UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea magistrale in Ingegneria Informatica e dell'Automazione

IMPLEMENTAZIONE DI METODI PROBABILISTICI LIDAR-BASED PER LA LOCALIZZAZIONE INDOOR FINALIZZATI ALLA NAVIGAZIONE AUTONOMA DI UNA PIATTAFORMA ROBOTICA MOBILE

IMPLEMENTATION OF INDOOR LOCALIZATION PROBABILISTIC LIDAR-BASED METHODS FOR AUTONOMOUS NAVIGATION OF A MOBILE ROBOTIC PLATFORM

Relatore:
Prof. Ippoliti Gianluca

Candidato:
Andreozzi Carmen

Correlatori:
Ing. Di Buò Gianluca
Prof. Orlando Giuseppe

A.A. 2022 / 2023

Ai miei genitori e alle mie sorelle

A Fabrizio e alla sua famiglia

Alle mie amiche

A Gianluca e a tutti i collaboratori di IDEA

A nonna Vera

Sommario

Questa tesi ha come scopo quello di illustrare il lavoro svolto durante l'esperienza di tirocinio presso l'azienda IDEA Soc. Coop. di Ancona.
L'obiettivo del progetto consiste nello sviluppo di un sistema di localizzazione e navigazione autonoma in ROS che permetta ad un AMR a guida differenziale di muoversi autonomamente all'interno di un ambiente indoor.
Un algoritmo di navigazione autonoma prevede la generazione di una mappa dell'ambiente circostante e la stima della posizione del robot. Ai fini di questo progetto, questi task sono stati svolti separatamente, in modo da poter testare ed utilizzare diversi approcci per ogni attività. Di conseguenza, i passi procedurali seguiti per il raggiungimento dell'obiettivo sono:

- **Mapping**: vengono utilizzate le informazioni relative alle scansioni effettuate dal sensore RP LIDAR A1 per generare una mappa;
- **Localizzazione locale**: vengono utilizzati i dati odometrici che forniscono informazioni sulla posizione e sulla velocità del robot dal punto di vista locale;
- **Localizzazione globale**: l'odometria e le scansioni laser vengono fuse per stimare lo stato del robot all'interno della mappa. Per questo task sono stati presi in considerazione due diversi approcci: un filtro particellare e un filtro di Kalman esteso.
- **Navigazione autonoma**: ottenuta la mappa e la localizzazione all'interno di essa, si può fornire all'algoritmo un goal che verrà raggiunto tramite una

pianificazione della traiettoria e dei comandi di velocità inviati alle ruote.

Nel lavoro di tesi sono state ripercorse tutte queste tappe, partendo da un primo approfondimento sulla struttura hardware e software fino ad arrivare all'analisi dei test svolti.

Di seguito è riportato lo schema di funzionamento generale.

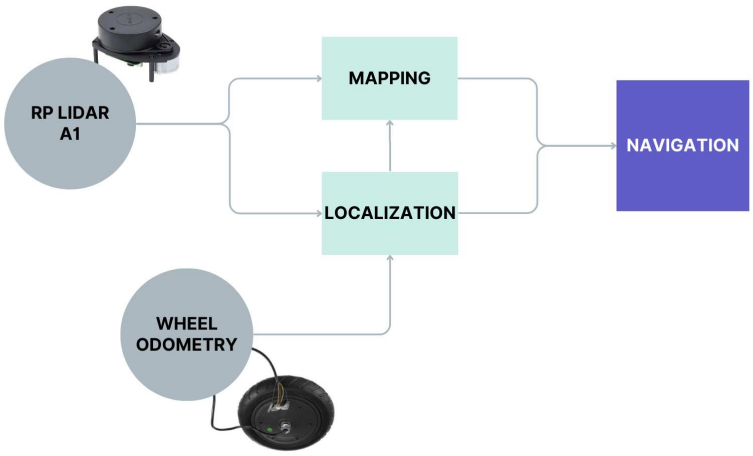

Figura 1: Schema di funzionamento

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Autonomous Mobile Robot (AMR)

Un AMR (Autonomous Mobile Robot) è una tipologia di robot in grado di mappare l'ambiente circostante con lo scopo di muoversi e spostarsi autonomamente, evitando gli ostacoli. Per raggiungere questo obiettivo, gli AMR sono caratterizzati da tecnologie innovative sia dal punto di vista hardware che dal punto di vista software, da sensori all'avanguardia fino ad arrivare alle più avanzate tecniche di Intelligenza Artificiale.

Figura 1.1: Esempi di AMR

Inoltre, è bene distinguere gli AMR dagli AGV (Automated Guided Vehicle): questi ultimi sono robot in grado di seguire una missione predeterminata entro le tempistiche desiderate. In particolare, la differenza tra le due tipologie consiste

nella necessità degli AGV di avere a disposizione un tracciato da seguire, diversamente dagli AMR che, grazie ai sensori e alle tecnologie utilizzate, interpretano autonomamente l'ambiente circostante per evitare gli ostacoli durante la navigazione. In aggiunta, è possibile implementare soluzioni per cui l'AMR sia in grado di modificare dinamicamente la propria traiettoria sulla base di un obiettivo da raggiungere.

Figura 1.2: Esempio di AGV

Ad oggi, i settori che richiedono l'utilizzo degli AMR sono sempre più numerosi. Questa tipologia di robot, infatti, non risulta più conveniente soltanto dal punto di vista operativo, ma anche per quanto riguarda attività pericolose per l'uomo, come ad esempio lo spostamento di carichi pesanti.

1.2 L'azienda

Figura 1.3: Logo IDEA

L'azienda in cui è stato sviluppato questo lavoro di tirocinio e tesi si chiama IDEA Soc. Coop. IDEA è nata nel 2007 ad Ancona come spin-off accademico

dell'Università Politecnica delle Marche grazie alla vincita del concorso eCapital nel 2006 da parte dei soci. IDEA fornisce ai clienti soluzioni personalizzate e innovative, lavorando nei seguenti settori:

- **Automazione**: IDEA progetta e realizza sistemi di automazione industriale per l'automatizzazione, la supervisione e l'ottimizzazione del processo produttivo;
- **Elettronica**: IDEA si occupa di tutte le fasi di progettazione di circuiti elettronici o apparecchiature complete;
- **Industra 4.0**: si tratta della digitalizzazione e dell'interconnessione dei processi industriali;
- **Robotica**: IDEA si occupa dell'integrazione di sistemi robotizzati intelligenti per la Smart Manufactoring;
- **Sistemi di Visione**: l'azienda si dedica anche alla progettazione e allo sviluppo di sistemi di visione artificiale innovativi e personalizzati.

Questo lavoro di tirocinio e tesi rientra all'interno dell'azienda nel settore di **Ricerca e Sviluppo**, in quanto consiste nella realizzazione di un progetto innovativo. Infatti, IDEA possiede un reparto costantemente impegnato in attività progettuali in ambito nazionale ed europeo, con l'obiettivo di monitorare l'evoluzione del contesto competitivo e le attuali tendenze dell'innovazione e ricercare le best practices da implementare nella progettazione di nuove soluzioni per il mercato.

Capitolo 2

Architettura hardware

In questa sezione saranno elencati i principali componenti hardware che costituiscono il robot, suddividendoli in base alla struttura dello stesso. Infatti, il robot presenta una disposizione degli elementi hardware su più livelli, oltre al sistema di sicurezza, quello di alimentazione e, infine, quello di movimento che è costituito dalle diverse tipologie di ruote.
Innanzitutto, però, è bene svolgere un'analisi cinematica del robot per capire al meglio il funzionamento dello stesso.

2.1 Modello cinematico

In generale, un robot mobile può assumere diverse configurazioni relativamente alla meccanica delle ruote. In particolare, un veicolo può essere:

- a **guida differenziale** se è caratterizzato da due ruote motrici indipendenti montate sullo stesso asse;
- a **guida sterzante** se il moto è dato da un asse motore e la direzione è data da una ruota o un asse sterzante.

In questo caso, l'AMR è messo in movimento grazie ad un sistema a **guida differenziale**: due ruote motorizzate poste lungo lo stesso asse sono in grado di

muoversi a velocità differenti, una indipendente dall'altra. Questa caratteristica fornisce al robot la capacità di sterzare, poiché per permettere un movimento curvilineo al robot durante il suo tragitto è essenziale che la velocità della ruota sinistra sia diversa da quella della ruota destra.
Questa configurazione, come mostrato nell'immagine successiva, conferisce al robot due gradi di libertà: sterzando, grazie alla guida differenziale, il robot può ruotare attorno all'asse z, mentre la traslazione può essere effettuata solo lungo x.

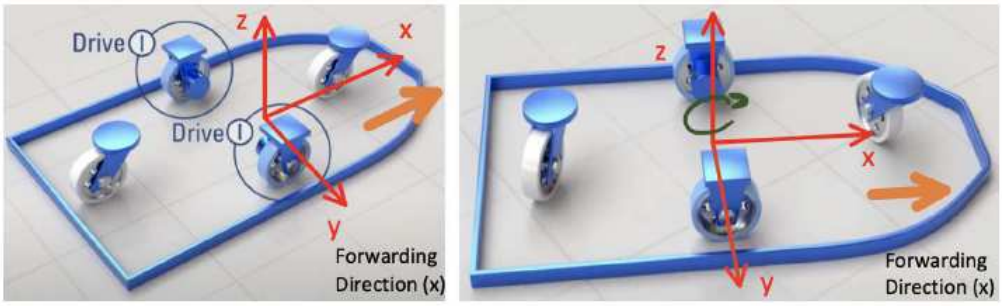

Figura 2.1: Configurazione robot a guida differenziale

L'immagine seguente illustra lo schema tramite il quale è possibile svolgere un'analisi cinematica del robot.

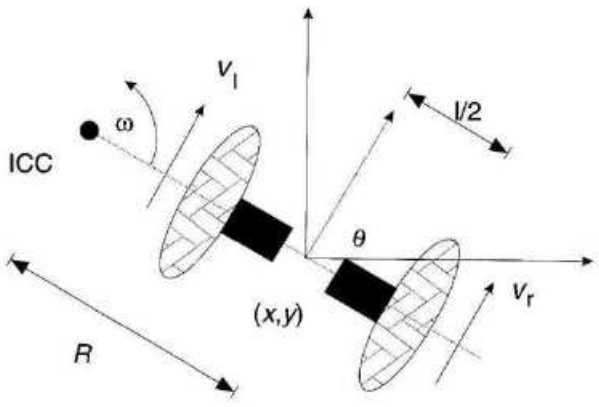

Figura 2.2: Modello cinematico

Ognuna delle due ruote è caratterizzata dalla propria velocità espressa in *metri al secondo* (V_l per la ruota sinistra e V_r per la ruota destra), l è la distanza tra le due

ruote, *ICC (Instantaneous Center of Curvature)* si riferisce al punto attorno al quale ruota il sistema, ω è la velocità di rotazione espressa in *radianti al secondo* mentre R corrisponde alla distanza tra l'ICC e il punto medio (x, y) dell'asse su cui sono poste le ruote.

In particolare, si ha:

$$ICC = [x - Rsin(\theta), y + Rcos(\theta)] \tag{2.1}$$

Dalla relazione che lega la velocità lineare e quella angolare sappiamo che:

$$V_r = \omega(R + \frac{l}{2}) \tag{2.2}$$

$$V_l = \omega(R - \frac{l}{2}) \tag{2.3}$$

Di conseguenza, si ottiene:

$$R = \frac{l}{2}\frac{V_l + V_r}{V_l - V_r} \tag{2.4}$$

$$\omega = \frac{V_r - V_l}{l} \tag{2.5}$$

Una volta ottenute queste relazioni relative alla velocità angolare e alla distanza tra il centro di istantanea rotazione e il punto medio dell'asse, è possibile immaginare tre diversi scenari:

- **Moto rettilineo**: si ha quando le due velocità relative alla ruota destra e a quella sinistra coincidono, ossia quando $V_r = V_r$ e R è infinito. Di conseguenza, il moto risulta lineare e rettilineo poiché la velocità angolare ω è nulla;
- **Rotazione attorno alla ruota destra**: si ottiene quando $V_r = 0$ e $R = \frac{l}{2}$. Analogo è il discorso relativamente alla rotazione attorno alla ruota sinistra, ottenuto per $V_l = 0$;
- **Rotazione attorno al centro del robot**: si ha quando $V_r = -V_l$ e $R = 0$. Di conseguenza, la rotazione si ha attorno al punto medio dell'asse delle ruote.

2.1.1 Cinematica diretta

In generale, la cinematica diretta consiste nel trovare una trasformazione adeguata con lo scopo di ottenere la posizione dell'end effector, data la posizione nello spazio dei giunti.

Nel caso di un robot a guida differenziale, si assume che esso si trovi ad una posizione (x, y) e che sia orientato rispetto all'asse z di un angolo θ. Inoltre, si suppone che il robot sia posizionato perfettamente nel punto medio dell'asse che separa le due ruote.

Conoscendo la posizione del centro di istantanea rotazione ICC e potendo manipolare le velocità V_r e V_l relative ad entrambe le ruote, all'istante $t + \delta t$ la posa del robot sarà:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} cos(\omega\delta t) & -sin(\omega\delta t) & 0 \\ sin(\omega\delta t) & cos(\omega\delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - ICC_x \\ y - ICC_y \\ \theta \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega\delta t \end{bmatrix} \tag{2.6}$$

Da questa formula si evince che la posa all'istante $t + \delta t$ è pari alla **matrice di rotazione**, la quale ha la funzione di esprimere la rotazione del vettore attorno l'asse z, moltiplicata per un vettore 3x1 che indica la posizione del robot rispetto al frame che ha come origine il centro di curvatura istantanea ICC.

La quantità ottenuta da questa moltiplicazione viene poi sommata ad un nuovo vettore, sempre di dimensioni 3x1, che a sua volta è dato dalle coordinate del centro del frame che ha ICC come origine, espresso rispetto al sistema fisso.

2.1.2 Cinematica inversa

La cinematica inversa, a differenza di quella diretta appena descritta, consiste nel determinare tutte le possibili configurazioni assunte dai giunti del robot, data la posa dell'end-effector, ossia il vettore che ne descrive la posizione e l'orientamento.

In generale, è possibile descrivere la posizione di un robot dotato della capacità di muoversi in una particolare direzione $\theta(t)$ ad una determinata velocità $V(t)$.

La posizione è descritta dalle seguenti formule:

$$x(t) = \int_0^t V(t)cos[\theta(t)]dt \tag{2.7}$$

$$y(t) = \int_0^t V(t)sin[\theta(t)]dt \tag{2.8}$$

$$\theta(t) = \int_0^t \omega(t)dt \tag{2.9}$$

Come già accennato, in questo caso particolare la configurazione scelta per il robot è a guida differenziale. Ciò significa che le velocità delle ruote possono essere l'una diversa ed indipendente dall'altra. Di conseguenza, è possibile descrivere la posizione di un robot con questa particolare configurazione come segue:

$$x(t) = \frac{1}{2}\int_0^t [v_r(t) + v_l(t)]cos[\theta(t)]dt \tag{2.10}$$

$$y(t) = \frac{1}{2}\int_0^t [v_r(t) + v_l(t)]sin[\theta(t)]dt \tag{2.11}$$

$$\theta(t) = \frac{1}{l}\int_0^t [v_r(t) - v_l(t)]dt \tag{2.12}$$

In questo caso, è importante sottolineare che la configurazione a guida differenziale introduce il concetto di costanti **non-olonomiche** per quanto riguarda la determinazione della posizione del robot stesso. Infatti, un robot con questa configurazione non è in grado di muoversi lateralmente lungo il suo asse.
Detto ciò, è possibile analizzare gli scenari che possono accadere al robot utilizzando la guida differenziale.

- Se il robot percorre una traiettoria rettilinea con $v_l = v_r = v$, le equazioni diventano:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + vcos(\theta)\delta t \\ y + vsin(\theta)\delta t \\ \theta \end{bmatrix} \tag{2.13}$$

- Se, invece, il robot effettua una rotazione sul posto tale per cui $v_r = -v_l$ e $R = 0$, le equazioni saranno:

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + 2v\delta t/l \end{bmatrix} \tag{2.14}$$

2.2 Movimento

Come già spiegato in precedenza, L'AMR viene messo in movimento tramite due ruote motorizzate poste lungo lo stesso asse che sono in grado di muoversi a velocità differenti, fornendo al robot la capacità di sterzare. In particolare, le ruote incorporano al loro interno motori **Brushless DC (BLDC)**.
Un'ulteriore caratteristica di queste ruote consiste nella presenza di **sensori ad effetto Hall**, i quali riescono a rilevare i campi elettromagnetici generati con lo scopo di ottenere informazioni sulla posizione del rotore da inviare ai driver.
La scelta di utilizzare questa soluzione nasce da una convenienza dal punto di vista economico, visto che risulta più vantaggioso rispetto ad un sistema in cui le ruote e i motori sono separati.
Le caratteristiche tecniche principali sono:

- Tensione: 36 V
- Potenza: 250 W
- Spessore: 0.05 m
- Diametro della ruota: 0.25 m

Figura 2.3: Ruota motorizzata

Oltre alle due ruote motorizzate, sono state utilizzate anche delle **ruote a sfera** poste agli angoli del robot in modo tale da permettere una maggiore stabilità. Infatti, grazie ad esse, è possibile appoggiare oggetti pesanti in prossimità di uno degli angoli del robot senza provocarne il ribaltamento.

Figura 2.4: Ruota a sfera

Per quanto riguarda questa tipologia di ruote, è stato necessario inserire un sistema di ammortizzazione, come mostrato nell'immagine seguente:

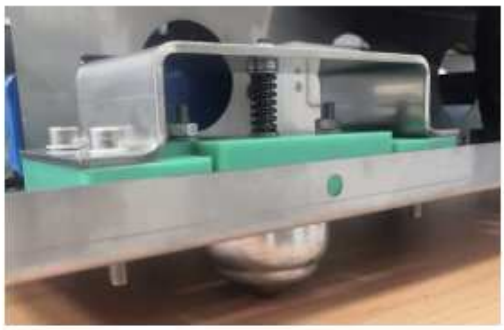

Figura 2.5: Sistema di ammortizzazione

2.3 Primo livello

Il **primo livello**, ossia quello più in basso, contiene un **mini PC Asus PB50** che si occupa dell'esecuzione dei nodi ROS al fine di comunicare con tutti gli altri componenti. In particolare, il PC gioca un ruolo fondamentale dal punto di vista della gestione del movimento, inviando e ricevendo dati agli altri componenti che si occupano del controllo.

Figura 2.6: Mini PC Asus PB50

2.4 Secondo livello

Nel **secondo livello**, invece, si trova la maggior parte della componentistica hardware. I componenti più rilevanti sono:

- Tre **batterie al piombo** collegate in serie, realizzate in materiale ABS in modo tale da aumentare notevolmente la resistenza del contenitore batteria. La singola unità ha una capacità di 13Ah, una tensione nominale di 12V con un peso di 4.1 kg.

Figura 2.7: Batteria al piombo

- I **driver** dei motori, che hanno il compito di pilotare i motori Brushless DC posti all'interno delle ruote, comunicando loro le azioni da svolgere rispettando sempre i vincoli operativi. In questo specifico caso, sono state scelte due Brushless Motor Driver Board 6V-60V 400W ZS-X11D1. La funzione dei driver consiste nell'inviare segnali **PWM** (Pulse Width Modulated) alle ruote motorizzate per pilotare la loro rotazione.

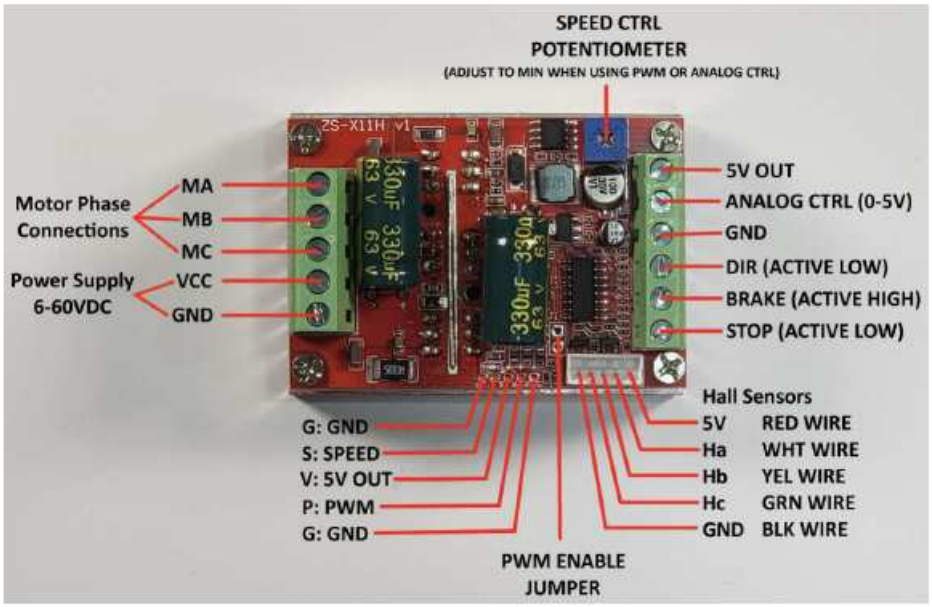

Figura 2.8: Driver dei motori BLDC

- Un **Arduino Mega 2560 REV3 board** tramite il quale vengono controllati i driver precedentemente illustrati. Si tratta di una scheda programmabile con microcontrollore che comprende un IDE facile da utilizzare, grazie al quale si possono scrivere e caricare codici. Inoltre, risulta molto conveniente dal punto di vista economico. Nel progetto illustrato in questa tesi, l'Arduino è collegato direttamente al mini PC e ai driver, giocando in questo modo un ruolo fondamentale dal punto di vista del controllo del movimento del robot.

Figura 2.9: Arduino Mega 2560 REV3 board

- Due **convertitori DC-DC**, i quali hanno come scopo quello di convertire la

tensione nominale fornita dalla batteria in tensioni desiderate. In particolare, nel caso dell'AMR di questo progetto, la tensione di alimentazione è pari a 36 V, la quale viene partizionata dai convertitori in due canali, uno da 19 V e un altro da 24 V.

