



UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

**Sviluppo su piattaforma autopilota di una missione di ricerca
con sciami di droni**

Relatore:

Prof. Mario G.C.A Cimino

Prof. Pierfrancesco Foglia

Dott. Salvatore Arancio Febbo

Candidato:

Giorgio Charles Sorrentini

Introduzione

L'uso dei **droni** è sempre più diffuso in svariati settori grazie alla loro capacità di operare in ambienti inaccessibili, raccogliendo dati senza mettere a rischio vite umane. Un'applicazione particolarmente rilevante è la ricerca e il **soccorso di dispersi in scenari di emergenza**.

Questo elaborato si concentra sullo sviluppo e il miglioramento di **algoritmi per il coordinamento di sciami** di droni, classificati come **UAV** (Unmanned Aerial Vehicle), impiegati nelle operazioni di ricerca post-catastrofe. In particolare, il lavoro si inserisce nel progetto **4DDS – 4D Drone Swarms**, un'iniziativa che sviluppa sciami di droni autonomi coordinati nello spazio e nel tempo, con applicazioni in diversi ambienti operativi, tra cui aereo, terrestre, superficiale e subacqueo.

Lo studio presentato prende in esame il sisma del 24 agosto 2016 ad Amatrice, utilizzando una ricreazione precedentemente realizzata in un ambiente di simulazione per testare algoritmi di coordinamento di sciami aerei nella rilevazione di dispersi. L'attuale implementazione si basa sul modello **Boids**, che permette un volo coeso evitando collisioni e dispersione dello sciame. Tuttavia, questo modello *non garantisce una copertura ottimale dell'area di ricerca, in quanto non prevede una distribuzione strategica dei droni*.

L'**obiettivo** di questo lavoro è quindi migliorare l'algoritmo esistente, sviluppando una logica che permetta agli sciami di droni di mantenere una formazione specifica e **coprire un'area** definita per tutta la durata della missione. Questo approccio mira a rendere più efficiente il processo di scansione e individuazione dei dispersi, riducendo i tempi di intervento e migliorando l'efficacia delle operazioni di soccorso.

1 Ambiente di Sviluppo

In questo capitolo vengono descritte le varie componenti software che hanno permesso lo svolgimento del progetto.

1.1 Elenco Software Componenti

1. Gazebo Classic
2. PX4 Autopilot
3. ROS2
4. XRCE-DDS (MicroXRCEAgent)

1.1.1 Gazebo Classic

Gazebo Classic è il software di simulazione 3D utilizzato per riprodurre l'ambiente operativo e i modelli di droni scelti. Genera dati relativi alla **posizione** (coordinate 3D), **velocità** e **informazioni sensoriali** per ogni drone simulato. Gazebo-ROS-Pkgs è un'estensione che include plugin aggiuntivi per simulare **sensori avanzati**, migliorando la fedeltà della simulazione.

Modalità di simulazione:

- **SITL (Software in the Loop):** Tutta la simulazione avviene su PC, senza necessità di hardware reale.
- **HTIL (Hardware in the Loop):** Permette di testare l'hardware reale del drone in un ambiente virtuale.

In questa tesi è stato scelto l'approccio **SITL** per la disponibilità, flessibilità e sicurezza nei test.

Percorsi delle cartelle di interesse:

- **Modelli droni:** /home/fourdds/PX4-Autopilot/Tools/simulation/gazebo-classic/sitl-gazebo_classic/models
- **Mondi simulati:** /home/fourdds/PX4-Autopilot/Tools/simulation/gazebo-classic/sitl-gazebo_classic/worlds

All'interno della cartella `../worlds` è la directory nella quale è salvata la ricreazione dello scenario di Amatrice usata nelle simulazioni.

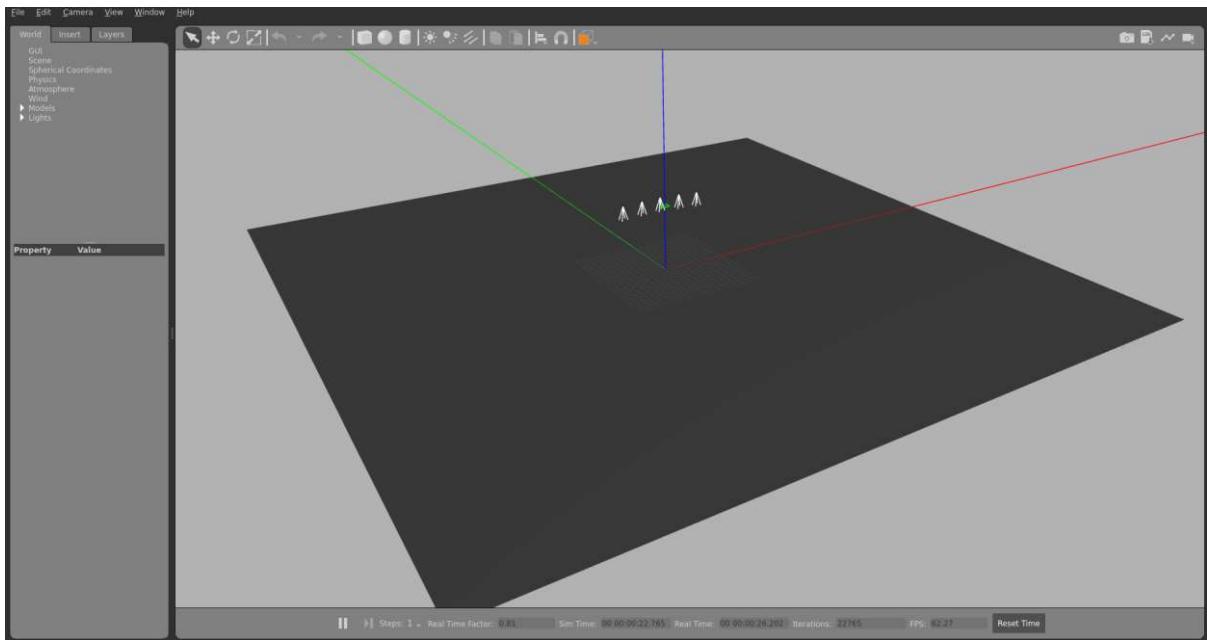


Figura 1.1.1.1: Simulazione all'interno di Gazebo di uno sciame di droni Iris in volo

1.1.2 PX4 Autopilot

PX4 Autopilot è una piattaforma software open-source progettata per il controllo di droni e veicoli autonomi senza pilota. È il software di **controllo on-board** del singolo drone, responsabile della stabilità, navigazione e risposta ai comandi.

- Si interfaccia con *Gazebo*, ricevendo dati simulati e calcolando la prossima azione del drone.
- L'architettura è pensata per offrire un elevato grado di flessibilità per gli sviluppatori per adattare il codice a seconda delle esigenze. Difatti, può funzionare anche senza ROS2 per missioni autonome di base, ma la sua integrazione con ROS2 permette una gestione avanzata della flotta.

In questa tesi questa componente software è stata usata per il controllo di quadricotteri.

Cartella di riferimento: /home/fourdds/PX4-Autopilot

1.1.3 ROS2 (Robot Operating System 2)

ROS2 è un insieme di librerie e software open-source utilizzati per sviluppare applicazioni avanzate per robot e droni. ROS2 si basa su un'architettura distribuita e utilizza **DDS** (Data Distribution Service) come middleware, consentendo una comunicazione più robusta e flessibile tra i vari componenti del sistema. Supporta svariati linguaggi di programmazione, tra cui C++ e Python, offre strumenti per simulazione, gestione dei pacchetti e controllo dei robot ed è compatibile con diversi sistemi operativi.

Si tratta fondamentalmente del **software di controllo off-board** utilizzato per la gestione multi-drone. Le sue principali funzioni includono:

- **Pianificazione delle missioni:** Definizione degli obiettivi di alto livello.
- **Coordinamento della flotta:** Comunicazione tra droni per strategie collettive.
- **Avoidance:** Rilevamento ed evitamento degli ostacoli.

- **Interfaccia con la stazione base:** Monitoraggio e controllo remoto.

ROS2 utilizza **DDS (Data Distribution Service)** per permettere una comunicazione asincrona tra i vari componenti del sistema tramite meccanismi di *publish/subscribe*.

Per questo progetto i pacchetti ROS2 per il coordinamento dei droni saranno implementati in C++.

Cartella di riferimento: /home/fourdds/ros2_ws/src

1.1.4 XRCE-DDS MicroXRCEAgent

XRCE-DDS è il middleware di comunicazione tra PX4 e ROS2, traducendo i dati tra il protocollo **MAVLink** di PX4 e DDS di ROS2. È un'implementazione di **DDS-XRCE** (Data Distribution Service for eXtremely Resource Constrained Environments), progettato per dispositivi a risorse limitate, come droni embedded.

Il funzionamento principale di questo software è lo **smistamento dei messaggi** tra i *publisher* e *subscriber* per gestire la comunicazione tra PX4 e ROS2.

Flusso dati:

- **PX4 → ROS2:** Stato del drone (es: posizione, orientamento, velocità attuale).
- **ROS2 → PX4:** Logica di coordinamento dello sciame e Comandi di navigazione (es: parametri di navigazione).

Vantaggi dell'utilizzo di XRCE-DDS:

- **Comunicazione asincrona** tra i droni, garantendo sincronizzazione tra più unità.
- **Riduzione del traffico di rete**, evitando colli di bottiglia nelle comunicazioni.
- **Scalabilità**, permettendo la gestione di più droni contemporaneamente.

