UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale: INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Simulazione di un algoritmo Slam e navigazione autonoma per un AMR progettato su piattaforma ROS

Simulation of a Slam algorithm and autonomous navigation for an AMR designed on ROS platform

Relatore:
Prof. Ippoliti Gianluca

Tesi di Laurea di:
Giangiacomi Gabriele

Correlatore:
Dott. Di Buò Gianluca

A.A. 2021/2022

Ringraziamenti

Questa tesi è dedicata in primo luogo alla mia famiglia, che mi ha supportato economicamente ed emotivamente in tutti questi anni per la realizzazione di questa laurea, spronandomi ad andare avanti nei periodi meno facili. Altri ringraziamenti vanno ai componenti dell'azienda IDEA soc.coop tra cui il tutor aziendale e correlatore Gianluca Di Buò, ad Alessandro Ciancaglione e a tutto lo staff per avermi ospitato e aiutato in questo progetto di tirocinio che sicuramente mi ha fatto crescere dal punto di vista professionale. Un ulteriore ringraziamento va al professore, nonché relatore di questa tesi, Gianluca Ippoliti per l'opportunità che mi ha dato di svolgere il tirocinio in questa azienda. Inoltre, dedico questa tesi a tutti i miei amici, universitari e non, che mi sono stati a fianco durante questo percorso.

Indice

Capitolo

1 Introduzione

1.1 Il progetto

Il seguente progetto di tirocinio nasce con l'idea di far muovere in modo del tutto autonomo un AMR (Autonomous Mobile Robot) a guida differenziale in un ambiente con ostacoli. Il robot, scansionando l'area circostante e fornito un obbiettivo da raggiungere all'interno della mappa, deve cercare il miglior tragitto da intraprendere, in termini di minor tempo impiegato e che eviti gli ostacoli presenti nel cammino. All'inizio del mio percorso di tirocinio il robot era già stato costruito da un altro studente, il quale aveva iniziato questo progetto riuscendo a farlo muovere manualmente attraverso un sistema di comunicazione Client-Server tra nodi ROS (un software di sviluppo di applicazioni robot) collegati alle varie componenti hardware del robot, tra cui controller delle ruote, scheda di controllo Arduino ecc. In questo lavoro di tesi ho implementato, attraverso il ROS, la scansione di un ambiente di simulazione virtuale circostante al modello simulato del robot reale e la guida autonoma attraverso l'uso di un algoritmo Slam e un filtro di particelle di cui parlerò in seguito. Futuri sviluppi consisteranno nel fare il porting da simulazione a modello reale, passando agli algoritmi implementati, i dati del sensore Lidar utilizzato, i dati di odometria delle ruote verificando la stabilità del robot ed il corretto funzionamento dell'algoritmo di navigazione autonoma.

1.2 L'azienda

Il luogo in cui ho sviluppato questo progetto di tesi è l'azienda "IDEA Soc. Coop" situata in Ancona. Essa è una società di ingegneria che si occupa di automazione industriale, robotica, elettronica, domotica e ICT. È composta da giovani tecnici e ingegneri e offrono prodotti e servizi di qualità per soddisfare le esigenze dei loro clienti. Il periodo di tirocinio si è svolto da ottobre 2022 a dicembre 2022.

Fig 1.1: Logo azienda IDEA Soc. Coop

1.3 Cos'è un AMR

Un AMR (Autonomous Mobile Robot) è un robot mobile capace di navigare in un ambiente in modo autonomo, senza la necessità di dispositivi di guida fisici o elettromeccanici, come invece sono richiesti dagli AGV (Automated Guided Vehicle) i quali utilizzano marcatori fisici disposti nell'ambiente nel quale devono navigare ad esempio sensori, magneti, nastri.

Gli AMR sono molto utilizzati nelle applicazioni industriali e nel trasporto di materiali. Mentre gli AGV seguono bande magnetiche o altri dispositivi

preimpostati per localizzarsi nello spazio, gli AMR sfruttano laser scanner e telecamere 3D che consentono di ricevere informazioni sull'ambiente circostante, riconoscere ostacoli e modificare dinamicamente il proprio percorso.

Fig 1.2: Esempio di AMR

Gli AMR sono in grado di riconoscere gli ostacoli che si presentano nel loro percorso e di aggirarli creando mappe dell'ambiente attorno ad essi, mentre gli AGV possono solamente rilevare gli ostacoli ma non aggirarli a causa del fatto che seguono percorsi prestabiliti dettati appunto dai marcatori ambientali. Quindi gli AMR sono molto più flessibili da questo punto di vista.

Capitolo

2 Il robot

Come precedentemente accennato nella parte introduttiva, il modello fisico del robot era già stato creato da uno studente che prima di me ha svolto, sempre nella stessa azienda, il tirocinio di laurea magistrale. Di seguito riporto una breve descrizione della struttura e di quelle che sono le sue componenti hardware principali.

2.1 Principali componenti hardware

Ruote: le ruote sono motorizzate tramite due motori brushless (BLDC) ciascuno avente il proprio driver di controllo e aventi le seguenti caratteristiche:

- Tensione: 36 V
- Potenza: 250 W
- Diametro: 25 cm
- Spessore: 5 cm

Fig 2.1: Motore brushless (BLDC)

Ruote a sfera: sono state montate quattro ruote a sfera negli angoli del robot del diametro di 3 cm.

Fig 2.2: Ruota a sfera

IMU: è stata usata l'IMU di Adafruit 9-DOF

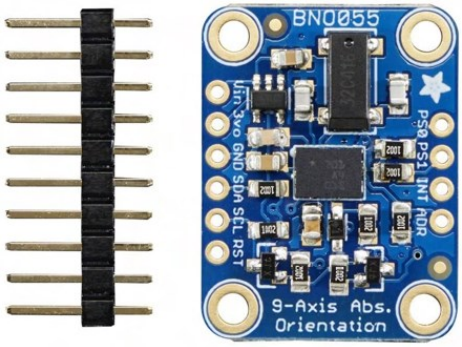

Fig 2.3: IMU Adafruit 9 DOF

Arduino Mega: è stato implementato il microcontrollore Arduino Mega 2560 REV3 per comunicare con i driver dei motori brushless

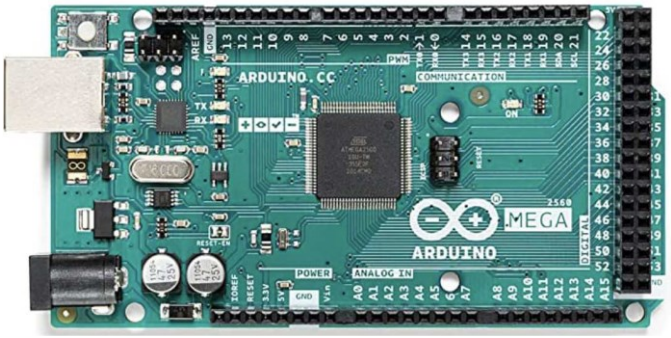

Fig 2.4: Arduino Mega 2560 REV3

Lidar: è stato scelto l'RPLidar A1 della Slamtec per il suo prezzo contenuto e per la sua buona performance. Questa componente è fondamentale per la mappatura dell'ambiente attorno al robot.

Specifiche:

- Range di misura: 0.15 m – 12 m
- Dimensioni: 96.8 x 70.3 x 55mm
- Tensione: 5V
- Range angolare: 360°
- Peso: 170 g

Fig 2.5: Slamtec RPLidar A1

Mini PC: per controllare i movimenti del robot è stato usato un mini-PC Asus PB50. Esso interagisce con la scheda Arduino e con il Lidar inviando comandi che l'utente trasmette da terminale in remoto.

Fig 2.6: Mini PC Asus PB50

Struttura: l'AMR è composto da acciaio inox per una buona solidità e resistenza alla corrosione.

Fig 2.7: Visuale sinistra del robot

Fig 2.8: Visuale destra del robot

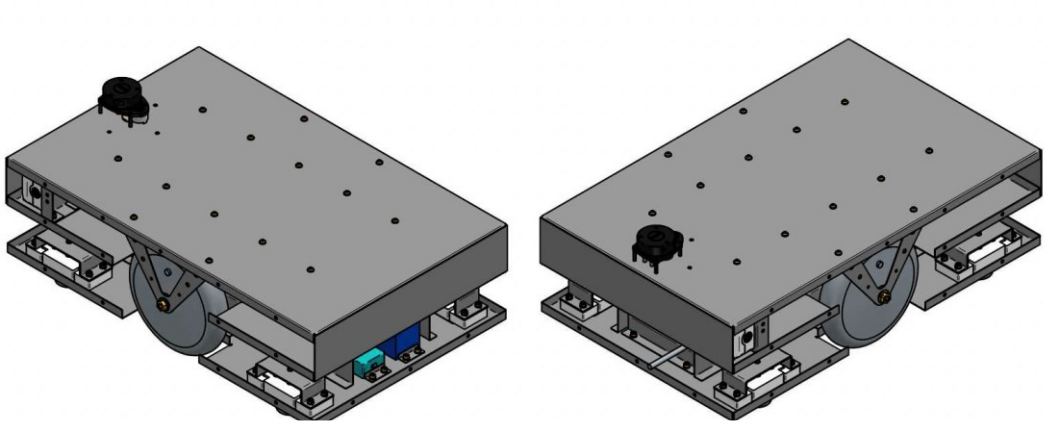

Fig 2.9: Modello 3D del telaio

2.2 Accenni al modello cinematico

L'AMR in questione è un robot a guida differenziale, vale a dire che è costituito da due ruote montate sullo stesso asse ed ogni ruota può essere guidata avanti o indietro indipendentemente dall'altra. Per far compiere al robot un movimento curvilineo, è necessario far variare le velocità di ciascuna ruota. Questo fa sì che il robot attui un movimento rotatorio attorno ad un punto situato nell'asse di congiunzione tra la ruota destra e sinistra chiamato "Centro di curvatura istantaneo" (ICC). La posizione del robot è espressa tramite le coordinate (x, y).

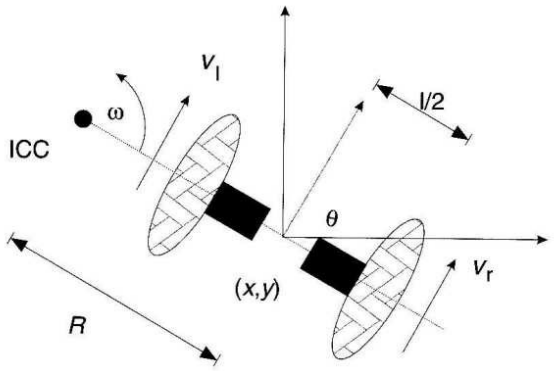

Fig 2.10: Cinematica guida differenziale

Nella guida differenziale la velocità angolare delle ruote determina spostamenti rettilinei o curvilinei. Se si conosce la velocità angolare delle singole ruote allora possono essere definite le seguenti formule:

$$v(t) = \frac{\omega(t)_R + \omega(t)_L}{2} r \qquad (2.1)$$

$$\omega(t) = \frac{\omega(t)_R - \omega(t)_L}{l} r \qquad (2.2)$$

dove $\omega(t)_R$ e $\omega(t)_L$ sono rispettivamente le velocità angolari della ruota destra e sinistra, r rappresenta il raggio delle ruote motrici, l è la distanza tra le ruote lungo l'asse che le congiunge, mentre $\omega(t)$ e $v(t)$ sono la velocità angolare del veicolo e la velocità tangenziale.

Sfruttando le precedenti equazioni, il modello cinematico del robot è dato dalle seguenti equazioni:

$$\dot{x}(t) = v(t)\cos(\theta) \qquad (2.3)$$

$$\dot{y}(t) = v(t)\sin(\theta) \qquad (2.4)$$

$$\dot{\theta}(t) = \omega(t) \qquad (2.5)$$

con $\theta(t)$ differenza angolare tra un sistema di riferimento globale fisso e il sistema di riferimento locale del robot, quest'ultimo individuato da un'origine in un punto P solidale al robot. È fondamentale definire questi due sistemi di riferimento al fine di localizzare il robot nel piano di spostamento.

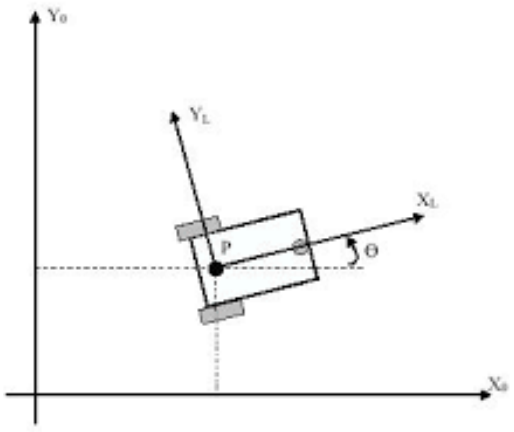

Fig 2.11: Sistema di riferimento globale e locale

Se indichiamo con R la distanza tra il punto centrale nell'asse di congiunzione tra le ruote e il centro di curvatura istantaneo, ω la velocità angolare del sistema e con v_R e v_L le velocità destra e sinistra delle ruote, si possono ricavare R ed ω in questo modo:

$$R = \frac{v_L + v_R}{v_R - v_L} \qquad (2.6)$$

$$\omega = \frac{v_R - v_L}{l} \qquad (2.7)$$

In base ai valori delle velocità delle ruote si possono distinguere tre casi di movimento per una guida differenziale:

1) il robot si muove in avanti quando $v_L = v_R$,infatti R tenderà ad infinito e ω sarà 0
2) il robot ruota su sé stesso quando $v_L = -v_R$ o viceversa, in questo caso R sarà uguale a 0 e il robot compirà una rotazione attorno al suo centro di massa
3) il robot ruota a sinistra (o a destra) quando $v_L = 0$ $(v_R = 0)$.

Capitolo

3 Ambiente di simulazione

Come detto all'inizio, lo studente che prima di me ha iniziato questo progetto ha utilizzato un particolare software per lo sviluppo di robot, il ROS.

3.1 Cos'è il ROS

Il nome "**ROS**" sta per "Robot Operating System" ma, al contrario di quanto si possa immaginare, non è un sistema operativo. Piuttosto è un framework, ovvero un insieme di librerie, pacchetti e strumenti usati per facilitare la programmazione di robot. In particolare, è un middleware open source, cioè un software basato su un meccanismo di comunicazione Client/Server tra moduli (**pacchetti**) che sono sottoprogrammi di un'applicazione robot, dove ogni modulo svolge un compito preciso. Questi moduli contengono dei nodi. I **nodi** non sono altro che programmi eseguibili dove ognuno può gestire una componente del robot (motori, sensori, localizzazione, pianificazione del percorso ecc.).

I nodi possono essere di due tipi: **publisher** o **subscriber**. I primi pubblicano delle informazioni mentre i subscriber le leggono. Insieme, possono essere visualizzati in un grafo che li connette mostrando come comunicano tra loro e che cosa si comunicano.

Ci sono diverse versioni ROS che sono state sviluppate, ma quella che ha utilizzato lo studente prima di me, e che quindi io ho ripreso, è la versione Noetic 1. Inoltre, il principale sistema operativo che supporta ROS è Ubuntu. Nel mio caso ho usato la versione Ubuntu Focal 20.04.5 su macchina virtuale.

3.2 Comunicazione tra nodi

I nodi sono contenuti all'interno di pacchetti, i quali contengono librerie e codici sorgente. La comunicazione tra nodi avviene attraverso:

- **Topics**: sono variabili che contengono informazioni. I nodi comunicano tra loro inviando e ricevendo i topic.
- **Services**: sono servizi che i nodi possono offrire o di cui usufruire per comunicare tra loro in modo sincrono. Sono utili anche per cambiare delle impostazioni del robot.
- **Actions**: azioni basate sui topic e che forniscono un'architettura client/server asincrona tra nodi, ovvero se il client invia una richiesta che richiede del tempo al server per elaborarla, esso può monitorare lo stato del server in ogni momento e nel frattempo dedicarsi ad altro.

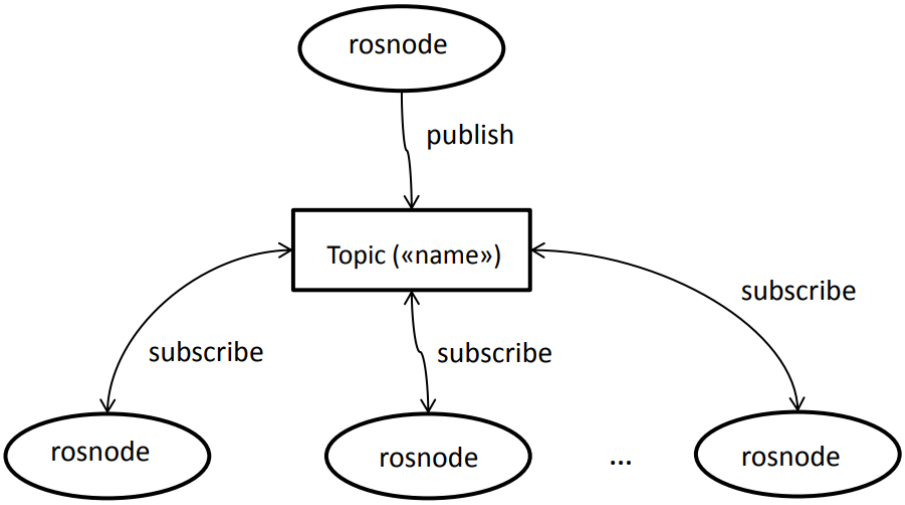

Fig 3.1: Grafico esemplificativo di comunicazione tra nodi ROS

Tutti i nodi ROS, per poter comunicare tra loro, fanno riferimento a un nodo principale che avvia l'intero sistema e che tiene traccia di ogni singolo nodo

publisher e subscriber, dei servizi e dei topic inviati. Questo nodo viene chiamato **Master**.

