UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Creazione di uno scenario pilota per il volo di droni in pattuglia basato su simulatori industriali open source

Relatori:

Prof: Mario G.C.A. Cimino

Prof: Cosimo Antonio Prete

Prof: Pierfrancesco Foglia

Candidato:

Giacomo Bertelli

ANNO ACCADEMICO 2022/2023

Indice

1 Introduzione **1**

2 Tecnologie utilizzate **1**
2.1 PX4 Autopilot . . . 1
2.2 Flight controller . . . 1
2.3 Companion computer . . . 1
2.4 MAVlink e MAVSDK . . . 2
2.5 Ground control station . . . 2
2.6 Simulatore, SITL, HITL . . . 2

3 Studi preliminari **3**
3.1 Ambiente di sviluppo . . . 3
3.1.1 Simulatori . . . 4
3.1.2 QGroundControl . . . 5
3.1.3 Python . . . 5
3.1.4 Altri strumenti utili . . . 5
3.2 Libreria MAVSDK-Python . . . 5
3.2.1 Script di esempio . . . 6

4 Scelte progettuali iniziali **8**
4.1 Architettura centralizzata o distribuita . . . 8
4.2 Comunicazione tra droni . . . 8
4.3 Individuazione degli obiettivi . . . 9
4.3.1 Dipendenza dall'hardware . . . 9
4.3.2 Dipendenza dal simulatore . . . 9
4.3.3 Difficoltà implementative . . . 9

5 Sviluppo del software **10**
5.1 Classe SystemWrapper . . . 10
5.2 Classe DronePosition . . . 11
5.2.1 Costruttori . . . 12
5.2.2 Conversione in altri formati . . . 12
5.2.3 Spostamenti relativi . . . 13
5.2.4 Calcolo delle distanze . . . 14
5.2.5 Conversione in stringa . . . 14
5.3 Classe Swarm . . . 14
5.3.1 Costruttore . . . 14
5.3.2 Operazioni di base sui droni . . . 15
5.3.3 Acquisizione delle posizioni . . . 16
5.3.4 Controllo dei droni . . . 16
5.3.5 Riconoscimento degli obiettivi . . . 17

6 Esempio di utilizzo **18**
6.0.1 Codice dimostrativo 18
6.0.2 Macro struttura del codice 21

1 Introduzione

L'obiettivo di questa tesi è quello di semplificare l'implementazione di algoritmi per il volo di droni in pattuglia utilizzando simulatori industriali open source. Più nello specifico, si è cercato di sviluppare una libreria che riesca a dare all'utente un'interfaccia per il controllo di droni reali indipendente dai protocolli utilizzati e dai dettagli fisici dei droni stessi. Il codice qui sviluppato non è da intendersi come punto di arrivo, ma anzi come punto di partenza per lavori futuri; non sono stati infatti presi in considerazione algoritmi specifici ma si è sempre cercato di mantenere il codice il più generale possibile.

Le tecnologie prese in esame rappresentano lo standard industriale in materia sia di simulazione che di droni, così da permettere, in futuro, di arrivare a un prodotto di livello, con possibili applicazioni reali.

2 Tecnologie utilizzate

In questo capitolo verranno illustrate le tecnologie utilizzate durante lo sviluppo del progetto a un livello di dettaglio sufficiente a rendere chiaro il seguito. Dove non specificato diversamente il linguaggio di programmazione utilizzato sarà Python 3.10.6.

2.1 PX4 Autopilot

PX4[1] è un software di controllo open source che mette a disposizione uno standard per mantenere sia il software che l'hardware dei droni. Nonostante PX4 sia compatibile con varie famiglie di droni (ala fissa, decollo verticale, ecc.), nel seguito ci occuperemo solamente di multicotteri e, nello specifico, di quadricotteri. Di seguito verranno elencati brevemente i vari componenti che si interfacciano con PX4.

2.2 Flight controller

Il flight controller[2] è un microcontrollore su cui viene eseguito PX4 e che si occupa del controllo dei sensori, dell'attuazione dei motori, e della comunicazione wireless con i dispositivi di terra (Ground Control Station e Controller).

2.3 Companion computer

Il flight controller non deve essere confuso con il companion computer, un dispositivo anch'esso installato sul drone che ha capacità di calcolo molto più elevata. Il companion computer si occupa di task più onerose come per esempio la computer vision o l'obstacle avoidance. Per i nostri scopi il companion computer non è richiesto dato che tutto il carico computazionale è spostato su un calcolatore esterno.

2.4 MAVlink e MAVSDK

MAVlink[3] è un protocollo di comunicazione molto efficiente e robusto che viene utilizzato sia per comunicare con il drone, sia per lo scambio di messaggi tra i vari componenti presenti a bordo. Esempi di messaggi possono essere la telemetria trasmessa dal drone a terra, i comandi di movimento trasmessi dal controller ai motori, ecc.

Trattandosi di un protocollo di basso livello non ci addrentremo oltre nella sua trattazione ma faremo riferimento, da ora in avanti, alla libreria MAVSDK[4] che permette l'interfacciamento con il drone in modo più semplice e intuitivo. Ciò che fa MAVSDK è fornire all'utente una serie di funzioni di alto livello che internamente si occupano di gestire il protocollo MAVLINK. MAVSDK-Python[5] è la libreria che permetterà lo sviluppo del codice illustrato nei capitoli successivi.

2.5 Ground control station

La ground control station è un software che viene eseguito su un dispositivo di terra (come per esempio un PC) e che permette di controllare i droni tramite un'interfaccia grafica. Nel seguito, la ground control station utilizzata sarà QGControl[6] (QGC in breve) che permetterà di visualizzare uno o più droni in tempo reale su una mappa. Sebbene QGC permetta di programmare e di eseguire missioni, nel seguito verrà utilizzata solamente per monitorare gli spostamenti dei droni, che saranno controllati diversamente.

2.6 Simulatore, SITL, HITL

Durante le prime fasi di sviluppo del software di controllo è impensabile effettuare il testing su un drone reale, per questo sono stati sviluppati vari simulatori che permettono di testare le modifiche (sia hardware che software) in un ambiente virtuale.

I simulatori forniscono input realistici (vento, disturbi, accelerazioni, ecc.) ai sensori del drone (anch'esso simulato) e reagiscono opportunamente alle azioni del drone. Nei simulatori più all'avanguardia è possibile modificare l'ambiente così da permettere test più specifici aggiungendo, per esempio, ostacoli o altri elementi di interesse o di disturbo.