2 Installazione e Studi Preliminari

In questo capitolo saranno illustrate le varie fasi per dare inizio al progetto di tesi.

2.1 Installazione Ambiente di Sviluppo

L'ambiente di sviluppo è configurato per un sistema operativo Linux Ubuntu 22.04 ed è stato distribuito attraverso una semplice cartella compressa **4DDS.zip** contenente il disco virtuale **4DDS.tar**.

Per poterlo usare su un sistema operativo Windows 10/11, come nel caso di questo progetto, è prima necessario installare **WSL2** (Windows Subsystem for Linux 2). Una volta completata l'installazione, si importa il disco virtuale. Per questi passaggi, i comandi sono i seguenti:

```
wsl --install  
wsl -import 4DDS . 4DDS.tar
```

Al termine dell'operazione sarà presente nella medesima cartella in cui è situato 4DDS.tar un disco virtuale chiamato **ext4.vhdx**.

Per accedere all'ambiente di sviluppo e all'utente per avere tutti privilegi necessari per l'accesso alle varie directory, eseguire i seguenti comandi da terminale:

```
wsl -d 4DDS  
su fourdds
```

Per lo sviluppo e modifica del codice, si parte dalla cartella di lavoro `/home/fourdds` usando un IDE come Visual Studio Code per poter accedere alle cartelle e file interessati.

2.2 Avvio Simulazione

Per poter avviare una Simulazione richiede l'esecuzione di tre comandi in tre finestre di comando distinte. I comandi sono i seguenti nell'ordine scritti:

```
MicroXRCEAgent udp4 -p 8888 -v
```

Questo lancia il broker XRCE-DDS permettendo lo scambio di messaggi tra PX4, Gazebo e i nodi ROS2.

Nella finestra successiva, bisogna spostarsi nella cartella `/home/fourdds/ws` per configurare l'ambiente di lavoro caricando variabili d'ambiente e percorsi necessari per eseguire correttamente il software.

```
cd ws  
source setup.bash
```

Poi tornare indietro per avviare l'ambiente di simulazione:

```

cd..
./PX4-Autopilot/Tools/simulation/gazebo-classic
sitol_multiple_run.sh -n <num_droni>

```

In quest'ultimo, `num_droni` indica il numero di droni nella simulazione e deve essere specificato.

Per fermare il simulatore Gazebo, è necessario premere CTRL + C nella finestra dove è stato avviato l'ultimo processo.

Infine, nella terza e ultima tab, si andrà ad attivare la logica dei droni, che permetterà loro di alzarsi in volo e coordinare lo sciame. Prima bisogna andare nella cartella `/home/fourdds/ros2_ws` per compilare l'ambiente ROS2 e mettere in esecuzione il programma per attivare la logica dei droni, che permetterà di coordinare lo sciame di droni:

```

cd ros2_ws
source setup.bash
bash run.sh -n <num_droni>

```

2.3 Tipologia di drone utilizzato

Il drone utilizzato nelle simulazioni è il modello Iris, un quadricottero molto comune per la sua stabilità di volo e compatibilità con numerosi strumenti e sensori per rilevamento dell'ambiente circostante.

Il file di descrizione è situato nella cartella:

`~/PX4-Autopilot/Tools/simulation/gazeboclassic/sitol_gazebo_classic/models/iris/iris.sdf.jinja`

Inoltre, il drone è stato equipaggiato con due sensori specifici:

- **LiDAR**: Uno scanner laser per la mappatura e rilevamento ostacoli.
- **Fotocamera FPV** (First-Person View): Utilizzata per la visione in prima persona e per l'identificazione dei target.

Questi sensori permettono di simulare il rilevamento dell'ambiente circostante in cui il drone deve navigare autonomamente, per evitare ostacoli e individuare i target.



Figura 2.3.1: Drone Iris in ambiente Gazebo con i due sensori lidar

3 Scenario di Simulazione

In questo capitolo viene approfondito lo scenario di riferimento utilizzato per le simulazioni effettuate.

3.1 Sisma di Amatrice 2016

Lo scenario preso in considerazione è stato quello di Amatrice quando è stata colpita dal violento terremoto che provocò danni notevoli alla città e alla popolazione locale.

Questo scenario è stato preso dalla cartella `/home/fourdds/PX4-Autopilot/Tools/simulation/gazebo-classic/sitl-gazebo_classic/worlds` dove sono presenti tutti i mondi simulati disponibili.

Questa realizzazione di Amatrice rappresenta principalmente il centro storico della città, la parte più danneggiata dal sisma, come mostrato nelle immagini sottostanti.



Figura 3.1.1: Ripresa di Amatrice prima e dopo il terremoto del 2016 [1] [2]

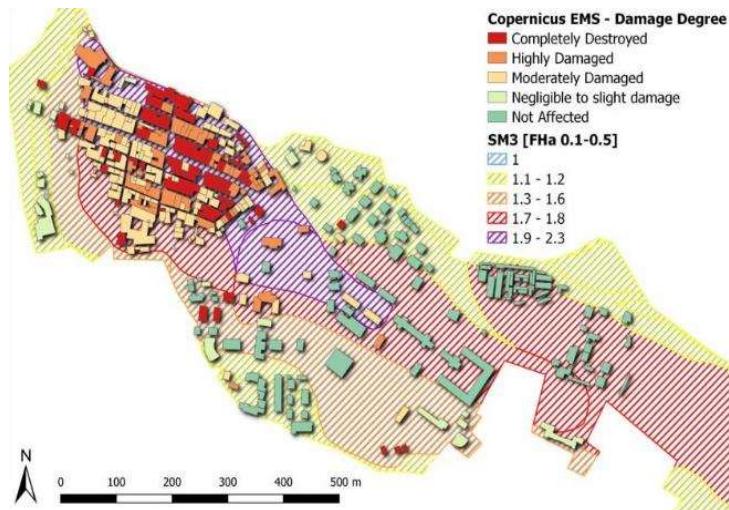


Figura 3.1.2: Grado di danneggiamento degli edifici post-terremoto [3]

Questo rappresenta un ottimo esempio come caso d'uso dei droni perché i danni che la città ha subito risultò difficile accedere alle aree colpite, ostacolando i soccorsi e rallentando la **ricerca dei dispersi**. Droni equipaggiati con sensori avanzati, come lidar, fotocamere e rilevatori di segnali GSM, potrebbe essere

fondamentale per individuare rapidamente le persone intrappolate sotto le macerie. Un caso d'uso più che rilevante per mettere alla prova queste tecnologie.

Lo scenario in questione si può selezionare tramite il comando di avvio della simulazione:

```
./PX4-Autopilot/Tools/simulation/gazebo-classic sitl_multiple_run.sh -n <num_droni> -w <nome_scenario>
```

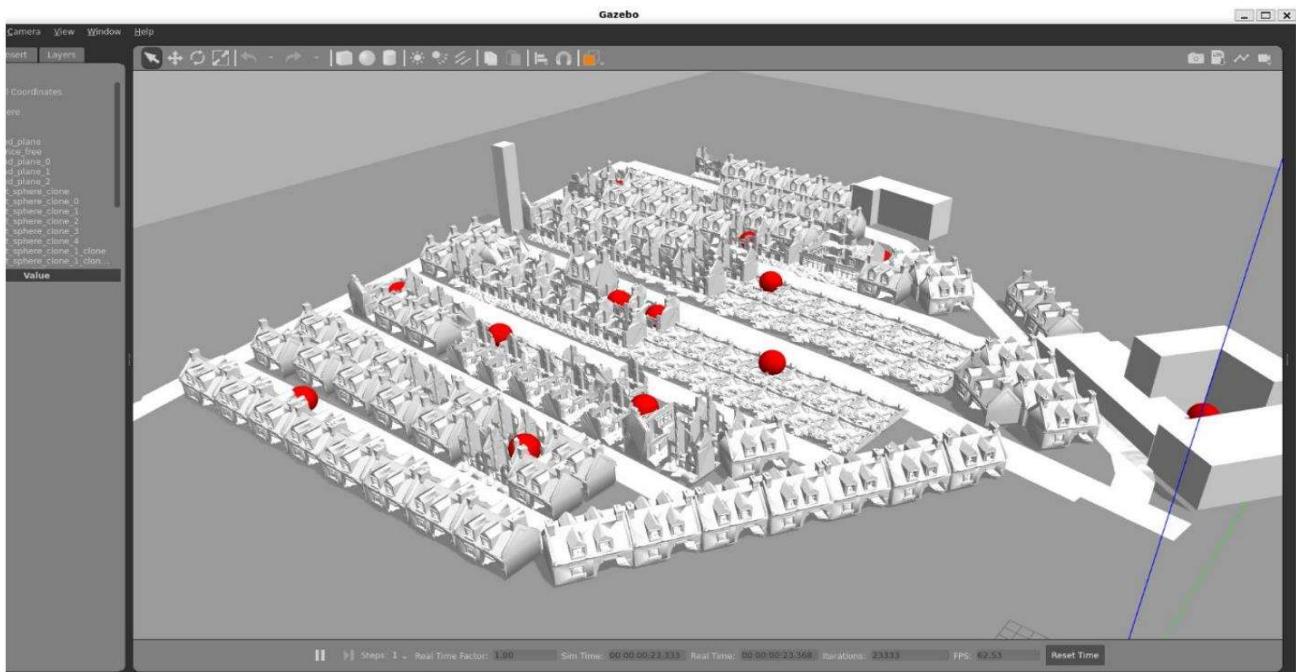


Figura 3.2.5: Scenario di Amatrice importato su Gazebo con target sferici rossi

4 Sviluppo del codice sorgente per la simulazione

Come anticipato all'introduzione, lo scopo del lavoro presentato è di modificare il codice sorgente per permettere allo sciame di droni di potersi disporre una formazione tale da coprire una determinata area superficiale per migliorare la capacità della flotta di rilevare gli obiettivi ricercati nella zona di interesse. Per questo, le parti di codice interessate per le modifiche da apportare sono quelle attinenti alla logica di flocking e alla comunicazione dei pacchetti specifici per il coordinamento dello sciame.

