



UNIVERSITÀ DEGLI STUDI DI FIRENZE  
SCUOLA DI INGEGNERIA - DIPARTIMENTO DI INGEGNERIA  
DELL'INFORMAZIONE

---

Tesi di Laurea Triennale in Ingegneria Informatica

**Sviluppo e valutazione di un sistema  
SLAM basato su ORB \_ SLAM3 su ROS**

*Candidato*  
Emanuele Nencioni

*Relatore*  
Prof. Carlo Colombo

*Correlatore*  
Marco Fanfani, PhD

---

Anno Accademico 2021/2022

# Indice

<b>Introduzione</b>	<b>i</b>
<b>1 ORB SLAM e ROS</b>	<b>1</b>
1.1 SLAM . . . . .	1
1.1.1 Visual SLAM . . . . .	3
1.2 ORB SLAM . . . . .	6
1.2.1 Features ORB . . . . .	6
1.2.2 Tracking and local mapping . . . . .	8
1.2.3 Loop closure . . . . .	9
1.2.4 conclusioni . . . . .	11
1.3 ORB SLAM 2 . . . . .	11
1.4 ORB SLAM3 . . . . .	14
1.4.1 Differenze tra ORB SLAM 2 e 3 . . . . .	14
1.4.2 <i>Visual-intertial</i> SLAM . . . . .	17
1.4.3 Maggiori dettagli sulle nuove funzionalità . . . . .	17
1.5 ROS . . . . .	19
1.5.1 ROS Filesystem Level . . . . .	19
1.5.2 ROS Computational Graph Level . . . . .	20
<b>2 Creazione wrapper di ORB SLAM3</b>	<b>25</b>

2.1	Sistema Autonomo nella Formula Student . . . . .	25
2.1.1	Passaggio da ORB SLAM2 a ORB SLAM3 . . . . .	28
2.2	orb_slam3_ros_wrapper . . . . .	29
2.2.1	Architettura . . . . .	31
2.2.2	Refactoring . . . . .	35
<b>3</b>	<b>Valutazione</b>	<b>47</b>
3.1	Valutazione odometrica . . . . .	48
3.1.1	<i>The KITTI Vision Benchmark Suite</i> . . . . .	48
3.1.2	<i>Structure from Motion</i> . . . . .	56
3.2	Valutazione computazionale . . . . .	60
3.2.1	Confronto con ORB SLAM3 nativo . . . . .	60
3.3	Conclusioni . . . . .	63
	<b>Bibliografia</b>	<b>65</b>

# Introduzione

La Formula Student è una competizione di ingegneria che si svolge in tutto il mondo e che coinvolge studenti universitari di ingegneria meccanica, elettrica e informatica. A Firenze, il team di formula Student è il Firenze Race Team. Lo scopo della competizione è quello di progettare, costruire e testare una vettura da corsa monoposto con o senza pilota (*driverless*). Una delle sfide principali nella progettazione di questi veicoli, è la loro capacità di navigare autonomamente in ambienti sconosciuti. Una tecnologia chiave che viene utilizzata per risolvere questa sfida è nota come SLAM (*Simoultaneous Localization and Mapping*), che consente ai veicoli di creare una mappa dell'ambiente circostante e di determinare la loro posizione all'interno di esso, in tempo reale. Questa tesi si focalizza sullo sviluppo di un *wrapper* per usare ORB SLAM3 in ROS e il confronto con ORB SLAM2, già in uso, modificato per farlo funzionare nel contesto della Formula Student. In questa tesi si cercherà di esporre il progetto, comprendere quali vantaggi e svantaggi si ottengono dalla versione originale di ORB SLAM3 e dalla vecchia generazione. Infine per capire se è valido o meno, si analizzerà in dettaglio gli errori di precisione risultanti dalla traiettoria che i due algoritmi forniscono e anche valutazioni computazionali che sono fondamentali, dato che nel veicolo non è possibile utilizzare hardware troppo potente per eccessivo consumo o peso.



# Capitolo 1

## ORB SLAM e ROS

Prima di iniziare l'esposizione del progetto occorre soffermarsi sulla definizione di un sistema SLAM. Si partirà con una definizione generale di SLAM e di ORB SLAM [6] per poi passare ad analizzare in dettaglio i sistemi slam precedentemente citati. Infine verrà analizzato ROS.

### 1.1 SLAM

SLAM sta per Simoultaneous Localization and Mapping, cioè contiene due operazioni principali:

- ***Mapping***: costruzione di una mappa dell'ambiente circostante. Questa avviene in seguito alla raccolta dati prelevati vari sensori presenti sul veicolo o sul robot.
- ***Localization***: calcolo della posa, cioè della posizione e orientamento del dispositivo dopo aver conosciuto l'ambiente intorno ad esso stesso. Di solito è necessario avere prima una mappa dell'ambiente circostante per ricavare la posizione del veicolo.

Ci sono vari tipi di sistemi SLAM, ciascuno, con i suoi pro e contro, utilizza diverse strategie, tecniche, algoritmi per riuscire a realizzare in modo migliore possibile le due operazioni sopra citate. In generale si hanno dei sistemi in grado di prendere dati da sensori diversi, altri che si focalizzano su un unico o pochi sensori. I sensori più comunemente usati sono:

- **Camere:**
  - **monoculari:** un'unica camera in bianco e nero o a colori. Permette di vedere l'ambiente circostante ma non di trarne troppe informazioni, poiché essendo una camera sola, le informazioni sulla distanza degli oggetti sono difficili da trovare. Sicuramente è un sensore che vede largo utilizzo quando si hanno altri sensori.
  - **RGB-D:** camere monoculari dotate anche di un proiettore a luci infrarosse (IR), capace di calcolare la distanza degli oggetti vicini, utile per ambienti di piccole dimensioni.
  - **Stereo:** l'uso di due o più fotocamere permette, scattando immagini contemporaneamente, di triangolare la posizione di oggetti in modo molto accurato, anche a distanze elevate(sopra i 20 metri).
- **Lidar:** (*Light Detection and Ranging*) sensore che proietta un laser sulle superfici tutte intorno a sé. Per ricavare la misura, viene calcolato l'intervallo tra quando il laser è stato sparato a quando, una volta colpita una superficie, la luce rimbalza ritornando indietro fino al lidar. Utilizzando questa informazione, questo sensore può calcolare la distanza degli oggetti o superfici intorno a sé con un'elevata precisione, raggiungendo anche i 100 metri di range di utilizzo.
- **IMU:** *Inertial Measurement Unit*, unità che contiene un giroscopio ed un accelerometro per avere informazioni delle accelerazioni sui 3 assi e le

rispettive velocità angolari. Le fotocamere moderne (Stereo o RGB-D), ormai, sono quasi tutte equipaggiate di questo sensore. Questo permette di compensare i movimenti della fotocamera durante l'acquisizione di immagini così da ottenere immagini più nitide. Però solitamente non sono molto accurati per essere usati nei sistemi SLAM.

- **GPS:** sistema di posizionamento globale su base satellitare per determinare la posizione di un oggetto sulla Terra. Sistemi di questo tipo possono arrivare anche ad una precisione altissima, spesso vengono utilizzati insieme ad IMU e LIDAR per migliorare di più la mappatura.

A seconda dell'utilizzo, è più o meno conveniente usare una combinazione di questi sensori. Ad esempio, usare tutti questi sensori potrebbe portare ad un eccessivo uso della potenza computazionale di un calcolatore, che deve perdere tempo a confrontare e unire tutte le informazioni diverse.

### 1.1.1 Visual SLAM

In particolare si analizzerà in dettaglio ORB SLAM [6], che di fatto implementa una tecnica chiamata *Visual SLAM*, un particolare tipo di SLAM in cui si utilizzano una o più fotocamere per costruire e mantenere una mappa dell'ambiente, senza usare nessun altro tipo di sensore. Tramite solo le immagini, questi sistemi riescono a ricostruire l'ambiente e a determinare la posa del veicolo.

Questi sistemi SLAM utilizzano una tecnica chiamata *feature extraction* per estrarre dei punti distinti dell'immagine, detti *keypoints*. Ogni *keypoint* ha poi un descrittore, che serve per identificarlo univocamente, solitamente i descrittori sono generati raccogliendo informazioni in un *patch*, un'area, in-

torno al *keypoint* stesso. Esistono varie tecniche di estrazione di queste caratteristiche e di seguito se ne propongono alcune:

- *SIFT*: *Scale-Invariant Feature Transfor*, il sistema esegue una scala di Gauss sull'immagine per individuare i punti di interesse, un punto dell'immagine con elevata variazione di luminosità rispetto all'area circostante (spesso succede nei bordi degli oggetti). Una volta individuati questi punti, il sistema calcola un descrittore SIFT per ognuno, utilizzando la distribuzione di intensità (la luminosità) dei pixel nel *patch* intorno alla *feature*. Questo tipo di descrittore non è altro che un vettore di 128 elementi che andrà a rappresentare in modo univoco il punto prima individuato.
- *FAST*: *Features from Accelerated Segment Test*, è una tecnica di estrazione di *features* ideata per essere veloce ed efficiente, rendendola adatta per l'uso in sistemi real time.

### Estrazione delle caratteristiche FAST

1. Individuazione del punto di interesse: il sistema esamina l'immagine pixel per pixel, seleziona i punti dove l'intensità del pixel supera una soglia predefinita rispetto all'intensità dei pixel circostanti; ciò per cercare di dare una prima stima veloce dei possibili candidati per essere *feature*.
2. Test accelerato: si prendono i vari punti candidati (punto 1) e il sistema confronta l'intensità di ciascuno di questi con l'intensità dei pixel ad essi circostanti, utilizzando una finestra di 16 pixel. Dal confronto escono fuori 3 classi: più chiari, più scuri, simili al punto centrale (il punto candidato come *feature*). Se il numero di

questi pixel, sopra una certa soglia, è maggiore o minore, allora il punto viene confermato come *feature*.

In più il sistema potrebbe calcolare un vettore descrittore delle *features* trovate. In questo caso il descrittore è un vettore di 64 elementi (calcolato sempre nello stesso modo delle SIFT).

- *BRIEF*: *Binary Robust Independent Elementary Features*, tecnica d'estrazione di descrittori progettata per essere robusta ed efficiente. Per prima cosa, si rende l'immagine più liscia usando un *Gaussian Kernel* per prevenire che il descrittore sia troppo sensibile al rumore ad alta frequenza. Per estrarli, il sistema utilizza una finestra di pixel intorno ad una *feature*, seleziona poi all'interno della finestra un certo numero di coppie di pixel dei quali viene calcolata l'intensità (le coppie possono essere generate da una combinazione qualsiasi di pixel della finestra). Dopodiché si genera il vettore descrittore in questo modo: ogni coppia definisce un valore ad una posizione specifica nel vettore, 1 se il primo elemento della coppia ha intensità maggiore del secondo, 0 altrimenti. Ne consegue, quindi, proprietà di efficienza in termini di occupazione dello spazio in memoria e velocità nelle operazioni di confronto, essendo un vettore binario (ad esempio si potrebbe usare anche la distanza di Hamming). La modalità di campionamento delle coppie permette di estrarle in modo deterministico.

Generalmente, le *features* estratte con la tecnica SIFT sono più robuste e affidabili rispetto all'estrazione con le altre due tecniche; diversamente FAST e BRIEF consentono di velocizzare la lenta estrazione che hanno le *feature* SIFT e i relativi descrittori. Tutte queste tipologie di *features* vengono utilizzate allo stesso modo: il sistema le usa per trovare i keypoints, e identificarli

unicamente attraverso l'uso dei descrittori. Si confrontano in frames successivi e, nel caso siano trovate delle corrispondenze, si generano i cosiddetti *match*, ovvero sono gli stessi *keypoints* anche se spostate in diverse posizioni rispetto ai frame precedenti. In questo modo è possibile capire se ci sono stati cambiamenti di scala, rotazioni o distorsioni e quindi risalire ai movimenti fatti dalla camera.

Solitamente i sistemi *Visual SLAM* per sfruttare questo sistema delle *features*, generano i *keyframe*. I *keyframes* sono dei frame (quindi immagini) di riferimento, ovvero ogni frame successivo a questo avrà i suoi stessi *keypoints* o la maggior parte di essi e saranno calcolati i cambiamenti di scala, rotazione o distorsione rispetto a questo *keyframe* fino al ricalcolo del successivo, per poi ripetere il meccanismo. La frequenza di questi *frame chiave* deve essere decisa in modo che i *keypoint* dell'ultimo *keyframe* siano visibili nei frame successivi e nel *keyframe* successivo, altrimenti si rischia di perdere il *tracking*; nonostante ciò la frequenza non deve essere troppo alta altrimenti il costo computazionale aumenta vertiginosamente.

## 1.2 ORB SLAM

### 1.2.1 Features ORB

ORB-SLAM [6] (*Oriented Fast and Rotated BRIEF SLAM*) è un sistema SLAM che utilizza solo una fotocamera monoculare. Questo tipo di algoritmo si basa sulla ricerca di *features* ORB [7]. Quest'ultime sono l'unione delle FAST e le BRIEF:

- si usa la velocità di ricerca delle *features* FAST, però, queste non hanno una componente di orientamento, o di scala. Perciò quello che viene

eseguito è usare un *multiscale image pyramid* (fig. 1.1). La tecnica consiste in una sequenza della stessa immagine, in cui ogni livello è la versione con risoluzione più bassa di quello precedente. L'algoritmo di estrazione delle ORB, dopo aver creato la piramide, usa l'algoritmo FAST per ottenere le *features* da ogni livello dell'immagine. In questo modo, determinando le *feature* ad ogni livello, ORB le sta localizzando su scale diverse. Infine viene assegnato un orientamento ad ogni *key-point* come "rivolto a sinistra" o "rivolto a destra", dipendentemente da come i livelli di intensità cambiano intorno alla *feature*.

- si usano i descrittori delle BRIEF, cioè si prendono tutte le *feature* precedentemente trovate e viene calcolato questo tipo di descrittori. Le BRIEF non sono invarianti alla rotazione, perciò si usano gli rBRIEF (*rotation-aware BRIEF*), quindi, anche quando l'immagine risulta ruotata, si riesce ad ottenere approssimativamente lo stesso descrittore. ORB cerca di aggiungere questa funzionalità senza andare a perdere l'efficienza computazionale delle BRIEF standard.

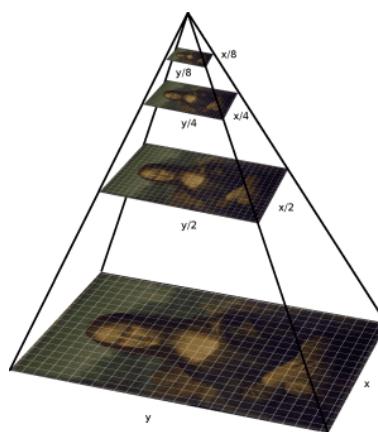


Figura 1.1: esempio di multiscale pyramid

Queste tipologie di *features*, sono state sviluppate come alternativa alle *features* SIFT (*Scale-invariant Feature Transform*) e SURF (*Speeded Up Robust Features*) e sono state progettate per essere più veloci e meno costose da calcolare, ottimo per sistemi real time.

### 1.2.2 Tracking and local mapping

Per riuscire ad eseguire le operazioni classiche di ORB, il sistema è diviso in 3 parti: *Tracking*, *local mapping* e *loop closing*.

