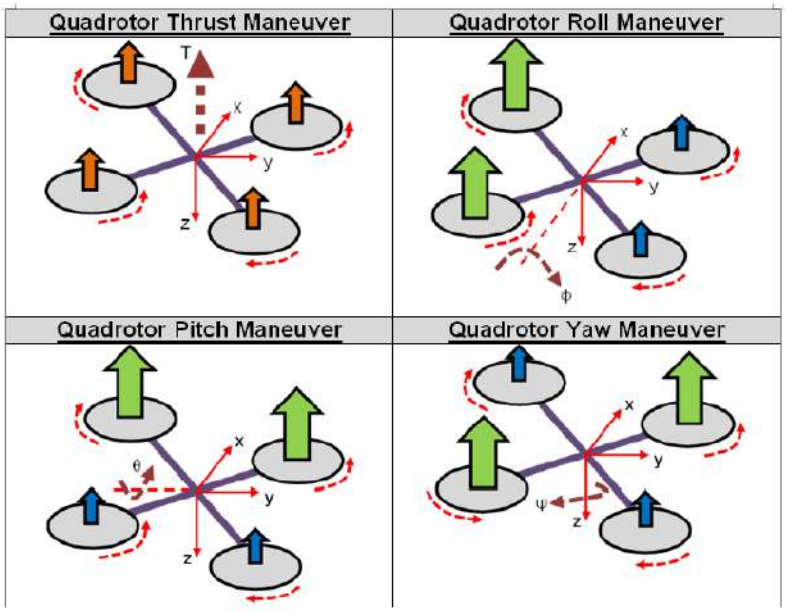


Figure 1.13: Flight Mechanisms for Quadrotor in Cross Configuration.

1.2.5 The rotational matrix

1.2.5.1 Kinematics and Quaternions

Kinematics is the process of describing the motion of objects without focusing on the forces involved. it also involves quaternions, which are used to describe rotations in three-dimensional space. The angular velocities $[\omega_x, \omega_y, \omega_z]$ are fed into a quaternion calculator, resulting in the quaternion vector q. Quaternions are represented as $[q_1, q_2, q_3, q_4]$, where:

- $[q_1, q_2, q_3]$ define the Euler axis in three-dimensional space.
- $[q_4]$ is the angle of rotation around that axis.[28]

1.2.5.2 DCM and Euler Angles

The DCM relates the input angular velocities to the Euler angles using one of 12 permutations of possible rotation sequences. The rows of the DCM show the axes of Frame A represented in Frame B, and the columns show the axes of Frame B represented in Frame A. The angles of rotation ϕ, θ, and ψ are used to rotate from orientation A to orientation B.

1.2.5.3 Euler Angles and Rotation Sequences

Euler angles describe the orientation of a rigid body through three successive rotations about different axes. There are two main types of rotation sequences[28]:

- **Proper Euler Angles**: Involves rotations about three different axes.
- **Tait-Bryan Angles**: Involves rotations where one axis is repeated, but not consecutively.

Table 1.1: Notations for Quadrotor Translational & Rotational Motions.

States	**Description**
x_i	Quadrotor position along the x-axis in the inertia frame.
y_i	Quadrotor position along the y-axis in the inertia frame.
z_i	Quadrotor position along the z-axis in the inertia frame.
$\dot{x}_i$	Quadrotor velocity along the x-axis in the inertia frame.
$\dot{y}_i$	Quadrotor velocity along the y-axis in the inertia frame.
$\dot{z}_i$	Quadrotor velocity along the z-axis in the inertia frame.
$\ddot{x}_i$	Quadrotor acceleration along the x-axis in the inertia frame.
$\ddot{y}_i$	Quadrotor acceleration along the y-axis in the inertia frame.
$\ddot{z}_i$	Quadrotor acceleration along the z-axis in the inertia frame.
x_b	Quadrotor position along the x-axis in the body frame.
y_b	Quadrotor position along the y-axis in the body frame.
z_b	Quadrotor position along the z-axis in the body frame.
x_v	Quadrotor position along the x-axis in the vehicle frame.
y_v	Quadrotor position along the y-axis in the vehicle frame.
z_v	Quadrotor position along the z-axis in the vehicle frame.
u	Quadrotor velocity along the x-axis in the body frame.
v	Quadrotor velocity along the y-axis in the body frame.
w	Quadrotor velocity along the z-axis in the body frame.
ϕ	Quadrotor roll angle with reference to inertia frame axis.
θ	Quadrotor pitch angle with reference to inertia frame axis.
ψ	Quadrotor yaw angle with reference to inertia frame axis.
p	Quadrotor roll rate along the x-axis in the body frame.
q	Quadrotor pitch rate along the y-axis in the body frame.
r	Quadrotor yaw rate along the z-axis in the body frame.

1.2.5.4 Determining Possible Sequences

For a valid sequence of Euler angles:

- There are 3 choices for the first axis.
- There are 2 remaining choices for the second axis.
- There is 1 remaining choice for the third axis.

Thus, the total number of sequences is $3 \times 2 \times 1 = 6$.

Tait-Bryan Angles (6 sequences):

- One axis is repeated, but not consecutively.

- Examples: XYZ, YZX, ZXY.

Hence, there are a total of 12 unique Euler angle sequences for describing 3D rotational motion, and because the Tait-Bryan angles correspond directly to intuitive notions of orientation. Roll, pitch, and yaw describe rotations about the quadrotor's forward axis (X-axis), sideways axis (Y-axis), and vertical axis (Z-axis), respectively, we chose the XYZ form in our final year project.

First Rotation

The first rotation is a rotation of ψ around the Z_0 axis which aligns the Y_0 axis with the Y' axis and the X_0 axis with the X' axis. Its rotation matrix is:

$$R_\psi = \begin{bmatrix} c_\psi & -s_\psi & 0 \\ s_\psi & c_\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

Where: $c_\psi = \cos(\psi)$ and $s_\psi = \sin(\psi)$.

Second Rotation

The second rotation is a rotation of θ around the Y' axis which aligns the X' axis with the X_1 axis and the Z_0 axis with the Z' axis. Its rotation matrix is:

$$R_\theta = \begin{bmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{bmatrix} \tag{1.2}$$

Where: $c_\theta = \cos(\theta)$ and $s_\theta = \sin(\theta)$.

Third Rotation

The third rotation is a rotation of ϕ around the X_1 axis which aligns the Z' axis with the Z_1 axis and the Y' axis with the Y_1 axis. Its rotation matrix is:

$$R_\phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\phi & -s_\phi \\ 0 & s_\phi & c_\phi \end{bmatrix} \tag{1.3}$$

Where: $c_\phi = \cos(\phi)$ and $s_\phi = \sin(\phi)$.

Total Rotation Matrix

Finally, as shown in the figure 1.14 we obtain the total rotation matrix which allows us to go from the body frame to frame the global frame by successively multiplying the three previous rotation matrices:

$$R_b^i = R_\phi \cdot R_\theta \cdot R_\psi = \begin{bmatrix} c_\theta c_\psi & c_\psi s_\theta s_\phi - s_\psi c_\phi & s_\psi s_\phi + c_\psi c_\phi s_\theta \\ c_\theta s_\psi & c_\psi c_\phi + s_\psi s_\theta s_\phi & -c_\psi s_\phi + s_\psi c_\phi s_\theta \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \tag{1.4}$$

Notice: This matrix may also be referred to as a direction cosine matrix (DCM), because the elements of this matrix are the cosines of the unsigned angles between the body-fixed axes and the world axes.

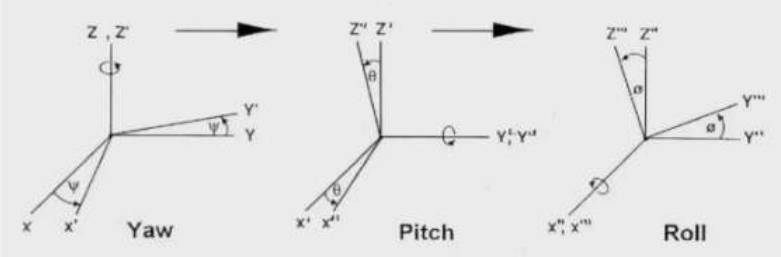


Figure 1.14: Euler-angle rotation sequence

1.2.6 Quadrotor dynamics

The following assumptions were made in deriving the quadrotor dynamics:

- The quadrotor's center of gravity is located at the origin of the body frame.
- The quadrotor is considered a rigid body.
- The quadrotor is symmetrical with respect to the x and y-axes, as described in [29, 30, 31, 32, 33, 34].
- The quadrotor propellers are rigid.
- The thrust and drag exerted on the quadrotor are proportional to the square of the propellers' angular speed.[35]

1.2.7 Aerodynamic Forces

In this section we will talk about the different aerodynamic forces which are:

1.2.7.1 Quadrotor Thrust Force

The thrust generated by the propellers along the z-axis in the body coordinate frame (z_b) can be described by the following equations:

$$T^b = -K_T \left(w_1^2 + w_2^2 + w_3^2 + w_4^2 \right) \tag{1.5}$$

$$T^b = -K_T U_1 \tag{1.6}$$

where:

- K_T is the propeller thrust coefficient.
- U_1 is the thrust control input for the propellers' rotational speed.

1.2.7.2 Quadrotor Roll Moment

The roll moment for the quadrotor along the x-axis in the body coordinate frame can be expressed for the cross configuration as follow:

- **Plus Configuration**

$$\tau_\phi = K_T l_y \left(-w_2^2 + w_4^2\right) \tag{1.7}$$

- **Cross Configuration**

$$\tau_\phi = K_T l_y \left(-w_1^2 - w_2^2 + w_3^2 + w_4^2\right) \tag{1.8}$$

$$\tau_\phi = K_T l_y U_2 \tag{1.9}$$

where:

- l_y is the length of the moment arm along the body y-axis.
- U_2 is the roll control input for the propellers' rotational speed.

1.2.7.3 Quadrotor Pitch Moment

The pitch moment for the quadrotor along the y-axis on the body coordinate frame in cross configurations can be expressed as:

- **Plus Configuration**

$$\tau_\theta = K_T l_x \left(w_1^2 - w_3^2\right) \tag{1.10}$$

- **Cross Configuration**

$$\tau_\theta = K_T l_x \left(w_1^2 - w_2^2 - w_3^2 + w_4^2\right) \tag{1.11}$$

$$\tau_\theta = K_T l_x U_3 \tag{1.12}$$

where:

- l_x is the length of the moment arm on the body x-axis.
- U_3 is the pitch control input for the propellers' rotational velocity.

1.2.7.4 Quadrotor Yaw Moment

The yaw moment for the quadrotor along the z-axis on the body coordinate frame in plus and cross configurations are the same and can be expressed as:

$$\tau_\psi = K_D \left(w_1^2 - w_2^2 + w_3^2 - w_4^2\right) \tag{1.13}$$

$$\tau_\psi = K_D U_4 \tag{1.14}$$

where:

- K_D is the propellers' drag coefficient.
- U_4 is the yaw control input for the propellers' rotational velocity.

1.2.7.5 Summary of Aerodynamic Forces and Moments

The relationship between the aerodynamic forces and the propellers' rotational velocity can be represented in matrix form. The matrices for the quadrotor in plus and cross configurations are expressed as follows:

- **Plus Configuration**

$$\begin{bmatrix} T^b \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} -K_T & -K_T & -K_T & -K_T \\ 0 & -K_T l_y & 0 & K_T l_y \\ K_T l_x & 0 & -K_T l_x & 0 \\ K_D & -K_D & K_D & -K_D \end{bmatrix} \begin{bmatrix} w_2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} \tag{1.15}$$

- **Cross Configuration**

$$\begin{bmatrix} T^b \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} -K_T & -K_T & -K_T & -K_T \\ -K_T l_y & K_T l_y & K_T l_y & -K_T l_y \\ K_T l_x & -K_T l_x & K_T l_x & -K_T l_x \\ K_D & K_D & -K_D & -K_D \end{bmatrix} \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} \tag{1.16}$$

Here, we need to underline that the inverse matrix

$$\begin{bmatrix} -K_T & -K_T & -K_T & -K_T \\ -K_T l_y & K_T l_y & K_T l_y & -K_T l_y \\ K_T l_x & -K_T l_x & K_T l_x & -K_T l_x \\ K_D & K_D & -K_D & -K_D \end{bmatrix}^{-1} \tag{1.17}$$

is called the mixer matrix because, using it, we can find the angular velocity of our system if we have the torque.

1.2.7.6 Quadrotor Equations of Motion

According to Newton's second law of dynamics in the inertial frame:

$$m\ddot{X} = \sum F_{\text{ext}} \tag{1.18}$$

Where:

- $m \in \mathbb{R}^+$ is the total mass of the quadrotor.
- $X = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \in \mathbb{R}^3$ is the position vector of the quadrotor in the inertial frame.
- $\sum F_{\text{ext}} \in \mathbb{R}^3$ is the vector of external forces.

The external forces applied to the quadrotor are:

Gravitational Force:

This is the gravitational force of the earth. It is given by:

$$P = \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{1.19}$$

Where $g \in \mathbb{R}^+$ is the acceleration due to gravity on earth.

Lift Force:

This is the total force generated by the rotation of the four rotor blades. It is directed upwards, meaning it tends to lift the quadrotor. The lift force is calculated in the body frame, so by multiplying in the rotation matrix we get:

$$T^i = \mathrm{R}_b^i \cdot \begin{bmatrix} 0 \\ 0 \\ -T^b \end{bmatrix} \tag{1.20}$$

Where:

- $T = \sum_{i=1}^{4} f_i$ is the total lift force from the four blades.
- f_i is the lift force produced by the rotation of the i-th blade, given by:

$$f_i = b \cdot w_i^2 \tag{1.21}$$

With $b \in \mathbb{R}^+$ being the lift coefficient. By substituting the expression for external forces into equation (1.5), we obtain the following system of differential equations after simplification:

$$\begin{cases} \ddot{x} = \frac{-T}{m}(c_\phi s_\theta c_\psi + s_\phi s_\psi) \\ \ddot{y} = \frac{T}{m}(s_\phi c_\psi - c_\phi s_\theta s_\psi) \\ \ddot{z} = \frac{-T}{m}(c_\phi c_\theta) + g \end{cases} \tag{1.22}$$

1.2.7.7 Rotation Dynamics

According to Newton's second law of dynamics:

$$\frac{d(J\Omega)}{dt} = \sum \Gamma_{\text{ext}} \tag{1.23}$$

And since the angular velocity is expressed in the frame attached to the quadrotor, then:

$$\frac{d(J\Omega)}{dt} = J\dot{\Omega} + \Omega \wedge J\Omega \tag{1.24}$$

Which gives us:

$$J\dot{\Omega} = -\Omega J\Omega + \sum \Gamma_{\text{ext}} \tag{1.25}$$

Where:

- $J = \begin{bmatrix} J_x & 0 & 0 \\ 0 & J_y & 0 \\ 0 & 0 & J_z \end{bmatrix} \in \mathbb{R}^{3\times3}$ is the inertia matrix of the quadrotor.

- $\sum \Gamma_{\text{ext}} \in \mathbb{R}^3$ is the vector of total external moments.

- $\Omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \in \mathbb{R}^3$ is the vector of instantaneous rotational velocities in the quadrotor's frame.

The external moments applied to the quadrotor are:

By substituting the expression for external moments 1.15 or 1.16 into equation 1.25, we obtain the following system of differential equations after simplification:

$$\begin{cases} \dot{p} = \left(\frac{J_y - J_z}{J_x}\right) qr + \frac{\tau_\phi}{J_x} \\ \dot{q} = \left(\frac{J_z - J_x}{J_y}\right) pr + \frac{\tau_\theta}{J_y} \\ \dot{r} = \left(\frac{J_x - J_y}{J_z}\right) pq + \frac{\tau_\psi}{J_z} \end{cases} \tag{1.26}$$

1.2.8 Relation between Euler angles and angular velocities

If a solid body rotates at a constant speed, its angular velocity w^b (angular velocity of quadrotor in body coordinates) is constant. However, the variations of the Euler angles will be variable because they depend on the instantaneous angles between the axes of the two frames. The sequence of Euler angles is obtained from three successive rotations: yaw, pitch, and roll. The variation ψ requires two rotations, θ requires one rotation, and ϕ requires no rotation[36]:

$$w^b = R_\phi R_\theta \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + R_\phi \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} \tag{1.27}$$

Which gives us:

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & s_\phi c_\theta \\ 0 & -s_\phi & c_\phi c_\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{1.28}$$

Here, the vector $\begin{bmatrix} p & q & r \end{bmatrix}^T$ represents the angular velocity components of the quadrotor in the body frame:

- p: Roll rate (angular velocity around the x-axis)

- q: Pitch rate (angular velocity around the y-axis)

- r: Yaw rate (angular velocity around the z-axis)

The transformation of the quadrotor's angular velocities from the body frame coordinate to the inertia frame coordinate can be represented by the following equation:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & s_\phi \frac{s_\theta}{c_\theta} & c_\phi \frac{s_\theta}{c_\theta} \\ 0 & c_\phi & -s_\phi \\ 0 & \frac{s_\phi}{c_\theta} & \frac{c_\phi}{c_\theta} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1.29}$$

Notice: If the pitch angle is equal to $\pi/2$, we will encounter what is called gimbal lock. A gimbal is a pivoted support that allows rotation of an object about a single axis. In a three-dimensional gimbal system, three gimbals are arranged mutually perpendicular to each other, allowing for movement in any direction.

Gimbal lock occurs when two of the three gimbals align, effectively reducing the system to a two-dimensional one. This alignment causes a loss of one degree of freedom, which can lead to problems in controlling and maneuvering the object. In practical terms, this means that certain orientations become impossible to achieve or control accurately.

In our case, we don't have to worry about gimbal lock because our drone is designed for security purposes, so it does not need to align the pitch angle to $\pi/2$ and also for simplification purposes and as most studies in the literature work with a simplified model [37, 38, 39], we assume that the roll and pitch angles are of small amplitude, i.e., $|\phi| \leq \frac{\pi}{6}$ and $|\theta| \leq \frac{\pi}{6}$, which allows us to have $\sin(\phi) \approx \phi$, $\sin(\theta) \approx \theta$, $\cos(\phi) \approx 1$ and $\cos(\theta) \approx 1$. We also assume that the angular velocities around the three axes of the quadrotor are small, and therefore equation (1.16) becomes:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & s_\phi \frac{s_\theta}{c_\theta} & c_\phi \frac{s_\theta}{c_\theta} \\ 0 & c_\phi & -s_\phi \\ 0 & \frac{s_\phi}{c_\theta} & \frac{c_\phi}{c_\theta} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \approx \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1.30}$$

And equation 1.29 becomes:

$$\begin{cases} \ddot{\phi} = \left(\frac{J_y - J_z}{J_x}\right) \dot{\theta}\dot{\psi} + \frac{\tau_\phi}{J_x} \\ \ddot{\theta} = \left(\frac{J_z - J_x}{J_y}\right) \dot{\phi}\dot{\psi} + \frac{\tau_\theta}{J_y} \\ \ddot{\psi} = \left(\frac{J_x - J_y}{J_z}\right) \dot{\phi}\dot{\theta} + \frac{\tau_\psi}{J_z} \end{cases} \tag{1.31}$$

1.2.9 Aerodynamic Effects and Uncertainties

There are many aerodynamic and gyroscopic effects associated with a quadrotor that modify the model presented above. Most of these effects only cause minor disturbances and do not justify being taken into account, even if they are important for the design of a complete system. Blade flapping and induced drag, however, are fundamental effects that are significantly important in understanding the natural stability of quadrotors. These effects are particularly important as they induce forces in the x-y plane of the quadrotor, its underactuated directions, which cannot be easily dominated by a high-gain control.

1.2.9.1 Air Friction

The chassis of the quadrotor as well as the propellers offer resistance to the air. This generates a friction force that opposes the linear and rotational movement of the quadrotor. This force

is proportional to the square of the difference between the speed of the quadrotor and that of the wind, and it depends on the geometry of the quadrotor. Its expression is given by[40, 41]:

$$F_r = \begin{bmatrix} -A_x|\dot{x} - w_x|(\dot{x} - w_x) \\ -A_y|\dot{y} - w_y|(\dot{y} - w_y) \\ -A_z|\dot{z} - w_z|(\dot{z} - w_z) \end{bmatrix} \tag{1.32}$$

Where:

- $A_x \in \mathbb{R}^+$, is the viscous friction coefficient along the X_0 axis;
- $A_y \in \mathbb{R}^+$, is the viscous friction coefficient along the Y_0 axis;
- $A_z \in \mathbb{R}^+$, is the viscous friction coefficient along the Z_0 axis;
- $w_x \in \mathbb{R}^+$, is the wind speed along the X_0 axis;
- $w_y \in \mathbb{R}^+$, is the wind speed along the Y_0 axis;
- $w_z \in \mathbb{R}^+$, is the wind speed along the Z_0 axis.

1.2.9.2 Gyroscopic Effect

The rotational motion of the propeller-rotor combination generates a gyroscopic effect that acts on the quadrotor in the body coordinate frame. The gyroscopic effect is contributed by the rotor's moment of inertia, the rotor's angular velocity, and the body attitude rate, which can be expressed by equation:

$$\mathbf{G}^b = I_{\text{rotor}}(w_b \times \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix})\Omega \quad = I_{\text{rotor}} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \Omega \tag{1.33}$$

where:

- w_b is the angular velocity of the quadrotor (body coordinate frame) during the flight,
- w is the sum of the 4 rotors' rotational velocities (i.e. $w = w_1 + w_2 + w_3 + w_4$),
- I_{rotor} is the rotor moment of inertia given by:

$$I_{\text{rotor}} = \left(\frac{1}{4} m_{\text{motor}} \cdot r_{\text{motor}}^2\right) + \left(\frac{1}{12} m_{\text{prop}} \cdot L_{\text{prop}}^2\right), \tag{1.34}$$

- m_{motor} is the motor mass,
- r_{motor} is the motor radius,
- m_{prop} is the propeller mass, and
- L_{prop} is the propeller length.

Thus,

$$\mathbf{G}^b = I_{\text{rotor}} \begin{bmatrix} q \\ p \\ 0 \end{bmatrix} \Omega \tag{1.35}$$

Assuming the attitude control system functions as intended and effectively regulates the angular dynamics to near-hover conditions, the UAV's body rates approach zero during flight. Given the constant and small moment of inertia of the rotor-propeller combination (Irotor), the product Gb in the final equation remains small as long as the roll and pitch attitude rates in b are maintained near zero. Consequently, the gyroscopic effect's contribution to the quadrotor's total moment is minimal and can be initially neglected during the initial phase of the linear control design approach.

1.2.9.3 Propeller Flapping

The propeller flapping effect is created when the propeller moves horizontally. This movement creates a difference in speed, and thus thrust, between the part of the propeller that attacks the airflow and the part that withdraws from the airflow. This difference in thrust between these elements causes the propeller plane to tilt, which changes the direction of the thrust vector[40, 41].

1.2.9.4 Ground Effect

The ground effect is created when a surface, sufficiently close to the propeller, disturbs the airflow generated by the propeller, thereby improving the thrust of the propeller. At low speed, this effect can be modeled by[42, 41]:

$$\left|\frac{T_{ES}}{T_0}\right| = \frac{1}{1 - \left(\frac{r}{4h}\right)^2} \tag{1.36}$$

Where:

- $T_{ES} \in \mathbb{R}^+$, is the thrust generated by the propeller with the ground effect;
- $T_0 \in \mathbb{R}^+$, is the thrust generated by the propeller without the ground effect;
- $r \in \mathbb{R}^+$, is the radius of the propeller;
- $h \in \mathbb{R}^+$, is the height of the propeller relative to the ground.

It is noted that the thrust increases due to the ground effect, but this effect decreases significantly even at low heights. The thrust increase is only 7% when the height is equal to the radius of the propeller[42].

In this respect, we will treat all these effects as well as all the variations due to the simplifications we have made and any possible modeling errors such as uncertainties and disturbances, and

we group them in the vector $\rho_d \in \mathbb{R}^4$ that we add to equations (1.31) and (1.22) so that they become:

$$\begin{cases} \ddot{x} = \frac{T}{m}(c_\phi s_\theta c_\psi + s_\phi s_\psi) + \rho_{dx} \\ \ddot{y} = \frac{T}{m}(c_\phi s_\theta s_\psi - s_\phi c_\psi) + \rho_{dy} \\ \ddot{z} = \frac{T}{m}(c_\phi c_\theta) - g + \rho_{dz} \end{cases} \tag{1.37}$$

$$\begin{cases} \ddot{\phi} = \left(\frac{J_y - J_z}{J_x}\right)\dot{\theta}\dot{\psi} + \frac{\tau_\phi}{J_x} + \rho_{d\phi} \\ \ddot{\theta} = \left(\frac{J_z - J_x}{J_y}\right)\dot{\phi}\dot{\psi} + \frac{\tau_\theta}{J_y} + \rho_{d\theta} \\ \ddot{\psi} = \left(\frac{J_x - J_y}{J_z}\right)\dot{\phi}\dot{\theta} + \frac{\tau_\psi}{J_z} + \rho_{d\psi} \end{cases} \tag{1.38}$$

1.3 Controller design

1.3.1 Pole placement and LQR Control Techniques

1.3.1.1 State-Space Model

To put the quadrotor equations into state-space form, we choose the state vector:

$$X = [x \quad y \quad \dot{x} \quad \dot{y} \quad \phi \quad \theta \quad \psi \quad \dot{\phi} \quad \dot{\theta} \quad \dot{\psi} \quad z \quad \dot{z}]^T \tag{1.39}$$

And to simplify the calculations, we chose to work with the plus configuration:

$$\begin{cases} U_1 = T = K_T(w_1^2 + w_2^2 + w_3^2 + w_4^2) \\ U_2 = \tau_\phi = K_T \cdot l_y(w_4^2 - w_2^2) \\ U_3 = \tau_\theta = K_T \cdot l_x(w_3^2 - w_1^2) \\ U_4 = \tau_\psi = K_D \cdot (-w_1^2 + w_2^2 - w_3^2 + w_4^2) \end{cases} \tag{1.40}$$

The obtained state representation is as follows:

$$\begin{cases} \dot{x}_1 = x_3 \\ \dot{x}_2 = x_4 \\ \dot{x}_3 = \frac{U_1}{m}(c_\theta s_\psi s_\phi - s_\theta c_\phi) + \rho_{dy} \\ \dot{x}_4 = \frac{U_1}{m}(c_\theta s_\psi c_\phi + s_\theta s_\phi) + \rho_{dx} \\ \dot{x}_5 = x_8 \\ \dot{x}_6 = x_9 \\ \dot{x}_7 = x_{10} \\ \dot{x}_8 = \left(\frac{J_y - J_z}{J_x}\right) x_9 x_{10} + \frac{U_2}{J_x} + \rho_{d\phi} \\ \dot{x}_9 = \left(\frac{J_z - J_x}{J_y}\right) x_8 x_{10} + \frac{U_3}{J_y} + \rho_{d\theta} \\ \dot{x}_{10} = \left(\frac{J_x - J_y}{J_z}\right) x_8 x_9 + \frac{U_4}{J_z} + \rho_{d\psi} \\ \dot{x}_{11} = x_{12} \\ \dot{x}_{12} = g - \frac{U_1}{m}(c_\theta c_\psi) + \rho_{dz} \end{cases} \tag{1.41}$$

1.3.1.2 1st Lyapunov method

Now, to work with these two methods, we need to linearize around the equilibrium point, which is the origin. Applying linearised equations to this LQR and pole placement design was a bit

difficult since their state-space equations would not conform to the format of a traditional state space. According to [43], Taylor series could be applied in linearising non-linear equations. However, in the LQR design, linearisation was simply achieved by approximating 1.41. Using the linearisation process removes the offsets from the dynamics of the system. These offsets are accounted for in the controller design after the gains are calculated and found to achieve satisfactory results [18]. The same technique was implemented in this application to design the LQR controller. The following equations show the linear equations that were achieved after applying this technique:

$$\begin{cases} \ddot{\phi} = \frac{1}{I_x} U_2 \\ \ddot{\theta} = \frac{1}{I_y} U_3 \\ \ddot{\psi} = \frac{1}{I_z} U_4 \\ \ddot{x} = u_x \frac{1}{m} U_1 \\ \ddot{y} = u_y \frac{1}{m} U_1 \\ \ddot{z} = u_z \end{cases} \tag{1.42}$$

where

$$\begin{cases} u_x = \cos\theta \sin\phi \cos\psi + \sin\phi \sin\psi \\ u_y = \cos\theta \sin\phi \sin\psi - \sin\phi \cos\psi \\ u_z = g - (\cos\phi \cos\theta) \frac{1}{m} U_1 \end{cases} \tag{1.43}$$

Another technique that was adopted in this LQR and pole placement design was the reduction technique [18]. This was implemented in order to simplify the mathematical calculations here. With this technique, the state vector which is given as $X = (\phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}, x, y, z, \dot{x}, \dot{y}, \dot{z})^T$, though consisting of 12 members, will be reduced into the following: altitude, attitude, and position controls as in 1.39. The altitude control would be characterised by $(z, \dot{z})^T$ while the attitude control would be characterised by $(\phi, \theta, \dot{\phi}, \dot{\theta}, \dot{\psi})^T$. On the other hand, the position control would be characterised by $(x, y, \dot{x}, \dot{y})^T$. After linearization at the origine :

$$\mathbf{X} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \phi \\ \theta \\ \psi \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ z \\ \dot{z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{1.44}$$

we obtain the following matrices:

For the position:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \\ \dot{x_4} \end{bmatrix} = A_{xy} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + B_{xy} \begin{bmatrix} U_x \\ U_y \end{bmatrix} \tag{1.45}$$

$$A_{xy} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, B_{xy} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ a & 0 \\ 0 & a \end{bmatrix} \tag{1.46}$$

For the attitude:

$$\begin{bmatrix} \dot{x_5} \\ \dot{x_6} \\ \dot{x_7} \\ \dot{x_8} \\ \dot{x_9} \\ \dot{x_{10}} \end{bmatrix} = A_{ypr} \begin{bmatrix} x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix} + B_{ypr} \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix} \tag{1.47}$$

$$A_{ypr} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, B_{ypr} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{1}{J_x} & 0 & 0 \\ 0 & \frac{1}{J_y} & 0 \\ 0 & 0 & \frac{1}{J_z} \end{bmatrix} \tag{1.48}$$

For the altitude:

$$\begin{bmatrix} \dot{x_{11}} \\ \dot{x_{12}} \end{bmatrix} = A_z \begin{bmatrix} x_{11} \\ x_{12} \end{bmatrix} + B_z \begin{bmatrix} U_z \end{bmatrix} \tag{1.49}$$

$$A_z = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, B_z = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.50}$$

where symbol a in matrix B_{xy} is equal to $\frac{1}{m}U_1$

1.3.1.3 Pole placement

Now, for the pole placement, finding the gain is relatively easy. We just use the MATLAB function 'place' and choose the poles. For the xy plane, the poles are $[-1, -2, -3, -4]$, for

attitude, the poles are $[-1,-2,-3,-4,-5,-6]$, and for altitude, the poles are $[-1,-2]$. The poles are chosen so that we can get the fastest dynamics possible. After executing the MATLAB code, we find the following results:

$$K1 = \begin{bmatrix} 0.6008 & 0.8003 & 0.0317 & 0.0127 \\ 0.0242 & 0.0104 & 1.5992 & 1.1997 \end{bmatrix} \tag{1.51}$$

$$K2 = \begin{bmatrix} 0.9000 & 0.5250 & 0.0000 & 0.0000 & -0.0000 & -0.0000 \\ 0.0000 & 0.0000 & 2.2500 & 0.8250 & -0.0000 & -0.0000 \\ -0.0000 & -0.0000 & -0.0000 & -0.0000 & 0.0026 & 0.0039 \end{bmatrix} \tag{1.52}$$

$$K3 = \begin{bmatrix} 2.0000 & 3.0000 \end{bmatrix} \tag{1.53}$$

- **Simulation Results**

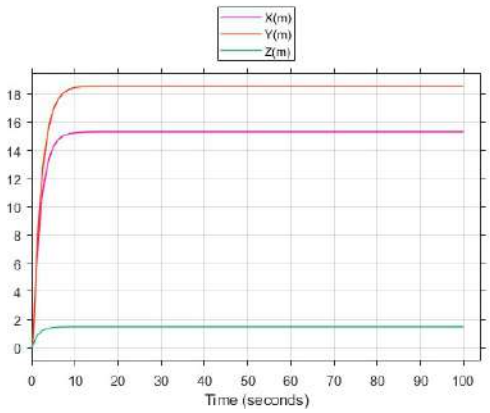
  We designed two systems for our quadrotor using Simulink. The first one is for the linear system to test the gains, and the second one is for the nonlinear system. Both blocks take the commands of position x, y, z, yaw and output the system position, yaw, pitch, roll of the system, and the angular velocities.

Results of simulation

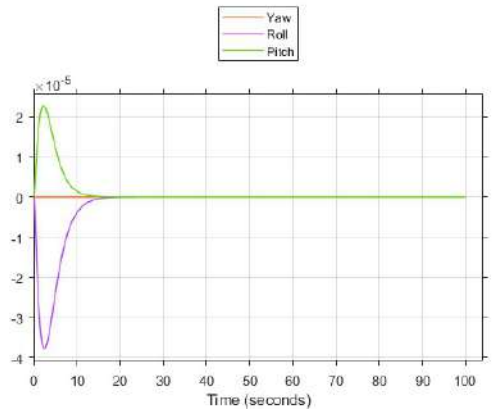
- **Linear Model**

  In Figure 1.16,Here we can see that the position is not tracking the setpoint, which is why we add a precompensator after that we can see In Figure 1.15, the position is perfectly tracking the setpoints, which are $x = 1$, $y = 2$, and $z = 3$, and also the yaw, pitch, and roll angles are satisfying the conditions in Equation 1.30.

  In Figure 1.17, we can see that the four angular velocities are giving us very good results. Our brushless motors, as discussed in the chapter on identification, have a maximum angular velocity of 363 tr/s. In the four plots, the maximum is about 255 tr/s, so the results are acceptable. 1.30.

