Informatica - Area scientifica
Dipartimento di Scienze matematiche, informatiche e multimediali
Università di Udine

Progetto di Droni e Sistemi Robotici

Giabardo Diletta (144176)
Rasera Giovanni (143395)

Anno accademico 2023/2024

Indice

1 Specifiche del progetto

Ambiente di sviluppo : ROS2 + Gazebo
Simulatore drone Gazebo : sjtu
Algoritmo Path Planning: Artificial Potential Field

Configurazione degli scenari:

[a] movimento lungo percorso da un punto A ad un punto B senza ostacoli;

[b] movimento lungo percorso da un punto A ad un punto B con ostacoli interposti;

[c] disposizione di target waypoint numerati in modo da realizzare un percorso in un ambiante tridimensionale, con l'aggiunta di ulteriori ostacoli. La posizione dei waypoint può essere scelta arbitrariamente e il drone raggiunge i waypoint rispettandone l'ordine di numerazione. Le posizioni di waypoint e ostacoli sono fisse durante la simulazione;

[d] come [c] ma con la disposizione dei waypoint numerati e degli ostacoli generata casualmente. Le posizioni di waypoint e ostacoli sono fisse durante la simulazione e anche in questo caso il drone deve raggiungere i waypoint rispettandone l'ordine di numerazione;

[e] [Opzionale] come [d]/[c] ma le posizioni di ostacoli e waypoint variano con continuità durante la simulazione con movimenti randomici e velocità inferiore rispetto a quella del drone.

Ambiente di Test:

Prima di procedere al setup dell'enviroment finale abbiamo proceduto a realizzare tutti i punti richiesti in un ambiente di Test, nello specifico in locale utilizzando Python.
Poi, dopo aver soddisfatto le richieste in questo ambiente abbiamo trasferito il tutto nell'enviroment definito.

Nello specifico, partendo dal file **3_CombinedPotentialFields_template.py**, abbiamo aggiunto la componente drone che è stata posta in mezzo al campo partendo da un punto iniziale x,y.
Successivamente, in base alla forza che percepiva in quel determinato punto, si spostava di conseguenza.

```
# movement
droneGridPos = np.array([math.floor(drone[0]), math.floor(drone[1])])
forceOnDrone = np.array([
  delx[droneGridPos[1]][droneGridPos[0]],
  dely[droneGridPos[1]][droneGridPos[0]]
])

# calculate force to apply
realForceOnDrone = (np.array([
  forceOnDrone[0] / maxForseX,
  forceOnDrone[1] / maxForseY
]))

print(f"droneGridPos: {droneGridPos}")
print(f"forceOnDrone: {forceOnDrone}")
print(f"realForceOnDrone: {realForceOnDrone}")

# move the drone
drone = drone + realForceOnDrone
```

Questo ci ha permesso di muovere il drone verso il goal evitando l'ostacolo.
Tuttavia non è stata implementata una soluzione al problema dei minimi locali, infatti in alcune occasioni il drone si blocca. Questo problema accade per esempio quando goal, ostacolo e drone sono allineati nel seguente modo:

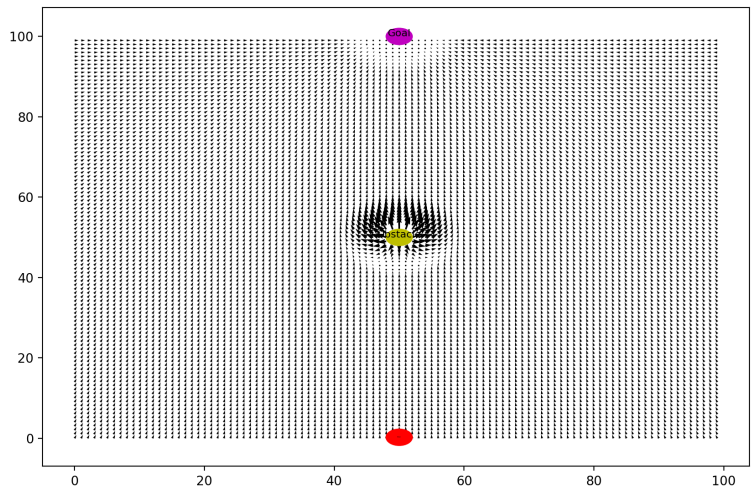


Figura 1.1: Drone: (50,0), Ostacolo: (50,50), Goal: (50,100)

Successivamente il campo potenziale generato viene utilizzato per comunicare al drone, in ambiente Gazebo, gli input di velocità negli assi x e y.

