DE – 40 (EE)

ARSLAN, IBRAHIM, USAMA, M. YOUSUF

IMPLEMENTATION OF SLAM IN QUADCOPTER

COLLEGE OF
ELECTRICAL AND MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY, RAWALPINDI.

2022

NUST College of Electrical & Mechanical Engineering

PROJECT REPORT

IMPLEMENTATION OF SLAM IN QUADCOPTER

Submitted By:

PC Arslan Khan
40-EE(B)-280779

NC Ibrahim Shafqat
40-EE(A)-241899

NC Usama Faisal
40-EE(B)-259021

NC Muhammad Yousuf Khan
40-EE(B)-246668

Project Supervisor:

Dr. Fahad Mumtaz Malik
(Head of Department)

DEPARTMENT OF ELECTRICAL ENGINEERING


بسم الله الرحمن الرحيم

NUST College of Electrical & Mechanical Engineering

PROJECT REPORT

IMPLEMENTATION OF SLAM IN QUADCOPTER

Submitted to the Department of Electrical Engineering in partial fulfillment of the requirements
for the degree of

Bachelor of Engineering in Electrical
2022

Project Supervisor:

Dr. Fahad Mumtaz Malik
(Head of Department)

DECLARATION

We now declare that no portion of work referred to in this project has been submitted in support of an application for another degree or qualification of this or any other university or other institutes of learning. If any act of plagiarism is found, we are fully responsible for every disciplinary action taken against us depending upon the seriousness of the proven offense, even the cancellation of our degree.

Arslan Khan

Ibrahim Shafqat

Usama Faisal

M. Yousaf Khan

The project, **IMPLEMENTATION OF SLAM IN QUADCOPTER,** is presented by:

1	PC Arslan Khan	40-EE(B)-280779
2	NC Ibrahim Shafqat	40-EE(A)-241899
3	NC Usama Faisal	40-EE(B)-259021
4	NC M. Yousaf Khan	40-EE(B)-246668

(Dr. Fahad Mumtaz Malik)
Project Advisor & Supervisor

ACKNOWLEDGMENTS

First, "**الحمد لله**."

All thanks to Allah (SWT), who created us when we were nothing and provided us with everything. We also thank Allah (SWT) for making it possible for us to imagine, design, build, and perform our project to the best of our abilities, for without His help, nothing would have been possible.

While building this project, we spent much time theoretically and were distrait with which tools we should use to build the project and which methodology to follow? To do this, our supervisor **Dr. Fahad Mumtaz Malik**, supported us completely. Thanks for believing in us, guiding us to the right path, giving us all the advice, and helping us finish our project. He gave us the desire to learn more and opened our eyes to the spiritual world.

We want to thank our parents who made it possible for us to be sitting here in this position, blessed with the resources required to create what we did.

THANKS AGAIN TO ALL WHO HELPED US!

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The project is about implementing SLAM (Simultaneous Localization and Mapping) technology in quadcopters. The algorithm to be implemented is VINS-mono. This project can be divided into four main stages.

1st stage consists of understanding the drone components, dynamics, and control and making it functional for RC control.

2nd stage, the drone's autopilot (flight controller) is to be implemented. For this, real-time data from the Inertial Measurement Unit (IMU) and some peripheral sensors are combined to calculate the drone's relative orientation, velocity, and position. The data is first filtered, biased, and calibrated to get these properties. Now using these attributes, a control system will be implemented for self-stabilization and drone navigation.

3rd stage revolves around getting data, extracting features, and making a visual 3D map of the environment using the camera.

4th stage is the simultaneous integration of the second and third stages of in-flight computers. This part includes the state estimation, localization, and path planning using SLAM's algorithm, i.e., VINS-mono. Finally, the drone is made autonomous to move independently from one point to the other on the map. The drone will also map the environment around it and continuously update it using real-time data from sensors.

CHAPTER 01: INTRODUCTION

1.1 Background

Unmanned Aerial Vehicles (UAVs) have become more and more popular in a wide field of applications nowadays. Although it was first launched for military purposes now it is diversified into other areas. Consider agriculture for example, where they are now used for field observations or for chemicals distribution for fast and efficient growth. They can patrol over wide forest areas as a fireguard, also can be used for traffic observation in the cities. Along with this there is another field that has shoot into skies known as "artificial intelligence" that is to achieve various degrees of autonomy.

Now both the above fields have gained the key-role in academic research because it is obvious that if UAV gets specification of autonomy, then the benefits from the tasks that we mentioned above would be increased tremendously. This is the reason self-driving cars and autonomous quadcopters are attracting scientists and engineers towards them. From engineering point of view, no doubt autonomous UAV design poses a lot of challenges. It is very complex multidisciplinary process, covering disciplines from hardware design, sensors and measurement, programming, networking, etc. to mathematic modeling and control theory, artificial intelligence, computation, machine learning, computer vision, image, and signal processing, etc. Therefore, it is very interesting for researchers from various fields, and there is still plenty of room for improvements and new approaches, as this field is relatively fresh and unexplored.

1.2 Objectives

- Inertial Navigation for Attitude Control
- Design autopilot to achieve stabilization
- Altitude and Attitude control using STM32 for autonomous flight
- State estimation and localization using VINS-Mono SLAM Algorithm

1.3 Design Workflow

Using Arduino as microcontroller:

Here after inertial navigation that is finding the attitude UAV can make itself to follow the commands of transmitter. From transmitter joystick values motion corresponds to the length of pulses that is received and measured by receiver, but it is not that simple to measure from each channel Arduino must call sub-routines and there due to speed and memory limitations and due to problem of refresh rate its efficiency slows down.

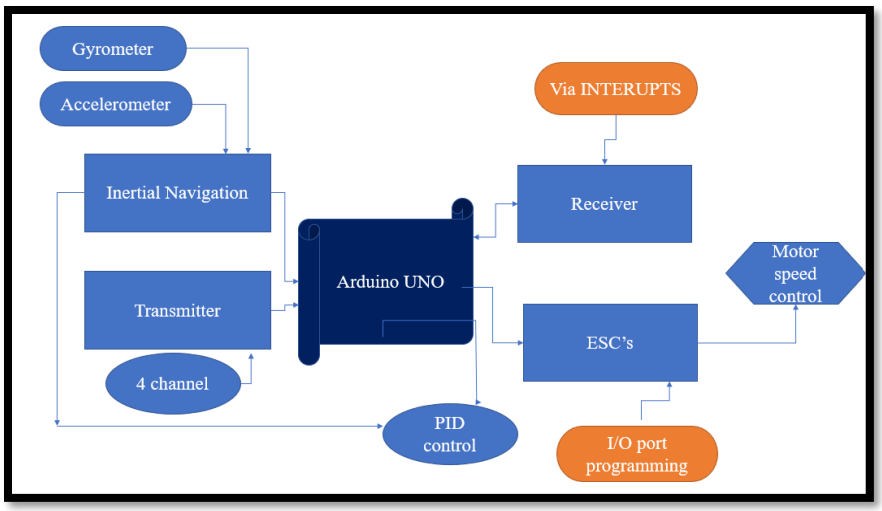

Figure 1: Arduino Design Workflow

Thus, to tackle these issues we make transition from Arduino to STM 32 as follows:

Using STM 32 as microcontroller:

Here STM32 with frequency speed 4 times than Arduino uno which instead of calling subroutines it simply measures the pulse length by using it internal clock counter where register hold the initial and final value and then further processing is also on register level programming using pointer constants that is resultantly much faster and efficient thus resulting in stable flight.

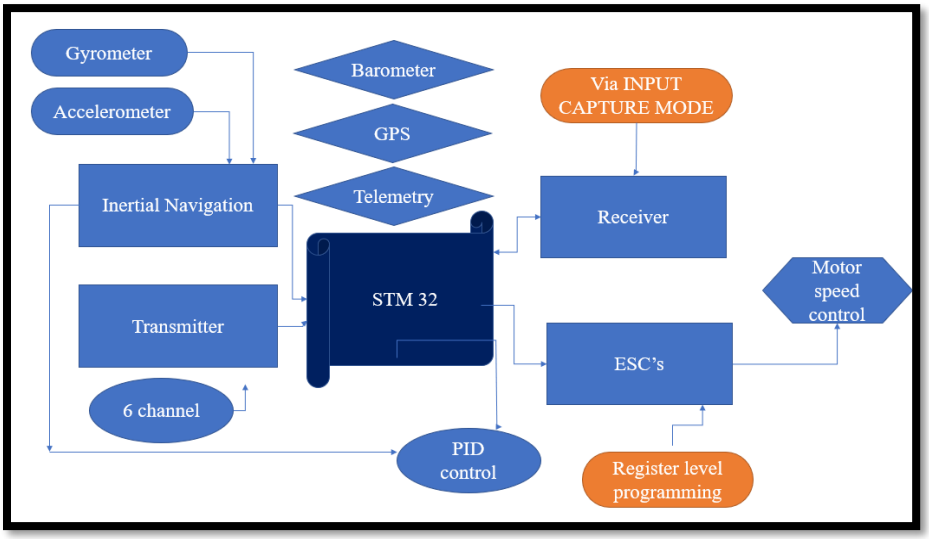

Figure 2: STM32 Design Workflow

1.4 Literature Review

1.4.1 Attitude determination

It is defined as the position or orientation of an object with reference to a coordinate system. There are various sensors for position detection. These include external sensors that measure distance, and active o dead reckoning sensors that use calculations to determine a person or object's current position. Inertial sensing relies on a variety of sensors to measure movement and orientation. These include accelerometers, gyroscopes, and magnetometers. Active sensors are more effective than passive sensors, but they are only useful in certain environments. Furthermore, different techniques are needed to determine attitude from different sensors. Accelerometers and gyroscopes are commonly used in MEMS-based inertial measurement units (IMUs), which are often used in the development of mechanical setups. The attitude of these sensors is discussed using a 6-DOF IMU, the ADIS16350 from Analog Devices. This device has three accelerometers and three rate gyros in three different axes. These sensors are inexpensive and less accurate, but their efficiency can be improved by combining the complementary properties of these devices. In addition to this Vision-based sensors are affordable and lightweight and can be easily deployed with quadcopters for various applications, especially in areas with poor GPS reception, like indoors. Quadcopters are commonly used for target detection and tracking. This thesis deals with the development of a way to track and control a UAV in collaboration with a moving ground vehicle.

1.4.2 Quadcopter control

The control of quadcopters has made progress over time, with significant advances being made in various applications. It is very easy to achieve with multiple algorithms and methods to choose from. A quadcopter has four rotors that are evenly spaced around the perimeter of a square structure. The propellers on an airplane generally spin in the same direction. Two propellers spin clockwise, and the other two spin counterclockwise. By varying the speed of each rotor, the thrust needed to lift and maneuver the craft can be generated. To control the quadcopter, we need to control four rotors independently.

The work in this thesis is focused on developing a tracking and control system for a UAV in concert with a moving ground vehicle. A control system for a quadcopter to follow a moving ground vehicle is developed using feedback. A novel algorithm is developed for vision-based estimation for detecting and tracking of the quadcopter. Six degrees of freedom (6DOF) tracking of the quadcopter is achieved with a ground vehicle-mounted monocular camera that tracks colored markers on the UAV. Finally, the UAV is given a control input for the PID controller to maintain a predetermined path, e.g., Maintain a position 25 meters in front of the vehicle or weave back and forth in front of the vehicle.

Several researchers have presented their approaches to achieving control for the quadcopter, based on a PID controller for attitude stabilization of a quadcopter. It is very easy to achieve with multiple algorithms and methods to choose from. A quadcopter has four rotors that are evenly spaced around the perimeter of a square structure. The propellers on an airplane generally spin in the same direction. Two propellers spin clockwise, and the other two spin counterclockwise. By varying the speed of each rotor, the thrust needed to lift and maneuver the craft can be generated. To control the quadcopter, we need to control four rotors independently. Several researchers have presented their approaches to achieving control for the quadcopter, based on a PID controller for attitude stabilization of a quadcopter.

1.4.3 Visual and Inertial based Navigation System

Our goal is to develop a visual-inertial-based algorithm for locating the quadcopter in relation to the ground vehicle. Multiple frames make solving the problem difficult and time-consuming to implement. We process one image at a time. Heavy computational-load algorithms cannot be used on low-power computers onboard a typically small quadcopter. The main challenge addressed here is tracking and estimating the relative altitude between the quadcopter and ground vehicle based on monocular camera measurements. Using sensor fusion of a monocular camera with GPS or other inertial-based sensors can increase accuracy in measurements obtained.

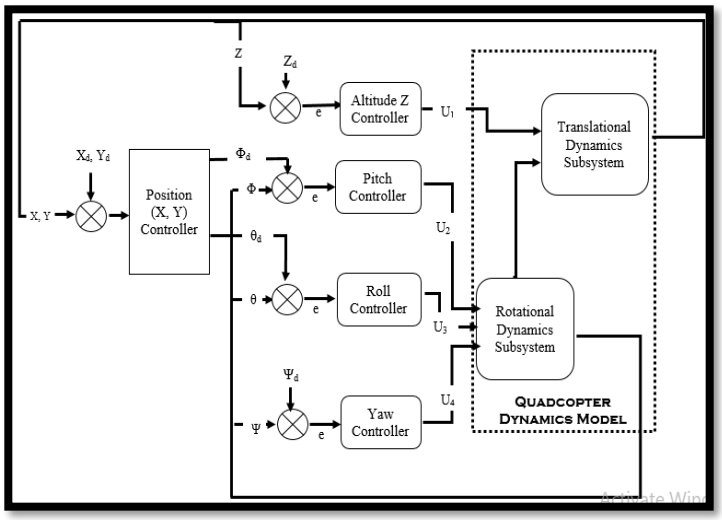

Figure 3: Control System

It is often the case that cameras have an advantage overactive range sensors when it comes to target tracking problems, as the camera is passive, and the target is unaware of observation. Currently, simultaneous localization and mapping (SLAM) techniques are widely used and implemented for monocular vision systems. Inn an algorithm authors use artificial features to reconstruct the 3D pose of an aircraft from a 2D image obtained from a monocular camera. Both approaches propose an algorithm to simplify the pose estimation problem for a quadcopter. These systems utilize an on-board cam era setup for simultaneous localization and mapping (SLAM) with the UAV but require lot of computation. The primary restriction of this technique is that for precise height adjustment, the quadrotor ought to have critical movement in the field of view, which fizzles for a float condition. Quadcopters are tracking down applications in different fields while implementing SLAM.

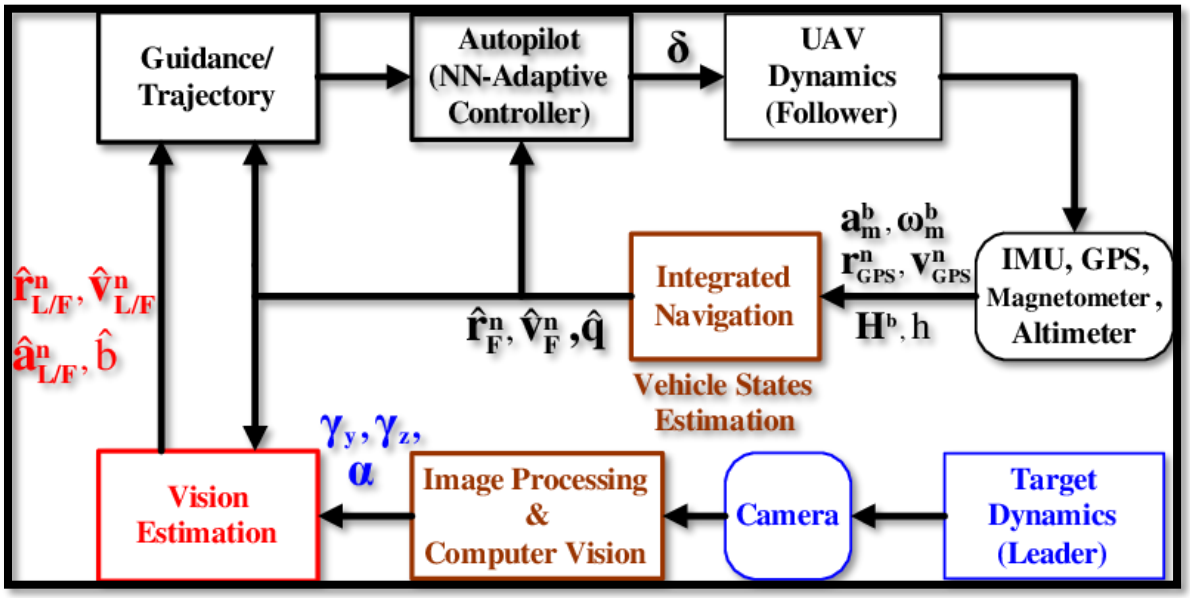

Figure 4: Autonomous Control system

CHAPTER 02: HARDWARE

2.1 Hardware Components

The essential components for a quadcopter and its navigation are:

Circuit	DIY / prototype board
Weight without battery	768gr / 1.69lb
Weight with battery	1077gr / 2.37lb
Propellers	10” x 4.5”
Flight time	15 - 20 minutes
Battery	3 cell / 5000mAh (309gr / 0.68lb)
Motors	EMAX XT2216 910kV
ESC's	Hobbywing Skywalker 30A
Frame	Readytosky Carbon Fiber
Transmitter	EzUHF

2.1.1 Lithium-ion Batteries

The battery acts as a source to the quadcopter. It supplies energy to all the electronic components on the frame through power distribution cables. First, Nickel Metal Hybrid or Nickel Cadmium based batteries were used. However, their use has decreased while the use of lithium batteries has increased. The electrical part of the quadcopter is very critical part concerning to the control and operation.

Specifications:

- **Product Type:** Li-po battery pack
- **Capacity:** 5000mAh
- **Voltage:** 11.1V
- **Max Continuous Discharge:** 60C (300A)
- **Weight:** 342g
- **Balance Plug:** JST-XH
- **Discharge Plug:** XT90
- **Charge Rate:** 1-3C Recommended

Figure 5: Li-po Battery

2.1.2 Frame

Frame is a structure where all the components and other peripherals are added. It is like a skeleton in which different parts are arranged so that they are uniformly distributed to get stability of the drone. They are placed in such a way to keep the center of gravity of the quadcopter in the middle.

Specifications:

- **Brand:** Readytosky
- **Color:** Black
- **Material:** Carbon Fiber
- **Weight:** 454g

- **Item Dimension (L x W x H):** 11.42 x 7.09 x 2.36 inches

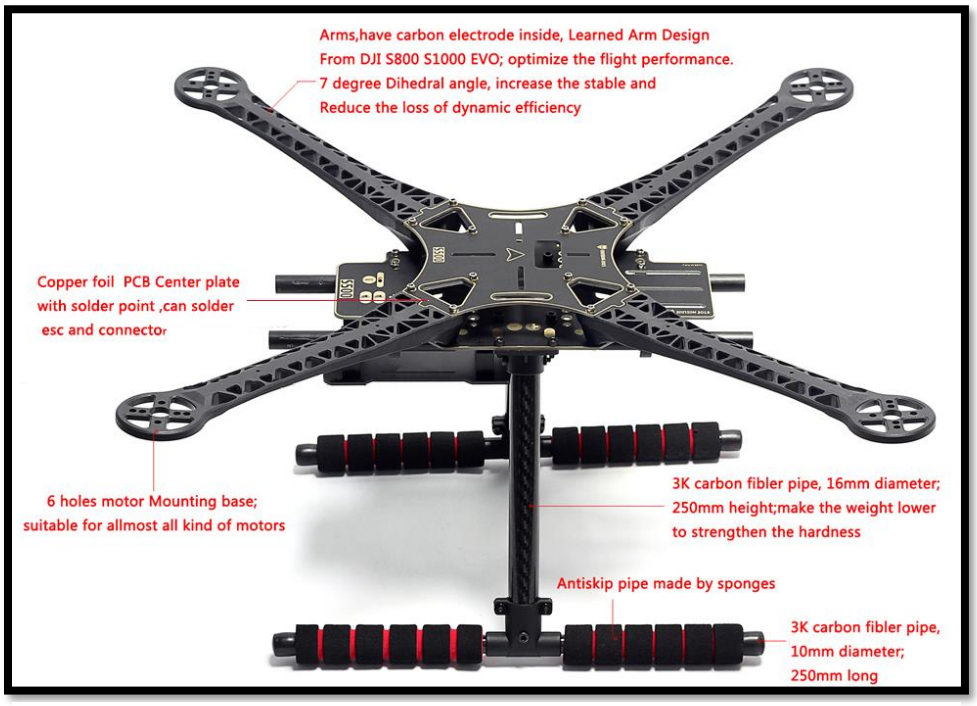

Figure 6: Quadcopter Frame

2.1.3 Flight Control / Board

It is the flight controller which is central to the whole function of the quadcopter. It makes a log of the takeoff place just in case the need arises for the drone to go back to its takeoff location without being guided. This is known as "return to home" feature. It determines and calculates the drone's altitude in respect to the amount of power it consumes.

The flight controller will interpret inputs from the IMU, flight computer and other on board sensors. It will use the data of the different sensors in regulating motor speeds, via ESC.

Arduino

Arduino UNO is a most common microcontroller board having ATmega328P. This microcontroller consists of 14 pins that are digital and analog and can be used as input or output. Among these 14 pins, there are 6 pins that can be used as analog PWM outputs. Arduino UNO also includes a 16 MHz crystal oscillator / resonator, reset button and a USB connector. It also supports UART, I2C and SPI communication protocol. Moreover, an EEPROM is also available with 1Kbs of memory which is volatile. To program it, we can use the Arduino IDE, Arduino CLI and Web Editor.

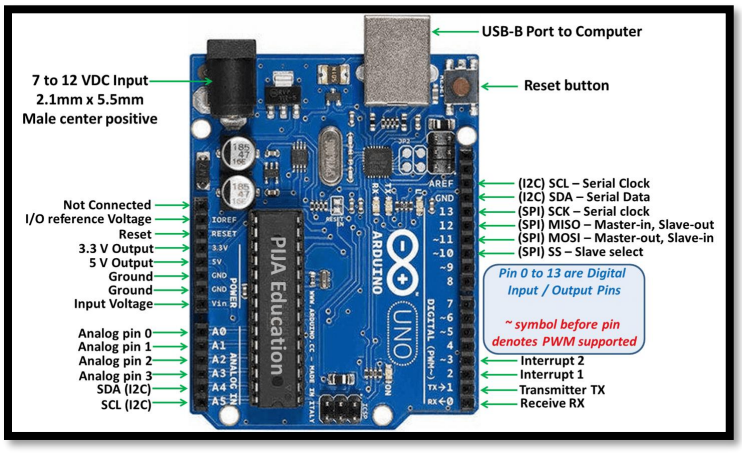

Figure 7: Arduino UNO

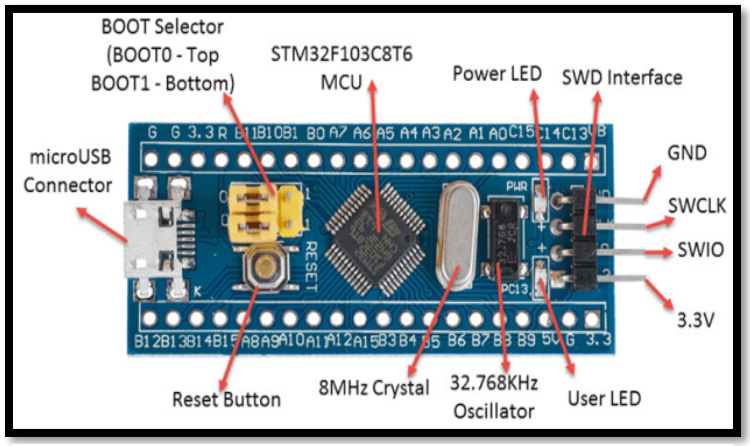

Figure 8: STM32 Microcontroller Board

STM32

STM32 is another microcontroller board which can be used as a flight controller for quadcopter. It is almost 4 times faster than Arduino and supports several functionalities. It belongs to ARM cortex M3 family having high-performance and 32-bit RISC core processor. Since it has 72 MHz frequency, it is high-speed. With 12-bit ADCs, three timers each 16-bit and One PWM timer, it supports three USARTs, two I2C and SPI and a USB and a CAN communication interfaces. This device operates on 2 to 3.6 V power and 5 V can damage it. It comes in different packages from 36 pins to 100 pins.

It has following specifications:

- **Flash Memory (KB):** 64KBs – 128 KBs, **SRAM:** 20KBs
- **Clock Frequency:** 72 MHz
- **Communication Protocol:** I2C, UART, SPI, CAN, and USB

2.1.4 BLDC Motors

Motors are essential for propeller's rotation. This enhances a thrust force for propelling the drone. The BLDC motor is electrically commutated by power switches instead of brushes. Compared with a brushed DC motor or an induction motor, the BLDC motor has many advantages:

- Higher efficiency and reliability
- Lower acoustic noise
- Smaller and lighter
- Greater dynamic response
- Better speed versus torque characteristics
- Higher speed range
- Longer life

Figure 9: BLDC Motor

How do BLDC Motors Work?

BLDC motors are different from other DC motors because their rotor contains no coils and is simply made of a permanent magnet. The stator contains coils that induce a magnetic field when a current is passed through them. When any given stator coil is energized, the rotor will be attracted to any powered pole in the stator. Designers have cleverly created electronic commutators that power the stator poles around the rotor on and off in sequence, thus leading the rotor around and causing rotation on the output shaft. By adjusting the magnitude and direction of current flow through the stator, operators can generate a range of speeds and torques, all synchronous to the input frequency.

These motors rely on electronic commutation to function. It is the electrical analog to a brushed DC motor's brush/commutator ring, which are used to switch the current from winding to winding

mechanically. BLDC motors are frictionless, produce no brushing dust, and are quieter than brushed DC motors.
BLDC motors, due to their electronic commutation, are highly efficient and last much longer than other electric motors. For this reason, they are best suited for continuous operations where the motor will be powered for extended periods.

Specifications

- Peak, Locked Rotor, & Rated Torque
- Motor Size
- Rated Voltage
- No-Load Speed & Speed Range

2.1.5 Propellers

It is a mechanical device for propelling a boat or aircraft, consisting of a revolving shaft with two or more broad, angled blades attached to it. It is a replaceable part of a drone that generate airflow by rapidly spinning which acts as a propulsion system that creates lift and enables a drone to fly. Changing the speed of specific propellers enables it to ascend, descend, hover in place and it also affects the drone's yaw, pitch, and roll.

Specifications:

- Very strong and lightweight
- **Material:** Carbon Fiber
- **Size:** 10* 4.5 inch
- **Length:** 34.8mm
- **Diameter:** 27.9mm
- **Number of cells:** 3s max
- **Max thrust:** 960g
- **Shaft diameter:** 3mm
- **Recommended prop size:** APC 11"
- **Weight:** 65g

Figure 10: Propellers

2.1.6 ESC

ESC is an electronic control board that varies the motor's speed and acts as a dynamic brake. The sensors help the ground pilot to approximate the height at which the quadcopter is running in. It is attained by gauging the amount of power used by all the motors. Altitude is associated with power drain from the power reservoirs.

Specification:

- **Type:** Brushless
- **Continuous / Peak Current:** 30A/40A
- **Input:** 2-3S Li-po
- **BEC Output:** 5V, 2A
- **Size:** 68.0x25.0x8.0mm
- **Weight:** 37g

Figure 11: Hobbywing Skywalker ESC

2.1.7 MPU-6050

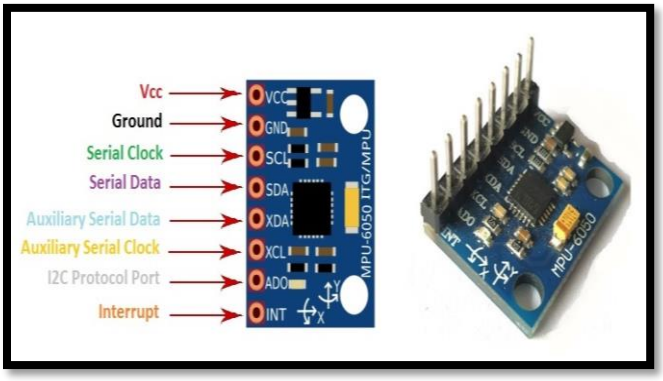

Figure 12: MPU6050

The MPU-60X0 is the world's first integrated 6-axis motion tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, and a Digital Motion Processor™ (DMP) all in a small 4x4x0.9mm package. Its dedicated I2C sensor bus directly accepts inputs from an external 3-axis compass to provide a complete 9-axis Motion Fusion™ output.
The MPU-60X0 features three 16-bit analog-to-digital converters (ADCs) for digitizing the gyroscope outputs and three 16-bit ADCs for digitizing the accelerometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of ±250, ±500, ±1000, and ±2000°/sec (DPS) and a user-programmable accelerometer full-scale range of ±2g, ±4g, ±8g, and ±16g.
Communication with all device registers is performed using I2C at 400kHz or SPI at 1MHz (MPU-6000 only). Additional features include an embedded temperature sensor and an on-chip oscillator with ±1% variation over the operating temperature range.