36 V Battery	19 V channel	Alimenta il mini PC
	24 V channel	Alimenta i relè nel sistema di sicurezza

Figura 2.10: Trasformatore DC-DC WINGONEER XL4016E1 (19 V channel)

Figura 2.11: Trasformatore DC-DC AZ-Delivery LM2596S LM2596 (24 V channel

2.5 Terzo livello

L'ultimo livello, ossia quello più in alto, è caratterizzato soltanto da un componente: un sensore **RPLIDAR A1** in grado di mappare l'ambiente circostante tramite impulsi laser, grazie ai quali viene misurata la distanza tra il robot e gli

ostacoli che ha intorno. La scelta di questo sensore è nata dai costi vantaggiosi e dalla facilità con cui può essere integrato con l'ambiente ROS. La posizione scelta, inoltre, permette al dispositivo di mappare l'ambiente nel migliore dei modi.

Figura 2.12: RPLIDAR A1

Essendo questo sensore il fulcro del progetto, i principi di funzionamento e l'integrazione con gli altri componenti saranno illustrati più avanti.

2.6 Sistema di sicurezza

Oltre ai componenti appena illustrati, è anche presente un **sistema di emergenza** tramite il quale è possibile togliere alimentazione alle fasi dei motori per ottenere lo spegnimento del robot in caso di pericolo. Questo meccanismo è stato implementato inserendo un **bottone di emergenza** mostrato nell'immagine seguente:

Figura 2.13: Bottone di emergenza

Inoltre, sono stati integrati nel sistema due **Finder Relays 62.33 24V** in grado di monitorare la tensione delle tre fasi con lo scopo di prevenire che una di esse esca al di fuori dell'intervallo prestabilito. Più precisamente, i relè controllano in ogni istante le tensioni di tutte e tre le fasi del sistema e intervengono togliendo l'alimentazione se una qualsiasi di esse dovesse uscire dall'intervallo. Grazie a questo, si evitano danni ai vari sottocomponenti del robot.

Figura 2.14: Finder Relays 62.33 24V

2.7 Sistema di alimentazione

Il sistema di alimentazione è caratterizzato principalmente da tre batterie che, come spiegato precedentemente, vengono collegate in serie fornendo all'intero sistema una tensione di 36V. Inoltre, è stato creato un sistema caratterizzato da un selettore, il quale permette di effettuare uno switch tra due diverse configurazioni: se il selettore è impostato ad off il collegamento in serie viene disabilitato

per permettere una fase di ricarica delle batterie in maniera separata l'una dall'altra, se invece il selettore è posto ad on il collegamento in serie viene riabilitato permettendo l'alimentazione del sistema e il conseguente funzionamento.
L'immagine seguente illustra l'implementazione pratica, effettuata all'interno di un apposito box per camblaggi:

Figura 2.15: Sistema di alimentazione con selettore

Per garantire una maggiore sicurezza al sistema di alimentazione, all'interno della scatola è stato aggiunto un fusibile (e il suo corrispondente portafusibile) con lo scopo di interrompere la corrente elettrica qualora venisse superato il limite prestabilito. Questo permette di evitare guasti che causano corti o flussi non consentiti di corrente, i quali possono causare danni a persone o oggetti.

Figura 2.16: Fusibile a cartuccia 10A 250V ca

Figura 2.17: Portafusibile da pannello 10A 250V ca

2.8 Struttura complessiva del robot

Nel complesso, il robot presenta la seguente struttura:

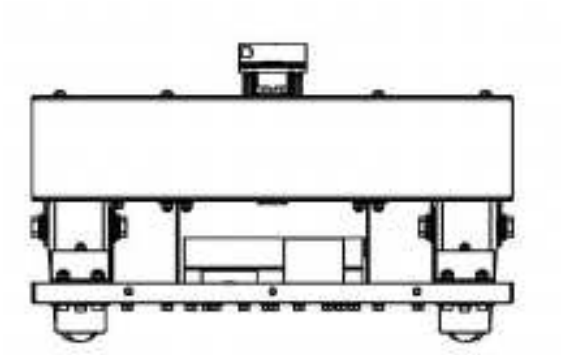

Figura 2.18: Vista frontale

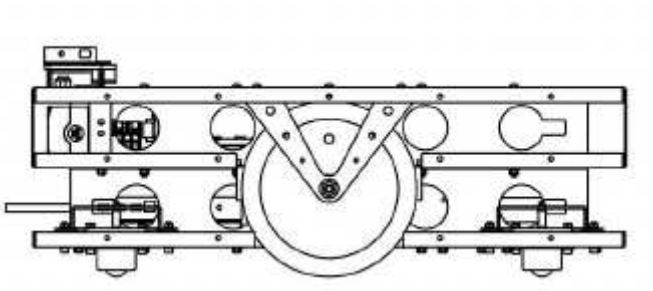

Figura 2.19: Vista laterale

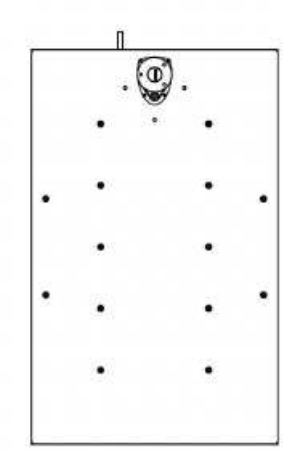

Figura 2.20: Vista dall'alto

2.9 Schema elettrico

Di seguito è riportato lo schema elettrico:

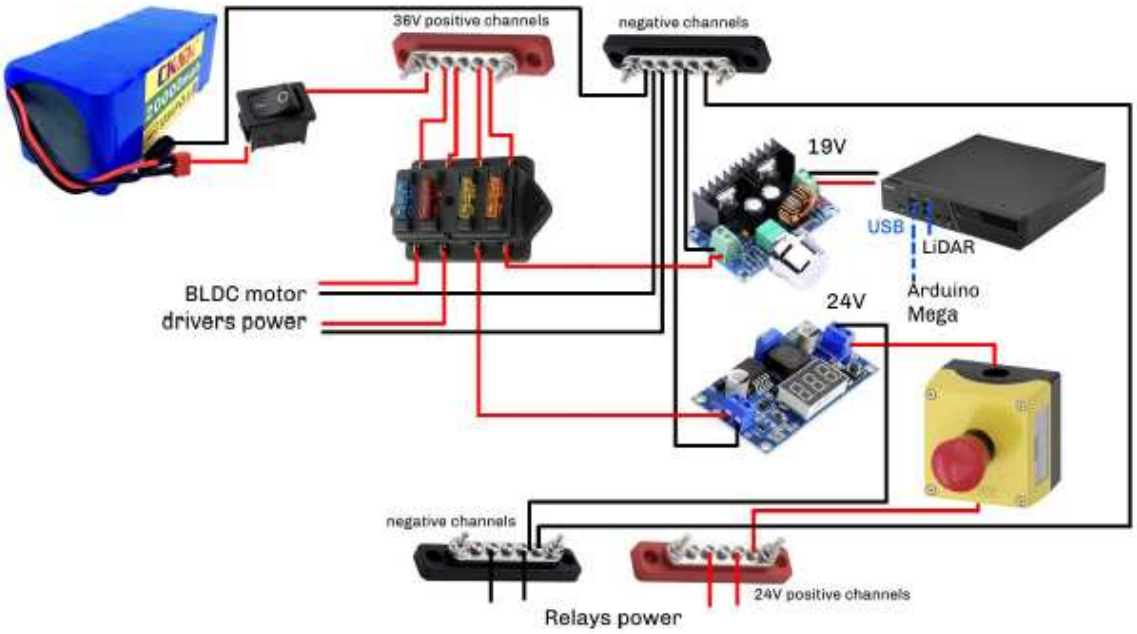

Figura 2.21: Collegamenti tra il sistema di alimentazione e i componenti principali

Figura 2.22: Collegamenti tra le ruote e i driver

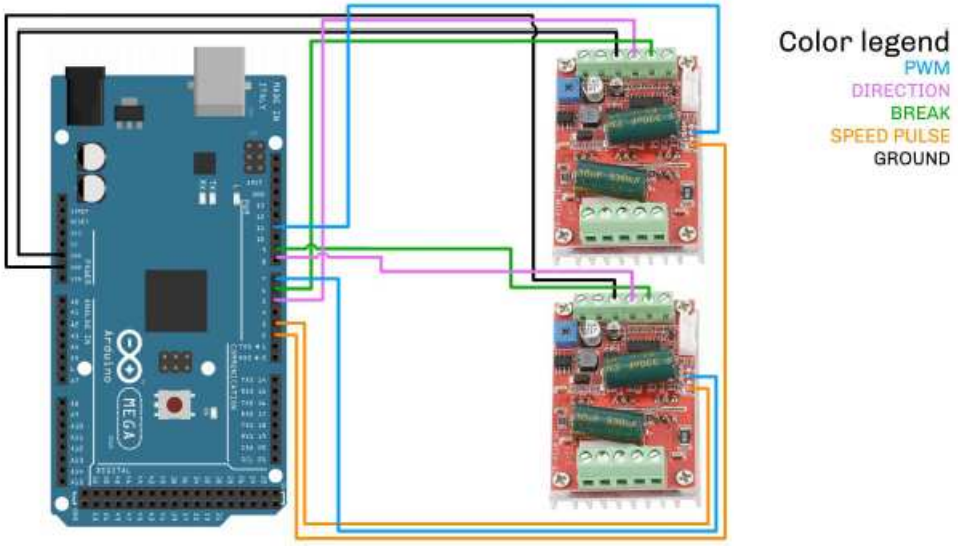

Figura 2.23: Collegamenti tra Arduino e i driver

Capitolo 3

Robot Operating System (ROS)

Figura 3.1: Logo ROS

Il **Robot Operating System (ROS)** è un framework open-source nato nel 2007 per la progettazione e lo sviluppo di applicazioni robot. Infatti, nonostante il nome, non è un vero e proprio sistema operativo, bensì è un **Software Development Kit (SDK)**, ossia un insieme di librerie e strumenti che permettono di generare e sviluppare movimenti e comportamenti complessi per sistemi robotici. In particolare, ROS nasce per essere utilizzato con sistemi operativi Linux, come ad esempio Ubuntu e Debian. Inoltre, i linguaggi di programmazione utilizzati per la scrittura dei codici sono in prevalenza C++ e Python.
A seconda dei pacchetti e dei servizi messi a disposizione, ROS fornisce diverse distribuzioni che vengono aggiornate annualmente, permettendo una continua evoluzione e un continuo sviluppo delle funzionalità.

Per questo progetto è stata scelta la distribuzione **ROS Noetic**, compatibile con **Ubuntu 20.04**.

3.1 Architettura dell'ambiente ROS

Alla base di un'applicazione ROS c'è il concetto di **nodo**, ossia un'unità di elaborazione che ha come scopo quello comunicare con il resto dei nodi grazie agli strumenti messi a disposizione dal framework. Ciò che si ottiene, quindi, è una rete di nodi interconnessi tra loro che accedono a servizi e scambiano messaggi.
Quando un nodo viene avviato, esso si metterà in comunicazione con il nodo principale di tutta l'architettura ROS, ossia il **Master**. Il nodo Master ha come compito quello di tenere traccia di tutti i nodi e di ognuna delle loro funzioni. Ogni nodo, infatti, può essere di due tipi:

- **publisher**: nodi che generano le informazioni;
- **subscriber**: nodi che le ricevono.

Ciò che mette in relazione i nodi publisher e i nodi subscriber è il concetto di **topic**, ossia l'entità tramite la quale vengono scambiati i messaggi.
In generale, i nodi publisher pubblicano i dati sotto forma di messaggi all'interno di un determinato topic al quale i nodi subscriber devono sottoscriversi per poter ottenere le informazioni di cui necessitano.
In particolare, ad ogni topic è associato un determinato tipo di messaggio; di conseguenza, ogni nodo che ha intenzione di fungere da publisher o da subscriber per uno specifico topic deve poter gestire il tipo di messaggio utilizzato per la trasmissione delle informazioni.
ROS mette a disposizione la possibilità di visualizzare a schermo il contenuto del topic desiderato. Il comando da eseguire da terminale che permette questa funzionalità è il seguente:

```
$ rostopic echo <topic>
```

Un altro elemento importante per quanto riguarda l'architettura ROS è il concetto di **service**. I servizi sono funzionalità che un nodo può offrire o di cui può usufruire. Proprio per garantire questa proprietà dei nodi, ROS utilizza il paradigma **client-server** grazie al quale un nodo, se necessario, può accedere ad un servizio messo a disposizione da un altro nodo per ottenere un'informazione specifica.

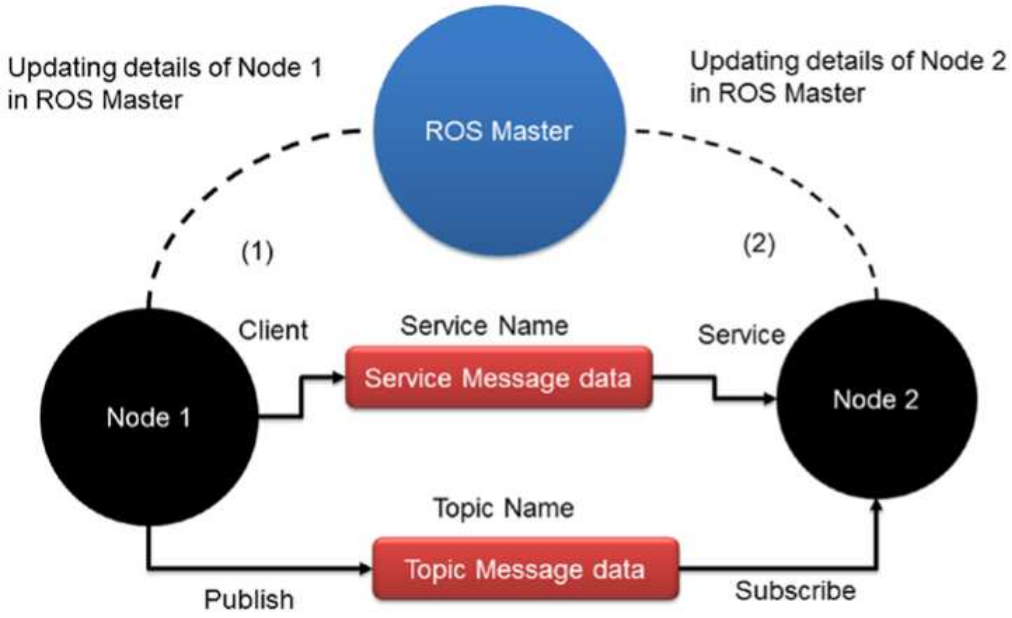

Figura 3.2: Funzionamento architettura ROS

Uno dei vantaggi principali di ROS consiste nella sua modularità, ossia la proprietà per cui l'intera struttura è suddivisa in unità distinte (moduli) che comunicano tra di loro svolgendo funzionalità ben precise. Questo consente di implementare e sviluppare separatamente i vari moduli, permettendo la riutilizzabilità degli stessi in molteplici contesti.

Alla base della modularità in ROS c'è il concetto di **package**. Un package è l'unità organizzativa principale di un'applicazione ROS e consiste in un insieme di librerie, file eseguibili, scripts, e così via.

Un package deve contenere un **manifest**, ossia un file XML chiamato *package.xml* in cui vengono elencate diverse proprietà come il nome del package, la versione, gli autori e la lista delle dipendenze.

Per creare un package in ROS è necessario, innanzitutto, creare un **catkin workspace** e, successivamente, eseguire da terminale il seguente comando:

```
$ catkin_create_pkg <pkg_name> [depend1] [depend2]
```

In particolare, un catkin workspace è un ambiente di lavoro in cui può essere contenuto un intero progetto ROS. Nello specifico, **Catkin** consiste in una raccolta di file **CMake** necessari per la compilazione dei package utilizzati. Ciò significa che tramite Catkin è possibile ampliare i vantaggi dell'ambiente ROS, dal momento in cui questo sistema di build personalizzato è in grado di permettere una migliore organizzazione dei progetti, un'elevata portabilità del codice e un maggiore supporto per l'identificazione automatica dei package che costituiscono l'ambiente di lavoro.
Catkin è in grado di generare, durante il processo di build, un numero elevato di **target**, ossia delle entità che possono essere librerie, file eseguibili, header, script, e così via. Catkin può creare i diversi target contemporaneamente, tenendo conto delle dipendenze. Per fare tutto ciò, necessita di diverse informazioni relative alla compilazione, come ad esempio i vari codici sorgente o le dipendenze reciproche. A tal proposito, all'interno di un package è sempre presente un file chiamato *CmakeList.txt* che contiene al suo interno un elenco di informazioni fondamentali come il nome del package, le librerie, una lista dei file necessari alla compilazione, informazioni relative ai messaggi e ai servizi, e così via.

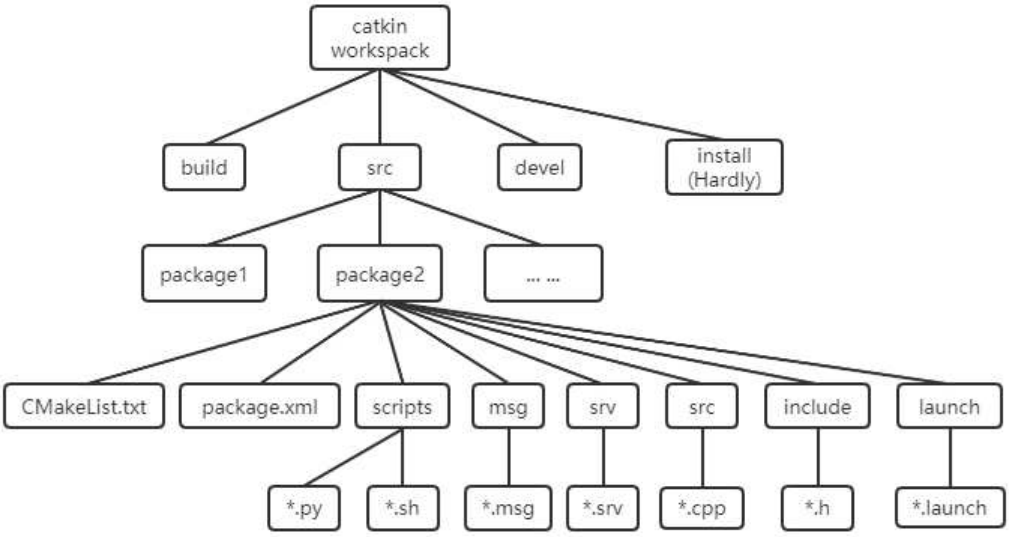

Figura 3.3: Catkin workspace

3.2 RViz

RViz è un tool messo a disposizione da ROS per visualizzare i vari movimenti del robot all'interno di uno spazio 3D. In particolare, è utile poiché è possibile integrare all'interno di questo spazio di visualizzazione tutti i dati provenienti dai sensori, con lo scopo di rappresentarli graficamente.
RViz può essere lanciato da terminale tramite il seguente comando:

```
$ rosrun rviz rviz
```

Una volta avviato RViz da terminale, apparirà la seguente schermata:

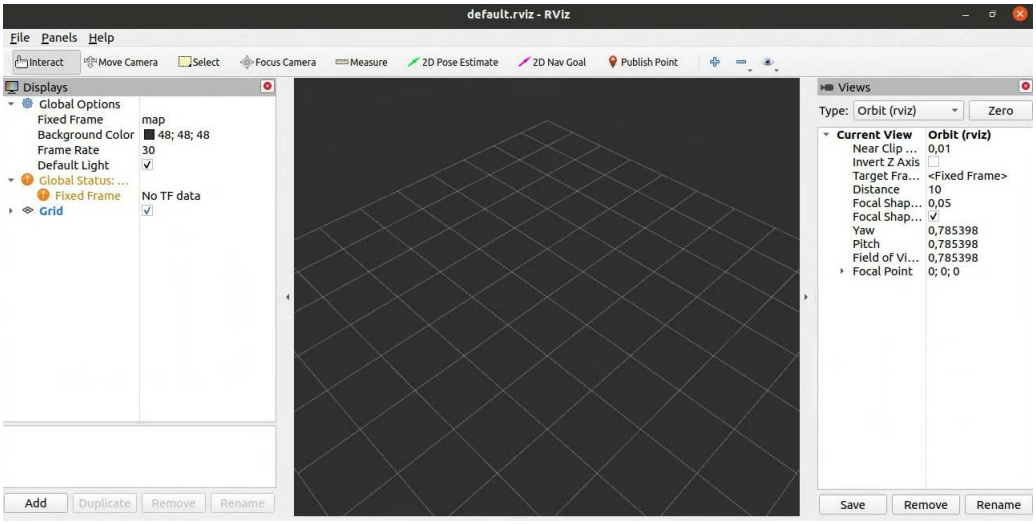

Figura 3.4: Schermata di default RViz

Per utilizzare al meglio RViz, è necessario settare correttamente tutte le impostazioni nel menù visibile a sinistra dell'immagine. In **Global Options** sono comprese le informazioni relative al frame fisso, al colore dello sfondo, e così via. Inoltre, cliccando la voce *Add* in basso a sinistra, si possono aggiungere ulteriori informazioni affinché RViz le possa leggere, permettendone l'integrazione durante la visualizzazione.

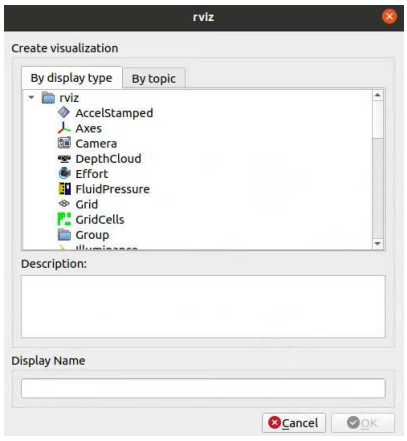

Figura 3.5: Funzionalità di RViz

Come vediamo, è disponibile una lista di elementi che possono essere integrati, come ad esempio le informazioni pubblicate tramite i topic dai nodi che com-

pongono il progetto. In questo modo, RViz può includerle nella visualizzazione. Un esempio di topic che RViz può leggere è l'odometria, particolarmente utile in questo progetto.