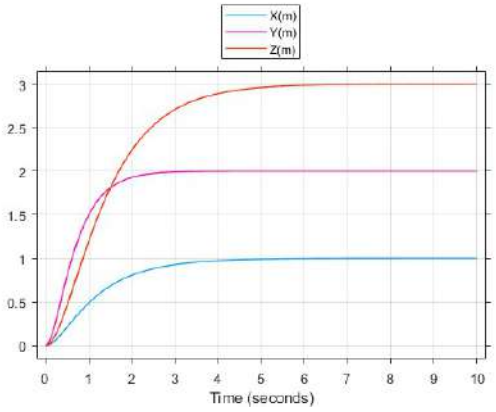


(a) XYZ of our Linear system

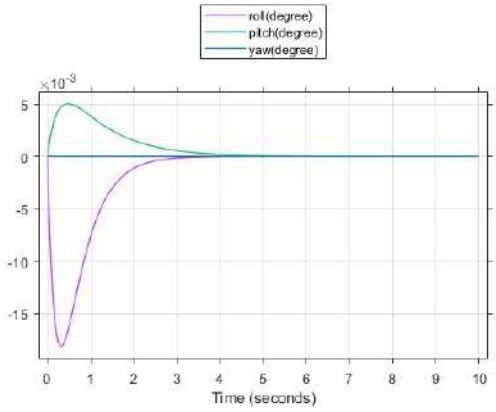


(b) yaw pitch roll Linear system

Figure 1.15: position and attitude of our Linear system

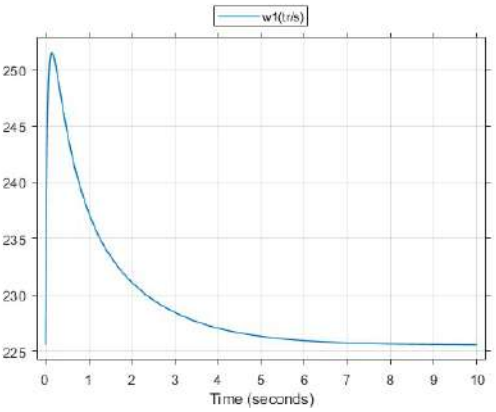


(a) XYZ of our Linear system with compensator

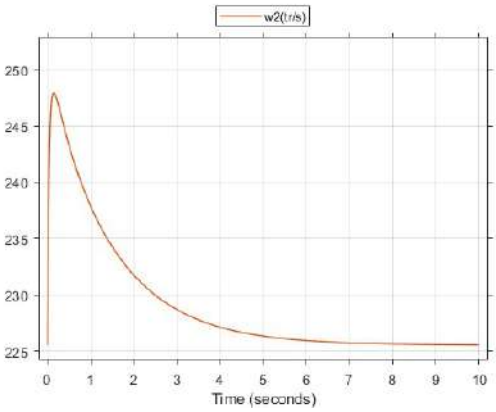


(b) yaw pitch roll Linear system with compensator

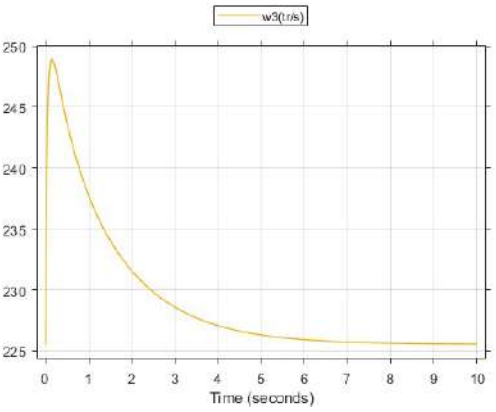
Figure 1.16: Position and attitude of our linear system with compensator



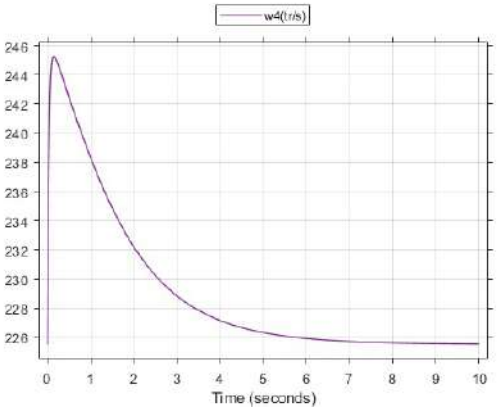
(a) angular velocity of motor 1



(b) angular velocity of motor 2



(c) angular velocity of motor 3

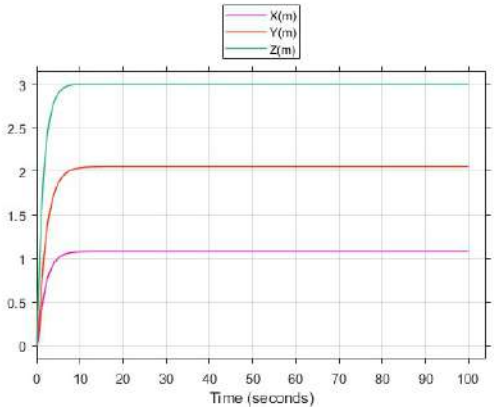


(d) angular velocity of motor 4

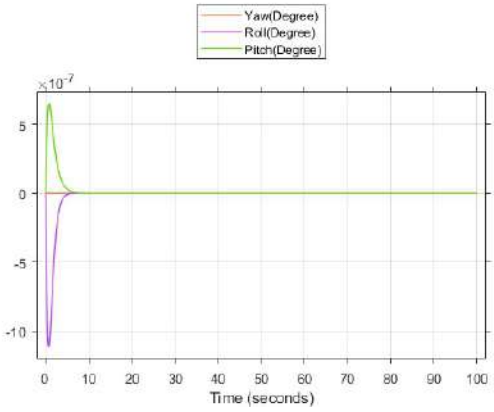
Figure 1.17: angular velocity of the four motors

Nonlinear Model

As we can see in 1.18, our nonlinear model gives us good results With some steady error because the precompensator we designed was based on the linear equations. The condition regarding the angles in Equation 1.30 is verified. In Figure 1.19, we can see that the four angular velocities are good results for the nonlinear system. Our brushless motors, as discussed in the chapter on

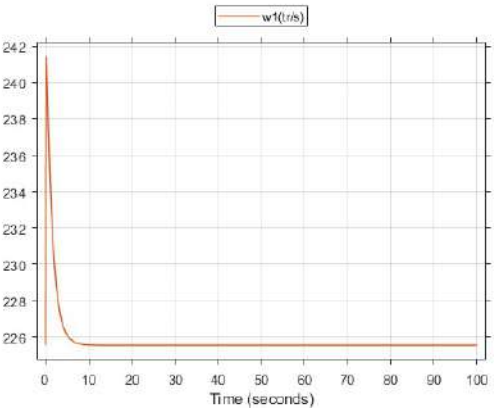


(a) XYZ of our Nonlinear system with compensator

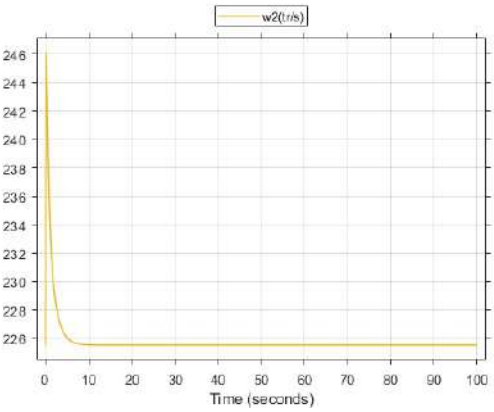


(b) yaw pitch roll of our Nonlinear system with compensator

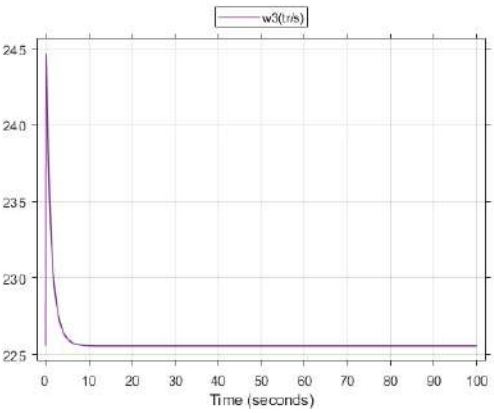
Figure 1.18: postion and attitude of our Nonlinear system with compensator



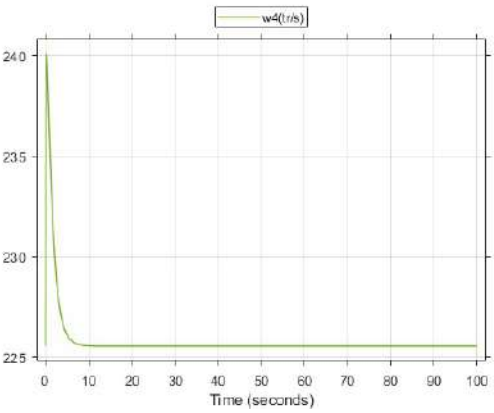
(a) angular velocity of motor 1



(b) angular velocity of motor 2



(c) angular velocity of motor 3



(d) angular velocity of motor 4

Figure 1.19: angular velocity of the four motors

identification, have a maximum angular velocity of 363 tr/s. In the four plots, the maximum angular velocity reaches about 246 tr/s, indicating that the results are acceptable.

1.3.1.4 LQR design

In LQR designs, the system's performance index is characterised by a cost function J for which the controller seeks to minimise [43]. This cost function is given by the formula:

$$J = \int_0^\infty [x^T Q x + u^T R u] dt \tag{1.54}$$

where Q is the state weighting matrix with real symmetry and positive semi-definite in nature. R is the control weighting matrix of real symmetry but positive definite in nature [44]. These weighting matrices help determine the relative importance of the existing error as well as the energy expenditure of the system [45]. It is, therefore, important that, for a successful LQR design, these parameters be chosen accurately. Here, a hybrid form of the classical approach based on the Bryson's method [46] and the trial-and-error methods are combined. This method was chosen for its ability to offset the disadvantages of just using the trial-and-error or Bryson's method. The Bryson's method was first used in determining the initial Q and R weighting matrices. The trial-and-error, then, was relied on to fine-tune these two parameters to achieve a better performance of the controller [47]. The Bryson's Rule: According to this rule, Q and R are diagonal matrices whose diagonal elements are, respectively, expressed as the reciprocals of the squares of the maximum acceptable values of the state variable (x) and the input control variable (u). The diagonal elements Q_{ii} of matrix Q, thus, can be written as [47]:

$$Q_{ii} = \frac{1}{(\text{maximum acceptable value of } X_i)^2} \tag{1.55}$$

where $i = (1, 2, 3, \ldots)$ The diagonal elements R_{jj} of matrix R, also, can be written as [44]:

$$R_{jj} = \frac{1}{(\text{maximum acceptable value of } u_j)^2} \tag{1.56}$$

where $j = (1, 2, 3, \ldots k)$ Applying Bryson's rule to the state-space equation for attitude, as in [45], the following initial Q and R values were obtained:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.57}$$

$$R = 0.3 \tag{1.58}$$

Also applying Bryson's rule to the state-space equation for position, the following initial Q and R values were obtained;

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad R = 0.5 \tag{1.59}$$

Finally applying Bryson's rule to the altitude, the following initial Q and R values were achieved;

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad R = 1 \tag{1.60}$$

Obtaining the feedback gain matrix through Riccati equation
After obtaining the Q and R matrices above, they had to be substituted into the algebraic Riccati equation, to solve for P [8].

$$A \times P + P \times A - P \times B \times R^{-1} \times B^T \times P + Q = 0 \tag{1.61}$$

With P solved, the feedback gain matrix (K) would then be calculated using [8].

$$K = R^{-1} \times B^T \times P \tag{1.62}$$

MATLAB was used as it provides a convenient way of solving for K by just using the following command

$$K = lqr(A, B, Q, R) \tag{1.63}$$

The MATLAB command, was also used to derive the K values for attitude, position, and altitude controllers. In the altitude controller, for instance, where

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad R = 1 \tag{1.64}$$

the values obtained for K are:

$$K_1 = \begin{bmatrix} 1.0000 & 1.1832 & -0.0000 & -0.0000 \\ 0.0000 & 0.0000 & 1.0000 & 1.1832 \end{bmatrix} \tag{1.65}$$

$$K_2 = \begin{bmatrix} 1.0000 & 122.4786 & -0.0000 & -0.0000 & -0.0000 & -0.0000 \\ -0.0000 & -0.0000 & 1.0000 & 122.4786 & 0.0000 & 0.0000 \\ -0.0000 & -0.0000 & 0.0000 & 0.0000 & 1.0000 & 16.1555 \end{bmatrix} \tag{1.66}$$

$$K_3 = \begin{bmatrix} 1.0000 & 1.7321 \end{bmatrix} \tag{1.67}$$

Simulation results

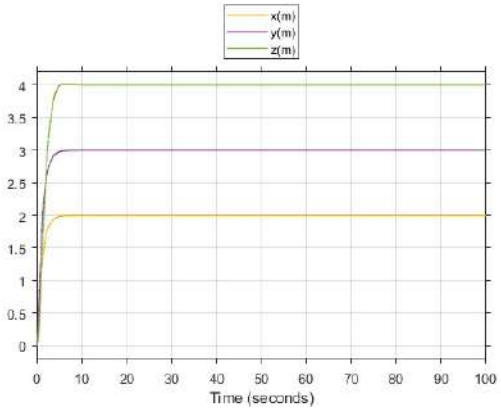
- **Linear system**

The structure of our Simulink blocks remains the same as before 1.30. These blocks accept commands for position $x, y, z,$ yaw and provide the system's position, yaw, pitch, roll, and angular velocities as output. In Figure 1.20, we can see that the position is perfectly tracking the setpoints, which are $x = 2$, $y = 3$, and $z = 4$. Additionally, the yaw, pitch, and roll angles are satisfying the conditions in Equation 1.30.

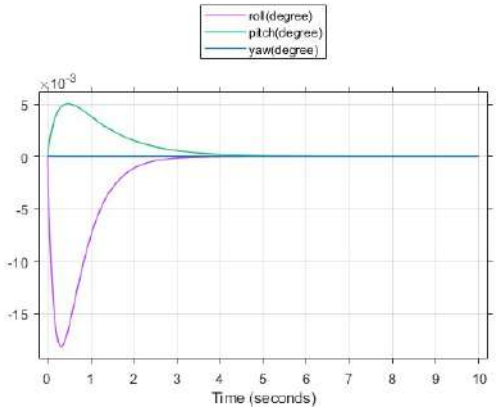
In Figure 1.21, we can see that the four angular velocities are giving us very good results. Our brushless motors, as discussed in the chapter on identification, have a maximum angular velocity of 363 tr/s. In the four plots, the maximum is about 247 tr/s, making these results acceptable.

- **Noninear system**

In Figure 1.22, we can see that the position is perfectly tracking the setpoints, which are $x = 2$, $y = 3$, and $z = 4$. Additionally, the yaw, pitch, and roll angles are satisfying the conditions specified in Equation 1.30 . In Figure 1.23, we can see that the maximum value of our angular velocity is 750 tr/s, which is significantly greater than 363 tr/s. This is why we will not apply this method in our flight controller.

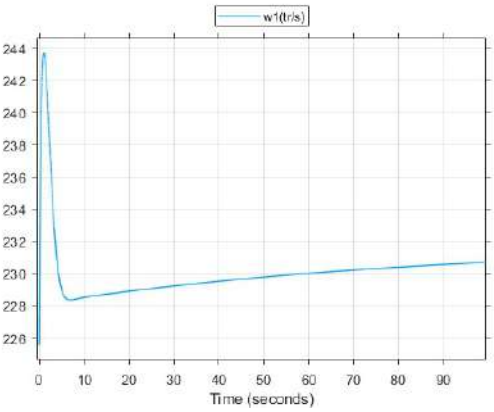


(a) X Y Z of our Linear system

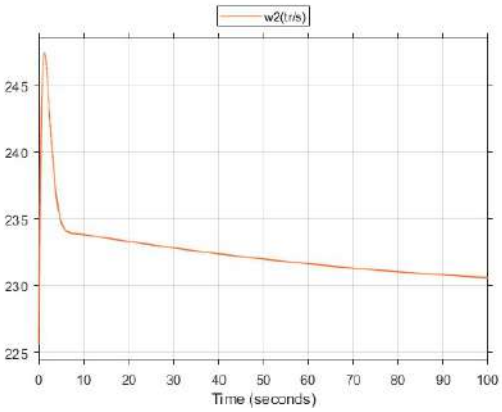


(b) yaw pitch roll of our Linear system

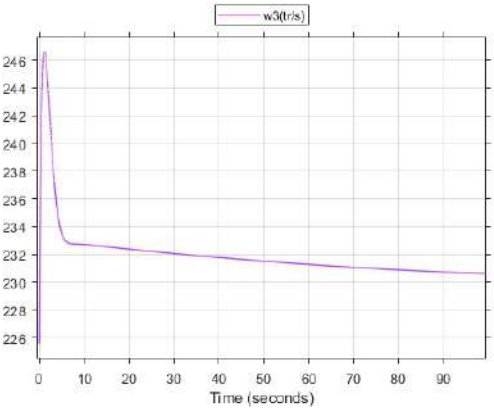
Figure 1.20: position and attitude of our Linear system



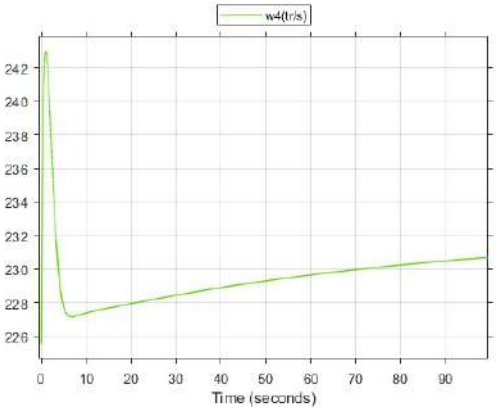
(a) angular velocity of motor 1



(b) angular velocity of motor 2



(c) angular velocity of motor 3



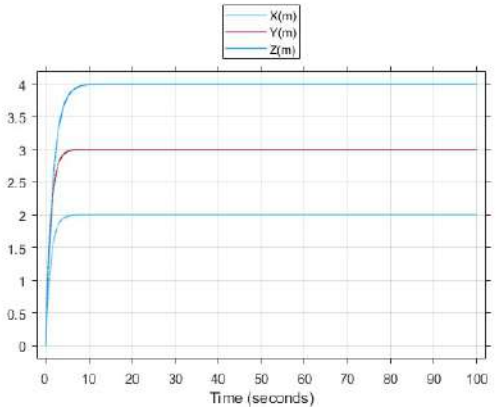
(d) angular velocity of motor 4

Figure 1.21: angular velocity of the four motors

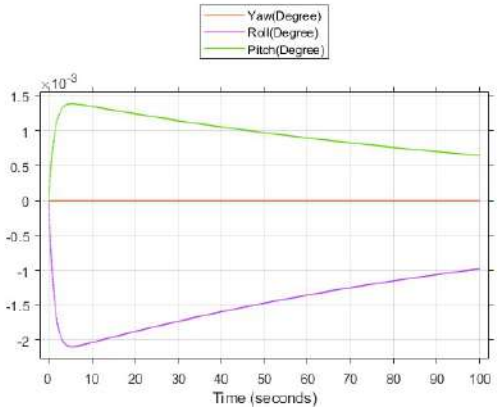
1.3.2 Overview of PID controller

1.3.2.1 PID components

The quadrotor model in this final project uses a PID controller, as elaborated in [48], [49], and [50], to stabilize the attitude during flight. A PID controller consists of three tunable gain values: Proportional gain (i.e., K_P), Integral gain (i.e., K_I), and Derivative gain (i.e., K_D), as shown in Figure 1.24. The transfer function of a PID controller can be represented by equation

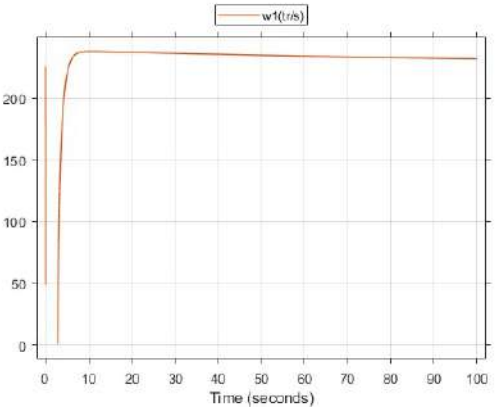


(a) X Y Z of our nonLinear system

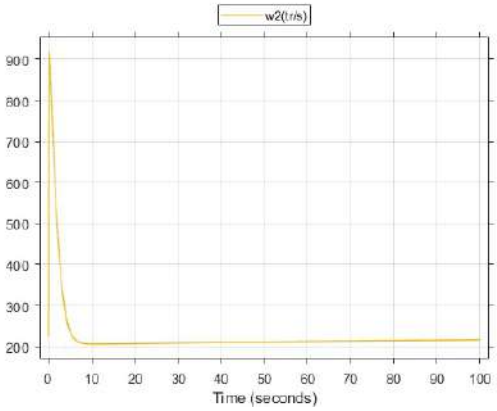


(b) yaw pitch roll of our Nonlinear system

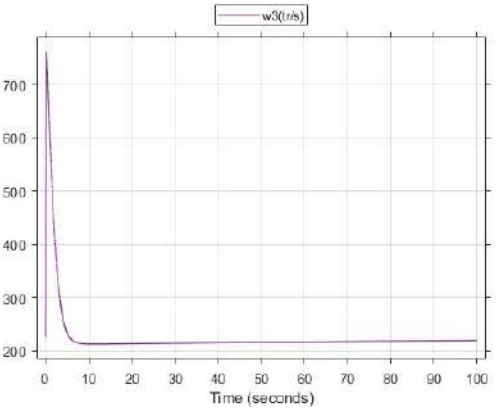
Figure 1.22: position and attitude of our Nonlinear system



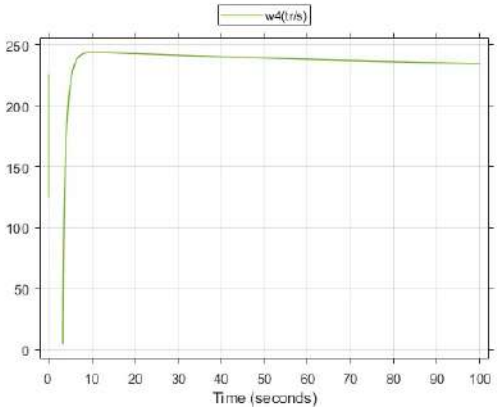
(a) angular velocity of motor 1



(b) angular velocity of motor 2



(c) angular velocity of motor 3



(d) angular velocity of motor 4

Figure 1.23: angular velocity of the four motors

1.68.

$$G(s) = K_P + \frac{K_I}{s} + K_D \cdot s \tag{1.68}$$

Each gain in the PID controller can be tuned to modify a particular transient response parameter of the feedback system (see Figure 1.25). The effects of increasing each gain value separately are elaborated below:

1. **Proportional Gain,** K_P

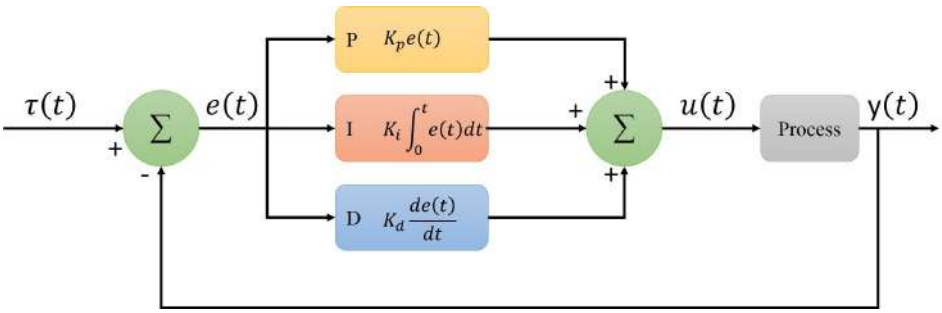


Figure 1.24: Illustration of System designed with PID Controller

The K_P value is increased to reduce the time required for the output signal to reach the desired signal (i.e., system response time, t_r). By increasing the K_P value alone in the PID controller, a steady-state error can be reduced and expected to be between the desired signal and the output signal. However, setting an overly high K_P value will also propagate any inherent disturbance signal within the system and cause the system to undergo unstable oscillations.

2. **Integral Gain,** K_I
   The K_I value is increased to eliminate the steady-state error of the feedback system. However, as the integral term introduces a pole at the origin of an S-plane plot, the system might become increasingly unstable when the K_I value is increased (i.e., the system will become increasingly oscillatory in the steady-state).

3. **Derivative Gain,** K_D
   The K_D value is increased to reduce the overshoot, M_P, and the settling time, t_d, of the feedback system's output signal. Although derivative control does not affect the steady-state error directly, it introduces damping to the feedback system. This allows the system to use a larger K_P value, which improves the system's steady-state performance. As described in [49] and [50], "derivative control operates on the rate of change of the actuating error and not the actuating error itself; this mode is never used alone." Therefore, K_D gain is generally used in combination with K_P and K_I control actions.

1.3.2.2 Advantages of PID Control for Drone Design

Using PID control for the design of a drone controller offers several advantages. Firstly, simplicity and ease of implementation are notable benefits. PID controllers are straightforward to design, implement, and understand, making them accessible to many practitioners due to their fundamental nature taught in engineering. Secondly, proven effectiveness is a significant advantage. PID control has been proven effective in stabilizing drone orientation and position in numerous studies and practical applications. It demonstrates reliable performance and robustness, handling disturbances and uncertainties well to ensure stable flight in various conditions. Thirdly, PID control provides flexibility. It allows easy tuning of proportional, integral, and derivative gains to achieve desired performance and can be combined with other control methods for enhanced performance. Lastly, stability and precision are key strengths. PID controllers are effective at maintaining stable hover and accurate trajectory following, achieving high accuracy in maintaining desired setpoints for flight parameters.

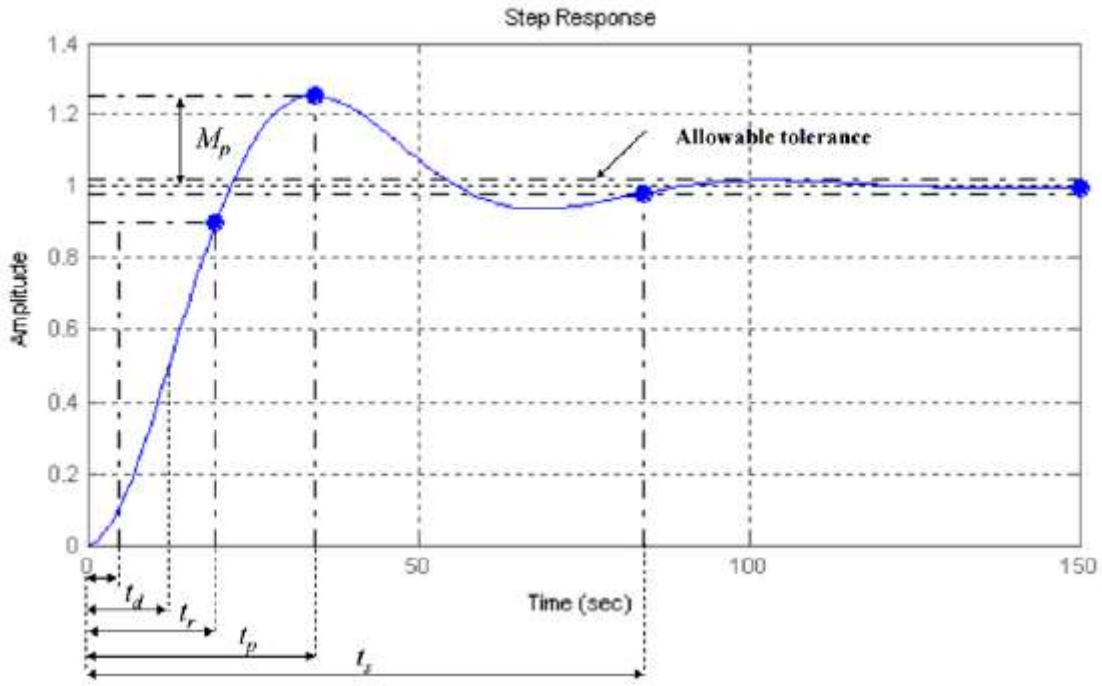


Figure 1.25: Transient Response for a Feedback System (from [3]).

1.3.3 Cascade Control

1.3.3.1 Principle

Cascade control involves multiple nested control loops, where each loop has its own specific measurement and control objectives. In this system, we have three nested control loops: the first controller for position, the second for velocity, and the third for attitude. The position control loop (C1(p)) sets the desired velocities for the velocity control loop (C2(p)). The velocity control loop adjusts the desired attitudes for the attitude control loop (C3(p)). Finally, the attitude control loop sets the desired attitude rates for the attitude rate control loop (C4(p)).

1.3.3.2 Example of two controllers

For example if we had two controllers, Each controller in the hierarchy receives its setpoint from the output of the previous controller and uses its measurement signals to minimize the control errors. The secondary controllers (inner loops) change quickly, while the primary controllers (outer loops) change slowly. This hierarchical structure ensures that disturbances affecting the inner loops do not propagate to the outer loops, thereby enhancing the stability and performance of the control system.

The diagram1.26 illustrates this cascade structure with the control loops and their interactions clearly labeled.

If the intermediate variable is the controlling variable of $H1(p)$, it is referred to as "cascade on the controlling variable". Otherwise, it is referred to as "cascade on an intermediate variable". This type of regulation is justified when the system has a large inertia with respect to a disturbance on the controlling variable or on an intermediate variable. The internal loop should be tuned first, followed by the external loop with the slave controller closed.

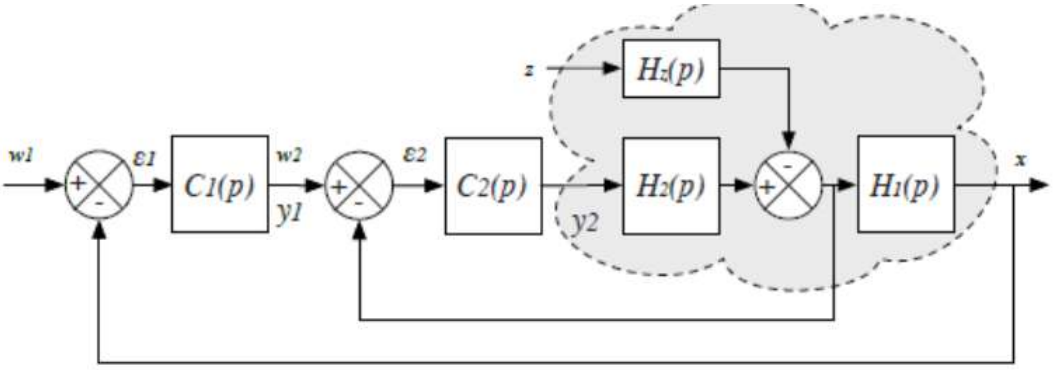


Figure 1.26: Cascade control structure

1.3.4 Ziegler-Nichols closed-loop tuning method

The Ziegler-Nichols closed-loop tuning method allows you to use the ultimate gain value, K_u, and the ultimate period of oscillation, P_u, to calculate K_c. It is a simple method of tuning PID controllers and can be refined to give better approximations of the controller. You can obtain the controller constants K_c, T_i, and T_d in a system with feedback. The Ziegler-Nichols closed-loop tuning method is limited to tuning processes that cannot run in an open-loop environment.

Determining the ultimate gain value, K_u, is accomplished by finding the value of the proportional-only gain that causes the control loop to oscillate indefinitely at steady state. This means that the gains from the I and D controllers are set to zero so that the influence of P can be determined. It tests the robustness of the K_c value so that it is optimized for the controller. Another important value associated with this proportional-only control tuning method is the ultimate period (P_u). The ultimate period is the time required to complete one full oscillation while the system is at steady state. These two parameters, K_u and P_u, are used to find the loop-tuning constants of the controller (P, PI, or PID). To find the values of these parameters, and to calculate the tuning constants, we use the following procedure:

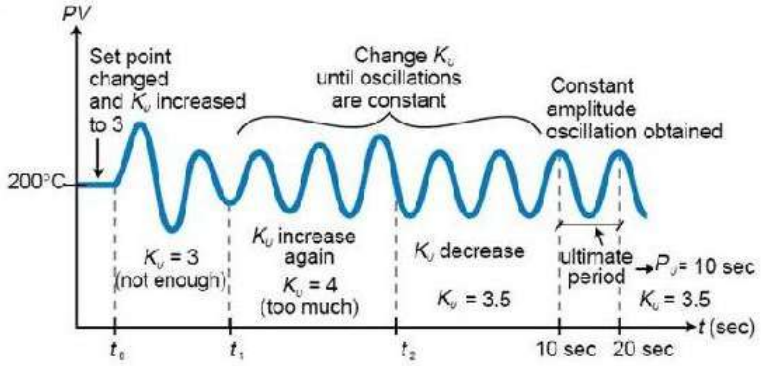


Figure 1.27: System tuned using the Ziegler-Nichols closed-loop tuning method

1.3.4.1 Closed Loop (Feedback Loop)

Remove integral and derivative action. Set integral time (T_i) to 999 or its largest value and set the derivative controller (T_d) to zero.

Create a small disturbance in the loop by changing the set point. Adjust the proportional gain, increasing and/or decreasing, until the oscillations have constant amplitude.

Record the gain value (K_u) and period of oscillation (P_u).

	K_c	T_i	T_d
P	$K_u/2$	-	-
PI	$K_u/2.2$	$P_u/1.2$	-
PID	$K_u/1.7$	$P_u/2$	$P_u/8$

Table 1.2: Ziegler-Nichols Tuning Parameters

The Ziegler-Nichols approach offers several advantages, including the ease of experimentation, as it only requires changing the P controller, and the inclusion of the dynamics of the entire process, which provides a more accurate representation of how the system behaves. However, there are also disadvantages: the experiment can be time-consuming, and there is a risk of venturing into unstable regions while testing the P controller, which could cause the system to become uncontrollable.

1.3.5 Implemented controllers

The control modeling of our quadrotor is summarized and presented in the block diagram, as shown in Figure 1.28 below. A hierarchical approach that was presented in [47] divides the control tasks into different levels. Each level is responsible for specific aspects of the control process, with distinct bandwidth requirements.

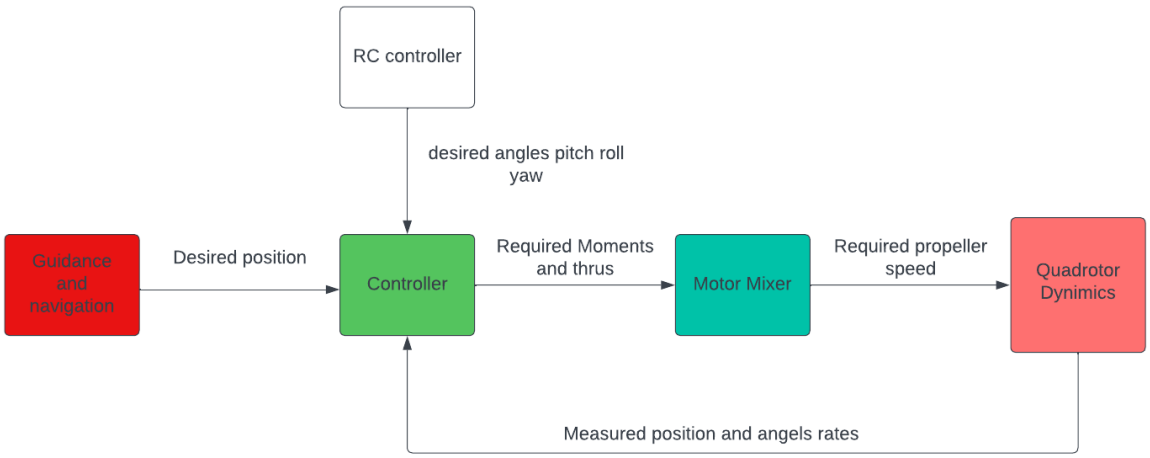


Figure 1.28: Block Diagram of Quadrotor Control Model

1.3.6 Guidance and Navigation

In this block, we output the desired positions (x, y, z, yaw) that we want our quadrotor to follow. These trajectories are directly sent to the controller block to generate the desired angles

(roll, pitch, yaw) that will be regulated. It is important to note that if we are using an RC, this block won't send anything because the desired angles (roll, pitch, yaw) are then directly sent through the RC.

1.3.7 Controller Blocks

The control of the quadrotor's position and attitude is accomplished by the design of the feedback controller and the method was well documented in [51] and [52].As we previously mentioned, we will use the cascade form and PID for the controller design. In general, we use two controllers: the first one is the position and velocity controller, which operates in cascade mode, and the second one is the attitude controller. The figure 1.29 illustrates our design strategy.

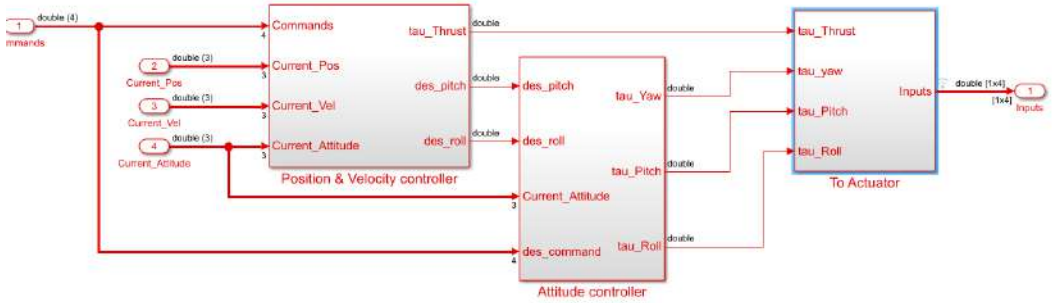


Figure 1.29: Controller Block

1.3.7.1 Position & Velocity Controller Block

As you can see in 1.30 our Simulink code, we aim to reduce the position error using the pitch, yaw, and roll angles. To adjust the attitude of our drone based on the position, we use the velocity. This means that we first reduce the position error by adjusting the velocity along each axis. Then, we reduce the error in velocity using the attitude angles.

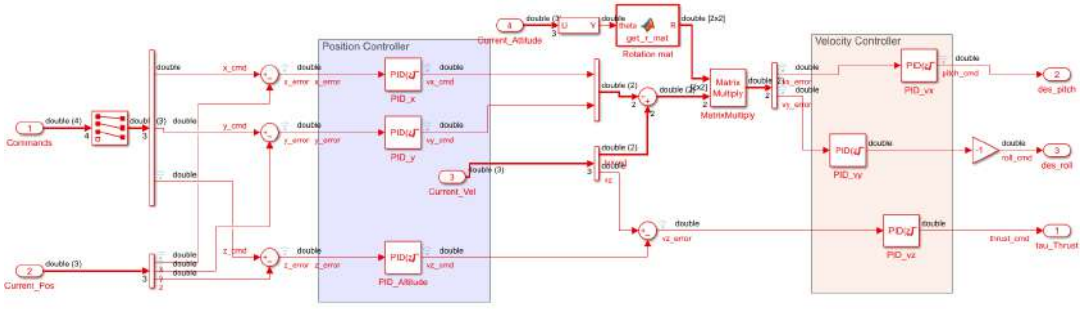


Figure 1.30: Simunlik blocks of Position & Velocity Controller Block

1.3.7.2 Tuning the PID Controllers

As previously mentioned, this is a cascade control structure. This means we need to start by tuning the inner loop first, followed by the outer loop. In our case, we will begin by tuning the velocity controller using the Ziegler-Nichols method, followed by the position controller.

- **Velocity and Position Controller**

  - **Velocity Controller**
    Since we are using a cascade configuration, the position controller must be in open loop while tuning the velocity controller. This allows us to regulate the velocity controller effectively, as shown in Figure 1.31.

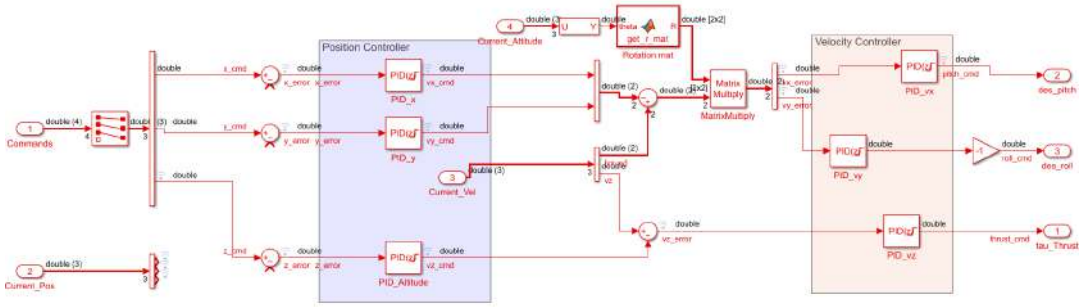


Figure 1.31: Open loop position controller

  - **Position Controller**
    After tuning the velocity controller, we proceed to tune the position controller. During this step, the velocity controller operates with the regulated gains in a closed loop configuration.

