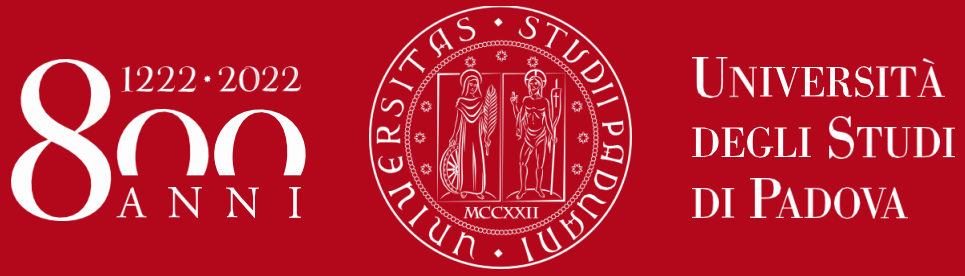


UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Dipartimento di Tecnica e Gestione industriale
Corso di laurea triennale in Ingegneria Meccatronica

TECNICHE DI LOCALIZZAZIONE INDOOR PER DRONI BASATE SU FIDUCIAL MARKER

Laureandi:
Civiero Matteo 1223955
Lovato Alessio 1216896
Piccina Alberto 1218038

Relatore
Michieletto Giulia

Correlatore
Michieletto Stefano

Indice

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

ABSTRACT

Abstract

Lo scopo di questa ricerca è quello di **analizzare le prestazioni di diverse famiglie di fiducial marker** per identificare quella più vantaggiosa nell'utilizzo indoor per la rilevazione della posizione di un UAV multirotore.

Lo studio delle diverse famiglie di marker, e quindi successivamente la scelta del marker migliore, è stato basato sulla revisione ed analisi di diversi paper che ne valutano il comportamento sotto altri ambiti di ricerca.

La valutazione delle prestazioni del marker si basa sulla **precisione della localizzazione tridimensionale e stima della posa**, rilevamento e analisi dei tag sotto l'influenza di blur delle immagini dovuto allo spostamento trasversale e longitudinale.

Per compiere queste valutazioni è stato quindi predisposto un set di prove che mettono a confronto le prestazioni delle diverse tipologie di marker utilizzando un drone multirotore.

Per selezionare i marker più idonei rispetto al nostro obiettivo ci siamo **basati su diversi studi e documenti riguardanti i diversi marker**, anche quando utilizzati in modo diverso da quello prefissato.

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

RICERCHE SUI FIDUCIAL MARKER

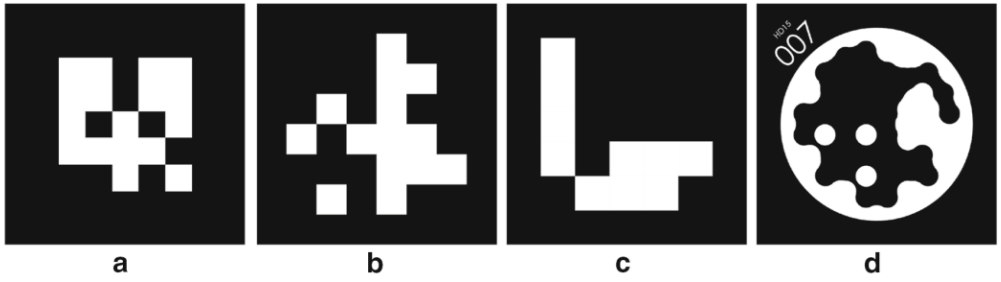
- Analisi dei paper
- ArUco
- STag
- Whycode

Analisi dei paper

Come punto di riferimento abbiamo selezionato [1] in cui viene riportata una panoramica circa i fiducial marker di cui si possono trovare pubblicazioni affidabili; lo stesso ne riporta caratteristiche, vantaggi e svantaggi in base alle possibili applicazioni. Vengono inoltre riportati **confronti** molto interessanti tra i fiducial marker più utilizzati negli ultimi anni, eseguendo esperimenti significativi per valutarne le performance in scenari diversi.

In particolare vengono analizzati:

- **ARTag**, uno dei primi fiducial marker e di conseguenza meno performante;
- **AprilTag**, già utilizzato per la localizzazione del nostro drone;
- **ArUco**, uno dei marker più diffusi in ambito UAV;
- **STag**, che si prospetta essere in rapida evoluzione nei prossimi anni ma con tuttora ottime prestazioni.



Esempi di fiducial marker: a)ARTag, b)AprilTag, c)ArUco e d)STag

Analisi dei paper

Tutti questi marker sono di forma **quadrata e monocromatici**, ma utilizzano tecniche diverse di codifica che determinano la configurazione interna del quadrato.
Di conseguenza ogni pacchetto ha un proprio algoritmo in grado di riconoscere il marker, di decodificarne l'identità e di stimarne posizione e orientamento rispetto alla camera.
Nel nostro caso la posizione della camera è stata poi riportata al centro del modello del nostro drone.

Abbiamo poi approfondito la ricerca [1] analizzando singolarmente altri tag (anche con forme e caratteristiche differenti dai precedenti) che ci sembravano interessanti e andando a recuperare altre ricerche specifiche su di essi.

In particolar modo ci siamo concentrati su **ArUco**[2], **STag**[4], **Whycode**[5], TRIP, InterSense, Pi-Tag e Fourier Tag, selezionando in conclusione i primi tre.

ArUco

ArUco fa parte della famiglia di marker a **forma quadrata** il cui contenuto è identificato da una matrice di elementi binari codificata sulla base dei colori bianco (1) e nero (0).
Nello specifico questo marker è stato sviluppato specificamente per la stima di posizione della camera.

ArUco è uno dei marker più utilizzati nella navigazione basata su questo tipologia di sistema, tanto da essere citato ed utilizzato in più di **100 ricerche scientifiche**.

Uno dei punti di forza di questo marker è il fatto che non possiede dizionari standard.
Come sostenuto dalle prime righe del paper [2] dov'è stato presentato, ArUco non è solo un sistema di rilevamento di marker, ma un vero e proprio generatore di dizionari personalizzato.

Gli autori sostengono alcuni punti fondamentali del loro lavoro nella generazione dei marker:

- Non esiste un set predefinito di marker;
- Un **dizionario** può essere generato contenendo solo il **numero di marker necessari**;
- Visto il punto 2, può essere **minimizzata** di volta in volta la **distanza tra marker** e il numero di bit di transizione;
- In base al numero di identificativi necessari, può essere scelto il numero di bit del marker che contiene l'informazione.

Sebbene il marker sia stato scelto come candidato da confrontare per quanto detto in precedenza, ci teniamo a condividere i risultati di un recente paper [3] che confronta i marker ArUco con AprilTag3.
Secondo i risultati di questa ricerca infatti, nella maggior parte dei casi le **prestazioni di Apriltag sono superiori**.

ArUco - pacchetto

Nonostante la popolarità di questo marker è comunque abbastanza **difficile trovare pacchetti** già sviluppati **per ROS**, funzionanti e con dell'adeguata documentazione riguardo l'utilizzo.

Inizialmente è stato utilizzato il pacchetto ROS presente al seguente repository: https://github.com/tuw-robotics/tuw_marker_detection.
Questo repository infatti si basa sulla ricerca iniziale [2] ed è sviluppato sul software ArUco presente alla pagina https://sourceforge.net/projects/aruco/, pagina gestita dagli sviluppatori del marker.
La limitazione introdotta dal suddetto pacchetto è quella di non riuscire ad utilizzare marker di dimensione diversa all'interno della stessa mappa.

Dopo alcune ricerche senza risultati proficui, si è deciso di passare alla versione sviluppata da **OpenCv**, come visto in [3]. Questa versione può vantare dizionari già precompilati della dimensione di **1000 marker**, contro i 250 della precedente versione di ArUco. La documentazione può essere trovata a:
https://docs.opencv.org/4.x/d9/d6d/tutorial_table_of_content_aruco.html.
A questo punto è stato sufficiente trovare un nuovo pacchetto basato questa libreria ed il risultato è stato:
https://github.com/CopterExpress/clover/tree/master/aruco_pose.

Questo pacchetto è gestito da un'azienda e al momento della scrittura della tesi il codice è tutt'ora mantenuto e **distribuito con licenza MIT**[1]. Inoltre quest'ultima scelta ci dà la possibilità di gestire marker di dimensione diversa e fornisce un nodo che, basandosi sui marker rilevati, stima la posizione in base all'origine di una mappa inserita.
Tutta la documentazione necessaria riguardo l'utilizzo può essere trovata al link precedente.

1) https://opensource.org/licenses/MIT

STag

Il marker è formato da un bordo quadrato ed un **pattern circolare al centro**: questo permette una stabilità delle stime di posizione migliore rispetto agli altri marker.

