Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in
Ingegneria Aerospaziale
A.a. 2024/2025

Sessione di laurea Dicembre 2025

Configurazione e validazione di un drone con autopilota PX4 in facility di volo indoor

Relatori:

Dr. Stefano Primatesta

Riccardo Enrico

Candidato:

Matteo Bisceglia

Abstract

Questa tesi si pone l'obiettivo di configurare un quadricottero dotato di autopilota PX4 e companion computer nella facility di volo indoor del Dipartimento di Ingegneria Meccanica e Aerospaziale del Politecnico di Torino, dotata di sistema di tracciamento Vicon. In una prima fase, è stata realizzata una simulazione in ambiente Ubuntu con gli stessi strumenti poi usati nel volo reale quali ROS2, PX4 Autopilot, Micro XRCE-DDS e QGroundControl, andando a verificare la corretta comunicazione tra questi e la capacità del drone di volare in ambiente simulativo Gazebo grazie ai dati di odometria che successivamente sarebbero stati forniti dal sistema Vicon. Con lo scopo di testare la configurazione, la posizione del drone in Gazebo è stata trasmessa a ROS2 tramite un bridge apposito ed è stata quindi fornita al PX4 tramite un opportuno topic, andando poi a verificare l'azione di filtraggio con i sensori del drone dell'autopilota. Tale configurazione in ambiente simulato rappresenta anche un ambiente preliminare di test in cui svolgere le prove sperimentali, propedeutiche ai test reali nella facility di volo indoor. Nella seconda fase si è passati al drone reale. Sul companion computer si è installato e settato quanto necessario per interfacciare il flight controller con il sistema Vicon. Sono stati scelti gli opportuni parametri dell'autopilota già inizialmente validati in simulazione, ed è stata stabilita la comunicazione tra il drone e il sistema Vicon, per poi effettuare i test sperimentali nella gabbia.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni i velivoli a pilotaggio remoto hanno avuto una diffusione estremamente rapida, trovando impiego in ambito civile, industriale e accademico. La crescente maturità delle tecnologie di controllo e percezione ha reso possibile lo sviluppo di droni autonomi in grado di svolgere missioni complesse senza supervisione costante dell'operatore umano. Tuttavia, la validazione sperimentale di algoritmi di navigazione, stima e controllo rimane un aspetto critico: le condizioni atmosferiche, le limitazioni normative, i rischi operativi e la difficoltà nel riprodurre test identici rendono spesso complesso verificare in modo sicuro e ripetibile il comportamento del velivolo in ambiente reale.

Per rispondere a queste esigenze, diversi gruppi di ricerca europei hanno sviluppato infrastrutture indoor dedicate alla sperimentazione controllata. Tra le più note si trovano la Flying Machine Arena dell'ETH Zurich e la CyberZoo della TU Delft, entrambe dotate di sistemi di motion capture ad alta precisione e ampi volumi di volo per droni autonomi. A queste si affianca anche l'Innovative Air Mobility Lab di Tecnalia, che mette a disposizione una grande area indoor equipaggiata con sistemi di tracking ottico e strumenti avanzati per la valutazione del comportamento dei velivoli in condizioni controllate, incluso un generatore di profili di vento per simulare ambienti operativi realistici. La presenza di infrastrutture di questo tipo conferma l'importanza, a livello europeo, di disporre di spazi sicuri e ripetibili per lo sviluppo e la validazione delle tecnologie di navigazione autonoma.

In questo stesso contesto si inserisce la gabbia di volo indoor del Politecnico di Torino, dotata di sistema Vicon, che permette di ottenere posizione e velocità del drone nei 6 gradi di libertà con elevata accuratezza e frequenze elevate, offrendo un ambiente ideale per la sperimentazione di algoritmi di guida e stima. Il lavoro presentato in questa tesi si concentra sulla configurazione, integrazione e validazione di un quadricottero equipaggiato con autopilota PX4 e companion computer per

operare all'interno di questa facility, utilizzando il sistema Vicon come sorgente primaria di odometria.

La prima fase del lavoro ha previsto la creazione di una simulazione completa in ambiente Ubuntu utilizzando Gazebo, ROS 2, PX4 Autopilot, Micro XRCE-DDS e QGroundControl, riproducendo fedelmente il setup che sarebbe stato poi utilizzato nelle prove reali. In simulazione è stata verificata la corretta comunicazione tra i vari moduli software, il funzionamento del sistema di odometria e la capacità del filtro EKF2 di fondere i dati simulati in maniera coerente. Successivamente, la stessa architettura è stata replicata sul drone reale, realizzando nodi ROS 2 dedicati per ricevere la posa dal server Vicon, trasformarla nel formato compatibile con PX4 e trasmetterla all'autopilota tramite Micro XRCE-DDS. Infine, sono stati condotti test sperimentali nella gabbia indoor per valutare l'efficacia della configurazione, analizzare il comportamento del filtro EKF2 in presenza di dati reali e studiare l'effetto dei principali parametri dell'autopilota.

La tesi è strutturata nel seguente modo. Il Capitolo 1 descrive il kit PX4 Vision Autonomy Development Kit e le sue principali componenti hardware; il Capitolo 3 presenta il sistema Vicon utilizzato nella facility indoor; il Capitolo 4 illustra l'architettura software basata su PX4, ROS 2 e Micro XRCE-DDS; il Capitolo 5 tratta la simulazione SITL e le prove in ambiente virtuale; il Capitolo 6 riporta la configurazione del drone reale e i test sperimentali svolti nella gabbia; il Capitolo 7 conclude il lavoro discutendo i risultati ottenuti e i possibili sviluppi futuri.

Capitolo 2

PX4 Vision Autonomy Development Kit

Figura 2.1: Drone PX4 Vision Autonomy Development Kit

Il drone utilizzato per il progetto di tesi è il PX4 Vision Autonomy Development Kit [1] che permette di eseguire operazioni di volo autonomo[1] e di computer vision [2] anche se quest'ultima funzionalità non verrà sfruttata in questo caso specifico poiché non verrà usata la camera del drone. In questo capitolo verranno trattate le

[1]La capacità di volare senza che l'operatore umano impartisca direttamente i comandi

[2]La capacità del drone, nello specifico del companion computer, di interpretare le immagini provenienti dalle telecamere

principali caratteristiche hardware ad alto livello del modello di veicolo in esame. Di seguito si descrive il contenuto del kit.

- Componenti principali:
  - 1x Pixhawk 4 or Pixhawk 6C (for v1.5) flight controller
  - 1x PMW3901 sensore optical flow
  - 1x TOF distance sensor a infrarossi (PSK-CM8JL65-CC5)
  - 1x Structure Core depth camera
    - Vision camera con campo visivo di 160°
    - Camera stereo a infrarossi
    - Onboard IMU[3]
    - Potente processore NU3000 Multi-core depth
  - 1x UP Core computer (4GB di memoria & 64GB eMMC)
    - Intel® Atom™ x5-z8350 (up to 1.92 GHz)
    - Sistemi operativi compatibili: Microsoft Windows 10 full version, Linux (ubilinux, Ubuntu, Yocto), Android
    - FTDI UART connesso al flight controller
    - USB1: USB3.0 porta utilizzata per l'esecuzione di PX4 avoidance environment da chiavetta USB2.0 (la connessione di una periferica USB3.0 può disturbare il GPS).
    - USB2: porta USB2.0 con connettore JST-GH. Può essere utilizzata per una seconda camera, LTE, etc. (o tastiera/mouse in fase di sviluppo)
    - USB3: USB2.0 con porta JST-GH connessa alla depth camera
    - HDMI: HDMI out
    - SD card slot
    - WiFi 802.11 b/g/n @ 2.4 GHz (connesso all'antenna esterna #1). Permette al computer di accedere alle reti WiFi per connettersi a Internet o per gli aggiornamenti.
- Specifiche meccaniche
  - Telaio: Full 5mm 3k twill in fibra di carbonio
  - Motori: T-MOTOR KV1750
  - ESC: BEHEli-S 20A ESC

[3]Inertial Measurement Unit

  - GPS: M8N GPS module
  - Power module: Holybro PM07
  - Larghezza alla base: 286mm
  - Massa: 854 grammi senza batteria né eliche
  - Telemetria: ESP8266 connessa al flight controller (tramite l'antenna esterna #2). Permette la connessione wireless con la ground station

- Una chiavetta USB2.0 contenente il software per eseguire:
  - Ubuntu 18.04 LTS
  - ROS Melodic
  - Occipital Structure Core ROS driver
  - MAVROS
  - PX4 avoidance

- Cavi assortiti, 8x eliche, 2x battery straps (installati) e altri accessori (che possono essere utilizzati per connettere periferiche aggiuntive).

Nel kit non sono invece inclusi la batteria ed il radiocomando. La batteria deve essere una 4S LiPo[4] con connettore femmina XT60 e deve avere una lunghezza inferiore ai 115 mm per entrare tra il power connector e l'asta del GPS. Il radiocomando invece dovrà essere compatibile con il PX4 e nel nostro caso viene utilizzato un FrSky Taranis QX7 e un ricevitore FrSky R-XSR reciver montato sul drone. E' inoltre necessaria la ground station QGroundControl.
In questo capitolo verranno dunque descritti i componenti principali del PX4 Vision Autonomy Development Kit, a prescindere da quale sarà la configurazione usata nella fase di test del progetto dove nello specifico non verranno utilizzati i sensori: GNSS, optical flow, distance sensor, depth camera.

[4]Lithium Polymer a 4 celle

2.1 Flight Controller

Il flight controller adempie principalmente a:

- leggere i dati dei sensori o da sistemi di motion capture esterni (Vicon);
- fondere tramite filtro di Kalman (EKF2[5]) i dati dei sensori e della motion capture (se presente) per avere una stima univoca;
- controllare il volo (stabilizzazione, controllo di posizione e velocità, ecc.);
- gestire la comunicazione dell'autopilota con l'esterno.

Figura 2.2: Pixhawk 6C

Nel caso in esame l'autopilota è un Pixhawk 6C [2] che è un computer embedded ad alte prestazioni che ospita il firmware dell'autopilota. È equipaggiato di un microcontrollore STMicroelectronics STM32H743, basato su architettura ARM Cortex-M7 operante a 480 MHz dotato di 2 MB di memoria flash e 1 MB di RAM, la scheda dispone inoltre di un processore ausiliario STM32F103 dedicato alle funzioni I/O e alla gestione delle comunicazioni di basso livello. Questa configurazione consente l'esecuzione di algoritmi di controllo complessi con elevata frequenza di aggiornamento, assicurando la stabilità del veicolo anche in presenza di disturbi o variazioni dinamiche significative.

La scheda presenta al suo interno diversi sensori inerziali e ambientali:

- Due unità IMU ridondanti (ICM-42688-P e BMI055) che forniscono misure di accelerazione e velocità che poi vengono integrate per risalire all'odometria del drone
- Un magnetometro IST8310 per la determinazione dell'attitude calcolata a partire dal nord magnetico terrestre

[5]Extended Kalman Filter

- Un barometro MS5611 che fornisce la pressione che verrà poi usata per la determinazione della quota

La presenza di più tipologie di sensori che forniscono uno stesso tipo di dato e la loro ridondanza permette di avere una misura meno affetta da rumore e di migliore qualità in quanto il filtro EKF2 fonde tutti i dati derivanti dai sensori per ottenere un'unica misura dell'odometria. C'è da notare come i dati "grezzi" provenienti dai vari sensori (con diverse unità di misura tra loro: Pa, μT, m/s^2 ecc...) non vengono convertiti subito in un dato odometrico relativo al singolo sensore poiché questo risulterebbe essere troppo dispendioso per l'autopilota e non sarebbe conveniente a livello numerico. Quindi l'autopilota va a cambiare la stima dello stato complessivo del drone confrontando i dati grezzi dei sensori con un modello numerico che gli permette di capire quanto cambiare questa stima.
Essendo la Pixhawk 6C un sistema dual processor, essa è composta da:

- FMU[6] Processor: STM32H743 o 32 Bit Arm® Cortex®-M7, 480MHz, 2MB memory, 1MB SRAM Figura 5: Flight controller Pixhawk 6C 21. E' il Processore principale che quindi è imputato ad effettuare le operazioni a maggiore complessità computazionale e a priorità temporale alta, nello specifico si occupa di: far girare il firmware PX4, gestione sensori, comunicazioni esterne, logging e supervisione di sistema monitorando lo stato dell' IO processor e intervenendo in caso di failure

- IO Processor: STM32F103 o 32 Bit Arm® Cortex®-M3, 72MHz, 64KB SRAM. E' un processore dedicato all'hardware di basso livello per scaricare lavoro dalla FMU e si occupa di: gestione delle uscite PWM[7] verso gli ESC, Gestione ingressi PWM/SBUS del radiocomando, failsafe dei motori in caso di perdita di segnale da FMU e inizializzazione e calibrazione base all'avvio

Il flight controller presenta le seguenti interfacce:

- 16- PWM servo outputs raggruppati in due porte, 8 nella porta I/O PWM OUT e 8 nella porta FMU PWM OUT

- 3 porte seriali con scopo generico
  - TELEM1 - Full flow control, con corrente massima 1.5A
  - TELEM2 - Full flow control
  - TELEM3

[6]Flight Management Unit

[7]Pulse Width Modulation

- 2 porte GPS
  - GPS1 - Full GPS port (GPS con safety switch)
  - GPS2 - Basic GPS port
- 1 porta I2C che supporta la calibrazione I2C EEPROM dedicata e collocata sul sensor module
- 2 CAN Buses con controllo silenzioso individuale o ESC RX-MUX control
- 2 porte di debug:
  - FMU Debug
  - I/O Debug
- R/C input dedicata a Spektrum / DSM e S.BUS, CPPM, analog / PWM RSSI
- S.BUS output dedicato
- 2 Power input ports (Analog)

Il Pixhawk 6C viene mantenuto ad una temperatura di 40 85° grazie a resistori per il riscaldamento, permettendo così un opportuno funzionamento delle IMU. Inoltre la presenza di sensori inerziali all'interno di esso fa si che debba essere posizionato in prossimità del centro di grafità dell'aeromobile.

2.2 Power Management Board

Per Power Management Board[3] si intende un componente che svolge la funzione di:

- Power Module - componente che:
  - Regola la tensione - la batteria LiPo4S fornisce una tensione variabile tra i 16.8 e i 13V a seconda del suo livello di carica, il flight controller ha bisogno invece di 5V stabili in ingresso e questi gli vengono forniti tramite la porta POWER1 o POWER2.
  - Misura tensione e corrente della batteria, questi dati vengono inviati sempre tramite il cavo POWER al PX4 che stima la carica restante (e la proietta in QGroundControl) e attiva il failsafe in caso di carica bassa
- Power Distribution Board - è la piattaforma di distriuzione della potenza che preleva corrente ad alto amperaggio (fino a 120A) dalla batteria per traferirla agli ESC e ai motori

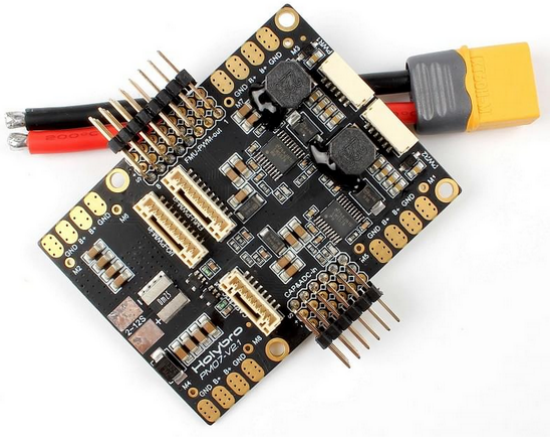

Figura 2.3: Power module

I flight controller di precedente generazione avevano dei pin più grandi da cui quindi uscivano dei connettori più grandi in grado di trasportare una maggiore corrente e che potevano portare corrente direttamente ai motori (i motori di tali droni richiedevano correnti minori). Invece i connettori JST-GH che allo stato dell'arte escono dalla flight controller possono portare una bassa corrente poiché sono pensati per un segnale PWM a bassa potenza che ha l'unico scopo di far arrivare un'informazione. Come rappresentato in figura 2.4:

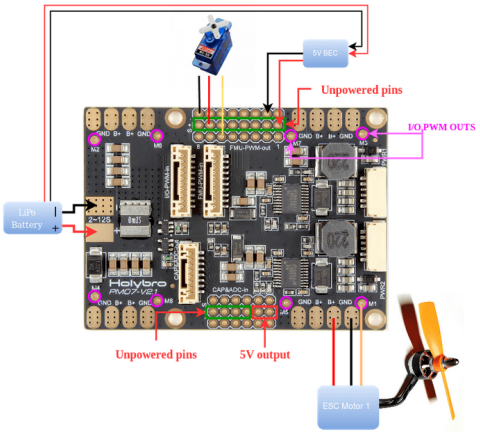

Figura 2.4: Cablaggio PM07

- In uscita dalla batteria c'è un segnale di alta potenza che quindi non può essere portato dai JST-GH, ma dal connettore XT-60

- In uscita dalla flight controller si ha un segnale che va nella *FMU-PWM-out* che è un segnale PWM di bassa potenza che poi viene smistato verso gli ESC per indicare quanta corrente deve entrare nei motori
- Negli ESC entrano anche segnali ad alta corrente provenienti dai pad *B+* e *GND* che fanno si che ai motori arrivi la potenza necessaria per il loro effettivo funzionamento

2.3 Companion Computer

Il companion computer impiegato nel PX4 Vision Autonomy Development Kit è l'UP Core[4], un'unità di elaborazione compatta basata su architettura Intel® Atom™ progettata per applicazioni embedded che richiedono elevata capacità computazionale a fronte di dimensioni e consumi ridotti. Il companion computer è totalmente distinto dal flight controller, ma ci può comunicare tramite: USB-seriale, UART delle porte TELEM, MAVLink via rete, CAN bus oppure interfacce di basso livello come I2C/SPI. Infatti la sua fonte di alimentazione non è la PM07 ma la sua carrier board. La carrier board è una struttura che:

- fornisce alimentazione all'UP Core
- espone porte USB, CSI e Wi-Fi
- permette il montaggio fisico del companion

Essa si connette ad alcune linee di un connettore a 100 pin ad alta velocità posizionato sotto l'UP core, rendendolo così expandable. Il companion computer è compatibile con molti sistemi operativi, tra cui Linux.

