Università Politecnica delle Marche

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Algoritmi di Navigazione e Orientamento per Veicoli Autonomi e Robotici in Ambiente ROS

Navigation and Orientation Algorithms for Autonomous and Robotic Vehicles in ROS Framework

Candidato:
Damiano Santarelli

Relatore:
Prof. Gianluca Ippoliti

Correlatore:
Ing. Gianluca Toscano

22 ottobre 2021

Indice

Elenco delle figure

Elenco delle tabelle

Abstract

L'obiettivo di questa tesi consiste nella ricerca e nello sviluppo di algoritmi di navigazione eseguiti su sistema operativo Linux utilizzando il framework ROS. Attraverso tali strumenti si desidera ottenere un sistema che elabori e percorra autonomamente un certo percorso, dati il punto A di partenza e il punto B di arrivo. La prima e più corposa parte del lavoro è consistita nell'implementazione in simulazione di un veicolo autonomo. A tal proposito è stato compiuto un lavoro di ricerca degli algoritmi più efficienti per poter implementare le funzionalità di Visual Odometry e di sensor fusion tramite filtro di Kalman esteso, al fine di ottenere l'odometria del veicolo in tempo reale. Tali algoritmi sono stati implementati all'interno dell'ambiente ROS e sono stati utilizzati per il corretto funzionamento di Navigation2, anch'esso eseguito tramite ROS, per garantire un'efficace navigazione autonoma del veicolo. Successivamente si è passati alla fase applicativa del lavoro di tesi, ovvero all'implementazione degli algoritmi studiati in simulazione su un veicolo reale in scala 1:10. Per garantire la corretta comunicazione tra il controllore a basso livello a bordo del veicolo e il sistema di navigazione sono stati sviluppati script in linguaggio Python. Inoltre sono stati approfonditi il tema delle dipendenze di Linux, la struttura del sistema operativo dell'hardware in questione (ARM64) e, in particolare, il metodo di installazione e configurazione delle librerie necessarie al funzionamento dei pacchetti sopra citati.

Introduzione

Quello della guida autonoma sta diventando un tema sempre più centrale all'interno dei dibattiti riguardanti la mobilità. Negli ultimi anni c'è stato infatti uno sviluppo considerevole in questo campo, dovuto soprattutto all'esponenziale miglioramento delle piattaforme hardware. L'aumento delle capacità di calcolo da parte dei chip di ultima generazione ha reso possibile l'impiego di algoritmi computazionalmente complessi come quelli di machine learning, deep learning e reti neurali, sui quali si basa l'intelligenza artificiale. Di conseguenza anche numerosi algoritmi di guida autonoma, anni fa inattuabili per mancanza di tecnologie adatte, hanno subito uno sviluppo molto importante che li ha portati oggi ad essere implementati all'interno di vetture reali. Basti pensare alla casa costruttrice americana Tesla, che è pronta ad introdurre sul mercato il suo algoritmo di Full Self Driving, il quale permette all'auto di navigare verso qualsiasi destinazione senza il bisogno di alcun intervento da parte del conducente. Tuttavia, nonostante gli sviluppi più consistenti siano maturati negli ultimi 15 anni, la guida autonoma possiede una storia lunga più di un secolo, dove l'intelligenza artificiale rappresenta soltanto il culmine di una tecnologia che è in via di sviluppo già da parecchio tempo.

Oltre ad un breve inquadramento storico, durante questo capitolo introduttivo saranno illustrati gli obiettivi della tesi ed il software utilizzato per realizzarli. Il cospicuo lavoro iniziale di ricerca dei pacchetti più adatti, si è focalizzato nel massimizzarne la stabilità e allo stesso tempo l'efficienza. Si è quindi cercato di utilizzare software che fossero testati e costantemente aggiornati al fine di ottenere un risultato efficace.
Tutti gli algoritmi utilizzati in questa tesi sono sviluppati per funzionare nell'ambiente ROS, acronimo di Robot Operating System. Proprio per massimizzarne la stabilità, si è scelto di utilizzare due versioni diverse di ROS, in particolare ROS1 "Noetic" e ROS2 "Foxy", ed utilizzare il pacchetto `ros1_bridge` per la comunicazione tra i due framework.
Il risultato finale sarà quello di connettere tra loro tutti gli algoritmi utilizzati per eseguire i singoli task ed arrivare ad un sistema complesso che riesca a navigare autonomamente verso una certa destinazione.

Un po' di Storia

La guida autonoma è un argomento affascinante che da sempre ha caratterizzato le evoluzioni della tecnica. Prima ancora dell'automobile, si può parlare di guida autonoma con riferimento ai siluri semoventi inventati da Robert Whitehead nella seconda metà dell'800. In aeronautica, del resto, già dieci anni dopo il primo volo dei fratelli Wright gli aerei venivano dotati di auto pilota e, certamente, anche nel settore navale il progresso tecnologico aveva implementato le prime soluzioni sul tema. Più difficile, invece, applicare questi concetti all'automobile.
In realtà, alla guida autonoma l'uomo pensava già un secolo fa. I veicoli elettrici, del resto, a cavallo tra '800 e '900 erano considerati il futuro della mobilità e i motori azionati da combustibili fossili erano visti in declino e si riteneva fossero destinati a sparire dalla faccia della terra entro pochi anni. Anche la guida autonoma, oggi cavallo di battaglia dei principali marchi costruttori premium mondiali, in realtà non è niente di nuovo rispetto a quanto non fosse già stato quantomeno teorizzato. Le smisurate possibilità offerte dalla tecnologia d'oggi, insomma, hanno solo reso realizzabile e non più fantascientifico quanto era già stato pensato molti decenni addietro.

Si ha notizia di un primissimo tentativo di dare concretezza al concetto della guida autonoma a metà degli anni 20 a New York: a bordo di una Chandler del 1926 la Houdina Radio Control installò una piccola rete di motori elettrici, tutti afferenti una grande antenna montata sul tetto. Da una seconda macchina, una strumentazione inviava impulsi elettrici all'antenna e da qui ai dispositivi, che erano in grado di muovere il veicolo senza bisogno di intervento umano. La Linrrican Wonder fu protagonista di una dimostrazione pubblica tra Broadway e la 5th Avenue a Manhattan.
Non si ebbero più notizie rilevanti su questo affascinante tema fino alla metà degli anni 30 quando iniziarono a moltiplicarsi gli studi sulla possibilità di creare automobili e taxi senza pilota allo scopo di liberare le città americane dal traffico sempre più opprimente e dalla scarsità dilagante di parcheggi.
Poi, nel 1939, alla Fiera Mondiale svoltasi a New York una prima grande visione del futuro: il padiglione della General Motors, opera di Norman Bel Geddes e denominato Futurama, creava sensazione: in esso una costruzione di tanti possibili scenari sul futuro dell'architettura e della mobilità. La costruzione anticipava il futuro architettonico delle città americane di almeno 20 anni disegnando un ambiente popolato da automobili radio controllate mosse da campi magnetici. Secondo Geddes il modello di funzionamento delle autostrade prendeva spunto da quanto già esisteva in Italia e Germania ma migliorato con l'introduzione dei principi di funzionamento propri delle ferrovie, dotate di sistemi per la velocità controllata e l'analisi della posizione dei treni per evitare collisioni. L'idea era semplice: si guidava tranquillamente l'automobile fino all'ingresso di un autostrada, si inseriva il pilota automatico e la vettura sarebbe rimasta nella propria corsia fino all'uscita (utilizzando bande magnetiche, rotaie su cui scorrevano ruote metalliche accoppiate ai pneumatici ecc.).

La seconda guerra mondiale interruppe ogni progetto, costringendo all'oblio ogni idea e sperimentazione sulla guida autonoma per le automobili. Nuovi risultati furono raggiunti nei primi anni 50: General Motors e la RCA (Radio Corporation of America), colosso americano dell'elettronica e della produzione musicale, crearono

un piccolo prototipo di automobile guidato e controllato attraverso cavi annegati nel pavimento. L'invenzione scatenò la curiosità di Leland M. Hancock (ingegnere del Nebraska Department of Roads) e del suo capo L. N. Ress, che iniziarono una serie di studi a partire dall'infrastruttura autostradale dell'epoca. Ci vollero cinque anni ma, finalmente, nel 1958 (anno in cui la Chrysler Imperial fu la prima automobile a proporre il Cruise Control), i due ingegneri e la RCA Labs presentarono i risultati: lungo un tratto di strada pubblica nella città di Lincoln, 400 piedi (121 metri) erano letteralmente farciti di una rete di sensori capaci di controllare la posizione di un veicolo e di rilevare la presenza di ostacoli. L'automobile, dotata della tecnologia in grado di dialogare con la sensoristica, era capace di accelerare, frenare o sterzare da sola.
Il risultato fu molto promettente, al punto che, secondo la maggior parte dei costruttori automobilistici dell'epoca, la guida autonoma sarebbe diventata una tecnologia perfettamente funzionante e fruibile a partire dal 1975.

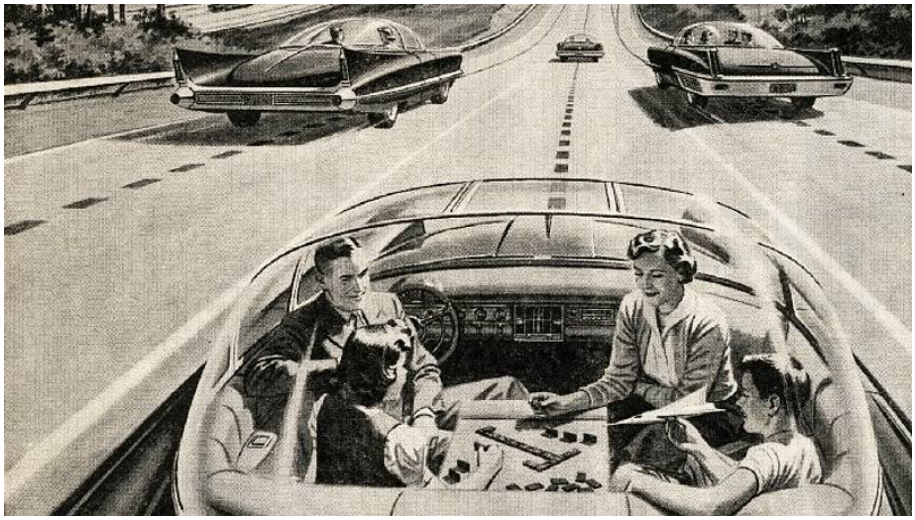


Figura 1: La guida autonoma immaginata negli anni '50

Studi sul tema della guida autonoma furono condotti un po' ovunque. In Inghilterra se ne interessò anche il Governo e negli anni 60 il Transport and Road Research Laboratory realizzò una serie di prototipi dotati di tecnologia in grado di captare sensori posti nell'asfalto e assumere comportamenti attivi in conseguenza. L'idea era partita già prima che fosse inaugurata la M1, la prima autostrada del Regno Unito. Ma poiché già si disegnavano scenari a tinte fosche sul futuro del traffico sulle strade inglesi, il Governo varò un programma di ricerca per studiare soluzioni concrete e percorribili. Il risultato fu innanzitutto una DS19 dotata di un sistema "preistorico" ma efficace: la vettura, interagendo con cavi magnetici inseriti nell'asfalto, era in grado di procedere da sola e raggiungere anche velocità molto elevate (fu testata fino a 130 km/h), frenare e sterzare. L'obiettivo era di arrivare a conoscere, in tempo reale, posizione e direzione dei veicoli sulle strade. Controllando sterzo, distanza tra essi e velocità, un computer centrale avrebbe potuto calcolare la condizione di traffico migliore che evitasse congestioni e... automobilisti nervosi! Ulteriori test furono eseguiti con una Standard Vanguard e con una Austin Mini.
Gli studi, che proseguirono anche negli anni 70, tennero conto, naturalmente, dei costi per la realizzazione di tutte le infrastrutture. Fu calcolato che per fine secolo lo Stato si sarebbe rifatto di tutte le spese per dotare la rete autostradale di tutto

il sistema di sensori necessario. Ma a metà della decade il progetto fu chiuso.

Nel 1987 è stato lanciato l'Eureka Prometheus Project (PROgraMme for a European Traffic of Highest Efficiency and Unprecedented Safety, 1987-1995) un programma di finanziamento lanciato dai membri dell'Eureka (organizzazione europea per la ricerca tecnologica applicata allo sviluppo produttivo) e che ha stanziato 749 milioni di euro per progetti nel campo della guida autonoma.
Grazie a questi fondi, nel 1994 sono state presentate a Parigi la VaMP e la sua gemella, Vita-2. La VaMP, costruita sulla base di una Mercedes 500 SEL dal team dello scienziato Ernst Dickmanns, dall'Università di Monaco di Baviera e da Mercedes Benz, è guidata completamente da un computer, capace di agire su sterzo, acceleratore e freni grazie alla capacità di analisi in tempo reale dell'ambiente circostante. La visione computerizzata di VaMP riesce a riconoscere strade e veicoli. Nel 1995 ha percorso 2000 chilometri da Monaco a Copenaghen e ritorno, anche a velocità di 180 orari, richiedendo in pochissime occasioni l'intervento umano.

Gli Obiettivi Principali

Tra gli obiettivi principali prefissati per questa tesi c'è sicuramente quello di ottenere un segnale affidabile di odometria. La soluzione più efficace per questo tipo di task è rappresentata dalla Visual Odometry, che rappresenta un approccio molto interessante poiché evita completamente il problema dello slittamento degli pneumatici, che si presenta invece nel caso di utilizzo di un encoder. In più, grazie allo sviluppo del Machine Learning gli algoritmi per questo tipo di task stanno migliorando esponenzialmente. Alle informazioni di Visual Odometry è stato unito un segnale di accelerazione proveniente da una IMU, utilizzando un algoritmo di Sensor Fusion.
Altra parte integrante del lavoro di tesi è il sistema di navigazione che, ricevendo informazioni dai sensori ha il compito di elaborare un percorso ed inviare comandi al veicolo per farlo arrivare a destinazione.
Un ulteriore obiettivo che si desidera raggiungere durante questo lavoro di tirocinio e di stesura di tesi, è quello di comprendere le problematiche relative ad un sistema di guida autonoma e riuscire a ricavare soluzioni efficaci allo scopo di arrivare ad un sistema finale funzionante e capace effettivamente di muoversi autonomamente. In più, la comprensione e la conoscenza approfondita del sistema permetteranno anche di scoprire nuove strade che porteranno verso sviluppi futuri che potranno essere oggetto di un ampliamento di questo lavoro. La costante evoluzione del campo della guida autonoma offre spunti ed approcci sempre nuovi alla risoluzione dei problemi e delle sfide presenti e per questo è molto importante seguire questa evoluzione e far parte di essa offrendo, se pur in minima parte, il proprio contributo alla comunità tecnologica.

Gli argomenti appena introdotti saranno trattati in dettaglio durante il primo capitolo, mentre in quelli successivi saranno illustrati i test svolti in simulazione, l'hardware del veicolo reale in scala 1:10 e le prove finali effettuate con il veicolo.

Capitolo 1

Studio del Software per la Realizzazione dell'Algoritmo

Durante questo capitolo sarà illustrato nel dettaglio tutto il software utilizzato per realizzare il sistema di navigazione autonoma di un veicolo di tipo Ackermann, a partire dal controllo del veicolo a basso livello fino ad arrivare agli algoritmi di navigazione utilizzati all'interno dei vari server di Navigation2.

La prima fase di studio è stata necessaria oltre che per trovare il software più adatto, anche per definire la struttura del sistema. L'obiettivo, come introdotto nel paragrafo precedente, è quello di ricavare i dati di odometria direttamente da un sistema di visione sfruttando un algoritmo di visual odometry, senza quindi utilizzare encoder per misurare la velocità di rotazione degli pneumatici. Inoltre, tale valore di odometria sarà migliorato fondendolo con le informazioni provenienti da una IMU. Infine, il rilevamento di ostacoli lungo il percorso sarà demandato ad un sensore LIDAR. Fatta tale premessa, di seguito viene mostrato uno schema riassuntivo dei task da adempiere e del software corrispondente:

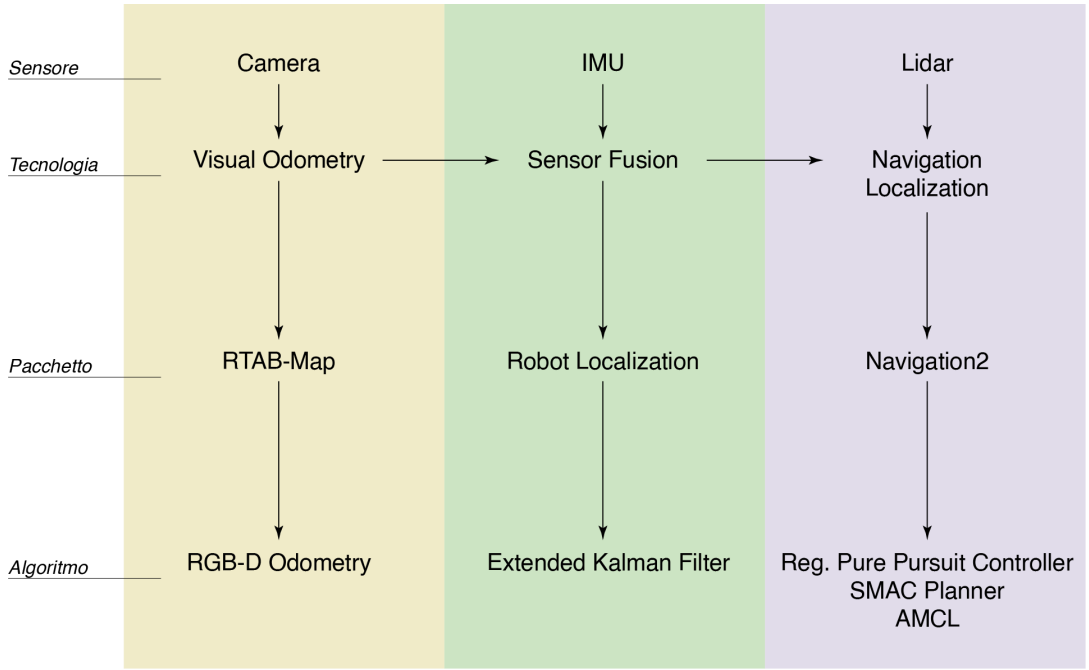


Figura 1.1: Task da eseguire e software utilizzato

Nel seguito saranno illustrati il framework ROS, il sistema in cui vengono eseguiti tutti gli algoritmi, e l'ambiente di simulazione 3D Gazebo.

1.1 Gazebo: Introduzione e Feature Principali

All'interno di questa parte sarà illustrato il funzionamento del simulatore 3D Gazebo, oltre alle sue potenzialità e ai motivi per i quali si è scelto proprio questo ambiente di sviluppo per realizzare la simulazione dell'algoritmo di navigazione autonoma trattato in questo elaborato.

Gazebo rappresenta una soluzione open source per la realizzazione di simulazioni di robotica. Al suo interno integra più di un physics engine ad alta performance, come l'Open Dynamics Engine, che riesce grazie ai calcoli elaborati internamente, a stimare e simulare il comportamento fisico dei corpi rigidi che si decidono di inserire all'interno dell'ambiente.

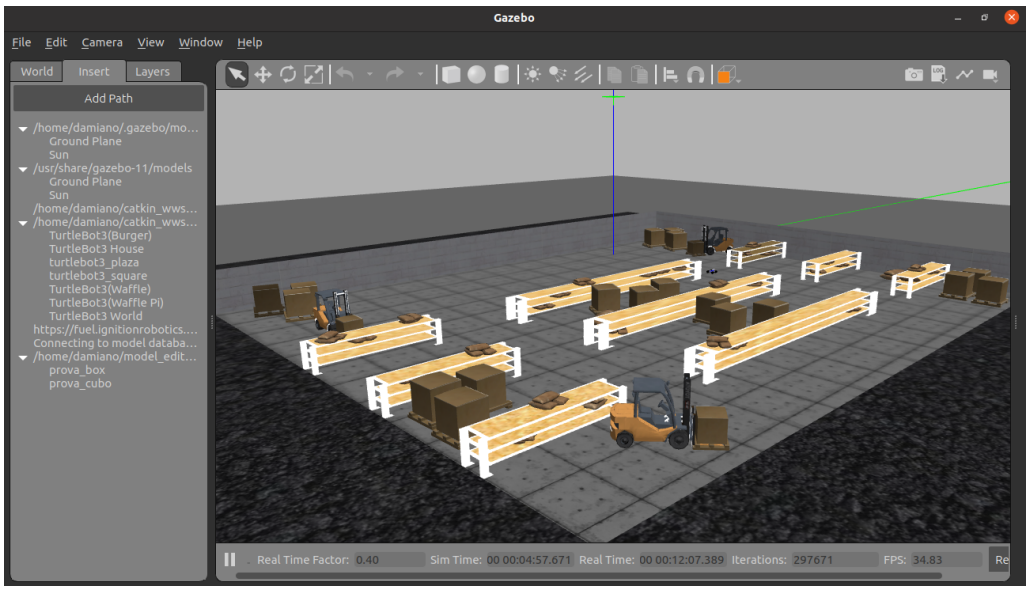


Figura 1.2: Esempio di visuale nell'ambiente Gazebo

Tale algoritmo integra un "rigid body dynamics simulation engine", atto a stimare la dinamica dei corpi rigidi all'interno della simulazione, e un "collision detection engine", necessario per stimare il comportamento di tali corpi durante una collisione.

Grazie a Gazebo è possibile caricare un ambiente virtuale e far interagire con esso un robot, che in questo elaborato è stato rappresentato da un veicolo autonomo a quattro ruote.

Oltre alla simulazione grafica e della dinamica dei corpi che si muovono all'interno dell'ambiente, Gazebo fornisce anche dei plugin per poter rappresentare in maniera virtuale vari tipi di sensori. Dal motore fisico vengono estrapolate tutte le informazioni necessarie per poter simulare il comportamento di un sensore reale. I sensori che sono stati implementati in questo elaborato sono una IMU, che sfrutta i dati di accelerazione del veicolo, un LIDAR, che si serve delle coordinate dei corpi rigidi presenti all'interno dell'ambiente e ricava una distanza da essi, e di una camera. I dati raccolti in simulazione rappresentano un grande vantaggio in termini di tempi di sviluppo degli algoritmi, nonostante tali dati siano privi dei disturbi caratteristici

presenti in un test reale.

Un veicolo, come anche un ambiente grafico come quello mostrato in figura, è composto di varie parti, che possiedono vincoli le une con le altre. Ad esempio, il sensore LIDAR è vincolato in maniera rigida con lo chassis del veicolo, mentre gli pneumatici costituiscono con esso una coppia cinematica di tipo rotoidale. Pertanto, per poter attuare la simulazione è necessario descrivere tutti i link che caratterizzano un sistema di corpi, e anche tutti i vincoli che agiscono tra essi. Nei file descrittivi (.urdf) sono descritte le dimensioni dei link, il tipo e la posizione del giunto che collega un link all'altro, ma anche le aree e i volumi di collisione relativi a tali corpi. Inoltre, all'interno della descrizione dei link che caratterizzano i sensori, viene anche dichiarato il plugin di Gazebo che dovrà essere eseguito per effettuarne la simulazione. È molto importante che questi file siano scritti in maniera corretta e coerente con la fisica del veicolo e che siano ricchi di dettagli, al fine di realizzare una simulazione aderente il più possibile alla realtà. Gazebo infatti rappresenta, in questa prima fase, il mondo reale, pertanto l'obiettivo è quello di sviluppare il proprio algoritmo con il suo ausilio, per poi provare la stessa configurazione sul proprio robot, servendosi delle informazioni provenienti dai sensori reali montati all'interno del veicolo.

1.2 Introduzione al Framework ROS

ROS è un meta-sistema operativo, pensato per controllare i sistemi Robot. Infatti, integra molte funzionalità proprie di un sistema operativo, come il controllo di dispositivi a basso livello, il passaggio di messaggi tra processi, la gestione di pacchetti. In esso sono disponibili anche strumenti e librerie per scrivere ed eseguire codice tra più computer. Anche se viene utilizzato per far interagire tra di loro processi (nodi) e possiede un proprio kernel che viene eseguito (roscore), ROS non è un vero e proprio sistema operativo. Infatti è necessaria una macchina Linux sulla quale ROS può essere installato ed eseguito. In questo elaborato è stato utilizzato il sistema operativo Linux Ubuntu 20.04. L'infrastruttura di comunicazione tra processi ROS, il "ROS runtime graph", consiste in una rete peer-to-peer in cui i vari processi sono connessi tra loro, formando un albero. Sono presenti diversi stili di comunicazione in questa infrastruttura, come lo scambio di informazioni sincrono tra servizi, lo streaming asincrono di dati tra topic, salvataggio di dati in un Server. ROS non è un framework Real-Time, tuttavia è possibile integrarlo con sistemi di terze parti Real-Time. L'obiettivo principale del sistema ROS è la capacità di poter riutilizzare il codice per scopi diversi. Infatti, ROS è un framework di processi (nodi), che possono essere eseguiti individualmente e possono essere collegati fra loro a runtime. Questo consente a tali processi di poter essere raggruppati in "pacchetti" o "stacks" per poter essere riutilizzati in altri progetti.
È questo il caso del Navigation2, successore del Navigation Stack nella versione 2 di ROS, necessario in questo lavoro per poter implementare gli algoritmi di navigazione indoor basati su SLAM. Navigation2 è un insieme di processi che prende in input informazioni dall'odometria e dai sensori, ovvero dai relativi topic, e fornisce in output comandi in velocità per controllare i movimenti del robot. Per farlo si basa su una mappa statica, che in questo elaborato sarà aggiornata in tempo reale tramite la tecnologia SLAM, grazie ad un pacchetto aggiuntivo.

Nei prossimi paragrafi saranno illustrate le principali componenti del framework ROS, in particolare quelle che permettono una comunicazione veloce ed efficace tra i vari task che compongono l'algoritmo. L'ambiente ROS si basa sui Nodes, nei quali i task veri e propri vengono implementati. I vari Nodes che compongono il sistema possono comunicare tra di loro attraverso tools come Topic, Services, Actions, Parameters. Nel seguito sarà illustrato il funzionamento di tali componenti strutturali dell'ambiente ROS.

1.2.1 ROS Nodes

Ogni Node del sistema ha un compito ben preciso, come ad esempio il controllo di un sensore, di un attuatore, una funzione di navigazione etc.

Ogni nodo può comunicare con altri nodi inviando o ricevendo informazioni attraverso le strutture di ROS (topics, services, actions, parameters).