3.3 URDF

L'**Unified Robot Description Format (URDF)** è un linguaggio XML che permette di descrivere in maniera molto dettagliata la struttura e la cinematica di un sistema robotico. Viene frequentemente utilizzato nel contesto dell'automazione per generare una rappresentazione standardizzata dei modelli robotici, fondamentale per progetti di simulazione, controllo e programmazione di robot.
Per rappresentare in maniera accurata e dettagliata la struttura di un robot (che sia semplice o complesso) URDF utilizza un'organizzazione gerarchica. In particolare, i principali elementi includono la definizione dei giunti, dei link e delle componenti di sensoristica.
Infatti, L'URDF permette di definire la presenza di sensori come telecamere o, come in questo caso, LIDAR. Anche le componenti di questo tipo possono essere inserite visivamente e collegate agli altri link per mezzo dei giunti.
I principali tag utilizzati in un file URDF sono i seguenti:

- <**robot**>: si tratta del tag radice, il quale contiene l'intera descrizione del robot;
- <**link**>: definisce un componente della struttura del robot, specificandone le caratteristiche fisiche più importanti come le proprietà geometriche;
- <**joint**>: si tratta della connessione tra due link. I giunti possono essere di vario tipo in base alla struttura e alla cinematica del robot: possono essere di tipo prismatico se permettono solo una traslazione, oppure rotazionali se permettono la rotazione di un link rispetto all'altro. In ogni caso, un giunto descrive il movimento relativo che avviene tra due link del robot;
- <**sensor**>: permette di aggiungere alla struttura del robot un sensore. An-

che in questo caso possono essere specificate le caratteristiche fisiche più importanti, oltre che la posizione e l'orientamento;

- <**visual**>: si tratta del tag responsabile della rappresentazione visuale di un componente, definendo la geometria, i materiali e le varie trasformazioni;
- <**collision**>: viene utilizzato per specificare come vengono calcolate le collisioni dal punto di vista geometrico;
- <**origin**>: è il tag utilizzato per definire la posizione e l'orientamento di un componente rispetto al suo sistema di riferimento.

Questi sono alcuni dei tag principali che sono stati utilizzati all'interno del file URDF generato per questo progetto. Tuttavia, esistono numerosi altri tag che possono essere utilizzati e integrati nel codice per permettere una descrizione più completa e dettagliata della struttura del robot.

3.3.1 Libreria Tf

La libreria **tf (transform library)** è uno degli elementi più importanti nella progettazione di un sistema robotico in ambiente ROS. Si tratta di uno strumento fondamentale per la gestione delle trasformazioni geometriche tra i diversi frame di riferimento che caratterizzano il sistema robotico.
Questa libreria mette a disposizione la possibilità di gestire in maniera flessibile ed efficiente la posizione e l'orientamento degli oggetti facenti parte la struttura del robot, consentendo a quest'ultimo di interfacciarsi ed interagire in maniera coerente con l'ambiente circostante.
Per quanto riguarda il concetto di **trasformazione geometrica**, si tratta di un elemento fondamentale poiché descrive come un sistema di coordinate è posizionato e orientato rispetto ad un altro all'interno dell'intera struttura. Si tratta di un aspetto che deve essere gestito in maniera particolarmente precisa, dal momento in cui la libreria tf gestisce le trasformazioni relative anche ai frame dei sensori, riportando le osservazioni al sistema di riferimento del robot. Di conseguenza, i valori ottenuti grazie alle trasformate geometriche devono essere calcolati accura-

tamente, per non rischiare di introdurre degli errori nelle misurazioni dei sensori. Oltre a questi aspetti principali, tf è stata progettata per gestire la variazione nel tempo di tutti i valori e di tutte le trasformazioni. Infatti, un robot che si muove nello spazio sarà soggetto a delle variazioni nella posizione e nell'orientamento di tutti i vari frame. Di conseguenza, tf è in grado di aggiornare costantemente le trasformazioni e garantire una rappresentazione sempre coerente. Inoltre, quando i dati vengono ricevuti a frequenze diverse o con dei ritardi temporali, la libreria è in grado di interpolare le trasformazioni esistenti fornendo una rappresentazione *smooth* della posa dei frame.
La libreria tf in ROS pubblica le trasformazioni all'interno del sistema, consentendo a tutti i nodi di essere sempre a conoscenza di tutte le trasformazioni in corso. Queste possono essere lette sfruttando le funzionalità di tf. In particolare, per stampare la trasformazione tra due frame può essere utilizzato il seguente comando da terminale:

```
$ rosrun tf tf_echo frame_source frame_target
```

Ad esempio, se si volesse verificare qual è la posa del sensore LIDAR, il cui frame è chiamato **laser**, rispetto al frame di riferimento del robot chiamato **base_link**, si può utilizzare questo comando ottenendo il seguente output:

```
idea@idea-E810:~$ rosrun tf tf_echo base_link laser
At time 0.000
- Translation: [0.300, 0.000, 0.110]
- Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
            in RPY (radian) [0.000, -0.000, 0.000]
            in RPY (degree) [0.000, -0.000, 0.000]
```

Figura 3.6: tf echo

Un altro elemento importante per quanto riguarda tf è l'**albero delle trasformazioni**, noto anche come grafo delle trasformazioni o TF tree. Si tratta di una rappresentazione grafica dei collegamenti e delle relazioni di trasformazione tra tutti i frame di riferimento coinvolti all'interno del sistema robotico. Questo grafo risulta particolarmente utile per comprendere come e da chi sono state svolte tutte le trasformazioni, visualizzando graficamente quali sono i frame coinvolti.

3.3.2 Visualizzazione del robot in RViz

Una volta che è stato definito il file URDF e che è stata utilizzata la libreria tf per la definizione delle relazioni tra i frame, è possibile visualizzare il robot su RViz. In particolare, aprendo il software richiamando il file URDF corrispondente si ha il seguente risultato:

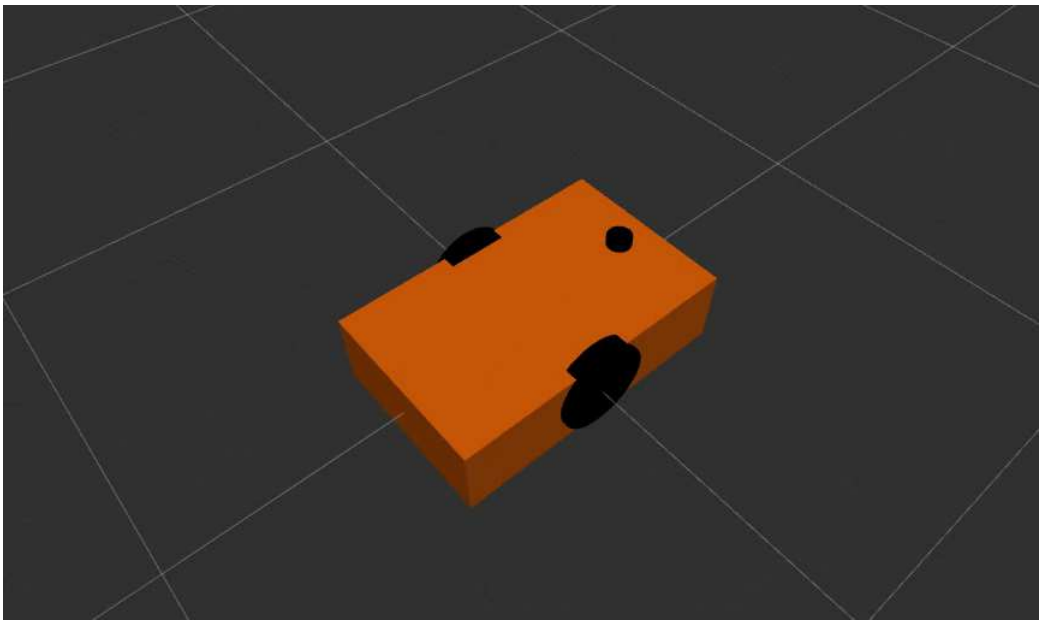

Figura 3.7: Visualizzazione del robot su RViz

Inoltre, RViz mette a disposizione la possibilità di togliere momentaneamente il modello creato tramite URDF e di visualizzare soltanto i frame di riferimento con le rispettive trasformazioni pubblicate da tf. Di conseguenza, spuntando le apposite voci nel menù del software, il risultato è il seguente:

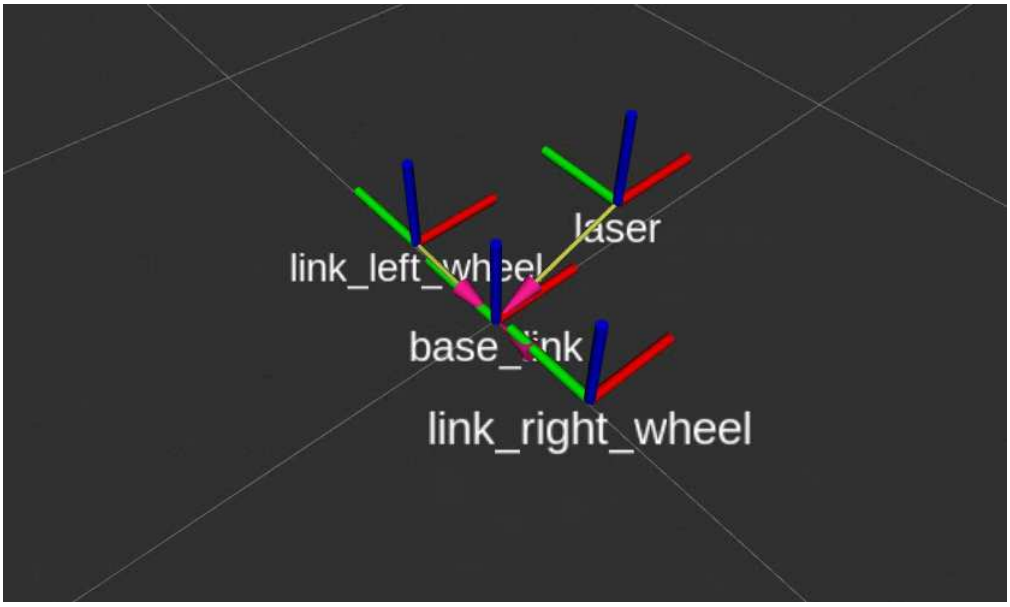

Figura 3.8: Visualizzazione dei frame su RViz

Capitolo 4

Richiami agli sviluppi precedenti

Il lavoro svolto durante il progetto, come già spiegato in precedenza, si basa sulle attività già portate a termine nei mesi precedenti da altri studenti. In questa sezione saranno spiegate brevemente le nozioni fondamentali per comprendere al meglio il punto di partenza del progetto.

4.1 Comunicazione e controllo

Il lavoro svolto si è basato sull'integrazione di tutti i componenti hardware e sull'utilizzo dell'ambiente ROS per permettere il movimento del robot senza l'ausilio dell'intervento umano. In particolare, per comprendere al meglio la struttura del robot e i principi di funzionamento, è necessario analizzare le modalità con cui sono stati implementati gli algoritmi per la comunicazione tra i componenti e il loro controllo.
In generale, il controllo del robot è suddiviso su tre livelli:

- Il **controllo di alto livello** si basa sull'ambiente ROS per la comunicazione, tramite i nodi, con l'Arduino e gli altri hardware che costituiscono i livelli

più bassi;

- Il **controllo di livello intermedio** prevede l'utilizzo di Arduino che, dopo aver scambiato i dati necessari con il controllo di alto livello, comunica con quello più basso per fornire alle ruote motorizzate i comandi adeguati;
- Il **controllo di basso livello** che interagisce direttamente con i motori, in modo tale da alimentare le fasi e gestire il comportamento delle ruote.

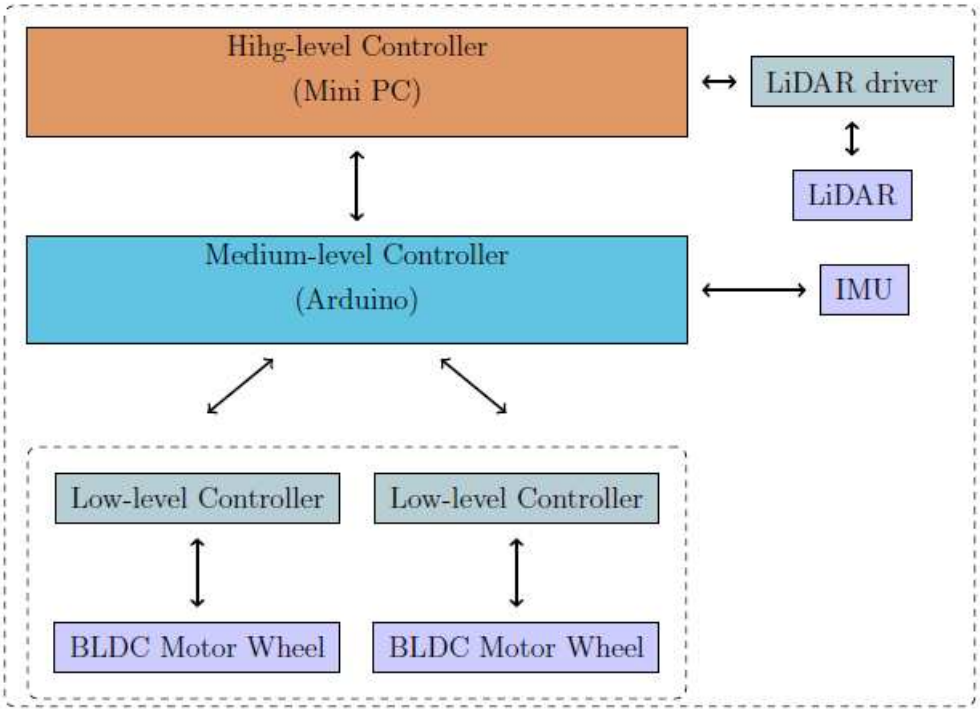

Figura 4.1: Comunicazione tra i livelli di controllo

4.1.1 Controllo di alto livello

Come già spiegato, il ruolo chiave nella comunicazione svolta dal controllo di alto livello con quelli più bassi è giocato dai nodi ROS che vengono eseguiti nel mini PC Asus e in Arduino.

In particolare, il paradigma utilizzato per inviare i comandi desiderati relativi al movimento del robot è di tipo *client-server*.

A tal proposito lo studente precedente ha implementato due nodi fondamentali per il funzionamento di questo paradigma.

- **wheel_Command_Client**: si tratta di un nodo che funge da client poiché attende che vengano forniti i comandi che costituiscono l'input per il robot. Il nodo, una volta avviato da terminale, permette di visualizzare un menù user-friendly in cui sono elencati tutti i comandi ammessi per movimentare il robot.

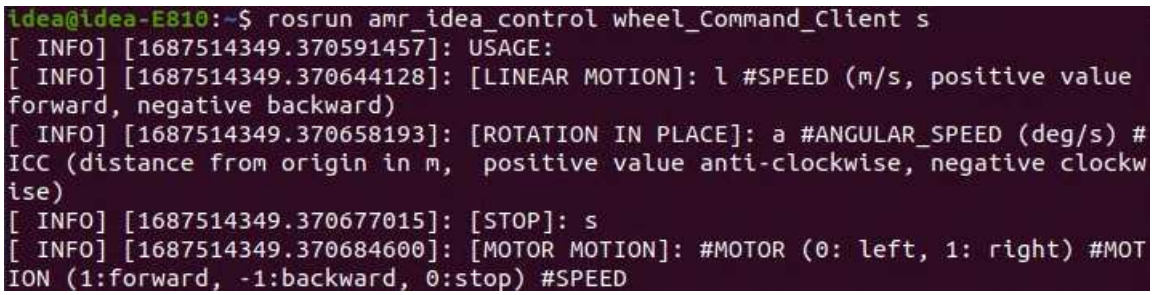

Figura 4.2: wheel_Command_Client

Come è possibile vedere dall'immagine, le opzioni a disposizione dell'utente sono:

- *Linear motion*: il robot viene movimentato in avanti o all'indietro seguendo una traiettoria rettilinea. Il valore numerico inserito ne determina la velocità in metri al secondo.
- *Rotation in place*: inserendo dei valori adeguati relativi al centro istantaneo di rotazione e alla velocità angolare, è possibile permettere una rotazione del robot. In particolare, se il valore inserito per l'ICC risulta positivo, l'AMR svolgerà una rotazione in senso orario e viceversa.
- *Stop*: inserendo questo comando, il robot viene immediatamente stoppato poiché le ruote vengono bloccate.
- *Motor motion*: con questo comando è possibile controllare uno dei due motori, ossia una delle due ruote motorizzate. In particolare, è necessario inserire un valore booleano per identificare la ruota (0 per la ruota sinistra e 1 per la ruota destra) e la velocità di rotazione desiderata.

Nell'esempio illustrato nell'immagine, la lettera *s* inserita nel comando identifica il comando *stop*, grazie al quale il robot rimane fermo.

- **wheel_Command_Service**: si tratta del nodo che ricopre il ruolo di service. In particolare, una volta che il client viene eseguito da terminale correttamente, insieme ai comandi adeguati, esso effettuerà un controllo sui dati per assicurarne la correttezza e la conformità, per poi inviarli al service. Quest'ultimo ha come funzione quella di pubblicare un topic appena riceve i dati dal client chiamato **wheel_Command**, in cui viene registrato un vettore di tipo *std_msgs/Float64.msg*. In questo vettore sono presenti:
  - *motionL*: il senso di rotazione della ruota sinistra;
  - *speedL*: la velocità di rotazione della ruota sinistra espressa in RPM;
  - *motionR*: il senso di rotazione della ruota destra;
  - *speedR*: la velocità di rotazione della ruota destra, anch'essa espressa in RPM;

  Per visualizzare ciò che viene pubblicato all'interno del topic durante il movimento, è possibile eseguire il seguente comando da terminale:

  ```
  $ rostopic echo wheel_Command
  ```

  Nell'immagine seguente è illustrato un esempio, considerando una traiettoria rettilinea in avanti, con velocità pari a 0.5 *m/s*.

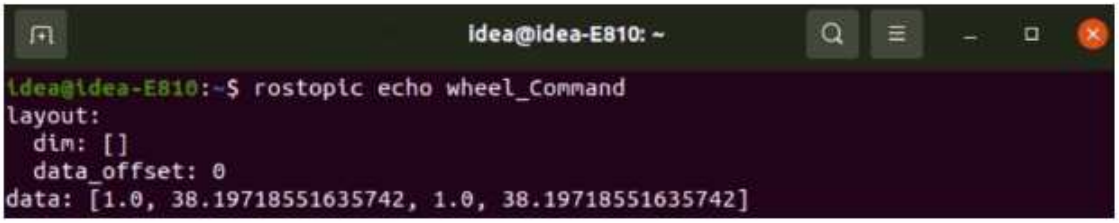

Figura 4.3: wheel_Command

4.1.2 Controllo di livello intermedio

Il controllo intermedio riveste un ruolo fondamentale per il funzionamento dell'AMR. Infatti, è stato implementato per inviare i segnali ai driver delle ruote, ma soprattutto per svolgere il compito di **controllore PI** relativamente alla velocità di rotazione delle ruote motorizzate.
Alla base del controllo intermedio c'è l'Arduino MEGA, il quale comunica direttamente con il mini PC e con i driver dei motori brushless. In particolare, Arduino è stato utilizzato per implementare dei nodi ROS in grado di pubblicare due topic in formato *std_msgs/Float64.msg*. I topic sono:

- *wheel0_Vel*: la velocità effettiva di rotazione della ruota sinistra espressa in RPM;
- *wheel1_Vel*: la velocità effettiva di rotazione della ruota destra, anch'essa espressa in RPM.

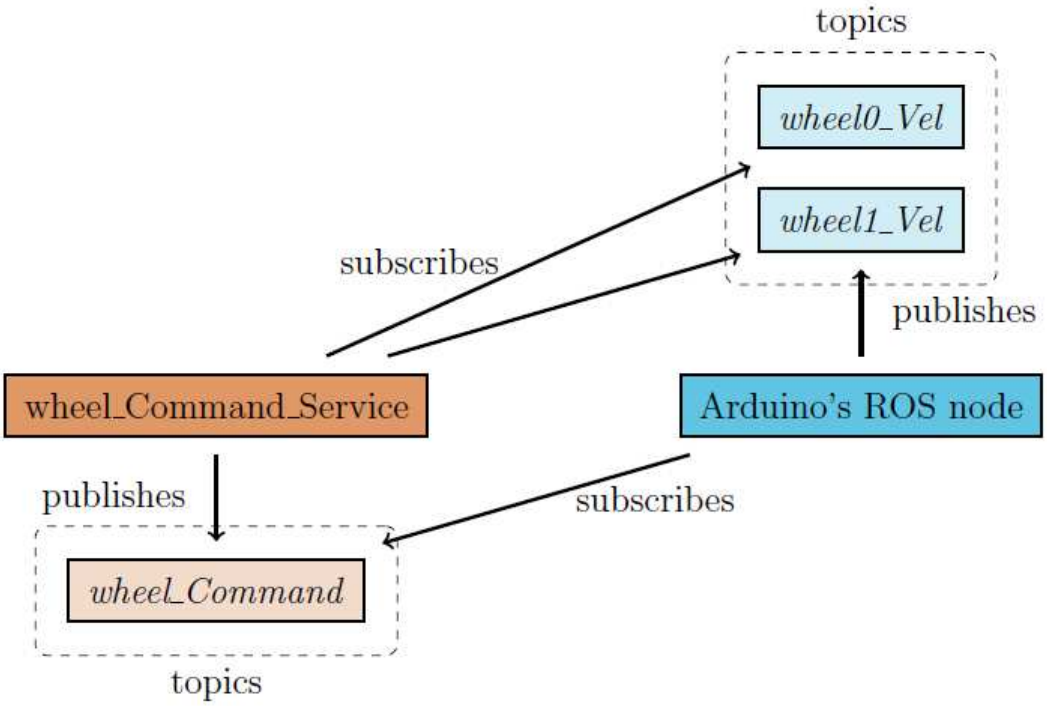

Figura 4.4: Comunicazione tra alto livello e livello intermedio

Per quanto riguarda il controllo implementato in Arduino, il funzionamento pre-

vede un metodo basato sul concetto di **timer interrupt**. In particolare, ad ogni intervallo di tempo specificato il controllore verifica che le velocità delle due ruote siano state rilevate. In caso negativo, ogni 0.2 secondi si genera un'interruzione grazie alla quale viene invocato il metodo **PICallBack**, il quale si occupa del rilevamento delle velocità delle ruote dai driver dei motori.

D'altro canto, nel caso in cui la velocità delle ruote venisse adeguatamente misurata, il controllo di livello intermedio si occupa di esaminarne il valore per assicurarne la correttezza.