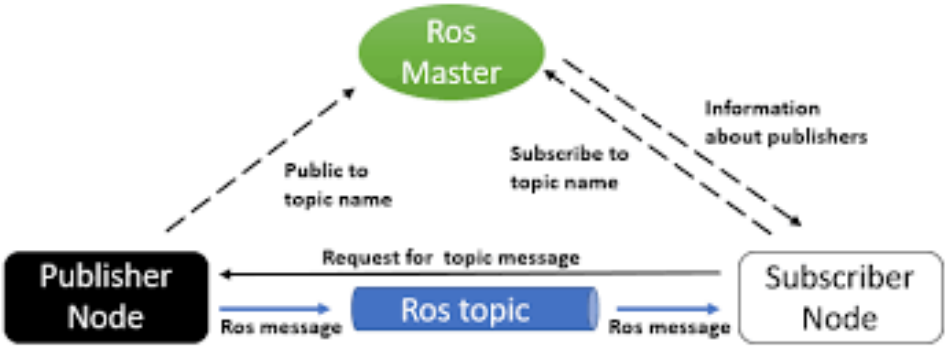

Fig 3.2: Interazione nodo Master con altri nodi

Per poter interagire con il sistema ROS, l'utente ha a disposizione numerosi comandi da avviare su terminale. Ne riporto solo alcuni:

- **roscore**: è il comando per avviare il nodo Master senza il quale gli altri nodi non vengono attivati.
- **rosrun**: è il comando per avviare un solo nodo.
- **roslaunch**: è il comando per avviare più nodi contemporaneamente (questo comando avvia automaticamente anche il Master).
- **rosnode**: è il comando per ottenere informazioni su un particolare nodo o una lista di nodi attivi in quel momento.
- **rostopic**: è il comando per avere informazioni su un particolare topic, ad esempio da quali nodi viene pubblicato o sottoscritto (visualizzato) o anche per avere una lista di topic attualmente attivi.

ROS usa principalmente due linguaggi di programmazione per lo sviluppo di applicazioni robot: il **C++** e il **Python**. Quando in un certo workspace si creano pacchetti che conterranno codici, i quali a loro volta avvieranno nodi, bisogna specificare che tipo di linguaggio si vuole utilizzare. Per fare ciò, sono disponibili due librerie: **roscpp** e **rospy**. Queste vengono definite alla creazione di un pacchetto ROS come "dipendenze" del pacchetto per stabilire se si userà codice C++ piuttosto che Python. Ci sono comunque anche altre librerie ROS che permettono di scrivere codice in altri linguaggi ad esempio Java o Javascript.

3.3 Rviz e Gazebo

L'ambiente di simulazione nel quale si è svolto il lavoro contenuto in questa tesi, viene reso disponibile da ROS tramite di due tool principali di visualizzazione 3D: **Gazebo** e **Rviz**.

3.3.1 Gazebo

Gazebo è un simulatore dinamico 3D utilizzato per simulare modelli robotici complessi in ambienti interni ed esterni. Fornisce la possibilità di testare il modello robotico nell'ambiente simulato e incorporare i dati dai sensori. Esso rappresenta la simulazione del mondo reale nel quale il robot si muove e interagisce. È capace di simulare grandezze fisiche che agiscono sul robot come gravità, inerzia o attrito ma anche condizioni atmosferiche. Si possono creare manualmente delle mappe ambientali a proprio piacimento con muri, oggetti di vario tipo, edifici od ostacoli che sono già presenti in Gazebo oppure utilizzare delle mappe preconfigurate di diversa grandezza.

Gazebo non è altro che un nodo contenuto in un pacchetto denominato "gazebo_ros" che contiene servizi e plugins per l'interfaccia con l'ambiente.

Può essere avviato da terminale attraverso il comando "**rosrun gazebo_ros gazebo"** o semplicemente con **"gazebo"**.

Se all'avvio non viene configurato in nessun modo, ad esempio non gli viene passata nessuna mappa o nessun modello di robot, la sua schermata iniziale sarà vuota come nella figura seguente.

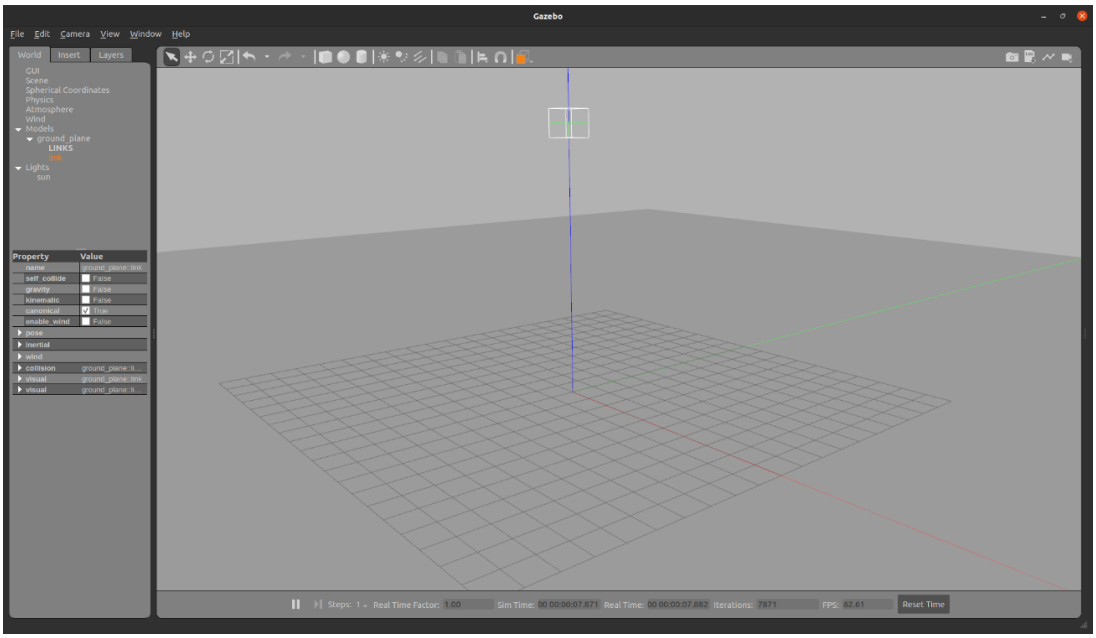

Fig 3.3: Schermata iniziale Gazebo (mondo vuoto)

3.3.2 Rviz

Rviz è un acronimo per "Ros visualization" ed è un potente strumento di visualizzazione 3D per ROS. Consente all'utente di visualizzare il modello del robot simulato, di registrare le informazioni dei sensori del robot e di riprodurle visualizzandole su schermo. In pratica permette di vedere quello che il robot percepisce dall'ambiente circostante attraverso i dati dei sensori o telecamere che sono installate su di esso e che vengono visualizzate su Rviz come dati immagine.

Rviz può essere avviato con il comando **"rosrun rviz rviz"** ed appare con la seguente schermata.

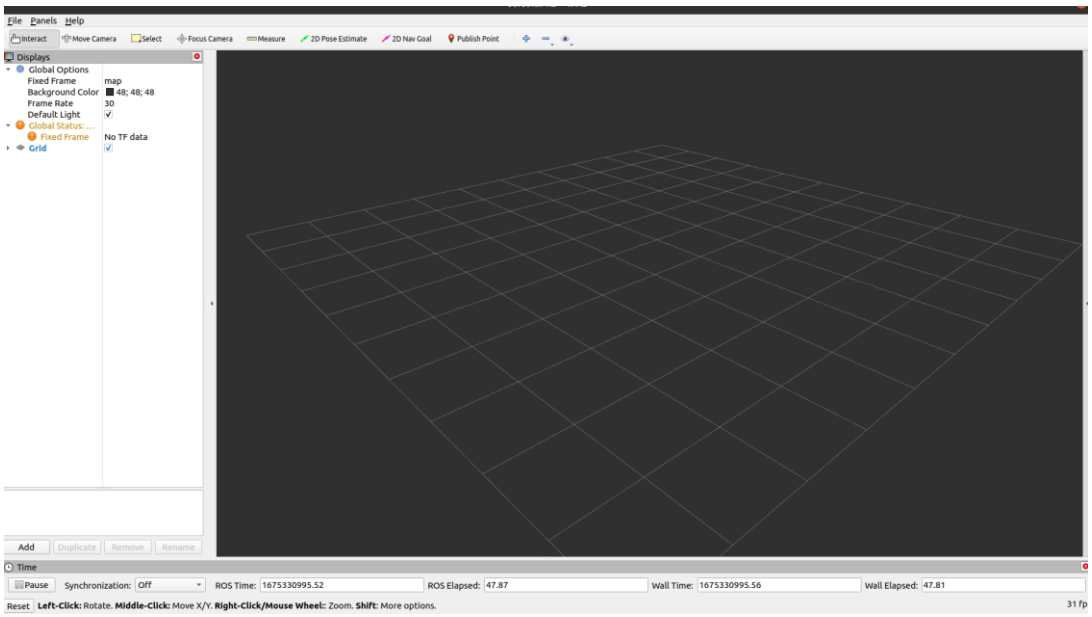

Fig 3.4: Schermata di avvio Rviz

3.4 Considerazioni progettuali

C'è però da considerare un fatto. Il lavoro svolto in questa tesi è di pura simulazione vale a dire che Gazebo, tramite opportuni plugins che si vedranno in seguito, pubblica i dati relativi alla posizione del robot e quindi dell'odometria rispetto ad un sistema di riferimento assoluto che è l'origine dell'ambiente virtuale di Gazebo e pubblica i dati sulla posizione degli ostacoli nella mappa recepiti dal sensore lidar.

In pratica Gazebo funge da nodo publisher che pubblica tali informazioni e vengono lette da Rviz che le mostra su schermo.

Ovviamente nella realtà non ci sarà Gazebo a produrre questi dati ma dovrà essere creato e programmato un nodo che pubblicherà un topic il quale conterrà la posizione del robot nel mondo reale, cioè i dati odometrici delle ruote, e un nodo che pubblicherà un topic contenente i dati del sensore RPLidar A1.

Per quanto riguarda quest'ultimo nodo, in ROS è già stato creato un pacchetto chiamato **"rplidar_ros"** che gestisce la scansione degli RPLidar versioni A1, A2, e A3 della Slamtec. Tramite questo pacchetto si può avviare un file launch e visualizzare su Rviz i dati del sensore tramite il relativo topic di scansione.

Quindi nel passaggio da simulazione a realtà non ci sarà bisogno di Gazebo e Rviz prenderà le informazioni riguardanti il modello reale del robot.

3.5 Workspace

L'ambiente di lavoro e di sviluppo della simulazione del robot è all'interno di una cartella denominata "**amr_description**", situata a sua volta dentro la cartella "**src**" (source) e ancora dentro la directory definitiva "**catkin_ws**" che contiene il workspace ROS dell'intero progetto, comprensivo di quello creato dallo studente precedente.

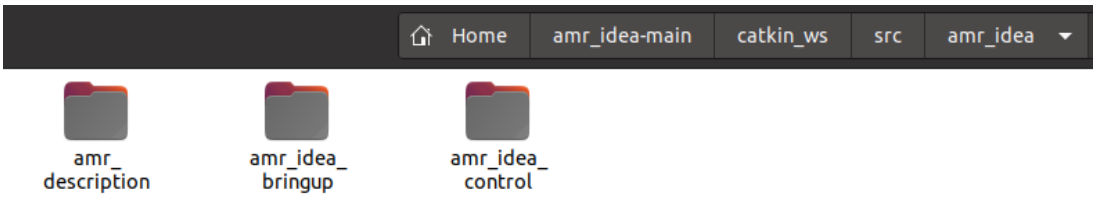

Fig 3.5: Cartella amr_description all'interno del workspace

La directory "amr_description" contiene diversi file di configurazione per la simulazione del robot:

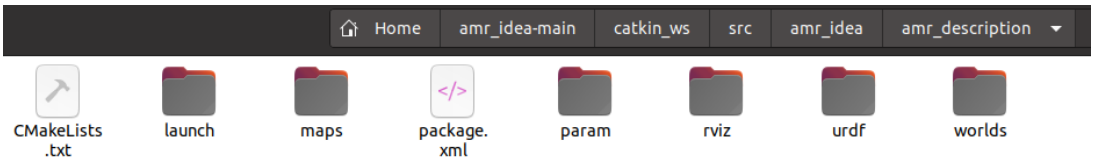

Fig 3.6: Contenuto cartella amr_description

Facendo un elenco molto rapido del contenuto di queste cartelle possiamo dire che:

- la cartella "**launch**" contiene i vari file launch per avviare diversi nodi, tra cui Gazebo, Rviz, l'algoritmo Slam usato per mappare l'ambiente, e il file di navigazione autonoma.
- la cartella "**maps**" contiene le mappe create dall'algoritmo Slam implementato.
- la cartella "**param**" contiene vari parametri per il metodo di navigazione autonoma.
- la cartella "**rviz**" contiene il file di descrizione del modello virtuale del robot con una serie di impostazioni salvate per essere visualizzato appunto su Rviz.
- la cartella "**urdf**" contiene i file di descrizione del robot, ovvero dimensioni, ruote, sensore, posizioni dei giunti che verrano poi visualizzati su Gazebo.
- la cartella "**worlds**" contiene i file ambiente di Gazebo nel quale il robot si muoverà.

Questi contenuti verranno comunque spiegati man mano che si procederà con i capitoli.

3.6 URDF (Unified Robot Description Format)

L'**Urdf**, acronimo di **"Unified Robot Description Format"**, è un file Xml che descrive le caratteristiche fisiche di un robot e degli elementi che lo compongono (telaio, ruote, sensori ecc.). Per essere visualizzato correttamente su Gazebo, le componenti dell'Urdf vengono racchiusi da dei tag che permettono di definire il "ruolo" di quella componente e le sue caratteristiche.

Ecco alcuni tag che vengono utilizzati nell'Urdf:

- **<robot>**: è il tag principale che racchiude tutte le componenti del robot.
- **<sensor>**: è il tag utilizzato per definire le caratteristiche di un sensore montato sul robot, ad esempio una camera 3D o un sensore laser (lidar) come in questo caso.
- **<link>**: è il tag usato per indicare una componente rigida del robot con proprietà visive, inerziali e di collisione. I diversi link che compongono il robot sono collegati tra loro attraverso i giunti formando come una struttura ad albero di interconnessione. Infatti, i link possono essere di due tipi: **parent link** e **child link**. Ogni child link è collegato al suo parent link. Questa cosa è fondamentale per creare un modello di robot consistente e visualizzabile correttamente su Gazebo e Rviz.
- **<joint>**: è il tag per definire un giunto del robot che collega due link tra loro.

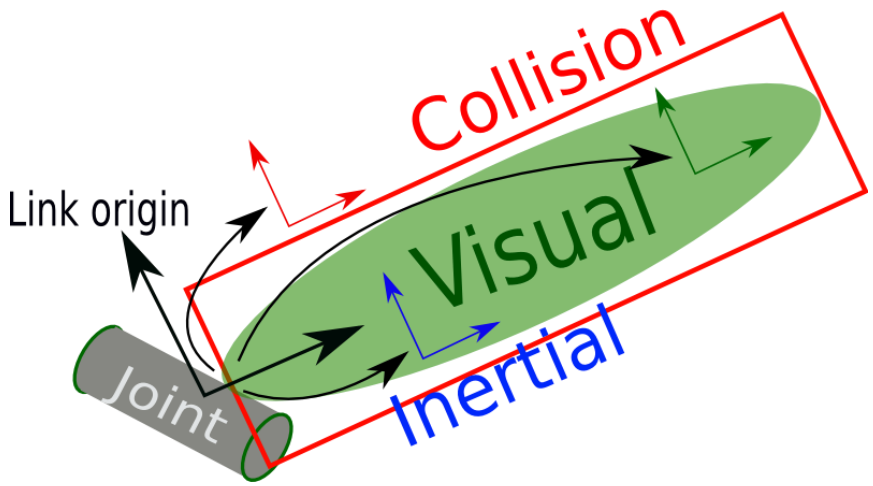

Fig 3.7: Grafico di connessione giunto del robot con un link

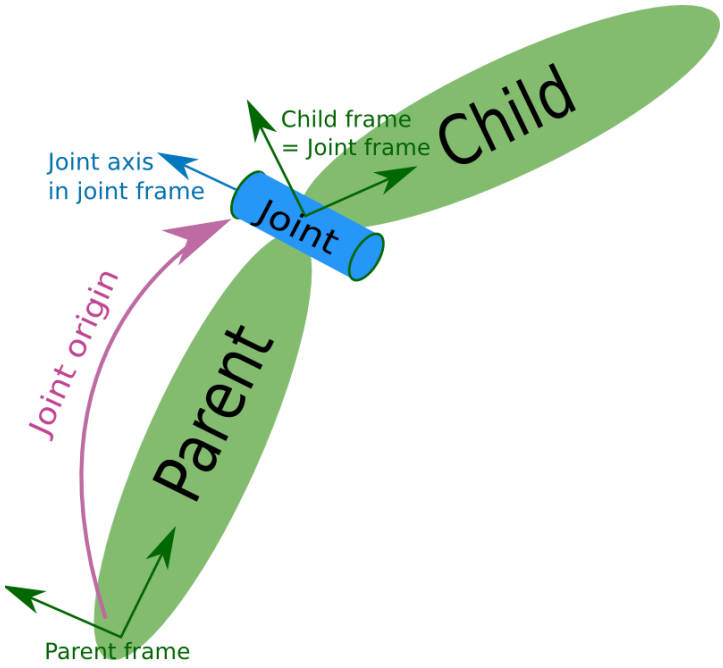

Fig 3.8: Grafico di connessione parent link e child link

Ogni link può inoltre contenere al suo interno altri tag come <inertial>, <origin>, <geometry>, <collision>, <visual> per definire le proprietà inerziali

del link, la posizione d'origine del link, le proprietà visive e di collisione con gli oggetti.

Nell'immagine successiva è rappresentato uno schema di collegamento tra il link del corpo principale del robot (parent) e i link delle ruote (child) adottato per la realizzazione del modello virtuale dell'amr in questo lavoro.