2.4 Modulo GNSS

Nelle missioni outdoor l'impiego di un sistema GNSS è praticamente indispensabile: è ciò che permette all'autopilota di stabilire la posizione del veicolo e di seguirne gli spostamenti con un'accuratezza adeguata alla navigazione. Per questa ragione il modulo GNSS fornisce all'autopilota le informazioni necessarie per operare correttamente. Il flight controller Pixhawk 6C, inoltre, è in grado di gestire fino a due ricevitori GNSS allo stesso tempo, collegabili tramite BUS CAN oppure tramite UART. Se si montano entrambi, non svolgono esattamente lo stesso ruolo: il modulo principale è quello essenziale, soprattutto quando non si riceve la posizione da un sistema di motion capture indoor; il secondo, invece, è opzionale e viene aggiunto dall'utente come ridondanza.

PX4 supporta un'ampia gamma di ricevitori GNSS e praticamente tutte le principali costellazioni, incluse quelle SBAS. Oltre al ricevitore satellitare, questi moduli integrano anche un magnetometro che l'autopilota utilizza per ricavare la direzione e lo yaw del velivolo sfruttando il campo magnetico terrestre. Per evitare disturbi su questa misura, è importante installare il modulo il più lontano possibile da fonti di interferenza elettromagnetica come motori e circuiti di potenza.

Figura 2.5: Modulo GNSS e Pixhawk

Il controller verifica anche che il modulo sia montato correttamente, cioè rivolto verso l'alto e orientato in avanti. Se non è possibile installarlo così e si rende necessario ruotarlo, PX4 può riconoscere automaticamente rotazioni di almeno 45°. Per inclinazioni diverse bisogna invece impostare a mano gli angoli di Eulero appropriati. Un altro fattore critico è la distanza del modulo GNSS dal centro di gravità del drone: più questa aumenta, più diventa importante indicarla tramite i parametri EKF2_GPS_POS_X, EKF2_GPS_POS_Y e EKF2_GPS_POS_Z. Questo aspetto è particolarmente rilevante quando si utilizza un sistema RTK a precisione centimetrica. Senza questa informazione, l'autopilota potrebbe assumere che il ricevitore si trovi nel centro del velivolo, introducendo compensazioni non necessarie durante il mantenimento della posizione.

I moduli GNSS includono anche alcuni elementi di sicurezza e segnalazione: il safety switch, un buzzer e un LED RGB. Prima di far decollare il drone è necessario attivare fisicamente il safety switch, mentre buzzer e LED servono a comunicare lo stato del veicolo tramite suoni e colori, rendendo immediato capire se il drone è acceso, armato o ha rilevato un errore. La figura successiva mostra il modulo Holybro M8N[5][6] utilizzato sul drone, insieme al safety switch e al LED.

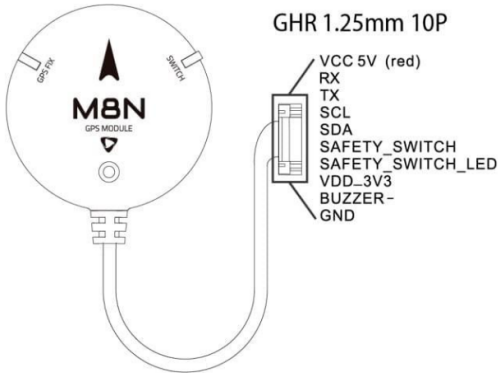

Figura 2.6: Modulo GNSS M8N

C'è infine una differenza pratica nel modo in cui vengono collegati i due moduli GNSS: quello primario, che gestisce più funzioni, deve essere connesso alla porta principale dell'autopilota, che dispone di più terminali; il modulo secondario, che funge solo da ricevitore GNSS di riserva, si collega invece alla porta dedicata, dotata di un cablaggio più semplice che non supporta le altre funzionalità. La rappresentazione nella figura chiarisce bene questa distinzione.

2.5 Camera

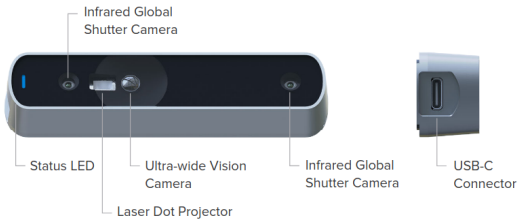

Figura 2.7: Structure Core Depth Camera

La presenza di una camera a bordo del drone consente al veicolo di acquisire informazioni visive dell'ambiente circostante fondamentali per compiti di navigazione autonoma / rilevamento ostacoli e di individuazione della pose del drone (posizione e attitude). Nel caso del PX4 Vision Autonomy Development Kit il modulo di visione utilizzato è quello della Structure Core Depth Camera[7] che è costituita da:

- 160° wide vision camera: una camera che cattura immagini ad ampio Field of View in 2 dimensioni

- Camere stereo ad infrarossi: una coppia di sensori infrarossi che acquisiscono immagini simultaneamente e da posizioni diverse, questo consente al depth processor di effettuare una triangolazione per acquisire la profondità degli oggetti
- Onboard IMU: serve a migliorare le misure dello stereo matching soprattutto in caso di movimenti troppo rapidi del drone o non perfetto fissaggio della camera
- NU3000 multi-core processore di profondità: un processore dedicato che esegue in hardware gli algoritmi di matching stereo, calcolo della profondità, funsione visual-inertial (stereo + IMU). Fornisce in uscita il calcolo della posa

2.6 Optical Flow

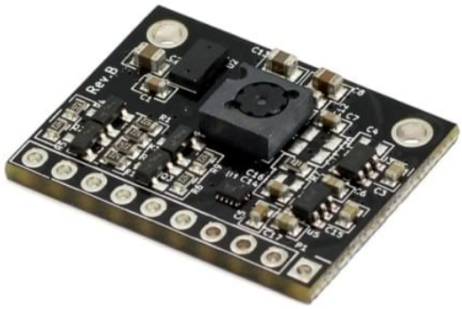

Figura 2.8: Optical Flow PMW3901

L'optical flow è un piccolo sensore a basso consumo di potenza, particolarmente utile in assenza di GPS. Esso è costituito da:

- sistema ottico: formato da una lente e da un sensore di immagine monocromatico a bassa definizione
- ASIC[8] per il calcolo del flusso ottico

L'ASIC prende le immagini acquisite dal sistema ottico in frame successivi e confrontandole riesce ad ottenere una variazione di posizione laterale ΔX e ΔY espressa in pixel, sarà poi l'autopilota infatti ad usare questo dato per calcolare velocità e successivamente posizioni, che il sensore non può conoscere perché non

[8]Application Specific Integrated Circuit, un circuito progettato specificatamente per questa applicazione

ha la distanza a cui si trovano gli oggetti.
Nel veicolo oggetto di studio è montato un PMW3901[8][9] che è in grado di adempiere al suo posto per oggetti che stanno ad una distanza dalla camera a partire dagli 80 mm, il costruttore infatti non impone costrizioni per quel che riguarda il limite superiore della distanza degli oggetti.

2.7 Distance Sensor

Figura 2.9: Distance sensor

Il distance sensor è uno strumento essenziale per leggere i dati di variazione di posizione laterale espressa in pixel in uscita dall'optical flow. Optical flow e distance sensor sono distinti a livello hardware ma i loro dati vengono presi dall'autopilota e integrati tra loro. Il distance sensor utilizzato è il Lanbao PSK-CM8JL65-CC5 ToF Infrared Distance Measuring Sensor[10]. Questo sensore rileva la distanza degli oggetti andando ad individuare la differenza di fase tra il fascio infrarosso emesso e quello che gli ritorna dopo aver rimbalzato sull'oggetto di interesse. E' caratterizzato da:

- HALIOS photoelectric detection technology che conferisce al sensore una forte adattabilità ad ambienti outdoor luminosi e a variazioni di temperatura
- una limitata influenza del colore e della rugosità della superficie di interesse sull'output del sensore
- ridotte dimensioni: 38 mm x 18 mm x 7 mm, peso $\leq$ 10g
- un range di 0.17 m -8 m

2.8 Cablaggi

L'architettura che appare nel diagramma di figura 2.10[11] mostra chiaramente come i vari componenti del sistema si collegano tra loro. Usano una serie di

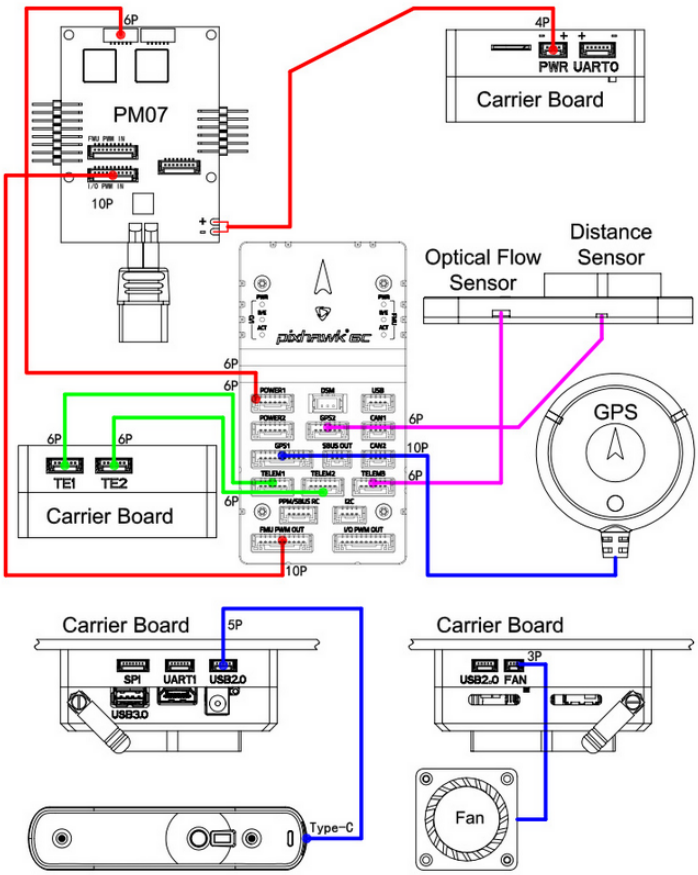

Figura 2.10: Cablaggi PX4 Vision Dev Kit V1.5

connettori specifici. Ognuno ha un ruolo preciso per distribuire l'alimentazione e trasmettere i segnali. La scheda PM07 riceve la potenza dalla batteria attraverso il connettore XT60. Poi ridistribuisce la tensione regolata agli altri elementi del drone. Lo fa con connettori JST-GH a 6 e 10 pin. Le uscite a 6 pin servono per alimentare il Pixhawk 6C. Lo collegano alle porte POWER1 e POWER2. Oltre ai fili per tensione e massa, questi connettori portano anche le linee analogiche. Quelle linee riportano al flight controller la tensione e la corrente della batteria. Il connettore a 10 pin della PM07 gestisce più segnali. Espone canali di ingresso per PWM e linee di comunicazione extra. Sono dedicate ai controlli motore o a dispositivi che richiedono più linee. In questa parte del cablaggio ci sono anche le intestazioni FMU PWM-IN e I/O PWM-IN. Rappresentano i punti dove la PM07 riceve i segnali PWM. Li genera il processore principale del Pixhawk, che è l'FMU. Oppure il processore ausiliario, che è l'I/O. La PM07 non elabora questi segnali. Li inoltra semplicemente verso gli ESC insieme alla potenza della batteria.

Per quanto riguarda l'alimentazione, l'immagine mostra il connettore PWR sulla carrier board dell'UP Core. Si collega con un cavo a 4 pin all'uscita PWR della PM07. Questo è l'ingresso principale a 5 V per il companion computer. I simboli più e meno su entrambi i lati indicano la polarità corretta. Il cavo porta la linea positiva e la massa. Gli altri pin extra non si usano. Servono solo per compatibilità con lo standard del connettore. In pratica, la PM07 fornisce al companion tutta la potenza che gli serve per funzionare.

Il Pixhawk 6C è il nodo centrale dello schema. Usa connettori JST-GH per quasi tutte le interfacce. Le porte TELEM1 e TELEM2 sono a 6 pin. Collegano il flight controller alla carrier board dell'UP Core. Permettono il passaggio dei messaggi MAVLink. E in generale tutto il traffico seriale per la comunicazione tra autopilota e companion computer. Altre porte a 6 pin, come CAM1 e CAM2, si usano per sensori extra del kit. Ad esempio l'optical flow e il sensore di distanza ToF. Nel diagramma appaiono con linee viola. La porta GPS ha un connettore a 10 pin. Trasporta alimentazione, segnali UART per il ricevitore GNSS e linee I2C per il magnetometro.
La parte bassa della figura si concentra sulla carrier board dell'UP Core. Qui ci sono porte USB3.0 e USB2.0. C'è anche un connettore a 5 pin che collega la Structure Core depth camera al companion computer. Questo cavo USB è evidenziato in blu. Trasmette immagini, dati IMU e stime di posa dalla camera. Nella stessa area si vede un connettore piccolo a 3 pin. È per la ventola di raffreddamento della CPU dell'UP Core.

Nel complesso lo schema evidenzia una struttura ordinata e modulare. Si basa su tre tipi principali di connettori ricorrenti. I 6 pin per alimentazione regolata e interfacce seriali. I 10 pin per gruppi di segnali più complessi, come PWM e GPS. E i 3-5 pin per periferiche specifiche, tipo ventole e USB. Questo modo di fare permette di integrare tutto in un'unica architettura coerente: flight controller, PM07, companion computer e sensori. La potenza e i segnali seguono percorsi chiari e facili da controllare durante configurazione e manutenzione.