In particolare, al valore reale viene sottratto quello di riferimento per ottenere l'errore tra i due segnali. Se l'errore si trova al di sotto di una certa soglia (posta pari ad 1.5 RPM), la velocità di rotazione può essere considerata corretta; se invece il residuo tra la velocità effettiva e quella desiderata risulta superiore rispetto alla soglia, si applicano delle azioni di accelerazione/decelerazione in base all'errore rilevato.

Capitolo 5

RPLIDAR A1

5.1 Principi di funzionamento

Come già accennato in precedenza, il sensore RPLIDAR è in grado di mappare l'ambiente circostante tramite impulsi laser, grazie ai quali viene misurata la distanza tra il robot e gli ostacoli che ha intorno.
In particolare, il modello RPLIDAR A1 contiene un motore, oltre al sistema di scansione dell'ambiente a 360°, che gli permette, una volta alimentato, di iniziare a ruotare per poter effettuare la mappatura.

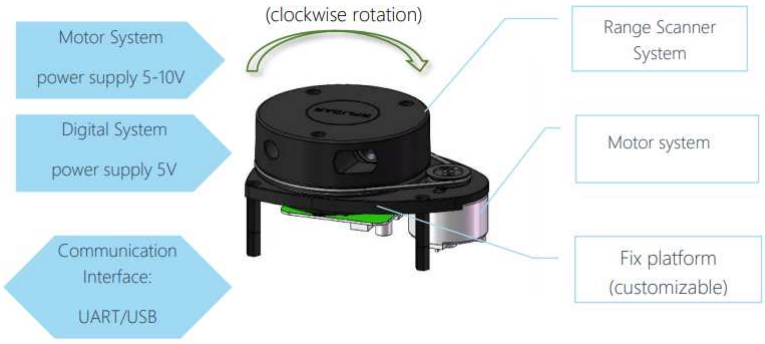

Figura 5.1: Funzionamento RPLIDAR A1

La velocità di rotazione del motore, in questo caso, risulta particolarmente importante dal momento in cui la frequenza di scansione laser dipende da quanto velocemente gira il sensore. Di conseguenza, è necessario un meccanismo di speed detection.
Il meccanismo alla base di un sensore LIDAR è detto **laser triangulation ranging principle**: il sensore emette dei segnali laser infrarossi che vengono successivamente riflessi dall'oggetto da rilevare. Il segnale che ritorna verso il sensore viene campionato da un sistema di vision acquisition per poi, dopo aver elaborato i dati, fornire in output il valore della distanza misurata e il valore dell'angolo tra l'oggetto e il sensore.

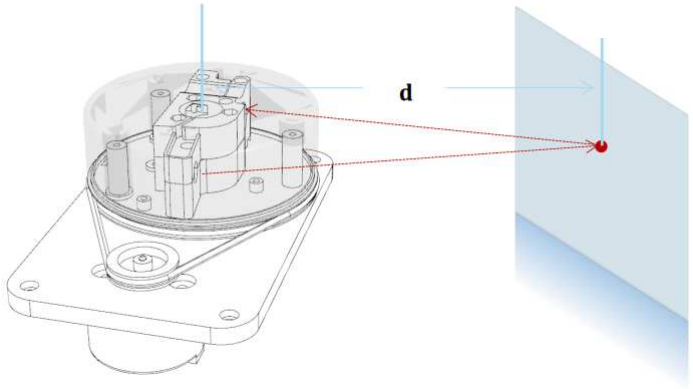

Figura 5.2: Laser triangulation ranging principle RPLIDAR A1

La comunicazione utilizzata per questo sensore è basata sul protocollo UART; tuttavia, in base alle necessità è possibile customizzare questo aspetto.
Nel caso dell'AMR utilizzato in questo progetto, la connessione tra il sensore e il mini PC avviene tramite USB.

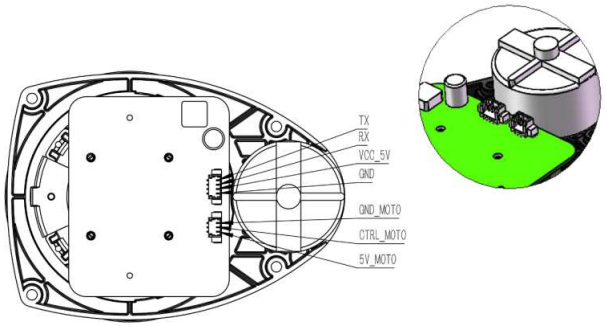

Figura 5.3: Protocollo di comunicazione RPLIDAR A1

Le prestazioni del modello utilizzato per questa applicazione sono schematizzate nella tabella seguente:

Item	Unit	Min	Typical	Max	Comments
Distance Range	Meter(m)	TBD	0.15 - 6	TBD	White objects
Angular Range	Degree	n/a	0-360	n/a	
Distance Resolution	mm	n/a	<0.5 <1% of the distance	n/a	<1.5 meters All distance range*
Angular Resolution	Degree	n/a	≤1	n/a	5.5Hz scan rate
Sample Duration	Millisecond(ms)	n/a	0.5	n/a	
Sample Frequency	Hz	n/a	≥2000	2010	
Scan Rate	Hz	1	5.5	10	Typical value is measured when RPLIDAR A1 takes 360 samples per scan

Figura 5.4: Prestazioni RPLIDAR A1

5.2 Integrazione con l'ambiente ROS

ROS permette una perfetta integrazione dei sensori LIDAR all'interno dei vari progetti. In particolare, il primo step da svolgere consiste nello scaricare i pacchetti necessari, messi a disposizione da Slamtec, ossia l'azienda produttrice dei sensori LIDAR. Tramite l'installazione di questi pacchetti è possibile ottenere i vari file launch che permettono di avviare il sensore e visualizzare le scansioni

registrate dal LIDAR in RViz.

Il package installato, che nel workspace di questo progetto si chiama *rplidar_ros*, contiene un file launch per ogni modello di LIDAR disponibile. Per avviare il file è necessario usare da terminale il seguente comando:

```
$ roslaunch rplidar_ros view_rplidar_a1.launch
```

Nello specifico, il file avviato è il seguente:

```
<?xml version="1.0"?>
<launch>
  <include file="$(find rplidar_ros)/launch/rplidar_a1.launch" />

  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
   rplidar_ros)/rviz/rplidar.rviz" />
</launch>
```

In questo file, oltre ad essere avviato RViz per visualizzare le scansioni del sensore, è richiamato un ulteriore file launch, il quale ha lo scopo di avviare il sensore e di settare tutti i parametri necessari. Il codice è riportato di seguito:

```
<?xml version="1.0"?>
<launch>
  <node name="rplidarNode"          pkg="rplidar_ros"  type="
   rplidarNode" output="screen">
  <param name="serial_port"         type="string" value="/dev/
   ttyUSB0"/>
  <param name="serial_baudrate"     type="int"    value="115200"/
   >
  <param name="frame_id"            type="string" value="laser"/>
  <param name="inverted"            type="bool"   value="false"/>
  <param name="angle_compensate"    type="bool"   value="true"/>
  </node>
</launch>
```

Una volta avviati questi file, RViz si aprirà automaticamente mostrando la seguente schermata:

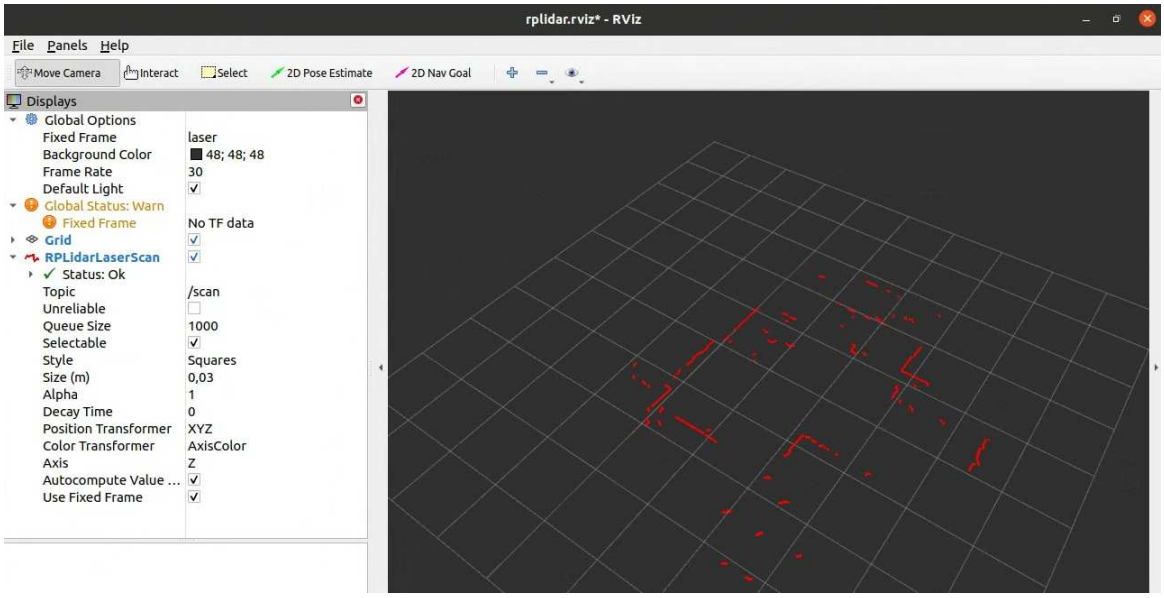

Figura 5.5: Scansioni LIDAR su RViz

Come mostrato nell'immagine, RViz fornisce la possibilità di aggiungere la voce *RPLidarLaserScan* che permette di visualizzare le scansioni nell'ambiente.
In particolare, la mappa visibile è relativa al magazzino dell'azienda.
Il topic pubblicato dal sensore è chiamato *scan* ed è possibile visualizzare i messaggi che vengono pubblicati utilizzando il comando seguente:

```
$ rostopic echo scan
```

L'output generato, come mostrato nell'immagine successiva, è una serie di valori relativi alle scansioni effettuate.

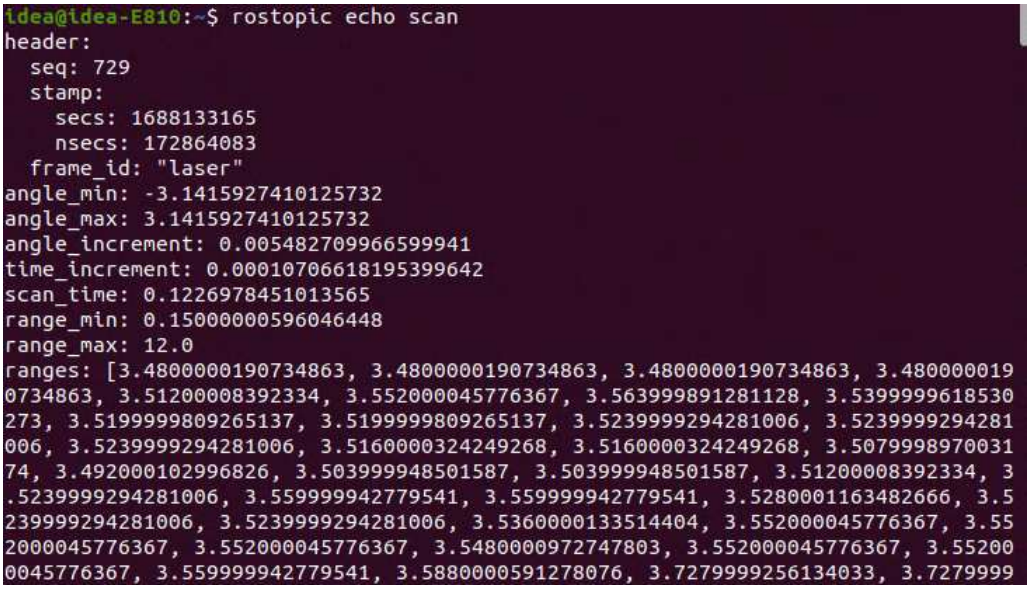

Figura 5.6: Topic scan

Capitolo 6

Mapping

In robotica il problema del **mapping** consiste nella capacità di un robot di acquisire un modello spaziale relativo all'ambiente in cui si muove. Infatti, prima ancora di procedere con l'implementazione di un algoritmo che sia in grado di generare una traiettoria e inviare i comandi di velocità al robot per seguirla, è fondamentale che venga analizzato l'ambiente circostante per generare una mappa in cui l'AMR possa localizzarsi. In questo capitolo, perciò, sarà esaminato singolarmente il problema relativo alla costruzione di mappe, il quale può essere affrontato utilizzando diverse tecniche messe a punto nel corso degli anni.
In particolare, il problema della rappresentazione dell'ambiente e quello della sua localizzazione possono essere considerati uno il duale dell'altro: un'elevata precisione nella costruzione della mappa permette migliori prestazioni nella stima della localizzazione del robot. Inoltre, un ambiente mappato correttamente garantisce una migliore decisione nella pianificazione della traiettoria dal punto di vista dell'algoritmo di navigazione autonoma. Di conseguenza, lo step relativo al problema di map rapresentation consiste in un punto chiave nello sviluppo del progetto.
Per ottenere una rappresentazione precisa dell'ambiente circostante, il robot deve essere provvisto di sensori che gli permettano di ottenere informazioni relative all'ambiente e agli ostacoli presenti. In questo caso, come spiegato precedente-

mente, il sensore utilizzato è il LIDAR. I sensori utilizzati per il problema di mapping sono ovviamente soggetti ad errori più o meno rilevanti, i quali devono essere tenuti conto dall'algoritmo di localizzazione. Infatti, una volta acquisiti i dati per la generazione della mappa, nell'algoritmo di localizzazione si effettua sempre una fase di matching tra le informazioni contenute nella rappresentazione dell'ambiente e quelle relative alle acquisizioni in tempo reale, in modo tale da valutare la precisione della mappa e correggere la stima della posizione del robot all'interno di essa.

Tornando al singolo problema di mapping, negli anni sono state introdotte diverse tipologie di mappe, soprattutto per quanto riguarda la rappresentazione di ambienti indoor come ad esempio magazzini, case ed uffici.

- **Mappe topologiche**: le mappe topologiche vengono utilizzate per rappresentare l'ambiente in modo astratto, ossia come un insieme di percorsi connessi e intersezioni. Infatti, questa tipologia di mappe viene prevalentemente utilizzata per rappresentare ad esempio degli uffici, i quali presentano generalmente una serie di spazi aperti e lunghi corridoi che li separano.

  Proprio per come sono caratterizzate, le mappe topologiche possono essere rappresentate tramite una struttura a grafo. Ad esempio, i vari spazi potrebbero essere considerati come i nodi del grafo, mentre i corridoi potrebbero essere graficati come gli archi che li collegano.

  Un approccio di questo tipo viene utilizzato in genere quando il robot è provvisto di sensori di visione, poiché è fondamentale che esso sia in grado di capire in quale nodo del grafo si trova. Una volta che si hanno informazioni sul nodo, è possibile tenere traccia degli spostamenti analizzando quali archi vengono percorsi dopo che si è lasciato un nodo per raggiungerne un altro.

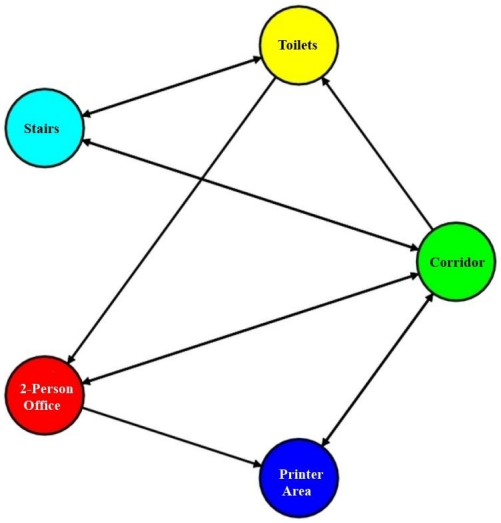

Figura 6.1: Esempio di mappa topologica di un ambiente indoor

- **Mappe metriche basate sull'estrazione di features**: una feature-based map è una tipologia molto utilizzata e consiste nella rappresentazione di una mappa come un set di features prese dall'ambiente. In particolare, una feature è un punto di riferimento estratto dallo spazio che può essere considerato come tale per diverse proprietà. Infatti, una feature può essere catalogata in base a proprietà geometriche (lunghezza, larghezza, ecc.) oppure in base ad altre caratteristiche come colore e posizione. In genere, le feature considerate vengono scelte sulla base dei dati sensoriali disponibili e, di conseguenza, vengono utilizzate per generare la mappa ma anche per fornire informazioni utili all'algoritmo di localizzazione. Nell'immagine seguente, ad esempio, è riportato un esempio di una mappa generata tramite l'estrazione di features, che in questo caso consistono in angoli e spigoli presenti all'interno dell'ambiente.

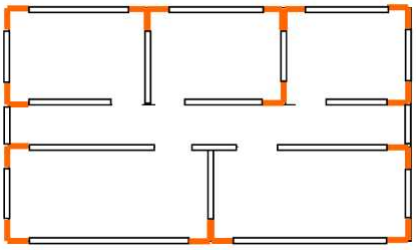

Figura 6.2: Esempio di mappa feature-based di un ambiente indoor

- **Occupancy grid map**: le mappe basate sulle "griglie di occupazione" vengono generate discretizzando lo spazio circostante come se fosse un insieme di celle indipendenti tra loro. Ad ognuna di queste celle è assegnata una variabile che indica se la singola cella è occupata o meno da un ostacolo. La variabile assegnata può essere binaria (0 se non è occupata da ostacoli, 1 altrimenti) oppure può rappresentare una probabilità. Anche in questo caso, le mappe vengono generate a partire dalla lettura dei dati provenienti dai sensori, i quali vengono processati da un algoritmo di assegnazione delle probabilità basato sulla stima Bayesiana. Le fasi di lettura, stima e correzione consentono un aggiornamento incrementale della mappa e quindi permettono la realizzazione di task di localizzazione del robot, ma allo stesso tempo necessitano di un elevato sforzo computazionale.

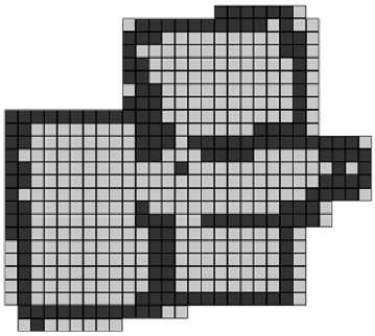

Figura 6.3: Esempio di occupancy grid map di un ambiente indoor

Per quanto riguarda lo sviluppo di progetti nell'ambiente ROS, sicuramente questa ultima tipologia è molto più utilizzata rispetto alle altre.

ROS mette a disposizione una tipologia di messaggio che può essere scambiato tra i nodi proprio per rappresentare una mappa basata su griglia. Il messaggio, che si chiama *nav_msgs/OccupancyGrid* comprende tutte le informazioni necessarie a descrivere la mappa. In particolare, troviamo le seguenti informazioni:

```
# The time at which the map was loaded
time map_load_time
# The map resolution [m/cell]
float32 resolution
# Map width [cells]
uint32 width
# Map height [cells]
uint32 height
# The origin of the map [m, m, rad].  This is the real-world pose of the
# cell (0,0) in the map.
geometry_msgs/Pose origin
```

Figura 6.4: nav_msgs/OccupancyGrid

6.1 Hector_mapping

Per svolgere il task di mapping, per questo progetto è stato utilizzato **Hector_mapping**, un algoritmo messo a disposizione da ROS, il quale genera una mappa considerando le letture provienienti dal sensore LIDAR. Il vantaggio di questo algoritmo è che, a differenza degli altri sistemi di mapping presenti in ROS, non necessita dei dati odometrici del robot, ma genera la mappa soltanto sulla base delle informazioni sensoristiche e della posizione del frame relativo al laser. In particolare, Hector mapping è un modulo facente parte di un pacchetto ROS chiamato **Hector SLAM** che fornisce le funzionalità per effettuare Simultaneous Localization and Mapping. Tuttavia, per lo sviluppo di questo progetto, si è deciso di utilizzare soltanto la parte relativa al mapping, poichè risulta precisa e accurata visto che si basa soltato sulle scansioni laser. Come sarà spiegato successivamente, il task relativo alla localizzazione sarà svolto da altri algoritmi. Hector mapping, per realizzare il task, necessita delle informazioni relative al topic *scan* e alle trasformazioni di coordinate dal frame *laser* al frame *base_link*. A sua volta, l'algoritmo pubblicherà le informazioni sulla mappa generata tramite un messaggio di tipo *nav_msgs/OccupancyGrid*. Nel file di tipo launch messo a

disposizione da ROS una volta installato il pacchetto, è possibile settare tutti i parametri necessari, come ad esempio la risoluzione desiderata della mappa, il nome del frame di riferimento, l'origine della mappa, il numero di celle da utilizzare e così via.
Prima di testare l'algoritmo, è necessario inserire il robot all'interno di un ambiente chiuso e limitato. A tal proposito, sono stati progettati dei pannelli in legno e degli appositi supporti con lo scopo di montare e smontare facilmente un ambiente dotato di pareti.