I simulatori trattati di seguito saranno Gazebo, Gazebo Classic[7] e JMAVSim.[8] I primi due sono i più accurati e personalizzabili, mentre il terzo rappresenta un'alternativa più leggera e minimale per effettuare semplici test.

Mentre i simulatori appena descritti permettono di simulare l'hardware e l'ambiente, PX4 mette a disposizione due metodi per il testing del software: il primo, utilizzato in questo progetto, prende il nome di SITL (Software In The Loop) e consiste nel far eseguire lo stack di volo allo stesso computer che esegue il resto del codice; il secondo prende invece il nome di HITL (Hardware In The Loop) e permette di interfacciare un ambiente simulato con un flight controller reale.

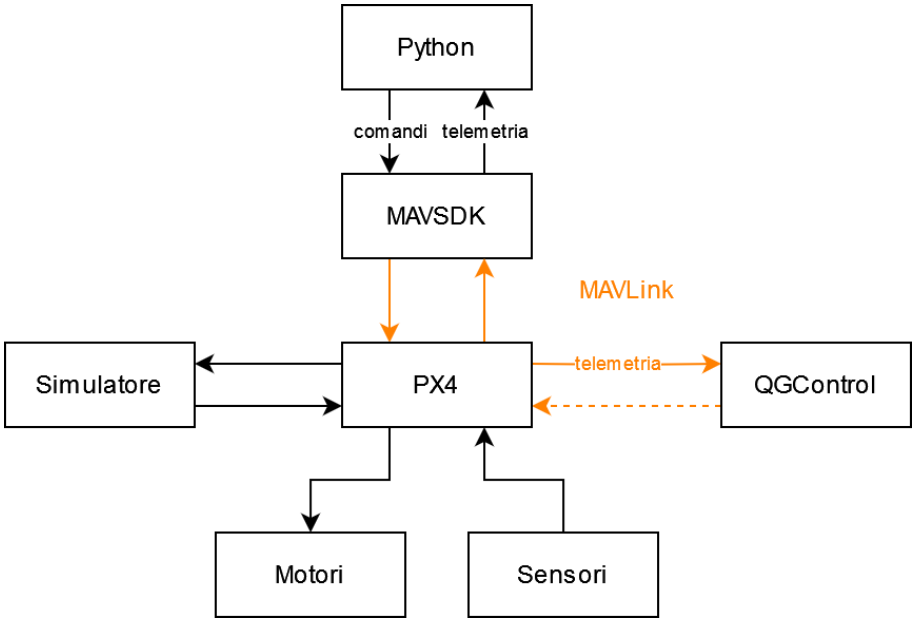

Figura 1: Scambi di messaggi che avvengono tra i vari componenti

In Figura 1 è possibile vedere uno schema semplificato che illustra le relazioni tra i vari componenti appena descritti. I dettagli di alcune relazioni verranno trattati più accuratamente in seguito, per gli altri si rimanda alla documentazione ufficiale.

3 Studi preliminari

Nella prima fase del progetto, l'attenzione è stata rivolta alla comprensione approfondita delle tecnologie sopra illustrate. Questo passaggio è stato fondamentale per capire quali fossero i ruoli di ciascun componente e quale fossero le funzionalità disponibili.

Le principali difficoltà incontrate sono state quelle dovute alla mancanza di documentazione o alla scarsa qualità della stessa. Dato l'utilizzo in ambito industriale di questi prodotti, la maggior parte del materiale disponibile era infatti rivolto ad un pubblico esperto.

Le principali fonti di informazioni sono state le pagine di documentazione ufficiale di PX4[1] e MAVSDK,[4] sfortunamenente poco aggiornate e carenti di informazioni basilari.

3.1 Ambiente di sviluppo

Per prendere familiarità con i vari componenti è stato installato, fin da subito, l'ambiente di sviluppo suggerito da PX4 seguendo la guida ufficiale.[9]

Il lavoro è stato eseguito su un PC Intel Core i7-8750H con scheda video NVIDIA GeForce GTX 1050 Ti e Windows 11. Dato che la maggior parte dei programmi

necessari sono sviluppati per Linux, si è deciso di utilizzare l'ambiente WSL2 per facilitare il testing. Come sarà chiaro più avanti, questa scelta presenta degli importanti trade-off di cui è necessario prendere atto.

Di seguito verranno illustrati i principali tentativi fatti e risultati ottenuti. Per un elenco dettagliato dei passaggi seguiti fare riferimento alla repository del progetto.[10]

3.1.1 Simulatori

I componenti che hanno dato maggiori problemi durante l'installazione sono stati i simulatori Gazebo, Gazebo Classic e JMAVSim. Nonostante non sia necessario installare più di un simulatore, è preferibile avere a disposizione sia una versione di Gazebo per simulazioni realistiche, sia JMAVSim, che è più leggero dal punto di vista computazionale, per testing durante lo sviluppo.

Gazebo

Per quanto riguarda Gazebo è necessario fare un'importante premessa prima di procedere con la trattazione. Fino ad Aprile 2022 erano presenti due versioni del simulatore:[11] Gazebo e Ignition. Gazebo rappresentava la prima architettura del simulatore, mentre Ignition era la versione più nuova. Dopo Aprile 2022 però è stato effettuato un cambio di nomi: quello che prima era chiamato Gazebo è stato rinominato Gazebo Classic, mentre il vecchio Ignition è stato rinominato Gazebo. Questo cambiamento, seppur banale, deve essere tenuto in considerazione durante la ricerca di informazioni in rete poiché il termine "Gazebo" può riferirsi a due versioni del simulatore distinte e non compatibili tra di loro. Da ora in avanti verrà utilizzata la nomenclatura corrente.

Entrambe le versioni del simulatore sono compatibili solamente con una specifica versione di Ubuntu: Gazebo è installabile solamente su Ubuntu 22.04 e successivi, mentre Gazebo Classic su Ubuntu 20.04.

Dal momento che Gazebo Classic ha già raggiunto il termine dello sviluppo[7] (verrà mantenuto fino al 2025 e non verranno rilasciate altre versioni), la scelta più ovvia sembrerebbe quella di utilizzare Gazebo. Al momento della scrittura però Gazebo, essendo più recente, ha meno funzionalità e una documentazione più scarna rispetto al suo predecessore.[9] Utilizzando una macchina virtuale o, come in questo caso, WSL, si possono avere i benefici di entrambe le versioni mantenendo il flusso di lavoro piuttosto lineare.