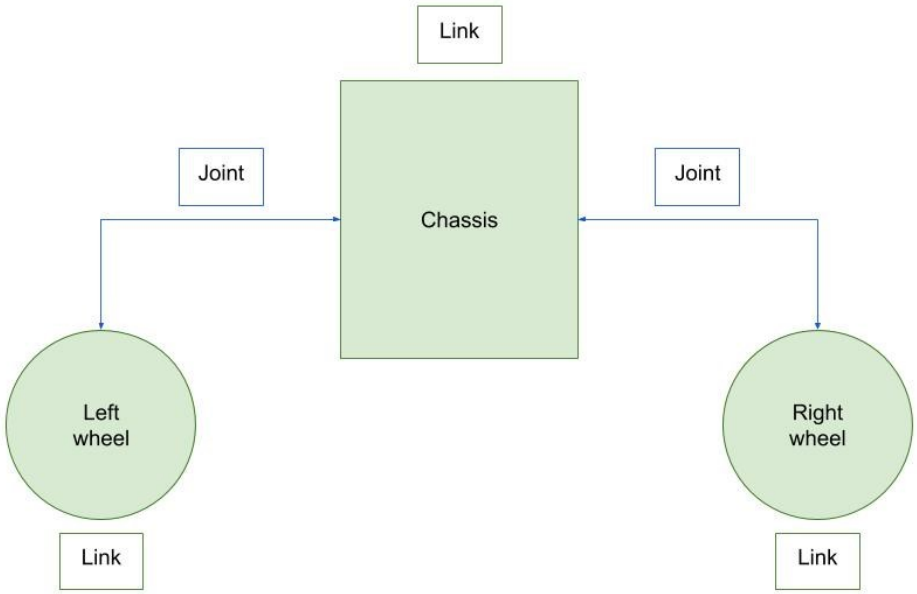

Fig 3.9: Schema di collegamento tra link telaio del robot (chassis) e link delle ruote

3.6.1 Libreria Tf

Un ruolo molto importante nel sistema di connessione ad albero tra i vari link del robot, lo svolge un pacchetto ROS "tf".

Tf è un pacchetto che permette all'utente di tenere traccia nel tempo delle coordinate di più link (anche chiamati frame). Un robot, infatti, può essere composto da diversi link come telecamere, sensori, braccia ecc. Questo

pacchetto stabilisce una relazione tra le coordinate (x,y,z) dei frame in una struttura ad albero. Ad esempio, supponiamo di avere un link relativo alla base del robot (cioè il corpo principale) e un link di un sensore posizionato ad una certa distanza da quello della base. Se rispetto al link del sensore arrivano i dati di scansione dell'ambiente circostante e vogliamo che il robot li possa evitare, allora servirà un sistema di trasformazione di coordinate da quelle del frame del laser a quello della base. Questa operazione è gestita da tf.

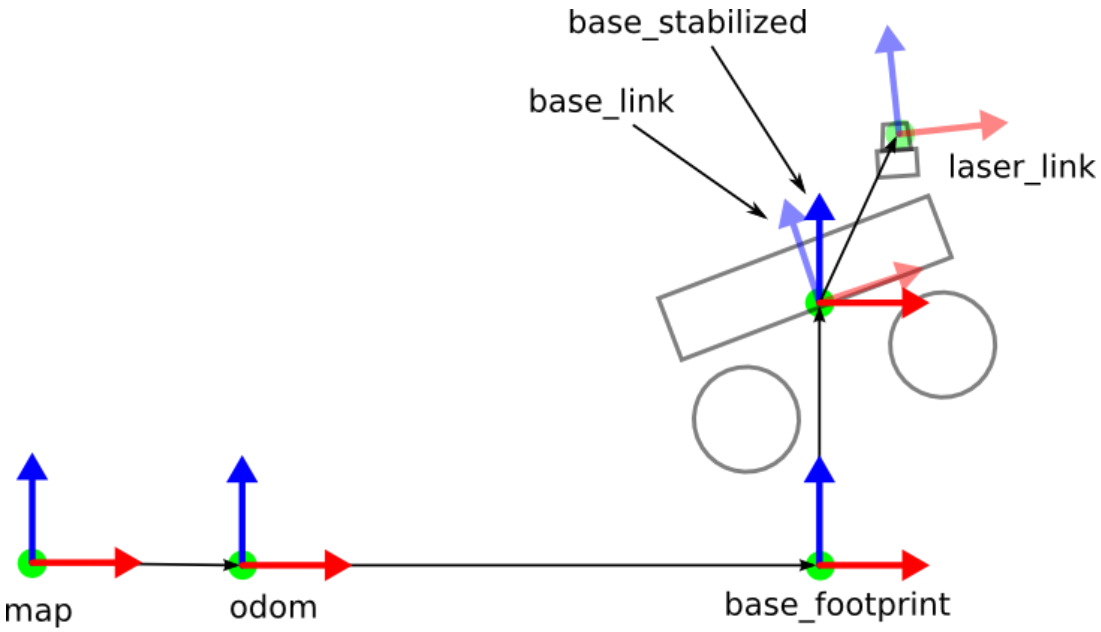

Fig 3.10: Grafico di collegamento tra le coordinate dei link

Nell'immagine sopra:

- la freccia rossa indica l'asse x, quella blu l'asse z mentre l'asse y è entrante
- il frame "map" indica il frame di riferimento alla mappa ambientale in cui il robot si muove. È un frame fisso ed è un punto arbitrario nella mappa

- il frame “odom” è un frame fisso che rappresenta l’origine del sistema di riferimento del robot, ovvero il punto di partenza.
- il frame “base_footprint” è un frame che ha origine direttamente sotto il centro del robot. Come il robot si muove, cambiano le coordinate x e y mentre z rimane 0
- il frame “base_link” sta nella parte centrale della base del robot, ha le stesse coordinate x, y di “base_footprint”, ma diverso valore di z

È importante inoltre definire i link parent e child (genitore e figlio) perché tf presuppone che le trasformazioni di coordinate vengano fatte dal genitore al figlio.

Di seguito è riportato lo schema ad albero di collegamento tra i vari frame del robot, registrato in un certo istante di tempo tramite il comando da terminale “**rosrun tf view_frames**”.

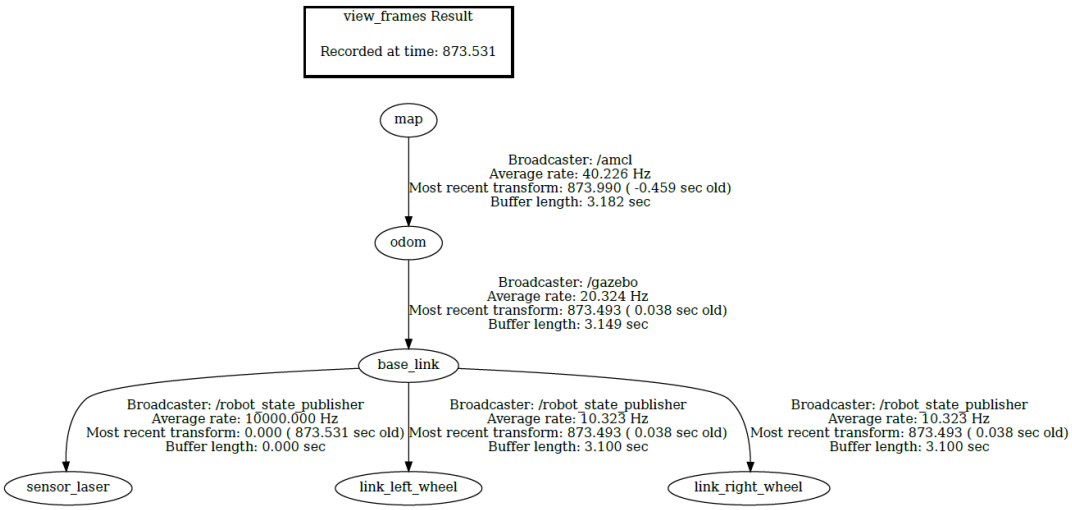

Fig 3.11: Collegamenti tra i frame del robot (albero tf)

3.7 Codice

All'interno della cartella "urdf" ho creato il file "**amr.xacro**" che rappresenta il file urdf del robot.

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

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="amr_robot">

  <xacro:include filename="$(find amr_description)/urdf/materials.xacro"/>
  <xacro:include filename="$(find amr_description)/urdf/amr.gazebo"/>



  <link name="base_link">
    <!-- pose and inertial -->
    <pose>0 0 0.1 0 0 0</pose>
    <inertial>
      <mass value="5"/>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <inertia ixx="0.0395416666667" ixy="0" ixz="0" iyy="0.106208333333" iyz="0" izz="0.106208333333"/>
    </inertial>
    <!-- body -->
    <collision name="collision_chassis">
      <geometry>
        <box size="0.8 0.5 0.22"/>
      </geometry>
    </collision>
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0.03"/>
      <geometry>
        <box size="0.8 0.5 0.22"/>
      </geometry>
      <material name="blue"/>
    </visual>
```

Fig 3.12: Codice iniziale amr.xacro

Nel codice ho chiamato "**base_link**" il nome del link in riferimento al corpo principale del robot che rappresenta il telaio. È di dimensioni 50×80×22 cm, esattamente le stesse del robot reale.

Le ruote destra e sinistra sono state dimensionate rispetto alle dimensioni originale delle ruote brushless motorizzate, ovvero 25 cm di diametro e 5 cm di spessore.

```
<link name="link_left_wheel">
    <inertial>
      <mass value="0.2"/>
      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
      <inertia ixx="0.000526666666667" ixy="0" ixz="0" iyy="0.000526666666667" iyz="0" izz="0.001"/>
    </inertial>
    <collision name="link_left_wheel_collision">
      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
      <geometry>
        <cylinder length="0.05" radius="0.125"/>
      </geometry>
    </collision>
    <visual name="link_left_wheel_visual">
      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
      <geometry>
        <cylinder length="0.05" radius="0.125"/>
      </geometry>
    </visual>
</link>
```

Fig 3.13: Link ruota sinistra

```
<link name="link_right_wheel">
    <inertial>
      <mass value="0.2"/>
      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
      <inertia ixx="0.000526666666667" ixy="0" ixz="0" iyy="0.000526666666667" iyz="0" izz="0.001"/>
    </inertial>
    <collision name="link_right_wheel_collision">
      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
      <geometry>
        <cylinder length="0.05" radius="0.125"/>
      </geometry>
    </collision>
    <visual name="link_right_wheel_visual">
      <origin rpy="0 1.5707 1.5707" xyz="0 0 0"/>
      <geometry>
        <cylinder length="0.05" radius="0.125"/>
      </geometry>
    </visual>
</link>
```

Fig 3.14: Link ruota destra

```
<joint name="joint_left_wheel" type="continuous">
  <origin rpy="0 0 0" xyz="0 0.24 0.01499"/>
  <child link="link_left_wheel"/>
  <parent link="base_link"/>
  <axis rpy="0 0 0" xyz="0 1 0"/>
  <limit effort="10000" velocity="1000"/>
  <joint_properties damping="1.0" friction="1.0"/>
</joint>
```

Fig 3.15: Giunto ruota sinistra

```
<joint name="joint_right_wheel" type="continuous">
  <origin rpy="0 0 0" xyz="0 -0.24 0.01499"/>
  <child link="link_right_wheel"/>
  <parent link="base_link"/>
  <axis rpy="0 0 0" xyz="0 1 0"/>
  <limit effort="10000" velocity="1000"/>
  <joint_properties damping="1.0" friction="1.0"/>
</joint>
```

Fig 3.16: Giunto ruota destra

Come si può notare, i giunti delle ruote collegano il child link (link della relativa ruota) al parent link, cioè il base_link. Inoltre, i giunti sono definiti di tipo "continuous" il che significa che le ruote gireranno attorno al proprio asse in modo continuo.

3.7.1 Plugin di Gazebo

In testa al codice del file "amr.xacro" sono stati aggiunti due header file: "**materials.xacro**" e "**amr.gazebo**".

Il file "**materials.xacro**" contiene solamente delle definizioni di colori in formato RGB che si possono applicare alle varie componenti del robot.

Il file "**amr.gazebo**" invece, contiene due **plugin** cioè pezzi di codice che sono integrati al file dell'Urdf e che vengono compilati per simulare delle funzionalità aggiuntive del robot. Ad esempio, in questo caso ho usato un plugin che fornisce un controller per robot a guida differenziale che mi ha permesso di controllare tramite tastiera i movimenti del robot e la velocità delle ruote.

Ho usato poi, un plugin che simula i raggi laser del lidar e che trasmette su un particolare topic i dati del sensore acquisiti dall'ambiente Gazebo e che verranno visualizzati su Rviz.

```
<plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_controller">
      <legacyMode>false</legacyMode>
      <alwaysOn>true</alwaysOn>
      <updateRate>20</updateRate>
      <leftJoint>joint_left_wheel</leftJoint>
      <rightJoint>joint_right_wheel</rightJoint>
      <wheelSeparation>0.48</wheelSeparation>
      <wheelDiameter>0.25</wheelDiameter>
      <torque>0.1</torque>
      <commandTopic>cmd_vel</commandTopic>
      <odometrySource>world</odometrySource>
      <odometryTopic>odom</odometryTopic>
      <odometryFrame>odom</odometryFrame>
      <publishWheelTF>false</publishWheelTF>
      <publishOdom>true</publishOdom>
      <publishOdomTF>true</publishOdomTF>
      <publishTf>1</publishTf>
      <publishWheelJointState>false</publishWheelJointState>
      <robotBaseFrame>base_link</robotBaseFrame>
</plugin>
```

Fig 3.17: Plugin controllo a guida differenziale

Come si può vedere nell'immagine 3.17, nel plugin di guida differenziale sono definiti dei tag attraverso i quali Gazebo configura l'assetto del robot per la guida.

I tag più importanti sono:

- **<leftjoint>** e **<rightjoint>**, usati per definire i giunti delle ruote destra e sinistra **joint_left_wheel** e **joint_right_wheel**.
- **<commandTopic>**, dove viene dichiarato il topic attraverso il quale si invieranno i dati sulla velocità delle ruote. Questo argomento è

fondamentale per comandare tramite tastiera il robot e farlo muovere in diverse direzioni. Il topic per la velocità è nominato "**cmd_vel**".

- **<odometryTopic>** e **<odometryFrame>** sono tag che racchiudono il topic "**odom**" che contiene i dati di odometria del robot e della sua posizione.
- **<publishOdom> e <publishOdomTf>** servono a Gazebo per pubblicare il topic "odom".
- **<publishTf>** serve per pubblicare la trasformazione del sistema di coordinate tra i frame.
- **<robotBaseFrame>** indica il link principale di riferimento dell'intera struttura del robot rispetto al quale calcolare l'odometria.

```
<gazebo reference="sensor_laser">
    <sensor type="ray" name="rplidar_a1">
      <pose>0 0 0 0 0 0</pose>
      <visualize>true</visualize>
      <update_rate>40</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>720</samples>
            <resolution>1</resolution>
            <min_angle>-3.14159</min_angle>
            <max_angle>3.14159</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.15</min>
          <max>12.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_laser.so">
        <topicName>/scan</topicName>
        <frameName>sensor_laser</frameName>
      </plugin>
    </sensor>
</gazebo>
```

Fig 3.18: Plugin sensore laser

Per quanto riguarda il plugin del sensore laser, ho cercato di simulare nel modo più fedele possibile il sensore RPLidar A1. Al plugin viene passato, come riferimento, il link "sensor_laser" precedentemente osservato, per indicare che tutto ciò che seguirà all'interno del plugin verrà applicato a quel link. Riferendomi alle specifiche, ho impostato un range angolare di 360° tramite i tag **<min_angle>** e **<max_angle>** dove gli angoli sono in radianti.

Ho impostato un range di distanza minimo di 15 centimetri e massimo di 12 metri. Il tag **<visualize>** contiene una variabile booleana che se impostata a "true", come nel mio caso, consente di vedere su Gazebo i raggi laser di colore blu che partono dal sensore e irraggiano l'ambiente circostante. Infine, il plugin pubblica i dati del sensore su un topic chiamato "**scan**". Questo topic sarà sottoscritto poi dall'algoritmo slam che leggerà i dati di scansione e creerà una mappa circostante al robot. Ma questo si vedrà in seguito.

3.8 Robot nell'ambiente virtuale

Una volta realizzato il codice mostrato in precedenza, ho creato il file launch "**gazebo_rviz.launch**" che avvia diversi nodi ROS, tra cui Rviz e Gazebo per far visualizzare su schermo il modello del robot virtuale.

```
<?xml version="1.0"?>
<launch>
    <param name="robot_description" command="$(find xacro)/xacro '$(find amr_description)/urdf/amr.xacro'"/>

    <!-- Send joint values -->
    <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher"/>

    <!-- Combine joint values -->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"/>

    <!-- Show in Rviz -->
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find amr_description)/rviz/robotmodel.rviz"/>

    <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="false"/>
    <arg name="gui" value="true"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="recording" value="false"/>
    <arg name="headless" value="false"/>
    <arg name="world_name" value="$(find amr_description)/worlds/willow_garage.world"/>
    </include>

    <node pkg="gazebo_ros" type="spawn_model" name="spawn_model" output="screen" args="-urdf -param
robot_description -model amr_robot"/>
</launch>
```

Fig 3.19: File launch gazebo_rviz

Innanzitutto viene definito un parametro "**robot_description**" che contiene il file urdf del robot ovvero il file "amr.xacro".

Vengono avviati poi due nodi: "**robot_state_publisher**" e "**joint_state_publisher**".

Joint state publisher legge il parametro "robot_description" che contiene l'urdf e pubblica le coordinate dei giunti del robot in un topic denominato "**joint_states**".

Robot state publisher utilizza l'URDF specificato dal parametro "robot_description" e le posizioni dei giunti dal topic "joint_states" per calcolare la cinematica di avanzamento del robot e pubblicare i risultati tramite tf.

Viene poi avviato il nodo Rviz passandogli come argomento un file di configurazione del modello del robot chiamato "**robotmodel**" e salvato nella cartella "rviz", in modo da poter caricare direttamente l'urdf del robot e di visualizzarlo. Senza questo passaggio, altrimenti, all'avvio di Rviz non si vedrebbe subito il modello del robot ma bisognerebbe, manualmente attraverso l'interfaccia grafica, far leggere a Rviz il parametro "robot_description" così da poterlo visualizzare.

Si lancia poi Gazebo e in particolare gli viene passato, nell'argomento "**world_name**", un ambiente virtuale già disponibile su Gazebo e preconfigurato che ho salvato come "**willow_garage.world**" nella cartella "**worlds**".

Infine, si avvia un nodo per far mostrare il robot nell'ambiente Gazebo, passandogli come parametri l'urdf del robot e riuscendolo a vedere su schermo.

