

# **Simulazione del Digital Twin di un Drone**

Industrial Internet of Things

**Antonino Maio**

## Indice

<b>1 Scopo del progetto</b>	<b>2</b>
<b>2 Introduzione</b>	<b>3</b>
<b>3 Ambiente di Simulazione del Drone</b>	<b>4</b>
3.1 PX4-Autopilot . . . . .	5
3.1.1 Installazione . . . . .	5
3.2 ROS 2 Foxy . . . . .	8
3.2.1 Installazione . . . . .	10
3.3 Micro XRCE-DDS . . . . .	13
3.3.1 Installazione . . . . .	13
3.4 QGroundControl . . . . .	14
3.5 Simulazione della missione . . . . .	15
3.6 Acquisizione dei dati tramite ROS 2 . . . . .	17
3.6.1 Build dei pacchetti e Utilizzo . . . . .	18
<b>4 Simulazione del Digital Twin su MATLAB</b>	<b>20</b>
4.1 Subsampling dei dati . . . . .	20
4.2 Livescript Matlab . . . . .	20
4.2.1 Elaborazione dei dati . . . . .	21
4.2.2 Simulazione del Digital Twin . . . . .	22
<b>5 Conclusioni</b>	<b>24</b>

## 1 Scopo del progetto

L'obiettivo di questo progetto è quello di realizzare la simulazione di un drone utilizzato per monitorare una regione di spazio con lo scopo di verificare la presenza del batterio **Xylella fastidiosa** negli ulivi delle zone monitorate, questo sarà realizzabile tramite la gestione delle camere presenti sul drone unite ad un algoritmo di classificazione che permetterà di analizzare le immagini acquisite dal drone e poi comunicare un'eventuale presenza del batterio nelle piante monitorate.

La parte di gestione delle camere, sviluppo dell'algoritmo di classificazione e pianificazione di una tratta per ottenere il massimo di informazioni riguardo le piante è soggetto di un'altra parte di questo progetto che non è stata affrontata durante lo sviluppo di questa e che quindi non verrà citata nella presente relazione.

Il focus principale di questa relazione sarà relativo alla simulazione della missione di un drone con lo scopo finale di acquisire i dati relativi al viaggio del drone stesso sia per possibili elaborazioni future sia per realizzarne un **Digital Twin** del drone stesso durante la sua missione per poterlo monitorare ed analizzare in tempo reale.

La relazione sarà divisa come di seguito: in una prima parte verrà introdotto e descritto il concetto di Digital Twin con le relative migliorie che l'utilizzo dello stesso può introdurre; nella seconda parte verrà descritto l'ambiente utilizzato per ottenere la simulazione di una missione del drone e come sono stati acquisiti i dati del viaggio; nella terza parte verrà descritta la simulazione in MATLAB utilizzata per ottenere il Digital Twin vero e proprio, evidenziando i limiti imposti dall'hardware nella realizzazione della stessa; infine verranno delineate le conclusioni e gli eventuali sviluppi futuri.

## 2 Introduzione

Il concetto di **Digital Twin** corrisponde ad una rappresentazione digitale di un oggetto fisico, questo permette quindi di ottenere informazioni riguardo l'oggetto stesso senza doverci essere fisicamente vicini. Esso è strettamente legato all'ambiente IoT poiché per ottenere tali informazioni a distanza è imprescindibile l'utilizzo di sensori o dispositivi IoT capaci di acquisire dati relativi all'oggetto stesso e poi trasmetterli in modo tale che essi possano essere elaborati in tempo reale.

Un Digital Twin è quindi una vera e propria copia dell'oggetto reale, che può essere un attrezzatura, un veicolo o un'insieme di essi, che viene costantemente aggiornata in tempo reale con i dati acquisiti e permette di simulare il comportamento dell'oggetto reale al fine di analizzarlo senza doverci essere a stretto contatto fisico. L'accuratezza della simulazione del Digital Twin dipende dall'accuratezza dei dati acquisiti di sensori stessi.

Tra i principali vantaggi che l'utilizzo dei Digital Twin comporta si possono annoverare i seguenti: riduzione del **time-to-market** permettendo uno sviluppo accelerato dei prodotti grazie alle simulazioni realizzate; la **manutenzione predittiva**, essendo un modello simulato dai dati reali, attraverso un Digital Twin è possibile notare e correggere un guasto in un componente al minimo segnale e quindi ancor prima che tutto il sistema possa essere intaccato dal problema stesso, andando a ridurre i downtime del sistema e sfociando in una maggiore affidabilità dell'intero sistema, che quindi si traduce in una **miglioria delle prestazioni**; infine il più semplice dei vantaggi introdotti dall'utilizzo del Digital Twin è quello di avere un **monitoraggio a distanza** del dispositivo al quale questo approccio è applicato, permettendo quindi di monitorare anche sistemi che si trovano in situazioni avverse per l'essere umano.

### 3 Ambiente di Simulazione del Drone

Per ottenere una simulazione del drone durante il suo viaggio sono stati impiegati diversi tools, nel seguente capitolo verranno innanzitutto brevemente descritti e successivamente verrà descritto il metodo d'installazione utilizzato e quindi anche il procedimento per arrivare allo scopo prefissato.

Per ottenere la demo della realizzazione è stata utilizzata una VM con Ubuntu 20.04, nella quale sono stati installati ed interfacciati tra di loro tutti i seguenti tools:

- PX4-Autopilot;
- QGroundControl;
- ROS 2 Foxy;
- uXRCE-DDS;

### 3.1 PX4-Autopilot

PX4-Autopilot è un sistema open-source realizzato per controllare e simulare veicoli terrestri ed aerei, concentrandosi maggiormente su veicoli a pilotaggio remoto e senza equipaggio. Esso supporta diversi tipi di veicoli, come descritto in precedenza, tra aerei e terrestri, e permette inoltre non solo la simulazione, ma anche il controllo a distanza degli stessi.

#### 3.1.1 Installazione

Per simulare il viaggio del drone è necessario installare la toolchain di sviluppo di PX4, per farlo bisogna configurare l'ambiente di sviluppo nel seguente modo:

```
git clone https://github.com/PX4/PX4-Autopilot.git --recursive  
bash ./PX4-Autopilot/Tools/setup/ubuntu.sh
```

Figure 1: Installazione PX4

Una volta installato è possibile far partire una prima build del sistema per valutarlo e verificare che funzioni tutto, la simulazione verrà fatta utilizzando **gazebo-classic** come simulatore, questo successivamente non verrà più visualizzato per evitare un sovraccarico computazionale che comporterebbe un innalzamento dei tempi di simulazione.