Nonostante quanto appena detto, non è stato possibile utilizzare Gazebo su Ubuntu 22.04 ad un frame rate maggiore di 4 fps per problemi di compatibilità con la scheda video. Per questo motivo si è deciso di non utilizzare questa versione del software dato che avrebbe ostacolato notevolmente lo sviluppo. Per quanto riguarda Gazebo Classic invece l'installazione è andata a buon fine su Ubuntu 20.04 e dai test effettuati le simulazioni avevano un frame rate di circa 60 fps.

JMAVSim

A differenza dei simulatori appena discussi, JMAVSim, in teoria, non dovrebbe avere dipendenze dalla versione di Ubuntu utilizzata. Nella pratica però sono emersi problemi di compatibilità su Ubuntu 20.04 (dove invece veniva eseguito correttamente Gazebo Classic).

Conclusioni

Date le problematiche appena illustrate si è preferito, nella fase iniziale, utilizzare solamente JMAVSim, ritenuto più che sufficiente per semplici simulazioni di test con pochi droni.

Dato che la scelta del simulatore non condiziona in alcun modo lo sviluppo del codice, nel seguito, ove possibile, verranno evitati riferimenti al simulatore specifico.

3.1.2 QGroundControl

L'installazione di QGC è stata portata a termine senza difficoltà seguendo la procedura illustrata nella documentazione.

3.1.3 Python

Come accennato in precedenza, durante lo sviluppo è stata utilizzata la versione 3.10 di Python. Eccetto che per le librerie specifiche per il progetto, discusse in dettaglio in seguito, l'unica accortezza presa è stata quella di utilizzare `pipenv`[12] per la gestione degli ambienti virtuali. Questa scelta si è rivelata molto utile per permettere il testing di librerie diverse senza la preoccupazione di alterare gli altri progetti.

3.1.4 Altri strumenti utili

Durante le fasi di test, per semplificare il flusso di lavoro, sono stati sviluppati alcuni semplici script basati su `tmux`[13] per lanciare più istanze del simulatore in un unico comando. Gli script in questione sono disponibili sulla repository del progetto.[10]

3.2 Libreria MAVSDK-Python

La maggior parte del lavoro svolto in questa tesi si è sviluppato intorno alla libreria MAVSDK-Python che permette di interfacciarsi con uno o più droni tramite script python. Come accennato in precedenza, la libreria permette di generare e inviare sequenze di messaggi MAVLink costruite per effettuare le più comuni operazioni di controllo del drone.

Prima di analizzare uno script di esempio è utile capire quali sono le operazioni necessarie per controllare un drone simulato:

1. Avviare un'istanza del simulatore SITL. Il drone verrà reso disponibile sulla porta specificata, di default la porta UDP 14540.

2. Avviare un'istanza di un server MAVSDK che si interfacci con il drone. Questa operazione viene gestita dalla libreria ed è perlopiù trasparente agli occhi dell'utente.
3. Connettersi al drone sulla porta specificata.
4. Far decollare il drone.
5. Ottenere la telemetria e controllare lo spostamento del drone.
6. Far atterrare il drone.

L'operazione che, per ovvi motivi, richiede la maggior attenzione è quella di controllo del drone. Nel nostro caso utilizzeremo solamente la modalità di controllo a set point, indicheremo cioè al drone delle coordinate a cui spostarsi e sarà il controllore a eseguire le azioni necessarie per portare il drone nella posizione richiesta. Nonostante il drone sia controllato dall'esterno, non faremo mai uso della modalità `offboard`,[14] il cui nome può trarre in inganno. In questa modalità sarebbe infatti necessario controllare direttamente il movimento dei motori, prestando quindi attenzione a tutti quegli aspetti che, normalmente, vengono svolti dal flight controller.

3.2.1 Script di esempio

Adesso che abbiamo trattato da un punto di vista funzionale i passaggi da seguire, possiamo analizzare un semplice script che permette di far decollare e atterrare un drone.

```
import asyncio
from mavsdk import System

async def run():
    drone = System()
    await drone.connect(system_address="udp://:14540")

    status_text_task =
↪   asyncio.ensure_future(print_status_text(drone))

    print("Waiting for drone to connect...")
    async for state in drone.core.connection_state():
        if state.is_connected:
            print(f"-- Connected to drone!")
            break

    print("Waiting for drone to have a global position
↪   estimate...")
    async for health in drone.telemetry.health():
        if health.is_global_position_ok and
↪   health.is_home_position_ok:
            print("-- Global position estimate OK")
            break
```

```
    print("-- Arming")
    await drone.action.arm()

    print("-- Taking off")
    await drone.action.takeoff()

    await asyncio.sleep(10)

    print("-- Landing")
    await drone.action.land()

    status_text_task.cancel()

if __name__ == "__main__":
    asyncio.run(run())
```

Dove la funzione `print_status_text`, che si occupa di stampare alcune informazioni sul drone, non è stata riportata per brevità.

Nonostante il codice sia di facile comprensione, è opportuno fare attenzione a qualche dettaglio che diventerà importante successivamente. Prima di tutto va notato che la maggior parte dei metodi della classe `System` sono asincroni. Il perché risulta chiaro se si pensa che questi metodi svolgono azioni che possono durare a lungo e che potrebbero addirittura non terminare; basti pensare all'azione di `takeoff` che porta il drone ad una quota determinata: il drone potrebbe impiegare secondi, se non minuti, a raggiungere la quota richiesta o potrebbe addirittura non raggiungerla per problemi di vario tipo. Si vede quindi che è necessario permettere al flusso di esecuzione del programma di continuare mentre l'azione richiesta viene svolta in background.

Sebbene necessaria, questa caratteristica può portare a errori se non gestita correttamente: osservando il codice sopra riportato si vede infatti che la chiamata di `drone.connect` non risulta in un'immediata connessione con il drone, tant'è che, poche righe sotto, il flusso di esecuzione viene forzatamente interrotto in attesa che la connessione sia ultimata.

Se non sono stati fatti errori, l'esecuzione dello script dovrebbe far decollare e atterrare il drone sul simulatore. Per avere una miglior panoramica degli spostamenti dei droni, e per testare tutti i componenti insieme, prima di eseguire lo script è suggeribile avviare anche QGC che dovrebbe automaticamente connettersi al drone e mostrare la sua posizione in tempo reale sulla mappa.