Gyroscope Features

- The triple-axis MEMS gyroscope in the MPU-60X0 includes a wide range of features:
- Digital-output X-, Y-, and Z-Axis angular rate sensors (gyroscopes) with a user-programmable full-scale range of ±250, ±500, ±1000, and ±2000°/sec
- The external sync signal connected to the FSYNC pin supports image, video, and GPS synchronization
- Integrated 16-bit ADCs enable simultaneous sampling of gyros
- Enhanced bias and sensitivity temperature stability reduce the need for user calibration
- Improved low-frequency noise performance
- Digitally programmable low-pass filter
- Gyroscope operating current: 3.6mA
- Standby current: 5µA
- Factory calibrated sensitivity scale factor

Accelerometer Features

- The triple-axis MEMS accelerometer in MPU-60X0 includes a wide range of features:
- Digital-output triple-axis accelerometer with a full-scale programmable range of ±2g, ±4g, ±8g, and ±16g
- Integrated 16-bit ADCs enable simultaneous sampling of accelerometers while requiring no external multiplexer
- Accelerometer normal operating current: 500µA
- Low power accelerometer mode current: 10µA at 1.25Hz, 20µA at 5Hz, 60µA at 20Hz, 110µA at 40Hz
- Orientation detection and signaling
- Tap detection
- User-programmable interrupts
- High-G interrupt

2.1.8 Transmitter

Transmitter provides 4 channels to the ESCs through receiver with each of the channel basically corresponds to joystick position of all four commands like throttle, roll, pitch, and yaw. It basically converts these into pulse width which determines the speed of motor

Figure 13: EzUHF

Specifications:

- FHSS Link 433MHz/70cm Ham band operation
- Options for 431-433MHz, 433-435MHz, 435-437MHz.
- Clean GFSK modulation
- With 200mW low-power mode, 600mW (~28dBm) output power
- Power switch, boost power when at long range
- Standard SMA antenna connector, compatibility with 433MHz/70cm Ham-band antennas
- Flexible buffered, AC-coupled, PPM input, ensures compatibility with 3.3v thru 9.6v PPM levels
- Standard USB port (Mini-A) for firmware upgrades, and system configuration
- Single pushbutton learning of failsafe positions
- Binding function, which allows binding Single Tx->Rx, Multi Tx->Rx, and Single Tx->Multi Rx
- Mates with 8 channels, and 4 channel EzUHF receivers
- Power Requirements: 9-12v DC 250 mA @ 12v, in 600mW mode, 85mA @ 12v, in 200mW mode

2.1.9 Receiver

To make our quadcopter get certain angles we need to transmit the signal to the receiver through which electronic speed controllers (ESC's) get pulses varying from 1000 to 2000 microsecond. Once receiver signal is received the next step is to transmit these signals towards motor such that we get speed variation for achieving correct angles. But it is not as simple we cannot transmit signals directly to motors but us 4 ESC's that perform this job.

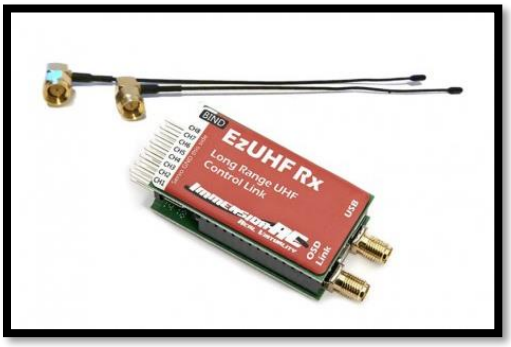

Figure 14: EzUHF Receiver

Specifications:

- 8-Channels RC long-range receiver
- Each antenna input contains a 7-section ceramic low-pass filter for countering interference
- Programmable MK mode, emits raw PPM on a chosen channel
- Reliable FHSS Link (Frequency Hopping, Spread Spectrum)
- 433MHz/70cm Ham band operation
- -112dBm Sensitivity
- Bi-directional ready * (contains onboard transmitter)

2.1.10 Remote Controller

The RC (remote controller) receives 4 types of inputs:

1. Main Sticks
2. Potentiometers
3. Trims
4. Switches

The analog inputs (sticks and pots) go through a calibration phase. The sticks can also go through Expo and Dr filters before going to the mixer. The mixer does it all. It directs each input to the desired output (CH1….CH16). It controls how the inputs are added. It also controls the timing of each function. After the inputs are processed by the mixer, they are directed to the relevant output channels. The limit procedure takes over and makes sure no output goes too far. Finally, the channels are encoded and sent to the RF module to take that nice little hike through the air to your model (drone's receiver antenna).

Specifications:

- **Channel:** Up to 24 supported
- **Display:** 128 * 64 LCD
- **Stick modes:** 1, 2, 3, 4
- **Encoder type:** ppm/pcm
- **Module Interface:** JR Compatible
- **Simulator Interface:** Yes (JR & Futaba)
- **Buzzer/Speaker:** Yes
- **Battery Compartment:** 112 x 44 x 27mm
- **Weight:** 723g

Figure 15: Turnigy 9XR PRO RC

2.1.11 Camera

When we talk about a drone or a drone camera, we're typically referring to a quadcopter configuration. These models dominate the market almost exclusively thanks to their relative simplicity compared to single-rotor or other systems.

Quadcopter drone cameras maintain stability, direction, and motion using four rotors: two rotate clockwise and the other two counterclockwise. A mounted camera sits at the center to take photos and record videos for visual mapping to be used in implementation of SLAM.

Specifications:

- **Brand:** GoPro Hero 7
- **Dimensions:** 1.75 x 2.44 x 1.26 inches
- **Weight:** 119g
- **Special feature:** Image-stabilization
- **Video Capture Resolution:** 1080p, 4K
- **Image Capture Speed:** 30 fps
- **Connectivity Technology:** USB

Figure 16: GoPro Hero 7 Camera

2.1.12 GPU

Figure 17: NVIDIA JETSON NANO

General purpose graphics processing units (GPGPUs) are ideal for a wide range of applications relevant to UAVs (unmanned aerial vehicles) unmanned systems and robotics. Due to their ability to quickly process large amounts of data, they are ideal for running sophisticated artificial intelligence (AI), machine learning, deep learning, and computer vision algorithms.
Algorithms such as SLAM (simultaneous localization and mapping) can allow vehicles to map and travel autonomously within their environment by making use of data from sensors such as cameras. We have used the NIVIDIA Jetson NANO as GPU for our SLAM computations.

Specifications:

GPU	128 Core Maxwell, 472 GFLOPs (FP16)
CPU	4 core ARM A57 ® 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	16 GB eMMC
Video Encode	4K @ 30 I 4x 1080p @_1 30 I 8x 720p (2) 300-L164/H.165)
Video Decode	4K 60 I 2x 41(@ 30 I 8x 1080p ® 30 I 16x 720p ® 30 I (1-1.264/11.265)
Display	HDMI 2.0 or DP1.2 I eDP 1.4 I DSI (1 x 2), 2 simultaneous
SDI OISPI / 12C	lx SDIO lx SPI 5x SysIO 13x GPIOs 6x 12C

2.1.13 GPS

The M8N GPS Module with Compass provides the best performance and easiest RF integration of all the NEO-M8 configurations. The NEO form factor allows easy migration from previous NEO generations. Sophisticated RF-architecture and interference suppression ensure maximum performance even in GNSS-hostile environments. NEO-M8 combines a high level of robustness and integration capability with flexible connectivity options. The future-proof NEO-M8N includes an internal Flash that allows simple firmware upgrades for supporting additional GNSS systems.
Ublox NEO-M8N GPS Module for APM APM2.52 APM Flight Controller with Case and GPS Antenna Mount to protect the GPS preventing electromagnetic interference. A new generation Ublox GM8N GPS Module with Compass, with low power consumption and high precision, the ultimate accuracy is 0.6 meters, almost 0.9 meters, greater than the previous generation NEO-7N 1.4-1.6 meters accuracy. Support GPS/QZSS L1 C/A, GLONASS L10F, BeiDou B1 protocol, and mode or more. The NEO-M8N and ceramic antenna make this a very accurate receiver, fast locks & lots of satellites, and the stand is very sturdy to protect the GPS preventing electromagnetic interference. Moreover, M8N GPS Module with compass Can be integrated with APM 2.8.
GPS/QZSS L1 C/A, GLONASS L10F, BeiDou B1

Specifications:

- SBAS L1 C/A: WAAS, EGNOS, MSAS
- Galileo-ready E1B/C (NEO-M8N)

- Onboard Compass
- Assistance AssistNow GNSS Online
- AssistNow GNSS Offline (up to 35 days)
- AssistNow Autonomous (up to 6 days)
- OMA SUPL & 3GPP compliant
- Oscillator TCXO (NEO-M8N/Q),
- Crystal (NEO-M8M)
- RTC crystal Built-In
- Noise figure On-chip LNA (NEO-M8M). Extra LNA for
- lowest noise figure (NEO-M8N/Q)
- Anti-jamming Active CW detection and removal. Extra
- onboard SAW bandpass filter (NEO-M8N/Q)
- Memory ROM (NEO-M8M/Q) or Flash (NEO-M8N)
- Supported antennas Active and passive
- Odometer Travelled distance
- Data-logger for position, velocity, and time (NEO-M8N)

2.1.14 Barometer

The MS5611-01BA is a new generation of high-resolution altimeter sensors from MEAS Switzerland with SPI and I2C bus interface. This barometric pressure sensor is optimized for altimeters and variometers with an altitude resolution of 10 cm. The sensor module includes a high linearity pressure sensor and an ultra-low power 24-bit ΔΣ ADC with internal factory calibrated coefficients. It provides a precise digital 24 Bit pressure and temperature value and different operation modes that allow the user to optimize for conversion speed and current consumption. A high-resolution temperature output allows the implementation of an altimeter/thermometer function without any additional sensor. The MS5611-01BA can be interfaced to virtually any microcontroller. The communication protocol is simple, without the need of programming internal registers in the device. Small dimensions of only 5.0 mm x 3.0 mm and a height of only 1.0 mm allow for integration in mobile devices. This new sensor module generation is based on leading MEMS technology and latest benefits from MEAS Switzerland proven experience and know how in high volume manufacturing of altimeter modules, which have been widely used for over a decade. The sensing principle employed leads to very low hysteresis and high stability of both pressure and temperature signal. The barometer that is being used is MS5611.

Specifications:

- High resolution module, 10 cm
- Fast conversion down to 1 ms
- Low power, 1 µA (standby < 0.15 µA)
- QFN package 5.0 x 3.0 x 1.0 mm3
- Supply voltage 1.8 to 3.6 V
- Integrated digital pressure sensor (24-bit ΔΣ ADC)
- Operating range: 10 to 1200 mbar, -40 to +85 °C
- I2C and SPI interface up to 20 MHz
- Excellent long-term stability

2.1.15 Telemetry System

The telemetry system that we are using is with the help of APC-220 Wireless RF Modules with USB converter. For the ground station, the Arduino UNO clone with LCD breakout board with buttons are used. The APC-220 telemetry maintains $\sim 3.3V$ on the receiver pin and it is suitable for 3.3 V STM32 microcontroller. The other specifications of APC-220 telemetry are:

- Working frequency: 431 MHz to 478 MHz
- Power: 3.3-5.5V
- Current: < 25 - 35mA
- Working temperature: $-20°C \sim +70°C$
- Range: 1200 m line of sight (1200 bps)
- Interface: UART/TTL
- Baud rate: 1200 – 19200 bps
- Baud rate (air): 1200 – 19200 bps
- Receiver buffer: 256 bytes
- Size: 37 mm X 17 mm x 6.6 mm
- Weight: 30 g

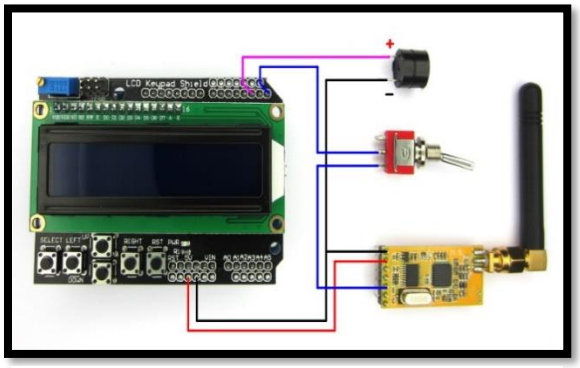

Figure 18: Telemetry System

2.2 Choice of Hardware

The motor size is the general dimensions of the motor, its form factor, and any other important geometric information. BLDC motors come in many forms, and their size will partly dictate their performance in each application.

2.2.1 How much does a drone weigh?

It's difficult to know the weight of your drone at the planning stage - after all, you didn't choose the motors yet, so how can you determine their weight? Nevertheless, a reasonable estimate is sufficient for our drone thrust calculator. You can always adjust it later once you know more about the construction of your multi-copter.

We split the total drone weight into three main components:

- Drone weight is the weight of the main body - the frame, motors, propellers, landing gear, and so on.
- Battery weight is the weight of your LiPo battery.
- Equipment weight is the weight of detachable equipment - for example, if you're planning to use the drone for aerial photography, this will include the weight of the camera.

2.2.2 Drone thrust to weight ratio

Before you can determine the optimal thrust of the drone motor, you need to pick one more important parameter: the thrust-to-weight ratio (TWR). Now, first we will see what is thrust to weight ratio in Drone.

The figure illustrates the main dynamics of the drone. For the motion of the drone the only force that the drone can exert is the Thrust from the Propellers.

For our drone to make it hover the minimum TWR is the 2:1 for any drone so in the first part of making the drone. The first thing is to be considered is the weight of the drone, so the choice of the motor is the main part. The higher the thrust to weight ratio, the easier it is to control your drone in elaborate aerobatics.

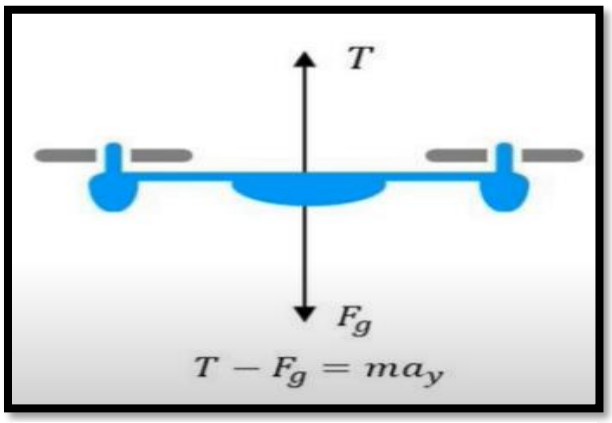

Figure 19: Drone Thrust to Weight Ratio

2.2.3 Calculating the motor thrust: an example

To get our drone airborne, we need the right amount of thrust from its motor. Using the steps below to calculate the required amount of thrust and choose the right engine to support our drone.

STEPS:

Step 1: Determine the weight of the drone. Double the drone's weight to calculate the minimum amount of thrust we'll need from a motor.

Step 2: Adding 20% to that total to ensure that our drone will be able to hover. As in our case the weight of our drone is 1650 grams, we'll need 3300 grams of thrust to get it off the ground, plus 600 additional grams to get it to hover. The number you come up with is our thrust total is 3900.

Step 3: Dividing our thrust total by the number of motors the drone requires as our thrust total is 3 900 grams, we will need 975 grams of thrust per motor.

Step 4: Now Browsing motors online or in stores. Each motor should list the amount of thrust it provides. This thrust number must match or exceed the number you came up with in Step 3. In our case we choose Emax with thrust of 1000 g per motor and the specs listed in Hardware section.

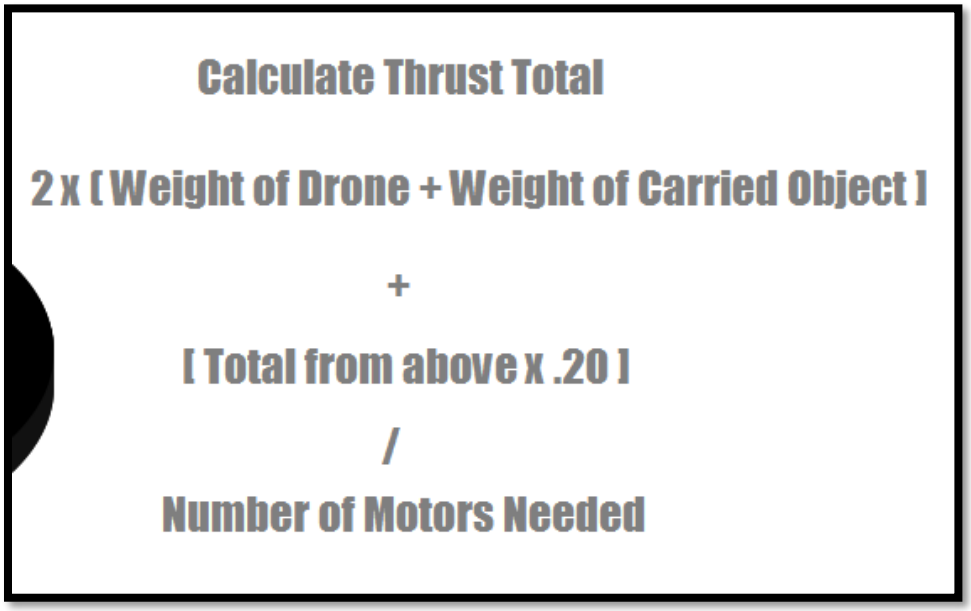

Figure 20: Calculating Motor Thrust

CHAPTER 03: DRONE DYNAMICS AND CONTROL

3.1 Setting Up the Control Problem

Many entry-level vehicles now have advanced control system that enables them to fly autonomously and with little human intervention. Their four propellers are spun precisely to control the quadcopter in 6dof. We'll look at how to create a control system that allows a quadcopter to hover at a fixed flight level. We can add more rotors and call ourselves hexacopter or octocopter.

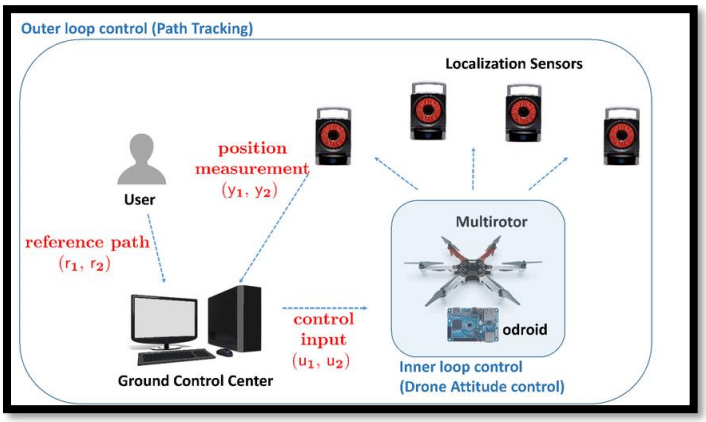

Figure 21: Outer Loop Control

Sensors:

We will begin with the sensors after we have completed the hardware. The sensor is a camera on the bottom. It captures an image at 60 fps and employs an image processing technique known as optical flow to assess how objects move between frames. From this apparent motion, the quadcopter can estimate horizontal motion and speed.

A pressure sensor inside the quadcopter measures altitude indirectly. The air pressure drops fractionally as the drone climbs in altitude. We can use this slight change in pressure to estimate whether the quadcopter's altitude is increasing or decreasing.

An Inertial Measurement Unit is the final sensor (IMU). It is made up of a 3-axis accelerometer for measuring linear acceleration and a 3-axis gyroscope for measuring angular rate. We can estimate the quadcopter's attitude relative to gravity and rotation speed using the IMU and our knowledge of gravity acceleration. We can determine altitude using ultrasound and pressure, and rotational and translational motion using the IMU and camera.

Actuators:

We have four motors, each with its propeller. These motors' configuration and spin direction are the most important things to recognize. These four motors are laid out in an X configuration instead of the plus configuration. The only difference between these two is which motors you send commands to when pitching and rolling the quadcopter. In general, the underlying concepts are the same for both.

The spin direction is the most ingenious part of a quadcopter's motor configuration. Opposing motors spin in the same direction as each other but in the opposite direction as the other pair. It is necessary to make sure that thrust, roll, pitch, and yaw can be commanded independently of each other. That means we can command one motion without affecting the others. Complex fluid dynamics around the drone mean that all motion is coupled in some way. We have a set of sensors that we can use to estimate the state of our quadcopter directly or indirectly. System states include angular position and rates, altitude, and horizontal velocity. The states we estimated depend on the control architecture and what we are trying to accomplish.

Finally, with knowledge of the state of the system and an understanding of what we want our quadcopter to do, we can develop a controller, basically, an algorithm that runs in software that takes in our set point and estimated state and calculates those precise motor commands that will inject the necessary forces and torques.

The first thing we should notice is that this is an underactuated system. We only have four actuators, but we have 6 degrees of freedom - three translational directions, up and down, left, and right, forward, and backward, and three rotational directions, roll, pitch, and yaw. Since we do not have an actuator for every motion, we already know that some directions are uncontrollable at any given time. For example, our quadcopter cannot move left, at least not without first rotating in that direction. The same goes for forwarding and backward motion as well. The under-actuation problem can be solved by developing a control system that couples rotations and thrust to accomplish the overall goals.

How do we generate thrust, roll, pitch, and yaw?

How do we generate thrust, roll, pitch, and yaw with just four motors, and why the spin direction allows us to decouple one motion from the other? A motor produces thrust by spinning a propeller which pushes air down, causing a reaction force that is up. If the motor is placed in a position where the force is applied through the center of gravity of an object, then that object will move in pure translation, with no rotation at all. Moreover, if the force of the thrust is exactly equal to and the opposite of the force of gravity, then the object will hover.

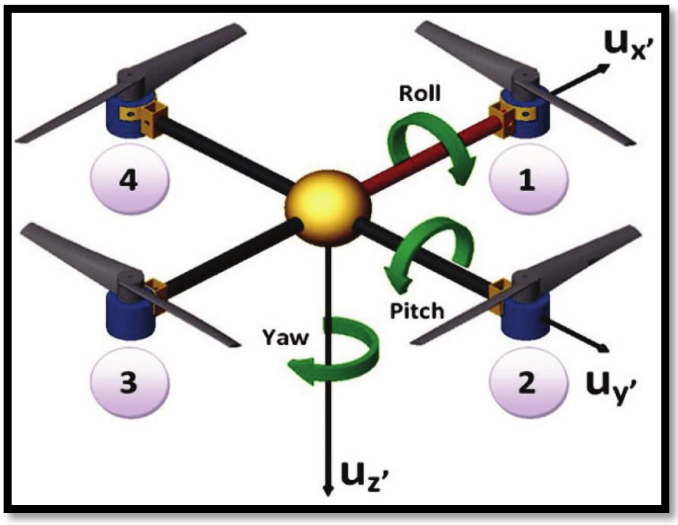

Figure 22: Roll, Pitch, and Yaw

A force at a distance from the center of gravity produces a translational motion and torque or rotating moment around the center of gravity. If our motor is attached to the bar, as it rotates, the torque will stay constant since the distance between the force and center of gravity stays the same. However, the force is no longer always in the opposite direction of gravity; therefore, our bar will begin to move off to the side and fall out of the sky. Now, if there is a counter force on the opposite side of the center of gravity, and each force is half of the force of gravity, then the object will again stay stationary because the torques and forces will cancel each other out.

However, our actuators do not generate force purely when it generates thrust. Since it accomplishes thrust by rotating and torquing a propeller with mass, our actuators are also generating a reaction torque in the opposite direction. Furthermore, if both of our motors spin in the same direction, then the torque is doubled, and our bar would start to rotate.

We could spin the two motors in opposite directions to counter this torque. That would work fine in 2 dimensions, but a bar with only two motors would not be able to generate torques in the third dimension; that is, we would not be able to roll this bar. So, we add a second bar with two more motors to create our quadcopter. With this configuration, we can hover by accelerating each motor until they produce a force 1/4 that of gravity. Moreover, if we have two counter-rotating motors, the torques from spinning the propellers will balance, and the drone will not spin. It does not matter where you put the counter-rotating motors if there are two in one direction and two in the other.

However, quadcopter developers settled on a configuration with opposing motors spinning in the same direction, so there must be a reason for this. And there is. It is because yaw, or the flat spinning motion, interacts with roll and pitch.
Does yaw affect roll and pitch? Suppose the rotating motor pairs are on the same side. In that case, slowing one pair down and increasing the other pair will cause an imbalance of forces about the center of gravity, and the vehicle will either pitch or roll depending on which side the motor pairs are on. However, if we separate the two motors and place them on opposite sides of the drone, the forces will balance each other out. It is why the motor configuration and spin directions are so critical. We can now send commands to the four motors so that the vehicle will yaw but not roll, pitch, or change its thrust.
Similarly, we can look at roll and pitch. To roll, we would decrease one of the lefts/right pairs and increase the other, causing a rolling torque, and to pitch, we would decrease one of the front/back pairs and increase the other, causing a pitching torque. Both motions would not affect yaw since we are moving counter-rotating motors in the same direction, and their yaw torque would continue to cancel each other out. We need to increase or decrease all four motors to change thrust. In this way, roll, pitch, yaw, and thrust are the four directions that we have direct control over. Furthermore, the commands to the motors would be a mix of the amount of thrust, roll, pitch, and yaw required.
As we now know, we can command thrust by setting all four motors to the same speed. Then we can create yaw by increasing two motors spinning in the same direction and decreasing the other two. Pitch is created by increasing or decreasing the front motor pair and then commanding the back pair in the opposite direction. Roll is the same, but with a left/right pair. It is our simple motor mixing algorithm that can convert between the intuitive roll, pitch, yaw, and thrust and the less intuitive motor speeds. If we wanted to maintain altitude, we would increase the thrust so that the vertical component is still canceling out the downward pull of gravity.

3.2 How Do You Get a Drone to Hover?

In the control system the plant is the quadcopter itself. It takes four-motor speeds as inputs, which spin the propellers, generating forces and torques that affect its output state. The output that we want is to have the quadcopter hover at a fixed altitude. So, how do we commend these four motors autonomously so that happens?
Let us assume that rather than do it autonomously, we want to command the quadcopter manually. We have a remote control with toggles that directly control all four motor speeds. The left toggle would control the two front motors, and the right would control the two rear motors. It puts you in the feedback path because you can see where the drone is and then react to its position and attitude by moving the four toggles in specific ways.
If you want to increase thrust, speed up all four motors by moving the two toggles in this direction. Yaw requires that two opposing motors increase speed and the other two decrease speed so that yawing left, for example, would require this kind of toggle motion. Then to roll the vehicle, you would increase one of the left/right pairs and decrease the other. Moreover, you would increase one of the front/back pairs and decrease the other to pitch the vehicle. In this way, the feedback controller, could get the drone to hover by expertly changing the commands to these four motors.
While possible, thinking in terms of motor speed seems hard, and we want to make our job easier, so instead, we can use the motor mixing and command thrust, roll, pitch, and yaw directly. When we command thrust, for example, this single thrust command gets split evenly to all four motors, the yaw command is distributed positive to two motors and negative to the other two, and so on. Now our remote-control toggles are aligned with the intuitive roll, pitch, yaw, and thrust rather

than mind-bending motor speeds. It is the controller configuration that many operators use when manually flying their quadcopters. Nevertheless, it turns out that thinking in terms of roll, pitch, yaw, and thrust is also beneficial when developing an autonomous control system. So, we will keep the motor mixing algorithm and build a feedback control system with it in the loop.

Let us get rid of the human operator and think about how we can accomplish the same thing autonomously. We will start by focusing on the thrust command. Thrust is always in the same direction relative to the drone airframe along the Z-axis of the quadcopter. That means that if the drone is flying level and the z-axis is aligned with the gravity vector, increasing thrust causes the drone to increase its altitude rate, which is how fast it is rising, and decreasing thrust drops the altitude rate. That is straightforward. However, if our drone is flying at a steep pitch or roll angle, increasing thrust is coupled to both the altitude rate and the horizontal speed.

If we are building a controller for a racing drone that is likely to fly at extreme roll and pitch angles, then we need to consider this coupling. However, for our simple hover controller, I will assume that the roll and pitch angles are always small. This way, changing the thrust only meaningfully impacts altitude rate and nothing else. With this information, we can start our control design.

Let us build a controller that uses thrust to adjust the altitude. If we can measure the drone altitude, we can feed it back to compare it to an altitude reference. The resulting error is then fed into an altitude controller that uses that information to determine how to increase or decrease thrust. For now, we can think of this controller as some form of a PID controller. With this controller, if the drone is hovering too low, the altitude error will be positive, and the thrust command will increase, causing all four motors to speed up simultaneously, and the drone will rise. If the drone is too high, all four motors will slow down, so the drone will descend.

We have developed our simple altitude controller, which will hover our quadcopter. Moreover, you know that is not the case because there are disturbances, like wind gusts, that will induce a little roll or pitch into the system. When that happens, the thrust will adjust altitude and create horizontal motion, and the drone will start to move away from you. Furthermore, what good is a hover controller that maintains altitude but requires you to chase after it or crashes into walls? That is hardly covering.

We can command thrust, roll, pitch, and yaw independently; that is, we can command one action without affecting the others. Knowing this, we can create three more feedback controllers, one for roll, pitch, and yaw, the same way we did for thrust. Now, the output of the plant is more than just altitude. We will also need to measure or estimate the roll, pitch, and yaw angles. We have four independent or decoupled controllers, one for thrust and an altitude controller, since we are claiming small roll and pitch angles. And then three controllers trying to maintain 0 degrees in roll, pitch, and yaw, respectively. It should maintain altitude and keep the quadcopter facing forward and level with the ground.

It is a better hover controller than our first one. However, again, it is still not perfect. The wind might initially introduce a little roll angle, but our roll controller will remove that error and get the drone back to level flight. However, during the roll, the thrust vector is not straight up and down; therefore, the drone will have moved horizontally. Then another gust comes and causes another roll or pitch error, and the drone walks away a little more. So even though the drone will not run away continuously like our first controller, this controller will still allow it to meander away slowly.