Capitolo 3

Sistema di tracciamento Vicon

Il sistema di tracciamento Vicon è una piattaforma di motion capture ottico dotata di alta precisione e progettata per ricostruire in tempo reale la posizione e l'orientamento di corpi rigidi all'interno di un volume di misura precedentemente calibrato. Le camere del sistema Vicon emettono un fascio luminoso infrarosso che permette di individuare dei marker, posti sull'oggetto di interesse, in uno spazio bidimensionale. Infatti non avendo le camere singole la misura della profondità, è come se ogni camera, la quale ottiene la posizione dei marker in 2 dimensioni, in 3D individuasse una retta che per ottenere una posizione dovrà essere intersecata con le rette delle altre camere.
Questo lavoro di triangolazione viene eseguito nel nostro caso dal software Vicon Tracker che si ritrova a gestire per un dato istante un numero di immagini pari al numero di camere che ritraggono gli elementi riflettenti. A questo punto stabilire la posizione del marker è un problema numerico, con la complicazione però che occorre determinare quale tra i punti riflessi nell'immagine sia quello in esame. Una volta conosciuta l'effettiva posizione dei marker in 3D è possibile calcolare l'attitude del drone a partire dalla precedente calibrazione dei marker sul drone.
Nel caso specifico della facility utilizzata per questo lavoro il sistema Vicon utilizza delle telecamere Vicon Vero v2.2[12] caratterizzate da:

- compattezza (83 x 80 x 135 mm)
- una risoluzione di 2.2 megapixel
- frame rate fino a 330 fps.
- un illuminatore a infrarossi ad anello: è composto da una serie di LED disposti concentricamente intorno all'obiettivo progettati per emettere radiazione

infrarossa uniforme e stabile. Esso serve a far risaltare sulla scena gli elementi riflettenti che sono resi tali da milioni di microsfere in vetro del diametro di 40-70 micron inglobate in uno strato di resina.

- sensore global shutter: sono in grado di leggere simultaneamente tutti i pixel di un'immagine. Si differenziano dai rolling shutter che leggono una riga per volta ad istanti temporali consecutivi ma diversi, questo non li rende ideali per la rilevazione di oggetti in movimento poiché possono dar vita ad immagini distorte

Le camere del Vicon eseguono un attività di pre-processing delle immagini che consiste nel trovare il centroide dei marker. Questa è un'attività necessaria poiché per ogni immagine si avrà una macchia luminosa più grande del centimetro di diametro degli elementi riflettenti e che può andare dai 4 ai 20 pixel, è dunque necessario trovarne il baricentro per passarlo poi al software di tracking.
Il software di tracking una volta calcolata la poose in 6 DOF la trasmette alla rete e può farlo attraverso una varietà di protocolli di streaming. Quello usato nel nostro caso è il VRPN[1], un protocollo leggero basato su UDP che trasmette le pose del rigid body come sequenza di messaggi contenti posizione e orientamento nel sistema di riferimento del Vicon. A questo punto si è scelto di lanciare due nodi:

- nodo VRPN client: si connette al server VRPN del Vicon Tracker, riceve le pose e le converte automaticamente in messaggi ROS2 standard (geometry_msgs/PoseStamped) rendendola immediatamente disponibile nel grafo ROS2

- nodo vicon_to_px4: legge il messaggio in uscita dal nodo VRPN client contenente la pose che va a pubblicare in \fmu\in\vehicle_visual_odometry

3.1 Architettura del sistema Vicon e della rete locale

Il sistema di motion capture è basato su un'architettura nella quale i componenti Vicon e i dispositivi ROS2 convivono in un'unica LAN privata. L'infrastruttura è costituita da tre elementi principali:

- *Switch PoE Ethernet per le telecamere Vicon*: ogni telecamera Vicon è collegata tramite cavo ethernet ad un PoE switch che permette di fornirgli alimentazione e scambio di dati in un singolo cavo dedicato per ognuna delle 12 telecamere.

[1]Virtual Reality Peripheral Network

Lo switch inoltre collega tutte le telecamere al PC master tramite un'unica porta uplink ethernet

- *Computer Vicon Master*: è il PC che esegue il software Tracker e al tempo stesso funge da server della rete Vicon e nodo centrale dell'intero sistema. Esso è il dispositivo di riferimento della rete LAN e deve essere sempre raggiungibile dai client, in più tramite il software Tracker apre un socket su una porta pubblicando in streaming la posa del drone ed i client VRPN (ROS2) vi si connettono, questo lo rende un server di rete
- *Router (Wi-Fi + DHCP*: esso estende la LAN Vicon via Wi-Fi permettendo a laptop e companion computer di connettersi alla stessa rete del PC master ed assegna indirizzi IP dinamici (DHCP) ai dispositivi che lo richiedono. In realtà il master e il companion computer del drone hanno un indirizzo IP statico, in modo tale che ogni volta che viene avviato il sistema questo è sempre lo stesso

Capitolo 4

Architettura Software

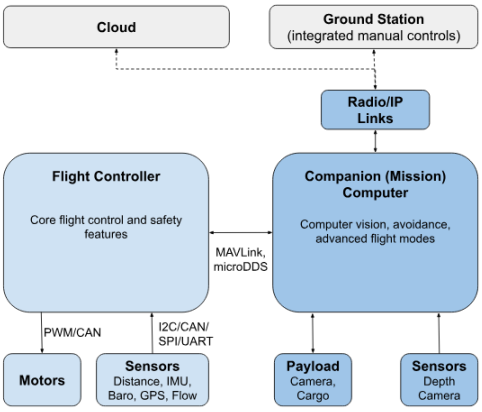

Figura 4.1: Tipica architettura per unmanned vehicle

L'architettura complessiva del sistema di volo non può limitarsi a guardare il velivolo solamente dal lato hardware o software, ma deve tenere in considerazione l'integrazione di questi due aspetti . Sotto questa luce in figura 4.1[13] vengono riportati i principali componenti sia hardware che software che intervengono in una tipica configurazione di veicolo unmanned dotato di due on-vehicle computers: il flight controller e il companion computer. Questi ultimi comunicano attraverso protocolli dedicati (MAVLink o microDDS) che possono viaggiare su UART, UDP, TCP, USB, radio telemetrica, Wi-Fi ed ethernet. A completare l'architettura, oltre a quanto già visto nel capitolo 2 riguardo a connessioni con motori e sensori, ci sono la ground station e l'eventuale infrastruttura cloud che forniscono telemetria, comandi e operazioni di monitoraggio.

4.1 PX4

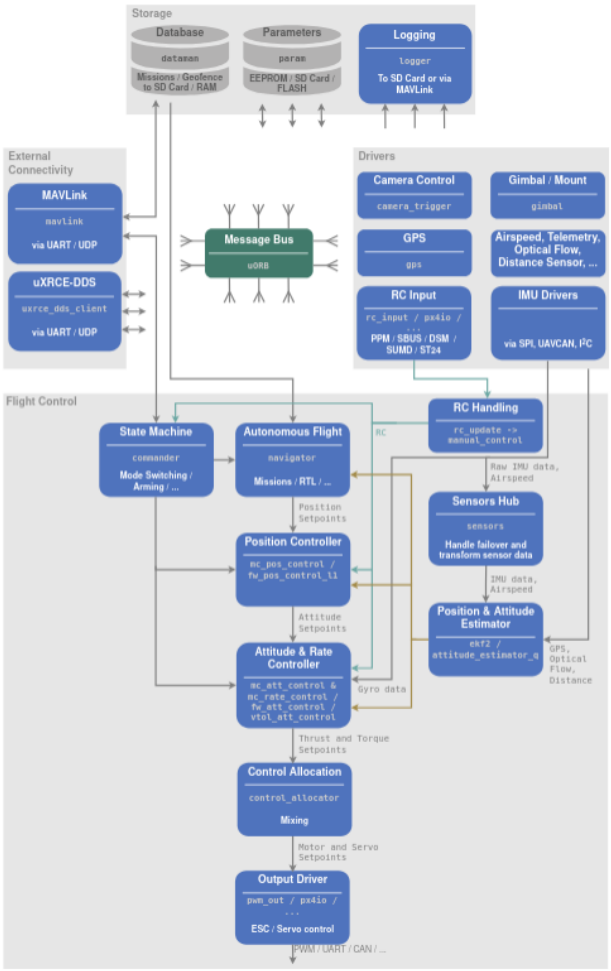

Figura 4.2: High level PX4 software architecture

Il software installato sul flight controller è il PX4 e viene eseguito su sistema operativo NuttX. Tutti gli airframes[1] disponibili su PX4 condividono una stessa base di codice il quale si può definire *reactive*. Questo vuol dire che:

- il PX4 è diviso in molti moduli indipendenti e ognuno di questi può essere modificato, sostituito e riutilizzato su airframes diversi (applicando in alcuni casi le dovute modifiche)

[1]Configurazione predefinita di veicolo: multirotore, ala fissa, VTOL, rover, barca ecc ...

- la comunicazione viene effettuata tramite passaggio di messaggi in maniera asincrona

- il sistema può funzionare con un carico di lavoro variabile

Come si può vedere dalla figura 4.2[14] i moduli comunicano tra di loro tramite il message bus *uORB*: un sistema publish-subscriber che realizza un passaggio asincrono di messaggi. Ciò significa che ogni componente fisico pubblica attraverso il suo driver dedicato dei messaggi uORB ad una propria frequenza (tipicamente l'IMU a circa 1kHz mentre gli altri sensori a poche decine di Hz). Gli stimatori (ad esempio EKF2) e i controllori lavorano ad una frequenza ancora diversa dell'ordine delle centinaia di Hz e prendono l'ultimo messaggio mandato dai driver dei componenti, a prescindere che essi lavorino a frequenze superiori o inferiori. Si noti come questo funzionamento è essenziale per le applicazioni real time poiché fà si che stimatori e controllori non si fermino mai ad aspettare nuove misure.
Lo schema architetturale di figura 4.2 si può dividere in *flight stack* (in figura va sotto il nome di flight control e si trova nella parte inferiore del grafico) e *middleware* (nel grafico tutto ciò che non è flight control).
Il *middleware* consiste principalmente in:

- i driver per i sensori embedded: il modulo software che dialoga con il sensore fisico, traducendo i segnali dell'hardware in messaggi uORB

- tutto ciò che riguarda la comunicazione del drone con il mondo esterno: GCS[2], companion computer ecc...

- uORB publish-subscribe message bus

Rientra inoltre nel gruppo middleware il *simulation layer* che consente di eseguire la missione di un veicolo simulato, supporta sia la configurazione *SITL*[3] che *HITL*[4] con quest'ultima che consiste nell'autopoilota fisico presente nel loop.
Il *flight stack* del PX4 è un insieme di algoritmi di guida, navigazione e controllo per i droni autonomi ed include i controllori per tutti i tipi di airframes supportati. In figura 4.3 si può osservare la pipeline del flight stack nella sua interezza. Si notano i seguenti blocchi:

- estimator: prende uno o più input dai sensori e calcola lo stato del veicolo

[2]Ground Control Station

[3]Software In The Loop

[4]Hardware In The Loop

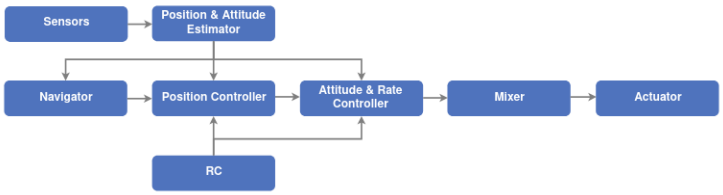

Figura 4.3: Architettura del flight stack

- controller: prende in input la variabile di stato che si vuole controllare e il setpoint che la variabile di stato deve raggiungere (navigator), in uscita dà la correzione da attuare
- mixer: prende in input i valori di forza e coppia sui tre assi dal controllore e li applica nel caso reale regolando i regimi rotativi nei vari motori. Si nota come questa applicazione sia specifica di ogni airframe poiché varia ad esempio la configurazione dei motori rispetto al centro di gravità e la sua inerzia rotatoria

Quindi nello schema di figura 4.3: i sensori rilevano lo stato attuale del veicolo, dove ci saranno dei sensori che daranno misure diverse (seppur in alcuni casi di un ordine di grandezza piccolo) di uno stesso stato. Ebbene queste misure verranno fuse dall'estimatore (nel nostro caso il filtro di Kalman esteso) per ottenere uno stato 6 DOF univoco. Lo stato viene passato così al position controller che prende in input lo stato target del navitor e il comando del pilota che può arrivare da radiocomando (questo passa il comando anche al controllore di attitude & rate. E' interessante notare come tra gli output del position controller ci sia uno stato di attitude, poiché il drone per raggiungere una determinata posizione dovrà assumere un determinato assetto. Infine i comandi di coppia e forza vengono passati al mixer che li traduce in comandi di rate per gli attuatori, trasmessi tramite gli ESC.

4.2 QGroundControl

QGroundControl[15] è una ground control station che connettersi agli autopiloti PX4 o ArduPilot è in grado di:

- fornire l'health status del velivolo e il suo stato e dei suoi sensori in tempo reale
- realizzare un mission planning per il volo autonomo
- fornire supporto per gestire multi-vehicle simulations
- regolare i parametri dell'autopilota e monitorare lo stato dell'autopilota, del drone e dei suoi sottosistemi grazie alla MAVLink Console

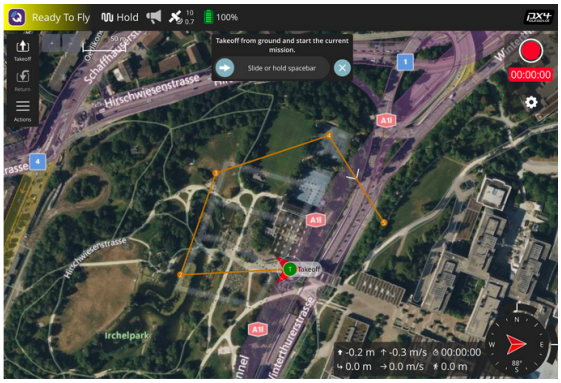

Figura 4.4: Schermata principale di QGroundControl

- visualizzare la mappa 3D dell'ambiente dove si muove il drone
- in caso di camere del veicolo proiettare il video di quest'ultime in tempo reale

4.3 ROS2

ROS è un meta-sistema operativo specifico per applicazioni robotiche, questo vuol dire che fornisce servizi tipici di un sistema operativo quali:

- hardware abstraction: l'applicazione non comunica direttamente con il sensore/attuatore ma parla con un driver ROS. Questa non è una vera astrazione completa come nei kernel OS: dipende comunque dai driver Lunix sottostanti. ROS li organizza, non li sostituisce
- low-level device control: ROS non controlla direttamente i dispositivi hardware del robot ma lo fa tramite nodi, driver e middleware
- implementazioni di funzionalità comuni. Per eseguire operazioni come: protocolli di comunicazione tra nodi, messaggi standardizzati, algoritmi di SLAM, mappe, trasformazioni TF, ROS fornisce librerie pronte
- message-passing tra i processi. ROS consente ad un numero arbitrario di processi (chiamati nodi) di scambiarsi dati in modo strutturato attraverso topic, servizi o azioni. Questi processi possono risiedere su uno stesso computer o più computer, essere implementati in linguaggi differenti ed essere avviati in momenti diversi in maniera indipendente
- package management. ROS gestisce dipendenze , build system (ad es. *colcon*), worksapce e versioni dei pacchetti

ROS[16] fornisce inoltre strumenti e librerie per sviluppare che sono stati ampliamente utilizzati in questa tesi, come: *rviz*, *gazebo*, *rqt*, *rosbag*, *plotjuggler* ecc...
ROS1 è stato inizialmente sviliuppato nel 2007 allo Stanford Artificial Intelligence Laboratory e dal 2013 è gestito dalla Open Source Robotics Foundations. Al giorno d'oggi ROS è uno standard per la programmazione robotica e oltre ai vantaggi fino ad ora elencati si può annoverare quello di avere una community molto attiva con la quale l'utente si può confrontare nell'implementazione dei progetti robotici.
ROS2 è la nuova versione di ROS nata per venire incontro alle rinnovate esigenze del mondo della robotica tra le quali:

- un approccio maggiormente orientato verso le applicazioni industriali, essendo ROS nato in ambiente universitario e di ricerca
- una migliore gestioni delle comunicazioni: la comunicazione in ROS1 non era abbastanza sicura, non era compatibile con protocolli real time ed era *single point of failure*. Infatti in ROS esisteva un nodo centrale al quale tutti i nodi, anche se distribuiti su più computer, devono registrarsi ottenendo da lui le informazioni per contattare gli altri nodi. Di conseguenza se il master smetteva di funzionare o risultava irraggiungibile, l'intero sistema perdeva la capacità di coordinare i nodi.