Per simulare i dati di un drone non realmente esistente si utilizza la simulazione SITL (*Software In The Loop*), ma PX4 permette anche l'implementazione di una simulazione HITL (*Hardware In the Loop*) attraverso la quale si può direttamente realizzare un digital twin partendo da un software installato direttamente su un drone reale dal quale acquisire i dati.

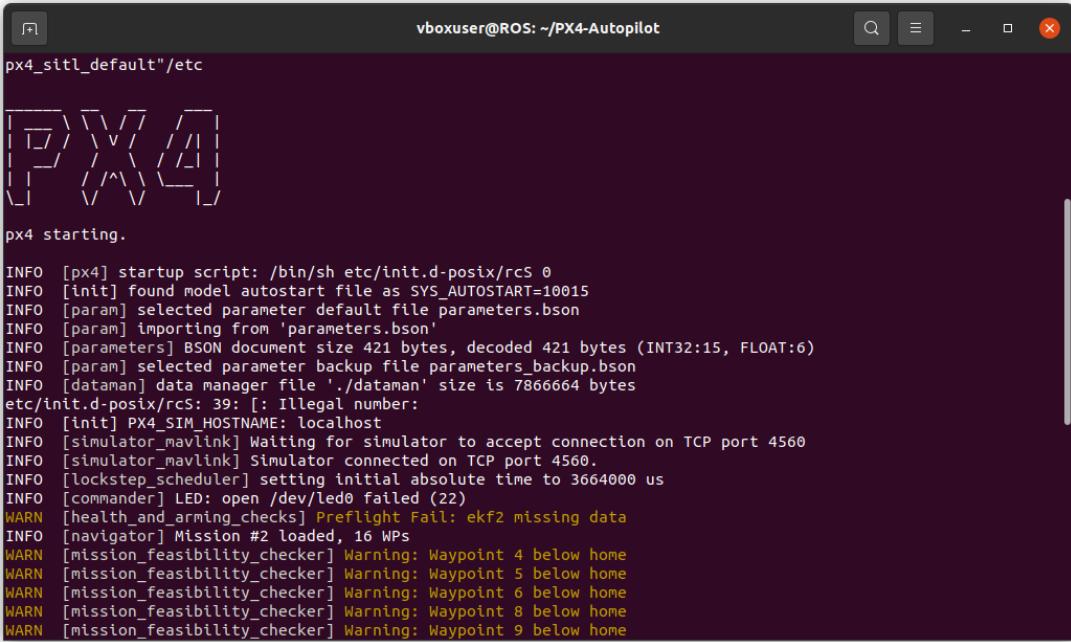
Come simulatore è stato scelto gazebo-classic in quanto PX4 si interfaccia direttamente con esso utilizzando le API di Gazebo, non richiedendo quindi

l’interfacciamento con protocolli di messaggistica terzi quali ad esempio MAVLink.

```
cd PX4-Autopilot/  
make px4_sitl gazebo-classic
```

Figure 2: Prima run PX4

Dopo aver eseguito i precedenti comandi sul terminale partirà l’interfaccia di PX4 e contestualmente il simulatore Gazebo verrà avviato mostrando un drone come nelle seguenti figure:



```
vboxuser@ROS: ~/PX4-Autopilot  
px4_sitl_default"/etc  
| ____ \ \ \ / / |  
| _/ / \ \ / / |  
| | / ^ \ \ / |  
| \ \ \ \ \ / |  
px4 starting.  
INFO [px4] startup script: /bin/sh etc/init.d-posix/rcS 0  
INFO [init] Found model autostart file as SYS_AUTOSTART=10015  
INFO [param] selected parameter default file parameters.bson  
INFO [param] importing from 'parameters.bson'  
INFO [parameters] BSON document size 421 bytes, decoded 421 bytes (INT32:15, FLOAT:6)  
INFO [param] selected parameter backup file parameters_backup.bson  
INFO [dataman] data manager file './dataman' size is 7866664 bytes  
etc/init.d-posix/rcS: 39: [: Illegal number:  
INFO [init] PX4_SIM_HOSTNAME: localhost  
INFO [simulator_mavlink] Waiting for simulator to accept connection on TCP port 4560  
INFO [simulator_mavlink] Simulator connected on TCP port 4560.  
INFO [lockstep_scheduler] setting initial absolute time to 3664000 us  
INFO [commander] LED: open /dev/led0 failed (22)  
WARN [health_and_arming_checks] Preflight Fail: ekf2 missing data  
INFO [navigator] Mission #2 loaded, 16 WPs  
WARN [mission_feasibility_checker] Warning: Waypoint 4 below home  
WARN [mission_feasibility_checker] Warning: Waypoint 5 below home  
WARN [mission_feasibility_checker] Warning: Waypoint 6 below home  
WARN [mission_feasibility_checker] Warning: Waypoint 8 below home  
WARN [mission_feasibility_checker] Warning: Waypoint 9 below home
```

Figure 3: Interfaccia terminale PX4

Utilizzando il comando *commander takeoff* nel terminale è possibile far decollare il drone nella simulazione di Gazebo