Il linguaggio di programmazione scelto è stato C++.

4.1 Moduli interessati e la Logica attualmente implementata

I moduli responsabili della logica di coordinamento dello sciame di droni e i valori dei parametri coinvolti sono:

- **Drone.cpp**, gestisce il movimento del singolo drone e la comunicazione con la *Base Station*, trasmettendo i dati di navigazione e ricevendo i parametri di coordinamento per il singolo drone all'interno della flotta.
- **BaseStation.cpp**, coordina il movimento dell'intera flotta di droni e comunica con i singoli dispositivi per determinare e trasmettere i comandi necessari per mantenere lo sciame coeso ed evitare collisioni tra loro e/o con altri ostacoli.
- **Parameters.hpp** e **Options.yaml**, sono i moduli nella quale sono definiti tutti i parametri utilizzati nei due moduli menzionati. Non solo quelli relativi alla logica di coordinamento ma anche altri essenziali (es: velocità massima, coordinate di spawning, numero veicoli, perimetro della mappa ecc..).

Per quest'ultimi della lista, **Parameters.hpp** permette di fare riferimento e usare all'interno del codice sorgente i parametri definiti all'interno di **options.yaml**. In questo modo è possibile modificare i valori dei vari parametri direttamente da quest'ultimo file senza dover ricompilare il codice sorgente ogni volta.

Per i primi due moduli menzionati, sono quelli in cui è stata implementata la logica di flocking seguendo il **modello di Boids**. Questo modello è stato coniato da Craig Reynolds, uno sviluppatore software esperto in computer grafica e videogiochi. Sviluppò questo modello di coordinamento di un gruppo di entità mobili durante il periodo lavorativo a Sony Computer Entertainment come ricercatore per simulare il movimento di uno stormo di piccioni.

Il *flocking* è un comportamento collettivo osservato in natura in diverse specie di animali, che siano terrestri o marini. Il **modello di Boids** per simularlo si basa su tre regole fondamentali:

1. **Separazione** – Ogni entità evita di avvicinarsi troppo agli altri per prevenire collisioni.
2. **Allineamento** – Ogni entità adatta la propria direzione e velocità a quelle dei vicini per mantenere la coesione del gruppo.
3. **Coesione** – Ogni entità si muove verso il centro del gruppo per evitare dispersioni.

Queste semplici regole generano comportamenti complessi e realistici, utilizzati in simulazioni di intelligenza collettiva e grafica computerizzata. Nel nostro progetto, il modello di Boids è implementato per coordinare la flotta di droni, garantendo movimenti autonomi e organizzati.

4.2 Funzionalità di interesse di ciascun modulo

Nei moduli `Drone.cpp` e `BaseStation.cpp` sono presenti diverse funzioni che gestiscono questa logica di flocking e la comunicazione tra i droni e la stazione base. Qua verranno elencate le funzioni più rilevanti di ciascun modulo relative al lavoro svolto.

4.2.1 Drone.cpp

sendGeoPing e *flockingSubscriptionCallback*

Queste due funzioni sono essenziali per la comunicazione con Base Station. La prima serve per comunicare i dati di navigazione del drone individuale come le coordinate geografiche e la velocità lungo l'asse nord ed est.

```
void fdds::Drone::sendGeoPing()
{
    fdds_messages::msg::GeoPing msg;
    msg.vehicle_id = vehicleId;

    msg.latitude_m = latLonMetres(0);
    msg.longitude_m = latLonMetres(1);

    msg.speed_north_m_s = currentSpeed(0);
    msg.speed_east_m_s = currentSpeed(1);

    msg.spawn_positions_x = spawn_positions(0);
    msg.spawn_positions_y = spawn_positions(1);

    /* Stampa dei parametri inviati...
     */
    geoPingPublisher->publish(msg);
}
```

E la seconda per ricevere i parametri di flocking aggiornati da *Base Station*, ovvero *coesione*, *allineamento* e *separazione* assieme alla *direzione* per giungere a destinazione. Questi sono essenziali per il coordinamento della flotta.

```
void fdds::Drone::flockingSubscriptionCallback(fdds_messages::msg::Flocking::ConstSharedPtr msg)
{
    cohesion = Eigen::Vector2d{msg->cohesion_x, msg->cohesion_y};
    alignment = Eigen::Vector2d{msg->alignment_x, msg->alignment_y};
    separation = Eigen::Vector2d{msg->separation_x, msg->separation_y};
    direction = Eigen::Vector2d{msg->direction_x, msg->direction_y};

    /* Stampa dei parametri ricevuti...
     */
}
```

flockingLogic

Questa funzione calcola la velocità del drone come un vettore bidimensionale basandosi sui parametri di flocking del modello implementato, assieme ad altre forze repulsive per gestire i confini della mappa in cui sono istanziati i droni ed evitare di poterli far uscire dal perimetro virtuale.

```

void fdds::Drone::flockingLogic()
{
    ...

    // Calcolo della velocità
    Eigen::Vector2d velocity = Eigen::Vector2d::Zero();

    // Aggiunta della forza di coesione
    velocity += options.cohesionFactor * cohesion;

    // Aggiunta della forza di allineamento
    velocity += options.alignmentFactor * alignment;

    // Aggiunta della forza di separazione
    velocity += options.separationFactor * separation;
    // Aggiunta della direzione parameters.directionFactor
    velocity += (direction.normalized() * options.directionFactor);

    // Aggiunta della forza di boundary handling
    Eigen::Vector2d boundaryForce = handleBoundaries();
    velocity -= boundaryForce;

    currentSpeed = velocity;

    // Controllo restrizione velocità massima
    double currentNorm = currentSpeed.norm();
    if (currentNorm > options.maxSpeed)
    {
        double scale = options.maxSpeed / currentNorm;
        currentSpeed *= scale;
    }

    // Invio della velocità e geoping
    sendVelocity(currentSpeed, altitudeParameter);
    sendGeoPing();
}

```

Come mostrato, la velocità viene calcolata e aggiornata in funzione dei quattro parametri ricevuti dalla `flockingSubscriptionCallback` e ciascuno di essi viene amplificato da un certo fattore moltiplicativo definito in `options.yaml`.

Alla fine della funzione, viene chiamata la `sendGeoPing` per comunicare con *Base Station* e trasmettere i dati necessari per l'aggiornamento dei parametri.

4.2.2 BaseStation.cpp

advertiseFlocking

Questa funzione trasmette a `Drone.cpp` i parametri di flocking aggiornati al singolo drone con un certo numero identificativo. Quindi *Base Station* non comunica tutti i droni simultaneamente ma a ciascuno individualmente.

```

void fdds::BaseStation::advertiseFlocking(const int vehicle_id,
                                           const Eigen::Vector2d& cohesion,
                                           const Eigen::Vector2d& alignment,
                                           const Eigen::Vector2d& separation,
                                           const Eigen::Vector2d& formation)
{
    fdds_messages::msg::Flocking msg;
    msg.cohesion_x = cohesion(0);
    msg.cohesion_y = cohesion(1);

    msg.alignment_x = alignment(0);
    msg.alignment_y = alignment(1);

    msg.separation_x = separation(0);
    msg.separation_y = separation(1);

    msg.direction_x = direction(0);
    msg.direction_y = direction(1);
    flockingPublishers[vehicle_id - 1]->publish(msg);
}

```

geoPingCallback

E' la funzione centrale che risponde ad ogni messaggio di *geoPing* di ciascun drone in comunicazione con *BaseStation*.

```

void fdds::BaseStation::geoPingCallback(fdds_messages::msg::GeoPing::ConstSharedPtr msg)
{
    drone_position_relative << msg->spawn_positions_x, msg->spawn_positions_y;

    Eigen::Vector2d drone_position{msg->latitude_m, msg->longitude_m};
    Eigen::Vector2d drone_speed{msg->speed_north_m_s, msg->speed_east_m_s};
    swarmPositions[msg->vehicle_id - 1] = drone_position;
    swarmSpeeds[msg->vehicle_id - 1] = drone_speed;

    int num_cohesion = 0;
    Eigen::Vector2d cohesion = Eigen::Vector2d::Zero();

    int num_alignment = 0;
    Eigen::Vector2d alignment = Eigen::Vector2d::Zero();

    int num_separation = 0;
    Eigen::Vector2d separation = Eigen::Vector2d::Zero();
}