Improving this control system to do just that. What roll and pitch are doing while hovering? It is tempting to say that both angles should be zero, which is how we set up this current version of the controller. However, they may need to be non-zero to hover. For example, if we want to hover in

a strong constant wind, the drone will have to lean into the wind at some angle to maintain its ground position. So rather than specifying that we want level flight, we need a ground position controller, which will recognize when the drone is wandering off and make the necessary corrections to bring it back to the reference points X and Y.

Nevertheless, just because we do not want to pick specific roll and pitch angles does not mean we do not need the roll and pitch controllers. So, our control system needs to couple position errors with roll and pitch. It is a complicated sounding set of maneuvers, but luckily the resulting controller is simple to understand. Our position controller takes the position error as an input and outputs roll and pitch angles. Our roll and pitch controllers are trying to follow these reference angles. So instead of us, as the designer, picking roll and pitch reference angles, we are letting the position controller create them for us. In this way, the position controller is the outer loop, generating the reference commands for the inner loop roll and pitch controllers. These are cascaded loops.

The measured yaw angle also feeds into the position controller. The reason is that the X, Y position error is relative to the ground, or the world reference frame, whereas roll and pitch are relative to the drone's body. Therefore, the pitch does not always move the drone in the X world direction, and the roll does not always move the drone in the Y world direction. It depends on how the drone is rotated or its yaw angle. So, if we need to move the drone to a specific spot in the room, it needs to know yaw to determine whether roll, pitch, or some combination of two is needed to achieve that. Our position controller uses yaw to convert between the world X, Y, & the body X, Y frame.

Let us say the quadcopter is flying level at the correct altitude but a little too far left of where it took off. It will result in a position error that feeds into the position controller. The proportional part of the controller will multiply that error by a constant, requesting that the drone roll to the right. The roll controller will see a roll error because the drone is still level and request a roll torque. It will play through the motor mixing algorithm and request that the motors on the left side of the drone speed up and the motors on the right side slow down. It will roll the drone to the commanded angle. Now the drone will begin to move to the right, but since the vertical component of thrust is slightly smaller when rolled, the drone will also start to lose altitude. The altitude controller will sense this error and increase the thrust command accordingly.

As the drone continues moving right, the position error is dropping; therefore, the requested roll angle through the proportional path is also dropping, bringing the drone back to level. However, if the drone is traveling too fast, then some horizontal momentum in the direction of travel needs to be removed. It is where the derivative term in the position controller is useful. It can sense that the drone is closing in on the reference position quickly and start to remove roll earlier, even causing it to roll in the opposite direction to slow the drone down and stop right where we want it.

There are two glaring obstacles to creating and tuning it. First, this requires estimating the yaw, roll, pitch, altitude, and XY position states. These are the signals we feedback. Furthermore, second, we need to tune six PID controllers that all interact with each other, specifically four directly coupled in cascaded loops. We will handle the first problem by combining the measurements from the four sensors we have and, in some cases, using a model and a Kalman filter to estimate each feedback state. If we look at the controller portion of this feedback system, the outer loop XY position controller generates the reference pitch and roll angles for the inner loop pitch/roll controller. There are also the Yaw and altitude controllers, which feed into the motor mixing algorithm.

3.3 How to Build the Flight Code

We need to code the control logic in a way that we can put it on the quadcopter. We will call this the flight code. And second, we will need to tune and tweak the flight code until the hover performance is what we're looking for. Flight control software is just a small part of the overall flight code that will exist on the quadcopter. There is also code to operate and interface with the sensors and process their measurements. There is code to manage the batteries, the Bluetooth interface, and LEDs. There is also a code to manage the motors speeds, and so on.

One option to implement the flight controller would be to write the C code by hand and then compile the entire flight code with your changes to the flight controller and finally load the compiled code to the quadcopter. This is a perfectly reasonable approach to creating flight code, but it has a few drawbacks when developing feedback control software. With this method, we don't have an easy way to tune the controllers except by tweaking them on the hardware.

We have a pretty good start on the flight control system from the architecture that we developed in the last video, but it's not all the software that we need to write. This is only the controller part of the control system. For example, the drone has a sensor that measures air pressure, and this air pressure reading is what is passed into our flight control system. However, we're not trying to control the drone to a particular pressure, we're trying to control an altitude. Therefore, there is additional logic needed to convert all the measured states coming from the sensors into the control states needed by the controller. We'll call this block the state estimator.

In addition to the state estimator and the controllers, there's also logic we must write that determines whether to set the shutdown flag. We could leave this code out, but then we'd run the risk of the drone flying away or causing harm to nearby observers. We could check for things like the rotation rate of the drone is above some threshold or the position or altitude is outside of some boundary set. Creating this code is relatively easy and can save us from damaging the hardware or other people.

Now, we need to think about data logging. All the firmware that exists on the quadcopter records data that we have access to during and after the flight. Since the quadcopter doesn't know about the software that we've written, we need to make sure we have a data logging set for the variables that we're interested in. You can now see that we have the XY position outer loop controller feeding into the roll pitch inner loop controller. And independent of those, we have the yaw and altitude controllers. Overall, six PID controllers work together to control the position and orientation of the quadcopter.

If we look at just the altitude controller, which is set up as proportional and derivative, you'll see that it might be implemented slightly differently than you're used to. Rather than a single altitude error term feeding into the P and D branches, the P gain is applied to the altitude error derived from the ultrasound, whereas the D gain is applied to the vertical rate measurement directly from the gyro. This way, we don't have to take a derivative of a noisy altitude signal, we already have a derivative from a different sensor. One that is less noisy than taking a derivative of the ultrasound sensor. The output of these PID controllers is force and torque commands which all feed into the mixing algorithm. This produces the required thrust per motor and then that thrust command is converted into a motor speed command through this block.

There are two steps involved in taking the raw sensor measurements and generating the estimated states. First, we process the measurements and then we blend them with filters to estimate the control states. Let's look at the details of the sensor processing block. This looks daunting at first, but the underlying concept is simple. Along the top, the acceleration and gyro data are calibrated by subtracting off the bias that has been previously determined. By removing the bias, zero

acceleration and zero angular rates should result in a zero measurement. The next step is to rotate the measurements from the sensor reference frame to the body frame. And lastly, filter the data through a low pass filter to remove the high-frequency noise.
Similarly, the ultrasound sensor has its own bias removed. And the optical flow data just has a pass/fail criterion. If the optical flow sensor has a valid position estimate, and we want to use that estimate, then this block sets the validity flag. There's always more sensor processing that can be done, but we'll see shortly that our drone hovers quite nicely with just this simple amount of bias removal, coordinate transformation, and filtering.
Now that we have filtered and calibrated data, we can begin the task of combining measurements to estimate the states we need for the controllers. The two orange blocks are used to estimate altitude and XY position. If you look inside these blocks, you'll see that each of them uses a Kalman filter to blend the measurements and a prediction of how we think the system is supposed to behave to come up with an optimal estimation.
The other non-orange block estimates roll, pitch, and yaw and it does it using a complementary filter instead of a Kalman filter. A complementary filter is a simple way to blend measurements from two sensors together and it works well for this system. In the description of this video, I also linked a complimentary video on complementary controllers that I posted to my channel if you are interested. With the state estimation and controller subsystems described, we can now move on to the other important, but less flashy subsystems.
There is the logging subsystem that is saving a bunch of signals like the motor commands and position references to .mat files. These are the values that we can download and plot after the flight. We also have the crash predictor flag subsystem. The logic here is just checking the sensed variables like position and angular velocity for outliers. When that happens, it sets the flag that shuts down the quadcopter. This is where you could add additional fault protection logic if you wanted to.
There is also the sensor data group, which is just pulling individual sensor values off the sensor bus so that we have access to them elsewhere in the code. And finally, there is the landing logic block. This block will overwrite the external reference commands with landing commands if the landing flag is set. Once again, I'll remove the switch and landing portion to simplify the logic since we don't want to execute a precision landing.
I must change one other thing here because the reference block from the top level of the model isn't part of the auto coder that runs on the drone. So, it won't get loaded onto the drone and executed. But we can move this logic into the flight code right now. Since I know I just want the drone to hover, I'm going to hardcode the reference values in this block. There we go. This will keep the drone at an X Y position of 0 and 0 and an altitude of -0.7 meters. Remember, the z-axis is down in the drone reference frame, so this is an altitude of 0.7 meters up. This is no longer the landing logic, but instead is the block that is generating our reference commands. And we don't need these inputs anymore since the reference commands are now hardcoded values.
That completes the very quick walkthrough of the entire flight control software that is in this quadcopter model. And you should now sort of understand how each of these subsystems contributes to getting our quadcopter to hover safely, whether it's the sensor processing and filtering or the various feedback controllers or even the logic that sets the stop flag. We need it all to have a successful control system. I haven't yet spoken about how to tune all these controllers, that will be a future video in this series. For now, we can rely on the fact that the default gains delivered with the model are already tuned well.

3.4 PID Control and Tuning

For controlling a quadcopter, we used the feedback PID control whose concept is illustrated below and explained in detail above in chapter 3. First, we compare the quadcopter attitude that is gyro angles with the input receiver signals and their difference is termed as error that is needed to multiply with P, I and D. For getting values of PID we tune our quadcopter by using a technique in which first we target D controller, we go on increasing its value until our quadcopter starts shuddering then reducing its value until it gets normalized. After this value of P gain start incrementing from 0.3 with steps of 0.2 until it gets stable flight.

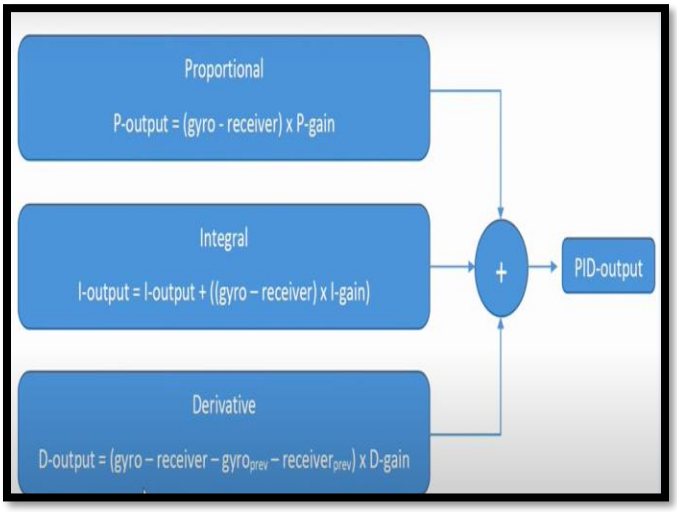

Figure 23: PID Control & Tuning

3.5 Sensor Fusion

Sensor fusion is an integral part of the design of autonomous systems; things like self-driving cars, RADAR tracking stations, and the Internet of Things all rely on sensor fusion of one sort or another. "What is sensor fusion and how does it help in the design of autonomous systems?" This will be a good first video if you're new to the topic because we're going to go over a broad definition of what it is and then show a few scenarios that illustrate the various ways sensor fusion is used.

The high-level definition is that sensor fusion is combining two or more data sources in a way that generates a better understanding of the system. "Better" here refers to the solution being more consistent over time, more accurate, and more dependable than it would be with a single data source. For the most part, we can think of data as coming from sensors, and what they're measuring provides the understanding of the system; for example, things like how fast it's accelerating, or the distance to some object. But a data source could also be a mathematical model, because as designers we have some knowledge of the physical world, and we can encode that knowledge into the fusion algorithm to improve the measurements from the sensors.

Sense refers to directly measuring the environment with sensors. It's collecting information from the system and the external world. For a self-driving car, the sensor suite might include RADAR, LIDAR, visible cameras, and many more. But simply gathering data with sensors isn't good enough, because the system needs to be able to interpret the data and turn it into something that can be understood and acted on by the autonomous system. This is the role of the perceive step: to make sense of, well, the sensed data. For example, let's say this is an image from a vehicle's camera sensor. The car must ultimately interpret the blob of pixels as a road with lane lines, and that there's something off to the side that could be a pedestrian about to cross the street or a stationary mailbox. This level of understanding is critical for the system to determine what to do next. This is the planning step, where it figures out what it would like to do and finds a path to get there. Lastly, the system calculates the best actions that get the system to follow that path. This last step is what the controller and the control system is doing.

So where does sensor fusion come in? Well, it sorts of straddles sense and perceive, as it has a hand in both capabilities. It's the process of taking the multiple sensor measurements, combining

them, and mixing in additional information from mathematical models with the goal of having a better understanding of the world with which the system can use to plan and act. The four different ways that sensor fusion can help us do a better job at localization and positioning of our own system, as well as detecting and tracking other objects.

As a simple example, let's take a single accelerometer and place it on a stationary table so that it's only measuring the acceleration due to gravity. If this was a perfect sensor, the output would read a constant 9.81 m/s^2. However, the actual measurement will be noisy - how noisy depends on the quality of the sensor. This is unpredictable noise so we can't get rid of it through calibration, but we can reduce the overall noise in the signal if we add a second accelerometer and average the two readings. If the noise isn't correlated across the sensors, fusing them together like this reduces the combined noise by a factor of the square root of the number of sensors. So, four identical sensors fused together will have half the noise of one. In this case, all that makes up this very simple sensor fusion algorithm is an averaging function.

We can also reduce noise by combining measurements from two or more different types of sensors, and this can help if we must deal with correlated noise sources. For example, let's say we're trying to measure the direction your phone is facing relative to north. We could use the phone magnetometer to measure the angle from magnetic north, easy. However, just like with the accelerometer, this sensor measurement will be noisy. And if we want to reduce that noise, then we may be tempted to add a second magnetometer. However, at least some contribution of noise is coming from the moving magnetic fields created by the electronics within the phone itself. This means that every magnetometer will be affected by this correlated noise source and so averaging the sensors won't remove it.

Two ways to solve this problem are to simply move the sensors away from the corrupting magnetic fields—hard to do with a phone—or to filter the measurement through some form of a low-pass filter, which would add lag and make the measurement less responsive. But another option is to fuse the magnetometer with an angular rate sensor, a gyro. The gyro will be noisy as well, but by using two different sensor types, we're reducing the likelihood that the noise is correlated and so they can be used to calibrate each other. The basic gist is that if the magnetometer measures a change in the magnetic field, the gyro can be used to confirm if that rotation came from the phone physically moving or if it's just from noise.

There are several different sensor fusion algorithms that can accomplish this blending, but a Kalman filter is probably one of the more common methods. The interesting thing about Kalman filters is that a mathematical model of the system is already built into the filter. So, you're getting the benefit of fusing together sensor measurements and your knowledge of the physical world.

The second benefit of sensor fusion is that it can increase the reliability of the measurements. An obvious example is that if we have two identical sensors fused together, like we had with the averaged accelerometers, then we have a backup in case one fails. We lose quality if one sensor fails, but at least we don't lose the whole measurement. We can also add a third sensor into the mix and the fusion algorithm could vote out the data of any single sensor that is producing a measurement that differs from the other two.

An example here could be using three pitot tubes to have a reliable measure of an aircraft's air speed. If one breaks or reads incorrectly, then the airspeed is still known using the other two. So duplicating sensors is an effective way to increase reliability; however, we must be careful of single failure modes that affect all the sensors at the same time. An aircraft that flies through freezing rain might find that all three pitot tubes freeze up and no amount of voting or sensor fusion will save the measurement.

Again, this is where fusing together sensors that measure different quantities can help the situation. The aircraft could be set up to supplement the air speed measurements from the pitot tubes with an airspeed estimate using GPS and atmospheric wind models. In this case, air speed can still be estimated when the primary sensor suite is unavailable. Again, quality may be reduced, but the airspeed can still be determined, which is important for the safety of the aircraft.

Losing a sensor doesn't always mean the sensor failed. It could mean that the quantity they're measuring drops out momentarily. For example, take a RADAR system that is tracking the location of a small boat on the ocean. The RADAR station is sending out a radio signal which reflects off the boat and back and the round-trip travel time, the doppler shift of the signal, and the azimuth and elevation of the tracking station are all combined to estimate the location and range rate of the boat. However, if a larger cargo ship gets between the RADAR station and the smaller boat, then the measurement will shift instantly to the location and range rate of that blocking object.

So, in this case, we don't need an alternate sensor type or a secondary RADAR tracking station to help when the measurement drops out because we can use a model of the physical world. An algorithm could develop a speed and heading model of the object that's being tracked. And then when the object is out of the RADAR line of sight, the model can take over and make predictions. This of course only works when the object you're tracking is relatively predictable and you don't have to rely on your predictions long term. Which is pretty much the case for slow-moving ships.

The third benefit of sensor fusion is that we can use it to estimate unmeasured states. Now, it's important to recognize that unmeasured doesn't mean unmeasurable; it just means that the system doesn't have a sensor that can directly measure the state we're interested in.

For example, a visible camera can't measure the distance to an object in its field of view. A large object far away can take up the same number of pixels as a small but close object. However, we can add a second optical sensor and through sensor fusion, extract three-dimensional information. The fusion algorithm would compare the scene from the two different angles and measure the relative distances between the objects in the two images. So, in this way, these two sensors can't measure distance individually, but they can when combined. In the next video, we'll expand on this concept of using sensors to estimate unmeasured states by showing how we can estimate position using an accelerometer and a gyro.

For now, though, I want to move onto the last benefit I'm going to cover in this video. Sensor fusion can be used to increase the coverage area. Let's imagine the short-range ultrasonic sensors on a car which are used for parking assist. These are the sensors that are measuring the distance to nearby objects, like other parked cars and the curb, to let you know when you're close to impact. Each individual sensor may only have a range of a few feet and a narrow field of view. Therefore, if the car needs to have full coverage on all four sides, additional sensors need to be added and the measurements fused together to produce a larger total field of view. Now, more than likely, these measurements won't be averaged or combined mathematically in any way since it's usually helpful to know which sensor is registering an object so that you have an idea of where that object is relative to the car. But the algorithm that pulls all these sensors together into one coherent system is still a form of sensor fusion.

So hopefully you can start to see that there are a lot of different ways to do sensor fusion, and even though the methods don't necessarily share common algorithms or even have the same design objective, the general idea behind them is ubiquitous: Use multiple data sources to improve measurement quality, reliability, and coverage, as well as be able to estimate states that aren't measured directly. The fact that sensor fusion has this broad appeal across completely different types of autonomous systems is what makes it an interesting and rewarding topic to learn.

CHAPTER 04: FLIGHT CONTROLLER

Now, let us design a flight controller. At first, we implemented a low-level flight controller using Arduino with limited functionalities. We proceed on STM32 for flights that meet practical challenges and make our quadcopter able to be designed for automation using the VINS algorithm.

4.1 Flight Controller using Arduino

We can divide our flight controller into three stages from the basics of auto-leveling to a more efficient autonomous flight controller.

Auto leveling of the quadcopter

Auto-leveling means whenever our drone disrupts a wind gust or obstacle interference, it instantly balances itself. For doing this, our first target was to perform inertial navigation using MPU-6050.

Inertial Navigation:

MPU-6050 consists of the following two sensors:

- 3-axis accelerometer: measures the accelerations along its axes
- 3-axis gyroscope: measures the rotational velocity around its axes

Accelerometers

Accelerometers sense all the accelerations applied to them, which are then converted into a voltage and later translated into a binary number that autopilot can understand.

Working principle

The basic underlying working principle of an accelerometer is such as a dumped mass on a spring. When this device experiences acceleration, the mass gets displaced till the spring can easily move the mass, with the same rate equal to the acceleration it sensed. Then this displacement value is used to measure the give the acceleration.

Gyroscopes

Gyros measure the rotational velocity around their axis. Measuring the rotational velocity is of primary importance, as it can be integrated to estimate the actual tilt angle.

Working principle

The working principle of a gyroscope is based on gravity. It is explained as the product of angular momentum experienced by the torque on a disc to produce a gyroscopic precession in the spinning

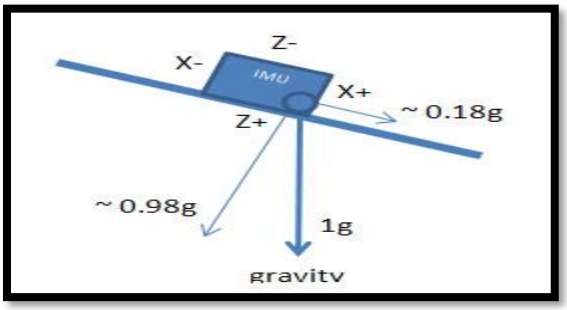

Figure 24: IMU Reading

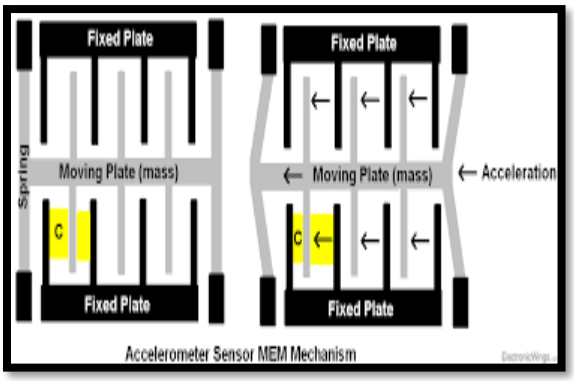

Figure 25: Accelerometer MEM Mechanism

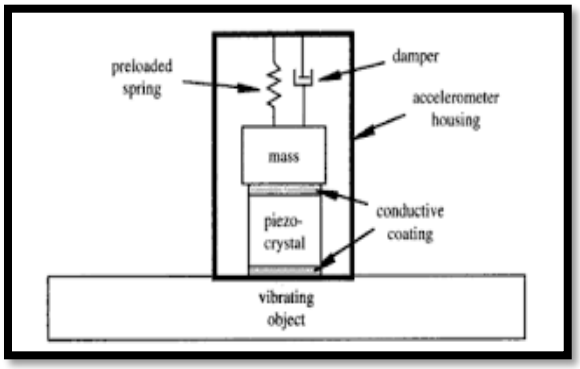

Figure 26: Gyroscope Mechanism

wheel. We know that the rotating object possesses angular momentum, which needs to be conserved. It is done because when there is any change in the axis of rotation, there will be a change in the orientation, which changes the angular momentum. Therefore, it can be said that the working principle of a gyroscope is based on the conservation of angular momentum. (MPU Datasheet for Gyroscope in Appendix A1). Here we have selected Full-Scale Range = 1; thus, our gyro gives an output equal to 65.5 for rotation of 1°/s.

$$angle(°) = \frac{raw\ gyro\ output}{65.5}$$

However, the above formula is valid only if getting gyro output after every 1 sec. But this is not the case as the refresh rate for our gyroscope is 250 Hz thus gyro adds its values every 4ms.

$$angle(°) = \frac{raw_{gyro}\ output}{65.5} * 250 = raw_{gyro}\ output * 0.0000611$$

Calculate the traveled pitch angle and add this to the angle pitch variable.

Computation of roll and pitch using gyroscope:

$$angle_{pitch} += gyro_{pitch} * 0.0000611$$
$$angle_{roll} += gyro_{roll} * 0.0000611$$

Simply from the above two lines, if our drone continuously rotates, add gyro output values every 4ms. Also, note that if the direction of rotation changes, then gyro gives negative values as output.

Angle inclination problem

The above code does not tackle the situation when our drone is inclined and starts rotation along the yaw axis, which is the value of pitch and roll that remains unchanged during yaw rotation. To tackle this problem, we map a relation between yaw rotation and pitch value.

We get a sinusoidal relation; thus, the IMU has yawed transfer the roll angle to the pitch angle and then that pitch angle to roll angle as there lies inverse sinusoid relation of roll and pitch, thus sign change.

$$angle_{pitch} -= angle_{roll} * \sin(gyro_{yaw} * 0.000001066)$$
$$angle_{roll} += angle_{pitch} * \sin(gyro_{yaw} * 0.000001066)$$

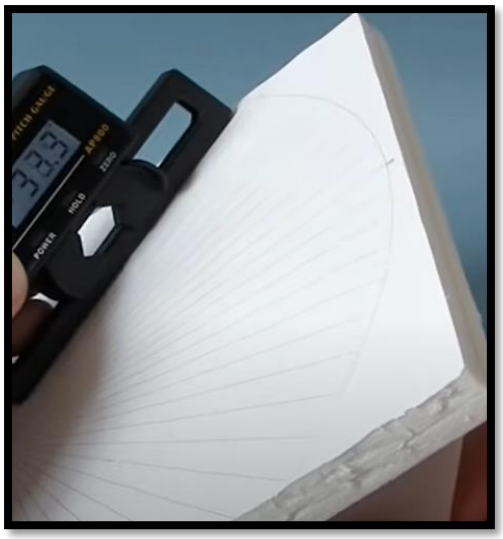

Figure 27: Angle Inclination Problem

Computation of yaw using gyroscope

Here it is important to note that we are not computing yaw angle but simply getting rotation as several degrees per second.

$$gyro_{yaw}input = (gyro_{yaw}input * 0.7) + ((\frac{gyro_{yaw}}{65.5}) * 0.3)$$

The **drawback** of using a gyroscope:
It drifts a little over time. Furthermore, when the quadcopter remains at rest, it produces a bias.

For our case, we selected AFS-SEL=1; thus, 1g accelerometer gives an output equal to 4096.

Calculate the total accelerometer vector:

$$acc\ total\ vector = sqrt\left((acc_x * acc_x) + \left(acc_y * acc_y\right) + (acc_z * acc_z)\right);$$

As gravitational acceleration is projected into three-dimensional spring system of accelerometer, we can find roll and pitch by its resolution along their axis.

Calculate the pitch angle:

if(abs(acc_y) < acc_total_vector)
*{angle_pitch_acc = asin((float)acc_y/acc_total_vector) * 57.296}*

Calculate the roll angle:

if(abs(acc_x) < acc_total_vector)
*{angle_roll_acc = asin((float)acc_x/acc_total_vector) * -57.296}*

The **drawback** of using an accelerometer:
It produces noise during motion; thus, only average is usable.

Complementary filter

How to get an output that produces a better result of angles measurement? We have seen that the gyro is not useful at rest conditions while the accelerometer is useless during motion; thus, we take each value in their respective suitable periods.

Using a complementary filter, we pass accelerometer value by a low pass filter that gives incoming new data less weightage than previous data, resulting in a reduction of noise but has a drawback of less response time. While gyro meter value is passed through a high pass filter, meaning that when it is in rest mode or slow-motion mode, its value would be ignored.

This all done by following code of line:

$$\text{angle}_{\text{pitch}} = \text{angle}_{\text{pitch}} * 0.9996 + \text{angle}_{\text{pitch}}\text{acc} * 0.0004;$$

$$\text{angle}_{\text{roll}} = \text{angle}_{\text{roll}} * 0.9996 + \text{angle}_{\text{roll}}\text{acc} * 0.0004;$$

Transmitter and Receiver:

Until now we have measured angles of our quadcopter, now to order our quadcopter to get certain angles we need to transmit the signal to the receiver through which ESCs get pulses varying from 1000 to 2000 microsecond as we change joystick position, to complete understand its working lets dive deep into it.

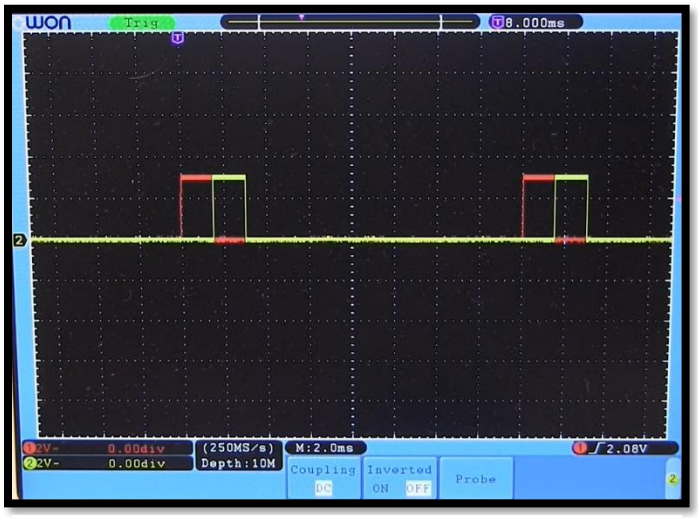

Figure 28: PWM

To get stabilized our drone update its value 250 times per second using IMU but the problem is that receivers have refresh rate of only about 62 Hz which means that we cannot wait for receiver to update angle values thus here we need important functionality of Arduino ATMEGA 328P that is external change interrupts.

From fig below we have four PCINT pins available at pins 8,9,10 and 11 which are then connected to signal pins of each of the four receivers.