Altri script di esempio più complessi possono essere trovati sulla repository di MAVSDK-Python.[15]

4 Scelte progettuali iniziali

Una volta appurate le funzionalità e le limitazioni delle tecnologie a disposizione si è proceduto a valutare la migliore architettura hardware e software per portare a termine il progetto. Di seguito verranno illustrate le decisioni prese e le motivazioni che hanno portato a percorrere una strada piuttosto che un'altra.

4.1 Architettura centralizzata o distribuita

La prima scelta effettuata è stata quella di stabilire se fosse da preferire un'architettura distribuita oppure una centralizzata. La prima prevede che il software di controllo dei droni sia in esecuzione sui droni stessi, la seconda, invece, consiste nell'eseguire il medesimo codice su un unico calcolatore separato. Il fattore che, come accennato in precedenza, ci ha fatto scegliere la seconda opzione è stata la semplicità realizzativa a fronte di un basso numero di aspetti negativi. Scegliendo infatti un'architettura distribuita sarebbe stato necessario, in primo luogo, trovare un modo per permettere a ciascun drone di comunicare con gli altri; seppur possibile, questa scelta avrebbe complicato notevolmente sia l'hardware (richiedendo un maggior numero di periferiche), sia il software, non più atomico. I principali vantaggi di questa struttura sarebbero stati quelli di eliminare la dipendenza da una stazione di terra e quello di minimizzare il tempo di reazione dei droni. Il primo vantaggio illustrato è stato ritenuto superfluo per gli scopi del progetto, mentre il secondo si è rivelato, a tutti gli effetti, non significativo, dato che il software di controllo da sviluppare si occuperà di prendere decisioni sul breve-lungo periodo (nell'ordine dei minuti).

Fatta questa scelta fondamentale, l'hardware necessario viene di conseguenza: oltre al corpo del drone avremo bisogno di un flight controller, di un modo per far comunicare il drone con il computer di terra e del computer stesso. Come detto nel primo capitolo, la comunicazione tra drone ed esterno è gestita in autonomia dalla flight board, che rende quindi inutile l'uso di un companion computer. Il computer di terra dovrà poi essere equipaggiato, durante l'utilizzo reale, con un'antenna in grado di comunicare con i droni; nel nostro caso faremo però uso solamente delle interfacce simulate messe a disposizione dai simulatori.

4.2 Comunicazione tra droni

Il primo requisito del progetto era quello di permettere a due o più droni di scambiarsi informazioni. Avendo però deciso di fare uso di una struttura di controllo centralizzata, questa funzionalità è stata realizzata automaticamente dato che, tramite il protocollo MAVLink e la librerira MAVSDK, le informazioni di telemetria sono già disponibili senza bisogno di ulteriori azioni. Non avendo nessuna logica di alto livello in esecuzione sui droni, svanisce anche la necessità di trasmettere l'informazione al drone stesso: il software centralizzato ha una visione di insieme sullo stato dei droni che gli permette di prendere decisioni di conseguenza.

4.3 Individuazione degli obiettivi

Il secondo requisito di progetto era quello di rendere i droni capaci di individuare un generico obiettivo, così da permettere, in futuro, il testing di algoritmi di ricerca in flocking.

In un primo momento è stata valutata la possibilità di utilizzare sensori hardware (simulati) installati sul drone. Questa scelta, nonostante fosse la più realistica, si è rivelata poco pratica in questa fase. I principali svantaggi individuati possono essere riassunti in tre categorie che verranno approfondite in seguito: dipendenza dall'hardware, dipendenza dal simulatore, difficoltà implementative.

4.3.1 Dipendenza dall'hardware

L'uso di un sensore reale porta con sé la necessità di scegliere una tipologia specifica di sensore che potrebbe non essere compatibile con tutti i modelli di drone. In questa fase, non sapendo quali siano le applicazioni pratiche del progetto, sarebbe preferibile non prendere scelte che potrebbero limitare o rendere più difficile lo sviluppo futuro.

4.3.2 Dipendenza dal simulatore

Come illustrato in precedenza, è stato osservato che l'installazione dei simulatori può rivelarsi ostica in base all'hardware a disposizione; per questo motivo è stato di nuovo scelto di adottare una strategia il più generale possibile, che non fosse quindi vincolata ad uno specifico simulatore. In base al programma utilizzato sarebbe infatti necessario utilizzare modelli diversi di drone e di sensore. Dato l'utilizzo prevalente di JMAVSim, che non permette l'installazione di altri sensori sul drone, quanto detto si è rivelato ancora più importante durante lo sviluppo.

4.3.3 Difficoltà implementative

In aggiunta a quanto appena detto, va considerato il fatto che l'utilizzo di un sensore reale avrebbe richiesto la presenza di un obiettivo reale compatibile con il sensore scelto. Questo fatto, oltre che reintrodurre i due problemi appena visti, avrebbe richiesto di modificare manualmente l'ambiente di simulazione per cambiare la posizione dell'obiettivo ed effettuare test in condizioni diverse.

La scelta presa per superare i problemi appena illustrati ha richiesto di separare logicamente il codice di gestione del drone dal codice per l'individuazione dell'obiettivo, così da poter introdurre un'interfaccia drone-sensore, indipendente dal sensore utilizzato. Tramite questa scelta, analizzata in maggior dettaglio in seguito, sarà necessario fornire al codice di controllo un valore numerico che rappresenti la lettura del sensore, indipendentemente da cosa questa lettura indichi.

Una volta introdotta questa generalizzazione, la scelta fatta è stata quella di creare un sensore e un obiettivo virtuali basati sulle coordinate GPS. All'avvio del programma verranno creati uno o più obiettivi virtuali a coordinate GPS arbitrarie, il codice di gestione del sensore verrà poi sostituito da un semplice controllo della

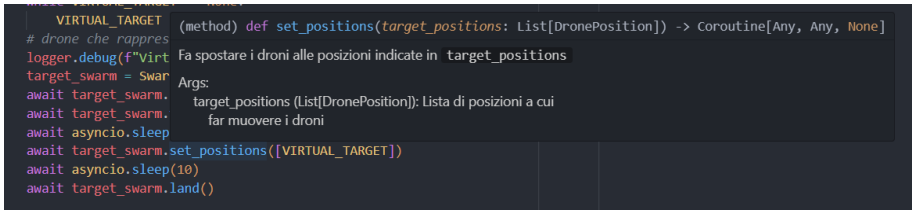

Figura 2: Esempio di suggerimento mostrato dall'IDE VS Code

distanza tra il drone e i target virtuali. Fintanto che il codice rispetta l'interfaccia descritta poco sopra non saranno necessari cambianti nel caso in cui si decida di passare al testing di un sensore reale.