Infatti STag è stato ideato proprio per **contrastare** fenomeni di **jittering** di rilevazione dei tag, caratteristica che rende questo marker più preciso e stabile degli altri grazie anche al metodo di decodifica del tag adottato da questa famiglia di marker.

Tra i punti fondamentali che rendono STag appetibile e paragonabile ad altri tag più diffusi sono presenti:

- **Elevata frequenza di rilevazione**: migliora l'accuratezza della stima di posizione poiché c'è ridondanza delle informazioni ottenute;
- La forma ''ibrida'' del marker permette un'ottima rilevazione del tag anche a distanze elevate e con angoli di visione accentuati;
- STag permette di **specificare** quali **marker** possono essere **rilevati**;
- È una famiglia di marker sviluppata recentemente, questo ci porta a pensare che verrà mantenuta per più tempo e che possa ricevere molte migliorie con l'avanzare del tempo.

Un'altra motivazione per la quale STag è stato preso in considerazione è stata l'analisi del paper [4], dove viene dimostrato che questa famiglia di marker è paragonabile con ArUco e AprilTag in termini di prestazioni.
Ci teniamo solo ad evidenziare il fatto che purtroppo possieda un **utilizzo delle risorse della CPU** ben più **elevato** rispetto agli altri tag sopra citati, motivo che rende l'utilizzo di STag più limitato (fare riferimento al paper [1]).

STag - pacchetto

Per questo tag abbiamo utilizzato il pacchetto scaricabile da GitHub: https://github.com/usrl-uofsc/stag_ros.git

Essendo il **pacchetto** sviluppato per **ROS1**, è stato necessario utilizzare un bridge ROS1-ROS2 per poter effettivamente collegarlo con Gazebo: https://github.com/ros2/ros1_bridge.git

Nel nostro caso studio, abbiamo optato per la libreria di marker **HD15** che comprende ben 756 marker diversi, ma il pacchetto comprende ben 7 dizionari di marker ognuna con un numero di tag diverso al suo interno.

È possibile configurare i parametri relativi alla libreria in uso e ai marker mediante i file .yaml presenti all'interno della directory **cfg** del pacchetto stag_ros (ad esempio single.yaml e single_config.yaml).
In base ai file di configurazione impostati, è possibile far partire la detection dei marker mediante l'apposito file launch:

```
roslaunch stag_ros stagNode.launch
```

È possibile visualizzare anche quali marker sono identificati tra quelli imposti e la loro posizione rispetto alla camera mediante il topic:

```
rostopic echo /stag_ros/markers
```

N.B. : per la generazione del file single_config.yaml, che comprende tutti i tag presenti nella mappa, è stato utilizzato uno script che a partire dal file apriltag_virtual_map.yaml genera automaticamente il file .yaml pronto per essere inserito nella directory cfg del pacchetto stag_ros.

Whycode

Abbiamo scelto questo fiducial marker perché dal paper [5] risulta avere una **frequenza di rilevazione** molto **elevata**, una precisione di posizione e orientamento confrontabile con AprilTag ed addirittura **migliore all'aumentare dell'altezza**; tutto questo grazie alla forma circolare del marker che ne facilita la rilevazione.
Si può scegliere inoltre il numero di bit per la codifica, da cui dipende il numero di markers del dizionario. Per gli esperimenti sono stati scelti n = **13 bit**.
Il marker è diviso in n + 1 settori circolari, di cui uno è costante per garantire il riconoscimento della rotazione. La codifica si basa sulla teoria *Necklace Encoding*.

Per questo tag abbiamo utilizzato il pacchetto scaricabile da GitHub: https://github.com/gestom/whycon-orig/tree/whycon-ros-full.

Seguendo le indicazioni presenti nella pagina è possibile scaricare il programma e utilizzarlo, collegandolo ad una qualsiasi camera attraverso i driver appositi.

Whycode - pacchetto

Il pacchetto è stato sviluppato per **ROS1**, perciò è stato necessario utilizzare *ros1_bridge* per poter collegare i nodi di gazebo e di questo programma.

Digitando il comando, dopo aver modificato il file launch con il nome dei topic della camera, verrà avviato il programma:

roslaunch whycon_ros whycon.launch

Sarà possibile visualizzare l'immagine della camera con i tag identificati; si aprirà anche una finestra in cui sarà possibile cambiare alcuni parametri legati ai tag.

Con **rostopic echo /whycon_ros/markers** è possibile visualizzare i marker identificati e la loro posizione rispetto al centro della camera.

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

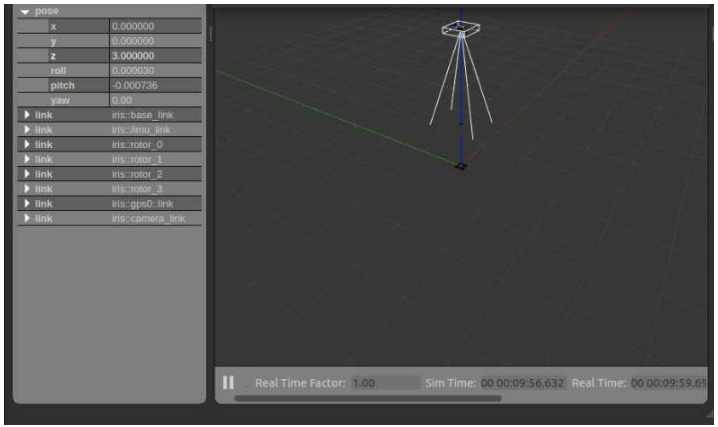
IMPOSTAZIONE AMBIENTE

- Setup iniziale
- PX4
- Gazebo
- Comandi offboard
- Generazione mappe
- Marker_to_visual_odometry

Setup iniziale

Al fine di valutare il corretto funzionamento dei pacchetti scelti, li abbiamo **testati tramite** la **webcam** del computer. Tramite questo test siamo riusciti effettivamente a capire se il tag veniva rilevato e riconosciuto dal pacchetto.

Il passaggio successivo è stato quello di utilizzare **Gazebo** in modo tale da poter impostare un ambiente di simulazione che ci potesse permettere di effettuare qualche **test statico**: per farlo abbiamo modificato la texture di asphalt_plane in modo tale da inserire un solo tag. A questo punto è stato sufficiente importare nella simulazione il **modello** di una qualsiasi **camera** (noi abbiamo optato per la camera classica di Gazebo), posizionarla nel punto desiderato nello spazio e verificare se le **trasformate** ottenute in uscita **combaciassero** con le **coordinate** impostate da noi precedentemente.



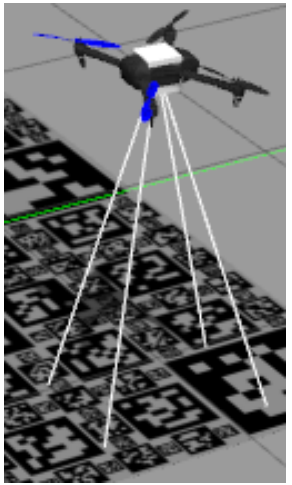
Pacchetto camera per ros2: https://gitlab.com/boldhearts/ros2_v4l2_camera
Installazione pacchetto gazebo_ros_pkgs (ROS2): https://classic.gazebosim.org/tutorials?tut=ros2_installing&cat=connect_ros

Setup iniziale

Dopo aver verificato il corretto funzionamento di Gazebo e l'attendibilità delle trasformate in uscita, abbiamo fatto in modo di **collegare** la **camera** di Gazebo con il modello del **drone**. Così facendo si muoveranno assieme e questo ci ritornerà particolarmente utile quando daremo i comandi offboard per eseguire dei movimenti preimpostati e automatici.

Il collegamento tra i due modelli è stato fatto modificando il file **iris.sdf** presente nella sezione models di PX4-Autopilot. La camera è stata posta al centro del modello del drone e collegata tramite un joint.

A questo punto abbiamo ripetuto i test statici precedenti con lo scopo di verificare se il collegamento camera-drone fosse saldo.
Dopo aver eseguito questi test, possiamo confermare quanto citato sui paper analizzati ad inizio progetto, ovvero che la stima di posizione relativa al rilevamento di un singolo marker è molto affidabile per tutti e 3 i marker in esame.