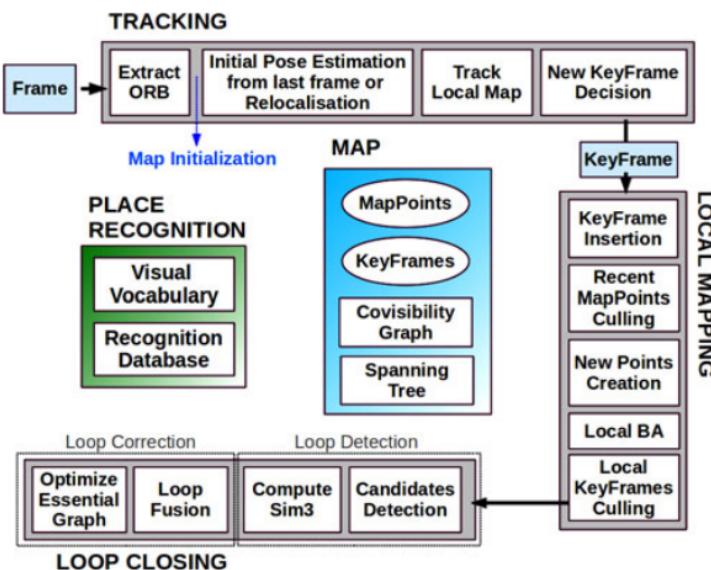


Figura 1.2: *block diagram* del sistema di ORB-SLAM. Si possono vedere tutti gli step che i 3 blocchi eseguono. Si vedono anche i principali componenti della mappa e del *Place Recognition*

Il *tracking* è incaricato di localizzare la camera in ogni frame e decide quando inserire il nuovo *keyframe*. Si introduce il concetto di *Bundle Adjustment*(BA): fornisce stime accurate delle localizzazioni della camera, fornisce anche ricostruzioni geometriche sparse se vengono forniti una buona rete di match e delle buone ipotesi iniziali. Per prima cosa viene eseguito un *feature*

*matching* iniziale con il frame precedente e si ottimizza la posa usando il *motion-only* BA (*motion-only* prevede l'ottimizzazione solo tra il frame attuale e il precedente). Se il sistema si perde, allora viene avviato il modulo di *place recognition* (vedi 1.2.3, Bags of Words) per eseguire una ri-localizzazione globale. *Local Mapping* processa i nuovi *keyframe* e esegue un *local BA* per ottenere un'ottima ricostruzione dell'ambiente intorno alla posa. Vengono cercate nuove corrispondenze per *features* ORB che non hanno un match tra il nuovo *keyframe* e i *keyframe* vecchi collegati ad esso nel grafo di co-visibilità. Tutto questo per triangolare nuovi punti. In una fase successiva, vengono eliminati i punti tramite una "*culling policy*". Questo per mantenere soltanto i punti di qualità più alta. Il *local mapping* si occupa poi di eliminare anche i *keyframe* ridondanti (cioè quei *keyframe* molto simili tra loro).

### 1.2.3 Loop closure

ORB SLAM [6] supporta il ***Loop closing***: è una tecnica utilizzata per migliorare la qualità della mappa e la precisione della localizzazione. Quando la fotocamera ripassa su una zona mappata precedentemente, il sistema di SLAM, confrontando le immagini attuali con quelle già presenti nella mappa, cerca dei punti in comune; se questi vengono trovati, allora l'algoritmo può utilizzarli per correggere la mappa e migliorare la precisione della localizzazione. La *loop closure* (fig. 1.3) è utile per ridurre l'errore accumulato nel tempo sulla localizzazione (*drift*).

#### Bags of words

ORB SLAM utilizza un modulo basato su DBoW2 [2] per cercare di velocizzare il *place recognition* e quindi effettuare *relocalization* (in caso il

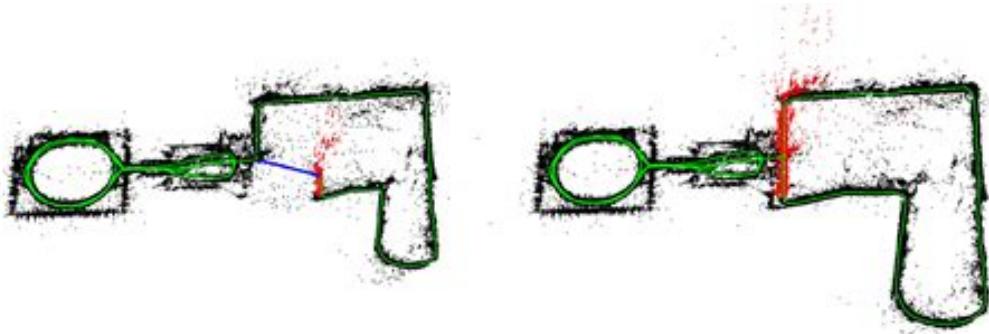


Figura 1.3: Esempio di *loop closure*. A sx la mappatura prima della *loop closure*, dove il vettore blu indica il *drift*, a dx dopo aver eseguito *loop closure* e riduzione del *drift*.

sistema si perda) e *loop detection*. *Visual words*, gli elementi delle *bags*, non sono altro che una discretizzazione dello spazio del descrittore, noto come *visual vocabulary*. Il vocabolario è creato *offline* con i descrittori ORB estratti da un grande insieme di immagini. Se le immagini sono abbastanza generali lo stesso vocabolario può essere usato per diversi ambienti, riuscendo sempre ad ottenere buone performance. Il sistema genera un database incrementale che contiene un indice inverso; questo immagazzina, per ogni *visual word* del vocabolario, l'informazione del *keyframe* in cui è stato visto, perciò, si può accedere al database in modo molto efficiente. Il database è aggiornato anche quando un *keyframe* viene eliminato. Siccome potrebbe esserci sovrapposizione tra i frame, quando si effettua una *query* al database potrebbe non esserci un'unica risposta con il punteggio migliore. Il modulo originale di DBoW2 tiene conto di ciò sommando i punteggi delle immagini che sono vicine nel tempo. Questo ha dei limiti, poiché non si include la possibilità di vedere lo stesso luogo inserito in tempi diversi, del tipo: percorrere più giri di pista o girare intorno ad un quartiere. Quindi si raggruppano i *keyframe* che sono collegati nel grafo di co-visibilità che mantiene delle informazioni di co-visibilità tra *keyframe*. Quando si vuole computare le corrispondenze

tra due set di *features* ORB, il sistema si può vincolare al "brute force matching" ed è sufficiente per quelle *features* che appartengono allo stesso nodo nel *vocabulary tree*. In questo modo si velocizza la ricerca.

Il grafo di co-visibilità è un grafo indiretto in cui i nodi sono i *keyframe*; un arco tra due *keyframe* indica che gli stessi hanno dei punti in comune nella mappa. Questo grafo è utilizzato successivamente nella fase di *loop closure*, quando occorre ottimizzare e ridurre il *drift*. La fase di ottimizzazione prevede di eseguire una *pose-graph optimization* su vincoli di somiglianza per ottenere consistenza globale.

#### 1.2.4 conclusioni

Per concludere, la parte interessante di ORB SLAM [6] è la capacità di utilizzare soltanto una camera monoculare, senza l'ausilio di altri sensori, riuscendo a raggiungere ottimi risultati di mappatura *visual odometry*, ovvero un calcolo della posa solo attraverso la fotocamera, misurando gli spostamenti successivi tra frame. Questo punto di forza presenta anche uno svantaggio: usando una camera monoculare, la profondità non è osservata e quindi la scala della mappa e la stima della traiettoria sono sconosciuti; in più l'avvio del sistema richiede una multi-vista o delle tecniche di filtraggio per ottenere una mappa iniziale, perché che non può essere triangolata dal primo frame. Infine, questo tipo di slam soffre di *drift* di scala e potrebbe fallire se si eseguono rotazioni pure nell'esplorazione.

## 1.3 ORB SLAM 2

Il sistema SLAM attualmente in uso da Firenze Race Team è ORB SLAM2 [4]. Questo sistema è esattamente l'evoluzione successiva di ORB

SLAM [6]. Le principali aggiunte riguardano:

- **Architettura.** ORB SLAM [6], utilizza un’architettura monolitica, in cui tutti i componenti dell’algoritmo (come *tracking* o il *mapping*) sono integrati in un unico sistema. Al contrario ORB SLAM2 [4] utilizza un’architettura a moduli, in cui ogni componente è implementato come un modulo indipendente , ognuno facile da sostituire o aggiornare (fig. 1.4).
- **Efficienza.** ORB SLAM2 [4] utilizza tecniche ottimizzate per il confronto tra immagini e per la gestione della memoria, ciò lo rende più adatto per l’utilizzo su sistemi con risorse *hardware* limitate.
- **Funzionalità.** ORB SLAM2 [4] offre diversi capacità aggiuntive rispetto al precedente, quali il supporto per fotocamere stereo o RGB-D, risolvendo i problemi di utilizzare una sola camera monoculare. Riutilizzo delle mappe già costruite in precedenza per sfruttarle in modalità *localization only*, disattivando così il nodo di mappatura. Usando fotocamere stereo o RGB-D, si può inizializzare il *keyframe* iniziale dal primo frame, settando la sua posa come l’origine e creando quindi una mappa iniziale da tutti i *keypoints* stereo.

### In dettaglio le principali differenze (fig 1.4)

- **Full BA:** dopo aver fatto *loop closure* e la *pose graph optimization*, è stata aggiunta l’operazione di *Full BA* per ottenere la soluzione ottima. L’ottimizzazione può diventare molto costosa, perciò viene fatta su un *thread* diverso. Questo però porta ad un’altra sfida, ovvero unire l’output della BA con il corrente stato della mappa. Vale a dire che, quando la BA finisce, bisogna unire il sottoinsieme di *keyframe* e punti

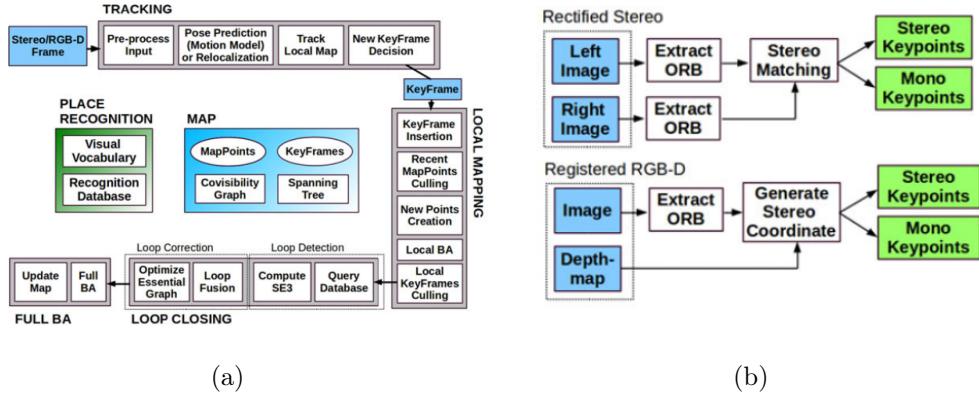


Figura 1.4: Possiamo vedere in (a), le principali differenze con orb slam. (b) mostra come l'input viene processato.

ottimizzati con quelli non inseriti quando la full BA era sotto esecuzione. Questo è risolto con la propagazione della correzione dei *keyframes* aggiornati e quelli non, attraverso lo *spanning tree*, una nuova aggiunta di questo sistema SLAM, che mantiene collegati tutti i *keyframe*. I punti non aggiornati sono trasformati in accordo con la correzione applicata al loro *keyframe* di riferimento. Se un nuovo loop è rilevato mentre l'ottimizzazione è attiva, allora questa viene abortita per poter richiudere il loop, rieseguire la *pose graph optimization* e poterla così riavviare.

- **Input:** il sistema adesso supporta anche l'uso di *keypoints* stereo, definiti da 3 coordinate,  $x_s = (u_L, v_L, u_R)$ , dove  $(u_L, v_L)$  sono le coordinate dell'immagine sinistra e  $u_R$  è la coordinata orizzontale della destra. Dalle camere stereo vengono estratte le *features* e si calcola la posizione dei keypoints su entrambe le immagini, per quelli trovati nell'immagine di sinistra, si cerca un match su quella di destra (*Stereo matching*). Questo può essere velocizzato se le immagini sono del tipo *stereo rectified* e quindi le linee epipolari sono orizzontali. Dopodiché si genera il *key-*

*point* stereo unendoli dai match: si prende le coordinate del keypoint dell’immagine della camera sinistra e la coordinata orizzontale dello stesso sull’immagine della fotocamera destra. Per le camere RGB-D, si calcola una specie di coordinata orizzontale come emulazione:

$$u_R = u_L - \frac{f_x b}{d}$$

dove  $f_x$  è la lunghezza della focale orizzontale,  $b$  è la *baseline* tra il proiettore della luce e la camera ad infrarossi,  $d$  è la distanza calcolata dalla camera RGB-D. Viene poi aggiunta una classificazione dei punti trovati: *close* punti vicini se la sua profondità è inferiore di 40 volte la *baseline* RGB-D o stereo, altrimenti è considerato come *far* (lontano). I punti vicini possono essere triangolati con sicurezza poiché la profondità è correttamente stimata e fornisce informazioni di scala, rotazione e traslazione. I punti lontani invece forniscono un’accurata informazione di rotazione, ma informazioni di scala e traslazione più povere. Si triangolano i punti lontani quando sono supportati da più viste.

## 1.4 ORB SLAM3

ORB SLAM3 [1] porta ulteriori funzionalità rispetto ad ORB SLAM2 [4]. Di seguito verranno analizzate in modo generale.

### 1.4.1 Differenze tra ORB SLAM 2 e 3

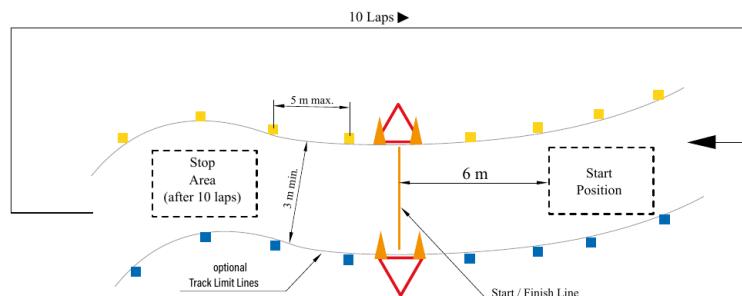
ORB SLAM3 [1] differisce per diversi aspetti da ORB SLAM2:

- **IMU.** Supporto ad integrazione dell’IMU con la fotocamera. Può essere usato per fornire informazioni di movimento supplementari al sistema



(a)

- Yellow/Blue Cone
- ▲ Small/Big Orange Cone
- △ Red TK Marking & TK Equipment  
(Shape undefined)



(b)

Figura 1.5: test con pilota (a), mappa di come è formato un pezzo di pista della formula student con regole (b)

SLAM; può aiutare a ridurre gli errori di localizzazione e a migliorare la stabilità del sistema, soprattutto in situazioni dove si hanno immagini delle videocamere scarsamente distinctive o in condizioni di scarsa illuminazione.

- **Multi mappa.** Se viene persa la posizione, il sistema è in grado di creare istantaneamente una nuova mappa e accedendo ai dati precedenti è in grado di correggersi e unirsi a mappe passate. Questo concetto è molto utile soprattutto se si ripassa più volte in uno stesso ambiente, andando ad ottimizzare sempre di più aree difficili da mappare causa agenti esterni. In un uso come la *formula student* è decisamente un dettaglio rilevante.
- **Place recognition.** Utilizzando BDoW2, il problema che sorge è una richiesta di consistenza temporale. Occorre fare un match con 3 keyframe successivi della stessa area per riconoscerla a pieno, questo prima di verificare una consistenza geometrica, cercando di aumentare la precisione. Di conseguenza il sistema risulta molto lento a chiudere il loop. In ORB SLAM3 [1], viene proposto un altro metodo: I fotogrammi chiave candidati vengono controllati per la coerenza geometrica, dopodiché si controllano poi per la coerenza locale con 3 frame vicini (che nella maggior parte dei casi sono già nella mappa). Questa strategia dovrebbe aumentare la recall ed intensificare l'associazione dei dati, migliorando l'accuratezza della mappa a scapito di un costo computazionale leggermente superiore.
- **Astrazione camera.** Il codice adesso è agnostico al modello della camera usata e permette l'aggiunta di nuovi modelli semplicemente fornendo le funzioni di projection, unprojection e Jacobian.

Quindi le parti più toccate dallo sviluppo di ORB SLAM 3 [1], sono quelle relative a *tracking* e a *loop closure* che adesso si chiama *loop and map merging*.

