

Università di Pisa

Dipartimento di ingegneria dell'informazione
Ingegneria robotica e dell'automazione

Sistemi Subacquei

Pianificazione del moto di un AUV per la circumnavigazione di una macchia di inquinante

Docenti:

Riccardo Costanzi
Andrea Munafò

Elaborato a cura di:
Sebastiano Bianco
Matteo Mariani
Alberto Regoli

Anno Accademico 2022-2023

Indice

Introduzione	1
1 Descrizione della missione	3
1.1 Correlazione tra i gruppi	3
1.2 Costruzione del nodo Planning	5
1.2.1 File dei parametri	5
1.2.2 Elenco dei subscriber	6
1.2.3 Elenco dei publisher	9
1.3 Vincoli del problema	11
1.4 Scelte progettuali	12
2 Costruzione della missione	13
2.1 Costruzione della missione	14
2.2 Algoritmo 1: esplorazione	14
2.3 Algoritmo 2: circumnavigazione	18
2.3.1 Ricerca del valore di inquinante massimo relativo	18
2.3.2 Costruzione dei waypoint O_1 e O_2	25
2.3.3 Conclusione Circumnavigazione	32
2.3.4 Costruzione dei "reverse waypoint"	35
2.3.5 Costruzione dell'algoritmo di ritorno	38
3 Validazione algoritmi	41
3.1 Validazione algoritmo di esplorazione	41
3.2 Validazione algoritmo 2	42
4 Risultati sperimentali	45
4.1 Prove intermedie	45
4.1.1 Pubblicazione dei waypoint di esplorazione	45
4.1.2 Errore nell'angolo di orientazione	46
4.2 Prove al lago	47
Conclusioni	51

Elenco delle figure

0.1	Laghetti di Campo (PI).	1
0.2	Zeno.	2
1.1	Publisher e Subscriber dei vari gruppi.	3
1.2	Rqt_graph complessivo.	4
1.3	Macchina a stati del robot.	8
1.4	Area di lavoro e griglia circoscritta.	11
1.5	Sistemi di riferimento utilizzati.	12
2.1	Descrizione del funzionamento degli algoritmi attraverso un flowchart.	13
2.2	Rappresentazione della traiettoria di ispezione, in cui il waypoint più vicino al robot è WP0: in blu è rappresentata la traiettoria a "clessidra", in viola la traiettoria a "rombo".	14
2.3	Sistemi di riferimento	21
2.4	Rappresentazione degli angoli su cui si basa la valutazione sulla matrice <i>elementiDaEstrarre</i> .	21
2.5	Rappresentazione delle configurazioni possibili della matrice <i>elementiDaEstrarre</i> al variare dell'angolo di matrice <i>orientation</i> .	22
2.6	Costruzione waypoint O_1 e O_2 .	25
2.7	Posizioni possibili del waypoint O_1 .	26
2.8	Configurazioni "gomito basso" (sinistra) e "gomito alto" (destra).	26
2.9	Traiettoria circumnavigazione.	27
2.10	Grandezze geometriche per il calcolo dei waypoint.	28
2.11	Posizione di O_1 .	30
2.12	Esempio di traiettoria per la circumnavigazione.	32
2.13	Rappresentazione dell'angolo di <i>Heading</i> e <i>distanceFinal</i> necessari per concludere la missione.	33
2.14	Esempio di chiusura della missione con una macchia di inquinante concava.	34
2.15	Traiettoria dei <i>Reverse waypoint</i> quando Zeno è entrato nell'inquinante.	36
2.16	Traiettoria dei <i>Reverse waypoint</i> quando l'inquinante è a contatto con il perimetro dell'area di lavoro.	37
2.17	Traiettoria di ritorno (in giallo).	38
3.1	Esplorazione tramite la tecnica <i>clessidra-rombo</i> .	41
3.2	Situazione al bordo dell'area di lavoro e della griglia.	43
4.1	Interfaccia grafica simulatore.	45
4.2	Rappresentazione dell'angolo di orientazione α_{rot} .	46
4.3	L'immagine a sinistra rappresenta la traiettoria eseguita da Zeno durante la sperimentazione; l'immagine a destra mostra la traiettoria visualizzata su <i>Rviz</i> , dove i colori rappresentano la percentuale di inquinante.	47
4.4	L'immagine a sinistra rappresenta la traiettoria desiderata; l'immagine a destra mostra la traiettoria visualizzata su <i>Rviz</i> , dove i colori rappresentano la percentuale di inquinante.	48

Elenco delle figure

4.5	Rappresentazione grafica delle componenti della posizione del robot in terna Local: in blu, il risultato della sperimentazione; in verde, il risultato desiderato.	49
4.6	Confronto grafico delle componenti della posizione del robot in terna Local: in blu, il risultato della sperimentazione; in verde, il risultato desiderato. . .	49
4.7	Confronto tra i valori di inquinante rilevati e quelli forniti dal gruppo sensing.	50

Elenco delle tabelle

1.1 Topic e relativa tipologia di messaggio.	4
--	---

Listings

1.1	File dei parametri.	5
1.2	Subscriber di <i>virtual_map</i>	6
1.3	Definizione del messaggio <i>OccupancyGrid</i>	6
1.4	Subscriber di <i>flag_map</i>	7
1.5	Subscriber di <i>reliable_map</i>	7
1.6	Subscriber di <i>zeno/mm/mission_status</i>	7
1.7	Subscriber di <i>nav_status</i>	8
1.8	Definizione del messaggio <i>NavStatus</i>	8
1.9	definizione dei messaggi custom	9
1.10	Publisher di <i>zeno/mm/upload_mission</i>	9
1.11	Definizione del messaggio <i>WaypointList</i>	9
1.12	Publisher di <i>zeno/mm/start_mission</i>	10
1.13	Publisher di <i>zeno/mm/pause_mission</i>	10
1.14	Publisher di <i>zeno/mm/delete_mission</i>	10
1.15	Publisher di <i>posWay</i>	10
1.16	Definizione messaggio <i>Float32MutliArray</i>	11
2.1	Funzione che trasforma le componenti di un punto da terna ECEF a terna NED.	15
2.2	Funzione che restituisce la matrice di rotazione da terna NED a terna Local.	15
2.3	Trasformazione dei WP in terna Local.	16
2.4	Costruzione dei WP posti nella mezzeria dei lati di lavoro.	16
2.5	Definizione della lista dei WP in terna Local.	16
2.6	Funzione che restituisce il vertice più vicino al robot per l'avvio della esplosione.	16
2.7	Funzione che costruisce la traiettoria in terna Local.	16
2.8	Funzione che converte le componenti da terna Local in terna ECEF.	17
2.9	Funzione che costruisce la lista dei WP in terna ECEF.	17
2.10	Funzione che costruisce il messaggio.	17
2.11	Pubblicazione della lista dei WP in terna ECEF.	17
2.12	Reshape in forma matriciale ottenendo la <i>mapVirtual_matrix</i>	18
2.13	Reshape in forma matriciale ottenendo la <i>mapFlag_matrix</i>	18
2.14	Reshape in forma matriciale ottenendo la <i>mapReliable_matrix</i>	18
2.15	Funzione utilizzata per l'estrazione matriciale.	19
2.16	Funzione che restituisce la sotto-matrice proveniente dalla <i>mapVirtual_matrix</i>	19
2.17	Funzione utilizzata per determinare la dimensione della matrice da estrarre dalla <i>productPositionFlag</i> al variare della distanza del robot dal bordo della mappa.	19
2.18	Funzione utilizzata per determinare il punto di massimo relativo di inquinante più distante dal robot.	23
2.19	Definizione e inizializzazioni delle variabili.	27
2.20	Posizione di O_2 in terna Local.	28

Listings

2.21 Definizione di α	28
2.22 Costruzione parametri geometrici al variare di ϑ	28
2.23 Posizione di O_1 per "gomito basso" e "gomito alto".	29
2.24 Posizione di O_1 in <i>terna matrice</i>	29
2.25 Calcolo del theta_alto.	29
2.26 Posizione di O_1	31
2.27 Punto di inizio della Circumnavigazione.	32
2.28 Funzioni per concludere la circumnavigazione.	33
2.29 Traiettoria di ritorno.	34
2.30 Costruzione dei <i>reverse waypoint</i>	36
2.31 Pubblicazione dei <i>reverse waypoint</i>	37
2.32 Traiettoria di ritorno.	38
2.33 Definizione possibili traiettorie con la funzione <i>switchfinal</i>	39

Introduzione

Nel seguente elaborato si mostra la pianificazione di una missione per la ricerca e la circumnavigazione di un agente inquinante all'interno di uno scenario operativo tramite il controllo di un Autonomous Underwater Vehicle (AUV).

Questa relazione è un modulo del macro-progetto *Inquinante* e si occupa della costruzione della traiettoria da imporre al robot. Il monitoraggio della macchia è stato fatto sia in simulazione sia in ambiente reale ai "Laghetti di Campo".



Figura 0.1 – Laghetti di Campo (PI).

L'AUV utilizzato è *Zeno* di proprietà dell'Università di Pisa e prodotto dall'azienda *MDM Team*. Il veicolo ha capacità di manovrabilità elevate (grazie ai suoi 8 motori) e possibilità di sostituzione veloce della batteria facilitando le ispezioni e rilievi subacquei. Inoltre Zeno, in figura 0.2, può operare anche come Remote Operated Vehicle (ROV) grazie ad una boa

Introduzione

di collegamento o ad un classico cavo ombelicale, ovvero la modalità utilizzata per le prove al lago ¹.



Figura 0.2 – Zeno.

L'elaborato si suddivide in 4 capitoli, nei quali verranno esaminati il codice, le simulazioni e i dati raccolti ponendo attenzione al percorso di sviluppo del progetto e all'interazione tra i vari gruppi.

Il codice è stato scritto in linguaggio *Python* ed è stato usato l'ambiente *ROS* per la costruzione dei nodi e la verifica delle loro connessioni. Invece le simulazioni sono state fatte sull'interfaccia grafica fornita dall'azienda *MDM Team*.

¹/<https://www.dii.unipi.it/news/news/il-robot-subacqueo-zeno-nei-laghetti-di-campo-il-monitoraggio-ambientale>

Capitolo 1

Descrizione della missione

In questo capitolo ci proponiamo di discutere gli aspetti relativi alla missione da svolgere per il nostro gruppo di planning, ovvero l'esplorazione e la circumnavigazione di una macchia di inquinante all'interno di un'area di lavoro.

1.1 Correlazione tra i gruppi

Il lavoro da svolgere è stato ripartito su 3 gruppi principali:

- **gruppo sensing:** si occupa della modellazione del sensore e della creazione della macchia d'inquinante all'interno dell'area di lavoro;
- **gruppo mapping:** si occupa della realizzazione della mappa dell'ambiente in real-time sulla base della posizione in cui si trova il robot e delle misure del sensore;
- **gruppo planning:** si occupa di definire un algoritmo per la pianificazione del moto del robot sulla base delle informazioni fornite dal gruppo di *mapping*.

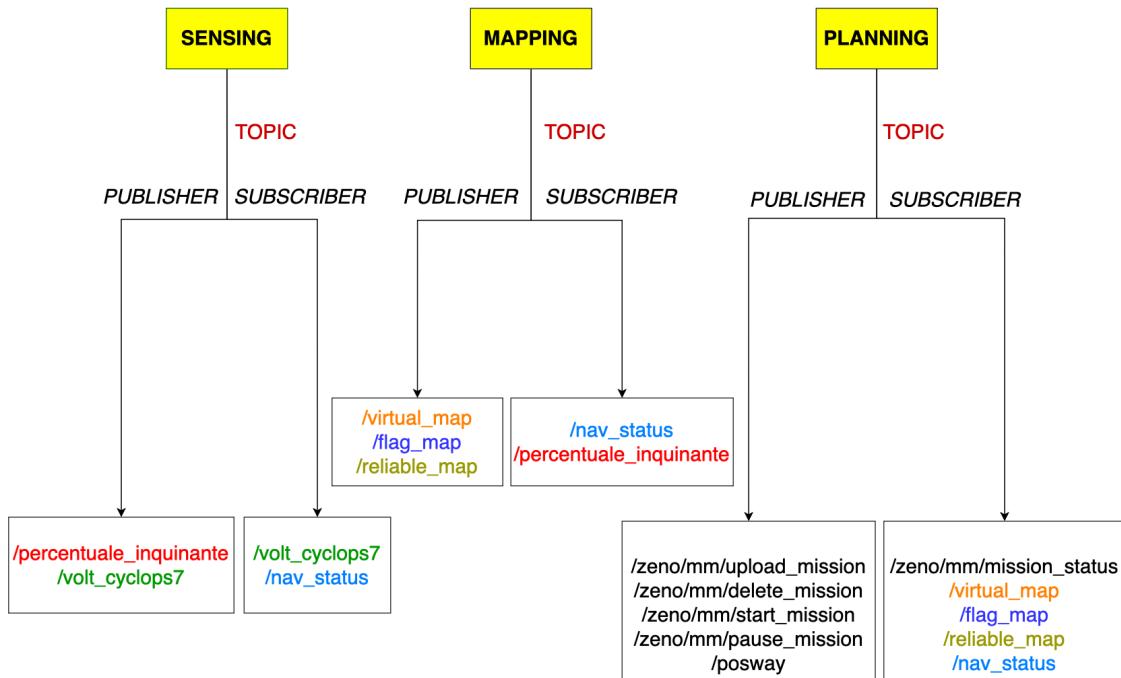


Figura 1.1 – Publisher e Subscriber dei vari gruppi.

In figura 1.1 vengono mostrate le *topic* utilizzate dai vari gruppi e i rispettivi *publisher*/*subscriber*. Per quanto riguarda le *topic*, sono state decise e concordate all'unanimità sulla

Capitolo 1 Descrizione della missione

base delle richieste promosse da ciascun gruppo: ad esempio per il gruppo di planning, è necessario avere la *topic virtual_map* (proveniente dal gruppo di mapping) riguardante la mappa generata in modo da poter ricevere le informazioni riguardanti la percentuale d'inquinante in base della posizione del robot. Dopo aver effettuato una serie di ragionamenti ed avere delineato il nome di tutte le *topic* utili, si è pensato alla tipologia di messaggio da trasmettere per ciascuna di esse come mostrato nella seguente tabella:

Nome Topic	Tipologia messaggio	Tipo di gruppo
percentuale_inquinante volt_cyclops7	sensor_msgs Float32	SENSING
virtual_map flag_map reliable_map	OccupancyGrid OccupancyGrid OccupancyGrid	MAPPING
zeno/mm/upload_mission zeno/mm/delete_mission zeno/mm/start_mission zeno/mm/pause_mission zeno/mm/mission_status posway	WaypointList Empty Empty Empty String Float32MultiArray	PLANNING
nav_status	NavStatus	TUTTI

Tabella 1.1 – Topic e relativa tipologia di messaggio.

Una volta definita la tipologia di messaggi, è stata fatta una prova generale per verificare le connessioni tra i nodi e il corretto passaggio dei dati: per fare ciò sono stati eseguiti i vari nodi ed utilizzato il comando *ROS rqt_graph*.

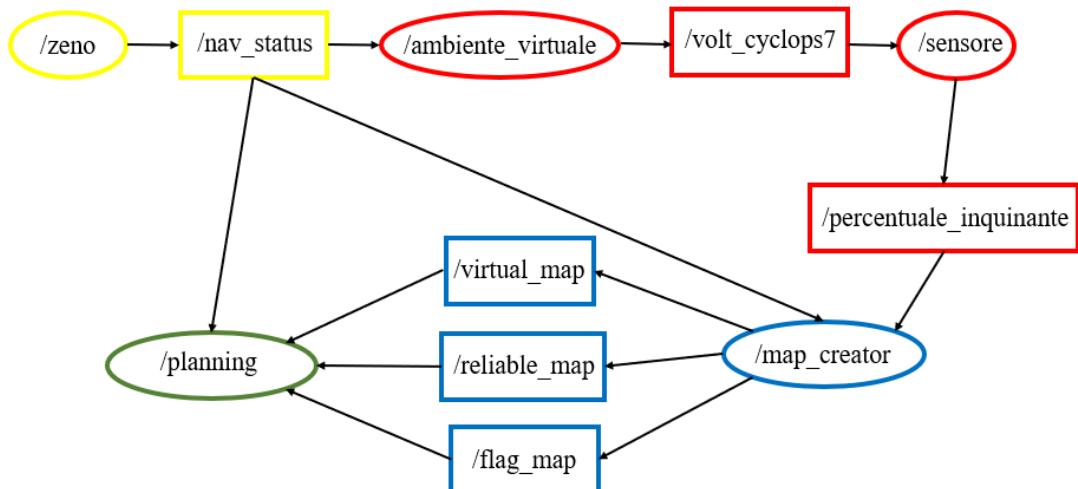


Figura 1.2 – Rqt_graph complessivo.

Come si può osservare in figura 1.2 vi sono tutti i collegamenti tra i 3 gruppi:

- il gruppo di **sensing** riceve la posizione del robot e la utilizza per estrarre il corrispondente valore della misura in volt, la quale sarà inviata al nodo sensore per trasformarla in percentuale;
- il gruppo di **mapping** riceve tale informazione e aggiorna le 3 mappe;
- il gruppo di **planning** riceve le 3 mappe e attua il corretto algoritmo di pianificazione in base alla posizione della macchia.

1.2 Costruzione del nodo Planning

Il nostro gruppo ha definito un unico nodo *ROS* denominato *planning* costituito da una classe python: all'interno del costruttore vengono definiti tutti i *publisher* e *subscriber*, citati nella sezione precedente, mentre nel corpo della classe vengono definite le funzioni di callabck per ciascun *subscriber*. Di seguito si propone una trattazione in merito allo "scheletro" del nodo mettendo in risalto anche il file dei parametri, in comune a tutti e 3 i gruppi.

1.2.1 File dei parametri

In questa sottosezione si analizza il **file dei parametri** utilizzato da tutti i gruppi per definire variabili di inizializzazione: nell'ambiente *ROS* questo tipo di file prende il nome di *parameter server* e rappresenta un dizionario che i nodi utilizzano per archiviare e recuperare i parametri in fase di esecuzione.

```

1 # Parametri missione Inquinante
2
3 # Sistemi di riferimento
4 Olocal: [0, 0, 0] # origine LOCAL espressa in NED
5
6 # Dati Griglia
7 dimensioni_celle : [1,1] # [m x m]
8 r : 3 # raggio di interpolazione [m]
9 size : 7 # dimensione della matrice productPositionFlag
10
11 # Dati sensore
12 frequency: 10 # [Hz]
13 std_perc: 0.0026
14
15 # Soglie circumnavigazione
16 soglia : 0.6 # in modo che il minimo sia 0.6 (inquinante*affidabilita)
17 sogliaAvvicinamento: 0.4 # soglia per avvicinarsi alla macchia (
    inquinante*affidabilita)
18
19 # Vertici area di lavoro in 11
20 WP0: [43.706144064520120, 10.476042510093047]
21 WP1: [43.706683743126035, 10.475253099163552]
22 WP2: [43.706108194470780, 10.474939590251552]
23 WP3: [43.705697317829156, 10.475719979341854]
```

Listing 1.1 – File dei parametri.

Capitolo 1 Descrizione della missione

Nel file dei parametri sono presenti:

- **dati della griglia:** necessari a definire la mappa e la dimensione della matrice estratta;
- **dati sensore:** necessari a definire i parametri per il modello del sensore;
- **soglie di circumnavigazione:** necessarie per l'esecuzione dell'algoritmo circumnavigazione;
- **vertici area di lavoro:** necessari a definire l'area in cui il robot lavora in coordinate geodetiche.

1.2.2 Elenco dei subscriber

Di seguito vengono mostrati tutti i nodi di tipo *subscriber* utilizzati: principalmente si occupano di leggere le mappe in ingresso, la posizione del robot e lo stato del robot.