Simulazione

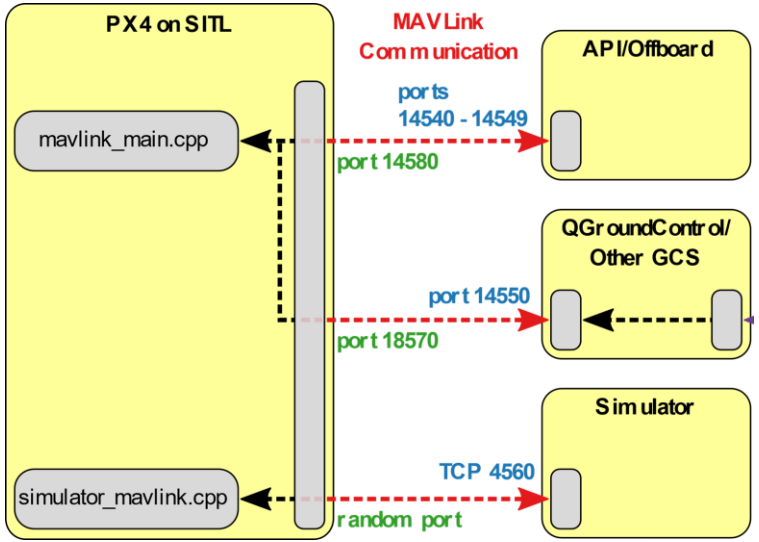
Approccio SITL (Software in the Loop)

Un nuovo approccio al testing è quello di **simulare l'hardware** montato all'interno del drone all'interno di un software. Questo ci permette in primo luogo di limitare i possibili danni ai nostri prototipi ed in secondo luogo ci dà la possibilità di eseguire diverse tipologie di testing con le più disparate condizioni.

Dal momento che *PX4* simula il comportamento reale del drone, è necessario avere anche il radiocomando per controllare lo UAV. Abbiamo quindi optato per l'utilizzo di un ulteriore software (***QGroundControl***) che simula il radiocomando.

Il simulatore usato è ***Gazebo***, uno dei più utilizzati in ambiente robotico.

Infine i comandi per muovere il drone potrebbero essere inviati utilizzando *QGroundControl,* ma un metodo più efficiente è quello di utilizzare script offboard come verrà spiegato successivamente.



I software da noi utilizzati sono stati:

- *PX4 Autopilot* (SITL) - https://px4.io/
- *Gazebo simulator* - https://gazebosim.org/home
- *QGroundControl* - http://qgroundcontrol.com/
- Comandi offboard personalizzati

PX4 - Mavlink

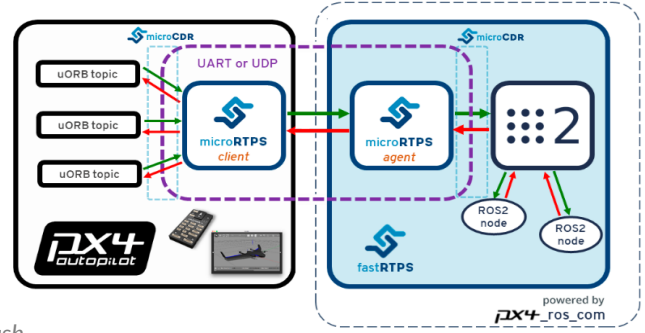
Mavlink è il **protocollo** di messaggi che viene utilizzato da PX4 per interfacciarsi con *QGroundControl*, i programmi offboard e il simulatore. Questo protocollo non solo è utilizzato per le comunicazioni tra la stazione di terra e il drone, ma anche per le comunicazioni tra le componenti del drone stesso. I **messaggi** *Mavlink* sono **definiti** attraverso dei file **XML**.
L'utilizzo di questo protocollo rende possibile avvicinare ancora di più la simulazione alla realtà.

Maggiori informazioni su questo protocollo e il suo utilizzo possono essere trovati alla pagina https://mavlink.io/en/.

MAVLINK
MICRO AIR VEHICLE COMMUNICATION PROTOCOL

PX4 - Ambiente

Per poter collegare PX4 a ROS2 è necessario utilizzare un **Bridge microRTPS**; esso permette uno scambio dati tra client ed agent in tempo reale.



Ora è necessario predisporre il workspace di ROS2 per l'interfacciamento con PX4:

1. Tramite il comando *git clone* copiare all'interno della cartella src i repository ai seguenti link:
   - https://github.com/PX4/px4_msgs
   - https://github.com/PX4/px4_ros_com
2. Commentare le righe 128 e 128 del file *px4_ros_com/scripts/build_ros2_workspace.bash*
3. All' interno della cartella *px4_ros_com/scripts/* eseguire il file tramite comando *bash build_ros2_workspace.bash*
4. Installare micrtoRTPS seguendo le indicazioni alla pagina https://docs.px4.io/main/en/dev_setup/fast-dds-installation.html#fast-rtps-gen (**N.B.** La cartella dove viene clonato il repository git non è rilevante.)
5. (**Facoltativo**) nel caso l'ultimo comando non funzionasse, prima di riprovare, da terminale eseguire il comando *sudo apt install ros-galactic-rmw-fastrtps*
   Se non funziona seguire i sottopunti
   - Scaricare in qualsiasi posizione il file alla pagina https://github.com/PX4/PX4-Autopilot/blob/main/Tools/setup/ubuntu.sh
   - Rendere il file eseguibile tramite il comando chmod +x ubuntu.sh
   - Eseguire da terminale il comando wget https://raw.githubusercontent.com/PX4/PX4-Autopilot/main/Tools/setup/requirements.txt
   - Eseguire il file tramite il comando bash ubuntu.sh --no-nuttx
6. Eseguire il comando sudo apt update
7. All' interno della cartella *px4_ros_com/scripts/* eseguire il file tramite comando *bash build_ros2_workspace.bash*
8. Entrare all'interno del workspace di ROS2
9. Compilare il workspace con il comando:

   colcon build --cmake-args -DCMAKE_BUILD_TYPE=RELWITHDEBINFO --symlink-install --packages-skip ros1_bridge

N.B. Per fare in modo che il bridge funzioni, è necessario lanciare l'agent microRTPS nel workspace ROS2 con il comando micrortps_agent -t UDP

PX4 - Firmware

PX4 è un firmware tuttora in continuo aggiornamento da parte degli sviluppatori che implementano man mano nuove funzionalità.

La sua installazione è semplice:

1. Clonazione del repository git tramite il comando `git clone`
2. Da terminale ci si sposta all'interno della cartella appena clonata
3. **(Facoltativo)** Nel nostro caso ci siamo dovuti spostare in un altro branch con `git checkout`
4. Avviare la compilazione del firmware tramite `make px4_sitl_rtps gazebo`

   È inoltre possibile compilare altri tipi di ambienti e modelli presenti all'interno della cartella «Models» di PX4. Alcune delle opzioni di simulazione disponibili sono visionabili alla pagina http://docs.px4.io/main/en/simulation/gazebo.html.
5. A questo punto PX4, una volta compilato, si occupa dell'avvio dell'intero ambiente di simulazione e nel terminale dal quale è stato lanciato il comando viene avviata la shell di PX4.
6. **(Facoltativo)** Nel nostro caso, a causa di problemi nel takeoff, al primo avvio è stato necessario cambiare alcuni parametri tramite shell PX4 come riportato nel seguente issue: https://github.com/PX4/PX4-Autopilot/issues/19919

Repository: https://github.com/PX4/PX4-Autopilot Branch utilizzato: 10a2b4c9f7d34234459b4d3604b99a33491a9d83

Gazebo

La simulazione dell'ambiente viene lanciata direttamente da PX4.

Quando viene lanciata un'istanza di Gazebo dalla shell di PX4 è possibile vedere, in base al comando, quali sono stati i modelli utilizzati per la creazione del mondo.
Il comando citato in precedenza utilizza:

- Empty_world
- asphalt_plane
- IRIS

Il modello «Empty_world» carica un mondo vuoto, come dice il nome, all'interno del quale troviamo una superficie di asfalto («asphalt_plane» per l'appunto) e il modello del quadricottero «IRIS».
Come detto in precedenza, questi due ultimi modelli sono stati modificati in base alle nostre esigenze.

Tutti i modelli possono essere trovati all'interno della cartella: PX-Autopilot/Tools/sitl_gazebo/models

Offboard

Ad occuparsi della pubblicazione dei comandi offboard è il pacchetto **px4_ros_com**. Quest'ultimo infatti è l'interfaccia di connessione di ROS2 con PX4 attraverso il bridge microRTPS.
Alcuni esempi di codice per la generazione di comandi offboard possono essere trovati nella cartella px4_ros_com/src/examples/offboard.

I comandi offboard sono solo uno dei tanti metodi per poter comandare un drone.
Con questo metodo i comandi vengono passati al drone sotto forma di una **traiettoria di punti** da seguire nel tempo, con una frequenza temporale impostabile. In questo modo oltre che la posizione futura è possibile anche **imporre la velocità** con cui il drone deve muoversi.

Tutte le posizioni e l'imbardata del drone passate attraverso i comandi offboard, nel caso di una simulazione SITL, fanno riferimento all'origine di generazione del drone nel mondo Gazebo.