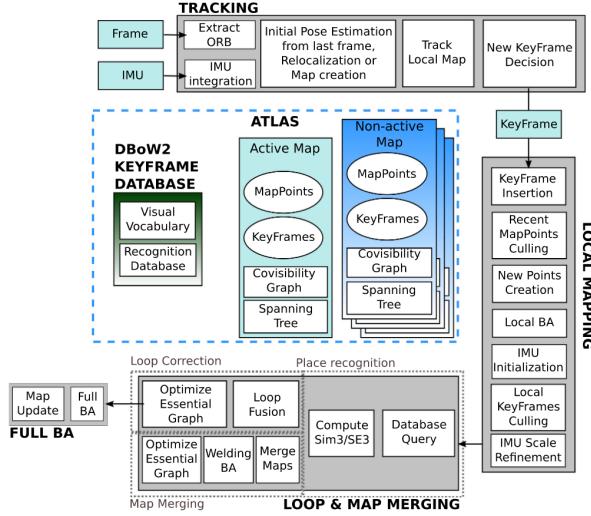


Figura 1.6: Componenti principali di ORB SLAM3 [1]

### 1.4.2 *Visual-intertial* SLAM

ORB SLAM3 [1], non solo si basa su ORB SLAM2 [4], ma anche su ORB SLAM-VI [5]. Questo particolare sistema, è stato uno dei primi in grado di implementare il riuso di mappe e l'uso dell'IMU, riuscendo in modo economico, a raggiungere risultati impressionanti con solo l'uso di una camera monoculare e di un sensore IMU. ORB SLAM3 [1] riesce a prendere i punti migliori di ORB SLAM2 [4] e di ORB SLAM-VI [5] riuscendo a creare un sistema multi-mappa e multi-sessione capace di lavorare sia in modalità *visual* che *visual inertial* con sensori monoculari, stereo o RGB-D, usando modelli pinhole e fisheye.

### 1.4.3 Maggiori dettagli sulle nuove funzionalità

Adesso si analizzeranno in dettaglio quelle che sono le principali novità di questa:

- **Atlas.** Rappresentazione multi-mappa composta da un insieme di map-

pe disconnesse. Si ha una mappa detta attiva, usata in real time come per ORB SLAM2 [4], mentre tutte le altre mappe sono referenziate come non attive. ORB SLAM3 [1], anche qui, usa un sistema unico basato su un *database* DBoW2 di *keyframes* usato per la re-localizzazione, *loop closing* e *map merging*.

- **Tracking thread.** Usando la modalità *visual-inertial*, la velocità e i *bias* dell'IMU sono stimati includendo i residui inerziali nell'ottimizzazione. Se il sistema si perde (*track lost*), il *tracking* thread cerca di ri-localizzare il frame corrente in tutte le mappe dell'*Atlas*. Nel caso venga trovata la mappa di appartenenza, la mappa attiva viene cambiata in caso di necessità oppure viene disattivata con l'inizializzazione di una interamente nuova.
- **Local mapping thread.** Vengono inizializzati e raffinati i parametri dell'IMU usando una tecnica chiamata MAP-estimation (stima Maximum a Posteriori).
- **Loop and map merging thread.** Rileva regioni in comune tra la mappa attiva e il resto dell'*Atlas* alla frequenza del rate di creazione di nuovi *keyframes*. Se l'area comune appartiene alla mappa attiva, allora si esegue l'operazione di *loop correction*; se invece appartiene ad un'altra mappa, allora entrambe le mappe vengono unite (*merging*) in un'unica che diventa poi la mappa attiva. Dopo entrambe queste due operazioni, viene lanciato un full BA in un thread indipendente per raffinare ancora di più la mappa senza intaccare le performance *real-time*.

## 1.5 ROS

ROS sta per *Robot Operating System* anche se in realtà non è un vero e proprio sistema operativo, ma più un *framework* di basso livello che viene eseguito su Linux. Una specie di *middleware*, ovvero un *layer* che è responsabile di gestire le comunicazioni tra programmi in un sistema distribuito. Questo ci rende possibile la programmazione in sotto programmi, ognuno che si occupa di funzioni specifiche, le quali, messe insieme, riescono a risolvere problemi complessi, come appunto la guida autonoma. ROS è stato quindi progettato per aiutare a creare rapidamente applicazioni per robot che integrano sensori, attuatori, schede di controllo e altre componenti via hardware. Si hanno 3 livelli di concetti: Livello *Filesystem*, livello *Computational Graph*, livello *Community*.

### 1.5.1 ROS Filesystem Level

Concetti che coprono le risorse di ROS su disco:

- **Packages.** I pacchetti sono l'unità principale per l'organizzazione del software in ROS. Un package può contenere processi ROS, i cosiddetti nodi (vedi 1.5.2), una libreria, datasets, file di configurazione o qualsiasi altra cosa che è utile organizzare insieme.
- **Metapackages.** Pacchetti specializzati per rappresentare un gruppo di altri pacchetti relativi
- **Package Manifests.** Manifesti che forniscono metadati su un pacchetto, includendo nome, versione, descrizione, informazioni sulla licenza etc.

- **Message types.** Descrizione di messaggio. Definisce la struttura di un messaggio.
- **Service types.** Descrizione di un servizio. Definisce la struttura dati della richiesta e della risposta.

### 1.5.2 ROS Computational Graph Level

Rete *peer-to-peer* di processi ROS che stanno processando dati insieme.

#### Concetti basilari di *Computation Graph*

- *Nodes*: processi che eseguono il calcolo. Un nodo ROS è scritto usando una libreria *client* di ROS come roscc o rospy ( per c++ e Python rispettivamente).
- *Master*: fornisce la registrazione del nome e la ricerca per il resto del *Computational Graph*. Senza di questo, i nodi non sarebbero in grado di scambiarsi messaggi o richiamare servizi.
- *Messages*: nodi che comunicano tra loro passandosi i messaggi (definiti in 1.5.1).
- *Parameter Server*: permette ai dati di essere immagazzinati in una locazione centrale. Fa parte del master.
- *Topics*: identificatori di *Bus* dove i nodi possono scambiarsi i messaggi. In generale, i nodi non sanno con chi stanno comunicando, invece, un nodo che genera dei dati li **pubblica** (*Publisher*) su un *topic* rilevante. Un nodo interessato in dei dati, si **sottoscrive** (*Subscriber*) a quel *topic* e riceverà tutti i messaggi. Ad un *topic* possono collegarsi multipli *Subscriber* e *Publisher*. Ogni topic è fortemente tipizzato dal tipo di

messaggio ROS usato per la pubblicazione. I nodi possono ricevere messaggi da quel topic solo con il tipo corrispondente. Il nodo *Master* o i *Publisher* non impongono la consistenza di tipo, ma i *Subscribers* non stabiliranno il trasporto dei messaggi se non c'è corrispondenza di tipo. Ogni *Subscriber* si avvale di una funzione di *callback* per poter ricevere ed elaborare i messaggi che arrivano sul topic in ascolto in *real-time* (fig 1.7).

- *Services*: permettono di creare una comunicazione sincronizzata in stile *client/server* tra i nodi. Molto utile per cambiare un' impostazione o richiedere una specifica azione da un nodo.
- *Bags*: sono formati per il salvataggio e la riproduzione di messaggi ROS. Questi sono un meccanismo importante per immagazzinare i dati, ad esempio quelli provenienti dai sensori che possono essere difficili da collezionare ma sono necessari per sviluppare e testare gli algoritmi.

Il *Master* agisce come un *nameservice* (gestisce la mappatura e i nomi del sistema ai loro indirizzi di rete) nel ROS *Computational Graph*. Immagazzina le registrazioni delle informazioni su *topics*, *services* per i nodi ROS. Siccome i nodi comunicano con il *Master* per riportare la loro registrazione delle informazioni, potrebbero ricevere le informazioni su altri nodi registrati e quindi stabilire connessioni appropriate.

I nodi, tra loro, stabiliscono direttamente la propria connessione; il *Master* fornisce solo informazioni di *lookup*, come un server DNS, perciò i nodi che si sottoscrivono ad un *topic* richiederanno la connessione ad un nodo che pubblica su quel *topic*. Il protocollo viene stabilito concordatamente, il più comune in ROS è chiamato TCPROS che utilizza *socket* TCP/IP standard. Questa architettura permette di avere operazioni disaccoppiate dove i nomi

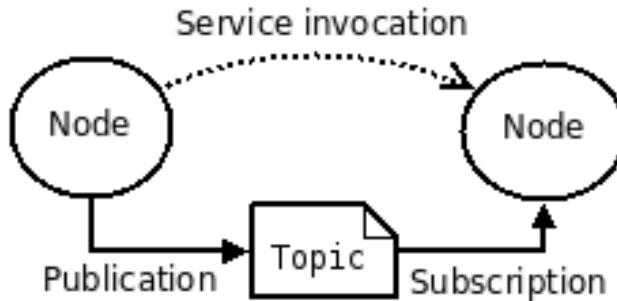


Figura 1.7: Schema del funzionamento di uno scambio di messaggi tra due nodi usando un *topic*

sono i mezzi principali per costruire un sistema più grande e complesso. I nomi hanno un ruolo molto importante in ROS: nodi, *topics*, servizi e parametri hanno tutti un nome.

### ROS Community Level

La parte che mantiene risorse per abilitare la comunità di scambiare software e conoscenze. Ogni messaggio pubblicato ha associato un *header* che contiene un *timestamp*, un numero di sequenza e anche un campo chiamato *frame\_id*.

### Frames

I frame in ROS non sono altro che sistemi di riferimento relativi ad un particolare sensore o punto del veicolo. Esistono dei frame predefiniti:

- **map.** Il frame ha la sua origine in un punto fisso arbitrario scelto nel mondo.
- **odom.** Ha la sua origine dove il robot è inizializzato. Il punto è comunque fissato nel mondo.

- **base\_footprint.** Posizione in 2D al centro del robot(senza l'altezza) che rappresenta il movimento del robot, perciò questo sistema di riferimento non è fisso ma è solidale al robot.
- **base\_link.** Posizione in 3D solidale al robot.

Infine ci possono poi essere dei *frame\_id* personalizzati in base alla tipologia dei sensori che si utilizzano. La trasformazione da un frame ad un altro è statica nel caso in cui i sistemi di coordinate sono fissi tra loro ( quindi c'è solo una traslazione che non varia nel tempo); invece avviene una trasformazione più complessa tra frame non vincolati, le cui posizioni possono cambiare nel tempo in modo indipendente, ad esempio tra *base\_link* che si muove rispetto a *map*. ROS fornisce un *package* chiamato **tf(transform listener)** che aiuta ad effettuare le trasformazioni cercando di semplificare questi concetti.



# Capitolo 2

## Creazione wrapper di ORB SLAM3

### 2.1 Sistema Autonomo nella Formula Student

Il sistema autonomo della Formula Student è strutturato tramite una serie di algoritmi eseguiti in modo concorrente, ognuno con un compito specifico.

- **Cone Detection:** rete neurale convoluzionale che, analizzando le immagini della camera stereo, riesce a trovare la posizione dei coni al suo interno.
- **SLAM:** insieme alla Cone Detection, rappresenta la parte di Visione del veicolo. Come descritto nel cap.1, implementando ORB SLAM2 [4], è in grado contemporaneamente di mappare l'ambiente e localizzare il veicolo all'interno di esso, usando la sola camera Stereo. La mappatura subisce poi uno step di fusione con le informazioni della Cone Detection, per estrapolare al meglio la posizione dei coni e passarla poi all'algoritmo successivo insieme alla posizione.

- **Trajectory Estimation:** algoritmo di ricerca che, ricevuta la mappa dell’ambiente dei soli coni e la posizione del veicolo, è in grado di ricavare una traiettoria che segue il circuito. Inizialmente, quando ancora il circuito è sconosciuto ed è quindi visibile solo la parte frontale al veicolo, l’algoritmo cerca di seguire una traiettoria più centrale possibile ai coni; una volta concluso un intero giro del tracciato, ottimizza la traiettoria.
- **Controls:** algoritmo che, data la traiettoria in ingresso, si occupa di controllare i vari attuatori del veicolo e cerca, tramite *feedback*, di fare in modo che il veicolo segua la traiettoria fornita dall’algoritmo precedente, minimizzando il più possibile l’errore.

Nella competizione esistono diverse tipologie di prove denominate missioni. Queste fanno in modo che il veicolo sia pronto per un determinato tipo di tracciato: si ha lo **Skidpad**, dove il veicolo deve seguire il circuito che forma un 8, seguendo un determinato pattern. **Acceleration** dove invece il veicolo deve andare dritto e accelerare più che può, per poi fermarsi. **Track drive** e **Autocross** dove il veicolo deve muoversi in un circuito vero, in cui: nel primo caso, può fare un giro di prova lento per avere una mappatura migliore del circuito e successivamente il veicolo deve arrivare fino a 10 giri per poi fermarsi; nel secondo caso, la vettura deve fare un unico giro, quindi senza giro di prova. Tutti i tipi di tracciati sono sempre delineati da coni stradali. Siccome il veicolo ha necessità di eseguire tutti questi algoritmi in modo concorrente e collaborativo per raggiungere l’obbiettivo della missione selezionata, allora per cercare di integrarli è stato scelto di usare ROS. Questo porta a molti vantaggi:

- **uso comprovato.** ROS è ampiamente utilizzato nell'industria della robotica, perciò lo studio di questo framework non è fine a se stesso.
- **multi-platform.** ROS 1 è supportato e testato su linux, mentre ROS 2 adesso funziona anche in piattaforme quali Windows, MacOS ma anche dispositivi embedded.
- **open source.** In un ambiente come la Formula Student dove i fondi sono limitati, l'aspetto *open source* è fondamentale e inoltre garantisce pieno supporto da tutta la comunità.
- **modularità.** ROS è progettato con una architettura modulare, che permette di integrare e riusare il codice tra progetti differenti in modo molto facile. Inoltre è possibile far comunicare nodi ROS scritti in linguaggi differenti come python o c++.

Si hanno ovviamente anche degli svantaggi:

- **alto costo computazionale.** Usare ROS richiede un alto costo computazionale, ciò lo rende sconsigliabile da usare in sistemi Embedded. Fortunatamente, l'intero algoritmo autonomo descritto precedentemente, viene eseguito da un computer Rugged, con un hardware molto potente in grado di gestire ROS senza alcun problema. Rimane comunque un alto costo computazionale ed è infatti consigliabile, ove possibile, l'ottimizzazione di tutti i possibili algoritmi.
- **performance Real Time limitate.** Pur essendo una buona scelta per la Formula Student, dove comunque non si raggiungono velocità troppo elevate, per altri ambiti dell'Automotive risulta molto limitato per applicazioni che richiedono molta più precisione nei tempi o sincronizzazioni.

- **Safety:** ROS non è stato progettato specificatamente per le applicazioni critiche per la sicurezza, ciò può rappresentare una limitazione per i progetti che richiedono standard di sicurezza rigorosi. Per il sistema autonomo del Firenze Race Team, la safety viene raggiunta andando ad implementare dei nodi che aggiungono questa funzionalità.

Quest'analisi di pro e contro è stata necessaria per capire il motivo della scelta di ROS come piattaforma di integrazione dei vari algoritmi del sistema autonomo progettato insieme ai ragazzi del Firenze Race Team: Per il caso d'uso della Formula Student i soli vantaggi di modularità e multi-piattaforma, sovrastano in maniera netta tutti gli svantaggi.

### 2.1.1 Passaggio da ORB SLAM2 a ORB SLAM3

Si è scelto di provare ad eseguire questa evoluzione generazionale da ORB SLAM2 [4] al 3 [1], come riportato in 1.4.1.

- **Supporto Multimappa:** rende il sistema più robusto di ORB SLAM2 [4].
- **IMU:** Aggiunge la possibilità futura, in caso dell'acquisto da parte del Firenze Race Team di un sensore IMU preciso, di usarlo per migliorare la stabilità del sistema.
- **Astrazione camera:** A differenza di ORB SLAM2 [4], adesso risulta molto più facile provare l'algoritmo con camere diverse da quelle presenti nel database di ORB SLAM2 [4].