- **Attitude Controller**
  In the Attitude controller, we regulate the yaw, pitch, and roll angles using the corresponding torques, as shown in the figure 1.32

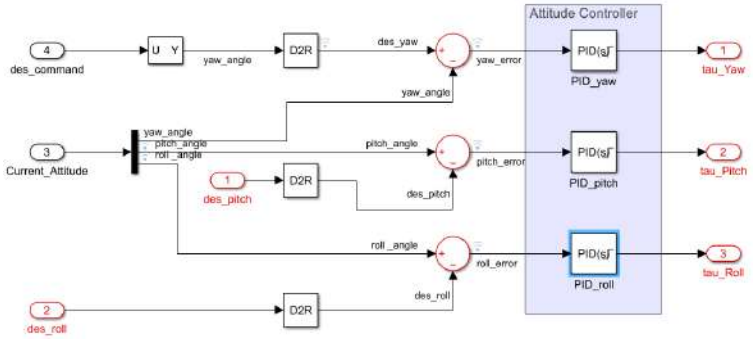


Figure 1.32: Attitude Controller

All the results and simulations are shown in the appendix 3.9. The following table 1.3 shows the gains that we found using the Ziegler-Nichols method.

Table 1.3: Summary of PID Controller Gains using Ziegler-Nichols Method

Command	K_{cr}	K
Roll	55	25
Pitch	60	27
Thrust	8	4
V_x	6.7	3.35
V_y	6.5858	3.2929
V_z	4.5	2.25
Torque Yaw	0.01	0.005
Torque Pitch	0.054	0.027
Torque Roll	0.06	0.03

1.3.8 Conclusion

In conclusion, this chapter covered the characteristics of the Pixhawk 2.4.8 flight controller, including its operating system and flight stack architecture. We then explored the quadrotor mathematical model, identifying all necessary parameters to write the state-space representation. Following this, we designed three types of controllers using different methods: the LQR method, pole placement, and finally, the PID method. These steps ensure a stable and optimal flight performance.

Chapter 2

Visual SLAM overview and testing

2.1 Introduction

Simultaneous localization and mapping (SLAM) is a groundbreaking technology in the field of robotics. For nearly four decades, it was a popular problem between researchers, a popularity that could be explained by how incredibly useful and versatile this technology is. Simultaneous localization and mapping ,as its name suggests, has two main functions that work in harmony and synchronicity. It addresses the dual challenge of constructing a map of an unknown environment all while keeping track of the robot's location. This complex task is solved by an ecosystem based on a set of sensors that keep track of the external environment and form the hardware half of the equation and a set of algorithms that form the software half.

SLAM gives the robot the ability to represent and reason about the relationship between itself and other objects in its vicinity hence reducing the locational uncertainty is a necessary step in order to achieve more complex tasks like navigation and obstacle avoidance amongst various others.This is how SLAM found its way to numerous application like autonomous navigation with self driving cars or military exploration with drones or even augmented and virtual reality (AR and VR) applications. But the bread and butter of SLAM is used for indoor applications where there's no access to a global positioning system (GPS).

In this chapter we're going to address the problem of SLAM for indoor environments, as well as the most important works that helped develop the SLAM ecosystem to what it is today. We'll also dive deeper in the classification of those algorithms,based on the hardware and software used, and of course go in depth in our proposed method.

2.2 Literature review

In the earliest works, the SLAM problem was dubbed a "State Estimation" problem which was instigated by Smith and Cheeseman's work in 1986 [53] that dealt with the estimation of spatial uncertainty. In 1991, building upon the foundation laid in [53], Leonard et al. [54] were the first to implement a probabilistic approach to solving this estimation problem. They utilized an Extended Kalman Filter (EKF) to localize a mobile robot by tracking the geometric shape of landmarks. In 1996, Durrant-Whyte and Bailey [55] took a significant step by combining the environment states with the vehicle states into one large vector to estimate them simultaneously, thereby formulating the first SLAM problem. This integration led to the development of the first recorded solution to the problem in 2001, known as EKF-SLAM [56].

Following these pioneering efforts, various improvements and alternative methods emerged to address the limitations of EKF-SLAM. One such advancement was the introduction of FastSLAM by Montemerlo et al. in 2002 [57], which leveraged a particle filter to manage the high computational costs associated with EKF-SLAM, allowing for more efficient handling of larger maps. FastSLAM 2.0, an improved version of this algorithm, was later introduced in 2003 [**?**], further enhancing the accuracy and efficiency of SLAM systems.

In parallel, research on Graph-Based SLAM started gaining traction. Thrun and Montemerlo's work in 2006 [58] highlighted the advantages of representing the SLAM problem as a graph of poses connected by constraints derived from observations and odometry. This representation paved the way for optimization techniques such as Pose Graph Optimization (PGO), which refined both the robot's trajectory and the map by minimizing a global error function.

The development of robust feature extraction and matching techniques also significantly contributed to the evolution of Visual SLAM. For instance, the introduction of ORB-SLAM by Mur-Artal et al. in 2015 [9] marked a significant milestone. ORB-SLAM utilized Oriented FAST and Rotated BRIEF (ORB) features, known for their efficiency and robustness in real-time applications, and combined them with sophisticated optimization techniques like Bundle Adjustment to achieve high accuracy in pose estimation and mapping.

As research progressed, newer SLAM algorithms started incorporating advanced machine learning techniques. For example, Deep Learning-based SLAM approaches began to emerge, leveraging Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) to enhance feature extraction, data association, and overall system robustness in challenging environments.

The journey from the initial state estimation problems to contemporary SLAM solutions underscores a continuous evolution driven by both theoretical advancements and practical implementations, each building upon the successes and addressing the limitations of its predecessors. This rich history of development highlights the collaborative and iterative nature of progress in the field of SLAM, setting the stage for future innovations and applications.

2.3 Problem statement

The goal of this thesis is to deploy an unmanned aerial vehicles (UAVs) for social security tasks such as exploration and surveillance, so there's a critical need to design systems that are both cost-effective and compact. This is a direct consequence of the fact that we want to reproduce these systems for widespread use in various scenarios. Given these constraints, the use of Lidar sensors, which are typically expensive and bulky, is not feasible which exludes Lidar-based SLAM. This leaves us with techniques that utilize cameras as a promising alternative to lidars due to their affordability and lightweight nature.

Moreover, these applications require real-world and real-time implementation which demands that the algorithms used are highly optimized to ensure timely processing of data. Out of all state of the art visual slam algorithms, ORB-SLAM meets this criterion through its efficient use of computational resources, making it the most suitable for deployment on UAVs with limited processing power. In the following sections we'll be providing the necessary theoretical background needed in order to establish an ORB-SLAM framework.

2.4 The classical visual slam framework

2.4.1 Mathematical formulation of the SLAM problem

We know that SLAM the problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within that environment. The SLAM problem is typically formulated in a probabilistic framework and consists of two main components: the motion model and the measurement model.

2.4.1.1 State Vector

The state vector $\mathbf{x}_t$ at time t includes the robot's pose $\mathbf{p}_t$ and the map $\mathbf{m}$:

$$\mathbf{x}_t = [\mathbf{p}_t, \mathbf{m}]$$

where:

- $\mathbf{p}_t$ is the pose of the robot at time t, typically represented as position and orientation.
- $\mathbf{m}$ represents the map, which can be a collection of landmarks or a continuous representation of the environment.

2.4.1.2 Motion Model

The motion model describes how the robot's pose evolves over time based on the control inputs. It is represented as:

$$\mathbf{p}_t = f(\mathbf{p}_{t-1}, \mathbf{u}_t) + \mathbf{w}_t \tag{2.1}$$

where:

- f is the motion function that describes how the previous pose $\mathbf{p}_{t-1}$ and the control input $\mathbf{u}_t$ produce the new pose $\mathbf{p}_t$.
- $\mathbf{w}_t$ represents the process noise.

2.4.1.3 Measurement Model

The measurement model relates the observed measurements to the robot's pose and the map. It is represented as:

$$\mathbf{z}_t = h(\mathbf{p}_t, \mathbf{m}) + \mathbf{v}_t \tag{2.2}$$

where:

- h is the measurement function that maps the current pose $\mathbf{p}_t$ and the map $\mathbf{m}$ to the expected measurement $\mathbf{z}_t$.

- $\mathbf{v}_t$ represents the measurement noise.

So the whole problem is how to estimate the state vector based on these two equations.

A typical VSLAM eco-system can be dissected into five very different but equally important steps as shown in 2.1

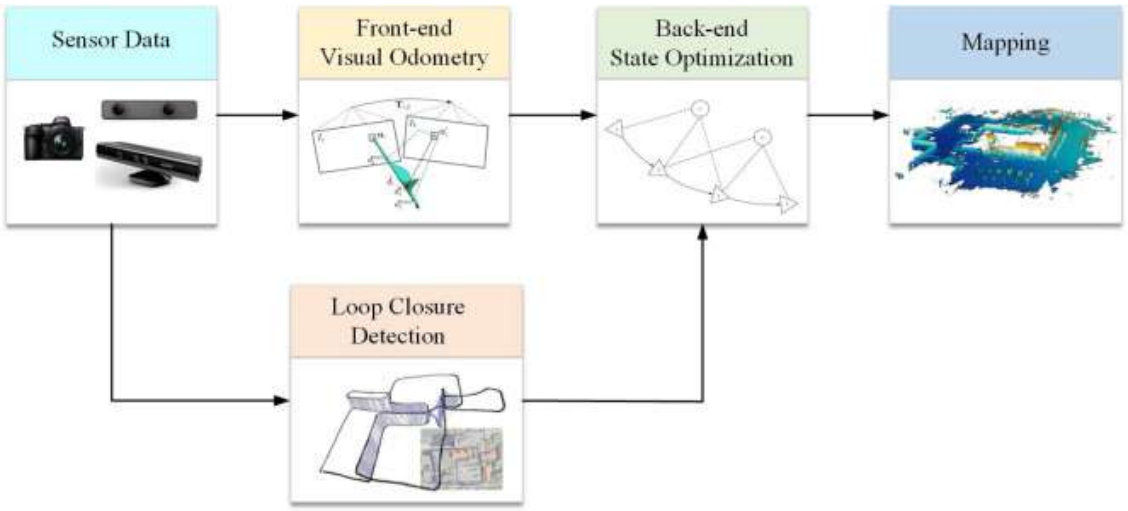


Figure 2.1: The Visual SLAM framework [4]

2.4.2 Data Acquisition

As visual SLAM refers to the process of using visual features to localize and construct a map of the environment, then data acquisition mainly means the process of getting images using different camera sensors and prepossessing them before passing them through the SLAM pipeline. So before advancing further in the SLAM core we need to explain how the data acquisition varies by changing the camera sensor.

2.4.2.1 Monocular camera

Monocular SLAM refers to the SLAM achieved through a Monocular cameras. These are cameras that use one lens for the acquisition, they are very simple and relatively very cheap which makes them incredibly attractive to researchers which is why MonoSLAM [59] or monocular slam is one of the most researched SLAM algorithms out there. But how does the transformation from the 3D world in meters to a 2D image plane in pixels work ?

Figure 2.2: The transformation pipeline

This is described by several geometrical models [60], the simplest one is the pinhole camera model [61] used for monocular cameras. We'll explain this model based on figure 2.3.

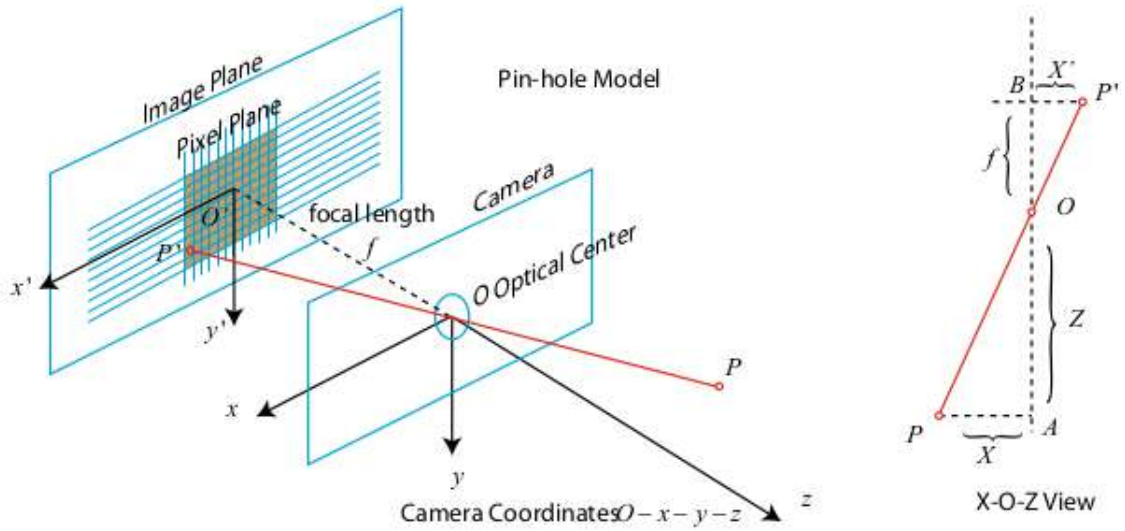


Figure 2.3: The pinhole camera model[5]

Let O - x - y - z be our used camera coordinate system, with the z axis being in the front of the camera and the camera's optical center O representing the camera aperture . The point P is in the 3D camera frame, and after its projection to the 2D image sensor frame O' - x' - y' - z' we get the point P'. The coordinates of the point P in the camera frame $P = \begin{bmatrix} X & Y & Z \end{bmatrix}^\top$,and the coordinates of P' in the imaging plane are $P' = \begin{bmatrix} X' & Y' & Z' \end{bmatrix}^\top$. The final thing we need to define is the physical distance between the camera lens and the imaging plane which is denoted by f and is know as the focal length. Now using the law of similarity of triangles, we get the following equation :

$$\frac{Z}{f} = \frac{X}{X'} = \frac{Y}{Y'} \tag{2.3}$$

which means by putting X' and Y' to the left side we get :

$$X' = f\frac{X}{Z},\ Y' = f\frac{Y}{Z} \tag{2.4}$$

The formula 2.4 indicates the spatial relationship between the point P and its image . But as it is shown in figure 2.2 the end goal is to end up in the pixel plane through the sampling and the discretization of the image, and with that comes the new pixel plane denoted as o - u - v. o is the upper left corner of the image meaning the origin is translated by $\begin{bmatrix} c_x & c_y \end{bmatrix}^\top$, and of course there's not only a translation but an apparent "zoom" due to the scale change of the image. So we define the two quantities α and β that represents respectively the scale on the u and the v axis. So the relationship between the coordinates of P' and the pixel coordinate system $\begin{bmatrix} u & v \end{bmatrix}^\top$ is :

$$u = \alpha X' + c_x \tag{2.5a}$$
$$v = \beta Y' + c_y \tag{2.5b}$$

If we inject this into 2.4 we get the following relationship :

$$u = f_x\frac{X}{Z} + c_x \tag{2.6a}$$
$$v = f_y\frac{Y}{Z} + c_y \tag{2.6b}$$

α and β are in pixels/meter, which means that f_x, f_y and c_x, c_y are in pixels. And if we write this in matrix form, we get the following:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z} \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{2.7}$$

The matrix showcased in 2.7 is called the intrinsic parameter matrix. These parameters are fixed during the manufacturing and are generally giving by the manufacturing company.In the case of their unavailability, we perform calibration using a set of famous algorithms such as the Zhang Zhengyou algorithm [62].

Now, we successfully got the latter part of the transformation pipeline 2.2 that allows the transformation from the camera coordinates to the pixel coordinates using the intrinsic parameters matrix. Now for the former part that allows the transformation from the world coordinates to the camera, we use what we call the extrinsic parameter matrix which is a 4×4 homogeneous transformation matrix that takes this form :

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} R_{11} & R_{12} & R_{13} & t_1 \\ R_{21} & R_{22} & R_{23} & t_2 \\ R_{31} & R_{32} & R_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{2.8}$$

where $\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix}$ is the rotation matrix that represents the orientation of the camera with respects to the world frame and $\begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$ that represents the translation vector from the origin of the world frame to the origin of the camera frame. By getting both the intrinsic and the extrinsic parameter matrices we get proceed with our data acquisition as shown in 2.9.

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z} \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} & t_1 \\ R_{21} & R_{22} & R_{23} & t_2 \\ R_{31} & R_{32} & R_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{2.9}$$

The transformation from the 3D world frame to the 2D image is a loss of a dimension, that is translated by the loss of the very important quantity in SLAM which is Depth. So by using a single image from a monocular camera getting depth is impossible, and a critical consequence of loosing depth is loosing scale because the size of a particular object in an image is relative to the other object in the same image. This mean that getting a 3D structure using monocular camera can't be done by using a single frame, we need multiple frames. Another solution to this problem is adding either another camera to create a different type of model that we call a stereo system [63] or adding a Time Of Flight (TOF) system to create depth cameras.

2.4.2.2 Stereo and RGB-D cameras

As the work presented in this thesis will deal primarily with a monocular camera system, we won't go into to much detail in the explanation of both types of cameras but we will cite some works that go into much more depth.
Stereo and RGB-D cameras have the same purpose of eliminating the scale ambiguity that we get from using a monocular camera by creating a 3D structure of the environment from a single snapshot, but they have very different ways of achieving that purpose.

Figure 2.4: Getting depth from disparity [5]

Stereo cameras consist of two synchronized monocular cameras separated by a distance known as the baseline, the larger the baseline the further the camera can estimate depth. The way that's done is very similar to how the human eyes work, by using the principle of disparity.

- **Step 1:** Take two snapshots from both the cameras in the stereo system that are separated by the baseline.

- **Step 2:** Perform "stereo matching", which is the process of matching features from the image captured by the left camera to the one captured from the right and vice-versa.

- **Step 3:** After matching the features, calculate the disparity between each matching features from the same image pair, which means how much a features is displeased in one camera frame with respect to the other.

- **Step 4:** Turn that disparity into a pixel intensity, which means the features who are more displaced will be more intense or more luminescent while the other features will be less intense.

After the fourth step you should get an image as shown in figure 2.4,where the luminescent object are the the closest and the darker objects are the furthest. This can be verified by a simple eye test; if you put two hands in front of each other and then very quickly close one eye at a time, you can tell that the hand that's closest to you seems like it's moving more from one eye to the other. By using the same logic, we can estimate depth as shown in figure **??**. Again this is just an intuitive understanding of the stereo principle for depth estimation, to get a deeper understanding of the mathematical model you can view [63].

RGB-D cameras on the other hand provide a more active approach to calculating depth. By emitting an infrared light beam on the target object and then calculating the distance based on the object and the camera as well as the structure of the returned light beam. Which means that from a single picture u can get a full on point cloud depth estimation.

2.4.3 Visual Odometry

Now after getting our input data from the camera model, we move on to the frontend of our visual SLAM pipeline also known as Visual Odometry (VO), which was first introduced in 1986 by the revolutionary work of Hans P. Moravec in [64]. VO refers to the process of getting a estimates of the camera movement based on the consecutive information extracted from images, so this task can be done with one chain of movement at a time, which is why we can say that VO has a short term memory. This can be done by choosing pixels in our image as landmarks (features), then tracking the 3D position of those pixels based on adjacent frames. But how do we choose these landmarks and how do we describe them ?

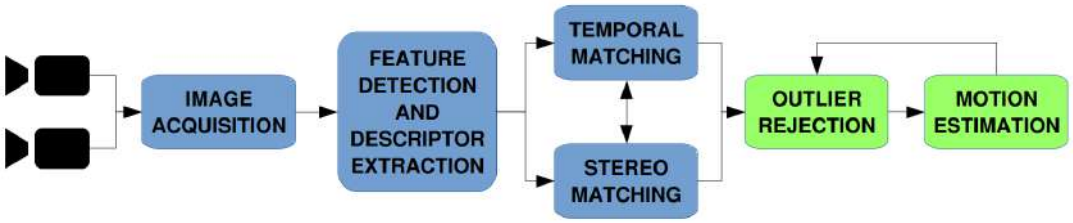


Figure 2.5: Visual Odometry pipeline [6]

The work done in [65] was the first that came up with the pipeline presented in figure 2.5, in which it describes the features using key points and descriptors.

2.4.3.1 Feature detection

This part of the pipeline deals with the selection landmarks for the SLAM problem using various criterias. We will be presenting only a few solutions while keeping our chosen method for later on in this chapter.

Features are selected using a set of rules that can be resumed in these four points :

1. **Repeatability:** In order for us to be able to track this specific feature, we need to detect it in multiple frames. Meaning we need to ensure that this feature can be found in different images.

2. **Distinctiveness:** On the other side of repeatability, we need the feature to be distinctive. So that the feature will be recognized in the following frame easily and not be confused with a bunch of others.

3. **Efficiency:** The number of features need to be relatively small so that the computational load can be kept to a minimum. A general good criteria for this is that the number of feature should be far smaller than the number of pixels.

4. **Locality:** We need to be able to trace that image directly to the a specific area of the image, hence the feature need to only be related to that specific location.

These features can be divided into either corners or Binary Large Objects (Blobs).

2.4.3.2 Corner detection

A corner is a highly distinctive feature, as it is invariant to translation, rotation and even pixel illumination. It can be considered as simply an intersection of two edges, so if we're able to detect an edge successfully we'll be able to detect their intersection and hence detect the corner. An edge, in the world of image processing is considered as a shift in pixel intensity, so if we detect that big change in intensity we'll be able to say with a certain degree of confidence that we have been able to detect an edge. Using this logic, we're going to look at two of the most popular corner detectors and review their suitability for our application.

1. **The Harris detector:**

   Let $I(x, y)$ be the intensity of an image at the specific pixel (x, y), and let w be the window (the local part of the image) that contains the pixel (x, y). If we want to quantify the rate of change of intensity of a pixel in the x and y direction, then we can use the image gradients :

   $$I_x = \frac{\partial I}{\partial x}, I_y = \frac{\partial I}{\partial y} \tag{2.10}$$

   In order to calculate the change of intensity in the same window w, we need to offset the pixel (x, y) by a quantity $(\delta u, \delta v)$ that keeps it inside the window to get the intensity of the offseted pixel $(x + \delta u, y + \delta v)$. Now we can define our change function $f(u, v)$:

   $$f(u, v) = \sum_{(x,y)\in w} \left[I(x + \delta u, y + \delta v) - I(x, y)\right]^2 \tag{2.11}$$

   where $f(u, v)$ is the sum of squared differences (SSD) that measures the shift in intensity in both direction x and y.

   If we substitute the Taylor equation :

   $$I(x + \delta u, y + \delta v) \approx I(x, y) + \begin{bmatrix} I_x & I_y \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{2.12}$$

   we get the following :

   $$f(u, v) = \sum_{(x,y)\in w} \left(\begin{bmatrix} I_x & I_y \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \right)^2$$

   $$f(u, v) = \begin{bmatrix} \delta u & \delta v \end{bmatrix} \left(\sum_{(x,y)\in w} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{2.13}$$

   And if we transfer the sum to inside the matrix, we get the following :

   $$f(u, v) = \begin{bmatrix} \delta u & \delta v \end{bmatrix} \begin{bmatrix} \sum_{(x,y)\in w} I_x^2 & \sum_{(x,y)\in w} I_x I_y \\ \sum_{(x,y)\in w} I_x I_y & \sum_{(x,y)\in w} I_y^2 \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{2.14}$$

   The matrix in 2.14 is called the structure matrix which accumulates all the information about the image gradient in the x and y direction (and the xy direction as well). We're basically encoding information on the change of intensity.

   $$H = \begin{bmatrix} \sum_{(x,y)\in w} I_x^2 & \sum_{(x,y)\in w} I_x I_y \\ \sum_{(x,y)\in w} I_x I_y & \sum_{(x,y)\in w} I_y^2 \end{bmatrix} \tag{2.15}$$

Now for us to able to tell in which direction the change of intensity is more dominant around a specific point (x, y), we can look at the values inside the structure matrix.

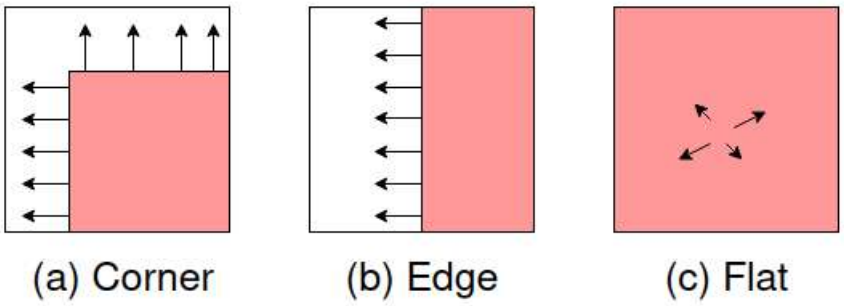


Figure 2.6: Corner detection with the image gradient

If we find ourselves in this case :

$$H = \begin{bmatrix} \gg 1 & \approx 0 \\ \approx 0 & \gg 1 \end{bmatrix} \tag{2.16}$$

Then we successfully detected a corner as shown in figure 2.7(a), as the gradients are found equally dominant in the x and y directions. However if we found ourselves in the following cases :

$$H = \begin{bmatrix} \approx 0 & \approx 0 \\ \approx 0 & \gg 1 \end{bmatrix} \tag{2.17}$$

and

$$H = \begin{bmatrix} \approx 0 & \approx 0 \\ \approx 0 & \approx 0 \end{bmatrix} \tag{2.18}$$

Then respectively detected an edge and a flat region as shown in figure 2.7(b) and 2.7(c).

An efficient way of getting the same result is by checking the eigenvalues of the structure matrix H, we can say that a corner has successfully been detected simply by finding two large eigenvalues.

$$\det \begin{bmatrix} h_{11} - \lambda & h_{12} \\ h_{21} & h_{22} - \lambda \end{bmatrix} = 0$$

And using a specific score function to test this criteria, we'll get very accurate results. For example the Harris score function for corner strength detection that was first presented in 1988 by Harris in [66]:

$$R = \det(H) - k \cdot (\text{trace}(M))^2$$

where:

- $\det(H)$ is the determinant of our structure matrix H,
- $\text{trace}(H)$ is the trace of the same matrix H,
- k is a sensitivity parameter (usually $0.04 \leq k \leq 0.06$).

This will result in the following set of rules :

a. If $R \approx 0$ then $\lambda_1 \approx \lambda_2 \approx 0$, this is a flat region.

b. If $R < 0$ then $\lambda_1 \gg \lambda_2$ or $\lambda_2 \gg \lambda_1$, this is an edge.

c. If $R \gg 0$ then $\lambda_1 \approx \lambda_2 \gg 0$, this is a corner.

By putting everything together, we get the following algorithm for the Harris corner detection:

Algorithm 1 Harris Corner Detection

Input: Grayscale image I, sensitivity parameter k
Output: Harris corner response map R

Step 1: Compute image gradients
$I_x \leftarrow \text{convolve}(I, \text{Sobel kernel}_x)$
$I_y \leftarrow \text{convolve}(I, \text{Sobel kernel}_y)$
▷ The Sobel kernel is an operator used for computing the image gradients of the image, basically a derivation of the image's pixel intensity.

Step 2: Compute products of gradients
$I_{x^2} \leftarrow I_x \odot I_x$
$I_{y^2} \leftarrow I_y \odot I_y$
$I_{xy} \leftarrow I_x \odot I_y$

Step 3: Apply Gaussian filter to the products of gradients
$S_{x^2} \leftarrow \text{convolve}(I_{x^2}, \text{Gaussian filter})$
$S_{y^2} \leftarrow \text{convolve}(I_{y^2}, \text{Gaussian filter})$
$S_{xy} \leftarrow \text{convolve}(I_{xy}, \text{Gaussian filter})$
▷ The Gaussian filter is applied to smooth out the image

Step 4: Compute the Harris response
for each pixel (x, y) in the image **do**
$H \leftarrow \begin{bmatrix} S_{x^2}(x,y) & S_{xy}(x,y) \\ S_{xy}(x,y) & S_{y^2}(x,y) \end{bmatrix}$
$\det(M) \leftarrow S_{x^2}(x,y) \cdot S_{y^2}(x,y) - (S_{xy}(x,y))^2$
$\text{trace}(M) \leftarrow S_{x^2}(x,y) + S_{y^2}(x,y)$
$R(x,y) \leftarrow \det(M) - k \cdot (\text{trace}(M))^2$
end for

Return: Harris corner response map R

ENP Example :

We applied the Harris corner detection algorithm on a frame taken of Ecole Nationale Polytechnique 2.7 which classifies as a relatively large image that contains very distinct features.

Commentary :

After applying the algorithm, we can see that the Harris detector provides accurate localization of corner points with a detected number of features of 84746, however a lot of features are condescend on each other. We noticed that when it comes to such a large image, despite of the decently accurate results, the algorithm took too long to compute

the points with an execution time of 0.124 seconds which eliminates all our hopes of using it in real-time processing.

Figure 2.7: Harris corner detection on Ecole Nationale Polytechnique

2. **The Features from Accelerated Segment Test (FAST) detector**

   The principle behind corner detection in the FAST algorithm is derived from the human perceptual sense of corners, where significant changes in gray levels occur in all directions. The algorithm defines a corner point as a pixel that exhibits substantial intensity differences with enough pixels in its neighborhood, so if a pixel p exhibits a significant difference in intensity compared to a predefined number of pixels in its surrounding circular neighborhood, it is classified as a corner.

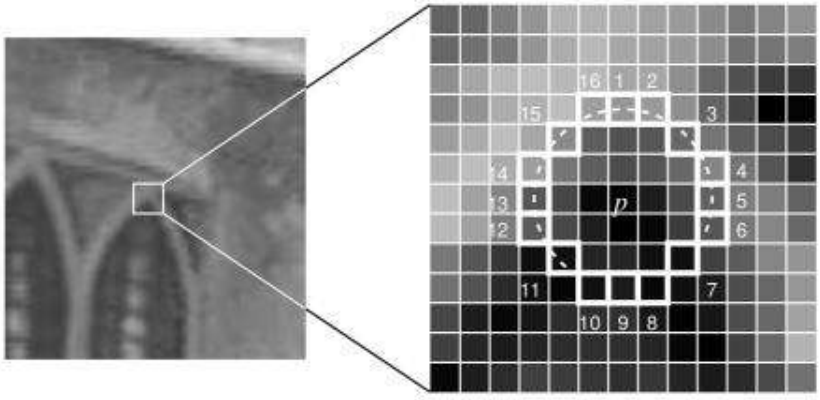


Figure 2.8: The 16 considered neighboring pixels

Here are the steps of the Fast algorithm :

a. **Pixel Selection**:

   For each candidate pixel p, consider a circle of 16 surrounding pixels. Their position form what is known as a Bresenham circle of radius 3 as shown in figure 2.8, which ensures that the selected pixels are equidistant from the candidate pixel.

b. **Intensity Threshold**:

Define a threshold t which is used to determine whether the intensity of the candidate pixel p significantly differs from the intensities of the surrounding pixels. So for each pixel P_i in the circle, we compare its intensity $I(P_i)$ with the intensity $I(p)$ of the candidate pixel:

$$\Delta I_i = |I(P_i) - I(p)|$$

If ΔI_i exceeds the threshold t, then the pixel P_i is considered significantly different in intensity from p.
We note that a common default value for the intensity threshold t is around 10 to 20 but it depends on various factors such as image contrast and noise. We will show the effect of the threshold in the ENP example.

c. **Initial Test**:

We then perform a quick test by comparing p to the pixels at positions 1, 5, 9, and 13 in the circle. These four pixels are chosen because they are spaced evenly around the candidate pixel, providing a quick and efficient initial check. If at least three of these pixels are either all brighter or all darker than p by at least t, proceed to the next step. Mathematically, this can be expressed as:

$$\text{Initial Test Pass} = \left(\sum_{i \in \{1,5,9,13\}} ((I(P_i) > I(p) + t) \vee (I(P_i) < I(p) - t)) \right) \geq 3$$

If the initial test passes, continue with the full test.

d. **Full Test**:

If the initial test is passed, compare p with all 16 pixels in the circle. Count how many pixels are at least t brighter or t darker than p. If there are at least 12 pixels that meet this criterion, classify p as a corner. This can be formalized as:

$$\text{Full Test Pass} = \left(\sum_{i=1}^{16} ((I(P_i) > I(p) + t) \vee (I(P_i) < I(p) - t)) \right) \geq 12$$

If the full test passes, p is identified as a corner point.

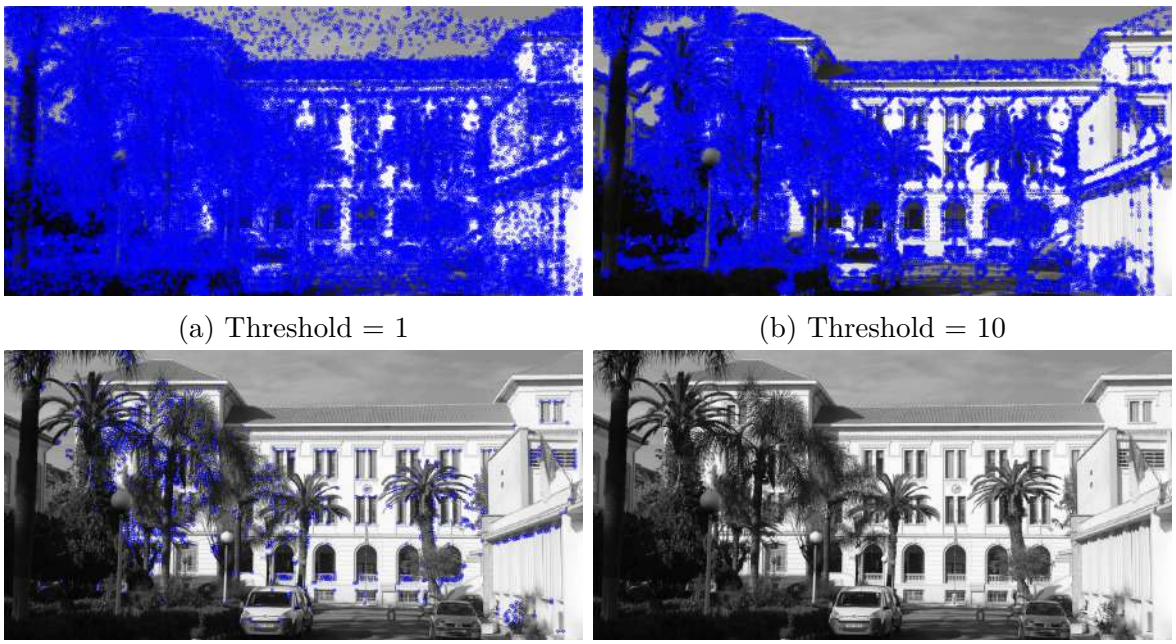
ENP Example :

We applied the Fast algorithm to the same image for comparison as shwon in figure 2.9.

Commentary:

Threshold	Number of Features	Execution Time (seconds)
1	40916	0.0107
10	24817	0.0059
100	912	0.0004
1000	0	0.0001

Table 2.1: Number of Features and Execution Time for Different Threshold Values



(a) Threshold = 1 (b) Threshold = 10

(c) Threshold = 100 (d) Threshold = 1000

Figure 2.9: FAST feature detection with different threshold values on Ecole Nationale Polytechnique

The table above shows the execution times for the FAST algorithm with various threshold values. As the threshold increases, the execution time decreases significantly as well as the number of features. This behavior can be explained by the nature of the FAST algorithm:

- The initial test in the FAST algorithm involves comparing the intensity of the candidate pixel to the intensities of the pixels at positions 1, 5, 9, and 13 in the circle of 16 surrounding pixels, this means that with higher threshold values, fewer pixels will meet the criteria in the initial test, thus reducing the number of candidate corners that need to be processed further in the full test. This means that fewer features are chosen, resulting in faster execution times.

The FAST algorithm is incredibly quick and optimized for real-time application as shown by the results. So we confidently say that we'll be choosing it for our method.

2.4.3.3 Binary Large Object detection

Binary Large Objects or blobs in the world of image processing are defined as a region in an image or a set of connected pixels that share properties such as intensity or color that separates them from surrounding regions. The main difference between blobs and corners, is that corners represents the outer structure of the image while blobs form the internal features of it. This is further highlighted in figure 2.10

The process of blob detection was first shown by Marr in 1980 [67], where we were introduced to the Difference of Gaussian algorithm that became the golden standard for blob detection. The algorithm is done in 3 steps.

- **Step 1 :** This step is responsible for finding a stable set of features that are invariant to the scale change of the image as well rotation and translation, and it has already been proven in [68] that the only scale-space kernel that allows the examination of an image in different scales is the Gaussian function.