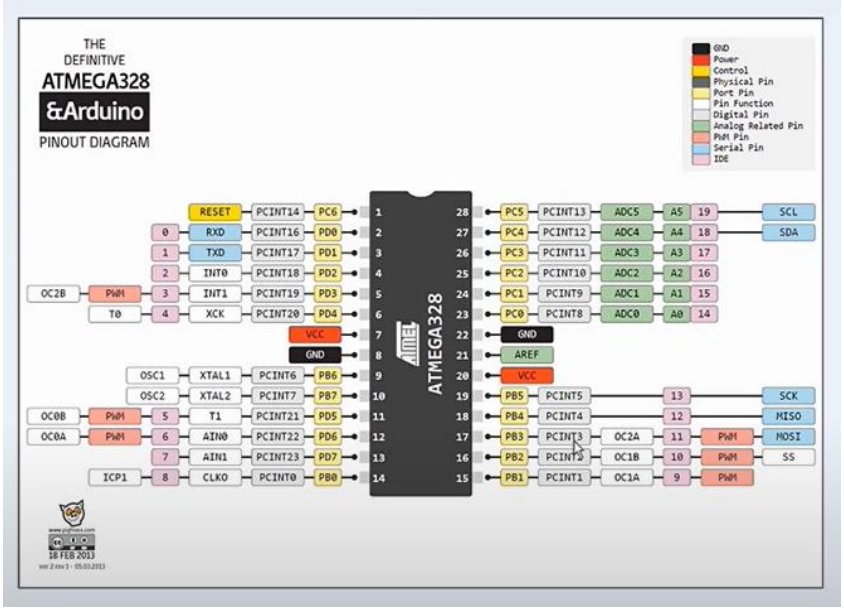

Figure 29: PINOUT DIAGRAM

To understand the functionality of subroutines, consider the PCICR (register)

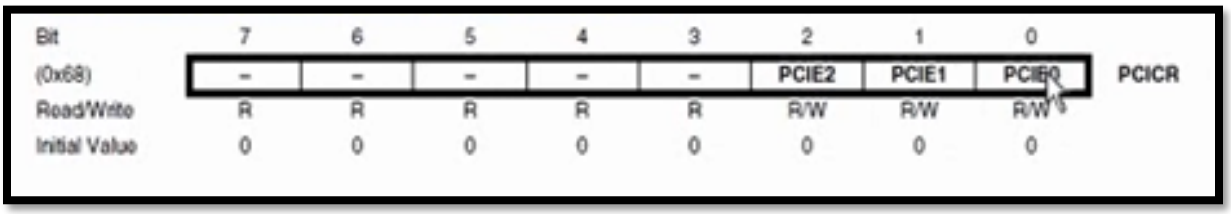

Figure 30: Pin Change Interrupt Control Register (PCICR)

PCICR |= (1 << PCIE0); //Set PCIE0 to enable PCMSK0 scan
If we set PCIE 0 pin then it means first seven interrupt pins that are PCINT[7:0] would cause an interrupt if any of them pin set on and this is determined by PCIMSK0 register.

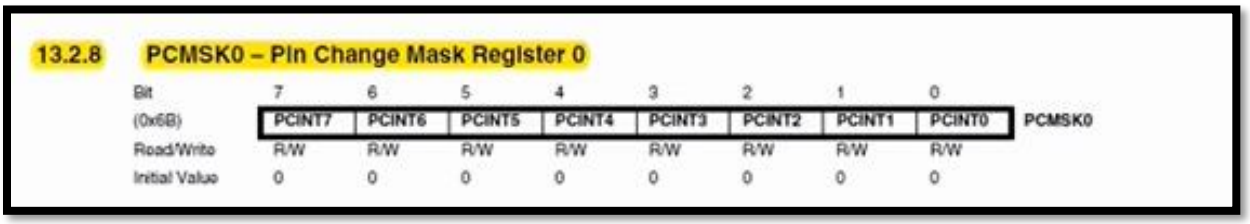

Figure 31: Pin Change Mask Register 0 (PCMSK0)

PCMSK0 |= (1 << PCINT0);
PCMSK0 |= (1 << PCINT1);
PCMSK0 |= (1 << PCINT2);
PCMSK0 |= (1 << PCINT3);
Simply moving 1 to each of four bits of register and these would only be interrupted once any of the digital pins 8 to 11 which are basically the bits of port B registers are triggered.

Here PINB is register having pins of port B that are connected to four receiver signal pins. From 4 channels of transmitter, let us assume that first channel is being triggered and measured pulse width using following code lines of interrupt subroutine:

```
ISR(PCINT0_vect){current_time = micros();
//Channel 1
if(PINB & B00000001){
if(last_channel_1 == 0){
last_channel_1 = 1;
timer_1 = current_time;}}
```

If the channel for receiver 1 is set, then we check for how much time the digital pin remained set on. As when it gets turn off following commands would run that would measure the total length of receiver signal:

```
else if(last_channel_1 == 1){ //Input 8 is not high and changed from 1 to 0.
last_channel_1 = 0; //Remember current input state.
receiver_input[1] = current_time - timer_1;} //Channel 1 is current_time - timer_1.
```

THE ESCs:

Once the receiver signal is received the next step is to transmit these signals towards motor such that we get speed variation for achieving correct angles. But it is not as simple we cannot transmit signals directly to motors but us 4 ESC's that perform this job. Here lets first explore its importance and then working.

Bit	7	6	5	4	3	2	1	0	
0x04 (0x24)	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 33: The Port B Data Direction Register (DDRB)

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 32: The Port B Data Register (PORTB)

The ESC's have refresh rate of about 250 Hz as gyro angular data is updating its pulse after every 4ms and as we know that pulse width foe ESC's range from 1ms to 2ms corresponding to extreme joystick position of any of the four motions.

Drawback of using Arduino UNO as microcontroller:

Program loop at 250 Hz gives us following time specifications:

- Total time available for measuring each updated pulse = 4 ms
- Reading gyro angular data takes almost 300 us
- PID calculations take about 600us

Thus, it means we have about 3ms available to measure the pulse width thus until now it all goes fine with Arduino UNO but again things are not that simple. At a time four subroutines may need to be called and the programming level for Arduino having been user friendly like having if-else statements create problem as it increases compilation time many times and make whole calculations nearly to impossible for 16 MHz speed microcontroller.

4.2 Flight Controller using STM32

To get more efficient and accurate result we make transition from Arduino to STM 32 which has both memory and speed almost 4 times more than that and it uses register-level programming which in effect not only increase functionality but also speed up computation time. Here again we can't wait for receiver as its refresh rate is about 12ms thus we would use interrupt service routine, but calculation method is not same as it will use input capture mode method to perform this task. As explained earlier that pulse range from 1ms to 2ms we only need to find the rising and falling edge time of pulse and STM perform this operation using its internal clock timer where it captures the counter value to store in register as below

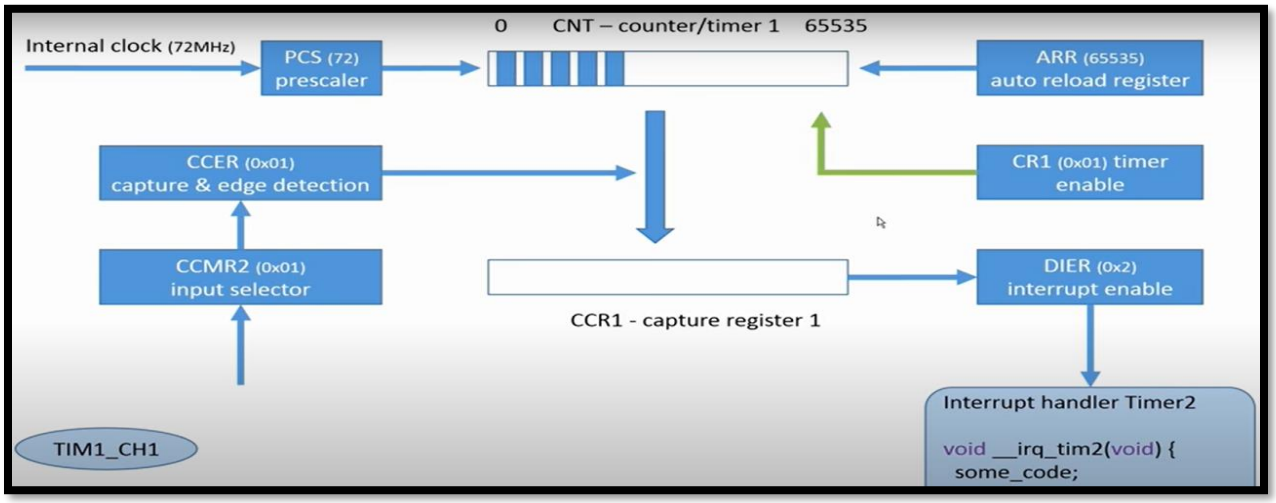

Figure 34: Flight Controller Loop

Here when Timer_1 signal that is connected to receiver is activated CCER (the edge detection register activates the capture register that in response capture the value of counter, after that it call subroutine where it toggles the bit of CCER register to make it enable to detect the falling edge net time, thus after getting both values their subtraction provides us the required result.

Here important thing to notice is that the whole calculation I performed just by toggling the bit of register and for that STM uses pointer float constants where it stores the address of register by constant and then by calling that constant address it dives into that location to manipulate the bits.

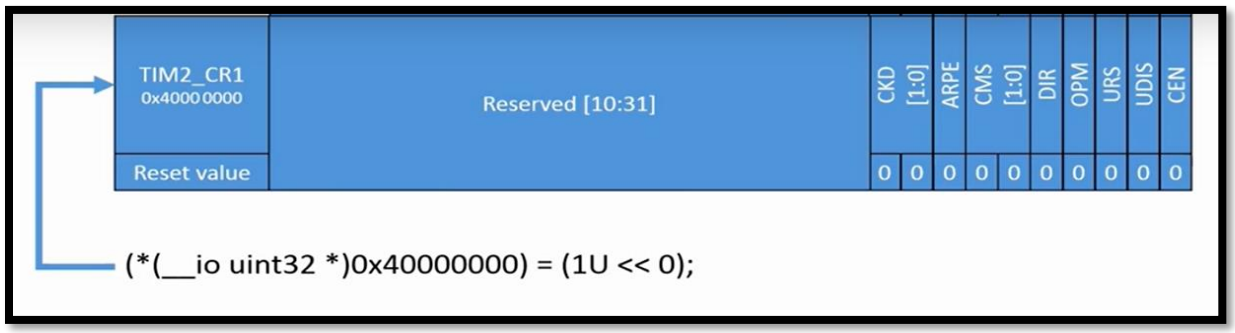

Figure 35: Pointer for Indirect Addressing

Electronic Speed Control (ESC)

After getting values for throttle, roll, pitch, and yaw by measuring the pulses width coming through receiver and then multiplying each with PID gain values we found out the values for ESCs as ESCs must transmit these signals to control the speed of motor by switching each of six relay switches.

$$if\ (throttle > 1800)\ throttle = 1800;$$
$$esc_1 = throttle - pid_{output\ pitch} + pid_{output\ roll} - pid_{outputyaw};\ //\ front\ right\ motor$$
$$esc_2 = throttle + pid_{output\ pitch} + pid_{output\ roll} + pid_{outputyaw};\ //\ rear\ right\ motor$$
$$esc_3 = throttle + pid_{output\ pitch} - pid_{output\ roll} - pid_{outputyaw};\ //\ front\ left\ motor$$
$$esc_4 = throttle - pid_{output\ pitch} - pid_{output\ roll} + pid_{outputyaw};\ //\ rear\ left\ motor$$

It is also important to consider the sign for each motion for all ESC's having concept of dynamics. Here, in the main loop we have measured the pulse length for each ESC which is used in the following circuit to compare with counter value for making falling edge.

In CCR1 register value of pulse width is kept which we measured above thus now to toggle bit of Timer_CH1 we compare its value with counter value and when this AND gate dissatisfy condition, we get signal low and thus the falling edge. This measurement from ESC is then converted into the speed of each motor.

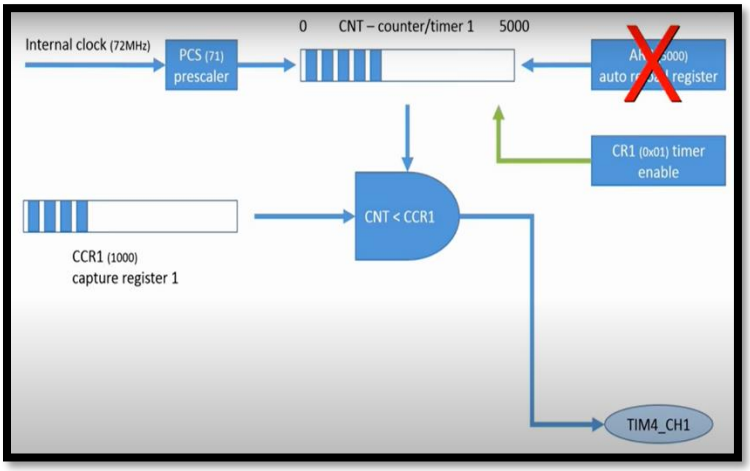

Figure 36: Input Capture Mode

Motor Speed Control

As BLDC motor as explained at start has six poles with each pole either attract or repel the magnet of rotor resulting in rotor movement and this attraction or repulsion is due to the current value that is passed through motor solenoid winding. To control those six poles, we have six different paths each with specific direction through two switches from total of six switches. From each channel of transmitter, the width of pulse for each corresponding motion basically determines switching speed of these paths with the help of ESCs.

This same process is repeated for each of four motor and by different combination of speed of each of motor basically we achieve all kinds of quadcopter motion.

4.3 Autonomous Flight Controller using STM32

STM32 version 2 consists of GPS, barometer, and Telemetry system implementation. The GPS will be used for heading lock and GPS hold function while the barometer will be used for altitude hold programming. The telemetry system that we will be implementing will be able to send the data of different variables such as IMU-angles, satellite data etc., to the ground station.

GPS Hold Function

The GPS hold function involves following steps:

- Getting the raw data from the GPS

- Filtering the data to get only the desired variables such as latitude, longitude, fix-type, and number of satellites that the GPS has locked.
- The values of variables from above data will be then used to implement PD-controller for GPS-hold function. The only P controller can only be used. It will be high if the error between desired coordinates and actual coordinates is larger and will be low vice versa. But quadrotor will not stop and continue to oscillate around the GPS hold position. To give the drone a frictional force or drag force to stop at GPS-Hold position, the D-controller will be used. The D-controller along with the P-controller will eventually make it to stop at the hold position. i.e.,

$$P_{out} = (GPS_{actual} - GPS_{desired}) * P_{gain}$$
$$P_{out} = (GPS_{actual} - GPS_{desired}) * D_{gain}$$

There are other problems that are needed to be addressed as follows:

- Refresh Rate
- Virtual Heading
- Inverting PD-gain

Refresh Rate:

This is the main issue while integrating GPS with quadcopter's flight controller. Since the GPS refresh rate is 5 Hz. The overall refresh rate of controller is 250 Hz. So, we need to increase the refresh rate as it can have a huge impact on overall performance of PD-control. What happens is that we have used long floating point type for GPS coordinates. Thus, we have six digits after degrees in latitude or longitude.

As above equator the change in one degree we have 111 km change in location. Thus, performing following calculations for the gain:

$$111 \text{ km } /1{,}000{,}000 = 11 \text{ cm}$$

Or 11 cm change in location will corresponds to 1 digit change our latitude or longitude coordinate. Now, if quadrotor is moving with 5.5 m/sec then, let's suppose our PD-controller make ESC to give 300 µs contiunuous pulses to stop.

$$5.5_{\text{m/sec}} \text{ / Refresh rate } = \ 1.1_{\text{m/200ms}}$$

Whereas the refresh rate is 5 Hz for GPS.

$$\frac{1.1_{\text{m}}}{11_{\text{cm}}} = 10_{\frac{\text{points}}{\text{200ms}}}$$

This means we will need a gain of $300/10 \ = 30$.

Now, if we use the same gain for slowly moving quadrotor e.g., 1.4 m/s,

$$1.4_{\text{m/s}} /_{5} = 28 \ _{\text{cm/200ms}}$$

$$\frac{28 \text{ cm}}{11 \text{ cm}} = 2 \text{ or } 3 \text{ points/200ms}$$

Thus, using 30 as gain, there will be 60 or 90 µs. This, 30 µs is enough difference to make the drone unstable. To remove this error, we need to increase the refresh rate. The refresh rate can be increased by doing the interpolation. We can take the first and tenth value and interpolate the values between them i.e., ten values making refresh rate 10 times more.

Virtual heading:

This is another problem that is needed to be solved. In this problem, the nose of the quadcopter is facing in the direction other than north. Since the longitude and latitudes are calculated using north as heading reference thus, we need to rotate the nose of the drone towards north always.

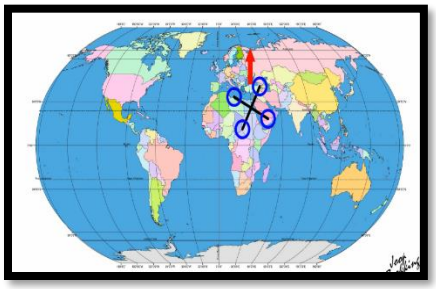

Figure 37: Virtual Heading

To do this, we will use the trigonometric identities using the yaw angle converted into radians as input to the cosine. Moreover, we will limit the maximum correction to 300. This way we will still have the control with the pitch and roll stick on the transmitter.

Inverting PD-gain:

The PD-gain should be inverted when we are below the equator line of earth since the longitude decreases below the equator. Thus, to make correct adjustments, the PD-gain of the controller should be inverted. To invert the PD-gain we need to know where in the world we are, i.e., in which part of the world e.g., NW, NE, SW, SE. This can be done by using the data from the GPS. The GGA line can be used to read the North-South and East/West indicator. These variables will tell us to either change the PD-gain sign or not. Thus, we can implement it easily.

Altitude Hold Mode:

The altitude hold function of quadrotor is implemented by firstly filtering the pressure data. There were two issues to address for getting the data from barometer as follows:

- Light sensitivity
- Conversion time and Filtering

Light Sensitivity:

The light sensitive pressure sensor gives very irregular values of pressure even when a torch is flashed at it. To save it from light, we needed to put the pressure sensor in the box with holes such that no light enters inside.

Conversion Time and Filtering:

The ADC of barometer has a conversion time of about 9 ms. Thus, we need to wait for two loops (4ms each) after requesting pressure data from the sensor. When the pressure data is received, we need to request for temperature data which again takes two loops of 4ms (250 Hz) for getting it ready. Then calculations are done for altitude measurement. This process takes a very long time as compared to overall refresh rate of whole loop.

To increase the refresh rate, we can pull the temperature data after every 20 cycles. In this way, we can have double rate as compared to the previous one. But this method gets us peaks in the measurements after every 20 cycles due to update temperature values. To remove these peaks, we can perform averaging and thus the resulting signal is somewhat smooth. Still these averaged values contain some noise and cannot be used for implementation in altitude control. So, we filter it using complementary filter and then these values we get are very slow. Thus, we combine the fast values (from averaging) with the slow values (from complementary filter) to get the smooth and fast values to be used in PD-controller for altitude hold programming. The altitude hold mode is nothing but a PD-controller that registers the value of throttle and maintains the height when asked. The PD-controller works same as explained in the GPS-hold function.

Telemetry System:

The APC 220 transceivers are used for sending the data from STM32 to the ground where the other transceiver is connected Arduino UNO clone and LCD display.

The receiver side contains buzzer and switch. The buzzer indicates if any error occurs during flight while the switch is used for programming the Arduino.

We can send multiple data over the transceiver to be displayed on the LCD but as APC 220 uses UART protocol, we can send only 8-bits per cycle. This limits us such that our 32-bit long latitude and longitude values cannot be sent with one cycle. Thus, to overcome this we perform the right shift after every byte sending. This way we can send the whole 32-bit integer in 4 cycles. The others variable can also be displayed such as IMU angles, flight modes and battery voltage etc.

CHAPTER 05: FLIGHT COMPUTER

5.1 Monocular Camera

The SLAM system that uses only one camera is called Monocular SLAM. This sensor structure is particularly simple, and the cost is particularly low. Therefore, the monocular SLAM has been very attractive to researchers. You must have seen the output data of a monocular camera: photo. What can we do with a single photo? A photo is essentially a projection of a scene onto a camera's imaging plane. It reflects a three-dimensional world in a two-dimensional form. One dimension lost during this projection process is the so-called depth. In a monocular case, we cannot obtain the distance between objects in the scene and the camera by using a single image. Later, we will see that this distance is critical for SLAM. Because we humans have seen many photos, we formed a natural sense of distance for most scenes, helping us determine the distance relationship among the objects in the image. For example, we can recognize objects in the image and correlate them with their approximate size obtained from daily experience. The close objects will occlude the distant objects; the sun, the moon, and other celestial objects are infinitely far away; an object will have a shadow if it is under sunlight. This common sense can help us determine the distance of objects, but certain cases confuse us, and we can no longer distinguish an object's distance and proper size. We cannot determine whether the figures are real people or small toys purely based on the image itself. Unless we change our view angle, explore the three-dimensional structure of the scene. It may be a big but far away object, but it may also be a close but small object. They may appear to be the same size in an image due to the perspective projection effect.

Since the image taken by a monocular camera is just a 2D projection of the 3D space, if we want to recover the 3D structure, we must change the camera's view angle. Monocular SLAM adopts the same principle. We move the camera and estimate its motion, as well as the distances and sizes of the objects in the scene, namely the scene's structure. So how do we estimate these movements and structures? From everyday experience, we know that if a camera moves to the right, the objects in the image will move to the left, which gives us inspiration for inferring motion.

On the other hand, we also know that closer objects move faster while distant objects move slower. Thus, when the camera moves, the movement of these objects on the image forms pixel disparity. By calculating the disparity, we can quantitatively determine which objects are far away and which objects are close. However, even if we know which objects are near and far, they are only relative values. For example, when we are watching a movie, we can tell which objects in the movie scene are bigger than the others, but we cannot determine the real size of those objects – are the buildings real high-rise buildings or just models on a table? Is it a real monster that destroys a building, or just an actor wearing special clothing? Intuitively, if the camera's movement and the scene size are doubled simultaneously, monocular cameras see the same. Likewise, by multiplying this size by any factor, we will still get the same picture. It demonstrates that the trajectory and map obtained from monocular SLAM estimation will differ from the actual trajectory and map with an unknown factor, called the scale 1. Since monocular SLAM cannot determine this real scale purely based on images, this is also called scale ambiguity. In monocular SLAM, depth can only be calculated with translational movement, and the real scale cannot be determined. These two things could cause significant trouble applying monocular SLAM to real-world applications. The fundamental cause is that depth cannot be determined from a single image. So, to obtain real-scaled depth, we start to use stereo and RGB-D cameras.

5.2 Classical Visual SLAM Framework

Let us look at the classic visual SLAM framework:

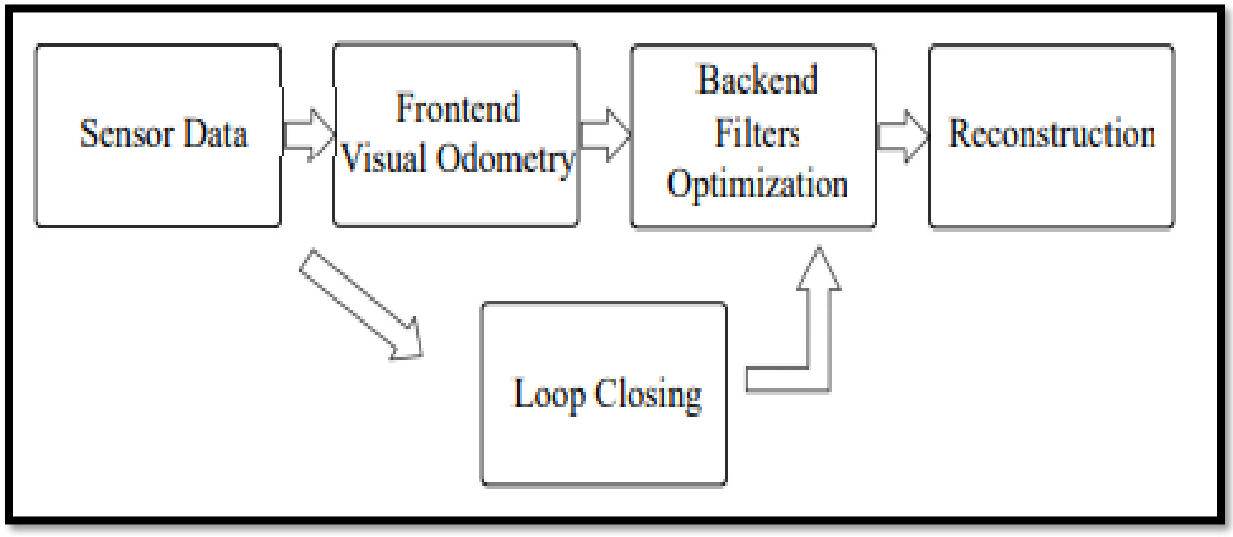

Figure 38: Visual SLAM Framework

A typical visual SLAM workflow includes the following steps:

Step 1: Sensor data acquisition. In visual SLAM, this mainly refers to acquiring and preprocessing camera images. For a mobile robot, this will also include the acquisition and synchronization with motor encoders, IMU sensors, etc.

Step 2: Visual Odometry (VO). VO's task is to estimate the camera movement between adjacent frames (ego-motion) and generate a rough local map. VO is also known as the front end.

Step 3: Backend filtering / optimization. The backend receives camera poses at different time stamps from VO and results from loop closing and then applies optimization to generate a fully optimized trajectory and map. It is also known as the backend because it is connected after the VO.

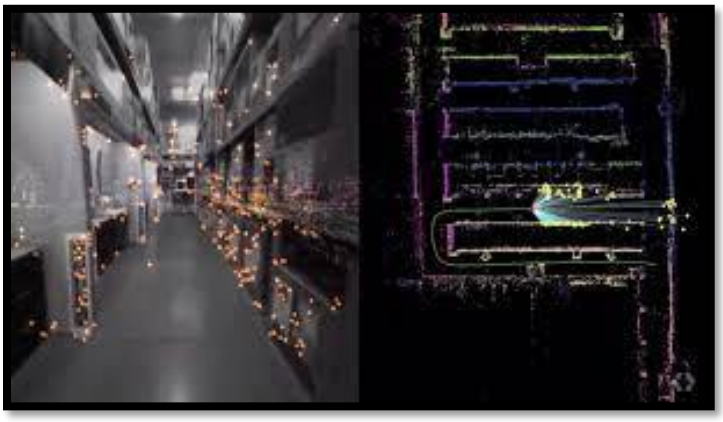

Figure 39: SLAM

Step 4: Loop Closing. Loop closing determines whether the robot has returned to its previous position to reduce the accumulated drift. If a loop is detected, it will provide information to the backend for further optimization.

Step 5: Reconstruction. It constructs a task-specific map based on the estimated camera trajectory. The classic visual SLAM framework results from more than a decade's research endeavor.

The framework and algorithms have been finalized and provided as essential functions in several public vision and robotics libraries. Relying on these algorithms, we can build visual SLAM systems performing real-time localization and mapping in static environments. Therefore, we can conclude that if the working environment is limited to fixed and rigid with stable lighting conditions and no human interference, the visual SLAM problem is solved [8]. The readers may not have fully understood the concepts of the modules mentioned above yet, so we will detail each

module's functionality in the following sections. However, a deeper understanding of their working principles requires certain mathematical knowledge, which will be expanded in this book's second part. For now, an intuitive and qualitative understanding of each module is good enough.

5.3 Visual Odometry

Visual odometry is concerned with the movement of a camera between adjacent image frames, and the simplest case is, of course, the motion between two successive images. For example, when we see the images in Fig. 1-8, we will naturally tell that the right image should be the result of the left image after a rotation to the left with a certain angle (it will be easier if we have a video input). Let us consider this question: how do we know the motion is "turning left"? Humans have long been accustomed to using our eyes to explore the world and estimate our positions, but this intuition is often difficult to explain, especially in natural language. When we see these images, we will naturally think that the bar is close to us, but the walls and the blackboard are farther away. When the camera turns to the left, the bar's closer part starts to appear, and the cabinet on the right side moves out of sight. With this information, we conclude that the camera should be rotating to the left.

However, if we go a step further: can we determine how much the camera has rotated or translated in units of degrees or centimeters? It is still difficult for us to give a quantitative answer. Because our intuition is not good at calculating numbers, a computer's movements must be described with such numbers. So, we will ask: how should a computer determine a camera's motion only based on images? As mentioned earlier, in computer vision, a task that seems natural to a human can be very challenging for a computer. Images are nothing but numerical matrices in computers. A computer has no idea what these matrices mean (this is the problem that machine learning is also trying to solve). In visual SLAM, we can only see blocks of pixels, knowing that they are the results of projections by spatial points onto the camera's imaging plane. To quantify a camera's movement, we must first understand the geometric relationship between a camera and the spatial points.

Figure 40: Visual Odometry

Some background knowledge is needed to clarify this geometric relationship and the realization of VO methods. Here we only want to convey an intuitive concept. For now, you just need to take away that VO can estimate camera motions from images of adjacent frames and restore the 3D structures of the scene. It is named odometry because it is like real wheel odometry, which only calculates the ego-motion at neighboring moments and does not estimate a global map or an absolute pose. In this regard, VO is like a species with only a short memory. Assuming we have visual odometry, we can estimate camera movements between every two successive frames. Connecting the adjacent movements naturally constitutes the robot trajectory movement and addresses the localization problem.