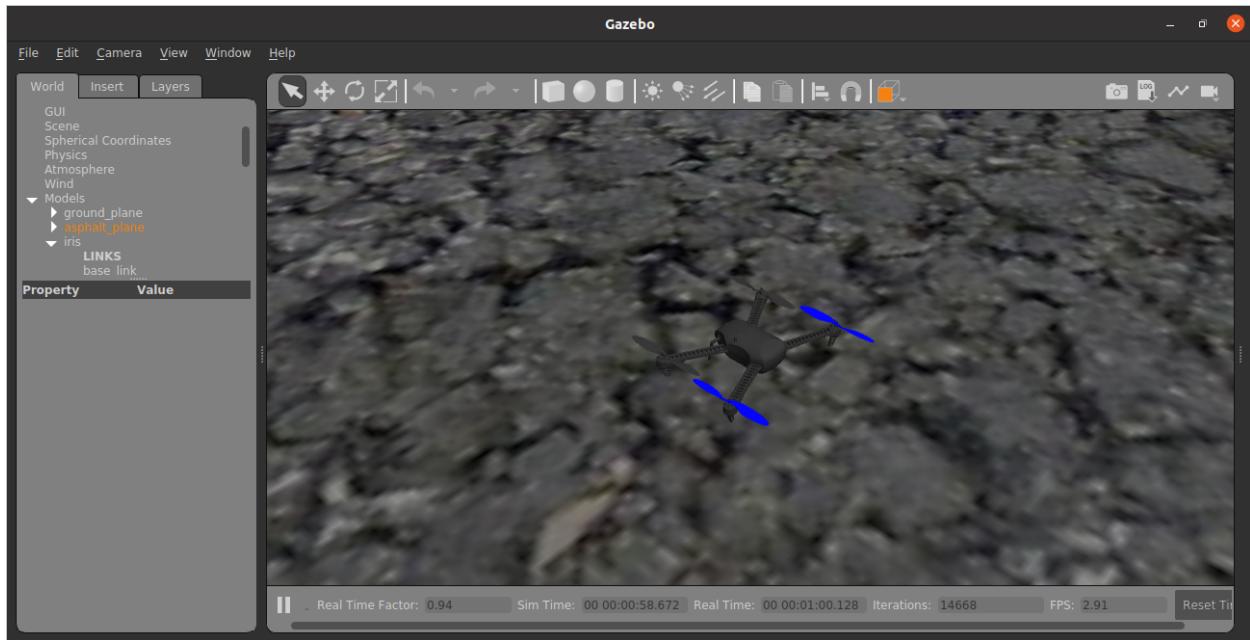


Figure 4: Simulatore Gazebo

### 3.2 ROS 2 Foxy

ROS (**R**obotics **O**perating **S**ystem), nonostante il nome non è un sistema operativo tradizionale, quanto piuttosto un sistema composto da middleware che permettono ad algoritmi sviluppati in linguaggi diversi di scambiare messaggi tra di loro e frameworks utili allo sviluppo di applicazioni di tipo robotico. Il progetto ROS inizia nel 2007 e dal 2017 è stata rilasciata la prima versione di ROS2, la distribuzione utilizzata nel progetto descritto in questa relazione è ROS2 Foxy, rilasciata nel 2020.

I principali concetti del progetto ROS sono i seguenti:

- **Open Source**, tutto il codice di ROS è disponibile pubblicamente, il che permette di implementare facilmente librerie compatibili con esso;
- **Leggero**, ROS è sviluppato per essere il più leggero possibile, utilizzando librerie standalone senza dipendenze da ROS stesso, in modo tale che applicazioni scritte per ROS possano essere utilizzate anche con altri frameworks;
- **Indipendenza dal linguaggio di programmazione**, ROS supporta diversi linguaggi di programmazione moderni, quali ad esempio Python, C++, Lisp ed Octave;
- **Peer-To-Peer**, un sistema sviluppato con ROS è un sistema distribuito, composto quindi da tanti processi che runtime vengono connessi con una topologia peer-to-peer.

ROS 2 è stato invece sviluppato per far fronte ad alcune limitazioni di ROS 1, soprattutto per quanto riguarda prodotti commerciali. Le sue caratteristiche principali sono:

- **Sistemi con più robot** invece che applicazioni basate su singoli robot;
- **Supporto al QoS**, permettendo ai programmati dei nodi publisher e subscriber di avere un maggior controllo su vengono scambiati i messaggi;
- **Nuovo protocollo di comunicazione**, ROS2 utilizza il middleware DDS per scambiare messaggi, esso è più flessibile e adatto a sistemi distribuiti.

ROS 2 fornisce tre differenti tipi di interfacce e conseguentemente tre differenti tipi di nodi:

1. **Topic**: Questo tipo di nodo viene principalmente usato in caso di stream di dati continue, i dati vengono pubblicati continuamente e indipendentemente da qualsiasi subscriber. In questo caso si ha una conessione di tipo *many-to-many*;
2. **Service**: Questo tipo di nodi sono basati su una comunicazione di tipo *request/response* e vengono usate per chiamate a routine che terminano velocemente. Un service implica l'utilizzo di una coppia client/server;
3. **Action**: Le Action vengono utilizzate per realizzare routine che durano più tempo rispetto ai Service, ma utilizzano lo stesso tipo di comunicazione.

### 3.2.1 Installazione

Per installare ROS 2 Foxy è innanzitutto necessario impostare la lingua del sistema in modo tale che supporti UTF-8, nel sistema utilizzato per realizzare la demo è stata impostata la lingua inglese come di seguito:

```
sudo apt update && sudo apt install locales  
sudo locale-gen en_US en_US.UTF-8  
sudo update-locale LC_ALL=en_US.UTF-8  
LANG=en_US.UTF-8  
export LANG=en_US.UTF-8
```

Figure 5: Impostazioni lingua per ROS 2

Successivamente bisogna impostare le sorgenti utili ad installare ROS 2, i comandi utilizzati sono riportati di seguito in Fig. 6.

```
sudo apt install software-properties-common  
sudo add-apt-repository universe  
sudo apt update && sudo apt install curl -y  
sudo curl -sSL  
https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o  
/usr/share/keyrings/ros-archive-keyring.gpg  
echo "deb [arch=$(dpkg --print-architecture)  
signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://pa  
ckages.ros.org/ros2/ubuntu  
$(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee  
/etc/apt/sources.list.d/ros2.list > /dev/null
```

Figure 6: Setup sources per ROS 2

Una volta impostate le sorgenti è possibile procedere con l'installazione di ROS 2 Foxy, nell'esempio specifico riportato in questa relazione è stata installata la versione desktop, comprensiva di demo e tutorial, sono presenti anche altre versioni tra cui *Base Install* e *Development Tools*.

```
sudo apt install ros-foxy-desktop python3-argcomplete
```

Figure 7: Installazione ROS 2 Foxy

Anche per ROS 2 è stato previsto un esempio di funzionamento per valutare l'installazione e verificare che funzioni tutto come previsto, in questo caso vengono utilizzati due esempi presenti già nell'installazione desktop che permettono di instaurare un semplice esempio talker-listener per poter inoltre verificare l'utilizzo delle API di C++ e Python da parte di ROS 2.