5 Sviluppo del software

Dopo aver illustrato le scelte prese possiamo passare all'analisi dettagliata del codice prodotto. Va tenuto in mente che questo codice rappresenta il punto di partenza per possibili progetti futuri e che, per questo, molte delle scelte implementative sono state prese per permettere una maggior flessibilità in futuro.

Durante lo sviluppo sono state anche prese varie accortezze per facilitare l'utilizzo del codice prodotto in altri progetti. Oltre all'utilizzo di commenti per chiarire le sezioni di codice di più difficile comprensione, è stato fatto uso della libreria `typing`[16] per permettere la tipizzazione dei parametri in ingresso e in uscita alle varie funzioni. Questa funzionalità permette all'utente di ricevere suggerimenti dall'IDE sul tipo dei parametri che una funzione si aspetta o sul tipo che restituisce.

Un'altra feature di cui è stato fatto uso sono le docstrings,[17] speciali commenti che possono essere utilizzati sia per generare automaticamente una breve documentazione di classi e funzioni, sia a runtime tramite l'attributo `__doc__` per accedere alla documentazione.

In Figura 2 è riportato un esempio di hint generato dall'IDE VS Code per la funzione `set_positions` a partire da quanto appena detto.

5.1 Classe SystemWrapper

La classe `SystemWrapper` ha lo scopo di nascondere parte della complessità vista precedentemente nello script di esempio e gestire in modo atomico la connessione con la classe `System` di MAVSDK. Nonostante non necessario, questo ha permesso, nel seguito, di creare codice più semplice e di conseguenza più leggibile. Sebbene in questa fase la gestione del `System` sia piuttosto semplice, è possibile che in futuro la complessità aumenti notevolmente con l'aggiunta di funzionalità o con il passaggio al testing su droni reali.

```
from loguru import logger
from mavsdk import System
```

```python
import random

class SystemWrapper:
    @logger.catch
    def __init__(self,
                 system_addr:int) -> None:
        self.system_addr = system_addr
        self.server_port = random.randint(1000, 65535)

        logger.debug(f"Creating System:
        ↪  system_addr={self.system_addr},
        ↪  server_port={self.server_port}")
        self.system = System(port=self.server_port)

    @logger.catch
    async def connect(self) -> System:
        logger.debug(f"Connecting to system@{self.system_addr}")
        await self.system.connect(f"udp:// {self.system_addr}")
        async for state in self.system.core.connection_state():
            if state.is_connected:
                logger.debug("Connection completed")
                break

        logger.debug("Waiting for drone to have a global position
        ↪  estimate...")
        async for health in self.system.telemetry.health():
            if health.is_global_position_ok and
            ↪  health.is_home_position_ok:
                logger.debug("Global position estimate OK")
                break
        return self.system
```

Come si può notare, il costruttore della classe prende come unico parametro di ingresso un intero che rappresenta l'indirizzo a cui è possibile contattare il drone e crea un'istanza della classe `System` della libreria MAVSDK.

La classe fornisce poi un metodo per connettersi al `System` che gestisce sia la connessione vera e propria che l'attesa di una telemetria sufficiente per procedere con la fase di arm. La struttura della `connect` permette di capire perché sia necessaria questa classe: molte delle funzioni messe a disposizione dalla libreria vengono spesso, nella pratica, utilizzate per portare a termine macro operazioni che possono essere raggruppate insieme.

5.2 Classe DronePosition

Per semplificare la gestione delle posizioni dei vari droni è stata creata una classe destinata, prima di tutto, a contenere le coordinate spaziali del drone (latitudine,

logitudine e altezza) e a permettere conversioni sia di unità di misura che di formato. Come sarà chiaro tra poco, non tutte le funzioni della libreria MAVSDK utilizzano lo stesso formato per una posizione e, non meno importante, questi formati possono risultare poco pratici da utilizzare dal punto di vista dell'utente.

5.2.1 Costruttori

Sono stati realizzati due costruttori per questa classe: il primo, quello di default, prende in ingresso tre `float` che rappresentano, nell'ordine, latitudine in gradi, longitudine in gradi e altezza assoluta sul livello del mare in metri. Il secondo costrutture invece, realizzato mediante il decoratore `@classmethod`, prende come unico parametro un'istanza della classe `telemetry.Position`,[18] utilizzata da MAVSDK per restituire la posizione di un drone.

```
def __init__(self,
             latitude_deg:float,
             longitude_deg:float,
             absolute_altitude_m:float) -> None:
    self.latitude_deg = latitude_deg
    self.longitude_deg = longitude_deg
    self.absolute_altitude_m = absolute_altitude_m

@classmethod
def from_mavsdk_position(cls, pos:telemetry.Position) -> None:
    return cls(pos.latitude_deg, pos.longitude_deg,
    ↪  pos.absolute_altitude_m)
```

5.2.2 Conversione in altri formati

La funzione `goto_location`[19] richiede i parametri `latitude_deg`,`longitude_deg`, `absolute_altitude_m`, `yaw_deg`. Per questo motivo si è reso necessario inserire un metodo per convertire dal formato standard a quello appena illustrato. L'unico parametro, opzionale è un'altra istanza della classe `DronePosition` che rappresenta la posizione di partenza del drone. Dato che stiamo lavorando con quadricotteri, l'angolo con cui il drone si dirige verso la posizione specificata non è univocamente definito. Per semplificare il codice, l'utente della classe può quindi specificare la posizione di partenza del drone così che l'angolo di yaw venga automaticamente calcolato; così facendo il drone giungerà a destinazione effettuando uno spostamento lungo il proprio asse longitudinale.