Subscriber virtual_map

La seguente riga di codice definisce il nodo *subscriber* relativo alla *topic virtual_map* generata dal gruppo di mapping:

```
1 self.subMapInqu = rospy.Subscriber("/virtual_map", OccupancyGrid, self.  
    callback_map)
```

Listing 1.2 – Subscriber di *virtual_map*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *nav_msgs/OccupancyGrid* ed è così costituito¹:

```
1 # This represents a 2-D grid map, in which each cell represents the  
# probability of occupancy.  
2  
3 Header header  
4  
5 #MetaData for the map  
6 MapMetaData info  
7  
8 # The map data, in row-major order, starting with (0,0). Occupancy  
# probabilities are in the range [0,100]. Unknown is -1.  
9 int8[] data
```

Listing 1.3 – Definizione del messaggio *OccupancyGrid*.

Come riportato dalla documentazione ufficiale di *ROS*, il messaggio è definito da:

- **header:** contiene parametri come *tempo* e *frame_id* della mappa;
- **info:** contiene parametri quali *risoluzione*, *larghezza*, *altezza*, *origine* della mappa;
- **data:** contiene i dati relativi a ciascuna cella della mappa (0-100 inquinante, -1 dove il robot non è mai stato).

¹http://docs.ros.org/en/melodic/api/nav_msgs/html/msg/OccupancyGrid.html

Subscriber flag_map

La seguente riga di codice definisce il nodo *subscriber* relativo alla *topic flag_map* generata dal gruppo di mapping:

```
1 self.subMapAff = rospy.Subscriber("/flag_map", OccupancyGrid, self.  
callback_flagmap)
```

Listing 1.4 – Subscriber di *flag_map*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *nav_msgs/OccupancyGrid*. L'unica differenza rispetto alla *topic virtual_map* è che nel campo **data** i valori sono così imposti:

- **100** dove il robot è già stato almeno una volta;
- **0** dove il robot non è mai stato;
- **-1** dove il robot non può andare in quanto al di fuori dell'area di lavoro.

Subscriber reliable_map

La seguente riga di codice definisce il nodo *subscriber* relativo alla *topic reliable_map* generata dal gruppo di mapping:

```
1 self.subMapOcc = rospy.Subscriber("/reliable_map", OccupancyGrid, self.  
callback_reliablemap)
```

Listing 1.5 – Subscriber di *reliable_map*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *nav_msgs/OccupancyGrid*. L'unica differenza rispetto alle due *topic* precedenti è che nel campo **data** i valori sono così imposti:

- **0-100** rappresenta il valore percentuale di affidabilità di una determinata cella.

Subscriber zeno/mm/mission_status

La seguente riga di codice definisce il nodo *subscriber* relativo alla *topic zeno/mm/mission_status* utile a leggere lo stato in cui si trova il robot:

```
1 self.subZenoStatus = rospy.Subscriber("zeno/mm/mission_status", String,  
self.callback_ZenoStatus)
```

Listing 1.6 – Subscriber di *zeno/mm/mission_status*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *String* e basa il suo funzionamento sulla seguente macchina a stati:

Capitolo 1 Descrizione della missione

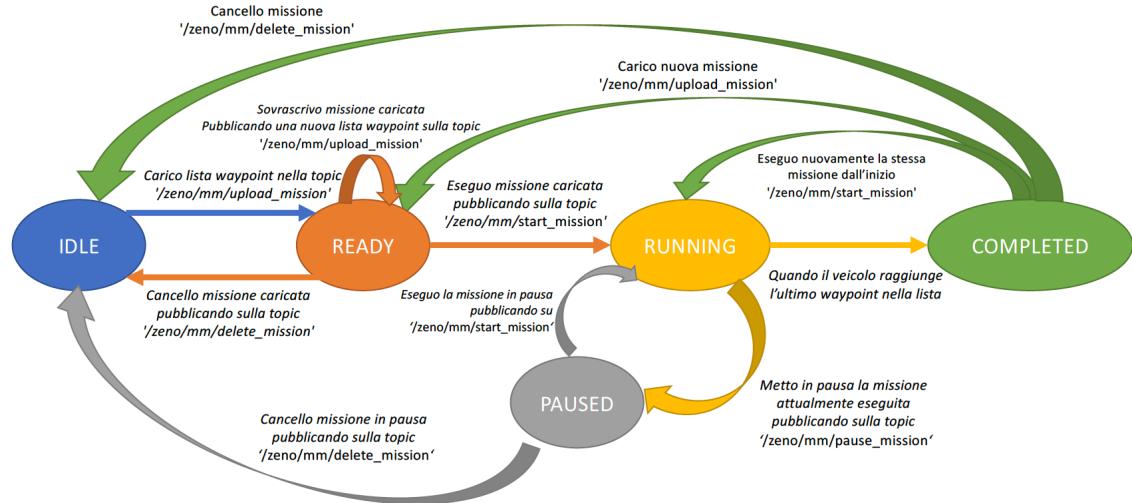


Figura 1.3 – Macchina a stati del robot.

La lettura dello stato del robot permette al nostro gruppo di capire in quale configurazione si trova l'AUV ed agire di conseguenza caricando ed eseguendo i waypoint utili all'esplorazione e alla circumnavigazione della macchia.

Subscriber nav_status

La seguente riga di codice definisce il nodo *subscriber* relativo alla *topic nav_status* utile a leggere lo stato in cui si trova il robot:

```
1 self.subPosition = rospy.Subscriber("/nav_status", NavStatus, self.
    callback_pose)
```

Listing 1.7 – Subscriber di *nav_status*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *NavStatus* ed è un custom marta_msgs/NavStatus così costituito:

```
1 Header header
2
3 marta_msgs/Position position
4
5 marta_msgs/Euler orientation
6
7 marta_msgs/Quaternion quaternion
8
9 geometry_msgs/Vector3 ned_speed
10
11 geometry_msgs/Vector3 omega_body
12
13 uint8 gps_status
14
15 bool initialized
```

Listing 1.8 – Definizione del messaggio *NavStatus*.

Il messaggio è dipendente da altri messaggi custom così definiti:

```

1 Position.msg
2   float64 latitude
3   float64 longitude
4   float64 depth
5
6 Euler.msg
7   float64 roll
8   float64 pitch
9   float64 yaw
10
11 Quaternion.msg
12   float64 w
13   float64 x
14   float64 y
15   float64 z

```

Listing 1.9 – definizione dei messaggi custom

La lettura dello stato di navigazione permette al nostro gruppo di estrarre la posizione del robot (in coordinate geodetiche), a cui corrisponde una cella della mappa, e verificare se vi è un'elevata percentuale di inquinante.

1.2.3 Elenco dei publisher

Di seguito vengono mostrati tutti i nodi di tipo *publisher* utilizzati per pubblicare i waypoint generati, cambiare lo stato del robot e generare marker per la visualizzazione su *Rviz*.

Publisher *zeno/mm/upload_mission*

La seguente riga di codice definisce il nodo *publisher* relativo alla *topic* *zeno/mm/upload_mission* utile a pubblicare la lista di waypoint:

```

1 self.pubUploadMission = rospy.Publisher("zeno/mm/upload_mission",
                                          WaypointList, queue_size=10)

```

Listing 1.10 – Publisher di *zeno/mm/upload_mission*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *WaypointList* ed è così costituito:

```

1 WaypointList.msg
2   Waypoint[] waypoint_list
3
4 Waypoint.msg
5   marta_msgs/Position position
6   float32 speed
7   string control_mode
8   float32 altitude

```

Listing 1.11 – Definizione del messaggio *WaypointList*.

Dopo aver generato la lista dei waypoint di esplorazione o circumnavigazione sulla base della posizione del robot e della macchia, vengono pubblicati: in questo modo si ha il cambiamento di stato del robot da *IDLE* a *READY*. Dopo averli caricati ed esserci assicurati del cambiamento di stato, il passo successivo è quello di far partire la missione attraverso il prossimo *publisher*.

Capitolo 1 Descrizione della missione

Publisher zeno/mm/start_mission

La seguente riga di codice definisce il nodo *publisher* relativo alla *topic* *zeno/mm/start_mission* utile ad iniziare la missione:

```
1 self.pubStartMission = rospy.Publisher("zeno/mm/start_mission",Empty,  
queue_size=10)
```

Listing 1.12 – Publisher di *zeno/mm/start_mission*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *Empty* e quindi non presenta alcun tipo di campo al suo interno. Tale messaggio permette di modificare lo stato del robot da *READY* a *RUNNING* e dare così avvio alla missione.

Publisher zeno/mm/pause_mission

La seguente riga di codice definisce il nodo *publisher* relativo alla *topic* *zeno/mm/pause_mission* utile a mettere in pausa la missione:

```
1 self.pubPauseMission = rospy.Publisher("zeno/mm/pause_mission",Empty,  
queue_size=10)
```

Listing 1.13 – Publisher di *zeno/mm/pause_mission*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *Empty* e quindi non presenta alcun tipo di campo al suo interno. Tale messaggio permette di modificare lo stato del robot da *RUNNING* a *PAUSED* e mettere così in pausa la missione: viene utilizzato quando si vuole caricare una nuova lista di waypoint.

Publisher zeno/mm/delete_mission

La seguente riga di codice definisce il nodo *publisher* relativo alla *topic* *zeno/mm/delete_mission* utile a cancellare la missione:

```
1 self.pubDeleteMission = rospy.Publisher("zeno/mm/delete_mission",Empty,  
queue_size=10)
```

Listing 1.14 – Publisher di *zeno/mm/delete_mission*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *Empty* e quindi non presenta alcun tipo di campo al suo interno. Tale messaggio permette di modificare lo stato del robot da *PAUSED* a *IDLE* ed eliminare così la missione per permettere il caricamento di una nuova lista di waypoint.

Publisher posWay

La seguente riga di codice definisce il nodo *publisher* relativo alla *topic* *posWay* utile a pubblicare i waypoint caricati e la posizione del robot che verranno poi utilizzati da un secondo nodo per la visualizzazione tramite Marker in *Rviz*:

```
1 self.pubPosWayZeno = rospy.Publisher("/posWay",Float32MultiArray,  
queue_size=10)
```

Listing 1.15 – Publisher di *posWay*.

Come già anticipato nella tabella 1.1 il messaggio è di tipo *Float32MultiArray* ed è così costituito²:

²http://docs.ros.org/en/melodic/api/std_msgs/html/msg/Float32MultiArray.html

```

1 MultiArrayLayout layout      # specification of data layout
2 float32[]      data        # array of data

```

Listing 1.16 – Definizione messaggio *Float32MultiArray*.

L'elemento fondamentale del messaggio sono i **data** definiti da una lista di *Float32*: qui verranno inseriti i waypoint in coda e la posizione del robot in testa.
 Questa lista verrà poi letta da un nodo (esterno a quello di planning) che ne definirà i *Marker* per la successiva visualizzazione grafica su *Rviz*.

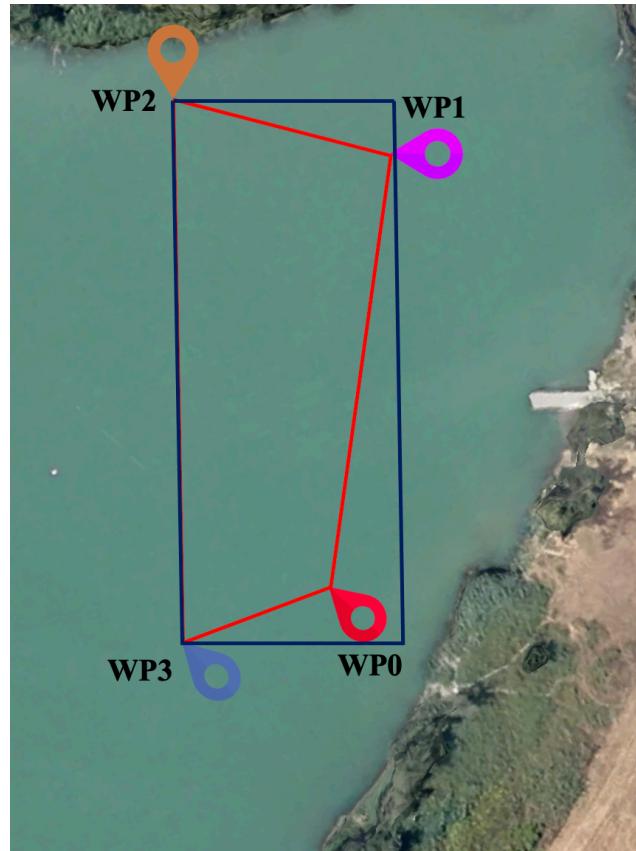
1.3 Vincoli del problema

I vincoli relativi alla pianificazione della circumnavigazione sono di diverso tipo:

- **vincoli di base**: dati dalla traccia del progetto;
- **vincoli di algoritmo**: dovuti alle particolari specifiche richieste dall'algoritmo.

Il primo **vincolo di base** è quello di non sapere dove si trova la macchia di inquinante inizialmente: questo implica un algoritmo con una prima fase di esplorazione e una successiva circumnavigazione non appena vengono rilevati valori di inquinante superiori ad una certa soglia.

Il secondo vincolo, legato al precedente, è quello di non poter attraversare la macchia di inquinante: ciò esclude la costruzione di algoritmi che eseguono una perlustrazione generale dell'area per individuare la macchia e successivamente circumnavigarla.

**Figura 1.4** – Area di lavoro e griglia circoscritta.

Capitolo 1 Descrizione della missione

Definita l'area di lavoro come mostrato in figura 1.4 è necessario non oltrepassarne il confine durante la missione. Come si può notare, oltre all'area di lavoro è stata costruita una griglia rettangolare esterna che la circoscrive in modo da generare una mappa di navigazione e successivamente avere sistemi di riferimento paralleli alle righe e alle colonne per uno spostamento più fluido.

Infine vi è un limite sulla distanza alla quale vengono posizionati i waypoint ovvero non possono essere ad una distanza inferiore ad un metro o maggiore di un km rispetto alla posizione di zero.

I **vincoli di algoritmo** sono legati ai diversi waypoint necessari per la pianificazione della traiettoria: questi verranno approfonditi nel capitolo 2.

1.4 Scelte progettuali

La scelta migliore per semplificare la lettura delle 3 mappe in forma matriciale è quella di posizionare l'origine della terna *Local* coincidente con quella della terna *NED*: tale coordinata viene letta dal file dei parametri dopo essere stata caricata dal gruppo di mapping.

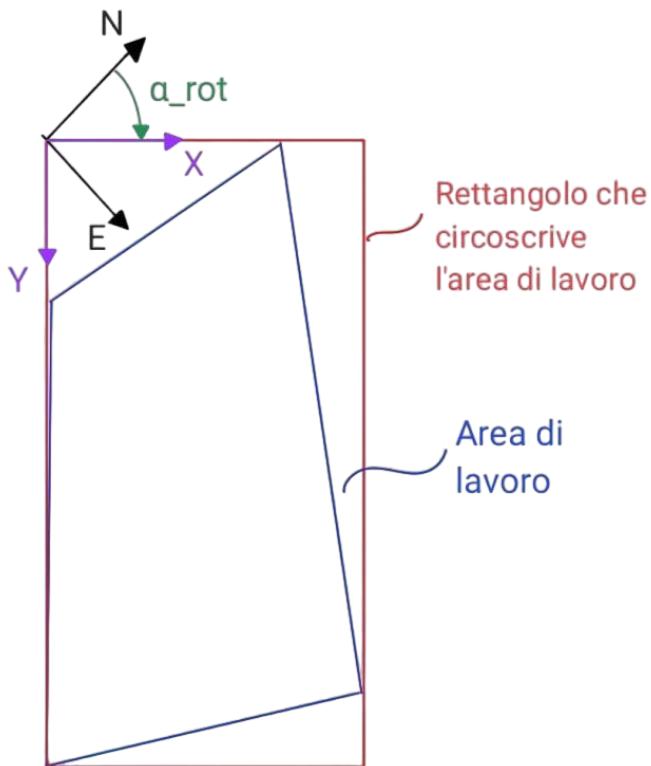


Figura 1.5 – Sistemi di riferimento utilizzati.

Questa decisione dà la possibilità di avere l'asse x coincidente con le righe e l'asse y coincidente con le colonne, questo perché l'origine della terna *Local* indica la componente [0,0] della matrice.

Infatti, la risoluzione con cui vengono costruite le mappe, pari a celle di 1[m] x 1[m], permette di associare alla posizione del robot le componenti [i,j] della matrice. Per esempio, si osserva che se la posizione del robot in terna Local è pari a [a, b] (con a e b valori di tipo float in [m]), allora esso si trova nella cella [int(a), int(b)].

Capitolo 2

Costruzione della missione

In figura 2.1 è mostrato il diagramma relativo al funzionamento degli algoritmi. All'interno sono presenti i *processi* rappresentati con dei rettangoli e le *decisioni* prese rappresentate con dei rombi.

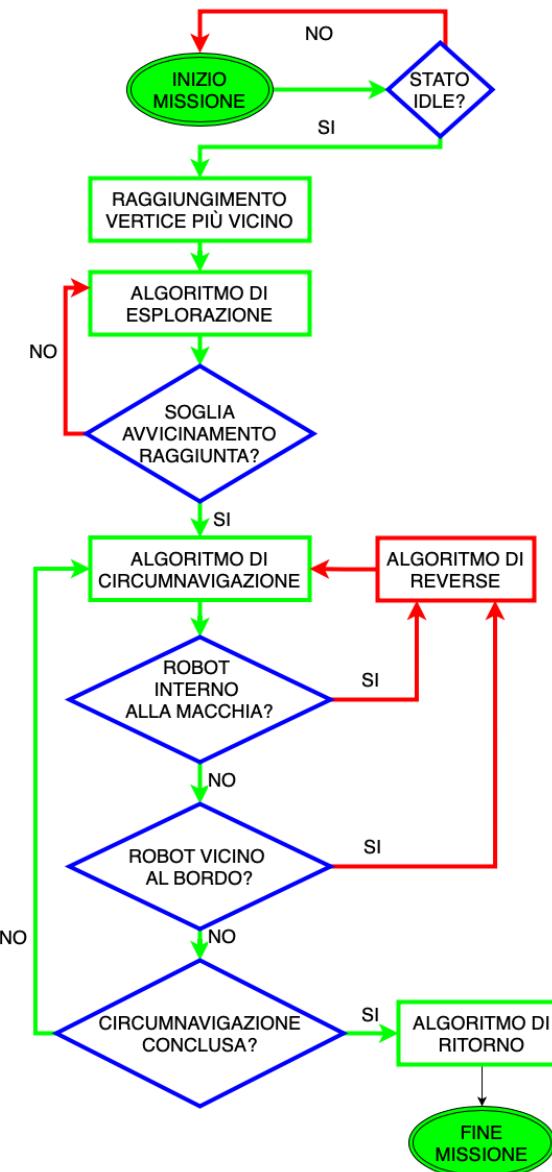


Figura 2.1 – Descrizione del funzionamento degli algoritmi attraverso un flowchart.

2.1 Costruzione della missione

La missione per la ricerca della macchia di inquinante si suddivide in due fasi:

- esplorazione dell'area di interesse;
- circumnavigazione della macchia.

La prima permette di scandagliare la zona di lavoro, affinché si possa raccogliere il maggior numero di dati possibili dal sensore, invece la seconda permette di circumnavigare l'inquinante a una data soglia e far tornare al punto di raccolta Zeno.

2.2 Algoritmo 1: esplorazione

In questa sezione si parlerà della perlustrazione iniziale che vede l'obiettivo di identificare la posizione della macchia di inquinante che in seguito verrà circumnavigata. Cosa necessaria sono i waypoint che delimitano l'area di lavoro, da cui può effettivamente avere inizio la descrizione dell'algoritmo di perlustrazione.

Il primo passo consiste nell'ordinare i vertici dell'area di lavoro in senso antiorario, dove il WP0 corrisponde al waypoint più a Est. In seguito all'immissione di Zeno in acqua, per avviare la missione di esplorazione, si determina il vertice dell'area di lavoro più vicino al robot e da lì inizia la missione.