```
gabriele@gabriele-VirtualBox:~$ roslaunch amr_description gazebo_rviz.launch
... logging to /home/gabriele/.ros/log/a0f3b5e2-a403-11ed-b94b-17e3239c44a4/roslaunch-gabriele-Virt
ualBox-3193.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://gabriele-VirtualBox:38111/

SUMMARY
========

PARAMETERS
 * /gazebo/enable_ros_network: True
 * /robot_description: <?xml version="1....
 * /rosdistro: noetic
 * /rosversion: 1.15.15
 * /use_sim_time: True

NODES
  /
    gazebo (gazebo_ros/gzserver)
    gazebo_gui (gazebo_ros/gzclient)
    joint_state_publisher (joint_state_publisher/joint_state_publisher)
    robot_state_publisher (robot_state_publisher/robot_state_publisher)
    rviz (rviz/rviz)
    spawn_model (gazebo_ros/spawn_model)
```

Fig 3.20: Avvio da terminale gazebo_rviz.launch

Togliendo la riga di codice in cui viene caricato l'ambiente "willow_garage.world", il robot viene comunque visualizzato ma nel mondo vuoto di Gazebo in questo modo:

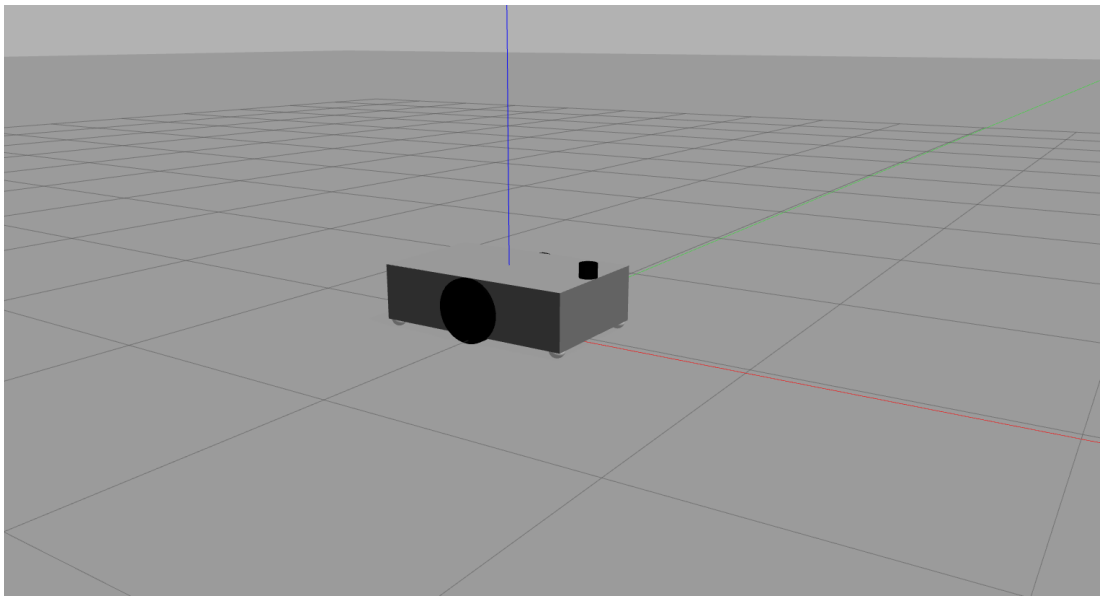

Fig 3.21: Modello di simulazione del robot su Gazebo

Mentre su Rviz è visualizzato in questo modo:

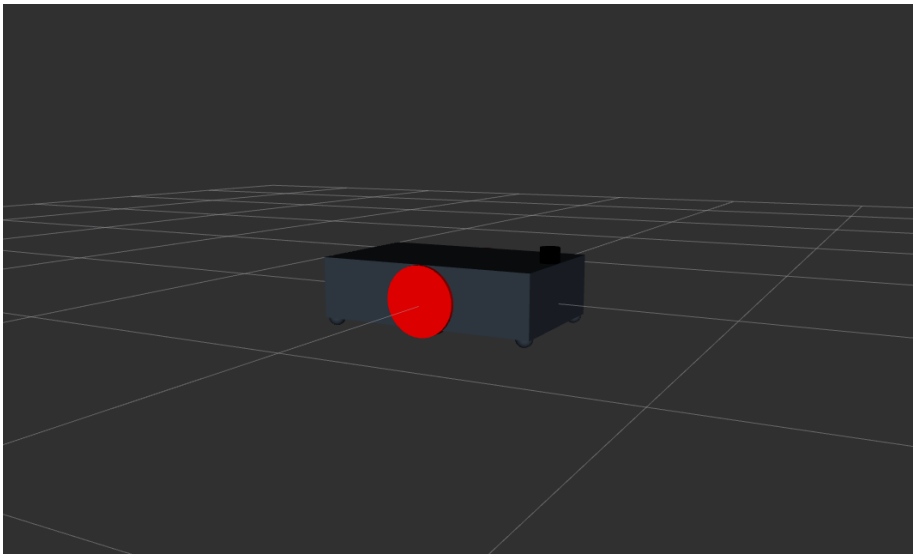

Fig 3.22: Modello di simulazione del robot su Rviz

Se si pongono degli ostacoli su Gazebo, i raggi laser del sensore posto in cima al robot vengono riflessi dagli oggetti ed il plugin pubblica il topic "scan" contenente i dati sensoristici mentre Rviz si sottoscrive a tale argomento facendo vedere su schermo la forma degli ostacoli tramite piccoli quadratini rossi.

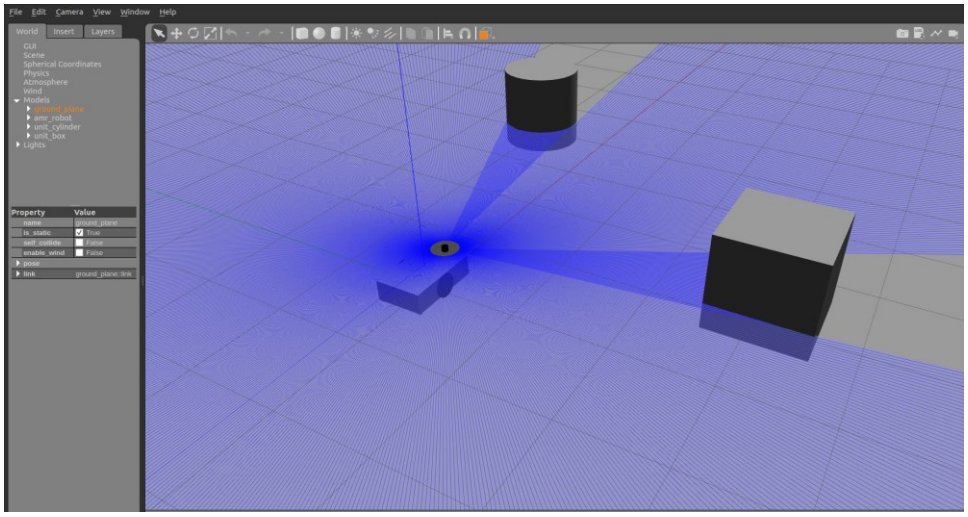

Fig 3.23: Oggetti disposti su Gazebo e colpiti dai raggi laser

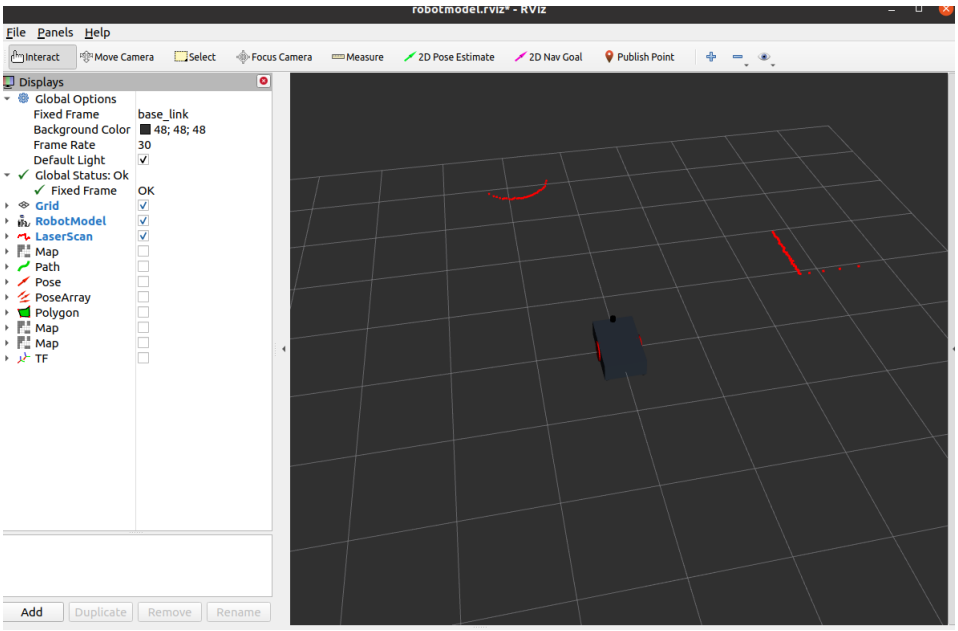

Fig 3.24: Dati sensore laser visualizzati su Rviz

Di seguito le immagini del robot nell'ambiente "willow_garage.world"

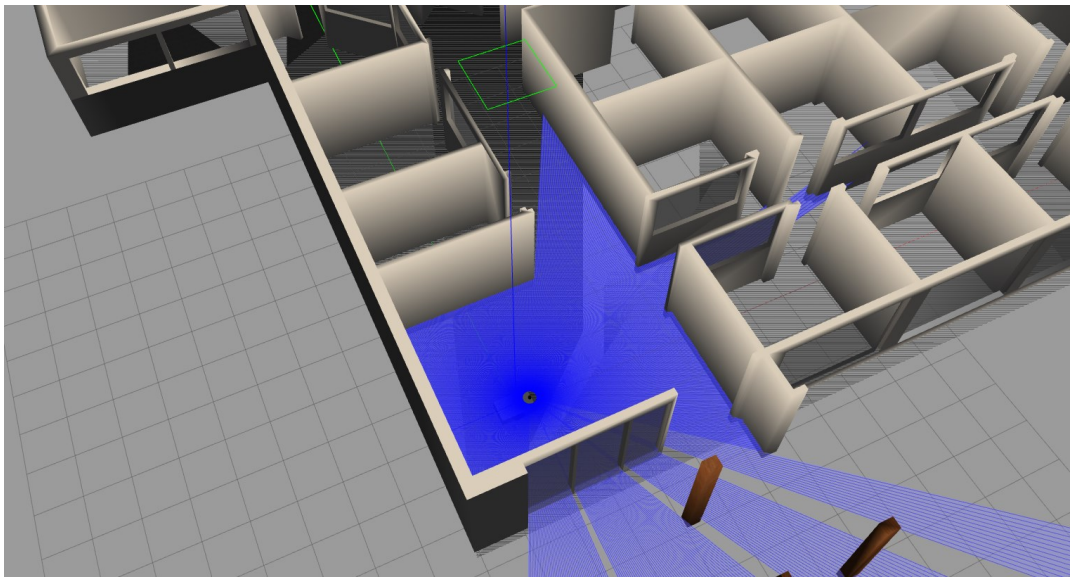

Fig 3.25: Robot nella posizione iniziale nell'ambiente

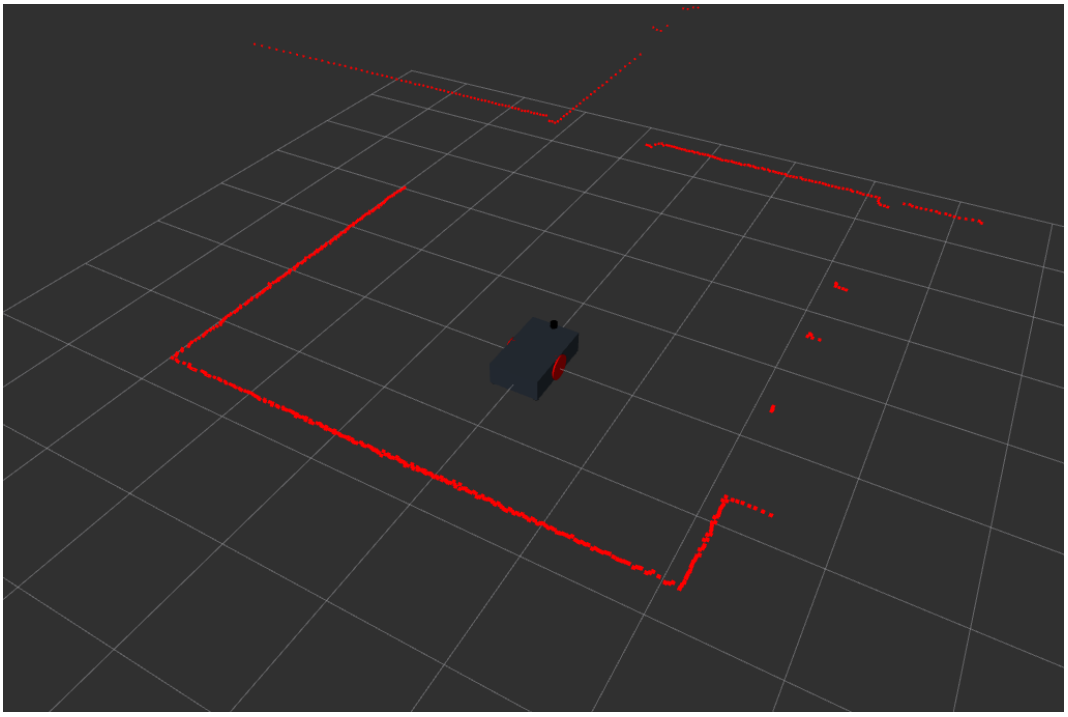

Fig 3.26: Dati del sensore visualizzati su Rviz

Per manovrare il robot da tastiera e controllare la sua velocità, viene in aiuto un pacchetto ROS che è stato scaricato, di nome "**teleop_twist_keyboard**". Esso contiene un nodo il cui codice sorgente è scritto in Python e permette di controllare tramite tastiera del proprio PC la velocità del robot, pubblicando il topic "cmd_vel" nel quale vengono inviati messaggi di velocità. Questo argomento viene poi sottoscritto da Gazebo ed in particolare dal plugin di guida differenziale attraverso il tag <commandTopic>.

```
gabriele@gabriele-VirtualBox:~$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py