Quando si ha a che fare con ROS occorre essere familiari con il concetto di distribuzione. Una distribuzione è un rilascio ufficiale e coordinato dell'intero ecosistema ROS che contiene una collezione stabilizzata di pacchetti e librerie, ed un insieme definito di dipendenze e versioni compatibli con la garanzia che tutti questi elementi funzionino insieme. Le distribuzioni sono dunque un qualcosa di necessario in un ambiente eterogeneo come quello della robotica poiché costituiscono un punto di riferimento comune che allinea tutto l'ecosistema.

Distribuzione ROS 2	**Versione Ubuntu compatibile**
ROS 2 Dashing Diademata	Ubuntu 18.04
ROS 2 Foxy Fitzroy	Ubuntu 20.04
ROS 2 Humble Hawksbill	Ubuntu 22.04
ROS 2 Jazzy Jalisco	Ubuntu 24.04

Tabella 4.1: Compatibilità tra distribuzioni ROS 2 e versioni di Ubuntu.

Scegliere la corretta distribuzione di ROS2 in relazione alla versione di Ubuntu risulta cruciale in caso di installazione di ROS2 via binario come riportato nella tabella 4.1. Nel nostro caso specifico si è scelto dunque di installare ROS2 Humble sul pc usato per la simulazione del setup sperimentale che aveva Ubuntu 22.04, mentre si è usato ROS2 Jazzy sul companion computer del drone che aveva Ubuntu 24.04.

4.4 MAVLink e MAVROS

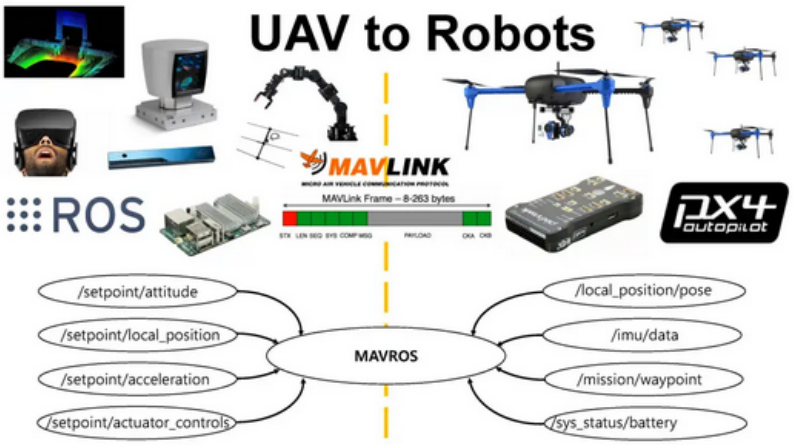

Figura 4.5: MAVROS e MAVLink

MAVLink[17] e MAVROS agiscono allinterno di ROS1/ROS2 ma non sono allo stesso livello. MAVLink è un protocollo di messaggistica molto leggero con IDE di piccole dimensioni adatto a comunicazioni in tempo reale con poco delay come quelle con: ground station, telemetrie radio, MAVROS e può venire anche usato per il caricamento dei parametri dell'autopilota con QGroundControl. MAVLink segue un design pattern ibirido tra un publish-subscribe ed un point-to-point:

- è un publish-subscribe nel senso che c'è un publisher che manda in broadcast e i subscriber ascoltano, ma è una logica molto più semplificata rispetto ad un middleware come DDS (si approfondirà questo concetto nel paragrafo successivo). PX4 pubblica dei messaggi MAVLink ad una certa frequenza fissa e tutti i dispositivi connessi allo stesso link MAVLink ricevono quei pacchetti senza che il PX4 debba sapere chi sono. Viene applicata questa logica principalmente per messaggi riguardanti tutto ciò che viene scambiato con una certa periodicità come: telemetria, heartbeat, attitude/setpoint target e status del sistema

- è un point-to-point per: mission protocol, parameter protocol, file transfer, log streaming e in generale qualsiasi transazione che richiede conferma e per tutte le operazioni che non possono andare perse. Il point-to-point è una transazione affidabile tra due peer[5] con una richiesta che attende una risposta

[5]Dispositivo che può inviare o ricevere dati ed un partecipante alla comunicazione, su un piano paritario, senza gerarchie

e se la richiesta non arriva vengono usati timeout e ritrasmissioni

Facendo riferimento alla figura 4.5, ROS/ROS2 possono comunicare con l'autopilota usando il protocollo MAVLink ma per far si che ciò avvenga è necessario il bridge MAVROS poiché ROS a differenza di PX4 non è compatibile a ricevere dati in protocollo MAVLink. AVROS svolge quindi un ruolo fondamentale di adattamento tra i due ecosistemi: decodifica i pacchetti MAVLink provenienti dall'autopilota e li converte in topic, servizi e parametri ROS, e viceversa. In questo modo i nodi ROS possono interagire con PX4 attraverso un'interfaccia uniforme, senza doversi occupare della gestione del protocollo, del parsing dei messaggi o delle logiche di comunicazione di basso livello. Inoltre MAVROS fornisce un insieme esteso di plugin che implementano i principali sottoprotocolli MAVLink (missioni, parametri, setpoint, stato del veicolo), rendendo possibile il controllo del drone e la lettura della telemetria direttamente all'interno dell'ambiente ROS. Si fa infine notare al lettore come questa configurazione MAVLink+MAVROS non è stata scelta per il setup sperimentale di questo progetto in quanto si è preferito optare per il più moderno uXRCE-DDS per i motivi che verranno elencati nel paragrafo precedente.

4.5 uXRCE-DDS

Micro XRCE-DDS[18] è un middleware di comunicazione ultra-leggero progettato per permettere ai dispositivi con risorse estremamente limitate — come microcontrollori, sensori embedded o autopiloti — di partecipare a un sistema basato su DDS (Data Distribution Service), lo stesso standard usato da ROS 2. Infatti tutta la comunicazione interna di ROS2 si appoggia su veri middleware DDS (Fast DDS, Cyclone DDS, ecc...) ed è per questo che l'agent di uXRCE-DDS non ha bisogno di un bridge come MAVROS per comnuicare con ROS.
Micro XRCE-DDS non è un singolo componente ma è diviso in due parti operative: un client che gira sull'autopilota e un agent sul companion computer. Il client implementa una versione minima del modello DDS avendo l'autopilota delle capacità computazionali molto ristrette che comunque risulta sufficiente per mandare all'agent i messaggi tramite protocolli di trasporto UART, UDP, TCP o customizzati dall'utente (nel nostro caso specifico è stato utilizzato un UART/seriale con baud rate di 921600 sia per l'agent che per il client). Come si può vedere dalla figura 4.6 il uXRCE-DDS client dialoga con i uORB topics tramite una logica publisher-subscriber poiché questi sono il sistema interno di messaggistica di PX4, quindi il client legge questi topic, li converte e li spedisce al mondo esterno che è ROS2. Si nota che, come per il collegamento tra uORB topic e uXRCE-DDS client, tutte le connessioni sono bidirezionali poiché esiste sia il caso in cui il client vuole comunicare con l'agent per mostrargli i suoi topic che il caso opposto in cui si vogliono inviare comandi al PX4.

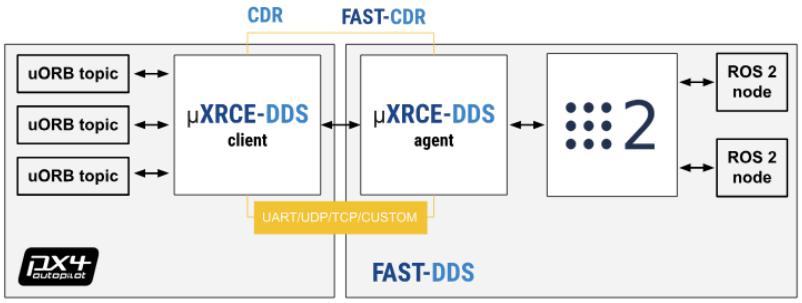

Figura 4.6: Pipeline uXRCE-DDS

Lato companion computer c'è il uXRCE-DDS agent che, dopo che il client ha serializzato i messaggi uORB in un formato ultra-compatto, li deserializza ricostruendo il messaggio nel formato standard DDS usando la libreria FAST-CDR. Una volta che il messaggio è distribuito da FAST-DDS, esso è identico ad un messaggio generato da un nodo ROS2 nativo e perciò può dialogare con essi. FAST-DDS per funzionare ha bisogno però di conoscere la struttura del messaggio che arriva dal client, per farlo ha bisogno del pacchetto *px4_msgs* che dunque dovrà essere allineato alla rispettiva versione dell'autopilota
Le principali differenze tra la comunicazione MAVLink-MAVROS e uXRCE-DDS sono:

- Micro XRCE-DDS è nativo per ROS2 perché parla direttamente DDS, lo stesso middleware usato da ROS2. MAVLink invece deve passare attraverso MAVROS (pensato per ROS1) e poi attraverso un ulteriore bridge per arrivare a ROS2. Questo introduce più livelli, più conversioni e più latenza. Con XRCE-DDS invece PX4 entra direttamente nel dominio DDS di ROS2 senza nessun ponte intermedio.

- MAVLink manda byte grezzi e non conosce il significato del dato, invece con uXRCE-DDS DDS sa: cosa contiene il messaggio, come serializzarlo, come controllare l'integrità e come distribuirlo ai subscriber corretti

- MAVLink usa per lo più scambi best-effort senza vere garanzie, mentre Micro XRCE-DDS eredita i QoS di DDS, permettendo comunicazioni affidabili, con history, deadline e matching automatico tra publisher e subscriber. In pratica non è più un semplice flusso di pacchetti, ma un sistema molto più controllato.

4.6 Ubuntu

Nel contesto di un sistema basato su ROS2 e PX4 Autopilot la scelta di utilizzare Linux Ubuntu[19] non è soltanto una prassi nella comunità robotica ma in questo caso una vera e propria necessità tecnica. ROS2 infatti è progettato e testato principalmente su Ubuntu che rappresenta la piattaforma di riferimento per la distribuzione dei pacchetti binari, il supporto delle librerie middleware (come Fast-DDS) e l'intero ecosistema di tool utili allo sviluppo e al debugging. Anche PX4 è progettato per funzionare nativamente su Linux: la compilazione del firmware, le utility di diagnostica, la gestione delle porte seriali e l'esecuzione dell'Agent Micro XRCE-DDS richiedono un ambiente POSIX completo che Ubuntu è in grado di fornire. A conferma del fatto che Ubuntu sia la giusta scelta per il sistema operativo del companion computer c'è anche il fatto che il PX4 Vision Autonomy Development Kit presenta già installato Ubuntu 18.04 LTS, abbinato con ROS Melodic.

Capitolo 5

Simulazione

Ora che è stato chiarito l'ambiente in cui si andrà ad operare, si può procedere con la spiegazione della prima vera attività di questo progetto: la realizzazione di una simulazione SITL per andare a verificare il setup che poi sarebbe stato utilizzato durante le prove sperimentali con il drone. La simulazione è stata realizzata in primis per provare l'autopilota PX4 Autopilot in modo da settarne i parametri utili al raggiungimento dei task. Per la creazione dell'ambiente simulativo è stato essenziale l'utilizzo del software Gazebo che verrà trattato nel paragrafo successivo.

5.1 Gazebo

Gazebo[20] è un simulatore nato per il campo della robotica, open source, utilizzato per il test e lo sviluppo di piattaforme e algoritmi robotici in un ambiente virtuale realistico. Gazebo è completamente compatibile con ROS costituendo così uno standard nella simulazione robotica, ma può funzionare anche senza di esso. Le sue principali caratteristiche sono:

- distributed simulation: nel caso delle simulazioni più pesanti come quelle multi-veicolo è possibile migliorare le performance eseguendo la simulazione su più computer o più server

- dynamic asset loading: è un ottimizzazione delle risorse che ben si accoppia con le distributed simulations. Consiste nell'andare a scegliere quando utilizzare o meno gli asset simulativi allo scopo di migliorare le performance

- performance regolabili andando a modificare il time step, è possibile quindi andare a realizzare simulazioni real time o più veloci o più lente del real time a piacimento

- cross-platform support: le librerie interne di Gazebo possono essere usate e compilate su Linux e macOS
- cloud integration: consente di visualizzare, caricare e scaricare modelli simulativi sul cloud-hosted server di Gazebo
- l'integrazione con ROS è garantita da un bridge apposito per ogni release di ROS. Nello specifico nella realizzazione della fase di sperimentazione di questo progetto è stato utilizzato un bridge per ROS2 Humble
- si può inserire del rumore nella simulazione che può essere anche personalizzato in base alle necessità
- grafica 3D
- motore del modello fisico molto realistico
- usa plugin che possono essere sfruttati per dare funzionalità aggiuntive alla simulazione come plugin riguardanti il motore fisico, il rendering e GUI libraries
- plugin simulation systems: gazebo fornisce un meccanismo per caricare sistemi customizzati che possono direttamente interagire con la simulazione
- IPC asincrono: una comunicazione intra-process che si basa su topic e servizi simili a quelli di ROS, ma in un formato diverso
- la possibilità di interagire sia da interfaccia grafica che da linea di comando
- l'utilizzo della *Gazebo web application* che permette di trovare nuovi asset simulativi e gestire quelli del proprio team di lavoro, partecipare a competizioni di simulazione ed eseguire simulazioni su piattaforme cloud

5.2 Simulazione SITL con Gazebo, ROS 2 e uXRCE-DDS

Il diagramma di figura 5.1 rappresenta i moduli che intervengono in una simulazione SITL con Gazebo, ROS2, uXRCE-DDS e le relative porte e comunicazioni. Il primo blocco grande sulla sinistra è proprio quello del SITL che è una ambiente attraverso il quale il PX4 non gira su un autopilota fisico, come nel caso reale del Pixhawk, ma viene come processo nativo sul computer che lo esegue, che in questo caso ha Linux Ubuntu 22.04. Una volta avviato il SITL, il PX4 carica un file di parametri che vengono detti di *autostart* che differiscono da quelli di *runtime* poiché i primi servono a far partire il PX4 con l'airframe scelto mentre i secondi impostano delle caratteristiche dell'autopilota quando il suo firmware è in esecuzione. Una volta che il modello è stato caricato l'autopilota abilita i seguenti moduli SITL specifici:

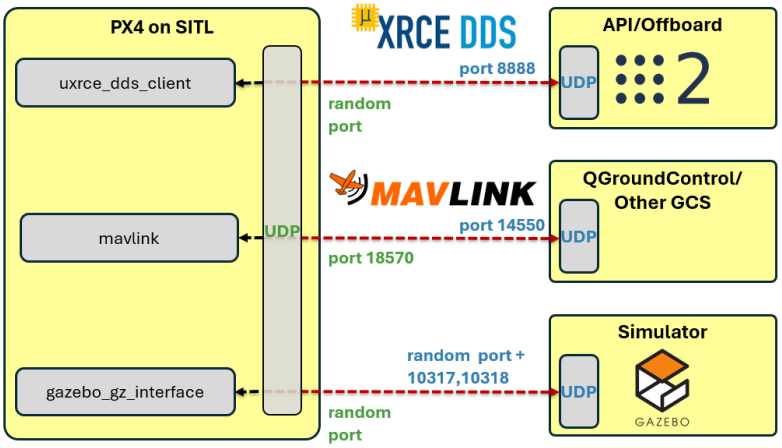

Figura 5.1: Architettura SITL

- *sensors_sim*: sostituisce i driver hardware trasformando i dati di Gazebo in messaggi uORB usati dal firmware

- *gazebo_gz_interface* è il modulo che comunica direttamente con Gazebo: riceve da esso i dati dei sensori simulati e gli invia i comandi dei motori calcolari dall'autopilota. In questa comunicazione e in questo specifico progetto la porta 10317 viene usata per il discovery server[1], la porta 10318 viene usata per la discovery multicast[2]e le porte dinamiche (effimere) sono quelle che portano i dati veri e propri (IMU, pose/velocità, comandi motori ecc...)