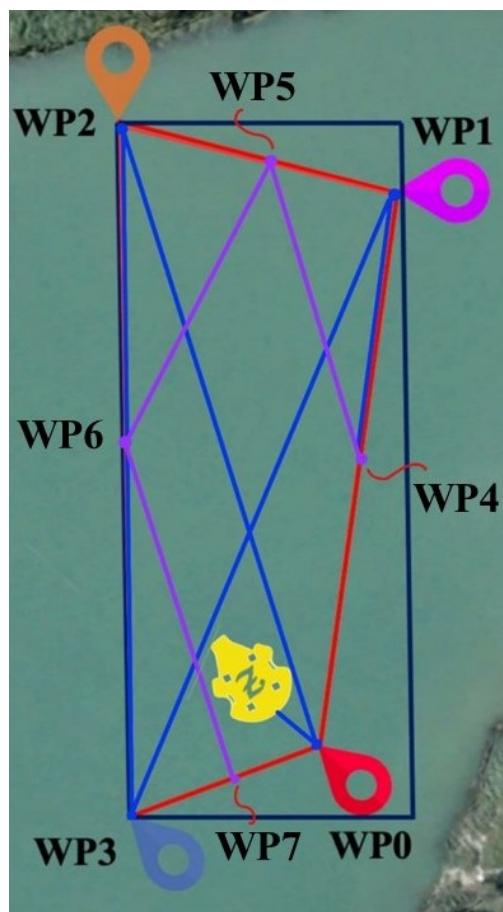


Figura 2.2 – Rappresentazione della traiettoria di ispezione, in cui il waypoint più vicino al robot è WP0: in blu è rappresentata la traiettoria a "clessidra", in viola la traiettoria a "rombo".

Al variare del vertice di partenza si costruisce la traiettoria di ispezione della mappa che permette di far muovere il robot prima lungo le diagonali principali dell'area di lavoro e successivamente, nell'ipotesi di non aver trovato la macchia di inquinante, di coprire le zone non ancora attraversate, ovvero la traiettoria passa per i punti di mezzeria dei lati della zona di interesse.

In questo modo si sviluppa la traiettoria di esplorazione in 2 fasi: la prima, in cui si percorre una traiettoria a forma di 'clessidra', la seconda, in cui si percorre un tragitto a forma di 'rombo'.

Tale strategia si reputa ottima per raccogliere un numero sufficiente di misure per la determinazione circa la posizione della macchia da circumnavigare.

L'algoritmo ha inizio dalla conoscenza dei vertici dell'area di lavoro in componenti in terna ECEF. Sfruttando la funzione *ll2ne* si ottengono le componenti in terna NED.

```

1 def ll2ne(l10, ll):
2     if len(l10)==2 and len(ll)==2:
3         lat0 = l10[0]
4         lon0 = l10[1]
5         lat = ll[0]
6         lon = ll[1]
7
8         lat = lat * math.pi/180
9         lon = lon * math.pi/180
10        lat0 = lat0 * math.pi/180
11        lon0 = lon0 * math.pi/180
12
13        dlat = lat - lat0
14        dlon = lon - lon0
15
16        a = 6378137.0
17        f = 1 / 298.257223563
18        Rn = a / math.sqrt(1 - (2 * f - f * f) * math.sin(lat0) * math.sin(lat0))
19        Rm = Rn * (1 - (2 * f - f * f)) / (1 - (2 * f - f * f) * math.sin(lat0) *
20                                         math.sin(lat0))
21
22        n = dlat / math.atan2(1, Rm)
23        e = dlon / math.atan2(1, Rn * math.cos(lat0))
24        ne = [n,e]
25        return ne

```

Listing 2.1 – Funzione che trasforma le componenti di un punto da terna ECEF a terna NED.

Successivamente ci si avvale della costruzione della matrice di rotazione *C_ne2lo* che permette la trasformazione delle componenti in terna NED in terna Local rappresentato in figura 1.5, con l'utilizzo dell'angolo di rotazione *alpha_rot*.

```

1 def matrixNED2Local(alpha_rot):
2     C_ne2lo = [[math.cos(alpha_rot), math.sin(alpha_rot), 0],
3                 [-math.sin(alpha_rot), math.cos(alpha_rot), 0],
4                 [0, 0, 1]]
5
6     return C_ne2lo

```

Listing 2.2 – Funzione che restituisce la matrice di rotazione da terna NED a terna Local.

Capitolo 2 Costruzione della missione

A tal punto è possibile definire i primi waypoint in terna Local:

```
1 for i in WP:  
2     w_local_i = f.positionLocal(i, self.C_ne2lo)  
3     self.w_local.append(w_local_i)  
4 w_local_0 = self.w_local[0]  
5 w_local_1 = self.w_local[1]  
6 w_local_2 = self.w_local[2]  
7 w_local_3 = self.w_local[3]
```

Listing 2.3 – Trasformazione dei WP in terna Local.

I successivi waypoint, posti nella mezzeria dei lati dell'area di lavoro per la costruzione della traiettoria a forma di rombo, in terna Local:

```
1 w_local_4 = 0.5*(w_local_0 + w_local_1)  
2 w_local_5 = 0.5*(w_local_1 + w_local_2)  
3 w_local_6 = 0.5*(w_local_2 + w_local_3)  
4 w_local_7 = 0.5*(w_local_3 + w_local_0)  
5 w_local_8 = w_local_0
```

Listing 2.4 – Costruzione dei WP posti nella mezzeria dei lati di lavoro.

Si definisce la lista *w_localTotal* contenente tutti i waypoint necessari per la prima traiettoria di esplorazione.

```
1 w_localTotal = [w_local_0, w_local_1, w_local_2, w_local_3,  
2     w_local_4, w_local_5, w_local_6, w_local_7, w_local_8]
```

Listing 2.5 – Definizione della lista dei WP in terna Local.

La funzione *distance* permette di determinare il vertice più vicino alla posizione del robot, appena immesso in acqua, da cui avrà inizio la missione.

```
1 def distance(vertices, P_local):  
2     for i in vertices:  
3         d.append(math.sqrt((P_local[0]-i[0])**2 + (P_local[1]-i[1])**2))  
4     min_index = d.index(min(d))  
5     return min_index
```

Listing 2.6 – Funzione che restituisce il vertice più vicino al robot per l'avvio della esplorazione.

In base a quale vertice risulta più vicino al robot, ottenuto in uscita dalla funzione *distance*, i waypoint contenuti all'interno della lista *w_localTotal* vengono ordinati determinando la traiettoria, chiamata *w_trajectory*.

```
1 def switch(way1, w_localTotal):  
2     if np.all(way1 == w_localTotal[0]): # se il waypoint piu vicino e WPO  
3         w_trajectory = [w_localTotal[0], w_localTotal[2],  
4             w_localTotal[3], w_localTotal[1], w_localTotal[4],  
5             w_localTotal[5], w_localTotal[6], w_localTotal[7],  
6             w_localTotal[8]]  
7     elif np.all(way1 == w_localTotal[1]): # se il waypoint piu vicino e WP1  
8         w_trajectory = [w_localTotal[1], w_localTotal[3],  
9             w_localTotal[2], w_localTotal[0], w_localTotal[4],  
10            w_localTotal[5], w_localTotal[6], w_localTotal[7],  
11            w_localTotal[8]]  
12    elif np.all(way1 == w_localTotal[2]): # se il waypoint piu vicino e WP2  
13        w_trajectory = [w_localTotal[2], w_localTotal[0],  
14            w_localTotal[1], w_localTotal[3], w_localTotal[6],  
15            w_localTotal[7], w_localTotal[4], w_localTotal[5],  
16            w_localTotal[8]]  
17    elif np.all(way1 == w_localTotal[3]): # se il waypoint piu vicino e WP3
```

```

18     w_trajectory = [w_localTotal[3], w_localTotal[1],
19                     w_localTotal[0], w_localTotal[2], w_localTotal[6],
20                     w_localTotal[7], w_localTotal[4], w_localTotal[5],
21                     w_localTotal[8]]
22     return np.array(w_trajectory)

```

Listing 2.7 – Funzione che costruisce la traiettoria in terna Local.

La pubblicazione dei waypoint prevede che siano espressi in terna ECEF, allora si utilizza la funzione *waypointLocal2ll*.

```

1 def waypointLocal2LL(w_local,C_lo2ne):
2     w_ned = np.array(np.dot(C_lo2ne,w_local) + Olocal)
3     W_ned2ll = np.array(geodetic_functions.ned2lld(WP_origin,w_ned))
4     return W_ned2ll

```

Listing 2.8 – Funzione che converte le componenti da terna Local in terna ECEF.

Infine si definisce il messaggio:

```

1 def waypointsList (w_trajectory,C_lo2ne):
2     wayList = WaypointList()
3     for e in w_trajectory:
4         w_ned2ll.append(waypointLocal2LL(e,C_lo2ne))
5     for i in w_ned2ll:
6         way = waypoints(i)
7         wayList.waypoint_list.append(way)
8     return wayList

```

Listing 2.9 – Funzione che costruisce la lista dei WP in terna ECEF.

e si pubblica la lista di waypoint, nella topic *upload_mission*:

```

1 def waypoints(w_ned2ll):
2     way = Waypoint()
3     way.position.latitude = w_ned2ll[0]
4     way.position.longitude = w_ned2ll[1]
5     way.position.depth = w_ned2ll[2]
6     way.speed = 0.5 # m/s
7     way.control_mode = "depth"
8     way.altitude = 0.0
9     return way

```

Listing 2.10 – Funzione che costruisce il messaggio.

Definite tutte le funzioni necessarie, queste vengono richiamate nella *callback_pose* e nel momento in cui lo stato di Zeno diventa *IDLE*, si calcolano i waypoint necessari per la missione di esplorazione una volta sola, sfruttando la flag *self.primoPasso*. Infine, si pubblica la lista di waypoint in terna ECEF, denominata *wayList*.

```

1 if self.primoPasso:
2     self.minIndex = f.distance(w_localTotal[0:4], self.P_local)
3     way1 = w_localTotal[self.minIndex]
4     self.w_trajectory = f.switch(way1, w_localTotal)
5     wayList = f.waypointsList(self.w_trajectory,self.C_lo2ne)
6     self.pubUploadMission.publish(wayList)
7     wayList=[]

```

Listing 2.11 – Pubblicazione della lista dei WP in terna ECEF.

2.3 Algoritmo 2: circumnavigazione

L'idea dell'algoritmo di circumnavigazione si basa sulla costruzione di due waypoint, come si vedrà in seguito. Uno di essi sarà posizionato nella zona relativa al valore di inquinante scelto per circumnavigare la macchia. La determinazione di tale punto sarà analisi della seguente sezione. L'altro verrà costruito successivamente in modo geometrico.

2.3.1 Ricerca del valore di inquinante massimo relativo

Il gruppo mapping fornisce 3 mappe in formato *OccupancyGrid*. Il primo passo consiste nel trasformarle in formato matriciale, ottenendo:

- ***mapVirtual_matrix***: contiene valori -1 se in quelle celle non sono state effettuate misurazioni o valori compresi tra 0 e 100, che rappresentano la percentuale di inquinante misurato dal sensore, se sono state fatte misure.

```
1 mapVirtual_matrix = np.array(msg_virtualMap.data).reshape((int(height), int(length)))
```

Listing 2.12 – Reshape in forma matriciale ottenendo la *mapVirtual_matrix*.

- ***mapFlag_matrix***: contiene valori -1, 0 e 100: -1 se tali celle sono al di fuori dell'area di lavoro; 0 se non sono state effettuate misurazioni; 100 se ci sono stato almeno una volta.

```
1 mapFlag_matrix = np.array(msg_flagMap.data).reshape((int(height), int(length)))
```

Listing 2.13 – Reshape in forma matriciale ottenendo la *mapFlag_matrix*.

- ***mapReliable_matrix***: contiene valori tra 0 e 100 che rappresentano l'affidabilità con cui sono state interpolate le misurazioni esterne alla cella occupata dal robot. Si puntualizza che l'interpolazione non si effettua su tutta la mappa, ma solo su una porzione di essa, precisamente su un intorno di raggio 3 metri della posizione occupata da Zeno

```
1 mapReliable_matrix = np.array(msg_reliableMap.data).reshape((int(height), int(length)))
```

Listing 2.14 – Reshape in forma matriciale ottenendo la *mapReliable_matrix*.

Tutte le matrici sono di dimensioni *height* x *length*, valori ottenuti dal gruppo mapping dal file *circ_rect* che caratterizzano la grandezza della mappa che circoscrive l'area di lavoro. Essendo valori di tipo *float* è necessario trasformarli di tipo *int* per rappresentare le dimensioni delle mappe in formato matriciale.

Determinate le tre matrici, come già accennato nel capitolo 1.4, risulta la scelta migliore porre il sistema di riferimento locale sul vertice della mappa in corrispondenza della cella $[0,0] = [i,j]$, in modo da avere l'asse x corrispondente all'asse su cui viene incrementata la componente i-esima della matrice, 'l'asse riga', e l'asse y corrispondente all'asse su cui viene incrementata la componente j-esima della matrice, 'l'asse colonna'.

Per tener conto dell'affidabilità per le misure predette, si effettua un prodotto elemento per elemento tra la matrice *mapVirtual_matrix* e la matrice *self.mapReliableSubMatrix* ottenendo *positionMatrix*, ancora in formato matriciale. Successivamente, si divide per 10000 assicurandosi di avere valori di tipo *float*, decimali, dell'ordine di 10^{-1} , al fine di confrontare i valori con le soglie.

2.3 Algoritmo 2: circumnavigazione

Per l'obiettivo di progettazione, non è necessario effettuare valutazioni su tutta la mappa, allora si costruisce la sotto-matrice di dimensione 7x7 al variare della cella corrispondente alla posizione del robot nelle mappe. Quindi per il prossimo passo sarà necessario conoscere la cella occupata dal robot, che risulterà essere quella centrale della sotto-matrice estratta, cioè la cella [3,3].

In questa operazione viene utilizzata la funzione *submatrix*, la quale richiede in ingresso la matrice da cui effettuare l'operazione di estrazione, la riga e la colonna da cui ha inizio l'estrazione, che rappresentano la nuova cella [0,0], e la dimensione finale della matrice estratta, cioè 7.

```
1 def submatrix(matrix, startRow, startCol, size):
2     return matrix[startRow:startRow+size,startCol:startCol+size]
```

Listing 2.15 – Funzione utilizzata per l'estrazione matriciale.

Tale operazione si effettua sulla *positionMatrix/10000.00*, ottenendo:

```
1 positionMatrix = (mapVirtual_matrix)*(self.mapReliableSubMatrix)
2
3 productPositionFlag = f.submatrix(positionMatrix/10000.00, self.P_local[0].
4     astype(int)-int(size/2), self.P_local[1].astype(int)-int(size/2), size)
```

Listing 2.16 – Funzione che restituisce la sotto-matrice proveniente dalla *mapVirtual_matrix*

Quest'ultima matrice contiene i valori su cui si baseranno le scelte sul movimento da far compiere al robot.

Un problema affrontato che riguarda la dimensione della matrice *productPositionFlag* è il caso in cui il robot è nei pressi dei bordi della mappa che circoscrive l'area di lavoro. Per poter mantenere la matrice *productPositionFlag* di dimensione 7x7 nelle vicinanze del perimetro è necessario utilizzare il metodo del Padding al fine di aggiungere almeno 3 celle lungo ogni riga e colonna. Ciò non è stato possibile e si è ricorso all'utilizzo di una dimensione minore e dipendente dalla distanza dai bordi, espressa dalla variabile *new_size*.

```
1 # Vengono costruite 4 casistiche:
2 # 1 - [0:size/2+1,0:height-size/2-1]
3 # 2 - [0:length-size/2-1,height-size/2+1:height]
4 # 3 - [length-size/2-1:length,size/2+1:height]
5 # 4 - [size/2+1:length,0:size/2+1]
6 # All'interno di ciascuna casistica vengono eseguiti dei controlli per
7 # verificare se il robot è troppo vicino al bordo.
8 # Se la variabile distanza minima diventa True, il robot esegue i
# waypoint reverse in modo da portarsi lontano dal bordo.
9 if not reverseWaypoint and not self.algoritmoRitorno:
10 if self.P_local[0] <= int(size/2)+1 and self.P_local[1] <= (length-int(
11     size/2)-1):
12     dim = int(min(self.P_local[1],self.P_local[0]))
13     flagNewSize = True
14     if dim%2 != 0:
15         dim = dim-1
16     if dim <= 3:
17         distanza_minima = True
18         new_size = 2*dim + 1 # dimensione della nuova matrice da estrarre
19         productPositionFlag = f.submatrix(positionMatrix/10000.00, self.P_local
20             [0].astype(int)-dim, self.P_local[1].astype(int)-dim, new_size)
21
22 elif self.P_local[0] < (height - int(size/2)-1) and (length - int(size/2)
23     -1) < self.P_local[0] < length:
24     dim = int(min(self.P_local[1],height-self.P_local[0]))
```

Capitolo 2 Costruzione della missione

```

21 flagNewSize = True
22 if dim%2 != 0:
23     dim = dim/2 - 1
24     if dim <= 3:
25         distanza_minima = True
26         new_size = 2*dim + 1 # dimensione della nuova matrice da estrarre
27         productPositionFlag = f.submatrix(positionMatrix/10000.00, self.P_local
28                                         [0].astype(int)-dim, self.P_local[1].astype(int)-dim, new_size)
29
30     elif (height - int(size/2)-1) < self.P_local[0] < height and int(size/2)
31         +1 < self.P_local[1] < length:
32         dim = int(min(length - self.P_local[1], height - self.P_local[0]))
33         flagNewSize = True
34         if dim <= 3:
35             distanza_minima = True
36             new_size = 2*dim + 1 # dimensione della nuova matrice da estrarre
37             productPositionFlag = f.submatrix(positionMatrix/10000.00, self.P_local
38                                         [0].astype(int)-dim, self.P_local[1].astype(int)-dim, new_size)
39
40     elif int(size/2)+1 < self.P_local[0] < height and self.P_local[1] <= int(
41         size/2)+1 :
42         dim = int(min(length - self.P_local[1], self.P_local[0]))
43         flagNewSize = True
44         if dim%2 != 0:
45             dim = dim - 1
46             if dim <= 3:
47                 distanza_minima = True
48                 new_size = 2*dim + 1 # dimensione della nuova matrice da estrarre
49                 productPositionFlag = f.submatrix(positionMatrix/10000.00, self.P_local
                                         [0].astype(int)-dim, self.P_local[1].astype(int)-dim, new_size)
50             else:
51                 productPositionFlag = f.submatrix(positionMatrix/10000.00, self.P_local
                                         [0].astype(int)-int(size/2), self.P_local[1].astype(int)-int(size/2),
                                         size)
52         flagNewSize = False
53         distanza_minima = False

```

Listing 2.17 – Funzione utilizzata per determinare la dimensione della matrice da estrarre dalla *productPositionFlag* al variare della distanza del robot dal bordo della mappa.

In seguito, si valuta il posizionamento di due ulteriori sistemi di riferimento.

Il primo viene posizionato nella cella [0,0] della sub-matrix, con assi x e y posizionati come nel sistema di riferimento locale.

Il secondo ha origine nel centro della cella occupata da Zeno, con assi x e y posizionati nella medesima direzione e verso degli assi della prima terna, utile per definire il punto più lontano dal robot.

Un’ulteriore valutazione sulle matrici è legata alla possibilità di posizionare i waypoint, necessari alla circumnavigazione, alle "spalle" del robot. Ciò potrebbe produrre la situazione in cui il robot farebbe un percorso "avanti e indietro" tra gli stessi waypoint, a ripetizione.