Innanzitutto per utilizzare ROS 2 è necessario eseguire il comando *source* sul file di setup, in modo tale da preparare l'ambiente, successivamente tramite il comando *ros2 run* vengono fatti partire gli esempi che comunicheranno tra di loro, di seguito riportati i comandi e gli output dei nodi.

```
source /opt/ros/foxy/setup.bash  
ros2 run demo_nodes_cpp talker
```

Figure 8: Nodo publisher in C++

```
source /opt/ros/foxy/setup.bash  
ros2 run demo_nodes_py listener
```

Figure 9: Nodo listener in Python

```
[INFO] [1694360279.871423423] [talker]: Publishing: 'Hello World: 1'
[INFO] [1694360280.871432273] [talker]: Publishing: 'Hello World: 2'
[INFO] [1694360281.871360262] [talker]: Publishing: 'Hello World: 3'
[INFO] [1694360282.871715500] [talker]: Publishing: 'Hello World: 4'
[INFO] [1694360283.871694349] [talker]: Publishing: 'Hello World: 5'
[INFO] [1694360284.871249095] [talker]: Publishing: 'Hello World: 6'
[INFO] [1694360285.871514486] [talker]: Publishing: 'Hello World: 7'
[INFO] [1694360286.871381527] [talker]: Publishing: 'Hello World: 8'
[INFO] [1694360287.871936198] [talker]: Publishing: 'Hello World: 9'
[INFO] [1694360288.871645833] [talker]: Publishing: 'Hello World: 10'
[INFO] [1694360289.871331813] [talker]: Publishing: 'Hello World: 11'
[INFO] [1694360290.871271755] [talker]: Publishing: 'Hello World: 12'
[INFO] [1694360291.871677580] [talker]: Publishing: 'Hello World: 13'
[INFO] [1694360293.059444342] [talker]: Publishing: 'Hello World: 14'
[INFO] [1694360293.874921167] [talker]: Publishing: 'Hello World: 15'
[INFO] [1694360294.871349527] [talker]: Publishing: 'Hello World: 16'
[INFO] [1694360295.873686054] [talker]: Publishing: 'Hello World: 17'
[INFO] [1694360296.871384761] [talker]: Publishing: 'Hello World: 18'
[INFO] [1694360297.873051085] [talker]: Publishing: 'Hello World: 19'
[INFO] [1694360298.871312481] [talker]: Publishing: 'Hello World: 20'
[INFO] [1694360299.871341608] [talker]: Publishing: 'Hello World: 21'
[INFO] [1694360300.871360465] [talker]: Publishing: 'Hello World: 22'
[INFO] [1694360301.874537157] [talker]: Publishing: 'Hello World: 23'
[INFO] [1694360302.892047994] [talker]: Publishing: 'Hello World: 24'
```

Figure 10: Output del nodo talker

```
ml
from pkg_resources import load_entry_point
[INFO] [1694360279.921779027] [listener]: I heard: [Hello World: 1]
[INFO] [1694360280.873335571] [listener]: I heard: [Hello World: 2]
[INFO] [1694360281.872764926] [listener]: I heard: [Hello World: 3]
[INFO] [1694360282.872856452] [listener]: I heard: [Hello World: 4]
[INFO] [1694360283.873736658] [listener]: I heard: [Hello World: 5]
[INFO] [1694360284.872465791] [listener]: I heard: [Hello World: 6]
[INFO] [1694360285.873036691] [listener]: I heard: [Hello World: 7]
[INFO] [1694360286.872725119] [listener]: I heard: [Hello World: 8]
[INFO] [1694360287.873979968] [listener]: I heard: [Hello World: 9]
[INFO] [1694360288.873191701] [listener]: I heard: [Hello World: 10]
[INFO] [1694360289.872660680] [listener]: I heard: [Hello World: 11]
[INFO] [1694360290.873568172] [listener]: I heard: [Hello World: 12]
[INFO] [1694360291.873276468] [listener]: I heard: [Hello World: 13]
[INFO] [1694360293.062166197] [listener]: I heard: [Hello World: 14]
[INFO] [1694360294.877448382] [listener]: I heard: [Hello World: 15]
[INFO] [1694360294.873104147] [listener]: I heard: [Hello World: 16]
[INFO] [1694360295.876203464] [listener]: I heard: [Hello World: 17]
[INFO] [1694360296.875674317] [listener]: I heard: [Hello World: 18]
[INFO] [1694360297.874642872] [listener]: I heard: [Hello World: 19]
[INFO] [1694360298.873221929] [listener]: I heard: [Hello World: 20]
[INFO] [1694360299.873079750] [listener]: I heard: [Hello World: 21]
[INFO] [1694360300.872993546] [listener]: I heard: [Hello World: 22]
```

Figure 11: Output del nodo listener

### 3.3 Micro XRCE-DDS

Per far sì che ROS 2 possa comunicare con PX4 è necessario utilizzare un middleware DDS, in questo caso è stato utilizzato **Micro XRCE-DDS**, che è un protocollo di comunicazione spesso utilizzato in applicazioni di robotica poiché affidabile, leggero, open-source e soprattutto scalabile tra sistemi distribuiti.

#### 3.3.1 Installazione

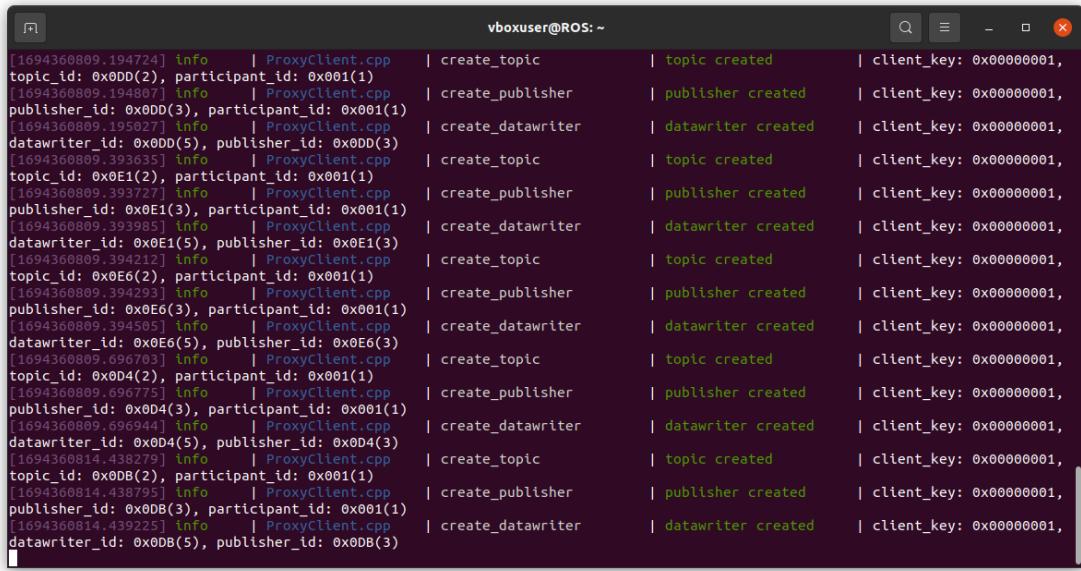
L'installazione riportata di seguito è quella *standalone*, che quindi permette il build dell'agent dal sorgente e la connessione con il simulatore di PX4.

```
git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
cd Micro-XRCE-DDS-Agent
mkdir build
cd build
cmake ..
make
sudo make install
sudo ldconfig /usr/local/lib/
MicroXRCEAgent udp4 -p 8888
```

Figure 12: Installazione uXRCE-DDS

In particolare, il comando *MicroXRCEAgent udp4 -p 8888* permette di far partire l'agent che si connetterà automaticamente al client nel simulatore di PX4.