- *simulator_interface* riceve i dati dei sensori da *gazebo_gz_interface* e li manda a *sensors_sim*, inoltre prende i comandi da mandare ai motori e li manda a *gazebo_gz_interface*

- *mavlink* serve per lo scambio dell'autopilota con QGroundControl di telemetria, arming, parametri, missioni e dati sullo stato del velivolo. Nella configurazione del progetto apre le porte UDP 18570 (lato PX4) e 14550 (lato QGroundControl)

- *uxrce_dds_client* è il client uXRCE-DDS ma che viene eseguito sul pc e non sul microcontrollore. Nella configurazione del progetto apre le porte 8888 lato agent e delle porte dinamiche lato client

[1]Scoprire quali processi Ignition sono presenti nella simulazione

[2]Diffondere e ricevere informazioni sui servizi e sui topic disponibili nella simulazione

5.3 Filtro EKF2

Il filtro EKF2[21][22] risponde all'esigenzadi una *state estimation* da parte del software di PX4. *State estimation* è il processo che determina gli stati del velivolo a partire dai dati raccolti dai sensori. EKF sta per Extended Kalman Filter che differisce da un semplice filtro di Kalman poiché quest'ultimo funziona solo per modelli lineari mentre quello esteso no. Quella dell'EKF è una tecnica matematica che parte da una predizione dello stato del drone calcolato a partire dal sensore dell'IMU (che quindi risulta essere il sensore più importante) e lo corregge con i dati degli altri sensori. Gli stati a cui giunge EKF2 sono 24 e sono i seguenti:

- 4 stati per l'attitude rappresentata come quaternioni per evitare il gimbal lock
- 3 stati per la velocità
- 3 stati per la posizione
- 9 stati per descrizione di errori e bias nei sensori IMU e magnetometro
- 3 stati per l'estimazione del campo magnetico terrestre
- 2 stati per la velocità del vento, dove si hanno solo due stati e non 3 poiché la componente verticale è poco osservabile

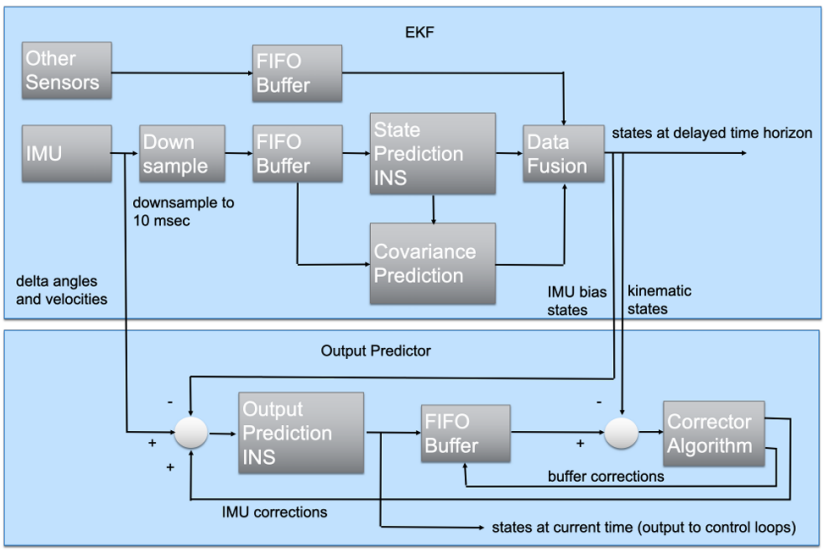

Figura 5.2: Architettura EKF2

In figura 5.2 viene riportata l'architettura alla base del filtro EKF2 di PX4 che è costituito da due moduli distinti ma comunicanti: EKF e Ouput Predictor. Nell'EKF si ha i dati di accelerazione lineare e velocità angolare dell'IMU che sono dati

che arrivano ad una frequenza molto alta e che quindi subiscono un'operazione di *Downsample* poiché il blocco *State Prediction-Data Fusion-Convariance Prediction* lavora ad una frequenza fino a 100 Hz. I dati IMU ridotti entrano nel *FIFO Buffer*[3] che memorizza tra i 100 e i 200 millisecondi di dati, ciò avviene perché l'IMU è un sensore a bassissimo ritardo che però dovrà confrontare lo stato predetto dalla sua misura con misure di sensori con un ritardo di questo ordine di grandezza: fondere misure con questo delay porterebbe a divergenza. Anche i dati degli altri sensori (nel nostro caso il Vicon) vengono memorizzati in un buffer per gestire dati che arrivano a frequenze diverse e con ritardi diversi dai vari sensori. A questo punto i dati IMU entrano nel blocco *State Prediction INS*[4] che attraverso una serie di integrazioni dei dati IMU si calcola uno primo stato del drone, questa operazione però è affetta da errori di integrazione che crescerebbero nel tempo se non venissero corretti da altre misure. Confrontando i dati IMU con lo stato calcolato nel blocco State Prediction si ottiene una *Covariance Prediction* che è una misura dell'errore dello stato calcolato a partire dai dati IMU, questo cresce nel tempo poiché aumentano gli errori di integrazione. A questo punto il blocco *Data Fusion* fonde i dat IMU e quelli degli altri sensori insieme alla misura dell'errore dei dati IMU, ottenendo così uno *stato delayed* del drone che ha un ritardo rispetto allo stato presente di 100-200 millisecondi, troppo grande per essere usato.
Per far fronte a questo problema c'è il blocco Ouput Prerdictor che prende direttamente i dati dell'IMU che hanno un ritardo molto basso e ci vengono sottratti gli errori a dinamica lenta dell'IMU e le IMU corrections. Il risultato viene usato dall'*Output Prediction INS* per ottenere lo stato *non-delayed*, che è l'ouput del filtro EKF2 con una latenza inferiore ai 100 μs, il quale viene immagazzinato in un *FIFO Buffer* per poi venir confrontato ad un comune orizzonte temporale con lo stato in uscita dal blocco EKF; la loro differenza entra nel *Correction Algorithm* che in uscita dà le IMU corrections che vengono anche immagazzinate nel Buffer.

5.4 Fake Vicon

E' stata denominata *fake vicon* l'attività svolta all'interno di questo progetto di tesi per andare a simulare la configurazione del drone che poi sarà ripresa nella configurazione reale[23]. Il caso reale vede il drone che per navigare deve prendere la pose nei 6 gradi di libertà dal sistema Vicon e pubblicarla nel topic */fmu/in/-vehicle_visual_odometry* che verrà usato a sua volta dal filtro EKF2 di PX4 per ricavarne una pose in 6 gradi di libertà fusa con i sensori IMU in /fmu/out/vehicle_odometry. E' possibile simulare questa configurazione poiché su pc è possibile

[3]First In First Out: il primo dato che entra è il primo dato che esce

[4]Inertial Navigation System

eseguire tutti i componenti necessari per far si che ciò avvenga (uXRCE-DDS, PX4 Autopilot, ROS2 e QGroundControl) ad eccezione dei sistema Vicon. Essendo che la fase simulativa è uno step essenziale non solo per prendere dimestichezza con i vari strumenti ma soprattutto per provare la logica del nodo ROS che dovrà leggere il dato e pubblicarlo su vehicle_visual_odometry e per cominciare ad individuare i parametri di runtime (da qui in poi verranno chiamati semplicemente "parametri") necessari per il task di volo reale. Per ovviare all'impossibilità di avere un "Vicon simulato", si usano i dati provenienti dalla simulazione Gazebo come se venissero da una fonte visual e vengono inseriti in vehicle_visual_odometry. Per far questo però occorre prima eseguire un bridge gazebo che converta il messaggio */model/x500_0/odometry* che è un messaggio non ROS di tipo *gz.msgs.Odometry*, in un topic ROS2 omonimo ma convertito in *na_msgs/msg/Odometry*.

```
1  #!/usr/bin/env python3
2  import rclpy
3  from rclpy.node import Node
4  from rclpy.qos import QoSProfile, QoSReliabilityPolicy,
       QoSHistoryPolicy
5
6  from nav_msgs.msg import Path, Odometry
7  from geometry_msgs.msg import PoseStamped
8  from px4_msgs.msg import (
9      VehicleOdometry,
10     VehicleAttitude,
11 )
12
13
14 class MocapNode(Node):
15     def __init__(self):
16         super().__init__('mocap_node')
17
18         qos_be = QoSProfile(
19             reliability=QoSReliabilityPolicy.BEST_EFFORT,
20             history=QoSHistoryPolicy.KEEP_LAST,
21             depth=1
22         )
23
24         # Subscribers
25         self.create_subscription(
26             Odometry,
27             '/model/x500_0/odometry',
28             self.listener_cb,
29             10
30         )
31         self.create_subscription(
32             VehicleAttitude,
33             '/fmu/out/vehicle_attitude',
```

```
34              self.attitude_cb,
35              qos_be
36          )
37
38          # Publishers (pose/path)
39          self.pose_pub = self.create_publisher(PoseStamped, '/x500/
    pose', 10)
40          self.path_pub = self.create_publisher(Path, '/x500/path',
    10)
41
42          # Publisher (visual odom towards PX4)
43          self.odom_pub = self.create_publisher(
44              VehicleOdometry,
45              '/fmu/in/vehicle_visual_odometry',
46              10
47          )
48          # Mocap odom publisher (kept for future use if needed)
49          self.mocap_odom_pub_ = self.create_publisher(
50              VehicleOdometry,
51              '/fmu/in/vehicle_mocap_odometry',
52              10
53          )
54
55          self.path = Path()
56          self.path.header.frame_id = 'map'
57          self.path_cnt = 0
58          self.att_msg = None
59          self.get_logger().info('MocapNode started')
60
61      def listener_cb(self, msg: Odometry):
62          # Log position from Gazebo
63          self.get_logger().info(
64              f"Received message from Gazebo: pos=({msg.pose.pose.
    position.x:.2f}, "
65              f"{msg.pose.pose.position.y:.2f}, {msg.pose.pose.
    position.z:.2f})",
66              once=True
67          )
68
69          # Publish PoseStamped
70          ps = PoseStamped()
71          ps.header = msg.header
72          ps.header.frame_id = 'map'
73          ps.pose = msg.pose.pose
74          self.pose_pub.publish(ps)
75
76          # Path publishing (downsampled)
77          if self.path_cnt == 0:
78              self.path.header.stamp = ps.header.stamp
```

```
79              self.path.poses.append(ps)
80              self.path_pub.publish(self.path)
81          self.path_cnt = (self.path_cnt + 1) % 10
82
83          # Publish VehicleOdometry (visual odom) in NED frame
84          odom = VehicleOdometry()
85          sec = msg.header.stamp.sec
86          nsec = msg.header.stamp.nanosec
87          now_us = sec * 1_000_000 + nsec // 1_000
88          odom.timestamp = now_us
89          odom.timestamp_sample = now_us
90
91          p = msg.pose.pose.position
92
93          # Gazebo odom: ENU -> PX4 NED
94          # ENU = (x = East, y = North, z = Up)
95          # NED = (x = North, y = East, z = Down)
96          odom.pose_frame = VehicleOdometry.POSE_FRAME_NED
97          odom.position = [float(p.y), float(p.x), -float(p.z)]
98
99          if self.att_msg:
100             odom.q = [float(q) for q in self.att_msg.q]
101         else:
102             odom.q = [1.0, 0.0, 0.0, 0.0]
103
104         odom.velocity_frame = VehicleOdometry.
    VELOCITY_FRAME_UNKNOWN
105         odom.velocity = [float('nan')] * 3
106         odom.angular_velocity = [float('nan')] * 3
107         odom.quality = 1
108
109         self.odom_pub.publish(odom)
110
111     def attitude_cb(self, msg: VehicleAttitude):
112         self.att_msg = msg
113
114
115 def main(args=None):
116     rclpy.init(args=args)
117     node = MocapNode()
118     rclpy.spin(node)
119     node.destroy_node()
120     rclpy.shutdown()
121
122
123 if __name__ == '__main__':
124     main()
```

Listing 5.1: Node `mocap_node` per la simulazione SITL

Il codice del nodo ROS2 che fa questo è *mocap_node.py* che è stato riportato nel listing 5.1. Il codice:

- importa il framework ros (rclpy) e la classe base per creare nodi, le QoS per gestire l'affidabilità dei messaggi, i tipi di messaggi ROS standard (Path, Odometry,PoseStamped) ed i messaggi generati da PX4 (VehicleOdometry e VehicleAttitude) usati per la comunicazione con EKF2
- crea il nodo ros *mocap_node*. La QoS BEST_EFFORT viene configurata per i dati non critici: se qualche pacchetto si perde, non serve ritrasmissione (tipico nei dati streaming PX4)
- si mette in ascolto di /model/x500_0/odometry che ora è un messaggio nav_msgs/msg/Odometry e dice che ogni volta che arriva un messaggio sul topic a cui è in ascolto fa partire la callback *self.listenere_cb*. Il nodo si mette in ascolto anche di /fmu/out/vehicle_attitude e ogni volta che su esso arrivano messaggi viene chiamata la callback *self.attitude_cb* che salva l'ultimo assetto ricevuto
- pubblica */x500/pose* e */x500/path* che servono solo per visualizzare in RViz di ROS2 l'attitude e la traiettoria
- pubblica in */fmu/in/vehicle_visual_odometry* ed in */fmu/in/vehicle_mocap_odometry* la pose espressa nel messaggio *VehicleOdometry*. Il primo topic è quello effettivamente utilizzato, il secondo no ma è stato lasciato per eventuali modifiche future che richiedano proprio questo topic
- chiama la *listener:cb* che:
  - converte *Odometry* in *PoseStamp* per uso dei tool ROS (RVIZ)
  - pubblica la path ogni 10 campioni per non saturare RVIZ
  - crea e invia *VehicleOdometry* verso PX4, si nota come occorra passare sia il *timestamp* il quale è il tempo di quando viene pubblicato il messaggio, che il *timestamp_sample* che indica quando la misura è stata ottenuta. Compilare questi due campi è essenziale per il funzionamento del filtro EKF2 come vedremo anche nel capitolo dell'applicazione reale. E' importante sottolineare come nel caso simulativo sia stato assegnato lo stesso valore ad entrambi poiché in SITL non si hanno latenze significative, questo non varrà per il volo in gabbia
  - viene effettauta la conversione da sistema di riferimento ENU a sistema di riferimento NED.
    Questa operazione è necessaria poiché, come si può vedere dalla tabella 5.1 congiuntamente con l'immagine di figura 5.3, tutto ciò che vive nel

Frame	PX4	ROS
Body	FRD (X Forward, Y Right, Z Down)	FLU (X Forward, Y Left, Z Up), usually named `base_link`
World	FRD or NED (X North, Y East, Z Down)	FLU or ENU (X East, Y North, Z Up), with the naming being `odom` or `map`

Tabella 5.1: Uso dei sistemi di riferimento in PX4 e ROS.

mondo ROS (incluso Gazebo e, quando verrà trattato, i dati in uscita dal Vicon) ha come sistema di riferimento world ENU e come body FLU, mentre PX4 usa come world NED e come body FRD. Sono necessari sia il sistema di riferimento world che quello body poiché la posizione è espressa nel sistema di riferimento world, mentre l'attitude è l'orientamento del body rispetto al rispettivo world

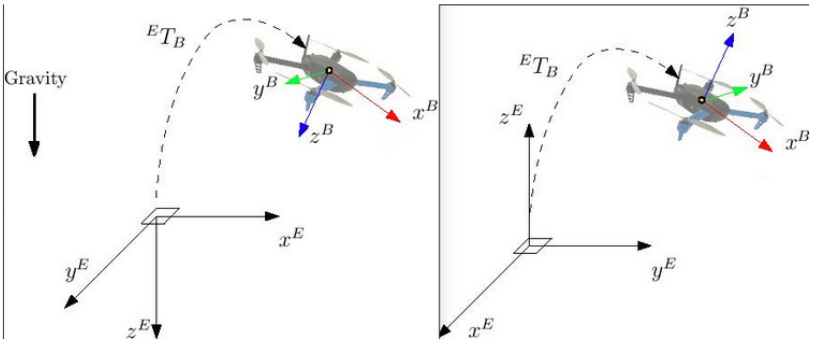

Figura 5.3: Sistemi di riferimento NED/FRD ed ENU/FLU

  - per quel riguarda l'orientamento invece il codice non usa l'attitude di Gazebo poiché in questa prima fase si era interessati a lavorare principalmente sulla posizione, quindi prende il quaternione pubblicato da PX4 e lo copia nel messaggio *VehicleOdometry* che viene inviato

  - non vengono fornite le velocità ed EKF2 le ignora

- definisce il main che avvia ROS, esegue il nodo finché non viene terminato e lo chiude