Per evitare tale circostanza, si costruisce un’ulteriore sotto-matrice a partire dalla *productPositionFlag*, chiamata *elementiDaEstrarre* che varia al variare dell’angolo *orientation*, il quale è legato all’angolo di *yaw* del robot, ψ . Esso corrisponde all’angolo che si crea tra l’asse x della terna body, asse che è in direzione del "muso" del robot sul piano di simmetria, e l’asse Nord della terna NED, assumendo valori nell’intervallo $[-\pi, \pi]$.

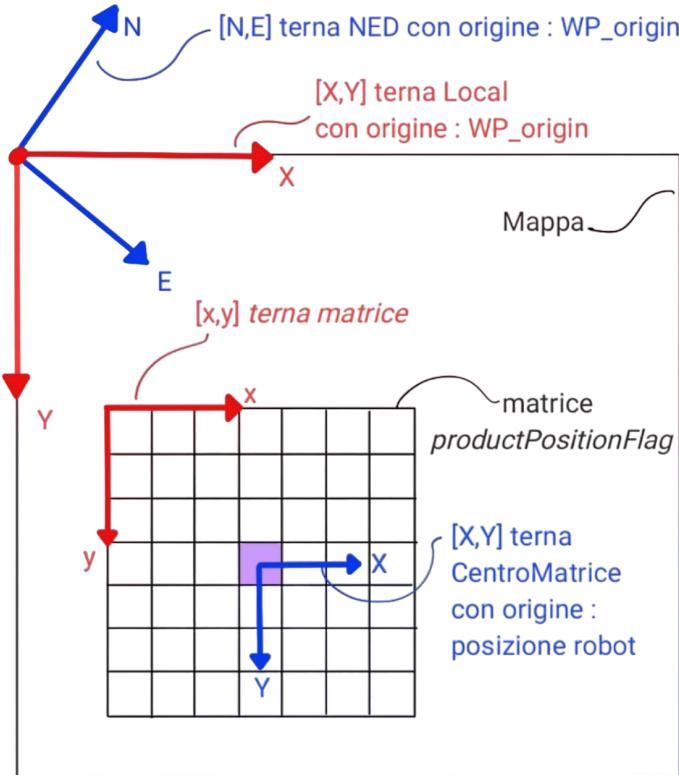


Figura 2.3 – Sistemi di riferimento

In seguito alla costruzione geometrica rappresentata in figura 2.4 è stato possibile costruire la matrice `elementiDaEstrarre` nelle varie configurazioni. Quest'ultima è delimitata dal rettangolo rosso, nella rappresentazione in figura 2.5. Inoltre, si osserva che la cella occupata dal robot in tale matrice dipende dalla casistica in cui ci si trova. Questa informazione si ricava dal vettore `origineEstratta` rappresentata dalla cella di colore viola.

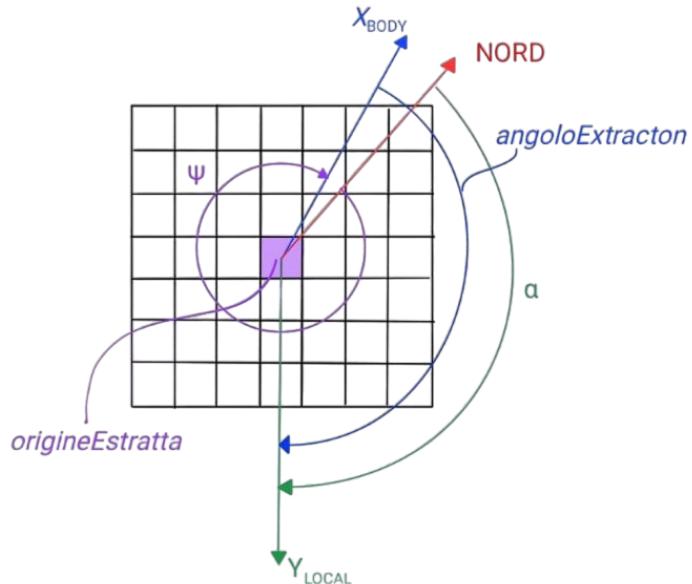


Figura 2.4 – Rappresentazione degli angoli su cui si basa la valutazione sulla matrice `elementiDaEstrarre`.

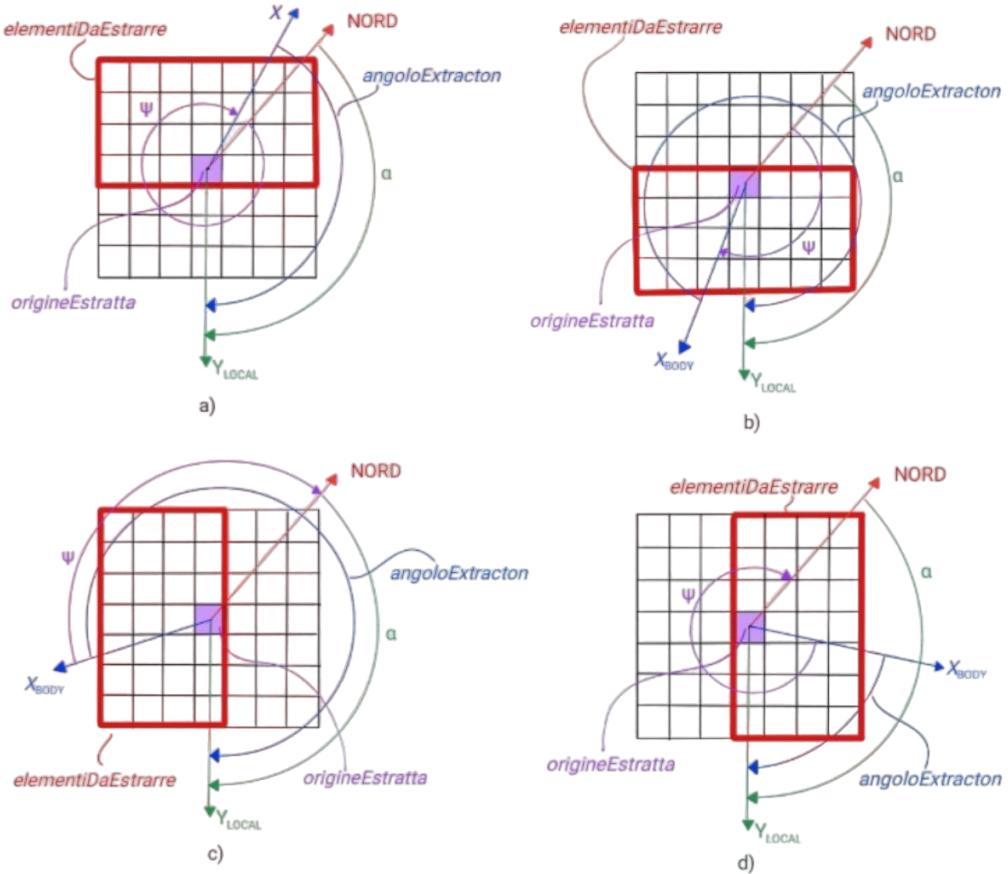


Figura 2.5 – Rappresentazione delle configurazioni possibili della matrice *elementiDaEstrarre* al variare dell’angolo di matrice *orientation*.

L’idea con cui avviene la costruzione dei waypoint, necessari per l’algoritmo di circumnavigazione, ha inizio dalla lettura della matrice *elementiDaEstrarre* e dalla rilevazione del suo valore più grande e minore, o al più uguale, alla *soglia di avvicinamento* o alla *soglia*. Per far ciò, si utilizza la funzione matrice *sogliaMaxDistance*.

Quest’ultima necessita di avere informazioni in ingresso, quali: posizione del robot in terna Local, la matrice *elementiDaEstrarre*, la posizione *origineEstratta* e la matrice *mapFlagSubmatrix*. In primis, si valuta la presenza di valori pari a -1 all’interno della matrice *mapFlagSubmatrix*, per valutare se il robot sia nelle vicinanze del bordo dell’area di lavoro. Se non ci sono valori pari a -1, si determina dalla matrice *elementiDaEstrarre* il valore o i valori più alti, ma minori, al più uguali, della soglia di avvicinamento, ipotizzando di rilevare per prima la *soglia di avvicinamento* e solo successivamente il valore pari alla *soglia* di inquinante da circumnavigare. Essi vengono impilati nel vettore *maxInquinante* e si determina la posizione in cui è stato rilevato, salvando le coordinate matriciali in termini [i,j] all’interno del vettore *indice*.

Nella matrice *elementiDaEstrarre* si cerca il valore, o i valori più elevati, compresi tra il valore della soglia e della soglia più un suo scarto, pari a 0.05. Vengono poi impilati nel vettore *maxInquinante* e si determina la posizione in cui sono stati rilevati, salvando le coordinate matriciali in termini [i,j] all’interno del vettore *indice*. Se ci sono valori pari a -1 all’interno della matrice *mapFlagSubmatrix*, la flag *reverseWaypoint* viene definita True, avviando la produzione dei *reverseWaypoint*. Se la dimensione del vettore *indice* è

maggiori di 1, si valuta la distanza euclidea della cella, che contiene i valori memorizzati nel vettore *indice*, rispetto alla cella *origineEstratta*. Si impilano queste distanze in un vettore *distZeno*: della massima distanza *maxDistance*, si valuta la posizione all'interno di *distZeno*. Sfruttando quest'ultimo valore, definito come *maxDistanceIndex*, si rileva la posizione della cella in cui è stato osservato il valore di inquinante maggiore, definito come *positionMax*. Se la dimensione del vettore *indice* è pari a 1, il valore *positionMax* è univoco.

Definita la matrice di rotazione *Matrice_map2body*, tra *terna matrice* e *terna CentroMatrice*, si determina la cella in cui posizionare il waypoint O_2 rispetto alla posizione del robot, *positionMax_centromatrice*.

```

1 def sogliaMaxDistance(posizioneZenoLocal,elementiDaEstrarre,origineEstratta
2   ,mapFlagSubMatrix):
3   global algoritmo2 # flag necessario per avviare l'algoritmo 2. E
4   inizializzato a False
5   global flagAlg1 # flag per l'algoritmo 1. E inizializzato a True
6   flattenMatrix = elementiDaEstrarre.flatten() # matrice vista come vettore
7   1x(nxn)
8   maxInquinante = [] # lista di massimi valori del prodotto elemeto per
9   elemento delle matrici Mapvirtual_matrix e MapReliable_matrix
10  reverseWaypoint = False # flag necessaria per avviare l'algoritmo dei
11  waypoint reverse. Waypoints necessari nel caso in cui trovo il valore
12  -1 all'interno della matrice MapflagSubMatrix.
13  distZeno = []
14  indice = []
15
16  # se si e in algoritmo 1 o 2:
17  if flagAlg1 or algoritmo2:
18    if not np.any(mapFlagSubMatrix== -1) or np.any(np.isnan(
19      mapFlagSubMatrix)):
20      for i in range(0,elementiDaEstrarre.shape[0]):
21        for j in range(0,elementiDaEstrarre.shape[1]):
22          if np.any(flattenMatrix >= soglia):
23            if (soglia<=elementiDaEstrarre[i,j] < soglia+0.05):
24              maxInquinante.append(elementiDaEstrarre[i,j])
25            # si estrae il massimo indice relativo, cioe si prende il
26            # valore di inquinante pari alla soglia presente nella cella piu lontana
27            # dal robot
28            indice = np.argwhere(soglia<=elementiDaEstrarre[i,j]< soglia
29            +0.05)
30            # settare le 2 flag, in modo da partire con l'algoritmo 2
31            algoritmo2 = True
32            flagAlg1 = False
33
34            # se si trovano elementi minori del valore di soglia da
35            circumnavigare:
36              # se non si trova un valore corrispondente alla soglia, si
37              osserva il massimo che si avvicina al valore di sogliaAvvicinamento.
38              # Il ciclo if serve per la ricerca della cella con valore pari
39              alla soglia di avvicinamento.
40              elif elementiDaEstrarre[i,j] < soglia:
41                if elementiDaEstrarre[i,j]>=sogliaAvvicinamento:
42                  maxInquinante.append(elementiDaEstrarre[i,j])
43                  indice = np.argwhere(elementiDaEstrarre == max(maxInquinante))
44                  algoritmo2 = True
45                  flagAlg1 = False
46                  # fin quando all'interno della matrice 'elementiDaEstrarre' non
47                  si leggono valori maggiori o uguali alla soglia di avvicinamento o
48                  # pari alla soglia da circumnavigare, si continua a navigare
49

```

Capitolo 2 Costruzione della missione

```

con la missione di esplorazione (algoritmo 1), settando a True la flag
'flagAlg1'
    else:
        flagAlg1 = True

    # se nella matrice 'mapFlagSubMatrix' si leggono valori pari a -1 e
    # la flag 'reverseWaypoint' e pari a True, si avvia l'algoritmo di
    produzione
    # dei waypoints reverse, tali da far tornare il robot sui suoi passi
    e continuare la navigazione. Questo avviene perche il valore -1 in tale
    mappa descrive lo scenario
    # in cui siamo vicini al bordo dell'area di lavoro e tale algoritmo
    consente di rimanere all'interno dell'area di lavoro.
else:
    if algoritmo2:
        reverseWaypoint = True

# se si determina un massimo relativo accettabile, l'algoritmo 2
diventa True, e se non siamo in algoritmo 1 e se non siamo nel caso di
reverseWaypoint = True:
    if algoritmo2 and not flagAlg1 and not reverseWaypoint:
        # se vi sono 2 o piu valori identici che sono accettabili, si
        determina il valore piu lontano dal robot e si pone in quella cella il
        waypoint o2
        if len(indice) > 1 :
            for i in indice:
                distance_x = i[0] # estraggo la componente x (riga)
                distance_y = i[1] # estraggo la componente y (colonna)
                d = math.sqrt((distance_x - origineEstratta[0])**2+(distance_y -
origineEstratta[1])**2)
                distZeno.append(d)
            maxDistance = max(distZeno) # calcolo la distanza massima

            maxDistanceIndex = distZeno.index(maxDistance)
            positionMax = indice[maxDistanceIndex]

        else:
            if len(indice[0])<2:
                positionMax = np.array([indice[0],0])
                origineEstratta = np.array([0, 0])
                maxDistance = 0
            else:
                positionMax = np.array(indice[0])
                maxDistance = math.sqrt((positionMax[0]-origineEstratta[0])**2+(positionMax[1]-origineEstratta[1])**2)

            # matrice di rotazione tra terna matrice e terna local (per avere l'
            asse: x == l'asse riga, l'asse y == l'asse colonna)
            Matrice_map2body = np.array([[1, 0],[0, 1]])
            positionMax_centromatrice = np.dot(Matrice_map2body,positionMax)-np.
dot(np.transpose(Matrice_map2body),origineEstratta)
            reverseWaypoint = False

return flagAlg1, maxDistance, positionMax_centromatrice, maxInquinante,
algoritmo2, reverseWaypoint
return flagAlg1, 0, 0 ,0, algoritmo2, reverseWaypoint

```

Listing 2.18 – Funzione utilizzata per determinare il punto di massimo relativo di inquinante più distante dal robot.

2.3.2 Costruzione dei waypoint O_1 e O_2

In questa sezione viene spiegata la costruzione dei waypoint necessari per la circumnavigazione dell'agente inquinante.

Determinato il valore massimo della matrice *elementiDaEstrarre* (sotto la soglia) e la cella corrispondente della griglia fornita, si calcola una coppia di waypoint (O_1 e O_2) che permetterà di compiere la prima iterazione della circumnavigazione.

Il punto O_2 viene posizionato nella cella di valore massimo della matrice *elementiDaEstrarre* e O_1 viene posizionato in modo da essere equidistante dalla posizione di Zeno e O_2 . Inoltre si definisce O_0 come la posizione di Zeno all'inizio del passo dell'algoritmo.

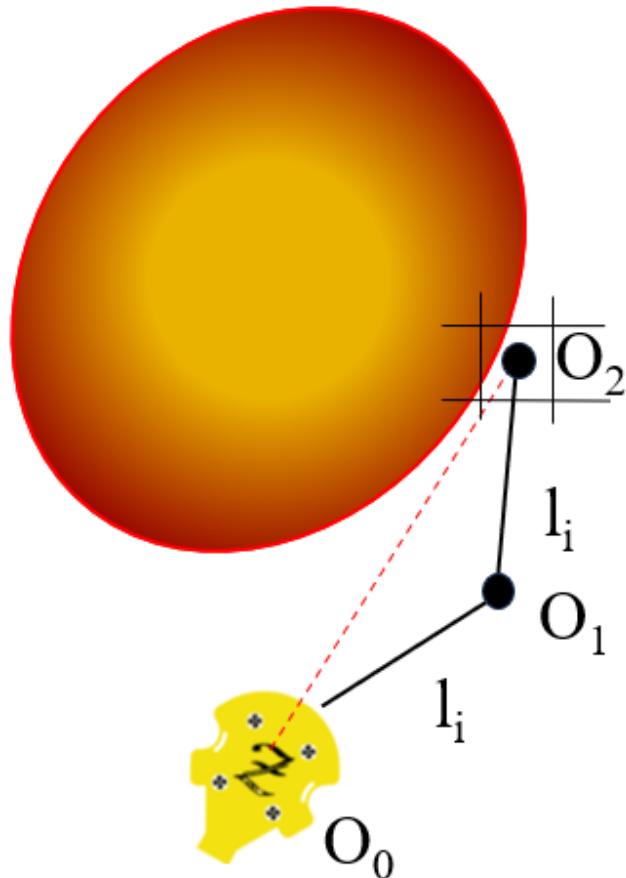


Figura 2.6 – Costruzione waypoint O_1 e O_2 .

Questa costruzione garantisce di trovare una retta di punti O_1 , perciò imponiamo dei vincoli affinché la ricerca porti alla traiettoria richiesta:

- valore prodotto inquinante per affidabilità di O_1 minore di quello di O_2 ;
- angolo ϑ compreso tra i 120 gradi e 150 gradi;
- distanza l maggiore di 1 metro.

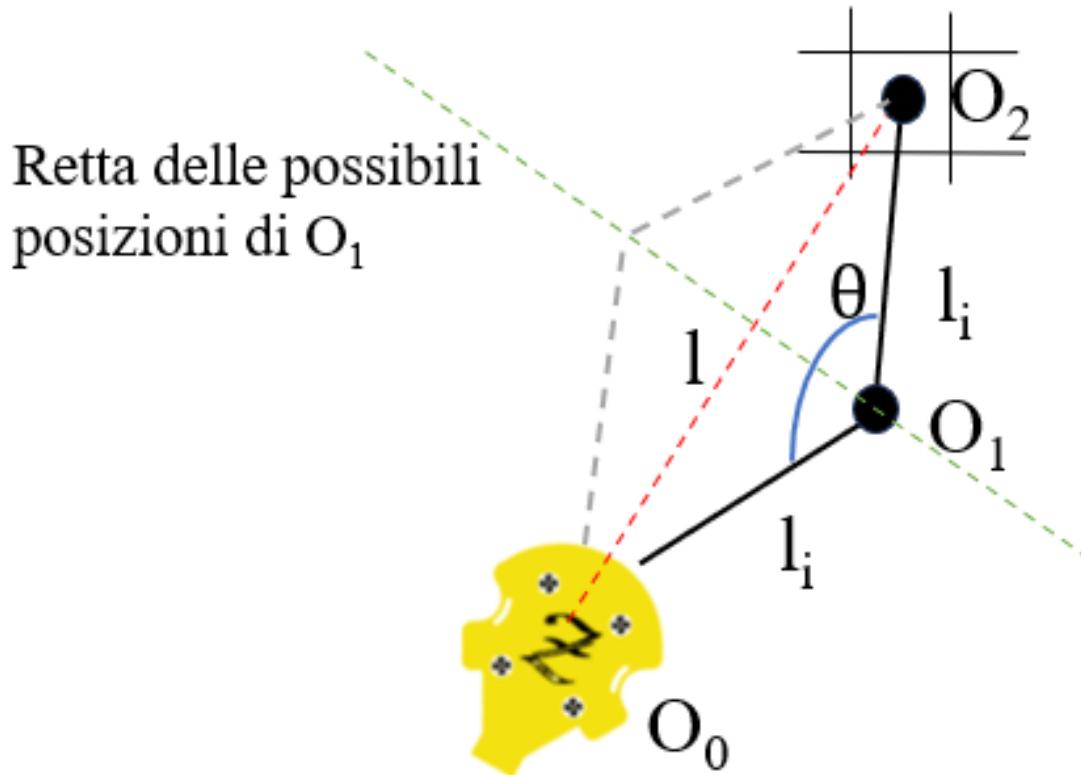


Figura 2.7 – Posizioni possibili del waypoint O_1 .