Una volta fatta partire la simulazione di PX4, i due inizieranno a comunicare e i vari topic pubblicati saranno visibili sul terminale come di seguito in fig. 13.



```
vboxuser@ROS: ~
[1694360809.194724] info | ProxyClient.cpp | create_topic | topic created | client_key: 0x00000001,
[1694360809.194807] info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x00000001,
[1694360809.195027] info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x00000001,
[1694360809.393635] info | ProxyClient.cpp | create_topic | topic created | client_key: 0x00000001,
topic_id: 0x0E1(2), participant_id: 0x001(1)
[1694360809.393727] info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x00000001,
publisher_id: 0x0DD(3), participant_id: 0x001(1)
[1694360809.393985] info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x00000001,
datawriter_id: 0x0E1(3), publisher_id: 0x001(3)
[1694360809.394212] info | ProxyClient.cpp | create_topic | topic created | client_key: 0x00000001,
topic_id: 0x0E6(2), participant_id: 0x001(1)
[1694360809.394293] info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x00000001,
publisher_id: 0x0E6(3), participant_id: 0x001(1)
[1694360809.394505] info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x00000001,
datawriter_id: 0x0E6(5), publisher_id: 0x0E6(3)
[1694360809.696703] info | ProxyClient.cpp | create_topic | topic created | client_key: 0x00000001,
topic_id: 0x0D4(2), participant_id: 0x001(1)
[1694360809.696775] info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x00000001,
publisher_id: 0x0D4(3), participant_id: 0x001(1)
[1694360809.696944] info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x00000001,
datawriter_id: 0x0D4(5), publisher_id: 0x0D4(3)
[1694360814.438279] info | ProxyClient.cpp | create_topic | topic created | client_key: 0x00000001,
topic_id: 0x0DB(2), participant_id: 0x001(1)
[1694360814.438795] info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x00000001,
publisher_id: 0x0DB(3), participant_id: 0x001(1)
[1694360814.439225] info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x00000001,
```

Figure 13: uXRCE-DDS in funzione

### 3.4 QGroundControl

L'applicazione **QGroundControl** è stata utilizzata per pianificare la missione del drone tramite la sua interfaccia grafica e per visualizzare i progressi dello stesso durante la simulazione

### 3.5 Simulazione della missione

Per simulare la missione è stato innanzitutto necessario impostare le coordinate di HOME del drone, per far sì che esso potesse muoversi nella regione di spazio desiderata, per la demo è stato deciso di simulare una missione che partisse dal Dipartimento di Ingegneria dell'Università di Messina e andasse a monitorare lo spazio antistante esso.

Le coordinate della posizione di HOME del drone devono essere impostate prima di far partire la simulazione di PX4, questo viene fatto tramite i comandi di seguito riportati in fig. 14.

```
cd PX4-Autopilot  
export PX4_HOME_LAT=38.25905277824533  
export PX4_HOME_LON=15.595626222658453  
export PX4_HOME_ALT=0
```

Figure 14: Impostazione coordinate HOME position

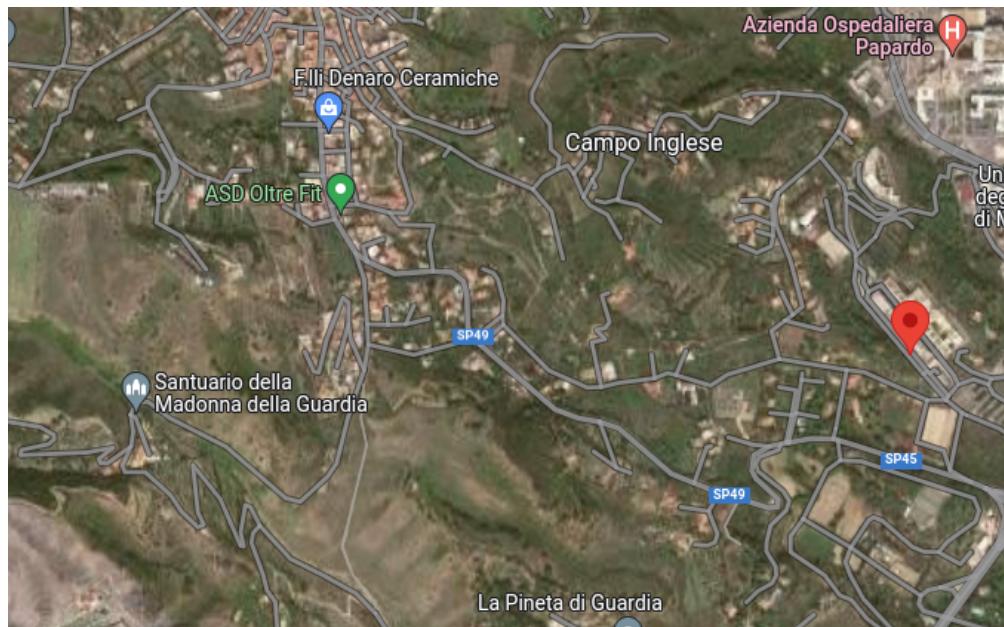


Figure 15: HOME position

Successivamente viene eseguita la simulazione di PX4, ma questa volta, per evitare un sovraccarico computazionale sull'hardware a disposizione e quindi ottenere dei tempi di simulazione ragionevoli, la simulazione viene lanciata senza UI di Gazebo, questo attraverso l'opzione *HEADLESS=1*.

```
HEADLESS=1 make px4_sitl gazebo-classic
```

Figure 16: Simulazione PX4 senza UI

Dall'interfaccia di QGroundControl è possibile pianificare la missione del drone, aggiungendo anche eventuali **Region of Interest** (ROI) in modo tale che il drone si giri per inquadrare per qualche secondo simulando quindi il tempo impiegato a classificare le immagini acquisite alla ricerca di eventuali malattie nelle piante inquadrate.