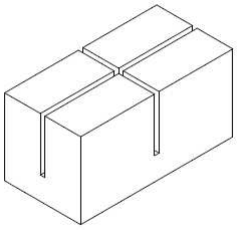

Figura 6.5: Supporto

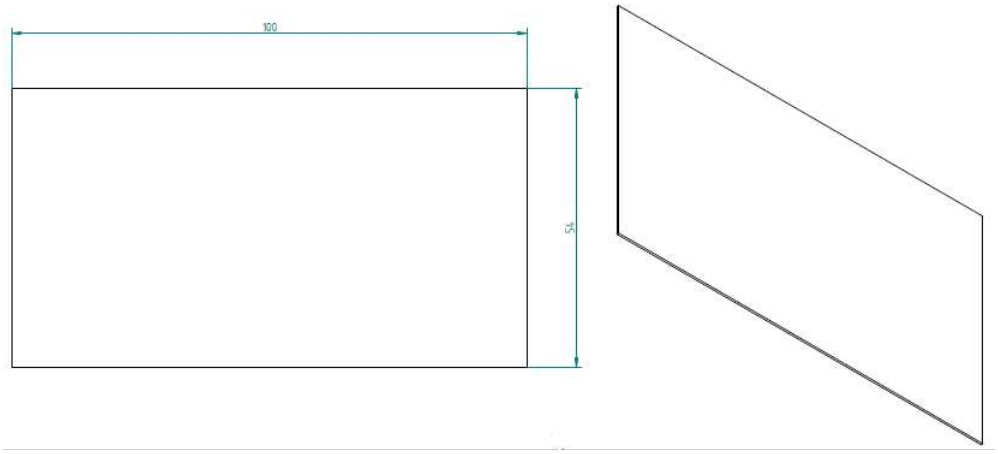

Figura 6.6: Pannello

Per un primo test dell'algoritmo di mapping, l'ambiente allestito è il seguente:

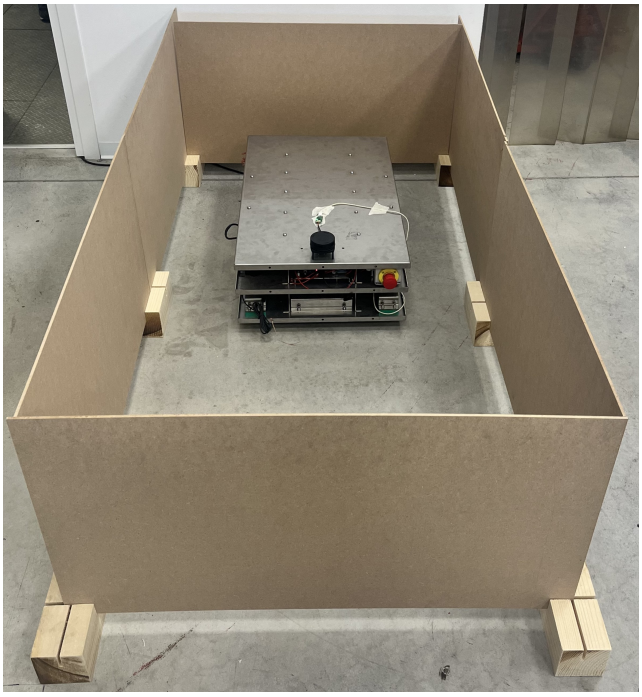

Figura 6.7: Primo allestimento dell'ambiente

Come si può vedere dall'immagine, i pannelli sono stati progettati utilizzando delle misure tali per cui il sensore LIDAR si trova in una posizione ottimale per il corretto rilevamento delle distanze.
Una volta montato l'ambiente, è possibile visualizzare tramite RViz il modello del robot e le relative scansioni laser, in modo tale da verificare il funzionamento del sensore. Il file di launch da utilizzare deve permettere l'avvio del sensore LiDAR, la lettura del modello URDF e l'avvio di RViz. Una volta avviato il tutto, ciò che si visualizza su RViz è riportato nella seguente immagine.

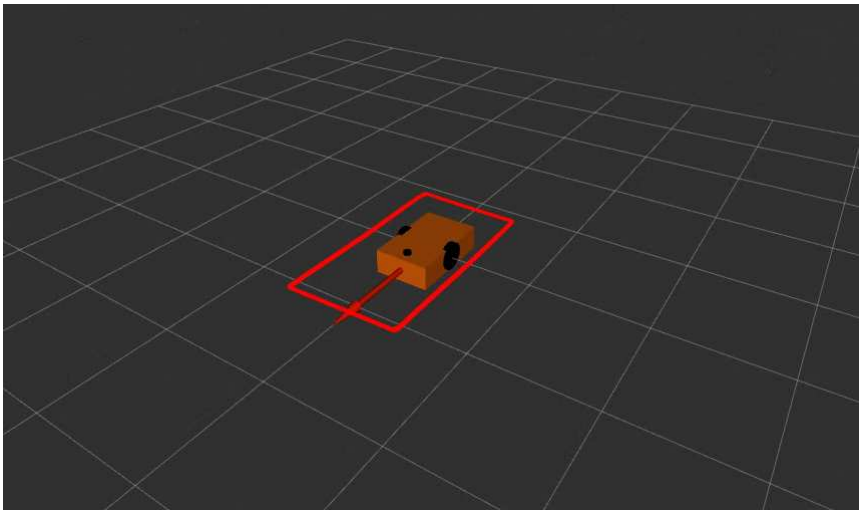

Figura 6.8: Visualizzazione su RViz

Per utilizzare l'algoritmo di mapping, è necessario avviarlo da terminale utilizzando il file launch messo a disposizione dal pacchetto. Il comando da inserire è il seguente:

```
$ roslaunch hector_mapping mapping_default.launch
```

A questo punto, l'algoritmo di mapping sarà in esecuzione e acquisirà le informazioni sensoriali per generare dinamicamente la mappa. Muovendo il LIDAR, le osservazioni si aggiorneranno permettendo il conseguente aggiornamento della mappa. Tuttavia, in questo caso, il robot è collocato in un ambiente tale per cui le pareti si trovano tutte ad una distanza rilevabile dal sensore anche senza doverlo spostare, visto che il modello scelto per il robot può arrivare a misurare ad un range di 6 metri.
L'algoritmo, perciò, finché è in esecuzione si occupa della generazione della mappa, la quale può essere esportata e salvata utilizzando da terminale il seguente comando:

```
$ rosrun map_server map_saver -f mappa
```

Map_server si tratta di un servizio messo a disposizione da ROS che permette di fornire le informazioni relative alla mappa durante lo svolgimento dinamico del

task di mapping da parte di Hector (o di qualsiasi altro algoritmo di mapping presente in ROS). **Map_saver**, a sua volta, è il responsabile della generazione dei file che contengono tutti i dati della mappa. Una volta scelto il nome, in questo caso "mappa", avviando da terminale il comando saranno automaticamente generati due file: *mappa.png* che contiene l'immagine salvata e *mappa.yaml*, il quale contiene informazioni come il path in cui è contenuto il file immagine, l'origine e la risoluzione della mappa.

Una volta effettuata questa procedura, l'algoritmo di mapping può essere terminato e la mappa appena generata può essere visualizzata in RViz insieme al modello del robot spuntando l'apposita voce dal menù. Il risultato ottenuto è il seguente :

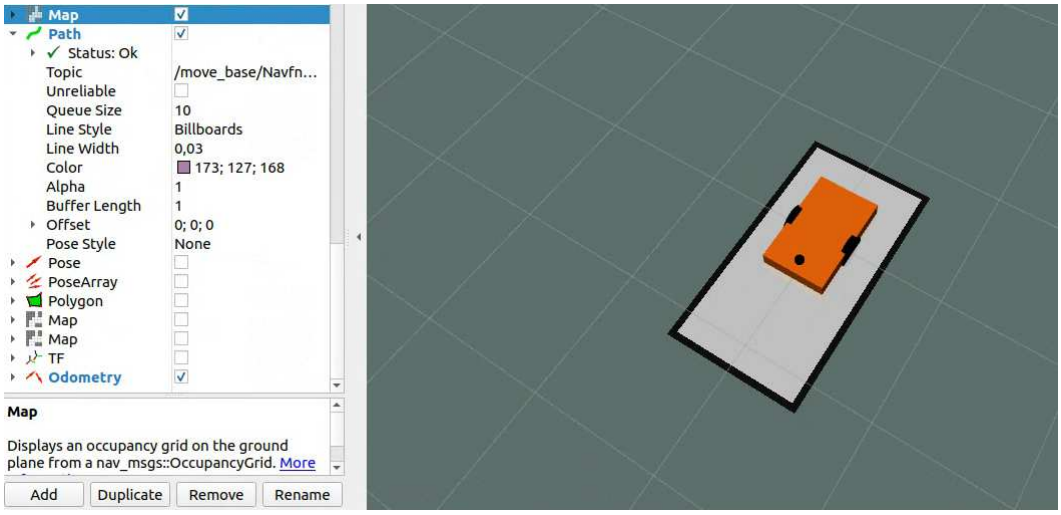

Figura 6.9: Mappa generata

Una volta seguiti tutti questi step, sarà creato un nuovo frame fisso chiamato **map**. Ottenuti questi risultati, la mappa generata può essere utilizzata per l'algoritmo di localizzazione e per quello di navigazione autonoma.

Capitolo 7

Localizzazione

Il concetto di **localizzazione** può essere considerato uno dei più importanti nello sviluppo di un sistema di navigazione autonoma per un AMR. Infatti, un robot in grado di muoversi autonomamente all'interno di un ambiente indoor deve poter generare una traiettoria appropriata sulla base degli ostacoli rilevati e della posizione assunta. Di conseguenza, il robot deve essere in grado di autolocalizzarsi all'interno della mappa per assicurare una corretta generazione del percorso da seguire e una movimentazione sicura ed affidabile, senza il rischio di sbattere contro muri, oggetti e persone. Il processo di localizzazione, perciò, consiste nella determinazione dinamica da parte del robot stesso della propria posizione e del proprio orientamento rispetto ad un frame di riferimento. Per un robot che si muove, la posa si può determinare in relazione ai suoi gradi di libertà. In generale, se lo spostamento può essere effettuato lungo ogni direzione dello spazio e ruotando liberamente rispetto ad ogni asse, i gradi di libertà sono 6 e la posa, data da posizione e orientamento, si esprime come:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \\ roll \\ pitch \\ yaw \end{bmatrix} \tag{7.1}$$

Nel caso di un AMR a guida differenziale, come spiegato nei capitoli precedenti, la direzione lungo cui si muove il robot è l'asse x, mentre quello attorno al quale è ammessa la rotazione è l'asse z. Di conseguenza, i gradi di libertà sono 2 e la posa può essere espressa considerando la sua posizione *(x,y)* all'interno del piano e l'orientamento rispetto all'asse z. Si ha perciò la seguente notazione:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ yaw \end{bmatrix} \tag{7.2}$$

In base al sistema di riferimento scelto per la determinazione di questa terna, si possono distinguere due tipi di localizzazione: quella **assoluta** e quella **relativa**.

- la **localizzazione relativa** o **locale** si ricava sfruttando i sensori propriocettivi del robot, ossia quelli che si occupano di misurare lo stato interno del robot come, ad esempio, la velocità delle ruote, la coppia motrice, l'accelerazione, e così via. Un esempio di sensore propriocettivo è l'encoder.
- la **localizzazione assoluta** o **globale** si ricava dai sensori esterocettivi, i quali consentono di caratterizzare l'interazione del robot con l'ambiente esterno. Di conseguenza, si trovano esternamente rispetto alla struttura del robot e si usano per monitorarne gli spostamenti all'interno della mappa in modo tale da evitare gli ostacoli presenti.

In genere, per ottenere un sistema di localizzazione il più accurato possibile, vengono integrate entrambe le tipologie: inizialmente si considera un approccio relativo, il quale sarà corretto utilizzando una localizzazione di tipo assoluto che prende in considerazione la mappa.

7.1 Odometria

Il primo step da svolgere per implementare un sistema di localizzazione consiste nell'estrapolare i dati odometrici del robot. L'odometria, in particolare, è una metodologia che viene utilizzata per misurare la posa di un robot mobile rispetto ad un sistema di riferimento fisso locale. A tal proposito, vengono utilizzate le informazioni relative alle velocità lineari e angolari delle ruote, le quali vengono integrate nel tempo per ottenere la posizione *(x,y)* nel piano e l'orientamento θ rispetto all'asse z.

Tuttavia, si tratta di un metodo che introduce un errore che si accumula notevolmente nel tempo, poiché si basa su informazioni incrementali che vengono integrate ad ogni istante di campionamento. Infatti, per ottenere valori trascurabili dell'errore, il robot si dovrebbe trovare in situazioni ideali che non accadono nella realtà: se una ruota, ad esempio, dovesse essere soggetta ad uno slittamento, lo spostamento effettivo non può corrispondere alla rotazione registrata. Questi sono errori che si verificano nel caso in cui l'ambiente in cui il robot è vincolato a muoversi sia caratterizzato da problematiche come terreni scivolosi o che non permettono il contatto adeguato della ruota, ossia scenari che possono essere molto frequenti nella realtà.

Inoltre, i dati odometrici vengono calcolati sfruttando le caratteristiche fisiche come il raggio e lo spessore della singola ruota, la distanza tra le due ruote, il disallineamento e così via. Di conseguenza, vengono introdotti ulteriori errori che potrebbero dipendere dalla possibile variazione, seppur minima, di queste grandezze durante la movimentazione.

In ROS è possibile pubblicare le informazioni odometriche utilizzando una particolare tipologia di messaggio chiamata *nav_msg/Odometry* e introducendo un

frame fisso locale chiamato *odom* che viene sottoposto ad una trasformazione di coordinate dalla libreria tf rispetto al frame mobile del robot *base_link*.

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Figura 7.1: Struttura messaggio *nav_msg/Odometry*

In questa tipologia di messaggio, il campo *pose* si riferisce alla stima della posizione e dell'orientamento rispetto all'header frame, in questo caso odom, mentre *twist* è la velocità lineare e angolare del robot rispetto al child frame, in questo caso base_link. In entrambi i casi, ROS mette a disposizione la possibilità di specificare una matrice di covarianza per l'incertezza della stima effettuata.

7.1.1 Codice

Come sarà spiegato più avanti, dal punto di vista della navigazione del robot sono state implementate due possibilità: un controllo tramite un joystick e un sistema di navigazione autonoma. In entrambi i casi, è necessaria l'estrapolazione dei dati odometrici. Tuttavia, il codice implementato per le due possibilità di navigazione è il medesimo, dal momento in cui le modalità di calcolo sono le stesse.
Il file implementato è il seguente:

```
1  #include <ros/ros.h>
2  #include <tf/transform_broadcaster.h>
3  #include <nav_msgs/Odometry.h>
4  #include "std_msgs/Float64MultiArray.h"
5  #include "geometry_msgs/Twist.h"
6  #include "geometry_msgs/Pose.h"
7  #include "geometry_msgs/Quaternion.h"
8  #include "wheel.h"
9
10 float x = 0;
11 float y = 0;
```

```
12 float th = 0;
13 float vx = 0;
14 float vy = 0;
15 float vth = 0;
16
17 void v_callback(const geometry_msgs::Twist &msg){
18
19     ::vx = msg.linear.x; // [m/s]
20     ::vth = msg.angular.z; // [rad/s]
21
22 }
```

In questa parte iniziale viene incluso tutto il necessario, come ad esempio le librerie che permettono di utilizzare determinate tipologie di messaggi in ROS e il file *wheel.h* che contiene tutti i dati fisici delle ruote, come ad esempio il raggio e la distanza tra le ruote. Di seguito sono state dichiarate e definite le grandezze in gioco, in particolare la posizione nel piano, l'orientamento e le velocità lineari ed angolari, in modo tale da settare a zero le posizioni, l'orientamento e le velocità iniziali. La funzione *v_callback* viene definita poiché il messaggio relativo alla velocità lineare lungo x e quella angolare attorno a z sono di volta in volta aggiornate tramite il topic che viene ricevuto dal nodo.

```
1  int main (int argc, char **argv)
2  {
3      ros::init (argc, argv, "odom_nav");
4
5      ros::NodeHandle n;
6      ros::Subscriber sub = n.subscribe("cmd_vel", 1, v_callback);
7      ros::Time last_time = ros::Time::now();
8      ros::Publisher pub = n.advertise<nav_msgs::Odometry>("
       robot_odom", 1);
9
10     tf::TransformBroadcaster odom_broadcaster;
11
12     ros::Rate r(80);
13
```

```
14      while(ros::ok()){
15
16          ros::spinOnce();
17          ros::Time current_time;
18          current_time = ros::Time::now();
19
20          double dt = (current_time - last_time).toSec();
21          double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
22          double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
23          double delta_th = vth * dt;
24
25          x += delta_x;
26
27          y += delta_y;
28
29          th += delta_th;
```

Nel main, successivamente, dopo aver inizializzato il nodo è necessario definire il publisher ed il subscriber. Quest'ultimo si sottoscrive al topic in cui sono contenuti i messaggi di velocità e richiama periodicamente la funzione *v_callback* per aggiornare i valori.

All'interno di un ciclo while vengono poi svolti i calcoli di integrazione: ad ogni istante di campionamento vengono estrapolati l'orientamento e le posizioni a partire dalle velocità correnti e vengono successivamente sommati a quelli calcolati all'istante di campionamento precedente.

```
1           geometry_msgs::Quaternion odom_quat = tf::
    createQuaternionMsgFromYaw(th);
2           geometry_msgs::TransformStamped odom_trans;
3           odom_trans.header.stamp = current_time;
4           odom_trans.header.frame_id = "odom";
5           odom_trans.child_frame_id = "base_link";
6
7           odom_trans.transform.translation.x = x;
8           odom_trans.transform.translation.y = y;
9           odom_trans.transform.translation.z = 0.0;
```

```
10
11          odom_trans.transform.rotation = odom_quat;
12
13          odom_broadcaster.sendTransform(odom_trans);
14
15          nav_msgs::Odometry odom;
16          odom.header.stamp = current_time;
17          odom.header.frame_id = "odom";
18
19          odom.pose.pose.position.x = x;
20          odom.pose.pose.position.y = y;
21          odom.pose.pose.position.z = 0.0;
22          odom.pose.pose.orientation = odom_quat;
23
24          odom.pose.covariance = {0.01, 0.0, 0.0, 0.0, 0.0, 0.0,
25                                  0.0, 0.01, 0.0, 0.0, 0.0, 0.0,
26                                  0.0, 0.0, 0.01, 0.0, 0.0, 0.0,
27                                  0.0, 0.0, 0.0, 0.01, 0.0, 0.0,
28                                  0.0, 0.0, 0.0, 0.0, 0.01, 0.0,
29                                  0.0, 0.0, 0.0, 0.0, 0.0, 0.01};
30
31          odom.child_frame_id = "base_link";
32          odom.twist.twist.linear.x = vx;
33          odom.twist.twist.linear.y = vy;
34          odom.twist.twist.angular.z = vth;
35
36          odom.twist.covariance = {0.01, 0.0, 0.0, 0.0, 0.0, 0.0,
37                                   0.0, 0.01, 0.0, 0.0, 0.0, 0.0,
38                                   0.0, 0.0, 0.01, 0.0, 0.0, 0.0,
39                                   0.0, 0.0, 0.0, 0.01, 0.0, 0.0,
40                                   0.0, 0.0, 0.0, 0.0, 0.01, 0.0,
41                                   0.0, 0.0, 0.0, 0.0, 0.0, 0.01};
42
43          pub.publish(odom);
44          last_time = current_time;
45          r.sleep();
46      }
```

```
47 }
```

Nell'ultima parte del codice, si procede con la trasformazione di coordinate tra il frame *odom* e il frame *base_link* che sarà poi inviata tramite la funzione *sendTransform* in modo da aggiornare tutti i collegamenti e le trasformazioni tra i frame. Infine, viene creato il messaggio con le informazioni odometriche. Per fare questo, si entra all'interno di ogni campo che caratterizza la struttura del messaggio e si assegna con l'informazione corrispondente. Come si può notare, la posa si riferisce al frame *odom*, mentre le velocità angolari e lineari sono calcolate rispetto a *base_link*. Ponendo il publisher all'interno del while, esso sarà in grado di pubblicare alla frequenza scelta i messaggi odometrici.
Per visualizzare ciò che viene stampato dal topic, si deve digitare il seguente comando dopo aver avviato l'intero progetto:

```
$ rostopic echo robot_odom
```

Ciò che verrà visualizzato è il seguente messaggio, considerando che il robot in una condizione di quiete con posizione *(0,0)* nel piano e con orientamento nullo rispetto all'asse z:

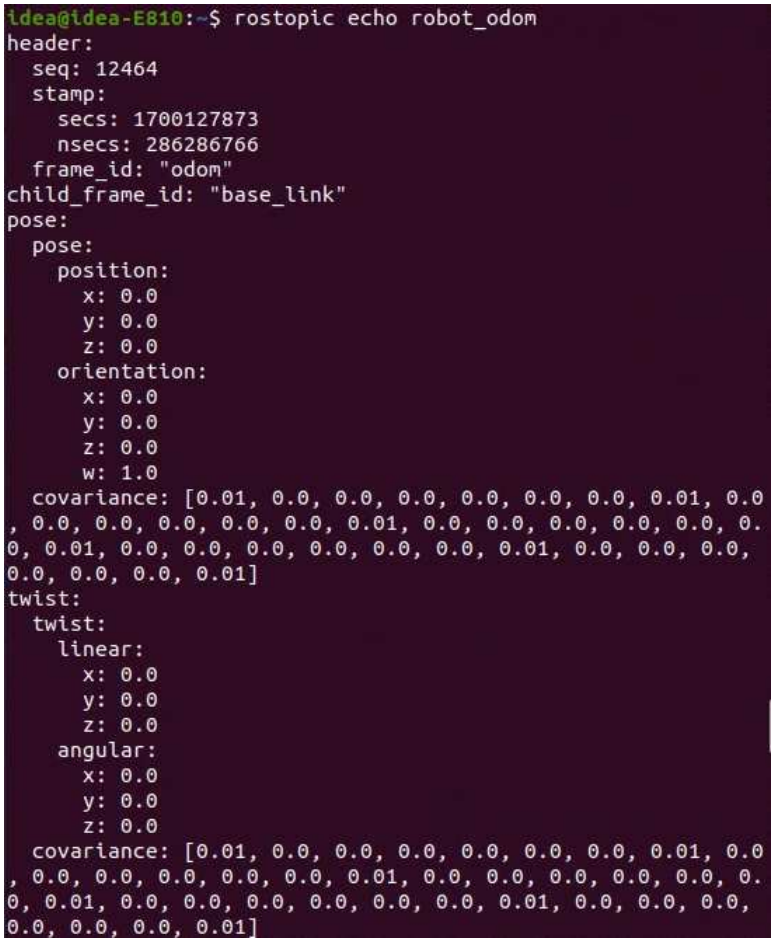

Figura 7.2: Topic robot_odom

L'odometria è utile anche dal punto di vista del tool di visualizzazione RViz, infatti dall'apposito menù è possibile aggiungere il topic *robot_odom*, in modo tale da permettere al software di leggere le posizioni dal topic e graficarle. Infatti, provando ad innescare ad esempio un movimento del robot in avanti, il risultato su RViz sarà il seguente:

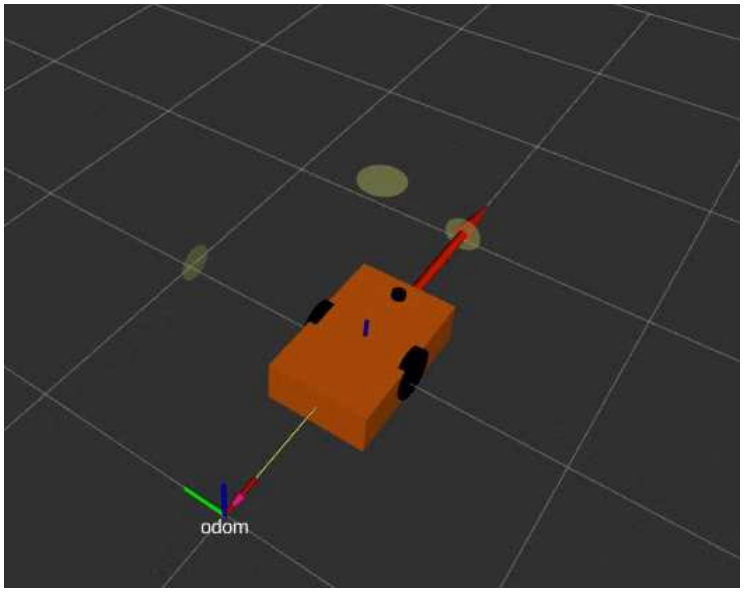

Figura 7.3: Spostamento in avanti

Come si può notare, il robot ha effettuato un movimento in avanti rispetto al frame odom, che rimane fisso e che può essere visualizzato in Rviz. Il frame base_link, invece, si trova al centro del robot ed è solidale agli spostamenti dell'AMR. Il messaggio che comparirà nel topic dopo aver testato un movimento di questo tipo è il seguente:

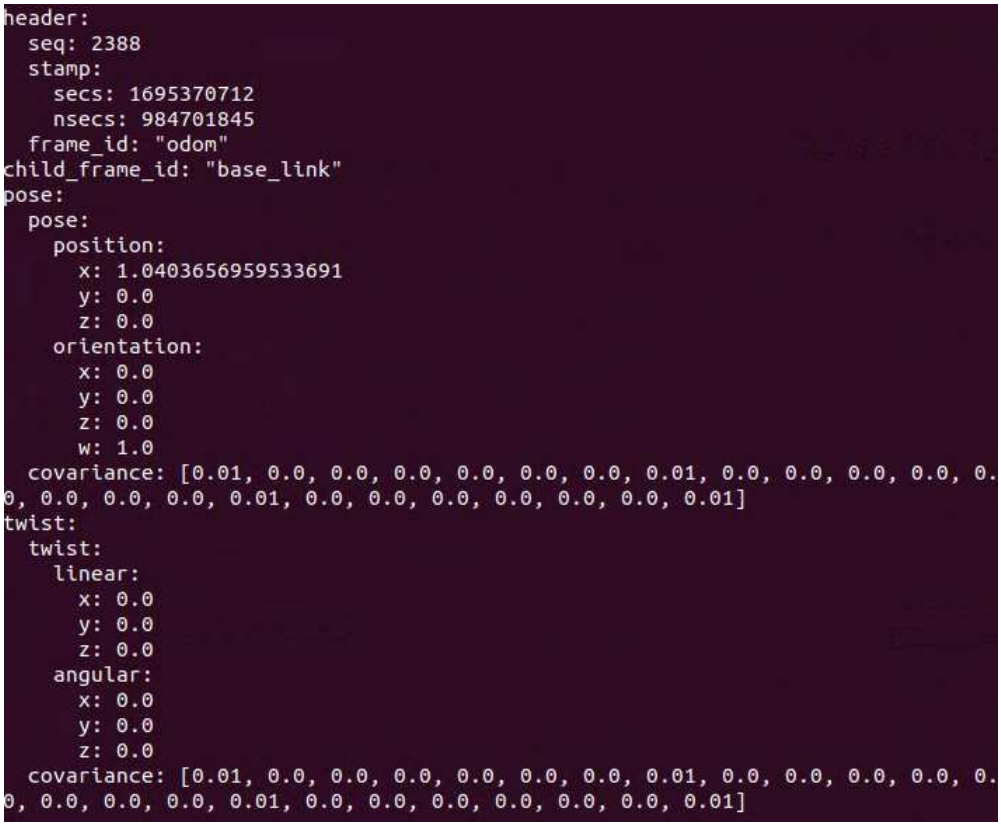

Figura 7.4: Topic robot_odom con spostamento in avanti

La velocità è nulla poichè il robot è stato fermato, ma la posizione indica qual è stata la traslazione del frame base_link rispetto ad odom. In questo caso, è stato effettuato un moto rettilineo perciò il valore che si modifica nel topic è relativo soltanto alla direzione lungo x.

Una volta introdotta l'odometria dal nodo *robot_odom*, è possibile sfruttare un comando messo a disposizione dalla libreria tf per visualizzare l'albero delle trasformazioni, in cui viene anche indicato il nodo responsabile della trasformazione. Il comando da utilizzare da terminale è il seguente:

```
$ rosrun tf2_tools view_frames.py
```

Questo comando avvia un programma che si mette in ascolto dei dati provenienti dalla libreria tf, per poi pubblicare un file pdf. Il risultato è il seguente:

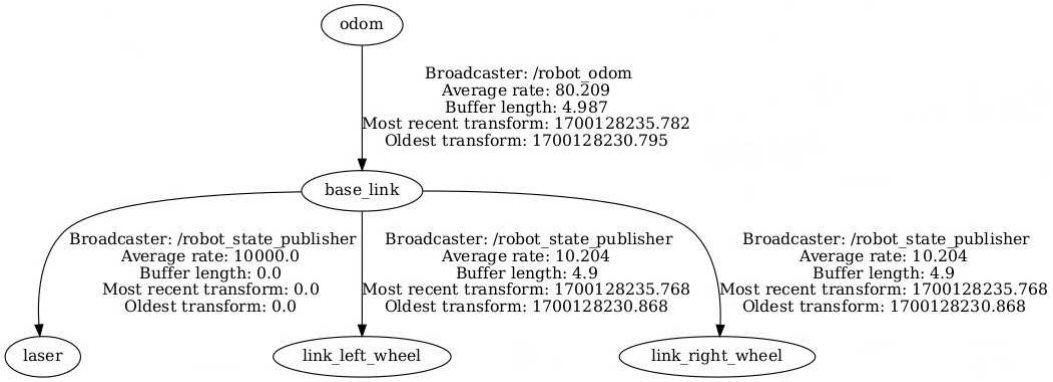

Figura 7.5: tf tree con il frame odom

7.2 Filtro di Bayes per la stima della posizione

Come spiegato in precedenza, i robot mobili sono in grado di tenere traccia della propria posizione a livello locale utilizzando i dati odometrici, calcolati sulla base delle informazioni provenienti dai sensori propriocettivi. Lo step successivo consiste nel correggere gli errori che si accumulano nel tempo dovuti alle limitazioni del calcolo dell'odometria utilizzando le osservazioni relative all'ambiente in cui si trova il robot. In questo modo, fondendo i dati odometrici con quelli relativi alla mappa, il robot può svolgere il task di localizzazione con una maggiore precisione. In genere, il processo di localizzazione di tipo globale consiste di due step principali:

- **prediction update**: in questa fase vengono predetti i valori relativi alla posizione locale utilizzando il modello cinematico;
- **measurement update**: il robot utilizza i sensori esterocettivi per correggere ciò che viene calcolato nello step precedente.

I metodi principali che hanno come scopo quello di svolgere questa tipologia di task sono metodi probabilistici, che consistono in implementazioni basate sul **filtro di Bayes**.

Si consideri un sistema dinamico a tempo discreto:

$$\begin{cases} x_{k+1} = f_k(x_k, w_k) \\ y_k = h_k(x_k, v_k) \end{cases} \tag{7.3}$$

dove

- x_k: stato del robot;
- y_k: uscita misurabile del sistema;
- w_k: disturbo di processo, ossia la modellazione di tutti i fattori che interferiscono con il sistema;
- v_k: disturbo di misura che modella le incertezze introdotte dai sensori.

L'ipotesi principale di questa formulazione assume che entrambi i disturbi siano rumore bianco: w_k e v_k sono perciò incorrelati tra loro e con lo stato iniziale x_0. Il problema della stima si pone come scopo quello di determinare per ogni istante il valore stimato $\hat{x}_k$ dello stato reale del sistema x_k.
In una prima fase di correzione si considerano la stima $\hat{x}_{k|k-1}$, ossia la stima dello stato all'istante k note $k-1$ osservazioni, e $y_k = h_k(x_k, v_k)$, ossia la misura all'istante k. Partendo da questi dati, si procede determinando la *stima aggiornata* (o filtrata) $\hat{x}_{k|k}$. Successivamente, si svolge la fase di predizione in cui, una volta ottenuta la stima corretta e aggiornata $\hat{x}_{k|k}$, si calcola la *stima predetta* $\hat{x}_{k+1|k}$ considerando il modello di stato $x_{k+1} = f_k(x_k, w_k)$.

In particolare, secondo l'approccio Bayesiano, la stima viene svolta determinando la densità di probabilità *PDF* della quantità che deve essere stimata, condizionata alle osservazioni. A tal proposito, si definiscono le seguenti funzioni di densità di probabilità:

- la *PDF corretta*: $p_{k|k}(x) = f(x_k|y^k)$;
- la *PDF predetta*: $p_{k+1|k}(x) = f(x_{k+1}|y^k)$.

dove y^k è la notazione utilizzata per esprimere le osservazioni dall'istante iniziale all'istante k.

La stima può essere determinata utilizzando diversi criteri, come ad esempio quello basato sul **minimo errore quadratico medio** che definisce il problema nel seguente modo:

$$\hat{x}_{k|k} = E[x_k|y^k] = \int x p_{k|k}(x) dx \tag{7.4}$$

con la corrispondente varianza espressa come:

$$P_{k|k} = \int (x - \hat{x}_{k|k})(x - \hat{x}_{k|k})^T p_{k|k}(x) dx \tag{7.5}$$

A questo punto, proprio come nel caso generale, il filtro di Bayes consiste di due fasi, partendo dalla PDF iniziale $p_{1|0}(x)$:

1. **Correzione**: considerando l'equazione di misura $y_k = h_k(x_k, v_k)$ e la PDF $p_{k|k-1}(x)$, si determina $p_{k|k}(x)$.
2. **Predizione**: considerando l'equazione di stato $x_{k+1} = f_k(x_k, w_k)$ e la PDF appena determinata $p_{k|k}(x)$, si determina $p_{k+1|k}(x)$.

Il filtro di Bayes nel corso del tempo è stato utilizzato come punto di partenza per l'implementazione di ulteriori algoritmi, i quali sono caratterizzati da diverse assunzioni iniziali al variare delle applicazioni, ma che seguono in ogni caso un approccio di tipo probabilistico.

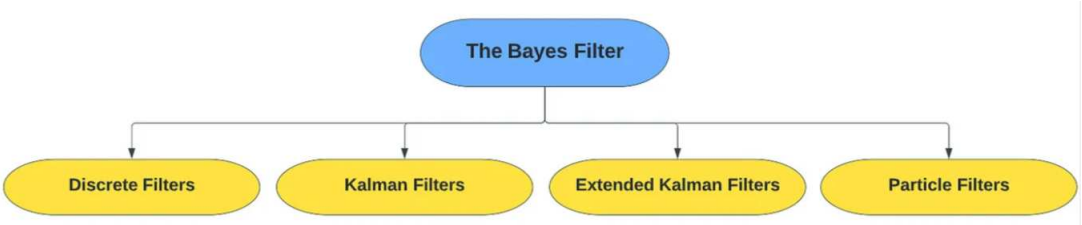

Figura 7.6: Algoritmi basati sul filtro di Bayes

Svolta questa panoramica generale, di seguito saranno spiegati due approcci proposti per il task di localizzazione, entrambi basati sul filtro di Bayes: la localizzazione Monte Carlo basata su un filtro particellare e la localizzazione basata su un filtro di Kalman esteso e sull'estrazione delle features ambientali.

7.3 Adaptive Monte Carlo Localization (AMCL)

La localizzazione Monte Carlo è un algoritmo che si basa su un **filtro particellare**, a sua volta implementato seguendo la logica bayesiana. I filtri di questo tipo vengono utilizzati poiché nel corso degli anni la potenza di calcolo degli elaboratori è aumentata notevolmente, permettendo di utilizzare metodi di campionamento che approssimano la distribuzione continua con una distribuzione discreta costituita da un alto numero di campioni.
Infatti, i filtri particellari consistono nella modellazione di una funzione di densità di probabilità dello stato del robot mediante una procedura di campionamento. Viene determinato un insieme di campioni chiamati **particelle** identificabili dalla coppia di valori (x_i, w_i) che determinano rispettivamente la posizione della particella i-esima e il peso che le è stato assegnato. La serie di particelle considerate definisce il vettore delle variabili di stato, infatti ogni singolo campione rappresenta la stima dello stato futuro del robot.
L'algoritmo utilizza il concetto di probabilità: ad ogni particella viene assegnata una probabilità grazie alla quale è possibile determinare quali tra tutti i campioni sono quelli che stimano nel modo migliore la posizione del robot nello spazio.
Anche in questo caso, l'algoritmo si struttura in diverse fasi, svolte ricorsivamente:

1. **Campionamento**: si genera l'insieme delle particelle tramite il campionamento da una distribuzione proposta;

2. **Assegnazione del peso**: ogni particella è caratterizzata da un peso che viene assegnato in base alla probabilità che ha la particella stessa di rispecchiare lo stato reale del robot. Più alta è la probabilità che la particella sia vicina al valor vero, più alto sarà il peso assegnato al campione.

3. **Ricampionamento**: si ricampionano le particelle considerando però solo quelle a cui è stato assegnato il peso più alto, eliminando di conseguenza

quelle con probabilità minore.

7.3.1 AMCL in ROS

Per poter ottenere una localizzazione basata su un approccio di questo tipo, ROS mette a disposizione un pacchetto chiamato **amcl**.
I topic che forniscono i dati di input all'algoritmo sono:

- *scan*: l'insieme di dati relativi alle scansioni laser effettuate dal sensore RPLIDAR A1;
- *tf*: i dati relativi ai legami esistenti tra i vari frame;
- *initialpose*: la posa iniziale del robot, in modo tale da poter inizializzare il filtro;
- *map*: la mappa precedentemente generata, all'interno della quale il robot deve essere localizzato.

I topic pubblicati, invece, sono relativi alle stime ottenute. In particolare, quello principale è *amcl_pose*, ossia la posa del robot stimata dall'algoritmo.

Per utilizzare l'algoritmo, è necessario integrare all'intero progetto l'apposito file di tipo launch messo a disposizione dal pacchetto. Questo file deve essere opportunamente inizializzato con i parametri necessari. Tra i più rilevanti troviamo il numero massimo e il numero minimo di particelle che si vogliono utilizzare, la posizione iniziale del robot, e così via.
Di seguito è riportato il codice utilizzato, il quale si può trovare all'interno del file *amcl_launch*:

```
<?xml version="1.0"?>
<launch>

  <arg name="scan_topic"     default="scan"/>
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
```

```
8
9   <node pkg="amcl" type="amcl" name="amcl" output = "screen">
10    <param name="use_sim_time"              value="true"/>
11    <param name="min_particles"              value="500"/>
12    <param name="max_particles"              value="5000"/>
13    <param name="kld_err"                    value="0.02"/>
14    <param name="update_min_d"               value="0.3"/>
15    <param name="update_min_a"               value="0.3"/>
16    <param name="resample_interval"          value="1"/>
17    <param name="transform_tolerance"        value="1"/>
18    <param name="recovery_alpha_slow"        value="0.001"/>
19    <param name="recovery_alpha_fast"        value="0.00"/>
20    <param name="initial_pose_x"             value="$(arg
    initial_pose_x)"/>
21    <param name="initial_pose_y"             value="$(arg
    initial_pose_y)"/>
22    <param name="initial_pose_a"             value="$(arg
    initial_pose_a)"/>
23    <param name="gui_publish_rate"           value="30.0"/>
```

In questa prima parte viene definita la posizione iniziale rispetto alla mappa, il nome del topic relativo alle scansioni al quale amcl si deve sottoscrivere per ottenere informazioni in tempo reale e tutti i parametri utili all'algoritmo come il numero minimo e massimo di particelle che si vogliono utilizzare, la frequenza di invio delle informazioni e così via.

```
1     <remap from="scan"                       to="$(arg scan_topic)
    "/>
2     <param name="laser_max_range"            value="12"/>
3     <param name="laser_max_beams"            value="30"/>
4     <param name="laser_z_hit"                value="0.95"/>
5     <param name="laser_z_short"              value="0.05"/>
6     <param name="laser_z_max"                value="0.05"/>
7     <param name="laser_z_rand"               value="0.5"/>
8     <param name="laser_sigma_hit"            value="0.2"/>
9     <param name="laser_lambda_short"         value="0.1"/>
10    <param name="laser_likelihood_max_dist" value="2.0"/>
```

```
    <param name="laser_model_type"          value="
    likelihood_field"/>

    <param name="odom_model_type"           value="diff"/>
    <param name="odom_alpha1"               value="0.5"/>
    <param name="odom_alpha2"               value="0.5"/>
    <param name="odom_alpha3"               value="0.5"/>
    <param name="odom_alpha4"               value="0.5"/>
    <param name="odom_frame_id"             value="odom"/>
    <param name="base_frame_id"             value="base_link"/>

  </node>

</launch>
```

In questa ultima parte vengono definiti tutti i parametri relativi ai frame odom e base_link e al sensore LIDAR come, ad esempio, il range delle scansioni.

Una volta integrato questo codice al file launch principale, è possibile visualizzare la mappa su RViz con il robot al suo interno che è in grado di localizzarsi a livello sia locale che globale. Per visualizzare le posizioni e le velocità stimate dall'algoritmo di localizzazione è possibile stampare il topic *amcl_pose* con il comando seguente:

```
$ rostopic echo amcl_pose
```

L'output ottenuto sarà il seguente:

```
idea@idea-E810:~$ rostopic echo amcl_pose
header:
  seq: 0
  stamp:
    secs: 1700128757
    nsecs: 335905027
  frame_id: "map"
pose:
  pose:
    position:
      x: -0.04550981515900069
      y: -0.027204309082680834
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0011516497363513087
      w: 0.9999993368512226
  covariance: [0.21571473344041753, -0.03349667181151572, 0.0, 0.0, 0.0, 0.0, -0
.03349667181151572, 0.20888912253485056, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0, 0.0, 0.0674863991042706]
```

Figura 7.7: Topic amcl_pose

L'algoritmo AMCL, come già specificato in precedenza, una volta avviato riceve in input i collegamenti tra i frame introdotti dalla libreria tf. Il suo compito, oltre alla stima della posizione e dell'orientamento rispetto alla mappa, è quello di aggiornare i legami tra i sistemi di riferimento inserendo la trasformazione tra il frame *map* e *base_link*.

Ottenuta questa trasformazione, l'albero dei frame sarà il seguente:

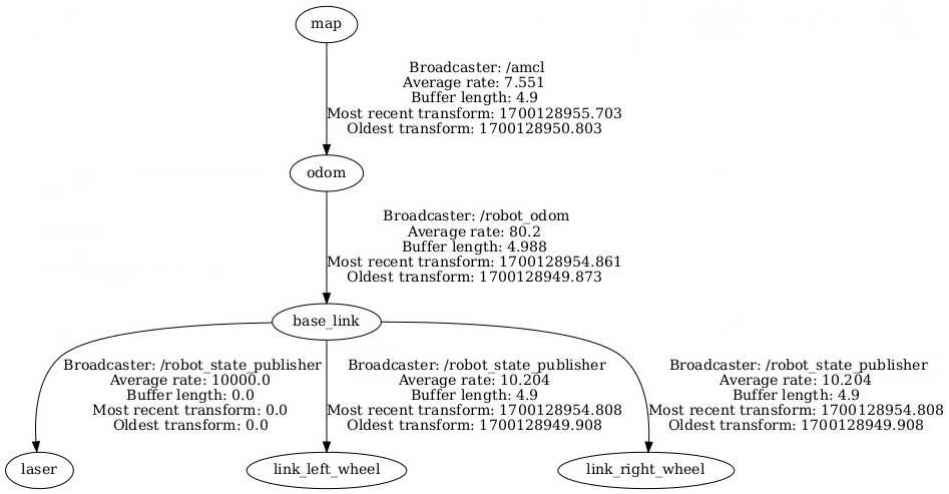

Figura 7.8: tf tree con il frame map

7.4 Filtro di Kalman esteso basato sull'estrazione delle features

Un ulteriore approccio da tenere in considerazione è quello basato su un **filtro di Kalman esteso**, anch'esso implementato sulla base del filtraggio bayesiano. Lo schema di funzionamento principale del filtro di Kalman utilizzato per lo scopo di questo progetto è il seguente:

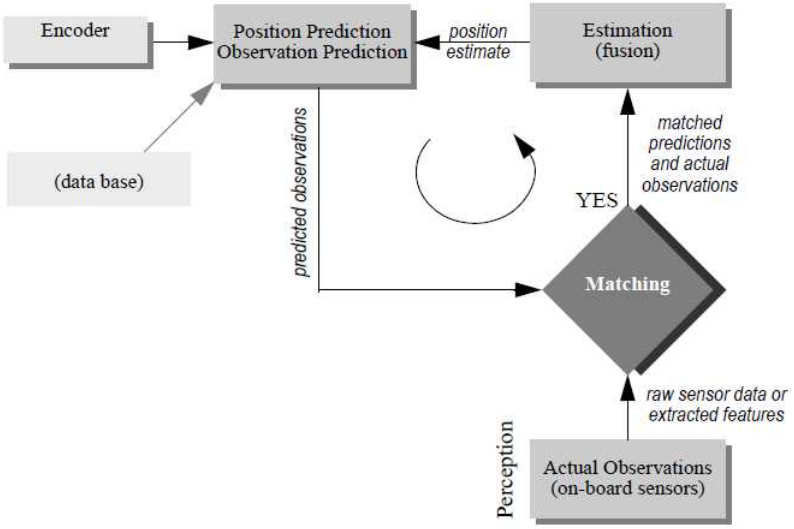

Figura 7.9: Schema EKF

Come mostra la figura precedente, anche la localizzazione basata sul filtro di Kalman consiste nello svolgimento di diversi step.
In particolare, gli step principali sono due: **prediction update** e **measurement update**. Quest'ultimo, a differenza del filtro particellare spiegato nel paragrafo precedente, è costituito da diversi sotto-passaggi.