Per semplicità di trattazione definiamo due casistiche, ovvero "gomito alto" se O_1 è sopra la congiungente tra posizione Zeno e O_2 e "gomito basso" se O_1 è sotto la congiungente tra posizione Zeno e O_2 .

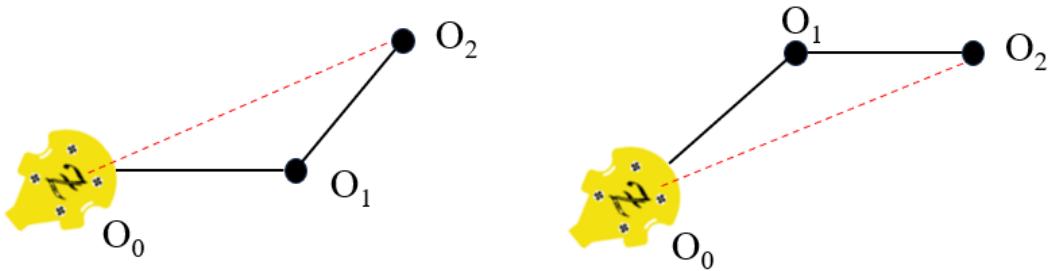


Figura 2.8 – Configurazioni "gomito basso" (sinistra) e "gomito alto" (destra).

La circumnavigazione della macchia avviene ripetendo la costruzione della coppia di waypoint in modo che il punto O_2 del passo precedente diventi la posizione iniziale di Zeno del passo successivo. La navigazione finisce quando il waypoint O_2 dell'i-esimo passo è nell'intorno del punto di inizio dell'algoritmo.

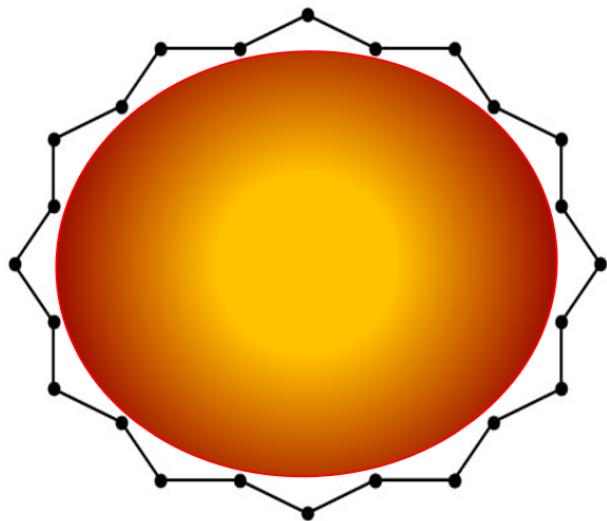


Figura 2.9 – Traiettoria circumnavigazione.

La funzione *planning_circumnavigazione* permette la costruzione della lista dei waypoint dando come ingresso *P_local*, *maxDistance*, *positionMax_centromatrice*, *elementiDaEstrarre*, *maxInquinante* e *origineEstratta*. Si definiscono e inizializzano delle variabili globali e locali necessarie per il calcolo.

```

1 def planning_circumnavigazione(P_local, maxDistance,
2     positionMaxCentromatrice, elementiDaEstrarre, maxInquinante,
3     origineEstratta):
4
5     # vettore che impila i valori del prodotto inquinante*affidabilita nella
6     # cella in cui si posiziona o1 nella configurazione a 'gomito alto'
7     vect_inquinante_alto = []
8
9     # vettore che impila i valori del prodotto inquinante*affidabilita nella
10    # cella in cui si posiziona o1 nella configurazione a 'gomito bassa'
11    vect_inquinante_basso = []
12
13    o1_basso = [] # configurazione a 'gomito basso' al variare di theta
14    o1_alto = [] # configurazione a 'gomito alto' al variare di theta
15
16    o1_basso_centrato = [] # configurazione a 'gomito basso' al variare di
17    # theta rispetto al centro della matrice elementiDaEstrarre
18    o1_alto_centrato = [] # configurazione a 'gomito alto' al variare di theta
19    # rispetto al centro della matrice elementiDaEstrarre
20
21    vect_o1_alto = [] # lista che raccoglie la cella nella matrice
22    # elementiDaEstrarre in cui si trova il waypoint o1 nel caso 'gomito alto',
23    #
24    vect_o1_basso = [] # lista che raccoglie la cella nella matrice
25    # elementiDaEstrarre in cui si trova il waypoint o1 nel caso 'gomito
26    # basso'
27
28    k = 0 # indice in cui ho il massimo valore della elementiDaEstrarre
29    theta_i = 120 # inizializzazione valore di theta tra 120 e 150 gradi
30    beta_i = 0 # inizializzazione angolo beta
31    theta_x = 0 # inizializzazione valore di theta ottimale
32    beta_x = 0 # inizializzazione valore di beta ottimale
33    theta_alto = 0 # inizializzazione valore di theta nel caso 'gomito alto'
34    theta_basso = 0 # inizializzazione valore di theta nel caso 'gomito basso'
35
36
```

Listing 2.19 – Definizione e inizializzazioni delle variabili.

Capitolo 2 Costruzione della missione

Definite le variabili possiamo trovare la posizione di O_2 nella cella a valore massimo della matrice *elementiDaEstrarre*, sotto il valore di soglia da circumnavigare (espresso in terna *Local*). La posizione viene costruita come posizione di Zeno sommata alla distanza relativa tra il robot e la cella considerata. Questo è possibile se non è attivo l'*if* relativo alla costruzione dei *reverseWaypoint2.3.4*.

```

1 if not reverse:
2   o2 = list(np.array([P_local[0]+positionMax_centromatrice[0], P_local[1]+
3     positionMax_centromatrice[1]]))

```

Listing 2.20 – Posizione di O_2 in terna Local.

In aggiunta si definisce il parametro α , che è l'angolo fra il vettore posizione robot-waypoint O_2 e l'asse x della terna *Local*, necessario per la generazione di O_1 .

```

1 alpha = math.atan2(positionMax_centromatrice[1], positionMax_centromatrice
[0])

```

Listing 2.21 – Definizione di α

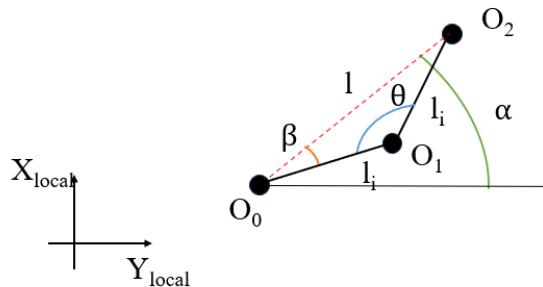


Figura 2.10 – Grandezze geometriche per il calcolo dei waypoint.

Il punto O_1 viene posizionato in modo da essere nella cella a valore più elevato possibile della matrice *elementiDaEstrarre*, ma minore dell'inquinante di O_2 per evitare che una configurazione ("gomito basso" o "gomito alto") faccia passare il robot nella macchia. Inoltre per garantire una traiettoria più "smooth" si sceglie l'angolo ϑ più grande possibile fra i 120° e i 150° da passare nella casella individuata precedentemente.

Si costruisce un ciclo *for* per scandagliare tutti possibili valori di ϑ , con passo 5 gradi, e costruire le grandezze geometriche necessarie, raccolte in liste, per la scelta della posizione migliore di O_1 .

```

1 # angolo fra i vettori posizione robot-o1 e o1-o2 che varia da 120 a 150
2   gradi con passo 5 gradi
3   for theta_gradi in range(120,155,5):
4
5     theta_i = theta_gradi*math.pi/180
6
7     beta_i = 0.5*math.pi-0.5*theta_i
8
9     # modulo dei vettori posizione robot-o1 = o1-o2
10    l_i = maxDistance/(2*math.sin(0.5*theta_i))

```

Listing 2.22 – Costruzione parametri geometrici al variare di ϑ .

Per le due casistiche possibili si hanno le componenti del punto in terza *Local* appoggian-
dosi alla posizione di Zeno per la costruzione.

```

1 # 'gomito basso' singola configurazione
2 o1_basso_i = list(np.array([P_local[0]+l_i*math.cos(alpha-beta_i),
3 P_local[1]+l_i*math.sin(alpha-beta_i)]))
4 o1_basso.append(o1_basso_i)
5
6 # 'gomito alto' singola
7 o1_alto_i = list(np.array([P_local[0]+l_i*math.cos(alpha+beta_i),
8 P_local[1]+l_i*math.sin(alpha+beta_i)]))
9 o1_alto.append(o1_alto_i)

```

Listing 2.23 – Posizione di O_1 per "gomito basso" e "gomito alto".

Adesso si cerca la cella in cui si trova il waypoint per poter scegliere quale casistica e angolo ϑ sono ottimali. Per fare ciò si esprime $o1_basso_i$ e $o1_alto_i$ rispetto a due sistemi di riferimento quali *terna centro matrice* e *terna matrice* rappresentati in figura 2.3.

Scegliendo celle della matrice *elementiDaEstrarre* di dimensioni 1 metro x 1 metro si ottiene direttamente la cella di O_1 nota la sua posizione in *terna matrice*.

```

1 o1_basso_centrato_i=o1_basso_i-P_local[0:2] # componenti in terna centro
2   matrice
3 o1_basso_centrato.append(o1_basso_centrato_i)
4 o1_alto_centrato_i=o1_alto_i-P_local[0:2] # componenti in terna centro
5   matrice
6 o1_alto_centrato.append(o1_alto_centrato_i)

# matrice di rotazione fra terna centrata e terna matrice
elementiDaEstrarre

7 Matrice_map2body = np.array([[1, 0],[0, 1]])
8 o1_basso_ternamatrix = list(np.dot(Matrice_map2body,np.transpose(
9   o1_basso_centrato_i))+origineEstratta)

10 o1_alto_ternamatrix = list(np.dot(Matrice_map2body,np.transpose(
11   o1_alto_centrato_i))+origineEstratta)

12 # Si raccoglie le celle poi nelle seguenti liste:
13 vect_o1_basso.append(o1_basso_ternamatrix)
14 vect_o1_alto.append(o1_alto_ternamatrix)
15

```

Listing 2.24 – Posizione di O_1 in terna matrice.

Per la configurazione a "gomito alto" si estraе il valore della cella della matrice *elementi-DaEstrarre* in cui si pone O_1 al variare di ϑ . Il valore viene accettato se sono rispettate le condizioni precedentemente indicate, ma se ciò non accadesse si pone pari a -1 e si salva per avere una corrispondenza biunivoca con l'indice della lista *vect_inquinante_alto*. Infine si valuta il massimo valore estratto *max_inq_o1_alto* e la cella in cui si trova *index_max_inq_o1_alto* per fornire il ϑ ottimale (*theta_alto*). Quest'ultimo si ottiene definendo una variazione di angolo, come il prodotto dell' ultimo valore di *index_max_inq_o1_alto* per il passo (5°), da sommare a 120° .

```
1 # estrazione valore della matrice elementiDaEstrarre
2 for i in vect_o1_alto:
3     if elementiDaEstrarre.shape[0] < 2:
4         inquinante_alto = elementiDaEstrarre.tolist()[0]
5     else:
```

Capitolo 2 Costruzione della missione

```

6     inquinante_alto = elementiDaEstrarre.tolist()[int(i[0]),int(i[1])]
7 # maxInquinante e il vettore dei possibili valori associati alla matrice
8     elementiDaEstrarre in cui posizionare il waypoint o2
9
10    if inquinante_alto <= max(maxInquinante):
11        vect_inquinante_alto.append(inquinante_alto[0])
12    else:
13        vect_inquinante_alto.append(-1)
14
15    max_inq_o1_alto = max(vect_inquinante_alto)
16    vect_inquinante_alto = np.array(vect_inquinante_alto)
17    index_max_inq_o1_alto = np.where(vect_inquinante_alto == max_inq_o1_alto)
18        [0]
19
20    # calcolo del theta ottimale per la configurazione o1_alto.
21    if len(index_max_inq_o1_alto) > 1:
22        k = index_max_inq_o1_alto[len(index_max_inq_o1_alto)-1]
23    else:
24        k=index_max_inq_o1_alto
25
26    # theta ottimale:
27    theta_alto=((120+5*k)*math.pi)/180

```

Listing 2.25 – Calcolo del theta_alto.

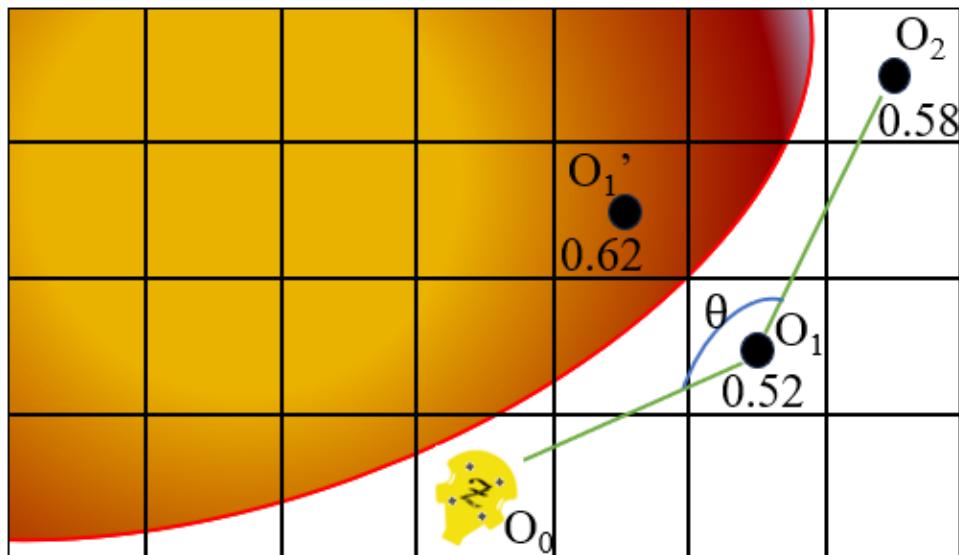


Figura 2.11 – Posizione di O_1 .

Analogamente si valuta per la casistica "gomito basso" il valore ottimale *theta_basso*. Il codice è lo stesso di quello già citato, ma con pedice "basso" per le variabili.

Infine vi è un confronto finale tra `max_inq_o1_alto` e `max_inq_o1_basso` e tra i relativi `theta_alto` e `theta_basso` (scegliendo sempre i valori maggiori) per definire la posizione di O_1 in terna *Local*. L'algoritmo si conclude con la generazione della lista dei waypoint `waypoint_list_circumnavigazione`, che vengono pubblicati sulla topic `zeno/mm/upload_mission` come per l'algoritmo di esplorazione.

```

1 if max_inq_o1_alto > max_inq_o1_basso:
2     theta_x = theta_alto
3     beta_x = 0.5*math.pi-0.5*theta_x
4     l_i = maxDistance/(2*math.sin(0.5*theta_x))
5     o1=list(np.array([P_local[0]+l_i*math.cos(alpha+beta_x),P_local[1]+l_i*
6         math.sin(alpha+beta_x),0]))
7
8     elif max_inq_o1_basso > max_inq_o1_alto:
9         theta_x = theta_basso
10        beta_x = 0.5*math.pi-0.5*theta_x
11        l_i = maxDistance/(2*math.sin(0.5*theta_x))
12        o1=list(np.array([P_local[0]+l_i*math.cos(alpha-beta),P_local[1]+l_i*math.
13            .sin(alpha-beta),0]))
14
15    elif max_inq_o1_alto == max_inq_o1_basso:
16        theta_x = max(theta_alto,theta_basso)
17        beta_x = 0.5*math.pi-0.5*theta_x
18        l_i = maxDistance/(2*math.sin(0.5*theta_x))
19        if theta_x == theta_alto:
20            o1 = list(np.array([P_local[0]+l_i*math.cos(alpha+beta_x),P_local[1]+l_i*
21                math.sin(alpha+beta_x),0]))
22
23 # lista dei waypoint per la missione circumnavigazione della macchia
24 waypoint_list_circumnavigazione=[o1,o2 ]
25
26 else:
27     waypoint_list_circumnavigazione = []
28
29 return waypoint_list_circumnavigazione

```

Listing 2.26 – Posizione di O_1 .

Per la pubblicazione dei waypoint si segue il procedimento riportato:

- si verifica se l'algoritmo 2 è attivo e non sono attivi l'algoritmo di ritorno o di costruzione dei *reverse waypoint* (*if flagAlg2 and not self.algoritmoRitorno and not reverseWaypoint:*);
- si verifica se lo stato di Zeno è *RUNNING*;
- pubblicazione di un messaggio *Empty* per passare da *RUNNING* a *PAUSED* sulla *topic zeno/mm/paused_mission*;
- cancellazione della missione precedente sulla *topic zeno/mm/delete_mission*;
- pubblicazione della nuova lista di waypoint sulla *topic zeno/mm/upload_mission*;
- pubblicazione di un messaggio *Empty* per avviare la missione sulla *topic zeno/mm/-start_mission*.

Per completare la missione si itera questo processo di costruzione ogni volta che il robot raggiunge il waypoint O_2 .

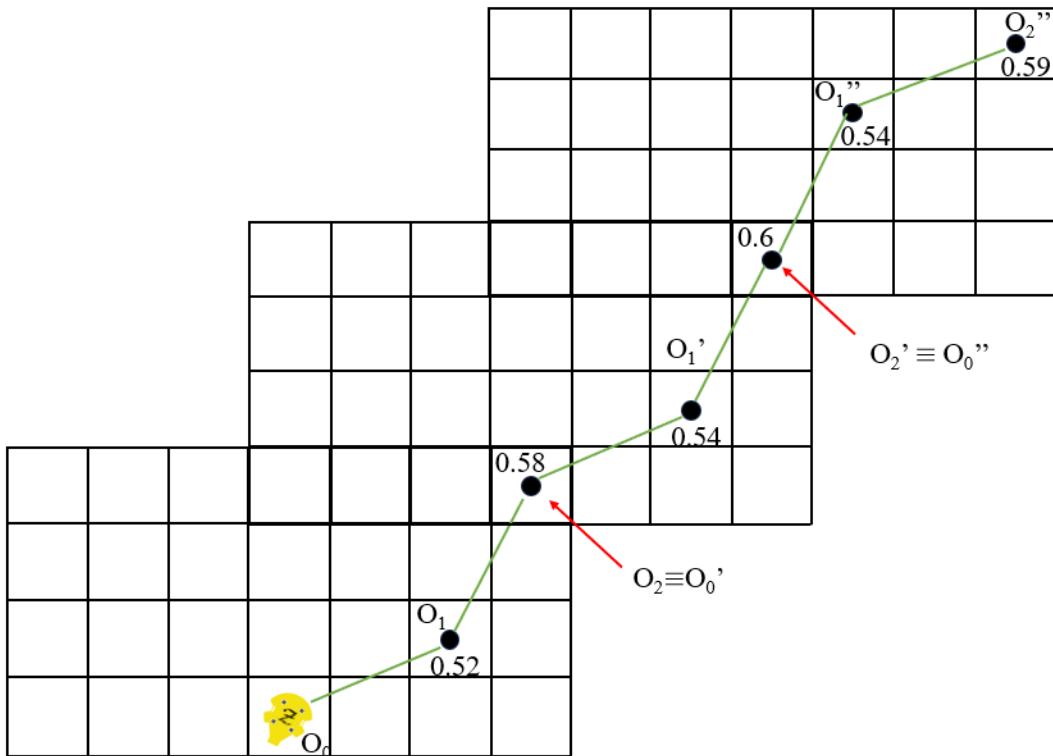


Figura 2.12 – Esempio di traiettoria per la circumnavigazione.