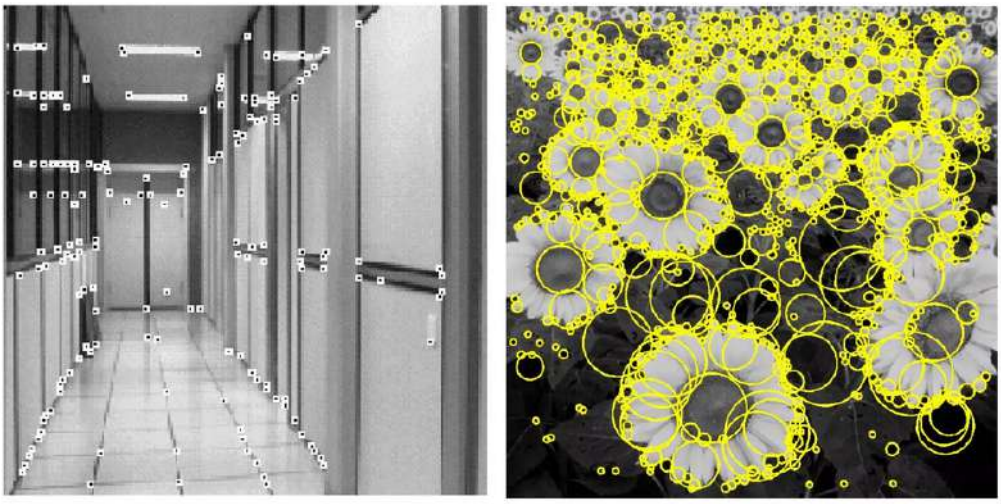


Figure 2.10: Corners vs Blobs

Let's define $L(x, y, \sigma)$ as the image scale space function, and it's a result of convoluting the input image multiple times at multiple scales with the Gaussian function $G(x, y, \sigma)$:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{2.19}$$

where

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.20}$$

We can explain this step in more simpler terms as just applying increasing degrees of Gaussian blur, blurring the image more and more, exactly as shown in figure 2.11

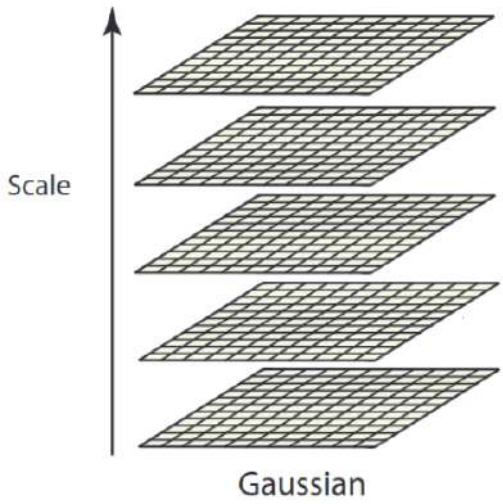


Figure 2.11: Gaussian filter scale

- **Step 2:** In this step, we'll try and find those stable keypoints in the scale space, by trying to find extremas which are points that locally stand out from the other regions. This is done through an operation called Difference Of Gaussians (DoG) performed over differently sampled images. As shown in figure 2.12.

  This step is done to increase visibility of corners, edges , blobs and every other detail present in the image.

We can also note that the Difference of Gaussians acts as a band-pass filter where we first filter out the high-frequencies by performing the normal blur, then by subtracting the blurred images 2.21 we only keep the frequencies that lie in between those two images.

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \tag{2.21}$$

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{2.22}$$

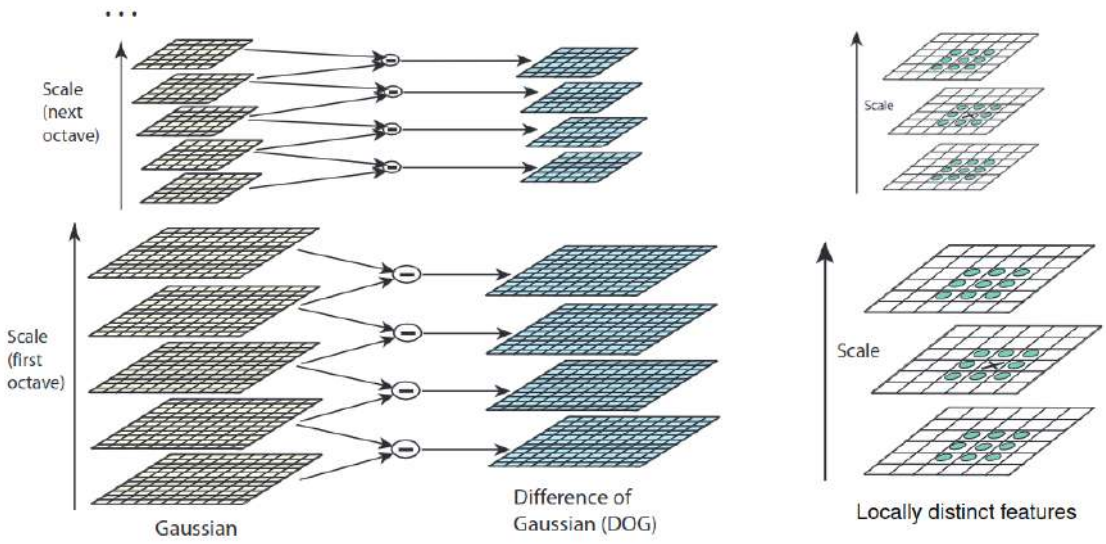


Figure 2.12: Difference of Gaussians

- **Step 3:** This step is called keypoint localization, as of now the DoG algorithm has produced various keypoints. Some of them are less useful than other, for example : not enough contrast or too many edges detected. So in order to keep only the most usefull points we go through a two part process : localization of those keypoints, then discard of the less expressive.

  1. **Localization:** We use a Taylor series expansion for the scale-space function $D(x, y, \sigma)$ around an initial keypoint :

     $$D(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x} \tag{2.23}$$

     Where $x = (x, y, \sigma)$ is the keypoint position in the scale space, and the gradient $\frac{\partial D}{\partial \mathbf{x}}$ and the Hessian matrix $\frac{\partial^2 D}{\partial \mathbf{x}^2}$ are calculated from the DoG images.

     And then to find the accurate position of the keypoint, we solve this equation for $\hat{x}$ which is the location of the extremum.

     $$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1}\frac{\partial D}{\partial \mathbf{x}} \tag{2.24}$$

     This gives the offset from the initial keypoint position to the refined position. If the offset $\hat{\mathbf{x}}$ is large, it indicates that the keypoint is not stable, and such keypoints are discarded.

  2. **Thresholding:** We discard features will low contrast through the following threshold test :

     $$D(\hat{x}) > threshold \tag{2.25}$$

     In the Lowe paper that presented this method [69], a value of 0.03 was used as a threshold. Which is the value we used in the example shown in figure 2.13

ENP Example :

We applied the DoG algorithm to the same image for comparison.

Figure 2.13: DoG blob detection on Ecole Nationale Polytechnique

Commentary:

The DoG algorithm due to it's scale invariance produces more accurate results, meaning it chooses the most robust features (Number of Blobs: 113),which makes the algorithm more suitable for complex and dynamic environments. However what it gains in robustness, it looses in time (Execution Time: 0.0985 seconds). Due to all the extra processing to achieve those results, DoG turned out to be significantly slower than the Harris algorithm which again make it non suitable for real-time applications.

2.4.3.4 Descriptor extraction

Now in order for us to be able to match keypoints or features to each other in different frames, we need to be able to describe them in a way that will allow us to both be able to distinguish them from each other while also keeping the operation of comparison as lightweight as possible when it comes to computation.

Lowe in his groundbreaking article published in 2004[69] introduced the notion of feature descriptors which are vectors that summarizes the local structure around the specific keypoint.

Based on what descriptor we use, meaning what that vector contains, we can formulate different types of features descriptor algorithms. In this part we'll present and test two different golden standards for descriptor extraction.

1. **Scale-Invariant Feature Transform (SIFT)**

   The SIFT descriptor [69] is one of the most robust and widely studied descriptors in the field, where we transform the image content into features that are invariant to image

translation, rotation and scale, while staying partially invariant to pixel illumination (if the image gradient remains unchanged).

The SIFT features are extracted from the difference of gaussians algorithm that we have shown previously,and they are given by a vector computed at a local extreme point in the scale space as explained previously as well. Here's the vector :

$$\Big(p, s, r, f\Big) \tag{2.26}$$

Where

- p is the pixel location of the feature on the image (x, y)
- s is the scale of the extrema in the scale-space from the DoG.
- r is the orientation of the point which we will explain briefly how to get.
- f which is a 128-dim descriptor that is generated from the local image gradients.

Now the question is, how de we calculate the orientation. This is done throughout the following process.

a. **Gradient Calculation**

After detetcting each keypoint, our SIFT algorithm calculates the image gradients in a local neighborhood around each of those keypoint. Those gradients are computed at the scale of the each selected feature, which we obtained during the scale-space extrema detection process.

Gradient Magnitude $m(x, y)$ and Orientation $\theta(x, y)$:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{2.27}$$

$$\theta(x,y) = \tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right) \tag{2.28}$$

where L represents the Gaussian-smoothed image at the appropriate scale.

b. **Orientation Histogram**

After computing the gradients, we create a histogram of their orientations taken within a circular window around each keypoint. This histogram typically covers 360 degrees of orientation.

Note that each sample's contribution to the histogram is weighted by its gradient magnitude and it's placement within the centered window. This reduces the influence of gradients far from the keypoint.

$$\text{weight} = m(x,y) \cdot \exp\left(-\frac{(x-x_k)^2 + (y-y_k)^2}{2\sigma^2}\right) \tag{2.29}$$

c. **Peak Selection**

Now form the peak of our histogram, we can select the dominant orientation. This orientation is assigned to that keypoint. Additionally, peaks within 80% of the highest peak are also used to create new keypoints with the same location and scale

but different orientations. This ensures robustness to changes in viewpoint and local appearance. This whole process is shown in figure 2.14.

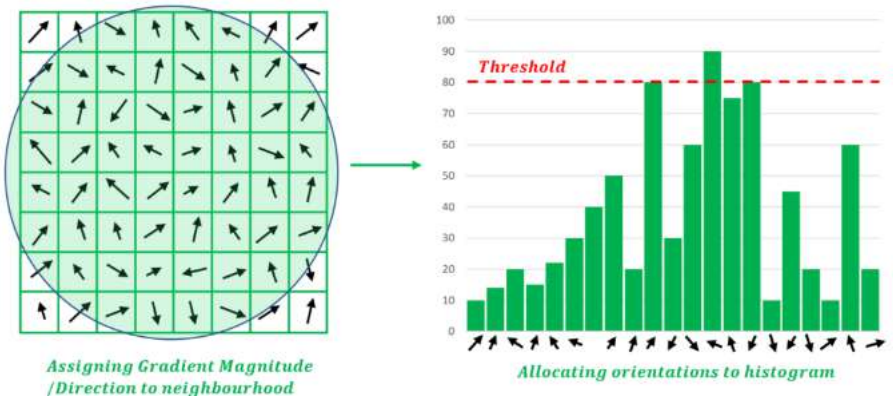


Figure 2.14: Allocating orientation to keypoints

Dominant Orientation:

$$\theta_k = \arg\max_{\theta} \text{hist}(\theta)$$

Now for the descriptor, we select a 16×16 window around the keypoint, where we compute the gradient of each pixel in the neighborhood, after that we rotate the coordinates of the gradients to align with the keypoint's orientation (this is what ensures invariance to image rotation). After that we create a histogram of the 8 gradient orientations for each sub-region in that window. Meaning 16 regions times 8 directions, that will give us a 128-dim vector that concatenates each histogram of those regions. This process is shown in figure 2.15.

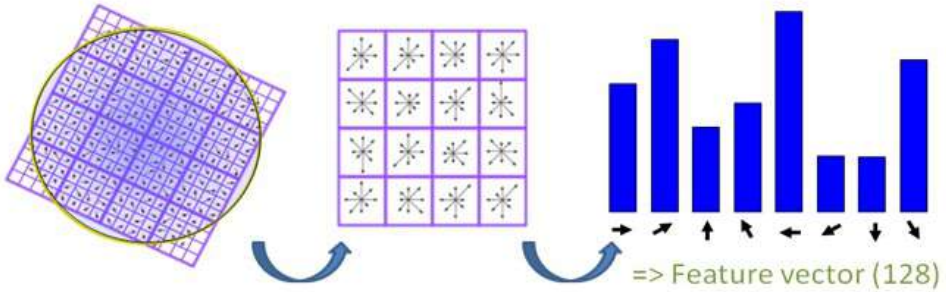


Figure 2.15: Creating the SIFT descriptor

Now in order to match the features together, we just compare the descriptors for each keypoints. Note that there are different methods to optimize this matching process and making it much more robust by dealing with outliers but we won't be discussing them in this thesis.

ENP Example :

Now we apply the SIFT algorithm with DoG blob detection on our ENP example image and a on a 45 degrees rotated variant of it.

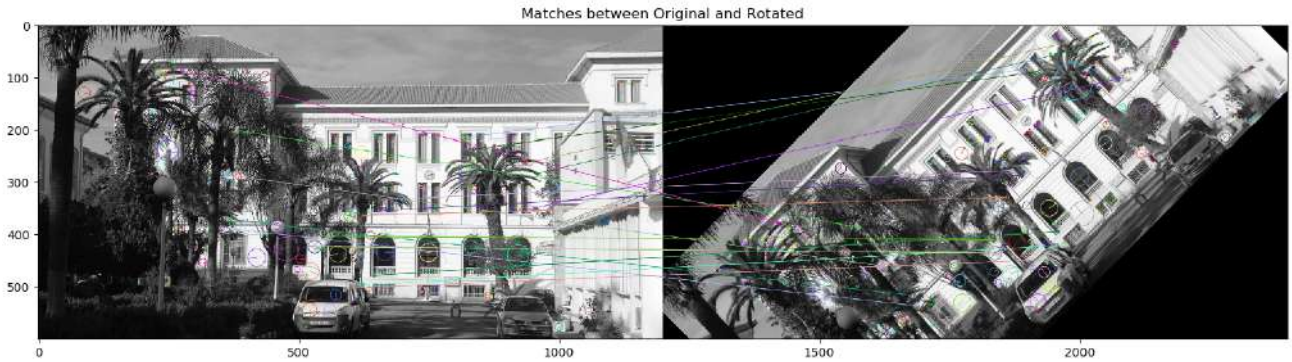


Figure 2.16: SIFT feature extraction on Ecole Nationale Polytechnique (rotated by 45 degrees

Commentary:

The SIFT algorithm's took 0.4522 seconds to execute which reflects its computational intensity due to the multi-step process, including scale-space construction and descriptor generation. Because of this longer processing time, SIFT achieves a high accuracy of 80.81% for matches, attributed to its robust and distinctive 128-dimensional descriptors and its invariance to scale and rotation. But its long computational time eliminates it from our real-time processes.

2. **Binary descriptors**

Although descriptors such as SIFT or SURF (Speeded Up Robust Features)[70] are incredibly robust and accurate, they trade off speed in order to get those desired results. Hence they're pretty much useless for applications that require online and in real time slam.

So most modern application today use what we call Binary descriptors, where the image patch around a feature can be described using a relatively small number of pair-wise intensity comparisons. So instead of the sophisticated image gradient histograms we only use small binary scripts. Following these steps :

- **Step 1:** After selecting a keypoint using a feature extraction method, we select a local path of pixels around that feature.
- **Step 2:** We select pairs of those pixels, following a specific distribution which we will address shortly.
- **Step 3:** For each chosen pair, we perform a comparison based on intensity values of the pixel pair

$$b = \begin{cases} 1 & \text{if } I(s_1) < I(s_2) \\ 0 & \text{otherwise} \end{cases} \tag{2.30}$$

So we're basically saying that if the pixel intensity of pixel s1 is smaller than that of pixel s2 return a 1, otherwise return a 0.

- **Step 4:** After each comparison, We concatenate all the pixel pair result into one bit string. Thus we get our descriptor.

These descriptors are incredibly compact, very fast to compute and specifically trivial in their way of comparison to achieve the matching as shown in equation 2.31 .

$$d_{\text{Hamming}}(B_1, B_2) = \sum \text{xor}(B_1, B_2) \tag{2.31}$$

This method of comparison called the Hamming distance is a metric used to measure the difference between two binary strings of equal length. It counts the number of positions at which the corresponding bits are different, so the smaller the hammer distance means fewer bits are different and the better it is.

Binary descriptors differ in their strategy of choosing the bit pairs. A very important thing is that if we choose a bit pair, we must keep the same pairs and maintain the same order in which those pairs are tested. This is how we keep the comparison viable, otherwise it will be useless. We'll be presenting the state of the art descriptor BRIEF than we'll present our chosen one later on.

- **Binary Robust Independent Elementary Features (BRIEF)**

  Also known as BRIEF is the first ever binary image descriptor, proposed by Calonder in his 2010 paper [71]. It' a 256 bit descriptor, meaning that it compares 256 pairs. Since BRIEF deals with the image at a pixel level, it's very sensitive to noise so a pre-smoothing step on the path using a Gaussian kernel is necessary. BRIEF provides five different geometries for choosing the bit pairs as shown in figure 2.17.

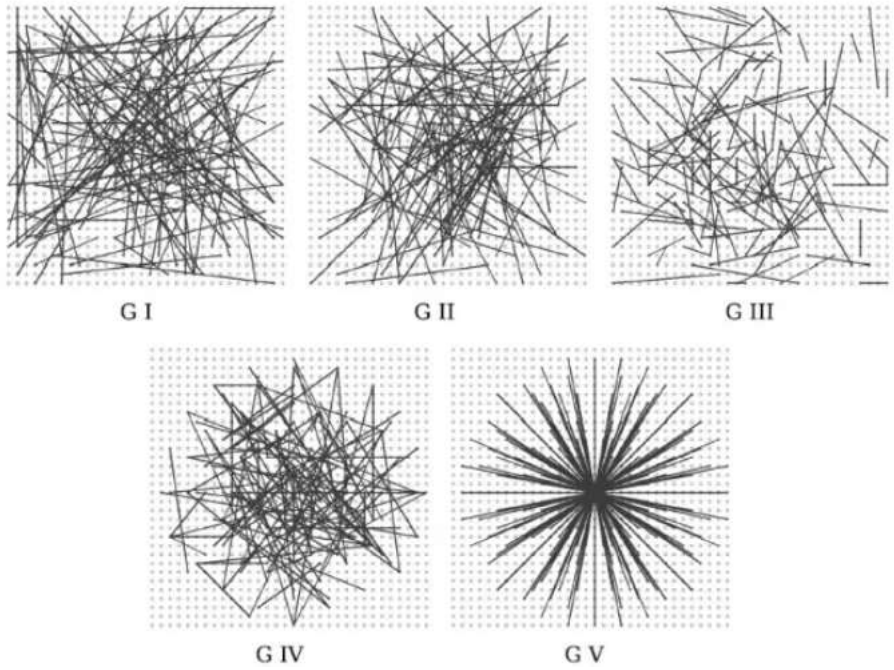


Figure 2.17: Different approaches to choosing BRIEF test locations

a. **Uniform (G I)**: Both x and y pixels in the random pair is drawn from a Uniform distribution or spread of $S/2$ around keypoint. The pair(test) can lie close to the patch border.

b. **Gaussian (G II)**: Both x and y pixels in the random pair is drawn from a Gaussian distribution or spread of $0.04 * S^2$ around keypoint.

c. **Gaussian (G III)**: The first pixel(x) in the random pair is drawn from a Gaussian distribution centered around the keypoint with a stranded deviation or spread of $0.04 * S^2$. The second pixel(y) in the random pair is drawn from a Gaussian distribution centered around the first pixel(x) with a standard deviation or spread of $0.01 * S^2$. This forces the test(pair) to be more local. Test(pair) locations outside the patch are clamped to the edge of the patch.

d. **Coarse Polar Grid (G IV)**: Both x and y pixels in the random pair is sampled from discrete locations of a coarse polar grid introducing a spatial quantization.

e. **Coarse Polar Grid (G V)**: The first pixel(x) in random pair is at $(0, 0)$ and the second pixel(y) in the random pair is drawn from discrete locations of a coarse polar grid.

We note that in [71], it has been proven that the first four distribution perform similarly while outperforming the last non random distribution (G V).
Despite of BRIEF speed and efficiency when it comes to feature matching, there are inherent problems such as it's lack of rotation and scale invariance that cause it to yield worst results in certain situation. We will tackle these problems later on.

ENP Example :

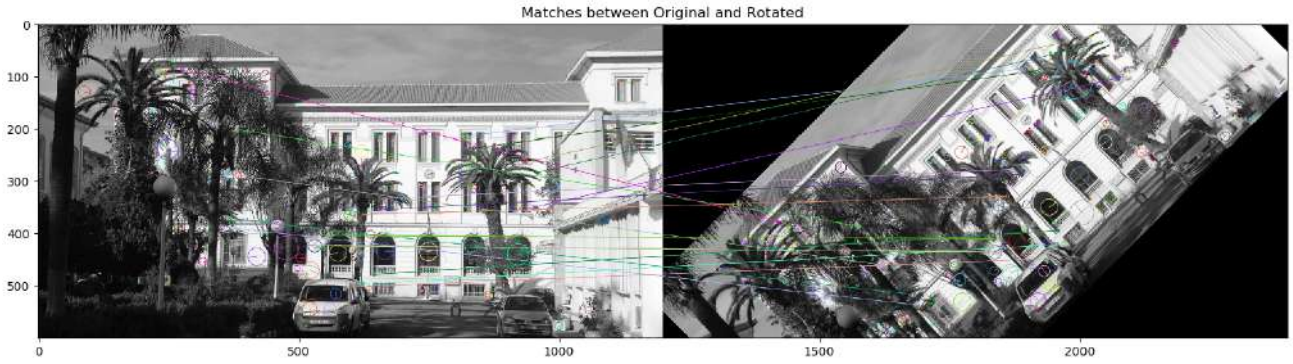


Figure 2.18: BRIEF feature extraction on Ecole Nationale Polytechnique (rotated by 45 degrees

Commentary :

The BRIEF algorithm achieved an impressive execution time of 0.0199 seconds, showcasing its efficiency due to its binary descriptor approach and the use of the FAST keypoint detector. However, the accuracy of matches was inadequate at 19.90%, reflecting BRIEF's limitation in handling significant scale and rotation changes. The article that created the algorithm [**?**] indicated that if the rotation is larger than 30 degrees, the matching looses all it's accuracy. Despite detecting 1377 matches, the algorithm's focus on speed over precision makes it suitable for real-time applications where minor inaccuracies in keypoint matching can be tolerated. So we will choose the BRIEF algorithm for our method but we will apply changes that we will discuss later on to make it more robust.

2.4.3.5 Motion estimation

Now, after we have been able to detect the keypoints or features using methods such as the Harris corner detection algorithm or the difference of gaussians, and after describing them and matching them together using descriptors such as the SIFT descriptor or the BRIEF descriptor, we need to estimate the motion. This step differs depending on the camera model and settings.

- If the camera model is monocular, then we only have the 2D estimate of the pixel coordinates. So the problem is estimating motion according to two sets of 2D points, this is solved using *epipolar geometry*. We will go more in depth into this method later on.

- If the camera is binocular (stereo), RGB-D or we obtain the distance through another method, then we have two sets of 3D points. This is solved using one of the most efficient alignment algorithms that's even used for Lidar-based SLAM, ICP or iterative-closest-point.

- If we have two sets that differ in nature, a 3D set and a 2D set then this problem is solved using PnP or Perspective-n-Point.

As this thesis will deal with a monocular camera setup, we will only be explaining the epipolar geometry method. But the following articles [72] and [73] are perfect for a more in depth understanding of the two other methods.

- **Epipolar Geometry**

  Let's assume we have a pair of matched feature points from two images as displayed in figure 2.19. Our goal is to find the motion between the two consecutive frames, and for easing the explanation process we'll look at the two frames as coming from a right and a left camera.

  The motion from the first frame to the second frame is descried by a rotation matrix R and a translation vector t. Each camera has its own coordinate system with axes $\hat{x}_l, \hat{y}_l, \hat{z}_l$ for the left camera and $\hat{x}_r, \hat{y}_r, \hat{z}_r$ for the right camera, and they're centered in O_l and O_r. Let's consider a 3D point P in our scene which is detected as a feature point in the right camera denoted as u_r and matched in the left camera image denoted as u_l.

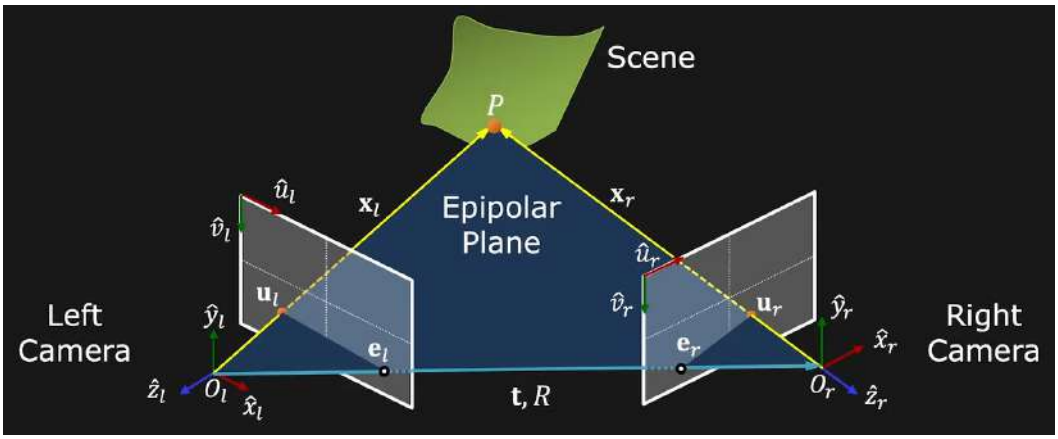


Figure 2.19: The epipolar constraint scene [7]

Now, if we consider the projection of the left camera center O_l into the right camera frame we get the point e_r and on the opposite side we get e_l. We define these two points as the epipoles of the scene, and they're unique to each frame pair. We also define the plane formed by the origins, the epipoles and the scene point P as the Epipolar plane which is unique to every point in the scene 2.20.

We can also define the normal vector n which is perpendicular to the epipolar plane. Hence we can calculate it as the cross product between the unknown translation vector t and the vector x_l.

$$n = t \times x_l \tag{2.32}$$

And since the normal vector is perpendicular to x_l then their dot product is null so we can define our epipolar constraint as follows :

$$x_l.(t \times x_l) = 0 \tag{2.33}$$

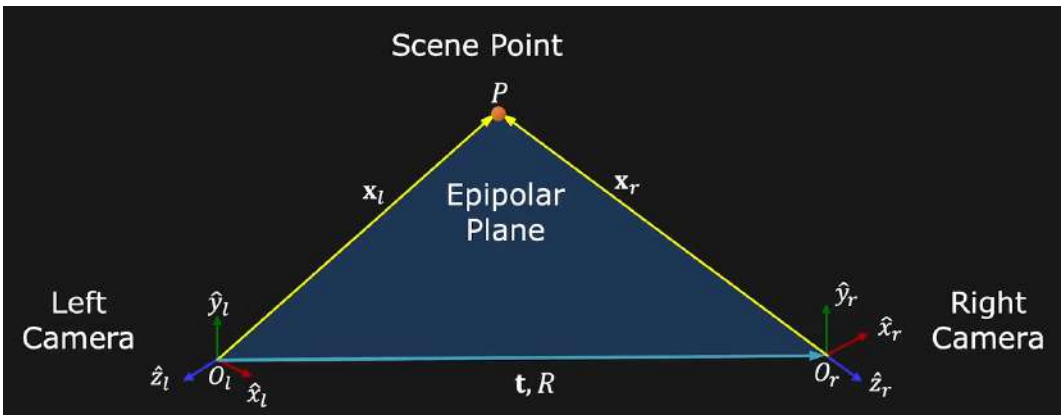


Figure 2.20: The epipolar plane [7]

The goal is to write the epipolar constraint equation with respect to our unknowns t and R.
So, firstly using the definition of a cross-product we get :

$$\begin{bmatrix} x_l & y_l & z_l \end{bmatrix} \begin{bmatrix} t_y z_l - t_z y_l \\ t_z x_l - t_x z_l \\ t_x y_l - t_y x_l \end{bmatrix} = 0 \tag{2.34}$$

Then we write it in a matrix-vector form to get the following:

$$x_l^T . [t]_\times . x_l = \begin{bmatrix} x_l & y_l & z_l \end{bmatrix} \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = 0 \tag{2.35}$$

Where $[t]_\times$ is a skew-symmetric matrix of the translation vector t between the two camera positions. We also define the homogeneous transformation from the right to the left coordinate frame as :

$$\mathbf{x}_l = \mathbf{R}\mathbf{x}_r + \mathbf{t}$$

with $\mathbf{t}_{3\times1}$ being the position of the right Camera in the left camera's frame, and $\mathbf{R}_{3\times3}$ being the orientation of the left camera in the right camera's frame.

$$\begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{2.36}$$

After substituting 2.36 in 2.35, we get the following equation :

$$\begin{bmatrix} x_l & y_l & z_l \end{bmatrix} \left(\begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} + \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \right) = 0 \tag{2.37}$$

And to further simplify this equation, we have the second term that is equivalent to the cross product of the translation vector t with itself $t \times t$, and we know this equates to zero.

$$\begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = t \times t = 0 \tag{2.38}$$

And if we multiply the first two matrices together, we get the Essential matrix $E = [t]_\times R$. A term first introduced in 1981 by Longuet [74] as a way to encapsulate the intrinsic geometry between two views in the same scenes. After substituting it in 2.37, we get our new epipolar constraint :

$$x_r^T.E.x_l = \begin{bmatrix} x_l & y_l & z_l \end{bmatrix} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} = 0 \tag{2.39}$$

As stated before, the translation matrix is a skew-symmetric matrix and the rotation matrix R is an orthonormal matrix. Their product can be decoupled using *Singular Value Decomposition* (SVD), this means that from the essential matrix we can extract our two unknowns t and R.

$$\begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} . \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.40}$$

So now we need to find the essential matrix E based on our new epipolar constraint equation 2.39. The equation still contains the coordinates of our point P in both the camera frames which are unknowns to us, so we replace them with their 2D projections 2.6 as explained in the Data acquisition section.

For that we use their perspective projection equations. We'll derive the formula for the left camera but it's the same for both views.

$$u_l = f_x^{(l)} \frac{x_l}{z_l} + o_x^{(l)} \tag{2.41}$$

$$v_l = f_y^{(l)} \frac{y_l}{z_l} + o_y^{(l)} \tag{2.42}$$

We multiply by z_l

$$z_l u_l = f_x^{(l)} x_l + z_l o_x^{(l)} \tag{2.43}$$

$$z_l v_l = f_y^{(l)} y_l + z_l o_y^{(l)} \tag{2.44}$$

Then we write it in matrix form to get the following equation based on the intrinsic parameters of our camera k:

$$z_l \begin{bmatrix} u_l \\ v_l \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_l \\ y_l \\ z_l \end{bmatrix} \tag{2.45}$$

$$z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r \\ y_r \\ z_r \end{bmatrix} \tag{2.46}$$

So we can express our unknowns coordinate vectors using the 2D projection points and the intrinsic parameter matrix K .

$$x_l^T = \begin{bmatrix} u_l & v_l & 1 \end{bmatrix} z_l K^{-1^T} \tag{2.47}$$

$$x_r = K^{-1} z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} \tag{2.48}$$

Now we substitute 2.47 and 2.48 in our new epipolar constraint equation 2.39 and we rewrite it in terms of image coordinates.

$$\begin{bmatrix} u_l & v_l & 1 \end{bmatrix} z_l K^{-1^T} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} K^{-1} z_r \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0 \tag{2.49}$$

z_l and z_r are the relative depths of the point P with respect to the two frames, meaning they can't be zero (otherwise the point P will be on the image plane). So we get the following :

$$\begin{bmatrix} u_l & v_l & 1 \end{bmatrix} K^{-1^T} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} K^{-1} \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0 \tag{2.50}$$

We define a new matrix, called the Fundamental matrix [75], as such

$$K^{-1^T} \begin{bmatrix} e_{11} & e_{12} & e_{13} \\ e_{21} & e_{22} & e_{23} \\ e_{31} & e_{32} & e_{33} \end{bmatrix} K^{-1} = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \tag{2.51}$$

This gives us the final form of our epipolar constraint equation.

$$\begin{bmatrix} u_l & v_l & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_r \\ v_r \\ 1 \end{bmatrix} = 0 \tag{2.52}$$

This form tells us that in order to calibrate our system, meaning in order to get the homogeneous transformation matrix that contains the translation vector t and rotation matrix R and get our visual odometry, we only need to estimate this Fundamental matrix. So our problem of visual odometry is solved by following these steps :

- **Step A :** Firstly, we need to find a number of corresponding features that are in the frames through one of the feature extraction algorithms that we presented earlier. And then match them together, then we write our epipolar constraint for each correspondence.

$$\begin{bmatrix} u_l^{(i)} & v_l^{(i)} & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_r^{(i)} \\ v_r^{(i)} \\ 1 \end{bmatrix} = 0 \tag{2.53}$$

  We expand the matrix to get the linear equation:

$$(f_{11}u_r^{(i)} + f_{12}v_r^{(i)} + f_{13})u_l^{(i)} + (f_{21}u_r^{(i)} + f_{22}v_r^{(i)} + f_{23})v_l^{(i)} + f_{31}u_r^{(i)} + f_{32}v_r^{(i)} + f_{33} = 0 \tag{2.54}$$

- **Step B:** Rearrange terms to form a linear system $\mathbf{Af} = 0$, where $\mathbf{f}$ is our only unkown.

$$\begin{bmatrix} u_l^{(1)}u_r^{(1)} & u_l^{(1)}v_r^{(1)} & u_l^{(1)} & v_l^{(1)}u_r^{(1)} & v_l^{(1)}v_r^{(1)} & v_l^{(1)} & u_r^{(1)} & v_r^{(1)} & 1 \\ u_l^{(2)}u_r^{(2)} & u_l^{(2)}v_r^{(2)} & u_l^{(2)} & v_l^{(2)}u_r^{(2)} & v_l^{(2)}v_r^{(2)} & v_l^{(2)} & u_r^{(2)} & v_r^{(2)} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_l^{(m)}u_r^{(m)} & u_l^{(m)}v_r^{(m)} & u_l^{(m)} & v_l^{(m)}u_r^{(m)} & v_l^{(m)}v_r^{(m)} & v_l^{(m)} & u_r^{(m)} & v_r^{(m)} & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.55}$$

- **Step C :** The properties of a homogeneous transformation are unaffected by scale meaning if we enlarge everything in a scene, the transformation would still be the same. Our fundamental matrix acts directly on the homogeneous coordinates, this indicates that we can set an arbitrary scale for $\mathbf{f}$ as follows

$$\|\mathbf{f}\|^2 = 1 \tag{2.56}$$

- **Step D :** We want $\mathbf{Af}$ as close to 0 as possible and $\|\mathbf{f}\|^2 = 1$:

  There are a lot of algorithms coined specifically for solving this problem, but they're all based on a least squared solution.

$$\min_{\mathbf{f}} \|\mathbf{Af}\|^2 \quad \text{such that} \quad \|\mathbf{f}\|^2 = 1 \tag{2.57}$$

- **Step E :** Now that we have our Fundamental matrix F, we can find the Essential matrix E

$$E = K_l^T.F.K_r \tag{2.58}$$

- **Step F :** And finally, using the liner algebra property of the Essential matrix E, we can decouple it using Singular Value Decomposition.

$$E = [t]_\times.R \tag{2.59}$$

  And the visual odometry problem is now solved.

2.4.3.6 Depth estimation through Triangulation

After extracting our features, describing them and matching them together, and after calculating our homogeneous transformation from the visual odometry, we need to get the coordinates of our chosen features in the 3D world.
This step relies heavily on the chosen camera sensor, we have touched on the methods of estimating depth briefly for Stereo and RGB-D cameras in section 2.4.2.2. So, we will address this problem for a monocular setup as the one we have, and we do so using a method called

Triangulation.