1.1 Setup Enviroment

Il progetto è stato realizzato a step, ed i seguenti sono gli scenari realizzati:

- Scenario a) Marker ai quattro angoli del mondo e goal;
- Scenario b) Marker ai quattro angoli del mondo, ostacoli e goal;
- Scenario c) Marker ai quattro angoli del mondo e goal;
- Scenario d) Marker ai quattro angoli del mondo, ostacoli e goal.
- Scenario e) Marker ai quattro angoli del mondo, ostacoli e goal.

1.2 Strumenti Utilizzati

Gazebo
Gazebo è un simulatore open-source per la robotica, che offre un ambiente 3D realistico per testare e sviluppare algoritmi robotici. Supporta la simulazione di sensori, oggetti e ambienti complessi, consentendo di verificare il comportamento del robot senza rischi nel mondo reale. È spesso utilizzato in combinazione con ROS per testare e validare sistemi robotici.
Utilizzo: nel progetto è stato utilizzato per simulare il movimento del drone e la dinamica sullo stesso. Oltre che alla creazione del mondo in cui opera il drone, il goal ed eventuali ostacoli.

ROS
ROS (Robot Operating System) è un framework open-source per lo sviluppo di software robotico, che fornisce librerie e strumenti per costruire applicazioni robotiche. Facilita la comunicazione tra diversi componenti di un robot, come sensori, attuatori e algoritmi di controllo. ROS supporta la modularità e la riusabilità del codice, accelerando lo sviluppo e la ricerca nel campo della robotica.
Utilizzo: all'interno del progetto è stato utilizzato per la comunicazione tra Gazebo e python, questo tramite l'utilizzo dei topic nei quali vengono pubblicati dei valori che permettono di prendere delle decisioni in merito ai movimenti del drone e portano i nuovi movimenti da effettuare.

RViz
RViz (Robot Visualization) è uno strumento di visualizzazione 3D per ROS, utilizzato per visualizzare dati da sensori, stato del robot e percorsi pianificati. Facilita il debugging e la simulazione, consentendo un'analisi in tempo reale del comportamento del robot. È altamente configurabile e supporta plugin per estensioni personalizzate.
Utilizzo: è stato utilizzato all'interno del progetto per visualizzare i dati dai sensori ed avere un feedback per la videocamera (che potrebbe essere utile in futuro per studi di Computer Vision).

Python
Python è un linguaggio di programmazione ad alto livello, noto per la sua sintassi semplice e leggibile, che facilita lo sviluppo rapido di applicazioni. È ampiamente utilizzato in vari campi, inclusi il web development, la data science, l'intelligenza artificiale e l'automazione. Grazie alla sua vasta libreria standard e alla comunità attiva, Python supporta una vasta gamma di funzionalità e integrazioni.
Utilizzo: è stato utilizzato per andare a costruire il PFM e per realizzare la comunicazione tra i PFM e Gazebo.

Docker
Docker è una piattaforma di containerizzazione che permette di creare, distribuire ed

eseguire applicazioni in container isolati e portatili. Questi container includono tutto il necessario per l'applicazione, garantendo portabilità e efficienza. Docker semplifica la gestione delle applicazioni e offre un ricco ecosistema di strumenti.
Utilizzo: è stato utilizzato per permettere di avere tutti gli strumenti utili per poter far girare il progetto senza andare ad intaccare il sistema sottostante.
Inoltre il docker presentato all'interno del progetto originale, presenta alcune scomodità, che sono state risolte andando ad implementare alcuni file in maniera customizzata, per quello che era necessario realizzare per questo progetto.

1.3 Algoritmo di Path Planning

Una volta definita la posizione dei vari oggetti, ad ogni istante t, avviene il calcolo del path planning.
Per fare in modo che il drone segua un path predefinito andiamo ad utilizzare il PFM (Potential Field Method) che va a costruire un campo elettrico all'interno del mondo, nel quale il drone avrà carica positiva, come gli ostacoli, e il goal avrà carica negativa. Quindi il drone andrà a navigare sul mondo seguendo il campo elettrico risultate dagli ostacoli, che andranno a respingere il drone, e il goal, che attrarrà il drone.