2.3.3 Conclusione Circumnavigazione

Questa parte di algoritmo si occupa di concludere la navigazione, basandosi su due controlli:

- verifica sull'angolo di *Heading*;
- valutazione della distanza di Zeno dal punto di inizio della circumnavigazione.

Queste verifiche implicano che al passo ennesimo il robot deve essere in un intorno della posizione e dell'orientazione da cui è partito. Per prima cosa si verifica con un *if* che il robot stia eseguendo l'algoritmo 2 (*algoritmo2 = True*) e non stia facendo la traiettoria di ritorno o la costruzione dei *reverseWaypoint* 2.3.4.

Dopo di che si va a utilizzare la variabile *o0*, nella quale è stata salvata la posizione di Zeno nell'istante in cui inizia la traiettoria della circumnavigazione.

```

1 if algoritmo2 and P_localFlag and not self.reverseWaypoint:
2     P_local100 = self.P_local
3     o0 = P_local100
4     P_localFlag = False

```

Listing 2.27 – Punto di inizio della Circumnavigazione.

Le funzioni utilizzate per il controllo sono *headingFunction* e *distanceFinal*.

La prima permette di verificare se dopo un certo numero di iterazioni l'angolo di *yaw* ha avuto un incremento di 360° e quindi il robot assume la stessa orientazione iniziale.

L'aumento ad ogni passo viene valutato come la differenza di orientazione fra quando Zeno è diretto verso O_2 e quando è diretto verso O_1 ; dopo di ché si salva nella variabile

2.3 Algoritmo 2: circumnavigazione

diffHeading al fine di sommare le componenti per avere l'incremento totale. Se quest'ultimo sarà maggiore di 350° o minore di 370° , allora sarà verificata la condizione sull'orientazione. La seconda funzione, inoltre, verifica che il punto in cui ho corretta orientazione sia a distanza minore di 3 metri dal punto iniziale dell'algoritmo. Questo permette di avere la posizione corretta e di evitare problematiche nel caso in cui la macchia avesse una forma concava e il robot si trovasse distante dal punto iniziale.

```

1 def headingFunction(headingVector):
2     diffHeading = headingVector[len(headingVector)]-headingVector[0]
3     if abs(diffHeading) > (350*math.pi)/180 and abs(diffHeading) < (370*math.
4         pi)/180:
5         headingTotFlag = True
6     else:
7         headingTotFlag = False
8     return headingTotFlag
9
10 def distanceFinal(P_local0, P_localCurrent):
11     distanceFinal = math.sqrt(([P_localCurrent[0]]-P_local0[0])**2 + (
12         P_localCurrent[1]-P_local0[1])**2)
13     if distanceFinal < 3:
14         distanceFinalFlag = True
15     else:
16         distanceFinalFlag = False
17     return distanceFinal,distanceFinalFlags

```

Listing 2.28 – Funzioni per concludere la circumnavigazione.

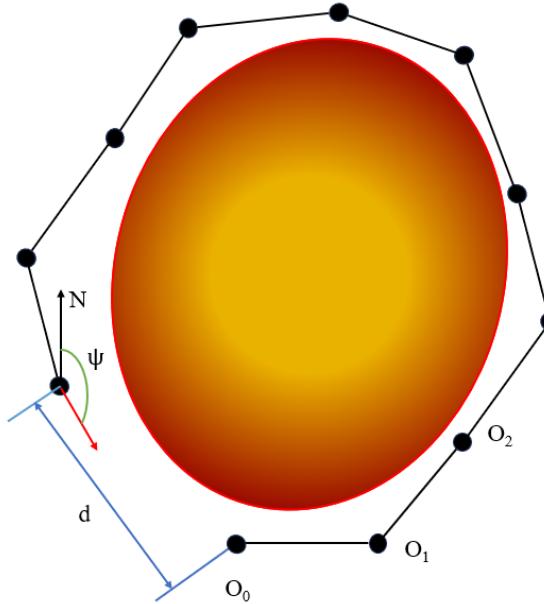


Figura 2.13 – Rappresentazione dell'angolo di *Heading* e *distanceFinal* necessari per concludere la missione.

Per la valutazione dell'angolo di *Heading* è necessario valutare la distanza euclidea dai waypoint di circumnavigazione con la funzione *normWaypoint* per capire se Zeno è diretto verso O_1 o verso O_2 . Se la distanza dal primo punto è minore di 1.5 metri si acquisisce dal *navStatus* l'angolo di *yaw* corrispondente e viene settata a *False* la variabile *O1flag* (per indicare che è stata calcolata per la prima volta la distanza), invece se sono a distanza

Capitolo 2 Costruzione della missione

minore di 1.5 metri dal secondo analogamente si salva l'orientazione fornita. Dopo di che i dati acquisiti vengono salvati nella variabile *orientationVectorWaypoint* che sarà l'input della funzione *headingFunction*.

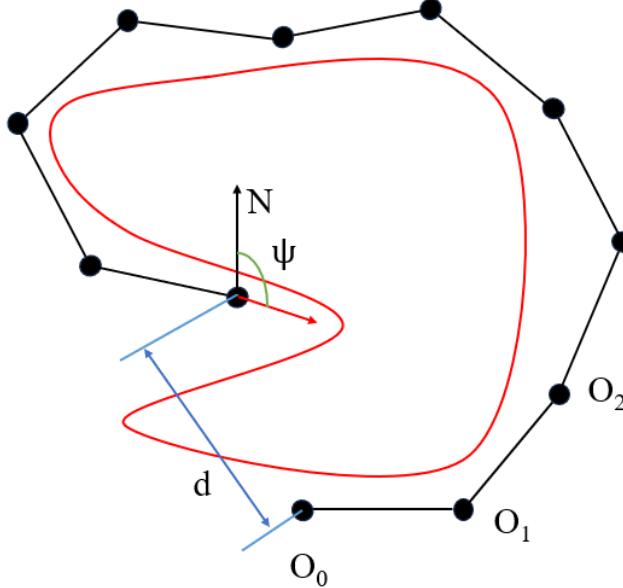


Figura 2.14 – Esempio di chiusura della missione con una macchia di inquinante concava.

Dopo il controllo sull'angolo si verifica la distanza da *P_localO0* per controllare se sono in un intorno della posizione iniziale.

```

1 elif not flagAlg2 and not reverseWaypoint:
2     normDistance = f.normWaypoint(self.waypointCircumnavigazione ,self.
3         .P_local)
4     o0list.append(o0)
5
6     if normDistance [1]<1.5:
7         o0_new = self.P_local
8         o0list.append(o0_new)
9         # angolo di orientazione del robot quando si trova nei pressi del
10        # waypoint2 (o2)
11        orientation02 = (navStatus_msg.orientation.yaw*180)/math.pi
12        if orientation02 < 0 :
13            orientation02 = 360 + orientation00
14
15        # lista che ingloba tutte le orientazioni rispetto al waypoint o2
16        orientationVectorWaypoint.append(orientation02)
17
18        # funzione per il calcolo dell'angolo di heading: restituira un      True
19        # se e stata completata la circumnavigazione.
20        headingFlag = f.headingFunction(orientationVectorWaypoint)
21        # funzione che restituisce la distanza mancante alla fine della
22        # circumnavigazione e il rispettivo flag per capire se e stata
23        # completata la missione.
24        distanceFinal,distanceFinalFlag = f.distanceFinal(P_local00 ,self.P_local)
25        print("FinalDistance , {}.".format(distanceFinal))
26
27        # verifica delle due condizioni (di fine circumnavigazione) e se      sono
28        # verificate viene eseguito l'algoritmo per il ritorno all'origine.
29        if headingFlag and distanceFinalFlag:
30            self.algoritmoRitorno = True

```

```

25
26 # viene verificata la condizione sulla minima distanza
27 if distanza_minima:
28     flagAlg2 = True
29     flagPaused = True
30     flagReady = True
31     flagRunning = True
32     flagIdle = True
33 else:
34     flagAlg2 = True
35     flagPaused = True
36     flagReady = True
37     flagRunning = True
38     flagIdle = True
39     O1flag = True
40
41 # condizione posta per verificare se la distanza a cui si trova il robot
42     rispetto al punto o1 è minore di 1.5 metri
43 elif normDistance[0]<1.5 and O1flag:
44     if flagO1first:
45         o0 = o0list[0]
46         flagO1first = False
47     else:
48         o0list.pop(0)
49         o0 = o0list[0]
50     # angolo di orientazione del robot quando si trova nei pressi del
51     # waypoint1 (o1)
52     orientation00 = (navStatus_msg.orientation.yaw*180)/(math.pi)
53     if orientation00 < 0 :
54         orientation00 = 360 + orientation00
55
56     # lista che ingloba tutte le orientazioni rispetto al waypoint o1
57     orientationVectorWaypoint.append(orientation00)
58     O1flag = False

```

Listing 2.29 – Traiettoria di ritorno.

2.3.4 Costruzione dei "reverse waypoint"

In questa sezione si indica il calcolo dei *reverse waypoint* necessari quando il robot entra dentro la macchia o quest'ultima si sviluppa nelle vicinanze del bordo dell'area di lavoro.

La costruzione fatta nelle sezioni precedenti implica che l'agente inquinante si sviluppi lontana dal perimetro dell'area di lavoro e che i valori forniti dalle mappe siano il risultato di una predizione. Nello studio presentato l'inquinante però, può posizionarsi in qualsiasi punto della zona di interesse e i dati derivano solamente da una fase di correzione. Per risolvere questi vincoli si generano dei *reverse waypoint*, ovvero dei waypoint che permettono a Zeno di tornare indietro mentre sta facendo la circumnavigazione.

Se il robot mentre sta costeggiando la soglia di inquinante vi entra all'interno, la missione viene fermata e si carica la lista dei *reverse waypoint* per farlo stare esterno alla macchia facendolo poi riprendere la traiettoria precedente. Questo scenario si presenta se la matrice *elementiDaEstrarre* ha solo valori superiori a 0.6 ± 0.05 . Analogamente se la zona da perlustrare è a contatto con un lato dell'area di lavoro, l'AUV non continua con il suo cammino, ma torna indietro evitando di arrivare in una zona non necessaria da scandagliare (rilevabile se almeno un valore della *mapFlagSubMAtrix* vale -1 o vale NaN).

Capitolo 2 Costruzione della missione

A livello geometrico i *reverse waypoint* sono O_0 e O_1 calcolati al passo i-esimo dall'algoritmo 2 e raccolti nella variabile w_{reverse} . Se l'inquinante si trova vicino al perimetro della zona di lavoro la sequenza sarà $[O_1, O_0]$, invece per l'altro caso sarà $[O_0, O_1]$ affinché Zeno sia nuovamente orientato verso la macchia.

```

1 # Costruzione dei reverse waypoint se Zeno entra nella macchia
2 if flagIdle and algoritmo2 and not distanza_minima and len(self.
   waypointCircumnavigazione):
3     self.w_trajectory = self.waypointCircumnavigazione
4     w_reverse = [o0, self.w_trajectory[0]]
5     wayList = f.waypointsList(self.w_trajectory, self.C_lo2ne) # ce li
       restituisce in terna NED
6     self.pubUploadMission.publish(wayList) # vengono caricati i waypoint
7     wayList = [] # viene svuotata la lista in modo che i nuovi waypoint non
       si aggancino ai precedenti
8     flagIdle = False
9
10    # Costruzione dei reverse waypoint se Zeno e' vicino al perimetro dell'
       area di lavoro
11 elif flagIdle and distanza_minima and len(self.waypointCircumnavigazione)
12   :
13   self.w_trajectory = self.waypointCircumnavigazione
14   w_reverse = [self.w_trajectory[0], o0]
15   wayList = f.waypointsList(w_reverse, self.C_lo2ne) # ce li restituisce in
       terna NED
16   self.pubUploadMission.publish(wayList) # vengono caricati i waypoint
17   wayList = [] # viene svuotata la lista in modo che i nuovi waypoint non
       si aggancino ai precedenti
18   flagIdle = False

```

Listing 2.30 – Costruzione dei *reverse waypoint*.

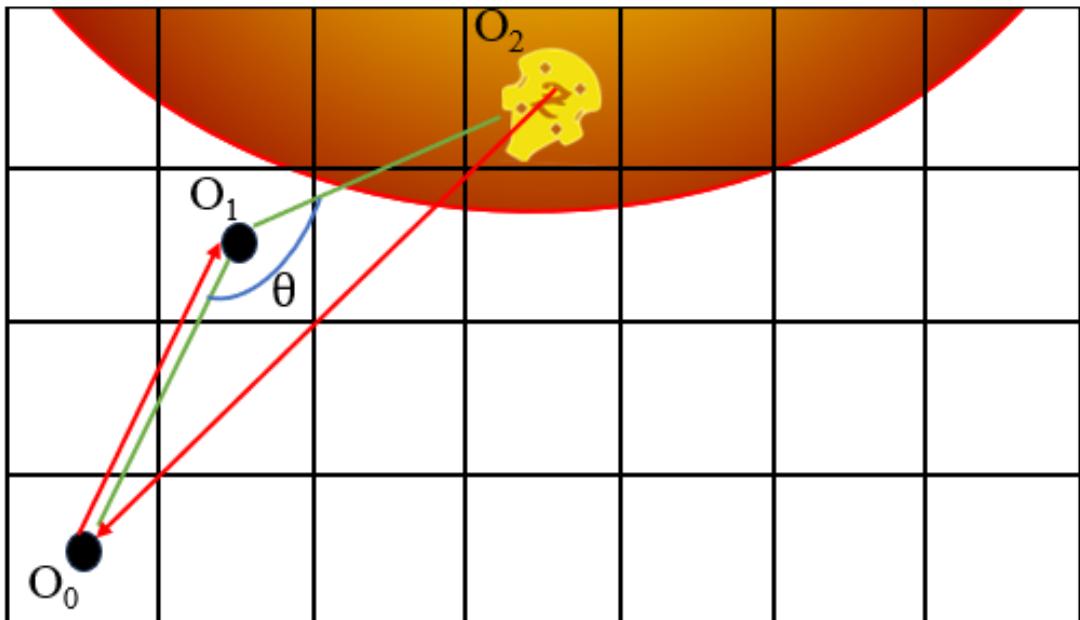


Figura 2.15 – Traiettoria dei *Reverse waypoint* quando Zeno è entrato nell'inquinante.

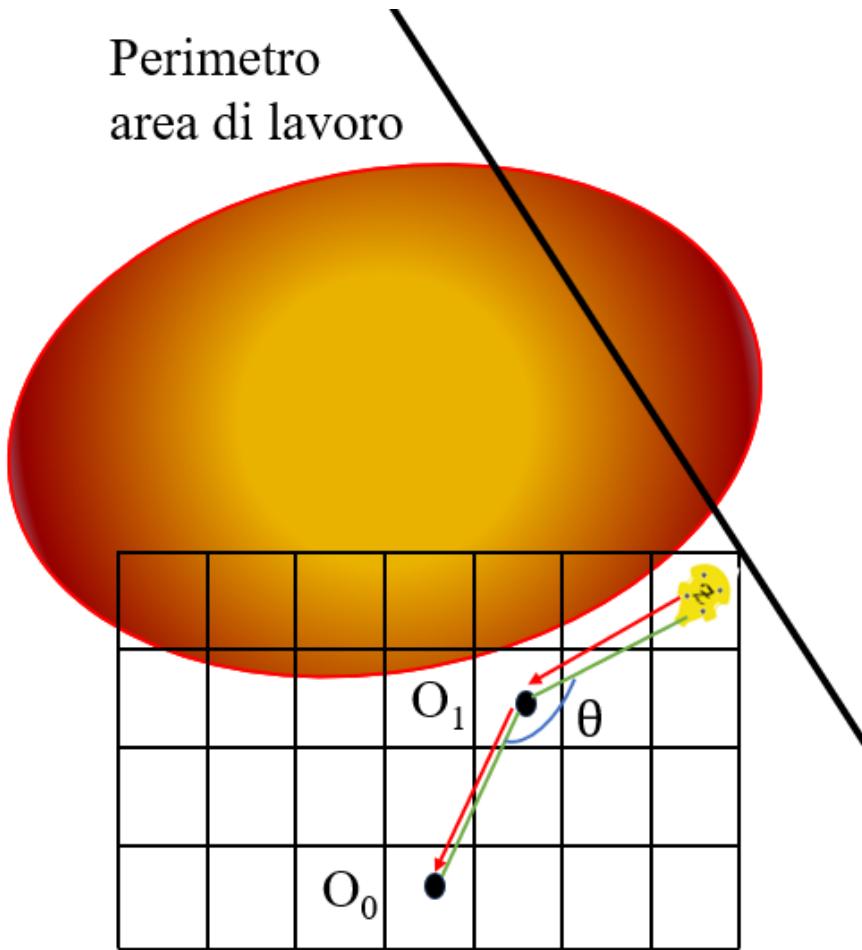


Figura 2.16 – Traiettoria dei *Reverse waypoint* quando l'inquinante è a contatto con il perimetro dell'area di lavoro.

Si riporta successivamente lo schema di pubblicazione dei *reverse waypoint*:

```

1 # viene verificata la condizione sullo stato di zeno e sull'algoritmo di
2     reverse per capire se è attivo
3 if self.reverseWaypoint and self.statusZeno == "RUNNING" :
4     self.pubPauseMission.publish(self.Mission) # viene messa in pausa la
5         missione
6
7 elif self.reverseWaypoint and (self.statusZeno == "PAUSED" or self.
8     statusZeno == "COMPLETED"):
9     wayList = f.waypointsList(w_reverse, self.C_lo2ne) # definizione della
10        lista di waypoint
11 self.pubDeleteMission.publish(self.Mission) # viene eliminata la missione
12        precedente
13
14 elif self.reverseWaypoint and self.statusZeno == "IDLE":
15     self.pubUploadMission.publish(wayList) # vengono caricati i waypoint
16     wayList = [] # viene svuotata la lista dei waypoint
17
18 elif self.reverseWaypoint and self.statusZeno == "READY":
19     self.pubStartMission.publish(self.Mission) # vengono pubblicati i
20         waypoint
21 self.reverseWaypoint = False

```

Listing 2.31 – Pubblicazione dei *reverse waypoint*.

2.3.5 Costruzione dell'algoritmo di ritorno

In questa sezione viene delineata la traiettoria da fornire all'AUV per essere recuperato o iniziare una nuova esplorazione.

Una volta conclusa la missione, il robot viene indirizzato verso il punto di raccolta posto nel vertice *WP0*. Tale punto è stato scelto in modo da portare il robot il più vicino possibile alla postazione di lavoro.