1.2.2 ROS Topics

Ogni topic all'interno del sistema si occupa di tenere traccia dei messaggi inviati dai nodi e li rende disponibili per altri nodi. Ogni topic possiede due principali liste di nodi:

- lista publishers: in tale lista sono presenti tutti i nodi che pubblicano dei messaggi nel topic. Nell'esempio del Turtlebot3 visto in precedenza, il nodo associato al laser scanner pubblica le informazioni relative a tale sensore nel topic `/scan`. Tale topic salva tutte le informazioni e le rende disponibili ai nodi subscribers.
- lista subscribers: i nodi subscribers sono tutti quei nodi che hanno bisogno di sapere le informazioni che vengono pubblicate nel topic. Facendo ancora riferimento all'esempio del Turtlebot3, un nodo che ha necessità di conoscere l'output del laser scanner, diventerà un subscriber del topic `/scan`

Ogni topic può possedere più nodi subscribers e publishers.

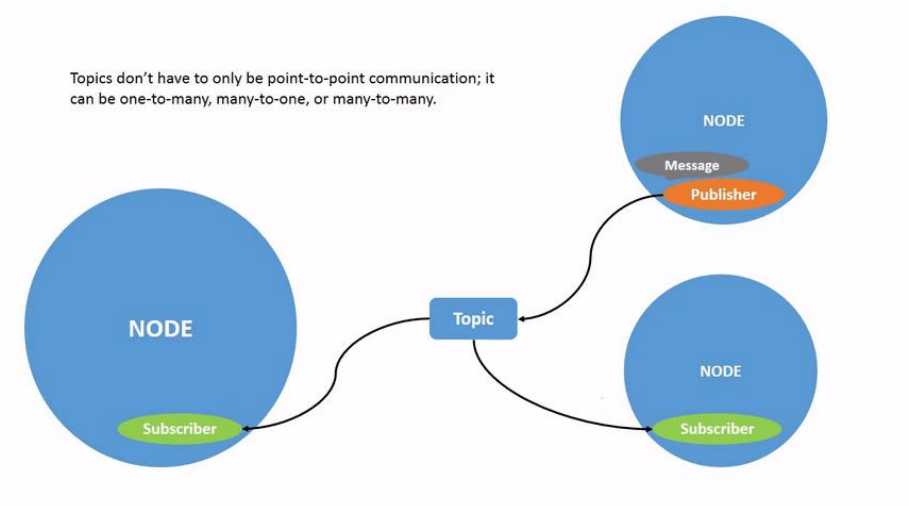


Figura 1.3: Funzionamento dell'infrastruttura Topic di ROS

1.2.3 ROS Services

Tale infrastruttura di comunicazione si basa su un modello di call-and-response. Un servizio è collegato ad uno o più nodi "Client", che effettuano un'azione di Call, richiedendo un'informazione, e ad un nodo "Server", che si occupa di fornire una Response al nodo che ha effettuato la Call. La Response passa comunque dal Service, che la indirizza verso il Client giusto.

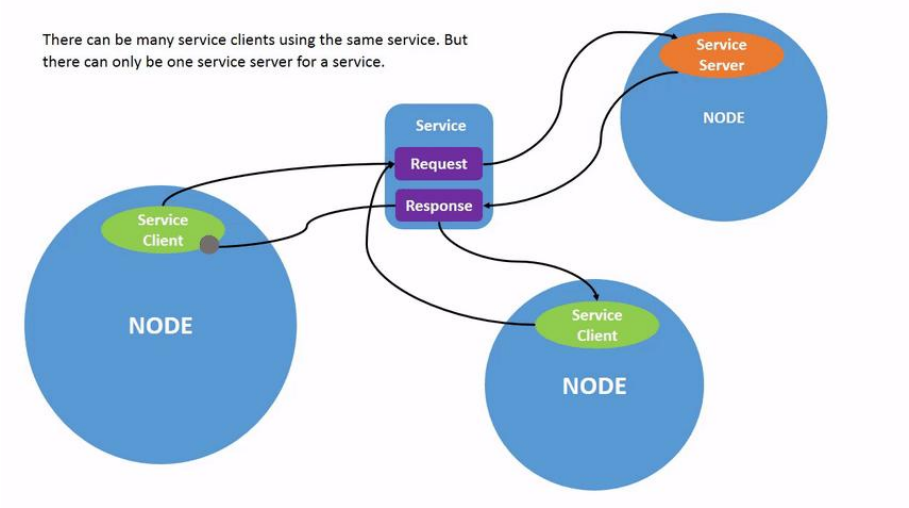


Figura 1.4: Funzionamento dell'infrastruttura Service di ROS

Come è possibile notare in Figura 1.4, ogni Service può avere nodi Client multipli, ma sempre un solo nodo Server.

1.2.4 ROS Parameters

Tali valori sono associati ai nodi del sistema e rappresentano le impostazioni dei nodi. I parametri di ogni nodo possono essere manipolati dinamicamente a runtime utilizzando i seguenti comandi:

```
# elencare tutti i parametri
ros2 param list

# mostrare il valore relativo ad un parametro specifico
ros2 param get <node_name> <parameter_name>

# impostare il valore di un parametro specifico
ros2 param set <node_name> <parameter_name> <value>

# salvare i valori dei parametri relativi ad un nodo in un file
ros2 param dump <node_name>

# caricare da file i dati di un parametro
ros2 run <package_name> <executable_name> --ros-args --params-file
    <file_name>
```

1.2.5 ROS Actions

Le ROS Actions sfruttano la struttura dei Topic e dei Services. Il loro funzionamento è simile a quello dei Services, infatti anch'esse si basano su Client e Server, ma il loro meccanismo è più complesso. Infatti le Actions aggiungono la possibilità di prelazione e feedback:

- Il nodo "Action Client" invia una richiesta (Goal) al nodo "Action Server";
- Il nodo "Action Server", mentre sta eseguendo la richiesta del Client, invia uno stream di feedback, e un "Result" alla conclusione

La particolarità delle Actions consiste nella possibilità di interrompere un Goal: il Goal può essere interrotto sia dal Client, inviando una richiesta di cancellazione al Server, che dal Server stesso, il quale ricevendo un nuovo Goal, inizia subito ad elaborare la nuova richiesta.

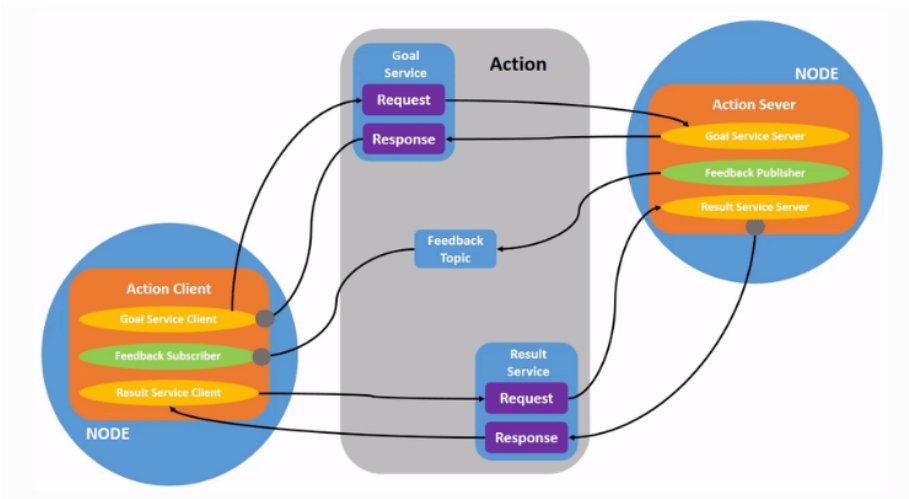


Figura 1.5: Funzionamento dell'infrastruttura Actions di ROS

1.2.6 Perché Utilizzare Entrambe le Versioni di ROS

In questo elaborato le versioni 1 e 2 di ROS sono state utilizzate in contemporanea, integrandole e sfruttando i pregi di entrambe. La ragione principale per cui è stato scelto questo tipo di configurazione risiede nella compatibilità e nell'ottimizzazione del sistema.

Perché utilizzare ROS1? La versione 1 di ROS ha molti anni di sviluppo alle spalle. Ciò significa che i driver relativi ai sensori e i pacchetti che implementano algoritmi di Visual Odometry e Sensor Fusion sono disponibili in numero più elevato. Oltre a ciò, l'algoritmo per la Visual Odometry utilizzato in questo elaborato (RTAB-Map), con il quale è possibile sfruttare appieno le potenzialità della RGB-D camera utilizzata per l'applicazione su veicolo reale, non è stato ancora sviluppato per la versione 2 di ROS. Pertanto, per questa parte del software, ROS1 rappresenta la scelta più adatta ed efficace.

Perché utilizzare ROS2? Il progetto ROS, partito nel 2007, è in continua evoluzione. L'obiettivo di ROS2 è portare un miglioramento sostanziale all'interno del framework, agendo su quelle parti del software che non è possibile aggiornare, se non con un cambio radicale di versione. Infatti non sarebbe possibile agire su parti come la modalità di esecuzione di nodi e l'aggiunta di strutture sulla stessa versione, pena la perdita di compatibilità con i pacchetti fino ad adesso sviluppati. Si è pertanto scelto di lasciare ROS1 così com'è in modo da non essere influenzato dallo sviluppo di ROS2.

Nonostante sia un sistema relativamente nuovo, ROS2 possiede già vari software compatibili con esso. Uno su tutti è Navigation2, il pacchetto per la navigazione

autonoma utilizzato in questa tesi. Esso possiede un livello di stabilità migliorato grazie al nuovo framework, con l'introduzione ad esempio dei lifecycle nodes, pertanto per questa parte del software è stata utilizzata la versione 2 di ROS (Foxy).

Per poter comunicare tra loro è stato implementato un bridge, ovvero un nodo che si occupa di trasferire i messaggi creati dai nodi da un framework all'altro, rendendoli così disponibili a tutto il sistema. Il funzionamento dettagliato del `ros1_bridge` e della comunicazione tra ROS1 e ROS2 sarà illustrata nei paragrafi successivi.

1.3 ROS1 Bridge: Introduzione e Feature Principali

In questa sezione si tratterà lo strumento `ros1_bridge`, molto utile nel caso si vogliano mettere in comunicazione pacchetti ROS che vengono eseguiti in ROS1 con altri pacchetti ROS compatibili con ROS2.

Il `ros1_bridge` viene installato e lanciato come nodo di ROS2 tramite l'apposito comando `ros2 launch`. La sua particolarità è di riuscire a leggere anche topic che vengono pubblicati all'interno dell'ambiente ROS1, purché quest'ultimo sia eseguito nella stessa macchina o comunque all'interno della stessa rete locale. Questa capacità, dovuta al particolare metodo di compilazione che sarà mostrato nell'Appendice A.5, gli permette di mettere in comunicazione le due versioni del framework, che altrimenti agirebbero come due compartimenti stagni. Il compito del `ros1_bridge` è quello di leggere tutti i messaggi pubblicati nei topic di ROS1 e pubblicarli, effettuando le dovute conversioni di tipo, se necessario, negli opportuni topic di ROS2, e viceversa.

Il bridge esegue un confronto tra i tipi di messaggi di ROS1 e quelli di ROS2 ed effettua un mapping automatico, per convertire i messaggi compatibili con una versione in messaggi adatti all'altra versione. Le regole di mappatura automatica vengono applicate in base ai nomi: quando viene trovata un'associazione la mappatura viene applicata. Inoltre, se si vogliono gestire tipi di messaggi non standard, o che hanno nomi molto differenti tra le due versioni di ROS, è possibile fornire regole di mappatura personalizzate.

Metodo di Associazione Auomatica dei Messaggi La mappatura automatica tra i messaggi viene effettuata basandosi sui nomi. In primo luogo i pacchetti ROS1 che terminano con il suffisso `_msgs` sono associati ai loro corrispondenti in ROS2 che terminano con il suffisso `_msgs` o `_interfaces`. In secondo luogo vengono associati tra loro i messaggi con lo stesso nome. Infine vengono associati tra loro i fields dei vari messaggi.

Se il nome del pacchetto, il nome del messaggio e tutti i nomi e tipi di campo sono gli stessi per un pacchetto di ROS1 e il corrispondente pacchetto di ROS2, soddisfando anche la condizione relativa ai suffissi sopra citata, i messaggi vengono automaticamente associati senza la necessità di aggiungere ulteriori specifiche.

Metodo di Associazione Personalizzata dei Messaggi Nel caso in cui sia necessario creare una regola ad hoc per l'associazione di due tipi di messaggi, può essere fatto utilizzando un file `yaml`. Esistono tre principali tipi di regole:

- package mapping rule: associa pacchetti di ROS1 e pacchetti di ROS2 e può quindi essere definita dai seguenti attributi:

  `ros1_package_name`

`ros2_package_name`

- message mapping rule: è definita dagli attributi della relativa package mapping rule e dai seguenti attributi specifici:

  `ros1_message_name`

  `ros2_message_name`

- field mapping rule: è definita dagli attributi della relativa message mapping rule e da un dizionario che mappa le caratteristiche dei campi del messaggio in ROS1 in quelle del messaggio in ROS2. Tale dizionario viene chiamato `fields_1_to_2`.

Tali regole di associazione tra pacchetti, messaggi e campi devono essere necessariamente create prima di eseguire il build del pacchetto `ros1_bridge` e nel caso questo sia già stato installato, nel momento in cui si creano nuove regole, la build va rieseguita. Al fine di rendere disponibile il file `yaml` creato al processo di build, è necessario esportarlo nel file `package.xml`, inserendo in tale file le seguenti righe di comando:

```
<export>
<ros1_bridge mapping_rules="my_mapping_rules.yaml"/>
</export>
```

I file `yaml` devono anche essere installati all'interno del file `CMakeLists.txt`:

```
install(
FILES my_mapping_rules.yaml
DESTINATION share/${PROJECT_NAME})
```

Queste operazioni sono molto importanti per permettere al bridge di riconoscere e mappare nella maniera corretta i messaggi presenti. Pertanto, poiché è necessario configurare il sistema in fase di compilazione, è preferibile conoscere in anticipo i tipi di messaggi che si desidera passare da un framework all'altro, così come i pacchetti che si prevede di utilizzare nel sistema finale.

1.4 Navigation2: Introduzione e Feature Principali

Il progetto Navigation2 rappresenta l'evoluzione del precedente Navigation Stack implementato con il framework ROS. Con l'uscita di ROS2, anche il Navigation Stack ha subito un aggiornamento.

Navigation2 è un pacchetto che implementa una serie di task, che lavorano all'interno dell'ambiente ROS2 e che si occupano di guidare un robot da un punto A verso un punto B, possedendo in memoria una mappa e orientandosi grazie a dei sensori, come LIDAR o camere. Navigation2 riceve in input le informazioni dai sensori ed elabora un segnale di velocità da inviare al robot pubblicando messaggi nel topic `cmd_vel`. I comandi da inserire nella shell per installare l'ambiente vengono mostrati di seguito:

```
sudo apt install ros-foxy-navigation2
sudo apt install ros-foxy-nav2-bringup
```

Il nodo centrale e più importante di Navigation2 è il "Behavior Tree" (BT), fornito in formato XML, che si occupa di prendere decisioni in conseguenza agli input che riceve. Il funzionamento del BT sarà illustrato nel dettaglio in seguito.

Gli strumenti principali di Navigation2 sono mostrati nell'elenco seguente, facendo riferimento ad un esempio di navigazione con Turtlebot3. Il comando necessario ad installare tale robot è il seguente:

```
sudo apt install ros-foxy-turtlebot3*
```

- Map Server: si occupa di salvare, memorizzare e rendere disponibili le mappe necessarie alla navigazione. Fornisce dati al topic "`/map`". Da tale topic vengono prese informazioni sulla mappa dai nodi di navigazione "AMCL" e dal programma RViz. Inoltre viene creata la "Global Costmap".

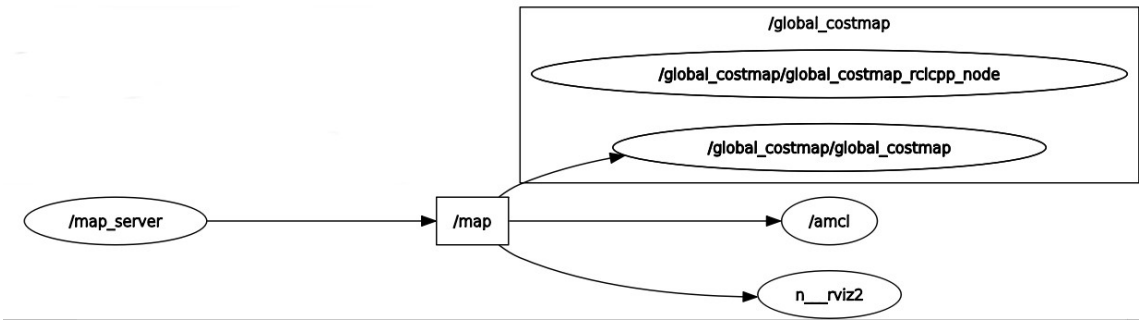


Figura 1.6: rqt_graph del topic `/map`

- AMCL: nodo che localizza il robot nella mappa. Legge le informazioni dai topic `/scan`, `/odom` e `/map` e ricava la posizione del robot;
- Nav2 Planner: pianifica un percorso da un punto A ad un punto B evitando gli ostacoli. La posizione degli ostacoli viene rilevata attraverso la mappa ed il sensore LIDAR in tempo reale. In particolare, il percorso da seguire viene

individuato sfruttando le informazioni contenute nella global costmap. Il nodo `/planner_server_rclcpp_node` comunica con il BT_Navigator attraverso la action `/compute_path_to_pose`:

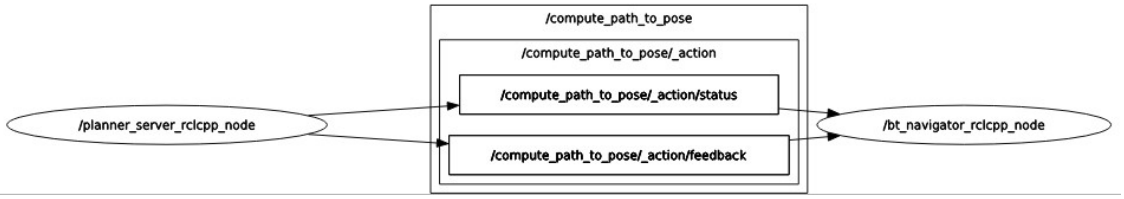


Figura 1.7: rqt_graph del nodo `/planner_server_rclcpp_node`

- Nav2 Controller: è il nodo che si occupa, dato il path da seguire, di regolare la velocità del robot per permettergli di seguire la traiettoria pianificata dal planner. Sfrutta la local costmap per capire se la posizione in cui si trova potrebbe essere una posizione dannosa e comunica con il recovery server in caso di necessità;
- Nav2 Costmap: converte i dati provenienti dalla mappa e dai sensori, in questo caso dal laser scanner, in una rappresentazione sotto forma di costmap[1]. Ne viene mostrato un esempio di seguito, dove i colori più scuri rappresentano un costo più alto:

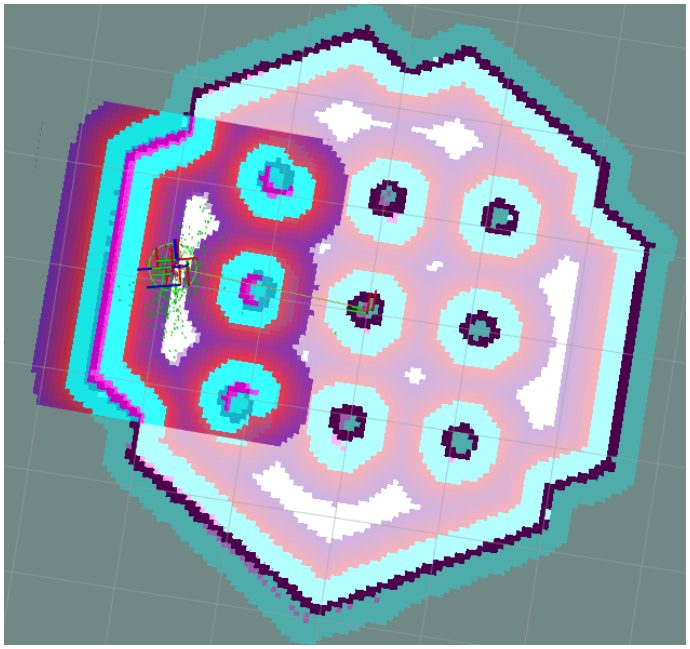


Figura 1.8: Esempio di costmap, associata all'esempio del Turtlebot3

- Nav2 Behavior Trees e BT Navigator: i behavior trees rappresentano il sistema decisionale del robot. Leggono i dati pubblicati nei principali topic di navigazione e in base a tali dati ricavano un comportamento da seguire.

[1]Una costmap rappresenta una mappa in cui ad ogni area viene associato un costo differente, in base alla difficoltà che incontrerebbe il robot passando all'interno di quell'area. Più difficile è l'attraversamento di un'area della mappa, più alto sarà il costo ad essa associato.

- Nav2 Recoveries: questo nodo si occupa delle procedure di emergenza da eseguire nel caso in cui il robot si trovi molto vicino ad un ostacolo o si verifichi un errore;
- Nav2 Waypoint Follower: rappresenta una feature capace di navigare verso waypoint multipli in sequenza;
- Nav2 Lifecycle Manager: un nodo che gestisce i lifecycle del sistema ed è un watchdog per i vari server;
- Nav2 Core: insieme di plugin per attivare algoritmi e behavior personalizzati.

Di seguito viene mostrata un'immagine di insieme dei nodi principali del sistema:

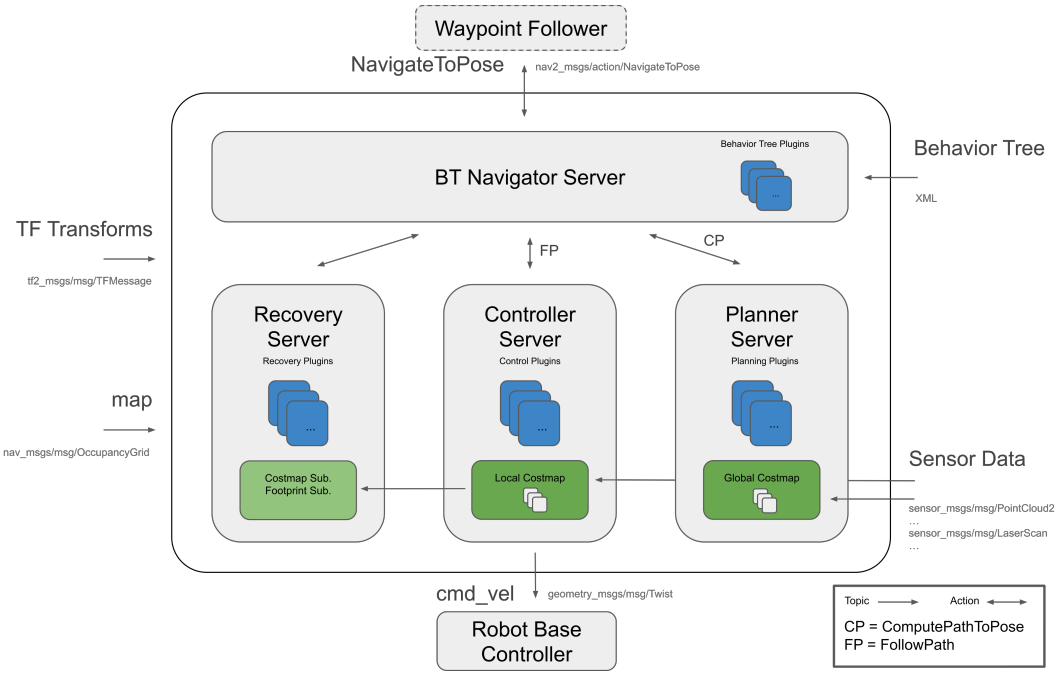


Figura 1.9: Mappa concettuale del sistema Navigation 2

1.4.1 Le ROS Actions all'interno di Navigation 2

In un ambiente in cui i compiti assegnati ai vari nodi possono essere lunghi, come ad esempio eseguire uno spostamento verso un nuovo waypoint, è molto utile avere degli Actions Server, che durante l'esecuzione del compito sono capaci di fornire dei feedback al nodo Client, mentre l'azione viene eseguita. Inoltre è possibile per il Client interrompere il task corrente per, ad esempio, assegnare un ulteriore nuovo waypoint al planner server. Quest'ultimo interrompe il task che stava eseguendo per pianificare un nuovo percorso. Feedback e risultati del goal assegnato possono essere raccolti in maniera sincrona registrando dei callback con gli Action Client, oppure in maniera asincrona.

All'interno del framework Navigation 2 sono presenti vari Action Server, che saranno illustrati di seguito:

- `/navigate_to_pose`: rappresenta l'azione che mette in comunicazione il Waypoint Follower (Client) con il BT_Navigator (Server), che durante la navigazione invia feedback sia al waypoint follower che al programma RViz:

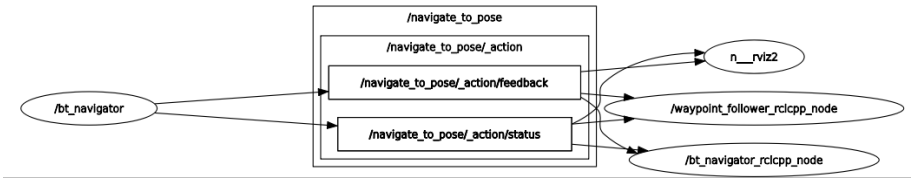


Figura 1.10: rqt_graph della action `/navigate_to_pose`

- `/follow_path`: il BT_Navigator in questo caso rappresenta il Client, che richiede informazioni al controller server sull'efficacia dell'inseguimento del path:

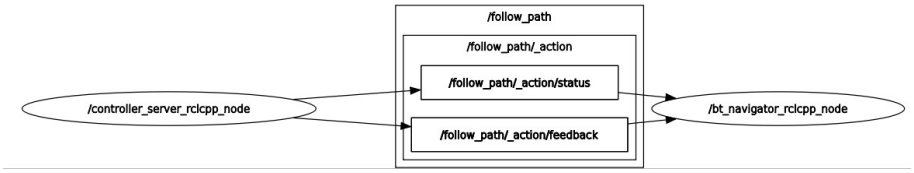
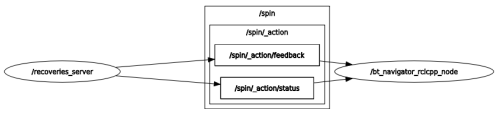


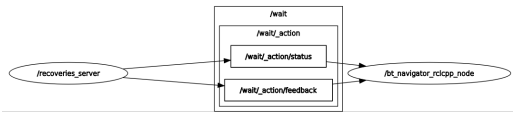
Figura 1.11: rqt_graph della action `/follow_path`

- `/spin`, `/wait`: tali actions collegano i nodi BT_Navigator (Client) e Recoveries_Server (Server). Il Recovery Server viene chiamato dal BT_Navigator quando il robot si posiziona in punti della costmap rischiosi. Questo si verifica ad esempio quando si rileva un ostacolo.