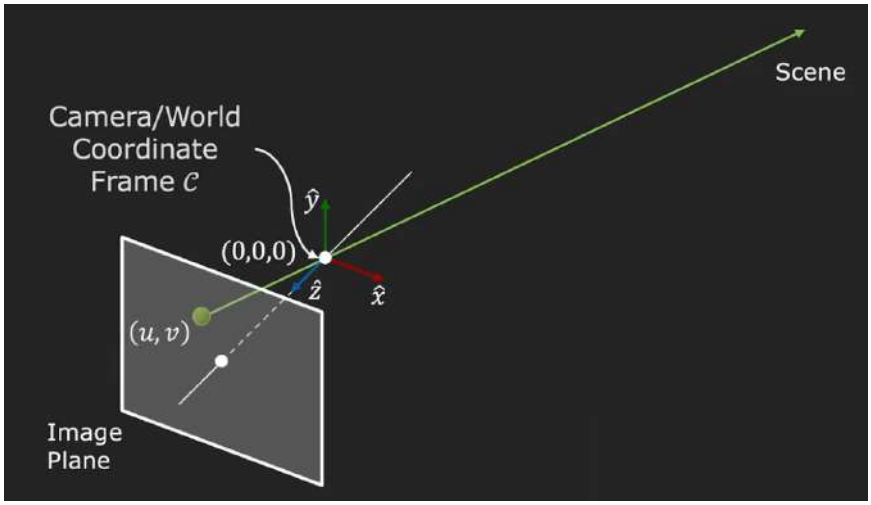


Figure 2.21: The 2D-3D outgoing ray[8]

Assuming we have a fully calibrated monocular system, meaning the intrinsic and extrinsic parameter matrices are in our disposal then we have the 3D-to-2D point projection as proven previously in 2.6. The 2D-to-3D projection is what we don't have, but we know with full certainty that they lay within a specific ray as shown in 2.21 that satisfies the following equations:

$$x = z\frac{u - o_x}{f_x} \tag{2.60}$$

$$y = z\frac{v - o_y}{f_y} \tag{2.61}$$

$$z > 0 \tag{2.62}$$

A single frame isn't enough for 3D image reconstruction, we need more information. We do so by adding another frame to make a stereo like system as shown in figure 2.27 In the given stereo vision setup, (u_l, v_l) and (u_r, v_r) are the 2D coordinates of the matched features in the left and right camera images, respectively. These coordinates correspond to the projections of the same 3D point (x, y, z) in the scene.

$$\begin{aligned} u_l &= f_x\frac{x}{z} + o_x, \\ v_l &= f_y\frac{y}{z} + o_y, \\ u_r &= f_x\frac{x - b}{z} + o_x, \\ v_r &= f_y\frac{y}{z} + o_y. \end{aligned} \tag{2.63}$$

The cameras are separated by a horizontal baseline b, which represents the translation vector $\mathbf{t}$ between the left and right frames. This means that we have a system of four equation to solve for four unknowns, and solving for (x, y, z) gives us our 3D projection.

$$x = \frac{b(u_l - o_x)}{(u_l - u_r)} \tag{2.64}$$

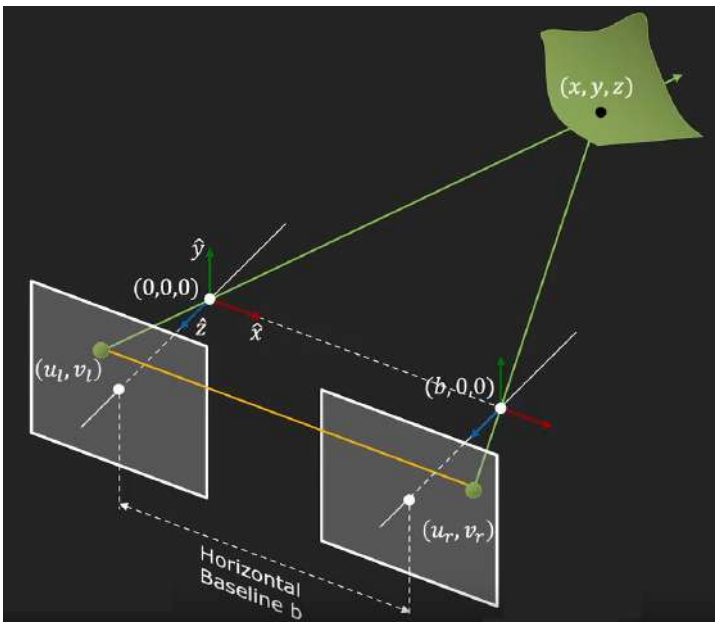


Figure 2.22: Finding 2D-3D outgoing ray using a stereo system [8]

$$y = \frac{bf_x(v_l - o_y)}{f_y(u_l - u_r)} \tag{2.65}$$

$$z = \frac{bf_x}{(u_l - u_r)} \tag{2.66}$$

Note that the z coordinate represents the depth of our point, and as we can see it is inversely proportional to $(u_l - u_r)$ which we call disparity, this tells us that the closer the point gets to the image the bigger the disparity will be. And this disparity is proportional to the baseline. Now we have successfully reconstructed the 3D coordinates of the features.

2.4.4 State Estimation

In a perfect world, the Visual SLAM problem is solved just using visual odometry. We were able to detect features and extract their 3D coordinates which form the landmarks of our map, as well as extract the trajectory of our camera by estimating its movement just from image frames. But the perfect case does not exist; the measurements are noisy due to their quality, which accounts for large errors that accumulate over time. Therefore, we care about how much noise our measurements contain, how it is carried from one time step to the other, and how confident we are in those measurements.

These problems are solved through backend optimization techniques that estimate our states, which include the robot's own trajectory as well as the environment map. This part is so vital that in early works [53], SLAM was dubbed a state estimation problem only. In contrast, the visual odometry part is usually referred to as the frontend, where it provides data to be optimized by the backend as well as initial values.

There are many state estimation algorithms, which can be categorized into two classes :

1. **Incremental methods :** These methods are all based on the Bayes filter where we have a prediction step and a correction step and our aim is to update the state estimate as new

measurements arrive. The different implementation of the Bayes filter based on different assumptions for some popular incremental methods that include:

- **Extended Kalman Filter :** Extended Kalman Filters (EKF) are used in SLAM to estimate the robot's pose and the map by maintaining a Gaussian distribution of the state belief. EKF-SLAM linearizes the non-linear motion and measurement, allowing the use of Kalman filter techniques. The algorithm involves predicting the state and covariance based on the motion model, updating them with sensor measurements, and correcting the estimates using the Kalman gain. EKF-SLAM is efficient for real-time applications and handles uncertainties in the state and measurements, but it may struggle with highly non-linear systems and large-scale environments due to linearization errors and computational complexity. It was introduced by [53].

- **Particle Filter:** Particle filters, are used in SLAM to estimate the distribution of a robot's pose and map by representing the belief with a set of particles. Each particle represents a possible state (pose) of the robot and is associated with a weight indicating its likelihood given the observations and controls. The algorithm involves initializing particles, predicting their movement based on the motion model, updating their weights based on sensor measurements, and resampling particles to focus on more likely states. Particle filters can handle non-Gaussian and multi-modal distributions, making them robust for complex, real-world environments, but they can be computationally intensive and require efficient resampling to avoid degeneracy.. This approach was introduced in [57] and [76].

2. **Batch Methods:** These methods reprocess all past measurements to optimize the state estimates. Given their relevance to our chosen approach, we will be exploring two of the most widely used batch methods in detail in the following sections.

2.4.4.1 Pose Graph Optimization (PGO)

Pose graph optimization is method that not only estimates the most recent pose of the camera but also helps refine the estimation of past poses over time. In this approach, these poses are referred to as nodes and are linked by edges as shown in figure 2.23, which represent constraints between poses that must be respected to accurately optimize the motion path. Consequently, the state of the robot is represented by a high-dimensional state vector containing selected poses to approximate the true trajectory.

1. **Mathematical formulation**

   The poses of the camera as well as the constraints, are retrieved through the process of visual odometry as discussed previously. The goal of the optimization approach is to find the node configuration (pose graph) that minimizes the error introduced by the constraints, we add landmark point nodes to make the graph more expressive, and form a non-optimized map of the environment as shown in figure 2.24.
   The graph consists of n nodes $x = x_{1:n}$, where each x_i is a pose of the robot at time t_i. A constraint or edge exists between the nodes x_i and x_j if :

   a. The robot moves from x_i to x_{i+1}, with the edge corresponding to visual odometry. As shown in figure 2.23

   b. if the robot observes the same part of the environment from x_i and x_j, representing measurements from both poses. As shown in figure 2.24

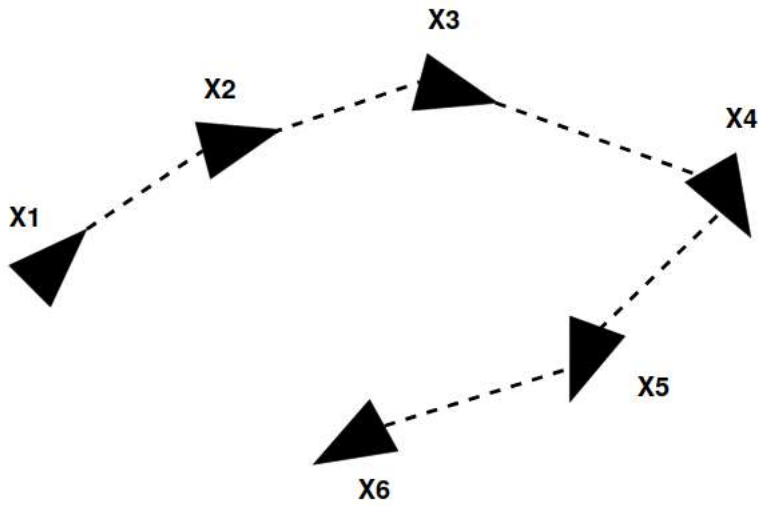


Figure 2.23: A pose graph of nodes and edges

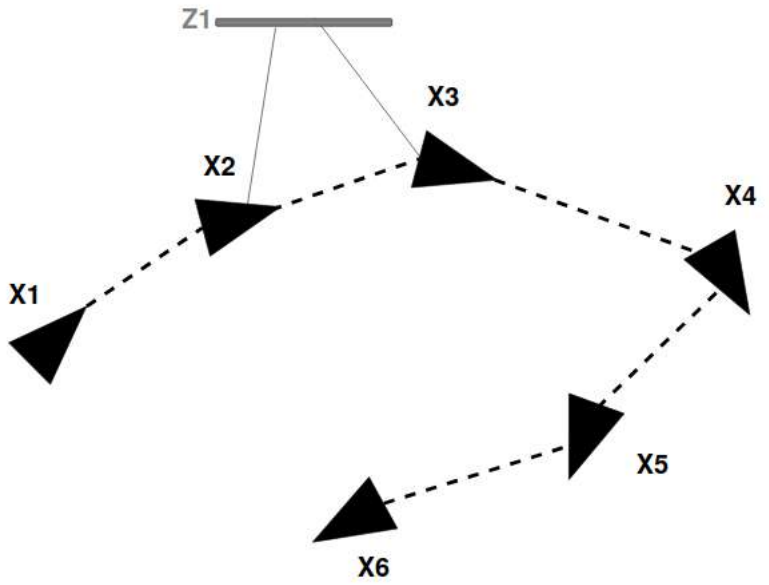


Figure 2.24: A pose graph with added landmarks

In pose graph optimization, the key challenge is dealing with the uncertainty in the constraints between poses, which can cause drift over time as shown in 2.25. This drift is a result of accumulated errors in the odometry and observations. To address this, we rely on the concept of loop closure to do a closed loop estimation. A loop closure is when the robots returns to a previously seen location, in our case when the camera detects the same set of features. When the loop closure takes place, the graph optimization algorithm runs to help improve the estimation of all poses in the state vector.
To express this approach we use homogeneous coordinates to show transformations between poses. The odometry-based edge is expressed as $(\mathbf{X_i^{-1}X_{i+1}})$ where this how the node i sees the node i+1, which describes how node i sees node j, is expressed as $(\mathbf{X_i^{-1}X_j})$. The pose graph consists of nodes representing robot poses and edges representing constraints or transformations between these poses.

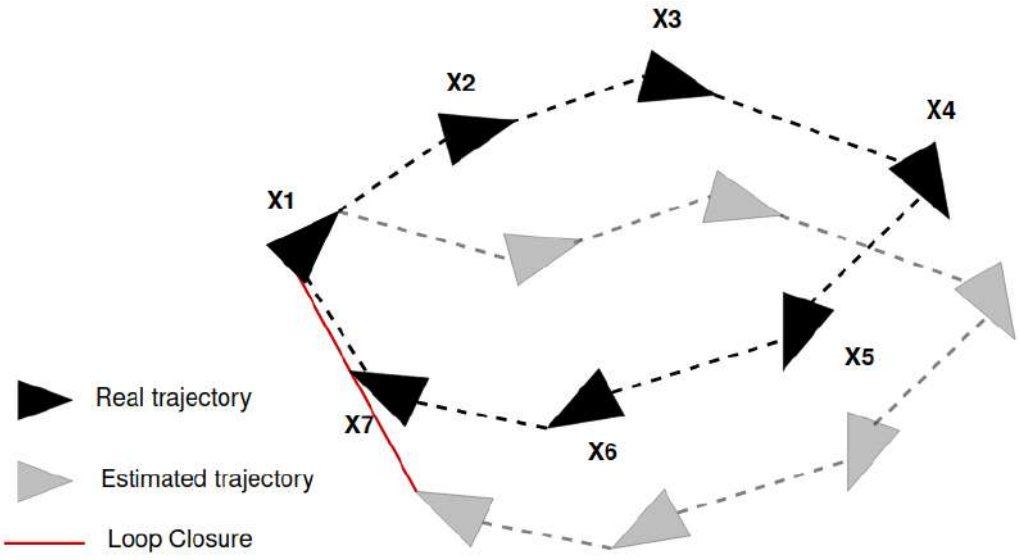


Figure 2.25: A pose graph with added landmarks

To incorporate the uncertainty, we define an information matrix Ω_{ij} for each edge, which encodes the confidence in the corresponding constraint. This matrix is typically the inverse of the covariance matrix of the measurement noise. The "bigger" Ω_{ij} is, the more that edge "matters" in the optimization process, the less noisy it is.

Using what we have we can now formulate our error function as depicted in figure 2.26

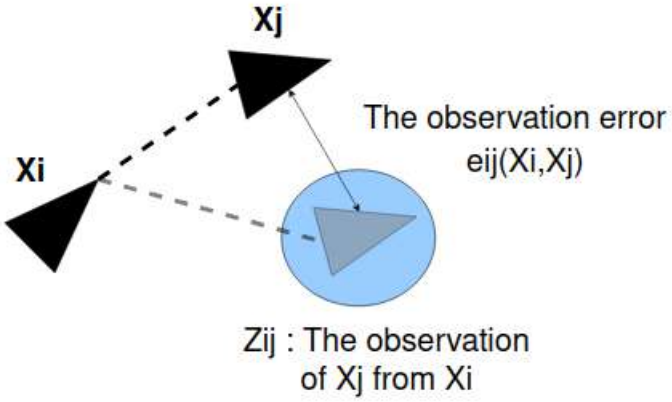


Figure 2.26: Formulation of the error

Mathematically, the error function for a single constraint is defined as:

$$e_{ij}(x_i, x_j) = t2v(Z_{ij}^{-1}(X_i^{-1}X_j)) \tag{2.67}$$

where Z_{ij} is the observed transformation between poses x_i and x_j, and $X_i^{-1}X_j$ is the estimated transformation. This basically indicates that we took a forward transformation form the node x_i to the node x_j and then a backward transformation from the observed x_j to x_i, and we check their difference which forms our error function e_{ij}.
Hence we can formulate our quadratic objective function to minimize the error as such :

$$x^* = \arg\min_x \sum_k e_k^T(x)\Omega_k e_k(x) \tag{2.68}$$

We note that the function $t2v$ converts the homogeneous transformation matrix to a vector representation for the purpose of simplifying the error function before optimizing it,as it

is simpler and less computationally costly to work with vectors rather than matrices. Here's a simple example for a 2D transformation matrix T:

$$T = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & t_x \\ \sin(\theta) & \cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

The $t2v$ function would convert this into:

$$t2v(T) = \begin{bmatrix} t_x \\ t_y \\ \theta \end{bmatrix}$$

2. **Objective function minimization**

   a. **Linearizing the Error Function**

   We can approximate the error functions around an initial guess x via Taylor expansion:

   $$e_{ij}(x + \Delta x) \approx e_{ij}(x) + J_{ij}\Delta x \tag{2.69}$$

   with

   $$J_{ij} = \frac{\partial e_{ij}(x)}{\partial x}$$

   b. **Derivative of the Error Function**

   A crucial result of having constraints depend only on the actual and previous node only, is that the error function does not depend on all state variables but only on x_i and x_j.
   This will have a direct result on the structure of the Jacobian because it will be non-zero only in the rows corresponding to x_i and x_j:

   $$\frac{\partial e_{ij}(x)}{\partial x} = \begin{bmatrix} 0 & \cdots & \frac{\partial e_{ij}(x)}{\partial x_i} & \cdots & 0 & \cdots & \frac{\partial e_{ij}(x)}{\partial x_j} & \cdots & 0 \end{bmatrix}$$

   c. **Jacobian and Sparsity**

   Error $e_{ij}(x)$ depends only on the two parameter blocks x_i and x_j:

   $$e_{ij}(x) = e_{ij}(x_i, x_j)$$

   The Jacobian will be zero everywhere except in the columns of x_i and x_j:

   $$J_{ij} = \begin{bmatrix} 0 & \cdots & \frac{\partial e_{ij}(x)}{\partial x_i} & \cdots & 0 & \cdots & \frac{\partial e_{ij}(x)}{\partial x_j} & \cdots & 0 \end{bmatrix}$$

   d. **Building a linear system out of Sparsity**

   By substituting 2.69 in the objective function 2.68, and after further development we find the following equation

   $$\sum_{(i,j)\in\mathcal{C}} \left(2e_{ij}(x_k)^T \Omega_{ij} J_{ij} \Delta x + \Delta x^T J_{ij}^T \Omega_{ij} J_{ij} \Delta x \right) \tag{2.70}$$

That can be written as $\Delta x^T H \Delta x + b^T \Delta x$. The optimal increment Δx is found by setting the gradient of the quadratic function to zero, so to find the node configuration that minimizes the error, we'll have to solve the following linear system :

$$H\Delta x = b$$

Where the he Hessian matrix is:

$$H_{ij} = J_{ij}^T \Omega_{ij} J_{ij}$$

And that the gradient vector is:

$$b_{ij}^T = e_{ij}^T \Omega_{ij} J_{ij}$$

Summing over all state variables and constraints we get :

$$b^T = \sum_{ij} b_{ij}^T = \sum_{ij} e_{ij}^T \Omega_{ij} J_{ij}$$

$$H = \sum_{ij} H_{ij} = \sum_{ij} J_{ij}^T \Omega_{ij} J_{ij}$$

Where it is non-zero only at the indices corresponding to x_i and x_j. This will give us a very sparse structure of H which is a direct result on the sparse structure of the Jacobian J_{ij}.

The process can be compacted through the following algorithm that runs each time a loop closure is detected:

Algorithm 2 Pose Graph Optimization Algorithm

1: **Initialize:** Initial guess for poses $\mathbf{x}$
2: **while** not converged **do**
3: Build the linear system: $(\mathbf{H}, \mathbf{b}) \leftarrow \text{buildLinearSystem}(\mathbf{x})$
4: Solve the sparse linear system: $\Delta\mathbf{x} \leftarrow \text{solveSparse}(\mathbf{H}\Delta\mathbf{x} = -\mathbf{b})$
5: Update the poses: $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}$
6: **end while**
7: **Return:** Optimized poses $\mathbf{x}^*$

Here's an example that shows how the PGO algorithm relied on that loop closure to correct the accumulated drift.

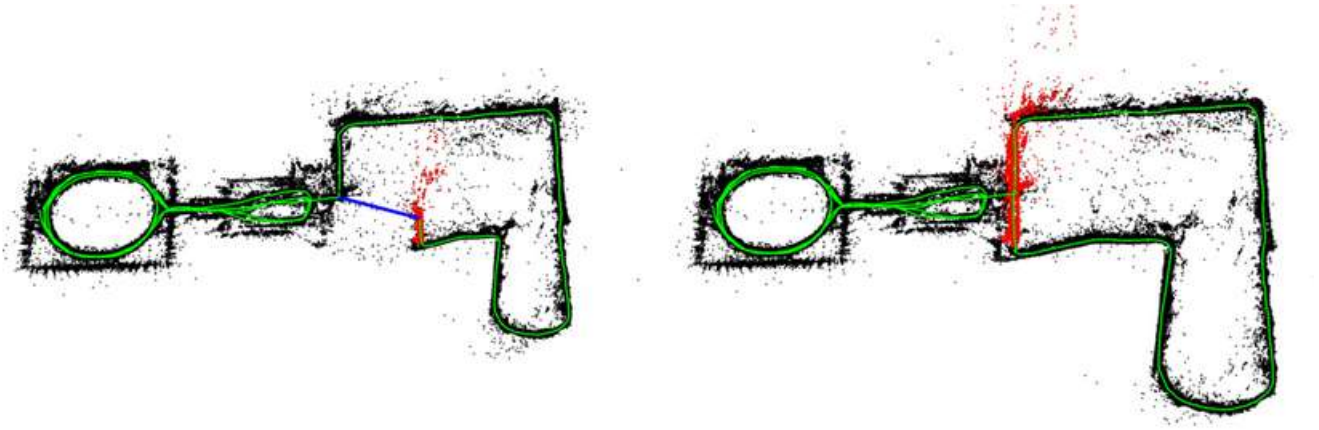


Figure 2.27: Loop Closure detected [9]

2.4.4.2 Bundle Adjustment (BA)

Bundle adjustment is a backend optimization technique used in visual slam with the goal of refining the 3D coordinates of the landmarks (features) and the camera poses simultaneously. This is done by minimizing the reprojection error as shown in figure 2.28, which is the difference between the observed 2D points in the images and their projected 3D points. The reprojection fault occur due to various factors such as the imperfections of the camera model from lens distortion, inaccurate intrinsic parameters and even bad camera calibration, as well is subpar feature matching. By optimizing these parameters, BA improves the accuracy of both the camera trajectory and the mapped environment.

1. **Mathematical Formulation**

   In BA, we define the problem as follows: Given a set of observed 2D points in multiple images, we aim to find the 3D coordinates of the corresponding landmarks and the camera poses that minimize the reprojection error. The reprojection error is the difference between the observed 2D points and the projected 3D points.

   The reprojection of a 3D point X_i in the world frame to a 2D point u_{ij} in the image frame of the j-th camera is given by:

   $$u_{ij} = \pi(K[R_j|t_j]X_i)$$

   where π is the projection function, K is the camera intrinsic matrix, R_j and t_j are the rotation and translation of the j-th camera respectively.

   The objective function to be minimized is:

   $$\min_{R_j,t_j,X_i} \sum_{i=1}^{M} \sum_{j \in \mathcal{O}(i)} \|u_{ij} - \pi(K[R_j|t_j]X_i)\|^2$$

   where $\mathcal{O}(i)$ is the set of observations of the i-th landmark.

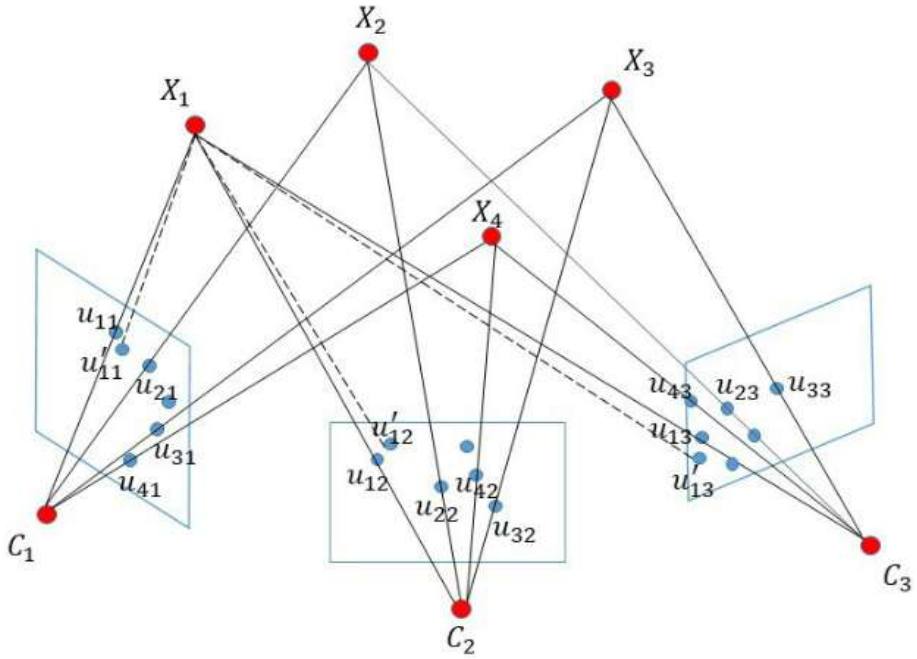


Figure 2.28: Bundle adjustement

2. **Objective function minimization**

We follow the same procedure for the PGO algorithm.

a. **Linearizing the Error Function**

To minimize the objective function, we linearize the error function around the current estimate using a first-order Taylor expansion:

$$e_{ij}(X_i, R_j, t_j) \approx e_{ij}(X_i^0, R_j^0, t_j^0) + \frac{\partial e_{ij}}{\partial X_i}\Delta X_i + \frac{\partial e_{ij}}{\partial R_j}\Delta R_j + \frac{\partial e_{ij}}{\partial t_j}\Delta t_j \tag{2.71}$$

where e_{ij} is the reprojection error, and ΔX_i, ΔR_j, Δt_j are the increments to the estimates.

b. **Jacobian and Sparsity**

The Jacobian matrix of the error function with respect to the parameters X_i, R_j, and t_j is sparse because each error term depends only on a subset of the parameters. This sparsity is exploited to efficiently solve the resulting linear system.

The Jacobian for the reprojection error can be written as:

$$J_{ij} = \begin{bmatrix} \frac{\partial e_{ij}}{\partial X_i} & \frac{\partial e_{ij}}{\partial R_j} & \frac{\partial e_{ij}}{\partial t_j} \end{bmatrix} \tag{2.72}$$

c. **Building a Linear System**

By stacking the linearized error functions for all observations, we obtain a linear system of the form:

$$H\Delta x = -b$$

where H is the Hessian matrix, Δx is the stacked vector of increments, and b is the gradient vector.

The Hessian matrix and the gradient vector are computed as:

$$H = \sum_{i,j} J_{ij}^T J_{ij}$$

$$b = \sum_{i,j} J_{ij}^T e_{ij}$$

d. **Solving the Linear System**

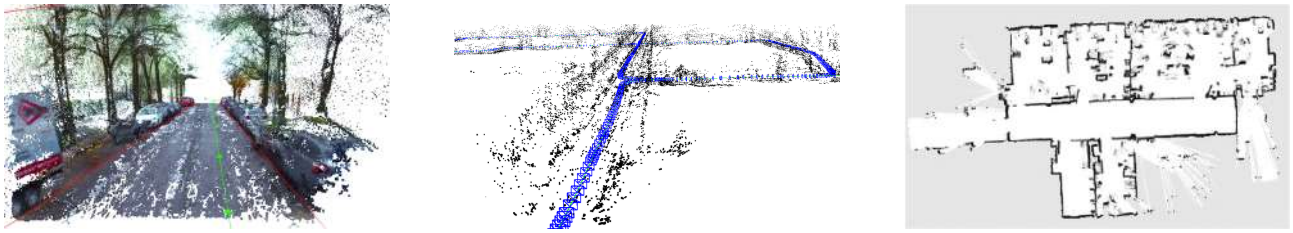
The linear system is solved iteratively using methods such as Gauss-Newton or Levenberg-Marquardt. The solution Δx is used to update the estimates of the camera poses and landmark positions:

$$x \leftarrow x + \Delta x$$

Bundle Adjustment achieves high accuracy by jointly optimizing the 3D points and camera poses. However, the main limitation of BA is its computational intensity. The optimization process is complex and can be slow, especially for large-scale maps or when performing global BA. Efficient implementations and approximations are necessary to make it feasible for real-time applications.

2.4.5 Mapping

Mapping has the goal of constructing a representation of the environment. This representation helps a robot or drone navigate and understand its surroundings. There are several types of mapping techniques, each with distinct characteristics and applications.



(a) Dense mapping (b) Sparse mapping (c) Grid mapping

Figure 2.29: Types of mapping

2.4.5.1 Dense mapping

Dense mapping involves creating a highly detailed and continuous representation of the environment. It captures fine-grained details of surfaces and objects, providing a rich 3D model. These maps are characterized by high resolution and complete surface representation, offering a continuous surface model that includes all visible surfaces in the environment. However, dense mapping is computationally intensive and requires significant resources and storage, which excludes it from our application.

2.4.5.2 Sparse mapping

Sparse mapping creates a representation of the environment using a limited number of significant features or landmarks, focusing on capturing essential elements rather than every detail. This method relies on key points or features, such as corners and edges, and is computationally less intensive compared to dense maps, making it suitable for real-time applications. Sparse maps are compact and require less storage since they only include crucial landmarks.This technique is ideal for mobile robotics or for our application of SLAM in drones, which is why it will be used in our method.

2.4.5.3 Grid mapping

Grid mapping involves dividing the environment into a grid of cells, with each cell representing a small area with a certain occupancy status (occupied, free, or unknown). This technique provides a structured and straightforward representation of space, often in the form of occupancy grids where each cell has a probability of being occupied. Grid maps are simple to understand and implement, making them a popular choice for many robotic applications. They are scalable to cover large areas by increasing the number of cells, though this increases memory usage. Grid mapping is commonly used in robotic navigation and path planning, autonomous vehicle systems, and environments where a straightforward representation of free and occupied space is needed.

2.5 The chosen approach ORB SLAM

2.5.1 Introduction to ORB SLAM

ORB-SLAM (Oriented FAST and Rotated BRIEF SLAM) is a feature-based simultaneous localization and mapping (SLAM) system that utilizes monocular, stereo, and RGB-D cameras. Developed by Mur-Artal, Montiel, and Tardos in 2015 [9],and it's renowned for its real-time performance, robustness, and accuracy. The system combines the efficiency of ORB features,which we will showcase shortly, with advanced SLAM techniques to provide a highly reliable solution for visual SLAM applications.

2.5.2 ORB feature extraction and matching

After testing multiple state of the art feature extraction methods and multiple feature descriptors, we proved that binary detectors and descriptors such as FAST and BRIEF, outperform the others when it comes to speed. But they lack robustness due to their lack of scale and rotation invariance, that's why we opted for the ORB features which provide additional changes to those two algorithms.

2.5.2.1 Oriented FAST

- **Assigning orientation :**

  As previously discussed, FAST does not have an orientation component, which is necessary for rotation invariance. To address this, ORB incorporates the orientation of keypoints detected by FAST. For each keypoint, the orientation is computed using the intensity centroid method. That is based on the moment of a patch of the image around the keypoint. The moments m_{pq} of the patch are defined as:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \tag{2.73}$$

  Where the zeroth moment m_{00} represents the sum of the pixel intensities in the image. Essentially, it is the total "mass" of the intensity function.

$$m_{00} = \sum_{x} \sum_{y} I(x,y) \tag{2.74}$$

  And the first moments m_{10} and m_{01} represent the weighted sums of the pixel intensities along the x and y axes, respectively.

$$m_{10} = \sum_{x} \sum_{y} xI(x,y) \tag{2.75}$$

$$m_{01} = \sum_{x} \sum_{y} yI(x,y) \tag{2.76}$$

  where $I(x,y)$ is the intensity at pixel (x,y). The centroid (C_x, C_y) of the patch is then calculated as:

$$C_x = \frac{m_{10}}{m_{00}}, \quad C_y = \frac{m_{01}}{m_{00}} \tag{2.77}$$

  The orientation of the keypoint is determined by the angle θ of the vector from the keypoint to the centroid:

$$\theta = \text{atan2}(m_{01}, m_{10}) \tag{2.78}$$

- **Assigning Scale :**

Scale invariance refers to the ability to extract the same feature even if the it is bigger or smaller size, meaning even if the camera gets closer or further. So its importance is understandable. Fast however doesn't provide that invariance so we need to include it though what is known as a scale pyramid.

1. **Blurring and Downsampling**: The image is repeatedly smoothed and downsampled to create a series of images at different scales. This is done using a Gaussian blur, which can be represented as:

$$I_\sigma(x, y) = I(x, y) * G_\sigma(x, y) \tag{2.79}$$

where $I(x, y)$ is the original image, $G_\sigma(x, y)$ is the Gaussian kernel with standard deviation σ, and $*$ denotes convolution.

2. **Constructing the Scale Pyramid**: Each level of the pyramid is constructed by applying a Gaussian blur followed by downsampling. Let I^0 be the original image, then the image at level s of the pyramid I^s can be constructed as:

$$I^s(x, y) = I^{s-1}(x', y') * G_{\sigma_s}(x, y) \tag{2.80}$$

where σ_s is the standard deviation of the Gaussian kernel at scale s, and (x', y') are the downsampled coordinates.

3. **FAST Corner Detection at Multiple Scales**: Apply the FAST corner detection algorithm on each level of the pyramid. For a point to be detected as a corner, it must satisfy the corner detection criteria at its respective scale.

$$\text{FAST}(I^s)$$

This results in a set of keypoints detected at each scale.

4. **Combining Keypoints**: The keypoints detected at different scales need to be combined. This involves scaling the coordinates of the keypoints detected in the downsampled images back to the original image scale. If a keypoint is detected at scale s with coordinates (x_s, y_s), its coordinates in the original image scale (x, y) can be computed as:

$$x = x_s \cdot 2^s$$
$$y = y_s \cdot 2^s$$

2.5.2.2 Rotated BRIEF

The key idea behind Rotated BRIEF (rBRIEF) is that, regardless of the rotation of the image, the keypoint's orientation remains consistent. That's why we use the orientation calculated by the oFAST algorithm , and each time that feature is detected it will be rotated to that orientation before being matched.

1. **Rotation of Sampling Pattern**

With the orientation θ determined, the BRIEF sampling pattern is rotated. For each point pair (x_i, y_i) in the BRIEF pattern, the rotated coordinates (x_i', y_i') are computed as:

$$\begin{pmatrix} x_i' \\ y_i' \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

2. **Descriptor Computation**

   Using the rotated sampling pattern, the binary descriptor is computed. For each pair of points (x'_i, y'_i) and (x'_j, y'_j), the binary string is formed based on intensity comparisons:

$$\text{rBRIEF}_k = \begin{cases} 1 & \text{if } I(x'_i, y'_i) < I(x'_j, y'_j) \\ 0 & \text{otherwise} \end{cases}$$

   where $I(x, y)$ represents the intensity at coordinates (x, y).

2.5.2.3 ENP Example

We apply the ORB feature algorithm to the same image and try and compare the results

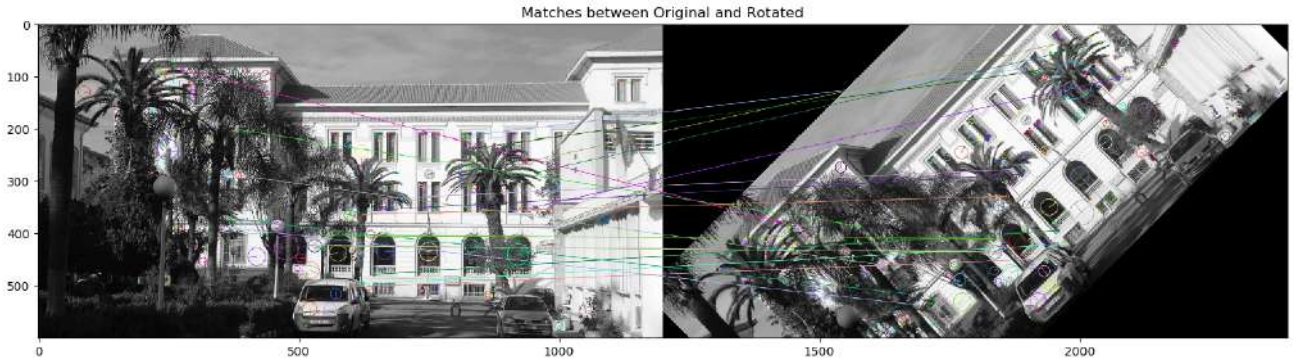


Figure 2.30: ORB feature extraction on Ecole Nationale Polytechnique (rotated by 45 degrees

- **Commentary**

  The ORB feature detection and matching process resulted in identifying 343 matches with a high matching accuracy of 92.42% in an efficient execution with a time of 0.0527 seconds. This highlights ORB's robustness and reliability in feature detection and matching, even under significant image transformations such as rotation. Which is due to the changes we made to the FAST and BRIEF algorithms.

- **Comparison**

Algorithm	Keypoint Detector	Execution Time (seconds)	Number of Matches	Matching Accuracy
ORB	oFAST + rBRIEF	0.0527	343	92.42%
SIFT	DoG	0.4522	N/A	80.81%
BRIEF	FAST	0.0199	1377	19.90%

Table 2.2: Comparison of Feature Detection Algorithms

Based on the performance metrics, we can see that ORB offers a good balance between speed and accuracy, making it suitable for real-time applications.And BRIEF, while extremely fast, lacks robustness against rotations and scale changes, necessitating enhancements for better performance. Finally SIFT, although accurate, is too slow for real-time processes.

2.5.3 System Overview

The ORB-SLAM algorithm workflow can be divided into three main threads that run in parallel as shown 2.31, which are tracking, local mapping and loop closing

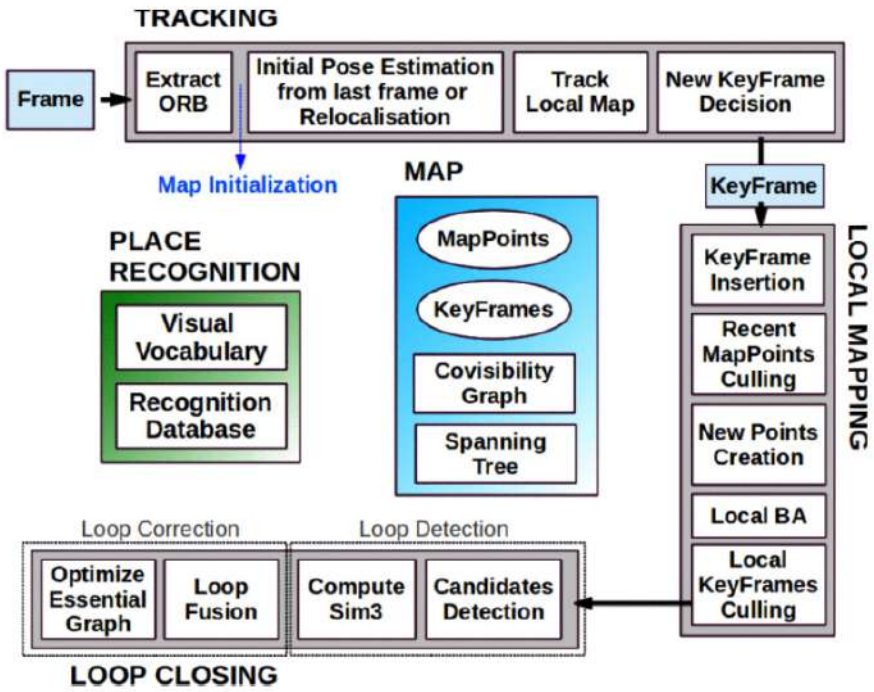


Figure 2.31: ORB SLAM Workflow [9]

We will explain each part separately in a brief manner because most of the algorithms used have already been presented extensively in the previous section. ORB SLAM just proposed a novel way on how to connect each part in the most optimized manner possible.