In fase di testing è stato utilizzato un calcolo dell'angolo di yaw approssimato utilizzando semplici formule trigonometriche. Questa approssimazione può essere soddisfacente per piccole distanze ma deve essere sostituita con calcoli più accurati per lunghe distanze o per ottenere risultati più veritieri.

```
def to_goto_location(self, prev_pos:'DronePosition'=None) ->
↪  List[float]:
    if prev_pos == None:
        yaw = 0
    else:
        # ATTENZIONE: i calcoli fatti qui sono solo
        ↪  un'approssimazione
        d_lat = self.latitude_deg - prev_pos.latitude_deg
        d_lon = self.longitude_deg - prev_pos.longitude_deg
        tan_angle = 90 + d_lon/d_lat
        yaw = math.atan(tan_angle)
    return (self.latitude_deg, self.longitude_deg,
    ↪  self.absolute_altitude_m, yaw)
```

5.2.3 Spostamenti relativi

Come appena visto, MAVSDK si aspetta sempre una posizione che fa uso di latitudine e longitudine. Sebbene questa scelta abbia senso dal punto di vista della gestione del drone, che fa uso del gps per raggiungere una certa posizione, può non essere altrettanto pratica dal punto di vista dell'utente, che potrebbe voler spostare un drone di un numero arbitrario di metri in una o più direzioni. Per semplificare questa operazione sono state innanzitutto create due funzioni che convertono gradi in metri e viceversa; come per il caso precedente queste conversioni sono molto approssimative[20] e sono accettabili solamente in casi di test su piccole distanze. In casi d'uso reali è necessario utilizzare apposite funzioni per il calcolo di distanze geografiche non trattate in questa tesi.

Una volta trovato il modo di convertire da un'unità di misura all'altra è stato realizzato un metodo che, data una terna di incrementi in metri, restituisce una nuova `DronePosition` le cui coordinate corrispondono a quella di partenza incrementata delle quantità specificate.

```
def deg_to_m(deg) -> float:
    # 1 deg = 111319.9 m
    # questo calcolo non è accurato
    return deg * 111319.9

def m_to_deg(m) -> float:
    # questo calcolo non è accurato
    return m / 111319.9

def increment_m(self, lat_increment_m, long_increment_m,
↪  alt_increment_m) -> 'DronePosition':
    # questo calcolo non è accurato
    new_lat = self.latitude_deg + m_to_deg(lat_increment_m)
```

```
    new_lon = self.longitude_deg + m_to_deg(long_increment_m)
    new_alt = self.absolute_altitude_m + alt_increment_m
    return DronePosition(new_lat, new_lon, new_alt)
```

5.2.4 Calcolo delle distanze

Il metodo qui illustrato si occupa di calcolare la distanza in metri tra due istanze della classe `DronePosition` facendo uso, in questo caso, della libreria `geopy`.[21] È importante specificare che la distanza tra i due punti è calcolata come se entrambi fossero situati sulla superficie terrestre, non tenendo quindi conto della quota.

```
def distance_2D_m(self, point:'DronePosition') -> float:
    point1 = (self.latitude_deg, self.longitude_deg)
    point2 = (point.latitude_deg, point.longitude_deg)

    distance = geo_distance.distance(point1, point2).meters
    return distance
```

5.2.5 Conversione in stringa

Per facilitare il debugging, è stato ridefinito il metodo `__str__` che indica come convertire un'istanza della classe in stringa. Come si può vedere di seguito, il metodo restituisce il nome di tutti gli attributi della classe accompagnati dal relativo valore.

```
def __str__(self):
    return '%s(%s)' % (
        type(self).__name__,
        ', '.join('%s=%s' % item for item in vars(self).items())
    )
```

5.3 Classe Swarm

La classe al centro di questo lavoro è la classe `Swarm` che permette di rappresentare e di gestire più droni come se fossero una singola entità. Dal punto di vista dell'utente la classe permette di gestitire un numero arbitrario di droni come se fossero una lista di punti nello spazio.

5.3.1 Costruttore

Il costruttore della classe si occupa di gestire la parte di basso livello della connessione con i droni facendo uso della classe `SystemWrapper` vista in precedenza. La classe ha un attributo statico `next_drone_address` che specifica quale sia il prossimo indirizzo del drone (simulato). Si è scelto di utilizzare una variabile statica per permettere l'istanziazione di più classi `Swarm` senza dare luogo a conflitti sugli indirizzi.

Il costruttore prende in ingresso tre parametri: il primo di essi, `target_scanner` verrà approfondito più avanti dato che richiede un'analisi più estesa, il secondo e il terzo, invece, indicano il numero di droni e gli indirizzi ai quali questi droni sono raggiungibili. Nel caso in cui `drones_addrs` sia `None` verranno utilizzati gli indirizzi standard a partire da `next_drone_address`. Il corpo della funzione si occupa infine di settare gli attributi privati della classe e di generare gli indirizzi dei droni se necessario.

```
class Swarm:
    next_drone_address = 14540
    def __init__(self,
                 target_scanner:Callable[[System], float],
                 drones_number:int,
                 drones_addrs:List[int]=None) -> None:
        self.__target_scanner = target_scanner
        self.__discoveries:List[float] = []
        self.__drones_number = drones_number
        self.__positions = []
        self.__drones:List[System] = []

        if drones_addrs == None:
            self.drones_addrs = []
            for i in range(drones_number):
                self.drones_addrs.append(Swarm.next_drone_address)
                Swarm.next_drone_address += 1
        elif drones_number != len(drones_addrs):
            raise ValueError; "The number of drones specified does
            ↪  not match with the list size"
        else:
            self.drones_addrs = drones_addrs
        logger.info(f"Creating swarm with {self.__drones_number}
        ↪  drones at {self.drones_addrs}")
```

5.3.2 Operazioni di base sui droni

Questi tre metodi si occupano semplicemente di estendere a ciascun singolo drone le operazioni viste per la classe SystemWrapper.

```
async def connect(self):
    logger.info("Connecting to drones...")
    for a in self.drones_addrs:
        logger.info(f"Connecting to drone@{a}")
        sysW = SystemWrapper(a)
        drone = await sysW.connect()
        logger.info(f"Coonection to drone@{a} completed")
        self.__drones.append(drone)
```

```
async def takeoff(self):
    logger.info("Taking off")
    for d in self.__drones:
        await d.action.arm()
        await d.action.takeoff()
    logger.info("Takeoff completed")

async def land(self):
    logger.debug("Landing")
    for d in self.__drones:
        await d.action.land()
    logger.debug("Landing completed")
```