Conversione coordinate del mondo Per andare a comunicare con gazebo la posizione del drone all'interno del mondo si è andati ad utilizzare un converter che andasse a convertire le coordinate del "mondo" 50x50 discreto di python al mondo di Gazebo 20x20 reale. Per fare questo siamo andati ad utilizzare questa formula:

$$p_x = \frac{(g_x - g_{min_x}) \cdot (p_{max_x} - p_{min_x})}{g_{max_x} - g_{min_x}} + p_{min_x}$$
$$p_y = \frac{(g_y - g_{min_y}) \cdot (p_{max_y} - p_{min_y})}{g_{max_y} - g_{min_y}} + p_{min_y}$$

Dove px e py rappresentano le coordinate convertite dal mondo gazebo a quello python.

L'implementazione vera e propria si trova nel file *converter_position.py* ed è la seguente:

```
1 abs_distance_x = abs(gmaxx - gminx)
2     abs_distance_drone_x = abs_distance_x - abs(gmaxx - gx)
3     div_len_x = abs_distance_x / pmaxx
4     px = Math.floor(abs_distance_drone_x / div_len_x)
```

Inoltre sono state utilizzate formule simili per fare il contrario:

$$g_x = \frac{(p_x - p_{min_x}) \cdot (g_{max_x} - g_{min_x})}{p_{max_x} - p_{min_x}} + g_{min_x}$$
$$g_y = \frac{(p_y - p_{min_y}) \cdot (g_{max_y} - g_{min_y})}{p_{max_y} - p_{min_y}} + g_{min_y}$$

Dove gx e gy rappresentano le coordinate convertite dal mondo python a quello gazebo.

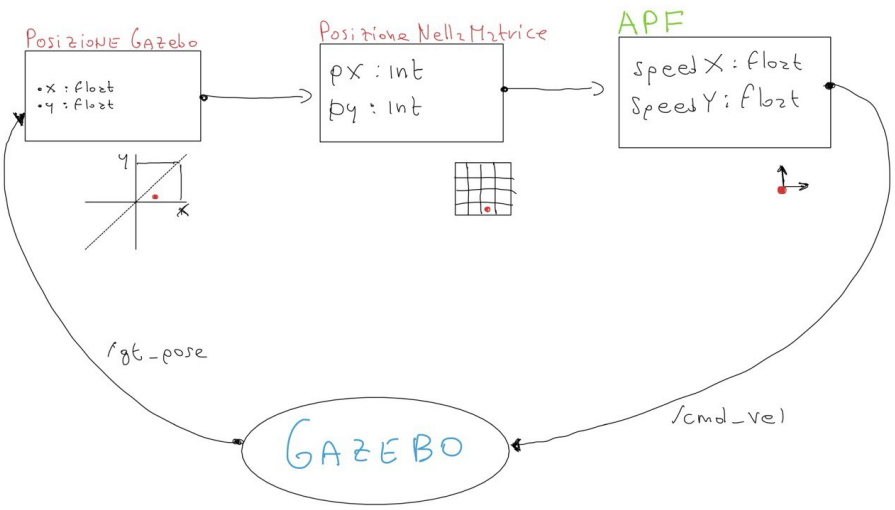
Per entrambe le formule:

- gminx = il valore minimo nell'asse delle x all'interno di gazebo;
- gminy = il valore minimo nell'asse delle y all'interno di gazebo;
- gmaxx = il valore massimo nell'asse delle x all'interno di gazebo;
- gmaxy = il valore massimo nell'asse delle y all'interno di gazebo;
- pminx = il valore minimo nell'asse delle x all'interno di python;
- pminy = il valore minimo nell'asse delle y all'interno di python;
- pmaxx = il valore massimo nell'asse delle x all'interno di python;
- pmaxy = il valore massimo nell'asse delle y all'interno di python;

Infine sono state fatte alcune modifiche per rendere più testabile il codice, in particolare: viene decomposto il calcolo in passaggi intermedi con l'uso di variabili intermedie per calcolare le distanze assolute e relative, la lunghezza del divisore e quindi le coordinate px e py.

Il codice della conversione dal mondo python a quello gazebo si trova nel seguente path del progetto:
src/sjtu_drone_apf_speed_control/sjtu_drone_apf/converter_position.py

Movimento Il processo che permette al drone di partire da un punto A ed arriva ad un punto B richiede controllo a catena chiusa. In particolare, quello che viene fatto, è una lettura della posizione del drone, questa viene tradotta da gazebo nella posizione della matrice del campo; il campo permette di estrarre gli input di velocità da dare al drone. La catena si chiude nel momento in cui viene fatto il controllo in velocità e viene ricalcolata la nuova posizione del drone.



Questo viene fatto per ogni movimento all'interno di ogni scenario.