Sulla carta, inoltre, viene anche riportata una migliore accuratezza rispetto ad ORB SLAM2 [4], perciò con tutte questi aspetti migliorativi è stato deciso di creare un wrapper per far funzionare ORB SLAM3 [1] con ROS e quindi

provare a vedere se si riesce a migliorare la parte di Visione del sistema autonomo.

## 2.2 *orb\_slam3\_ros\_wrapper*

ORB SLAM3 [1] fornisce una specie di interfaccia con ROS, dove il codice permette di prelevare le immagini da *topic* generici, senza però possibilità di pubblicare i risultati di mappatura e localizzazione e restando vincolati al sistema di visualizzazione incluso nello stesso ORB SLAM3 [1]. Il problema è dovuto alla metodologia usata per compilare quest'interfaccia che è antiquata e non più supportata. Perciò, per rispettare la necessità che lo SLAM si adatti a ROS e al sistema autonomo del Firenze Race Team, è stato scelto di creare un *wrapper* nuovo, con i seguenti requisiti:

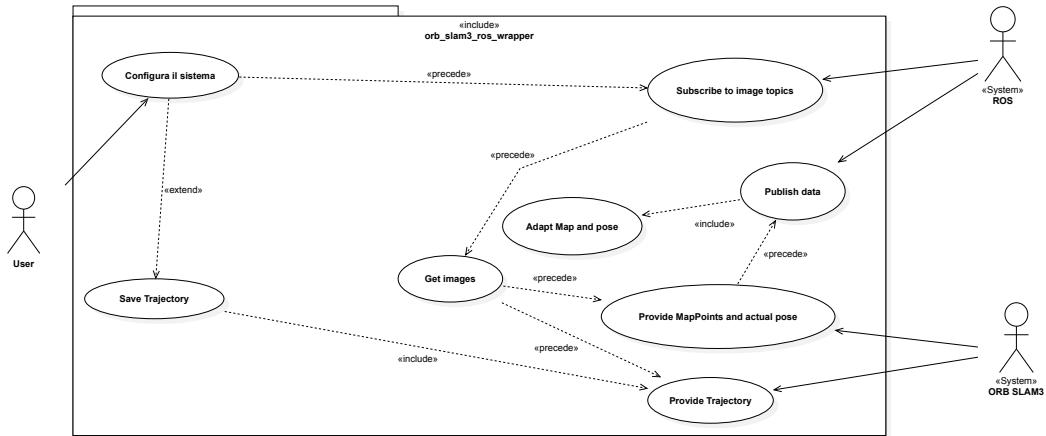


Figura 2.1: Use case diagram che racchiude tutti i requisiti funzionali del wrapper.

- **Portabilità:** il wrapper deve poter essere usato su qualsiasi calcolatore purché compatibile con ROS 1 (noetic) e con le librerie da cui dipende ORB SLAM3 [1]: **OpenCV** 4.4.0 o 3.2.0, **Eigen3**, almeno alla versione 3.1.0, **Pangolin** 0.8.

- **Flessibilità:** software in grado di adattarsi tranquillamente a cambiamenti nella camera o altri sensori usati. Anche in caso di cambiamento di un'altra componente del sistema autonomo, usando ROS questo requisito viene rispettato con facilità.
- **Usabilità:** rendere il software eseguibile in modo semplice tramite un file ".launch" o ".yaml" in cui racchiudere tutti i vari parametri di ROS e ORB SLAM3 [1].
- **Efficacia ed Efficienza:** il wrapper deve risolvere il problema specificato, cioè, in modo semplice e completo, riuscire a far comunicare ORB SLAM3 [1] con ROS, ricevendo in ingresso le immagini provenienti dalla fotocamera e pubblicando mappa e posizioni in tempo reale, con la perdita minore possibile di prestazioni rispetto ad ORB SLAM3 [1] nativo.

Prima di procedere con la progettazione, è stata fatta una ricerca con lo scopo di trovare una soluzione che fornisse un punto di partenza in modo da non dover partire da zero. Dai risultati della ricerca, sono emerse diverse soluzioni, ma la più in risalto risultava essere `orb_slam3_ros_wrapper`, un *wrapper* ROS già pronto che fa da *layer* di comunicazione tra ROS e il sistema SLAM usando direttamente l'API di ORB SLAM3 [1]. Di seguito si elencano i vari pregi e difetti:

- Pregi: Il codice del *wrapper* è separato da quello di ORB SLAM3 [1], il che rende semplice inter-cambiare il sistema slam nelle sue diverse versioni, a patto di non cambiare le sue API.
- Difetti: il codice dipende dalle API esposte da ORB SLAM3, lo sviluppo si articola in più steps, ovvero se si volesse cambiare parte del

codice in ORB SLAM3 [1], occorre ricompilarlo prima di usare le nuove *features* nel *wrapper*. Pertanto necessita fare una compilazione in più.

Questo codice risulta già essere presente di alcuni requisiti sopra citati. Non è stato sviluppato dagli autori di ORB SLAM3 [1], ma rimane comunque una buona scelta. Per migliorare fin da subito un po' i difetti, è stato inserito all'interno del *wrapper* anche la versione 1.0 di ORB SLAM3 [1], in modo che il *porting* risulti più veloce, avendo solo da compilare il codice senza dover ogni volta scaricarlo in caso di installazione su dispositivi diversi. Questo porta anche il vantaggio di mantenere il proprio versionamento in modo da avere piena libertà nella modifica di tutto il codice di ORB SLAM3 [1]; ciò risulta essere importante dato che, come vedremo, è stata necessaria una modifica dell'API, tuttavia quest'ultima compromette il sistema intercambiabile su cui era nato questo wrapper. Infine lo *step* in più che occorre fare per la compilazione di tutto il sistema, risulta essere una *feature*, poiché in caso di aggiunte o modifiche al *wrapper*, avendo il codice separato, la compilazione risulta molto più veloce e ciò avvantaggia molto hardware vecchi i quali non dispongono di troppi processori o thread, ma riescono benissimo a far girare il sistema SLAM.

### 2.2.1 Architettura

*orb\_slam3\_ros\_wrapper* ha una architettura(fig. 2.2) molto semplice, ma funzionale e pratica. Di seguito si analizzano in dettaglio i vari file con le proprie funzioni:

- *common.h*: file base che mantiene la dichiarazione delle funzioni comuni che vanno poi a richiamare il codice delle API di ORB SLAM3 [1].

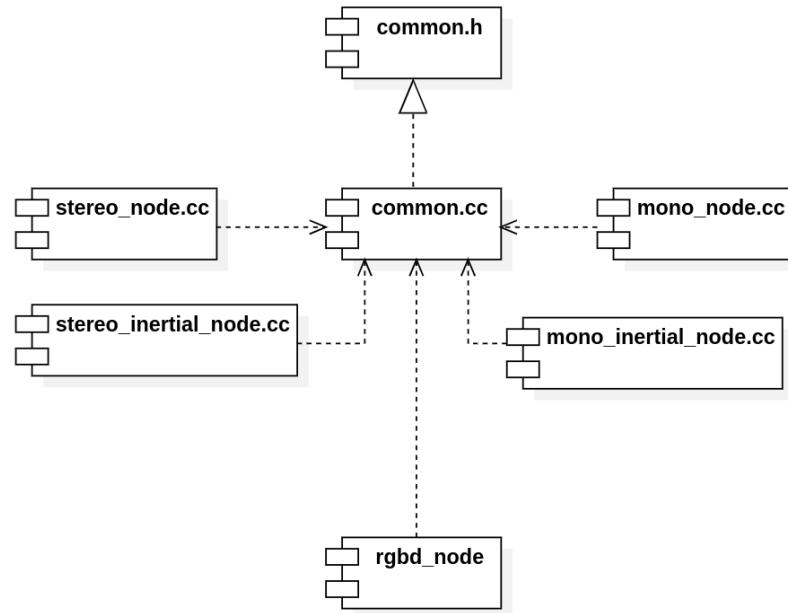


Figura 2.2: struttura più in dettaglio del codice di `orb_slam_3_ros_wrapper`.

Mantiene inoltre la dichiarazione di variabili globali usate da tutto il resto del wrapper.

- `common.cc`: file base che mantiene la definizione delle funzioni dichiarate in `common.h`.
- Il *wrapper* mantiene altri file .cc che servono per eseguire ORB SLAM3 [1] per le diverse tipologie di camere:
  - Stereo o `Stereo_inertial`
  - RGBD
  - Mono o `Mono_inertial`

In realtà le dichiarazioni che si trovano in `common.h` sono separate dal codice presente in `common.cc`, e questo è dovuto alla pessima progettazione del

*wrapper.*

Si vedrà ora di analizzare in dettaglio tutte le funzioni di common.cc per poi analizzare il codice degli altri file, così da far comprendere bene come funziona questo codice:

- **setup\_ros\_publishers:** funzione che prende in ingresso un oggetto di tipo `node_handle`, che fornisce un’interfaccia di gestione di un nodo ROS. La funzione prevede di attuare il setup dei *publisher* per pubblicare la posa della camera e la mappa dei punti locali.
- **publish\_ros\_camera\_pose:** funzione che provvede a pubblicare la posa della camera, prende in ingresso una matrice SE(3) contenente la posa.
- **publish\_ros\_tf\_transform:** funzione che prende in ingresso una matrice SE(3) e due frame: *child\_frame\_id*, *frame\_id*. Si richiama la funzione *SE3f\_to\_tfTransform()* per trasformare il dato dal tipo SE(3) ad un oggetto del package **tf**; dopodiché si trasforma la posa della camera dal frame *child\_frame\_id* al frame *frame\_id* e si pubblica usando le funzioni di **tf** rispettivamente: **StampedTransform** e **sendTransform**.
- **SE3f\_to\_tfTransform:** esegue la trasformazione da *Sophus::SEf* a *tf::Transform*, estraendo il vettore posizione e la matrice di rotazione per poi inserirla nel costruttore del nuovo oggetto del package **tf**.
- **publish\_ros\_tracked\_mappoints:** funzione che pubblica i punti real-time della mappa, ovvero i punti ORB che vengono estratti in tempo reale dalle immagini della fotocamera. Il procedimento è il seguente: prima i punti vengono convertiti in una *point cloud* di tipo *Sen-*

*sor\_msgs::PointCould2* per poi essere inviati tramite ROS. La conversione viene fatta mediante la funzione *tracked\_mappoints\_to\_pointcould*.

- **tracked\_mappoints\_to\_pointcloud:** la funzione prende in ingresso un vettore di oggetti di tipo *MapPoint* di ORB SLAM3, e crea un messaggio di tipo *Sensor\_msgs::PointCloud2*; quest'ultimo è formato dai seguenti campi:

- *header*, che contiene *frame\_id* e *timestamp*
- *height*, *width*, dimensioni della nuvola di punti in 2D
- *PointField[] fields*, vettore che descrive ogni punto della nuvola. Può essere ridimensionato a seconda di quanti canali si inserisce al suo interno (in questo caso solo 3 per le 3 dimensioni spaziali)
- *is\_bigendian*, campo booleano per indicare se i dati vengono rappresentati nel formato *big endian*
- *point\_step*, *row\_step*, lunghezza di un punto e di una riga in byte
- *data*, campo dove si inseriscono i dati dei punti

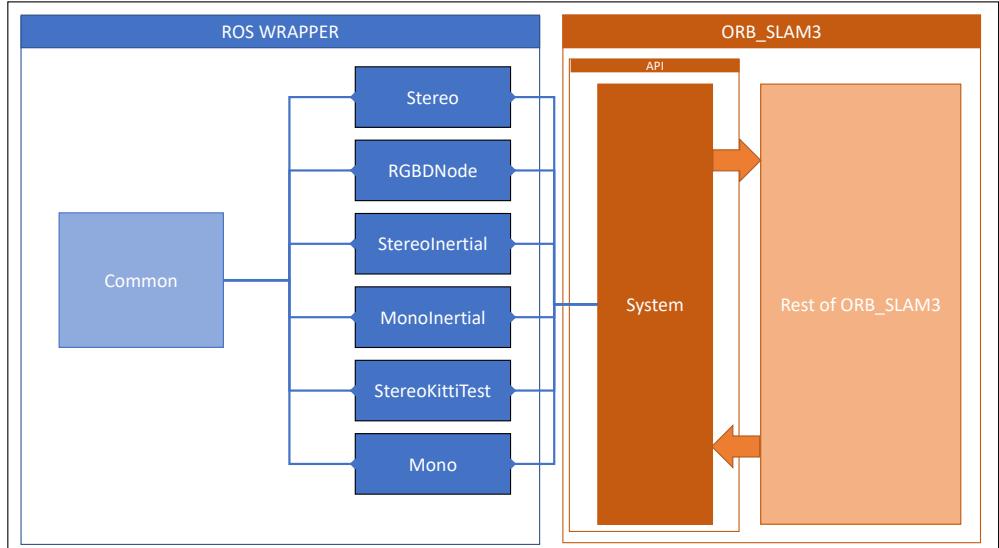
Il messaggio viene inserito in una sola riga poiché la nuvola non è ordinata. Dopo aver preparato l'*header* e tutti i campi prima di *data*, vengono inseriti finalmente i punti nella nuvola e il messaggio è pronto per essere inviato.

Negli altri file quali *rgbd\_node*, *stereo\_node* etc., il codice presenta una struttura molto simile tra loro: si trova il *main* che prende in ingresso tutti i vari parametri di funzionamento del nodo ros, quali i vari *topic* di immagini, i parametri della fotocamera. Si istanzia poi un oggetto di tipo *System* di ORB SLAM3 ed infine si registrano le *callback*. In ognuno dei file è presente

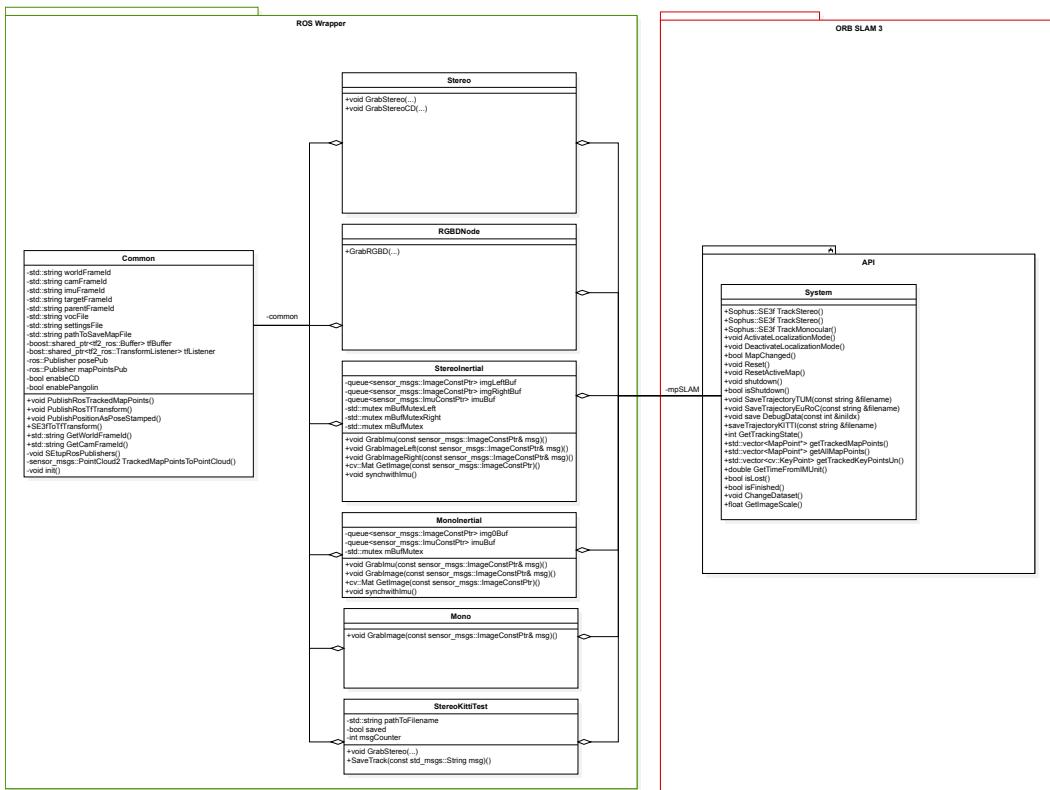
poi una classe chiamata *ImageGrabber*, che offre i metodi di *callback* per i vari *subscriber* per ottenere da ROS le immagini o i dati dell'IMU.