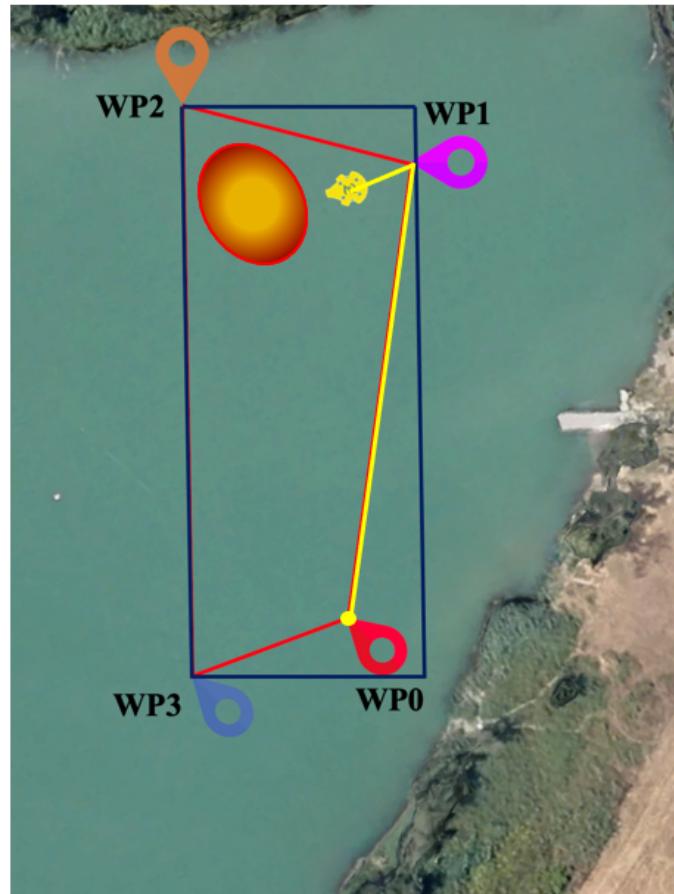


Figura 2.17 – Traiettoria di ritorno (in giallo).

Facendo un controllo sullo stato, si verifica la fine della circumnavigazione e si impone la nuova traiettoria al robot (solo se la variabile *self.algoritmoRitorno* è True).

Per prima cosa si valuta il vertice dell'area di lavoro più vicino alla posizione di Zeno e poi, in base a quello trovato, si impone una lista di waypoint con la funzione *switchfinal* che valuta ogni casistica. La lista ottenuta sarà composta da qualche vertice della zona di interesse in modo da far muovere l'AUV sul perimetro.

```

1 if self.algoritmoRitorno and self.statusZeno == "RUNNING" :
2     self.pubPauseMission.publish(self.Mission)
3
4 elif self.algoritmoRitorno and (self.statusZeno == "PAUSED" or
5     self.statusZeno == "COMPLETED"):
6     minIndexfinal = f.distance(w_localTotal[0:4], self.P_local)
7     # estrazione del waypoint per il ritorno (vertice più vicino)
8     wayfinal = w_localTotal[minIndexfinal]
9     # funzione che determina la traiettoria per raggiungere l'origine quando
10    si conclude la circumnavigazione

```

2.3 Algoritmo 2: circumnavigazione

```

10    self.w_trajectory = f.switchfinal(wayfinal, w_localTotal)
11    wayList = f.waypointsList(self.w_trajectory, self.C_lo2ne)
12    self.pubDeleteMission.publish(self.Mission)
13
14    elif self.algoritmoRitorno and self.statusZeno == "IDLE":
15        self.pubUploadMission.publish(wayList) # vengono caricati i waypoint
16        wayList = [] # viene svuotata la lista dei waypoint
17
18    # viene verificata la condizione sullo stato di zeno e sull'algoritmo di
19    # ritorno per capire se è attivo
20    elif self.algoritmoRitorno and self.statusZeno == "READY":
21        self.pubStartMission.publish(self.Mission)
22        self.algoritmoRitorno = False

```

Listing 2.32 – Traiettoria di ritorno.

Si riportano i possibili scenari della traiettoria di ritorno che può fare Zeno.

```

1 def switchfinal(wayfinal, w_localTotal):
2     if np.all(wayfinal == w_localTotal[0]): # WPO piu vicino
3         w_trajectoryfinal = [w_localTotal[0]]
4
5     elif np.all(wayfinal == w_localTotal[1]): # WP1 piu vicino
6         w_trajectoryfinal = [w_localTotal[1], w_localTotal[0]]
7
8     elif np.all(wayfinal == w_localTotal[2]): # WP2 piu vicino
9         w_trajectoryfinal = [w_localTotal[2], w_localTotal[3], w_localTotal[0]]
10
11    elif np.all(wayfinal == w_localTotal[3]): # WP3 piu vicino
12        w_trajectoryfinal = [w_localTotal[3], w_localTotal[0]]
13    return np.array(w_trajectoryfinal)

```

Listing 2.33 – Definizione possibili traiettorie con la funzione *switchfinal*.

Capitolo 3

Validazione algoritmi

In questo capitolo si approfondiscono gli aspetti principali dei due algoritmi, mettendo in risalto i diversi ragionamenti effettuati e le scelte tecniche.

3.1 Validazione algoritmo di esplorazione

Per quanto riguarda l'algoritmo 1, si è deciso di effettuare l'esplorazione tramite la tecnica *clessidra-rombo*, come già anticipato nel capitolo 2 sezione 2.2. Questa tecnica, a differenza di altre, permette di coprire l'area di lavoro più velocemente, presentando infatti un tempo medio di navigazione pari a 18 minuti e di scandagliare per prime le zone a probabilità maggiore per la presenza dell'inquinante.

Ovviamente il tempo impiegato dipende anche dalla distanza del vertice della zona di interesse più vicino dalla posizione iniziale del robot.

Se si confronta con una traiettoria fatta a transetti di ampiezza 7 metri si osserva come il tempo per la tecnica *rombo-clessidra* è minore di circa il 20%. Per questi motivi è stata scartata la tecnica di perlustrazione a *transetti*. Inoltre se la macchia fosse stata su un vertice opposto a quello di partenza del robot, sarebbe stata individuata solamente nella fase finale.

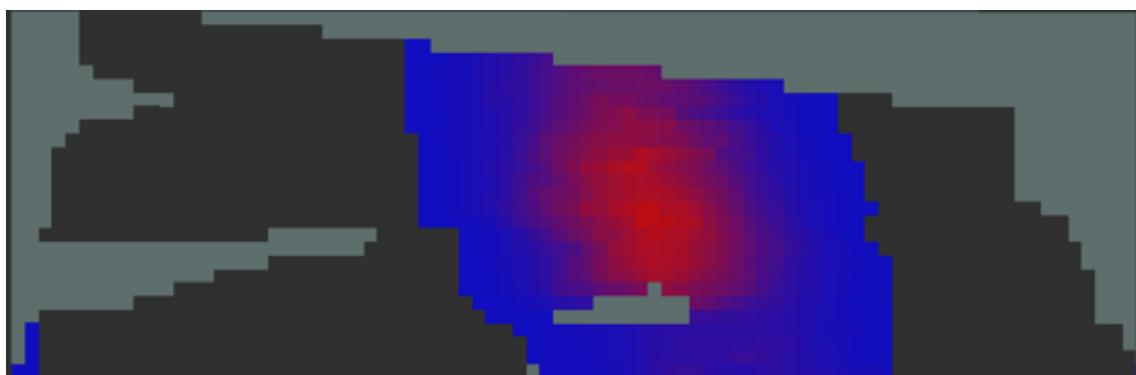


Figura 3.1 – Esplorazione tramite la tecnica *clessidra-rombo*.

In figura 3.1 si vede come il robot sia riuscito a trovare la macchia ancor prima di concludere la fase di esplorazione (in nero è raffigurata l'area coperta durante la missione).

Per quanto riguarda quindi la parte di esplorazione, non ci sono problematiche degne di nota.

3.2 Validazione algoritmo 2

Per quanto riguarda l'algoritmo 2, si è optato per una costruzione geometrica dei waypoint utili alla circumnavigazione, come già anticipato nel capitolo 2 sezione 2.3. Per fare ciò sono necessari diversi step, primo fra tutti la definizione della matrice *positionMatrix* che rappresenta il prodotto tra la matrice contenente i valori di inquinante (*mapVirtual_matrix*) e la matrice contenente i valori di affidabilità (*mapReliableSubMatrix*). Anziché basare l'algoritmo solamente sul valore della percentuale di inquinante, si è preferito eseguire il prodotto tra le due matrici in modo da iniziare la circumnavigazione solamente in prossimità di valori il cui prodotto fosse maggiore della *soglia di avvicinamento*; in questo modo si ha alta percentuale di inquinante e alta affidabilità delle misure.

Una prima problematica riscontrata è quella relativa alla **mancanza** di predizione dei valori di inquinante: in questo modo le celle davanti al robot presentano lo stesso valore di inquinante della posizione in cui si trova.

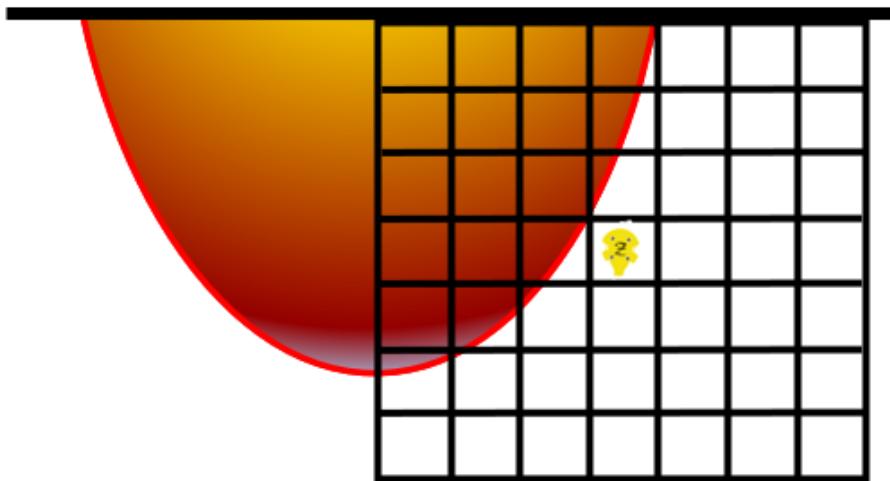
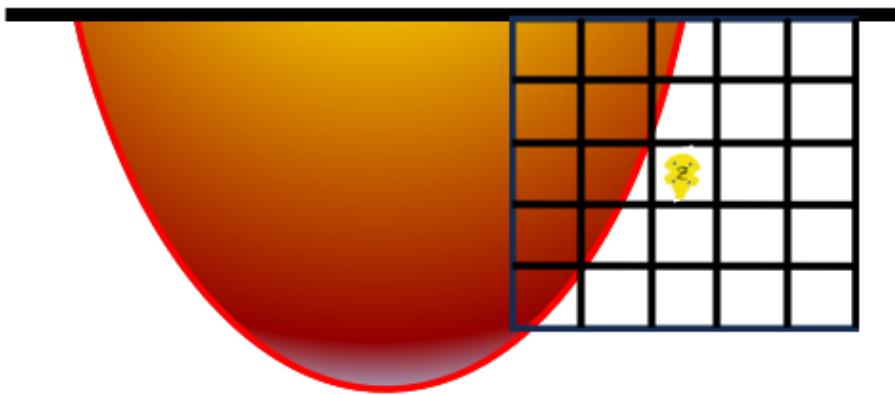
Tali valori vengono aggiornati a mano a mano che il robot si muove e ne viene fatta una media pesata: questo fattore influenza notevolmente l'algoritmo 2 in quanto vi è la possibilità che ponendo il waypoint O_2 in una cella con valore iniziale del 50%, questa possa assumere un valore differente nei passi successivi portando il robot all'interno della macchia.

Una volta ottenuta la *positionMatrix* si è deciso di estrarne una sottomatrice di dimensioni 7x7 (il cui elemento (3,3) è la posizione del robot) in modo da ridurne il carico computazionale. La dimensione è scelta pari a 7 per il semplice motivo che il gruppo di mapping ha definito un raggio di interpolazione pari a 3 metri: essendo però troppo piccolo, c'è la possibilità di incorrere in situazioni in cui i waypoint vengono posti a distanza ravvicinata dal robot, il quale potrebbe non muoversi e considerarli raggiunti.

Un'ulteriore motivazione dell'estrazione di una matrice 7x7 è legata al fatto che avendo un sensore puntuale, non sarebbe possibile eseguire una circumnavigazione con la sola misura in cui si trova il robot; in questo modo invece è possibile avere misure a 3 metri da Zeno e riuscire così a costruire i waypoint.

Considerando la matrice *productPositionFlag* si procede ora all'estrazione di un'ulteriore sottomatrice denominata *elementiDaEstrarre*: la dimensione varia in base all'orientazione del robot (angolo di yaw). I waypoint O_1 e O_2 vengono costruiti e posizionati all'interno di questa matrice. In particolare si è pensato di non costruire una traiettoria tramite un poligono già prefissato bensì effettuare la costruzione geometrica dei waypoint passo dopo passo. Il motivo è legato al fatto che avendo un raggio di interpolazione di soli 3 metri non avrebbe avuto senso mettere dei waypoint in celle della mappa con affidabilità bassa.

Un caso particolare, ma non raro, è la possibilità che vi sia la macchia vicino ai bordi dell'area di lavoro e della griglia. In questa situazione la dimensione della matrice di *productPositionFlag* non può rimanere 7x7 ma deve essere variabile in relazione alla vicinanza al bordo: in questo caso viene definita la variabile *new_size*. Per esempio se il robot dista 3 celle dal bordo (figura 3.2a), la dimensione rimane invariata mentre se ne dista 2 (figura 3.2b) significa che deve essere ridotta a 5 la variabile *new_size*.

(a) Matrice *productPositionFlag* con dimensione 7x7(b) Matrice *productPositionFlag* con dimensione 5x5**Figura 3.2** – Situazione al bordo dell'area di lavoro e della griglia.

I *reverse waypoint*, definiti nel capitolo 2 alla sottosezione 2.3.4, sono stati così costruiti in modo da far sì che il robot torni verso i waypoint (O_1 e O_0) appena raggiunti: in questo modo si fa sì che le celle di ritorno siano in una zona senza inquinante e abbiano un valore di affidabilità maggiore.

I *waypoint* utilizzati per l'algoritmo di ritorno, come già descritti nel capitolo 2 alla sottosezione 2.3.5, entrano in gioco quando la circumnavigazione si conclude: per comodità, il robot si dirige al vertice più vicino, che diventa il primo waypoint, e successivamente verso il waypoint più vicino alla zona di immissione/prelievo.

Capitolo 4

Risultati sperimentali

In questo capitolo vengono descritte le diverse prove effettuate, prima al simulatore e poi direttamente al lago.

4.1 Prove intermedie

Le prove intermedie sono state eseguite al simulatore tramite l'interfaccia grafica mostrata in figura 4.1.

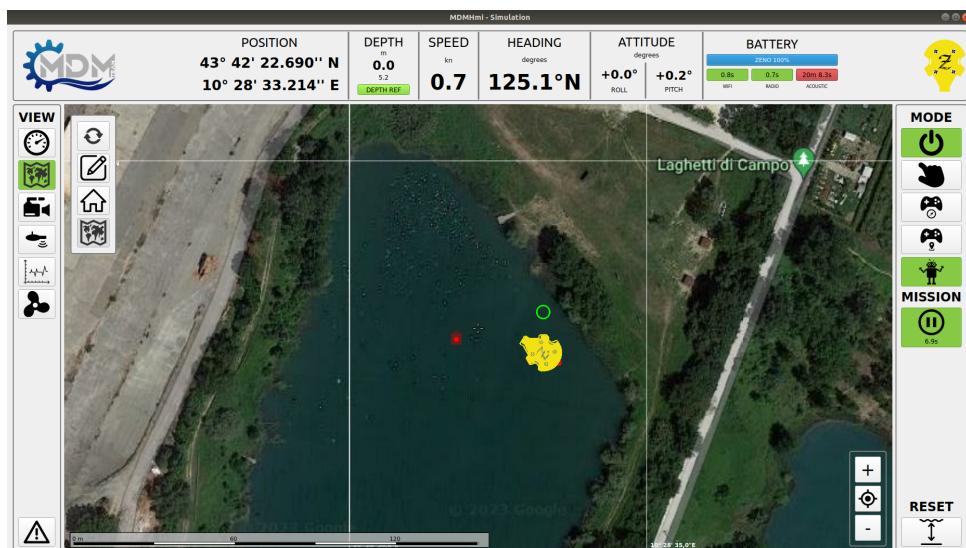


Figura 4.1 – Interfaccia grafica simulatore.

Grazie a quest'interfaccia¹, è stato possibile validare gli algoritmi di pianificazione descritti nel capitolo 2.

4.1.1 Pubblicazione dei waypoint di esplorazione

La prima simulazione è stata effettuata per verificare che i waypoint relativi alla parte di esplorazione, fossero pubblicati e seguiti correttamente da Zeno. Il problema riscontrato riguardava il tempo effettivo di completamento della missione (pari a 1 ora e 40 minuti) risultato eccessivo per i metri che il robot doveva effettivamente percorrere. Questo è dovuto al fatto che la pubblicazione della lista dei waypoint veniva fatta senza verificare lo stato effettivo di Zeno, il problema è stato risolto, facendo sì che la pubblicazione avvenisse solo una volta.

¹<https://www.mdmteam.eu>

4.1.2 Errore nell'angolo di orientazione

Verificata la correttezza della pubblicazione dei waypoint di esplorazione, durante la seconda simulazione, effettuata con tutti e tre i gruppi, si è notata un'errata posizione della macchia di inquinante (visualizzata su *Rviz*). Tale errore era dovuto a un errata costruzione dell'angolo di orientazione (*alpha_rot*), che al variare della posizione del robot riportava un valore di inquinante non rappresentativo rispetto al valore che il sensore riproduceva. Una volta trovato, l'errore è stato corretto e si è potuto così visualizzare la mappa correttamente.

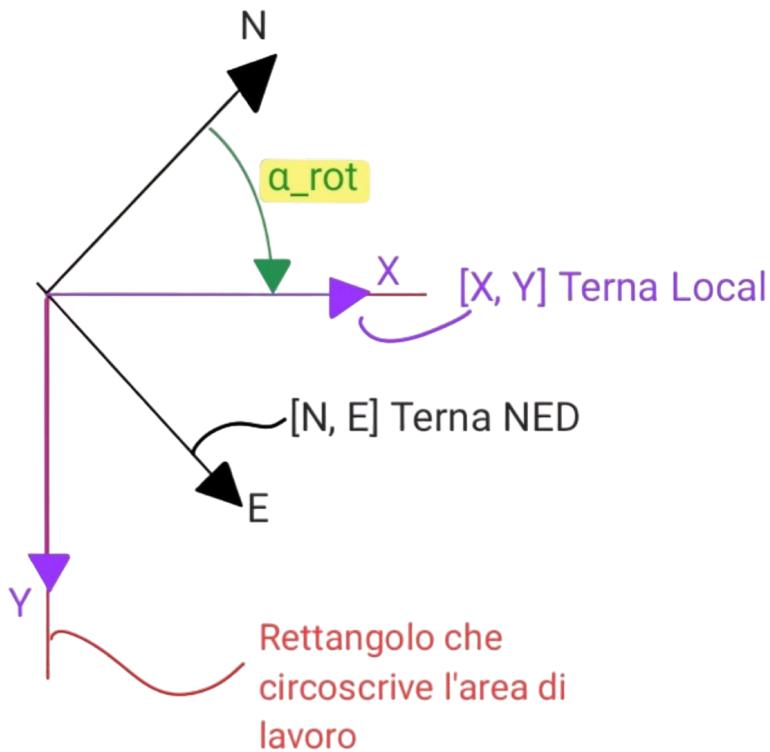


Figura 4.2 – Rappresentazione dell'angolo di orientazione α_{rot} .

Non tutti gli errori sono stati rilevati in largo anticipo rispetto alla data di sperimentazione al lago: ciò ha portato ad effettuare le dovute correzioni solo in seguito. Infatti, le successive prove al simulatore, utili a verificare l'esecuzione dell'algoritmo di circumnavigazione, hanno mostrato diversi errori computazionali che precedentemente non sono stati rilevati a causa della mancata corretta comunicazione tra gli algoritmi dei gruppi. Prova dopo prova gli errori relativi alla scrittura del codice sono stati sistematati, portando alla corretta definizione dei waypoint relativi alla circumnavigazione. Le correzioni sono state apportate basandosi sulle bag ottenute durante le prove, in modo da aver chiaro cosa stesse succedendo. Tuttavia nell'ultima simulazione eseguita, si è notato che lo stato del robot passava da RUNNING a PAUSED senza che l'algoritmo intervenisse. Dopo aver studiato la relativa *bag* si è notato come la posizione del robot non veniva più ricevuta: di conseguenza lo stato sarebbe rimasto a PAUSED per sempre. Nonostante ciò non è stato ancora appreso il motivo della pubblicazione di un messaggio di pausa durante la simulazione, cosa che dovrà essere corretto in uno sviluppo futuro.