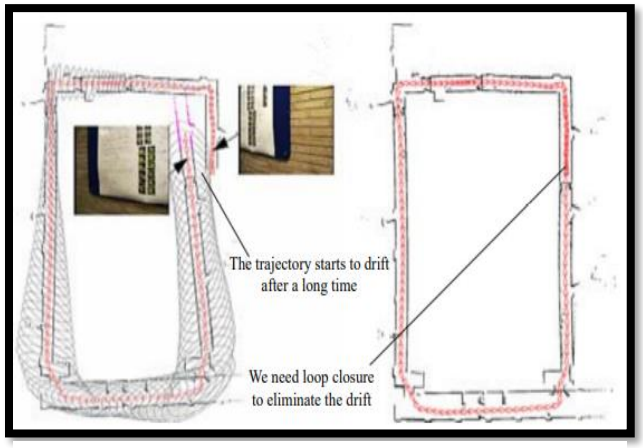

Figure 41: Trajectory Drifting

On the other hand, we can calculate the 3D position for each pixel according to the camera position at each time step, and they will form a map. Here, it seems that the SLAM problem is already solved with a VO. Or is it? Visual odometry is indeed a key technology for solving the visual SLAM problem. We will be spending a significant part explaining it in detail. However, using only a VO to estimate trajectories will inevitably cause accumulative drift. It is because the visual odometry (in the simplest case) only estimates the movement between two frames. We know that a certain error accompanies each estimate. Because of the way odometry works, errors from previous moments will be carried forward to the next moments, resulting in inaccurate estimation after some time. For example, the robot turns left 90◦ and then turns right 90◦. Due to error, we estimate the first 90◦ as 89◦, which can happen in real-world applications. Then we will be embarrassed to find that after the right turn, the robot's estimated position will not return to its origin. Even if the following estimates are perfectly calculated, they will always carry this 1◦ error compared to the true trajectory.

The accumulated drift will make us unable to build a consistent map. A straight corridor may be oblique, and a 90◦ angle may be crooked - this is an unbearable matter! We also need two other components to solve the drifting problem: the backend optimization four and loop closing. Loop closing is responsible for detecting whether the robot returns to its previous position, while the backend optimization corrects the shape of the entire trajectory based on this information.

5.4 Backend Optimization

Backend optimization mainly refers to dealing with the noise in SLAM systems. We hope all the sensor data is accurate, but, even the most expensive sensors still have a certain amount of noise. Cheap sensors usually have larger measurement errors, while expensive ones may be small. Moreover, many sensors' performance is affected by changes in the magnetic field, temperature, etc. Therefore, in addition to solving the problem of estimating camera movements from images, we also care about how much noise this estimation contains, how it is carried forward from the last time step to the next, and how confident we have in the current estimation. The backend optimization solves the problem of estimating the entire system's state from noisy input data and calculating their uncertainty. The state here includes both the robot's trajectory and the environment map.

In contrast, the visual odometry part is usually referred to as the front. In a SLAM framework, the frontend provides data to be optimized by the backend and the initial values. Because the backend is responsible for the overall optimization, we only care about the data itself instead of where it comes from. In other words, we only have numbers and matrices in the backend without those beautiful images. In visual SLAM, the frontend is more relevant to computer vision topics, such as image feature extraction and matching, while the backend is relevant to the state estimation research area. Historically, the backend optimization part has been equivalenting to "SLAM research" for a long time. In the early days, the SLAM problem was described as a state estimation problem, exactly what the backend optimization tries to solve. In the earliest papers on SLAM,

researchers at that time called it "estimation of spatial uncertainty". Although it sounds a little obscure, it reflects the SLAM problem's nature: the estimation of the uncertainty of self-movement and the surrounding environment. To solve the SLAM problem, we need state estimation theory to express the uncertainty of localization and map construction and then use filters or nonlinear optimization to estimate the states' mean and uncertainty (covariance).

5.5 Loop Closing

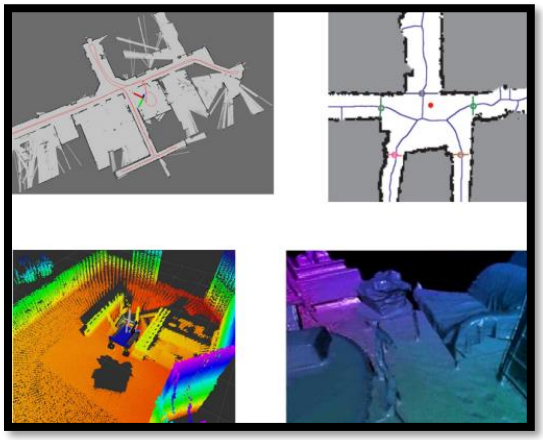

Figure 42: Loop Closing

Loop Closing, known as loop closure detection, mainly addresses the drifting problem of position estimation in SLAM. So how to solve it? Assuming a robot has returned to its origin after a movement period, the estimated does not return to the origin due to drift. How to correct it? Loop closing has a close relationship with both localization and map building. The main purpose of building a map is to enable a robot to know the places it has been. To achieve loop closing, we need to let the robot can identify the scenes it has visited before. There are different alternatives to achieve this goal. For example, as mentioned earlier, we can set a marker where the robot starts, such as a QR code. If the sign were seen again, we would know that the robot has returned to its origin. However, the marker is an intrusive sensor that sets additional constraints to the application environment. We prefer that the robot use its non-intrusive sensors, e.g., the image itself, to complete this task. A possible approach would be to detect similarities between images. Us humans inspire it. When we see two similar images, it is easy to identify that they are taken from the same place. The accumulative error can be significantly reduced if the loop closing is successful. Therefore, visual loop detection is essentially an algorithm for calculating similarities of images. Note that the loop closing problem also exists in laser-based SLAM. The rich information contained in images can remarkably reduce the difficulty of making a correct loop detection. After a loop is detected, we will tell the backend optimization algorithm, "A and B are the same point." Based on this new information, the trajectory and the map will be adjusted to match the loop detection result. This way, if we have sufficient and reliable loop detection, we can eliminate cumulative errors and get globally consistent trajectories and maps.

5.6 Mapping

Mapping means the process of building a map, whatever kind it is. A map is a description of the environment, but the way of description is not fixed and depends on the actual application. Let us take the domestic cleaning robots as an example. Since they move on the ground, a two-dimensional map with marks for open areas and obstacles, built by a single-line laser scanner, would be sufficient for navigation. Moreover, for a camera, we need at least a three-dimensional map for its six-degrees-of-freedom movement. Sometimes, we want a smooth and beautiful reconstruction result, not just a set of points but also a texture of triangular faces. Furthermore, at other times, we do not care about the map. We need to know things like "point A and point B are connected, while point B and point C are not," which is a topological way to understand the environment. Sometimes maps may not even be needed. For instance, a level-2 autonomous driving car can make a lane-following driving only knowing its relative motion with the lanes. For maps, we have various ideas and demands. Map building does not have a particular algorithm than the previously mentioned VO, loop closure detection, and backend optimization. A collection of

spatial points can be called a map. A beautiful 3D model is also a map, so it is a picture of a city, a village, railways, and rivers.
The form of the map depends on the application of SLAM. In general, they can be divided into two categories: metrical maps and topological maps. Metric Maps Metrical maps emphasize the exact metrical locations of the objects in maps. They are usually classified as either sparse or dense. Sparse metric maps store the scene in a compact form and do not express all the objects. For example, we can construct a sparse map by selecting representative landmarks such as the lanes and traffic signs and ignoring other parts.
In contrast, dense metrical maps focus on modeling all the things seen. A sparse map would be enough for localization, while for navigation, a dense map is usually needed. A dense map usually consists of several small grids at a certain resolution. It can be small occupancy grids for 2D metric maps or small voxel grids for 3D maps. For example, in a 2D occupancy grid map, a grid may have three states: occupied, not occupied, and unknown, to express whether there is an object. When a spatial location is queried, the map can provide information about whether the location is passable. We can see that all the grid statuses are stored on the map, thus being storage expensive. There are also some open issues in building a metric map. For example, in large-scale metrical maps, a small steering error may cause the walls of two rooms to overlap, making the map ineffective. Topological Maps Compared to accurate metrical maps, topological maps emphasize the relationships among map elements. A topological map is a graph composed of nodes and edges, only considering the connectivity between nodes. For instance, we only care that point A and point B are connected, regardless of how we could travel from point A to point B. It relaxes the requirements on precise map locations by removing map details and is, therefore, a more compact expression. However, topological maps are not good at representing maps with complex structures. VINS-Mono serves as position feedback in the control loop.

5.7 General layout

During normal operation, initialization is performed only once. The front-end is responsible for continuously extracting feature points and sending them to the back end; the back end is responsible for IMU data collection, pre-integration, and optimization/sliding window operations. Front-end and back-end loop continuously during operation. This system includes the following processes: feature extraction and publishing; IMU extraction and pre-integration; initialization; sliding window and optimization; loop closure detection.
The code mainly includes 3 nodes: feature_tracker, vins_estimator, pose_graph, of which feature_tracker is only responsible for feature point extraction and publishing, pose_graph key frame selection/pose graph establishment/loop detection, while vins_estimator has the most content, including initialization, sliding window, back-end content such as optimization and front-end content such as IMU pre-integration, and 2 threads are split in this node. A standard SLAM system is separated from the front and back ends, while the front end of the VINS system only includes optical flow calculations; initialization and other work are placed in vins_estimator. I understand that it does this because if the IMU data acquisition is placed in the front end, after the pre-integration is completed, it needs to be advertised to the ROS system for the back end to receive. Instead, it is better to let the back end directly receive the IMU data. It is processed in the backend, reducing the communication process.

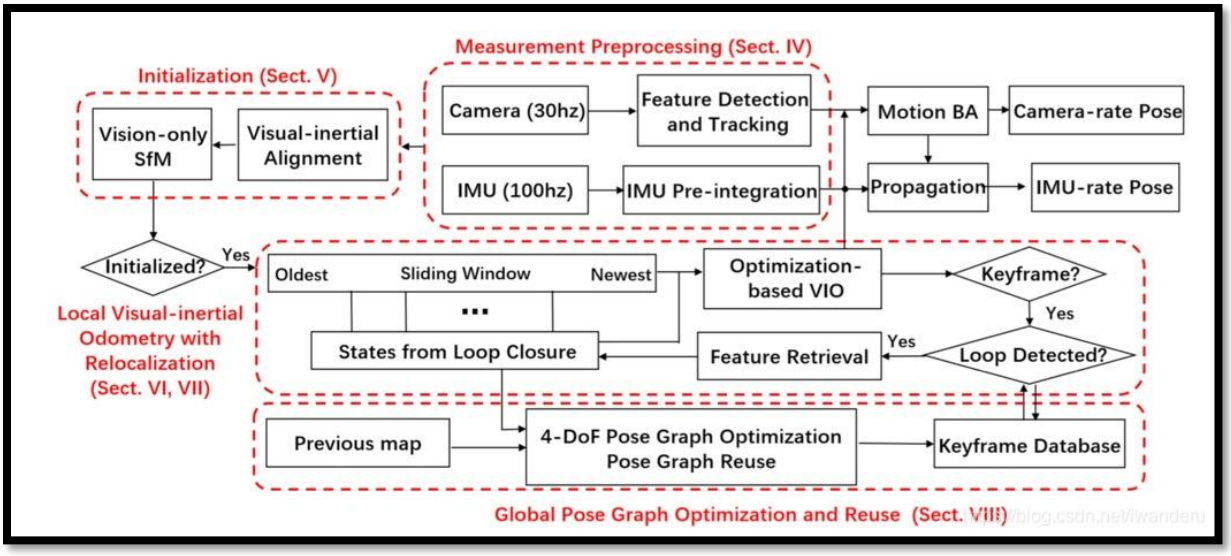

Figure 44: VINS Flowchart

The VINS node communication process is shown in the following figure,

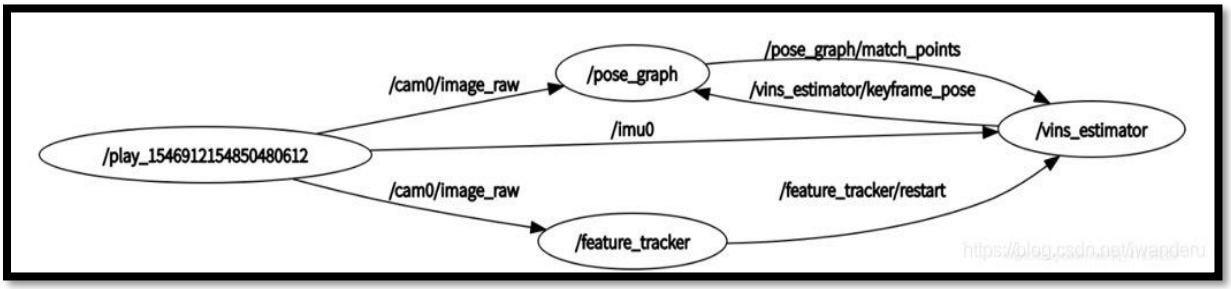

Figure 43: VINS Node Communication Process

Feature Tracker

feature_tracker is the front end of VINS. Its directory is under src/feature_tracker. The main function is to obtain the image frame of the camera and publish the feature points on the cur frame that meet the requirements in the format of sensor_msg::PointCloudPtr according to the preset frequency., so that RVIZ and VINS-estimator receive.

This package mainly includes 4 functions:

1. feature_tracker - contains all algorithm functions for feature extraction/optical flow tracking;
2. feature_tracker_node - the main entry for feature extraction, responsible for the function of a feature processing node;
3. parameters - responsible for reading from the configuration File parameters;
4. tic_tok - timer;note that there are also package.xml and CmakeLists.txt which define all external dependencies and executables.

feature_tracker_node

The feature tracker node is the main processing node responsible for successful feature extraction. The overall process is broken down into steps and briefly explained below.

STEPS:

Step 1: ROS is initialized, and handle and logger level is set.

Step 2: Configuration parameters from file config_euroc > euroc_config.yaml is read.

Step 3: camera internal parameters corresponding to each camera instance is read e.g., NUM_OF_CAM=1 is monocular.

Step 4: The IMAGE_TOPIC (usb_cam /image_raw**),** and the callback function img_callback is executed.

ros::Subscriber sub_img = n.subscribe(IMAGE_TOPIC, 100, img_callback);

IMAGE_TOPIC is usb_cam or image_raw defined in the configuration file. Once the image information is subscribed, the callback function img_callback is executed.

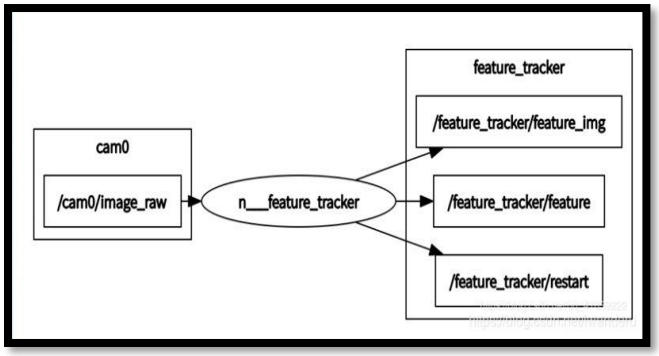

Figure 45: ROS Subscriber

img_callback

This function is the callback function of ROS. The main functions include: the readImage() function that uses the optical flow method to track the feature points of the newly arrived image and encapsulates the tracked feature points into feature_points and publishes it to the topic of pub_img and encapsulates the image into ptr is posted under pub_match.

Now we will dive into its processing details:

Processing

For the first frame of image, only the corresponding time stamp is recorded, and no features are extracted, because it does not have the previous frame of image and cannot obtain the optical flow.

Frequency control

Step 1: Re-initialize the frames with erratic time stamps;

Step 2: Update: last_image_time = img_msg->header.stamp.toSec();

Step 3: Calculate the relationship between the frequency of the currently accumulated pub_count frames and FREQ.

If the actual publishing frequency is greater than the set value, it will not be sent. Therefore, if you want to publish image frames, the actual frequency is lower than the set value. However, If the cumulative error between the actual frequency and the set frequency is greater than 0.01, the frame cannot be released.

Image format adjustment and image reading

The data of sensor_msgs::Image img is read and is converted into MONO8 format and receive it with cv::Mat show_img.

Extraction and optical flow tracking of the feature points of the latest frame for (core).

The most core function of img_callback() is readImage() function. This function implements the feature processing and optical flow tracking. It basically calls all the functions in feature_tracker.cpp.

readImage() passes 2 parameters:

The image of the current frame

The timestamp of the current frame.

See feature_tracker.cpp for the following content to see how the original function is written.

Optical flow tracking and failure point culling

Three major functions are implemented here: optical flow tracking, outlier culling, and the data in different containers is aligned.

- reduceVector() function is used for alignment which avoids a lot of pointer operations. Here,
- The position of the outlier is located by inBorder() and status() functions.

(5) The number of times the current feature point has been tracked

(6) The outliers are eliminated through the fundamental matrix rejectWithF()

rejectWithF(), the logic of this code, our understanding is to first convert the feature point coordinates (pixel coordinates) to normalized coordinates, then back to pixel coordinates, and then use findFundamentalMat() to find outliers.

(7) The tracking points are sorted, and the dense points are removed using setMask()

(8) The total number of feature points MAX_CNT are required to extract new feature points which is done by goodFeaturesToTrack(),

there are currently static_cast(forw_pts.size()) feature points, so n_max_cnt = MAX_CNT - static_cast(forw_pts.size()) feature points need to be added.

(9) New feature points are added by addPoints()

(10) The data is updated and acquisition of normalized coordinates

(11) Now the global id for the newly added feature points is Updated

<u>Encapsulate the image into a ptr instance of type cv_bridge::cvtColor and publish it to pub_match</u>

IMU pre-integration

This part of the content mainly appears in vins_estimator/src/factor/integration_base.h.

Continuous Form

The first question, why IMU points?

The IMU obtains the acceleration and angular velocity, and the pose information of the robot can be obtained through the integral operation of the IMU measurement.

IMU measurements include linear acceleration from the accelerometer and angular velocity from the gyroscope.

$$\hat{a}_t = a_t + b_{a_t} + R_w^t g^w + n_a$$

$$\hat{w}_t = w_t + b_{w_t} + n_w$$

The t subscript represents in the IMU coordinate system, and is affected by the acceleration bias ba, the gyroscope bias bw and the additional noise na, nw. The linear acceleration is the combined vector of the acceleration of gravity and the acceleration of the object. The superscript ^ represents

$$n_a \sim N(0, \sigma_a^2), \ n_w \sim N(0, \sigma_w^2)$$

the measured quantity of the IMU, and the value without the superscript represents the real quantity.
The additional noise is Gaussian noise.
The accelerometer bias and gyroscope bias are defined as random walks that vary with time and whose derivatives satisfy a Gaussian distribution.
For image frames k and k+1, the body coordinate system corresponds to bk and bk+1, and the

$$n_a \sim N(0, \sigma_a^2), \;\; n_w \sim N(0, \sigma_w^2)$$

$$p_{b_{k+1}}^w = p_{b_k}^w + v_{b_k}^w \Delta t_k + \iint_{t \in [t_k, t_{k+1}]} (R_t^w(\hat{a}_t - b_{a_t} - n_a) - g^w) dt^2$$

$$v_{b_{k+1}}^w = v_{b_k}^w + \int_{t \in [t_k, t_{k+1}]} (R_t^w(\hat{a}_t - b_{a_t} - n_a) - g^w) dt$$

$$q_{b_{k+1}}^w = q_{b_k}^w \otimes \int_{t \in [t_k, t_{k+1}]} \frac{1}{2} q_t^{b_k} \otimes \begin{bmatrix} (\hat{w}_t - b_{w_t} - n_w) \\ 0 \end{bmatrix} dt$$

position, velocity, and orientation state values PVQ can be measured according to the IMU in the [tk, tk+1] time interval, in the world coordinate system transmitted.
The three quantities on the right side of the equal sign are the quantities required by the IMU integration, which are the coordinates of the IMU coordinate origin in the world coordinate system at the time of bk+1, the speed, and the rotation angle, which are all added by the value corresponding to the time of bk. The change from bk to bk+1 is obtained, and this change is obtained by IMU integration.
Second question, why IMU "pre" integration?
The sampling frequency of the IMU is higher than the release frequency of the image frame, so the IMU information between two adjacent image frames needs to be integrated to align with the visual information.
Rw<-t, that is, qw<-t, is the amount that needs to be optimized, and this optimized amount is inside the IMU integral symbol. In the subsequent optimization process, this value is constantly changing, so it is necessary to Re-integration of the IMU repeatedly can get a really accurate PVQ value. To reduce the number of IMU integrations, it is hoped that the optimized quantity does not appear in the integration symbol, so the IMU pre-integration method is used. Multiply the above three equations by Rbk<-w on both sides of all equal signs, so that the content in the integral is only related to the measured quantity of the IMU and has nothing to do with the optimized state quantity. This is the IMU pre-integration.

$$R_w^{b_k} p_{b_{k+1}}^w = R_w^{b_k}(p_{b_k}^w + v_{b_k}^w \Delta t_k - \frac{1}{2} g^w \Delta t_k^2) + \alpha_{b_{k+1}}^{b_k}$$

$$R_w^{b_k} v_{b_{k+1}}^w = R_w^{b_k}(v_{b_k}^w - g^w \Delta t_k) + \beta_{b_{k+1}}^{b_k}$$

$$q_w^{b_k} \otimes q_{b_{k+1}}^w = \gamma_{b_{k+1}}^{b_k}$$

In the above formula, the left side of the equal sign is the required value, and the value in the parentheses on the right side of the equal sign does not need to be integrated and can be calculated directly:
The integration results α, β, and γ at this time can be understood as the relative motion of bk+1 to bk, and the corresponding dimensions are displacement, velocity, and quaternion respectively. Here α, β, γ can be approximated by first-order Taylor expansion.
Among them, the quantity with ^ is directly calculated from the IMU measurement quantity. dba and dbw are the variation of bias, and J is their Jacobian corresponding to α, β, γ. Therefore, to find α, β, and γ, we need to first find their scalar values directly calculated by the IMU

$$\alpha_{b_{k+1}}^{b_k} = \iint_{t\in[t_k,t_{k+1}]} R_t^{b_k}(\hat{a}_t - b_{a_t} - n_a)dt^2$$

$$\beta_{b_{k+1}}^{b_k} = \int_{t\in[t_k,t_{k+1}]} R_t^{b_k}(\hat{a}_t - b_{a_t} - n_a)dt$$

$$\gamma_{b_{k+1}}^{b_k} = \int_{t\in[t_k,t_{k+1}]} \frac{1}{2}\Omega(\hat{w}_t - b_{w_t} - n_w)\gamma_t^{b_k}dt$$

measurement, and then use bias to correct them to obtain the true value.
For this part of the code, see the evaluate() function of src/vins_estimator/estimator_node.cpp.

$$\alpha_{b_{k+1}}^{b_k} \approx \hat{\alpha}_{b_{k+1}}^{b_k} + J_{b_a}^{\alpha}\delta b_a + J_{b_w}^{\alpha}\delta b_w$$

$$\beta_{b_{k+1}}^{b_k} \approx \hat{\beta}_{b_{k+1}}^{b_k} + J_{b_a}^{\beta}\delta b_a + J_{b_w}^{\beta}\delta b_w$$

$$\gamma_{b_{k+1}}^{b_k} \approx \hat{\gamma}_{b_{k+1}}^{b_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}J_{b_w}^{\gamma}\delta b_w \end{bmatrix}$$

Discrete Form
The purpose of this part is to obtain the values of α, β and γ from bk to bk+1 . Thus α, β, γ provide an initial value, waiting for subsequent correction by bias and corresponding J.
As shown in the figure above, the sampling frequency of the IMU is much higher than the image frame release frequency of feature_tracker_node, which corresponds to the red and green lines in the above figure. If the two adjacent red lines correspond to times i and i+1, then within a certain delta_ti time range, its average acceleration and average angular velocity are:
Then the PVQ at time i+1 is respectively,
this formula can obviously be used. The program is calculated in an iterative manner, and the values from bk to bk+1 can be obtained recursively in this way. This part of the code appears many times in VINS.
For the first time, see the pridict() function of src/vins_estimator/estimator_node.cpp.

$$\hat{\alpha}_{i+1}^{b_k} = \hat{\alpha}_i^{b_k} + \hat{\beta}_i^{b_k}\delta t + \frac{1}{2}\bar{\hat{a}}_i\delta t^2$$

$$\hat{\beta}_{i+1}^{b_k} = \hat{\beta}_i^{b_k} + \bar{\hat{a}}_i\delta t$$

$$\hat{\gamma}_{i+1}^{b_k} = \hat{\gamma}_i^{b_k} \otimes \hat{\gamma}_{i+1}^{i} = \hat{\gamma}_i^{b_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2}\bar{\hat{w}}_i\delta t \end{bmatrix}$$

For the second occurrence, see the processIMU() function of src/vins_estimator/estimator.cpp

For the third occurrence, see the first half of midPointIntegration() in vins_estimator/src/factor/integration_base.h.

Error transfer matrix

Why is an error transfer function required? There are two main purposes, the first is to provide the corresponding Jacobian for different biases when calculating the values of α, β, and γ; the second purpose is to provide the information matrix for the IMU part in the back-end optimization part.

In fact, the measurement of the IMU at each moment has an error, and this error is Gaussian distributed. In the recursive process of IMU pre-integration, since the value at the next moment is obtained by adding the value at the previous moment and the current measurement value, the variance corresponding to the calculated value of PVQ at each moment is also continuously accumulated. Therefore, it is very important to find the corresponding error transfer function. Therefore, according to the definition of the error transfer function, we can establish an error transfer linear model of the following form, which describes the relationship between the error at

$$\delta z_{k+1}{}^{15\times1} = F^{15\times15}\delta z_k{}^{15\times1} + V^{15\times18}n^{18\times1}$$

the next moment and the current moment:

dzk+1 and dzk are the error of the next moment and the error of the previous moment, respectively.

Time error, the vector/matrix expression corresponding to the above equation is as follows:

For the derivation process of each term, please see the link provided in the reference.

Jacobian's iterative formula is:

$$J_{k+1} = FJ_k,\ J_0 = I$$

$$\begin{bmatrix} \delta\theta_{k+1} \\ \delta\beta_{k+1} \\ \delta b_{a_{k+1}} \\ \delta b_{w_{k+1}} \end{bmatrix} = \begin{bmatrix} 0 & f_{11} & 0 & 0 & -\delta t \\ 0 & f_{21} & I & f_{23} & f_{24} \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix} \begin{bmatrix} \delta\theta_k \\ \delta\beta_k \\ \delta b_{a_k} \\ \delta b_{w_k} \end{bmatrix} + \begin{bmatrix} 0 & -\frac{1}{2}\delta t & 0 & -\frac{1}{2}\delta t & 0 & 0 \\ -\frac{1}{2}R_k\delta t & v_{21} & -\frac{1}{2}R_{k+1}\delta t & v_{23} & 0 & 0 \\ 0 & 0 & 0 & 0 & \delta t & 0 \\ 0 & 0 & 0 & 0 & 0 & \delta t \end{bmatrix} \begin{bmatrix} n_{w_k} \\ n_{a_{k+1}} \\ n_{w_{k+1}} \\ n_{b_a} \\ n_{b_w} \end{bmatrix}$$

after having this J, you can get the real α, β, γ. Note that there have been many J calculations in VINS, one is here, and the other three are in back-end optimization. Be careful not to confuse them.

The covariance iteration formula is:

$$P_{k+1} = FP_kF^T + VQV^T,\ P_0 = 0$$

This covariance matrix can be used to calculate the information matrix of the IMU part of the back-end optimization.

The diagonal covariance matrix Q of the noise term is:

$$Q^{18\times18} = (\sigma_a^2, \sigma_w^2, \sigma_a^2, \sigma_w^2, \sigma_{b_a}^2, \sigma_{b_w}^2)$$

This part of the code corresponds to the second half of midPointIntegration(), as follows:

Interpretation of the IntegrationBase class

This part of the content mainly appears in vins_estimator/src/factor/integration_base.h.

Constructor is defined named IntegrationBase

This constructor reads 4 initial values at time i (12 variables in total), initializes acc_0 and linearized_acc (time i and time bk, respectively) with _acc_0, and initializes gyr_0 and linearized_gyr with _gyr_0 (time i and time respectively). bk time), initialize J with the identity matrix of 15 15, and initialize the covariance transfer matrix with the 0 matrix of 15 15. Initialize the 18*18 noise matrix with the IMU noise parameters read in the yaml file.

Propagation and re-propagation

For this the corresponding functions are propagated () and repropagate(). For propagate (), it is used more times, for PVQ propagation and error transfer from time i to time i+1 inside bk and bk+1.

estimator_node

main () main entry

1. ROS initialization, setting handle

ros::init(argc, argv, "vins_estimator");
ros::NodeHandle n("~");
ros::console::set_logger_level(ROSCONSOLE_DEFAULT_NAME, ros::console::levels::Info);

2. Read parameters and set state estimator parameters

readParameters(n);
estimator.setParameter();

This estimator.setParameter() needs attention, it is in estimator.cpp:

It reads the information matrix of the rotation/translation extrinsic parameters and the reprojection error part of the nonlinear optimization for each camera to the IMU coordinate system.

3. Publish the topic for RVIZ display. For the specific content of this module, please refer to the input and output

registerPub(n);

This function is defined in utility/visualization.cpp: void registerPub(ros::NodeHandle &n).

4. Subscribe to IMU_TOPIC and execute imu_callback

ros:Subscribersub_imu=n.subscribe(IMU_TOPIC,2000,imu_callback,ros::TransportHints().tcpNoDelay());This callback function is more important,

The imu_callback function is very important. This imu_callback mainly does three things .

1. First it put the IMU data in imu_buf and cache it.
2. Second, it obtains the PVQ at the current moment by IMU pre-integration,
3. Last the data is published to rviz, see the pubLatestOdometry () function of utility/visualization.cpp.