  Le diverse azioni di "recovery" vengono decise dal BT_Navigator, che si serve del Behavior Tree. Il BT_Navigator funge da action client: in base alla situazione in cui si trova il veicolo esso invia un Goal al server desiderato. Quando si trova in una situazione di emergenza, viene attivata la parte del behavior tree relativa al recovery server: viene richiesta in primis un'azione di spin, seguita da un'azione di wait. Se entrambe le azioni terminano con un Result positivo, il BT_Navigator chiama di nuovo in causa il Planner Server attraverso l'invio di un nuovo Goal e la navigazione può riprendere. Nel caso del progetto corrente l'azione di "spin", ovvero di girare su sé stesso, non è possibile. Infatti essa è un'azione tipica dei robot di tipo differenziale. Per il robot di tipo Ackermann in esame, è stato necessario sostituire l'algoritmo di spin con quello di "backup", appositamente modificato per essere eseguibile dal veicolo. Tale sostituzione è stata effettuata agendo sul behavior tree. La spiegazione dettagliata riguardo alla modifica effettuata sull'algoritmo di "backup" e sul behavior tree è illustrata nell'Appendice B.1



(a) rqt_graph della action `/spin`



(b) rqt_graph della action `/wait`

- `/compute_path_to_pose`: Tale azione viene richiesta dal BT_Navigator (client) al Planner Server, che ha il compito di elaborare il percorso da seguire:

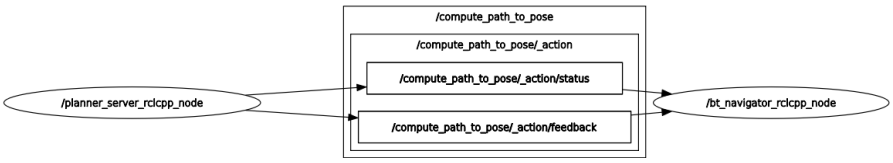


Figura 1.13: rqt_graph della action `/compute_path_to_pose`

- `/Follow_waypoints`: tale azione pone in collegamento il programma RViz con il waypoint follower. L'utente attraverso il programma sceglie un waypoint. RViz (client) richiede al waypoint follower (server) di navigare verso il waypoint.

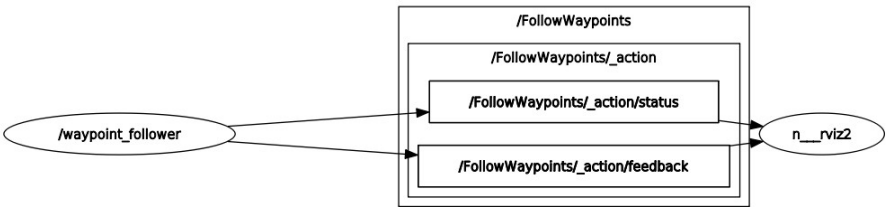


Figura 1.14: rqt_graph della action `/Follow_waypoints`

1.4.2 Lifecycle Manager

I nodi nell'ambiente Navigation 2 sono dei Managed Nodes, un tipo particolare di nodo di ROS 2. Il vantaggio di utilizzare dei managed nodes è di riuscire a controllare in maniera molto più efficace il comportamento del sistema. In ogni momento è possibile conoscere lo stato di un nodo, che ha un proprio lifecycle e può trovarsi nei seguenti stati principali:

- Unconfigured: stato in cui si trova un nodo appena istanziato;
- Inactive: stato che rappresenta un nodo inattivo ma che è già stato configurato ed è pronto ad essere attivato;
- Active: è lo stato principale del lifecycle di un nodo, in cui esso diventa operativo e interagisce con gli altri nodi tramite ROS;
- Finalized: stato in cui si trova un nodo in attesa di essere distrutto. L'unica azione possibile da questo stato è terminare il nodo.

Inoltre, per passare da uno stato ad un altro sono possibili 7 diverse transizioni:

- create
- configure
- cleanup

- activate
- deactivate
- shutdown
- destroy

Il seguente grafico mostra come le varie transizioni collegano i diversi stati del lifecycle di un nodo:

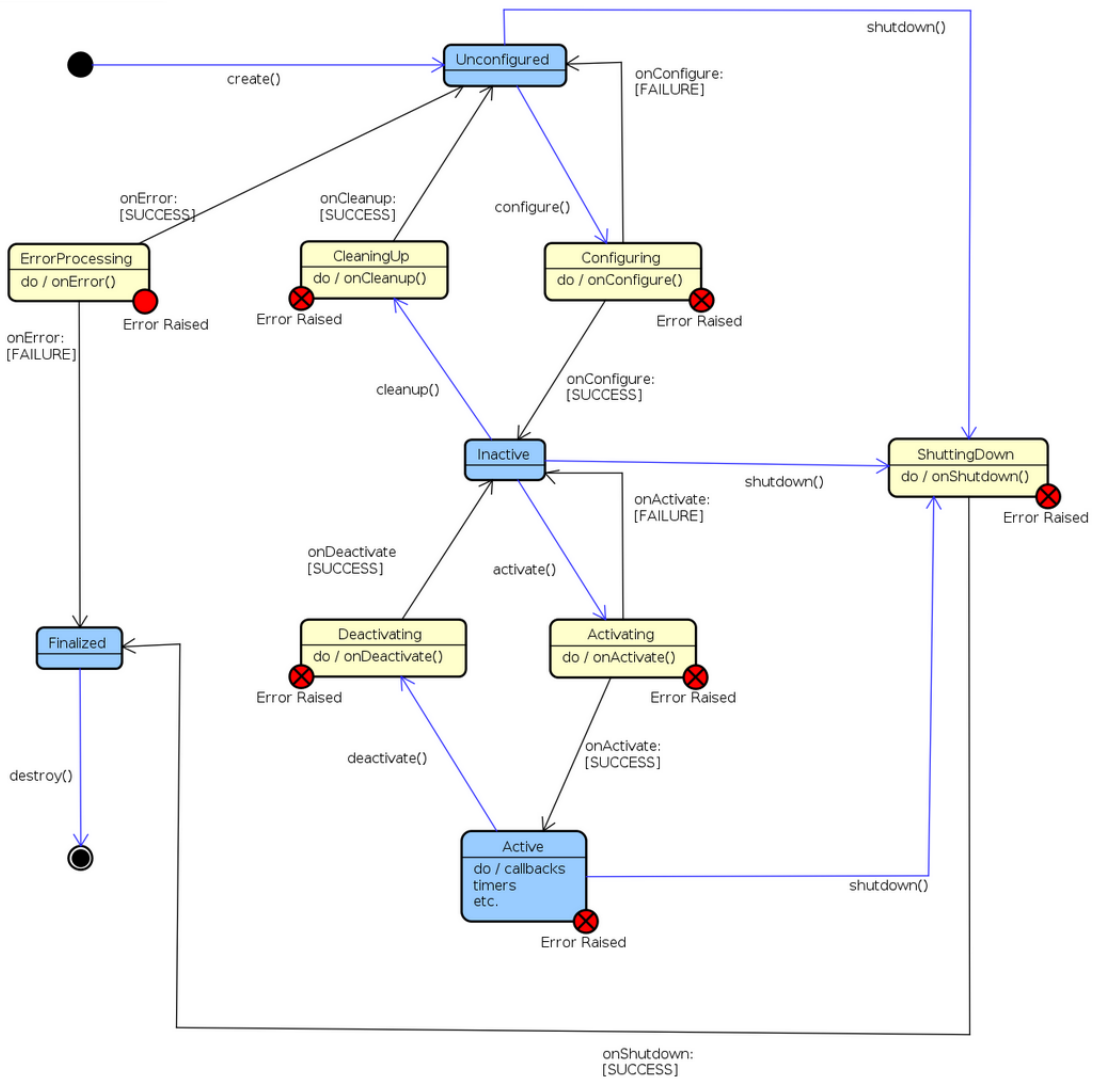


Figura 1.15: Grafo degli stati relativi al lifecycle di un nodo

Il Lifecycle Manager si occupa di configurare ed attivare correttamente tutti i nodi del framework Navigation 2 tramite i metodi `on_configure()` e `on_activate()` e di gestire ogni problema che un server potrebbe avere. Se si verifica un crash, il Lifecycle Manager effettua una transizione del sistema verso un altro stato per prevenire indesiderati failure.

1.4.3 Behavior Trees

I Behavior Trees (BT) rappresentano una struttura di task collegati attraverso una logica ben precisa, e sono la parte principale del framework Navigation 2 per quanto riguarda il sistema decisionale durante la navigazione. Un BT, rispetto ad esempio ad una macchina a stati finiti, rappresenta una soluzione molto più immediata nell'elaborazione e molto meglio comprensibile da un operatore umano. Per le applicazioni di navigazione robotica autonoma rappresentano la principale soluzione.

Il BT del sistema Navigation 2 è salvato all'interno di un file XML. Al caricamento del framework, l'albero viene caricato all'interno del nodo BT navigator, facendo un operazione di parsing del file XML. Una volta terminato il caricamento è possibile sfruttare il BT ed eseguire delle scelte per la navigazione.

1.4.4 Navigation Servers

All'interno dell'ambiente Navigation 2 sono presenti anche vari server necessari per svolgere i diversi compiti di navigazione. Il BT Navigator funge da client per tutti questi server che, in base alla scelta e al compito da eseguire in un determinato istante, invia delle richieste ai server designati a svolgere il compito specifico. All'interno di ogni server sono implementati più algoritmi, capaci di completare vari task. Quando il BT raggiunge un determinato nodo, chiama uno dei server tramite un'azione, al fine di completare il task richiesto. La scelta dell'algoritmo da utilizzare viene fatta dal BT, in base al tipo di callback inviato (es. `/spin` e `/wait`) al server.

In seguito si illustreranno tutte le funzioni che i navigation server sono in grado di svolgere.

Planner Server Il compito del Planner Server è di elaborare un percorso, in base al waypoint che viene ricevuto. Un planner può essere implementato al fine di soddisfare diversi tipi di funzioni obiettivo. Infatti il percorso può essere ottimizzato per:

- Calcolare il percorso più breve;
- Calcoalre il percorso che copre tutta l'area designata;
- Calcolare un percorso lungo strade predefinite.

L'algoritmo di planning scelto in questo elaborato è lo SMAC Planner. La scelta è ricaduta su di esso poiché risulta particolarmente adatto per il tipo di veicolo in questione, e permette di elaborare percorsi in base ai vincoli agenti.

Controller Server Se i planner sono necessari per calcolare un percorso su larga scala, il Controller Server si occupa di controllare i movimenti locali del robot, sfruttando i dati provenienti dai sensori. Il compito principale del controller server di Nav2 è assicurarsi che il robot riceva i giusti sforzi di controllo per riuscire a seguire il percorso globale impostato dal planner.

In particolare, in questa tesi è stato utilizzato il "Regulated Pure Pursuit Controller", che rappresenta un'ottima soluzione per i veicoli di tipo Ackermann. Esso infatti, elabora in maniera iterativa le velocità lineare e angolare da assegnare al veicolo per raggiungere un determinato punto che è stato ricavato in precedenza dal path.

Recovery Server Questa parte di Nav2 rappresenta il pilastro fondamentale per un sistema di tipo fault-tolerant. Infatti, gli algoritmi contenuti all'interno del Recovery Server sono necessari per eseguire tutte le procedure di emergenza per rendere di nuovo operativo il robot autonomo. Ad esempio, ogni qual volta il robot incontra un ostacolo dinamico non presente sulla mappa, il BT richiede una procedura di recovery all'omonimo server, che provvederà a portare il dispositivo in un punto sicuro in cui la normale navigazione potrà riprendere.

L'approccio con il Recovery Server utilizzato in questa tesi è stato di allontanare il veicolo dall'ostacolo nel momento in cui sia necessario. Quando entra in gioco il Recovery Server, viene eseguita una retromarcia per una distanza predefinita. Quando il veicolo è abbastanza lontano, il Planner Server può riprendere il controllo.

Il funzionamento dettagliato di questi algoritmi sarà illustrato nei paragrafi successivi.

1.4.5 Regulated Pure Pursuit Controller

Il "Regulated Pure Pursuit Controller" rappresenta una variante dell'algoritmo "Pure Pursuit Controller", il quale scopo è tracciare un percorso locale, dato un punto ad una distanza ravvicinata, preso dal path globale. È stata scelta questa variante dell'algoritmo poiché implementa termini di regolazione aggiuntivi sulla velocità lineare e per controllare un'eventuale collisione. Sono implementate anche le basi dell'algoritmo "Adaptive Pure Pursuit" che permettono di variare le "lookahead distances" in base alla velocità corrente.

Questo algoritmo implementa l'interfaccia del `nav2_core::Controller` per poter essere utilizzato all'interno di Navigation2 come azione del Controller Server. Come accennato in precedenza, l'idea alla base di questo algoritmo consiste nel trovare un punto nel path globale fornito dal Planner Server che si trovi di fronte al robot a breve distanza, per poi elaborare una traiettoria per raggiungere tale punto. Il processo si ripete in maniera iterativa finché il robot non raggiunge la destinazione finale. La figura seguente mostra tale processo:

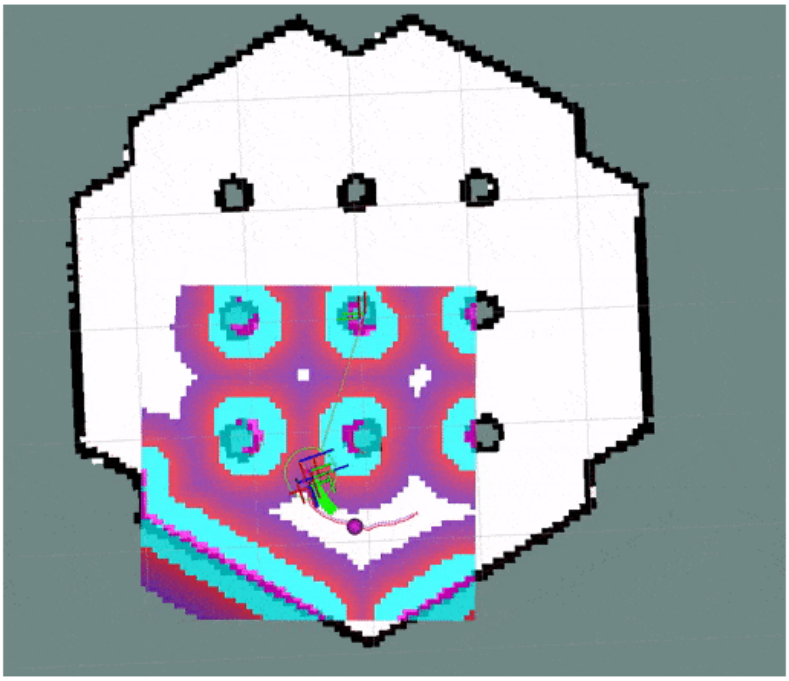


Figura 1.16: Illustrazione del punto sul global path e della local trajectory elaborata

La distanza dal robot alla quale trovare il punto da seguire è chiamata "lookahead distance", rappresenta un parametro dell'algoritmo e può quindi essere regolata a piacere.

Il global path è continuamente ridotto al punto più vicino al robot. Infatti, prima della scelta del lookahead point, la parte di global path presente all'interno della local costmap viene trasformata nel sistema di coordinate del robot e viene così ricavato il path che sarà fornito al pure pursuit controller (path di colore verde in Figura 1.17). Il punto che rappresenta l'intersezione tra il lookahead radius e il path di colore verde sarà il lookahead point. Il controller sfrutta questo punto per ricavare la curvatura necessaria a guidare il robot sul lookahead point. Tale curvatura viene utilizzata per ricavare valori di velocità lineare ed angolare da pubblicare all'interno del topic `cmd_vel`.

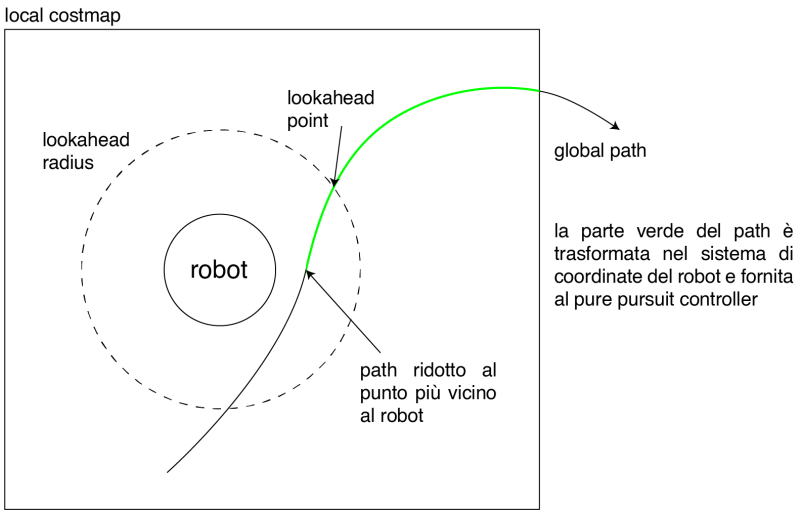


Figura 1.17: Metodo di ottenimento del lookahead point

È molto importante che anche il planner calcoli percorsi compatibili con i vincoli agenti sul veicolo. Se infatti il global path fornito dal planner avesse un raggio di curvatura troppo ridotto, il controller non riuscirebbe a trovare una coppia di velocità lineare e angolare capace di soddisfare tale curvatura. Lo SMAC Planner, che sarà illustrato nel seguente paragrafo, riesce a soddisfare tali requisiti poiché è anch'esso compatibile con i robot di tipo Ackermann.

Il controller, come del resto anche il planner, deve essere al corrente del tipo di veicolo che sta controllando e delle sue caratteristiche, per poter elaborare sforzi di controllo adeguati. È possibile infatti impostare dei parametri per descrivere nella maniera migliore il proprio robot. Per prima cosa è necessario impedire al controller di provare a far ruotare su sé stesso il veicolo. Infatti, essendo un robot di tipo Ackermann, questo non avrebbe senso. È possibile farlo impostando al valore `false` il parametro `use_rotate_to_heading`.

La lookahead distance può essere regolata in base alle dimensioni del robot e soprattutto in base alla frequenza con la quale si vuole aggiornare lo sforzo di controllo. Se si utilizza una lookahead distance ridotta, i relativi lookahead points saranno più vicini tra loro e di conseguenza ci sarà un maggior numero di iterazioni nel calcolo delle velocità.

Collision Detection Nell'algoritmo è implementata anche una collision detection attiva. In base alle velocità lineare e angolare correnti, si ricava la posizione che avrà il robot proiettato in avanti di un certo tempo e si controllano possibili collisioni. Questo approccio consente di controllare a distanze più elevate quando le velocità sono più elevate e a distanze minori se le velocità sono ridotte.

Modifica dei parametri Per poter cucire l'algoritmo su misura del veicolo di questa tesi, è stato necessario modificare alcuni parametri. Tali parametri, così come quelli degli altri nodi di Navigation2, sono raggruppati all'interno del file `nav2_params.yaml`. Accedendo a questo file è possibile cambiare tutti i parame-

tri necessari. Di seguito viene mostrata una tabella con le modifiche apportate al Regulated Pure Pursuit Controller e il relativo significato:

Nome parametro	Valore	Significato
`use_rotate_to_heading`	false	Stabilisce se il robot può ruotare su sé stesso
`xy_goal_tolerance`	0.3m	Tolleranza (x, y) nel posizionamento finale
`yaw_goal_tolerance`	0.3rad	Tolleranza di "yaw" nel posizionamento finale
`lookahead_dist`	0.6m	Lookahead distance

Tabella 1.1: Parametri dell'algoritmo di Regulated Pure Pursuit Controller

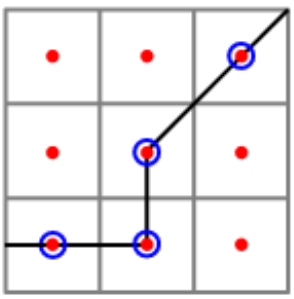
1.4.6 SMAC Planner

SMAC Planner è un plugin per il Planner Server di Navigation2. Esso contiene due plugin distinti, chiamati "SmacPlannerHybrid" e "SmacPlanner2D". Il primo dei due è compatibile con i veicoli di tipo Ackermann, pertanto è stato utilizzato per la programmazione del planner server di questa tesi e sarà illustrato nelle righe a seguire.

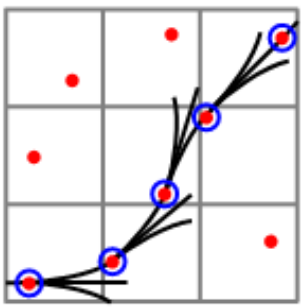
Il plugin "SmacPlannerHybrid" implementa al suo interno l'algoritmo Hybrid-A*, rendendolo di fatto compatibile con l'ambiente Navigation2 e in particolare con il Planner Server che dialoga con il resto del sistema.

Hybrid-A* Planner

Questo algoritmo è suddiviso principalmente in due parti: il primo step usa una variante di A* Search, un algoritmo di ricerca su grafi che individua un percorso da un dato nodo iniziale verso un dato nodo goal, mentre la seconda parte migliora la qualità della soluzione con una ottimizzazione numerica non lineare. Tale variante è applicata allo spazio di stato cinematico del veicolo, in modo tale da non considerare soltanto il centro di ogni cella ma permettere al veicolo di eseguire una traiettoria dolce, come mostrato in figura.



(a) Funzionamento convenzionale



(b) Esempio della variante

Figura 1.18: Esempi dell'algoritmo A* a confronto

Come nel caso convenzionale, lo spazio di ricerca (x, y, θ) è discretizzato, ma tale variante non permette la visita del solo centro delle celle. Infatti a ogni cella viene associato uno stato continuo in 3D del veicolo. Questo algoritmo non garantisce di trovare il percorso migliore in termini di distanza, ma garantisce che il percorso sia guidabile dal veicolo. Ci sono vari vincoli che è possibile impostare attraverso dei parametri: ad esempio il minimo raggio di curvatura è un parametro fondamentale per rispettare i vincoli fisici a cui è soggetto il veicolo Ackermann. Nella pratica, la soluzione trovata dall'algoritmo Hybird-A* staziona quasi sempre nelle vicinanze dell'ottimo globale. Questo permette alla seconda fase dell'algoritmo di trovarlo e di arrivare così alla soluzione ottima. L'algoritmo infatti, nella sua seconda fase utilizza un "gradient descent" per arrivare gradualmente a trovare l'ottimo globale, sempre che esso lo sia. L'algoritmo può pianificare percorsi sia in avanti che in retromarcia, permettendo al veicolo di districarsi anche nei casi in cui gli spazi a disposizione risultano ridotti.

Euristiche L'algoritmo è guidato da due euristiche. La prima ignora gli ostacoli ma tiene in considerazione la natura anolonoma del veicolo. In prima istanza quindi, si considera il punto di Goal $(x_g, y_g, \theta_g) = (0, 0, 0)$ e si calcola il percorso più breve verso il Goal a partire da qualsiasi punto (x, y, θ), assumendo completa assenza di ostacoli. L'effetto di questa euristica è di scartare rami di ricerca che approcciano il goal in maniera sbagliata. Dualmente, la seconda euristica tiene conto degli ostacoli presenti lungo il percorso ma ignora la natura anolonoma del veicolo. Questo approccio permette quindi di evitare vicoli ciechi e tutte le aree in cui sono presenti ostacoli.

Viene infine illustrato il parametro `minimum_turning_radius`. Esso consente di far rispettare all'algoritmo i limiti fisici del proprio veicolo Ackermann, ma può anche essere utilizzato con un robot di tipo differenziale per rendere più dolci le traiettorie percorse. Ovviamente bisogna anche tenere conto dell'ambiente in cui ci si sta muovendo: se il robot si trova in spazi stretti, il minimo raggio di sterzata non può essere elevato, altrimenti non sarà possibile trovare alcun percorso.

Modifica dei parametri Anche in questo caso è stato necessario modificare alcuni parametri dell'algoritmo. In particolare si è posto attenzione alle dimensioni del veicolo ed al suo raggio di sterzata:

Nome parametro	Valore	Significato
`tolerance`	0.5m	Tolleranza rispetto alla posa finale del veicolo
`minimum_turning_radius`	0.4m	Minimo raggio di sterzata
`reverse_penalty`	2.1	Penalità da applicare se viene scelto un movimento in retromarcia
`change_penalty`	0.15	Penalità da applicare se è presente un cambio di direzione
`non_straight_penalty`	1.5	Penalità da applicare se il moto non è in linea retta
`cost_penalty`	1.7	Penalità da applicare alle aree ad alto costo, cioè dove sono presenti ostacoli

Tabella 1.2: Parametri dell'algoritmo di SMAC Planner

1.4.7 AMCL

AMCL, acronimo di Adaptive Monte Carlo Localization, è un algoritmo che permette la localizzazione del veicolo all'interno di una mappa. Viene lanciato come nodo ROS da Navigation2 e permette, attraverso l'utilizzo di un particle filter, di ottenere la posizione e l'orientamento del veicolo all'interno della mappa, prendendo in input informazioni di odometria, tramite algoritmo di Sensor Fusion mostrato nella Sezione 1.7, scansioni LIDAR, la quale implementazione in simulazione viene illustrata in Appendice A.3, ed ovviamente, di una mappa.
I dati di odometria, come già visto nei precedenti paragrafi, sono affetti da un errore incrementale, che si propaga ed aumenta nel tempo. L'algoritmo AMCL sfrutta questi dati per ottenere una trasformazione TF dal frame `map` al frame `odom` e localizzare così il veicolo all'interno della mappa. Lo strumento utilizzato è il particle filter, che divide lo spazio di stato, ovvero l'insieme dei punti all'interno della mappa in cui il veicolo può trovarsi, in un insieme di particelle. In questo modo si entra in un dominio discreto e il numero di stati diventa finito. In base agli input ricevuti, AMCL stima una probabilità che il veicolo si trovi in un determinato stato discreto e, all'aumentare degli input, riesce ad ottenere una stima sempre più precisa della posa del veicolo all'interno della mappa. In Figura 1.19 viene mostrata la conversione tra stato continuo e stato discreto. Per semplicità, nel grafico è stato mostrato soltanto uno spostamento lungo una direzione (x) e ad esso è associata una probabilità per ogni posizione.