```

```

    {
        for (int i = 0; i < swarmSize; i++)
        {
            const auto& drone_i_position = swarmPositions[i];
            const auto& drone_i_speed = swarmSpeeds[i];

            const auto distance = (drone_i_position - drone_position).norm();
            if (distance > 0.0001)
            {
                if (distance < options.separationRadius && i != msg->vehicle_id - 1)
                {
                    separation += (options.separationRadius - distance)*(drone_position - drone_i_position).normalized();
                    num_separation++;
                }
                else if (distance < options.cohesionRadius)
                {
                    cohesion += drone_i_position;
                    num_cohesion++;
                }

                alignment += drone_i_speed;
                num_alignment++;
            }
        }

        if (num_cohesion != 0)
        {
            cohesion /= num_cohesion;
            cohesion -= drone_position;
        }

        if (num_alignment != 0)
            alignment /= num_alignment;

        if (num_separation != 0)
            separation /= num_separation;
    }

    if (msg->vehicle_id == 1)
    {
        zigzag_delta(drone_position); // Calcolo e Aggiornamento della Direzione
    }

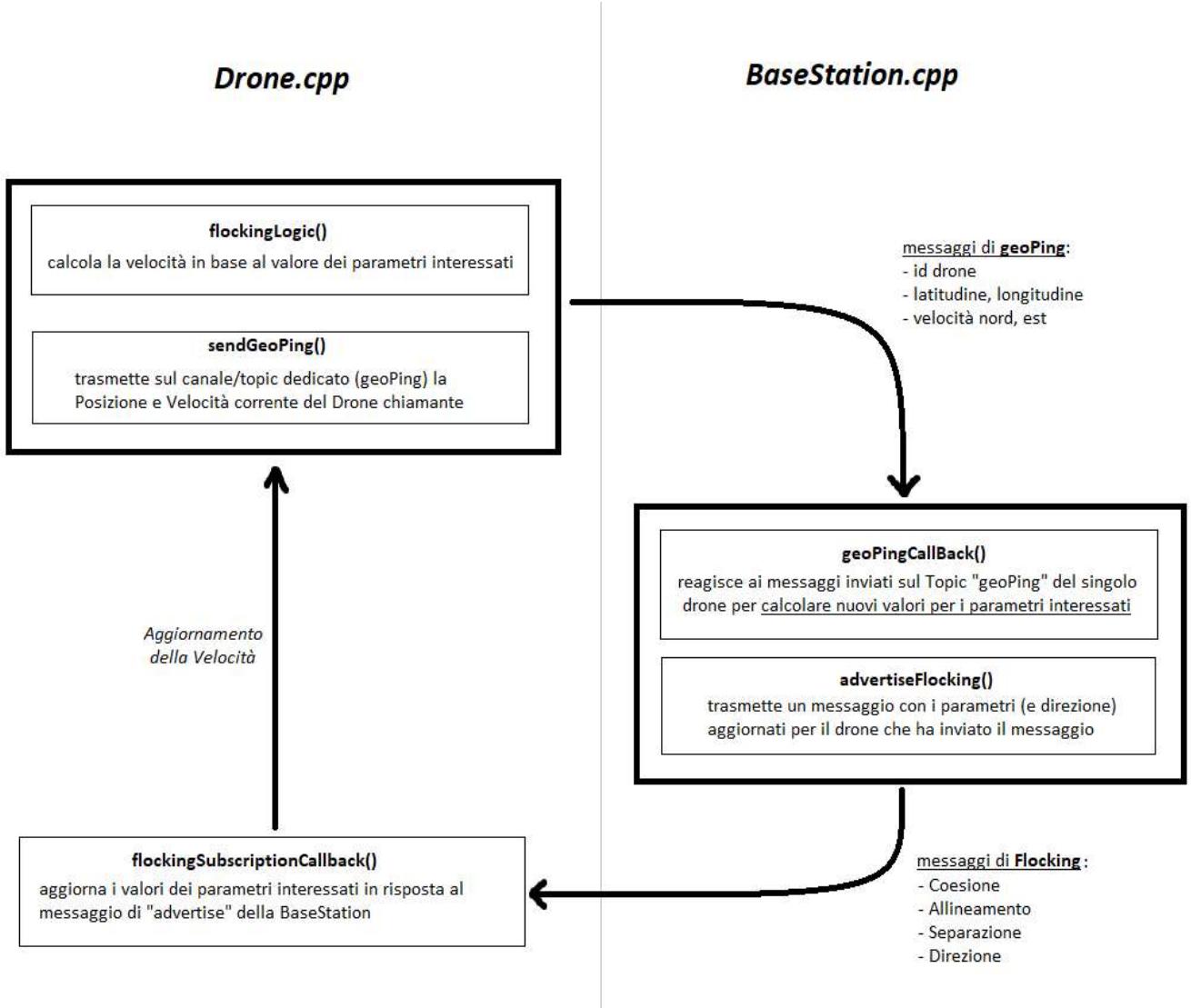
    advertiseFlocking(msg->vehicle_id, cohesion, alignment, separation, formation);
}

```

Per ogni messaggio ricevuto, Base Station ne estrae le coordinate geografiche trasmesse assieme alle componenti vettoriali della velocità. Questi dati sono poi sfruttati per calcolare i tre parametri di flocking per il singolo drone di cui è stato ricevuto il messaggio in funzione di tutto il resto dei droni della flotta cui la stazione base è a conoscenza. Il valore finale è una semplice media aritmetica dei valori ottenuti rispetto a ciascun altro drone.

Alla fine della funzione, viene calcolato e aggiornato anche il parametro della direzione da una funzione predefinita del modulo in questione, `zigzag_delta()`, terminando con la trasmissione di tutti questi valori al drone di cui è stato ricevuto il messaggio di *geoPing*.

Questa è un'illustrazione riassuntiva del flusso di comunicazione tra Drone.cpp e BaseStation.cpp¹:



4.2.3 Parameters.hpp e Options.yaml

Parameters.hpp definisce una struttura dati che contiene come campi tutti i parametri configurabili specificati in options.yaml. Nelle classi Drone e BaseStation, è presente un campo come istanza di questa struttura Parameters, denominata options, che consente di accedere a questi parametri in modo centralizzato nel proprio codice. Questo campo può essere visto utilizzato all'interno dei codici mostrati precedentemente, in particolare flockingLogic() e geoPingCallback().

¹ Di questi due moduli, Drone.cpp e BaseStation.cpp, è presente un ulteriore modulo dedicato alla definizione della classe relativa a ciascuno. Sono state omesse in questa illustrazione del codice per risparmiare dettagli eccessivi in quanto presentano diversi campi e metodi non rilevanti alle funzioni interessate.

A seguire nella presentazione saranno prese in considerazione per mostrare i campi aggiunti necessariamente per poter implementare la nuova logica aggiunta al codice sorgente.

La classe `Parameters` viene definita come tale:

```
namespace fdds
{
    struct Parameters
    {
        int numVehicles;

        float separationFactor;
        float separationRadius;

        float avoidanceRadius;
        float avoidanceFactor;

        float cohesionRadius;
        float cohesionFactor;

        float alignmentRadius;
        float alignmentFactor;
    };

    explicit Parameters(
        const int num_vehicles,
        const float separationFactor,
        const float separationRadius,
        const float avoidance_radius,
        const float avoidance_factor,
        const float cohesion_radius,
        const float cohesion_factor,
        const float alignment_radius,
        const float alignment_factor
    )
        : numVehicles(num_vehicles),
        separationFactor(separationFactor),
        separationRadius(separationRadius),
        avoidanceRadius(avoidance_radius),
        avoidanceFactor(avoidance_factor),
        cohesionRadius(cohesion_radius),
        cohesionFactor(cohesion_factor),
        alignmentRadius(alignment_radius),
        alignmentFactor(alignment_factor)
    {}

    Parameters() = delete;
};
```

```

    inline Parameters loadOptions(bool verbose)
    {
        Yaml::Node config_file;
        Yaml::Parse(config_file, "/home/fourdds/.swarm/options.yaml");

        return Parameters(
            config_file["num_vehicles"].As<int>(),

            config_file["separationFactor"].As<float>(),
            config_file["separationRadius"].As<float>(),

            config_file["avoidanceRadius"].As<float>(),
            config_file["avoidanceFactor"].As<float>(),

            config_file["cohesionRadius"].As<float>(),
            config_file["cohesionFactor"].As<float>(),

            config_file["alignmentRadius"].As<float>(),
            config_file["alignmentFactor"].As<float>
        );
    }
};

```

E in quest'ultima immagine relativa a `Parameters`, mostra il codice interessato per il caricamento dei valori per ciascun campo della classe ottenuti accedendo al file `options.yaml` (immagine sottostante):

```

! options.yaml ×

home > fourdds > .swarm > ! options.yaml >
1   alignmentFactor: 6.0
2   alignmentRadius: 8.0
3
4   cohesionFactor: 0.1
5   cohesionRadius: 8.0
6
7   separationFactor: 1.0
8   separationRadius: 3.0
9
10  avoidanceFactor: 1.5
11  avoidanceRadius: 3.0
12
13  num_vehicles: 3

```

4.3 Logica di Formazione da implementare

Il codice finora illustrato rappresenta la base su cui è stata sviluppata la logica di formazione per consentire ai droni di disporsi in modo tale da poter coprire un'area definita dall'utente.

L'idea alla base della nuova logica ideata è distribuire i droni lungo l'asse perpendicolare alla direzione calcolata nella funzione `geoPingCallback()`, in modo da aumentare l'area di copertura effettiva dello sciame e limitare la sovrapposizione dei campi di rilevamento dei singoli droni. Questo limiterebbe ridondanze nella raccolta dei dati, cercando di garantire che ogni drone esplori una porzione unica dell'area e ottimizzando l'efficienza della missione.