2.5.3.1 Tracking

1. **Feature Extraction:**

   The ORB feature extraction is performed as presented in [9] via an 8-level pyramid scale. This method has been shown to improve accuracy by ensuring a homogeneous distribution of detected corners. The pyramid scale is divided into grids, with each cell containing at least five features. The benchmark number of features, as indicated in [77], is 2000. This distribution helps maintain consistency and robustness in feature detection across different scales.

2. **Pose Prediction:**

   In ORB-SLAM, the tracking thread uses a motion model to predict the camera's current pose based on previous poses and velocities. This prediction provides a good initial estimate for the pose optimization process. One commonly used motion model is the constant velocity model, which assumes that the camera moves with a constant velocity between frames.

   The constant velocity model assumes that the camera's motion between consecutive frames can be approximated by a constant velocity. Mathematically, this can be ex-

pressed as:

$$\mathbf{T}_{k+1} = \mathbf{T}_k \mathbf{T}_{k-1}^{-1} \mathbf{T}_k \tag{2.81}$$

Here, $\mathbf{T}_k$ represents the camera pose at time step k, and $\mathbf{T}_{k-1}$ represents the camera pose at the previous time step $k-1$. The term $\mathbf{T}_{k-1}^{-1}\mathbf{T}_k$ computes the relative transformation between the previous two frames, which is then applied to the current pose $\mathbf{T}_k$ to predict the pose $\mathbf{T}_{k+1}$ at the next time step.

3. **Initial Pose Estimation and Local Map Tracking:**

   The local map consists of keyframes and map points that are close to the current camera pose $\mathbf{T}_{k+1}$. ORB features are extracted from the current frame and matched with the features in the local map using a descriptor matcher. The matching process involves finding correspondences between the 2D keypoints in the current frame and the 3D map points in the local map, using the predicted pose to narrow down the search space.

4. **Keyframe Decision:**

   Deciding whether or not to insert a new keyframe depends on three main factors: tracking quality, distance from the last keyframe, and scene changes.

   - **Tracking Quality:** The system checks the number of points that are being successfully tracked in the current frame. If the number of tracked points falls below a certain threshold, it may indicate that a new keyframe is needed to stabilize the tracking. Additionally, the spatial distribution of matched points across the image is considered. If the matches are unevenly distributed, a new keyframe may be required to ensure robust tracking.

   - **Distance from the Last Keyframe:** The system measures the distance the camera has moved since the last keyframe. If this distance exceeds a certain threshold, it suggests that a new keyframe should be inserted to capture the new area of the environment. Similarly, the system evaluates the rotational change since the last keyframe. Significant rotation may warrant a new keyframe to maintain accurate orientation tracking.

   - **Scene Changes:** If the viewpoint changes significantly, which can be detected by large changes in the relative positions of the tracked points, a new keyframe might be necessary. Changes in lighting conditions can affect tracking as well. If there are substantial changes in illumination, inserting a new keyframe can help adapt to these conditions. The introduction of new objects or significant changes in the scene can lead to the insertion of a new keyframe to capture these new elements accurately.

2.5.3.2 Local Mapping

This process involves several key components:

- **Insertion of New Keyframes:**

  When a new keyframe is added to the system, it is integrated into the local map. This step is essential to keep the local map consistent and up-to-date with the latest visual information.

- **Optimization of the Local Map:**

  The core of the local mapping is local bundle adjustment as explained in 2.28, in order to refine the camera poses of keyframes and the 3D positions of the map points, we apply BA within a local window that contains only a specific set of features and keyframes.

- **Search for New Correspondences:**

  Another important task in local mapping is searching for new correspondences. The mapping thread searches for matches between unmatched features in the new keyframe and features in connected keyframes within the graph. By finding new correspondences, the system adds to the the map and improves tracking accuracy. This search process uses descriptors and geometric constraints to validate potential matches, ensuring robustness and reliability.

- **Map Point Culling:**

  To maintain the quality and manage the complexity of the map, local mapping includes the culling of low-quality map points. Map points that are not observed by multiple keyframes and that have high reprojection errors, or are located outside the view are considered for removal. This culling process helps in maintaining a high-quality map by eliminating points that could degrade the overall accuracy and consistency.

- **Redundant Keyframe Removal:**

  Finally, the local mapping process involves the removal of redundant keyframes. Keyframes that have a high degree of overlap with others and do not significantly contribute to the map's accuracy are identified and removed. This step ensures that the map remains compact and computationally efficient, facilitating real-time performance.

2.5.3.3 Loop Detection and Correction

1. **Loop Detection:**

   The system employs a place recognition module that uses bag-of-words (BoW) to quickly identify similar places. This module uses ORB features to create a visual dictionary, which allows the system to recognize revisited places even under different viewpoints or lighting conditions. When a potential loop closure is detected, the system verifies it by comparing the geometric consistency of the features.

2. **Pose Graph Optimization (PGO):**

   Once a loop is detected, a similarity transformation is applied to correct the drift. This transformation aligns the current frame with the corresponding frame in the detected loop.

   Pose graph optimization (PGO) is then performed to refine the pose estimates of all keyframes and ensure global consistency as explain previously in 2.4.4.1.

3. **Similarity Transformation:**

   When a loop closure is detected, the system computes a similarity transformation to align the current pose with the corresponding pose in the loop. This transformation compensates for the accumulated drift and corrects the trajectory. The similarity transformation

includes scaling, rotation, and translation components, ensuring that the map is globally consistent.

The transformation can be expressed as:

$$\mathbf{T}_{global} = \mathbf{S}\mathbf{T}_{local}$$

where $\mathbf{S}$ is the similarity transformation matrix, $\mathbf{T}_{local}$ is the local pose, and $\mathbf{T}_{global}$ is the globally optimized pose.

4. **Global Bundle Adjustment:**

   After applying the similarity transformation, the system performs global bundle adjustment to refine the entire map. This step ensures that all keyframes and map points are optimally aligned, further improving the accuracy and consistency of the map.

   The objective function for global bundle adjustment is similar to local bundle adjustment but applied to the entire map.

2.5.4 Testing the algorithm

In this section, we will be testing our algorithm against a benchmark dataset.

2.5.4.1 The EuRoC MAV Dataset for Drones

The EuRoC MAV Dataset is designed for evaluating visual-inertial SLAM (Simultaneous Localization and Mapping) algorithms, particularly for drones. It was collected using a MAV with stereo cameras and an IMU (Inertial Measurement Unit) in various indoor environments. The dataset is widely used in the robotics and computer vision communities due to its high-quality data and accurate ground truth trajectories.

- **Key Features of the EuRoC MAV Dataset**
  1. **Stereo Images**: Synchronized stereo images captured at 20 Hz, essential for visual SLAM.
  2. **IMU Measurements**: Inertial data recorded at 200 Hz, crucial for motion estimation.
  3. **Ground Truth Trajectories**: Accurate trajectories obtained using high-precision motion capture systems.
  4. **Diverse Environments**: Includes sequences from:
     - **Vicon Room**: Controlled environment with accurate ground truth, considered easy.
     - **Machine Hall**: Large industrial environment with challenges like featureless areas and dynamic objects, considered medium to hard.
     - **Outdoors**: Environments with significant lighting changes and dynamic elements, considered hard.

2.5.4.2 Test Results

We conducted benchmark tests on three sequences from the EuRoC MAV Dataset to evaluate the performance of our algorithm. The selected sequences represent different levels of difficulty:

- **MH_01_easy** (Machine Hall, Easy)
- **V1_02_medium** (Vicon Room, Medium)

The following figures show the comparison between the ground truth and the estimated trajectories for the selected sequences, along with the maps generated by our SLAM algorithm:

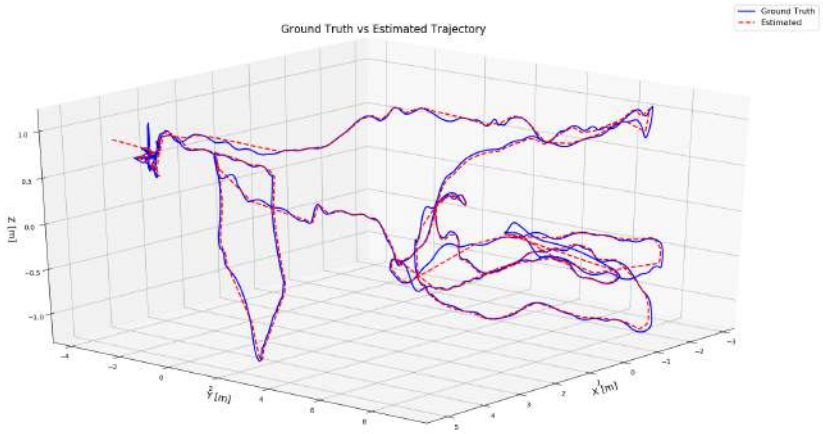


Figure 2.32: Trajectory comparison for MH_01_easy (Machine Hall, Easy).

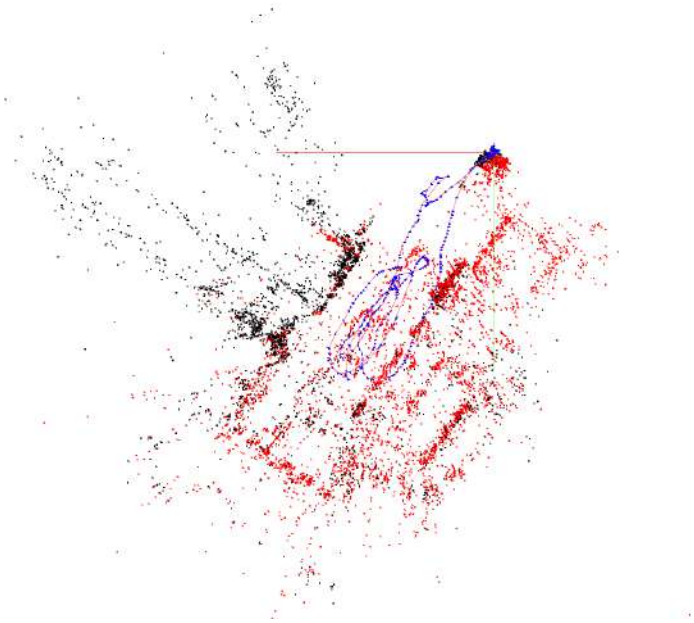
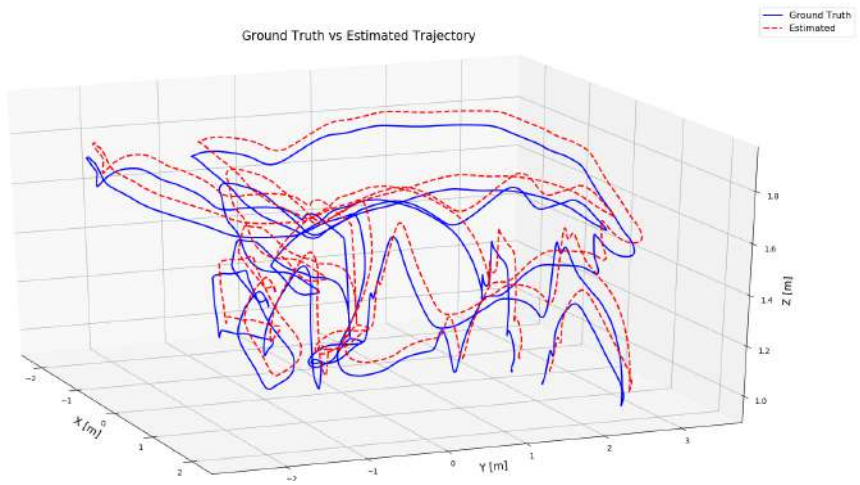


Figure 2.33: Generated map for MH_01_easy (Machine Hall, Easy).

2.5.4.3 Commentary

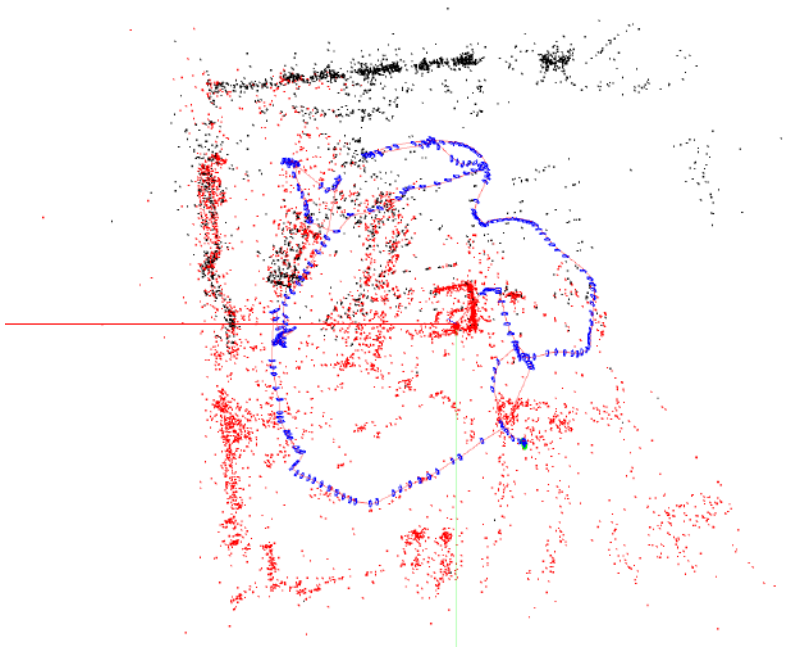
MH_01_easy: The trajectory for MH_01_easy shows a close alignment between the estimated and ground truth paths with a RMSE value of 0.1403 m, demonstrating the algorithm's effectiveness in controlled, easy environments. The generated map accurately represents the environment, further confirming the algorithm's reliability.

V1_02_medium: For V1_02_medium, the estimated trajectory maintains good accuracy with an RMSE value of 0.25, though some deviations are present, highlighting the increased complexity and challenges in medium difficulty environments. The generated map captures most features but shows slight inconsistencies in dynamic areas.



(a) Trajectory comparison for V1_02_medium (Vicon Room, Medium).

Figure 2.34: Trajectory for V1_02_medium (Vicon Room, Medium).



(a) Generated map for V1_02_medium (Vicon Room, Medium).

Figure 2.35: Map for V1_02_medium (Vicon Room, Medium).

2.5.4.4 Commentary

MH_01_easy: The trajectory for MH_01_easy shows a close alignment between the estimated and ground truth paths with a RMSE value of 0.1403 m, demonstrating the algorithm's effectiveness in controlled, easy environments. The generated map accurately represents the environment, further confirming the algorithm's reliability.

V1_02_medium: For V1_02_medium, the estimated trajectory maintains good accuracy with an RMSE value of 0.25, though some deviations are present, highlighting the increased complexity and challenges in medium difficulty environments. The generated map captures most features but shows slight inconsistencies in dynamic areas.

2.6 Conclusion

In this chapter, we have introduced and discussed the implementation of Visual SLAM for UAVs using the ORB-SLAM algorithm. We covered the theoretical foundations of Visual SLAM, detailed the workings of the ORB-SLAM algorithm, and demonstrated its practical implementation and testing using the EuRoC MAV Dataset.

Our testing results show that the ORB-SLAM algorithm performs well in controlled environments (easy) but faces challenges in more dynamic and complex scenarios (medium and hard). The generated maps, while generally accurate, also reflect these difficulties, indicating areas for further improvement.

Chapter 3

Implementation

3.1 Introduction

In this chapter, we explore into the practical aspects of building and implementing a quadrotor UAV equipped with a robust control system and SLAM capabilities. We begin with a detailed exploration of the component selection process, discussing the criteria for choosing each part and analyzing the associated costs. This comprehensive guide ensures that the selected components align with the performance requirements and budget constraints.

Following the component selection, we move on to the implementation of the control system. This involves identifying key parameters such as mass, arm lengths, and moments of inertia, which are crucial for the accurate modeling and control of the UAV. We also discuss the implementation of the Extended Kalman Filter (EKF) for sensor fusion and state estimation, highlighting its importance in achieving precise control and navigation.

The chapter also covers the use of Software-in-the-Loop (SITL) simulations to test and validate the control algorithms in a virtual environment before deploying them to the real quadrotor. This approach helps in refining the control strategies and ensures their reliability.

Finally, we present the implementation of the SLAM algorithm using ROS and Docker, providing a detailed overview of the frameworks and tools used. We conduct various experiments to test the SLAM performance in different scenarios, analyzing the results to understand the algorithm's strengths and limitations

3.2 Component Selection and Cost Analysis

In this section, we will explore the various components required for building a drone, discuss the criteria for selecting each component, and analyze the associated costs. The objective is to provide a comprehensive guide to understanding the importance of each component and how to make informed decisions to ensure optimal drone performance while adhering to budget constraints.

3.2.1 Component Overview

The following components are essential for building a functional and efficient drone. Each component has been selected based on specific criteria to ensure the best performance and value.

3.2.1.1 Drone Frame

The drone frame is the structural backbone of a drone, serving as the skeleton on which all other components are mounted and providing protection for the electronics. A typical drone frame is constructed from carbon fiber plates and metal hardware as shown in figure 3.1, including:

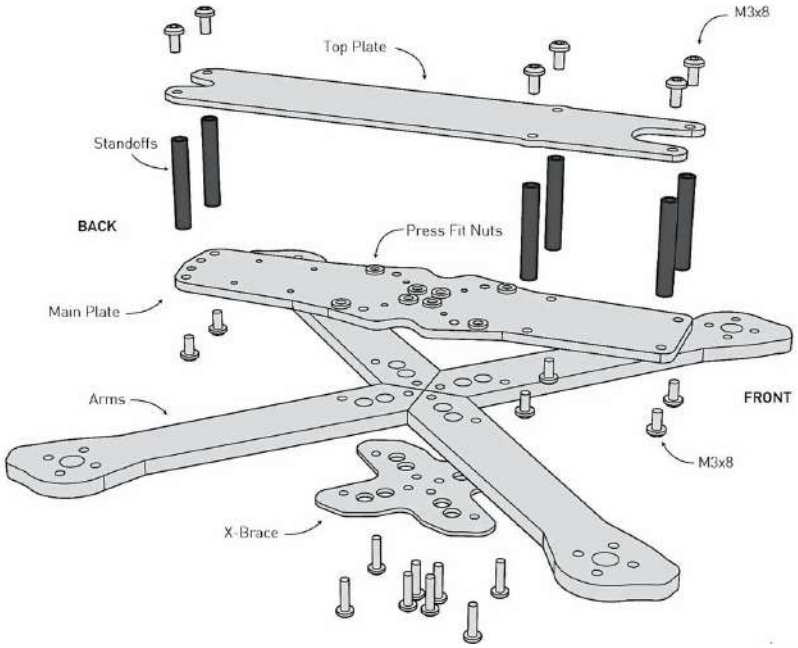


Figure 3.1: Typical drone frame [10]

- 4x Arms
- Top and bottom plates
- Camera mount
- Standoff
- Bolts

While most frames share a similar shape and construction, the choice of drone frame can significantly impact flight performance. Factors such as weight, aerodynamics, resonance frequency, and rigidity influence how well the drone flies. Frames not specifically designed for racing are often considered freestyle frames, which are versatile and suitable for various activities like freestyle flying, capturing cinematic shots, or casual cruising. In contrast, race frames prioritize performance, featuring lightweight and low-drag designs at the expense of durability and ease of build.

Freestyle frames are generally more durable and versatile, making them a better choice for non-racing applications. The ideal drone frame balances strength, practicality, and lightweight

construction. However, achieving this balance often requires trade-offs between protection, practicality, and weight.

Modern drone frames typically adopt a minimalist design with skinnier arms and thicker carbon fiber to save weight and reduce frame resonance.

Carbon fiber is the preferred material for drone frames due to its:

- Low cost
- Light weight
- Durability
- Rigidity
- High customizability

Despite its advantages, carbon fiber has downsides, including electrical conductivity, which can cause short circuits if live wires touch the frame, and the ability to block or attenuate radio frequencies, necessitating external antenna mounts for optimal signal strength.

QAV250 Frame: We chose the QAV250 Frame for our application. It's a lightweight and durable frame designed for small quadcopters. Which makes it ideal for agile and stable flight. THe specification of the frame can be found in 3.1 and its cost is 4,000 DA.

Specification	**Details**
Model	QAV250
Material	Carbon Fiber
Wheelbase (mm)	250
Height (mm)	80
Weight (gm)	160
Arm Size (L x W) mm	114 x 25
Motor Mounting Hole Dia. (mm)	3

Table 3.1: Specifications of the QAV250 Drone Frame

.

3.2.1.2 Propellers

Propellers, or props, are vital for an drone, generating the thrust needed for flight and maneuverability. Choosing the right propellers is crucial for optimal performance, as incorrect choices can lead to noise, reduced flight time, or motor failure.

- **Propeller Description Formats**

  Propeller sizes are given in inches ($1 = 2.54cm$), and are described in two formats:

  - **L x P x B**
  - **LLPP x B**

Figure 3.2: The chosen QAV250 frame

Figure 3.3: Different propeller sizes

.

Where:

- **L**: Length
- **P**: Pitch
- **B**: Number of blades

- **Factors Affecting Propeller Performance**
  - **Length**: The diameter of the disc created when the prop spins. Longer props generate more thrust but require more power and do not necessarily mean faster flight due to pitch importance.
  - **Pitch**: Distance a prop travels per revolution, measured in inches. Higher pitch means more thrust at high speeds but less at low speeds and more turbulence.
  - **Blades**: Increasing blade count increases surface area and thrust, but decreases efficiency and adds strain on the motor.
- **Propeller Types**
  - **Two-Blade**: More efficient, less drag, better for long-range flying.

- **Three-Blade**: Balanced between efficiency and power, popular for racing and freestyle.
- **Four-Blade/Hex-Blade**: Used for specific scenarios like indoor tracks, but generally less efficient.

- **Propeller Weight** Lighter props perform better, requiring less motor torque, leading to better responsiveness and compatibility with various motors.

Propeller 2 Blades 5045 : The 2 Blades 5045 CW CCW propellers are chosen for their efficient thrust and stability. These propellers are compatible with the drone's motors and frame. The cost of four pairs of propellers is 1,840 DA.

3.2.1.3 Motors

Choosing the right motors for an drone is crucial for achieving the desired performance, these are some of the criterias to look at when it comes to choosing a motor:

- **Motor Specifications**: The KV rating of a motor indicates its RPM per volt without load. Higher KV motors spin faster but produce less torque, making them suitable for lightweight setups, while lower KV motors generate more torque, ideal for carrying heavier loads. We also have the Stator size, denoted as "XXYY" (e.g., 2205), where "XX" is the stator diameter and "YY" is the stator height, also affects motor performance. Larger stators generally provide more torque and power.
- **Factors Affecting Motor Performance**: Torque is a critical factor, with higher torque motors capable of spinning larger props or more blades, thus providing more thrust. Efficiency is also important, as efficient motors convert more electrical power into thrust, improving flight time and performance.

Specification	**Details**
Framework	12N14P
KV Rating	2280Kv
Length	26.7 mm
Diameter	23 mm
No. of Cells	2-3S
Max Thrust	460g
Shaft Diameter	2 mm
Propeller Size	5"- 6"
Weight	18 g

Table 3.2: Motor Specifications

EMAX MT1806 Brushless 2280KV Motor: We opted for the EMAX MT1806 Brushless 2280KV motors. These motors are compatible with the propellers and ESCs, providing optimal thrust and control, more details are in 3.2.1.3.The cost of four motors is 11,200 DA. .

Figure 3.4: EMAX MT1806 Brushless

3.2.1.4 Battery

Batteries are chosen based on the following criterias :

- **Capacity**: Higher capacity provides longer flight times.
- **Voltage**: Appropriate voltage to match the drone's power requirements.
- **Discharge Rate (C rating)**: Determines how quickly the battery can deliver current. A higher discharge rate supports high-performance motors and electronics.
- **Weight**: Lightweight batteries improve flight efficiency and maneuverability.
- **Form Factor**: Compatibility with the drone's frame and available space.
- **Reliability**: Proven performance and durability in various conditions.

We chose the Li-po Battery 4200mAh 3s 35C 11.1V for its high energy density and discharge rate, which are critical for providing sustained power to the drone. This battery offers a good balance between capacity, weight, and performance, ensuring longer flight times. The cost of the battery is 12,200 DA.

3.2.1.5 Electronic Speed Controllers (ESC)

We chose four of the ESC 12A SimonK with a cost of 10,000 DA. This choice was based upon the following criterias

- **Current Rating**: Must handle the maximum current draw of the motors without overheating. The ESC can sustain a continuous load of 11A, with current peaks up to 16A for 1 second.
- **Voltage Compatibility**: Should match the voltage of the battery and motors.
- **Firmware**: SimonK firmware is preferred for its responsiveness and reliability in high-performance applications.
- **Size and Weight**: Compact and lightweight ESCs contribute to the overall efficiency and agility of the drone. Dimensions are 5cm x 5cm x 0.2cm.
- **Thermal Management**: Good heat dissipation to prevent overheating during prolonged use.

- **Reliability**: Proven durability and performance under various conditions.

Figure 3.5: 12A ESC SimonK

3.2.1.6 Power Distribution Board

The power distribution board (PDB) is a crucial component for managing and distributing power within the drone. It provides voltage to the motors through the ESCs and supplies power to the flight controller. This ensures that all components receive the necessary power to function correctly, maintaining the overall efficiency and stability of the drone. The cost of the power distribution board is 2,000 DA.

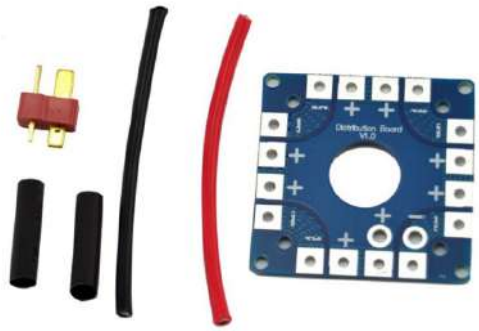


Figure 3.6: 12A ESC SimonK

3.2.1.7 Flight Controller

We chose a Pixhawk 2.4.8 PX4pix Flight Controller FULL Kit. We've talked about the flight controller extensively in the second chapter so we won't be repeating it here. The cost of the flight controller is 64,000 DA.

3.2.1.8 Remote Control

A remote control (RC) system is essential for manually controlling your vehicle from a transmitter. An RC system includes a ground-based remote control operated by the user to command the vehicle. The remote control has physical controls for specifying the vehicle's movement

(e.g., speed, direction, throttle, yaw, pitch, roll) and activating autopilot flight modes (e.g., takeoff, landing, return to home, mission).

Figure 3.7: FlySky FS-I6X 2.4 GHz AFHDS 2A RC Transmitter with fs-iA6B

For telemetry-compatible RC systems, the remote can also receive and display information from the vehicle (e.g., battery level, flight mode). The remote contains a radio module linked to a compatible module on the drone, which is connected to the flight controller. The flight controller interprets commands based on the current autopilot flight mode and vehicle status, then drives the motors and actuators accordingly. The complete system consists of two parts:

- **Transmitter (TX)**: The remote control used to command the drone.
- **Receiver (RX)**: The part that receives commands from the transmitter via radio waves, attached to the drone and connected to the flight controller.

We chose a FlySky FS-I6X 2.4 GHz AFHDS 2A RC Transmitter with fs-iA6B beacuse it was an affordable and user-friendly option that provided precise control over the drone. The cost of the RC transmitter is 32,000 DA.

3.2.1.9 Raspberry Pi 5 Model B

The Raspberry Pi 5 Model B is a powerful and versatile development kit, featuring an 8GB RAM and an efficient cooling system. It provides robust processing capabilities for various applications, including handling the complex computations required for visual slam. The cost of the Raspberry Pi 5 Model B Development Kit is 45,000 DA. It's equipped with the following :

- **Processor**: Equipped with a powerful quad-core ARM Cortex-A76 CPU, ensuring fast and efficient processing.
- **Memory**: 8GB of LPDDR4-3200 SDRAM, offering ample memory for multitasking and running complex applications.
- **Connectivity**: Multiple connectivity options, including USB 3.0, Gigabit Ethernet, and dual-band Wi-Fi.

Figure 3.8: Raspberry Pi 5 Model B full kit

- **Graphics**: Improved GPU for handling video and graphical tasks.
- **Expansion**: GPIO pins for connecting various sensors and peripherals, making it highly versatile for custom applications.
- **Cooling**: Comes with a fan cooler to maintain optimal temperatures during high-performance tasks.

3.2.1.10 Camera

We worked with the AUKEY Webcam Full HD which costs about 3,900 DA. The criterias for choosing a camera should be :

- **Resolution**: High-resolution cameras provide detailed images necessary for accurate SLAM processing but badly affect the computational time.
- **Frame Rate**: A higher frame rate ensures smoother video, which helps in better tracking and mapping.
- **Field of View**: A wider field of view allows the camera to capture more area, improving the SLAM algorithm's effectiveness.
- **Compatibility**: The camera must be compatible with the processing unit (e.g., Raspberry Pi) and SLAM software.
- **Latency**: Low-latency cameras provide real-time data, essential for SLAM applications.
- **Size and Weight**: Compact and lightweight cameras are preferable to avoid adding unnecessary weight to the drone.
- **Durability**: The camera should be durable and reliable under various environmental conditions.

Figure 3.9: AUKEY Webcam 1080p

3.2.1.11 Voltage Regulator

We chose the LM2596 Adjustable DC-DC Voltage regulator to provide 5 volts the raspberry pi 5. The cost of the voltage regulator is 700 DA.

Figure 3.10: LM2596 Adjustable DC-DC Voltage regulator

3.2.2 Total Cost

Component	Quantity	Cost (DA)
Li-po Battery 4200mAh 3s 35C 11.1V + Connector	1	12,200
Raspberry Pi 5 Model B (8 GB) Fan cooler	1	45,000
Pixhawk 2.4.8 PX4pix Flight Controller FULL Kit	1	64,000
FlySky FS-I6X 2.4 GHz AFHDS 2A RC Transmitter with fs-iA6B	1	32,000
Propeller 3 Blades 5045 CW CCW (Pair)	4	1,840
LM2596 Adjustable DC-DC Voltage Regulator	1	700
QAV250 Frame	1	4,000

ESC 12A SimonK	4	10,000
EMAX MT1806 Brushless 2280KV Motor	4	11,200
12V Power Distribution Board	1	2,000
AUKEY Webcam 1080p Full HD	1	3,900
Total Cost		**186,840**

Table 3.3: Cost Breakdown of Drone Components

The total cost for all the components listed above is 184,940 DA.

3.3 Control Implementation

3.3.1 Parameter Identification

To implement the control and observation algorithms on the quadrotor, we need to know the different parameters that govern its dynamics.

The parameters we need to identify are:

- The mass of the quadrotor
- The Arm lengths l_x and l_y
- The moment of inertia of the quadrotor
- The lift and drag coefficients

3.3.1.1 Mass

Using a digital scale, we weighed the quadrotor and obtained the following result:

$$m = 506g \tag{3.1}$$

3.3.1.2 Lengths l_x and l_y

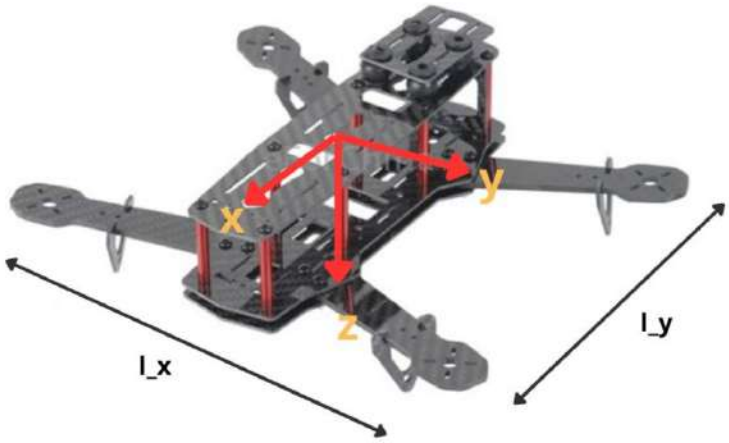


Figure 3.11: Quadrotor l_x and l_y

Using a ruler, we measured the two lengths l_x and l_y and obtained the following measurements:

$$l_x = 20.1cm \tag{3.2}$$

$$l_y = 15.6cm \tag{3.3}$$

3.3.1.3 Moment of Inertia

The moment of inertia of a solid describes the dynamic behavior of a body in rotation around a defined axis. It plays the same role in rotational dynamics as mass does in translational dynamics. The moment of inertia depends on the distribution of the solid's mass relative to a rotation axis. For a solid that undergoes rotational motion in space, the moment of inertia can be described by a symmetric matrix of dimensions 3x3.

$$J = \begin{bmatrix} J_{XX} & J_{XY} & J_{XZ} \\ J_{YX} & J_{YY} & J_{YZ} \\ J_{ZX} & J_{ZY} & J_{ZZ} \end{bmatrix} \tag{3.4}$$

For the same solid, different rotation axes can have different moments of inertia, and an infinite number can be found.

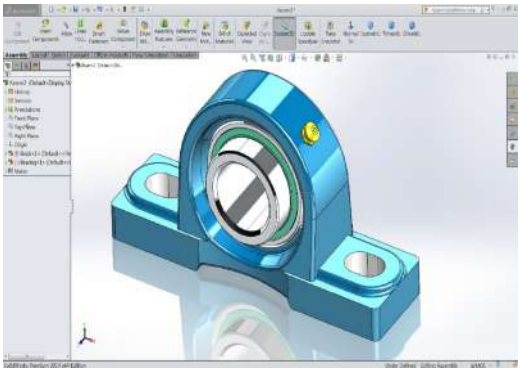
Generally, the inertia matrix is calculated using mathematical methods based on its expression, but for solids with complex geometric shapes, it is preferable to use software that calculates it directly or to proceed with an identification process. In our case, due to the lack of material we decide to work with Computer-Aided Design (CAD), we chose a software that can simulate our frame in a 3D environment with its carbon fiber construction. A good software choice is SolidWorks

3.3.1.4 Solidworks

SolidWorks is a robust CAD software that allows for 3D modeling and simulation. Developed by Dassault Systèmes, it is widely used in engineering and design fields for creating precise,

detailed models of mechanical parts and assemblies. SolidWorks offers comprehensive tools for designing, simulating, and documenting various types of products, making it an excellent choice for projects requiring detailed 3D representations and analysis. In our case, SolidWorks offers

(a) Design a motorcycle using SolidWorks



(b) Design a complex piece using SolidWorks

Figure 3.12: Examples of projects using SolidWorks

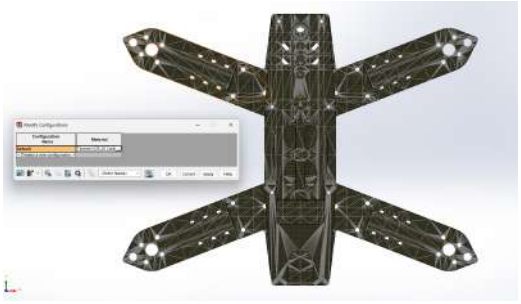
various types of shapes and geometries. However, the challenging part was the complex shape of our drone. We had to design each part individually to finally assemble the final version.

(a) Top Plate of the body Frame

(b) Camera support 1

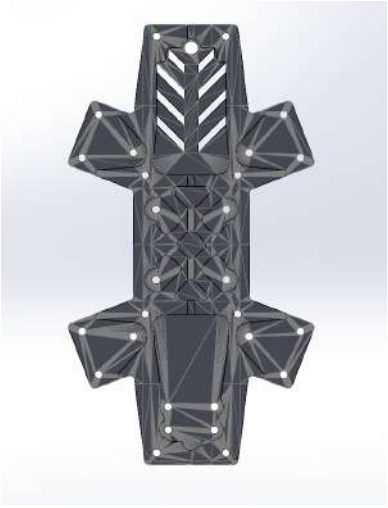
The next step is to add the carbon fiber material to our drone and observe the resulting moment of inertia as shown in figure.

(a) original Frame



(b) Frame made with fiber carbon

Figure 3.15: Applying the fiber carbon to the frame



(a) Bottom plate of the Frame

(b) camera support2

Figure 3.14: Different Views and Components of the Drone Frame

And then, by selecting the material properties in SolidWorks, we obtained the following results 3.5.