5.3.3 Acquisizione delle posizioni

Come già detto all'inizio del capitolo, lo scopo della classe è quello di fornire all'utente un modo per gestire un gruppo di droni trattandolo come una lista di punti nello spazio. Per permettere quindi all'utente di leggere le posizioni dei droni, si è pensato di utilizzare il decoratore `@property` per simulare il comportamento di una variabile. Analizzando il codice qui riportato è immediato vedere che la lettura delle posizioni dei droni non è un'operazione atomica: è infatti necessario leggere l'ultimo valore del generatore `position()` che restituisce la posizione del drone in un dato istante. Questa posizione deve poi essere, ovviamente, ripetuta per tutti i droni. Notiamo anche l'utilizzo del costruttore `from_mavsdk_position` discusso in precedenza.

```
@property
async def positions(self) -> List[DronePosition]:
    self.__positions = []
    for d in self.__drones:
        p = await anext(d.telemetry.position())
        pos = DronePosition.from_mavsdk_position(p)
        self.__positions.append(pos)

    return self.__positions
```

5.3.4 Controllo dei droni

Una volta permesso all'utente di leggere le posizioni dei droni, il passo successivo è quello di consentirne la modifica. La soluzione ottima, dal punto di vista della leggibilità, sarebbe quella di utilizzare il decoratore `@positions.setter`, python però non permette nativamente di costruire setter asincroni; nonostante sia disponibile la libreria `async-property` per ovviare a questo problema, si è preferito definire il

seguente metodo per non introdurre dipendenze superflue. Dualmente a quanto visto sopra, la funzione scorre la lista di posizioni passata come parametro e comanda a ciascun drone di spostarsi dove indicato tramite la funzione `goto_location`. Notiamo di nuovo l'utilità del metodo `to_goto_location` per effettuare la conversione della posizione nel formato richiesto.

```
async def set_positions(self,
↪  target_positions:List[DronePosition]):
        prev_pos = await self.positions
        print(prev_pos)
        for n, d in enumerate(self.__drones):
            pos = target_positions[n]
            logger.info(f"Moving drone@{self.drones_addrs[n]} to
            ↪  {pos}")
            await d.action.goto_location( *pos.to_goto_location(
            ↪  prev_pos[n]))
```

5.3.5 Riconoscimento degli obiettivi

Come già spiegato in precedenza, uno dei due requisiti del progetto era quello di permettere ai droni di rilevare vari tipi di obiettivo. Abbiamo visto che il metodo adottato è stato quello di creare un'interfaccia indipendente dal drone, dall'obiettivo e dal sensore. Per garantire la maggior generalità possibile quindi, il costrutture della classe Swarm prende in ingresso una funzione `target_scanner`, questa funzione deve necessariamente avere come unico parametro un'istanza della classe System e deve restituire un `float` compreso tra `0` e `1`. La funzione verrà poi chiamata dalla classe `Swarm` su ciascun drone per valutare la percentuale di riconoscimento corrente. Questa scelta permette alla classe di ignorare interamente i dettagli implementativi; starà all'utente preoccuparsene. Se, per esempio, l'utente volesse utilizzare una termocamera per determinare la presenza di incendi, la funzione `target_scanner` dovrebbe prelevare i dati dalla termocamera, valutare la probabilità che quei dati corrispondano ad un incendio e restituire un valore opportuno (vicino a `0` per una temperatura molto bassa e vicino a `1` per una temperatura molto alta).

```
@property
async def discoveries(self) -> List[float]:
    self.__discoveries = []
    for d in self.__drones:
        self.__discoveries.append(await self.__target_scanner(d))

    return self.__discoveries
```

6 Esempio di utilizzo

Con le funzionalità appena introdotte, l'utente ha a disposizione tutti gli strumenti necessari per mettere in pratica algoritmi di controllo più complessi. In questo capitolo verrà illustrato un esempio che mette in luce le potenzialità del software prodotto e che possa chiarirne gli aspetti più complessi.

6.0.1 Codice dimostrativo

Di seguito è stato riportato un semplice script per dare un'idea esaustiva di come utilizzare la classe `Swarm`.

```
from swarm import Swarm
import asyncio
import random
from droneposition import DronePosition
from loguru import logger
import math

# Sarà inizializzata a un'istanza della classe DronePosition
# che rappresenta la posizione dell'obiettivo virtuale
VIRTUAL_TARGET = None

# Funzione di testing che si occupa di creare un obiettivo
↪  virtuale
# nelle vicianze del punto di decollo dei droni. Più nello
↪  specifico
# l'obiettivo virtuale verrà generato all'interno di un cerchio di
↪  raggio
# `max_distance` dal punto di decollo del primo drone della
↪  `swarm`
async def create_virtual_target(swarm:Swarm, max_distance) ->
↪  DronePosition:
    assert max_distance > 1, "Il target non può coincidere con i
    ↪  droni"

    # recupero le informazioni sui droni
    starting_pos = await swarm.positions
    # creo un punto casuale in un cerchio di raggio `max_distance`
    ↪  centrato
    # nel punto di decollo del primo drone
    center = starting_pos[0]
    alpha = 2 * math.pi * random.random()
    u = random.random() + random.random()

    r = max_distance * (2 - u if u > 1 else u)
    target_x_incr = r * math.cos(alpha)
```

```
    target_y_incr = r * math.sin(alpha)
    target_z_incr = random.randrange(0, max_distance)

    # Il virtual target coinciderà con il punto di decollo
    ↪  traslato
    # delle quantità appena calcolate
    virtual_target = center.increment_m(target_x_incr,
    ↪  target_y_incr, target_z_incr)
    return virtual_target

# Funzione passata alla Swarm per il riconoscimento
↪  dell'obiettivo.
# In questo caso fa riferimento al "sensore" (simulato) destinato
# al riconoscimento dei virutal target
async def target_scanner(drone) -> float:
    # Recupero la posizione del drone
    p = await anext(drone.telemetry.position())
    drone_pos = DronePosition.from_mavsdk_position(p)
    # Calcolo la distanza tra il drone e il VIRTUAL_TARGET
    distance = drone_pos.distance_2D_m(VIRTUAL_TARGET)
    # Restituisco un valore che diminuisce esponenzialmente con la
    ↪  distanza
    discovery = math.exp(-distance/10)
    return discovery


async def main():
    global VIRTUAL_TARGET
    # Creo una swarm con 2 droni e mi ci connetto
    sw = Swarm(target_scanner, 2)
    await sw.connect()

    # Creo un VIRTUAL_TARGET (l'operazione è asincrona quindi
    ↪  attendo che
    # la creazione sia ultimata)
    while VIRTUAL_TARGET == None:
        VIRTUAL_TARGET = await create_virtual_target(sw, 100)
    logger.debug(f"Virtual Target: {VIRTUAL_TARGET}")

    # Per poter visualizzare la posizione del target su QGC creo
    ↪  una Swarm
    # composta da un singolo drone e faccio spostare il drone sul
    ↪  VIRTUAL_TARGET
    # (Questa operazione è solamente per scopi di debug e non ha
    ↪  niente a che
    # vedere con il funzionamento reale delle operazioni).
```

```
    # Dato che è solo per motivi di test, il valore della funzione
    ↪ target_scanner
    # non è significativo
    target_swarm = Swarm(lambda x: 0, 1)
    await target_swarm.connect()
    await target_swarm.takeoff()
    await asyncio.sleep(5)
    await target_swarm.set_positions([VIRTUAL_TARGET])
    await asyncio.sleep(10)
    await target_swarm.land()

    # Quanto segue è una serie di esempi che illustrano il
    ↪ funzionamento della
    # classe Swarm. Il comportamento atteso è che i droni si alzino
    ↪ in volo e
    # si spostino verso Nord, verso Est e in Alto di un numero di
    ↪ metri doppio
    # rispetto al proprio indice (il primo drone si sposta di 2
    ↪ metri a Est,
    # 2 metri a Nord e 2 metri in alto), il secondo invece di 4
    ↪ metri.
    # Durante gli spostamenti verranno stamapate le discoveries che
    ↪ saranno
    # tanto più alte quanto i droni saranno vicini al
    ↪ VIRTUAL_TARGET
    await sw.takeoff()
    disc = await sw.discoveries
    logger.info(f"Discoveries pre volo: {disc}")
    await asyncio.sleep(10)
    original_pos = await sw.positions

    new_poss = []
    for n, p in enumerate(original_pos):
        n += 1
        new_poss.append(p.increment_m(2*n, 2*n, 2*n))
    await sw.set_positions(new_poss)

    await asyncio.sleep(10)
    disc = await sw.discoveries
    logger.info(f"Discoveries in volo: {disc}")
    await sw.set_positions(original_pos)
    await asyncio.sleep(10)
    await sw.land()

if __name__ == "__main__":
    asyncio.run(main())
```