1.4 Problema dei minimi locali

Il problema dei minimi locali sussiste in particolari situazioni dove il campo si annulla, e di conseguenza il drone non si muove, visto che non percepisce alcuna forza su di esso.
In un primo approccio è stato verificato se la dinamica applicata al drone sjtu in ambiente Gazebo permetteva comunque di evitare che si cadesse in un punto di minimo locale. Questo è stato verificato poichè nell'ambiente python, in cui è stato costruito il PFM, è privo di dinamica, mentre nel mondo di Gazebo è presente. Di conseguenza l'inerzia causata dal movimento potrebbe far "saltare" al drone la cella dove nel PFM sarebbe presente un punto di minimo locale.
Quello che è stato osservato è che la dinamica del mondo Gazebo permette al drone di evitare i punti di minimo locale solo in certi casi. Sarà necessario, in futuro, implementare una logica di superamento dei minimi locali.

1.5 Gestione dei WayPoint

I waypoint sono gestiti dall'utente tramite il programma teleop.py. Questo permette di selezionare il numero del waypoint di interesse, che, negli esempi utilizzati, possono essere 1,2 o 3.

Per far partire la simulazione bisogna selezionare il pulsante **t** che permette di effettuare il takeoff, successivamente è necessario selezionare il tasto **g** che permetterà di azionare l'autonomia del drone. Appena viene premuto g il drone andrà verso il primo waypoint di interesse, nel momento in cui l'utente decide di passare al waypoint successivo deve premere il numero corrispondente.

In una versione più sofisticata, in futuro, si potrebbe sostituire l'azione umana con un nodo ROS che calcola la distanza tra il drone e il waypoint di interesse e decide quando selezionare il waypoint successivo.

Il valore dei waypoint è inserito all'interno del file: *waypoints.json*

1.6 Problematiche affrontate all'interno del progetto

1.6.1 Drone PX4

Inizialmente il progetto doveva essere realizzato tramite il drone PX4. Tuttavia, durante la realizzazione del progetto stesso, l'idea è stata abbandonata per diversi motivi tecnici e pratici. Nel particolare:

Supporto limitato per il simulatore:

- PX4 mostra limitazioni quando si tenta di compilarlo per il simulatore. In particolare, sembra che non supporti più certi tipi di veicoli, specialmente i droni, nel contesto della simulazione;
- La documentazione di PX4 non è aggiornata o corretta: indica di compilare per target specifici che non esistono più, rendendo difficile seguire le istruzioni e ottenere risultati funzionali nel simulatore;
- Anche se si riesce a compilare PX4 per alcuni target come gli aerei a doppia ala, i target per i droni non funzionano, probabilmente in futuri sviluppi si potrebbe cercare di trovare il target più adeguato.

Compatibilità con ROS:

- PX4 ha smesso di supportare direttamente ROS (Robot Operating System) e ora utilizza un protocollo di comunicazione proprietario chiamato UORB. UORB, pur essendo simile a ROS, richiede un "wrapper" per interfacciarsi con ROS, complicando ulteriormente l'integrazione;
- La necessità di utilizzare ROS Bridge per far comunicare ROS con UORB aggiunge un ulteriore strato di complessità, dato che non tutte le funzionalità sono supportate o compatibili. Questa complicazione rende più difficile sviluppare e mantenere il sistema, soprattutto in fase di simulazione e test.

Obsolescenza e documentazione:

- La documentazione non aggiornata di PX4 e i target non esistenti rendono difficile seguire il processo di sviluppo e simulazione;
- La confusione tra la compilazione per hardware reale (dove i target esistono) e per il simulatore (dove non esistono) crea ulteriori problemi, facendo sembrare che alcune funzionalità siano state tolte senza una chiara comunicazione agli utenti.

Per tutti i punti sopra elencati si è deciso di passare al sjtu drone, che riesce ad ovviare queste problematicità ed ha permesso al progetto di procedere.

1.6.2 Docker

In alcune distribuzioni Linux, la versione di gazebo che viene scaricata non è compatibile con ROS. Quindi, per andare a forzare il download di una cerca versione, è stato deciso di utilizzare un docker; questo ha permesso di scaricare tutte le versioni giuste e compatibili.
Difatti all'interno del docker creato è presente un file che dice quali sono le versione corrette e compatibili. Quindi il docker avrà tutto pronto e correttamente configurato.