Property	**Value**
Density	1880.0000 kilograms per cubic meter
Mass	0.2179 kilograms
Volume	0.0001 cubic meters
Surface area	0.0688 square meters
Center of mass (meters)	
X	0.0001
Y	0.0057
Z	0.0177

Table 3.4: Basic properties of the frame

Property	Value
Principal axes of inertia and principal moments of inertia: Taken at the center of mass	(kilograms * square meters)
Ix	(1.0000, -0.0025, 0.0001) Px = 0.0007
Iy	(0.0025, 1.0000, -0.0050) Py = 0.0008
Iz	(-0.0001, 0.0050, 1.0000) Pz = 0.0014
Moments of inertia:	(kilograms * square meters)
Ixx	0.0008
Ixy	0.0000
Ixz	0.0000
Iyx	0.0000
Iyy	0.0008
Iyz	0.0000
Izx	0.0000
Izy	0.0000
Izz	0.0014

Table 3.5: Inertia properties of the frame

and then we get the final matrix of moments of inertia:

$$J = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} 0.0008 & 0.0000 & 0.0000 \\ 0.0000 & 0.0008 & 0.0000 \\ 0.0000 & 0.0000 & 0.0014 \end{bmatrix} \tag{3.5}$$

3.3.1.5 Thrust Identification Method

To identify the thrust factor, a method involving the measurement of the force produced by the rotors was utilized. This process requires setting up the quadrotor in a way that allows for the precise measurement of the thrust force. The steps involved in this method are as follows:

1. The quadrotor is placed on a balance that has been tared, meaning that the balance reads zero when there is no weight on it.
2. The motors are run at the same speed, and for each speed setting, the value displayed by the balance (which represents the thrust force) and the pulse width modulation (PWM) signal in microseconds (μs) used to control the motors are recorded.
3. This procedure is repeated for various motor speeds to gather a comprehensive dataset.
4. The results obtained are summarized in a table and a figure, which show the relationship between the thrust force and the PWM signal. The thrust force (T) is modeled as a first-order polynomial function of the PWM signal (u):

$$T = a \cdot u + b \tag{3.6}$$

where a and b are coefficients determined through the identification process.

Due to material limitations, we used data from a previous final project [**?**], which had already conducted similar work on our quadrotor. This allowed us to leverage their results and proceed with our work confidently. The table below summarizes the coefficients obtained from the first-order approximation of the thrust relation with the PWM signal.

Coefficient	Value
a	0.0017
b	-0.9335

Table 3.6: Coefficients of the thrust approximation

3.3.1.6 Drag Moment Identification Method

To identify the drag moment factor, a method involving the measurement of the torque generated by the rotors was utilized. This process requires setting up the motor in a way that allows for the precise measurement of the drag moment. The steps involved in this method are as follows:

1. The motor is mounted vertically on an axis that allows it to rotate about a fixed point. This setup can be seen in Figure 3.16.

2. A balance is placed vertically and tared, meaning that the balance reads zero when there is no weight on it.

3. The motor is run in the clockwise direction, causing the axis to rotate in the opposite direction. The torque generated by the motor is then applied as a force on the balance.

4. The value displayed by the balance and the PWM signal in microseconds (μs) used to control the motor are recorded.

5. This procedure is repeated for various motor speeds to gather a comprehensive dataset.

6. The results obtained are summarized in a table and a figure, which show the relationship between the drag moment and the PWM signal. The drag moment (τ) is modeled as a first-order polynomial function of the PWM signal (u):

$$\tau = c \cdot u + d \tag{3.7}$$

where c and d are coefficients determined through the identification process.

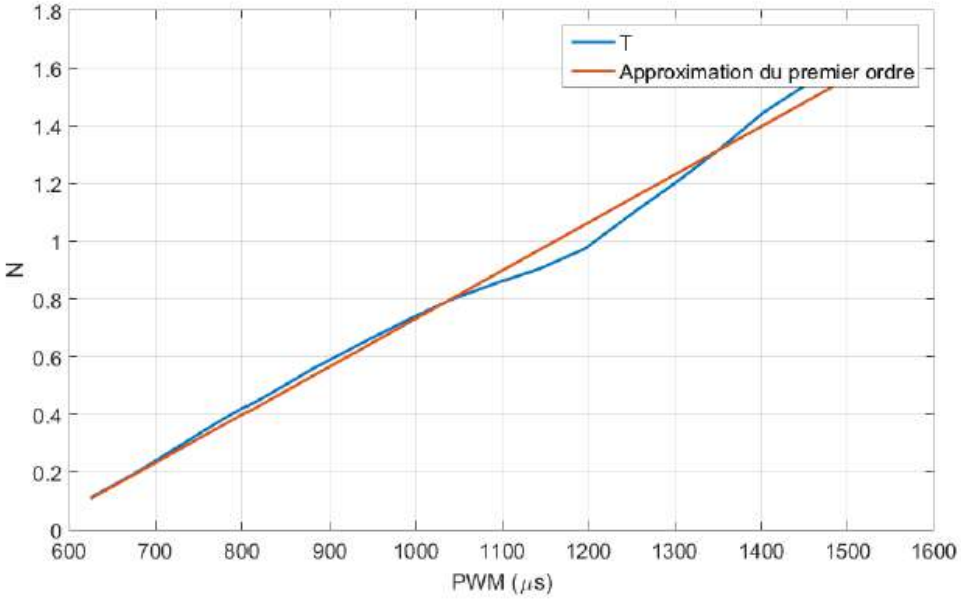


Figure 3.16: Setup used for drag moment identification

As previously mentioned, we will use the results from the previous final project [**?**], which performed the same work on our quadrotor. Their results are reliable. The table below summarizes the coefficients obtained from the first-order approximation of the drag moment relation with the PWM signal.

Coefficient	Value
c	1.2362e-8
d	3.0737e-5

Table 3.7: Coefficients of the drag moment approximation

3.3.2 Extended Kalman Filter (EKF)

The Extended Kalman Filter (EKF) is essential in modern control systems, particularly in robotics and unmanned vehicles, due to its ability to provide accurate state estimates from noisy sensor measurements. This accuracy is crucial for stability and performance, enabling precise control and navigation.

In essence, the EKF helps to:

- **Fuse Sensor Data:** Combine multiple sensor inputs to provide a comprehensive state estimate.
- **Filter Noise:** Mitigate the effects of sensor noise and errors.
- **Predict Future States:** Estimate the vehicle's future state based on current data and system dynamics.

3.3.2.1 EKF's principle

The EKF processes sensor measurements using an extended version of the Kalman Filter algorithm, which is designed for nonlinear systems. Here are the key components and steps in the EKF process:

1. **State Vector:** The EKF maintains a state vector that includes variables like position, velocity, orientation (quaternions), and biases.
2. **Prediction Step:** Using the system's dynamics model, the EKF predicts the next state and the associated uncertainty.
3. **Update Step:** The EKF updates its state estimates using new sensor measurements, taking into account their uncertainties.
4. **Covariance Matrix:** This matrix represents the uncertainties in the state estimates and is updated in each step to reflect the latest data.

The EKF runs on a delayed 'fusion time horizon' to accommodate different sensor delays relative to the IMU (Inertial Measurement Unit). Data from each sensor is buffered and used at the appropriate time, with delay compensation controlled by specific parameters.

A complementary filter propagates the states forward from the 'fusion time horizon' to the current time using buffered IMU data. The time constants for this filter are controlled by specific parameters to minimize errors.

3.3.2.2 Sensor Measurements Utilized by the EKF

The EKF can operate in different modes depending on the available sensor data:

- **IMU:** Provides three-axis body-fixed inertial measurements.
- **Magnetometer:** Offers three-axis magnetic field data or external vision system pose data.
- **Height Data:** Sources include GPS, barometric pressure sensors, range finders, or external vision systems.
- **GPS:** Supplies position and velocity measurements.
- **Optical Flow:** Provides velocity estimates from optical flow sensors.
- **External Vision Systems:** Offers position, velocity, or orientation measurements.

3.3.3 Simulation Results

Here are some simulation results that demonstrate the EKF's performance in estimating various states:

These plots show the estimated state versus the setpoint and actual measurements. The EKF effectively filters noise and provides accurate state estimates, which is critical for maintaining stability and precise control of the vehicle. The consistency between the estimated states and the setpoints demonstrates the EKF's capability to handle the dynamic conditions of the system.

Figure 3.17: Roll Angular Rate

Figure 3.18: Pitch Angle

Figure 3.19: Roll Angle

Figure 3.20: Yaw Angular Rate

3.3.4 Software-in-the-Loop (SITL) for Simulation of the Control Algorithm

Software-in-the-Loop (SITL) in PX4 is a simulation technique used to test flight control software without the need for physical hardware. It runs the full PX4 flight stack in a simulated environment, allowing for comprehensive testing and debugging before deploying to real hardware.

3.3.4.1 Benefits of SITL

- **Cost-effective**: No need for physical components which reduces costs.
- **Safe Testing Environment**: Allows for testing edge cases and failure modes without risk to physical hardware.
- **Development Speed**: Faster iterations as software changes can be tested immediately.

3.3.4.2 How SITL Works

SITL simulates the PX4 firmware on a host computer, typically using tools like jMAVSim or Gazebo for the simulated environment. The simulation runs in real-time, interacting with the same software that would run on the actual drone hardware.

3.3.4.3 Implementation of Custom Controller

In our project, we generate the C++ code from Simulink that uses the PID controller and then use QGroundControl to implement our custom controller. Next, we use Gazebo to simulate the environment.

3.3.4.4 QGroundControl

QGroundControl is an intuitive and powerful ground control station (GCS) application for flight control systems. It provides full flight control and mission planning functionality for autonomous vehicles. Key features include:

- **Cross-platform support**: Available on Windows, macOS, Linux, iOS, and Android.
- **Mission Planning**: Easy-to-use mission planning tools for complex waypoint and survey missions.
- **Real-time Data Monitoring**: Displays real-time telemetry data from the vehicle, including GPS position, battery status, and sensor readings.
- **Firmware Management**: Facilitates firmware upload and configuration for supported flight controllers.

QGroundControl is essential for configuring, monitoring, and controlling unmanned vehicles during development and testing.

3.3.4.5 Gazebo

Gazebo is a powerful robotics simulator that offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It provides the following features:

- **High-fidelity Physics**: Simulates the dynamics of robots and environments with high accuracy, including rigid body physics, contact, and sensor data.
- **Sensor Simulation**: Includes support for a variety of sensors, such as cameras, LiDAR, IMUs, and GPS.
- **Extensible and Flexible**: Integrates with various robotics frameworks, including ROS (Robot Operating System).
- **3D Visualization**: Offers 3D visualization tools for designing and debugging complex robotic systems.

Using Gazebo, developers can test and refine their robotic algorithms in a controlled, virtual environment before deploying them to real hardware, thus reducing risk and accelerating development.

3.3.4.6 Simulation

In this part, we will simulate the C++ code in Gazebo to evaluate the performance.

- First of all, we launch the C++ firmware code, then we choose a model of quadrotor to try the command, in our case we chose a quadrotor 5 inch. 3.21

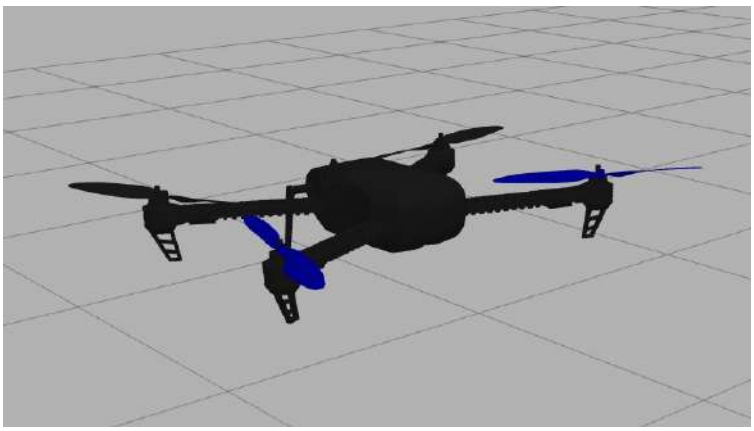


Figure 3.21: Quadrotor in Gazebo

- Then we open QGroundControl and the quadrotor will be automatically connected to it.3.22

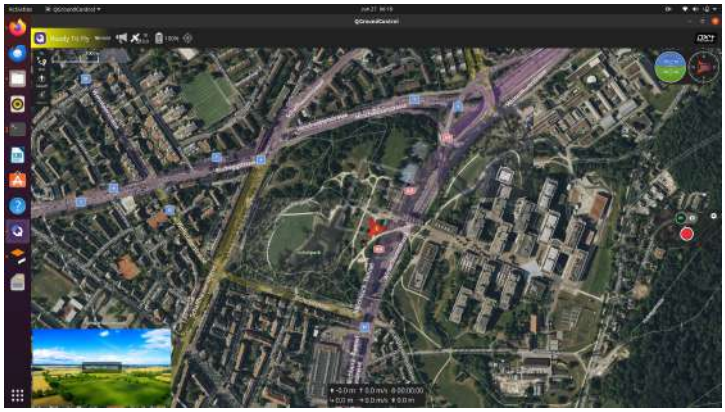


Figure 3.22: QGroundControl connected to the quadrotor

- We will define a path that our drone needs to follow, for example in Figure 3.23:

Figure 3.23: Path that we want our drone to follow

- Evaluating the results:
  In the map, the drone perfectly follows the path 3.24.

(a) (b) (c) (d)

Figure 3.24: The Quadrotor following the given path

3.3.5 Implementation in real quadrotor

To implement the quadrotor, first, we need to connect the flight controller to the computer using a USB cable. In QGroundControl, we install the custom firmware (the C++ code generated from Simulink) onto the Pixhawk. After completing the necessary calibrations, we can start the real-world simulation.

Figure 3.25: Quadrotor in flight

3.3.5.1 Results

We need to consider that we are using a remote controller, which means we provide the flight controller with setpoints for roll, pitch, and yaw. The flight controller, using our PID regulator, will attempt to track these commands. Here are the results of our flight:

Figure 3.26: Yaw Angle

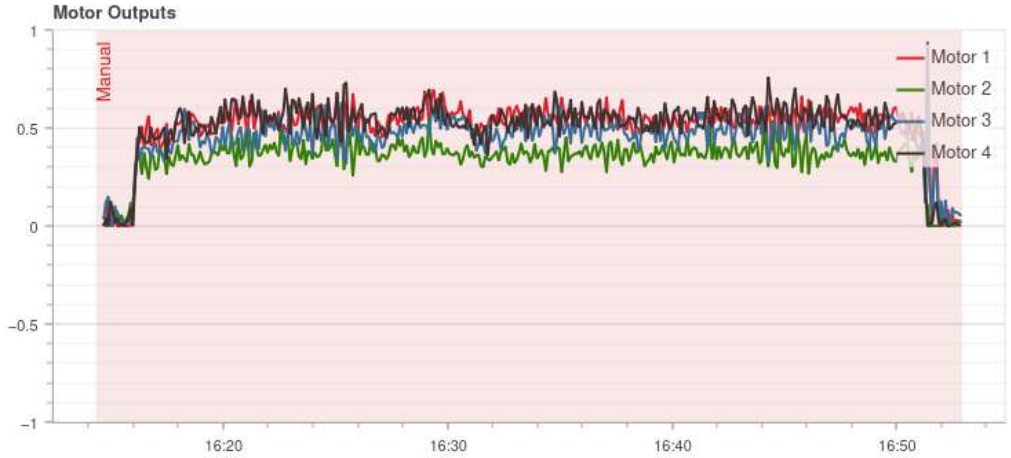


Figure 3.27: Motor Outputs

(a) Pitch Angle

(b) Roll Angle

Figure 3.28: Flight Performance Results: Pitch Angle, and Roll Angle

The results of our flight can be analyzed as follows:

- Yaw Angle (Figure 3.26): The yaw estimated angle closely follows the yaw setpoint, indicating that the yaw control loop is functioning well. The slight deviations are expected and within acceptable limits for manual control.

- Motor Outputs (Figure 3.27): The motor outputs show the individual thrust commands sent to each motor. The variability in motor outputs reflects the dynamic adjustments made by the PID controller to maintain stability and follow the given setpoints.

- Pitch Angle (Figure 3.28a): The pitch estimated angle tracks the pitch setpoint with some oscillations. These oscillations may indicate the need for further tuning of the PID parameters to improve stability.

- Roll Angle (Figure 3.28b): Similar to the pitch angle, the roll estimated angle follows the roll setpoint with noticeable oscillations. This suggests that while the controller is able to track the setpoint, there is room for optimization in the roll control loop.

Overall, the quadrotor demonstrates a good level of control, with the estimated angles closely following the setpoints. The motor outputs indicate that the PID controller is actively adjusting the thrust to maintain stability.

3.4 SLAM Implementation

Before delving into the actual implementation, we're going to present briefly the frameworks and softwares that we used.

3.4.1 Robot Operating System

Robot Operating System, abbreviated as **ROS**, is an open-source operating system designed for the development, programming, and simulation of robots. It facilitates these tasks through a collection of libraries, applications, and graphical tools. Additionally, ROS enables seamless communication among multiple robots. It offers essential functionalities such as hardware abstraction, device drivers, message passing, package management, and more, contributing to a comprehensive and efficient environment for robotic system development.

In addition, ROS is the most used operating system for robot programming, offering several tools facilitating robotics development, such as:

- Integration with other operating systems of particular robots.
- Human-machine interface and communication between machines.
- Test and simulation-based system.
- Implementation of multiple programming languages like C++ and Python.
- Open-source framework with the availability of extensive packages online.

3.4.1.1 ROS architecture

ROS architecture relies on a distributed communication system, enabling various nodes to interact through topics. These topics serve as communication channels for the exchange of messages between nodes.

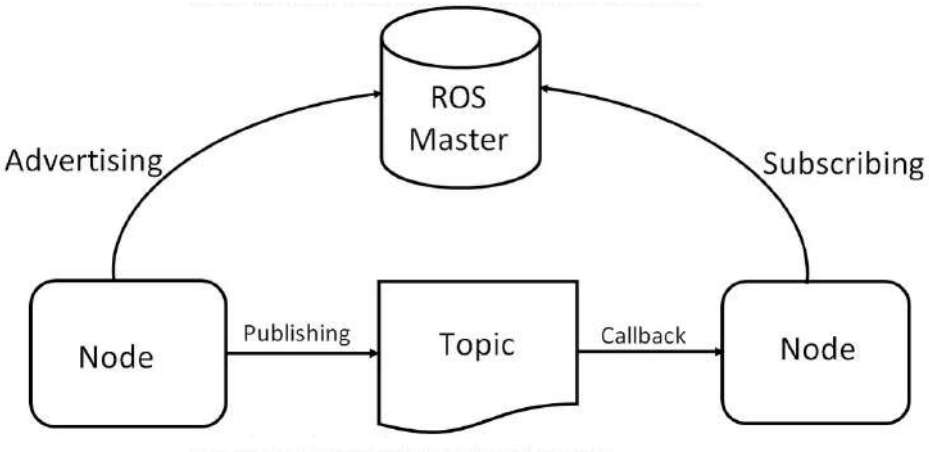


Figure 3.29: ROS Master Communication

Nodes are executable processes designed to perform specific tasks, such as perception, planning, or control. They communicate by publishing or subscribing to messages via topics and can be implemented in various programming languages like Python or C++. Additionally, you can employ the following commands for managing nodes:

```
$ rosrun          # Launch a ROS node from a package
$ rosnode         # List active ROS nodes
```

3.4.1.2 The components of a ROS package

In the world of ROS (Robot Operating System), a package stands as the fundamental building block of ROS. Within each package, one can house an array of elements, including libraries, datasets, configuration files, and more.

The `package.xml` file, referred to as a manifest, plays a crucial role by offering a comprehensive description of the package. This file encompasses essential details such as dependencies, metadata (version, maintainer, license), and more.

Beyond individual packages, ROS introduces the concept of stacks—a compilation of packages that collectively form a library. A stack manifest, similar to a package manifest, encapsulates the pertinent information related to a stack.

A package manifests itself as a directory containing a `package.xml` file. Conversely, a stack materializes as a folder housing a stack.xml file. This clear organizational structure facilitates efficient navigation and management of ROS code.

3.4.1.3 ROS workspace

A ROS workspace is a directory where ROS packages are built, modified, and organized. The workspace allows for the separation of various projects and their dependencies, making it easier to manage and develop complex robotic systems. Creating and setting up a ROS workspace involves a few straightforward steps:

1. Create a directory for the workspace and a subdirectory for source files:

   ```
   $ mkdir -p ~/workspace_name/src
   $ cd ~/workspace_name/
   ```

2. Initialize the workspace using ‘catkin make‘, a command that configures the workspace and sets up necessary build files:

   ```
   $ catkin_make
   ```

3. Source the workspace’s setup file to configure the environment. This command needs to be run every time a new terminal session is started to ensure the workspace environment is correctly set up:

   ```
   $ source devel/setup.bash
   ```

The first command creates a directory structure with ‘src‘ as the source directory where all your packages will reside. The second command runs ‘catkin make‘, which is a build tool for catkin-based workspaces that sets up the necessary build environment. Finally, sourcing the setup file ensures that the ROS environment variables are properly configured for the workspace, making all packages within the workspace available to ROS.

A typical ROS workspace might look like this:

```
workspace name/
 build
 devel
 src
 logs
```

- The ‘src‘ directory contains the source code for ROS packages. - The ‘build‘ directory holds intermediate files produced during the build process. - The ‘devel‘ directory contains the development environment setup files. - The ‘logs‘ directory keeps logs generated during builds.

3.4.1.4 ROS packages

To perform this operation in the terminal, we follow these steps:

```
$ cd ws_name
$ cd src
$ catkin_create_pkg pkg_name rospy roscpp std_msgs  # dependencies
$ cd ws_name
$ catkin_make
```

For a package to qualify as a catkin package and thus be usable by ROS, it must adhere to specific criteria:

- A `package.xml` file compliant with catkin, providing meta-information about the package, must be created.
- A `CMakeLists.txt` file that invokes catkin, must be created as well.
- Each package must reside in its directory; there cannot be more than one package in each folder.

3.4.1.5 GUI Tools

Apart from command-line utilities, ROS offers graphical user interface (GUI) tools to assist in development and visualization.

- **RViz**
  stands as a 3D visualization tool within the ROS framework, enabling users to visualize sensor data, robot models, and additional information in a 3D environment. It furnishes an interactive interface facilitating the observation and interaction with the robot's perception and planning capabilities.
- **RQT**
  is a graphical user interface framework for ROS, organized around plugins. It offers a set of robust tools and visualization plugins designed for diverse purposes, including system monitoring, debugging, data visualization, and parameter configuration. RQT is adaptable, allowing customization and extension using additional plugins to cater to specific development and analysis requirements.
- **Gazebo**
  stands as a formidable robot simulation environment seamlessly integrated with ROS. It empowers developers to design and simulate intricate robotic systems within a virtual

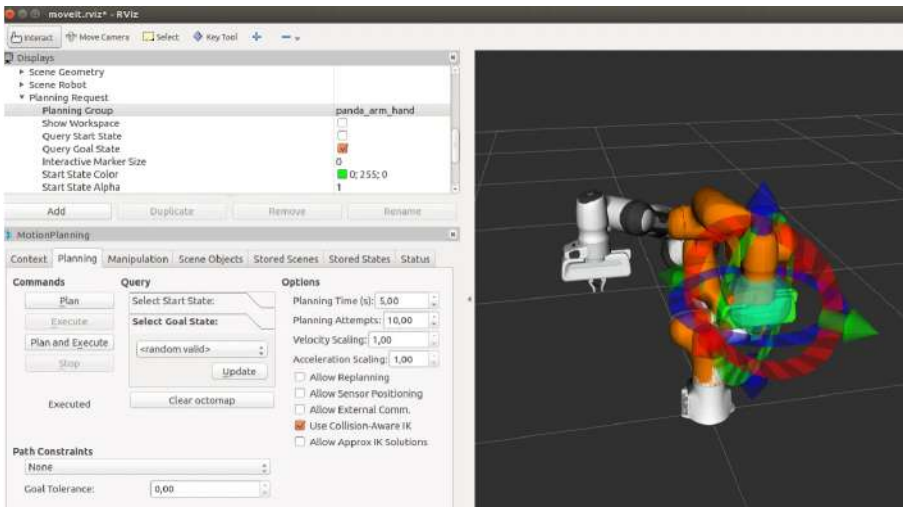


Figure 3.30: 3D Data Visualization using RViz [11]

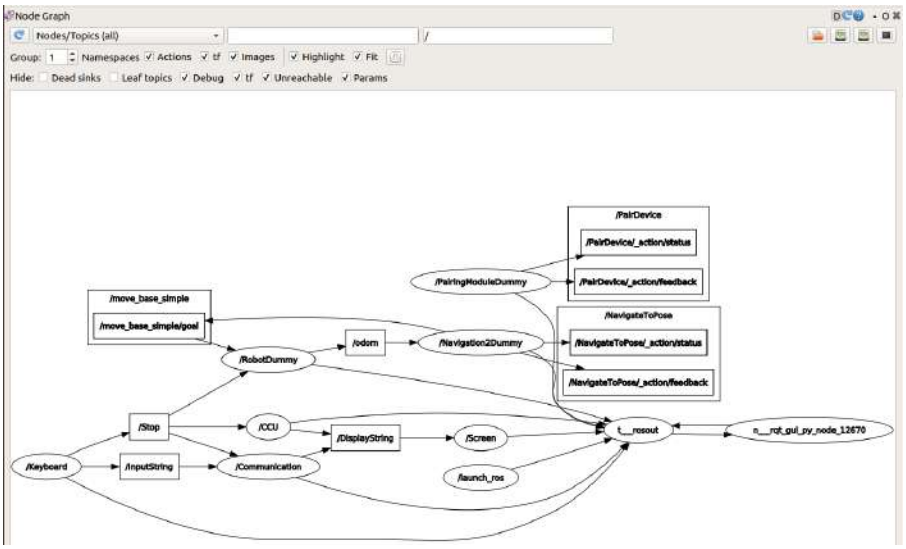


Figure 3.31: RQT tool [12]

space. Gazebo features a graphical user interface (GUI) that facilitates users in visualizing and interacting with simulated robots and their corresponding environments.

3.4.2 Docker

Docker is an open-source containerization platform by which you can pack your application and all its dependencies into a standardized unit called a container. Containers are light in weight which makes them portable and they are isolated from the underlying infrastructure and from each other. You can run the docker image as a docker container in any machine where docker is installed without depending on the operating system.

3.4.2.1 Docker Advantages

Docker gained its popularity due to its impact on the software development and deployment. The following are some of the main reasons for Docker becoming popular:

1. **Portability**: Docker facilitates the developers in packaging their applications with all

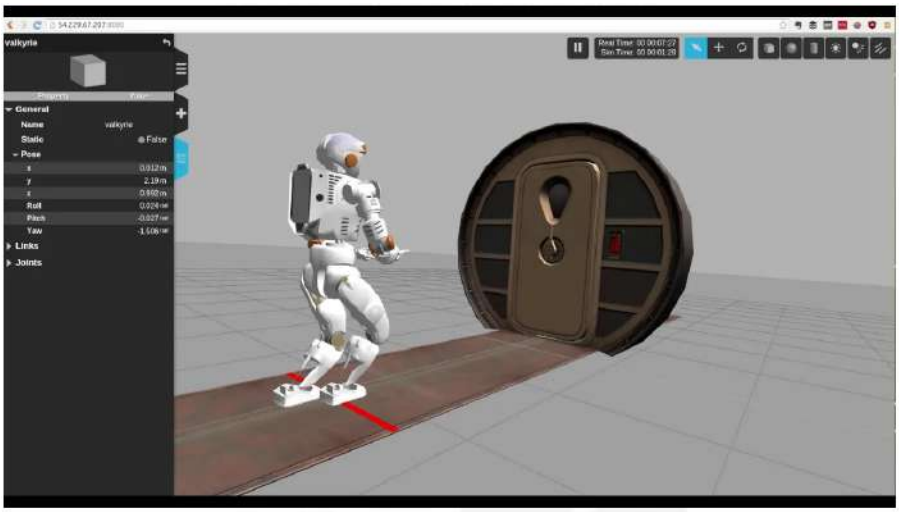


Figure 3.32: Simulation using Gazebo [12]

Figure 3.33: Docker logo [13]

dependencies into a single lightweight container. It ensures consistent performance across different computing environments.

2. **Reproducibility**: By encapsulating the applications with their dependencies within a container, it ensures software setups remain consistent across development, testing, and production environments.

3. **Efficiency**: Docker's container-based architecture optimizes resource utilization. It allows developers to run multiple isolated applications on a single host system.

4. **Scalability**: Docker's scalability features facilitate easier handling of applications during workload increments.

3.4.2.2 Dockerfile

The Dockerfile uses DSL (Domain Specific Language) and contains instructions for generating a Docker image. A Dockerfile defines the processes to quickly produce an image. While creating your application, you should create a Dockerfile in order since the Docker daemon runs all the instructions from top to bottom.

(The Docker daemon, often referred to simply as "Docker," is a background service that manages Docker containers on a system.)

- It is a text document that contains necessary commands which, on execution, help assemble a Docker image.

- A Docker image is created using a Dockerfile.

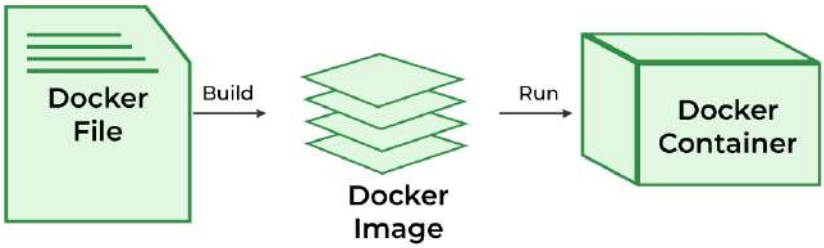


Figure 3.34: Dockerfile usage [14]

3.4.2.3 Docker Image

A Docker image is a file comprised of multiple layers used to execute code in a Docker container. They are a set of instructions used to create Docker containers. A Docker image is an executable package of software that includes everything needed to run an application. This image defines how a container should instantiate, determining which software components will run and how. A Docker container is a virtual environment that bundles application code with all the dependencies required to run the application. The application runs quickly and reliably from one computing environment to another.

3.4.2.4 Docker Container

A Docker container is a runtime instance of an image. It allows developers to package applications with all parts needed, such as libraries and other dependencies. Docker containers are the runtime instances of Docker images. Containers contain everything required for an application, so the application can be run in an isolated way.

3.4.2.5 Docker Architecture

Docker uses a client-server architecture. The Docker client communicates with the Docker daemon, which helps in building, running, and distributing the Docker containers. The Docker client and daemon can run on the same system or connect over a network. Docker uses REST API over a UNIX socket or a network for client-daemon communication. To learn more about Docker's architecture, refer to the official documentation.

3.4.2.6 Docker Commands

By introducing essential Docker commands, Docker has become a powerful tool in streamlining the container management process. It ensures seamless development and deployment workflows. Some commonly used Docker commands are:

- **Docker Run**: Used for launching containers from images, specifying runtime options and commands.

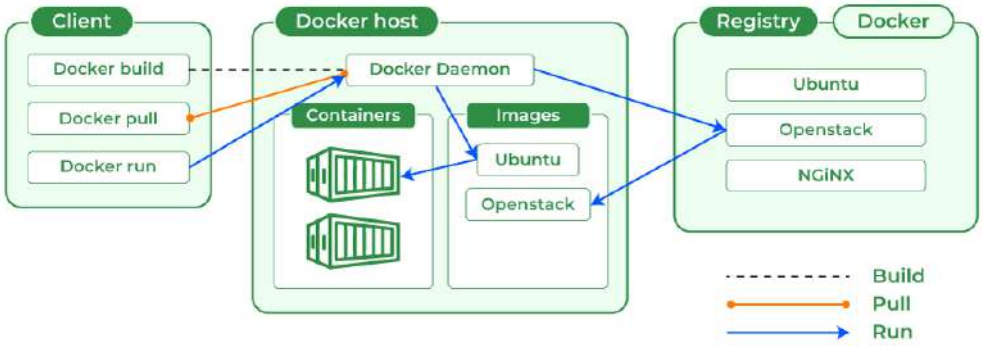


Figure 3.35: Docker Architecture [14]

- **Docker Pull**: Fetches container images from the container registry like Docker Hub to the local machine.
- **Docker ps**: Displays the running containers along with important information like container ID, image used, and status.
- **Docker Stop**: Halts the running containers, gracefully shutting down the processes within them.
- **Docker Start**: Restarts the stopped containers, resuming their operations from the previous state.
- **Docker Login**: Logs into the Docker registry, enabling access to private repositories.

3.4.2.7 Difference Between Docker Containers and Virtual Machines

The following are the differences between Docker containers and Virtual Machines:

Table 3.8: Comparison between Virtual Machines and Containers

	Virtual Machines	**Containers**
1.	Needs a hypervisor and a full OS inside	Talks to the host kernel
2.	Bigger footprint (RAM and storage space)	Smaller footprint (no RAM and differential storage)
3.	VMs consume storage space for each instance ∼1.2GB	Consumes very little space ∼2.5MB
4.	Heavier	Lightweight
5.	Virtual Machines startup time is in the order of minutes	Startup time is in the order of seconds
6.	Deployment is tough	Easy deployment with minimal requirements for running the application
7.	Slower	Faster
8.	Security issues of running OS	Security issues limited to applications

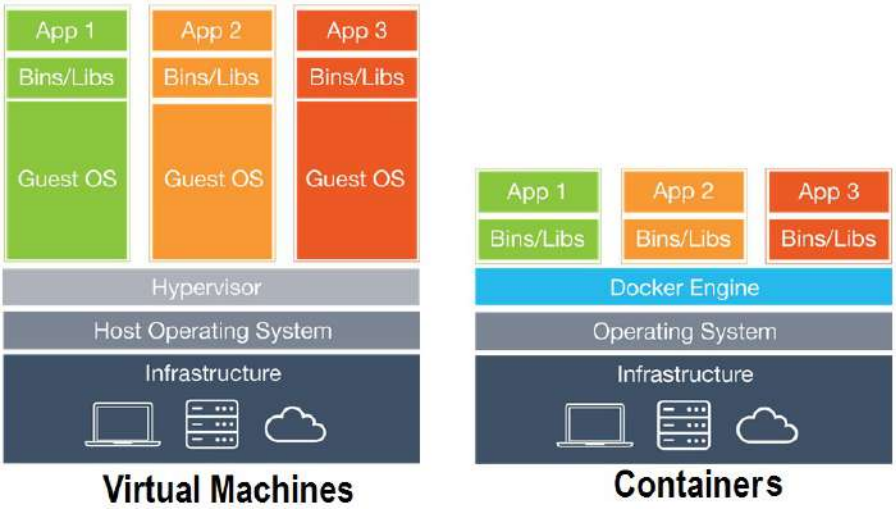


Figure 3.36: Containers vs VMs [13]

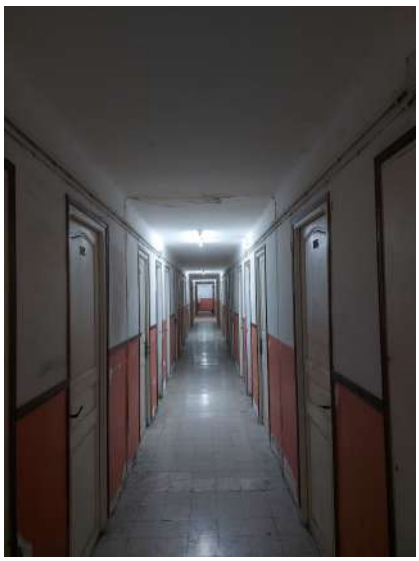
3.5 Experiments

Figure 3.37: The drone fleet created during this thesis

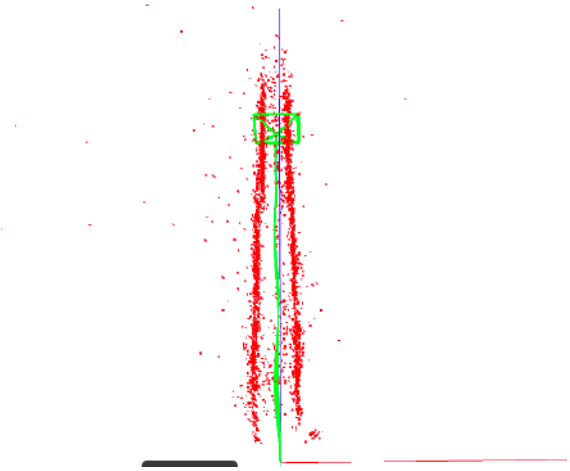
In this section we conducted 2 experiments , each one with the aim of showcasing a different aspect of the algorithm.

3.5.1 Straight corridor Experiment

We conducted an experiment in a corridor to test the performance in a constrained environment. The narrow and linear nature of the corridor provides a challenging scenario for visual SLAM algorithms due to repetitive patterns and limited features.



(a) The corridor chosen for experiment



(b) The corridor map

Figure 3.38: The corridor experiment setup and results

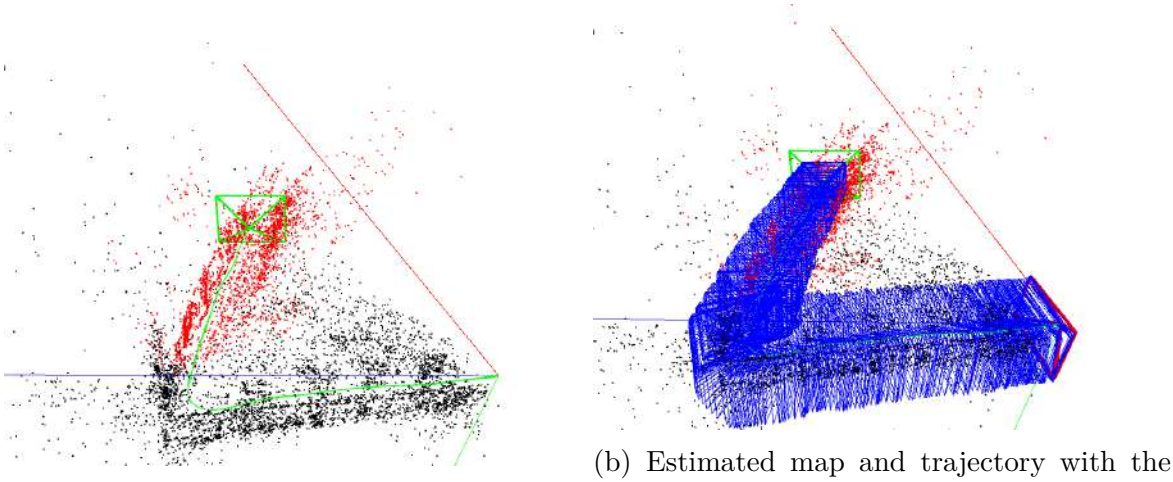
The results showed that ORB-SLAM was able to maintain accurate localization and mapping despite the challenging conditions. The algorithm successfully identified and tracked key features along the corridor, demonstrating robustness in feature-poor environments.