Reading from the keyboard  and Publishing to Twist!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5       turn 1.0
```

Fig 3.27: Avvio da terminale del nodo teleop_twist_keyboard.py

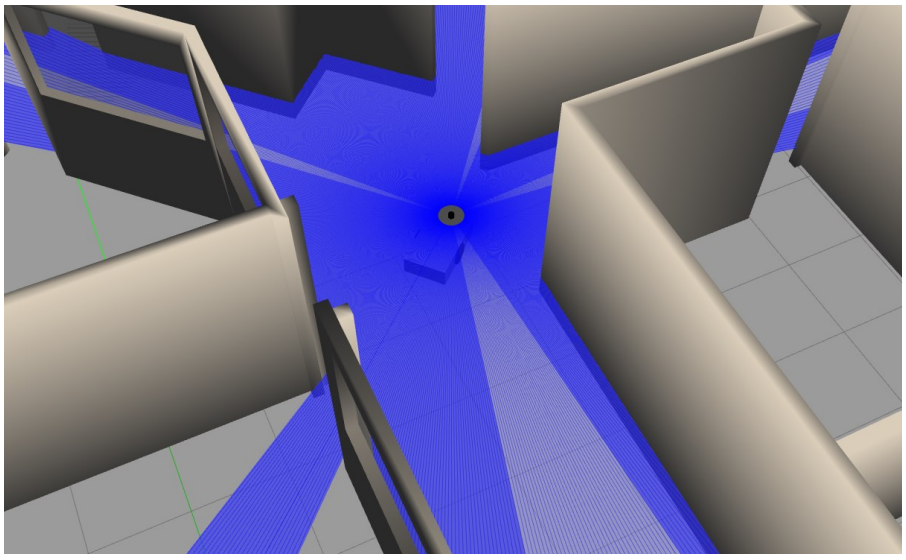

Fig 3.28: Movimento da tastiera del robot

Capitolo

4 Localizzazione e mapping ambientale

4.1 Tipi di localizzazione

Per localizzare un robot all'interno di un ambiente bisogna definire due **sistemi di riferimento**: uno **assoluto** e uno **relativo** al robot. La posizione generica di un robot che si muove nel piano è una terna del tipo (x, y, θ) con θ angolo di differenza di allineamento tra i due sistemi di riferimento. All'istante iniziale, la terna (x_0, y_0, θ_0) del sistema assoluto e la terna (x_1, y_1, θ_1) di quello relativo del robot, coincidono. Localizzare il robot, quindi, significa tenere traccia del suo sistema di coordinate all'aumentare del tempo rispetto al sistema di riferimento globale che rimane fisso.

Ci sono due tipi di localizzazione: **relativa** e **assoluta**.

- **Localizzazione relativa**: è un metodo di localizzazione che si basa sulla misurazione dell'incremento delle grandezze fornite dai sensori propriocettivi del robot (velocità, accelerazione, direzione ecc.). Nota la posizione del veicolo in un certo istante di tempo, con la lettura degli incrementi delle grandezze si stima la nuova posizione sfruttando relazioni cinematiche usando ad esempio tachimetri, accelerometri, encoder. Il problema di questa tipologia risiede nel fatto che questi tipi di grandezze, all'aumentare del tempo, tendono ad avere un errore. Quindi sommando questi errori, l'errore assoluto cresce notevolmente man mano che il robot si sposta e aumenta conseguentemente l'incertezza sulla stima della posizione.

- **Localizzazione assoluta**: è un metodo di localizzazione che si basa sulla lettura dei dati provenienti da sensori esterocettivi come laser o telecamere. Il problema di questo approccio è che dipende fortemente dall'ambiente in cui sono effettuate le misure. Un cambiamento dei parametri ambientali può portare a letture di misurazioni non consistenti.

Queste tipologie sono utilizzate entrambe e si usa solitamente la localizzazione relativa come metodo base di localizzazione che verrà integrata e corretta da quella assoluta attraverso un sistema di fusione di misure ottenuto, ad esempio, tramite un filtro di Kalman, il quale però non verrà trattato in questo testo.

4.2 Il concetto di SLAM

Nella robotica, la localizzazione e mappatura simultanea "**Simultaneous Localization And Mapping**" (SLAM) è il problema computazionale della costruzione o dell'aggiornamento di una mappa di un ambiente sconosciuto, tenendo contemporaneamente traccia della posizione di un agente al suo interno. Questo si può considerare un paradosso in quanto per riuscire a mappare un ambiente occorre sapere la propria posizione e per conoscere la posizione bisogna capire in che punto si è nell'ambiente. Gli algoritmi Slam vengono utilizzati nella navigazione e nella mappatura robotica come, ad esempio, in auto a guida autonoma, veicoli aerei senza pilota, rover planari e robot domestici.

Avendo una serie di controlli u_t e dei dati di osservazione di sensori o_t calcolati in certi istanti di tempo t, il problema consiste nello stimare lo stato del robot x_t e una mappa ambientale m_t .

Si può formulare il problema dello Slam in questo modo:

$$P\left(m_{(t+1)}, x_{(t+1)} \middle| o_{1:t+1}, u_{1:t}\right) \qquad (4.1)$$

Da questa relazione si deduce che i problemi della stima della mappa e della traiettoria del robot sono accoppiati e che quindi devono essere calcolati nello stesso momento.

Se la mappa dell'ambiente circostante è nota, lo SLAM si riduce ad un solo problema di localizzazione e quindi di calcolare la probabilità a posteriori che il sistema si trovi in una certa posizione.

Se invece, è nota la traiettoria, il problema consiste nello stimare la mappa dell'ambiente data la posa del robot e i dati dei sensori.

4.3 Tipologie di mappe

Gli algoritmi Slam acquisiscono i dati dai sensori, li elaborano stimando la posizione del veicolo e producono delle mappe. Riguardo a queste, se ne possono trovare di sei tipologie:

1. **Mappe a griglia**: rappresentano l'ambiente attraverso celle di occupazione dove ogni cella è una variabile aleatoria che indica la probabilità che, nel punto in cui si trova la cella stessa, ci sia un ostacolo o meno. È molto usata nelle rappresentazioni di mappe 2D oppure 3D.

2. **Mappe topologiche**: rappresentano l'ambiente in modo astratto, sotto forma di cammini connessi e intersezioni.

3. **Mappe di features**: rappresentano l'ambiente estraendo la posizione di punti di riferimento (landmarks).

4. **Mappe semantiche**: mettono in relazione gli oggetti con l'ambiente. Gli oggetti vengono etichettati e riconosciuti. Simile alla mappa topologica ma più dettagliata.

5. **Mappe di aspetto**: sono rappresentate da un grafo in cui nei vertici vengono associate delle immagini dell'ambiente connesse da archi. Ogni arco è rappresentato da un peso. Gli archi con peso più grande connettono immagini con alta probabilità che siano adiacenti tra loro.

6. **Mappe ibride**: sono l'unione di diverse tipologie di mappe.

4.4 Algoritmi SLAM

Ci sono diverse tipologie di algoritmi Slam, ognuno adatto a seconda delle diverse necessità. Per quanto riguarda la mappatura dell'ambiente, si possono dividere in quattro categorie principali:

- **Features-based Slam**: sono algoritmi basati sull'estrazione di landmarks ambientali e che tengono traccia delle coordinate, formando una mappa. In generale non sono algoritmi efficienti in quanto essendo basati su features, possono essere soggetti a perdite di dati.
- **View-based Slam**: sono algoritmi che creano mappe a partire dai dati dei sensori. Questi algoritmi localizzano il robot attraverso un sistema

di scan matching, ovvero calcolano quanto le misure dei sensori combaciano con la mappa più recente dell'ambiente. Quella che creano è una mappa di occupazione a griglia che fornisce informazioni sugli spazi liberi e occupati.

- **Appearance-based Slam**: sono algoritmi che di solito vengono utilizzati con fotocamere e possono essere combinati alle tipologie di Slam precedenti. Essi comparano delle nuove osservazioni ambientali con quelle precedenti per stabilire se un luogo è stato già osservato. Confrontano molte immagini e per questo hanno anche un alto costo computazionale.
- **Polygon-based Slam**: sono usati per il mapping 3D e associano segmenti planari ai features. Producono mappe molto precise e adatte a lavori di alto livello ma sono dispendiose a livello computazionale.

Lo Slam stima il movimento sequenziale, che include un certo margine di errore. L'errore si accumula nel tempo, causando una notevole deviazione dai valori reali. Può anche causare il collasso o la distorsione dei dati delle mappe, rendendo difficili le ricerche successive. Man mano che l'errore si accumula, il punto di partenza e quello di arrivo del robot non corrispondono più. Siamo di fronte a un cosiddetto problema di **"loop closure"**. Il concetto di loop closure in un contesto SLAM è la capacità di un veicolo di riconoscere che un luogo è già stato osservato. Applicando un algoritmo con loop closure, l'accuratezza sia della mappa che della posizione del robot può accrescere. Non è sempre facile da eseguire in quanto ci potrebbero essere simili circostanze strutturali e se il metodo funziona male potrebbe corrompere sia la mappa che la posa del veicolo. Questo concetto è molto importante per un

algoritmo Slam in quanto aiuta a ridurre l'errore cumulativo della posa stimata del robot e a generare una mappa globale coerente.

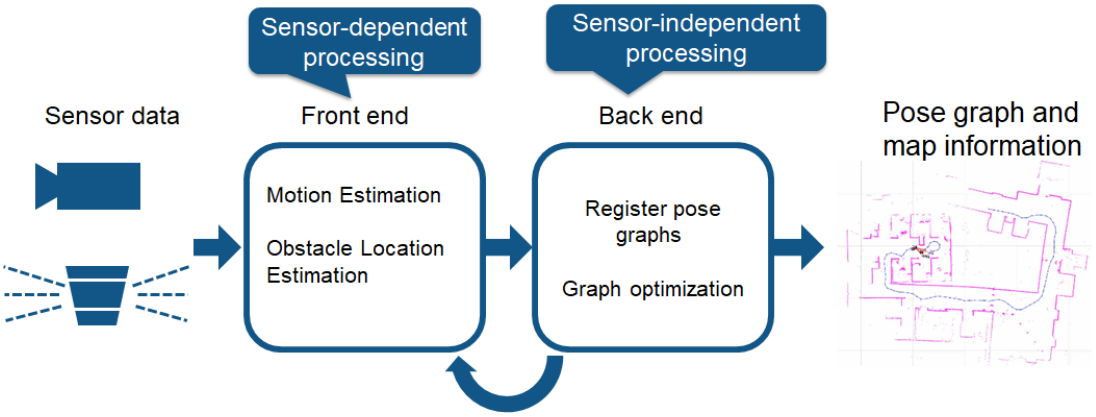

Fig 4.1: Diagramma di funzionamento Algoritmo Slam generico

Per quanto riguarda invece l'elaborazione dei dati uno dei principali algoritmi utilizzati è il **Filtering Slam**.

Filtering Slam è una categoria di algoritmi basati sul filtraggio dei dati. Se il robot usasse solamente i dati grezzi dei sensori, la sua navigazione non sarebbe ottimale perché i dati sono soggetti ad errori dovuti al rumore. È necessario quindi un meccanismo di filtraggio di dati.

Di questi algoritmi fanno parte, ad esempio, il **Particle Filter Slam** e **l'EKF Slam** (**Extended Kalman Filter**) però quest'ultimo non verrà trattato. Questi algoritmi sono utilizzati per stimare la posa corrente del robot e per ottimizzarne l'intera traiettoria.

Per questo progetto è stato utilizzato un algoritmo Slam basato su filtro particellare.

4.4.1 Filtro di Bayes

La posa di un robot è dovuta al suo stato corrente x_t. Quando il sistema entra in un nuovo stato x_{t+1}, dovuto a un comando di controllo del robot u_t, il nuovo stato produrrà una misura z_{t+1}. Un **filtro di Bayes** è un algoritmo che calcola la funzione di **densità di probabilità** per il vettore di stato x_t. Questo è svolto in due step: **predizione** $\overline{bel}(x_t)$ e **correzione** $bel(x_t)$

$$\overline{bel}(x_t) = \int p(x_t | u_t, x_{t-1}) bel(x_{t-1})\, dx_{t-1} \qquad (4.2)$$

$$bel(x_t) = \eta\, p(z_t \mid x_t)\, bel(x_t) \qquad (4.3)$$

dove nella fase di predizione (prima equazione) il nuovo stato x_t è predetto usando solo l'ingresso u_t e lo stato precedente x_{t-1}. La predizione è modificata dalla fase di correzione utilizzando i dati dei sensori per ottenere una migliore stima. Viene introdotto un fattore di scala η per ridimensionare la funzione di densità di probabilità risultante al fine di ottenere la somma integrale.

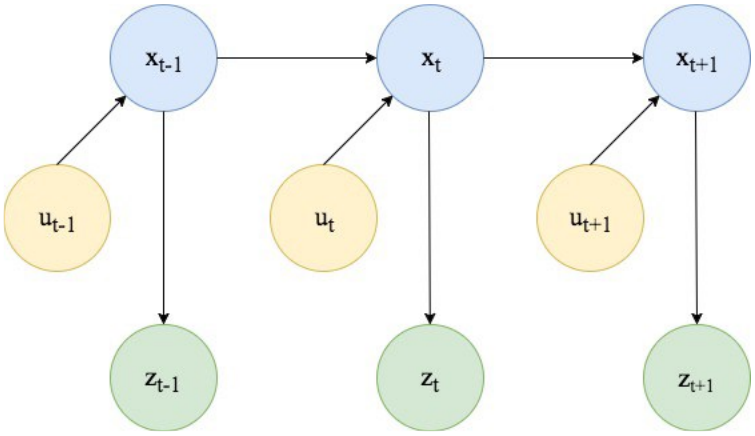

Fig 4.2: Relazione tra stato, misure e ingressi

4.4.2 Filtro particellare

Un **filtro particellare** è un'implementazione del filtro di Bayes, il quale definisce una serie di **M campioni** $x_M(t)$ chiamati anche **particelle**, dove ogni particella rappresenta una stima dello stato futuro del robot. L'insieme delle particelle compongono il vettore delle variabili di stato. Ogni particella è rappresentata da una probabilità di essere la stima verosimile dello stato reale del robot. Sono distribuite nello spazio in modo non uniforme. Se una particella ha un'alta probabilità significa che stima nel modo più corretto il vero stato del robot, cioè la sua posizione. Regioni di spazio con alta probabilità avranno maggiore densità di particelle, regioni con bassa probabilità ne conterranno poche o addirittura nessuna.

Un filtro particellare è costituito da tre passaggi

1) **Campionamento**: l'insieme di particelle viene generato dall'insieme precedente di particelle mediante campionamento da una distribuzione di stato proposta. Di solito, questa distribuzione è un modello di moto odometrico.
2) **Assegnazione peso d'importanza**: viene assegnato un peso a ciascuna particella. Il peso rappresenta quanto una particella si avvicini il più possibile allo stato previsto del robot.
3) **Ricampionamento**: viene utilizzato solo un numero finito di particelle per descrivere la distribuzione e vengono prese quelle con pesi più alti, mentre vengono scartate quelle con pesi più bassi. Poi, dopo il ricampionamento, tutte le particelle sono assegnate con lo stesso peso.

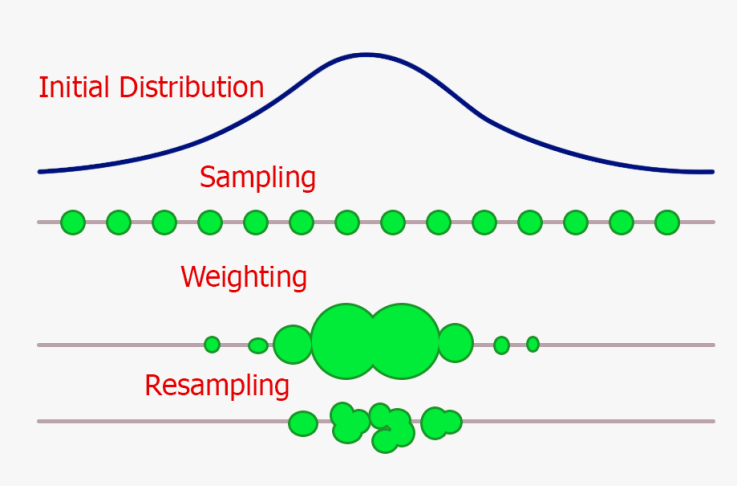

Fig 4.3: Grafico illustrativo funzionamento Particle Filter

4.5 SLAM Gmapping

In ROS ci sono diversi algoritmi Slam già sviluppati, scaricabili mediante pacchetti e pronti per essere installati e configurati, come ad esempio Hector Slam, Core Slam, Cartographer ed ognuno avente caratteristiche proprie. Per questo progetto ho usato l'algoritmo "**Gmapping**".

Ho scelto di utilizzare questo algoritmo in quanto è uno dei più utilizzati in ROS per lo scanning 2D degli ambienti tramite sensori laser, ha un buon sistema di loop closure e a livello di creazione di mappe risulta più che efficiente.

Gmapping è un algoritmo Slam che utilizza un filtro di particelle migliorato "**Rao – Blackwellized**", che permette di eseguire un campionamento sfruttando meno particelle rispetto ai normali filtri particellari ottenendo

comunque la stessa precisione e che simultaneamente scansiona l'ambiente circostante creando una **mappa di occupazione a griglia**.

Il filtro stima la probabilità a posteriori dello stato del robot

$$p(m, x_{1:t} | z_{1:t}, u_{1:t-1}) \qquad (4.4)$$

(dove m è la mappa ambientale), ovvero lo stato del robot in un certo istante data la misura dei sensori congiuntamente all'ingresso dell'istante precedente. Ogni particella rappresenta un possibile stato del robot e gli step del filtro sono quelli visti in precedenza per un filtro particellare:

- **Campionamento**: sono generate un set nuovo di particelle ${x_t}^{(i)}$ dal set di particelle campionato precedentemente.
- **Assegnazione del peso**: ad ogni particella è assegnato un peso ${w_t}^{(i)}$.
- **Ricampionamento**: particelle con peso minore sono scartate, si campiona la distribuzione e poi si assegna lo stesso peso alle particelle.

4.5.1 Gmapping in ROS

Gmapping in ROS è sostanzialmente un pacchetto che contiene un **nodo** chiamato "**slam_gmapping**", il quale include al suo interno il codice sorgente dell'algoritmo, scritto in C++, e un file launch per avviarlo con tutta una serie di parametri personalizzabili.

Quando arriva una scansione, Gmapping controlla se c'è una posa di odometria che viene fornita assieme ad una scansione laser o meno, cercando la trasformazione dal frame "**odom**" a "**base_link**" nei messaggi tf. Tutti i dati di odometria che vengono forniti senza una scansione laser associata, vengono scartati. L'algoritmo è dotato inoltre, da un **filtro di movimento** che viene

utilizzato per determinare se le informazioni provenienti dal sensore lidar devono essere elaborate o no. Se non c'è abbastanza movimento angolare o lineare del robot rispetto alla posizione che aveva nell'istante precedente, le informazioni sensoriali non vengono elaborate, altrimenti si.

Successivamente si attiva il filtro particellare Rao – Blackwellized (RBPF) per stimare lo stato del robot a partire dalle particelle campionate.

Un altro aspetto che influisce notevolmente sulle prestazioni di RBPF è la fase di ricampionamento. Durante il ricampionamento, se una particella viene sostituita con un'altra, la mappa associata alla particella sostituita deve essere copiata. Questo introduce una richiesta computazionale che può essere molto onerosa, dato che la mappa diventa sempre più grande. D'altra parte, il ricampionamento è necessario perché solo un numero finito di particelle rappresenta la distribuzione. Pertanto, è necessario un meccanismo per eseguire il ricampionamento quando ce n'è bisogno. In GMapping, il criterio per eseguire il ricampionamento si basa su una variabile che è una stima di quanto sia buona la distribuzione attuale delle particelle. Questa stima è chiamata "**dimensione effettiva del campione**" indicata come "$\boldsymbol{N_{eff}}$".

Questa variabile è ottenuta tramite la seguente formula:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} (\tilde{w}^i)^2} \qquad (4.5)$$

dove $\tilde{w}^i$ rappresenta il peso normalizzato della particella i-esima.

"N_{eff}" valuta quanto bene l'insieme di particelle si avvicini alla distribuzione di stato del robot. Nell'implementazione di GMapping in ROS, viene eseguita la fase di ricampionamento quando N_{eff} scende sotto un valore soglia pari a $\boldsymbol{N/2}$, con N numero di particelle. Questo riduce notevolmente il rischio di sostituire particelle buone.

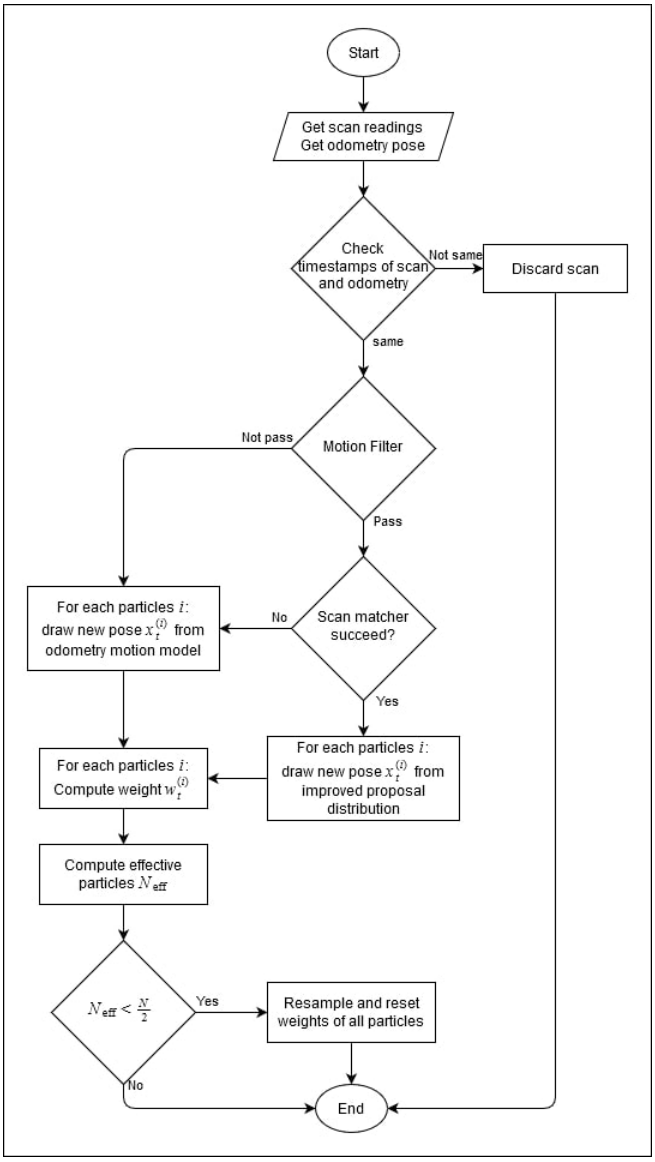

Fig 4.4: Flowchart di Gmapping in ROS

4.6 Gmapping su simulazione

Nell'ambiente di sviluppo ROS è possibile scaricare il pacchetto contenente l'algoritmo in due modi principali: o attraverso i comandi da terminale :"*sudo apt-get install ros-noetic-gmapping*" e "*sudo apt-get install ros-noetic-slam-gmapping*", oppure clonare il rispettivo repository disponibile su Github, nel proprio workspace. In questo caso ho deciso la seconda opzione.

Una volta scaricato il pacchetto, si trova all'interno un file launch che avvia il nodo "slam_gmapping" e che può essere configurato attraverso una serie di parametri. Ho scelto però di lasciare i loro valori predefiniti.