6.0.2 Macro struttura del codice

Di seguito è riportato un suggerimento di struttura del codice pensato per garantirne leggibilità e modularità. Le funzioni qui presenti non hanno un'implementazione reale ma servono semplicemente da segnaposto per funzioni realizzabili tramite quanto visto nel capitolo precedente. Oltre ad ampliare e perfezionare il codice prodotto starà a progetti successivi dedicarsi allo sviluppo degli algoritmi di controllo.

```
def main():
    # Eseguire le operazioni preliminari (connessione, decollo,
    ↪  ecc.)
    initial_setup()

    while True:
        # Recuperaere le posizioni dei droni
        get_drones_positions()
        # Recuperare informazioni sull'obiettivo acquisite dai
        ↪  droni
        get_drones_discoveries()
        # Implementare un algoritmo di controllo dei droni e farli
        # spostare di conseguenza
        move_drones()
```

Riferimenti bibliografici

[1] *PX4 User Guide*. Lug. 2023. URL: https://docs.px4.io/main/en.

[2] *Flight Controller Selection | PX4 User Guide*. Lug. 2023. URL: https://docs.px4.io/main/en/getting_started/flight_controller_selection.html.

[3] *Introduction · MAVLink Developer Guide*. Lug. 2023. URL: https://mavlink.io/en.

[4] *MAVSDK-Python API reference — MAVSDK-Python 1.4.8 documentation*. Giu. 2023. URL: http://mavsdk-python-docs.s3-website.eu-central-1.amazonaws.com.

[5] *MAVSDK-Python*. Lug. 2023. URL: https://github.com/mavlink/MAVSDK-Python.

[6] *QGC - QGroundControl - Drone Control*. Feb. 2020. URL: http://qgroundcontrol.com.

[7] *Gazebo*. Lug. 2023. URL: https://gazebosim.org/home.

[8] *jMAVSim*. Lug. 2023. URL: https://github.com/PX4/jMAVSim.

[9] *PX4 Development | PX4 User Guide*. Lug. 2023. URL: https://docs.px4.io/main/en/development/development.html.

[10] gamberoillecito. *tesi_droni*. Lug. 2023. URL: https://github.com/gamberoillecito/tesi_droni.

[11] *Open Robotics*. Lug. 2023. URL: https://www.openrobotics.org/blog/2022/4/6/a-new-era-for-gazebo.

[12] *pipenv*. Lug. 2023. URL: https://pypi.org/project/pipenv.

[13] tmux. *tmux*. Lug. 2023. URL: https://github.com/tmux/tmux/wiki.

[14] *Offboard Mode | PX4 User Guide*. Mar. 2023. URL: https://docs.px4.io/main/en/flight_modes/offboard.html#offboard-mode.

[15] mavlink. *MAVSDK-Python*. Lug. 2023. URL: https://github.com/mavlink/MAVSDK-Python/tree/main/examples.

[16] *typing — Support for type hints*. Lug. 2023. URL: https://docs.python.org/3/library/typing.html#module-typing.

[17] *PEP 257 – Docstring Conventions | peps.python.org*. Giu. 2022. URL: https://peps.python.org/pep-0257.

[18] *Telemetry — MAVSDK-Python 1.4.8 documentation*. Giu. 2023. URL: http://mavsdk-python-docs.s3-website.eu-central-1.amazonaws.com/plugins/telemetry.html#mavsdk.telemetry.Telemetry.position.

[19] *Action — MAVSDK-Python 1.4.8 documentation*. Giu. 2023. URL: http://mavsdk-python-docs.s3-website.eu-central-1.amazonaws.com/plugins/action.html#mavsdk.action.Action.goto_location.

[20] *Decimal degrees - GIS Wiki | The GIS Encyclopedia*. Apr. 2021. URL: http://wiki.gis.com/wiki/index.php/Decimal_degrees.

[21] *Welcome to GeoPy's documentation! — GeoPy 2.3.0 documentation*. Nov. 2022. URL: https://geopy.readthedocs.io/en/stable.