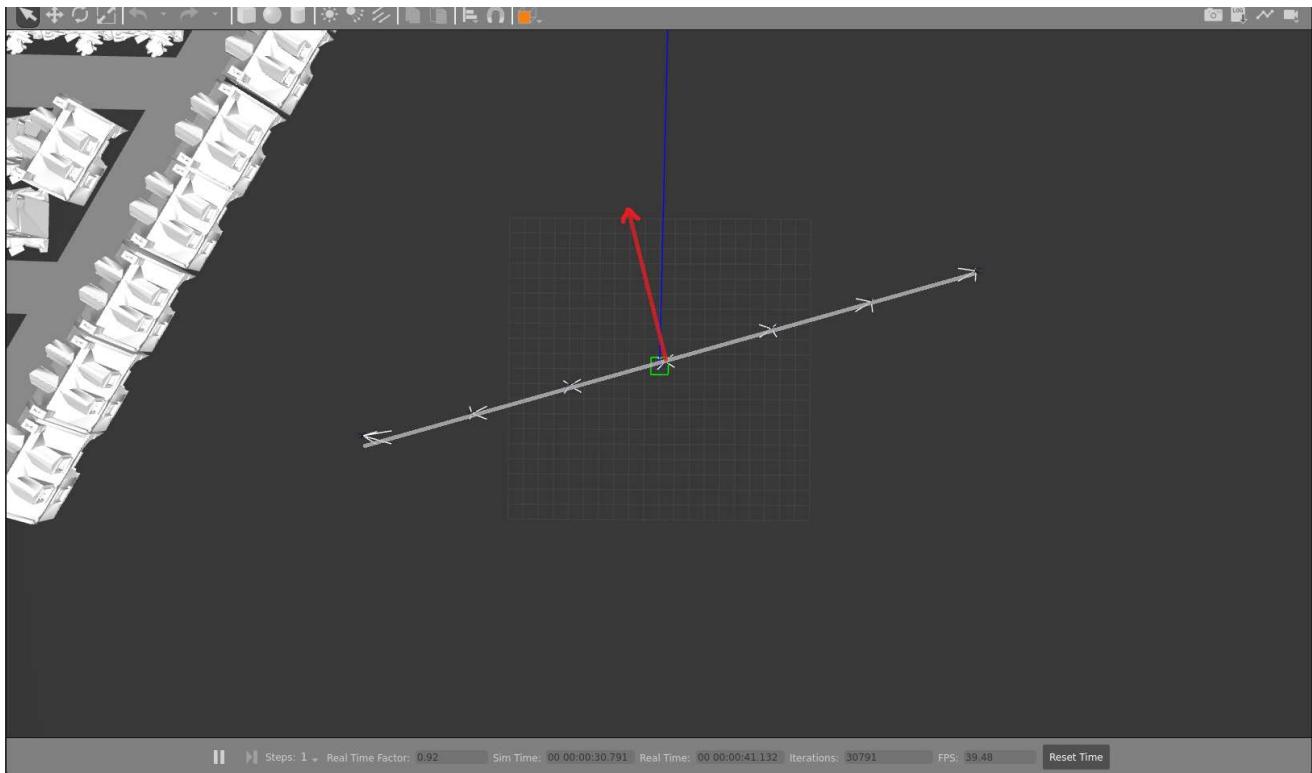


Figura 4.3.1: Sciame di 7 droni allineati alla partenza nello scenario di Amatrice

La spaziatura tra i droni verrà regolata da un nuovo parametro definito in `Parameters.hpp` e `options.yaml: coverArea`. Questo permetterà l'utente di assegnare un valore a scelta per disporre i droni da coprire una superficie specifica alle esigenze della missione. La distanza tra ciascun drone ovviamente dipende anche dal parametro `numVehicle`. Più droni, minore la distanza tra ciascuno per coprire l'area definita. Meno droni, più saranno distanti rischiando di avere punti ciechi nel campo di rilevamento totale dello sciame.

Per coordinare la disposizione dello sciame, è stato introdotto un nuovo parametro di flocking, `Formation`, che influisce sulla velocità di ogni drone per farlo posizionare lungo l'asse perpendicolare alla direzione di avanzamento. Di conseguenza, viene definito in `options.yaml` il relativo fattore moltiplicativo `formationFactor`.

Tuttavia, affinché si stabilisca la formazione in maniera stabile, prima di procedere con la navigazione è stata aggiunta una **nuova fase di volo**: quando i droni devono allinearsi, la flotta si ferma temporaneamente per permettere il posizionamento lungo l'asse. Solo una volta raggiunto l'allineamento, la navigazione riprenderà normalmente. Perciò, i droni si devono allineare alla partenza e ogni volta che cambia la direzione calcolata.

Per rendere più efficiente questo processo, è stato introdotto un parametro di tolleranza, `formationTolerance`, che evita che piccoli disallineamenti dovuti all'inerzia impediscano alla flotta di ripartire. Questo parametro definisce un margine accettabile entro cui i droni possono considerarsi allineati, riducendo i tempi di attesa soprattutto quando il numero di droni è elevato.

4.3.1 Modifiche apportate a Parameters.hpp e Options.yaml

```
namespace fdds
{
    struct Parameters
    {
        int numVehicles;

        float separationFactor;
        float separationRadius;

        float formationFactor;
        float formationTolerance;
        float coverArea;

        float avoidanceRadius;
        float avoidanceFactor;

        float cohesionRadius;
        float cohesionFactor;

        float alignmentRadius;
        float alignmentFactor;

        explicit Parameters(
            const int num_vehicles,
            const float separationFactor,
            const float separationRadius,
            const float formationFactor,
            const float formationTolerance,
            const float coverArea,
            const float avoidance_radius,
            const float avoidance_factor,
            const float cohesion_radius,
            const float cohesion_factor,
            const float alignment_radius,
            const float alignment_factor
        )
    };
}
```

```

        : numVehicles(num_vehicles),
          separationFactor(separationFactor),
          separationRadius(separationRadius),
          formationFactor(formationFactor),
          formationTolerance(formationTolerance),
          coverArea(coverArea),
          avoidanceRadius(avoidance_radius),
          avoidanceFactor(avoidance_factor),
          cohesionRadius(cohesion_radius),
          cohesionFactor(cohesion_factor),
          alignmentRadius(alignment_radius),
          alignmentFactor(alignment_factor)
      {}

      Parameters() = delete;
  };

  inline Parameters loadOptions(bool verbose)
  {
    Yaml::Node config_file;
    Yaml::Parse(config_file, "/home/fourdds/.swarm/options.yaml");

    return Parameters(
      config_file["num_vehicles"].As<int>(),
      config_file["separationFactor"].As<float>(),
      config_file["separationRadius"].As<float>(),
      config_file["formationFactor"].As<float>(),
      config_file["formationTolerance"].As<float>(),
      config_file["coverArea"].As<float>(),
      config_file["avoidanceRadius"].As<float>(),
      config_file["avoidanceFactor"].As<float>(),
      config_file["cohesionRadius"].As<float>(),
      config_file["cohesionFactor"].As<float>(),
      config_file["alignmentRadius"].As<float>(),
      config_file["alignmentFactor"].As<float>()
    );
  }
};

```

```

# options.yaml ✘
home > fourdds > .swarm > # options.yaml
  1  alignmentFactor: 6.0
  2  alignmentRadius: 8.0
  3
  4  cohesionFactor: 0.1
  5  cohesionRadius: 8.0
  6
  7  separationFactor: 1.0
  8  separationRadius: 3.0
  9
 10 avoidanceFactor: 1.5
 11 avoidanceRadius: 3.0
 12
 13 formationFactor: 0.5
 14 formationTolerance: 0.5
 15 coverArea: 40
 16
 17 num_vehicles: 3

```

4.3.2 Modifiche apportate a BaseStation.cpp e BaseStation.hpp

Questa nuova logica viene gestita prettamente dalla *BaseStation*. Per ogni messaggio di *geoPing* ricevuto da un drone, essa aggiornerà i parametri del modello di *Boids*, includendo quello nuovo per la formazione dello sciame. Per implementare questa logica, è stato necessario introdurre nuovi campi nella classe *BaseStation*, che consentono di calcolare e stabilizzare la disposizione dei droni lungo l'asse perpendicolare alla direzione di avanzamento.

I nuovi campi aggiunti sono i seguenti:

- **leaderPosition** (*Eigen::Vector2d*): memorizza la posizione del *drone leader* (*ID = 1*) durante il volo. Questo valore è necessario per convertire le coordinate della formazione, inizialmente calcolate in un sistema di riferimento relativo, nel sistema assoluto della missione.
- **alignPosition** (*Eigen::Vector2d*): rappresenta la posizione fissa che il *drone leader* deve mantenere durante la fase di allineamento. Poiché i droni subiscono forze repulsive per evitare collisioni, il leader potrebbe spostarsi involontariamente, rendendo instabile la formazione. Per risolvere questo problema, *alignPosition* viene fissata alla posizione del leader all'inizio della fase di allineamento o in seguito a un cambio di direzione significativo.
- **alignDirection** (*Eigen::Vector2d*): definisce la direzione lungo la quale i droni devono allinearsi prima della partenza. Stesso ragionamento del campo precedente: se questa direzione fosse continuamente ricalcolata in base alla posizione attuale del leader, eventuali spostamenti involontari causerebbero variazioni imprevedibili, impedendo ai droni di convergere stabilmente. Per questo motivo, *alignDirection* viene fissata in modo analogo a *alignPosition*.
- **aligned** (*bool*): indica se i droni hanno completato l'allineamento. Fino a quando questo valore è *false*, la flotta rimane ferma e i droni ricevono solo comandi di formazione e separazione; solo una volta raggiunto l'allineamento, vengono attivati i parametri di navigazione per procedere verso la destinazione.

```
// Posizione del Drone Leader (ID = 1)
Eigen::Vector2d leaderPosition = Eigen::Vector2d::Zero();
Eigen::Vector2d alignPosition = Eigen::Vector2d::Zero();
Eigen::Vector2d alignDirection = Eigen::Vector2d::Zero();

// Tien conto di quando i Droni si sono allineati alla Partenza o quando Cambia di Direzione
bool aligned = false;
```

L'introduzione di questi campi permette di gestire in modo più preciso e stabile la formazione, evitando oscillazioni e garantendo che i droni si dispongano correttamente lungo l'asse definito.