```
<launch>
    <param name="use_sim_time" value="true"/>
    <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen">
      <remap from="scan" to="/scan"/>
      <param name="map_update_interval" value="5.0"/>
      <param name="maxUrange" value="16.0"/>
      <param name="sigma" value="0.05"/>
      <param name="kernelSize" value="1"/>
      <param name="lstep" value="0.05"/>
      <param name="astep" value="0.05"/>
      <param name="iterations" value="5"/>
      <param name="lsigma" value="0.075"/>
      <param name="ogain" value="3.0"/>
      <param name="lskip" value="0"/>
      <param name="srr" value="0.1"/>
      <param name="srt" value="0.2"/>
      <param name="str" value="0.1"/>
      <param name="stt" value="0.2"/>
      <param name="linearUpdate" value="1.0"/>
      <param name="angularUpdate" value="0.5"/>
      <param name="temporalUpdate" value="3.0"/>
      <param name="resampleThreshold" value="0.5"/>
      <param name="particles" value="30"/>
      <param name="xmin" value="-50.0"/>
      <param name="ymin" value="-50.0"/>
      <param name="xmax" value="50.0"/>
      <param name="ymax" value="50.0"/>
      <param name="delta" value="0.05"/>
      <param name="llsamplerange" value="0.01"/>
      <param name="llsamplestep" value="0.01"/>
      <param name="lasamplerange" value="0.005"/>
      <param name="lasamplestep" value="0.005"/>
    </node>
</launch>
```

Fig 4.5: File launch Gmapping

Il significato di ciascuno di questi parametri può essere consultato sul sito ROS Wiki, ma uno dei più importanti è il parametro "**particles**" che indica appunto, il numero di particelle del filtro. Il numero di default di particelle è 30.

Ho creato poi, un semplice file launch denominato "**gmapping.launch**" che avvia sostanzialmente il file launch di Gmapping precedente

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

  <include file="$(find gmapping)/launch/slam_gmapping_pr2.launch">
  </include>

</launch>
```

Fig 4.6: File gmapping.launch

Una volta avviato Rviz e Gazebo con il modello virtuale del robot situato all'interno dell'ambiente, è stato possibile mappare le stanze di "willow_garage.world" lanciando questo file e comandando da tastiera il robot attraverso il nodo "**teleop_twist_keyboard.py**".

Nella pagina successiva, le immagini di elaborazione dei dati dell'algoritmo Gmapping e fase di ricampionamento su terminale.

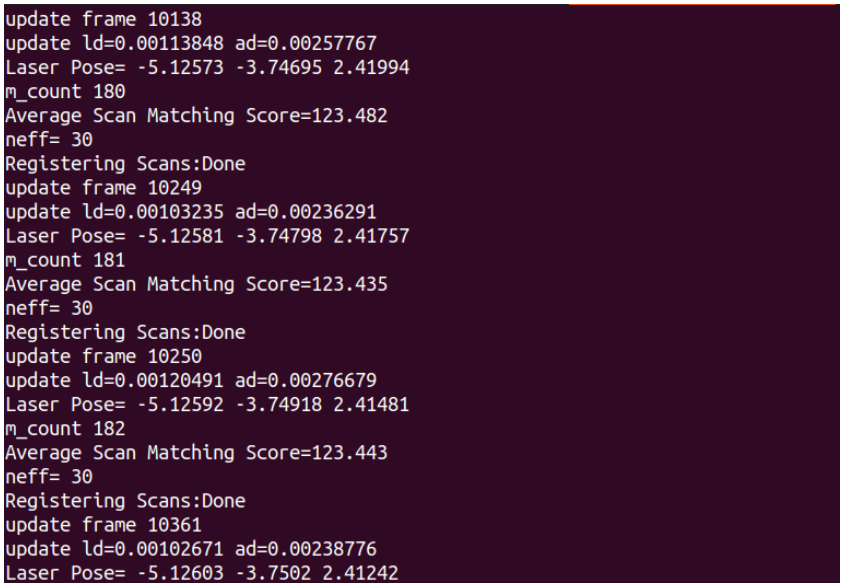

Fig 4.7: Calcolo dati Gmapping

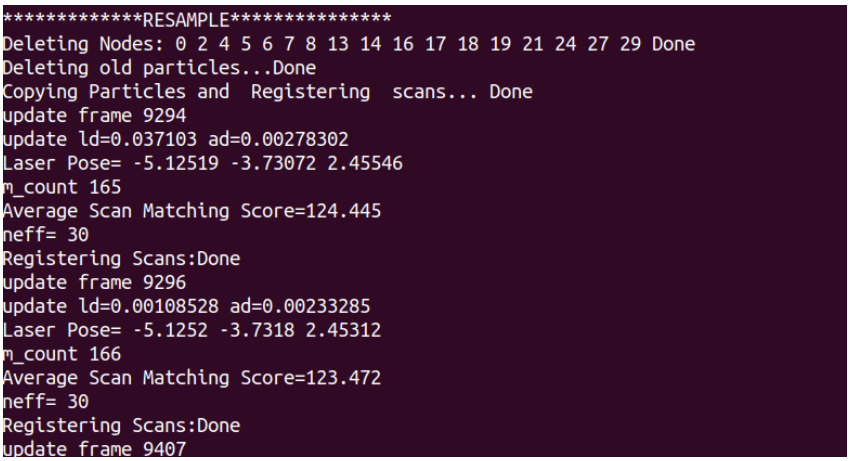

Fig 4.8: Fase di ricampionamento Gmapping

Per costruire la mappa dell'ambiente, il nodo Gmapping legge i dati del sensore laser sottoscrivendosi al topic "**scan**" e al topic "**tf**" pubblicati da Gazebo per la trasformazione di dati dal sensore alla base del robot e dalla base al frame "odom". Successivamente pubblica l'argomento "**map**" che

contiene i dati della mappa ambientale scansionata. Quest'ultimo topic viene poi sottoscritto da Rviz, il quale visualizza su schermo la mappa.

La mappa generata, come già detto, è una mappa di occupazione a griglia cioè rappresentata da celle dove ognuna rappresenta una probabilità che in quel punto ci sia un oggetto od ostacolo. La mappa può essere formata da tre colori

- nero: la cella in questione rappresenta uno spazio in cui c'è un ostacolo.
- grigio chiaro/bianco: la cella indica uno spazio libero in cui non ci sono ostacoli.
- grigio scuro: è una regione di spazio non ancora visitata e quindi sconosciuta.

Fig 4.9: Scansione ambiente tramite Gmapping

Attraverso un pacchetto fornito da ROS chiamato "**map_server**" e contenente un nodo chiamato "**map_saver**", è stato possibile salvare la mappa complessiva scansionata da Gmapping. Il nodo "map_saver ", salva i metadati della mappa in una cartella che ho chiamato "**maps**" nel workspace.

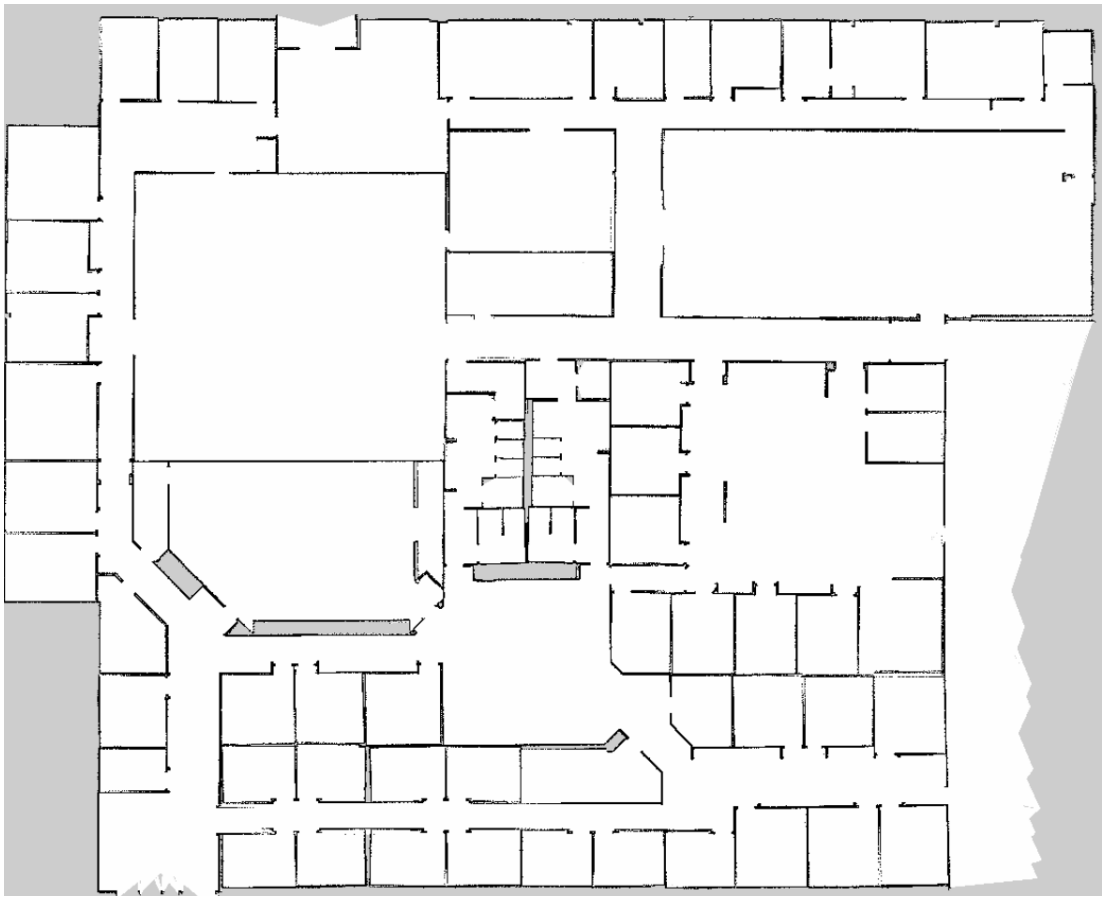

Fig 4.10: Mappa complessiva willow_garage.world

Capitolo

5 Navigazione autonoma

5.1 Pianificazione del percorso

Il problema di pianificazione di un percorso per un veicolo è dovuto al numero dei suoi gradi di libertà. Più essi sono, più ci saranno percorsi possibili da calcolare. In questo caso, il nostro robot si muove in uno spazio 2D e quindi su un piano. Lo scopo è quello di trovare un percorso ottimale, che eviti ostacoli, che abbia distanza minima e che connetta il punto di partenza e il punto di arrivo (obbiettivo). Questo di solito è ottenuto sfruttando un **pianificatore globale**, cioè un algoritmo che utilizza informazioni a priori sull'ambiente per creare il miglior percorso possibile, se esiste, e un **pianificatore locale** che in ogni istante ricalcola il percorso stimato iniziale per evitare eventuali ostacoli dinamici.

5.1.1 Pianificatore globale (global planner)

Il **pianificatore globale** ha bisogno, appunto, di una mappa globale già salvata per decidere il miglior percorso da fare per raggiungere un punto prestabilito.

Ci sono diversi algoritmi per pianificare un percorso, ma uno dei più efficaci e anche quello che usa ROS in uno dei suoi pacchetti, utilizzato per questo progetto, è l'algoritmo di **Dijkstra**.

Dijkstra usa un metodo grafico per risolvere il problema di path planning utilizzando una mappa divisa in celle dove in ogni cella viene assegnato un

nodo con un valore. Questo valore rappresenta il costo di attraversamento dello spazio per raggiungere tale cella.

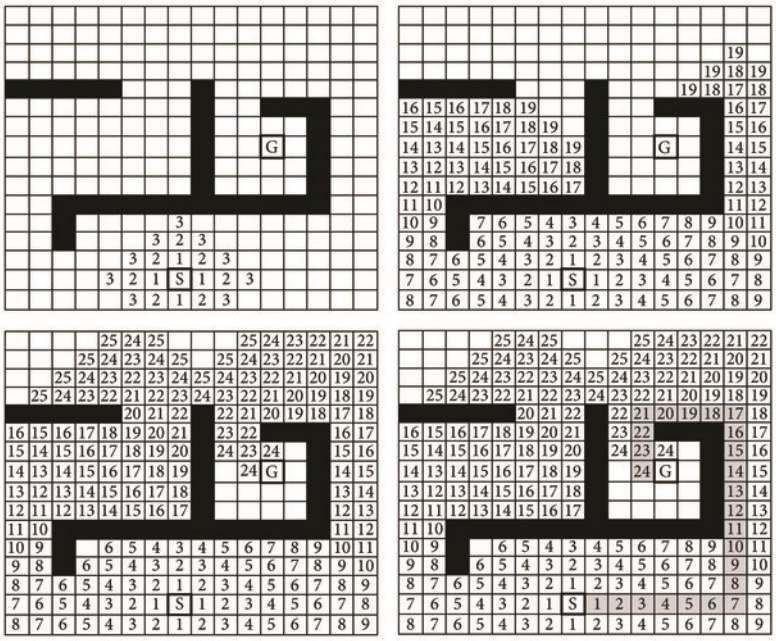

Fig 5.1: Metodo Dijkstra per il path planning

Nella figura 5.1, in alto a sinistra, il veicolo parte della cella "S". Le celle adiacenti a quella di partenza hanno valore 1 il che significa che occorre uno spostamento per raggiungere una di esse. Poi, subito dopo, ci sono quelle adiacenti con valore 2 e poi 3 ecc. Questo si ripete per ogni cella adiacente fino a raggiungere il punto target. La somma dei nodi delle celle con valore minimo rappresenterà il percorso più breve da compiere. Dopo che è stato trovato il percorso globale, tutti i nodi selezionati sono convertiti in posizioni (x, y) nel piano.

5.1.2 Pianificatore locale (local planner)

Il **pianificatore locale** crea nuove traiettorie tenendo conto degli eventuali ostacoli dinamici che si possono presentare nel cammino. Per ricalcolare il percorso, il pianificatore locale crea una mappa ridotta all'ambiente circostante il veicolo e viene aggiornata man mano che il robot si muove. Non può essere utilizzata la mappa intera in quanto i dati dei sensori non raggiungono ogni cella della mappa stessa. Con questo metodo di pianificazione locale è possibile modificare il percorso globale al fine di evitare i possibili ostacoli dinamici presenti, ma cercando comunque di far corrispondere il più possibile le celle della mappa scelte dal pianificatore globale.

5.2 ROS Navigation Stack

Nella piattaforma ROS è presente uno stack di navigazione che permette di far muovere autonomamente un veicolo verso un target desiderato. È il "**ROS Navigation Stack**".

Il ROS Navigation Stack è anch'esso un pacchetto, contenente a sua volta altri pacchetti che servono a far muovere in modo autonomo un robot. Riceve informazioni dall'odometria, dai flussi di dati sensoristici e dalla posizione di un obiettivo da raggiungere all'interno di una mappa emettendo comandi di velocità che vengono inviati alla base mobile.

Il suo concetto di funzionamento può sembrare abbastanza semplice ma non può essere utilizzato su qualsiasi robot e richiede dei prerequisiti.

Ad esempio, il robot sul quale si vuole applicare il ROS Navigation Stack deve innanzitutto essere programmato in ROS, deve disporre di un albero di

trasformazione tf e pubblicare i dati del sensore utilizzando tipi di messaggi corretti. Inoltre, lo stack di navigazione deve essere configurato in base alle caratteristiche della forma e della dinamica del robot, anche perché è stato pensato per robot a trasmissione differenziale. Richiede poi un laser montato sul robot per mappare e localizzarsi.

Il ROS Navigation Stack, come già accennato, è un pacchetto contenentene altri che servono a inviare messaggi di velocità al robot e a farlo muovere autonomamente. Ce ne sono diversi, ma quelli principali che ho utilizzato sono:

- **amcl**
- **move_base**
- **dwa_local_planner**
- **map_server**
- **nav_core**
- **navfn**

5.2.1 Nav_core e Move_base

"**Nav_core**" è un pacchetto ROS che fornisce interfacce comuni per azioni specifiche per la navigazione. Definisce, insieme a "**move_base**", la parte centrale di navigazione. Nav_core definisce delle classi chiamate "**BaseGlobalPlanner**", "**BaseLocalPlanner**" e "**RecoveryBehavior**" che servono a definire rispettivamente: il tipo di pianificatore globale che si utilizza, il pianificatore locale e comportamenti di ripristino del robot in caso il robot sia bloccato o ci siano imprevisti.