### 2.2.2 Refactoring

Per rendere il codice più chiaro ne è stato eseguito un *refactoring*(fig. 2.3) rendendolo più elegante, leggibile e scorrevole. L'obbiettivo era di cercare di mantenere la programmazione ad oggetti tenendo comunque saldo il concetto di semplicità, perciò sia Common che i vari file eseguibili, sono divenuti classi a tutti gli effetti. Questo design porta numerosi vantaggi: Common era diviso in modo pessimo senza alcun design in *common.h* che manteneva le dichiarazioni delle funzioni inutilmente e la dichiarazione di attributi globali usati nei vari file eseguibili (*stereo.cc*, *mono.cc*, etc...), mentre *common.cc* manteneva la definizione delle funzioni; perciò alla fine questo file era già a tutti gli effetti una classe. Ovviamente tenere globali delle variabili avrebbe portato a non pochi problemi, perciò, tramite l'incapsulamento di tutti gli attributi o addirittura tutta l'eliminazione di quelli non più usati, consente un aumento della sicurezza, una migliore organizzazione del codice riducendo in questo modo la complessità. In 2.2.2(Common) si espongono in dettaglio anche tutte le modifiche al codice. Sono state poi abolite tutte le "copie" di classi **ImageGrabber** in ognuno dei singoli file eseguibili evitando in questo modo di avere codice duplicato. Adesso, tutti questi file sono vere e proprie classi dotate di un proprio *main* per rendere i file ".cc" eseguibili. È stato infine aggiunto il codice che rende compatibile il *wrapper* con il sistema autonomo del Firenze Race Team. Come per la classe Common, anche qui si riesce ad ottenere vantaggi in termini di eleganza e chiarezza nel codice. Il codice infine è stato riscritto rispettando la pratica *Upper Camel Case*, seguendo la linea di ORB SLAM3, in modo che tutto fosse coerente.



(a)



(b)

Figura 2.3: (a) Block Diagram che rappresenta la struttura del wrapper. Possiamo vedere a destra che l'API di ORB SLAM3 è data dalla classe System. Ogni eseguibile/classe, ha un oggetto di tipo System, il quale permette di eseguire tutte le varie operazioni e di ricavare mappa e posizione. (b) class diagram completo del wrapper dopo l'operazione di Refactoring.

## Common

Di seguito vengono analizzate tutte le modifiche atte a far funzionare in modo ottimo tutto il codice del vecchio file common.cc

- **Common:** il costruttore della medesima classe, prende in ingresso un oggetto di tipo *node\_handler*, usato per istanziare i vari *Publisher* chiamando il metodo privato **setupRosPublisher**. Si occupa anche di richiamare il metodo privato **Init**.
- **Init:** In questo metodo è stato preso tutto il codice che veniva richiamato dal main di ogni singolo eseguibile che si riferisce all'istanziazione di tutti i parametri in input da ROS, rendendo in questo modo il codice comune, riuscendo a pulire tutti questi file di codice duplicato. Sono stati quindi aggiunti a **Common**, insieme ai loro metodi "Get", diversi attributi.
  - *camFrameId*: il frame della camera, al quale lo slam si riferisce, ad esempio se la fotocamera è stereo, allora il *cam\_frame\_id* corrispondente sarà quello relativo alla camera sinistra.
  - *worldFrameId*: relativo alla mappa, quindi l'identificatore del sistema di riferimento fisso.
  - *targetFrameId*: come accennato in 1.5.2, si può ricavare la posa della camera su un frame diverso che abbia trasformazione statica con *cam\_frame\_id*. Questo può essere utile in caso di utilizzo della camera in un veicolo o velivolo e si vuole direttamente la posizione di come si muove da un particolare punto di interesse(può essere il centro di massa, la posizione del guidatore, etc...).

- *parentFrameId*: siccome una camera può avere un albero di trasformazioni statiche, questo attributo serve ad indicare quale tra queste è la radice, in modo, poi, da riuscire a ricreare la mappa intorno alla camera impostando accuratamente tutti i vari frame.
- *imuFrameId*: il frame dell'IMU in caso sia disponibile. Attributo messo come default nullo in caso di non utilizzo dell'IMU in ORB SLAM3 [1].

Tutti questi frame id, sono usati soprattutto dentro la classe **Common**, perciò la loro presenza come attributo privato è essenziale. Procedendo, si hanno ancora altri attributi:

- *vocFile* stringa che indica la path al vocabulary file che servirà ad ORB SLAM3 per le Bag of Words(vedi 1.2.3).
- *settingsFile* stringa che indica dove è il file di configurazione contenente tutti i parametri della fotocamera in uso.
- *pathToSaveMapFile* con l'aggiunta della classe **stereoKittiTest**, è possibile salvare su file la mappatura fatta da ORB SLAM3 [1], perciò questa stringa indica dove andare a salvare il file.
- *enableCD* variabile booleana che indica se in ingresso si ha pure l'informazione della Cone Detection o meno.
- *enablePangolin* variabile booleana che indica se si vuole utilizzare il visualizzatore di mappa e traiettoria incluso in ORB SLAM3 che utilizza la libreria Pangolin.

Infine si sono resi attributi veri e propri le variabili *tfListener* e *tfBuffer* per gestire le trasformazioni fatte con la libreria **tf** e i due **Publisher**

**posePub** e **mapPointsPub** che pubblicano rispettivamente posa e mappa.

- **setupRosPublishers**: metodo identico al precedente che prevede di istanziare i due Publisher per pubblicare in ROS mappatura e posa.
- **PublishPositionAsPoseStamped**: constatato che *publish\_ros\_camera\_pose* pubblicava in ROS direttamente la posa della camera; che il sistema di coordinate della camera era diverso da quello di ROS(fig. 2.4), portando errori nella successiva visualizzazione in output della posa, è stato realizzata una soluzione a questi problemi: viene fatta una trasformazione della posa dal sistema di coordinate della camera a quello di ROS utilizzando il metodo *TransformFromMat*, dopodiché si è stata data la possibilità all'utente di cambiare il frame di riferimento della posizione, usando il *topic* dato al momento della costruzione *TargetFrameId*. Per fare ciò si usa il metodo *transformToTarget*, poiché, in questo modo, si può avere il riferimento della posa in relazione ad una posizione specifica, ad esempio: per una fotocamera stereo, ORB SLAM3 [1] pubblica la posa rispetto alla camera sinistra, con questo metodo è possibile avere la posa rispetto al centro della camera. La posa viene convertita in un messaggio di tipo *Geometry\_msgs::Pose*, che mantiene le coordinate del punto 3D insieme al quaternione che si riferisce all'orientamento e al *timestamp* di quando è stato generato il messaggio. Infine il messaggio viene pubblicato tramite il *Publisher* specifico. Come parametri di ingresso, il metodo acquisisce, come la precedente versione, la posa in formato *Sophus::SE3f* e il *timestamp* che confluirà nel messaggio. In più è stata aggiunta una variabile booleana per indicare la scelta di effettuare la trasformazione dal *camFrameId* al *targetFrameId*.

- **TrackedMappointsToPointcloud:** in questo metodo è stata aggiunta la possibilità di inserire il colore dei punti della mappa, ridimensionando il vettore *fields*. Questa impostazione è fondamentale per far funzionare il codice insieme al sistema autonomo poiché le informazioni dei coni vengono fuse congiuntamente agli oggetti di tipo **mapPoints** e il colore viene assegnato giallo o blu se il corrispondente ad un oggetto di tipo **MapPoint** si trova posizionato sopra un cono del tracciato.
- **TransformFromMat:** metodo che in ingresso si serve di una posa in formato *Sophus::SE3f* della camera e la restituisce per essere coerente con il sistema di coordinate di ROS. Questo perché, come già detto, il sistema di coordinate di ORB SLAM3 [1] è diverso rispetto a quello di ROS (fig. 2.4). Le operazioni fatte sono le classiche operazioni algebriche con matrici di rotazione.
- **TransformToTarget:** metodo che in ingresso usa una posa in formato *tf2::Transform* - frame di riferimento della posa, vale a dire rispetto al quale si vorrebbe la posa in ingresso. Per attuare la trasformazione si usa l'attributo *tfBuffer* che tramite la funzione **lookUpTransform** controlla tutto l'albero dei frame attuale in ROS e se è disponibile la trasformata, la esegue, altrimenti dà errore e restituisce la stessa posa iniziale.
- **PublishRosTfTransform:** questo metodo era già presente in *common.cc*, anche se con qualche bug. Risolte le problematiche, adesso è possibile usare anche questo metodo per pubblicare la posa della camera in modo che diventi un frame nell'albero di **tf**. Per fare ciò, basta richiamare all'interno della callback questa funzione in ingresso la posa, il *frame\_id* relativo alla mappa e il *parentFrameId*. Que-

sta aggiunta permette coerenza con il sistema dei frame di ROS e ciò porta a estremi benefici, ad esempio che se si vuole inserire un sensore, e sappiamo la sua distanza dalla camera, allora basta creare una *tf\_static\_transformation* e si può avere la posa del robot rispetto a questo nuovo sensore.

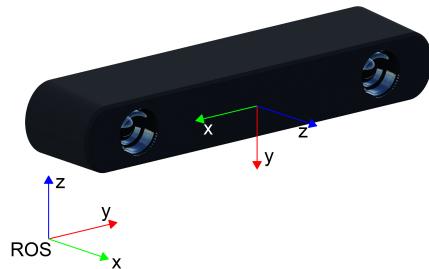


Figura 2.4: Orientamento assi ROS a sinistra, ORB SLAM3 a destra

L'ultimo metodo **SE3fToTfTransform** attualmente rimane non usato.

### Eseguibili

Tutti gli altri file, come accennato, sono divenuti classi (fig 2.3(b)), portando così ad eliminare le varie copie di **ImageGrabber** e integrando i suoi metodi e attributi all'interno della nuova classe. In questo modo si riesce a migliorare la qualità, la leggibilità, la manutenibilità che porta anche a consistenza del codice, risultando così una miglior esperienza e minor possibilità di bug. Si può vedere in fig. 2.5 tutta la serie di operazioni che un nuovo eseguibile attua per far funzionare ORB SLAM3 in real time:

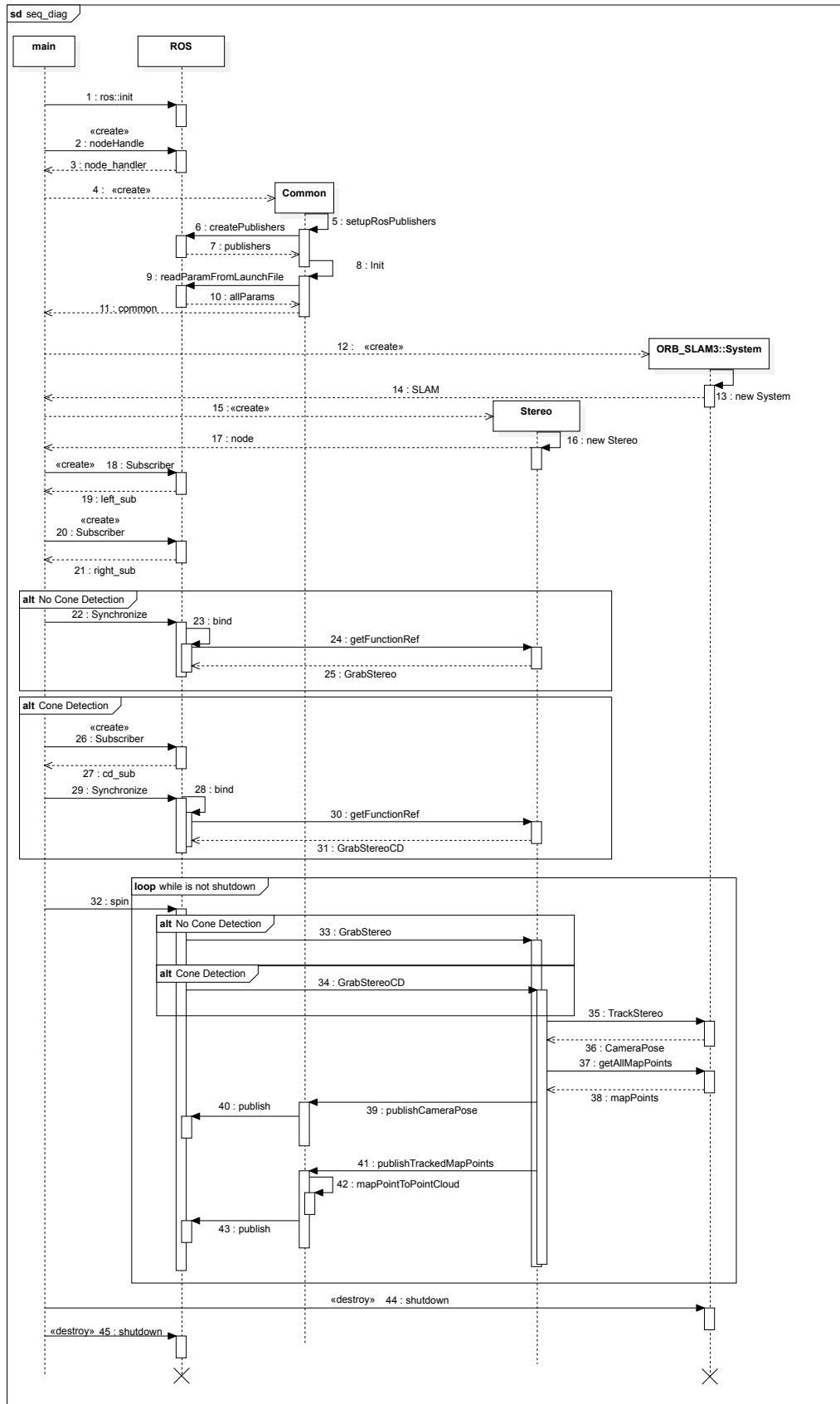


Figura 2.5: Sequence Diagram che rappresenta le operazioni svolte dal nodo ROS eseguibile per camere STEREO.