Per disporre i droni lungo l'asse perpendicolare alla direzione di avanzamento, è stata implementata la funzione *getFormationPosition()*, che calcola la posizione desiderata per ciascun drone in base ai dati forniti dai loro messaggi di *geoPing* e i nuovi campi definiti.

Il calcolo segue i seguenti principi:

1. Determinazione della spaziatura tra i droni

La distanza tra i droni è regolata dal parametro *coverArea*, che definisce l'area complessiva di copertura. La spaziatura effettiva è ottenuta dividendo *coverArea* per il numero di droni

(`numVehicles`). Tuttavia, per evitare configurazioni non ottimali, questa distanza è vincolata da due limiti:

- a. Non può essere maggiore del `cohesionRadius`, altrimenti i droni potrebbero disperdersi eccessivamente.
- b. Non può essere minore del `separationRadius`, per evitare che la forza di separazione li spinga continuamente fuori posizione, impedendo la convergenza.

2. Posizionamento del drone leader

Il *drone leader* (`droneId = 1`) viene fissato al centro della formazione. Durante la fase di allineamento, esso deve mantenere la posizione memorizzata in `alignPosition` per evitare spostamenti causati dalle forze repulsive degli altri droni. Una volta completato l'allineamento (`aligned = true`), può invece aggiornare la sua posizione in base a `leaderPosition`.

3. Distribuzione degli altri droni

Gli altri droni vengono posizionati simmetricamente rispetto all'asse perpendicolare. La loro disposizione segue la logica:

- a. I droni con *ID pari* vengono posizionati a destra rispetto al leader.
- b. I droni con *ID dispari* vengono posizionati a sinistra.
- c. L'offset di ciascun drone viene calcolato rispetto al numero di droni già posizionati, garantendo una distribuzione omogenea lungo l'asse.

4. Calcolo della direzione perpendicolare

La direzione lungo cui allineare i droni è data da un vettore ortogonale a `alignDirection`.

Normalizzandolo, si evita di introdurre scalature indesiderate. L'offset viene quindi proiettato lungo questa direzione per ottenere la posizione desiderata del drone.

5. Formazione e tolleranza di allineamento

La funzione `getFormationPosition()` viene chiamata all'interno di `geoPingCallback()` per calcolare il vettore di formazione come differenza tra la posizione desiderata e la posizione attuale del drone. Se la distanza è inferiore alla soglia `formationTolerance`, il valore viene azzerato per evitare correzioni continue che potrebbero causare oscillazioni indesiderate.

Questo è il corpo della funzione `getFormationPosition()`:

```
Eigen::Vector2d fdds::BaseStation::getFormationPosition(const int droneId)
{
    // Spaziatura tra i droni proporzionata alla Area di Copertura e Numero di Droni
    double spacing = options.coverArea / options.numVehicles;

    // Limiti dello "spacing" tra Droni
    if (spacing > options.cohesionRadius - 0.5)
        spacing = options.cohesionRadius - 0.5;

    else if (spacing < options.separationRadius + 0.5)
        spacing = options.separationRadius + 0.5;

    // Se il drone è il leader (ID 1), lo posizioniamo al centro in cima
    if (droneId == 1) {
        return aligned ? leaderPosition : alignPosition; // Se i droni NON sono ancora allineati,
                                                               // Leader deve "ancorarsi" nella posizione attuale per evitare
                                                               // di farsi spostare dalle altre forze repulsive calcolate
                                                               // rispetto ai droni durante la disposizione
    }

    // Determiniamo la posizione lungo l'asse perpendicolare
    int j = floor(droneId / 2);
    int offset = 0;

    offset = (droneId % 2 == 0) ? -j : +j; // Disposizione simmetrica con "pari" a DX e "dispari" a SX

    // Calcolo del vettore perpendicolare alla direzione
    Eigen::Vector2d perpDirection(-alignDirection.y(), alignDirection.x());
    perpDirection.normalize(); // Normalizzazione per evitare scalature non volute

    // Calcolo dell'offset lungo la direzione perpendicolare
    Eigen::Vector2d position = perpDirection * (offset * spacing);

    // Calcolo della posizione finale rispetto al leader
    return leaderPosition + position;
}
```

Successivamente viene illustrato il frammento di codice della `geoPingCallback()` per l'aggiornamento dei nuovi campi della classe `BaseStation` assieme al calcolo del parametro `Formation`:

```
/// 1. [DATI PER LA FORMAZIONE e RAGGIUNGIMENTO DELLA DESTINAZIONE]
/// - Calcolo della Direzione
/// - Salvataggio della Posizione del Leader
/// - Salvataggio della Direzione verso la Destinazione
if(msg->vehicle_id == 1)
{
    to_one_point(drone_position);
    //zigzag_delta(drone_position); // Aggiorna "direction" per mantenere la direzione di navigazione o aggiornarla
    leaderPosition = drone_position; // Aggiorna "leaderPosition" costantemente per mantenere la Flotta Allineata durante il Movimento

    // Aggiornamento "Posizione di Allineamento" alla Partenza o quando Cambia Direzione
    double angleDeg = std::acos(direction.dot(alignDirection) / (direction.norm() * alignDirection.norm()) * 180.0 / M_PI;
    if (alignPosition == Eigen::Vector2d::Zero() || angleDeg > 30.0)
    {
        alignPosition = drone_position;
    }

    // Aggiornamento "Direzione di Allineamento" alla Partenza o quando Cambia Direzione
    if (alignDirection == Eigen::Vector2d::Zero() || angleDeg > 30.0)
    {
        alignDirection = direction; // Nuova direzione di allineamento
        aligned = false; // Resetto la condizione di formazione per fermare la flotta per disporsi correttamente
    }
}
```

```

/// 2. [CALCOLO DEL PARAMETRO 'FORMAZIONE' e DISPOSIZIONE in FORMAZIONE prima della Partenza]
/// - Calcolo delle Posizioni dei Droni per Disporli lungo l'Asse Perpendicolare alla Direzione calcolata
Eigen::Vector2d formationPosition = getFormationPosition(msg->vehicle_id);
Eigen::Vector2d formation = (formationPosition - drone_position);

if (formation.norm() <= options.formationTolerance) // Considera una certa tolleranza da evitare che i Droni
    formation = Eigen::Vector2d::Zero();           // si correggano eccessivamente da aumentare il tempo di convergenza

```

Questa è la logica finale per poter calcolare questo nuovo parametro di flocking Formation.

Per quanto riguarda la nuova fase di volo per disporre i droni correttamente in formazione prima della partenza al decollo o al cambio di direzione, è stato aggiunto il seguente frammento di codice per far verificare al *drone leader* che tutti i droni della flotta fossero entro una certa tolleranza in posizione di formazione:

```

/// - Invia SOLO Parametri "Formazione" (per la Disposizione) e "Separazione" (per evitare che urtino tra loro) finché non sono tutti in Formazione
if (!aligned) // Una volta allineati alla Partenza, questo algoritmo viene saltato
{
    if(msg->vehicle_id == 1) aligned = true; // A controllare se lo Sciamo sia in Formazione sarà solo il Drone Leader

    for (int i = 0; i < swarmSize; i++)
    {
        const auto& drone_i_position = swarmPositions[i];
        const auto distance = (drone_i_position - drone_position).norm();

        // Controllo per la Separazione
        if (distance < options.separationRadius && i != msg->vehicle_id - 1)
        {
            separation += (options.separationRadius - distance) * (drone_position - drone_i_position).normalized();
            num_separation++;
        }

        // Controllo per la Formazione
        if (msg->vehicle_id == 1 &&      // Solo il Drone Leader controlla se i Droni NON sono in formazione
            (drone_i_position - getFormationPosition(i + 1)).norm() > options.formationTolerance)
        {
            aligned = false;
        }
    }

    // Calcolo della Separazione
    if (num_separation != 0)
        separation /= num_separation;

    // Se non ancora Allineati...
    if (!aligned)
    {
        direction = Eigen::Vector2d::Zero(); // Azzerà la Direzione così che i Droni non si spostano verso la Destinazione durante la Disposizione
        advertiseFlocking(msg->vehicle_id, cohesion, alignment, separation, formation); // parametri != 0: { Separation, Formation }

        direction = alignDirection;          // Re-impostata la Direzione a quella di Allineamento per evitare
                                            // che si aggiorni la Direzione di Allineamento al controllo iniziale
        return;
    }
}

```

Durante questa fase, vengono inviati solo i parametri di *Formazione* e *Separazione*, mentre *Cohesion*, *Alignment* e *Direction* sono azzerati o lasciati a 0 (all'inizio della funzione sono definiti come vettori nulli). Questo assicura che i droni rimangano stazionari durante l'allineamento senza iniziare il movimento verso la destinazione. Il parametro di *Separazione* evita collisioni tra i droni durante la disposizione lungo l'asse di formazione, mentre il parametro di *Formazione* garantisce che ciascun drone converga verso la propria posizione designata. Solo una volta che tutti i droni risultano allineati entro la tolleranza specificata, il *drone leader* imposta `aligned = true`, permettendo alla flotta di iniziare il movimento coordinato verso la destinazione.