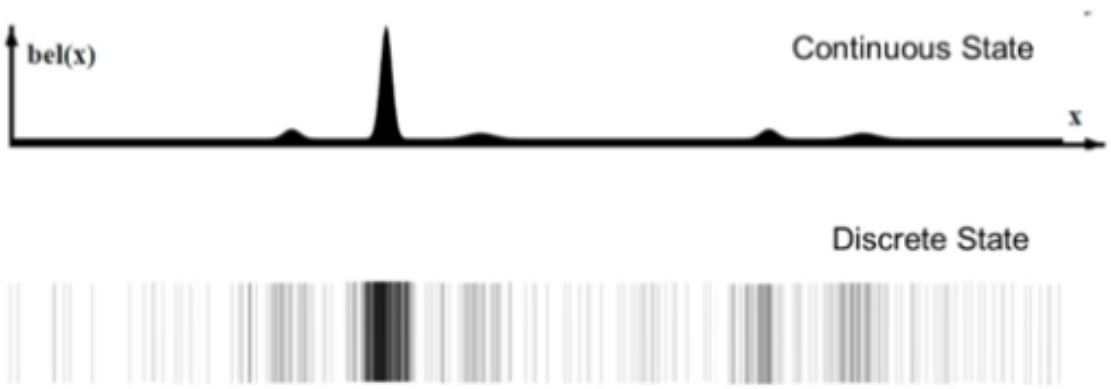


Figura 1.19: Probabilità del robot di trovarsi in una determinata posizione

Il particle filter riesce anche a minimizzare l'errore commesso dall'odometria. Infatti, nel tempo i valori di odometria risultano sempre meno accurati ed un intervento di correzione è sempre più necessario. A partire dai dati in input all'algoritmo, si ottiene un set di particelle, ognuna corrispondente ad una posa stimata del veicolo. Rispetto ad ogni particella si orientano le misurazioni fatte dal laser scanner e si calcola un fattore di correlazione che hanno queste ultime con i bordi della mappa, come in figura:

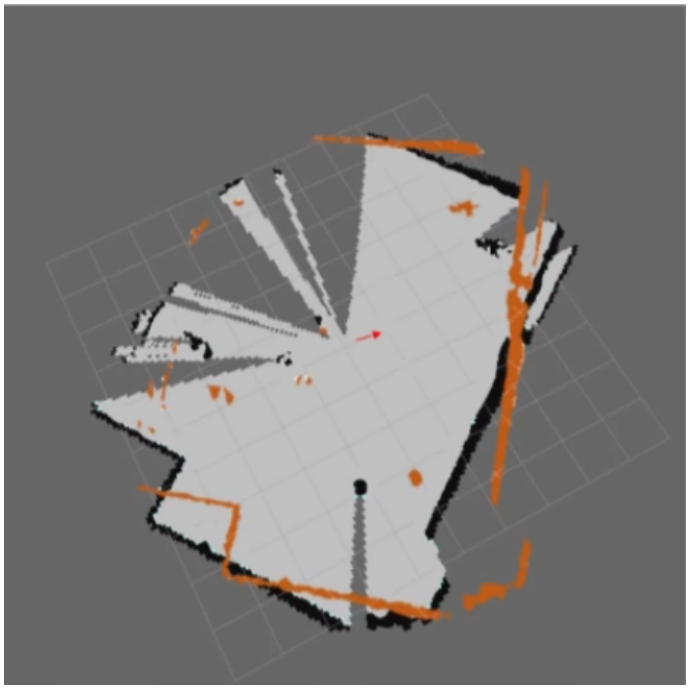


Figura 1.20: Comparazione tra misurazioni laser (arancio) e confini nella mappa (nero)

A queste misurazioni è associato un punteggio di correlazione, grazie al quale è possibile selezionare la posa ottimale. Tale punteggio, denominato con S, può essere calcolato come segue:

$$S = \frac{\sum_m \sum_n (A_{mn} - \bar{A})(B_{mn} - \bar{B})}{\sqrt{\left(\sum_m \sum_n (A_{mn} - \bar{A})^2\right)\left(\sum_m \sum_n (B_{mn} - \bar{B})^2\right)}} \tag{1.1}$$

dove A_{mn} rappresenta il pixel della mappa che si trova nella coordinata (m, n), mentre B_{mn} è il corrispettivo valore riguardante la scansione laser. $\bar{A}$ e $\bar{B}$ rappresentano i rispettivi valori medi della posizione dei pixel. I pixel relativi alle pareti hanno il valore 1, mentre quelli relativi a spazi liberi risultano 0. Sostanzialmente, facendo riferimento alle Figure 1.20 e 1.21 il risultato S dell'equazione 1.1 sarà proporzionale al numero di pixel di colore arancio che si sovrappongono con quelli di colore nero. L'equazione viene calcolata per ogni particella ed al termine, sarà scelta la posa in cui si avrà la miglior sovrapposizione:

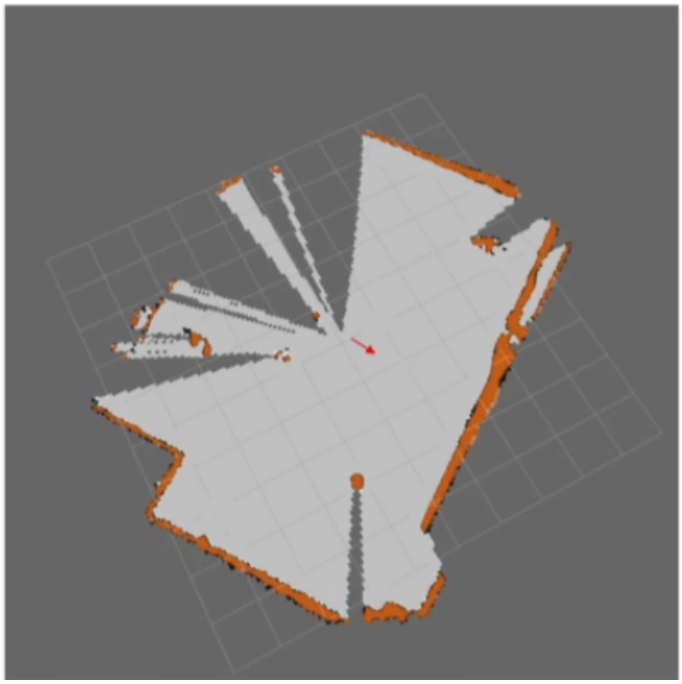


Figura 1.21: Particella con coefficiente di correlazione maggiore

In questo modo, AMCL riesce a correggere il drift nel posizionamento commesso dall'odometria ed a riposizionare correttamente il veicolo all'interno della mappa.

1.4.8 Backup Recovery

L'algoritmo di backup recovery viene chiamato in causa quando il veicolo si trova in una situazione di emergenza, ad esempio se si trova vicino ad un ostacolo e per il planner è impossibile elaborare un nuovo percorso. Se questo succede, il BT_Navigator invoca il Recovery Server inviando un Goal e inizia l'azione di backup. Tale azione consiste nel procedere in retromarcia per una distanza prefissata, che in questo progetto corrisponde a 30cm. In particolare, nel momento in cui l'algoritmo viene lanciato, il veicolo si trova nella posizione cartesiana $(x, y) = (0, 0)$ e il suo obiettivo è raggiungere la posizione $(x, y) = (-0.3, 0)$. Occorre notare che le coordinate cartesiane sono espresse rispetto al veicolo, quindi imporre che si debba percorrere -0.3m lungo l'asse x equivale ad imporre al veicolo di spostarsi di 30cm in

retromarcia lungo il proprio asse, senza sterzare. Una volta che questo spostamento è compiuto, il veicolo si trova a 30cm dall'ostacolo che ha interrotto la sua corsa, pertanto risulta possibile per il planner elaborare un nuovo percorso. Tale distanza è stata scelta in base alle caratteristiche fisiche del veicolo in questione, il quale raggio di sterzata risulta compatibile con gli spazi creati dal backup recovery.

1.4.9 Stima dello Stato

Per stimare lo stato del robot, ovvero la sua posizione all'interno della mappa, sono necessarie due trasformazioni principali:

- `map` → `odom`
- `odom` → `base_link`

La prima trasformazione viene fatta attraverso un sistema di Global Positioning, come ad esempio GPS o SLAM. In questa tesi il nodo di Nav2 adibito a questa trasformazione è rappresentato da `/amcl`, una tecnica di localizzazione basata su particle filter.

La seconda trasformazione viene eseguita dall'odometria. L'obiettivo dell'odometria è di ricavare localmente un frame, basandosi sul movimento del robot. Tale parametro può provenire da varie sorgenti, come LIDAR, encoder, IMU, sistemi di visione, che lavorando insieme possono fornirne un valore più accurato. Una volta ricavato un local frame, il sistema di global positioning aggiornerà il global frame in base al local frame. Nel corrente elaborato la trasformazione `odom` → `base_link` è stata eseguita grazie ad una RGB-D camera, servendosi perciò di un algoritmo di Visual Odometry, sincronizzato tramite un algoritmo di Sensor Fusion con i dati provenienti da una IMU.

1.4.10 Rappresentazione dell'Ambiente

Questa sezione tratta di come il robot riesce a percepire l'ambiente ed in particolare a distinguere le zone "pericolose" da quelle "sicure".

La rappresentazione dell'ambiente circostante viene effettuata tramite l'utilizzo di una costmap, che consiste in una griglia 2D di celle ad ognuna delle quali è associato un costo, che può essere: sconosciuto, libero, occupato, intermedio. Tale costmap (global), ottenuta consultando la mappa dell'ambiente, viene consultata dal planner per ricavare oltre che un percorso breve, anche sicuro e libero da collisioni. Una local costmap viene anche ricavata in tempo reale per calcolare gli sforzi di controllo locali o trovare ostacoli lungo il percorso non presenti nella mappa.

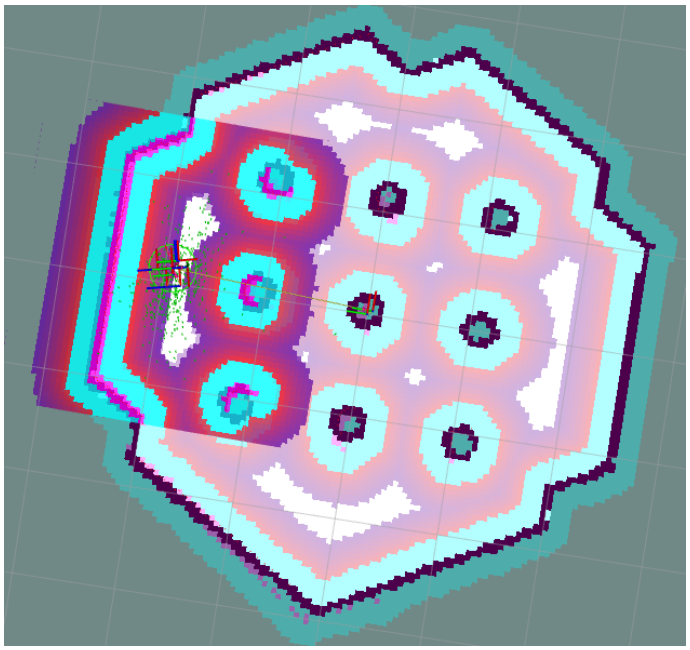


Figura 1.22: Esempio di costmap, associata all'esempio del Turtlebot3

Costmap Filters Tali elementi integrano la costmap e sono necessari per individuare zone particolari, come ad esempio aree da evitare o nelle quali si deve limitare la velocità del robot. Tale mappa rappresenta una "filter mask". I costmap filters sono implementati come plugin e si occupano di filtrare la costmap, basandosi sulle informazioni contenute all'interno della filter mask. Al fine di creare una costmap filtrata, viene effettuata una trasformazione lineare dei dati contenuti all'interno della filter mask e viene così creata una mappa delle feature. Fondendo le informazioni presenti nella costmap e nella mappa delle feature appena ottenuta, è possibile cambiare il comportamento del robot in base a dove si trova.

1.5 Veicoli Ackermann: Introduzione

Quasi tutti i veicoli a quattro ruote diffusi nel mondo, a partire dal ventesimo secolo, sono dotati di una tecnologia comune: il cinematismo di sterzo "Ackermann". Questa invenzione trova le sue radici quasi due secoli prima, quando nella seconda metà del 1700 fu applicato per la prima volta a carrozze trainate da cavalli. L'uomo che per primo pensò a questa soluzione si chiamava Erasmus Darwin. Filosofo, medico e naturalista, Darwin scelse di modificare il sistema sterzante delle sue carrozze, realizzato con la rotazione dell'intero assale anteriore, con le ruote vincolate ad esso. Il nuovo sistema, nonostante fosse molto all'avanguardia, non fu ben visto dalle cause costruttrici, che consideravano troppo complesso realizzarlo e manutenerlo in caso di guasto. L'idea fu ripresa 15 anni dopo, dall'artigiano Georg Lankensperger, il quale decise di depositare il brevetto in Inghilterra, affidando il compito ad un suo amico pubblicista, Rudolph Ackermann. Al momento di depositare il brevetto, Ackermann fornì il suo nome attribuendolo così all'invenzione.

Prima di vedere questa tecnologia applicata su scala industriale, passarono altri sessant'anni, quando l'ingegnere Charles Jeantaud perfezionò il brevetto di Ackermann, introducendo un quadrilatero nel sistema sterzante, rendendolo adatto ai veicoli dotati di trazione propria che iniziarono a diffondersi in quegli anni. In Figura 1.23 viene mostrato tale modello, utilizzato ancora oggi nelle automobili moderne.

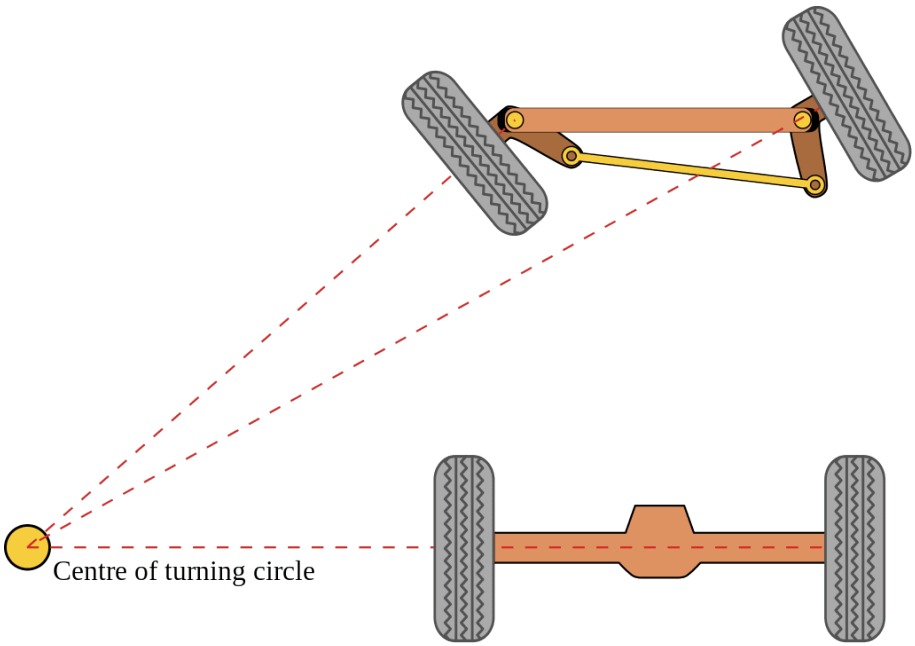


Figura 1.23: Modello sterzante Ackermann, particolare del quadrilatero

1.5.1 Modello Matematico

In questa sezione sarà trattato in dettaglio il funzionamento del modello sterzante Ackermann e, in particolare, come l'angolo di sterzata θ è correlato agli angoli di sterzata delle due ruote anteriori. Infatti, gli angoli che le ruote sterzanti assumono lungo la curva differiscono l'uno dall'altro, poiché se fossero coincidenti le ruote slitterebbero, a causa del vincolo che hanno le une con le altre. Considerando θ

come l'angolo di sterzata che dovrebbe assumere una ruota centrale immaginaria, senza ulteriori vincoli, tale angolo farà descrivere al veicolo una traiettoria circolare del tipo:

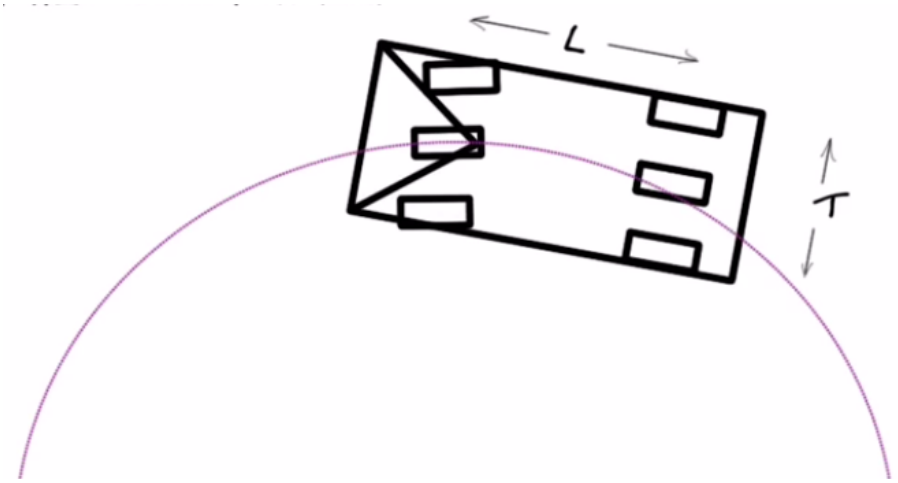


Figura 1.24: Traiettoria descritta dalla ruota immaginaria con un angolo di sterzo θ

Se si assegnassero gli stessi angoli sia alla ruota interna che a quella esterna, le traiettorie descritte non sarebbero compatibili con i vincoli che hanno le ruote, e inizierebbero a slittare, senza seguire una traiettoria ben definita:

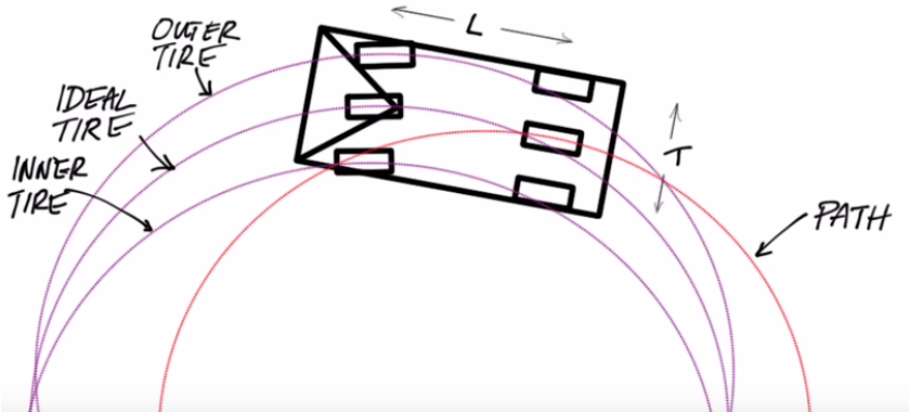


Figura 1.25: Traiettoria descritta dalle ruote sterzanti assegnando un angolo di sterzo θ

L'obiettivo è quindi quello di assegnare alle ruote sterzanti un angolo che non le faccia slittare e che faccia seguire al veicolo una traiettoria coerente con l'angolo θ desiderato, che si ricorda corrispondere all'angolo di sterzata della ruota centrale ideale.

Nel seguito sarà illustrato come ricavare gli angoli di sterzata delle due ruote anteriori del veicolo, a partire dall'angolo θ e dalle misure di interasse L e di carreggiata T. Nella seguente figura vengono mostrate le misure fondamentali per il calcolo:

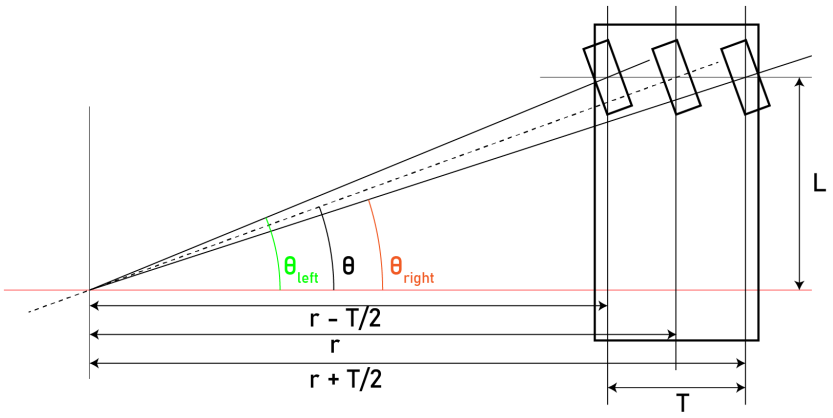


Figura 1.26: Misure fondamentali per il calcolo degli angoli di sterzata

Il parametro r corrisponde al raggio di curvatura che assume il veicolo. Infatti, con gli angoli di sterzata ricavati in questo modo, tutte le traiettorie assunte dalle ruote hanno lo stesso centro di curvatura C. Pertanto, pur assumendo traiettorie diverse, non si verifica uno slittamento. Utilizzando le regole della trigonometria è possibile ricavare i valori angolari assunti dalle due ruote sterzanti, partendo dall'angolo θ. Tale angolo corrisponde a quello fornito al controllore `ackermann_controller.py` che si occupa di far sterzare le ruote in simulazione e di far muovere il veicolo. Di seguito vengono mostrate le formule per ricavare tali angoli:

$$\theta_{left} = atan\left(\frac{r - \frac{T}{2}}{L}\right), \theta_{right} = atan\left(\frac{r + \frac{T}{2}}{L}\right) \tag{1.2}$$

dove r e T sono noti, mentre il raggio di curvatura r dipende da θ e corrisponde a:

$$r = Ltan\left(\frac{\pi}{2} - \theta\right) \tag{1.3}$$

Lo studio del modello appena mostrato si è reso necessario per poter convertire in maniera coerente i messaggi provenienti dal sistema Navigation2 in messaggi compatibili con il modello sterzante. Infatti i comandi prodotti dal Controller Server di Navigation2 sono di tipo "Twist". Essi contengono informazioni in velocità lineare ed angolare. L'obiettivo è ricavare da esse un angolo di sterzata da applicare al veicolo, utilizzando le formule appena viste. Un approfondimento relativo alla modalità implementativa e al codice utilizzato è mostrato in Appendice B.2.

1.6 Visual Odometry: Introduzione

Negli ultimi anni, con il rapido sviluppo dell'Intelligenza Artificiale e del Machine Learning, il campo della robotica ha compiuto un grande passo in avanti ed ora queste tecnologie sono regolarmente applicate ai più disparati campi, come quello militare, industriale, agricolo. Il focus principale della ricerca è stato quello del decision-making e del comportamento autonomo dei robot mobili. Tra le tecnologie sviluppate, soprattutto nell'ambito della navigazione indoor, c'è sicuramente il visual positioning, anche conosciuto come Visual Odometry. Questo processo utilizza sensori di visione per acquisire informazioni sotto forma di immagini e quindi stimare il moto del robot. Questo metodo viene preferito in alcuni ambiti, poiché ad esempio, l'odometria ricavata tramite encoder montato sulle ruote, nonostante sia molto meno costosa, commette un errore di posizionamento dovuto allo slittamento degli pneumatici oppure alle irregolarità del terreno.
I sensori di visione sono largamente utilizzati in robotica, nel campo della guida autonoma e in molte altre applicazioni, grazie al loro ricco apporto di informazioni e al costo relativamente basso rispetto ad altre tecnologie.

L'implementazione della Visual Odometry può essere realizzata utilizzando principalmente due metodi: il primo è chiamato "Feature-based Visual Odometry" (FVO) e il secondo è rappresentato dal "Dense Visual Odometry" (DVO), che non estrae feature dalle immagini. Il DVO generalmente utilizza le informazioni dei pixel relativi ad un frame per stimare la posa della camera. Tuttavia, a causa della grande quantità di dati e della complessità computazionale che richiede, con questo metodo non è possibile ottenere una performance in real-time.
D'altro canto, FVO rappresenta il metodo principale per realizzare un algoritmo di Visual Odometry, grazie alla sua stabilità, alla bassa quantità di calcoli necessari ed anche alla mole di ricerca che vi è stata dedicata, che lo ha reso un algoritmo più maturo ed affidabile rispetto agli altri.

1.6.1 L'Algoritmo FVO

L'algoritmo FVO è una soluzione per la Visual Odometry relativamente matura e molto utilizzata, che negli anni è stata sviluppata ed è diventata un sistema completo, che riesce a stimare la posa della camera a partire da una sequenza di immagini inviate dalla stessa. Di seguito viene mostrato il processo che l'algoritmo FVO segue per ottenere tale output:

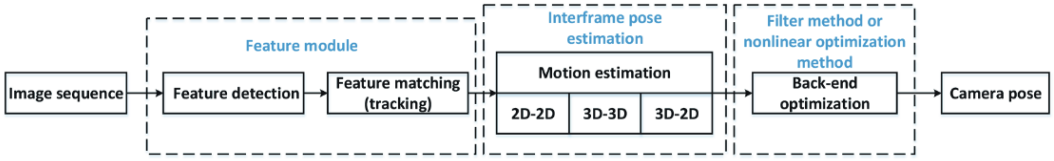


Figura 1.27: Processo implementativo dell'algoritmo FVO

Il processo consiste in tre moduli: feature, stima del moto e ottimizzazione back-end.

Il primo modulo applica prima di tutto un "Feature Point Detection", che consiste nel ricavare dei punti di riferimento nel frame inviato dalla camera. Successivamente viene effettuato un "Feature Matching" ed infine vi è la fase di "Mismatch Elimination".

Feature Point Detection Questa fase possiede due oggetti principali: il key point e il descriptor. Il key point rappresenta il punto individuato all'interno dell'immagine, mentre il descriptor consiste in una descrizione del punto individuato e fornisce una base per la fase successiva di Feature Matching. Esistono molte feature che possono essere applicate. Esse differiscono principalmente per la quantità di calcoli richiesti e per la loro bontà in determinati scenari. Ad esempio, ORB possiede buone caratteristiche in termini di scala, rotazione e calcolo della velocità. Per poter utilizzare la Feature Detection in real-time è necessario scegliere delle feature che siano elementari, indipendenti tra loro e quindi facilmente rilevabili. In ambiente indoor, gli angoli degli oggetti, delle strutture fisse rappresentano un buon esempio. Un feature point è un punto all'interno dell'immagine che possiede una rilevante variazione nei valori di grigio o uno spigolo pronunciato e che abbia le seguenti proprietà:

- Ripetibilità: lo stesso angolo può essere identificato in aree differenti;
- Distinguibilità: angoli differenti devono poter essere descritti con espressioni diverse;
- Vicinanza: le feature individuate dovrebbero essere relative ad aree dell'immagine adiacenti;
- Alta efficienza: il numero di feature points deve essere molto minore del numero di pixel presenti nell'immagine.