Nel nostro caso, dopo aver compreso come funzionasse la generazione dei percorsi del drone, ci è stato permesso di basarci su uno script personalizzato per creare i nostri percorsi; questo script infatti permette di creare movimenti più omogenei grazie ad un interpolazione polinomiale di quinto grado tra i punti.

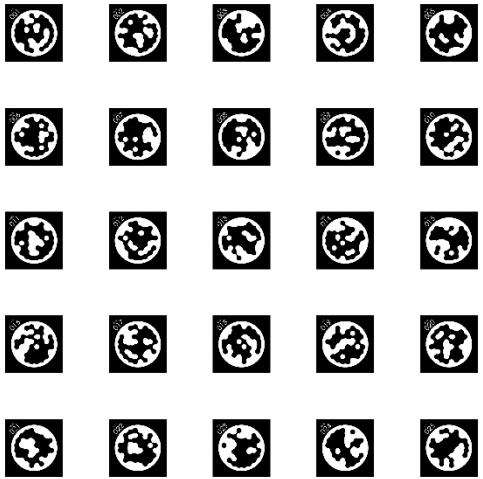
Generazione mappa

Inizialmente, dopo aver fatto il test con il singolo marker, è stata generata una mappa di dimensioni 3x3 metri contenente 25 marker di dimensione identica (0,4x0,4 metri) posizionati in maniera equidistante.

Questa decisione ci è sembrata subito non proseguibile in quanto ci sarebbe stata un'**altezza minima** alla quale il drone non sarebbe riuscito a determinare la posizione poiché non vedrebbe interamente i marker.

Il passo successivo è stato quello di utilizzare un **generatore di mappe** precedentemente sviluppato. Questo codice prende in ingresso un dizionario di marker (una cartella con dentro tutte le immagini dei marker) e in base a dei parametri che vengono impostati (come numero massimo di marker per ogni taglia e dimensione in metri delle varie taglie) genera una mappa con origine nell'angolo in alto a destra.
La mappa generata è stata successivamente convertita in .png e utilizzata come «asphalt_plane» in *Gazebo*. In questo modo, sapendo che la mappa viene posizionata al centro del mondo di *Gazebo*, siamo riusciti a calcolare la posizione dell'**origine della mappa rispetto a tale punto** e risulta {1,5; 1,5; 0} metri.

Il generatore di mappe inoltre genera anche un file .yaml contenente tutte le posizioni dei marker rispetto all'origine della mappa.



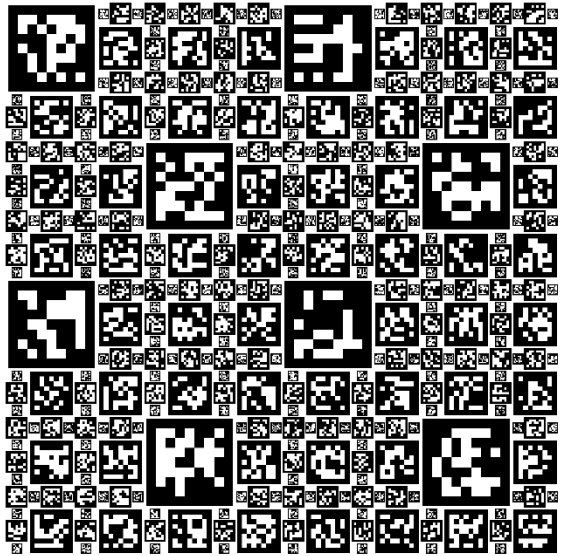
Mappa 3x3 con 25 marker STag

Lo script utilizzato è «Generator_map»

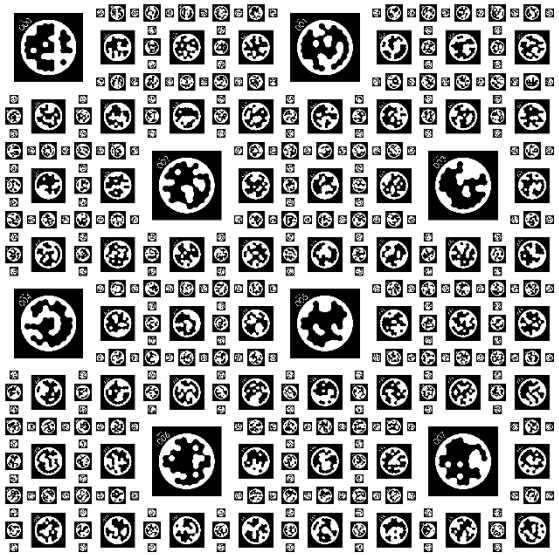
Mappe utilizzate

Dimensioni marker:

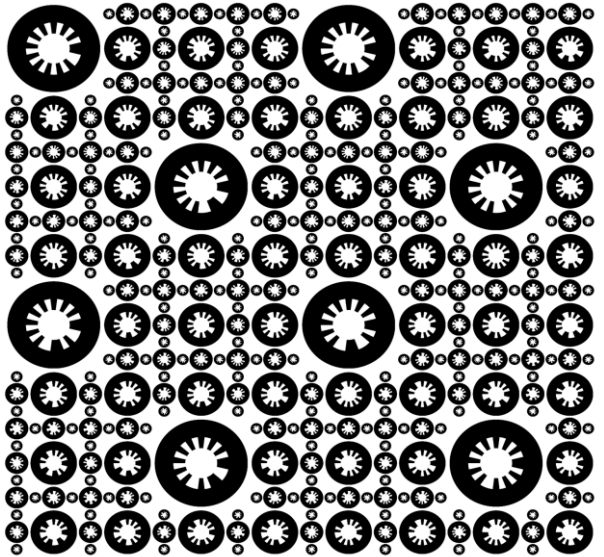
Taglia XL = 0,46m; **Taglia L = 0,23m;** **Taglia M = 0,115m;** **Taglia S = 0,0575m**



ArUco



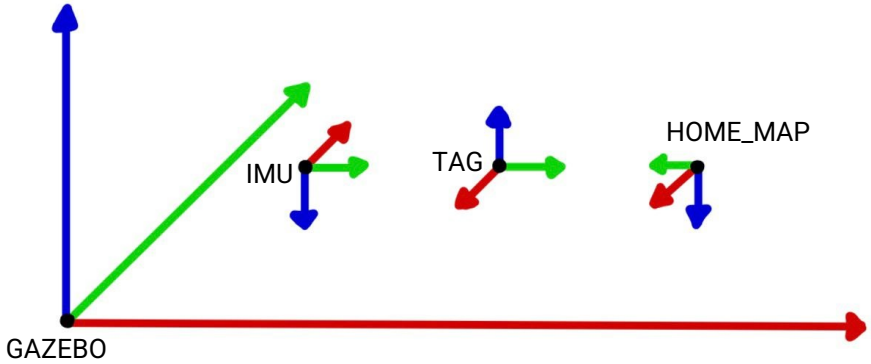
STag

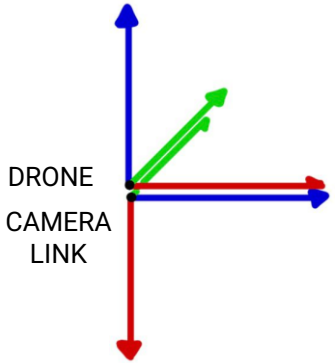


Whycode

Visual odometry

PX4 fornisce come topic in uscita */fmu/vehicle_odometry/out*; questo topic ci permette di conoscere la **posizione istantanea** del drone basandoci sui **movimenti inerziali** rilevati dalla **IMU** a partire dall'origine di generazione del drone nel mondo di *Gazebo*. Questa posizione, non essendo impostabile su PX4, è stata misurata sulla base della posizione iniziale del drone una volta aperto il simulatore; essa risulta sempre a {1; 1; 0} metri rispetto all'origine di *Gazebo*, con un errore massimo di 0,025 metri su tutti gli assi.
Attraverso alcuni comandi offboard inoltre siamo riusciti a ricostruire il sistema di riferimento della IMU, come visibile nello schema sottostante.





Tutti i pacchetti dei marker testati offrono in uscita una trasformata che corrisponde alla **posizione** e **rotazione relativa** di un marker rispetto alla camera montata sul drone.
Viene pubblicata una trasformata per ogni marker che il pacchetto riesce a riconoscere nelle immagini pubblicate dalla camera.

N.B. Le origini di camera link e drone sono solo ruotate tra loro, lo spostamento è puramente per scopo illustrativo degli assi

Visual odometry

Ricordando quanto detto in precedenza, ovvero che grazie al file .yaml generato conosciamo la posizione di ogni marker rispetto all'origine della mappa, possiamo calcolare la posizione del drone rispetto ad essa in ogni suo instante.