Dalla figura precedente (fig. 2.5), si possono vedere tutte le operazioni svolte dal wrapper nel caso di utilizzo di una fotocamera STEREO. Inizialmente, tramite la libreria **roscpp** si inizializza ROS e si crea un oggetto di tipo **nodeHandle**; questo verrà poi passato come parametro direttamente alla creazione di un nuovo oggetto di tipo **Common**, che prevederà di istanziare i suoi Publisher e di leggere tutti i parametri di ROS scritti nei file ".launch". Quest'ultimi sono file XML e permettono di definire una lista di parametri da passare al nodo, accessibili attraverso un oggetto di tipo **nodeHandle**. I parametri (*frame\_id*, la *path* dei file che servono ad ORB SLAM3 [1] etc.) vengono scritti in questo file, per poi essere salvati negli attributi interni dell'oggetto di tipo **Common**. Il passo successivo del codice, prevede di istanziare un nuovo oggetto di tipo **ORB\_SLAM3::System**, passandogli come parametro la *path* del vocabolario ORB e la path indicante la posizione del file di configurazione ".yaml", che come detto precedentemente contiene le informazioni di impostazione della camera utilizzata; queste ultime servono a ORB SLAM3 [1] per applicare le formule descritte in cap. 1 necessarie per trovare la distanza, ma anche per rimuovere la possibile distorsione che si ha nell'immagine. Successivamente, si crea un oggetto di tipo **Stereo** (in questo caso), passandogli per parametro sia l'oggetto di tipo **Common**, che l'oggetto di tipo SLAM appena creati. In questo caso sono entrambi puntatori perciò si è andati a passare l'indirizzo dell'oggetto di tipo **ORB\_SLAM3::System** con l'intento di creare direttamente un puntatore ad un oggetto di tipo **Common**. Una volta istanziato un oggetto di tipo **Stereo**, si può accedere ai metodi definiti al suo interno che diventeranno le **callback** degli oggetti di tipo **Subscriber**. Perciò ora, si può procedere a creare quest'ultimi. Un oggetto di tipo **Subscriber** prende come parametri: un oggetto di tipo **nodeHandle**, una stringa che indica il *topic* sul quale prele-

vare i messaggi e la lunghezza della coda dei messaggi. Una volta creati i due **Subscriber** (per immagine sinistra e destra), viene anche creato un oggetto sincronizzatore, che fornisce la libreria **roscpp**, di tipo **time\_synchronizer**. La libreria fornisce inoltre delle **policy** di sincronizzazione del messaggio, che indicano le regole da seguire quando si vuole sincronizzare due *topic* differenti. In questo caso viene utilizzata la *policy* **ExactTime**, poiché le fotocamere fin’ora usate pubblicano immagini perfettamente sincronizzate. Un’alternativa potrebbe essere **ApproximateTime**, in caso i *topic* da sincronizzare non abbiano *timestamp* perfettamente identici. Per creare l’oggetto di tipo **time\_synchronizer**, occorre passargli sia gli oggetti di tipo **Subscriber** precedentemente creati, sia la policy di sincronizzazione. Infine si richiamerà il metodo **registerCallback** dell’oggetto di tipo **time\_synchronizer** che effettuerà il vero **binding** della callback dall’oggetto di tipo **Stereo**. Questi passaggi finali, come si può vedere, sono alternativi, dipendentemente se è stata abilitata la **Cone Detection** o meno in ingresso. Nel caso sia stata abilitata, allora si ha un oggetto in più di tipo **Subscriber** che si sottoscrive al **topic** di output della Cone Detection per le informazioni di collocazione dei coni nell’immagine. Questo oggetto in più, si passa poi per parametro all’oggetto di tipo **time\_synchronizer**. Il metodo della classe **Stereo**, che nei parametri ha anche il campo per gestire il relativo tipo di messaggio, viene registrato come *callback*. Anche questo messaggio deve essere sincronizzato altrimenti si avrebbero problemi nell’individuare l’immagine a cui si riferiscono i dati che arrivano dalla Cone Detection.

Successivamente si esegue la funzione **spin**, della libreria **roscpp**, consentendo quindi di gestire gli eventi di comunicazione come messaggi in arrivo, perciò di eseguire il codice all’interno delle *callbacks* non appena arriva un

messaggio. Quindi nel diagramma, è illustrato il modo in cui, una volta arrivato un messaggio, questo viene gestito alternativamente chiamando il metodo **GrabStereo** o **GrabStereoCD**, in caso non si usi l'input della Cone Detection o si rispettivamente. Il codice successivo risulta uguale per entrambe le callbacks:

- si richiama il metodo **TrackStereo** di ORB SLAM3 [1], in caso di opzione con Cone Detection, allora a questo viene passato pure un oggetto di tipo **json**, contenente le informazioni dei coni. Il metodo restituisce subito la posa dell'immagine passata.
- dopodiché, si ottengono i punti ORB della mappa richiamando il metodo **GetAllMapPoints** ed infine si effettuano le pubblicazioni di posa e mappa rispettivamente richiamando i metodi di **Common** (**PublishCameraPose** e **PublishTrackedMapPoints**).

Come detto precedentemente, questo codice verrà eseguito quando si riceveranno immagini, altrimenti si rimane in attesa. In entrambi i casi se viene invocato il metodo **shutdown** da ROS, si ferma tutto rilasciando il thread principale alla funzione main. Infine si esegue lo shutdown del sistema SLAM e si riesegue un altro shutdown in modo da essere sicuri che tutto venga chiuso all'uscita del nodo.

### Modifiche aggiuntive

Come già accennato, è stata modificata la API di ORB SLAM3, questa purtroppo è stata necessaria in quanto serviva per avere un metodo che permettesse di accedere alla lista dei punti della mappa **GetAllMapPoints**. L'aggiunta di questo metodo era fondamentale altrimenti il *wrapper* non avrebbe funzionato a dovere, mancante appunto della possibilità di pubblicare la map-

pa. Per recuperare dentro System, la mappa di punti, si fa riferimento ad un oggetto di tipo **Atlas**(vedi 1.4.3), questo contiene tutte le mappe, inclusa quella attualmente attiva. Ottenendo un riferimento di quest'ultima, si riesce ad accedere a tutti i suoi **MapPoints**. È stato poi risolto un bug che comprendeva l'utilizzo della tipologia di fotocamere *Rectified*, ovvero che non presentano distorsioni: in pratica non era stato fornito del codice che inizializzasse le componenti della camera di destra. La risoluzione è stata appunto aggiungendo il codice predisposto a farlo.

Infine è stata inoltre aggiunta la classe **StereoKittiTest** per eseguire ORB SLAM3 sul dataset kitti [3] (vedi cap.3). Questa classe ha una callback in più, registrata ad un oggetto di tipo Subscriber indipendente, che serve per salvare la mappa generata su file. Si cerca di far funzionare tramite ROS le varie sequenze di questo dataset in modo poi da valutare il wrapper e ORB SLAM3 [1]. In questo caso, il dataset viene reso disponibile tramite un nodo ROS che pubblica le immagini esattamente rispettando le tempistiche dei loro *timestamp*. Una volta finita la sequenza, questo nodo invia il messaggio di "save", e, siccome c'è un oggetto di tipo Subscriber in ascolto, appena riceve questo messaggio, provvede a salvare la Sequenza delle pose richiamando il metodo di **ORB\_SLAM3::System SaveTrajectoryKitti** che creerà il file. Infine viene invocato il messaggio di shutdown di ROS per concludere l'esecuzione.

# Capitolo 3

## Valutazione

Questo capitolo tratterà della valutazione del wrapper da me sviluppato, descritto precedentemente (2.2). In particolare si esporranno per prima cosa le tecniche utilizzate per la valutazione per poi riportare i risultati ottenuti. Tutti i test sono stati eseguiti su un PC con un i7 6700K e 64GB di memoria RAM.

### Tecniche usate

Sono state usate 2 tecniche principali di valutazione.

- **Valutazione odometrica:** ovvero si confronteranno le traiettorie di ORB SLAM3 [1] con quelle generate da ORB SLAM2 [4]. La valutazione si dividerà in 2 tecniche diverse:
  - **confronto con ground truth:** si valuteranno i due sistemi slam utilizzando il dataset di odometria di KITTI [3], uno standard per quanto riguarda la *Computer Vision*. Questo dataset, disponendo di ground truth, permette di capire quale dei due sistemi è più accurato.

- **Confronto con SfM:** utilizzando un software per *Structure from Motion*, è possibile creare dei test fatti col nostro hardware, con i quali sarà poi possibile creare una visual odometry in modo offline, rendendola molto più accurata delle traiettorie dei due sistemi SLAM. Questa traiettoria verrà usata come *ground truth* in modo da valutare gli slam su tracciati e percorsi più consoni nel mondo della Formula Student.
- **Valutazione computazionale:** si confronterà il costo computazionale di ORB SLAM3 [1] con e senza il wrapper. Si valuterà anche la parte di *tracking* rispetto ad ORB SLAM2 [4].

## 3.1 Valutazione odometrica

### 3.1.1 *The KITTI Vision Benchmark Suite*

La *vision benchmark suite* di KITTI [3], fornisce un totale di 22 sequenze di immagini stereo in formato png *lossless* su cui fare testing, ma solo 11 di queste hanno *ground truth* pubblica.

Il setup per i test di KITTI, è rappresentato in figura 3.1: La vettura è fornita di quattro camere, due a colori modello Point Grey Flea 2 (FL2-14S3C-C)e 2 in bianco e nero modello Point Grey Flea 2 (FL2-14S3M-C), tutte con 1.4 Megapixels. Il setup è inoltre fornito da un Lidar della Velodyne, il modello HL-64E, e un sistema di navigazione inerziale (GPS/IMU) della OXTS, modello RT 3003. Infine si ha a disposizione i vari *timestamp* dell'esatto momento di estrazione dell'immagine o dato.



Figura 3.1: Il setup per raccogliere dati di visione. Possiamo vedere i vari sensori usati sul tetto del veicolo. In questo caso, solo il *Rig* di camere stereo viene considerato.



(a)



(b)

Figura 3.2: Immagini provenienti dal KITTI dataset [3], (a) immagine urbana, (b) immagine in superstrada

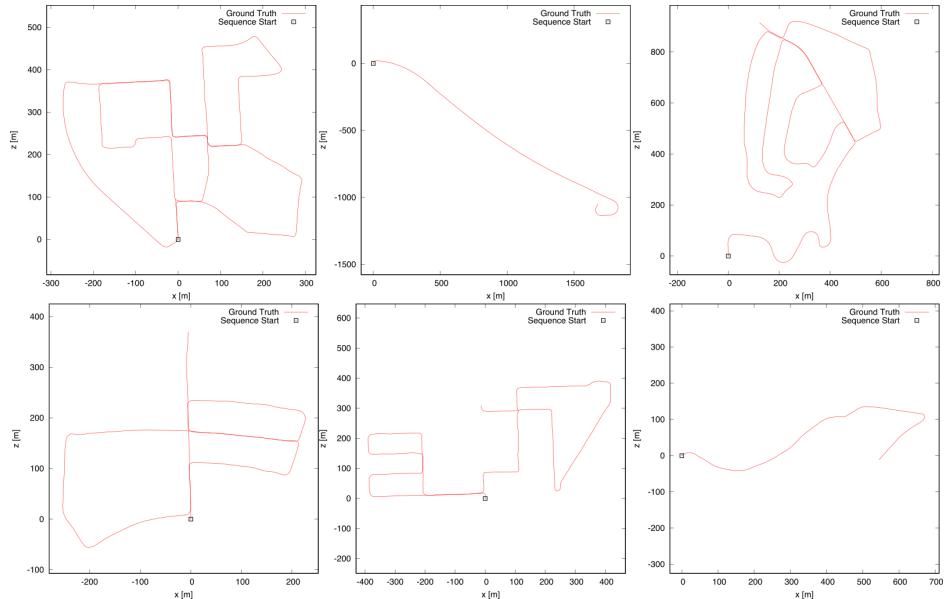


Figura 3.3: Alcune delle undici sequenze usate per testare ORB SLAM3 [1] e ORB SLAM2 [4]

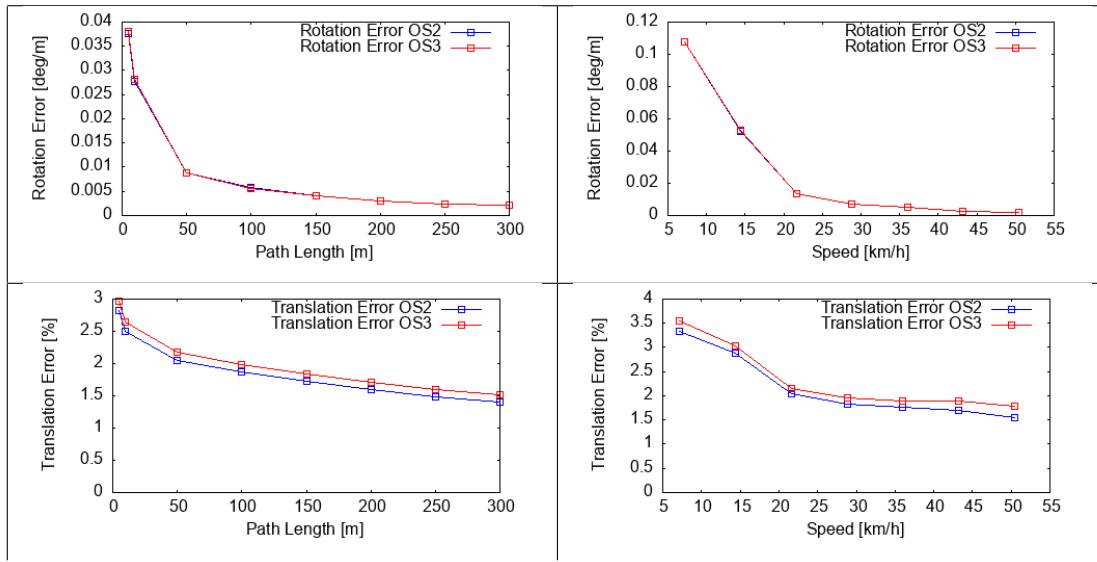
I test effettuati di seguito, riguardano il confronto delle traiettorie ricavate da ORB SLAM3 [1] e ORB SLAM2 [4] in relazione alla *ground truth* fornita dal dataset [3]. La sequenza di immagini è stata elaborata e pubblicata tramite un Nodo ROS, con la stessa frequenza ricavata dai *timestamp* forniti. Per avere una misura più accurata, sono state fatte 10 esecuzioni di ogni singola sequenza per poi estrarne la media. L'ambiente dei tracciati riguarda soprattutto ambiente urbano, ma sono stati fatti test pure in superstrada come si può vedere in figura 3.2. Tutti i test sono stati eseguiti facendo caricare lo stesso vocabolario di *visual words* di feature ORB.

Dalla figura 3.3 si può vedere l'andamento delle traiettorie delle sequenze del dataset, alcune presentano molti loop, con i quali si riuscirà a testare tutti i vari algoritmi di Global Bundle Adjustment. Altre sequenze compongono traiettorie molto semplici, in queste si potrà analizzare il drift accumulato man mano che la vettura si allontana dal punto iniziale.

## Risultati

Di seguito si analizzeranno le sequenze che risultano più rilevanti e poi si analizzeranno le medie dei vari risultati.

**Sequence 00** Il percorso 00 comprende un test molto lungo di 4541 immagini, circa 7 minuti di test, in cui si affrontano strade urbane con diversi loop. Di seguito i risultati:



Questi grafici sono la media di cammini con posizione iniziale e lunghezza del cammino variabile, in questo modo si cerca di capire il comportamento medio del sistema. Si evidenzia subito come ci sia un maggiore errore di traslazione passando da ORB SLAM2 [4] a ORB SLAM3 [1], mentre invece si ha una differenza irrilevante per quanto riguarda l'errore di rotazione. Molto probabilmente il nuovo metodo di chiusura del loop non è più accurato di quello proposto già dalla versione precedente. Di seguito si possono vedere i plot a confronto 3.4. Da queste immagini si capisce quanto i due sistemi slam sono simili tra loro: possiamo vedere che accumulano il drift esattamente alla stessa maniera, pur rimanendo in vantaggio ORB SLAM2 [4].