La Feature Detection ORB, la più utilizzata negli algoritmi di Visual Odometry, è un miglioramento dell'algoritmo FAST corners. L'idea principale di questo algoritmo consiste nell'individuare un pixel che sia più scuro o più chiaro dei suoi vicini, poiché è probabile che esso sia un angolo (corner). Per prima cosa si seleziona un pixel P nell'immagine, lo si prende come centro e si compara il suo colore con quello dei pixel vicini, selezionandoli attraverso un cerchio di un certo raggio fissato. Se la loro differenza supera la soglia prefissata, il pixel P può essere etichettato come "corner". Un esempio di rilevazione di un feature point è mostrato in Figura 1.28

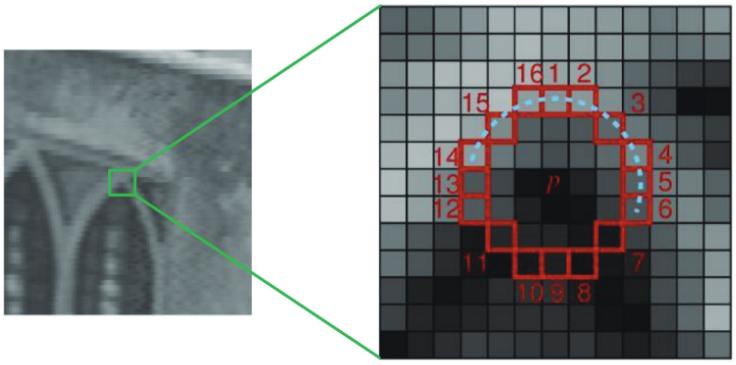


Figura 1.28: Individuazione di un feature point con metodo FAST

Feature Matching Il Feature Matching è uno step critico nel processo di FVO e si occupa di risolvere il problema di Data Association. Esso consiste nell'associazione, utilizzando il descriptor dei key point, dello stesso punto da frame diversi per poter, in una fase successiva, stimare il moto della camera. Il metodo più semplice per fare Feature Matching è la forza bruta (BF). Tuttavia questo metodo non è indicato se il numero di feature points è elevato, richiederebbe troppo tempo di calcolo. Viene quindi utilizzata la "Fast Library for Approximate Nearest Neighbors" (FLANN). Questo algoritmo è particolarmente adatto quando sono presenti molti feature points, poiché ha un'alta efficienza di calcolo. D'altro canto però, ci sono algoritmi che hanno un'accuratezza di matching più elevata. Con la terza fase del processo feature, chiamata "Mismatch Elimination", è possibile risolvere questi mismatch e massimizzare l'accuratezza della Visual Odometry.

Mismatch Elimination Questa fase è fondamentale per eliminare gli errori commessi nel processo di Feature Matching e quindi per aumentare l'accuratezza dell'algoritmo. Il metodo più semplice consiste nell'utilizzo di una soglia.

Motion Estimation Questo modulo si occupa di calcolare una matrice di trasformazione $T_{k,k-1}$ tra il frame corrente I_k e il frame precedente I_{k-1}. $T_{k,k-1}$ è calcolata utilizzando le informazioni provenienti dal modulo delle feature. Poiché nell'elaborato corrente è stata utilizzata una camera RGB-D, che riesce a rilevare anche la profondità, i key points sono disposti in un ambiente tridimensionale, sfruttando l'immagine RGB e la depth image. La Motion Estimation pertanto è una stima 3D-3D, che calcola il moto del robot partendo dall'associazione di tali punti.

1.6.2 RTAB-Map: Implementazione della Visual Odometry

Real-Time Appearance-Based Mapping è una soluzione molto usata nel campo dei sistemi di visione. Tale pacchetto include algoritmi per l'RGB-D SLAM e per la Visual Odometry, supportando vari tipi di sensori. È possibile infatti stimare un'odometria ricevendo informazioni da una camera a lente singola, da una stereo camera

oppure da un sensore RGB-D. Nel corrente elaborato è stata utilizzata una camera RGB-D Intel Realsense D435, che sarà illustrata in dettaglio nella sezione 3.2.
RTAB-Map implementa tutte le funzioni illustrate in precedenza, sfruttando l'algoritmo FVO per la Visual Odometry. Di seguito saranno illustrati i parametri più rilevanti dell'algoritmo, che sono stati adattati al problema specifico di questa tesi. È importante ricordare che la Visual Odometry è stata migliorata utilizzando un algoritmo di Sensor Fusion, sfruttando un Extended Kalman Filter. Oltre a migliorare la bontà dell'odometria introducendo le informazioni provenienti da una IMU, l'algoritmo di Sensor Fusion permette al sistema di funzionare anche in condizioni in cui la visibilità non è ottimale, come gli spazi stretti o troppo omogenei con pochi feature point.

Nome parametro	**Valore**	**Significato**
`subscribe_rgbd`	true	Valore booleano necessario ad informare RTAB-Map che l'input è di tipo RGB-D
`frame_id`	camera_link	Nome del sistema di riferimento al quale ricondurre il movimento ricavato con la Visual Odometry
`odom_frame_id`	odom	Nome del sistema di riferimento relativo all'odometria
`publish_tf`	false	Informa RTAB-Map che non deve pubblicare una trasformazione TF, da "odom" a "base_link"
`Odom/ResetCountdown`	1	Quando l'odometria viene persa, ad esempio in condizioni di scarsa informazione visiva, l'algoritmo viene resettato
`approx_sync`	false	Si utilizza una sincronizzazione esatta degli input, infatti tutte le immagini provengono dallo stesso dispositivo, quindi si può stabilire a priori che gli input saranno sincronizzati

Tabella 1.3: Descrizione dei parametri per la configurazione di RTAB-Map

In Appendice B.3 sarà mostrato in dettaglio il codice utilizzato per implementare l'algoritmo di Visual Odometry.

1.6.3 Test della Visual Odometry all'interno di un Magazzino Simulato

Nel corrente paragrafo sarà mostrato come la posizione del veicolo e in particolare il tipo di scenario che la camera cattura, è determinante per riuscire ad effettuare una buona stima dell'odometria.

Il tipo di test che è stato effettuato consiste nell'eseguire un tragitto all'interno del magazzino, mostrato in figura 1.29, passando in aree in cui l'algoritmo di Visual Odometry sarà sollecitato, sia per omogeneità dell'immagine catturata (mancanza di punti di riferimento) che per vicinanza ad oggetti (mancanza di un ampio campo visivo).

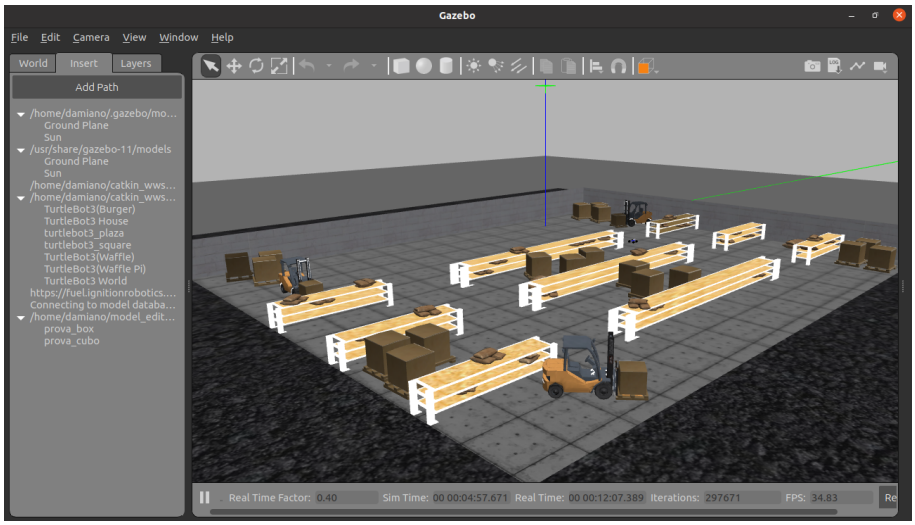


Figura 1.29: Magazzino simulato nel quale è stata svolta la simulazione

La velocità di riferimento è stata variata, anche per capire in che modo la Visual Odometry reagisce ad accelerazioni del veicolo.

In figura 1.30 viene illustrato il tragitto del veicolo tramite il topic odometrico, mostrato in maniera grafica con il programma RViz. Ciò che è possibile riscontrare è che nonostante il veicolo abbia compiuto un tragitto ad anello e sia ritornato esattamente alla posizione iniziale, è presente un drifting, che fa risultare il punto di arrivo leggermente distante da quello di partenza, di circa 2 metri più avanti. Nel prossimo paragrafo, nel quale questi dati saranno confrontati con quelli relativi all'odometria ottenuta tramite filtro di Kalman, saranno più chiare le circostanze nelle quali la Visual Odometry ottiene una stima meno precisa.

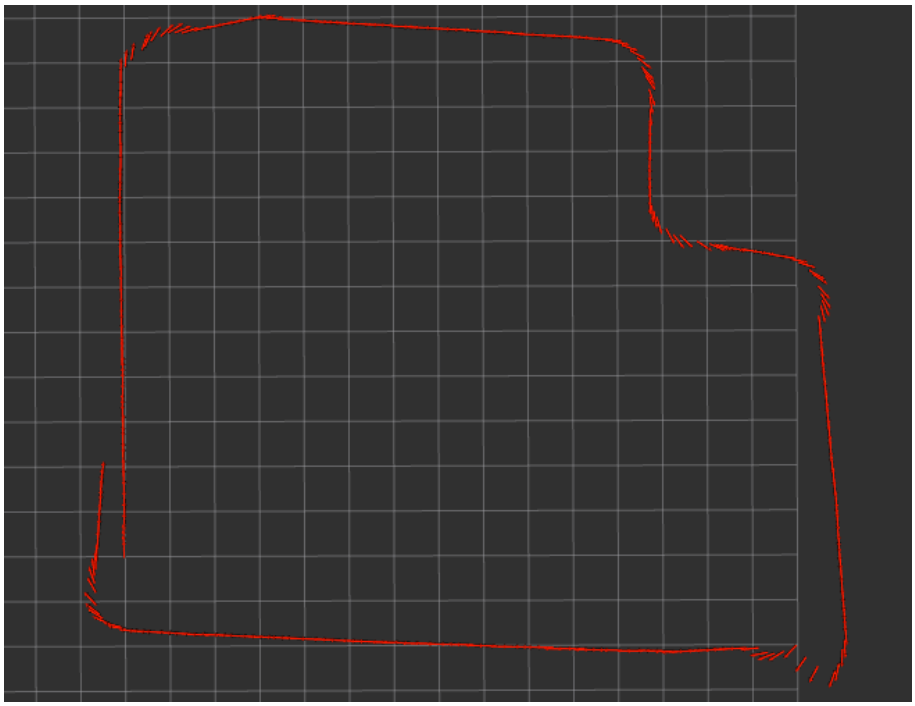


Figura 1.30: Output grafico della Visual Odometry ottenuta tramite RTAB-Map

Inoltre dal test effettuato si evince che la Visual Odometry presenta un lieve drifting lungo l'asse z, probabilmente dovuto alle oscillazioni relative agli ammortizzatori. Tale imprecisione può essere migliorata filtrando le informazioni provenienti dalla Visual Odometry, come sarà illustrato nel paragrafo relativo all'algoritmo di Sensor Fusion.

Infine, se si è in presenza di ambienti dai quali la RGB-D camera può carpire pochissime informazioni, come ad esempio se il veicolo si trova molto vicino ad un oggetto e non riesce a ricavare punti di riferimento, la Visual Odometry fallisce nella stima e fornisce in output un errore.

In conclusione, da tale test si evince che sarà necessario aggiungere un sensore a bordo del veicolo, in particolare una IMU, per poi fondere le informazioni provenienti da tale sensore con quelle ottenute tramite la Visual Odometry, e ricavare di conseguenza un'odometria migliorata.

1.7 Il Pacchetto Robot Localization: Sensor Fusion tramite EKF

L'utilizzo di un singolo strumento per stimare l'odometria, come per esempio la Visual Odometry, può produrre importanti derive nella stima della posa del veicolo, dovute all'errore cumulativo relativo al processo di stima che, ad ogni passo, per ricavare la posa alla quale il robot si trova, si basa sulla stima ottenuta al passo precedente. All'interno del pacchetto RTAB-Map per la Visual Odometry sono stati implementati vari strumenti che permettono di migliorare la stima odometrica e il mapping, come ad esempio il "Loop Closure Detection" che si occupa di riconoscere tramite salvataggio dei fotogrammi scattati negli istanti precedenti, punti in cui il veicolo è già passato, e correggere così il drifting odometrico chiudendo il loop. Tuttavia questo approccio richiede appunto che si ritorni al punto di partenza. Occorre quindi, al fine di avere una stima odometrica più precisa in tempo reale e di garantire un'affidabilità e una stabilità più elevata, adottare un sistema sensoriale complementare, che in questo caso è stato individuato in una IMU.
Un sensore di questo tipo infatti, al contrario della Visual Odometry, commette un errore di stima limitato e non dipendente dallo scenario visivo in cui si trova il veicolo. Ad esempio, se in una determinata posa l'ambiente risulta molto omogeneo o il campo visivo è limitato, il numero di feature points rilevato dal sistema di Visual Odometry risulterà basso e la stima sarà di cattiva qualità. D'altro canto, poiché per ricavare valori di posizione dal sensore inerziale è necessario integrare due volte le accelerazioni, l'incertezza che si ha con il solo utilizzo della IMU è maggiore. Pertanto, l'obiettivo sarà quello di adottare una strategia di fusione e sfruttare le caratteristiche positive dei due sistemi sensoriali. Resta comunque il fatto che si tratta di un dato di odometria e che come tale, basando la rilevazione all'istante k su quella effettuata all'istante $k-1$, esso è affetto da un errore che si accumula con il tempo. Sarà dunque compito dell'algoritmo AMCL di Navigation2, illustrato nel Paragrafo 1.4.7, correggere periodicamente il drifting odometrico e quindi la posizione del veicolo all'interno della mappa.

L'utilizzo di sensori multipli all'interno del sistema è spesso raccomandato dove l'informazione odometrica, nel caso corrente Visual Odometry, viene combinata con altri sensori, come Lidar, IMU, al fine di migliorare la stima finale. In generale l'accuratezza di queste tecnologie è altamente dipendente dal setup sensoristico. Ad esempio, considerando i dispositivi a bordo del robot, l'accuratezza della localizzazione dipende fortemente dal costo, dal numero di sensori utilizzati. In questa sezione sarà utilizzato un Extended Kalman Filter al fine di fondere insieme le informazioni provenienti da più sensori e migliorare così l'output odometrico finale.

Si è scelto di adottare proprio questa soluzione poiché rappresenta il giusto compromesso tra qualità della stima di posizionamento e invasività nell'installazione dei sensori. Infatti, l'utilizzo di una soluzione come un encoder rotativo nell'assale delle ruote avrebbe richiesto una modifica sostanziale nella meccanica del veicolo, mentre il tipo di sensori scelto risulta di molto più semplice implementazione anche se di contro, presenta un costo più elevato. Per poter realizzare tale compito è stato utilizzato il pacchetto di ROS "Robot Localization", compatibile con la versione

"Noetic" di ROS, per poter ottenere un valore di odometria migliorata rispetto alla Visual Odometry di partenza, effettuando un'operazione di sensor fusion con i dati provenienti dall'unità inerziale. Come già accennato in precedenza, il pacchetto Robot Localization lavora attraverso un Extended Kalman Filter, il funzionamento del quale sarà illustrato di seguito.

1.7.1 Applicazione dell'Extended Kalman Filter

L'Extended Kalman Filter (EKF) è stato utilizzato per fondere insieme le informazioni provenienti dalla IMU e dalla Visual Odometry.

Un filtro di Kalman lavora essenzialmente in due step che si ripetono in maniera iterativa:

- a: predizione della posa del robot $\hat{\boldsymbol{x}}_{k|k-1}$ al passo k: viene effettuata a partire dalla Visual Odometry, le cui informazioni sono incluse all'interno del vettore di input $\boldsymbol{u}_k$ e dallo stato corretto $\hat{\boldsymbol{x}}_{k-1|k-1}$ al passo k-1
- b: correzione della posa stimata $\hat{\boldsymbol{x}}_{k|k}$ al passo k. Questo step richiede per poter essere effettuato di conoscere il residuo di misura $\tilde{\boldsymbol{y}}_k$ al passo k tramite la misura del sensore inerziale $\boldsymbol{z}_k$ e la stima della misura a partire dallo stato, data dalla funzione $h(\hat{\boldsymbol{x}}_{k|k-1})$.

Le feature caratteristiche dell'EKF vengono così ricavate:

- Step di predizione: assumendo che la notazione $\boldsymbol{x}_{n|m}$ rappresenti la stima dello stato $\boldsymbol{x}$ all'istante n conoscendo le osservazioni all'istante $m <= n$, si avrà:

$$\begin{cases} \hat{\boldsymbol{x}}_{k|k-1} = f(\hat{\boldsymbol{x}}_{k-1|k-1}, \boldsymbol{u}_k) \\ \boldsymbol{P}_{k|k-1} = \boldsymbol{J}_k \boldsymbol{P}_{k-1|k-1} \boldsymbol{J}_k^T + \boldsymbol{Q}_k \end{cases} \tag{1.4}$$

  dove $\boldsymbol{P}_{k|k-1}$ rappresenta la previsione della matrice di covarianza al passo k, conoscendo le osservazioni al passo $k-1$. Tale matrice contiene l'errore di stima, mentre $\boldsymbol{J}_k$ rappresenta la matrice Jacobiana, che descrive le variazioni delle variabili le une rispetto alle altre. Il Jacobiano $\boldsymbol{J}_k$ sarà descritto da:

$$\boldsymbol{J}_k = \frac{\partial f}{\partial \hat{\boldsymbol{x}}}\Big|_{\hat{\boldsymbol{x}}_{k-1}, \boldsymbol{u}_k} \tag{1.5}$$

- Step di correzione: durante questo step viene calcolato il residuo di misura o innovazione $\tilde{\boldsymbol{y}}_k$, il Kalman Gain $\boldsymbol{K}$ che servirà per la correzione dello stato, viene aggiornato lo stato del sistema sfruttando le misure provenienti dai sensori e, infine, viene aggiornata la matrice di covarianza.

$$\begin{cases} \tilde{\boldsymbol{y}}_k = \boldsymbol{z}_k - h(\hat{\boldsymbol{x}}_{k|k-1}) \\ \boldsymbol{S}_k = \boldsymbol{H}_k \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T + \boldsymbol{R}_k \\ \boldsymbol{K}_k = \boldsymbol{P}_{k|k-1} \boldsymbol{H}_k^T \boldsymbol{S}_k^{-1} \\ \hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + \boldsymbol{K}_k \tilde{y}_k \\ \boldsymbol{P}_{k|k} = (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{P}_{k|k-1} \end{cases} \tag{1.6}$$

dove $\boldsymbol{H}_k$ rappresenta il seguente Jacobiano:

$$\boldsymbol{H}_k = \frac{\partial f}{\partial \boldsymbol{x}}\Big|_{\hat{\boldsymbol{x}}_{k|k-1}} \tag{1.7}$$

1.7.2 Creazione dell'Inertial Measurement Unit

Per riuscire a realizzare l'algoritmo di Sensor Fusion sopra descritto, è necessario inserire all'interno del veicolo simulato un'unità di misurazione inerziale. Il software Gazebo, oltre a permettere la simulazione grafica e fisica dei corpi rigidi, permette anche la simulazione di tali sensori, che possono essere creati in maniera molto semplice aggiornando il file descrittivo del veicolo, che in questo caso è rappresentato da `em_3905.urdf.xacro`. Infatti, è sufficiente definire all'interno di tale file un link, `imu_link`, definire la sua posizione e orientazione, oltre al giunto (fisso) che lo collega al `base_link` e dichiarare il plugin di Gazebo relativo al sensore inerziale. Il codice aggiunto nel file urdf del veicolo è il seguente:

```
<joint name="imu_joint" type="fixed">
        <axis xyz="1 0 0"/> <!-- 0 1 0 -->
        <origin xyz="0 0 0.19"/>
        <parent link="base_link"/>
        <child link="imu_link"/>
</joint>


<link name="imu_link">
        <inertial>
                <mass value="0.001"/>
                <origin rpy="0 0 0" xyz="0 0 0"/>
                <inertia ixx="0.0001" ixy="0" ixz="0" iyy
                   ="0.000001" iyz="0" izz="0.0001"/>
        </inertial>
        <visual>
                <origin rpy="0 0 0" xyz="0 0 0"/>
                <geometry>
                        <box size="0.001 0.001 0.001"/>
                </geometry>
        </visual>
        <collision>
                <origin rpy="0 0 0" xyz="0 0 0"/>
                <geometry>
                        <box size=".001 .001 .001"/>
                </geometry>
        </collision>
</link>

<gazebo>
        <plugin name = "imu_controller" filename = "
           libgazebo_ros_imu.so">
                <alwaysOn>true</alwaysOn>
                <updateRate>50.0</updateRate>
                <bodyName>imu_link</bodyName>
                <topicName>imu_data</topicName>
                <gaussianNoise>2.89e-08</gaussianNoise>
                <xyzOffsets>0 0 0</xyzOffsets>
                <rpyOffsets>0 0 0</rpyOffsets>
```

```
38                  <interface:position name="imu_position"/>
39          </plugin>
40  </gazebo>
```

Ciò che si occupa effettivamente di simulare il sensore inerziale è il plugin di Gazebo `libgazebo_ros_imu.so` che viene associato al link appositamente creato. Un passaggio molto importante che è necessario compiere quando si aggiunge un link all'interno del sistema, è pubblicare una trasformazione TF, ovvero stabilire la posizione e l'orientamento del sistema di riferimento solidale alla IMU (imu_link) con il sistema di riferimento relativo al veicolo (base_link). Tale trasformazione è specificata all'interno della definizione del giunto che collega i due link. In particolare il link "padre" corrisponde a `base_link` e il link "figlio" corrisponde a `imu_link`.

Il plugin `libgazebo_ros_imu.so` si occupa di pubblicare il topic `/imu_data`, con messaggi di tipo `sensor_msgs/Imu`. Tali messaggi, pubblicati all'interno del topic, contengono le informazioni fornite dalla IMU. Di seguito sono elencati i dati che il sensore fornisce al sistema:

- Orientamento, fornito in quaternioni (x, y, z, w);
- Matrice di covarianza (orientamento);
- Velocità angolare lungo i tre assi (x, y, z);
- Matrice di covarianza (velocità angolare);
- Accelerazione lineare lungo i tre assi (x, y, z).

Da tale topic saranno prelevate le informazioni necessarie all'operazione di sensor fusion che sarà effettuata dal pacchetto Robot Localization.

1.7.3 Creazione del nodo Robot Localization

Il Robot Localization Package è una raccolta di nodi volti alla stima dello stato. Contiene due nodi: `ekf_localization_node` e `ukf_localization_node`. Inoltre, Robot Localization fornisce il `navsat_transform_node`, che aiuta nell'integrazione con dati GPS.
Le principali features incluse all'interno di tale pacchetto sono:

- Fusione di un numero arbitrario di sensori. Nel caso del lavoro corrente sarà necessario fondere le informazioni provenienti dalla Visual Odometry con quelle fornite dalla IMU.
- Supporto di multipli tipi di messaggi, tra cui:

  `nav_msgs/Odometry`

  `sensor_msgs/Imu`

  `geometry_msgs/PoseWithCovarianceStamped`

  `geometry_msgs/TwistWithCovarianceStamped`

- Stima dello stato del robot con l'utilizzo dei nodi `ekf_localization_node` e `ukf_localization_node`. Tali nodi saranno utilizzati per fondere le informazioni dei sensori. Essi utilizzano un filtro di Kalman per la stima dello stato, come mostrato nel paragrafo precedente.

In questa sezione si porrà l'attenzione sul nodo `ekf_localization_node` del pacchetto "Robot Localization", che sfrutta un filtro di Kalman esteso per effettuare la fusione delle informazioni provenienti dal sensore inerziale e quelle di Visual Odometry. In particolare si descriverà il codice utilizzato per l'implementazione.

In primo luogo, al fine di lanciare tale nodo insieme agli altri nodi utilizzati nella simulazione del veicolo ackermann, si è aggiunto il seguente codice all'interno del file `.launch`:

```
<node pkg="robot_localization" type="ekf_localization_node" name="
    ekf_localization_with_imu">
<rosparam command="load" file="$(find ackermann_vehicle_description
    )/config/ekf_localization.yaml"/>
</node>
```

Il file `ekf_localization.yaml` è molto importante poiché contiene al suo interno tutti i parametri necessari al ROS Node sopra citato per poter funzionare correttamente. Inizialmente vengono definite le modalità di funzionamento e i frame ai quali il nodo deve associarsi:

```
frequency: 50

two_d_mode: true

publish_tf: true

odom_frame: odom
base_link_frame: base_link
world_frame: odom
```

Il parametro "frequency" definisce la frequenza di pubblicazione all'interno del topic `/odometry/filtered`, nel quale verrà pubblicata l'odometria migliorata, mentre il parametro `two_d_mode`, impostato con il valore "true", sta a significare che il robot si muoverà in un ambiente planare. L'effetto di questo parametro sarà di azzerare tutte le accelerazioni e le velocità lineari lungo l'asse z.
Il parametro `publish_tf` è impostato con il valore "true", infatti il nodo di localizzazione che implementa il filtro di Kalman esteso è quello incaricato di pubblicare la trasformazione TF dal frame `odom` al frame `base_link`.
In ultimo, i parametri `odom_frame`, `base_link_frame` e `world_frame` rappresentano i sistemi di riferimento ai quali il nodo deve associarsi e in particolare definiscono i frame rispetto ai quali il nodo deve creare la trasformazione TF. In questo caso, non avendo un sistema di mapping, i frame sono stati associati come illustrato nel codice sovrastante.
Successivamente vengono impostati i parametri di fusione dei sensori:

```
odom0: /ackermann_vehicle/stereo_camera/odom
#Values order: x, y, z, roll, pitch, yaw, vx, vy, vz, vroll, vpitch
    , vyaw, ax, ay, az
odom0_config: [false, false, false,
```

```
4  false, false, true,
5  true, true, false,
6  false, false, true,
7  false, false, false]
8  odom0_differential: false
9
10 imu0: /ackermann_vehicle/imu_data
11 imu0_config: [false, false, false,
12 false, false,  true,
13 false, false, false,
14 false, false,  true,
15 true,  false, false]
16 imu0_differential: false
```

All'interno delle variabili `odom0` e `imu0` vengono salvati i nomi dei topic, rispettivamente di Visual Odometry e IMU.
Le matrici `odom0_config` e `imu0_config` hanno la seguente struttura:

$$\begin{bmatrix} x & y & z \\ roll & pitch & yaw \\ \dot{x} & \dot{y} & \dot{z} \\ \dot{roll} & \dot{pitch} & \dot{yaw} \\ \ddot{x} & \ddot{y} & \ddot{z} \end{bmatrix} \tag{1.8}$$

Se l'elemento (i, j) della matrice ha il valore "true" allora la corrispondente variabile sarà utilizzata dal filtro di Kalman per stimare lo stato, altrimenti sarà scartata.

Nel caso della matrice `odom_config` i dati relativi alle posizioni (x, y, z) non sono stati presi in considerazione. Infatti, proveniendo dallo stesso sensore (RGB-D camera), i dati di posizione e di velocità risultano ridondanti, poiché uno è ottenuto dall'altro. Pertanto il risultato sarebbe stato quello di fornire informazioni duplicate al filtro.
Per quanto concerne il parametro $\dot{y}$, nonostante la velocità istantanea del robot risulti sempre lungo l'asse x, se si considera il sistema di riferimento solidale con esso, viene fornito al filtro per migliorare la stima dello stato. Si potrebbe dire lo stesso del parametro $\dot{z}$ che nella matrice è stato impostato al valore "false", ma in realtà tale parametro (con il valore 0) è stato già fornito al filtro nel momento in cui è stato impostato a "true" il parametro `two_d_mode`.