L'ultima modifica a `geoPingCallback()` copre il caso in cui la flotta sia arrivata a destinazione. Semplicemente, in questa condizione il parametro *Direction* dovrebbe risultare 0 e perché non ci siano

movimenti indesiderati, allora anche qua *BaseStation* pubblica il messaggio al drone con i parametri di flocking azzerati.

```
/// 3. [CONTROLLO ARRIVO]
/// - Se a questo controllo la Direzione è 0, allora la Flotta è attualmente ARRIVATA a Destinazione
if (direction == Eigen::Vector2d::Zero())
{
    Eigen::Vector2d null_vector = Eigen::Vector2d::Zero();
    advertiseFlocking(msg->vehicle_id, null_vector, null_vector, null_vector, null_vector); // parametri tutti a 0
    return;
}
```

A tal punto, per poter trasmettere anche il nuovo parametro di formazione, è quindi necessario modificare ulteriormente la funzione `advertiseFlocking` per la pubblicazione del messaggio di *flocking* della *BaseStation* in risposta ai droni con la quale è in comunicazione.

```
void fdds::BaseStation::advertiseFlocking(const int vehicle_id,
                                            const Eigen::Vector2d& cohesion,
                                            const Eigen::Vector2d& alignment,
                                            const Eigen::Vector2d& separation,
                                            const Eigen::Vector2d& formation)
{
    fdds_messages::msg::Flocking msg;
    msg.cohesion_x = cohesion(0);
    msg.cohesion_y = cohesion(1);

    msg.alignment_x = alignment(0);
    msg.alignment_y = alignment(1);

    msg.separation_x = separation(0);
    msg.separation_y = separation(1);

    msg.formation_x = formation(0);
    msg.formation_y = formation(1);

    msg.direction_x = direction(0);
    msg.direction_y = direction(1);
    flockingPublishers[vehicle_id - 1]->publish(msg);
}
```

4.3.3 Modifiche apportate a `Drone.cpp` e `Drone.hpp`

Ora che *BaseStation* è equipaggiata con il codice necessario per poter inviare il comando di formazione ai droni della flotta, bisogna adattare la controparte per poter finalizzare l'implementazione.

Prima di andare sulla logica di controllo, bisogna aggiungere il nuovo parametro `Formation` tra i campi privati della classe *Drone* in `Drone.hpp`:

```

    /// Vector to follow for cohesion.
    Eigen::Vector2d cohesion = Eigen::Vector2d::Zero();

    // Vector to follow for alignment.
    Eigen::Vector2d alignment = Eigen::Vector2d::Zero();

    // Vector to follow for separation.
    Eigen::Vector2d separation = Eigen::Vector2d::Zero();

    // Vector to follow for formation
    Eigen::Vector2d formation = Eigen::Vector2d::Zero();

    // Vecotr to follow for direction
    Eigen::Vector2d direction = Eigen::Vector2d::Zero();

```

Lo stesso vale per la funzione di ricezione dei messaggi di *flocking* da *BaseStation* per poter mantenere aggiornato questo nuovo campo della classe:

```

void fdds::Drone::flockingSubscriptionCallback(fdds_messages::msg::Flocking::ConstSharedPtr msg)
{
    cohesion = Eigen::Vector2d{msg->cohesion_x, msg->cohesion_y};
    alignment = Eigen::Vector2d{msg->alignment_x, msg->alignment_y};
    separation = Eigen::Vector2d{msg->separation_x, msg->separation_y};

    formation = Eigen::Vector2d{msg->formation_x, msg->formation_y};

    direction = Eigen::Vector2d{msg->direction_x, msg->direction_y};
}

```

Ora che la classe è dotata del parametro e ne riceve gli aggiornamenti appropriatamente, rimane solo che aggiungerlo al calcolo della velocità in *flockingLogic()*:

```

void fdds::Drone::flockingLogic()
{
    ...

    // Calcolo della velocità
    Eigen::Vector2d velocity = Eigen::Vector2d::Zero();

    // Aggiunta della forza di coesione
    velocity += options.cohesionFactor * cohesion;

    // Aggiunta della forza di allineamento
    velocity += options.alignmentFactor * alignment;

    // Aggiunta della forza di separazione
    velocity += options.separationFactor * separation;

    // Aggiunta della forza di formazione
    velocity += options.formationFactor * formation;

    // Aggiunta della direzione parameters.directionFactor
    velocity += (direction.normalized() * options.directionFactor);
}

```

5 Simulazione della Missione

5.1 Obiettivo

L'obiettivo della missione affidata allo sciame di droni è il rilevamento dei dispersi nel minor tempo possibile. Si vuole dimostrare che con un maggior numero di droni e/o un'area di copertura maggiore si riduce il tempo di ritrovamento.

Per valutare l'efficacia di diverse configurazioni dello sciame, è stato scelto di usare come algoritmo di scansione quello con strategia a pattern *zig-zag* già fornito dal codice sorgente e registrare il tempo impiegato per individuare l'80% dei target come indicatore di performance. Perciò per ogni prova è stato definito un numero di droni assieme ad una certa area di copertura e misurato il tempo necessario per portare a termine tale obiettivo.

Nei risultati sarà inevitabile la presenza di una certa variabilità fra le varie prove con la stessa configurazione, per questo si deve prendere in considerazione, oltre alla semplice media aritmetica, anche gli indici di **varianza** e **deviazione standard**. Questi ultimi sono misure statistiche che quantificano la dispersione dei dati rispetto alla media. La *varianza* indica quanto i valori si discostano mediamente dalla media al quadrato, mentre la *deviazione standard* ne rappresenta la radice quadrata, fornendo un'unità di misura direttamente confrontabile con i dati originali. Valori elevati di questi indici suggeriscono una maggiore variabilità nei risultati, mentre valori bassi indicano una maggiore stabilità e ripetibilità delle prove. In questo modo, forniscono una **misura dell'affidabilità** delle varie configurazioni testate, permettendo di valutare quanto siano consistenti nel tempo. Questo permette di certificare meglio la prevedibilità del comportamento di una configurazione rispetto a un'altra, facilitando lo studio della stabilità e l'identificazione dei fattori che influenzano la consistenza di una rispetto ad altre meno affidabili.

Questo approccio permette di determinare il numero ottimale di droni e area di copertura con l'obiettivo finale di massimizzare l'efficienza della missione.

5.2 Raccolta e Analisi dei Dati Sperimentali

Per ciascuna configurazione della flotta di droni, sono state condotte almeno cinque prove con un numero crescente di droni e area di copertura: 3, 5, 7 unità di droni e 20, 30, 40 unità di area di copertura. Quindi un totale di almeno quaranta cinque prove considerando ciascuna combinazione.

Durante ogni simulazione, il tempo impiegato è stato registrato in secondi e ogni momento di rilevamento del n-esimo target è stato salvato in file CSV generato automaticamente durante la simulazione. Ogni file CSV è stato in seguito processato mediante script Python, i quali hanno permesso di estrarre informazioni utili per l'analisi dei dati. I risultati sono stati rappresentati graficamente, fornendo un riferimento visivo intuitivo e dettagliato per l'interpretazione e il confronto di tutte le configurazioni sperimentate.

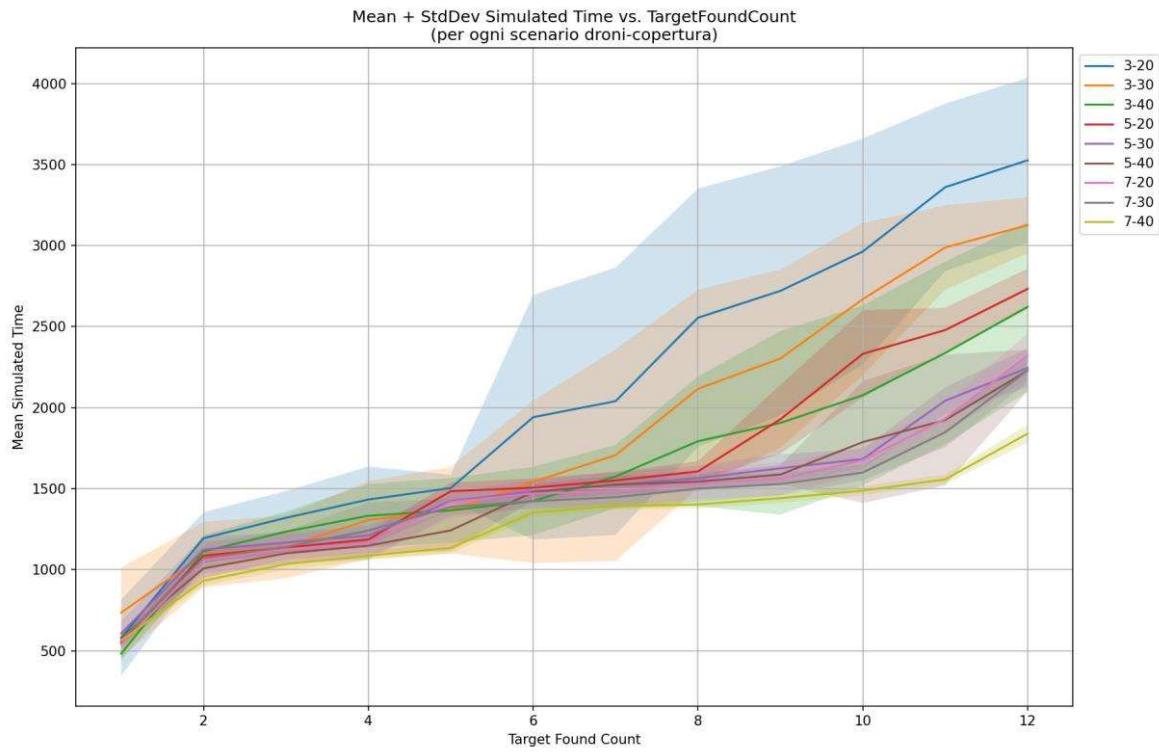


Figura 5.2.1: Tempo Medio e Variazione Standard per raggiungere l'obiettivo al variare delle configurazioni dello sciame

Dal grafico mostrato, si può chiaramente dimostrare quanto detto all'introduzione dell'obiettivo: l'aumento del numero dei droni e/o dell'area di copertura risulta in una riduzione nel tempo di ritrovamento.