5. Subscribe to /feature_tracker/feature and execute feature_callback

ros:Subscriber sub_image = n.subscribe("/feature_tracker/feature", 2000, feature_callback);

This part receives the information of all feature points in the cur frame published by feature_tracker_node, feature_callback **only does one thing**, which is to put all the feature points of the cur frame into feature_buf, which also needs to be locked. Note that all feature points of the cur frame are integrated into one data, that is, sensor_msgs::PointCloudConstPtr &feature_msg.

6. Subscribe to /feature_tracker/restart and execute restart_callback

ros::Subscriber sub_restart = n.subscribe("/feature_tracker/restart", 2000, restart_callback);

One thing restart_callback does is to reset all state quantities to zero and clear all the data in buf.

7. Subscribe to /pose_graph/match_points and execute relocalization_callback

ros::Subscriber sub_relo_points = n.subscribe("/pose_graph/match_points", 2000, relocalization_callback);

8. Create VIO main thread process() (VINS core!)

std::thread measurement_process{process};

This part is the most important and contains most of the VINS content and the hardest content.

The main thread of process():

The main thread will run into following manner

1. IMU and image data is aligned is paired first.

2. IMU data is processed:
The core code of this part is processIMU(), which is in estomator.cpp, and its function is IMU pre-integration,
3. Relocation/loopback detection operation
Return to the process() function of estimator_node.cpp. The function of the next code is to take out the last relocation frame in relo_buf, take out the information in it and execute setReloFrame()
4. Process the img information (core!)
(1) At the beginning, a new data structure is defined, explain:
(2) After the data structure is defined, the next step is to put data into the container.
(3) Process image processImage() (core!)
This part has a lot of content, which will be explained layer by layer in Next Section but for until it will be executed using ProcessImage() Function
estimator.processImage(image, img_msg->header)
5. Visualization
This step publishes odometer information, key pose, camera pose, point cloud and TF relationship to RVIZ. These functions are defined in utility/visualization.cpp, which are all ROS-related codes.
pubOdometry(estimator, header);
pubKeyPoses(estimator, header);
pubCameraPose(estimator, header);
pubPointCloud(estimator, header);
pubTF(estimator, header);
pubKeyframe(estimator);
if (relo_msg != NULL)
pubRelocalization(estimator);
6. PVQ information update of IMU
This step update IMU parameters [P,Q,V,ba,bg,a,g], locked them, pay attention to thread safety.
Here's a question, why update()?
Because after a process() loop is completed, the current PVQ state is different from the state at the beginning of the loop. So, we need to update the current PVQ state, which is tmp_X, based on the current data. Also, it must be locked.
Process Image processImage()
This function is in estimator.cpp. First look at the function name. The parameters it passes in are all the feature points on the current frame and the timestamp of the current frame.
void Estimator::processImage(const map<int, vector<pair<int, Eigen::Matrix<double, 7, 1>>>> &image, const std_msgs::Header &header)

1. **The key frame judgment**

VINS sliding window adopts such a strategy. It judges whether the current frame is a key frame. If it is a key frame, the margin will drop the oldest frame during the sliding window; if it is not a key frame, the margin will drop the previous frame.
Next, look at the logic in the addFeatureCheckParallax() function, which is defined in feature_manager.cpp.
There are 2 judgment indicators for judging key frames in this part. The first one is "the average parallax apart from the previous keyframe", which corresponds to parallax_num and parallax_sum / parallax_num in the code ; the second one is "If the number of tracked features goes below a certain threshold, we treat this frame as a new keyframe", which corresponds to last_track_num in

the code . Note that there is also a function in this part called compensatedParallax2(), which is used to calculate the disparity of the current feature point.

2. Store image data, time, and temporary pre-integration value in the image frame class

3. Update the initial value of temporary pre-integration

4. If you want to calibrate external parameters, then calibrate

5. Initialize

Why do we need to initialize?

For the monocular system:

- For depth information, we need to move the visual two-dimensional information i.e., triangulation.
- This "pseudo-depth", and its scale is random, not real, so the IMU is needed to calibrate this scale.
- To make the IMU calibrate this scale, the IMU also needs to be moved., get the P of PVQ.
- In addition, the IMU has bias, and vision can be used to calibrate the rotation bias of IMU.
- The world coordinate system can be obtained by initializing can be determined with the help of g.

This initialization does not occur **only once**, or once when the system restarts. Most of the time, the system is the state of NON-LINEAR. Because, during initialization, the initial values of scaler and bias can be determined. After the scaler is determined, the landmark points obtained during initialization are accurate, and the feature points obtained by PnP or BA are all true scales. After the initial value of bias is determined, it will be updated in real time in the subsequent nonlinear optimization process.

Fundamentals

If the external rotation parameter qbc is unknown, first estimate the external rotation parameter.

In fact, this part is not the initialized content in the VINS code**,** but the judgment and operation that needs to be completed before initialization, see 5.2-4. According to the rotation relationship between the IMU and the visual part, the following relationship can be obtained:

$$R_{c_{k+1}}^{b_k} = R_{b_{k+1}}^{b_k} \cdot R_c^b = R_c^b \cdot R_{c_{k+1}}^{c_k}$$

$$q_{b_{k+1}}^{b_k} \otimes q_c^b = q_c^b \otimes q_{c_{k+1}}^{c_k}$$

$$[\mathcal{Q}_1(q_{b_{k+1}}^{b_k}) - \mathcal{Q}_2(q_{c_{k+1}}^{c_k})] \cdot q_c^b = Q_c^b \cdot q_c^b = 0$$

Perform svd decomposition on A, where the singular vector corresponding to the smallest singular value is the qbc that needs to be solved.

Use SfM to determine the positional relationship of each pose and feature point relative to the c0 frame.

This part is like image-based 3D reconstruction. Triangulation and PnP can be used to convert the pose and feature point position of this string of ck frames. Determined (the feature point is a pseudo depth), and the external parameters qbc and pubic are added, and the pose of a series of bk frames is also determined.

The parameters to the IMU from the camera are (Rbc, qbc), and the attitude transformation to the IMU system of coordinates from the camera’s system can be obtained as:

Using the camera rotation constraint to calibrate the IMU angular velocity bias.

The objective function of the solution is shown in the following formula:

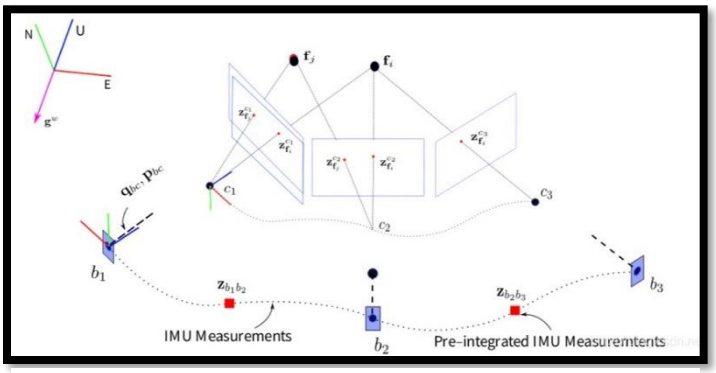

Figure 46: IMU and Pre-integrated IMU Measurements

After the SfM is completed and the external parameters are calibrated, the **first two values are known**, and we assume that the first two values are accurate. Ideally, the product of these three numbers should be a unit quaternion. Unfortunately, the third value is obtained by IMU pre-integration, and there is bias in the pre-integration. Therefore, by minimizing this objective function, the rotation bias can be calibrated!

In the IMU pre-integration part:

$$\gamma_{b_{k+1}}^{b_k} \approx \hat{\gamma}_{b_{k+1}}^{b_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2} J_{b_w}^{\gamma} \delta b_w \end{bmatrix}$$

bring it into the loss function, you can get:

$$q_{b_{k+1}}^{c_0}{}^{-1} \otimes q_{b_k}^{c_0} \otimes \gamma_{b_{k+1}}^{b_k} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

or:

$$\gamma_{b_{k+1}}^{b_k} = q_{b_k}^{c_0 -1} \otimes q_{b_{k+1}}^{c_0} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

after adding the residual of the bias, you can get that the

$$\hat{\gamma}_{b_{k+1}}^{b_k} \otimes \begin{bmatrix} 1 \\ \frac{1}{2} J_{b_w}^{\gamma} \delta b_w \end{bmatrix} = q_{b_k}^{c_0 -1} \otimes q_{b_{k+1}}^{c_0} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

real part does not need to be calibrated, so Only the imaginary part needs to be considered, that is:

$$J_{b_w}^{\gamma} \delta b_w = 2(\hat{\gamma}_{b_{k+1}}^{b_k}{}^{-1} \otimes q_{b_k}^{c_0 -1} \otimes q_{b_{k+1}}^{c_0})_{vec}$$

multiply both sides to construct the form of Ax=B. By using LDLT decomposition, the state quantity can be obtained:

$$J_{b_w}^{\gamma}{}^{T} J_{b_w}^{\gamma} \delta b_w = 2 J_{b_w}^{\gamma}{}^{T} (\hat{r}_{b_{k+1}}^{b_k}{}^{-1} \otimes q_{b_k}^{c_0 -1} \otimes q_{b_{k+1}}^{c_0})_{vec}$$

in fact, it is not like the Gauss-Newton method to solve iteratively, but it is more like using the direct method, that is, the matrix operation method to find the state quantity to be optimized.

STEPS

Step 1: The input of parameters and the definition of the container

In the formula, A and b correspond to Ax=b. The parameter passed in is all_image_frame, not just the frames within the sliding window. frame_i and frame_j read two adjacent frames in all_image_frame respectively.

Step 2: Set the last formula of this subsection to construct the Ax=b equation

The first formula in this subsection, there is a summation there, so you need to traverse all_image_frame and then stack A and b.

Step 3: ldlt decomposition

Step 4: Add angular velocity bias to the IMU pre-integration in the sliding window

Step 5: Recompute IMU integrals for all frames (important!)

Use the translation of IMU to estimate gravity/speed of each bk frame/scale scaler

First of all, it is necessary to clarify what the state quantity needs to be optimized, each frame's speed given with respect of bk system, and scaler of SfM and in the c0 frame:

$$\mathcal{X}_I^{3(n+1)+3+1} = [v_{b_0}^{b_0}, v_{b_1}^{b_1}, ..., v_{b_n}^{b_n}, g^{c_0}, s]$$

There is a legacy problem with this, which is why optimize for speed?

In the IMU pre-integration part, there are already the following formulas:

$$R_w^{b_k} p_{b_{k+1}}^w = R_w^{b_k}(p_{b_k}^w + v_{b_k}^w \Delta t_k - \frac{1}{2} g^w \Delta t_k^2) + \alpha_{b_{k+1}}^{b_k}$$

$$R_w^{b_k} v_{b_{k+1}}^w = R_w^{b_k}(v_{b_k}^w - g^w \Delta t_k) + \beta_{b_{k+1}}^{b_k}$$

But we don't know the w coordinate system, only the c0 coordinate system, so the above formula needs to be transferred to the c0 coordinate system:

$$\alpha_{b_{k+1}}^{b_k} = R_{c_0}^{b_k}(s p_{b_{k+1}}^{c_0} - s p_{b_k}^{c_0} + \frac{1}{2} g^{c_0} \Delta t_k^2 - R_{b_k}^{c_0} v_{b_k}^{b_k} \Delta t_k)$$

$$\beta_{b_{k+1}}^{b_k} = R_{c_0}^{b_k}(R_{b_{k+1}}^{c_0} v_{b_{k+1}}^{b_{k+1}} + g^{c_0} \Delta t_k - R_{b_k}^{c_0} v_{b_k}^{b_k})$$

Then, by solving above equations in the paper, following can be obtained:

also, by LDLT decomposition, the state quantity can be obtained:

$$H^T H \mathcal{X}_I^{10\times 1} = H^T b$$

$$\begin{bmatrix} -I\Delta t_k & 0 & \frac{1}{2} R_{c_0}^{b_k} \Delta t_k^2 & R_{c_0}^{b_k}(p_{c_{k+1}}^{c_0} - p_{c_k}^{c_0}) \\ -I & R_{c_0}^{b_k} R_{b_{k+1}}^{c_0} & R_{c_0}^{b_k} \Delta t_k & 0 \end{bmatrix} \begin{bmatrix} v_{b_k}^{b_k} \\ v_{b_{k+1}}^{b_{k+1}} \\ g^{c_0} \\ s \end{bmatrix} = \begin{bmatrix} \alpha_{b_{k+1}}^{b_k} + R_{c_0}^{b_k} R_{b_{k+1}}^{c_0} p_c^b - p_c^b \\ \beta_{b_{k+1}}^{b_k} \end{bmatrix}$$

Steps for implementation in code:

Step 1: Input of parameters and definition of container

In the formula, A and b correspond to Ax=b. Note that the parameter passed in is all_image_frame, not just the frames within the sliding window.frame_i and frame_j read two adjacent frames in all_image_frame, respectively.

Step 2: Set the last formula of this subsection to construct the Ax=b equation

Another thing to note is that **although only 2 adjacent frames are calculated in the formula, but in the code, it puts all the frames in, so the A matrix and b vector are much larger than the size in the formula, and in the There is an overlay operation in the code! The same is true for constructing H and J matrices in nonlinear optimization. **Here is where I got confused before looking at the code.

Step 3: ldlt decomposition, get the initial value of scale and g, and use a priori judgment

Step 4: Using the known prior condition of the modulo length of gw to further optimize gc0

Using the modulo length of gw to know this prior condition to further optimize gc0

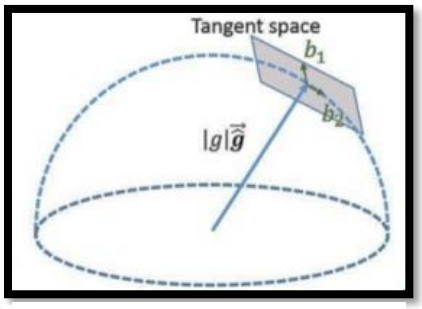

Figure 47: 2-DOF perturbation of gravity

However, there is a bug here, that is, g is optimized as a vector with 3 degrees of freedom, but in fact g has only 2 degrees of freedom, because its modulo length is known. So, here, we need to think of a way, that is, how to use a 2-degree-of-freedom expression to represent a 3-dimensional vector? Here, spherical coordinates are used for parameterization, that

$$\hat{g} = ||g||\bar{\hat{g}} + w_1 b_1 + w_2 b_2 = ||g||\bar{\hat{g}} + \vec{b}^{3\times 2}_{i}$$

is, a hemisphere is drawn with the modulus length of g as the radius. The blue line in the above figure corresponds to the direction of the measured value of gc0 (that is, the direction before optimization), at this intersection point Find a tangent plane and construct a coordinate system with gc0, b1, b2, then the coordinate values w1 and w2 on the axes b1 and b2 are the quantities we need. After finding out, add the right side of the equal sign together, which is the optimized gc0 value.

Note that the direction of b1 is obtained by the cross product of the direction of the measured value of gc0 and [1,0,0], and the direction of b2 is obtained by the cross product of the direction of the measured value of gc0 and b1.

In this way, the formula has been updated, and the optimization amount is reduced by 1 dimension:

$$\begin{bmatrix} -I\Delta t_k & 0 & \frac{1}{2}R_{c_0}^{b_k}\Delta t_k^2\vec{b} & R_{c_0}^{b_k}(p_{c_{k+1}}^{c_0} - p_{c_k}^{c_0}) \\ -I & R_{c_0}^{b_k}R_{b_{k+1}}^{c_0} & R_{c_0}^{b_k}\Delta t_k\vec{b} & 0 \end{bmatrix} \begin{bmatrix} v_{b_k}^{b_k} \\ v_{b_{k+1}}^{b_{k+1}} \\ w \\ s \end{bmatrix} = \begin{bmatrix} \delta\alpha_{b_{k+1}}^{b_k} + R_{c_0}^{b_k}R_{b_{k+1}}^{c_0}p_c^b - p_c^b - \frac{1}{2}R_{c_0}^{b_k}\Delta t_k^2||g||\bar{\hat{g}} \\ \beta_{b_{k+1}}^{b_k} - R_{c_0}^{b_k}\Delta t_k||g||\bar{\hat{g}} \end{bmatrix}$$

that is, the LDLT decomposition is also used:

Next is code parsing,

$$H^T H\mathcal{X}_I^{9\times 1} = H^T b$$

(1) the input of parameters and the definition of the container

In the above formula, A and b correspond to Ax=b. Note that the parameter passed in is all_image_frame, not just the frames within the sliding window. frame_i and frame_j read two adjacent frames in all_image_frame, respectively.

(2) Iteratively solves four times in total, and constructs and extends to the space

(3) Set the last formula of this subsection to construct the Ax=b equation

(4) ldlt decomposition to obtain the optimized state quantity x

Use gc0 and gw to determine the world coordinate system

(1) Find the rotation matrix from c0 to w system Rwc0 = exp([θu])

$$\mathbf{u} = \frac{\hat{\mathbf{g}}^{c_0} \times \hat{\mathbf{g}}^{w}}{\left\|\hat{\mathbf{g}}^{c_0} \times \hat{\mathbf{g}}^{w}\right\|}, \quad \theta = atan2(\left\|\hat{\mathbf{g}}^{c_0} \times \hat{\mathbf{g}}^{w}\right\|, \hat{\mathbf{g}}^{c_0} \cdot \hat{\mathbf{g}}^{w})$$

向量x乘是sin，.乘是cos，除之后便是tan

gc0 has been calculated, and gw has always been a known quantity, so the difference between them The included angle θ can be found quickly; so we can then use gc0 and gw as a cross product to get a rotation axis u; finally, rotate the c0 coordinate system by a θ around the rotation axis, and then we can find the c0 to w system Alignment relationship, that is, Rwc0 = exp([θu]).

(2) The variables contained by c0 coordinates system will be rotated to w-coordinate system and then multiplied by Rwc0.

(3) Restore the camera translation and feature point scale to metric units . The initialization is done!

Code Analysis

This part of the code is at the end of estimator::processImage(). Although the life cycle of the code in the initialization part is relatively short, the amount of code is huge! It is mainly divided into two parts. The first part is pure visual SfM to optimize the pose in the sliding window, and then fuse IMU information to optimize each state quantity according to the theoretical part.

initialStructure()

This is a large function, and multi-layered nesting dolls. And the initialization in principle, including pure visual SfM, loose coupling of vision and IMU, are all in this part.

The idea of this part is to determine whether the IMU has sufficient motion excitation by calculating the standard deviation of the linear acceleration of all frames in the sliding window to determine whether to initialize.As soon as it comes up, the data structure all_image_frame appears. It contains the visual and IMU information of all frames in the sliding window. It is a hash indexed by timestamp.

(1) The first cycle to find the average linear acceleration in the sliding window

(2) The second cycle, find the standard deviation of the linear acceleration in the sliding window.

Save the normalized coordinates of all features in f_manager in all frames to vector sfm_f

(1) Several containers are defined as soon as they come up, namely:

There are 2 things to pay attention to in this block, why is the capacity frame_count + 1? Because the capacity of the sliding window is 10, plus the current latest frame, the value of 11 frames needs to be stored!It can be found that it stores all the information of a feature point. After the container is defined, the next step is to put data into the container.

(2) Put data into the container

Here, why bother constructing a sfm_f instead of using f_manager directly? My understanding is that the amount of information in f_manager is greater than the amount of information required by SfM (f_manager contains a lot of pixel information), and different data structures serve different businesses, so here the author specially designed for SfM. Brand new data structure sfm_f, specialized services.Find the first frame (the lth frame) that meets the requirements in the sliding window (0-9), which has enough common viewpoint and parallelism with the latest frame (frame_count=10), and find out the two Relative position change relationship between frames.

(1) Define the container

(2) Parallax judgment between two frames, and obtain the relative pose change relationship between the two frames

Here, there is a new function relativePose(), this function is also the main function of 6.3.3, go in and take a look:First of all, figure out what this function is going to do?

a. Calculate the disparity between each frame (0-9) in the sliding window and the latest frame (10) until the first frame that meets the requirements is found as our lth frame;

The function of getCorresponding() is to find the normalized coordinates of all the feature points of the current frame and the latest frame under the corresponding 2 frames, and pair them for use in obtaining the relative pose.

b. Calculate the relative pose relationship between the l frame and the latest frame

The core formula here is m_estimator.solveRelativeRT(), which is very critical. The code here is very simple, that is, pass the corresponding point in, and then set the cv formula, but it is easy to

be confused about who is turning to whom for the obtained R and According to the previous learning content and memory and the reading of the following formula, this relative_R and relative_T are the rotation and translation of the latest frame to the lth frame!
And there is a new data structure m_estimator, analyze it. Data structure: MotionEstimator m_estimatorIt is defined in initial/solve_5pts.h, this class has no data members, only functions. So, kill this h file and integrate it into estimator.cpp!
The principle of this part corresponds to the following,
s1p1 = KP, s2x2 = K (RP + t).
At normalized plane, the coordinates are,
x1 = K-1p1, x2 = K-1p2.
Then they can get,
s2x2 = s1Rx1 + t.
The equation left multiplies t^ which is,
t^s2x2 = s1t^Rx1.
Then left multiplies x2T then gets,
0 = x2T (t^R) x1, or 0 = p2T (K-Tt^ RK-1) p1,
c. There are no frames that meet the requirements, and the entire initialization initialStructure() fails.
construct(): Solve the sfm problem in window for each image frame to determine the quaternion Q (rotation), translation vector T and feature point coordinates of all image frames relative to the reference frame. First, a container is defined, and let's look at the data structure of this container. So, this data structure is like MotionEstimator, mainly to realize the function, it has only one data member feature_num.
There are mainly two things done here. First, SfM, to get Q, T, sfm_tracked_points, pass in frame_count + 1, l, relative_R, relative_T, sfm_f parameters, and these three quantities are based on the representation. The second thing is marginization_flag = MARGIN_OLD. This shows that the first slidingwindow() mars the old frame in the late initialization.

(1) Take the lth frame as the reference coordinate system, and obtain the pose of the latest frame under the reference coordinate system
(2) Construct a container to store the pose of the lth frame in the sliding window relative to other frames and the latest frame
Note that these containers store relative motion, and the upper-case container corresponds to l-frame rotation to each frame. The lowercase container is used for global BA, and it is also used for l-frame rotation to each frame. The reason why this opposite rotation is saved in these two places is because this oppositely rotated matrix is required when triangulating for depth! To express the difference, these two types of containers are called coordinate system transformation matrices, not poses!
(3) For the lth frame and the latest frame, their relative motion is known and can be directly put into the container
Note that this piece has an operation that takes the opposite rotation, because the opposite rotation is required for triangulation!
(4) Triangulate the l frame and the latest frame and obtain the spatial coordinates of their common viewpoint on the l frame.
Note that there is one premise for triangulation: the (relative) poses of the two frames are known. Only in this way can the three-dimensional coordinates of their common viewpoints be restored.

The core of this function is triangulatePoint(Pose0, Pose1, point0, point1, point_3d). First, he took out the feature points of sfm_f and checked one by one to see if the feature points were observed in 2 frames. If they were observed, the triangulation operation was performed. Then look at how triangulatePoint() is written. The principle is as follows, for the 3D coordinates we require, it can be expressed in a homogeneous form, here, is the transformation from l frame to each frame, note that the representation of this block is the opposite of the representation we are used tox is the normalized plane coordinate of the camera:
λ is the depth value, the above conditions are known:

$$x = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

For the matrix (right side), its first row×(−u) The second row×(−v), and

$$T = \lceil R \quad T \rceil = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}$$

$$\begin{bmatrix} -r_2 + vr_3 \\ r_1 - ur_3 \end{bmatrix} X = 0$$

then add to get the third row. Therefore, it is linearly related, and the first two rows can be reserved.

Given a normalized plane coordinate x and a transformation matrix T, you can construct two about X system of linear equations. There are more than two image observations to find X.

$$\begin{bmatrix} -r_2^{(1)} + v^{(1)} r_3^{(1)} \\ r_1^{(1)} - u^{(1)} r_3^{(1)} \\ -r_2^{(2)} + v^{(2)} r_3^{(2)} \\ r_1^{(2)} - u^{(2)} r_3^{(2)} \\ \vdots \end{bmatrix} X = 0$$

Since the rank of the obtained matrix is greater than the number of unknowns, the above equation has no non-zero solution. Using SVD to find the least squares solution. This idea is very similar to the method of online calibration of IMU external parameters. Note that the x vector is four-dimensional, and the last number of the solved value needs to ensure that it is 1, so,

(5) For each frame after the lth frame in the sliding window, use PnP to find its pose with the previous frame, obtain the pose, and then triangulate with the latest frame to obtain the 3D coordinates of their common viewpoints

Here the modified form of structure of the source code is done to make it more logically smoother. triangulateTwoFrames() has been discussed before, and now we will pay attention to the solveFrameByPnP() function.PnP is used in section 7.7 of "SLAM14 Lectures". 2D-2D epipolar geometry is only used for the first time, that is, when there are no 3D feature point coordinates, once there are feature points, it will be used later. Find the pose in 3D-2D. Then it will enter the loop of PnP to find new poses, and then triangulate to find new 3D coordinates.
The solveFrameByPnP() code logic can be divided into 4 parts,
a. The first screening: pass all the feature points of the sliding window, those without 3D coordinates.
b. Because PnP is performed on the current frame and the previous frame, these feature points with 3D coordinates must be observed not only in the current frame, but also in the previous frame.

c. If these feature points with 3D coordinates appear in both previous frame and the current frame but the number is less than 15, then the entire initialization fails. Because it is passed up layer by layer.
d. Apply the formula of OpenCV to solve PnP.

(6) From the l+1st frame to the last frame of the sliding window, triangulate and supplement the 3D coordinates with the lth frame.
Now go back to the construct() function. In the previous step, find each frame after the l frame. The pose of the frame, and their 3D coordinates relative to the common viewpoint of the last frame are also obtained, but this is not enough. Now continue to add 3D coordinates, then triangulate with the lth frame.
(7) For each frame before the lth frame in the sliding window, use PnP to find its pose with the next frame, obtain the pose, and then triangulate with the lth frame to obtain the 3D coordinates of their common viewpoints
The frames after the l frame have all landed, and now the previous frames are resolved.
(8) Triangulate other unrestored feature points
So far, the poses of all image frames in the sliding window and the 3D coordinates of the feature points are obtained.
(9) Use ceres for global BA
a. Declare problem attention, because the quaternion is four-dimensional, but the degree of freedom is three-dimensional, so LocalParameterization needs to be introduced.
b. Add the amount to be optimized: global pose
Here, it can be found that only the pose is optimized, and the 3D coordinates of the feature points are not optimized!
c. Fixed a priori value
Because the l frame is the reference frame, the translation of the latest frame is also a priori. If it is not fixed, the original considerable amount will become insignificant.
d. Adding residual block
The method used here is still to minimize the reprojection error, so 2D-3D information is required. Note that there is no loss function added to this block.
e. shur elimination solution
There is a shur elimination knowledge point in this piece, review it. The shur elimination has two major functions. One is to use the sparse nature of the H matrix to accelerate the solution in the least-squares method, and the other is to solve the prior information matrix after the old frame is removed by the margin in the sliding window. This is the first usage of shur elimination.
d. Returns the 3D coordinates of the feature point l and the optimized global pose
At this point, the construct() function is all completed!

Provide initial RT estimates for image frames outside the sliding window, then solvePnP optimizes
Now go back to the estimator.cpp file again to see the next step. In the previous process, the state of all frames in the sliding window and the corresponding feature points under the l frame was obtained, and now all frames, that is, outside the sliding window, are also solved. This part of the code has a for loop in the outermost layer, which is to traverse all the image frames:
a. Boundary judgment: For the frames in the sliding window, set them as key frames, and obtain their corresponding rotation and translation from the IMU coordinate system to the l system

Although this part of the code looks simply, it is a bit convoluted and involves a lot of data structures. Here, it should be noted that Headers, Q and T, their sizes are all WINDOW_SIZE+1! The information they store is all in the sliding window, especially Q and T, they are the current visual frame to l frame (also the visual frame) tied to the rotation and translation. So at the beginning, judge whether it is a frame in the sliding window by the timestamp; if so, set it as a key frame; the next two lines are very important, pay attention to the data structure of ImageFrame. The image information also includes the corresponding IMU pose information and IMU pre-integration information, and here is the position where these frames first obtain their corresponding IMU pose information! That is, the rotation and translation of the bk->l frame!

b. Boundary judgment: If the timestamp of the current frame is greater than the timestamp of the i-th frame in the sliding window, then i++

The code is easy to understand, but I don't understand why it is written like this? This part is a bit like the two rulers in the kmp algorithm, a larger ruler is all_image_frame, and the other small ruler is Headers, which correspond to the length of all frames and the length of the sliding window respectively;

The small ruler is fixed, and the large ruler slides on it; each cycle, the large ruler slides by one grid; because the small ruler is relatively back, only the large ruler moves at the beginning, and the small ruler does not move; if the large ruler and the small ruler have the same scale If the scale of the large ruler is larger than that of the small ruler, the small ruler will move one division.

c. For all frames outside the sliding window, find the rotation and translation of their corresponding IMU coordinate system to frame l

This part is the same as the previous part of the idea. The only thing to pay attention to is the last two lines. This block obviously tells us that the rotation and translation of bk->l is passed in.

visualInitialAlign() (core!)

visualInitialAlign()

Basically, the theoretical part of initialization is in the visualInitialAlign() function.

Calculate gyroscope bias, scale, gravitational acceleration, and velocity

Go in and look at the specific content of VisualIMUAlignment(), which is in initial/initial_alignment.h. solveGyroscopeBias() corresponds to section 6.1.3. LinearAlignment() corresponds to sections 6.1.4 and 6.1.5.