1.6.3 Topic

Nel momento in cui si utilizzano gli script di stju drone, in particolare lo script *teleop.py* che si trova all'interno nella sezione di controllo del drone stesso, non è possibile mettersi in ascolto di qualsiasi topic. Quindi la sezione che sta in ascolto del topic non funziona e quando si prova ad iscriversi (tramite subscribe) al topic questo non fa vedere alcune informazione, nonostante il topic, da terminale, mostri i dati legati al topic stesso.
Per risolverlo è stato realizzato uno script separato che viene lanciato successivamente al lancio di Gazebo e di ROS, e che permette di leggere e scrivere i valori all'interno del topic.

2 Scenario [a]

Richiesta dello scenario a: movimento lungo percorso da un punto A ad un punto B senza ostacoli.

Procedimento: per poter rispondere alla richiesta siamo andati a settare l'enviroment. Poi è stato necessario realizzare un mondo gazebo che contenesse i marker ai quattro angoli del mondo, il goal ed il drone.
Successivamente è stato implementato il codice di comunicazione tra l'enviroment di gazebo e il mondo python, questo tramite l'utilizzo di ros. In particolare sono stati utilizzati i canali di comunicazione di ros: cmd_val; per inviare le informazioni di velocità negli assi x e y del drone. Il valore di cmd_val viaggia tra -1 ed 1 per ogni asse.

Risultati: i risultati ottenuti in questo scenario sono buoni e soddisfano le richieste effettuate.
Al seguente link è possibile visualizzare un video dove si realizza il seguente scenario:

https://www.youtube.com/watch?v=KEkBba3iZuc

3 Scenario [b]

Richiesta dello scenario b: movimento lungo percorso da un punto A ad un punto B con ostacoli interposti.

Procedimento: per poter rispondere alla richiesta siamo andati a settare l'enviroment. Poi è stato necessario realizzare un mondo gazebo che contenesse i marker ai quattro angoli del mondo, il goal, il drone e degli ostacoli.
Successivamente è stato implementato il codice di comunicazione tra l'enviroment di gazebo e il mondo python, questo tramite l'utilizzo di ros. In particolare sono stati utilizzati i canali di comunicazione di ros: cmd_val; per inviare le informazioni di velocità negli assi x e y del drone. Il valore di cmd_val viaggia tra -1 ed 1 per ogni asse.

Risultati: i risultati ottenuti in questo scenario sono buoni e soddisfano le richieste effettuate.
Al seguente link è possibile visualizzare un video dove si realizza il seguente scenario, dove l'ostacolo è stato posizionato nelle coordinate x=25 y=25:

https://www.youtube.com/watch?v=HAnecKL0Ad0

Al seguente link è possibile visualizzare un video dove si realizza il seguente scenario, dove l'ostacolo è stato posizionato nelle coordinate x=25 y=10:

https://www.youtube.com/watch?v=CHujgEFMi_c

Inoltre, all'interno del seguente link, è possibile trovare un altra simulazione con delle spiegazioni annesse:

https://www.youtube.com/watch?v=-cSxRWCwpmY

4 Scenario [c]

Richiesta dello scenario c: disposizione di target waypoint numerati in modo da realizzare un percorso in un ambiante tridimensionale, con l'aggiunta di ulteriori ostacoli. La posizione dei waypoint può essere scelta arbitrariamente e il drone raggiunge i waypoint rispettandone l'ordine di numerazione. Le posizioni di waypoint e ostacoli sono fisse durante la simulazione.

Procedimento: per poter rispondere alla richiesta siamo andati a settare l'enviroment. Poi è stato necessario realizzare un mondo gazebo che contenesse i marker ai quattro angoli del mondo, il goal ed il drone. Inoltre sono stati posizionati dei waypoint che, all'interno del mondo, sono visualizzati allo stesso modo del goal.
Successivamente è stato implementato il codice di comunicazione tra l'enviroment di gazebo e il mondo python, questo tramite l'utilizzo di ros. In particolare sono stati utilizzati i canali di comunicazione di ros: cmd_val; per inviare le informazioni di velocità negli assi x e y del drone. Il valore di cmd_val viaggia tra -1 ed 1 per ogni asse.

Risultati: i risultati ottenuti in questo scenario sono buoni e soddisfano le richieste effettuate.
Al seguente link è possibile visualizzare un video dove si realizza il seguente scenario:

https://www.youtube.com/watch?v=apwwHCmKB2k

5 Scenario [d]

Richiesta dello scenario d: come [c] ma con la disposizione dei waypoint numerati e degli ostacoli generata casualmente. Le posizioni di waypoint e ostacoli sono fisse durante la simulazione e anche in questo caso il drone deve raggiungere i waypoint rispettandone l'ordine di numerazione.