A questo punto conosciamo la posizione del drone rispetto all'origine IMU (che chiameremo ***Odometry***) e la posizione del drone rispetto all'origine della mappa data dai marker (***Visual_odometry***). Prendendo un terzo sistema di riferimento siamo in grado di calcolare la posizione reciproca dei due sistemi di riferimento; per fare ciò utilizzeremo l'**origine del mondo di Gazebo** dal momento che conosciamo la posizione delle altre due origini rispetto a quest'ultima.

Il pacchetto *apriltag_to_visual_odometry*, già sviluppato in precedenza, permette per l'appunto di riferire entrambe le trasformate alla stessa origine. Per fare ciò utilizza una **somma di rotazioni di vettori**, con i **quaternioni riportati al sistema finale**, come si può vedere nella formula sottostante:

$$Drone\_origin = quatRotate(q\_imu\_to\_origin, offset\_camera\_to\_origin) + quatRotate(q\_imu\_to\_camera, tf\_marker\_to\_camera) + quatRotate(q\_imu\_to\_marker, offset\_home\_map\_to\_marker) + offset\_imu\_to\_home\_map;$$

Per il calcolo della posizione serve solo una trasformata di un qualsiasi marker; a questo punto il pacchetto seleziona una tra quelle dei **marker disponibili** con la **taglia più grande**.
Alla fine del processo viene pubblicato un topic (*Visual_odometry*) confrontabile con il topic *Odometry* della IMU.
Questo processo porta a qualche errore di posizione dovuto innanzitutto **all'errore di posizione iniziale del drone** e in secondo luogo alle varie approssimazioni eseguite all'interno delle somme e moltiplicazioni.

N.B. La funzione quatRotate() calcola il vettore ruotato a partire da un quaternione e un vettore

SETUP SPERIMENTALE

- Test eseguiti
- Quadrato
- Variazione altitudine
- Serpentina
- Serpentina con variazione altitudine

Test eseguiti

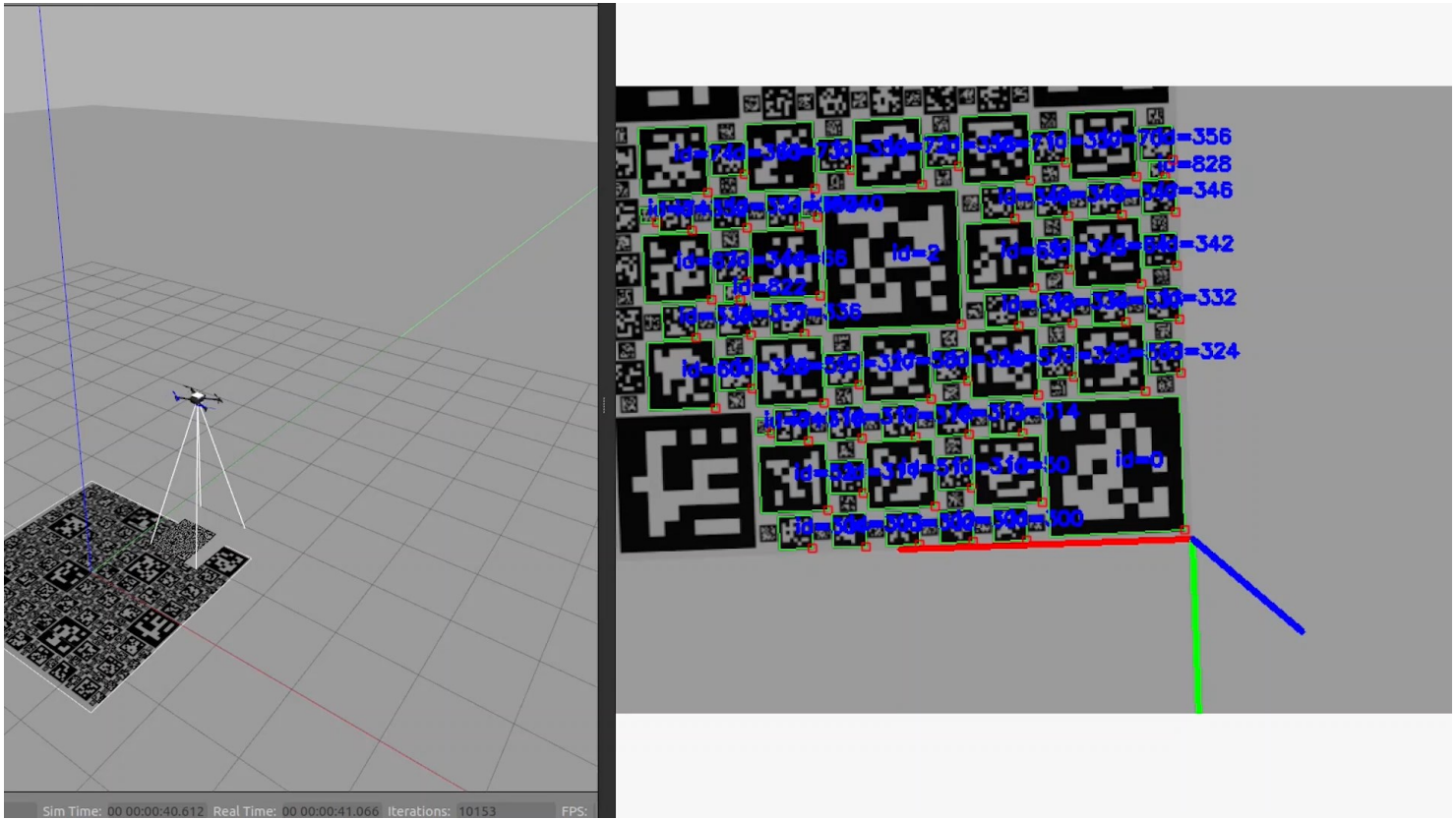
Per quantificare la precisione di ogni marker abbiamo deciso di effettuare dei test utilizzando un drone in movimento, per poter rendere il più realistico possibile lo scenario di utilizzo dei fiducial marker:

- Percorso lungo un quadrato di lato 2 metri a quote diverse;
- Movimenti lungo l'asse verticale, mantenendo costante la posizione sugli altri due assi;
- Percorso "a serpentina", simile al test del quadrato ma cambia il movimento;
- Percorso "a serpentina" con cambio graduale di quota;
- Percorso a zigzag;
- Percorso quadrato con velocità doppia;
- Percorso "a serpentina verticale" con velocità doppia.

La scelta di questi movimenti ci permette di poter osservare il comportamento delle rilevazioni sotto aspetti diversi:

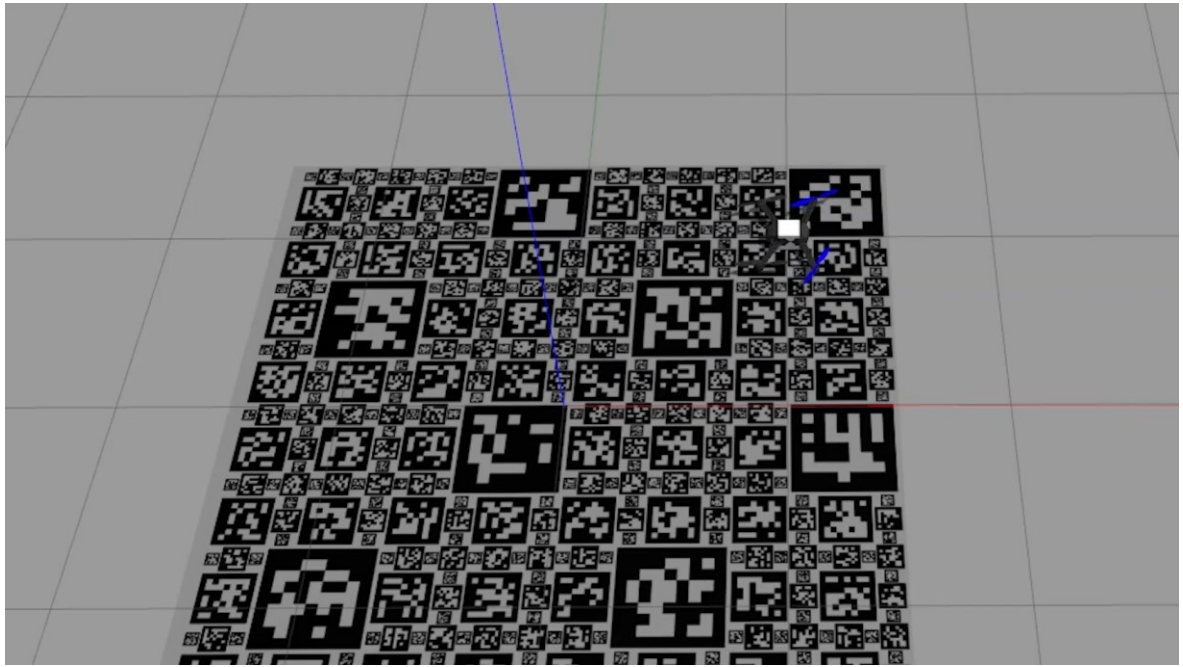
- Il percorso quadrato è il movimento più basilare che ci permette di testare un movimento lungo un solo asse;
- Il percorso "a serpentina" ci permette di vedere il comportamento con sequenze di marker diverse dal percorso quadrato;
- Il percorso verticale ci permette di testare i marker a varie altezze;
- Il percorso "a serpentina" con cambio di quota ci permette di combinare i due punti precedenti.