Pass the poses Ps and Rs of all image frames and set them as key frames

Recalculate the feature point depth of all f_manager

There are 2 rics here, the small ric is calculated by vins, and the big ric is read from yaml, there may not be any difference. It should be noted here that this triangulate() is not the same function that appeared before. It is defined in feature_manager.cpp, before it is in initial_sfm.cpp. The objects they serve are different data structures!

The bias of the IMU is changed, and the pre-integration in the sliding window is recalculated

This operation is repeated in the solveGyroscopeBias () part of 6.4.1, because 6.4.1 recalculated the angular velocity bias and improved it. Accuracy, calculate it again, and improve the accuracy again, even if it is commented out, is it the one in 6.4.1 that is commented out? try it.

After scaling the Ps, Vs, and depth scales s, it is transformed from frame l to frame image coordinate system relative to frame c0

By rotating gravity to the z-axis, the rotation matrix rot_diff between the world coordinate system and the camera coordinate system c0 is obtained
Rotation of all variables from the reference coordinate system c0 to the world coordinate system w
At this point, the initialization work is all completed! The amount of code is huge, half of the work is in the visual SfM (this part is only responsible for finding the camera pose), and the other half is the loosely coupled initialization mentioned in the paper! In the first half of initialStructure(), a new set of feature points such as sfm_f is created. Although there is data duplication with f_manager, their function is only used to match the pose, and this data structure is built in the stack area., all destroyed after the function ends. And the operation of finding l frames is very clever. The second half is the content of the thesis. See initialization.

5.8 Back-end nonlinear optimization

Theoretical basis

Bayes Model, Factor Graph and Least Squares
This part mainly recalls and summarizes Dr. Dong Jing's open course "Theoretical Basis of Factor Graph".
(1) Bayes model
Suppose that yellow is the pose of the robot at different times, blue is the landmarks observed by the robot, red is the robot's observation of the landmarks, and green is the robot's observation of its own motion.

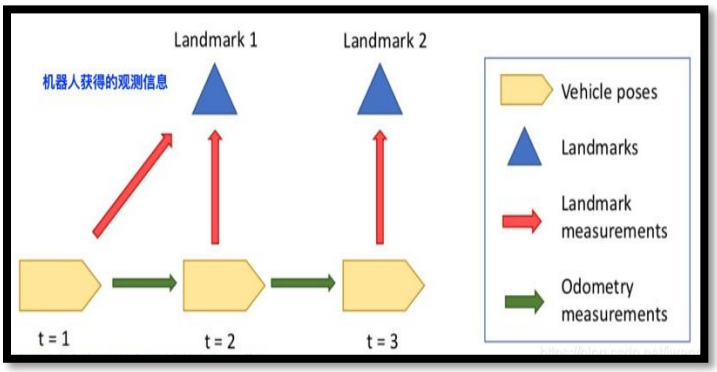

Figure 48: Pose Estimation Problem 1

Then, this process can be described by the following bayes-net:
Then, there can be such a definition: yellow - robot pose - is state quantity - X; blue - landmark point coordinates - is state quantity - -X; red - the robot's observation of the landmark point - is the observation amount - Z; green - the robot's observation of its own motion - is the observation amount - Z;

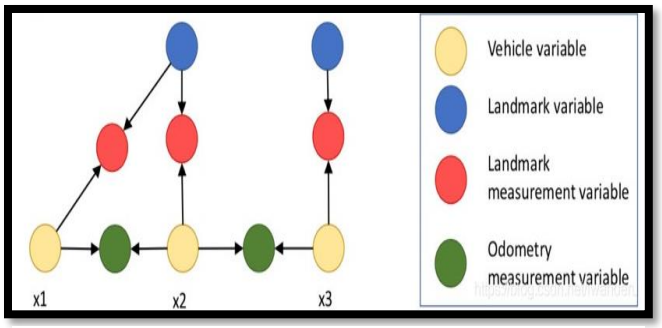

Figure 49: Pose Estimation Problem 2

The state quantity X corresponds to the quantity to be optimized, that is, the vertex in g20, and the ParameterBlock in ceres; the observation quantity Z, corresponds to the residual/factor, that is, the edge in g20, and the ResidualBlock in ceres;

Then, according to Bayes' rule, the joint probability of the observational quantity Z and the state quantity X is equal to the conditional probability multiplied by the marginal probability:

$$P(X, Z) = P(Z|X)P(X)$$

P(Z | X) is the probability corresponding to the observational quantity Z, also called the likelihood; P(X, Z) is the observational quantity The joint probability of Z and the state quantity X; P(X) is the prior probability of the state quantity, such as the world coordinate system, which is a priori. Regard, reference, or global coordinate system, if there is no prior information then the overall drift of the system can meet the solution requirements, then there is an additional degree of freedom that is not observable. The observation variable Z has a characteristic, such as a red

$$P(Z|X) = \prod_i P_i(Z_i|X_i)$$

$$\mathrm{P}(z_{11}, z_{12}, z_{13}, z_{21}, z_{22}|x_1, x_2, x_3, l_1, l_2) =$$

$$P(z_{11}|x_1, l_1)P(z_{12}|x_2, l_1)P(z_{13}|x_3, l_2)P(z_{21}|x_1, x_2)P(z_{22}|x_2, x_3)$$

circle on the left, which is only related to the state variables x1 and l1 and has nothing to do with the landmark point l2. Other observation variables also have similar characteristics, that is, **the two characteristics of sparsity and irrelevance, so the overall observation P(Z | X) can be obtained by multiplying each individual observation.**

(2) Factor diagram

First of all, it is necessary to clarify that our solution target is when P(X | Z) is the largest, that is, in the case of observing Z, the corresponding X; because often the state quantity X is unknown, and in the state X The observation Z of is known, i.e., P(Z | X), therefore, it is necessary to find the relationship between the quantity to be sought P(X | Z) and the known quantity P(Z | X):

$$P(X|Z) = \frac{P(Z|X)P(X)}{P(Z)} \propto P(Z|X)P(X)$$

P(X | Z) It is equal to the formula in the middle of the above formula, and is proportional to the formula on the right. P(Z) is the prior information of the observation, corresponding to the mathematical model of the measurement sensor, such as camera, IMU, wheel speedometer.

We want to find such a set of state quantities X, so that in the case of the current observation Z, the probability of appearing state P(X | Z) is the largest! It can be converted into a solution, the corresponding observation P(Z | X) in the current state is multiplied by a prior P(X) of a state quantity!

$$X^* = \arg\max_X P(X|Z) = \arg\max_X P(Z|X)P(X)$$

This is the legendary, maximum a posteriori estimation, which is transformed into a priori for multiplying the likelihood by the state quantity!

(3) Definition of factor

$$\phi(X_i) \doteq \psi(X_i, Z_i) \propto P_i(Z_i|X_i)$$

因子和概率的关系是 正比。

$$X^* = \arg\max_X \phi(X) = \arg\max_X \prod_i \phi_i(x_i).$$

Then, it can be converted into, the definition of factor is as follows:

$$\mathrm{P}(z_{11}, z_{12}, z_{13}, z_{21}, z_{22}|x_1, x_2, x_3, l_1, l_2) =$$

$$P(z_{11}|x_1, l_1)P(z_{12}|x_2, l_1)P(z_{13}|x_3, l_2)P(z_{21}|x_1, x_2)P(z_{22}|x_2, x_3)$$

$$\phi_i(X_i) = \exp\left(-\frac{1}{2} \| f_i(X_i) \|^2_{\Sigma_i}\right)$$

$$\phi(X) \doteq \phi(x_1, l_1)\phi(x_2, l_1)\phi(x_3, l_2)\phi(x_1, x_2)\phi(x_2, x_3)\phi(x_1)$$

the f(xi) inside is the residual of the observed quantity! That is, we often say IMU residual error, reprojection error, prior error, etc. The exponential function of e is used because the measurement noise of the sensor generally satisfies the gauss distribution.

When the factors are multiplied together, the corresponding maximum likelihood estimate is the least squares.

$$X^* = \arg\max_X \prod_i \phi_i(X_i) = \arg\max_X \log\Big(\prod_i \phi_i(X_i)\Big)$$
$$= \arg\min_X \prod_i -\log\big(\phi_i(X_i)\big) = \arg\min_X \sum_i \| f_i(X_i) \|^2_{\Sigma_i}$$

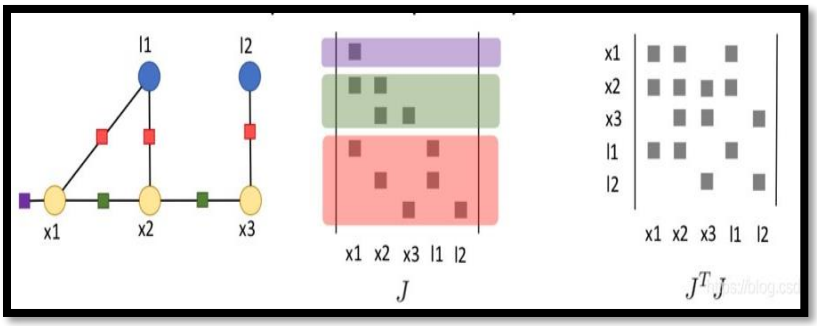

Figure 50: The corresponding positional relationship factor diagram of residuals, state quantities, J, and H.

The most convenient place is to be able to determine briefly what type of residuals this is, and to know which state quantity X of the current residuals corresponds to the J block. There is value, which block is 0! For example, the red box factor on the left, we can know briefly that the reprojection error model is applied, and the corresponding formula for J has already been established; it is connected to x1 and l1, indicating that only the J in these two positions has value. of.

However, it should be noted that **the factor graph of VINS is generally in the following form. The same feature point is observed by 2 frames, and a certain IMU factor relates to the PVQ and bias of the adjacent two frames.**

Therefore, it is very convenient to stuff each row into the J-large matrix! This is also the logic of g2o. Adding the state quantity is called adding vertex, and adding a residual, which is a row of the J matrix, is called adding edge!

Back-end model First determine the state quantity X,

$$\mathcal{X} = \left[\mathbf{x}_0, \mathbf{x}_1, \cdots \mathbf{x}_n, \mathbf{x}_c^b, \lambda_0, \lambda_1, \cdots \lambda_m\right]$$
$$\mathbf{x}_k = \left[\mathbf{p}_{bk}^w, \mathbf{v}_{bk}^w, \mathbf{q}_{bk}^w, \mathbf{b}_a, \mathbf{b}_g\right], k \in [0, n]$$
$$\mathbf{x}_c^b = \left[\mathbf{p}_c^b, \mathbf{q}_c^b\right]$$

including PVQ, bias, external parameters, and inverse depth of feature points of each frame. The number of states determines the dimension of the H matrix and the number of columns of the J matrix. Both types of PQ states are quantities that both vision and IMU optimize. The objective function is:

$$\min_{\mathcal{X}} \left\{ \|\mathbf{r}_p - \mathbf{H}_p\mathcal{X}\|^2 + \sum_{k\in\mathcal{B}} \left\| \mathbf{r}_{\mathcal{B}}\left(\hat{\mathbf{z}}_{b_{k+1}}^{b_k}, \mathcal{X}\right) \right\|^2_{\mathbf{P}_{b_{k+1}}^{b_k}} + \sum_{(l,j)\in\mathcal{C}} \rho\left(\left\| \mathbf{r}_{\mathcal{C}}\left(\hat{\mathbf{z}}_l^{c_j}, \mathcal{X}\right) \right\|^2_{\mathbf{P}_l^{c_j}} \right) \}$$

these three items are the marginalized prior information, the measurement residual of the IMU, and the visual reprojection error. This objective function explains why the vins system is a tightly coupled system, because the three objective functions are optimized together! For such an objective function, an incremental formula as shown in the following formula can be constructed to iteratively solve the optimal solution of the state quantity X:

$$\left(H_p + \sum {J_{b_{k+1}}^{b_k}}^T {P_{b_{k+1}}^{b_k}}^{-1} J_{b_{k+1}}^{b_k} + \sum J_l^{c_j T} P_l^{c_j -1} J_l^{c_j}\right)\Delta X = b_p + \sum {J_{b_{k+1}}^{b_k}}^T {P_{b_{k+1}}^{b_k}}^{-1} r_B + \sum J_l^{c_j T} P_l^{c_j -1} r_C$$

$$\left(\Lambda_p + \Lambda_B + \Lambda_C\right)\Delta X = b_p + b_B + b_C$$

According to the content of the previous section, we know that Jacobian and H can be added line by line (several lines and several lines), and then we will study how to add them. The above factor graph explains IMU constraints and visual constraints very classically.

(1)IMU constraint part

In 4.1.1.IMU pre-integration part, we already know:

$$R_w^{b_k} p_{b_{k+1}}^w = R_w^{b_k}\left(p_{b_k}^w + v_{b_k}^w \Delta t_k - \frac{1}{2} g^w \Delta t_k^2\right) + \alpha_{b_{k+1}}^{b_k}$$

$$\mathbf{r}_B(\mathbf{z}_{b_{k+1}}^{b_k}, \mathcal{X}) = \begin{bmatrix} \mathbf{r}_p \\ \mathbf{r}_q \\ \mathbf{r}_v \\ \mathbf{r}_{ba} \\ \mathbf{r}_{bg} \end{bmatrix} = \begin{bmatrix} \mathbf{q}_{b_i w}\left(\mathbf{p}_{w b_j} - \mathbf{p}_{w b_i} - \mathbf{v}_i^w \Delta t + \frac{1}{2}\mathbf{g}^w \Delta t^2\right) - \alpha_{b_i b_j} \\ 2\left[\mathbf{q}_{b_j b_i} \otimes \left(\mathbf{q}_{b_i w} \otimes \mathbf{q}_{w b_j}\right)\right]_{xyz} \\ \mathbf{q}_{b_i w}\left(\mathbf{v}_j^w - \mathbf{v}_i^w + \mathbf{g}^w \Delta t\right) - \beta_{b_i b_j} \\ \mathbf{b}_j^a - \mathbf{b}_i^a \\ \mathbf{b}_j^g - \mathbf{b}_i^g \end{bmatrix}$$

the left side of the equal sign minus the right side is the IMU residual:

the optimization variable is the two adjacent frames (bk, bk+1 time) p, q, v, ba, bg: $\left[p_{bk}^w, q_{bk}^w\right], \left[v_{bk}^w, b_{ak}, b_{\omega_k}\right], \left[p_{bk+1}^w, q_{bk+1}^w\right], \left[v_{bk+1}^w, b_{a_{k+1}}, b_{\omega_{k+1}}\right]$

In this part, the residual is 15-dimensional, and the state quantity is 7+9+7+9-dimensional. Therefore, for each additional IMU constraint, the total Jacobian matrix increases by 15 rows, increasing 7+9+7+9 columns!

Next, we need to figure out what is the Jacobian of the residual to the state quantity, just add the matrix block of this J to the corresponding position, and the other positions are still 0. Cui Shen deduced the result as follows.

Jacobian of residual to PVQ at time bk: 15*7

$$J[0]^{15\times 7} = \left[\frac{\partial r_B}{\partial p_{b_k}^w}, \frac{\partial r_B}{\partial q_{b_k}^w}\right] = \begin{bmatrix} -R_w^{b_k} & \left[R_w^{b_k}\left(p_{b_{k+1}}^w - p_{b_k}^w - v_{b_k}^w \Delta t_k + \frac{1}{2} g^w \Delta t_k^2\right)\right]^\wedge \\ 0 & -\mathcal{L}\left[q_{b_{k+1}}^w{}^{-1} \otimes q_{b_k}^w\right]\mathcal{R}\left[\gamma_{b_{k+1}}^{b_k}\right] \\ 0 & \left[R_w^{b_k}\left(v_{b_{k+1}}^w - v_{b_k}^w + g^w \Delta t_k\right)\right]^\wedge \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Residual to bk time speed/bias Jacobian: 15*9

$$J[1]^{15\times9}=\left[\frac{\partial r_B}{\partial v_{b_k}^w},\frac{\partial r_B}{\partial b_{a_k}},\frac{\partial r_B}{\partial b_{w_k}}\right]=\begin{bmatrix}-R_w^{b_k}\Delta t & -J_{b_a}^{\alpha} & -J_{b_\omega}^{\alpha}\\ 0 & 0 & -\mathcal{L}\left[q_{b_{k+1}}^{w}{}^{-1}\otimes q_{b_k}^{w}\otimes\gamma_{b_{k+1}}^{b_k}\right]J_{b_\omega}^{\gamma}\\ -R_w^{b_k} & -J_{b_a}^{\beta} & -J_{b_\omega}^{\beta}\\ 0 & -I & 0\\ 0 & 0 & -I\end{bmatrix}$$

Jacobian of residual to PVQ at time bk+1: 15*7

$$J[2]^{15\times7}=\left[\frac{\partial r_B}{\partial p_{b_{k+1}}^w},\frac{\partial r_B}{\partial q_{b_{k+1}}^w}\right]=\begin{bmatrix}R_w^{b_k} & 0\\ 0 & \mathcal{L}\left[\gamma_{b_{k+1}}^{b_k}{}^{-1}\otimes q_{b_k}^{w}{}^{-1}\otimes q_{b_{k+1}}^{w}\right]\\ 0 & 0\\ 0 & 0\\ 0 & 0\end{bmatrix}$$

Residual to the speed of bk+1 moment/bias Jacobian: 15*9

$$J[3]^{15\times9}=\left[\frac{\partial r_B}{\partial v_{b_{k+1}}^w},\frac{\partial r_B}{\partial b_{a_{k+1}}},\frac{\partial r_B}{\partial b_{w_{k+1}}}\right]=\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ R_w^{b_k} & 0 & 0\\ 0 & I & 0\\ 0 & 0 & I\end{bmatrix}$$

Then, when constructing the corresponding H matrix block, it is necessary to multiply the information matrix, JT (Pk+1) J + uI

$$P_{k+1}=FP_kF^T+VQV^T,\ P_0=0$$

(2) Visual constraint part

The residual error in this part is the reprojection error. Note that it is represented on the normalized plane:

$$\mathbf{r}_c=\begin{bmatrix}\frac{x_{c_j}}{z_{c_j}}-u_{c_j}\\ \frac{y_{c_j}}{z_{c_j}}-v_{c_j}\end{bmatrix}$$

but the difference from pure vision is that the rotation and translation in the state quantity X to be optimized are all IMUs. It is connected to the w system, not the camera connected to the w system:

$$[p_{b_i}^w,q_{b_i}^w]、\ [p_{b_j}^w,q_{b_j}^w]、\ [p_c^b,q_c^b]、\ \lambda_l$$

the state quantity here has two characteristics: the quantity to be optimized overlaps with the quantity to be optimized in the IMU constraint part, which again reflects tight coupling; different from IMU, The state quantity of each optimization of the IMU is two adjacent frames, but the two frames of visual optimization are not necessarily adjacent, so they are represented by i and j.

The coordinate relationship between the feature points in the normalized camera coordinate system and the camera coordinate system is:

$$\begin{bmatrix}x\\ y\\ z\end{bmatrix}=\frac{1}{\lambda}\begin{bmatrix}u\\ v\\ 1\end{bmatrix}$$

this block does not use depth, but uses the inverse depth, because the inverse depth satisfies the Gaussian distribution.For the l-th landmark point P, convert P viewing to its pixel coordinates

$$\begin{bmatrix}x_{c_j}\\ y_{c_j}\\ z_{c_j}\\ 1\end{bmatrix}=\mathbf{T}_{bc}^{-1}\mathbf{T}_{wb_j}^{-1}\mathbf{T}_{wb_i}\mathbf{T}_{bc}\begin{bmatrix}\frac{1}{\lambda}u_{c_i}\\ \frac{1}{\lambda}v_{c_i}\\ \frac{1}{\lambda}\\ 1\end{bmatrix}$$

under the current j-th cameracoordinate system:

this conversion process has transformation occurring from, bi coordinate system to w-coordinate system to bj coordinate system to cj coordinate system. The above equation is:
Then, the reprojection coordinates on the normalized plane are fcj = [xcj/zcj, ycj/zcj], and the reprojection error is the formula at the beginning.

$$f_{c_j} = \mathcal{P}_l^{c_j} = R_b^c R_w^{b_j} R_{b_i}^w R_c^b f_{c_i} + R_b^c (R_w^{b_j} (R_{b_i}^w p_c^b + p_{b_i}^w - p_{b_j}^w) - p_c^b)$$

In this part, the residual is 2-dimensional, and the state quantity is 7+7+m-dimensional, so for each additional visual constraint, the total Jacobian matrix increases by 2 rows and 7+7+m columns!
Because according to the chain rule, the calculation of Jacobian can be divided into:
1) The visual residual is derived from the reprojected 3D point fcj.

$$J[0]^{3\times 7} = \left[\frac{\partial f_{c_j}}{\partial p_{b_i}^w}, \frac{\partial f_{c_j}}{\partial q_{b_i}^w}\right] = \begin{bmatrix} \frac{1}{} & 0 & -\frac{x_{c_j}}{} \\ R_b^c R_w^{b_j}, -R_b^c R_w^{b_j} R_{b_i}^w \left(R_c^b \frac{1}{\lambda_l} \overline{P}_l^{c_i} + p_c^b\right)^\wedge \end{bmatrix}$$

$$J[1]^{3\times 7} = \left[\frac{\partial f_{c_j}}{\partial p_{b_j}^w}, \frac{\partial f_{c_j}}{\partial q_{b_j}^w}\right] = \left[-R_b^c R_w^{b_j}, R_b^c \left\{R_w^{b_j}\left[R_{b_i}^w \left(R_c^b \frac{\overline{P}_l^{c_i}}{\lambda_l} + p_c^b\right) + p_{b_i}^w - p_{b_j}^w\right]\right\}^\wedge\right]$$

$$J[2]^{3\times 7} = \left[\frac{\partial f_{c_j}}{\partial p_c^b}, \frac{\partial f_{c_j}}{\partial q_c^b}\right] = \begin{bmatrix} R_b^c \left(R_w^{b_j} R_{b_i}^w - I_{3\times 3}\right) \\ -R_b^c R_w^{b_j} R_{b_i}^w R_c^b \left(\frac{\overline{P}_l^{c_i}}{\lambda_l}\right)^\wedge + \left(R_b^c R_w^{b_j} R_{b_i}^w R_c^b \frac{\overline{P}_l^{c_i}}{\lambda_l}\right)^\wedge + \left\{R_b^c \left[R_w^{b_j}\left(R_{b_i}^w p_c^b + |p_{b_i}^w - p_{b_j}^w\right) - p_c^b\right]\right\}^\wedge \end{bmatrix}^T$$

$$J[3]^{3\times 1} = \frac{\partial f_{c_j}}{\partial \lambda_l} = -R_b^c R_w^{b_j} R_{b_i}^w R_c^b \frac{\overline{P}_l^{c_i}}{\lambda_l^2}$$

2) Multiply by fcj to derive the derivation of each optimization variable.
Then, when constructing the corresponding H matrix block, it is necessary to multiply the information matrix, JT (Pk+1) J + uI. The covariance of the visual constraint is related to the reprojection error when calibrating the camera's internal parameters. VINS takes 1.5 pixels in sqrt_info in the code, which corresponds to the normalized camera plane and needs to be divided by final information matrix and,f (focal length). The real information matrix here is the square of sqrt_info.

(3) The prior constraint parts

In the main process, it can be found that 6 things have been done, of which 1 and 3 are the most critical 2 steps. And optimization() happens to be in solveOdometry(), so it can be found that the code is optimized first and then marg.

solveOdometry() and optimization() (core!)

This block also does 2 things. The first thing is to triangulate the feature points according to the current pose, which also appeared once during initialization, see 6.4.3. Mainly look at the optimization() function. Part 1:

1) Nonlinear optimization

$$sqrt(\Omega_{vis}) = sqrt(\Sigma_{vis}^{-1}) = \left(\frac{1.5}{f} I_{2\times 2}\right)^{-1} = \frac{f}{1.5} I_{2\times 2}$$

a. Declare and introduce robust kernel function

b. Add various amounts to be optimized X - pose optimization amount

New data structures para_Pose[i] and para_SpeedBias[i] appear in this block, because ceres passes in all double types and is initialized in vector2double().

c. Add various quantities to be optimized X - camera external parameters

d. Add various quantities to be optimized X - IMU-image time synchronization error
The addition of the amount to be optimized is why there is no amount to be optimized (inverse depth) of the visual part?
e. vector2double()assigns value to ParameterBlock.
f. Add various residuals - a priori residuals
There are 2 new data structures in this block. On the first execution of this code, there is no prior information, so this is skipped. When it is executed the second time, it does. The assignment of last_marginalization_info appears in the code that follows.
Data structure: The data structure last_marginalization_infois defined in marginization_factor.cpp, which is more complicated. Another data structure is last_marginalization_parameter_blocks. It refers to the state quantity X corresponding to the prior matrix.
g. Add various residuals - IMU residuals
h. Add various residuals - reprojection residuals
Here it is necessary to note again that the residuals of the IMU are two adjacent frames, but the vision is not. Analyzing the code, it adds 2 frames, which are the last two frames where the same feature was observed.
h. Add various residuals - loop closure detection
i. Solve
j. double2vector()
There are 2 values to pay attention to here, namely Vector3d origin_R0, Vector3d origin_P0.This block fixes the prior information. At this point, the first part is over.
(2) Marginalization
The second part is marginalization. This part is only marginalized, not solved, and the solution is left to the first part of the next round of optimization. This part is very difficult to understand.
k.marg_old
1) First, add the residual information of the previous round:
Obviously, I want to marg now, and to construct a new prior H matrix, then add the legacy information of the previous old prior. This block uses drop_set to get rid of the prior information of the oldest frame.
2) Then, add the IMU information to be marg this time:
Obviously, the 0th frame information is marg dropped.
3) Then, add the visual information to be marg this time:
VINS threw away all the feature points seen in frame 0.
4) Synthesize the parameter blocks in the three ResidualBlockInfo into marginalization_info, where the residual and Jacobian of all ResidualBlock (residual items) are calculated, and parameter_block_data is the container of the parameter block. At this point, marg_old ends.
l.marg_new
If the second latest frame is not a key frame, discard (marginalize) the visual measurement of this frame and keep the IMU measurement value in the sliding window. (Other steps are the same as the previous step) else//If the second latest frame is not a key frame, discard (marginalize) the visual measurement of this frame and keep the IMU measurement value in the sliding window. (Other steps are the same as the previous step). At this point, optimization() ends.
failureDetection()
slideWindow()
manager.removeFailures()

5.9 Sliding window

In fact, this part is performed together with the back-end nonlinear optimization, and this part corresponds to the prior part of the loss function of the nonlinear optimization.

The last nonlinear optimization is over, and the final H matrix is the predecessor of the prior matrix of this round of nonlinear optimization.

Constructing a priori matrix

(1) Move poses and landmarks that need to be margin removed. Correspondingly, some columns of J matrix need to be deleted; some rows of H matrix need to be deleted. When operating the a priori matrix, it is necessary to move the row and column to be margined to the upper left corner of the H matrix, and the corresponding part of the b vector to be moved to the top, so that the shur method can be used to marg it, as shown in the following formula:

$$\begin{aligned} \mathbf{b}_m(k) &= \begin{bmatrix} \mathbf{b}_{mm}(k) \\ \mathbf{b}_{mr}(k) \end{bmatrix} = -\sum_{(i,j)\in\mathcal{S}_m} \mathbf{J}_{ij}^{\top}(k)\boldsymbol{\Sigma}_{ij}^{-1}\mathbf{r}_{ij} \\ \boldsymbol{\Lambda}_m(k) &= \begin{bmatrix} \boldsymbol{\Lambda}_{mm}(k) & \boldsymbol{\Lambda}_{mr}(k) \\ \boldsymbol{\Lambda}_{rm}(k) & \boldsymbol{\Lambda}_{rr}(k) \end{bmatrix} = \sum_{(i,j)\in\mathcal{S}_m} \mathbf{J}_{ij}^{\top}(k)\boldsymbol{\Sigma}_{ij}^{-1}\mathbf{J}_{ij}(k) \end{aligned} \tag{47}$$

mm corresponds to is the part to be marg, and rr is the part to stay.

(2) Marginalization: Constructing the prior matrix

This is the marginalization operation. After marginalization, the obtained prior matrix H and prior vector b are respectively:

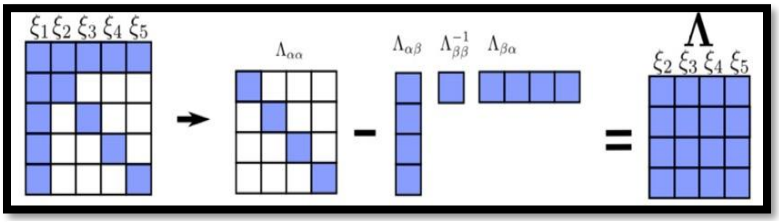

Figure 51: Marginalization: Constructing the prior matrix

$$\begin{aligned} \mathbf{b}_p(k) &= \mathbf{b}_{mr}(k) - \boldsymbol{\Lambda}_{rm}(k)\boldsymbol{\Lambda}_{mm}^{-1}(k)\mathbf{b}_{mm}(k) \\ \boldsymbol{\Lambda}_p(k) &= \boldsymbol{\Lambda}_{rr}(k) - \boldsymbol{\Lambda}_{rm}(k)\boldsymbol{\Lambda}_{mm}^{-1}(k)\boldsymbol{\Lambda}_{mr}(k) \end{aligned}$$

the sliding window will cause the H matrix to no longer be sparse.