Per quanto riguarda la matrice `imu0_config`, relativa alla IMU, e quindi importante per le misure di accelerazione lineare, è possibile notare che il parametro $\ddot{y}$ è "false". Infatti l'accelerazione istantanea lungo l'asse y sarà sempre nulla, ma il sensore inerziale potrebbe introdurre del rumore all'interno di tale valore, peggiorando la stima dello stato e provocando un veloce drifting.

1.7.4 Test dell'Odometria ottenuta tramite Sensor Fusion e Confronto con la Visual Odometry

In questa sezione saranno ripresi i test effettuati sulla Visual Odometry (Cap. 1.6) e saranno confrontati con l'odometria ottenuta tramite il filtro di Kalman esteso, fondendo i dati della Visual Odometry con le informazioni di accelerazione ricavate

dalla IMU. Il test svolto consiste in un primo spostamento del veicolo da un punto A ad un punto B del magazzino, per poi fermarsi e ritornare al punto di partenza. Per valutare l'efficacia del test si considererà la differenza tra la posizione iniziale e quella finale, mettendo a confronto le due odometrie. Il tipo di localizzazione utilizzata in questo progetto è di tipo "dead reckoning", ovvero si stima la posizione attuale del robot a partire dalla posizione stimata al passo precedente. Questo metodo è per definizione affetto da errore, poiché l'errore commesso in un determinato istante si propagherà per tutti gli istanti successivi. Lo scopo quindi è quello di ridurre al minimo tale errore. Una soluzione spesso adottata per risolvere questo problema è far intervenire un sistema GPS per correggere l'errore commesso.

Tali test sono finalizzati soltanto a provare la bontà della stima delle due odometrie, pertanto non è stato fatto utilizzo di una mappa per la navigazione, in quanto questa feature sarà implementata nelle sezioni successive.

Dal confronto delle due odometrie ottenute si evince come, nei punti in cui la camera non riesce ad avere un'ampia visuale e in quelli in cui l'inquadratura non è ricca di dettagli, la Visual Odometry tenda a driftare e l'errore continui ad aumentare. Grazie alla fusione dei dati del sensore inerziale, tramite il filtro di Kalman, si riesce a migliorare sensibilmente l'odometria.

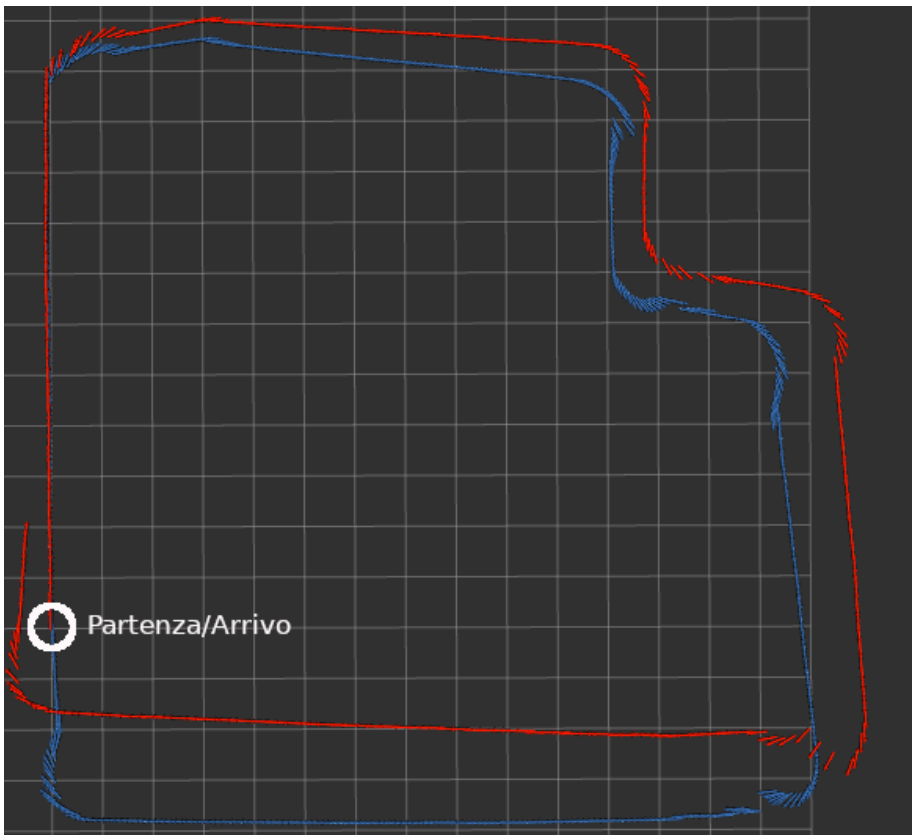


Figura 1.31: Confronto della Visual Odometry (tratto rosso) con l'odometria ottenuta tramite Sensor Fusion (tratto blu)

Durante il test, nei tratti rettilinei, è stata aumentata e poi diminuita la velocità. Dai dati è possibile notare come la Visual Odometry, se non integrata con un altro

sensore come la IMU, faccia fatica a riconoscere le variazioni di velocità del veicolo, infatti i tratti rettilinei differiscono in lunghezza. Il tratto blu (Kalman Filter) riesce a descrivere un loop quasi perfetto, mentre il tratto rosso (Visual Odometry) subisce un drifting di circa due metri. Inoltre, nei tratti in cui non sono presenti un buon numero di feature points, la qualità della Visual Odometry potrebbe calare e portare ad incertezze maggiori nella stima del posizionamento. In tali tratti, l'aiuto della IMU è ancora più importante, poiché riesce a correggere l'errore commesso dalla Visual Odometry. D'altro canto, in generale un sistema di visione riesce a stimare in modo migliore il posizionamento del veicolo rispetto al solo utilizzo di un sensore inerziale.

Capitolo 2

Simulazione e Test sul Sistema Finale

Dopo aver introdotto tutti i software utilizzati e la loro configurazione, in questo capitolo si passerà a descrivere il sistema finale e si illustreranno alcuni test svolti per certificare l'efficacia del lavoro svolto. La composizione degli algoritmi studiati nel Capitolo 1 è finalizzata a realizzare una simulazione il cui obiettivo è quello di navigare da un punto A verso un punto B, forniti entrambi dall'utente. Infatti il lavoro svolto durante questa tesi è focalizzato all'ottenimento di un sistema odometrico affidabile e all'interfaccia del sistema di navigazione con gli attuatori del robot. Tramite la corretta configurazione dei parametri del software Navigation2, del software di Visual Odometry RTAB-Map e dell'algoritmo di Sensor Fusion è stato possibile ottenere un ottimo risultato, che sarà mostrato in questa sezione e che sarà applicato in un modellino reale in scala 1:10 nel Capitolo 3.

Per questo test è stato fatto uso del software Gazebo per simulare il veicolo e l'ambiente in cui si muove, così come tutti i sensori necessari alla stima del posizionamento, quali la camera, il Lidar e la IMU. Nel prossimo Capitolo invece, Gazebo sarà sostituito da un veicolo reale che a bordo avrà lo stesso tipo di sensori testati in simulazione.

Le prove svolte in questa sezione consistono in una navigazione da un punto A verso un punto B, in primo luogo senza l'inserimento di ostacoli lungo il percorso, per testare sia l'efficacia dell'odometria che di Navigation2. Successivamente, una volta verificato il funzionamento del sistema in assenza di disturbi esterni, si è passato al secondo test, in cui è stato inserito un ostacolo lungo il percorso al fine di testare la feature di Obstacle Avoidance, tramite intervento del Recovery Server di Navigation2 e del ricalcolo del percorso da parte del Planner Server, una volta rilevato l'ostacolo.

Prima di passare alle simulazioni svolte, nella prossima sezione sarà descritta in dettaglio la configurazione del sistema e in particolare l'interfaccia tra i vari software studiati nel Capitolo 1.

2.1 Descrizione del Sistema Simulato Completo

Il framework ROS permette, grazie alla sua struttura, di eseguire diversi algoritmi in contemporanea utilizzando nodi multipli e tali algoritmi possono comunicare tra loro grazie alle infrastrutture messe a disposizione. Tramite le Actions, i Services e i Topics, tutti i nodi in fase di running all'interno di ROS possono interagire e dar vita ad un sistema complesso che in questo caso ha permesso la simulazione di un veicolo che si muova in maniera autonoma all'interno di un ambiente.
Di seguito è mostrata la struttura complessiva del sistema e in particolare l'interazione tra i nodi.

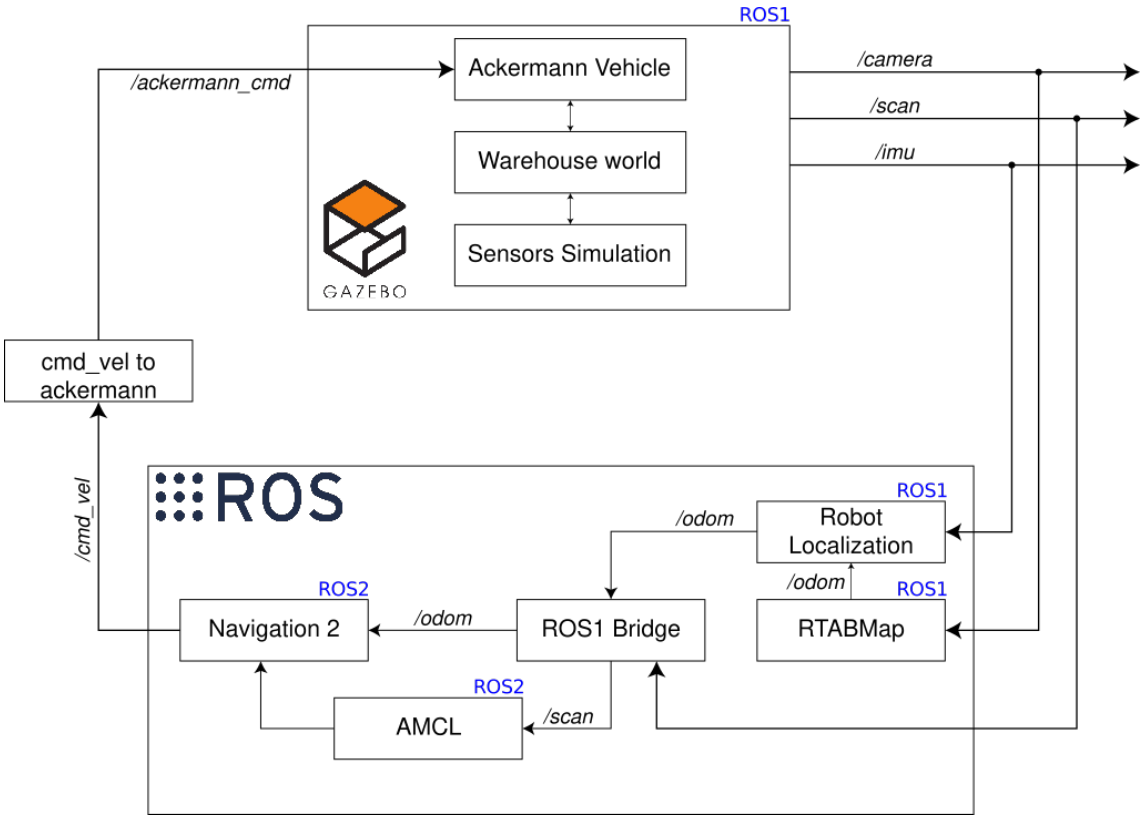


Figura 2.1: Mappa relativa alla configurazione del sistema simulato

Com'è possibile apprezzare in figura, l'algoritmo di Visual Odometry, una volta ricavato il valore di odometria, lo pubblica all'interno del topic `/odom`. Queste informazioni vengono lette dal nodo relativo al filtro di Kalman esteso che, fondendole con quelle provenienti dalla IMU, pubblica un nuovo topic di odometria e una trasformazione TF.
Il `ros1_bridge` legge tutti i dati elaborati dagli algoritmi eseguiti in ROS1 e si occupa di trascriverli in ROS2, all'interno dei relativi topic. Se il topic corrispondente non è ancora presente nell'ambiente ROS2, viene creato e viene riempito con le informazioni opportune. Il software Navigation2 legge tutti i dati relativi ai sensori e all'odometria per poter fare "obstacle detection" tramite il Controller Server. Questa feature è permessa grazie alla lettura della Local Costmap, che viene aggiornata in tempo reale leggendo i dati provenienti dal sensore LIDAR. Tale sensore è utilizzato anche dal pacchetto AMCL che si occupa di localizzare il robot all'interno

della mappa.

Navigation2 invia in output tramite il topic `cmd_vel` i comandi in velocità. Il tipo di questi messaggi prevede un dato in velocità lineare e uno in velocità angolare, pertanto è necessaria una conversione per ricavare un angolo di sterzata da applicare al veicolo. Lo script Python `cmd_vel_to_ackermann.py`, illustrato in dettaglio nell'Appendice B.2 si occupa proprio di questo: sfruttando le formule ricavate nella Sez. 1.5.1, in cui è stato studiato il modello matematico di un veicolo Ackermann, ricava un angolo di sterzata e lo pubblica all'interno del topic `ackermann_cmd`, che viene letto per azionare il veicolo all'interno dell'ambiente simulato in Gazebo.

Il software Gazebo riesce ad interfacciarsi con il framework ROS ed eseguire graficamente e fisicamente una simulazione del movimento del veicolo, fornendo anche le informazioni relative ai sensori. Gazebo esegue il veicolo, in accordo con i parametri salvati all'interno del nodo ROS `ackermann_vehicle`, il world "warehouse", che rappresenta un magazzino, e invia negli opportuni topic tutte le informazioni che un sensore reale ricaverebbe all'interno di un ambiente reale.

I topic `/camera`, `/scan` e `/imu` sono letti da Robot Localization e da RTAB-Map così da chiudere il loop di controllo.

2.2 Test di Navigazione da un Punto A verso un Punto B

In questa sezione sarà illustrato il test eseguito in simulazione per verificare il funzionamento del sistema. In primo luogo, come introdotto in precedenza, è stata effettuata una navigazione da un punto A verso un punto B. Il Planner Server (SMAC Planner) ha ricavato un percorso da seguire basandosi sulla global costmap, ottenuta dalla mappa fornita dal Map Server e in tale percorso non sono stati inseriti ostacoli.

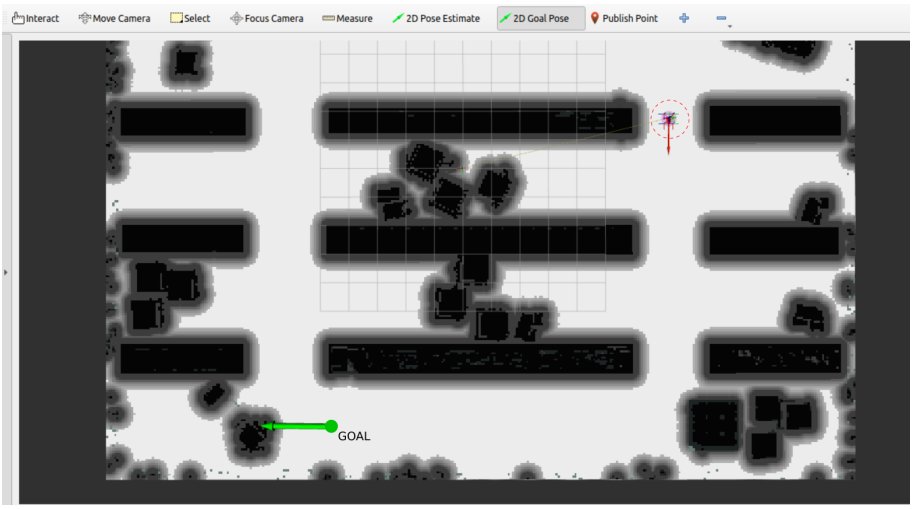


Figura 2.2: Set della posizione finale da raggiungere

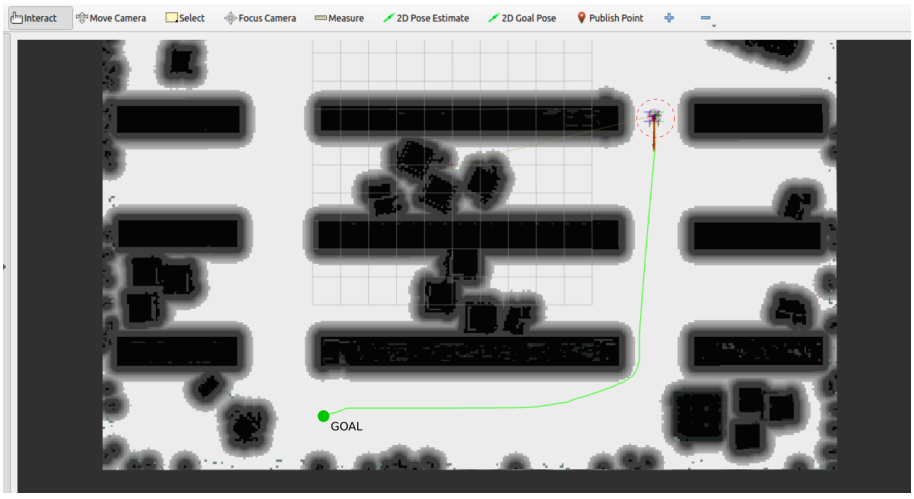


Figura 2.3: Calcolo del path da percorrere per arrivare a destinazione

In Figura 2.3 è mostrata un'istantanea del processo di simulazione, dopo aver definito il punto di arrivo. Il planner, basandosi sulla global costmap, ha ottenuto il percorso mostrato.

2.3 Test della Feature di Obstacle Avoidance

Come secondo test, per simulare un ostacolo lungo il percorso e quindi far intervenire il Recovery Server di Navigation2 e far ricalcolare il percorso al Planner Server, è stato inserito all'interno del magazzino un box, non presente nella mappa. In prima istanza, poiché tale oggetto non era rilevabile dal veicolo, al fine di arrivare al punto di arrivo ottimizzando il percorso, il planner ha ricavato il path rettilineo mostrato in Figura 2.4:

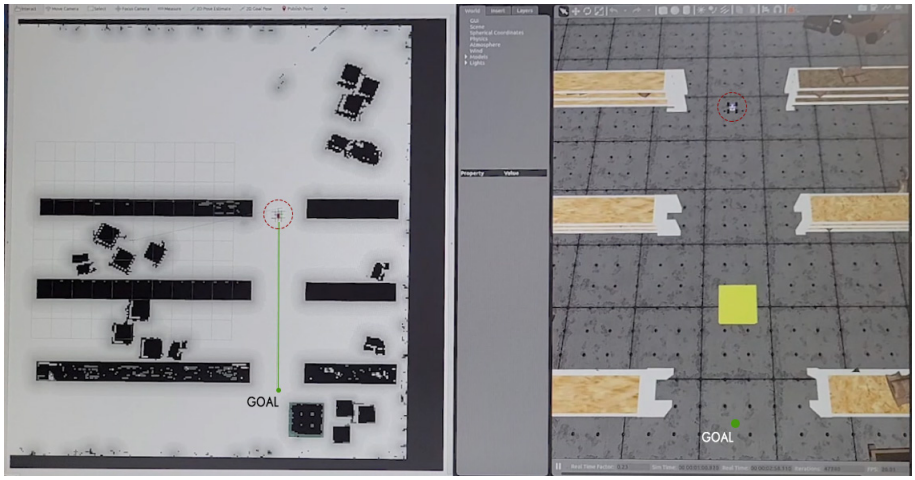


Figura 2.4: Path ricavato dal Planner Server non avendo rilevato l'ostacolo

Approcciandosi all'ostacolo, grazie al sensore Lidar il veicolo lo rileva e la costmap si aggiorna, mostrando la sagoma dell'oggetto:

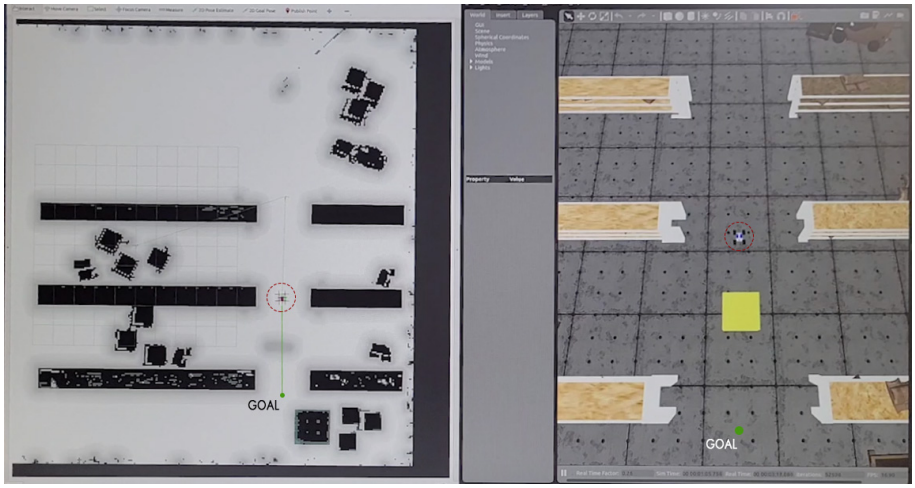


Figura 2.5: Aggiornamento della global costmap avendo rilevato l'ostacolo

Di conseguenza, il planner ricalcola il percorso e il veicolo riesce ad aggirare l'ostacolo, come mostrato nelle figure 2.6 e 2.7:

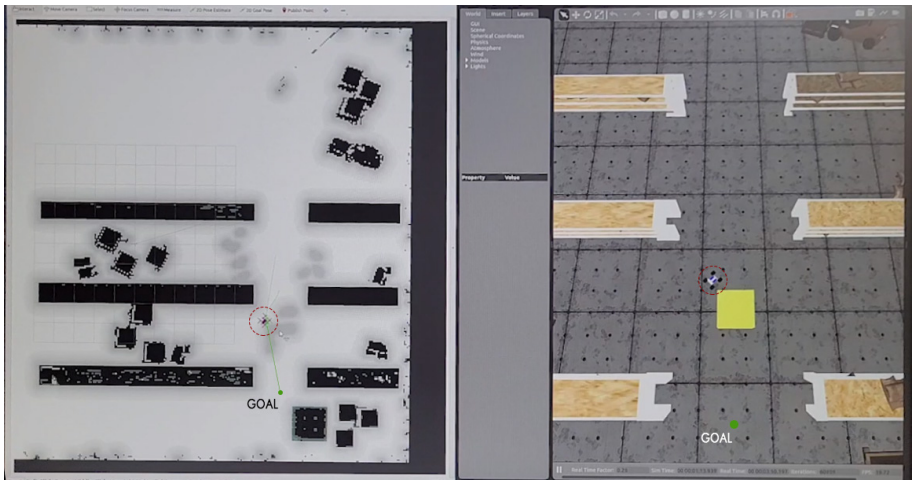


Figura 2.6: Ricalcolo del percorso e sterzatura intorno all'ostacolo

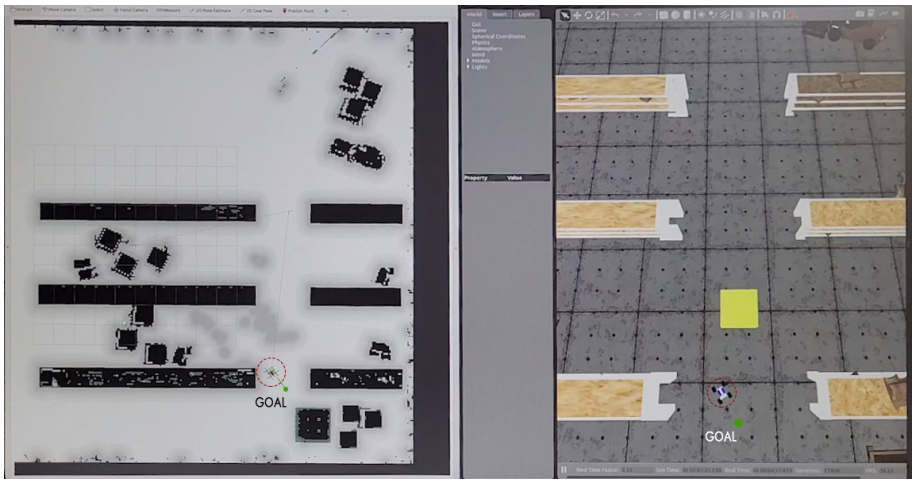


Figura 2.7: Approccio alla destinazione finale

Per ottenere i risultati appena mostrati, oltre alla modifica dei parametri relativi al Controller Server e al Planner Server descritti nelle Sezioni 1.4.5 e 1.4.6, che sono stati necessari per adattare gli algoritmi alla dinamica del veicolo, sono stati anche aggiornati alcuni parametri relativi alla global costmap e alla local costmap, per migliorare il comportamento del veicolo durante la navigazione. Mentre i parametri relativi alla dinamica del veicolo sono stati aggiornati a priori, poiché sono parametri oggettivi e conosciuti, quelli mostrati di seguito sono stati adattati migliorando alcuni comportamenti errati del veicolo constatati durante le simulazioni:

Nome parametro	**Valore**	**Comportamento Migliorato**
`inflation_radius`	0.7	Leggero aumento della distanza consentita dalla costmap da oggetti presenti nella mappa, per migliorare il comportamento del veicolo vicino ad angoli
`cost_scaling_factor`	3.0	Fattore di decadimento intorno all'area non consentita: non permette al veicolo di accedere alle zone "pericolose"

Tabella 2.1: Parametri relativi alla global costmap e alla local costmap

Capitolo 3

Applicazione degli Algoritmi Studiati su un Veicolo Reale in Scala

Questa sezione rappresenta l'obiettivo finale della tesi: realizzare un algoritmo di navigazione che permetta ad un veicolo reale (in scala 1:10) di spostarsi da un punto A verso un punto B in maniera completamente autonoma, interagendo con l'ambiente circostante ed evitando eventuali ostacoli che potrebbe incontrare lungo il suo percorso. Per poter realizzare questo, nel Capitolo 1 è stato fatto uno studio approfondito di tutto il software e di tutte le interfacce necessarie per una buona esecuzione del sistema, mentre nel Capitolo 2 tali algoritmi sono stati applicati in simulazione. Il passaggio da un ambiente simulato ad uno reale non è sempre immediato; infatti, molto spesso è necessario rimettere mano al codice sviluppato per aggiornarlo ed effettuare aggiustamenti, dovuti principalmente al fatto che ad esempio, un sensore reale non sarà mai perfetto come uno simulato, che estrapola informazioni dal physics engine di Gazebo, descritto da sole equazioni, mentre il sensore reale misura grandezze fisiche tramite elementi sensibili ed è affetto da rumore. In più, il modello dinamico del veicolo sviluppato in simulazione, per quanto preciso esso sia, non rispecchierà appieno il comportamento del suo corrispettivo nella realtà.
In questo capitolo si descriverà in primo luogo il veicolo e tutto l'hardware utilizzato a bordo di esso, per poi passare al software sviluppato per poter interfacciare il sistema di navigazione testato nel capitolo precedente con il veicolo stesso.
In particolare, il modellino in questione è un RC-Car Traxxas TRX-4, modificato appositamente per poter supportare ed eseguire algoritmi per la guida autonoma. A tal proposito, sul veicolo è stata installata un'unità di calcolo NVIDIA Jetson AGX Xavier, basata su processore ARM con architettura a 64 bit.
Su di essa andranno eseguiti tutti i task necessari al controllo dei vari sensori, attraverso il framework ROS, su sistema operativo Linux Ubuntu 18.04. Come sarà descritto in seguito, la scheda di sviluppo NVIDIA Jetson AGX Xavier è stata utilizzata soltanto per l'acquisizione dati dei sensori montati a bordo del veicolo, mentre gli algoritmi di odometria e di navigazione sono stati demandati ad un PC esterno con architettura x86 a 64 bit. Questo si è reso necessario a causa di un'incompatibilità dovuta all'architettura riscontrata durante l'installazione del pacchetto RTAB-Map, che non ha reso possibile applicare gli algoritmi studiati in simulazione sulla scheda di sviluppo NVIDIA.