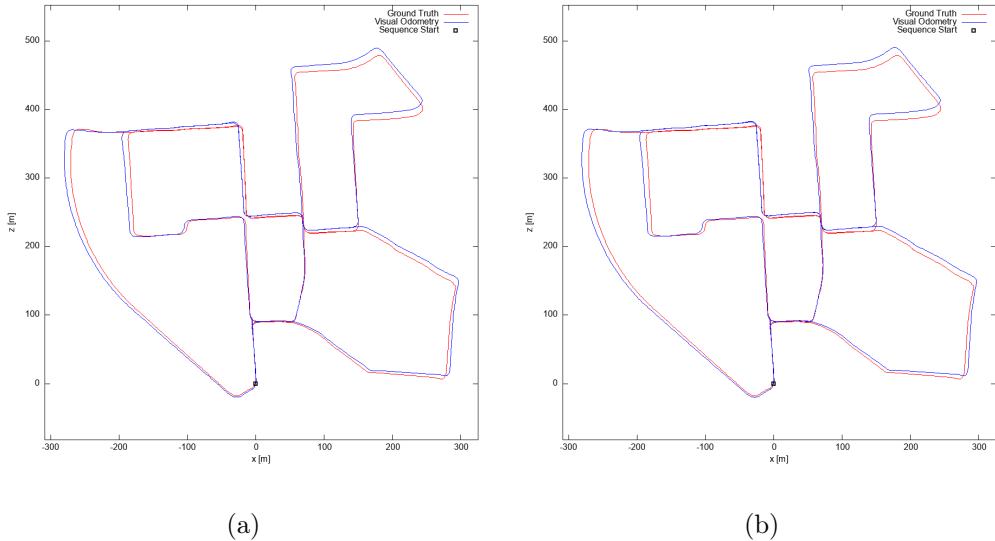
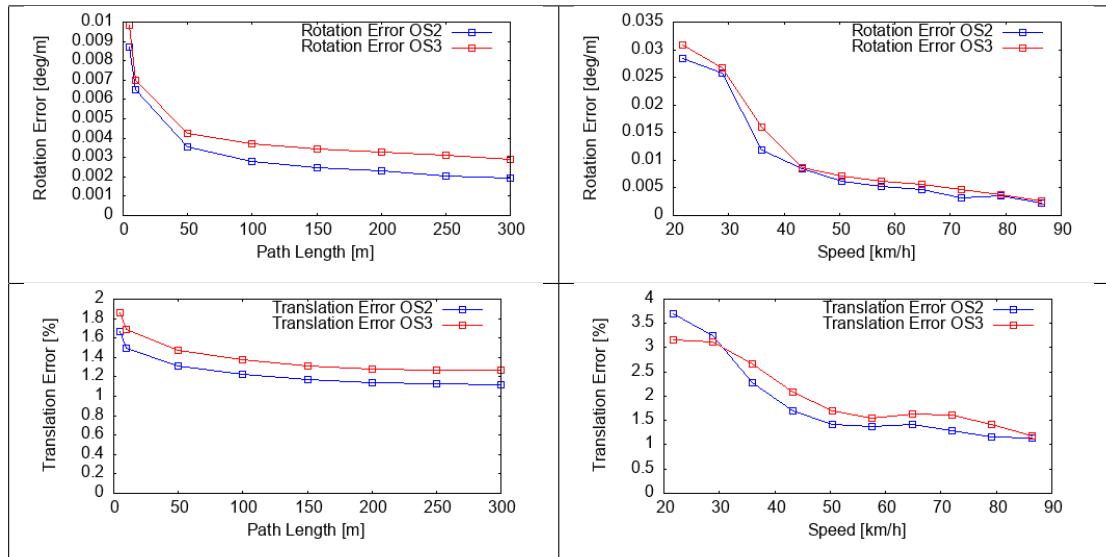


Figura 3.4: (a) traiettoria di ORB SLAM2 [4], (b) traiettoria di ORB SLAM3 [1]

**Sequence 01** Si è scelto di analizzare questa sequenza poiché non presenta alcun loop, perciò si può benissimo analizzare l’accumulo del drift in entrambi i sistemi.



La prima cosa che si può notare, è che l’errore è più marcato rispetto

alla sequenza precedente, confermando ancora di più la maggior precisione di ORB SLAM2 [4]. Si può notare anche un netto miglioramento per quanto riguarda l'errore di rotazione. Nei grafici delle traiettorie (fig 3.5), si può notare l'enorme differenza di drift che si accumula nei due sistemi.

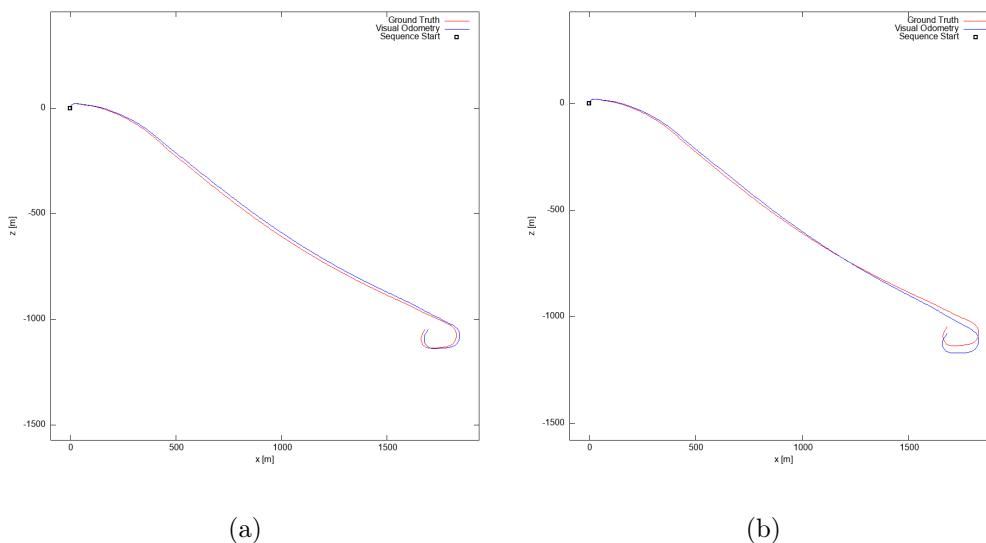
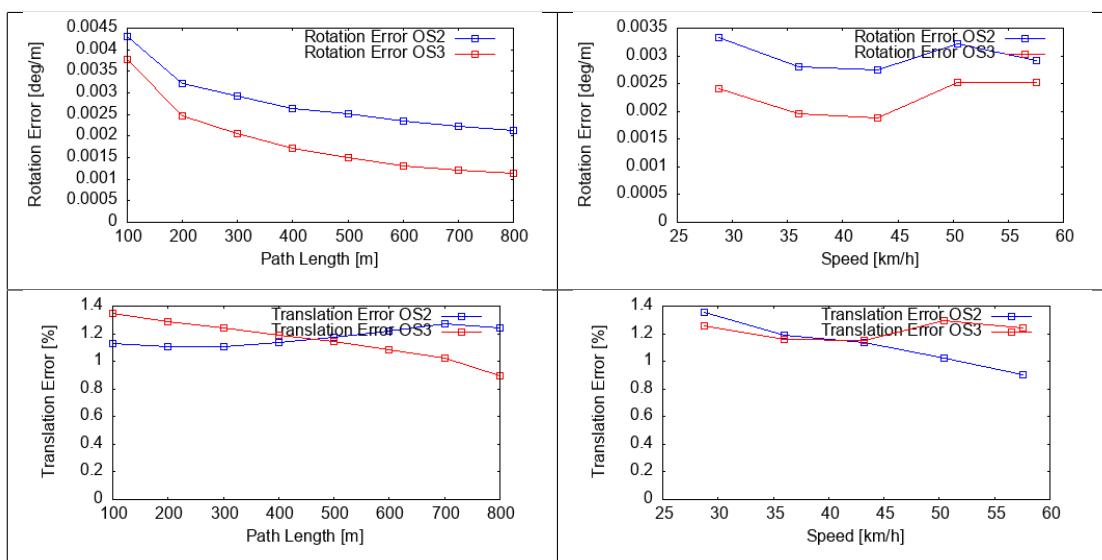


Figura 3.5: (a) traiettoria di ORB SLAM2 [4], (b) traiettoria di ORB SLAM3 [1]

**Sequence 09** In questa sequenza invece si ha tutt'altro risultato:



Si può notare come, sia in vantaggio ORB SLAM3 [1], soprattutto per l'errore di rotazione, rimanendo più preciso, ma anche per *path* di maggiore lunghezza. Andando a vedere com'è fatta la sequenza (fig. 3.6), si può notare come ORB SLAM3 [1] riesca a chiudere il ciclo, mentre il ORB SLAM2 [4] no, ciò conferma la maggiore velocità nella chiusura del loop. Ciò porta un estremo avvantaggiamento ad ORB SLAM3, poiché nella Formula Student è fondamentale sapere il prima possibile quando si è vicini al punto iniziale. A più alta velocità ORB SLAM2 [4] sembra riprendersi ottenendo un miglior risultato.

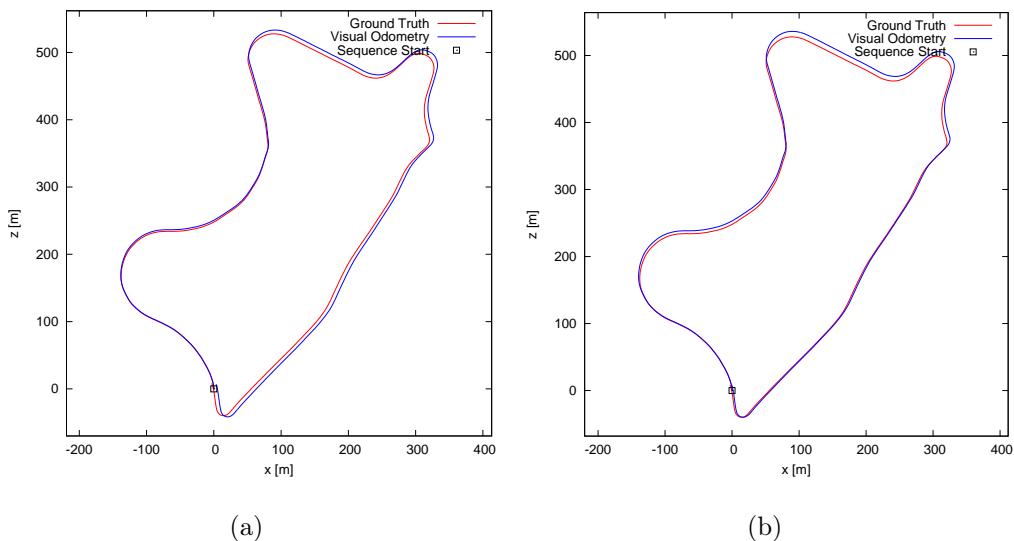
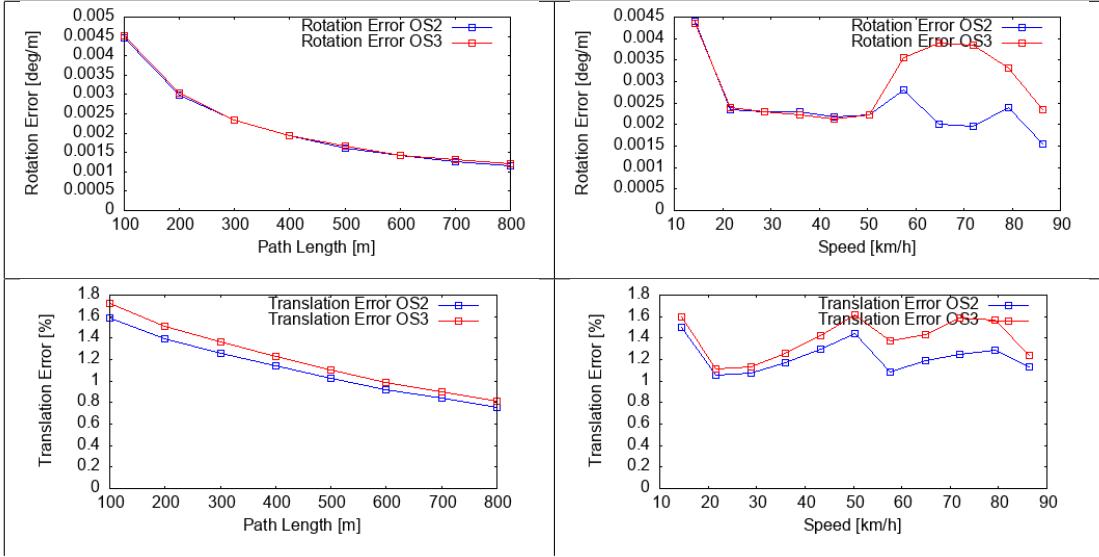


Figura 3.6: (a) traiettoria di ORB SLAM2 [4], (b) traiettoria di ORB SLAM3 [1]

**Valori medi** Di seguito si propongono i grafici dei valori medi su tutte e undici le sequenze.



Possiamo vedere come in realtà i due sistemi siano molto simili, soprattutto per l'errore di rotazione in relazione alla lunghezza del *path*. Mentre è sempre in leggero vantaggio per la precisione in tutti gli altri 3 tipi di grafo. Ciò denota che comunque ORB SLAM2 [4], rimane più preciso, pur avendo aggiunto tutte le tecniche nuove per cercare di migliorarlo. Possiamo notare uno strano comportamento nel grafico velocità-errore di rotazione, dove ORB SLAM3 [1] ha una curva di errore, con massimo a circa 75 km/h. Questo comportamento probabile sia dovuto a curve lunghe prese ad alta velocità, che comportano un errore più grande. In generale comunque l'errore trovato a queste velocità non è troppo rilevante in quanto, soprattutto nei primi anni, la vettura del Firenze Race Team non raggiungerà mai velocità così elevate.

Questi sono i risultati medi per tutte e undici le sequenze. Dimostrano ancora una volta quanto i due algoritmi, sono molto simili tra loro. I due sistemi SLAM hanno difetti e pregi in punti diversi, che comunque portano

Errore Medio		
Type of Error	ORB SLAM2	ORB SLAM3
Rotation Error (deg/m)	0.000040	0.000040
Translation Error (%)	<b>1.1601</b>	1.2406

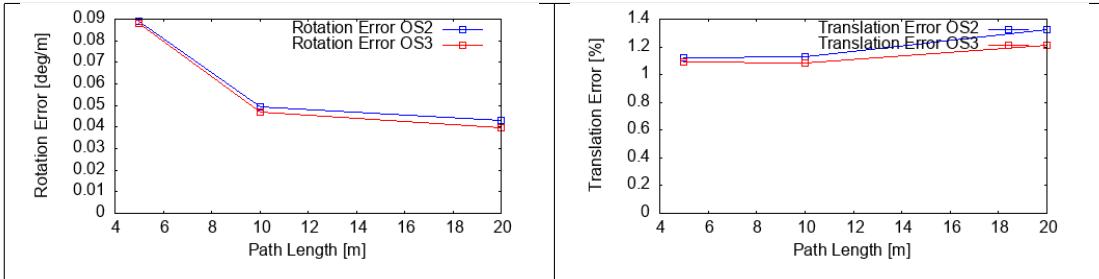
a ribilanciarli, sempre leggermente in vantaggio per ORB SLAM2 [4].

### 3.1.2 *Structure from Motion*

I test eseguiti con questa modalità, sono stati fatti estrapolando i dati da una camera stereo le cui immagini della sola camera sinistra sono state poi usate insieme ad **AliceVision** per creare una *Structure from Motion*. Questo passaggio è stato fatto per cercare di avere una Ground Truth; una Structure from Motion viene realizzata analizzando le immagini offline estraendo molte features con un elevato numero di descrittori. Ciò porta ad una maggiore precisione rispetto a sistemi SLAM online come quelli che si stanno valutando. Con una sola camera non è possibile trovare la scala reale delle pose e, data la grande quantità di ottimizzazioni, è anche necessario un allineamento. Perciò è stata generata una trasformazione usando il metodo di Horn in forma chiusa tra la traiettoria generata da **AliceVision** e i due sistemi SLAM. In particolare si effettua una trasformazione di scala per la traiettoria proveniente dal software di *Structure from Motion* e si applica alle due traiettorie dei sistemi SLAM la trasformazione di rotazione e traslazione precedentemente trovate con il metodo di Horn. Così facendo le traiettorie risultano più simili possibile alla traiettoria della Ground Truth. L'ultimo passaggio prevede una trasformazione generale delle 3 traiettorie in modo

che la prima posa risulti posizionata all'origine. Sono state usate le stesse tecniche di valutazione del KITTI [3] dataset. Infine i test sono stati presi da una media di 5 prove consecutive di ognuna delle sequenze proposte in modo da avere un risultato più accurato. Le immagini hanno una risoluzione di 1280x720, registrate con la fotocamera stereo Zed2i di Stereolabs. Entrambi i test sono stati fatti con 1200 features per immagine come parametro. Questi test sono stati fatti appositamente per cercare di riprodurre l'ambiente della formula student, per cercare di capire se si avesse risultati diversi dal dataset KITTI.

**Acceleration** Questo test prevede un tracciato dritto, è una versione più piccola(di circa 20 metri) di un tracciato della missione della Formula Student "Acceleration" , generato in un parcheggio. Il circuito è costruito usando i coni con standard da formula student. le immagini sono a 60 fps.