I **pianificatori globali** sviluppati in ROS e supportati dalla classe BaseGlobalPlanner di Nav_core sono:

- **global_planner**
- **navfn**
- **carrot_planner**

I **pianificatori locali** supportati dalla classe BaseLocalPlanner sono:

- **base_local_planner**
- **dwa_local_planner**
- **eband_local_planner**
- **teb_local_planner**
- **mpc_local_planner**

"**Move_base** "invece è un pacchetto che, dato un obbiettivo da raggiungere nella mappa, invia i comandi al robot per farlo arrivare al punto desiderato. È di fatto, il nodo principale che fa compiere la navigazione al veicolo e supporta i pianificatori locali e globali definiti prima. Esso fa interagire tra loro il local planner ed il global planner mantenendo due mappe, una relativa al pianificatore locale e una per il globale: "**local_costmap**" e "**global_costmap**". Qualora il robot dovesse avere problemi nel raggiungere il target fissato, move_base eseguirà dei tentativi di recupero.

5.2.2 Amcl

Amcl è un sistema di localizzazione probabilistica per un robot che si muove su un piano bidimensionale.

È l'acronimo di "**Adaptive Monte Carlo Localization**". La localizzazione Monte Carlo è un algoritmo che permette ai robot di localizzarsi utilizzando un **filtro a particelle**, in modo del tutto analogo a quanto visto per l'algoritmo Gmapping. Infatti, i filtri particellari sono anche chiamati metodi "Monte Carlo" e che sfruttano l'inferenza statistica bayesiana. Quindi i procedimenti eseguiti per la stima dello stato del robot sono praticamente gli stessi del filtro di Gmapping (**campionamento**, **assegnazione dei pesi**, **ricampionamento**) e ogni particella rappresenta un possibile stato del robot.

La differenza che c'è con Gmapping è che, mentre Gmapping è un algoritmo Slam che appunto compie sia la parte di localizzazione tramite filtro particellare sia la parte di mappatura dell'ambiente, Amcl compie solo la parte di localizzazione. Ma mentre in Gmapping la parte di localizzazione non viene pubblicata ma è "intrinseca" all'algoritmo, su Amcl invece viene pubblicata la posizione stimata dello stato.

Il pacchetto ROS "**Amcl**" che contiene questo algoritmo, però, ha bisogno di una mappa già salvata per localizzare il robot all'interno, perché si sottoscrive al topic "**map**" che contiene la mappa e anche al topic "**scan**" che contiene i dati del sensore laser e li confronta con la mappa stessa.

Questo si traduce in un problema Slam dove è nota a priori la mappa ambientale ma occorre localizzarsi all'interno di essa. Sostanzialmente quello che si fa è passare all'algoritmo Amcl, la mappa creata da Gmapping.

Il nodo Amcl si sottoscrive ai topic "scan", "map", "tf" e "**initialpose**" cioè la posizione iniziale e conosciuta del robot. Pubblica invece i topic "**amcl_pose**" ovvero la posizione stimata del robot nella mappa, "tf" per la trasformazione dei dati e "**particlecloud**" cioè una nuvola di particelle del filtro che viene visualizzata su schermo tramite Rviz che sono stime della posizione del robot. Questo metodo Monte Carlo è chiamato adattivo (adaptive) perché regola dinamicamente il numero di particelle nel filtro: quando la posa del robot è molto incerta, il numero di particelle aumenta; quando la posa del robot è ben determinata, il numero di particelle si riduce. Ciò consente al robot di fare un compromesso tra velocità di elaborazione e precisione di localizzazione.

5.2.3 Navfn

È il pianificatore globale utilizzato da move_base, supportato dall'interfaccia BaseGlobalPlanner di Nav_core. Calcola il percorso globale dalla posizione del robot al target utilizzando l'algoritmo di **Dijkstra**.

5.2.4 Dwa local planner

Il tipo di pianificatore locale che ho utilizzato, invece, è il "**Dynamic Window Approach**" (Dwa) Local Planner. È un pacchetto ROS che fornisce un controller che guida una base mobile nel piano cercando di evitare gli ostacoli dinamici che ci possono essere nel tragitto. Usando una mappa, il pianificatore crea una traiettoria cinematica per il robot per arrivare da un punto iniziale a una posizione obiettivo. Lungo il percorso, il pianificatore crea, almeno localmente attorno al robot, una funzione valore, rappresentata come una

mappa a griglia. Questa funzione codifica i costi di attraversamento delle celle della griglia. Il compito del controller consiste nell'usare questa funzione valore per determinare le velocità v_x, v_y, ω da inviare al robot.

L'idea di base dell'algoritmo è la seguente:

1) si campiona un insieme di coppie di velocità (v_x, v_y, ω) in uno spazio discretizzato attorno al robot, che rappresenta uno spazio dinamico (finestra dinamica) che cambia man mano che il robot si sposta.
2) per ogni velocità campionata, si esegue una simulazione in avanti a partire dallo stato attuale del robot per prevedere cosa accadrebbe se la velocità campionata fosse applicata per un breve periodo di tempo.
3) valuta il punteggio di ogni traiettoria risultante dalla simulazione in avanti, utilizzando una metrica che incorpora caratteristiche quali: vicinanza agli ostacoli, vicinanza all'obiettivo, prossimità al percorso globale e velocità. Scarta le traiettorie non percorribili, ovvero quelle che si scontrano con gli ostacoli.
4) sceglie la traiettoria con il punteggio più alto che rispetti le caratteristiche del punto precedente e invia la velocità associata al veicolo.
5) ripete l'operazione da capo.

5.3 Codice

Per l'implementazione della navigazione autonoma del robot ho creato il file "**amr_navigation.launch**". Questo file avvia tre nodi: amcl, move_base e map_server nel seguente modo.

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

    <arg name="open_rviz" default="false"/>
    <arg name="move_forward_only" default="false"/>
    <arg name="map_file" default="$(find amr_description)/maps/willow_garage.yaml"/>

    <!-- Map server -->
  <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)"/>

  <!-- AMCL -->
  <include file="$(find amr_description)/launch/amcl.launch"/>

  <!-- move_base -->
  <include file="$(find amr_description)/launch/move_base.launch">
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find amr_description)/rviz/robotmodel.rviz"/>
  </group>

</launch>
```

Fig 5.2: File amr_navigation.launch

All'inizio del file sono definiti degli argomenti:

- "**open_rviz**" è una variabile booleana che, se impostata a true, permette di avviare automaticamente Rviz. In questo caso è impostata a false perché questo file launch l'ho avviato assieme al file "gazebo_rviz.launch" che già apre Rviz.

- "**move_forward_only**" è anch'essa una variabile booleana che se viene settata a true permette al robot di muoversi solo in avanti e non indietro. Quindi dato che il robot reale deve muoversi anche indietro, la simulazione deve rappresentare il più possibile la realtà per cui è impostata a false.
- "**map_file**" è una variabile che legge i dati della mappa scansionata dall'algoritmo Gmapping e sono salvati nel file "willow_garage.yaml" nella directory "maps".

Viene avviato poi il nodo map_server che carica i dati della mappa dall'argomento map_file.

Il nodo amcl viene avviato dal file "**amcl.launch**". Contiene diversi parametri, anch'essi consultabili sul sito "ROS Wiki", che si possono configurare ma che ho lasciato come erano di default.

Di questi parametri si può, ad esempio, modificare il numero minimo e massimo di particelle del filtro oppure la posizione iniziale del robot dalla quale iniziare il campionamento. La posizione iniziale predefinita coincide con l'origine del sistema di riferimento $(0, 0, 0)$ che è una terna (x, y, a) con a angolo di rotazione. L'ho lasciata invariata perché il robot su Gazebo parte proprio dall'origine. Vengono passati ad amcl anche i frame "odom" per l'odometria del robot e "base_link".

Nella pagina seguente è mostrato il codice di "amcl.launch".

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

  <!-- Arguments -->
  <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"/>

  <!-- AMCL -->
  <node pkg="amcl" type="amcl" name="amcl">

    <param name="min_particles"             value="500"/>
    <param name="max_particles"             value="3000"/>
    <param name="kld_err"                   value="0.02"/>
    <param name="update_min_d"              value="0.20"/>
    <param name="update_min_a"              value="0.20"/>
    <param name="resample_interval"         value="1"/>
    <param name="transform_tolerance"       value="0.5"/>
    <param name="recovery_alpha_slow"       value="0.00"/>
    <param name="recovery_alpha_fast"       value="0.00"/>
    <param name="initial_pose_x"            value="$(arg initial_pose_x)"/>
    <param name="initial_pose_y"            value="$(arg initial_pose_y)"/>
    <param name="initial_pose_a"            value="$(arg initial_pose_a)"/>
    <param name="gui_publish_rate"          value="50.0"/>

    <remap from="scan"                      to="$(arg scan_topic)"/>
    <param name="laser_max_range"           value="3.5"/>
    <param name="laser_max_beams"           value="180"/>
    <param name="laser_z_hit"               value="0.5"/>
    <param name="laser_z_short"             value="0.05"/>
    <param name="laser_z_max"               value="0.05"/>
    <param name="laser_z_rand"              value="0.5"/>
    <param name="laser_sigma_hit"           value="0.2"/>
    <param name="laser_lambda_short"        value="0.1"/>
    <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="0.1"/>
    <param name="odom_alpha2"               value="0.1"/>
    <param name="odom_alpha3"               value="0.1"/>
    <param name="odom_alpha4"               value="0.1"/>
    <param name="odom_frame_id"             value="odom"/>
    <param name="base_frame_id"             value="base_link"/>
  </node>
</launch>
```

Fig 5.3: File amcl.launch

Viene infine avviato il nodo move_base tramite il file "**move_base.launch**" che permette la navigazione autonoma, sottoscrivendosi ad un topic chiamato "**move_base_simple/goal**" , che contiene le coordinate del punto da raggiungere, e inviando comandi di velocità pubblicando sul topic "**cmd_vel**". Nel file viene passato l'argomento "move_forward_only" che è stato impostato a false. Questo parametro viene letto dal dwa local planner che permetterà al robot di eseguire anche movimenti all'indietro.

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

  <!-- Arguments -->
  <arg name="cmd_vel_topic" default="/cmd_vel" />
  <arg name="odom_topic" default="odom" />
  <arg name="move_forward_only" default="false"/>

  <!-- move_base -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
    <rosparam file="$(find amr_description)/param/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find amr_description)/param/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find amr_description)/param/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find amr_description)/param/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find amr_description)/param/move_base_params.yaml" command="load" />
    <rosparam file="$(find amr_description)/param/dwa_local_planner_params.yaml" command="load" />
    <remap from="cmd_vel" to="$(arg cmd_vel_topic)"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <param name="DWAPlannerROS/min_vel_x" value="0.0" if="$(arg move_forward_only)"/>
  </node>

</launch>
```

Fig 5.4: File move_base.launch

All'interno del nodo move_base vengono caricati dei parametri per il nodo move_base , per il dwa_local_planner e per la mappa locale e globale generate dai rispettivi pianificatori. Questi parametri sono contenuti dentro dei file in

formato yaml nella cartella "**param**" e sono responsabili per la performance della navigazione e del movimento del robot. Ora verranno citati i più importanti.

```
#robot_radius: 0.25
footprint: "[ [0.4, 0.25], [0.4, -0.25], [-0.4, -0.25], [-0.4, 0.25] ]"
transform_tolerance: 0.4
map_type: costmap

obstacle_layer:
 enabled: true
 obstacle_range: 2.5
 raytrace_range: 3.5
 inflation_radius: 0.5
 track_unknown_space: false
 combination_method: 1

 observation_sources: laser_scan_sensor
 laser_scan_sensor: {data_type: LaserScan, topic: scan, marking: true, clearing: true}


inflation_layer:
  enabled:              true
  cost_scaling_factor:  5.0  # exponential rate at which the obstacle cost drops off
(default: 10)
  inflation_radius:     0.5   # max distance from an obstacle at which costs are incurred for
planning paths.

static_layer:
  enabled:              true
  map_topic:            "map"
```

Fig 5.5: File costmap_common_params.yaml

Nell'immagine precedente, ad esempio, sono mostrati i parametri comuni sia per la mappa locale che quella globale dei due pianificatori nel file "**costmap_common_params**". La variabile "**footprint**" rappresenta un vettore costituito da quattro punti di coordinate (x, y) che sono gli estremi dello spazio occupato dalla base del robot ovvero il frame base_link e viene utilizzato dagli

algoritmi di path planning per capire le dimensioni del robot e di conseguenza scegliere opportuni percorsi. Viene definito poi il topic "scan" dal quale si ottengono i dati laser di osservazione ed il topic della mappa "map".

Un altro importante file di configurazione di parametri è il "**local_costmap_params**". All'interno sono stabiliti dei parametri per la mappa locale a cui fa riferimento il local planner per individuare degli eventuali ostacoli dinamici. È di dimensioni personalizzabili e viene mantenuta sempre attorno al robot durante il movimento. Nel file ho impostato le dimensioni della mappa a 5×5 metri grazie ai parametri "width" e "height".

```yaml
local_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 2.0
  publish_frequency: 1.0
  static_map: false
  rolling_window: true
  width: 5
  height: 5
  resolution: 0.1
  transform_tolerance: 0.5

  plugins:
   - {name: static_layer,        type: "costmap_2d::StaticLayer"}
   - {name: obstacle_layer,      type: "costmap_2d::ObstacleLayer"}
```

Fig 5.6: File local_costmap_params.yaml

Infine, il file di configurazione del dwa local planner è responsabile della pianificazione locale del percorso e della navigazione del robot. Ci sono parametri per impostare la velocità lineare ed angolare massima e minima espresse in metri al secondo, l'accelerazione e parametri riguardo al percorso e al punto target che si deve raggiungere (goal).

```yaml
DWAPlannerROS:

# Robot Configuration Parameters
  max_vel_x: 0.30
  min_vel_x: -0.30

  max_vel_y: 0.0
  min_vel_y: 0.0

# The velocity when robot is moving in a straight line
  max_vel_trans:  0.30
  min_vel_trans:  0.11

  max_vel_theta: 2.75
  min_vel_theta: 1.37

  acc_lim_x: 2.5
  acc_lim_y: 0.0
  acc_lim_theta: 3.2

# Goal Tolerance Parametes
  xy_goal_tolerance: 0.05
  yaw_goal_tolerance: 0.17
  latch_xy_goal_tolerance: false

# Forward Simulation Parameters
  sim_time: 1.5
  vx_samples: 20
  vy_samples: 0
  vth_samples: 40
  controller_frequency: 10.0

# Trajectory Scoring Parameters
  path_distance_bias: 32.0
  goal_distance_bias: 20.0
  occdist_scale: 0.02
  forward_point_distance: 0.325
  stop_time_buffer: 0.2
  scaling_speed: 0.25
  max_scaling_factor: 0.2