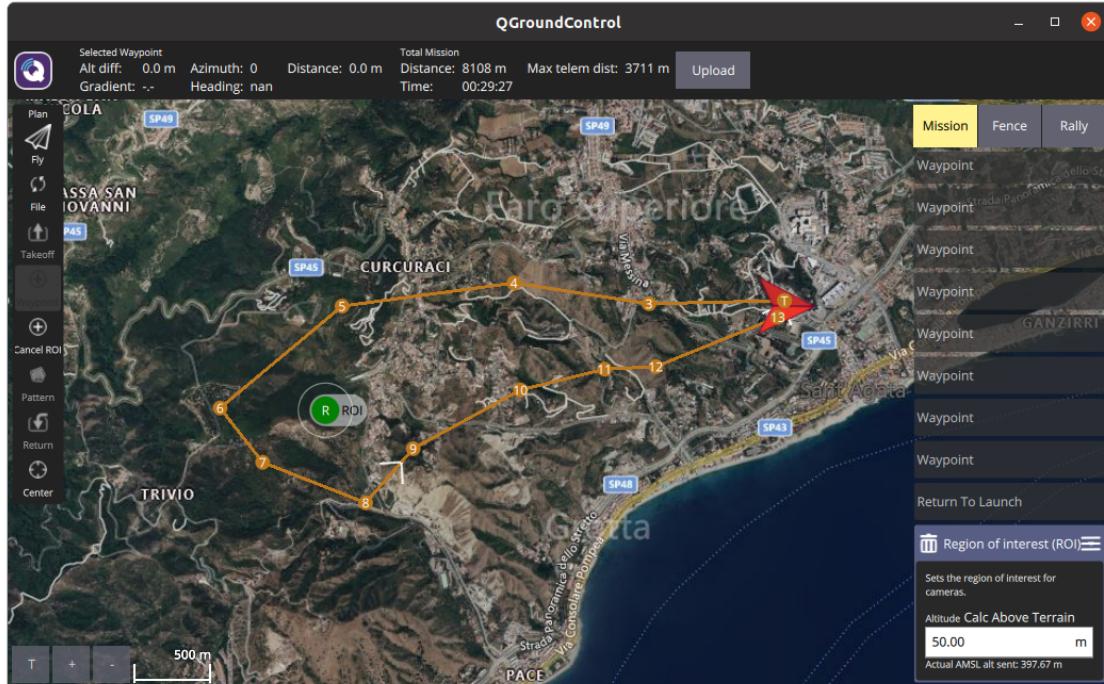


Figure 17: Planning della missione

### 3.6 Acquisizione dei dati tramite ROS 2

Per simulare una sorta di Digital Twin è necessario acquisire i dati del drone ottenuti tramite la simulazione, per questo motivo viene utilizzato ROS 2. Per evitare di sovraccaricare l'hardware a disposizione, i dati del drone verranno innanzitutto acquisiti real-time tramite ROS 2, successivamente verranno visualizzati su terminale e contestualmente salvati dentro un file *.csv*, i dati contenuti all'interno di esso verranno successivamente utilizzati per essere ulteriormente processati tramite MATLAB, la cui implementazione verrà descritta nel prossimo capitolo. Tale procedimento può anche essere eseguito real-time, inviando i dati da ROS 2 a MATLAB direttamente, senza doversi appoggiare ad un file *.csv* eseguendo le simulazioni in due istanti di tempo diversi, ma per vincoli di tipo hardware questo non è possibile.

Per l'approccio proposto sono quindi stati selezionati tra i vari topic disponibili e pubblicati dalla simulazione di PX4 SensorCombined e SensorGPS, che permettono quindi di ottenere i dati di tutti i sensori del drone, salvati nel SI e la coordinate GPS del drone stesso, quest'ultime verranno utilizzate su MATLAB per ricostruire il percorso del drone, realizzandone il **Digital Twin**.

Essendo la parte di pubblicazione già prevista all'interno di PX4, in questo ambito sono stati realizzati due nodi di tipo subscriber, uno per topic, tale implementazione può essere successivamente ampliata nel caso fossero necessari ulteriori dati del drone utilizzando gli altri topic presenti nel reference di uORB. Per realizzare i nodi subscriber è stato necessario realizzare un pacchetto di messaggi personalizzati relativi al tipo di messaggio che PX4 invia e un pacchetto di comunicazione con all'interno i nodi subscriber scritti in C++, entrambi sono contenuti all'interno della cartella *src* contenuta nella repo fornita con la presente relazione.

### 3.6.1 Build dei pacchetti e Utilizzo

Una volta scaricata la cartella *src* contenuta nella repo è possibile buildare attraverso il comando *colcon build* i pacchetti contenuti all'interno di *src*, nel caso *colcon* non sia installato è possibile installarlo con il comando che lo precede in fig. 18.

```
sudo apt install python3-colcon-common-extensions  
colcon build
```

Figure 18: Build dei pacchetti

Una volta buildati i pacchetti è possibile fare il source del setup dell'ambiente e quindi lanciare i nodi attraverso i propri *launch files* come di seguito riportato:

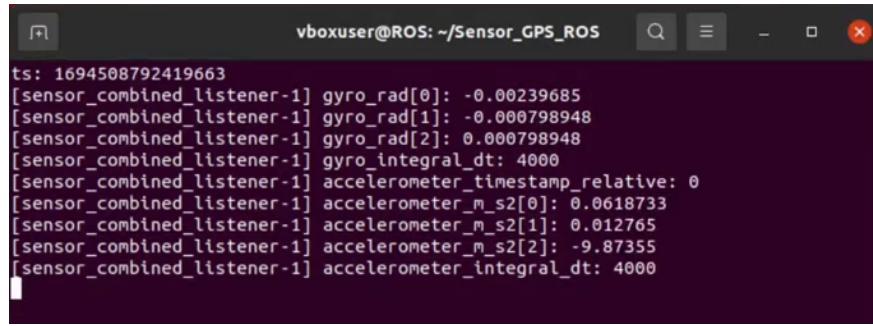
```
source install/local_setup.bash  
ros2 launch px4_ros sensor_combined_listener.launch.py
```