Procedimento: per poter rispondere alla richiesta siamo andati a settare l'enviroment. Poi è stato necessario realizzare un mondo gazebo che contenesse i marker ai quattro angoli del mondo, il goal, il drone ed degli ostacoli. Inoltre sono stati posizionati dei waypoint che, all'interno del mondo, sono visualizzati allo stesso modo del goal.
Successivamente è stato implementato il codice di comunicazione tra l'enviroment di gazebo e il mondo python, questo tramite l'utilizzo di ros. In particolare sono stati utilizzati i canali di comunicazione di ros: cmd_val; per inviare le informazioni di velocità negli assi x e y del drone. Il valore di cmd_val viaggia tra -1 ed 1 per ogni asse.

Risultati: i risultati ottenuti in questo scenario sono buoni e soddisfano le richieste effettuate.
Al seguente link è possibile visualizzare un video dove si realizza il seguente scenario:

https://www.youtube.com/watch?v=KEkBba3iZuc

Al seguente link è possibile visualizzare un altro video:

https://www.youtube.com/watch?v=EHNFr1AXRUs

Al seguente link è possibile visualizzare un altro video ancora:

https://www.youtube.com/watch?v=LyyLIMuBUq4

In questo video è possibile vedere uno dei fallimenti/problemi incontrati durante le simulazioni. In particolare si può vedere come il drone vada contro un ostacolo. Difatti tutti i video in questa relazione hanno comportato ore e ore di testing per arrivare alla soluzione ottimale, e alla realizzazione dei video finali.

https://www.youtube.com/watch?v=sxrG9n_AffM

6 Scenario [e]

Richiesta dello scenario e: come [d]/[c] ma le posizioni di ostacoli e waypoint variano con continuità durante la simulazione con movimenti randomici e velocità inferiore rispetto a quella del drone.

Procedimento: per poter rispondere alla richiesta siamo andati a settare l'enviroment. Poi è stato necessario realizzare un mondo gazebo che contenesse i marker ai quattro angoli del mondo, il goal, il drone ed degli ostacoli. Inoltre sono stati posizionati dei waypoint che, all'interno del mondo, sono visualizzati allo stesso modo del goal.
Successivamente è stato implementato il codice di comunicazione tra l'enviroment di gazebo e il mondo python, questo tramite l'utilizzo di ros. In particolare sono stati utilizzati i canali di comunicazione di ros: cmd_val; per inviare le informazioni di velocità negli assi x e y del drone. Il valore di cmd_val viaggia tra -1 ed 1 per ogni asse.

Risultati: non è possibile implementare lo scenario poichè per considerare una variazione in continuità degli ostacoli e dei waypoint era necessario poter leggere, tra i vari topic, lo stato del mondo. Tuttavia, all'interno dei topic disponibili non ce n'è uno che dia queste informazioni. Di conseguenza non è stato possibile realizzare lo scenario e, senza avere le informazione sulle posizioni dell'ostacolo, non è stato possibile neanche andare a calcolare il nuovo PFM.

7 Conclusioni

Il codice del progetto può essere trovato all'interno di questo link GitHub:

`https://github.com/GiovanniRaseraF/APF-PathPlanning-ROS-Gazebo`

Suggeriamo di seguire i requirement (che trova al seguente link `https://github.com/GiovanniRaseraF/APF-PathPlanning-ROS-Gazebo/blob/main/src/sjtu_drone_apf_speed_control/README.md`) alla lettera e di scaricare le versioni presentate, poichè versioni precedenti o successive potrebbero compromettere l'apertura del programma.

Considerazioni: a parer nostro l'utilizzo, il programma realizzato, è utile in ambienti aperti piuttosto che chiusi, questo perchè il drone sembra molto lento nei movimento e sembra instabile, in particolare quando ci si avvicina a degli ostacoli. La simulazione è complessa e il fatto di utilizzare docker ci ha aiutato molto poichè permette di non preoccuparsi del setup di ros e gazebo rendendo più semplice lo sviluppo a chi è meno esperto in questi sistemi.

Limitazioni: bisogna avere dei punti di riferimento come marker per poter tradurre il mondo da gazebo a python, inoltre bisogna essere a conoscenza della posizione degli ostacoli. Non potendo leggere in tempo reale la posizione degli ostacoli non è possibile ricalcolare il campo in tempo reale.