4.2 Prove al lago

Dopo aver eseguito diverse simulazioni, in data 1 giugno 2023 si è tenuta la prova finale ai Laghetti di Campo (PI). L'obiettivo era quello di verificare il corretto comportamento di Zeno per ciascun macro gruppo. Purtroppo una volta avviata la missione sono state riscontrate diverse problematiche che non hanno permesso il corretto funzionamento degli algoritmi.

L'errore principale è relativo all'algoritmo di esplorazione in quanto i waypoint dell'area di lavoro, definiti manualmente nel file dei parametri, non sono stati inseriti nell'ordine corretto. Ciò ha pregiudicato il funzionamento dell'algoritmo che ha portato il robot a compiere un percorso totalmente differente da quanto ci si aspettava mostrato in figura 4.3.

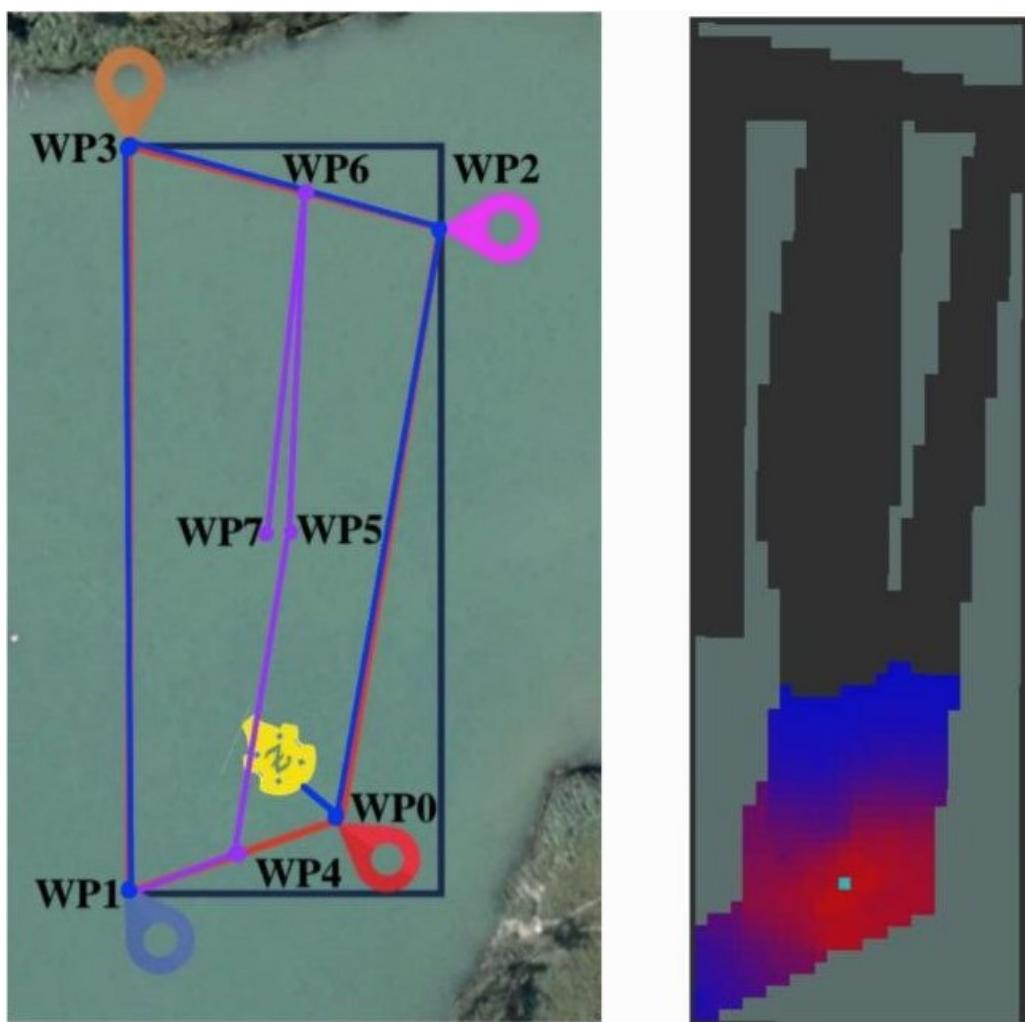


Figura 4.3 – L'immagine a sinistra rappresenta la traiettoria eseguita da Zeno durante la sperimentazione; l'immagine a destra mostra la traiettoria visualizzata su *Rviz*, dove i colori rappresentano la percentuale di inquinante.

La traiettoria di ispezione desiderata per migliorare la fase di ricerca della mappa, doveva essere come quella rappresentata in figura 4.4.

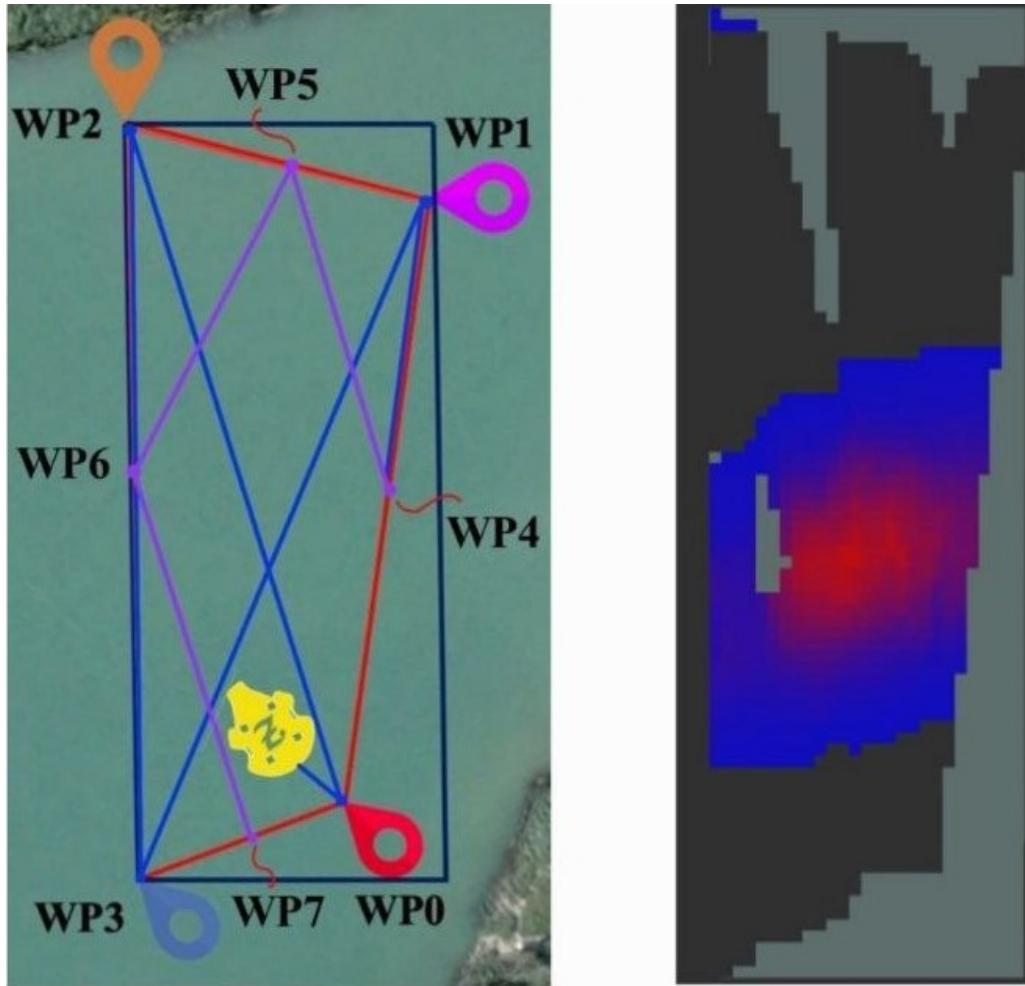


Figura 4.4 – L’immagine a sinistra rappresenta la traiettoria desiderata; l’immagine a destra mostra la traiettoria visualizzata su *Rviz*, dove i colori rappresentano la percentuale di inquinante.

Inoltre durante l’analisi dei dati postumi all’esperimento al lago, è stato rilevato che la posizione del robot in terna Local aveva componenti quasi sempre negative.

Siccome la scelta della terna Local è stata effettuata per semplificare l’identificazione della cella in cui vi è Zeno, avere componenti negative ha pregiudicato la costruzione della matrice *productPositionFlag*, la quale è risultata vuota. Questo perché non è possibile osservare la cella $[i,j]$ di una matrice se i e j assumono un valore negativo.

Questo problema è dovuto ad una variazione di segno all’angolo di orientazione del sistema di riferimento locale rispetto alla terna NED, che è stata apportata qualche istante prima della sperimentazione, causato dal mancato studio di alcune casistiche geometriche che dovevano essere studiate nel nodo *circ_rect*.

Questa modifica è stata apportata perché non veniva riprodotta la mappa che circoscrive l’area di lavoro e senza di essa la prova non poteva essere riprodotta.

In figura 4.5, in blu, sono rappresentate le componenti x e y della posizione del robot, chiamata *P_local*. In seguito, apportate le modifiche necessarie (effettuate in date seguenti alla sperimentazione), il risultato sperato durante la prova al lago è rappresentato nella figura 4.5 in verde, in cui sono descritte le componenti x e y della *P_local*.

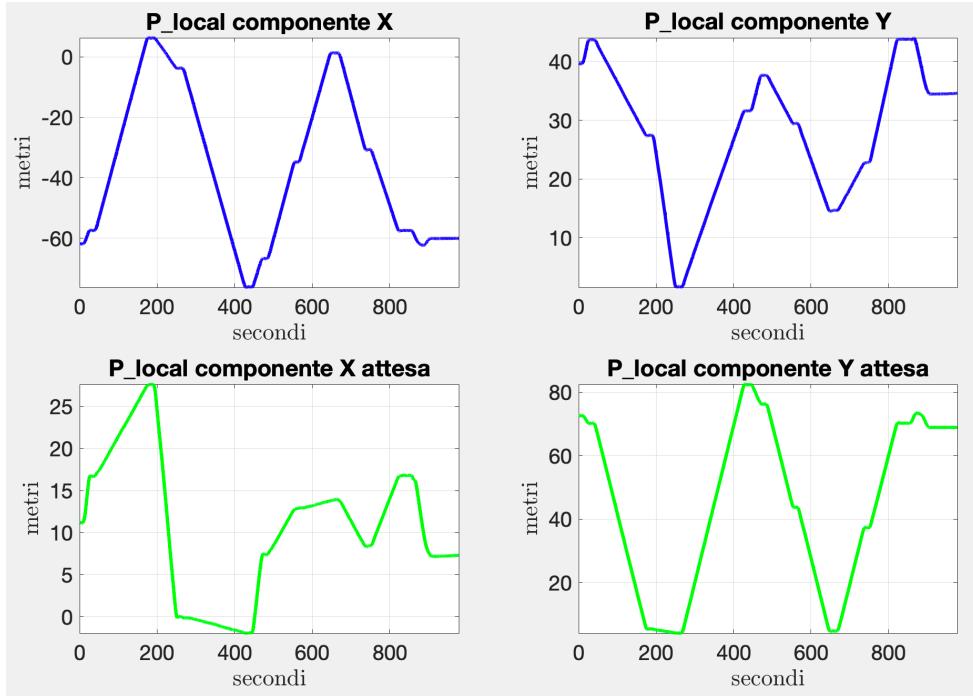


Figura 4.5 – Rappresentazione grafica delle componenti della posizione del robot in terna Local: in blu, il risultato della sperimentazione; in verde, il risultato desiderato.

In figura 4.6 è rappresentato il confronto tra le componenti della posizione del robot in terna Local, osservando le differenti zone in cui esse si presentano.

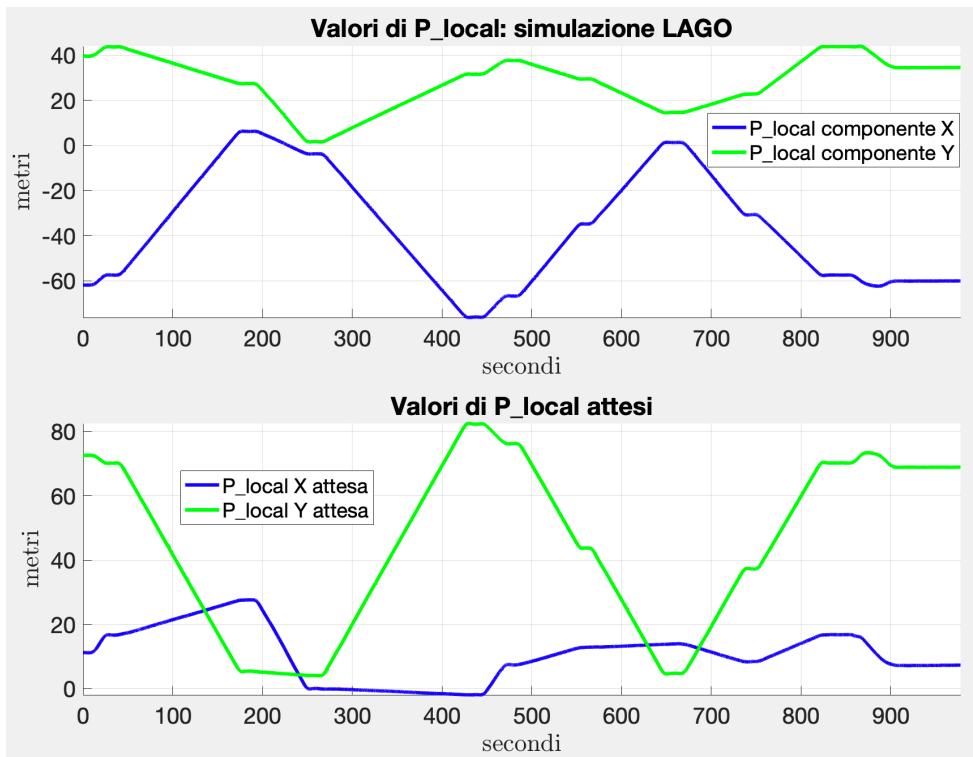


Figura 4.6 – Confronto grafico delle componenti della posizione del robot in terna Local: in blu, il risultato della sperimentazione; in verde, il risultato desiderato.

Capitolo 4 Risultati sperimentali

Al variare della posizione di Zeno, i valori di inquinante che dovevano essere rilevati sono rappresentati in figura 4.7, confrontati con l'andamento di quelli forniti dal gruppo sensing.

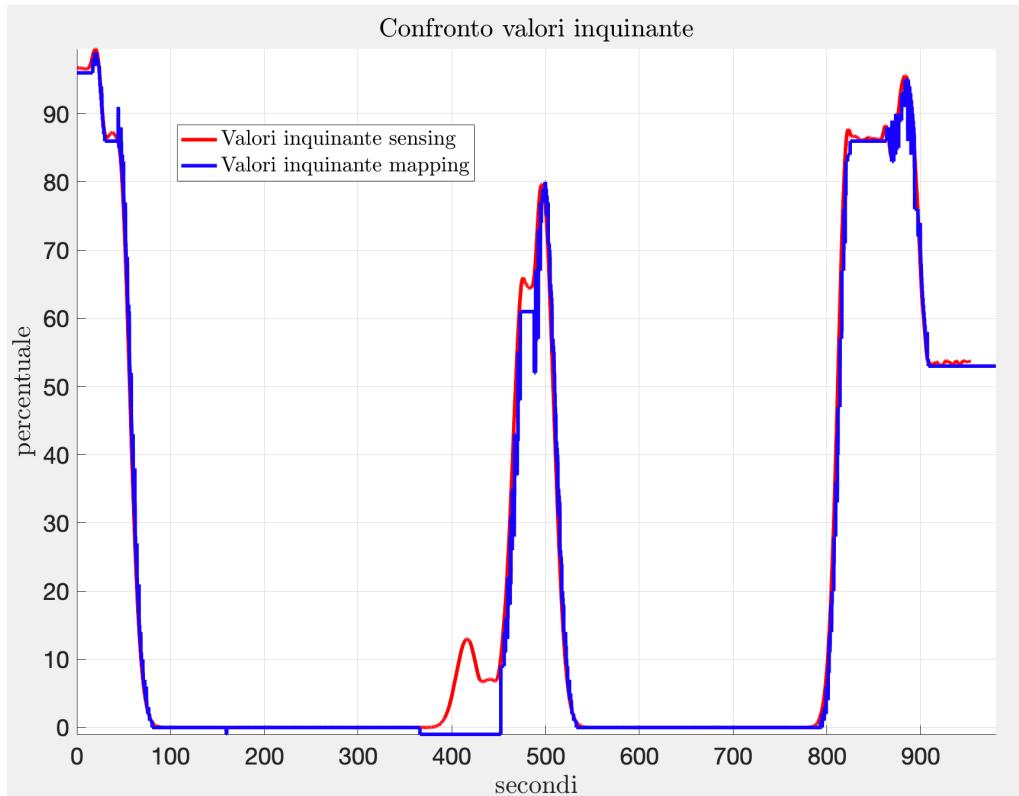


Figura 4.7 – Confronto tra i valori di inquinante rilevati e quelli forniti dal gruppo sensing.

Come si può osservare dal grafico, tra l'istante 350 e 450, i valori di inquinante forniti dal gruppo mapping sono pari a -1 nonostante quelli relativi al gruppo sensing fossero compresi tra 0 e 13. Questo è dovuto al fatto che il robot si trovava molto vicino al bordo e può essere successo che il GPS abbia fornito delle misure scostate dalla posizione vera di qualche metro.

Conclusioni

Dall'analisi dei dati raccolti è possibile vedere che l'algoritmo di esplorazione, a meno di errori umani, garantisce una traiettoria per la raccolta dei dati necessari per la localizzazione dell'agente inquinante.

Invece l'algoritmo per la circumnavigazione ha generato molte problematiche legate alle possibili casistiche considerate: ad esempio posizione variabile della macchia, costruzione adattiva dei waypoint della traiettoria basandosi sulla correzione, ecc.

La scelta di una pianificazione in real-time permette di avere una soluzione migliore, ma più complessa perché è necessario valutare più scenari: la mancanza di tempo ha impedito di completare l'analisi e garantire il compito richiesto.

In realtà sarebbe stato possibile utilizzare anche soluzioni più imprecise e semplici (ad esempio una traiettoria rettangolare costruita offline intorno alla macchia), ma che garantissero la circumnavigazione.

Sicuramente la costruzione di una missione più grezza, come backup della soluzione scelta, avrebbe permesso la completa raccolta dei dati nel caso in cui la traiettoria adattiva avesse fallito.

Lo sviluppo di questo progetto ha permesso di acquisire nuove conoscenze tecniche su *Python*, *ROS* e su come costruire una missione di esplorazione quando si utilizza robot come Zeno. Dal punto di vista interpersonale è stata accresciuta l'esperienza nel lavoro di gruppo e la gestione delle connessioni e delle informazioni fra i vari gruppi.

Alcuni possibili sviluppi futuri sono i seguenti:

- costruzione dei waypoint basandosi su una fase di predizione dell'inquinante, oltre alla correzione;
- gestione dei casi limite, ovvero algoritmo di fine circumnavigazione quando la macchia è a contatto con un lato dell'area di lavoro;
- pianificazione di una traiettoria ottimale di ritorno che garantisca che Zeno non passi attraverso la macchia mentre si dirige verso WP0;
- previsione di una nuova fase di esplorazione se la macchia è posizionata nei luoghi non scandagliati precedentemente e non è stata rilevata;
- concatenazione di varie missioni di circumnavigazione se vi sono almeno due macchie nella zona di interesse.