Test - video


Sim Time: 00 00:00:40.612 Real Time: 00 00:00:41.066 Iterations: 10153 FPS:
id=356
id=828
id=346
id=342
id=332
id=324
id=114
id=0
id=100

Test - problematiche

Purtroppo durante la fase di test all'improvviso PX4 non riusciva più a gestire la corretta orientazione del drone. Infatti, come visibile nel video sottostante, il drone dopo aver eseguito la rotazione imposta, esegue un piccolo cambiamento dell'angolo dell'imbardata non riconosciuto dalla IMU.



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

MATLAB E PLOTJUGGLER

- Salvataggio topic ROS
- Analisi MatLab

Salvataggio topic ROS

Per salvare i dati durante i test utilizziamo il comando:

ros2 bag record <nome_topic>

A questo punto utilizziamo il programma *"plotjuggler"* per poter estrarre i dati dalle bag ed esportarli in formato *csv*; questo programma è molto utile perché è in grado di allineare temporalmente i dati in basi ai valori di tempo derivanti dai messaggi, ponendo a *"NaN"* le caselle vuote.

Successivamente selezioniamo i dati di interesse, ovvero le componenti dei quaternioni, di posizione e il valore *"timestamp_sample"* che contiene l'informazione temporale, e importiamo i dati su MATLAB.

Matlab

Abbiamo creato lo script *“letture_csv.m”* per:

- interpolare le caselle di dati con valore *“NaN”* e avere degli array di dati omogenei;
- costruire un grafico delle traiettorie misurate dai due topic;
- misurare errore medio, deviazione standard e frequenza dei due topic.

A questo punto è possibile confrontare i diversi marker nelle condizioni scelte.

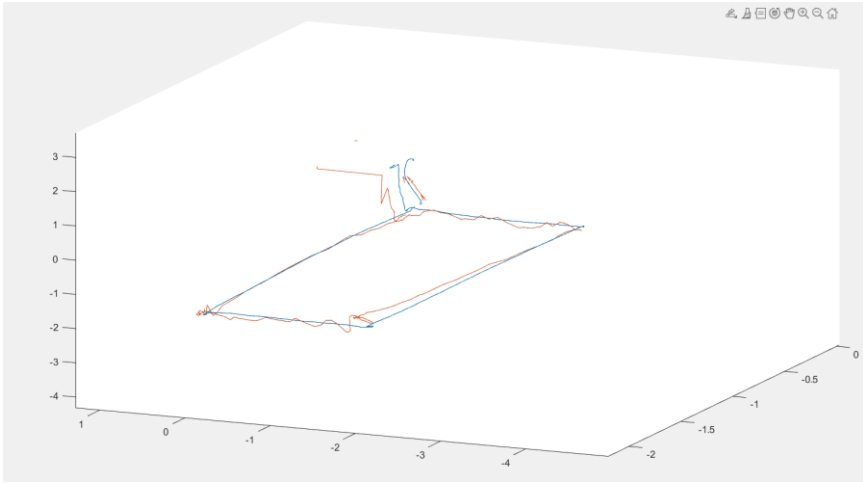
UNIVERSITÀ
DEGLI STUDI
DI PADOVA

RISULTATI

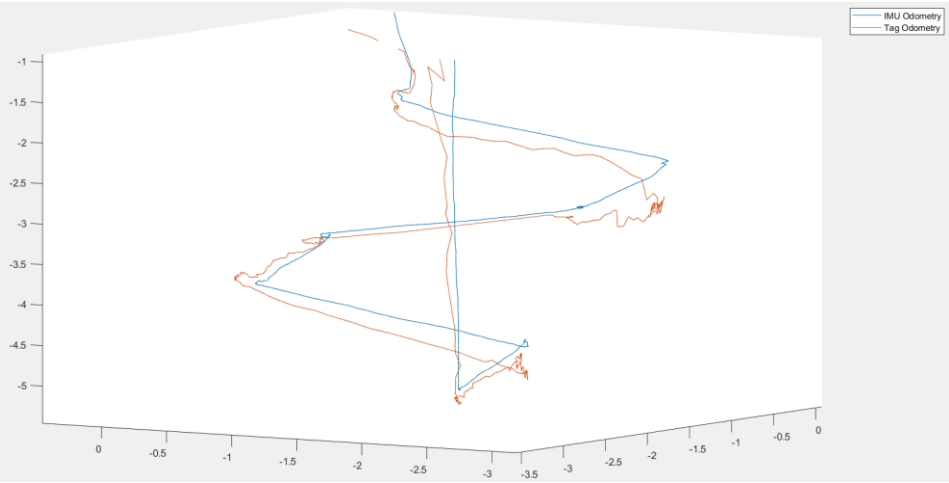
- ArUco
- STag
- Whycode
- STag vs ArUco

ArUco

Nel complesso ha una precisione abbastanza buona, pecca nei movimenti strettamente verticali oltre i 2m di distanza in z, in cui la rilevazione fallisce e peggiorano le prestazioni.
Rate medio informazione di posizione: 40-60 Hz