Per poter interagire con l'ambiente circostante ed effettuare tutte le misurazioni necessarie per la localizzazione, il veicolo è dotato di un LIDAR, di una IMU e di una RGB-D camera. Inoltre, la NVIDIA Jetson AGX Xavier può essere controllata dallo sviluppatore in modalità wireless, grazie ad un router Wi-Fi.
All'interno della scheda di sviluppo è stato installato l'ambiente ROS Melodic, necessario per acquisire dati dai sensori.

Gli step eseguiti in questa sezione saranno:

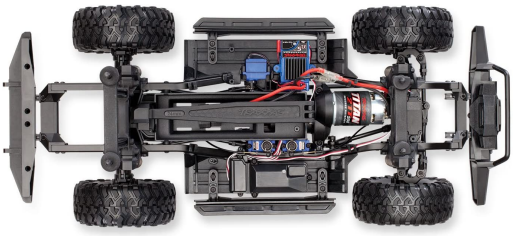
- Installazione di tutti i pacchetti necessari all'acquisizione dati dai sensori;
- Creazione di una mappa dell'ambiente di test tramite SLAM, utilizzando il pacchetto `hector_slam`;
- Settaggio del pacchetto `navigation2` fornendo la mappa acquisita tramite SLAM e impostazione del controller server e del planner server più opportuni;
- Scrittura di uno script in linguaggio Python per il controllo a basso livello del veicolo;
- Prova di navigazione autonoma all'interno dell'ambiente di test.

Nel seguito saranno descritte tutte le componenti utilizzate in questa prova. Successivamente saranno riportate le prove di guida autonoma effettuate.

3.1 Veicolo RC-Car Traxxas TRX-4

Il veicolo utilizzato per il test di guida autonoma è il Traxxas TRX-4. Esso è dotato di un motore a collettore Titan 21T 550 dedicato alla trazione, che può essere trasmessa a tutte le ruote, mentre per lo sterzo viene utilizzato un servomotore.

(a) Veicolo Traxxas TRX-4



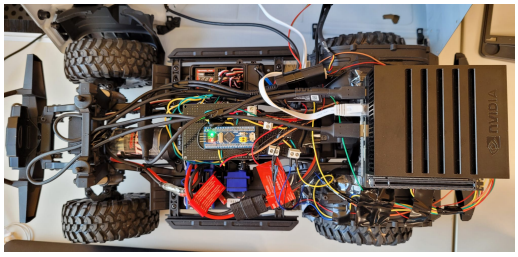
(b) Vista dall'alto dello chassis

Figura 3.1: Traxxas TRX-4

Grazie alle sue dimensioni, sullo chassis del veicolo è stato possibile montare tutti i dispositivi necessari per renderlo autonomo. In particolare, sulla parte posteriore è stata installata una NVIDIA Jetson AGX Xavier. Il veicolo in questione risulta particolarmente adatto per applicazioni outdoor, come ad esempio la navigazione su terreni sconnessi. Infatti esso è dotato di sospensioni a molla abbinate ad ammortizzatori in alluminio e di trazione integrale.

Direttamente sullo chassis è stata installata la scheda NVIDIA Jetson AGX Xavier, oltre ad una scheda di sviluppo per il controllo a basso livello, che si pone da interfaccia tra la NVIDIA Jetson e la ESC del motore. Inoltre, sul tetto del veicolo sono presenti un power bank con capacità da 25000mAh, un router GL-AR750 e una IMU Bosch BNO055.

(a) Vista dall'alto dei dispositivi installati sul tetto



(b) Vista dei dispositivi installati all'interno

Figura 3.2: Dispositivi installati sul veicolo

3.1.1 Motori installati

I motori che permettono al veicolo di muoversi ed essere controllato sono principalmente due. Il motore a collettore dedicato alla trazione è un Titan 21T 550, alimentato a 14V. Di seguito ne viene illustrata la scheda tecnica:

Specifica	Valore
Tipo Motore	Collettore
N. Avvolgimenti	21T
Diametro Motore	37mm
Lunghezza Motore	78mm
Diametro Albero	3.12mm
Connettore	Bullet 3.5mm
Tensione Input Max	14V

Tabella 3.1: Datasheet del motore Titan 21T 550

Figura 3.3: Motore Titan 21T 550

Il motore a collettore del veicolo viene controllato da una ESC (Electronic Speed Controller), che funge da interfaccia tra i segnali inviati dal controllore e l'effettivo sforzo di controllo fornito al motore. Tale dispositivo si occupa del controllo a basso livello ed è capace di generare una tensione negativa per invertire il senso di rotazione del motore e permette al controllore ad alto livello di fornire un input in velocità senza preoccuparsi di controllare la tensione fornita al motore con una modulazione PWM.

Il servomotore utilizzato per lo sterzo garantisce una buona coppia grazie alla sua composizione in metallo. Di seguito vengono elencate le caratteristiche principali:

Specifica	Valore
Struttura e Composizione	Ingranaggi in metallo, cuscinetto a sfera
Coppia	9kg-cm
Tempi di Spostamento	0.17 sec/60°
Dimensioni	L=55mm, H= 42.75mm, W=21.5mm

Tabella 3.2: Datasheet del servomotore installato nel veicolo

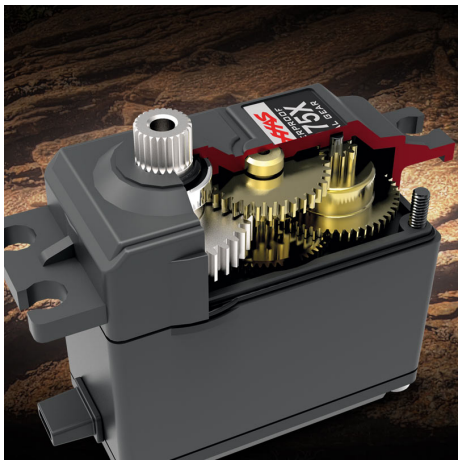


Figura 3.4: Metal Gear Servo Traxxas 2075X

3.1.2 Sensori Implementati

I sensori montati a bordo del veicolo servono sostanzialmente a ricavare una buona odometria e localizzarsi di conseguenza all'interno della mappa. La videocamera, una Intel Realsense D435 fornisce, oltre ad un'immagine a colori dell'ambiente che inquadra, anche una depth image, nella quale sono racchiuse le informazioni relative alla profondità. Il funzionamento dettagliato della camera con la descrizione di tutti i suoi componenti sarà illustrata nella Sezione 3.2.
La videocamera sarà sfruttata per ricavare una stima dell'odometria, in particolare una Visual Odometry, che sarà integrata con le informazioni provenienti dalla IMU (Unità di Misura Inerziale) grazie all'operazione di Sensor Fusion, che sarà performata dal pacchetto `robot_localization`, grazie ad un Extended Kalman Filter, il cui funzionamento è illustrato in dettaglio nella Sezione 1.7.

Il veicolo è dotato anche di un sensore LIDAR, in particolare un RPLIDAR, che fornisce una nuvola di punti 2D, parallela al terreno e situata ad un'altezza pari a quella del sensore. Tale mappa sarà molto importante per il nodo `AMCL` di Navigation2 e consentirà, insieme al dato di odometria, di localizzare istante per istante il veicolo all'interno della mappa e di seguire il path programmato dal Global Planner.

3.1.3 Dispositivi di Supporto (Power Bank, Batteria, Router)

Il veicolo è dotato di una batteria collegata ai motori, fornita insieme al veicolo. Tale batteria, con tecnologia LiPo, ha una capacità di 5800mAh, eroga una tensione di 7.4V e si occupa di alimentare il motore di trazione ed il servomotore del veicolo. I pacchi batteria LiPo sono composti da celle piatte da 3,7 volt. Le celle sono impilate e racchiuse in un involucro semirigido. Le batterie LiPo Traxxas sono disponibili in configurazioni a 2 celle[1] (7,4V), 3 celle (11,1V) e 4 celle (14,8V). Le batterie LiPo hanno una maggiore "densità di energia" rispetto alle batterie NiMH, il che significa che, a parità di volume, hanno più tensione e capacità di una batteria NiMH. Le

[1]Nel veicolo utilizzato in questo elaborato è stata adottata una batteria a 2 celle

batterie LiPo sostengono anche una tensione più elevata per una maggiore durata di ogni corsa.

Per quanto concerne l'alimentazione dell'unità di calcolo NVIDIA Jetson AGX Xavier e del router Wi-Fi è stato impiegato un power bank con una capacità di 25000mAh che eroga una tensione di 20V. I sensori invece, prendono la loro alimentazione direttamente dall'unità NVIDIA Jetson.

Il power bank è collegato alla scheda Jetson tramite semplici connettori DC, mentre la batteria LiPo fornisce energia alla ESC tramite connettori proprietari Traxxas.

3.2 Intel Realsense D435 Depth Camera

La Intel Realsense D435 è una RGB-D camera, capace di fornire depth images di alta qualità. Risulta pertanto particolarmente valida per lo scopo di questo progetto. Inoltre, il suo ampio angolo di campo è molto utile per individuare più feature points possibili, fondamentali per effettuare una buona Visual Odometry. Di seguito, in Figura 3.5, è possibile apprezzare un dettaglio di tutte le componenti della camera.

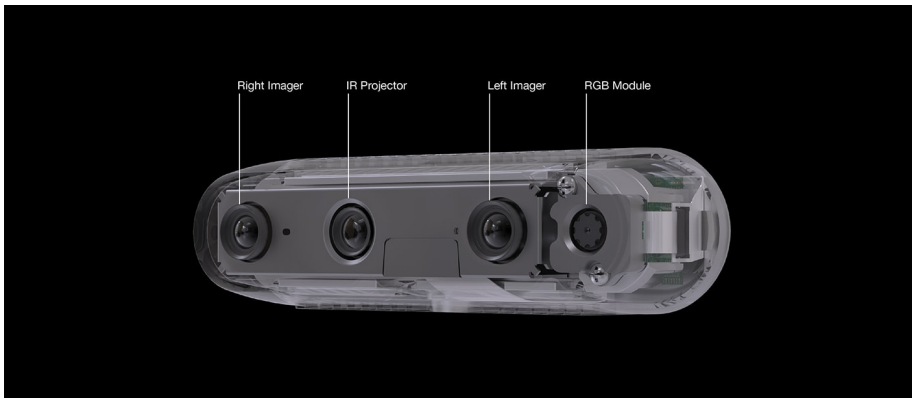


Figura 3.5: Moduli implementati all'interno della camera Intel Realsense D435

In particolare, all'interno della camera sono implementati:

- Left Imager e Right Imager: il modulo relativo alla profondità, o depth module, possiede due sensori, chiamati left imager e right imager. Questi due ricevitori di raggi infrarossi sono identici e configurati allo stesso modo. Essendo posizionati con un certo offset orizzontale, possiedono una differenza di prospettiva. Tale differenza viene elaborata dal processore interno della camera e viene fornito uno stream di dati che contiene già una depth image. Un esempio realizzato inquadrando il veicolo viene illustrato di seguito:

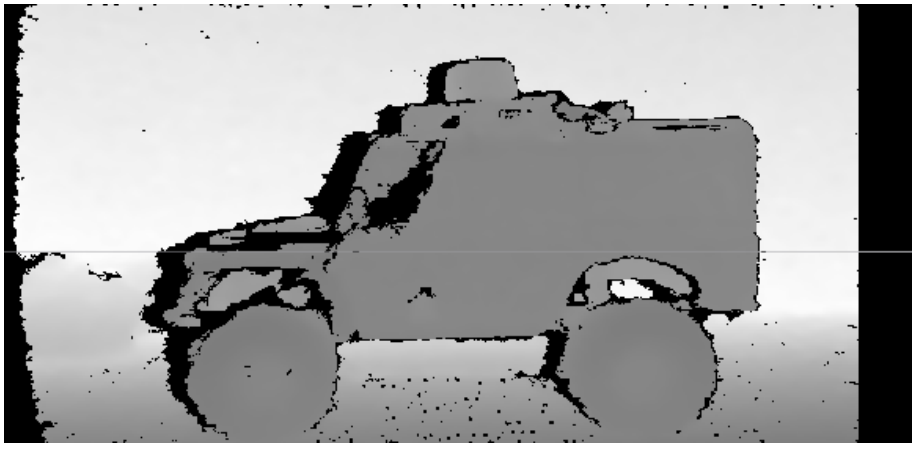


Figura 3.6: Esempio di depth image ricavata grazie al right imager e al left imager

- IR Projector: per permettere il corretto funzionamento del left imager e del right imager montati a bordo della RGB-D camera, è presente un proiettore ad infrarossi. Grazie a tale componente la camera è capace di aumentare la

consistenza e il contrasto della scena inquadrata, ottenendo un'immagine più nitida e ricca di informazioni.

- Color Sensor: in ultimo, è presente anche un sensore RGB che cattura immagini a colori. Le informazioni ricavate da tale sensore vengono fuse insieme a quelle fornite dai sensori per la profondità, al fine di ottenere una nuvola di punti a colori e quindi una ricostruzione 3D dell'ambiente. Tale operazione, come per la depth image, viene effettuata a bordo della camera grazie al processore integrato.

3.2.1 Posizionamento della Camera

La scelta della posizione della camera è molto importante per il corretto funzionamento della Visual Odometry. I requisiti più importanti sono:

- Campo visivo ottimale;
- Mancanza di oggetti che ostruiscono la visuale;
- Struttura di supporto della camera solida, per evitare movimenti.

In Figura 3.7 è mostrata la depth camera Intel Realsense D435 montata a bordo del veicolo attraverso un apposito supporto:

Figura 3.7: Struttura di montaggio della camera Intel Realsense D435

3.2.2 Principio della Visual Odometry

La camera invia al controllore principalmente due flussi di dati: uno di fotogrammi RGB e uno di depth images. Ogni fotogramma RGB contiene informazioni visive dell'ambiente, mentre l'altro flusso racchiude in ogni pixel dell'immagine ricevuta un'informazione relativa alla profondità alla quale si trova un oggetto catturato in quella posizione. Sfruttando queste informazioni, il pacchetto RTAB-Map riesce a

ricavare una Visual Odometry, molto utile per la localizzazione del veicolo durante la navigazione.

Sfruttando questi due flussi di informazioni, l'algoritmo FVO presente all'interno del pacchetto RTAB-Map riesce ad agganciare dei feauture points. Durante il movimento del veicolo, la posizione relativa tra i feature points cambia, e di conseguenza viene stimato un cambiamento di posizione del veicolo, correlato con una velocità ed un'accelerazione. La posizione del veicolo all'istante k dipende strettamente da quella all'istante $k-1$, pertanto l'errore commesso nella stima di posizione si propaga e si accumula nel tempo. Per ridurre questo errore si può ricorrere a sistemi di localizzazione supplementari, come il sistema GPS, utilizzato soltanto in applicazioni di tipo outdoor. In questo elaborato il valore di Visual Odometry è stato fuso con le informazioni provenienti dalla IMU per migliorare la stima e di conseguenza ridurre l'errore. Tramite il laser scanner e l'algoritmo di localizzazione AMCL, illustrato nella Sezione 1.4.7, è possibile migliorare ulteriormente la stima di posizione del veicolo all'interno della mappa.

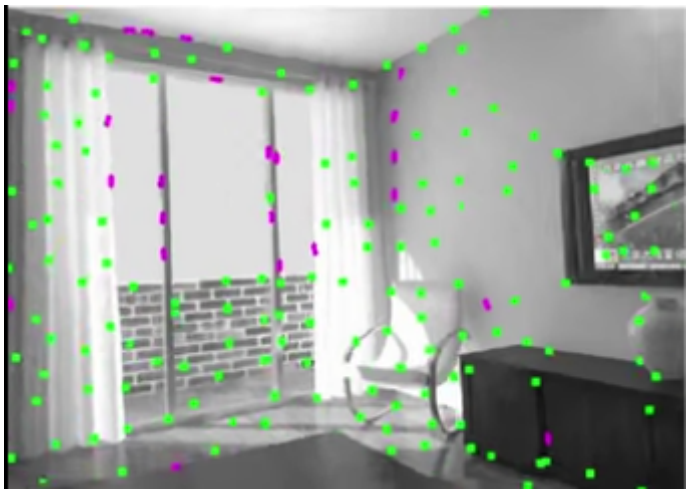


Figura 3.8: Markers catturati dalla Visual Odometry

Inoltre, tramite le immagini RGB e di profondità, è possibile ricavare una mappa 3D dell'ambiente, a partire da una nuvola di punti 3D che viene costruita facendo un matching dei due flussi, e tenendo conto dei movimenti effettuati dal veicolo. Per ogni immagine acquisita dal dispositivo - ad esempio 25 ogni secondo - la rispettiva nuvola di punti è fusa con quella ottenuta nelle posizioni precedenti per ottenere una mappa globale.

3.2.3 Implementazione e Topic Pubblicati

Per poter ottenere tutte le informazioni elaborate dalla camera Realsense D435 è necessario lanciare un apposito ROS Node che, facendo da tramite tra il processore della camera e l'ambiente ROS, pubblica tutti i dati di depth image, color image e depth cloud all'interno dei topic designati. I pacchetti di localizzazione e navigazione si iscriveranno a tali topic per leggere le informazioni della camera.

Il ROS node che si occupa di effettuare le operazioni sopra descritte può essere ottenuto installando il pacchetto `realsense-ros` dalla relativa repo in GitHub e lanciare il seguente comando:

```
roslaunch realsense2_camera rs_camera.launch
```

I topic pubblicati da tale nodo sono i seguenti:

- `/camera/color/camera_info`: topic relativo alle caratteristiche dell'immagine pubblicata, come dimensione in pixel e coefficienti di calibrazione e distorsione;
- `/camera/color/image_raw`: topic all'interno del quale vengono pubblicate le matrici di pixel in formato raw. Il tool grafico, che in questo caso corrisponde a RViz, si serve del topic `/camera/color/camera_info` per impostare correttamente i dati provenienti dal topic `/camera/color/image_raw`;
- `/camera/depth/camera_info`: informazioni relative all'immagine di profondità ricevuta, in modo analogo al caso precedente;
- `/camera/depth/image_rect_raw`: immagini di profondità ricevute in forma di matrici di pixel, con valori in scala di grigi, dove valori tendenti al nero (0) corrispondono ad oggetti vicini, mentre valori tendenti al bianco (255) corrispondono ad oggetti lontani.

3.3 RPLIDAR: Funzionamento e Topic Pubblicati

Il dispositivo LIDAR utilizzato in questo elaborato è un RPLIDAR A2. Esso riesce ad elaborare una mappa 2D dell'ambiente circostante, sfruttando la tecnologia del laser range scanner. A tale scopo, RPLIDAR è dotato di un motore che gli permette di ruotare su sé stesso di 360° ed effettuare quindi una scansione dell'ambiente, inviando impulsi laser e ricevendoli grazie ad un apposito ricevitore.

Figura 3.9: RPLIDAR A2

La scelta di questo sensore è ovviamente correlata con il suo scopo finale, che consiste nel monitoraggio dell'ambiente circostante al fine di evitare collisioni con oggetti vicini. Il dispositivo riesce a ricavare soltanto una mappa in 2D degli oggetti circostanti, pertanto una criticità potrebbe essere quella di non riuscire a rilevare oggetti più bassi dell'altezza di montaggio. Questo tipo di ostacoli è comunque molto specifico e non giustifica l'utilizzo di un LIDAR 3D che avrebbe comportato un costo molto più elevato.

Il principio di misurazione della distanza dall'oggetto che riflette il raggio laser si basa sulla triangolazione: ovvero, il diodo laser emette una radiazione luminosa in direzione di un bersaglio posto ad una distanza di misura *d*. Tale impulso viene riflesso dalla superficie e incide su un dispositivo di tipo CCD (Charge Coupled Device). All'interno del dispositivo CCD viene colpito un punto preciso, la cui distanza dal centro è direttamente proporzionale alla distanza di misura.

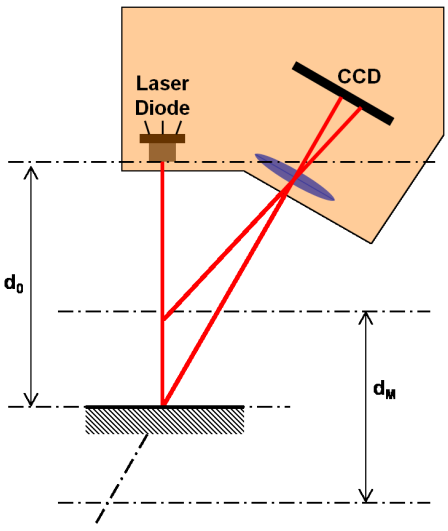


Figura 3.10: Principio di funzionamento del RPLIDAR A2

Di seguito viene mostrato il datasheet del dispositivo:

Parametro	**Min**	**Tipico**	**Max**
Range Distanza	n/a	0.15 - 6 m	n/a
Range Angolare	n/a	0 - 360°	n/a
Risoluzione @ Distanza	n/a	< 0.5 mm @ < 1.5m <1% distanza @ tutto il range distanza	n/a
Risoluzione Angolare	0.45	0.9	1.35
Frequenza Campionamento	2000Hz	>= 4000Hz	4100Hz
Rate di Scansione	5Hz	10Hz	15Hz

Tabella 3.3: Datasheet del sensore RPLIDAR A2

Il dispositivo RPLIDAR è equipaggiato con un motor driver che conferisce allo scanner la possibilità di regolare la propria velocità di rotazione. Esso infatti può essere controllato via PWM. Ovvero, regolando la frequenza e il duty cycle del segnale inviato al motor driver, è possibile regolare la velocità di rotazione. Una modifica di questo tipo potrebbe risultare necessaria nel caso in cui si desiderasse una precisione maggiore nella scansione o, viceversa, una frequenza di campionamento più elevata.

Durante il suo funzionamento a regime, il LIDAR invia alla NVIDIA Jetson una sequenza di informazioni, con una frequenza di 10Hz. All'interno di tali messaggi sono presenti tutti i punti ricavati durante l'ultima scansione a 360°. Come nel caso degli altri sensori, un ROS Node è responsabile dell'interfacciamento tra dispositivo e sistema software. In particolare, tale ROS Node si trova all'interno del ROS Package `rp_lidar_ros` e può essere lanciato da shell grazie al comando:

```
1 roslaunch rp_lidar_ros rp_lidar.launch
```

L'effetto del lancio di questo nodo è la pubblicazione del topic `/scan`, all'interno del quale sono presenti informazioni strutturate di tutte le scansioni effettuate dal LIDAR. Tale topic risulterà particolarmente importante per il nodo AMCL, responsabile della localizzazione del veicolo all'interno della mappa.

3.4 IMU Bosch BNO055

Lo Smart Sensor Bosch BNO055 è un'Unità di Misura Inerziale (IMU) che integra tre sensori al suo interno: esso è dotato di un giroscopio a tre assi, un accelerometro a 3 assi e di un sensore geomagnetico. Grazie all'elaborazione interna dei dati provenienti dai vari sensori, lo Smart Sensor è capace di fornire in output informazioni ottenute tramite sensor fusion, come orientamento in quaternioni, angoli di Eulero, vettore di rotazione, accelerazione lineare, gravità.

Questo tipo di sensori trova numerose applicazioni. Ad esempio sono molto utilizzati per il bilanciamento di robot mobili, come Segway. Essi vengono impiegati anche in ambito aeronautico, per controllare gli angoli di inclinazione dell'aeromobile (rollio, beccheggio, imbardata). Grazie alle misure di accelerazione lineare ed angolare, è capace di rilevare i movimenti di un veicolo. Una IMU può essere utilizzata ad esempio per il task di localizzazione del veicolo all'interno di una mappa, affiancata ad un sensore GPS, che possiede una funzione correttiva. Infatti, un sensore inerziale riesce a fornire informazioni di localizzazione con un rate molto alto, ma è affetto dal fenomeno di drifting, ovvero, la stima della posizione all'istante corrente dipende strettamente dalla stima effettuata al passo precedente, pertanto l'errore tende ad accumularsi nel tempo. Questo metodo di localizzazione viene chiamato dead reckoning. Per correggere l'errore di stima entra in gioco il sistema GPS, che ha il compito di correggere la posizione assoluta del veicolo. I due dispositivi vengono utilizzati in stretta collaborazione, poiché nonostante il GPS sia molto preciso, possiede una frequenza di refresh della posizione molto inferiore rispetto a quella della IMU. La configurazione sensoristica mostrata in questo esempio sarebbe particolarmente adatta ad un'applicazione di tipo outdoor, poiché però in questa tesi si è sviluppato un sistema che lavorasse in ambiente indoor, si è scelto di adottare una soluzione differente.

Come è possibile intuire dal paragrafo precedente, per eseguire un task di localizzazione, a causa del fenomeno di drifting, è necessario affiancare il sensore di misura inerziale con altri dispositivi. Nel caso di questo elaborato, i dati provenienti dalla IMU sono stati sfruttati per realizzare un'operazione di sensor fusion (Cap. 1.7) con le informazioni di odometria provenienti dalla stereo camera Intel Realsense D435. Nonostante l'introduzione di un EKF abbia migliorato l'output odometrico, l'errore di drifting non può essere eliminato del tutto. Esso infatti viene corretto periodicamente dall'algoritmo di localizzazione AMCL (Sezione 1.4.7), basato su particle filter.