5.5 Scelta dei parametri PX4

In questo paragrafo si assegnano i parametri[24] al PX4 in modo da raggiungere l'obiettivo di questa fase simulativa, i parametri vengono passati all'autopilota dalla MAVLinkConsole del QGroundControl e sempre da QGroundControl vengono salvati per poi fare il reboot del velivolo.
Sono stati impostati i seguenti parametri:

- **EKF2_BARO_CTRL 0**: non fa fondere al filtro EKF2 l'altezza barometrica, ovvero l'altezza calcolata tramite il barometro
- **EKF2_HGT_REF 3**: indica quale debba essere il riferimento principale per l'altitudine. In questo caso il valore 3 sta a significare che il riferimento principale è la vision, dunque il topic */fmu/in/vehicle_visual_odometry*
- **EKF2_EV_DELAY 10**: indica il ritardo che l'utente si aspetta in millisecondi tra la misura vision e la misura delle IMU. In altre parole è la differenza temporale tra il timestamp del visioin e il "vero" capture time che sarebbe stato registrato secondo il clock dell'IMU che è il clock del filtro EKF2. Questo valore non è quasi mai 0 poiché c'è sempre della latenza, ma l'unico modo per settarlo correttamente è empirico: osservare dai log file quale è l'offset tra gli IMU rate e gli EV rate. Il valore può venire ulteriormente approfondito trovando quello che minimizza la differenza tra il dato vision e la previsione del filtro: innovazione. In questo caso 10 ms è un valore basso, indicativo del fatto che non siamo in un sistema reale
- **EKF2_EV_CTRL** 3: indica quali misure fondere nell'estimatore EKF2. Il valore 3 sta a significare che il filtro fonde solo le misure di posizione orizzontale e verticale. Non fonde la velocità e nemmeno l'attitude, coerentemente al codice di *mocap_node.py*
- **EKF2_EV_NOISE_MD 1**: se a 0 (defalut) la misura del noise (varianza) è presa dal messaggio vision (varianza di vehicle_visual_odometry impostata nel codice *mocap_node.py*) e gli EV noise parameters (EKF2_EVP_NOISE) fungono da limite inferiore, se invece è posto a 1 come in questo caso l'observation noise viene settato direttamente dai parametri
- **EKF2_EVP_NOISE 0.05**: specifica la varianza da parametro per le misure di posizione, questa misura viene presa solo se *EKF2_EV_NOISE_MD* è impostato a 1. Più il valore di EKF2_EVP_NOISE è piccolo, più l'estimatore si fida della misura vision. 0.05 è il valore minimo
- **EKF2_EVP_GATE 4.0**: indica il numero di deviazioni standard sopra le quali il filtro ekf2 rifiuta la misura vision

- **SYS_HAS_GPS 0**: indica che non esiste un modulo GPS a bordo
- **EKF2_GPS_CTRL 0**: disattiva qualsiasi fusione di dati GPS nell'EKF2

5.6 Test simulativo

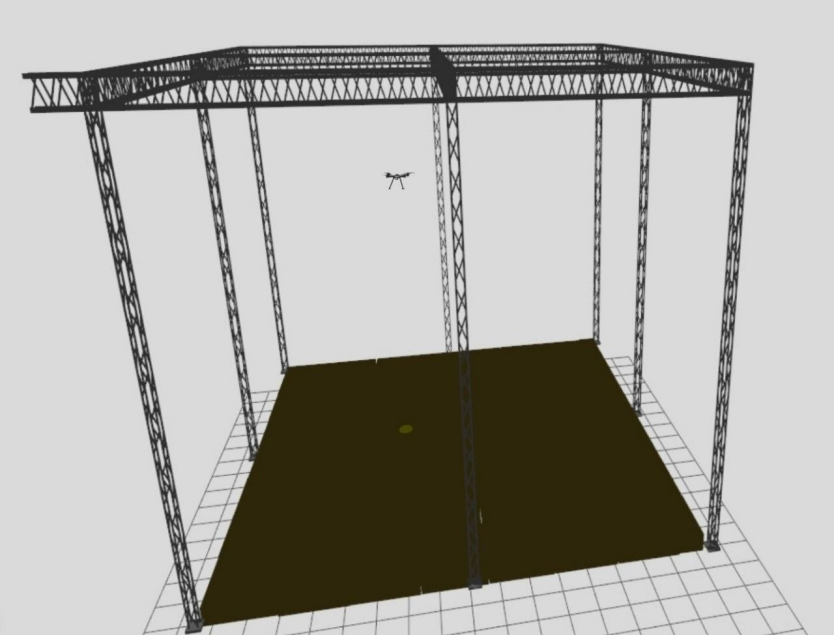

Figura 5.4: Simulazione del volo in gabbia in Gazebo

In questo paragrafo viene riportato il grafico su Plotjuggler di un test simulativo eseguito con il setup descritto in questo capitolo. In figura 5.4 è rappresentata la scena creata in ambiente Gazebo, dove è stato creato un world personalizzato di Gazebo definito tramite un file SDF. A partire da un modello di gabbia già esistente la struttura della gabbia è stata modificata per renderla di dimensione più simile a quella presente nel Politecnico di Torino. La gabbia così modificata è stata poi inclusa all'interno di un mondo dedicato in cui il file SDF si limita a descrivere la scena e posizionare il modello nella posa desiderata. L'ambiente di simulazione è stato inoltre configurato affinché, all'avvio della simulazione tramite il comando *make px4_sitl gz_x500*, il veicolo venga inizializzato direttamente all'interno della gabbia consentendo di eseguire il test in un ambiente delimitato che riproduce quello reale.

Nella figura vengono rappresentate le 3 componenti nello spazio della posizione del drone per il topic */fmu/in/vehicle_visual_odometry* e */fmu/out/vehicle_odometry*.

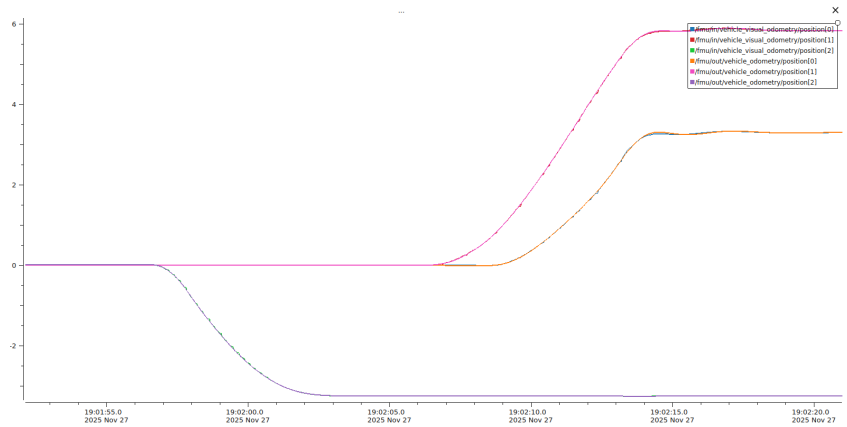

Figura 5.5: Simulazione del volo su Plotjuggler

Come si può notare dalla figura 5.5 ognuna delle coordinate del topic vehicle_odometry segue con buona fedeltà le coordinate di vehicle_visual_odometry, ma al tempo stesso è evidente che esse non coincidano perfettamente in ogni punto. Questo è un buon segno poiché significa che il filtro EKF2 sta effettivamente fondendo il dato visual, e al tempo stesso lo tiene nella giusta considerazione per avere una misura precisa di ciò che sta avvenendo. Tutto ciò conferma che tutta la configurazione utilizzata è funzionante per l'obiettivo che ci si è posti e consente di passare alla fase di volo reale in gabbia.

Capitolo 6

Test Reale

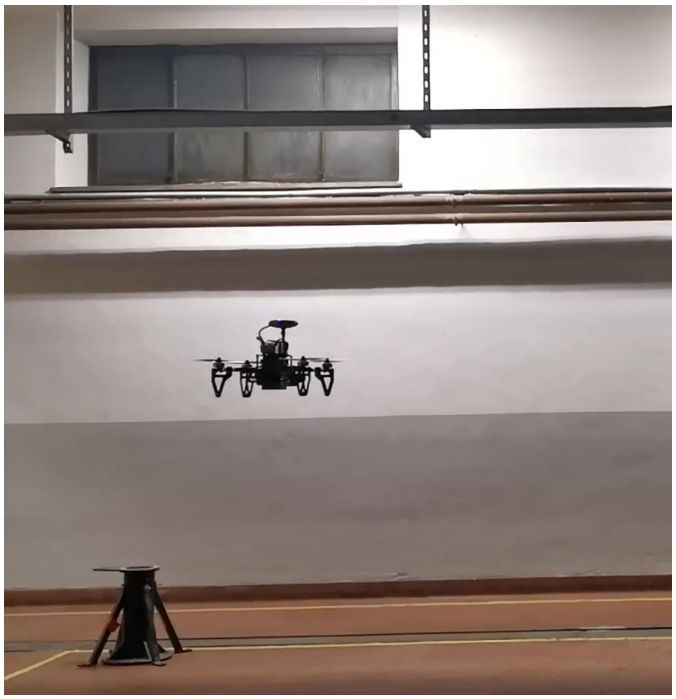

Figura 6.1: Volo reale in gabbia

In questo capitolo si tratta della configurazione del drone per il volo reale. Come già accennato nel capitolo 5 il setup del drone per il volo in gabbia sarà molto simile a quello simulativo. Nello specifico si è andati ad aggiornare il PX4 del microcontrollore alla versione main 1.16 e si sono installati sul companion computer una versione dei *px4_msgs* coerente con quella del PX4. Per consentire la comunicazione diretta tra il flight controller PX4 e il sistema ROS 2 installato sul companion

computer, è stato installato e configurato il middleware Micro XRCE-DDS, insieme alle sue dipendenze principali Fast-DDS (implementazione del protocollo DDS) e Fast-CDR (libreria di serializzazione dei dati); ciò permette di tradurre i topic di *px4_msgs* in flusso DDS standard. Infine si è optato per la release di ROS2 Jazzy.

6.1 Nodo *vicon_to_px4*

Come spiegato nel capitolo 3, il nodo ROS *vicon_to_px4* è quello che prende la pose in uscita dal nodo VRPN client e la pubblica in */fmu/in/vehicle_visual_odometry*.

```
1  #!/usr/bin/env python3
2  import rclpy
3  import numpy as np
4  from rclpy.node import Node
5  from scipy.spatial.transform import Rotation
6  from geometry_msgs.msg import PoseStamped
7  from px4_msgs.msg import VehicleOdometry
8
9  from rclpy.qos import (
10     QoSProfile,
11     QoSReliabilityPolicy,
12     QoSDurabilityPolicy,
13     QoSLivelinessPolicy,
14 )
15
16
17 class ViconToPx4Node(Node):
18     """
19     Convert Vicon pose (world ENU, body FLU) into PX4
       VehicleOdometry
20     (world NED, body FRD) and publish on /fmu/in/
       vehicle_visual_odometry.
21
22     Assumptions:
23       - Vicon Tracker publishes poses in world ENU, body FLU (x
       Fwd, y Left, z Up)
24       - PX4 expects world NED (x North, y East, z Down), body FRD
25         (x Fwd, y Right, z Down).
26     """
27
28     def __init__(self):
29         super().__init__("vicon_to_px4")
30
31         # Optional yaw offset in world NED
32         self.declare_parameter("yaw_offset_deg", 0.0)
```

```
33          self.yaw_offset_deg = float(self.get_parameter("
    yaw_offset_deg").value)
34
35          qos_profile = QoSProfile(
36              reliability=QoSReliabilityPolicy.BEST_EFFORT,
37              durability=QoSDurabilityPolicy.VOLATILE,
38              liveliness=QoSLivelinessPolicy.AUTOMATIC,
39              depth=5,
40          )
41
42          # Publisher towards PX4
43          self._odom_pub = self.create_publisher(
44              VehicleOdometry,
45              "/fmu/in/vehicle_visual_odometry",
46              10,
47          )
48
49          # Subscriber from Vicon
50          self._pose_sub = self.create_subscription(
51              PoseStamped,
52              "/vrpn_mocap/Obj1/pose",
53              self._pose_callback,
54              qos_profile,
55          )
56
57          self.get_logger().info(
58              "vicon_to_px4 node initialized. "
59              "Subscription: /vrpn_mocap/Obj1/pose (remappable). "
60              "Output: /fmu/in/vehicle_visual_odometry"
61          )
62
63      #
    ------------------------------------------------------------------
    ----------
      #
64
65      def _pose_callback(self, msg: PoseStamped):
66          odom_msg = self._convert_pose(msg)
67          self._odom_pub.publish(odom_msg)
68
69      #
    ------------------------------------------------------------------
    ----------
      #
70
71      def _convert_pose(self, pose_msg: PoseStamped) ->
    VehicleOdometry:
72          odom = VehicleOdometry()
73
74          # Current local timestamp (us)
```

```
75          odom.timestamp = int(self.get_clock().now().nanoseconds /
    1000)
76
77          # timestamp_sample: time when the measurement was taken
78          odom.timestamp_sample = int(
79              pose_msg.header.stamp.sec * 1e6 +
80              pose_msg.header.stamp.nanosec / 1e3
81          )
82
83          #
    ------------------------------------------------------------------
84          # POSITION -- world ENU -> world NED
85          #
    ------------------------------------------------------------------
86          x_enu = float(pose_msg.pose.position.x)
87          y_enu = float(pose_msg.pose.position.y)
88          z_enu = float(pose_msg.pose.position.z)
89
90          odom.position = [
91              y_enu,          # North
92              x_enu,          # East
93              -z_enu          # Down
94          ]
95
96          odom.pose_frame = VehicleOdometry.POSE_FRAME_NED
97
98          #
    ------------------------------------------------------------------
99          # ORIENTATION -- world ENU + body FLU -> world NED + body
    FRD
100         #
    ------------------------------------------------------------------
101         enu_quat = np.array([
102             pose_msg.pose.orientation.x,
103             pose_msg.pose.orientation.y,
104             pose_msg.pose.orientation.z,
105             pose_msg.pose.orientation.w
106         ])
107
108         R_enu_flu = Rotation.from_quat(enu_quat).as_matrix()
109
110         # WORLD: ENU -> NED
111         R_ned_enu = np.array([
112             [0, 1,  0],
113             [1, 0,  0],
114             [0, 0, -1]
115         ])
116
117         # BODY: FLU -> FRD
```

```
118         R_flu_frd = np.array([
119             [1,  0,  0],
120             [0, -1,  0],
121             [0,  0, -1]
122         ])
123 
124         # Composition
125         R_ned_frd = R_ned_enu @ R_enu_flu @ R_flu_frd
126 
127         # Optional yaw offset
128         if abs(self.yaw_offset_deg) > 1e-6:
129             yaw = np.deg2rad(self.yaw_offset_deg)
130             R_yaw = Rotation.from_euler("z", yaw).as_matrix()
131             R_ned_frd = R_yaw @ R_ned_frd
132 
133         odom.q = Rotation.from_matrix(R_ned_frd).as_quat().tolist
    ()
134 
135         #
    ------------------------------------------------------------------
136         # VELOCITIES AND VARIANCES
137         #
    ------------------------------------------------------------------
138         odom.velocity_frame = VehicleOdometry.VELOCITY_FRAME_NED
139         odom.velocity = [0.0, 0.0, 0.0]
140         odom.angular_velocity = [0.0, 0.0, 0.0]
141 
142         odom.position_variance = [0.001, 0.001, 0.001]
143         odom.orientation_variance = [0.001, 0.001, 0.001]
144         odom.velocity_variance = [0.01, 0.01, 0.01]
145 
146         odom.quality = 0
147 
148         return odom
149 
150 
151 def main(args=None):
152     rclpy.init(args=args)
153     node = ViconToPx4Node()
154     rclpy.spin(node)
155     node.destroy_node()
156     rclpy.shutdown()
157 
158 
159 if __name__ == "__main__":
160     main()
```

Listing 6.1: Node `vicon_to_px4` for the Vicon–PX4 pipeline

Concettualmente il nodo *vicon_to_px4* fa le stesse cose del codice del nodo usato in simulazione ma con la differenza principale che qui viene presa la pose dal sistema Vicon e non da Gazebo, nello specifico:

- ora il nodo si sottoscrive al topic */vrpn_mocap/Obj1/pose* dal quale prende la pose determinata dal sistema Vicon
- il timestamp e il timestamp_sample non sono più lo stesso valore come in simulazione. Questo perché il Vicon ha la pipeline $camera \rightarrow ViconPC \rightarrow VRPN \rightarrow ROS2$ che ha una sua latenza che non è trascurabile. Dando un timestamp_sample uguale al timestamp (che coincide con l'istante in cui viene pubblicato il topic vehicle_visual_odometry) come in simulazione, si andrebbe a prendere la pose trattandola come se fosse stata presa ad un istante più in avanti nel tempo di quello che in realtà è. Questa differenza può mandare in divergenza l'estimatore EKF2, occorre quindi assegnare al timestamp_ sample il valore del timestamp del topic /vrpn_mocap/Obj1/pose, che sebbene non coincide esattamente con l'istante esatto in cui la misura è stata catturata dal Vicon, è un ritardo più che accettabile
- adesso anche l'attitude viene presa come dato esterno e non viene semplicemente copiata dal PX4. Questo però necessita di un ragionamento sui sistemi di riferimento più accurato poiché la pose in uscita dal Vicon è in un sistema di riferimento diverso da quello che si aspetta il PX4. L'attitude che viene indicata dal Vicon sta a significare l'orientamento del sistema di riferimento body FLU rispetto al world ENU (si veda il paragrafo 5.4) ed essendo che il PX4 si aspetta l'attitude intesa come l'orientamento di un body FRD rispetto a un world NED occorre effettuare la seguente rotazione:

  $$R_{NED \rightarrow FRD} = R_{NED \rightarrow ENU} \times R_{ENU \rightarrow FLU} \times R_{FLU \rightarrow FRD}$$

  . Quindi essendo che l'attitude in uscita dal Vicon è il quaternione $q_{ENU \rightarrow FLU}$, occorre applicarvi la rotazione $R_{NED \rightarrow FRD}$ calcolata precedentemente per dare in pasto al PX4 un quaterione nel sistema di rifeirmento da lui atteso
- la posizione viene passata nello stesso modo di come è stato fatto nel paragrafo 5.4 trasformando la posizione da ENU a NED

6.2 Parametri di volo

In questa trattazione con *parametri di volo* si intendono i parametri assegnati all'autopilota quando è stato effettuato il test di volo vision in gabbia. Di seguito sono elencati i parametri più significativi e sono spiegati quelli che non sono stati già trattati nel paragrafo 5.5

- **EKF2_BARO_CTRL 0**
- **EKF2_HGT_REF 3**
- **EKF2_EV_DELAY 80**: differisce dal valore inserito in simulazione poiché qui il ritardo tra la misura e la pubblicazione del topic visual è maggiore
- **EKF2_EV_CTRL 11**: a differenza della simulazione il valore 11 sta a significare che nel filtro viene fusa non solo la posizione verticale ed orizzontale, ma anche lo yaw
- **EKF2_EV_NOISE_MD 1**
- **EKF2_EVP_NOISE 0.01**
- **EKF2_EVP_GATE 6.0**: valore più alto rispetto al caso simulativo, per far rifiutare la misura del Vicon il meno possibile
- **EKF2_GPS_CTRL 0**
- **SYS_HAS_GPS 0**
- **EKF2_OF_CTRL 0**: non fonDe la misura dell'optical flow
- **EKF2_RNG_CTRL 0**: non fonde la misura del range finder

6.3 Variazione dei parametri

In questo paragrafo si mostrano gli effetti in una prova in gabbia a motori spenti di alcuni di alcuni dei parametri scelti: EKF2_EVP_NOISE, EKF2_BARO_CTRL ed EKF2_EV_DELAY. In ognuna delle sezioni di questo paragrafo viene fatto variare, rispetto ai *parametri di volo*, solamente il parametro che dà il nome alla sezione stessa.

6.3.1 EKF2_EVP_NOISE 0.2

E' stato aumentato il velore del parametro EKF2_EVP_NOISE da 0.01 a 0.2, questo sta a significare che il filtro ora si fida meno della misura del Vicon prendendo più in considerazione il dato dell'IMU che è un dato ad alta frequenza ma affetto da errore. In figura 6.2 si nota dunque come il topic *vehicle_odometry* segua con scarsa precisione la misura visual di *vehicle_visual_odometry*, che con buona approssimazione può essere ritenuta la posizione reale del drone. Essendo che la posizione in uscita dall'EKF2 differisce troppo da quella reale si può concludere che questa configurazione non soddisfa i requisiti minimi per andare ad effettuare

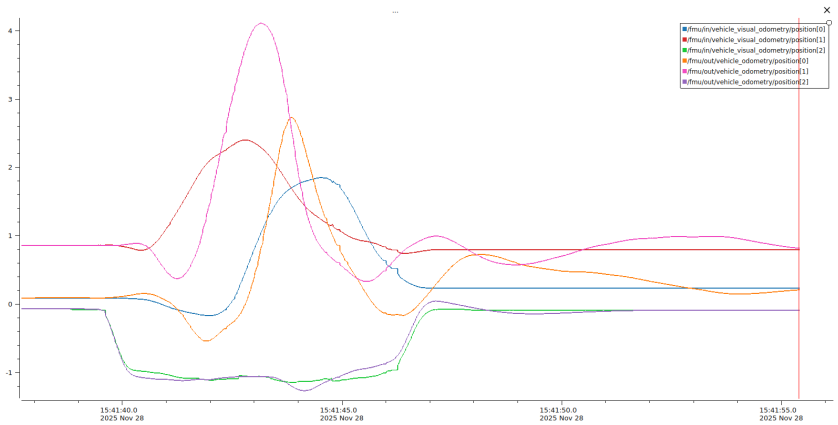

Figura 6.2: EKF2_EVP_NOISE 0.2

un volo vero e proprio.
Si può notare però che nonostante in alcuni punti la differenza tra *vehicle_visual_odometry* e *vehicle_odometry* sia molto alta (si veda ad esempio il picco sulla coordinata y), andando ad indicare una differenza tra il valore atteso di posizione e quello che gli arriva dalla misura, il filtro non rifiuta mai la misura visual, cosa che accade quando si trovano bruschi tratti discontinui. La ragione di ciò la si può trovare nella matematica che sta dietro al filtro di Kalman esteso. La misura viene accettata se:

$$\frac{innovation^2}{S} < gate^2 \tag{6.1}$$

Con l'*innovation variance* $S = HPH^T + R$ e $gate = EKF2\_EVP\_GATE$, dove H è la matrice di osservazione. P è la convarianza dello stato ed $R = EKF2\_EVP\_NOISE$. Appare dunque evidente che aumentando il valore EKF2_EVP_NOISE a parità di innovazione, le misure verranno scartate più difficilmente.

6.3.2 EKF2_BARO_CTRL 1

Settare il parametro EKF2_BARO_CTRL ad 1 significa far entrare nella fusione di EKF2 anche il dato dell'altezza barometrica. Il grafico presente in figura 6.3 mostra un comportamento fiscontinuo del topic *vehicle_odometry* che sta a significare che la misura visual viene rifiutata poiché l'equazione 6.1 non è più soddisfatta.
La quota barometrica è una misura sulle z e per questo può sembrare strano che vadano in instabilità la x e la y. Questo non deve però sorprendere poiché andando a fondere anche la quota barometrica cambia la covarianza interna P associata alle misure delle varie fonti di dati, quindi anche alle x ed alla y della misura visual, andando a variare così quanto il filtro si fida dei vari sensori.

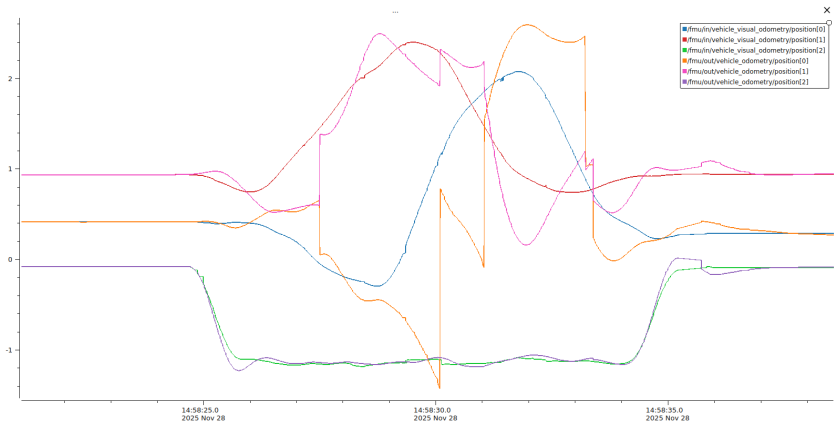

Figura 6.3: EKF2_BARO_CTRL 1

6.3.3 EKF2_EV_DELAY

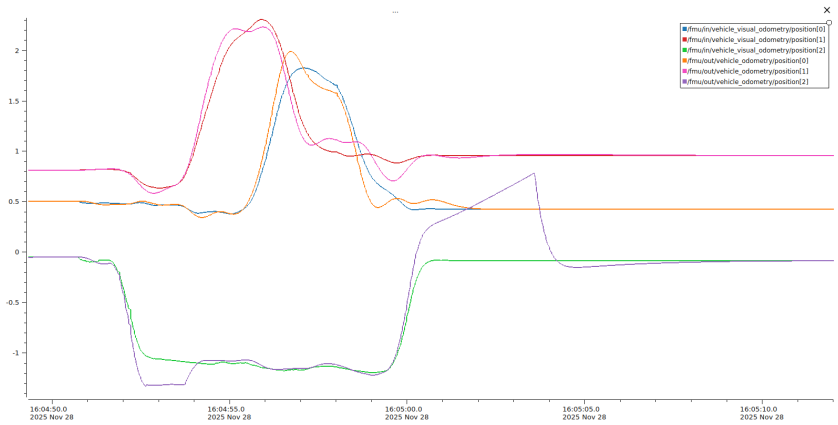

Figura 6.4: EKF2_EV_DELAY 150 ms

In questo paragrafo viene studiato il parametro EKF2_EV_DELAY variandone il valore in millisecondi. Nelle figure da 6.4 a 6.9 sono rappresentate le prove con valori diEKF2_EV_DELAY decrescente di 150, 80, 40, 20, 10 e 5 ms. Osservando i grafici si può osservare che il valore di 150 ms è un valore troppo grande di delay, così come 5 e 10 ms sono troppo piccoli. I risultati migliori si ottengono per un delay della misura vision supposto tra i 20 e i 40 ms, mentre il ritardo di 80 ms è accettabile ma meno preciso. Il punto però è che la rete ha un ritardo variabile che può crescere in caso di traffico maggiore, è stato perciò ritenuto più sicuro assegnare un delay di 80 piuttosto che di 20 o 40 per essere conservativi.