Figure 19: Launch file per i sensori

```
source install/local_setup.bash  
ros2 launch px4_ros gps_data_listener.launch.py
```

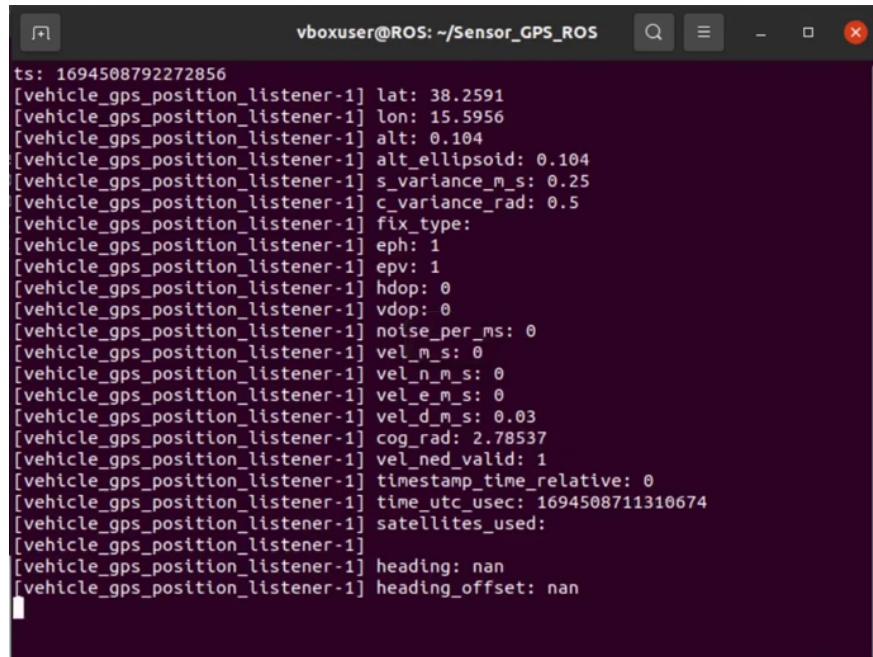
Figure 20: Launch file per il GPS

Gli output dei dati acquisiti tramite ROS potranno essere visualizzati anche su terminale come in fig. 21 e fig. 22.



```
vboxuser@ROS: ~/Sensor_GPS_ROS
ts: 1694508792419663
[sensor_combined_listener-1] gyro_rad[0]: -0.00239685
[sensor_combined_listener-1] gyro_rad[1]: -0.000798948
[sensor_combined_listener-1] gyro_rad[2]: 0.000798948
[sensor_combined_listener-1] gyro_integral_dt: 4000
[sensor_combined_listener-1] accelerometer_timestamp_relative: 0
[sensor_combined_listener-1] accelerometer_m_s2[0]: 0.0618733
[sensor_combined_listener-1] accelerometer_m_s2[1]: 0.012765
[sensor_combined_listener-1] accelerometer_m_s2[2]: -9.87355
[sensor_combined_listener-1] accelerometer_integral_dt: 4000
```

Figure 21: Output di dati per i sensori



```
vboxuser@ROS: ~/Sensor_GPS_ROS
ts: 1694508792272856
[vehicle_gps_position_listener-1] lat: 38.2591
[vehicle_gps_position_listener-1] lon: 15.5956
[vehicle_gps_position_listener-1] alt: 0.104
[vehicle_gps_position_listener-1] alt_ellipsoid: 0.104
[vehicle_gps_position_listener-1] s_variance_m_s: 0.25
[vehicle_gps_position_listener-1] c_variance_rad: 0.5
[vehicle_gps_position_listener-1] fix_type:
[vehicle_gps_position_listener-1] eph: 1
[vehicle_gps_position_listener-1] epv: 1
[vehicle_gps_position_listener-1] hdop: 0
[vehicle_gps_position_listener-1] vdop: 0
[vehicle_gps_position_listener-1] noise_per_ms: 0
[vehicle_gps_position_listener-1] vel_m_s: 0
[vehicle_gps_position_listener-1] vel_n_m_s: 0
[vehicle_gps_position_listener-1] vel_e_m_s: 0
[vehicle_gps_position_listener-1] vel_d_m_s: 0.03
[vehicle_gps_position_listener-1] cog_rad: 2.78537
[vehicle_gps_position_listener-1] vel_ned_valid: 1
[vehicle_gps_position_listener-1] timestamp_time_relative: 0
[vehicle_gps_position_listener-1] time_utc_usec: 1694508711310674
[vehicle_gps_position_listener-1] satellites_used:
[vehicle_gps_position_listener-1]
[vehicle_gps_position_listener-1] heading: nan
[vehicle_gps_position_listener-1] heading_offset: nan
```

Figure 22: Output di dati per il GPS

## 4 Simulazione del Digital Twin su MATLAB

Come descritto nel capitolo precedente, l'idea di base del progetto era quella di realizzare un Digital Twin che potesse andare in real-time con i dati acquisiti tramite ROS 2 dalla simulazione di PX4, questo non è stato possibile a causa delle limitazioni di tipo hardware riscontrate durante le simulazioni (30 punti simulati all'ora, su un totale di 3000 acquisiti durante la simulazione iniziale). È stato quindi scelto un approccio asincrono alle simulazioni stesse, salvando prima i dati generati dalla simulazione del drone ed acquisiti con ROS 2 su dei file .csv e solo successivamente tali dati sono stati utilizzati per effettuare una seconda simulazione, ricostruendo il percorso fatto dal drone su MATLAB, realizzando una sorta di Digital Twin che segue la simulazione principale.

### 4.1 Subsampling dei dati

Attraverso lo script Python presente nella cartella *utils* della repo è possibile sotto campionare i dati presenti nel csv in modo tale da ottenere un numero di punti da simulare che permettano di ottenere un tempo di simulazione ragionevole con l'hardware a disposizione, in questo caso è stato scelto di campionare un sample ogni 50, riducendo il numero totale di sample nel .csv da 3000 a 70.

### 4.2 Livescript Matlab

Il livescript MATLAB realizzato per ottenere la simulazione del Digital Twin è anch'esso presente nella repo dentro la cartella *matlab\_sim*, esso dopo aver acquisito i dati relativi alle coordinate del drone dal .csv li elabora e realizza una mappa in 2D e 3D. Tale mappa verrà successivamente utilizzata per visualizzare il movimento del drone sia su una cartina topografica sia inquadrando quello che verrebbe registrato da un'eventuale camera presente sul drone stesso.