Incredibilmente, questi test ribaltano leggermente i risultati ottenuti con il dataset KITTI [3], portando in vantaggio ORB SLAM3 [3]. Sono stati prodotti soltanto i grafici rispetto alla lunghezza del percorso poiché la velocità è rimasta costante durante tutto l'acceleration, perciò quei risultati sono irrilevanti. Dai valori medi presenti nei grafici precedentemente analizzati e dalla tab. 3.1.2, si può notare subito che l'errore di rotazione è di un ordine di grandezza più grande rispetto al dataset KITTI. Dalla fig. 3.7, si può notare benissimo come la ground truth non è minimamente a favore di ORB

Errore Medio Acceleration		
Type of Error	ORB SLAM2	ORB SLAM3
Rotation Error (deg/m)	0.001181	<b>0.001147</b>
Translation Error (%)	1.1544	<b>1.1082</b>

SLAM3 [1] o ORB SLAM2 [4]. Perciò pur non essendo troppo accurata, è possibile comunque usarla come ground truth per cercare di estrapolare comunque una valutazione approssimativa dei due sistemi SLAM. Sicuramente risulta molto più precisa di entrambi i sistemi SLAM real-time.

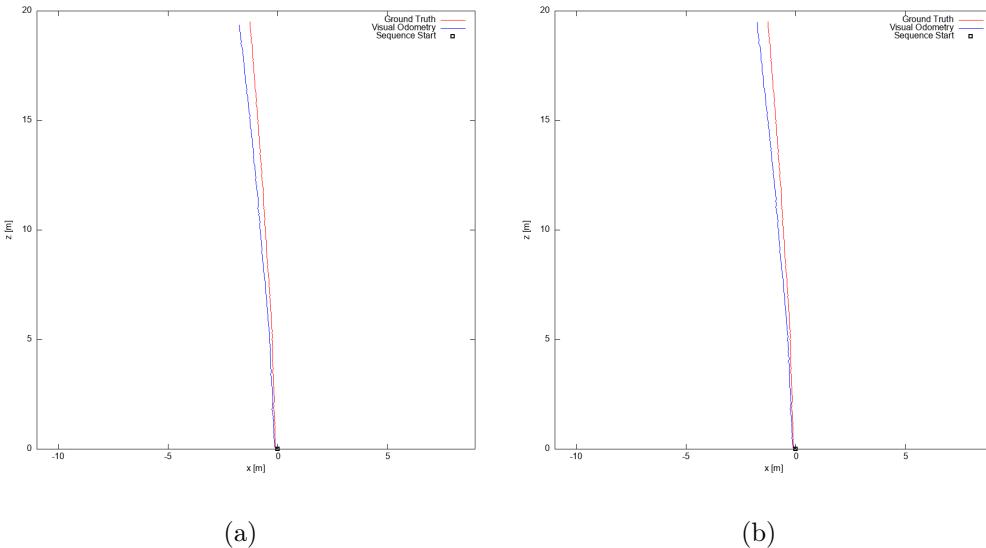
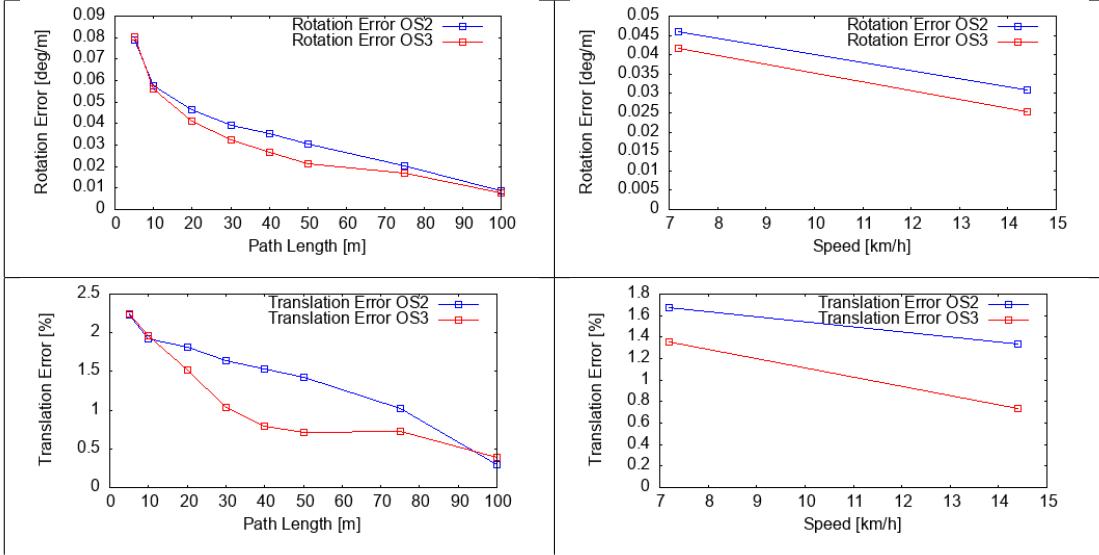


Figura 3.7: (a) traiettoria di ORB SLAM2 [4], (b) traiettoria di ORB SLAM3 [1]

**Track drive** In questa prova si esegue un solo giro di una pista, si riesce così a testare ancora una volta la velocità dei due algoritmi nel rilevare un loop. In questo test le immagini sono campionate a 20 fps dalla camera Zed2i.



Anche in questo caso senza alcun dubbio ORB SLAM3 [1] ha una migliore precisione rispetto ad ORB SLAM2 [4]. Guardando le traiettorie in fig. 3.8, ORB SLAM2 [4] ha un errore maggiore nel rettilineo e nella curva successiva, mentre ORB SLAM3 [1] ha solo un leggero drift laterale. Ovviamente non si può prendere per ottima la traiettoria della **Structure from Motion**, ma guardando anche la tab. 3.1.2, i risultati non sono poi così diversi dal dataset KITTI [3], con circa l'1% di errore di traslazione e circa lo stesso ordine di grandezza per quello di rotazione. Ciò conferma la possibilità di usare la *Structure from Motion* che fornisce AliceVision per valutare anche in futuro nuovi test del Firenze Race Team. Tornando ai due sistemi, ORB SLAM3 [1] risulta migliore in ogni test eseguito con questa tecnica e con l'hardware che il Firenze Race Team ha a disposizione. Perciò si può affermare che in un ambiente più consono alla formula student, si ha un migliore risultato.

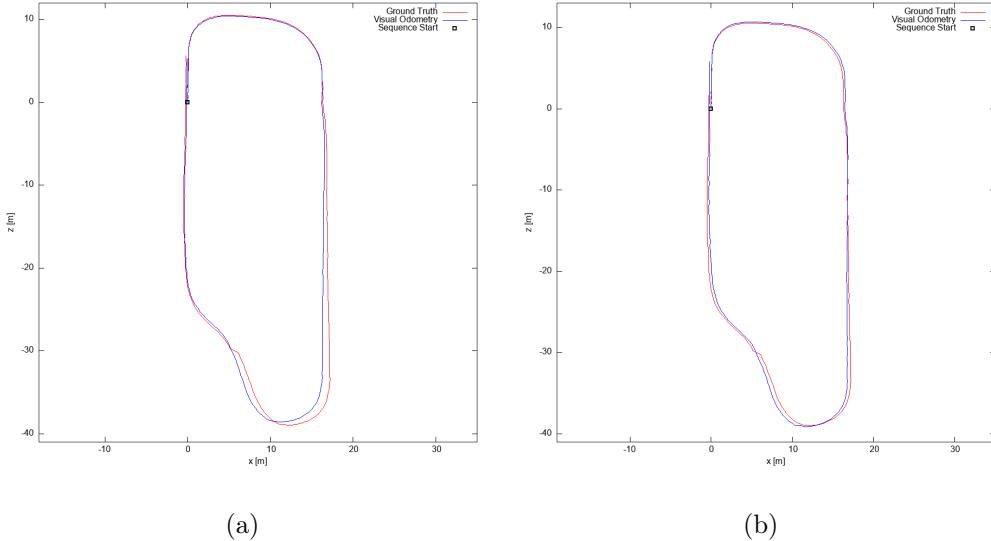


Figura 3.8: (a) traiettoria di ORB SLAM2 [4], (b) traiettoria di ORB SLAM3 [1]

Errore Medio Acceleration		
Type of Error	ORB SLAM2	ORB SLAM3
Rotation Error (deg/m)	0.000800	<b>0.000727</b>
Translation Error (%)	1.6698	<b>1.3521</b>

## 3.2 Valutazione computazionale

### 3.2.1 Confronto con ORB SLAM3 nativo

Di seguito si propongono vari tempi estratti dall'esecuzione di ORB SLAM3 [1] con e senza l'utilizzo del wrapper. In questo modo si quantifica tutta la perdita di prestazioni che si ha avviando il sistema in ROS. Si valuteranno tutti gli aspetti di ORB SLAM3 [1], grazie agli sviluppatori che hanno lasciato il codice per poter estrarre tutti questi tempi in modo molto semplice.

Test eseguiti sulla sequenza 00 del dataset KITTI, a 2000 feature per frame, facendo una media di 10 ripetizioni. Il wrapper riesce a comportarsi a livello

Operation set	Operation	ORB SLAM3	OS3 ros wrapper
Tracking	Orb Extraction	24.71359±3.55904	25.76672±5.40369
	Stereo Matching	4.59615±0.90247	5.64361±1.29272
	Pose Prediction	2.21871±0.84956	2.51272±0.98541
	LM track	4.51815±2.57844	5.62798±4.36470
	new KF Decision	0.23302±0.46206	0.31150±0.90404
Local Mapping	KF Insertion	7.71498±1.94253	9.38148±3.75025
	MP Culling	0.40786±0.09846	0.46799±0.45358
	MP Creation	22.09222±6.32344	27.70486±10.08730
	LBA	90.21890±71.43946	104.17778±77.28825
	KF Culling	4.66844±3.91325	5.97626±6.02706
Place Recognition	Database Query	6.01794±3.51149	6.24415±3.85100
	MP Culling	16.94626±5.74096	18.72483±6.30433
Loop Closing	Loop Fusion	15.06226±7.45652	15.63366±5.75580
	Essential Graph	447.10864±142.45692	549.68369±241.10940
Full GBA	GBA	5165.566±1794.34	6131.036±2578.725
	Map update	49.96812±18.54416	72.70873±42.03602

Tabella 3.1: Tabella che riassume tutti i tempi di esecuzione di ORB SLAM3. I tempi sono in ms.

di precisione esattamente come il nativo, eppure, come si può vedere dalla tabella, l'aggiunta di ROS aumenta i tempi di tutte le varie operazioni che avvengono all'interno del sistema. La parte più colpita da questo calo di prestazioni, riguarda più che altro la parte di ottimizzazione che è la parte computazionalmente più costosa del sistema, in particolare si ha anche ben

un secondo in più per quanto riguarda l'esecuzione della Full Global Bundle Adjustment. Mentre per il resto, si ha un calo di prestazioni non così netto, ma si ha un grande aumento di deviazione standard che sta a significare la variabilità dei tempi calcolati. Ciò significa che ROS diventa più pesante computazionalmente in relazione a cosa sta succedendo sul framework di ROS, portando molta più incertezza sui tempi di esecuzione.

### Confronto con ORB SLAM2

Il confronto avviene tenendo in considerazione le seguenti operazioni: l'estrazione delle *features* ORB, lo stereo matching, la predizione della posa e il successivo tracking nella mappa locale. infine la decisione di un nuovo keyframe(vedi **TRACKING** in fig. 1.4). Confrontato il wrapper sviluppato da me con ORB SLAM2 nativo. Di seguito la tabella dei tempi della sola parte di *tracking* di entrambi i sistemi SLAM:

Mean tracking times		
features per image	ORB SLAM2	ORB SLAM3
1000	45,2522(ms)	<b>29,6973</b> (ms)
1500	53,7255(ms)	<b>33,6823</b> (ms)
2000	65,491(ms)	<b>40,1916</b> (ms)
2500	72,8744(ms)	<b>45,363</b> (ms)
3000	85,1621(ms)	<b>52,28477</b> (ms)

Tabella 3.2: tabella che riassume i tempi di tracking a vari livelli di features estratte per immagine

I dati provengono dalla sequenza 00 del dataset KITTI, perciò si hanno ben 4151 immagini da cui poi ricavare media. Da questi dati possiamo vedere quanto è molto più ottimizzato ORB SLAM3 [1] rispetto al 2, consideran-

do che si ha un netto vantaggio di circa 20 secondi in questi passaggi. Ciò dimostra quanto, almeno la parte di tracking, sia stata maggiormente ottimizzata. Questo permette di elaborare molte più immagini al secondo, arrivando a circa 22 fps in media nel caso di 2500 features per immagine, quasi raddoppiando la capacità massima di ORB SLAM2 [4] che arriva a massimo 13 fps circa di media. Si può anche vedere in fig 3.9 una curva molto più piatta per ORB SLAM3 [1], mentre ORB SLAM2 [4] tende a salire molto più vertiginosamente. Un risultato eccezionale per ORB SLAM3 [1].

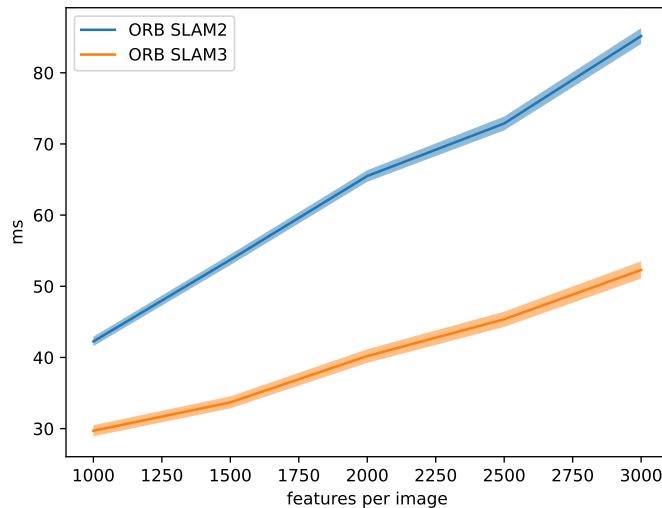


Figura 3.9: Andamento dei tempi al variare delle feature estratte.

### 3.3 Conclusioni

Dopo tutte queste valutazioni, si può concludere che il wrapper di ORB SLAM3 [1] risulta essere ben ottimizzato, dimostrando di essere più efficiente addirittura rispetto a ORB SLAM2 [4] nativo. Inoltre ha una più veloce loop detection, essenziale nel campo della Formula Student. ORB SLAM2 [4]

---

risulta essere in generale più preciso della nuova generazione, questo molto probabilmente è dovuto anche alla maggiore complessità sui descrittori delle features ORB che vengono estratti. Con l'hardware del Firenze Race Team, cercando di simulare l'ambiente della formula Student, si hanno risultati molto positivi, che portano in vantaggio ORB SLAM3. Ciò è rilevante in quanto gli ambienti dove funzionerà questo sistema SLAM, saranno sempre piste automobilistiche. Con la funzionalità della multimappa, si aumenta la robustezza del sistema andando a non eliminare tutta una mappatura fatta fino ad un track lost. Il supporto dell'IMU permetterà futuri sviluppi migliorando ancora di più la precisione. Si conclude quindi la tesi affermando che il passaggio da ORB SLAM2 [4] ad ORB SLAM3 [1] risulta una buona scelta per il sistema SLAM del Firenze Race Team.

# Bibliografia

- [1] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual-inertial, and multimap slam. *IEEE Transactions on Robotics*, 37(6):1874–1890, 2021.
- [2] Dorian Gálvez-López and Juan D Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.
- [3] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.
- [4] Raul Mur-Artal and Juan D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE transactions on robotics*, 33(5):1255–1262, 2017.
- [5] Raúl Mur-Artal and Juan D Tardós. Visual-inertial monocular slam with map reuse. *IEEE Robotics and Automation Letters*, 2(2):796–803, 2017.
- [6] Raúl Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. Orb-slam: A versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

- [7] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.