3.5.2 Corridor with rotation

The environment for this experiment consists of two corridors connected perpendicularly to each other. This setup introduces additional complexity for the algorithm due to the significant rotations.

(a) Half of the corridor junction

(b) Second half of the junction



(a) Estimated map and trajectory

(b) Estimated map and trajectory with the camera keyframes

Figure 3.40: Results of the Open Room Experiment

We can see that the algorithm faced challenges in accurately detecting rotations using only a monocular camera. The junction required significant rotational adjustments, which the algorithm struggled with due to the limited field of view and lack of depth information inherent to monocular vision. This led to occasional inaccuracies in the trajectory estimation and mapping process. This is a big limitation of using monocular cameras in environments requiring precise rotation detection.

3.6 Conclusion

This chapter provides a comprehensive overview of the practical implementation of a quadrotor UAV, covering both hardware selection and software integration. By carefully selecting components based on performance criteria and cost analysis, we ensure an optimal balance between functionality and budget.

The implementation of the control system, including parameter identification and EKF, demonstrates the importance of accurate modeling and sensor fusion for achieving stable flight. The use of SITL simulations facilitates thorough testing and validation of control algorithms, ensuring their effectiveness in real-world scenarios.

The SLAM implementation, utilizing ROS and Docker, showcases the power of modern frameworks and tools in developing advanced robotic systems. The experiments conducted highlight the capabilities and limitations of the SLAM algorithm, providing insights for future improvements.

Overall, this chapter lays a solid foundation for building and implementing a high-performance quadrotor UAV, offering valuable guidance for researchers and developers in the field of autonomous aerial systems.

General Conclusion

This thesis has explored the multifaceted aspects of developing and optimizing a quadrotor UAV, with a particular focus on control systems and SLAM algorithms. Through three comprehensive chapters, we have detailed our approach, findings, and the challenges encountered. In the first chapter, we delved into the characteristics of the Pixhawk 2.4.8 flight controller, examining its operating system and flight stack architecture. We developed a mathematical model for the quadrotor and identified the necessary parameters for the state-space representation. We then designed three types of controllers using different methods: the LQR method, pole placement, and the PID method, ensuring a stable and optimal flight performance.

The second chapter introduced and discussed the implementation of Visual SLAM for UAVs using the ORB-SLAM algorithm. We covered the theoretical foundations of Visual SLAM, detailed the workings of the ORB-SLAM algorithm, and demonstrated its practical implementation and testing using the EuRoC MAV Dataset. Our results showed that while the ORB-SLAM algorithm performs well in controlled environments, it faces challenges in more dynamic and complex scenarios, indicating areas for further improvement.

In the third chapter, we conducted the component selection and cost analysis, identified the parameters of our robot, and tested both SLAM and the controller in real-life scenarios. These tests yielded excellent results, confirming the effectiveness of our approaches.

Looking forward, there are several avenues for improving and extending the capabilities of our quadrotor UAV system. For the control algorithms, one significant enhancement would be to improve them with the goal of achieving full autonomy. This includes implementing advanced obstacle avoidance techniques, allowing the drone to perform complete missions on its own without human intervention.

Regarding the SLAM system, integrating IMU measurements from the drone and implementing inertial visual SLAM would help correct the rotation problems encountered with the current monocular setup. Furthermore, employing RGB-D and stereo cameras could provide richer data, enhancing the algorithm's accuracy and robustness in various environments.

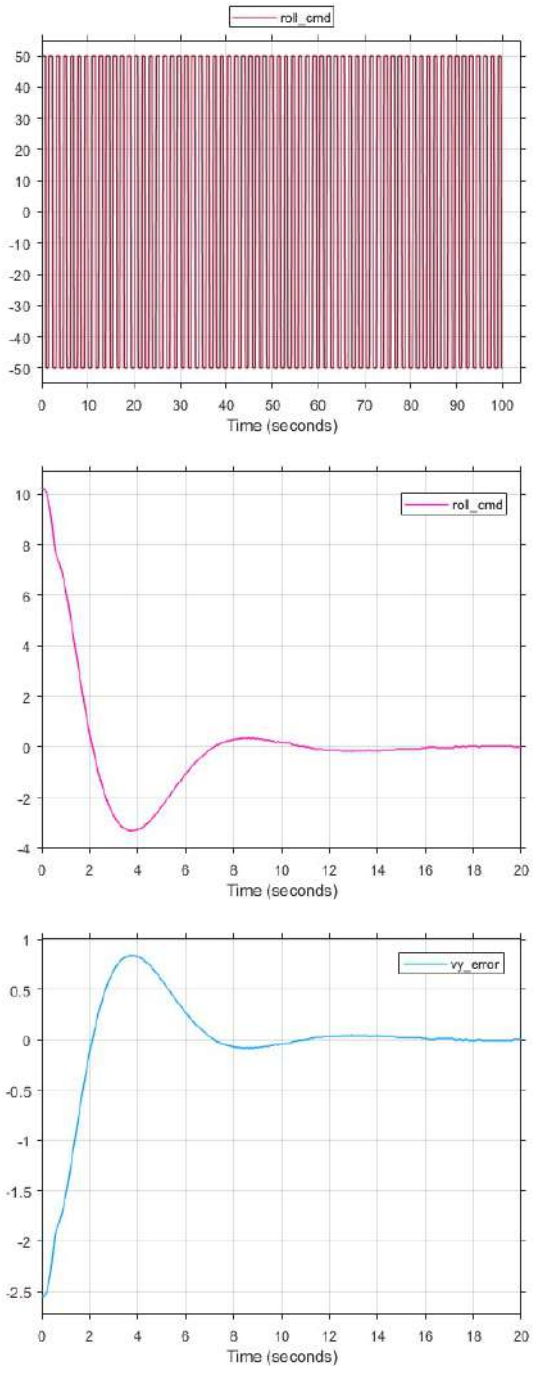
On the hardware side, switching to a larger structure could improve stability and provide better protection for the electronic components. Additionally, using a smaller battery and camera would reduce the overall weight, potentially increasing flight time and maneuverability.

Finally, advancing towards implementing multi-agent systems would represent the next significant milestone. By enabling multiple drones to work collaboratively, we can explore more complex and large-scale missions, pushing the boundaries of what is achievable with autonomous UAV systems.

In conclusion, while our current system has demonstrated substantial capabilities, these proposed improvements and extensions could significantly enhance the performance, reliability, and autonomy of our quadrotor UAV, paving the way for more advanced applications.

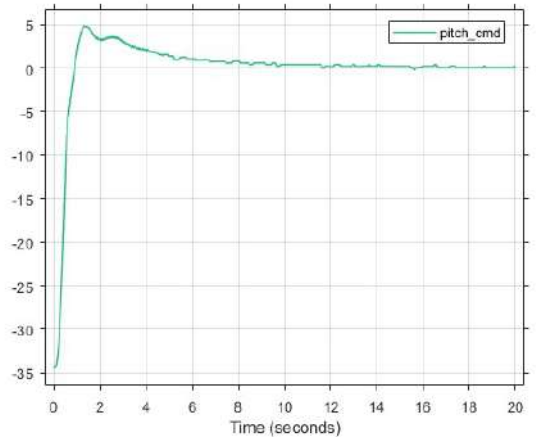
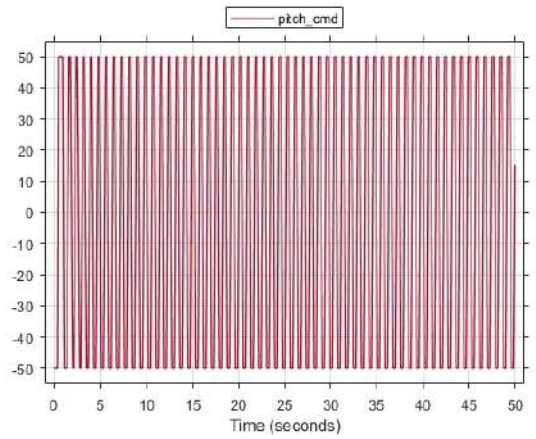
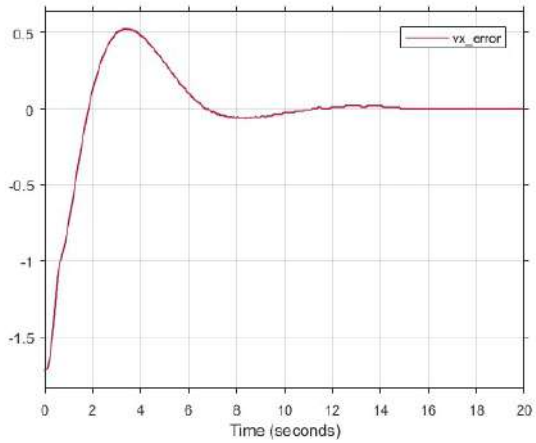
Appendix

Table 3.9: Tuning PID Controllers using Ziegler-Nichols Method

Command	Description	Figures
Roll	Oscillation for $K_{cr} = 55$, Tuned gain $K = 25$	

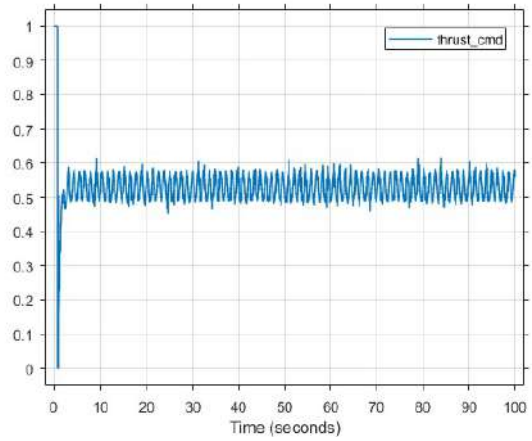
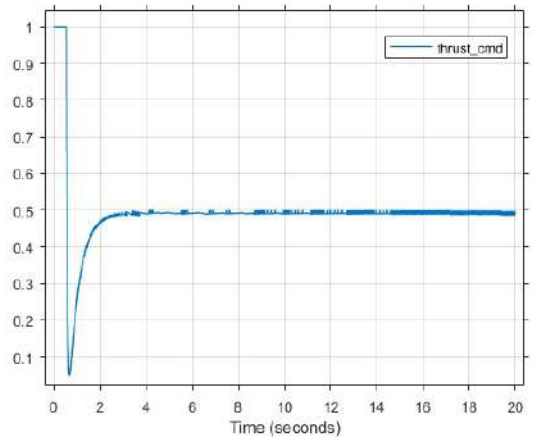
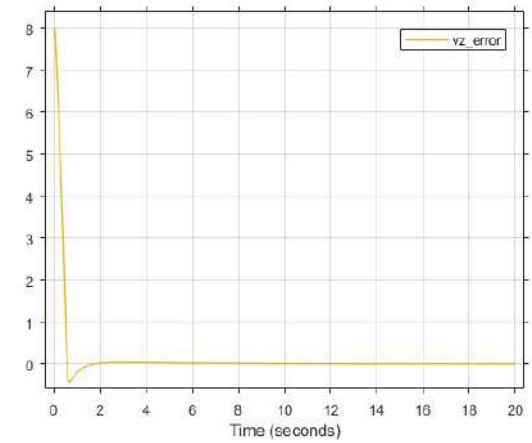
Continued on next page

Table 3.9 – *Continued from previous page*

Command	**Description**	**Figures**
Pitch	Oscillation for $K_{\text{cr}} = 60$, Tuned gain $K = 27$	  

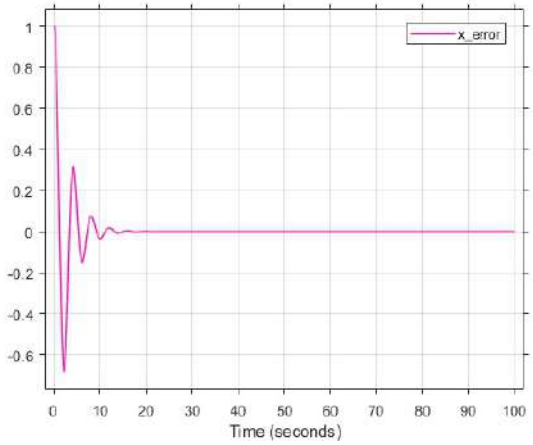
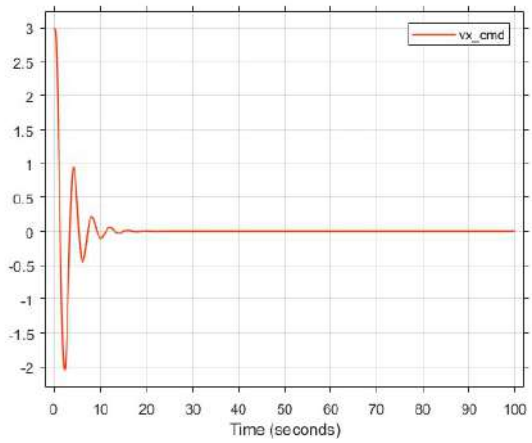
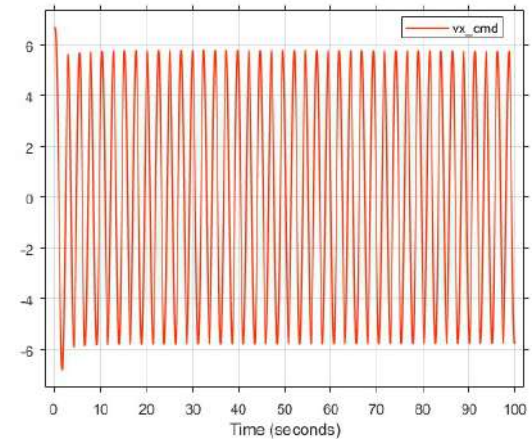
Continued on next page

Table 3.9 – *Continued from previous page*

Command	**Description**	**Figures**
Thrust	Oscillation for $K_{\text{cr}} = 8$, Tuned gain $K = 4$	  

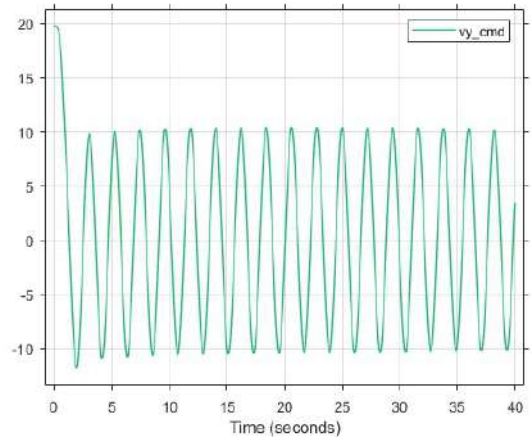
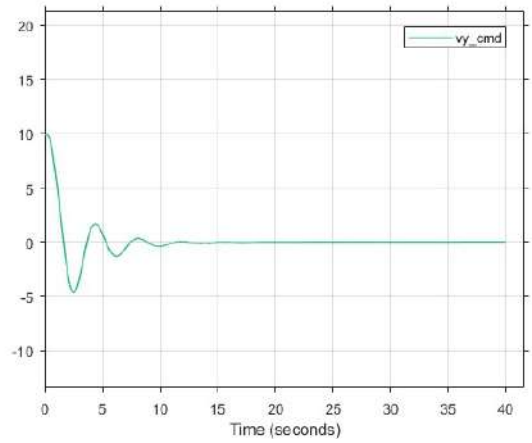
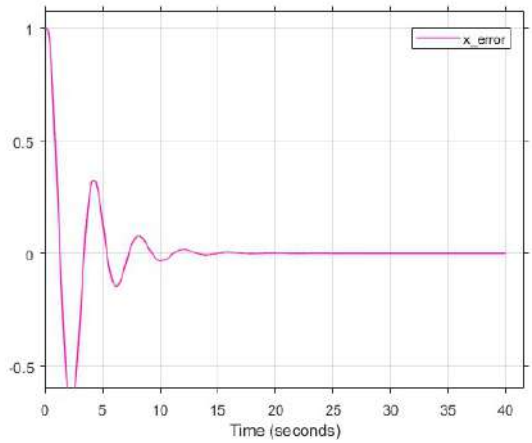
Continued on next page

Table 3.9 – *Continued from previous page*

Command	Description	Figures
V_x	Oscillation for $K_{\text{cr}} = 6.7$, Tuned gain $K = 3.35$	  

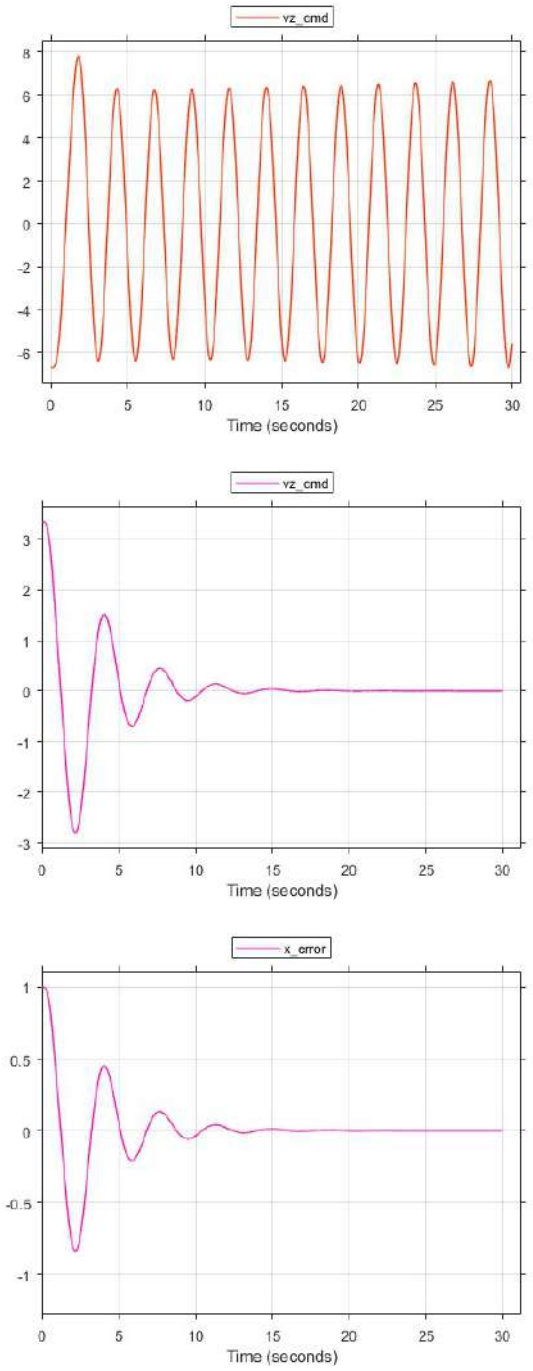
Continued on next page

Table 3.9 – *Continued from previous page*

Command	**Description**	**Figures**
V_y	Oscillation for $K_{\text{cr}} = 6.5858$, Tuned gain $K = 3.2929$	  

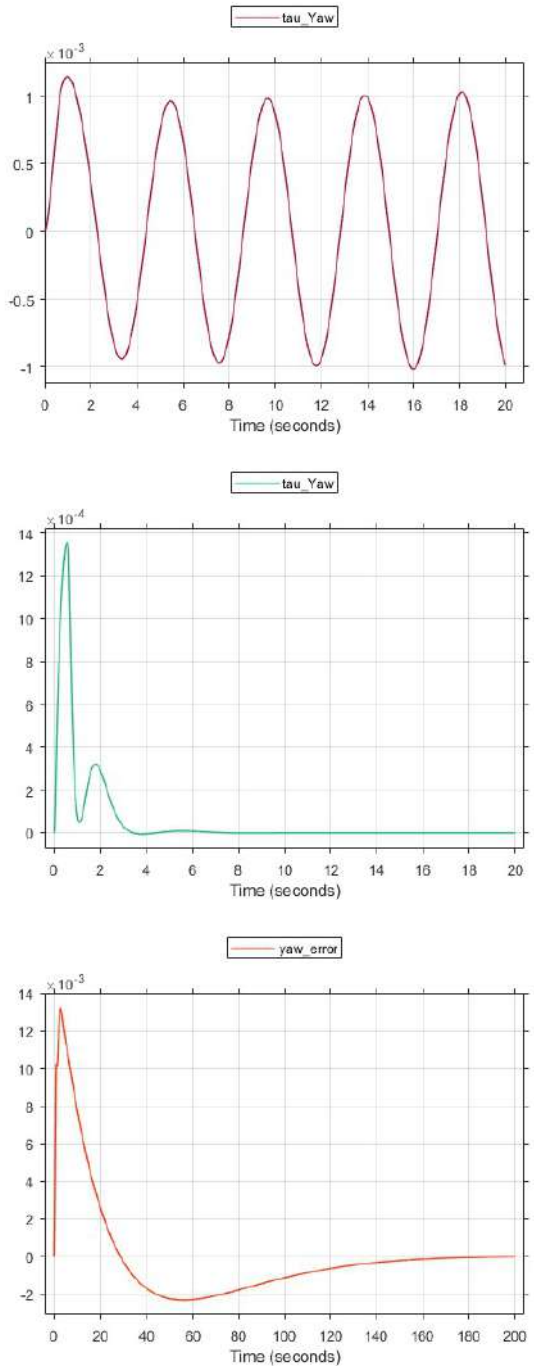
Continued on next page

Table 3.9 – *Continued from previous page*

Command	Description	Figures
V_z	Oscillation for $K_{\text{cr}} = 4.5$, Tuned gain $K = 2.25$	

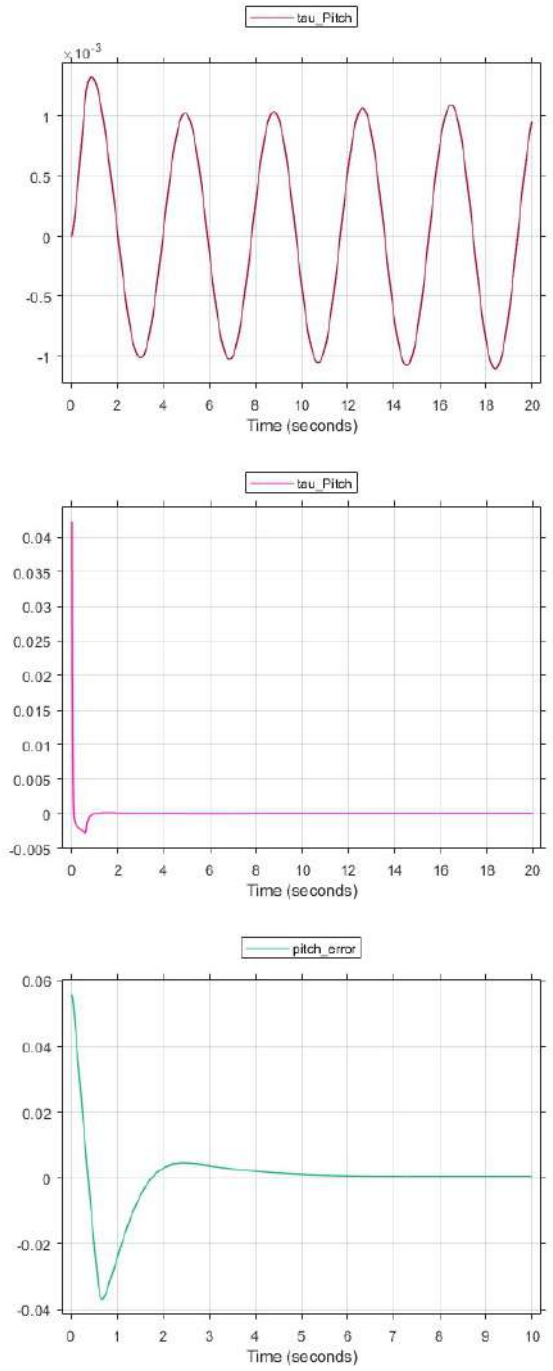
Continued on next page

Table 3.9 – *Continued from previous page*

Command	**Description**	**Figures**
Torque Yaw	Oscillation for $K_{cr} = 0.01$, Tuned gain $K = 0.005$	

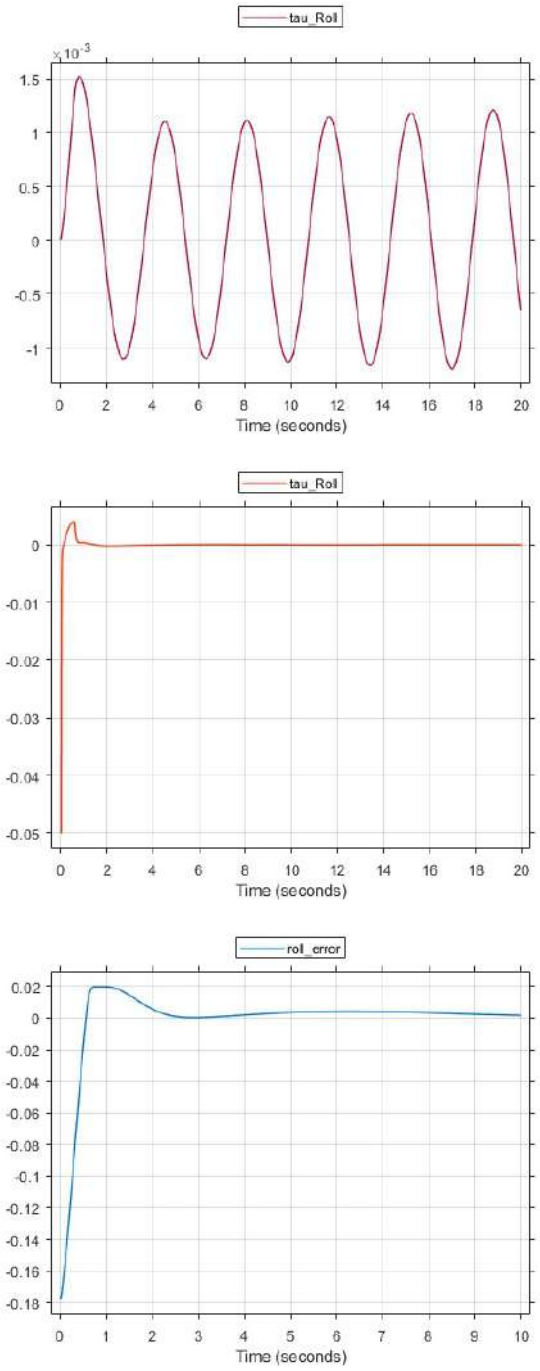
Continued on next page

Table 3.9 – *Continued from previous page*

Command	**Description**	**Figures**
Torque Pitch	Oscillation for $K_{cr} = 0.054$, Tuned gain $K = 0.027$	

Continued on next page

Table 3.9 – *Continued from previous page*

Command	**Description**	**Figures**
Torque Roll	Oscillation for $K_{cr} = 0.06$, Tuned gain $K = 0.03$	

Bibliography

[1] W. Z. Fum. Implementation of simulink controller design on iris+ quadrotor. Technical report, Naval Postgraduate School, 2015. Accessed: 2024-06-07, cited on pp. 4, 8, 12, 19, 31.

[2] L. Meier et al. Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[3] Performance specifications. Class notes for Missile Flight and Control, Dept. of Mechanical and Aerospace Engineering, Naval Postgraduate School, Monterey, CA, spring 2015.

[4] Song Zhang, Shili Zhao, Dong An, Jincun Liu, He Wang, Yu Feng, Daoliang Li, and Ran Zhao. Visual SLAM for underwater vehicles: A survey. *Computer Science Review*, 46(C), November 2022.

[5] Xiang Gao and Tao Zhang. *Introduction to Visual SLAM: From Theory to Practice*. Publisher Name, Publisher Address, 2021.

[6] Ivan Krešo and Sinisa Segvic. Improving the egomotion estimation by correcting the calibration bias. *VISAPP 2015 - 10th International Conference on Computer Vision Theory and Applications; VISIGRAPP, Proceedings*, 3:347–356, 01 2015.

[7] First Principles of Computer Vision. Perspective-n-point (pnp) in computer vision, 2021. Accessed: 2024-06-18.

[8] Shree Nayar. Computing 2d to 3d outgoing ray. YouTube video, 2021. Accessed: 2024-06-23.

[9] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós. Orb-slam: A versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[10] Oscar Liang. Flight controller guide, 2024. Accessed: 2024-06-07.

[11] MoveIt Community. Moveit: Motion planning framework. https://moveit.ros.org/, 2024. Accessed: 2024-06-26.

[12] Open Source Robotics Foundation. Robot operating system (ros) - wiki. http://wiki.ros.org/, 2024. Accessed: 2024-06-26.

[13] Docker. Docker documentation, 2024. Accessed: 2024-06-26.

[14] GeeksforGeeks. Introduction to docker, 2024. Accessed: 2024-06-26.

[15] Emad Samuel Malki Ebeid, Martin Skriver, Kristian Husum Terkildsen, Kjeld Jensen, and Ulrik Pagh Schultz. *A Survey of Open-Source UAV Flight Controllers and Flight Simulators*. University of Southern Denmark, 2023.

[16] MathWorks Pilot Engineering Group. *Pixhawk Pilot Support Package (PSP) User Guide Version 3.04*, 2018.

[17] Q. Jiang and L. Wang. Research on obstacle avoidance system and path planning of unmanned ground vehicle based on px4. *Journal of Robotics and Automation*, 35(2):123–134, 2021.

[18] V. Mazzia L. Comba A. Khaliq M. Chiaberge P. Gay. Uav and machine learning based refinement of a satellite-driven vegetation index for precision agriculture. *Sensors*, 20, 2020.

[19] W.Z. Fum. Implementation of simulink controller design on iris+ quadrotor. 2015.

[20] A. Polak. Px4 development kit for simulink, 2014.

[21] MathWorks. Embedded coder support package for px4 autopilots documentation, 2020.

[22] MathWorks Pilot Engineering Group. *Pixhawk Pilot Support Package (PSP) User Guide Version 3.04*. 2018.

[23] A. Khattab et al. Implementation of sliding mode fault tolerant control on the iris+ quadrotor. In *2018 IEEE Conference on Control Technology and Applications (CCTA)*, page 1724–1729, 2018.

[24] Nuttx real-time operating system, 2020.

[25] MathWorks. *Embedded Coder*, 2020.

[26] Kitware. CMake. cit. on pp. 20, 38.

[27] Pulse width modulation.

[28] Brendon Smeresky, Alexa Rizzo, and Timothy Sands. Kinematics in the information age. *Mathematics*, 6(9):148, 2018. Submission received: 27 June 2018 / Revised: 20 August 2018 / Accepted: 21 August 2018 / Published: 27 August 2018.

[29] S. Bouabdallah. Design and control of quadrotors with application to autonomous flying. Master's thesis, Dept. Sci and Eng., EPFL, Lausanne, Switzerland, 2007.

[30] W. R. Beard. Quadrotor dynamics and control, 2008. Online; accessed 3-Oct-2008.

[31] P. Corke. *Robotics, Vision and Control*. Springer, Berlin, 2013.

[32] A. G. Sidea, R. Y. Brogaard, N. A. Andersen, and O. Ravn. General model and control of an n rotor helicopter. In *J. Phys.: Conf. Series 570*, page 052004, 2014.

[33] T. Bresciani. Modelling, identification and control of a quadrotor helicopter. Master's thesis, Dept. Automatic Control, Lund Univ., Lund, Sweden, 2008.

[34] A. R. Partovi, Z. Y. K. Ang, H. Lin, and G. Cai. Development of a cross style quadrotor. In *AIAA Guidance, Navigation, and Control Conf.*, pages 2012–4780, Minneapolis, MN, 2012.

[35] Andrew Gibiansky. Quadcopter dynamics, simulation, and control. Master's thesis, University of California, Berkeley, 2012.

[36] Matko Orsag and Stjepan Bogdan. Influence of forward and descent flight on quadrotor dynamics. In Ramesh Agarwal, editor, *Recent Advances in Aircraft Technology*. 2012.

[37] Y. C. Choi and H. S. Ahn. Nonlinear control of quadrotor for point tracking: Actual implementation and experimental tests. *IEEE/ASME Transactions on Mechatronics*, 20(3):1179–1192, 2015.

[38] S. H. Dolatabadi and M. J. Yazdanpanah. Mimo sliding mode and backstepping control for a quadrotor uav. In *2015 23rd Iranian Conference on Electrical Engineering*, pages 994–999, 2015.

[39] C. Wu. Robust output feedback position control for quadrotor based on disturbance observer. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 446–451, 2015.

[40] R. Mahony, V. Kumar, and P. Corke. Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor. *IEEE Robotics Automation Magazine*, 19(3):20–32, 2012.

[41] Guillaume Charland-Arcand. Contrôle non linéaire par backstepping d'un hélicoptère de type quadrotor pour des applications autonômes. Maîtrise en génie électrique, École de Technologie Supérieure, Université du Québec, 2014.

[42] Matthew Rich. Model development, system identification, and control of a quadrotor helicopter. Master of science, Iowa State University, 2012.

[43] Katsuhiko Ogata. *Modern control engineering*. Prentice-Hall, Boston, 5th edition, 2010.

[44] K Hassani and W Lee. Optimal tuning of linear quadratic regulators using quantum particle swarm optimization. Technical report, University of Ottawa, School of Electrical Engineering and Computer Science 161 Louis Pasteur, Ottawa, Ontario, Canada, 2014.

[45] A Rahman and SM Ali. Design and analysis of a quadratic optimal control system for a type one plant model. Technical report, Department of Electronics and Communication Engineering, National University of Singapore, Singapore; School of Mechanical & Aerospace Engineering, Nanyang Technological University, Singapore, 2013.

[46] Arthur E Bryson. *Control of spacecraft and aircraft*. Princeton University Press, Princeton, NJ, 2015.

[47] Randal Beard. Quadrotor dynamics and control rev 0.1. https://scholarsarchive.byu.edu/facpub, 2008. Brigham Young University Faculty Publications.

[48] S. D. Hanford, L. N. Long, and J. F. Horn. A small semi-autonomous rotary wing unmanned air vehicle. In *AIAA InfoTech@Aerospace Conf.*, Washington, DC, 2005.

[49] K. Ogata. *Modern Control Engineering*. Prentice Hall, Upper Saddle River, NJ, 5th edition, 2010.

[50] S. Bouabdallah, A. Noth, and R. Siegwart. Pid vs lqr control techniques applied to an indoor micro quadrotor. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2451–2456, Sendai, Japan, 2004.

[51] W. R. Beard. Quadrotor dynamics and control. http://rwbclasses.groups.et.byu.net/lib/exe/fetch.php?media=quadrotor:beardsquadrotornotes.pdf, Oct 2008. [Online].

[52] P. Corke. *Robotics, Vision and Control*. Springer, Berlin, 2013.

[53] C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, 5:56–68, 1986.

[54] J. J. Leonard and H. F. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation*, 7:376–382, 1991.

[55] Hugh Durrant-Whyte, David Rye, and Eduardo Nebot. Localization of autonomous guided vehicles. *Robotics Research*, pages 613–625, Year.

[56] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241, 2001.

[57] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam: A factored solution to the simultaneous localization and mapping problem. 2002.

[58] S. Thrun and M. Montemerlo. Graph-based slam. 2006.

[59] A. J. Davison, I. D. Reid, N. D. Molton, and O. Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, June 2007.

[60] Peter Sturm, Srikumar Ramalingam, Jean-Philippe Tardif, and Simone Gasparini. Camera models and fundamental concepts used in geometric computer vision. *Foundations and Trends in Computer Graphics and Vision*, 6:1–183, 01 2011.

[61] Peter Sturm. Pinhole camera model. 01 2014.

[62] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. 1:666–673, 1999.

[63] Shunyi Zheng, Ruirui Wang, Changjun Chen, and Zuxun Zhang. 3d measurement and modeling based on stereo-camera. 2007. Luoyu Road 129, Wuhan, Hubei, P.R. China.

[64] Hans P. Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. pages 611–616, 1980.

[65] David G. Lowe. Object recognition from local scale-invariant features. pages 1150–1157, 1999.

[66] Chris Harris and Mike Stephens. A combined corner and edge detector. pages 147–151, 1988.

[67] David Marr and Ellen Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 207(1167):187–217, 1980.

[68] Jan J Koenderink. The structure of images. *Biological Cybernetics*, 50(5):363–370, Aug 1984.

[69] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[70] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. 2006.

[71] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. 2010.

[72] Paul J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[73] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. Epnp: An accurate o(n) solution to the pnp problem. *International Journal of Computer Vision*, 81(2):155–166, 2009.

[74] H. Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133–135, 1981.

[75] Olivier D. Faugeras. What can be seen in three dimensions with an uncalibrated stereo rig? 1992.

[76] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. 2003.

[77] Andreas Geiger, Philipp Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. pages 3354–3361, 2012.