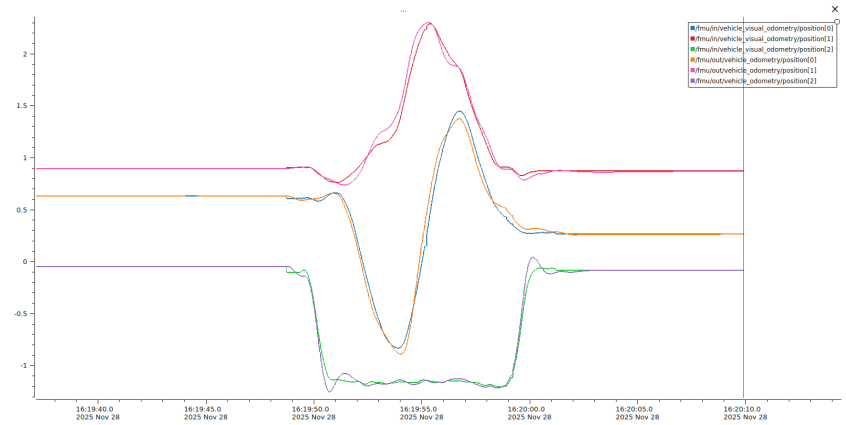

Figura 6.5: EKF2_EV_DELAY 80 ms

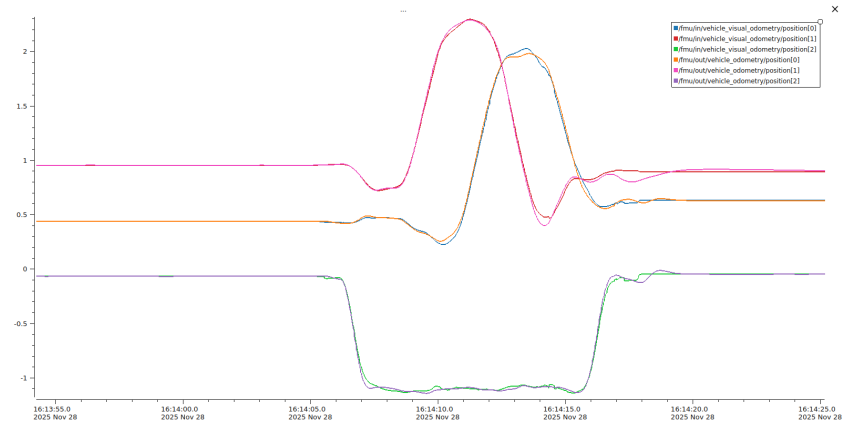

Figura 6.6: EKF_EV_DELAY 40 ms

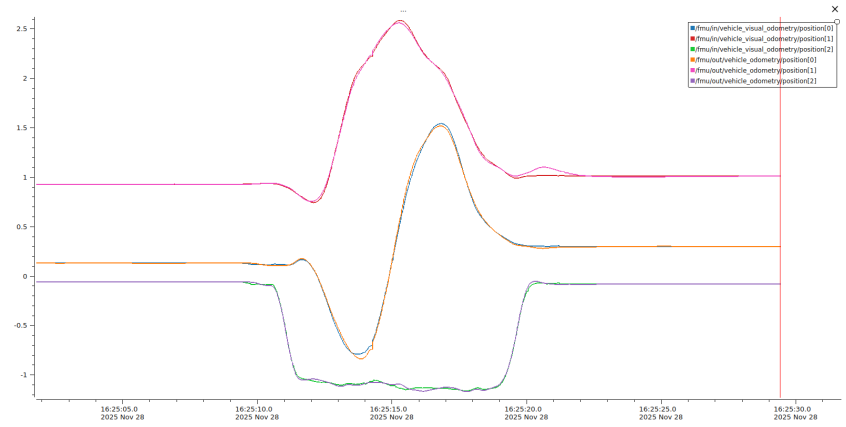

Figura 6.7: EKF2_EV_DELAY 20 ms

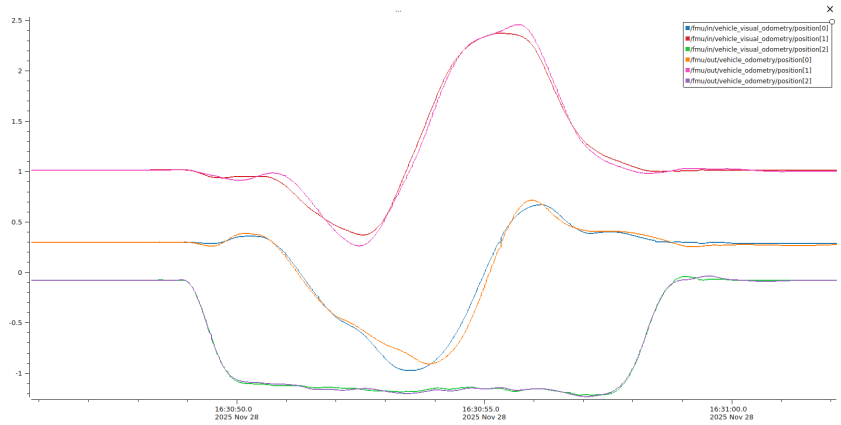

Figura 6.8: EKF2_EV_DELAY 10 ms

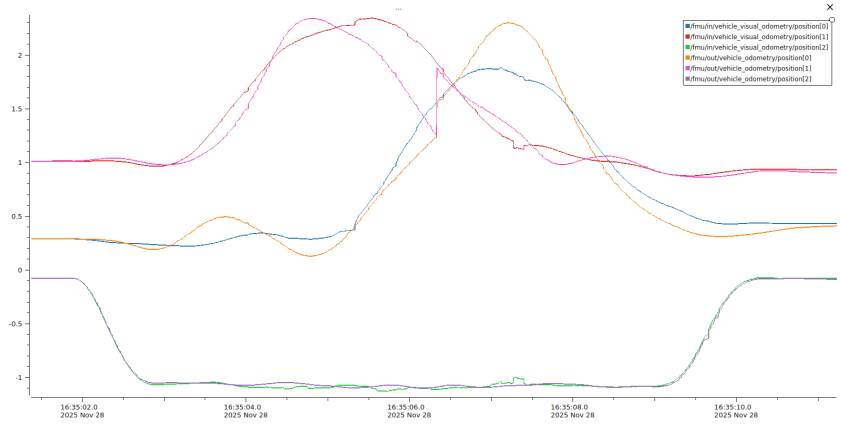

Figura 6.9: EKF2_EV_DELAY 5 ms

6.4 Prova di volo

In questa sezione vengono riportati i risultati delle prove di volo in gabbia eseguite nelle flight mode *Altitude* e *Position*.
La modalità *Position* è una mdoalità in cui PX4 controlla automaticamente X,Y e Z ed è dunque una modalità completamente assistita per il pilota che comanda una posizione e non un assetto. Per questo vengono attivati i controller in posizione, velocità ed attitude in modo tale che il comando di traiettoria del pilota in posizione e velocità venga usato come setpoint da raggiungere e che l'attitude sia controllata di conseguenza, unicamente dal PX4.
La modalità *Altitude* è invece una modalità semi-assistita in cui il PX4 mantiene automaticamente l'altitudine Z ma non controlla la posizione X ed Y. Per questo in questa modalità c'è un altitude un attitude controller poiché il controllore di quota accetta un setpoint di quota e regola la thurst in modo automatico, allo stesso modo l'attitude controller riceve dal pilota un setpoint di attitude (a differenza della modalità position) e regola i motori per mantenerlo.
Le prove di volo sono state eseguite con i *parametri di volo* del capitolo 6.2 e nei grafici 6.10 e 6.12 vengono riportati i topic */fmu/in/vehicle_visual_odometry* ed */fmu/out/vehicle_odometry*, dove quest'ultimo è denominato *estimator_odometry* poiché questi topic sono stati ricavati dai logfiles presi da QGroundControl che quindi ha rinominato i topic. Si nota come in entrambe le prove volo si ha un'ottima aderenza del topic in uscita dal filtro con il topic visual confermando che l'estimatore lavora in maniera corretta. I test sono stati eseguiti prima in modalità *Altitude* e poi in modalità *Position*, questo proprio perché *Position* usa dei controller oltre che in altitude e in attitude, in posizione e velocità rendendo il sistema più sensibile ad eventuali errori della misura visual (nel nostro caso le approssimazioni sono state a livello di delay e di timestamp). Infatti in altitude mode nonostante le misure visual continuino ad essere fuse dall'EKF2, non vengono usate le posizioni in X ed Y e le velocità in generale per i controllori, rendendo così questa modalità più "robusta" rispetto ad eventuali imprecisioni della misura visual. E' proprio per questo motivo che il pilota nella prova *Position* ha riscontrato una leggera oscillazione del drone in fase di decelerazione e nei cambi di direzione.

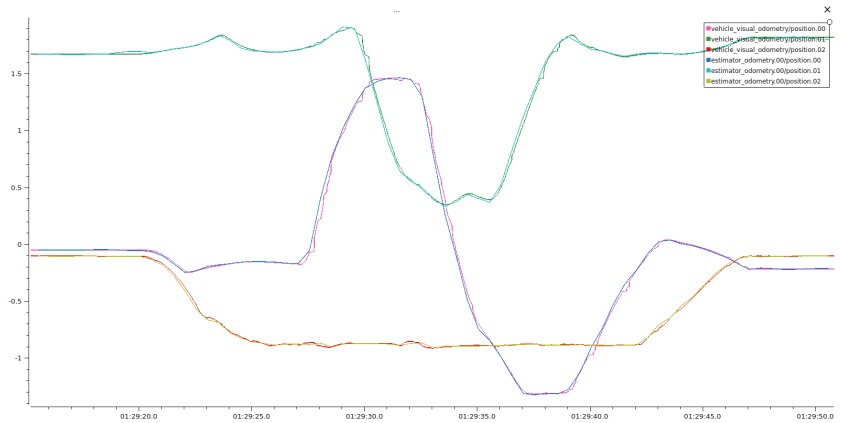

Figura 6.10: Prova di volo in Altitude Mode

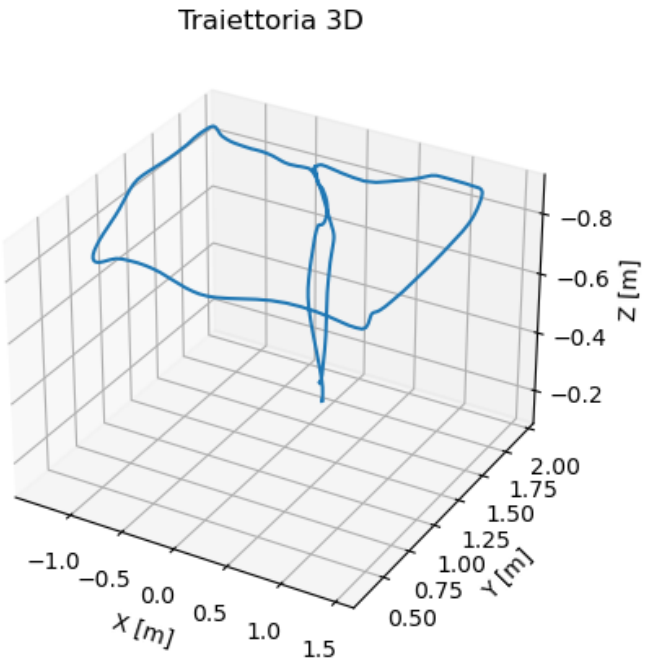

Figura 6.11: Traiettoria della prova di volo eseguita in Altitude Mode

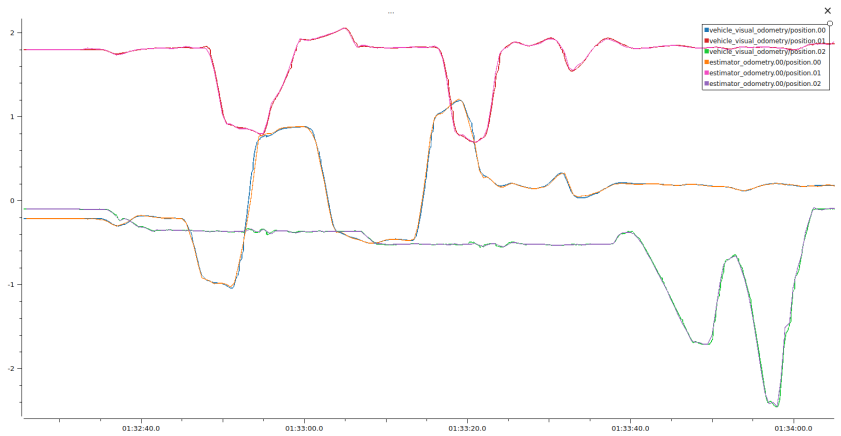

Figura 6.12: Prova di volo in Position Mode

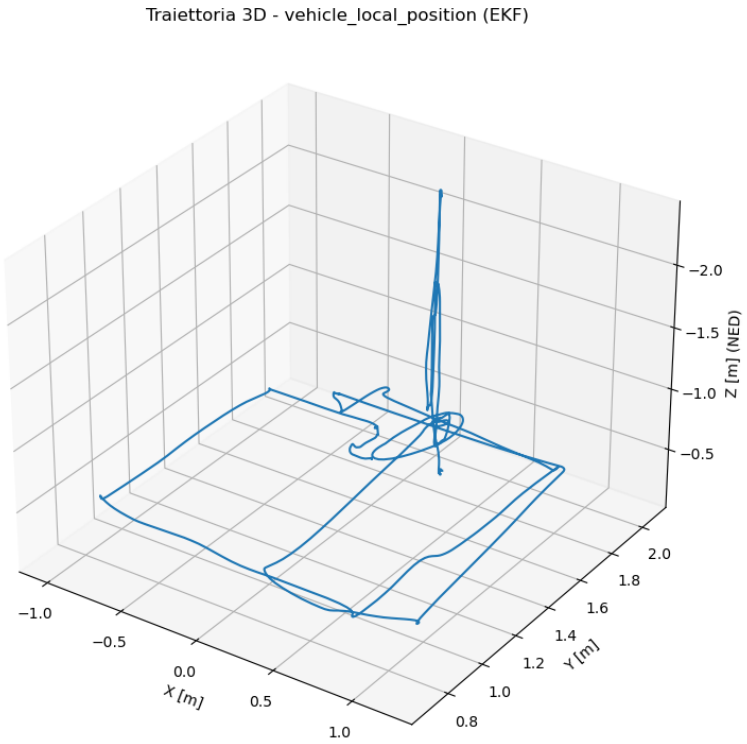

Figura 6.13: Traiettoria della prova di volo eseguita in Position Mode

Capitolo 7

Conclusioni e Lavori Futuri

Il progetto di tesi si può ritenere concluso con successo in quanto sia le prove simulative che quelle in gabbia sono riuscite a far acquisire all'autopilota il dato visual e a fonderlo correttamente nell'estimatore. Le prove in gabbia hanno mostrato delle qualità di volo assolutamente in linea con quelle previste dagli scopi del progetto ma è chiaro che ci sia modo di migliorarle. La principale fonte di degrado del dato di posizione fuso dall'estimatore è relativo alla gestione dei timestamp e del parametro *EKF2_EV_DELAY* poiché per l'estimatore è fondamentale sapere con la massima precisione a quale istante temporale risale una determinata misura del sistema Vicon.
Nel progetto infatti non sono stati misurati con precisione il tempo che impiega il dato misurato dalle camere a venire elaborato dal software in una posizione a 6 gradi di libertà, approssimando l'istante della misura delle camere a quello della pubblicazione del topic del nodo del client VRPN facendo credere all'estimatore che il dato in questione sia relativo ad un istante successivo rispetto a quello a cui è in realtà riferito. Per lavori futuri può essere studiato con maggiore precisione il parametro precedentemente riportato sul delay che essendo l'intervallo di tempo inserito dall'utente tra l'istante di tempo nel clock IMU e l'istante di pubblicazione del topic visual, è l'altro elemento che stimato correttamente, anche in relazione alla latenza della rete LAN, farebbe migiliorare la correttezza della stima del filtro. Un'altra correzione per poter migliorare i risultati è quella di far fondere al filtro i dati di velocità della misura vision, fornendogli così un dato in più sul quale effettuare la sua stima. Occorre infine annoverare anche la possibilità di configurare una flotta di droni operanti nella facility di volo indoor del Politecnico di Torino, sotto la rete LAN già sfruttata per questo progetto.

Bibliografia

[1] PX4 Autopilot. *PX4 Vision Kit — Complete Vehicle Setup*. Accessed: 2025-11-09. 2024. URL: https://docs.px4.io/main/en/complete_vehicles_mc/px4_vision_kit (cit. a p. 3).

[2] PX4 Autopilot. *Holibro Pixhawk 6C*. Accessed: 2025-12-03. 2024. URL: https://docs.px4.io/main/en/flight_controller/pixhawk6c#holybro-pixhawk-6c (cit. a p. 6).

[3] PX4 Autopilot. *Power Modules & Power Distribution Boards*. Accessed: 2025-12-03. 2024. URL: https://docs.px4.io/main/zh/power_module/ (cit. a p. 8).

[4] UP. *UP Core Specifications*. Accessed: 2025-12-04. 2023. URL: https://up-board.org/wp-content/uploads/datasheets/UP-core-DatasheetV0.3.pdf (cit. a p. 10).

[5] PX4 Autopilot. *Holybro M8N & M9N GPS*. Accessed: 2025-12-04. 2025. URL: https://docs.px4.io/main/en/gps_compass/gps_holybro_m8n_m9n (cit. a p. 11).

[6] PX4 Autopilot. *Mounting a Compass (or GNSS/Compass)*. Accessed: 2025-12-04. 2024. URL: https://docs.px4.io/main/en/assembly/mount_gps_compass.html (cit. a p. 11).

[7] Holybro. *Holybro Strucutre Core*. Accessed: 2025-12-04. 2025. URL: https://holybro.com/products/structure-core?srsltid=AfmBOooAeElYjOuqbE12EebQ6o2eZstLJxu9af0MJ-uIqHa-Eh2VnyAC (cit. a p. 12).

[8] PX4 Guide. *PMW3901-Based Flow Sensors*. Accessed: 2025-12-04. 2025. URL: https://docs.px4.io/main/en/sensor/pmw3901 (cit. a p. 14).

[9] Circuit Digest. *Interfacing PMW3901 Optical Flow Sensor With ESP32*. Accessed: 2025-12-04. 2023. URL: https://circuitdigest.com/microcontroller-projects/interfacing-pmw3901-optical-flow-sensor-with-esp32 (cit. a p. 14).

[10] RX Electronics. *PSK-CM8JL65-CC5 ToF Infrared Distance Measuring Sensor*. Accessed: 2025-12-04. 2024. URL: https://www.rxelectronics.nz/datasheet/47/101990440.pdf (cit. a p. 14).

[11] Holybro. *Wiring Diagram*. Accessed: 2025-12-04. 2024. URL: https://docs.holybro.com/drone-development-kit/px4-vision-dev-kit-v1.5/wiring-diagram (cit. a p. 14).

[12] Vicon. *Vicon Vero Technical Information*. Accessed: 2025-12-04. 2025. URL: https://www.vicon.com/hardware/cameras/vero/#technical-information (cit. a p. 17).

[13] PX4 Guide. *Companion Computers*. Accessed: 2025-12-04. 2025. URL: https://docs.px4.io/main/en/companion_computer/ (cit. a p. 20).

[14] PX4 Guide. *PX4 Architectural Overview*. Accessed: 2025-12-04. 2025. URL: https://docs.px4.io/main/en/concept/architecture (cit. a p. 22).

[15] QGroundControl. *QGroundControl Guide (Daily Builds)*. Accessed: 2025-12-04. 2025. URL: https://docs.qgroundcontrol.com/master/en/qgc-user-guide/index.html (cit. a p. 23).

[16] ROS. *ROS - Robot Operating System*. Accessed: 2025-12-04. 2025. URL: https://www.ros.org/ (cit. a p. 25).

[17] MAVLink. *MAVLink Developer Guide*. Accessed: 2025-12-04. 2025. URL: https://mavlink.io/en/ (cit. a p. 26).

[18] PX4 Guide. *uXRCE-DDS (PX4-ROS 2/DDS Bridge)*. Accessed: 2025-12-04. 2025. URL: https://docs.px4.io/main/en/middleware/uxrce_dds (cit. a p. 27).

[19] Ubuntu. *Ubuntu Italia Homepage*. Accessed: 2025-12-04. 2025. URL: https://www.ubuntu-it.org/ (cit. a p. 29).

[20] Gazebo. *Gazebo Homepage*. Accessed: 2025-12-04. 2025. URL: https://gazebosim.org/home (cit. a p. 30).

[21] Dr. Paul Riseborough. *'PX4 State Estimation' PX4 Developer Summit Zurich 2020*. Accessed: 2025-12-04. 2025. URL: https://px4.io/wp-content/uploads/2020/07/Paul-Riseborough-PX4-state-estimation-update.pdf (cit. a p. 33).

[22] PX4 Autopilot-youtube. *PX4 state estimation update - Dr. Paul Riseborough - PX4 Developer Summit 2019*. Accessed: 2025-12-04. 2025. URL: https://www.youtube.com/watch?v=HkYRJJoyBwQ (cit. a p. 33).

[23] PX4 Guide. *Using Vision or Motion Capture Systems for Position Estimation*. Accessed: 2025-12-04. 2025. URL: https://docs.px4.io/main/en/ros/external_position_estimation (cit. a p. 34).

[24] PX4 Guide. *Parameter Reference*. Accessed: 2025-12-04. 2025. URL: `https://docs.px4.io/main/en/advanced_config/parameter_reference` (cit. a p. 40).