Prediction update Il calcolo della posizione del robot $\hat{x_t}$ all'istante t si basa sulla localizzazione all'istante precedente $t-1$ e sull'input di controllo u_t. In particolare, si ha:

$$\hat{x_t} = f(x_{t-1}, u_t) \tag{7.6}$$

Per un robot a guida differenziale, è possibile sfruttare i dati odometrici calcolati ad ogni istante. Inoltre, utilizzando questo metodo si può calcolare la matrice di covarianza dell'errore di stima tramite la seguente espressione:

$$\hat{P}_t = F_x \cdot P_{t-1} \cdot F_x^T + F_u \cdot Q_t \cdot F_u^T \tag{7.7}$$

Dove P_{t-1} è la matrice di covarianza dell'errore di stima relativa all'istante precedente e Q_t è la matrice di covarianza dell'errore di misura associata al calcolo proveniente dal modello cinematico.

Measurement update Come accennato, questa fase consiste in diversi passaggi. In particolare, gli step da seguire sono:

1. **Observation step**: questo passaggio ha come scopo quello di ottenere tutte le misurazioni z_t provenienti dal sensore LIDAR all'istante t. In generale, esse consistono in un insieme di n osservazioni $z_t^i, (i = 0...n)$. Ogni singola osservazione può essere rappresentata in diversi modi, come ad esempio un landmark o una linea. In questo passaggio è importante sottolineare che in genere le misurazioni provenienti dal LIDAR vengono calcolate rispetto al frame di riferimento del sensore, di conseguenza è necessario fare attenzione a questo aspetto poiché per ottenere un'implementazione corretta del filtro ogni grandezza deve essere espressa relativamente allo stesso frame di riferimento.

2. **Measurement prediction**: a questo punto vengono usate la mappa M e la posizione precedentemente stimata $\hat{x_t}$ per generare un insieme di features ambientali $\hat{z}_t^j$ predette. Queste osservazioni attese sono quelle che il robot si aspetta di vedere quando assume una particolare posizione.

3. **Matching**: arrivati a questo punto si hanno le informazioni relative sia alle osservazioni reali che alle osservazioni predette. Questo step ha come scopo quello di effettuare un confronto tra queste due tipologie di informazioni per capire quali sono le osservazioni reali che meglio corrispondono a quelle attese, in modo tale da poterle utilizzare in fase di stima della posizione. Di

conseguenza, il risultato ottenuto dal confronto è chiamato **innovazione** ed è dato dalla seguente espressione:

$$v_t^{ij} = [{z_t}^i - \hat{z}_t^j] \tag{7.8}$$

Da questi passaggi è possibile calcolare la matrice di covarianza del processo di innovazione tramite la seguente formula:

$$\Sigma_{IN_t}^{ij} = H^j \cdot \hat{P}_t \cdot {H^j}^T + R_t^i \tag{7.9}$$

Dove H^j è il jacobiano e R_t^i rappresenta la matrice di covarianza dell'errore di misura legata alle osservazioni z_t^i.

4. **Estimation**: in questo ultimo step si calcola la stima della posizione del robot basandosi sulla predizione $\hat{x}_t$ e sulle osservazioni z_t^i all'istante t. Le formule utilizzate per effettuare questo calcolo sono le seguenti:

$$x_t = \hat{x}_t + K_t v_t \tag{7.10}$$

$$P_t = \hat{P}_t - K_t \cdot \Sigma_{IN_t} \cdot K_t^T \tag{7.11}$$

Dove K_t è il guadagno e si calcola come

$$K_t = \hat{P}_t \cdot H_t^T \cdot (\Sigma_{IN_t})^{-1} \tag{7.12}$$

In conclusione, le equazioni appena illustrate indicano che la stima della posizione del robot all'istante t corrisponde al valore della migliore predizione calcolata sulla base dell'odometria a cui viene sommato un termine correttivo dato dal guadagno moltiplicato per la differenza tra le osservazioni reali e quelle attese.

Capitolo 8

Navigazione

L'obiettivo finale di questo lavoro consiste nella navigazione dell'AMR all'interno di un ambiente indoor. Permettere al robot di muoversi significa determinare una traiettoria da seguire e fornire alle ruote dei comandi di velocità per raggiungerla, i quali vengono prima elaborati dall'Arduino. Quest'ultimo, infatti, provvederà a sua volta ad inviare i segnali PWM ai driver.
La navigazione del robot si tratta in generale di un processo che necessita della conoscenza dell'ambiente circostante. Proprio per questo motivo, la pianificazione del percorso e il controllo dei movimenti vedono come elemento principale il LIDAR.
Ai fini del progetto svolto, relativamente alla navigazione del robot sono state implementate due tecniche di natura molto diversa: un controllo tramite **joystick** e un sistema di **navigazione autonoma**.

8.1 Controllo tramite joystick

La navigazione del robot tramite joystick è stata inserita all'interno del progetto poiché risulta fondamentale in fase di mapping. Infatti, quando l'algoritmo di mapping rileva le scansioni laser per generare una mappa, se il robot si muove quest'ultima viene aggiornata dinamicamente. Di conseguenza, potrebbe risultare

particolarmente utile muovere il robot tramite joystick per ispezionare correttamente l'ambiente e permettere all'algoritmo di mapping di rilevare ogni dettaglio per una corretta generazione della mappa.
L'ambiente ROS mette a disposizione diversi pacchetti che permettono la perfetta integrazione di alcuni modelli di joystick all'interno del progetto. In particolare, il joystick utilizzato è un controller per Xbox 360, mostrato nell'immagine seguente:

Figura 8.1: Controller Xbox 360

Per utilizzare il joystick in ROS è necessario svolgere alcuni passaggi:

1. **Installare il pacchetto**: per utilizzare il controller, è necessario installare il pacchetto **teleop_twist_joy** tramite il seguente comando da terminale.

   ```
   $ sudo apt-get install ros-noetic-teleop-twist-joy
   ```

2. **Assicurare il riconoscimento del joystick**: è necessario collegare il joystick al computer e assicurarsi che venga riconosciuto. Ciò può essere fatto utilizzando l'apposito comando da terminale.

   ```
   $ jstest /dev/input/js0
   ```

3. **Configurare il joystick**: il pacchetto utilizzato mette a disposizione diversi file di configurazione, uno per ogni modello di joystick compatibile. Il controller utilizzato è per la console Xbox 360, perciò il file da personalizzare

sarà **xbox.config.yaml**. Al suo interno possono essere specificate le velocità lineari e angolari massime e minime e i tasti del joystick che devono essere premuti per abilitarlo.

4. **Lanciare teleop_twist_joy**: il pacchetto può essere avviato tramite il comando apposito.

```
$ roslaunch teleop_twist_joy teleop.launch
```

A questo punto, muovendo l'analogico del controller, verranno inviati i corrispondenti comandi di velocità. Questi saranno pubblicati all'interno del topic **cmd_vel**. La tipologia di messaggio è *geometry_msgs::Twist* e contiene sia la velocità lineare in m/s che quella angolare in rad/s. Stampando il topic, il risultato sarà il seguente:

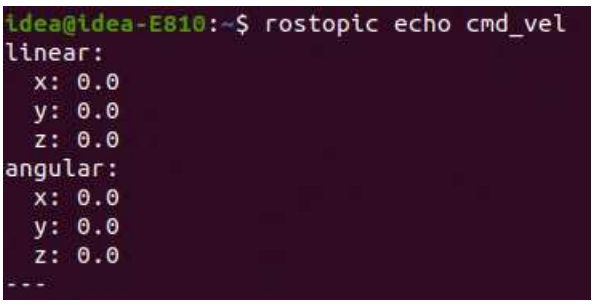

Figura 8.2: cmd_vel

L'implementazione per il controllo tramite joystick risulta utile sia perché, come spiegato precedentemente, aiuta nella fase di mapping per una migliore generazione della mappa, ma anche perchè il formato e le modalità con cui vengono stampati i comandi di velocità sono gli stessi dell'algoritmo di navigazione autonoma. Ciò significa che possono essere trattati allo stesso modo prima di essere inviati a Arduino.
A tal proposito, il nodo che ascolta il topic e si occupa dell'integrazione dei comandi nel robot verrà spiegato più avanti, dopo aver introdotto la navigazione autonoma.

8.2 Navigazione autonoma

Il sistema di navigazione autonoma ha come obiettivo quello di permettere la movimentazione del robot in un ambiente 2D. In particolare, le fasi principali che devono essere incluse sono:

1. **Mapping e Localizzazione**: come già spiegato, queste fasi sono preliminari e particolarmente importanti. Infatti, è necessario che il robot conosca precisamente la propria posizione all'interno della mappa, al fine di ottenere un risultato ottimale.

2. **Scansione dell'ambiente**: questa fase è necessaria anche in tempo reale e non solo durante il task di mapping. Infatti, il robot deve poter percepire l'ambiente dal momento in cui quest'ultimo può risultare dinamico, come ad esempio l'interno di un magazzino in cui possono transitare costantemente persone ed oggetti.

3. **Pianificazione della traiettoria**: la conoscenza della posizione del robot e delle informazioni sensoriali in tempo reale permettono di valutare tutte le possibili traiettorie che possono essere svolte all'interno della mappa per raggiungere un determinato obiettivo. La generazione delle traiettorie deve tener conto di tutti gli ostacoli rilevati o presenti nella mappa pre-generata.

4. **Integrazione dei comandi di velocità al robot**: ogni piattaforma robotica è caratterizzata da diverse tipologie di configurazione, come quella a guida differenziale nel caso dell'AMR di questo progetto. Di conseguenza, i comandi di velocità provenienti dal sistema di navigazione devono essere estrapolati e manipolati per renderli compatibili con la configurazione del robot prima di essere inviati alle ruote.

5. **Obstacle detection**: nonostante l'algoritmo sia a conoscenza della mappa e degli ostacoli presenti all'interno di essa, un ambiente nella realtà può risultare dinamico e in continuo cambiamento. Ciò significa che il robot deve essere in grado di rilevare ostacoli, fermarsi in anticipo e ricalcolare la

traiettoria.

6. **Iterazione**: dal momento in cui la generazione della traiettoria è basata sulla localizzazione, è necessario che il processo di navigazione autonoma svolga delle continue iterazioni per valutare il percorso migliore sulla base della nuova posizione assunta dal robot.

Per svolgere questi passaggi, ROS mette a disposizione il **Navigation Stack**, ossia un insieme di pacchetti e funzionalità che consentono alle diverse configurazioni di robot di svolgere il task di navigazione autonoma all'interno di ambienti complessi.
I componenti principali che costituiscono il Navigation Stack di ROS sono:

- **Map server**: è il nodo che gestisce le mappe generate, in particolare occupandosi della pubblicazione delle informazioni.
- **Algoritmo di localizzazione**: come già spiegato, è stato scelto AMCL, il quale fornisce in uscita la posizione globale e la invia all'algoritmo di navigazione autonoma.
- **global_planner**: si tratta del pianificatore globale che, a partire dalla mappa pre-generata, valuta quali solo le migliori traiettorie da seguire per raggiungere il goal. Gli algoritmi utilizzati per la generazione delle traiettorie sono molteplici. Uno dei più utilizzati è **Dijkstra**, il quale utilizza la teoria dei grafi per valutare la traiettoria migliore. In particolare, la mappa viene vista come un grafo in cui i nodi sono i punti del piano in cui il robot può trovarsi e gli archi sono i percorsi che li collegano. Per ogni nodo, a partire da quello iniziale, viene calcolata la distanza tra il nodo e quello adiacente, per capire qual è la distanza più breve. Di volta in volta questo viene aggiornato fino ad arrivare al punto desiderato.
- **base_local_planner**: si tratta di un pacchetto che ha come scopo quello di pianificare la traiettoria a livello **locale**. Si tratta di un metodo particolarmente adattabile visto che può essere utilizzato per varie tipologie di robot. Infatti, il pacchetto fornisce un file di configurazione in cui si possono

specificare alcune caratteristiche fisiche come la velocità massima e minima, l'accelerazione massima e la grandezza del robot in metri. Il controllore implementato in questo algoritmo crea una funzione di costo locale che assegna un valore alle particelle che si trovano intorno al robot, facenti parte della mappa. Il compito successivo del controllore è quello di valutare i pesi assegnati dalla funzione di costo e determinare le velocità da inviare al robot.

- **dwa_local_planner**: si tratta di un'implementazione del base_local_planner che utilizza l'algoritmo **Dynamic Window Approach**. L'algoritmo segue i seguenti step:

  1. Si considera un insieme di possibili velocità $(dx, dy, dtheta)$ che il robot potrebbe assumere a partire dalla posizione attuale per raggiungere il goal. Lo spazio delle velocità in cui si opera considera tutte le caratteristiche cinematiche e dinamiche del robot.

  2. Si simulano delle possibili traiettorie a partire da ogni campione dello spazio di velocità, predicendo il comportamento del robot se per un intervallo di tempo determinato si muovesse con le velocità campionate.

  3. Si determinano dei fattori per la valutazione della bontà delle traiettorie predette, come ad esempio quanto il percorso è in grado di evitare gli ostacoli e quanto il robot si avvicina al goal.

  4. L'algoritmo assegna dinamicamente un peso in base a quanto la traiettoria soddisfa i suddetti fattori.

  5. Si seleziona la traiettoria con il valore più alto assegnato.

  6. Questo ciclo viene iterato per permettere al robot di adattarsi a qualsiasi cambiamento e variazione in tempo reale dell'ambiente circostante.

- **move_base**: si tratta del nodo principale dell'intero Navigation Stack, poichè si occupa del controllo e del coordinamento di tutti i componenti del sistema di navigazione. In particolare, il suo scopo è quello di ricevere l'obiettivo

della navigazione e di coordinare ciò che viene calcolato dai pianificatori (globale e locale).

Navigation Stack, inoltre, mette a disposizione diversi file di configurazione relativi all'ambiente circostante. In particolare troviamo:

- **local_costmap_params**: si tratta della mappa che ha come centro il robot e si basa su ciò che viene visto in tempo reale dai sensori.
- **local_costmap_params**: si tratta della mappa intera che riflette l'ambiente circostante.
- **costmap_common_params**: è il file di configurazione in cui vengono definiti i parametri comuni delle due tipologie di mappe.

Lo schema generale di funzionamento del Navigation Stack di ROS è il seguente:

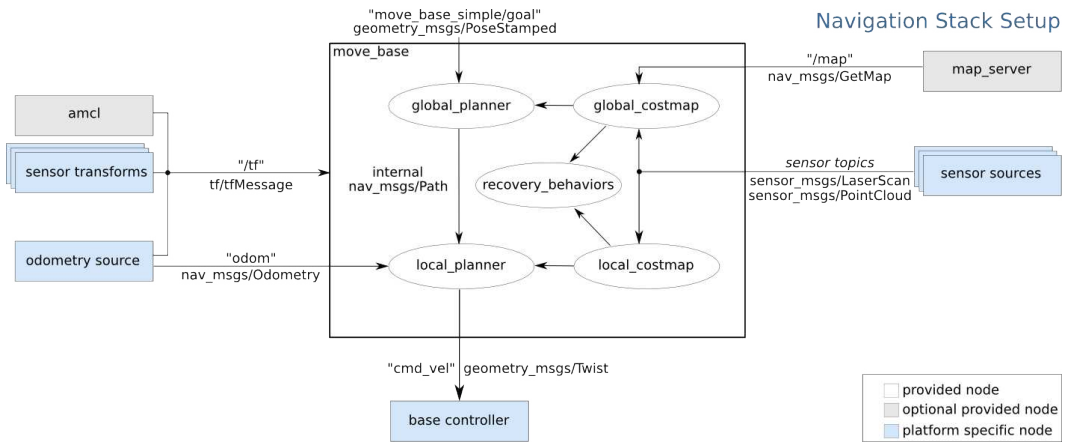

Figura 8.3: Schema di funzionamento Navigation Stack

8.3 Codice

Come si può vedere dallo schema della sezione precedente, dopo che il sistema di navigazione svolge i calcoli necessari e seleziona la traiettoria da seguire, sarà pubblicato il topic che contiene i comandi di velocità.
Per rendere il topic adattabile a tutte le configurazioni di robot, i valori espressi

si riferiscono alla velocità lineare lungo x del sistema e a quella angolare attorno all'asse z. Di conseguenza, si hanno a disposizione v_x e ω. L'AMR di questo progetto, però, comunica con Arduino inviando un vettore di dati relativi alla velocità in RPM della ruota sinistra e di quella destra separatamente, aggiungendo anche il verso di rotazione. Perciò, è necessario implementare un ulteriore nodo che si occupi della conversione dei valori pubblicati in cmd_vel grazie al modello cinematico del robot, affinché risultino compatibili con la tipologia di comunicazione implementata per Arduino. Inoltre, questo codice viene utilizzato anche dalla navigazione tramite joystick, poichè i comandi di velocità inviati sono nel medesimo formato.

Il codice è il seguente:

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include "std_msgs/Float64MultiArray.h"
#include "std_msgs/Float64.h"
#include "geometry_msgs/Twist.h"
#include "geometry_msgs/Pose.h"
#include "geometry_msgs/Quaternion.h"
#include "wheel.h"

class FromTwistToWheel
{

public:

  FromTwistToWheel()
  {
    pub = n.advertise<std_msgs::Float64MultiArray>("twist_toWheel", 1);
    sub = n.subscribe("cmd_vel", 1, &FromTwistToWheel::callback, this);
  }

```

```
22  void callback(const geometry_msgs::Twist &msg)
23  {
24    std_msgs::Float64MultiArray twist_joy_toWheel;
25    twist_joy_toWheel.data.resize(4);
26
27    float speedL = ((msg.linear.x - (msg.angular.z*L/2)) / (
     WHEEL_DIAM/2))*60/(2*PI);
28    float speedR =((msg.linear.x + (msg.angular.z*L/2)) / (
     WHEEL_DIAM/2))*60/(2*PI);
29
30    if((speedL < 0) && (speedR>0)){
31      twist_joy_toWheel.data[0] = BACKWARD;
32      twist_joy_toWheel.data[2] = FORWARD;
33
34    } else if((speedL < 0) && (speedR < 0)){
35      twist_joy_toWheel.data[0] = BACKWARD;
36      twist_joy_toWheel.data[2] = BACKWARD;
37
38    } else if ((speedL > 0) && (speedR>0)){
39      twist_joy_toWheel.data[0] = FORWARD;
40      twist_joy_toWheel.data[2] = FORWARD;
41
42    } else if ((speedL > 0) && (speedR < 0)){
43      twist_joy_toWheel.data[0] = FORWARD;
44      twist_joy_toWheel.data[2] = BACKWARD;
45
46    } else if((speedL == 0)&&(speedR == 0)){
47      twist_joy_toWheel.data[0] = 0;
48      twist_joy_toWheel.data[2] = 0;
49    }
50
51    twist_joy_toWheel.data[1] = abs(speedL);
52    twist_joy_toWheel.data[3] = abs(speedR);
53
54    pub.publish(twist_joy_toWheel);
55  }
56
```

```
57 private:
58
59   ros::NodeHandle n;
60   ros::Publisher pub;
61   ros::Subscriber sub;
62 };
63
64 int main(int argc, char **argv)
65 {
66   ros::init(argc, argv, "twist_toWheel");
67
68   FromTwistToWheel FTWObject;
69
70   ros::spin();
71
72   return 0;
73 }
```

All'interno del codice vengono utilizzate le formule apposite per effettuare per ogni messaggio ricevuto da cmd_vel la conversione da velocità lineare e angolare del robot a velocità in RPM della ruota sinistra e della ruota destra.
Inoltre, le grandezze vengono inserite all'interno di un vettore di quattro elementi: il primo e il terzo identificano il verso di rotazione, il secondo e il quarto indicano la velocità. Questo passaggio è fondamentale poiché l'algoritmo implementato in Arduino si sottoscrive al topic pubblicato da questo nodo e necessita i dati in questo formato.
Una volta permessa la corretta comunicazione tra il pc e Arduino, è possibile lanciare i progetti comprensivi di tutti i nodi necessari. Per quanto riguarda il joystick, il comando da digitare da terminale è il seguente:

```
$ roslaunch amr_idea joystick_control.launch
```

Il codice è il seguente:

```
1 <?xml version="1.0"?>
2
```

```
<launch>

    <node name="joint_state_publisher" pkg="joint_state_publisher
    " type="joint_state_publisher" output = "screen">
  </node>

    <node name="robot_state_publisher" pkg="robot_state_publisher
    " type="robot_state_publisher" output = "screen">
    </node>

    <param name="robot_description" command="$(find xacro)/xacro
    '$(find amr_idea)/urdf/robot.xacro'"/>

  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
    amr_idea)/rviz/robotmodel.rviz">
  </node>

  <rosparam file="$(find amr_idea)/config/diff_driver.yaml"
    command="load" />
    <node name="serial_node"        pkg="rosserial_arduino"
    type="serial_node.py">
  <param name="port"              type="string"
    value="/dev/ttyACM0"/>
    <param name="baud"              type="int"
    value="115200"/>
    </node>

    <node name="rplidarNode"          pkg="rplidar_ros"  type="
    rplidarNode" output="screen">
  <param name="serial_port"         type="string" value="/dev/
    ttyUSB0"/>
  <param name="serial_baudrate"     type="int"    value="115200"/
    >
  <param name="frame_id"            type="string" value="laser"/>
    <param name="inverted"            type="bool"   value="false"
    />
  <param name="angle_compensate"    type="bool"   value="true"/>
```

```
28   </node>
29
30     <arg name="joy_config" default="xbox" />
31     <arg name="joy_dev" default="/dev/input/js0" />
32     <arg name="config_filepath" default="$(find teleop_twist_joy)
     /config/$(arg joy_config).config.yaml" />
33     <arg name="joy_topic" default="joy" />
34
35     <node pkg="joy" type="joy_node" name="joy_node">
36     <param name="dev" value="$(arg joy_dev)" />
37     <param name="deadzone" value="0.3" />
38     <param name="autorepeat_rate" value="20" />
39     <remap from="joy" to="$(arg joy_topic)" />
40     </node>
41
42     <node pkg="teleop_twist_joy" name="teleop_twist_joy" type="
     teleop_node">
43     <rosparam command="load" file="$(arg config_filepath)" />
44     <remap from="joy" to="$(arg joy_topic)" />
45     </node>
46
47   <node name="robot_odom"         pkg="amr_idea"       type="
     robot_odom">
48   </node>
49
50     <node name="twist_toWheel"         pkg="amr_idea"       type="
     twist_toWheel">
51   </node>
52
53 </launch>
```

Per quanto riguarda la navigazione autonoma, il comando è il seguente:

```
$ roslaunch amr_idea autonomous_navigation.launch
```

Il codice del file launch è il seguente:

```
1 <?xml version="1.0"?>
```

```
2
3  <launch>
4
5      <node name="joint_state_publisher" pkg="joint_state_publisher
       " type="joint_state_publisher" output = "screen">
6    </node>
7
8      <node name="robot_state_publisher" pkg="robot_state_publisher
       " type="robot_state_publisher" output = "screen">
9      </node>
10
11     <param name="robot_description" command="$(find xacro)/xacro
       '$(find amr_idea)/urdf/robot.xacro'"/>
12
13   <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
       amr_idea)/rviz/robotmodel.rviz">
14   </node>
15
16   <rosparam file="$(find amr_idea)/config/diff_driver.yaml"
       command="load" />
17     <node name="serial_node"        pkg="rosserial_arduino"
       type="serial_node.py">
18     <param name="port"              type="string"
       value="/dev/ttyACM0"/>
19     <param name="baud"              type="int"
       value="115200"/>
20     </node>
21
22     <node name="rplidarNode"          pkg="rplidar_ros"  type="
       rplidarNode" output="screen">
23   <param name="serial_port"         type="string" value="/dev/
       ttyUSB0"/>
24   <param name="serial_baudrate"     type="int"    value="115200"/
       >
25   <param name="frame_id"            type="string" value="laser"/>
26     <param name="inverted"            type="bool"   value="false"
       />
```

```
27   <param name="angle_compensate"    type="bool"   value="true"/>
28   </node>
29
30   <arg name="map_file" default="$(find amr_idea)/maps/mappa.yaml"
      />
31   <node pkg="map_server" name="map_server" type="map_server" args
      ="$(arg map_file)">
32   </node>
33
34   <node name="robot_odom" pkg="amr_idea" type="robot_odom" output
       = "screen">
35   </node>
36
37   <include file="$(find amr_idea)/launch/amcl.launch"/>
38
39     <arg name="move_forward_only" default="true"/>
40     <include file="$(find amr_idea)/launch/move_base.launch">
41     <arg name="move_forward_only" value="$(arg move_forward_only)
      "/>
42     </include>
43
44     <node name="twist_toWheel" pkg="amr_idea" type="twist_toWheel
      ">
45   </node>
46
47 </launch>
```

8.4 Test e risultati

Ai fini della relazione finale è stato svolto un test di cui sono riportati i risultati ottenuti. Il test effettuato consiste in diversi passaggi:

1. **Allestimento dell'ambiente**: il primo passaggio consiste nell'utilizzare i pannelli acquistati per l'allestimento di un ambiente chiuso, all'interno del quale sarà inserito il robot.

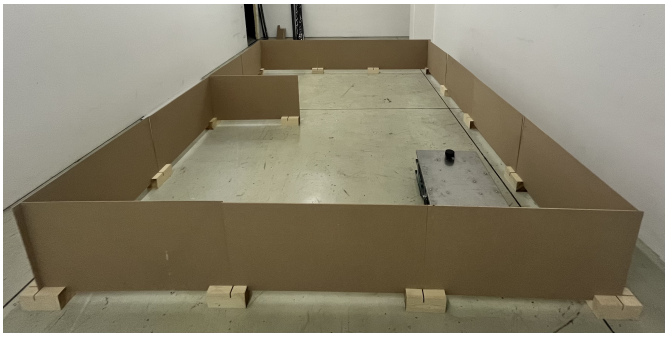

Figura 8.4: Ambiente di test

2. **Avvio della navigazione tramite joystick**: successivamente, viene avviato il progetto relativo al controllo tramite joystick, in modo tale da permettere al robot l'ispezione dell'ambiente.

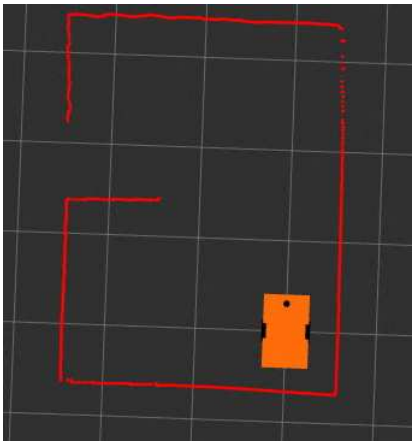

Figura 8.5: Robot in RViz

3. **Avvio dell'algoritmo di mapping**: grazie al controller, è possibile muovere il robot all'interno dell'ambiente senza tralasciare nessun punto, in modo tale da poter avviare l'algoritmo di mapping al fine di generare la mappa.

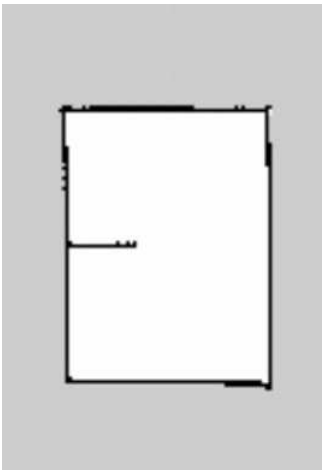

Figura 8.6: Mappa generata

4. **Avvio del progetto di navigazione autonoma**: a questo punto, la mappa è stata generata e le informazioni sono state salvate all'interno dei file appositi. Di conseguenza, disponendo di una mappa e di un sistema di localizzazione all'interno di essa, è possibile avviare la navigazione autonoma.

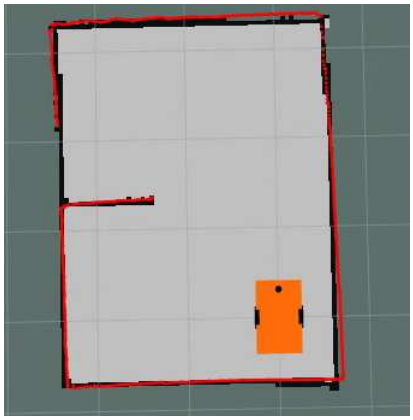

Figura 8.7: Robot nella mappa in RViz

5. **Selezione di un goal**: tramite l'interfaccia fornita da RViz, è possibili utilizzare il mouse per indicare un punto da raggiungere all'interno della mappa. Per farlo, è necessario cliccare in alto sul tasto **2D Nav Goal**, grazie al quale può essere specificata la posizione e l'orientamento desiderati.

A questo punto, l'algoritmo valuterà la migliore traiettoria da seguire sulla base della posizione iniziale e dell'obiettivo.

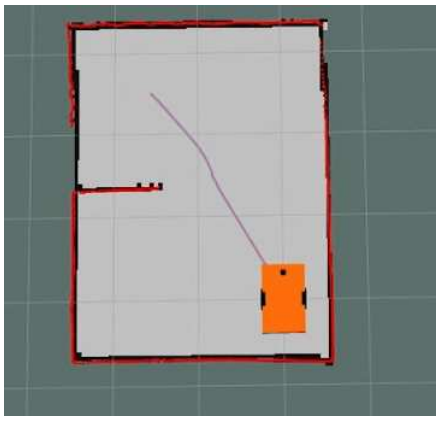

Figura 8.8: Traiettoria selezionata

6. **Raggiungimento dell'obiettivo**: il robot riceverà i comandi di velocità, i quali saranno estrapolati e manipolati al fine di essere inviati correttamente a Arduino. In questo modo, le ruote saranno messe in movimento e il robot inizierà a muoversi per raggiungere l'obiettivo.

Capitolo 9

Conclusioni

Questo lavoro di tirocinio e tesi è stato strutturato in diversi passaggi, ognuno fondamentale e strettamente collegato agli altri. Le principali aree che sono state trattate sono descritte di seguito.

- **Analisi della struttura hardware**: il primo grande obiettivo è stato quello di comprendere al meglio quali fossero e come funzionassero tutti i componenti hardware del robot. Per riuscirci, è stato necessario analizzare nel dettaglio ogni singolo dispositivo, dai driver che pilotano le ruote all'Arduino installato a bordo del robot. Il lavoro svolto sotto questo punto di vista non è stato solo di analisi, bensì anche di monitoraggio e manutenzione. Infatti, dopo aver capito il funzionamento di ogni componente, si è cercato nel corso del tempo di trovare soluzioni al fine di migliorare ed ottimizzare la struttura dell'AMR. Ad esempio, svolgendo vari test sono sorte alcune problematiche relativamente alle batterie. Di conseguenza, è stato progettato e costruito un nuovo sistema di alimentazione formato da un selettore in grado di utilizzare tre batterie al piombo collegate in serie durante la fase di funzionamento. Inoltre, sono state svolte delle analisi per capire quali componenti necessitano di essere sostituiti per avere risultati migliori e quali dispositivi possono essere integrati al sistema per ampliarne l'efficienza.

- **Analisi del software**: come già spiegato, sul robot è montato un mini PC Asus che ha il compito di controllare ad alto livello il robot tramite ROS. Di conseguenza, prima del lavoro sono stati analizzati profondamente i principi di funzionamento di questo middleware, a partire dai vari paradigmi utilizzati per la comunicazione dei nodi fino ad arrivare a tutti i pacchetti a disposizione per i vari task. Per un utilizzo ottimale di ROS è fondamentale la documentazione presente online: ogni pacchetto pre-implementato viene ampiamente spiegato nel sito ufficiale, in cui vengono analizzati i vari dettagli relativi agli algoritmi utilizzati, ai messaggi scambiati e agli input necessari. Durante tutto lo svolgimento del lavoro è stata svolta una continua ricerca per trovare gli algoritmi più efficienti e adattabili al sistema utilizzato.

- **Analisi del lavoro già svolto**: le attività sviluppate in questo periodo di tirocinio sono legate a quelle svolte in precedenza da altri tirocinanti. A tal proposito, in un primo momento l'attenzione è stata spostata sui task già portati a termine. In particolare, il primo studente ad utilizzare l'AMR si è occupato della struttura hardware e di una prima movimentazione delle ruote tramite ROS. In seguito, un'altra attività di tirocinio ha avuto come scopo quello di simulare il comportamento del robot utilizzando **Gazebo**, ossia un simulatore open-source in grado di fornire un ambiente 3D realistico per lo sviluppo di applicazioni robotiche. Questo lavoro consisteva nel simulare la guida autonoma del robot, utilizzando anche in questo caso Navigation Stack. Tuttavia, l'odometria proveniente dagli encoder e le scansioni provenienti dal LIDAR sono state simulate. Di conseguenza, l'ultimo step consisteva nell'integrare la struttura hardware agli algoritmi utilizzati per la simulazione, al fine di ottenere un sistema reale funzionante. Tutte queste implementazioni sono state fondamentali per avere delle basi da cui partire, anche se in seguito sono state utilizzate metodologie e approcci differenti per alcuni task, visto che in simulazione possono essere trascurati alcuni aspetti che nella realtà risultano particolarmente importanti.

- **Approfondimenti teorici**: durante tutto lo svolgimento del progetto è sta-

to necessario svolgere alcuni approfondimenti legati a degli aspetti teorici. In particolare, il processo di autolocalizzazione del robot all'interno della mappa può essere messo a punto soltanto dopo profonde ricerche relativamente alle varie tecniche di filtraggio. Innanzitutto, una prima analisi è stata svolta in relazione ai filtri particellari che caratterizzano l'algoritmo AMCL. Infine, sono stati ripercorsi tutti gli aspetti teorici e analitici del filtro di Kalman, per comprendere al meglio come esso potesse essere integrato all'interno del sistema.

- **Implementazione e raggiungimento degli obiettivi**: dopo aver analizzato la struttura hardware e i software utilizzati e dopo aver studiato approfonditamente a livello teorico gli algoritmi da utilizzare, l'intero lavoro si è basato sulla programmazione, sull'integrazione delle varie parti e sull'ottimizzazione. Sono stati approfonditi numerosi aspetti legati al linguaggio C++ e di come esso è legato a ROS tramite l'utilizzo di pacchetti, librerie ed algoritmi. Inoltre, è stato approfondito il funzionamento di Arduino e del suo IDE. L'integrazione di tutti questi componenti ha permesso il raggiungimento degli obiettivi.

9.1 Sviluppi futuri

Avendo a disposizione un tempo limitato per lavorare sull'AMR, sono stati pensati degli sviluppi che potranno essere applicati alla struttura del robot più avanti. Di seguito sono riportati nel dettaglio:

1. **Integrazione di ulteriori sensori**: un ruolo chiave all'interno del progetto è stato giocato senza dubbio dai vari sensori installati, come ad esempio il LIDAR. Sicuramente, più il numero di sensori utilizzati è alto, migliore sarà la percezione dell'ambiente. Di conseguenza, tutti i task (mapping, localizzazione, generazione di una traiettoria, ecc.) saranno svolti con maggiore precisione ed accuratezza. I sensori che potrebbero essere integrati sono i seguenti:

- Un **Adafruit 9-DOF IMU**, ossia un sistema elettronico in grado di misurare l'accelerazione e la velocità angolare di un corpo. In particolare, l'IMU (Inertial Measurement Unit) è dotata di diversi sensori per misurare le accelerazioni lungo x, y e z (accelerometro), la velocità angolare lungo x, y e z (giroscopio), il campo magnetico generato (magnetometro) e la pressione atmosferica (barometro).

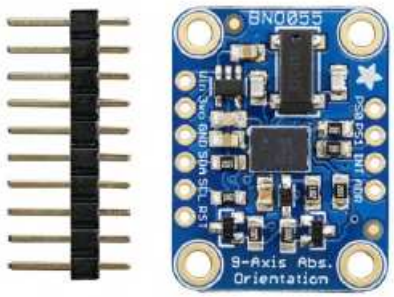

Figura 9.1: Adafruit 9-DOF IMU

- Una **telecamera di profondità Intel RealSense D457**, ossia una stereo camera costituita da due fotocamere poste ad una determinata distanza. In questo modo, la camera è in grado di ottenere informazioni sulla profondità degli oggetti e di percepire l'ambiente circostante in maniera tridimensionale, proprio come gli occhi umani. Questa camera è stata pensata per sviluppare un sistema di visione, ossia un insieme di componenti nato per interpretare dati visivi.

Figura 9.2: Telecamera Intel RealSense D457

2. **Completamento e testing del filtro di Kalman per la localizzazione**: come spiegato in precedenza, è stato implementato da zero un algoritmo basato sull'estrazione di features ambientali e su un filtro di Kalman per la localizzazione del robot all'interno della mappa. Tuttavia, sono stati individuati i passi elementari per la progettazione dell'algoritmo, per poi scrivere una prima bozza di codice. Di conseguenza, un possibile sviluppo futuro consiste nel migliorare e ottimizzare il codice ed effettuare dei test per verificarne il funzionamento.

Bibliografia

[1] How to Build an Indoor Map Using ROS. *https:// automaticaddison.com/ how-to-build-an-indoor-map-using-ros-and-lidar-based-slam/google_vignette*.

[2] Luigi Chisci. *Filtraggio non lineare*. 2019.

[3] Gervasio Danilo. *Design and development of an Autonomous Mobile Robot (AMR) based on a ROS controller*. A.A 2021/2022.

[4] Giangiacomi Gabriele. *Simulazione di un algoritmo Slam e navigazione autonoma per un AMR progettato su piattaforma ROS*. A.A 2021/2022.

[5] Prof. Ippoliti Gianluca. *Sinusoidal Pulse-Width-Modulation*.

[6] Sensori a effetto Hall. *https://it.rs-online.com/web/c/automazione-e-controllo-di-processo/sensori/sensori-a-effetto-hall/*.

[7] Shubham Nagla. *2D Hector SLAM of Indoor Mobile Robot using 2D Lidar*. 2020.

[8] Differential Drive Robots. *https://www.cs.columbia.edu/ allen/F17/NOTES/ icckinematics.pdf*.

[9] Davide Scaramuzza Roland Siegwart Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. 2011.

[10] LIDAR integration with ROS. *https://www.generationrobots.com/blog/en/lidar-integration-with-ros-quickstart-guide-and-projects-ideas/*.

[11] RP LIDAR Laser Scanner. *https://www.slamtec.com/*.

[12] Clovis Peruchi Scotti. *Design, Implementation and Evaluation of a Software Solution for Localization, Navigation and Mission Management Applied to Autonomous Mobile Robots*. 2010.

[13] Documentation - ROS Wiki. *https://wiki.ros.org*.

[14] Kaiyu Zheng. *ROS Navigation Tuning Guide*. 2016.

Appendice A

Motori Brushless DC

Il **motore brushless** è una tipologia di motore elettrico, il cui utilizzo oggi è sempre più frequente nel settore della robotica. In particolare, il compito principale di un motore elettrico è quello di convertire energia elettrica in energia meccanica, permettendo in questo modo il movimento del carico. Il motore brushless è costituito da un **rotore**, su cui sono alloggiati i magneti permanenti, e da uno **statore** su cui, proprio come in un motore asincrono, sono avvolti gli avvolgimenti di fase.

I motori brushless, in particolare, sono detti motori **sincroni**, poiché il campo magnetico generato dalle correnti di fase risulta sempre ortogonale e sincrono al campo magnetico generato dai magneti permanenti che costituiscono il rotore. Questo risultato si ottiene alimentando le fasi statoriche alternativamente. Il sincronismo, inoltre, si mantiene commutando le correnti negli avvolgimenti in funzione della posizione angolare del rotore. Questo viene garantito dalla presenza di un inverter che, fungendo da commutatore elettronico, permette di evitare il sistema spazzole-collettore tipico di una macchina a corrente continua.

I motori DC Brushless vengono anche chiamati **brushless trapezio** dal momento in cui le forze controelettromotrici indotte che si producono nello statore durante la rotazione presentano un andamento trapezoidale. Contrariamente, se le forze indotte presentano una forma d'onda sinusoidale, il motore sarà un AC Brushless

(brushless sinusoidale).

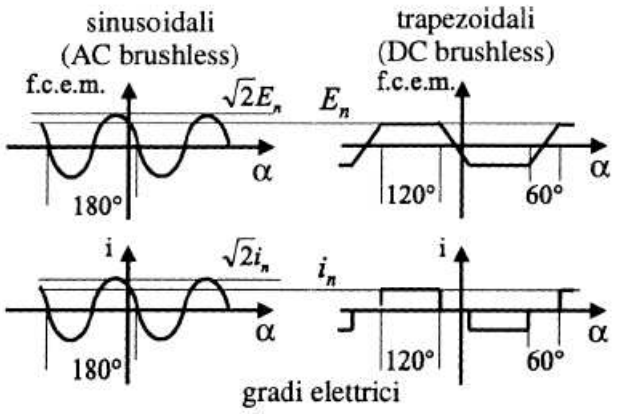

Figura A.1: Forme d'onda delle forze indotte nei motori Brushless

In genere, gli azionamenti di tipo brushless vengono impiegati nel caso di piccole-medie potenze a causa dei magneti permanenti, le cui dimensioni risulterebbero troppo elevate per potenze maggiori. Nei settori in cui vengono utilizzati, come nel caso della robotica, i vantaggi principali sono i seguenti:

- possono essere raggiunte velocità elevate a parità di potenza;
- vi è una riduzione significativa relativamente al peso e al volume;
- risultano più semplici dal punto di vista delle modalità costruttive e presentano una maggiore robustezza rispetto ai motori in corrente continua;
- l'assenza delle spazzole consente una frequenza di manutenzione molto più bassa.

A.1 Sensori ad effetto Hall

I sensori ad effetto Hall, generalmente integrati all'interno dei motori elettrici, sono dispositivi elettronici utilizzati per rilevare e misurare i campi magnetici. In particolare, rilevano l'**effetto hall**, ossia un fenomeno scoperto nel 1879 dall'omonimo scienziato, secondo cui se un magnete viene posizionato in maniera perpendicolare ad un conduttore di corrente costante, gli elettroni vengono spostati

da un lato creando una differenza di carica. Di conseguenza, questo fenomeno indica la presenza di un campo magnetico di cui è possibile misurare l'intensità. Grazie a questi sensori, si possono rilevare grandezze importanti come la velocità o lo spostamento dell'intero sistema.

Appendice B

Pulse Width Modulation (PWM)

PWM è la sigla di *Pulse Width Modulation*, ossia *Modulazione in larghezza d'impulso*. Si tratta di una tipologia di segnali caratterizzati da un impulso la cui durata dipende dal confronto con un segnale di riferimento. Questa tecnica di modulazione è prevalentemente utilizzata per il controllo di dispositivi di conversione di potenza, poiché è in grado di intervenire sulla quantità di energia inviata al carico quando la frequenza è costante.

Un segnale PWM presenta delle fasi di on e delle fasi di off ed è caratterizzato da un insieme di impulsi all'interno del periodo, la cui durata determina quanta energia trasferire al carico.

In particolare, questa tecnica di modulazione prevede un confronto del segnale con un segnale di riferimento generalmente sinusoidale, in modo tale da determinare la durata dell'impulso. Utilizzando come confronto un segnale di questo tipo, gli impulsi risulteranno di lunghezza variabile.

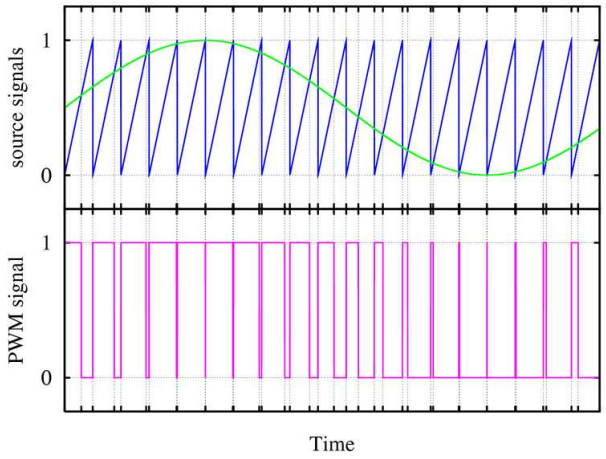

Figura B.1: Segnale PWM

I segnali PWM sono generalmente utilizzati in diversi ambiti, come ad esempio nel controllo dei convertitori di potenza per regolare la tensione di uscita o nei sistemi di controllo PID per assicurarsi che una grandezza sia il più possibile vicina ad un valore di riferimento.

Tuttavia, ai fini di questo progetto, sono stati utilizzati per il controllo della velocità dei motori brushless integrati all'interno delle ruote. La velocità con cui i motori ruotano, infatti, viene regolata grazie alla variazione della potenza trasferita al motore grazie alla modulazione della durata degli impulsi.