# Oscillation Prevention Parameters
  oscillation_reset_dist: 0.05

# Debugging
  publish_traj_pc : true
  publish_cost_grid_pc: true
```

Fig 5.7: File dwa_local_planner_params.yaml

5.4 Test su simulazione

Una volta avviato il file "gazebo_rviz.launch" che prepara l'ambiente e il robot, ho avviato il file "amr_navigation.launch" per la navigazione, ho configurato Rviz per la lettura dei topic ottenendo la seguente schermata.

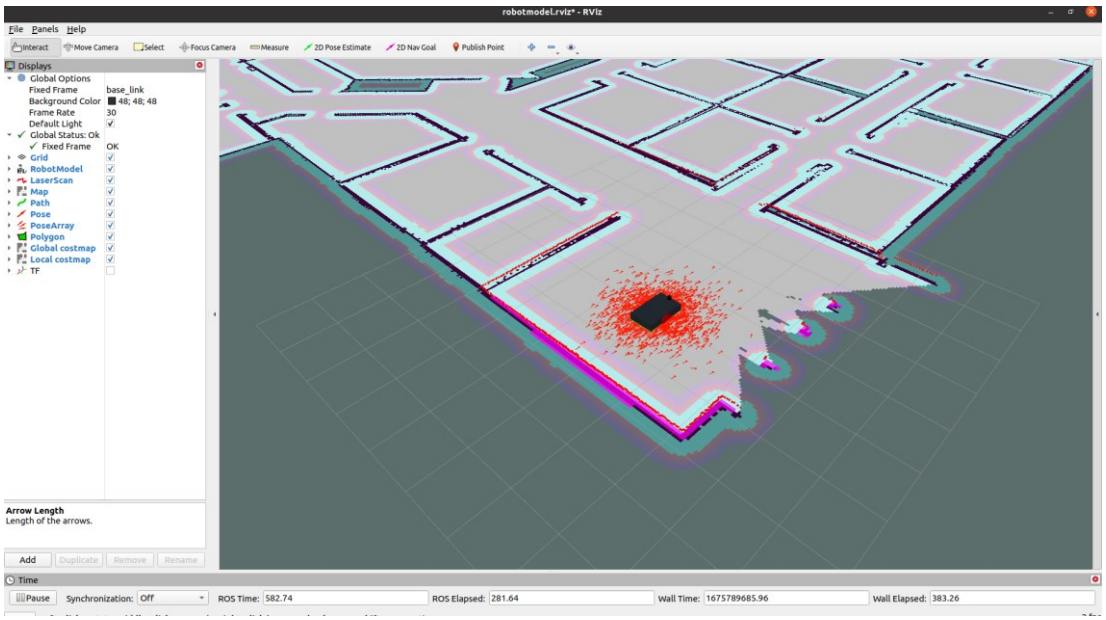

Fig 5.8: Schermata Rviz dopo l'avvio di amr_navigation.launch

Le frecce rosse attorno al robot che si possono osservare nell'immagine sopra sono le particelle dell'algoritmo Amcl che inizialmente sono posizionate nello spazio circostante il punto di origine (0, 0, 0). Esse sono pubblicate dal nodo amcl tramite il topic "**particlecloud**".

I tratti violacei dei muri sotto il robot rappresentano i confini della mappa locale gestita dal local planner.

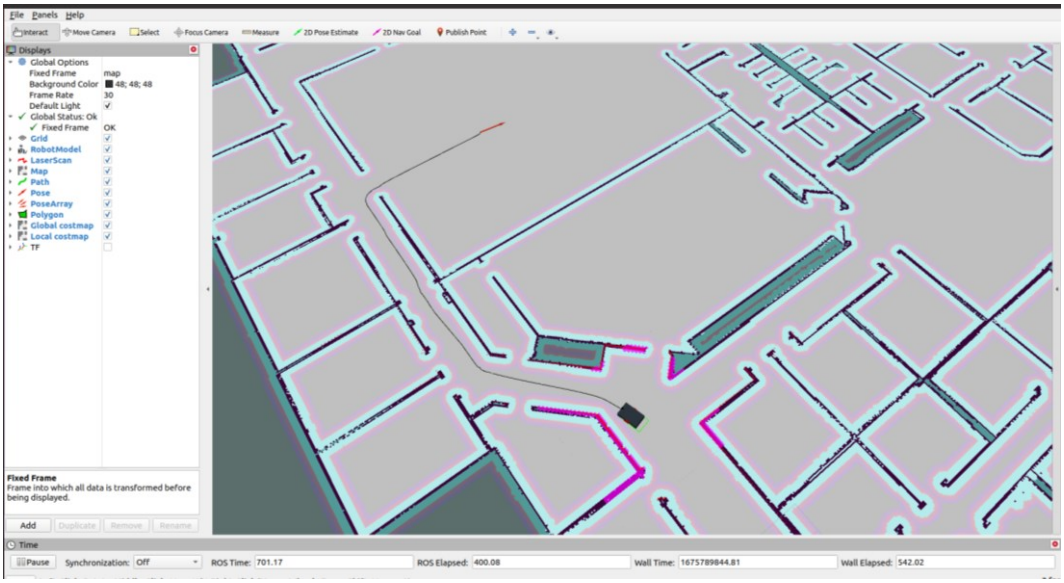

Fig 5.9: Assegnazione obbiettivo e path planning

Nell'immagine 5.9 invece si può vedere come, assegnato un punto sulla mappa da raggiungere, il robot calcola il percorso migliore e più breve per raggiungerlo e inizia a muoversi tramite il nodo move base e il dwa local planner.

Le particelle non sono visibili nell'immagine in quanto, man mano che il robot si sposta e riceve i segnali di velocità, esse convergono verso il centro del robot perché vengono prese solo le particelle con peso maggiore nella fase di ricampionamento e che stimano la posizione corretta.

Il target da raggiungere è stato impostato tramite l'interfaccia grafica di Rviz con un pulsante chiamato "**2D Nav Goal**". Nell'immagine 5.9 è rappresentato da una freccia rossa con la direzione che il robot deve assumere in quella posizione.

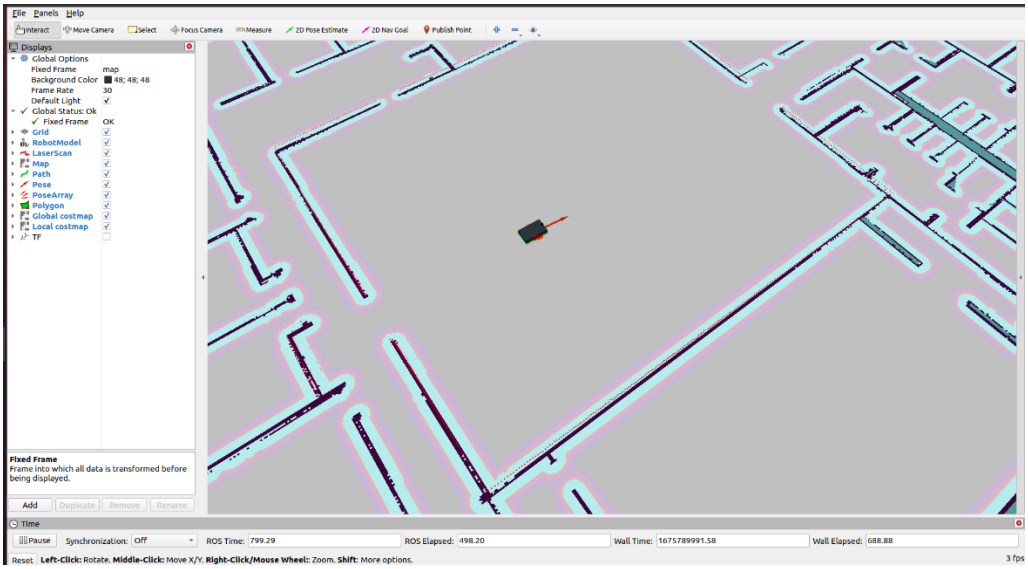

Fig 5.10: Obbiettivo raggiunto

È stato inoltre testato il dwa local planner per l'aggiramento di ostacoli dinamici. Stavolta ho inserito un ostacolo davanti alla porta che deve attraversare il robot impedendo di raggiungere il target.

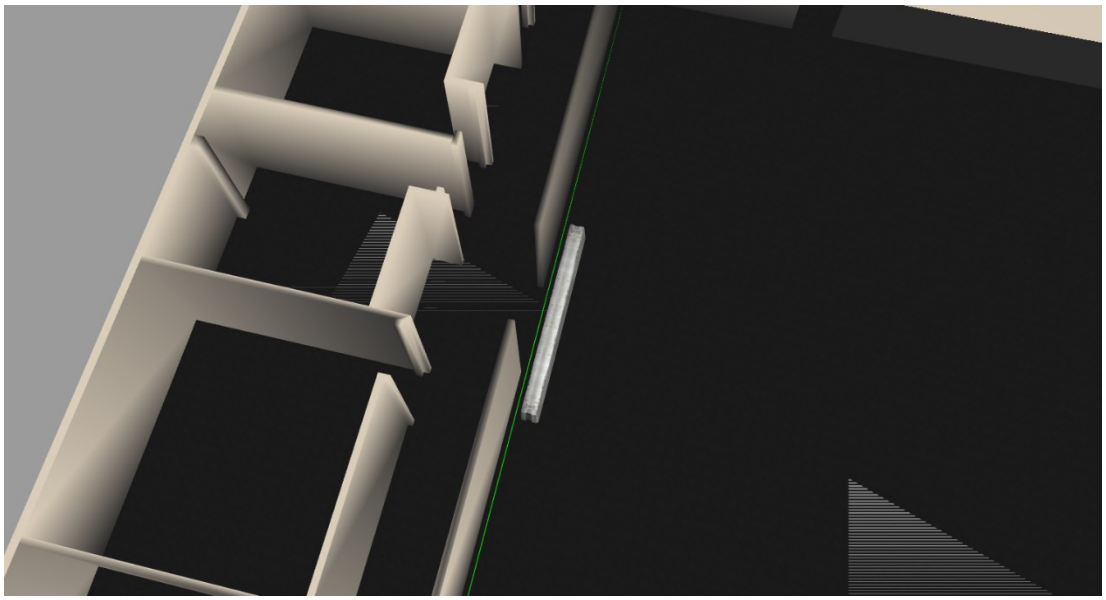

Fig 5.11: Inserimento ostacolo

Il robot, nella fase iniziale di individuazione del miglior percorso attraverso il pianificatore globale Navfn, è all'oscuro dell'inserimento dell'ostacolo. Mediante il dwa local planner però, è riuscito ad individuarlo nella mappa locale generata attorno al robot, ricalcolando il percorso da attraversare.

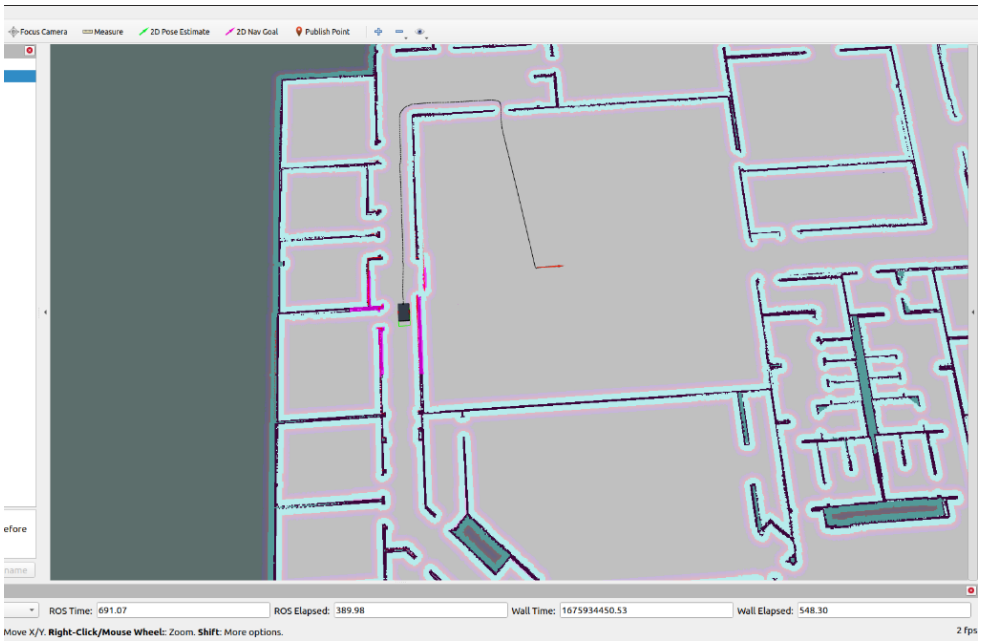

Fig 5.12: Percezione dell'ostacolo dinamico e percorso ricalcolato

Nella pagina seguente, invece, è mostrato il grafico di collegamento e di comunicazione tra i nodi ROS durante la navigazione autonoma ottenuto tramite il pacchetto "**rqt_graph**". I cerchi sono i nodi mentre i rettangoli sono i topic.

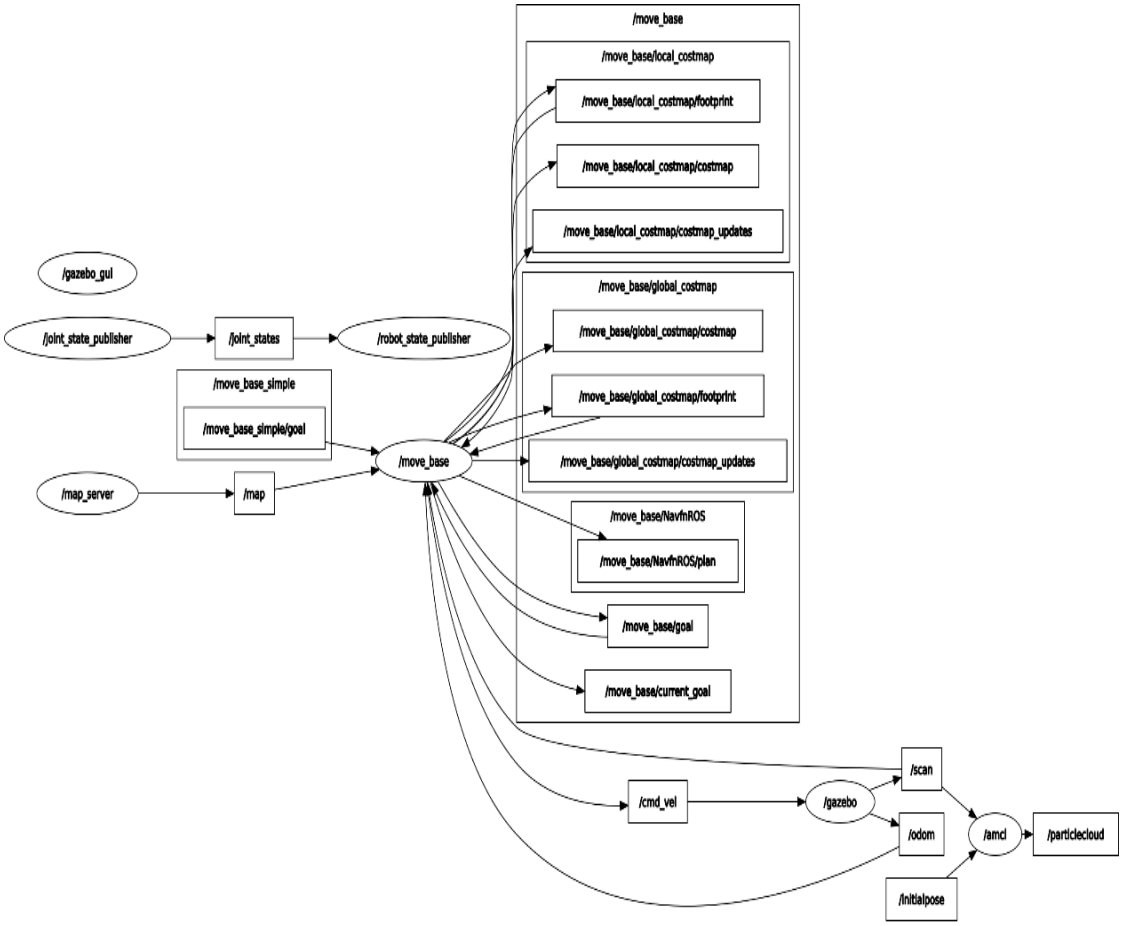

Fig 5.13: Rqt_graph navigazione autonoma

Capitolo

6 Sviluppi futuri

Il lavoro svolto, come si è visto, è compiuto su simulazione tramite Gazebo. Infatti è Gazebo, attraverso i plugin implementati, a pubblicare i dati di odometria del veicolo e i dati sensoristici del laser.

È stato fatto ciò per avere un'idea di come dovrebbe comportarsi, nell'ambiente reale, l'amr reale.

Nel mondo reale però non ci sarà un simulatore. Occorrerà perciò programmare in linguaggio ROS un nodo che pubblichi su un topic i dati odometrici delle ruote, sfruttando il codice realizzato dallo studente precedente nel suo progetto. Questo sarà utilizzato dal nodo Gmapping per la mappatura dell'ambiente reale e dal nodo amcl nella stima della posizione per la parte di navigazione autonoma. Per quanto riguarda i dati del sensore lidar, nel workspace è già stato scaricato un pacchetto nominato "rplidar_ros" che serve a far funzionare il sensore RPLidar A1 e visualizzare i dati reali del laser su Rviz tramite il topic "scan".

Bisognerà poi eseguire dei test cinematici sul robot per verificare il corretto funzionamento dell'algoritmo Amcl e del nodo move base per la navigazione autonoma. Sarà infatti move base, tramite il dwa local planner, che dovrà inviare comandi di velocità ai controller delle ruote.

Il metodo adottato per la navigazione autonoma è stato quello di creare prima una mappa dell'ambiente sfruttando Gmapping e poi utilizzare Amcl e move base fornendoli della mappa già salvata. In pratica la parte di localizzazione dello slam Gmapping non viene utilizzata ma viene solo sfruttata la mappa

generata, la quale viene passata ad Amcl che invece compie solo la parte di localizzazione. Quindi il robot già conosce l'ambiente e deve solo capire in che punto si trova all'interno di esso. Un'alternativa potrebbe essere quella di utilizzare solo l'algoritmo slam Gmapping, che compie sia la localizzazione sia il mapping ambientale, unito ad un algoritmo di esplorazione che invii comandi di velocità alle ruote e rilevi gli ostacoli in base ai dati del laser acquisiti in modo da evitarli. Il robot, quindi, sarà in un ambiente sconosciuto per lui e non conoscerà a priori la mappa ma la costruirà al momento e nello stesso tempo si localizzerà.

Bibliografia e sitografia

1) Gervasio Danilo, *Design and development of anAutonomous Mobile Robot (AMR) based on a ROS controller*, A.A 2021/2022
2) Urbinati Simone, *Strategia di aggiramento ostacoli per veicoli d'ausilio alla mobilità dei disabili*, anno 1999
3) Caldarelli Andrea, *Localizzazione e costruzione di mappe ambientali per un robot mobile: un approccio basato sul filtro di Kalman*, A.A 2005/2006
4) Mikael Berg, *Navigation with Simultaneous Localization and Mapping For Indoor Mobile Robot*, https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/260896
5) Bayu Kanugrahan Luknanto, *A Review of 2D SLAM Algorithms on ROS*, https://www.politesi.polimi.it/bitstream/10589/164687/3/A%20Review%20of%202D%20SLAM%20Algorithms%20on%20ROS.pdf
6) Guolai Jiang, Lei Yin, Shaokun Jin, Chaoran Tian, Xinbo Ma, Yongsheng Ou, *A Simultaneous Localization and Mapping (SLAM) Framework for 2.5D Map Building Based on Low-Cost LiDAR and Vision Fusion*, https://www.mdpi.com/2076-3417/9/10/2105
7) Peter Aerts, Eric Demeester, *Benchmarking of 2D-Slam Algorithms A Validation for the TETRA project Ad Usum Navigantium*, http://www.acro.be/downloadvrij/Benchmark_2D_SLAM.pdf
8) Johan Alexandersson och Olle Nordin, *Implementation of SLAM algorithms in a small-scale vehicle using model-based development*, https://liu.diva-portal.org/smash/get/diva2:1218791/FULLTEXT01.pdf
9) Eram Arfa, *Study and implementation of LiDAR-based SLAM algorithm and map-based autonomous navigation for a telepresence robot to be used as a chaperon for smart laboratory requirements*, https://elib.uni-

stuttgart.de/bitstream/11682/12098/1/MasterThesis_EramArfa_IAAS_WiSe2122.pdf

10) Gabriele Bolano, *SLAM multi-agente distribuito e decentralizzato per l'esplorazione coordinata di ambienti indoor*, https://core.ac.uk/download/pdf/79622717.pdf
11) Kaiyu Zheng, *ROS Navigation Tuning Guide*, https://kaiyuzheng.me/documents/navguide.pdf
12) Xuexi Zhang, Jiajun Lai, Dongliang Xu, Huaijun Li, Minyue Fu, *2D Lidar-Based SLAM and Path Planning for Indoor Rescue Using Mobile Robots,* https://www.hindawi.com/journals/jat/2020/8867937/
13) Pablo Marin-Plaza, Ahmed Hussein, David Martin, Arturo de la Escalera, *Global and Local Path Planning Study in a ROS-Based Research Platform for Autonomous Vehicles*, https://www.hindawi.com/journals/jat/2018/6392697/
14) http://wiki.ros.org/ROS/Tutorials
15) https://roboticsknowledgebase.com/wiki/state-estimation/adaptive-monte-carlo-localization/
16) https://kiranpalla.com/autonomous-navigation-ros-differential-drive-robot-simulation/introduction/
17) http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF
18) https://www.theconstructsim.com/exploring-ros-2-wheeled-robot-part-01/
19) https://automaticaddison.com/coordinate-frames-and-transforms-for-ros-based-mobile-robots/
20) http://wiki.ros.org/navigation?distro=noetic
21) https://it.wikipedia.org/wiki/Robot_Operating_System
22) https://classic.gazebosim.org/tutorials?tut=ros_gzplugins
23) https://github.com/XRobots/ReallyUsefulRobot/tree/main/ROS