Inizialmente i tempi di completamento calano rapidamente con l'aumentare dei droni, ma la riduzione si affievolisce progressivamente. Ciò potrebbe dipendere dal fatto che i target, pur distribuiti pseudo-casualmente, sono fissi e i droni seguono percorsi simili in tutte le configurazioni. Di conseguenza, è prevedibile una convergenza nei tempi di miglioramento. Inoltre, la deviazione standard diminuisce insieme al tempo medio, indicando maggiore affidabilità, soprattutto per configurazioni con più droni.

Un ritrovamento importante è come in alcune configurazioni con meno droni ma maggiore area di copertura mostrano tempi di ritrovamento dell'ultimo target inferiori o simili a quelle con più droni e copertura minore. Ciò suggerisce che la copertura influisca più del numero di droni sull'efficienza. Questo è rilevante perché dimostra come la nuova logica di formazione ottimizzi le prestazioni semplicemente regolando un parametro privo di costi aggiuntivi, a differenza dell'incremento del numero di droni. I grafici sottostanti mostrano meglio questa tendenza evidenziando appunto il tempo meglio di ritrovamento dell'ultimo target.

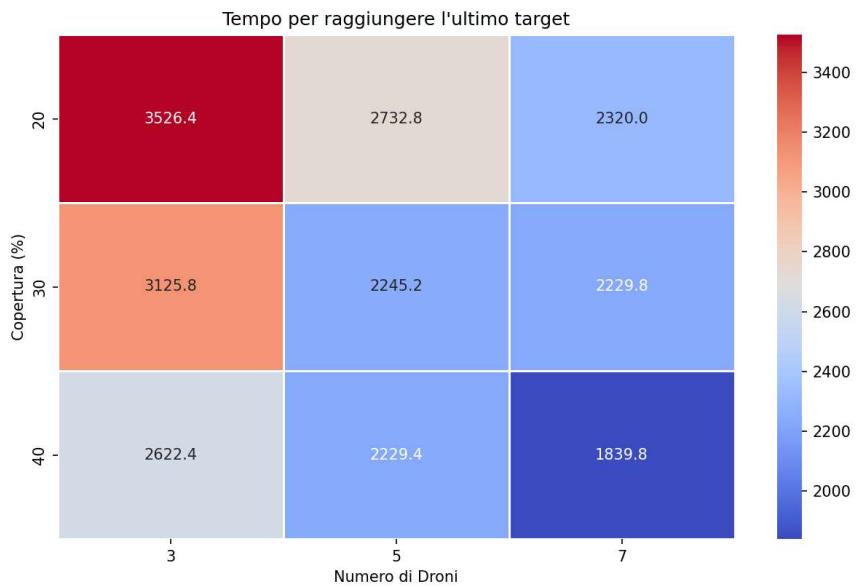


Figura 5.3.2: Mappa di calore che mostra il Tempo medio per il ritrovamento dell'ultimo target per ciascuna configurazione

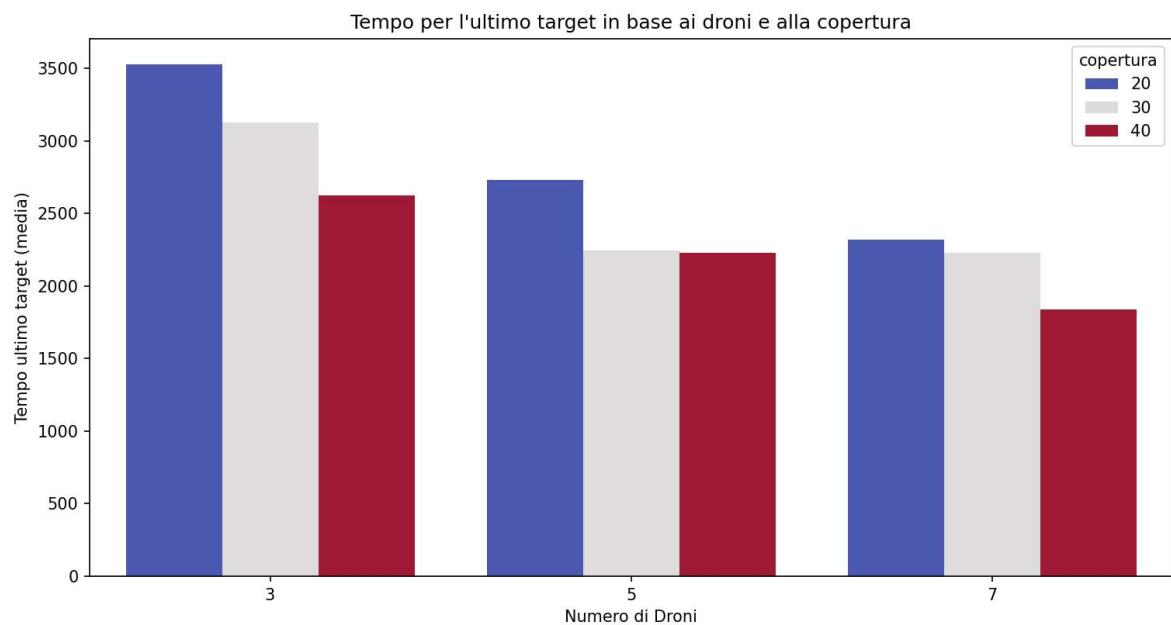


Figura 5.3.3: Istogramma che mostra il Tempo medio per il ritrovamento dell'ultimo target per ciascuna configurazione

Per esempio, il caso 3-40 risulta leggermente migliore del 5-20; lo stesso per i casi 5-40 e 7-30. Addirittura le configurazioni 5-40 e 7-30 sono praticamente uguali.

6 Conclusioni

In conclusione, i risultati di questa tesi dimostrano che aumentando il numero di droni e area di copertura si aumenta significativamente l'efficienza delle operazioni di ricerca. Inoltre, ha permesso di dimostrare che ottimizzando solamente l'area di copertura dei droni migliora le prestazioni della flotta dei droni nelle operazioni di ricerca, lasciando invariato il numero di unità impiegate. Quindi, tale approccio non solo migliora la velocità di rilevamento dei dispersi, ma risulta essere una strategia di costo-efficienza particolarmente vantaggiosa.

La capacità di raggiungere risultati comparabili o migliori con un minor numero di droni e una maggiore area di copertura pone in evidenza un aspetto critico per le operazioni di soccorso: la **gestione ottimale delle risorse disponibili**. In scenari reali, dove il budget e le risorse logistiche possono essere limitati, l'approccio dimostrato offre una soluzione pragmatica per massimizzare l'efficacia delle operazioni mantenendo al contempo i costi sotto controllo.

L'implicazione pratica di queste scoperte è profonda, soprattutto per le agenzie di soccorso e per gli organizzatori di missioni di ricerca in vasti ambienti. Adottando strategie basate su queste evidenze, è possibile implementare operazioni di soccorso più rapide ed efficienti, migliorando le probabilità di salvare vite umane e riducendo il periodo di esposizione al rischio sia per i dispersi che per i soccorritori.

Questa tesi fornisce quindi una base solida per la riconsiderazione delle attuali pratiche operative, incoraggiando un riorientamento verso modelli più sostenibili ed economicamente vantaggiosi, che potrebbero rivoluzionare il modo in cui le missioni di ricerca e soccorso vengono condotte in futuro.

Sitografia

1. <https://ingvterremoti.com/2019/08/23/un-aggiornamento-a-tre-anni-dal-terremoto-del-24-agosto-2016/>
2. <https://www.geopop.it/terremoto-di-amatrice-6-anni-fa-la-scossa-di-magnitudo-6-0-che-devasto-il-centro-italia/>
3. https://classic.gazebosim.org/tutorials?cat=guided_b&tut=guided_b2
4. <https://docs.px4.io/v1.12/en/simulation/gazebo.html>
5. <https://docs.px4.io/main/en/>
6. <https://px4.io/ecosystem/ecosystem-overview/>
7. <https://code.visualstudio.com/docs/remote/wsl>
8. <https://learn.microsoft.com/it-it/windows/wsl/install#install-wsl-command>
9. https://www.researchgate.net/figure/Typical-view-from-Pigeons-in-the-Park_fig1_2358480
10. [https://www.researchgate.net/publication/268077894 Optimisation Of Boids Swarm Model Base d On Genetic Algorithm And Particle Swarm Optimisation Algorithm Comparative Study](https://www.researchgate.net/publication/268077894_Optimisation Of Boids Swarm Model Based On Genetic Algorithm And Particle Swarm Optimisation Algorithm Comparative Study)