Constructing the H matrix of nonlinear optimization

This part is the content of the nonlinear optimization part about constructing the H matrix.

(1) Expand the dimension of the prior matrix. The expanded dimension is the dimension of the state quantity added by the next optimization. Compared with the direct Bundle Adjustment, there is an additional maintenance of the prior matrix.

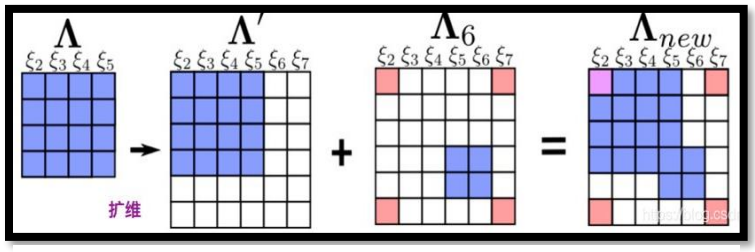

Figure 52: Superimpose with the H matrix of the IMU and the visual part

(2) Superimpose with the H matrix of the IMU and the visual part. In fact, in each round of optimization, the H matrix is increased piece by piece, so the H matrix is added first, and then each residual item is added, the total The H matrix will automatically increase accordingly.

Performing nonlinear optimization (FEJ)

This part is the content of the nonlinear optimization part about the LM/DOGLEG algorithm solution part. Finally, the optimized state quantity is obtained. This piece involves the FEJ issue. Let's talk about the conclusion first. FEJ means that in the total large H matrix, the value of the corresponding part of the prior matrix remains unchanged during the iterative process of this optimization.

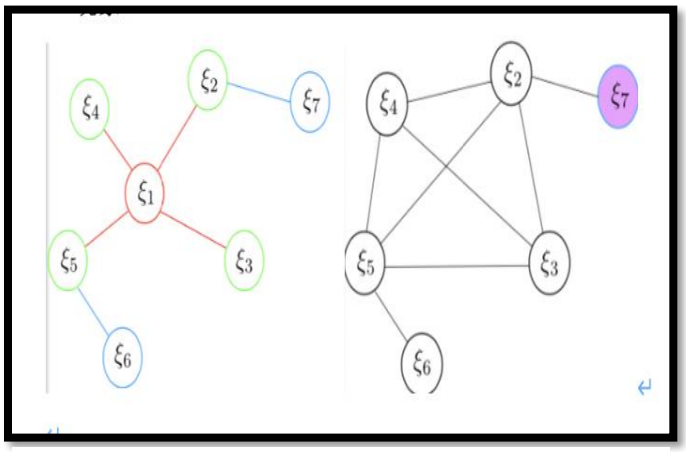

Figure 53: Performing Non-linear Optimization

Besides the reasons,

(1) When the state quantity 1 is merged, the originally independent quantities will become independent, which is manifested in the H matrix, which is the fill-in phenomenon.

(2) When adding new observations some parts of the H matrix are related to both the prior and the newly added information.

(3) If this part of the H matrix block is not fixed, a new linearization point will be obtained in each iteration, and this new linearization point will be different from the prior linearization point, which may lead to the occurrence of the null space of the information matrix. changes, thereby introducing erroneous information when solving. Because the information matrix Λ is not full of rank. The corresponding null space is N, and the incremental δx changes in the null space dimension when solving with Gauss-Newton which does not change our residuals. This means that the system can have multiple solutions x that minimize the loss function.

$$\mathbf{\Lambda}\delta\mathbf{x} = \mathbf{b}$$

$$\mathbf{\Lambda}\delta\mathbf{x} + \mathbf{\Lambda}\mathbf{N} = \mathbf{b}$$

(4) If the linearization point corresponding to the prior part is not fixed, the originally invisible information will become observable, and the problem of multiple solutions will become a definite solution. And this is wrong.

(5) Therefore, FEJ is used, which is the first estimated Jacobian. When different residuals calculate the Jacobian of the same state, the linearization points must be the same. This avoids null space degradation and makes unobservable variables observable. In the total large H matrix, the value of the corresponding part of the prior matrix remains unchanged during the iterative process of this optimization.

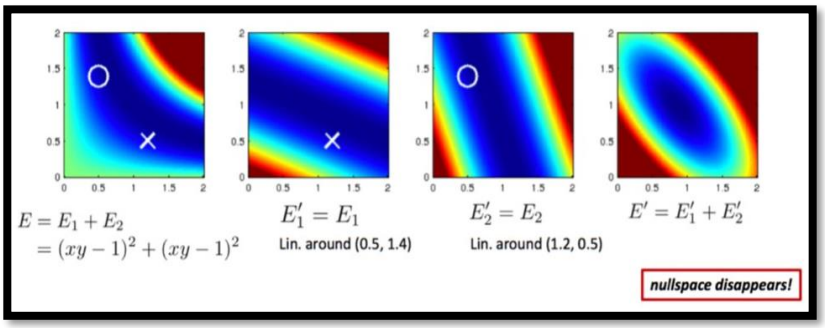

Figure 54: Windowed, Real-Time Optimization

(6) **The prior matrix does not change, but the prior residual changes!**
Although the prior information matrix is fixed, as the iteration progresses, the variables are continuously optimized, and the prior residual needs to follow the changes. Otherwise, the solution system may crash. The variation of the prior residuals can be approximated using a first-order Taylor approximation.

$$\mathbf{b}_{\mathrm{p}}^{\prime}=\mathbf{b}_{\mathrm{p}}+\frac{\partial \mathbf{b}_{\mathrm{p}}}{\partial \mathbf{x}_{\mathrm{p}}} \delta \mathbf{x}_{\mathrm{p}} \quad=\mathbf{b}_{\mathrm{p}}+\frac{\partial\left(-\mathbf{J}^{\top} \mathbf{\Sigma}^{-1} \mathbf{r}\right)}{\partial \mathbf{x}_{\mathrm{p}}} \delta \mathbf{x}_{\mathrm{p}} \quad=\mathbf{b}_{\mathrm{p}}-\mathbf{\Lambda}_{\mathbf{p}} \delta \mathbf{x}_{\mathrm{p}}$$

The other parts are the same as nonlinear optimization. But one thing to note is that in g20, the Jacobian is passed in. Jacobian knows both the IMU residual and the reprojection residual, but in this prior part, we know H, so we need to use the marg. The latter state quantity inversely solves a Jacobian.

Choi Shen said that vins did not use FEJ.8.2 Code Analysis The function slideWhindow() is used in initialization and nonlinear optimization. As for whether the marg is old or new.

if (marginalization_flag == MARGIN_OLD)
Deletes the first frame of the sliding window.

(1) Save the oldest frame information
(2) Move the information in the sliding window forward in turn
(3) Give the information at the end of the sliding window (10 frames) to the latest frame (11 frames) Note that in (2), the forward movement of all information has been achieved. At this time, the latest frame has become the 10th frame in the sliding window. Here, the information of the original latest frame is only used as the next latest frame. the initial value of.
(4) Newly instantiate an IMU pre-integration object for the next latest frame
(5) Clear the buf of frame 11
(6) Delete all the information corresponding to the oldest frame
(7) slideWindowOld()

if (marginalization_flag == MARGIN_NEW)
deletes the 10th frame of the sliding window.

(1) Take out the information of the latest frame
(2) The IMU pre-integration between the current frame and the previous frame is converted into the IMU pre-integration between the current frame and the previous two frames.
(3) Overwrite the information of the previous frame with the information of the latest frame
(4) Since the information of the 11th frame has been covered by the 10th frame, the 11th frame is now cleared
(5) Sliding window

CHAPTER 06: FUTURE WORK &APPLICATIONS

6.1 Future Works

6.1.1 Chip-based Flight Controller

Figure 55: Chip-based Flight Controller

Today, modular design is a popular approach in almost any industry. Let's take a look at what modular hardware is and what its benefits are. Modular hardware is a concept where a circuit is separated into modules instead of one complete setup. In electronics design, it means splitting a complete circuit into several PCBs. The PCBs are then connected to each other via cables or connectors.

The concept isn't new if you look at how computers are assembled. A single motherboard with slots for graphics cards, sound cards, and other peripherals is what modular hardware is all about.

In electronics, the modular design trend is highlighted by platforms like Arduino, Raspberry Pi, and BeagleBoard, which allow independent modules to be connected within an ecosystem. The modular concept offers more freedom for designers to explore possibilities at a lower cost.

It is true that there's less hassle when you're creating just one PCB for an entire circuit. However, there are times where taking a modular approach is the better choice.

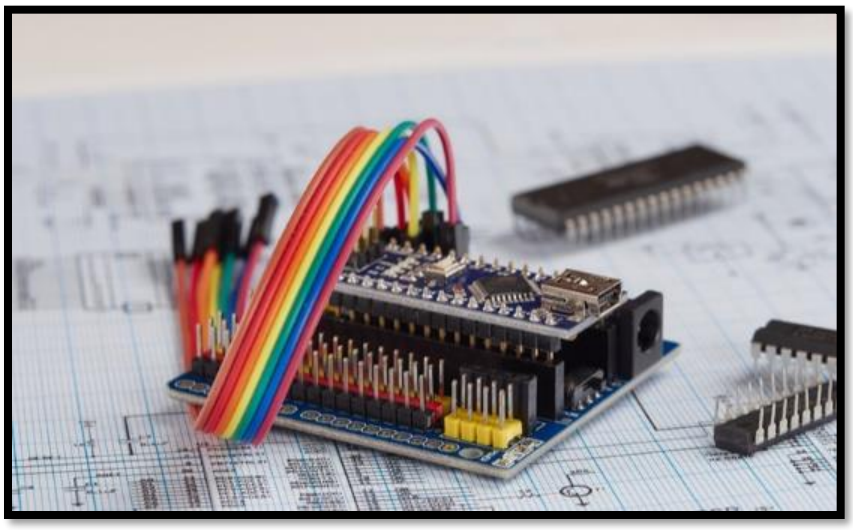

Figure 56: Module-based Flight Controller

6.1.2 Added Flight Controller Modes

Flight modes define how the autopilot responds to remote control input, and how it manages vehicle movement during fully autonomous flight.

The modes provide different types/levels of autopilot assistance to the user (pilot), ranging from automation of common tasks like takeoff and landing, through to mechanisms that make it easier to regain level flight, hold the vehicle to a fixed path or position, etc.

6.1.3 Way-Point Navigation

Satellite Navigational devices function around one premise and that is that they can mark a location anywhere on the earth. The term that we use for that is a "Waypoint" and the dictionary definition is that it is a reference point in physical space.

Before GPS devices we would have to take a map and open the map. We would have to visually look at the map and find the location that we're at or find the location that we want to move to and mark it with a pen. That's exactly the same with a GPS satnav. The points are stored electronically rather than on the map as a pen mark.

There are 3 important things we can do with GPS waypoints which takes it beyond the realms of a mark of the map.

1) From anywhere around the world, whether that's 20 meters away from here or 50 meters, 100 kilometers or 1,000 kilometers, we can come straight back from our starting point, an exact location in a straight line from anywhere.
2) Waypoints can be transferred onto a computer's digital mapping software.
3) We can share our waypoints with other people.

Drone Waypoint GPS Navigation is advanced technology. It is very useful both today and even more so into the future as drones take on new functions and roles in business and other areas.

Waypoint GPS Navigation allows a drone to fly on its own with it's flying destination or points preplanned and configured into the drone remote control navigational software. This instructs the drone where to fly; at what height; the speed to fly at and it can also be configured to hover at each waypoint. It is a route and destination planner for your drone.

6.1.4 Return to Home

The Return to Home feature is a failsafe mechanism that allows your drone to safely go back to a preset home point when it loses connection to the remote controller, the battery is getting too low, or the user chooses to do so. If you don't give your drone enough time to receive a strong GPS signal to set the home point, it will be recorded elsewhere while you are flying it.For example, if you take off without a home point and fly over a lake, the drone will assume that the place is the home point if it first establishes a GPS signal there.

The danger arises when the home point is set in a location where the drone cannot land safely. As a result, always take the time to set the home point before taking off.

6.2 Applications

6.2.1 Aerial Photography

Drones are now being used to capture footage that would otherwise require expensive helicopters and cranes. Fast paced action and sci-fi scenes are filmed by aerial drones, thus making cinematography easier. These autonomous flying devices are also used in real estate and sports photography. Furthermore, journalists are considering the use of drones for collecting footage and information in live broadcasts.

6.2.2 Delivery Drones

Major companies like Amazon, UPS, and DHL are in favor of drone delivery. Drones could save a lot of manpower and shift unnecessary road traffic to the sky. Besides, they can be used over smaller distances to deliver small packages, food, letters, medicines, beverages and the like.

6.2.3 Disaster Management

Drones provide quick means, after a natural or man-made disaster, to gather information and navigate debris and rubble to look for injured victims. Its high definition cameras, sensors, and radars give rescue teams access to a higher field of view, saving the need to spend resources on manned helicopters. Where larger aerial vehicles would prove perilous or inefficient, drones, thanks to their small size, are able to provide a close-up view of areas.

Figure 57: Application of Quadcopter

6.2.4 Precision Agriculture

Farmers and agriculturists are always looking for cheap and effective methods to regularly monitor their crops. The infrared sensors in drones can be tuned to detect crop health, enabling farmers to react and improve crop conditions locally, with inputs of fertilizer or insecticides. It also improves management and effectuates better yield of the crops. In the next few years, nearly 80% of the agricultural market will comprise of drones.

6.2.5 Robots First Respondors

Presence of thermal sensors gives drones night vision and makes them a powerful tool for surveillance. Drones are able to discover the location of lost persons and unfortunate victims, especially in harsh conditions or challenging terrains. Besides locating victims, a drone can drop supplies to unreachable locations in war torn or disaster stricken countries. For example, a drone can be utilized to lower a walkie-talkie, GPS locator, medicines, food supplies, clothes, and water to stranded victims before rescue crews can move them to some place else.

6.2.6 Wildlife Monitoring

Drones have served as a deterrent to poachers. They provide unprecedented protection to animals, like elephants, rhinos, and big cats, a favorite target for poachers. With its thermal cameras and sensors, drones have the ability to operate during the night. This enables them to monitor and research on wildlife without causing any disturbance and provides insight on their patterns, behavior, and habitat.

6.2.7 Law Enforcement

Drones are also used for maintaining the law. They help with the surveillance of large crowds and ensure public safety. They assist in monitoring criminal and illegal activities. In fact, fire investigations, smugglers of migrants, and illegal transportation of drugs via coastlines, are monitored by the border patrol with the help of drones.

As technology advances, drones will become more robust and advanced, accommodating longer flight times and heavier loads. The industry comes with immense opportunities for businesses, gradually becoming inevitable for them. It is, therefore, important for organizations to study the scope of drone technology in their area of business, build the required infrastructure, and test their services across it.

CHAPTER 07: CONCLUSION

In order to conclude the whole discussion of our project work ranging from making quadcopter enable to have a stable flight towards implementing a sort of autonomous features in it, firstly it is important to note the behavior and working of flight controller that is the interfacing of sensors like IMU for inertial navigation with the brain of quadcopter initially being Arduino UNO which was quite suitable for implementation of the auto-leveling flight controller and mainly understanding the basic control and functionalities of flight, but as explained above that to achieve more practical flight mode level we shifted to STM 32 for controlling flight where the larger memory, speed, and availability of pins functionalities for interfacing more sensors like a barometer for altitude level control and GPS for better inertial localization. After it debugs and manipulates the working on real-time telemetry operations through STM 32. Making these changes enabled us to present a stable flight that is ready to tackle practical challenges like a wind gust.
Until now it was the first part of our project, after it considering the need and noticing the limitations of manual flight mode we targeted Artificial intelligence features to embed inside it by making use of a camera and NVIDIA Jetson Nano as a flight computer in which we implemented the VINS MONO algorithm (Visual Inertial Navigation System) which is basically the technique of sensor fusion where data from inertial sensors and camera are integrated together to make quadcopter to achieve localization simultaneous with mapping its surroundings also known as SLAM. For mapping 3-d computer vision geometric reconstruction technique is applied where a moving camera captures data and then through algorithmic procedure in-flight computer the feature points were taken out of it which was projected to get the certain shape, this process continued as the camera moves to obtain knowledge of surroundings. Through this implementation, our drone would be able to select an optimum path for its destination by avoiding obstacles and thus would be able to provide benefits in real-life applications like military, agriculture, cargo services, etc. The flight controller stabilizes the drone, whereas the flight computer gets the quadcopter from point A to B. Both are important for safe and reliable autonomous drone simulation.

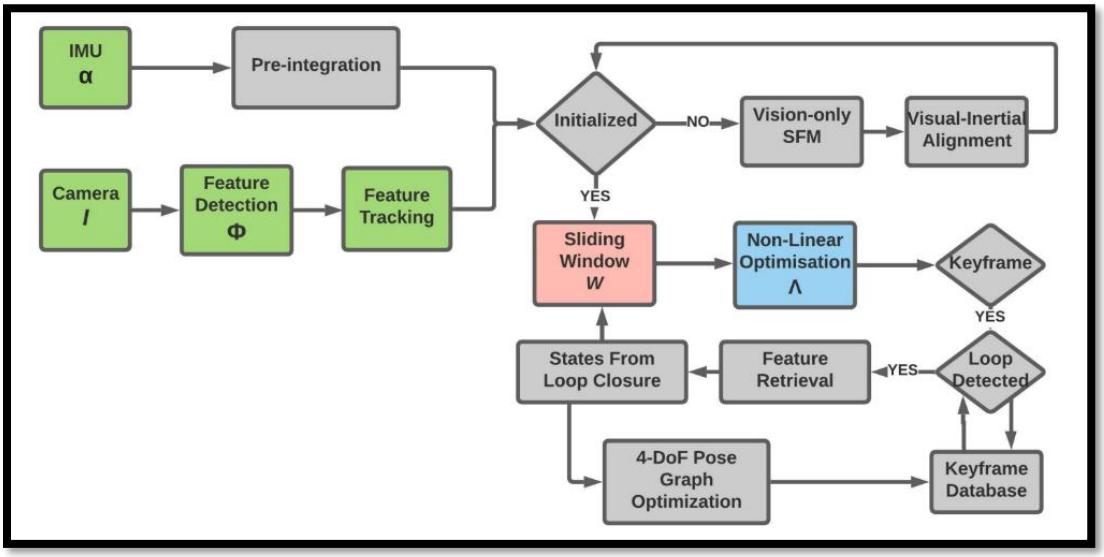

Figure 58: Summarized flowchart of Implementing SLAM

REFERENCES

[1] G. Klein and D. Murray, "Parallel tracking and mapping for small arworkspaces," in Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on. IEEE, 2007, pp. 225–234.
[2] C. Forster, M. Pizzoli, and D. Scaramuzza, "SVO: Fast semi-direct monocular visual odometry," in Proc. of the IEEE Int. Conf. on Robot. and Autom., Hong Kong, China, May 2014.
[3] J. Engel, T. Schops, and D. Cremers, "Lsd-slam: Large-scale di-Rect-monocular slam," in European Conference on Computer Vision. Springer International Publishing, 2014, pp. 834–849.
[4] R. Mur-Artal, J. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," IEEE Transactions on Robotics, vol. 31, no. 5, pp. 1147–1163, 2015.
[5] J. Engel, V. Koltun, and D. Cremers, "Direct sparse odometry," IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017.
[6] D. Galvez-L ´ opez and J. D. Tard ´ os, "Bags of binary words for fast ´ place recognition in image sequences," IEEE Transactions on Robotics, vol. 28, no. 5, pp. 1188–1197, October 2012.
[7] S. Shen, N. Michael, and V. Kumar, "Tightly-coupled monocular visualinertial fusion for autonomous flight of rotorcraft MAVs," in Proc. Of the IEEE Int. Conf. on Robot. and Autom., Seattle, WA, May 2015.
[8] Z. Yang and S. Shen, "Monocular visual–inertial state estimation with online initialization and camera–imu extrinsic calibration," IEEE Transactions on Automation Science and Engineering, vol. 14, no. 1, pp. 39–51, 2017.
[9] T. Qin and S. Shen, "Robust initialization of monocular visual-inertial estimation on aerial robots." in Proc. of the IEEE/RSJ Int. Conf. on Intell. Robots and Syst., Vancouver, Canada, 2017, accepted.
[10] P. Li, T. Qin, B. Hu, F. Zhu, and S. Shen, "Monocular visual-inertial
state estimation for mobile augmented reality." in Proc. of the IEEE Int. Sym. on Mixed adn Augmented Reality, Nantes, France, 2017, accepted.
[11] S. Weiss, M. W. Achtelik, S. Lynen, M. Chli, and R. Siegwart, "Realtime onboard visual-inertial state estimation and self-calibration of mavs
in unknown environments," in Robotics and Automation (ICRA), 2012 IEEE International Conference on. IEEE, 2012, pp. 957–964.
[12] S. Lynen, M. W. Achtelik, S. Weiss, M. Chli, and R. Siegwart, "A robust and modular multi-sensor fusion approach applied to mav navigation," in Proc. of the IEEE/RSJ Int. Conf. on Intell. Robots and Syst. IEEE, 2013, pp. 3923–3929.
[13] A. I. Mourikis and S. I. Roumeliotis, "A multi-state constraint Kalman filter for vision-aided inertial navigation," in Proc. of the IEEE Int. Conf. on Robot. and Autom., Roma, Italy, Apr. 2007, pp. 3565–3572.
[14] M. Li and A. Mourikis, "High-precision, consistent EKF-based visualinertial odometry," Int. J. Robot. Research, vol. 32, no. 6, pp. 690–711,May 2013.
[15] M. Bloesch, S. Omari, M. Hutter, and R. Siegwart, "Robust visual inertial odometry using a direct ekf-based approach," in Proc. of the IEEE/RSJ Int. Conf. on Intell. Robots and Syst. IEEE, 2015, pp. 298–304.
[16] S. Leutenegger, S. Lynen, M. Bosse, R. Siegwart, and P. Furgale, "Keyframe-based visual-inertial odometry using nonlinear optimization," Int. J. Robot. Research, vol. 34, no. 3, pp. 314–334, Mar. 2014.

[17] R. Mur-Artal and J. D. Tardos, “Visual-inertial monocular SLAM with map reuse,” arXiv preprint arXiv:1610.05949, 2016.
[18] K. Wu, A. Ahmed, G. A. Georgiou, and S. I. Roumeliotis, “A square root inverse filter for efficient vision-aided inertial navigation on mobile devices.” in Robotics: Science and Systems, 2015.
[19] M. K. Paul, K. Wu, J. A. Hesch, E. D. Nerurkar, and S. I. Roumeliotis, “A comparative analysis of tightly-coupled monocular, binocular, and stereo vins,” in Proc. of the IEEE Int. Conf. on Robot. and Autom., Singapore, May 2017.
[20] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, “isam2: Incremental smoothing and mapping using the bayes tree,” The International Journal of Robotics Research, vol. 31, no. 2, pp. 216–235, 2012.
[21] V. Usenko, J. Engel, J. Stuckler, and D. Cremers, “Direct visual-inertial ¨odometry with stereo cameras,” in Proc. of the IEEE Int. Conf. on Robot. and Autom. IEEE, 2016, pp. 1885–1892.
[22] T. Lupton and S. Sukkarieh, “Visual-inertial-aided navigation for highdynamic motion in built environments without initial conditions,” IEEE Trans. Robot., vol. 28, no. 1, pp. 61–76, Feb. 2012.
[23] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, “IMU preintegration on manifold for efficient visual-inertial maximum-a-posteriori estimation,” in Proc. of Robot.: Sci. and Syst., Rome, Italy, Jul. 2015.
[24] S. Shen, Y. Mulgaonkar, N. Michael, and V. Kumar, “Initializationfree monocular visual-inertial estimation with application to autonomous MAVs,” in Proc. of the Int. Sym. on Exp. Robot., Marrakech, Morocco, Jun. 2014.
[25] A. Martinelli, “Closed-form solution of visual-inertial structure from motion,” Int. J. Comput. Vis., vol. 106, no. 2, pp. 138–152, 2014.
[26] J. Kaiser, A. Martinelli, F. Fontana, and D. Scaramuzza, “Simultaneous state initialization and gyroscope bias calibration in visual inertial aided navigation,” IEEE Robotics and Automation Letters, vol. 2, no. 1, pp.18–25, 2017.
[27] M. Faessler, F. Fontana, C. Forster, and D. Scaramuzza, “Automatic reinitialization and failure recovery for aggressive flight with a monocular vision-based quadrotor,” in Proc. of the IEEE Int. Conf. on Robot. And Autom. IEEE, 2015, pp. 1722–1729.
[28] H. Strasdat, J. Montiel, and A. J. Davison, “Scale drift-aware large scale monocular slam,” Robotics: Science and Systems VI, 2010.
[29] B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” in Proc. of the Intl. Joint Conf. on Artificial Intelligence, Vancouver, Canada, Aug. 1981, pp. 24–28.
[30] J. Shi and C. Tomasi, “Good features to track,” in Computer Vision and Pattern Recognition, 1994. Proceedings CVPR’94., 1994 IEEE Computer Society Conference on. IEEE, 1994, pp. 593–600.
[31] R. Hartley and A. Zisserman, Multiple view geometry in computer vision. Cambridge university press, 2003.
[32] A. Heyden and M. Pollefeys, “Multiple view geometry,” Emerging Topics in Computer Vision, 2005.
[33] D. Nister, “An efficient solution to the five-point relative pose problem,” IEEE transactions on pattern analysis and machine intelligence, vol. 26,no. 6, pp. 756–770, 2004.
[34] T. Liu and S. Shen, “Spline-based initialization of monocular visualinertial state estimators at high altitude,” IEEE Robotics and Automation Letters, 2017, accepted.

[35] V. Lepetit, F. Moreno-Noguer, and P. Fua, “Epnp: An accurate o (n) solution to the pnp problem,” International journal of computer vision, vol. 81, no. 2, pp. 155–166, 2009.
[36] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, “Bundle adjustmenta modern synthesis,” in International workshop on vision algorithms. Springer, 1999, pp. 298–372.
[37] P. Huber, “Robust estimation of a location parameter,” Annals of Mathematical Statistics, vol. 35, no. 2, pp. 73–101, 1964.
[38] S. Agarwal, K. Mierle, and Others, “Ceres solver,” http://ceres-solver.org.
[39] G. Sibley, L. Matthies, and G. Sukhatme, “Sliding window filter with application to planetary landing,” J. Field Robot., vol. 27, no. 5, pp. 587–608, Sep. 2010.
[40] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, “Brief: Binary robust independent elementary features,” Computer Vision–ECCV 2010, pp. 778–792, 2010.
[41] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, “The euroc micro aerial vehicle datasets,” The International Journal of Robotics Research, 2016.
[42] C. Mei and P. Rives, “Single view point omnidirectional camera calibration from planar grids,” in Robotics and Automation, 2007 IEEE International Conference on. IEEE, 2007, pp. 3945–3950.
[43] L. Heng, B. Li, and M. Pollefeys, “Camodocal: Automatic intrinsic and extrinsic calibration of a rig with multiple generic cameras and odometry,” in Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, 2013, pp. 1793–1800.
[44] Y. Lin, F. Gao, T. Qin, W. Gao, T. Liu, W. Wu, Z. Yang, and S. Shen, “Autonomous aerial navigation using monocular visual-inertial fusion,” J. Field Robot., vol. 00, pp. 1–29, 2017
[55] J. Brooking, "Brooking.net," 31 01 2014. [Online]. Available: http://www.brokking.net/ymfc-32_main.html. [Accessed 31 10 2021].

APPENDIX

A1

MPU datasheet reference for gyroscope reading

VDD = 2.375V-3.46V, VLOGIC (MPU-6050 only) = 1.8V±5% or VDD, T_A = 25°C

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
GYROSCOPE SENSITIVITY						
Full-Scale Range	FS_SEL=0		±250		°/s	
	FS_SEL=1		±500		°/s	
	FS_SEL=2		±1000		°/s	
	FS_SEL=3		±2000		°/s	
Gyroscope ADC Word Length			16		bits	
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(°/s)	
	FS_SEL=1		65.5		LSB/(°/s)	
	FS_SEL=2		32.8		LSB/(°/s)	
	FS_SEL=3		16.4		LSB/(°/s)	
Sensitivity Scale Factor Tolerance	25°C	-3		+3	%	
Sensitivity Scale Factor Variation Over Temperature			±2		%	
Nonlinearity	Best fit straight line; 25°C		0.2		%	
Cross-Axis Sensitivity			±2		%	

MPU datasheet reference for accelerometer reading:

VDD = 2.375V-3.46V, VLOGIC (MPU-6050 only) = 1.8V±5% or VDD, T_A = 25°C

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS	NOTES
ACCELEROMETER SENSITIVITY						
Full-Scale Range	AFS_SEL=0		±2		*g*	
	AFS_SEL=1		±4		*g*	
	AFS_SEL=2		±8		*g*	
	AFS_SEL=3		±16		*g*	
ADC Word Length	Output in two's complement format		16		bits	
Sensitivity Scale Factor	AFS_SEL=0		16,384		LSB/*g*	
	AFS_SEL=1		8,192		LSB/*g*	
	AFS_SEL=2		4,096		LSB/*g*	
	AFS_SEL=3		2,048		LSB/*g*	
Initial Calibration Tolerance			±3		%	
Sensitivity Change vs. Temperature	AFS_SEL=0, -40°C to +85°C		±0.02		%/°C	
Nonlinearity	Best Fit Straight Line		0.5		%	
Cross-Axis Sensitivity			±2		%	