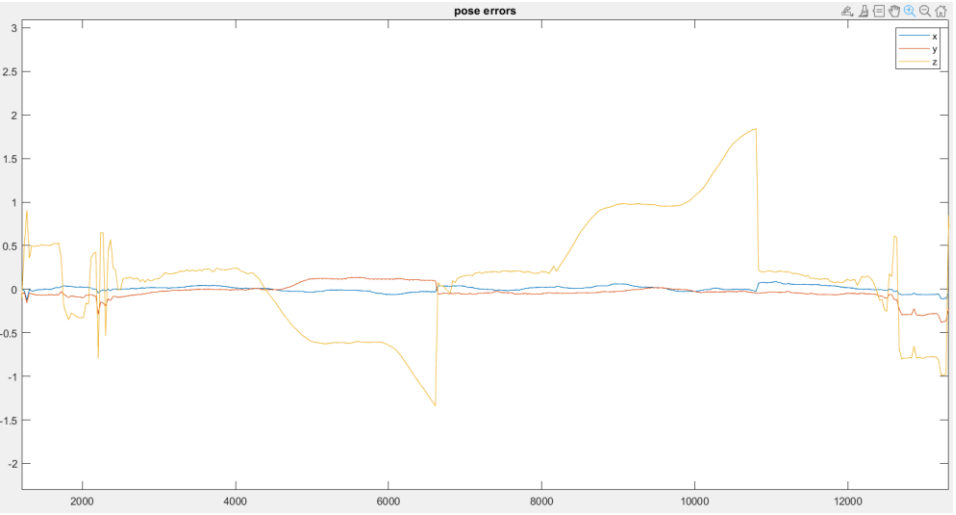


Traiettoria square_15

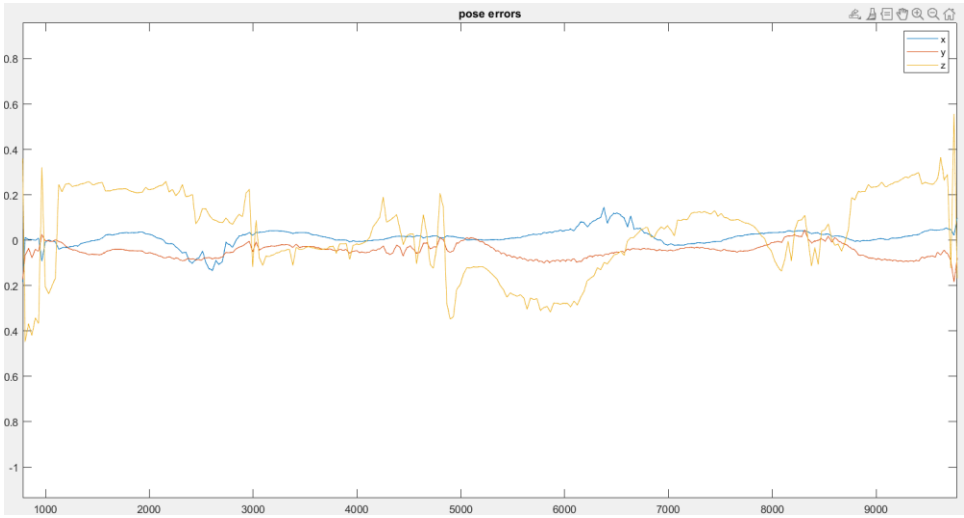


Traiettoria vert_s

ArUco

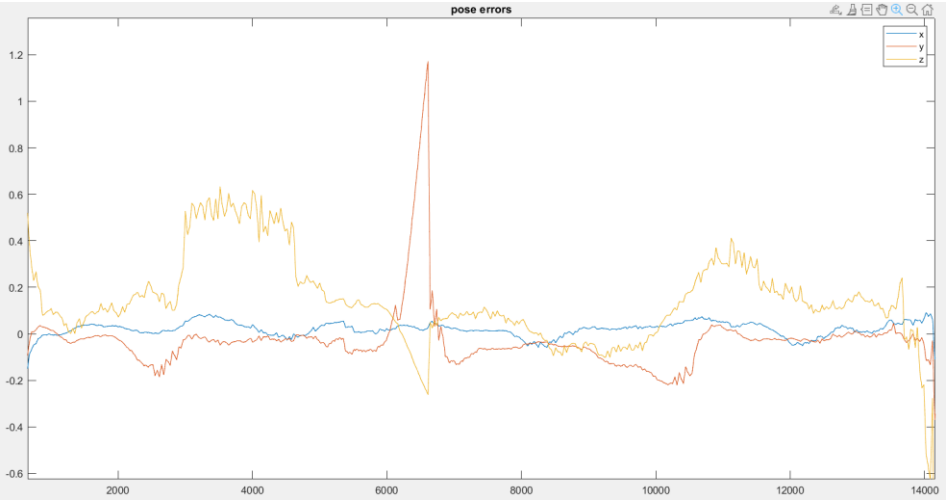


Errore posizione traiettoria verticale

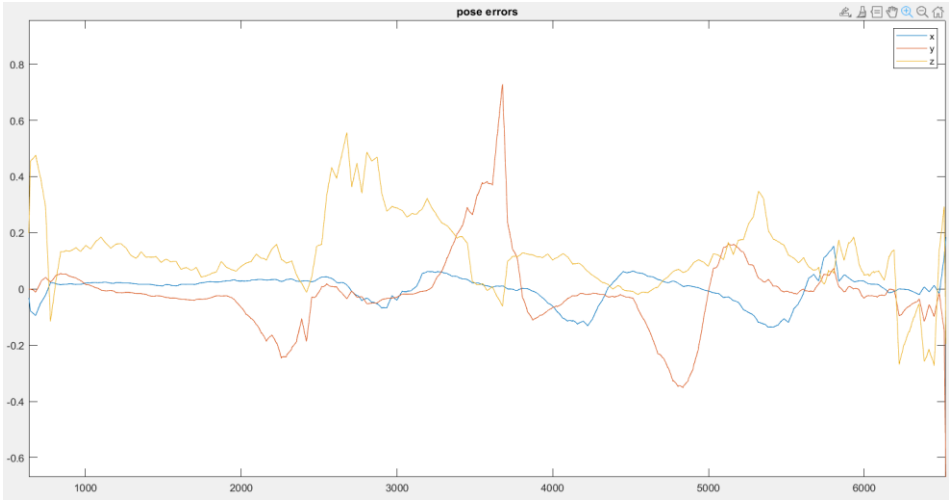


Errore posizione per traiettoria quadrata a 1,5m

Confronto a diverse velocità



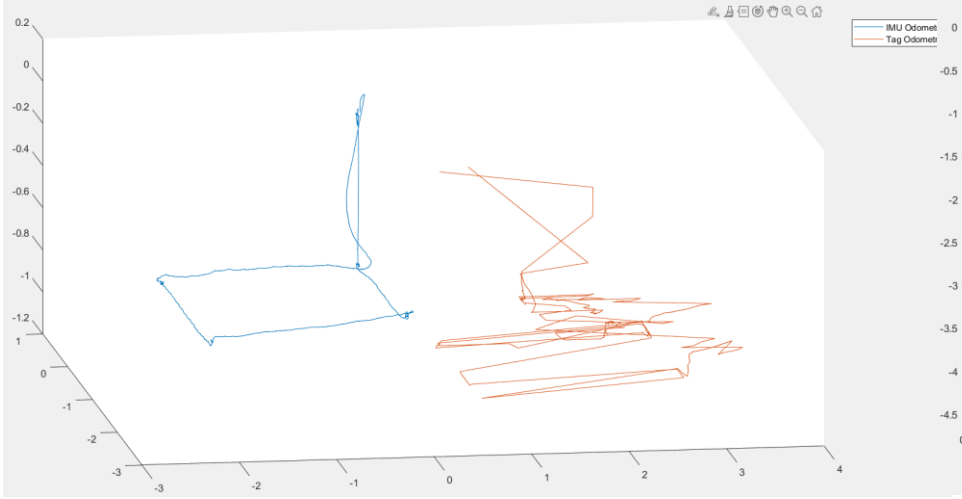
Traiettoria S verticale lenta



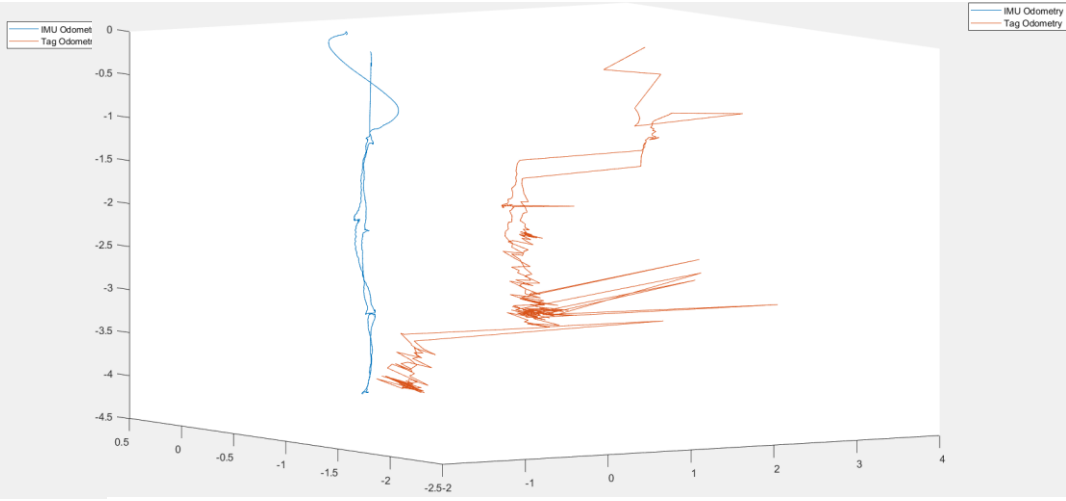
Traiettoria S verticale velocità doppia

Per velocità sostenibili per il frame-rate della camera, la velocità di movimento del drone non influisce molto sulla precisione di posizione.

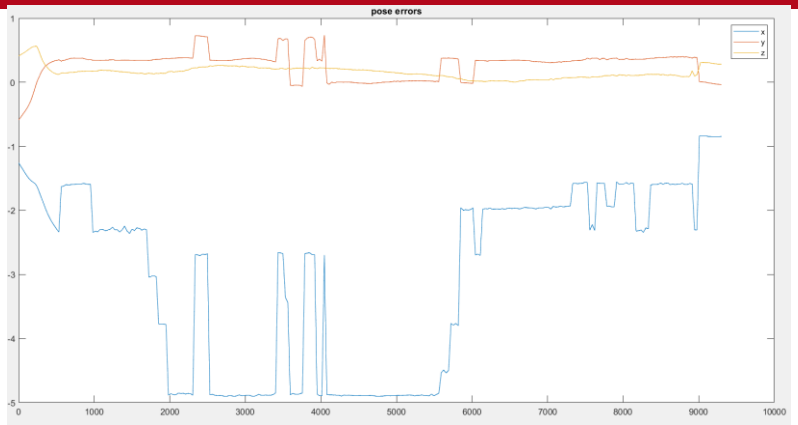
Prestazioni scarse in x, che migliora all'aumentare di z (contro le nostre aspettative), grande instabilità nonostante il pacchetto dovrebbe essere in grado di riconoscere marker di diverse dimensioni.
Ipotizziamo come causa di queste performance un elevato uso delle risorse oppure un codice non ottimizzato per questo utilizzo.
Rate medio informazione di posizione: 44 Hz