#### 4.2.1 Elaborazione dei dati

I dati relativi alle coordinate del drone in ogni istante vengono elaborati da MATLAB ottenendo la distanza cumulativa coperta durante la missione del drone (fig.23), essa è comprensiva non solo degli spostamenti della posizione, ma anche cambiamenti di altezza del drone stesso, questo viene fatto calcolando l'offset punto a punto per ogni valore acquisito da ROS 2.

```
distanceIncrementIn3D = hypot(hypot(dx, dy), dz);
cumulativeDistanceIn3D = cumsum(distanceIncrementIn3D);
totalDistanceIn3D = sum(distanceIncrementIn3D);
fprintf("During the entire mission the drone path is long %.2f
meters.\n",totalDistanceIn3D)
```

During the entire mission the drone path is long 11320.19 meters.

Figure 23: Elaborazione dei dati del drone

#### 4.2.2 Simulazione del Digital Twin

Come descritto in precedenza all'inizio del capitolo, la simulazione del Digital Twin comprende la realizzazione di un render 2D e 3D delle zone visitate dal drone, per questo motivo la simulazione è pesante a livello computazionale. All'interno del livescript viene innanzitutto visualizzata la posizione di partenza del drone, che come detto in precedenza è il Dipartimento di Ingegneria dell'Università di Messina.



Figure 24: Posizione di partenza del drone

Successivamente tutti i dati presenti all'interno del csv vengono elaborati per realizzare una visione del percorso eseguito dal drone durante la sua missione su una carta topografica, evidenziando punto di partenza e di arrivo, che in questo caso corrispondono.

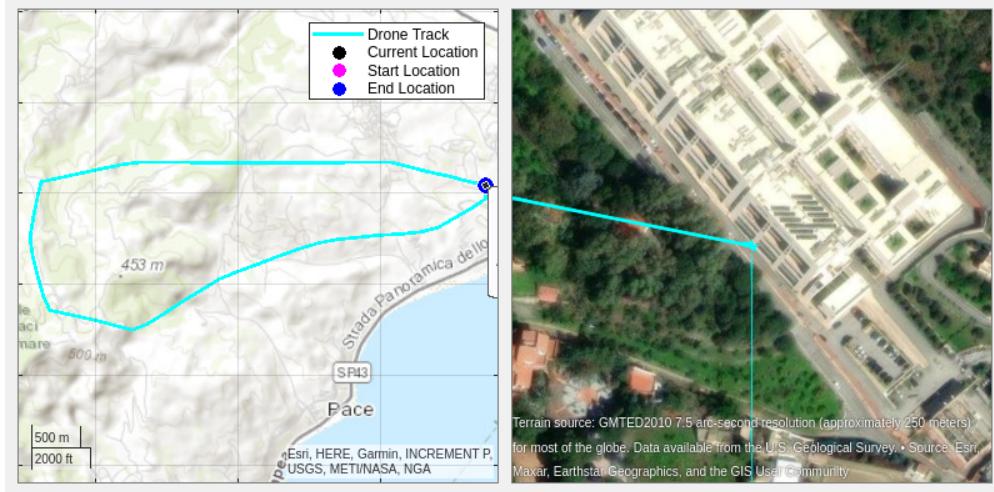


Figure 25: Layout della missione del drone

Come ultimo step all'interno del livescript viene eseguita una simulazione della missione effettuata dal drone, visualizzando da una parte la carta topografica che si aggiorna runtime, segnalando le coordinate e la distanza dal punto di partenza; mentre dall'altra parte viene visualizzato un render di quello che vedrebbe un'eventuale cam posta sul drone.

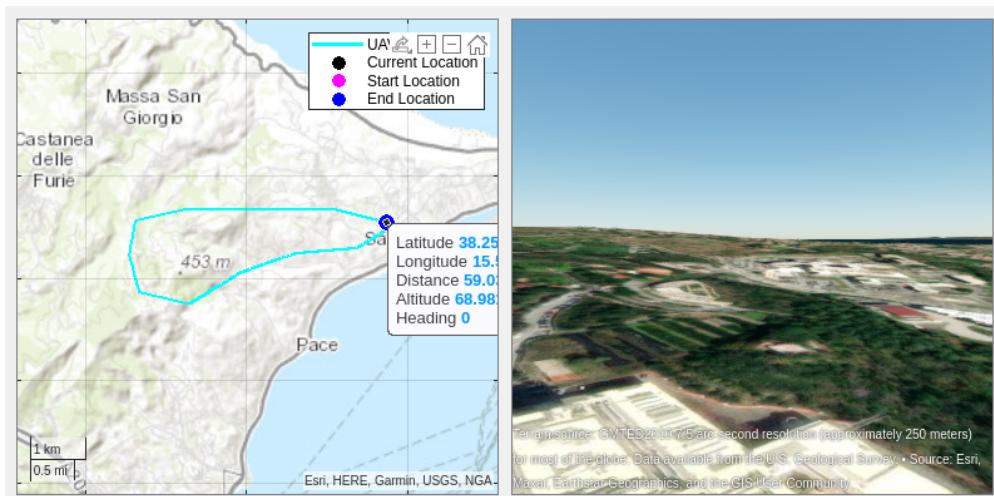


Figure 26: Simulazione del Digital Twin

## 5 Conclusioni

In conclusione, il progetto ha permesso di realizzare la simulazione di una missione di un drone, permettendo di impostare HOME position, personalizzare la missione stessa impostando vari waypoints e Region of Interest per permettere al drone simulato di concentrarsi su regioni di spazio che potessero essere interessanti per il monitoraggio di piante. È stato inoltre possibile acquisire i dati relativi a tutti i sensori del drone e alle coordinate indicanti la sua posizione per ogni istante di tempo. Tali dati sono stati elaborati successivamente su MATLAB per ottenere un monitoraggio della distanza percorsa dal drone e una simulazione del Digital Twin del drone stesso, visualizzando lo spostamento su una cartina topografica e un render delle possibili immagini acquisite dall'eventuale camera del drone.

Non è stato possibile realizzare un approccio real-time di comunicazione tra la simulazione PX4 e quella del Digital Twin su MATLAB a causa delle richieste di risorse delle simulazioni stesse e all'hardware sul quale le simulazioni sono state fatte girare. Nonostante ciò è comunque possibile prevedere di implementare la comunicazione real-time su un eventuale hardware consono alle risorse richieste da tali strumenti.