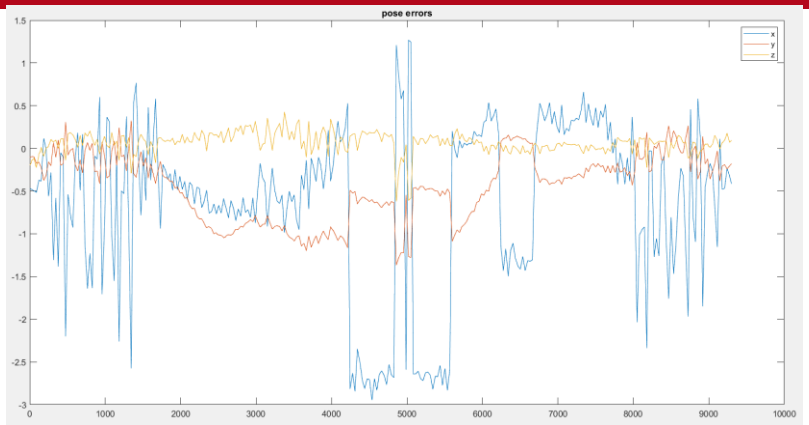
Traiettoria square_15



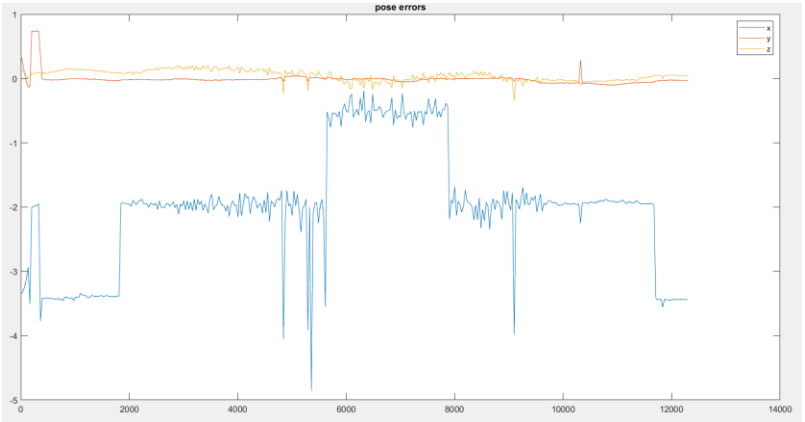
Traiettoria verticale



Errore di posizione per traiettoria quadrata a 0,7m

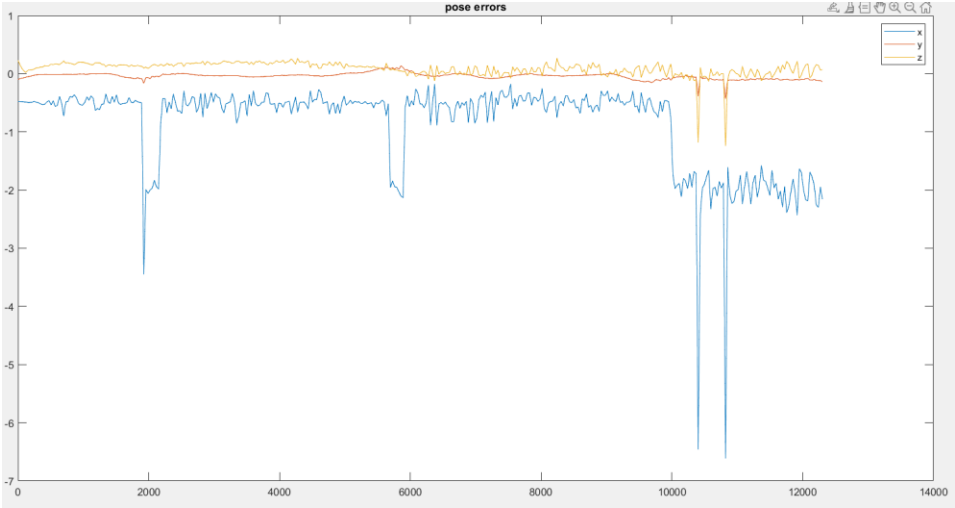


Errore di posizione per traiettoria quadrata a 5m

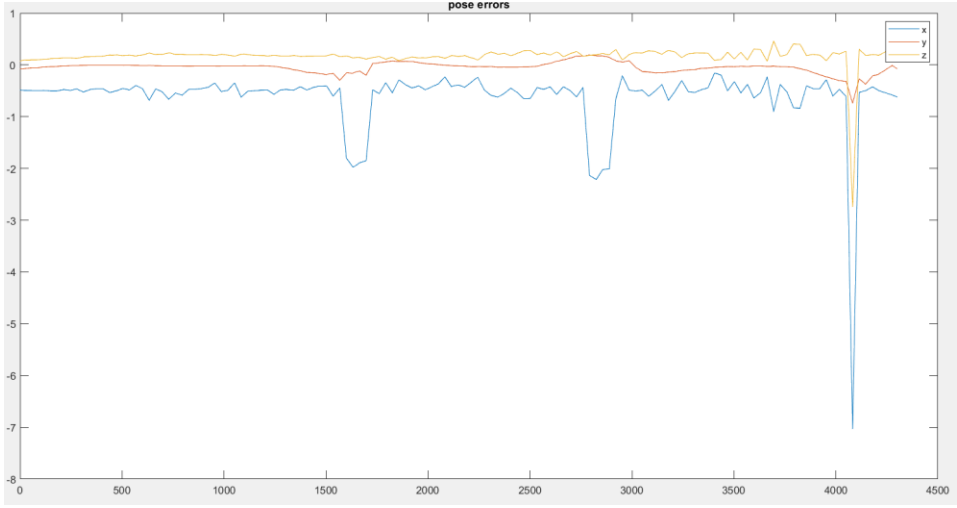


Errore di posizione per traiettoria verticale

Confronto a diverse velocità



Errore di posizione per traiettoria S verticale



Errore di posizione per traiettoria S verticale a velocità doppia

Le prime prove di test svolte con questo pacchetto hanno portato alla luce un errore nel codice, che comportava una **componente z** del quaternione ricavato dalle trasformate **sempre pari a 0**.

A questo punto abbiamo provato ad utilizzare altri pacchetti, disponibili su Github, come https://github.com/jiriUlr/whycon-ros e i suoi branch, ma in questi casi abbiamo notato degli **errori** molto consistenti nella corretta **rilevazione degli ID dei tag**, fondamentali per la localizzazione del drone, nel caso soprattutto di mappe con elevata quantità di marker.

Considerando che questo marker è solitamente utilizzato per la localizzazione di oggetti negli ambienti “realtà aumentata”, è possibile che **questi pacchetti non possano performare correttamente quando è necessario analizzare molti marker contemporaneamente**.

Inoltre, per lo stesso motivo, il pacchetto non è in grado di analizzare markers di dimensione diversa nella stessa immagine.

ArUco vs STag

ArUco

Mean_x	1,89cm	Std_x	4,20cm
Mean_y	3,64cm	Std_y	11,56cm
Mean_z	14,48cm	Std_z	25,60cm

STag

Mean_x	154,00cm	Std_x	120,90cm
Mean_y	15,01cm	Std_y	15,28cm
Mean_z	10,61cm	Std_z	8,79cm

Da questi esperimenti risulta ovvia la possibile scelta.
Nel caso si risolvesse il problema in x per STag risulterebbero circa confrontabili.
Non è stata svolta l'analisi del quaternione per il problema esposto precedentemente riguardo all'errore di rotazione del drone da parte della IMU e per degli errori di allineamento dei sistemi di riferimento in apriltag_to_visual_odometry.cpp.

800 1222·2022 ANNI

UNIVERSITÀ DEGLI STUDI DI PADOVA

BIBLIOGRAFIA

Bibliografia

[1] M. Kalaitzakis, B. Cain, S. Carroll, A. Ambrosi, C. Whitehead, N. Vitzilaios: Fiducial Markers for Pose Estimation, Journal of Intelligent & Robotic Systems, 26 Marzo 2021

[2] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez. 2014. "Automatic generation and detection of highly reliable fiducial markers under occlusion". Pattern Recogn. 47, 6 (June 2014), 2280-2292. DOI=10.1016/j.patcog.2014.01.005

[3] J. Kallwies, B. Forkel and H. -J. Wuensche, "Determining and Improving the Localization Accuracy of AprilTag Detection," 2020 IEEE International Conference on Robotics and Automation (ICRA), 2020, pp. 8288-8294, doi: 10.1109/ICRA40945.2020.9197427.

[4] B. Benligiray, C. Topal, C. Akinlar, STag: A stable fiducial marker system, Image and Vision Computing 89, (2019)

[5] P. Lightbody, T. Krajnìk, M. Hanheide: A Versatile High-Performance Visual Fiducial Marker Detection System with Scalable Identity Encoding, Lincoln Centre for Autonomous Systems Research, School of Computer Science, University of Lincoln, United Kingdom, SAC 2017, April 03-07, 2017, Marrakech, Morocco