Figura 3.11: IMU Bosch BNO055

Lo smart sensor Bosch BNO055 è alimentato da una tensione che ha un range di valori che va da 2.4V a 3.6V, ed è possibile interfacciarsi ad essa per ricevere i dati elaborati tramite I^2C o UART. Di seguito sono elencate le principali feature del componente:

- Dati in output: orientamento in quaternioni e angoli di Eulero, accelerazione lineare, gravità;
- Tre sensori installati:

  Accelerometro triassiale a 14bit

  Giroscopio triassiale a 16bit

  Magnetometro

L'accelerometro si occupa di misurare l'accelerazione lineare che un corpo possiede. In questo caso, trattandosi di un accelerometro a tre assi, esso riesce a misurare le accelerazioni lineari lungo tutti e tre gli assi coordinati del sistema corpo. Le caratteristiche principali dell'accelerometro montato a bordo del sensore inerziale Bosch BNO055 sono elencate di seguito:

- Funzionalità programmabili:
  - Range di accelerazione: ±2g/±4g/±8g/±16g
  - Larghezza di banda del filtro passa-basso: 1kHz - <8Hz
  - Modalità operative:
    * Normal
    * Suspend
    * Low power
    * Standby
    * Deep suspend

- Controllore di interrupt:
  - Generazione di segnale di interrupt in conseguenza di un movimento:
    - Rilevamento di moto
    - Rilevamento di rallentamento
    - Rilevamento di situazione "high-g"

Il giroscopio è un dispositivo di tipo propriocettivo ed è adibito alla misura di una velocità angolare. Anche in questo caso, poiché si tratta di un dispositivo triassiale, il sensore misura le velocità angolari del veicolo attorno a tutti e tre gli assi coordinati.

- Funzionalità programmabili:
  - Range di misura: da $\pm 125°/s$ a $\pm 2000°/s$
  - Larghezza di banda del filtro passa-basso: 523Hz - 12Hz
  - Modalità operative:
    - Normal
    - Fast power up
    - Deep suspend
    - Suspend
    - Advanced power save
- Controllore di interrupt:
  - Generazione di segnale di interrupt in conseguenza di un movimento:
    - Rilevamento di moto
    - Alta velocità angolare

In ultimo, lo smart sensor Bosch BNO055 è equipaggiato con un magnetometro. Il compito di tale dispositivo è di fornire una misura del campo magnetico terrestre, rispetto ai tre assi coordinati del veicolo. Poiché il veicolo sul quale è montata la IMU possiede parti in metallo e ogni metallo ha un suo campo magnetico, alla prima installazione è necessario calibrare il magnetometro perché misuri soltanto il campo magnetico terrestre. Questo sensore può essere utilizzato come bussola, nella modalità "compass". Esso infatti, calcolando la direzione del campo magnetico terrestre, riesce a ricavare la direzione geografica del Nord. Un altro possibile impiego per il magnetometro installato a bordo della IMU, potrebbe essere quello di ricavare l'orientamento del veicolo. Questa modalità, chiamata "M4G" consiste nel far eseguire al magnetometro il compito che normalmente avrebbe il giroscopio, ovvero ricavare l'orientamento del veicolo. Questo viene fatto in casi in cui è necessario un alto risparmio energetico. Infatti, il consumo in termini di energia del magnetometro è di molto inferiore rispetto a quello del giroscopio.

Di seguito vengono elencate le caratteristiche del sensore geomagnetico:

- Range di campo magnetico: $\pm 1300\mu$T (assi x, y); $\pm 2500\mu$T (asse z)

- Risoluzione del campo magnetico: 0.3μT
- Modalità operative:
  - Low power
  - Regular
  - Enhanced regular
  - High Accuracy
- Modalità energetiche:
  - Normal
  - Sleep
  - Suspend
  - Force

3.4.1 Principio di Funzionamento della IMU

La IMU (Inertial Measurement Unit) è costruita per misurare le grandezze di tipo inerziale. A tale scopo gli accelerometri ed i giroscopi sono posizionati in un modo ben preciso. La configurazione classica, che è quella utilizzata da questo modello di IMU, è detta strapdown, all'interno della quale gli assi sensibili dei sensori inerziali (giroscopi e accelerometri) sono mutuamente ortogonali in un sistema di riferimento cartesiano. I sensori sono connessi rigidamente all'involucro della IMU, che è a sua volta installata sull'oggetto di cui si vuole misurare il moto, mediante un sistema anti-vibrazione.

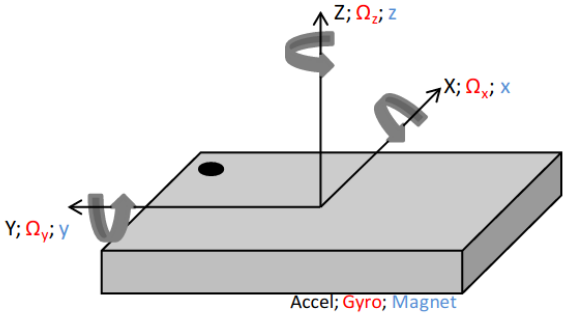


Figura 3.12: Configurazione dell'Unità di Misura Inerziale

Il cluster di sensori è dotato di un elettronica di supporto che ha lo scopo di elaborare i dati grezzi provenienti dalle misurazioni inerziali e convertirli in informazioni relative ad orientamento ed accelerazione del veicolo. Nella IMU è integrato un processore che permette di operare piccole forme di compensazione online dell'uscita del sensore basate sui dati di calibrazione ottenuti in laboratorio o sui dati di taratura ottenuti in fase di produzione.

All'interno dello smart sensor è presente anche un sensore di temperatura. È possibile infatti ottenere un output di tale grandezza in °C, ma essa viene principalmente utilizzata per motivi di compensazione: poiché la caratteristica statica dei

sensori è spesso dipendente dalla temperatura, per ottenere prestazioni soddisfacenti è necessario effettuare una compensazione mediante un circuito collegato ad un sensore di temperatura.

Tipicamente, le fonti principali di errore di misura relative alle IMU possono essere di vari tipi:

- Bias: valore additivo indipendente relativo ad accelerazione e velocità angolare, che si manifesta per il 90% all'accensione e per il 10% per effetti di temperatura.
- Fattore di scala: rapporto tra variazione del misurando e variazione della misura
- Non ortogonalità: la configurazione strapdown richiede ortogonalità tra gli assi sensibili dei sensori: se questa ortogonalità viene meno (incertezza nella costruzione del sensore) allora si manifesta un errore. Di solito questo errore viene corretto in fase di calibrazione mediante una serie di test che prevede rotazioni ortogonali.
- Rumore: deriva dalle instabilità elettriche e meccaniche, ha distribuzione gaussiana a riposo che tuttavia può variare durante il movimento dell'unità: è possibile limitarne (ma non annullarne) gli effetti con filtraggio.

3.5 NVIDIA Jetson AGX Xavier

La NVIDIA Jetson AGX Xavier rappresenta una soluzione embedded avanzata per gli sviluppatori di Intelligenza Artificiale ed è l'ideale per implementare algoritmi avanzati di computer vision. La sua potenza elevata e le sue dimensioni molto ridotte consentono allo sviluppatore di operare direttamente su piattaforme robotiche, che agiscono sul campo con prestazioni a livello di workstation e operano in modo completamente autonomo senza fare affidamento sull'intervento umano o sulla connettività cloud.
Le macchine intelligenti alimentate da Jetson AGX Xavier hanno la libertà di interagire e navigare in sicurezza nei loro ambienti, libere da terreni complessi e ostacoli dinamici, svolgendo attività del mondo reale con completa autonomia. Ciò include task come package delivery e industrial inspection, che richiedono alti standard di elaborazione per essere eseguiti con precisione. A tale scopo, la NVIDIA Jetson AGX Xavier può gestire algoritmi complessi, come quelli di elaborazione della Visual Odometry, sensor fusion, localization and mapping, obstacle detection e path planning, pregio che la rende particolarmente adatta al lavoro sviluppato in questo elaborato.

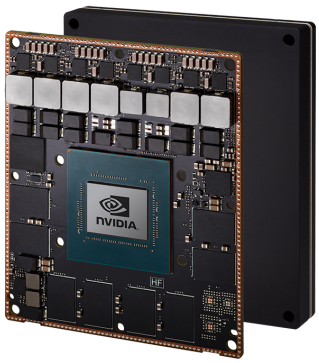


Figura 3.13: Vista dei chip integrati a bordo della NVIDIA Jetson AGX Xavier

Il principale punto di forza di questa soluzione embedded sta nelle dimensioni. In soli 100x87mm essa offre eccellenti prestazioni da workstation in dimensioni molto ridotte. Infatti, la NVIDIA Jetson AGX Xavier integra una GPU con una capacità di calcolo pari a 32 TeraOPS (TOPS), ovvero 32 Tera operazioni al secondo e una velocità di trasferimento I/O fino a 750Gbps.
Più specificatamente, la piattaforma integra una CPU ARM 8-core con architettura a 64bit, una GPU NVIDIA Volta dotata di 512 core, che includono 64 Tensor Cores, molto utili per accelerare i calcoli relativi ad applicazioni di deep learning. In particolare i Tensor Cores, integrati nella GPU NVIDIA Volta, vengono impiegati per accelerare ed ottimizzare complesse operazioni di algebra lineare ed elaborazione di segnali. Il data sheet completo della piattaforma embedded appena descritta viene mostrato di seguito:

Modulo NVIDIA Jetson AGX Xavier	
CPU	8-core NVIDIA Carmel 64-bit ARMv8.2 @ 2265MHz
GPU	512-core NVIDIA Volta @ 1377MHz with 64 TensorCores
Deep Learning	Dual NVIDIA Deep Learning Accelerators (DLAs)
Memory	16GB 256-bit LPDDR4x @ 2133MHz — 137GB/s
Storage	32GB eMMC 5.1
Vision	(2x) 7-way VLIW Vision Accelerator
USB	(3x) USB 3.1 + (4x) USB 2.0
Misc I/Os	UART, SPI, I2C, I2S, GPIOs
Power	10W / 15W / 30W profiles, 9.0V-20VDC input

Tabella 3.4: Specifiche della scheda di sviluppo NVIDIA Jetson AGX Xavier

3.5.1 GPU Volta

La GPU Nvidia Volta, mostrata in figura, integra 512 CUDA Cores e 64 Tensor Cores, con una massima frequenza di clock di 1.37GHz. Tali unità di calcolo sono suddivise in 8 SM (Streaming Multiprocessor), in ognuno dei quali sono presenti 64 CUDA Cores e 8 Tensor Cores. Ad ogni Multiprocessor è affiancata una cache L1 da 128KB. Per aumentare ancora la velocità di calcolo, i multiprocessori condividono un ulteriore memoria cache di tipo L2 da 512KB.

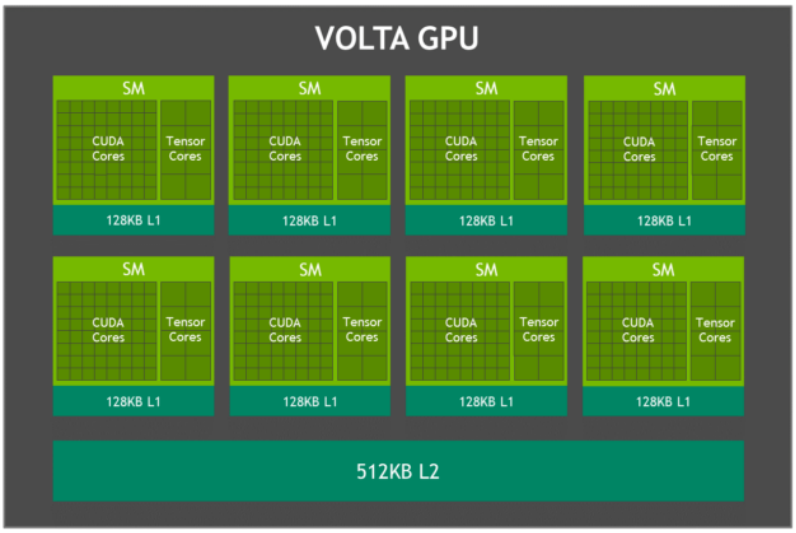


Figura 3.14: Schema compositivo della GPU Volta

Ogni SM è costituito da quattro blocchi di elaborazione separati denominati SMPS (Streaming Multiprocessor Partitions), ognuno con la propria cache L0, warp scheduler, dispatch unit e register file, oltre ai CUDA e ai Tensor Cores.

Tensor Cores

I Tensor Cores NVIDIA sono unità programmabili volte all'esecuzione di moltiplicazioni e somma di matrici, che vengono eseguiti in parallelo con i CUDA Cores. I Tensor Core implementano le nuove istruzioni HMMA a virgola mobile (Half-Precision Matrix Multiply and Accumulate) e IMMA (Integer Matrix Multiply and

Accumulate) per l'accelerazione di complessi calcoli di algebra lineare, elaborazione di segnali e inferenza[2] con deep learning.

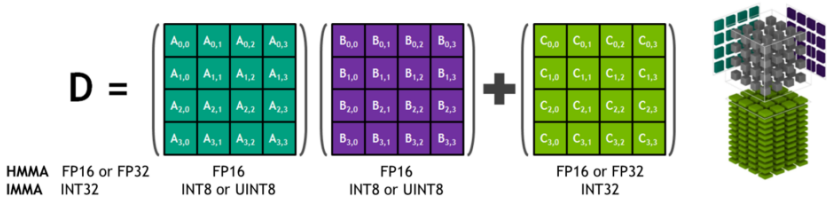


Figura 3.15: Principio di funzionamento di un Tensor Core

Il tipo di dato relativo agli elementi delle varie matrici dipende dal tipo di istruzione utilizzata per eseguire la moltiplicazione o l'accumulazione. Come si può apprezzare dalla figura, il Tensor Core si occupa di effettuare operazioni di algebra lineare, ed essendo studiato apposta per queste applicazioni, la precisione e la capacità di elaborazione sono sufficienti per evitare condizioni di overflow e underflow durante l'accumulazione. La tecnologia dei Tensor Cores ha accelerato significativamente i tempi di training relativi all'IA, da settimane a ore.

Deep Learning Accelerator (DLA) Jetson AGX Xavier dispone di due motori NVIDIA Deep Learning Accelerator (DLA), che si occupano dell'inferenza relativa alle Convolutional Neural Network (CNN). Questi componenti migliorano l'efficienza energetica e liberano la GPU, che può eseguire reti più complesse e attività dinamiche implementate dall'utente. Ogni DLA ha una prestazione fino a 5 TOP INT8 o 2.5 TFLOPS FP16 con un consumo di energia di soli 0.5-1.5 W. Il supporto dei DLA per l'accelerazione dei livelli della CNN, come convoluzione, deconvoluzione, funzioni di attivazione, min/max/mean pooling, normalizzazione della risposta locale e livelli completamente connessi.

Struttura della CPU La struttura delle CPU della NVIDIA Jetson AGX Xavier è mostrato nella figura sottostante ed è costituito da quattro cluster CPU NVIDIA Carmel dual-core, basati su ARMv8.2 con una frequenza di clock massima di 2.26 GHz. Ogni core include 128KB di cache L1 dedicati alle istruzioni e 64KB dedicati a dati, più una cache L2 da 2MB condivisa tra i due core. I cluster della CPU condividono una cache L3 da 4 MB.

[2]Quando una rete neurale ha terminato il suo processo di addestramento e sta applicando le conoscenze acquisite per svolgere determinati task, si trova nella fase di inferenza

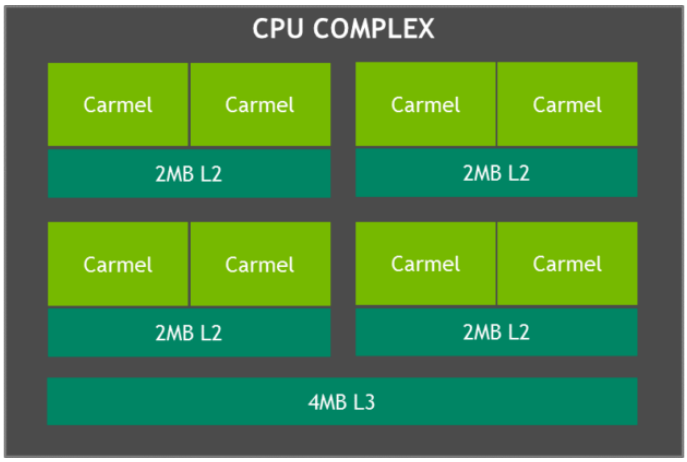


Figura 3.16: Struttura della CPU della NVIDIA Jetson AGX Xavier

Vision Accelerator La scheda NVIDIA Jetson AGX Xavier dispone di due motori Vision Accelerator, mostrati in Figura 3.17. Ciascuno integra un doppio processore VLIW (Very Long Instruction Word) a 7 vie che esegue algoritmi di computer vision come feature detection & matching, optical flow, e point cloud processing a bassa latenza e bassa potenza.

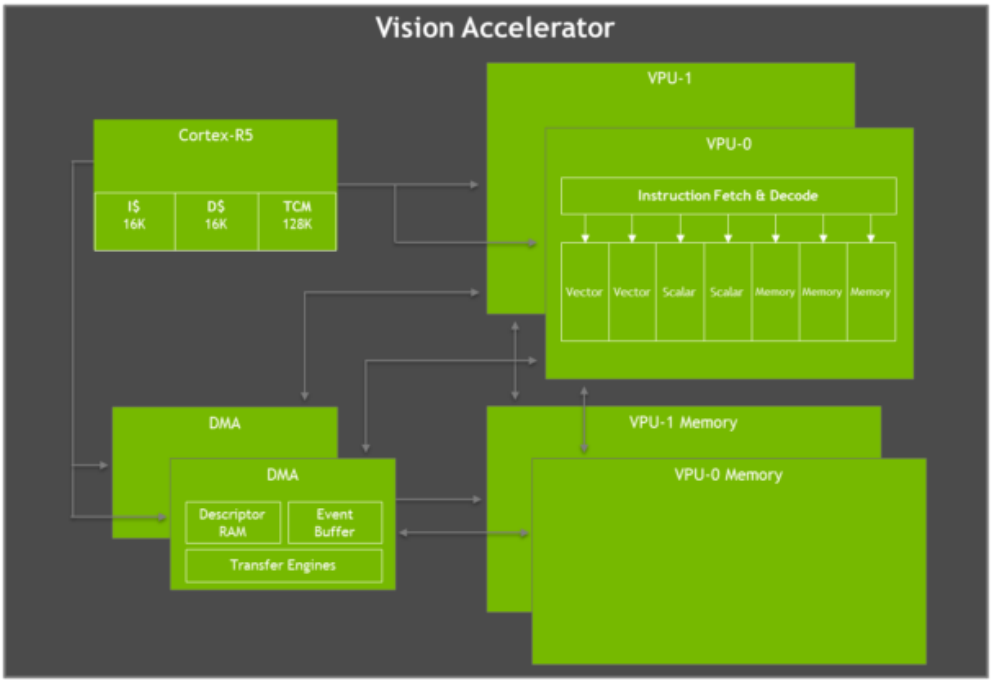


Figura 3.17: Struttura del Vision Accelerator

Ogni Vision Accelerator include un processore Cortex-R5 per comando e controllo. Inoltre, come descritto in precedenza, ogni unità possiede due VPU (Vector Processing Units) per l'elaborazione vettoriale (ciascuna con 192KB di memoria vettoriale su chip), e due unità DMA (Direct Memory Access) per lo smistamento dei dati. Le unità di elaborazione vettoriale a 7 vie contengono due slot per operazioni vettoriali, due per operazioni scalari e tre di memoria per contenere le istruzioni.

3.5.2 Interfacciamento con la Scheda

La scheda di sviluppo NVIDIA Jetson AGX Xavier esegue un sistema operativo Linux. È possibile accedere ed interfacciarsi con il sistema collegando un mouse e una tastiera alle porte USB e un monitor al connettore HDMI. Per poter essere collegati al sistema anche mentre il veicolo, con a bordo la scheda, è in movimento, si è proceduto ad installare un router, configurando il sistema come VNC Server, utilizzando l'omonimo programma. Grazie a questo accorgimento, è possibile collegarsi in remoto e controllare il sistema Linux della Xavier e di conseguenza tutto l'ambiente ROS, utilizzando un normale PC e collegandosi alla rete locale Wi-Fi del router montato a bordo del veicolo.
Una volta effettuato l'accesso tramite VNC Client, collegandosi all'indirizzo IP[3] della scheda, la NVIDIA Jetson AGX Xavier si presenta come una normale workstation, sulla quale è stato installato il sistema operativo Linux Ubuntu 18.04.

[3]L'indirizzo IP configurato per la scheda corrente è 192.168.8.145

3.6 Posizionamento dei Sistemi di Riferimento

In questa sezione sarà illustrato il metodo di posizionamento ed il codice utilizzato per descrivere il sistema di riferimento del veicolo e dei sensori montati a bordo dello stesso. Infatti è molto importante descrivere il posizionamento relativo di tali sistemi di riferimento, per poter elaborare correttamente tutti i dati. Ad esempio, la Visual Odometry viene ricavata in funzione della posizione della camera, che è spostata rispetto al centro del veicolo. Lo studio del posizionamento dei sistemi di riferimento e la loro descrizione all'interno di un file `.urdf` permette di tenere conto di tutti questi aspetti.

Il link `base_link` rappresenta il centro del veicolo e tutti i frame relativi ai sensori sono riferiti ad esso. Esistono infatti link "padre" e link "figlio". Questa definizione è molto importante per poter costruire correttamente il TF tree relativo alle trasformazioni di coordinate da un sistema all'altro. L'albero delle trasformazioni relativo al robot studiato in questa tesi viene mostrato nella seguente figura:

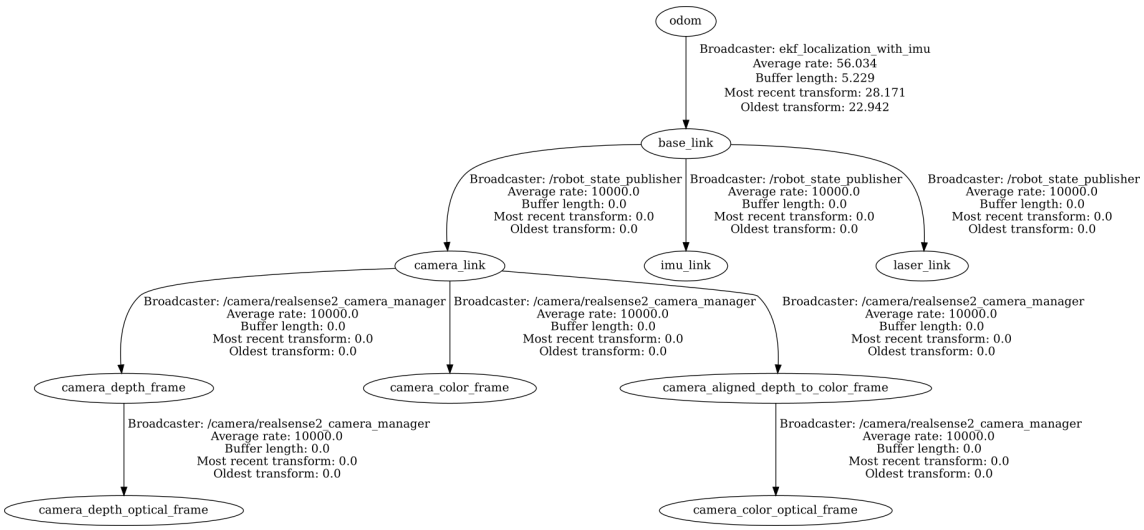


Figura 3.18: TF Tree del veicolo

Com'è apprezzabile dallo schema, la pubblicazione delle trasformazioni TF è realizzata da diversi nodi: il nodo `robot_state_publisher` si occupa di leggere la descrizione dei link che compongono il veicolo e di popolare il TF tree con tutti i frame di cui si conosce posizione ed orientamento a priori, cioè che restano fissi nel tempo. Questi frame sono:

- `base_link`: frame padre, a cui sono riferiti tutti gli altri frame del veicolo;
- `camera_link`: frame relativo alla camera: il nodo `camera_manager` parte da esso per pubblicare i frame relativi ad ogni ottica del dispositivo;
- `imu_link`: frame al quale si riferisce il sensore di misura inerziale;
- `laser_link`: frame al quale si riferisce il dispositivo LIDAR. La distanza rispetto ad un oggetto viene calcolata rispetto a questo frame e viene ricondotta al `base_link` grazie al TF tree.

Diverso invece è il discorso per il frame `odom`: esso rappresenta il frame pubblicato dal nodo `ekf_localization` che insieme ad esso, pubblica le informazioni di odometria all'interno del topic `/odom`. Questo frame varia la sua posizione nel tempo in base ai movimenti del veicolo ed è molto importante per la localizzazione.

Le trasformazioni TF relative al frame di odometria e quelli associati alle ottiche della camera vengono pubblicati automaticamente come spiegato precedentemente, mentre la struttura principale del veicolo deve essere descritta all'interno di un file `.urdf`, mostrato nell'Appendice B.4.

Capitolo 4

Test degli Algoritmi applicati sul Veicolo Reale

In questo capitolo saranno illustrati i test effettuati sul veicolo reale in scala 1:10 e saranno discussi i risultati ottenuti. Nella prima parte sarà mostrata in dettaglio la configurazione del sistema, con un focus sulle modalità di comunicazione tra i vari nodi.
Successivamente, sarà introdotto il metodo di acquisizione della mappa dell'ambiente, fondamentale per l'algoritmo di Adaptive Monte Carlo Localization che si occupa di localizzare il veicolo all'interno del terreno di prova e poter così realizzare la navigazione da un punto A verso un punto B.
Infine saranno descritti i test effettuati ed i risultati ottenuti. In particolare, sono stati condotti tre tipi di test, aggiungendo di volta in volta il numero di feature da utilizzare. In prima battuta si è verificata la bontà dell'algoritmo di planning e del controller, comandando al veicolo di spostarsi da un punto A verso un punto B, lungo un percorso in cui non erano presenti ostacoli. L'algoritmo di navigazione quindi, ha ricavato un percorso da seguire in base alla global costmap, valorizzata attraverso la mappa acquisita precedentemente. Il secondo test è consistito in un'ulteriore navigazione, con il veicolo che ha dovuto districarsi all'interno di uno spazio stretto. L'intento di questo test è di far arrivare il veicolo in prossimità di una collisione e far intervenire il Recovery Server, riportare il veicolo lontano da ostacoli e riprogrammare il percorso verso la destinazione finale. Durante la terza ed ultima prova si è inserito un ostacolo lungo il percorso dopo che il veicolo aveva già programmato il suo itinerario: si è stabilita una destinazione, il veicolo ha ottenuto il path da seguire e soltanto dopo si è inserito un ostacolo lungo il percorso. Il veicolo ha individuato l'oggetto e riprogrammato il suo piano per arrivare a destinazione.

4.1 Configurazione del Sistema Finale

Per il corretto funzionamento del sistema finale sono stati utilizzati due dispositivi: la NVIDIA Jetson AGX Xavier per l'acquisizione dati dai sensori e un PC con architettura x86 per l'elaborazione di tali dati e gli algoritmi di navigazione. Questa soluzione si è resa necessaria a causa dell'incompatibilità del pacchetto RTAB-Map con un architettura ARM e la conseguente impossibilità di utilizzare tale pacchetto a bordo della NVIDIA Jetson. Di seguito viene mostrato uno schema che riassume la struttura del sistema:

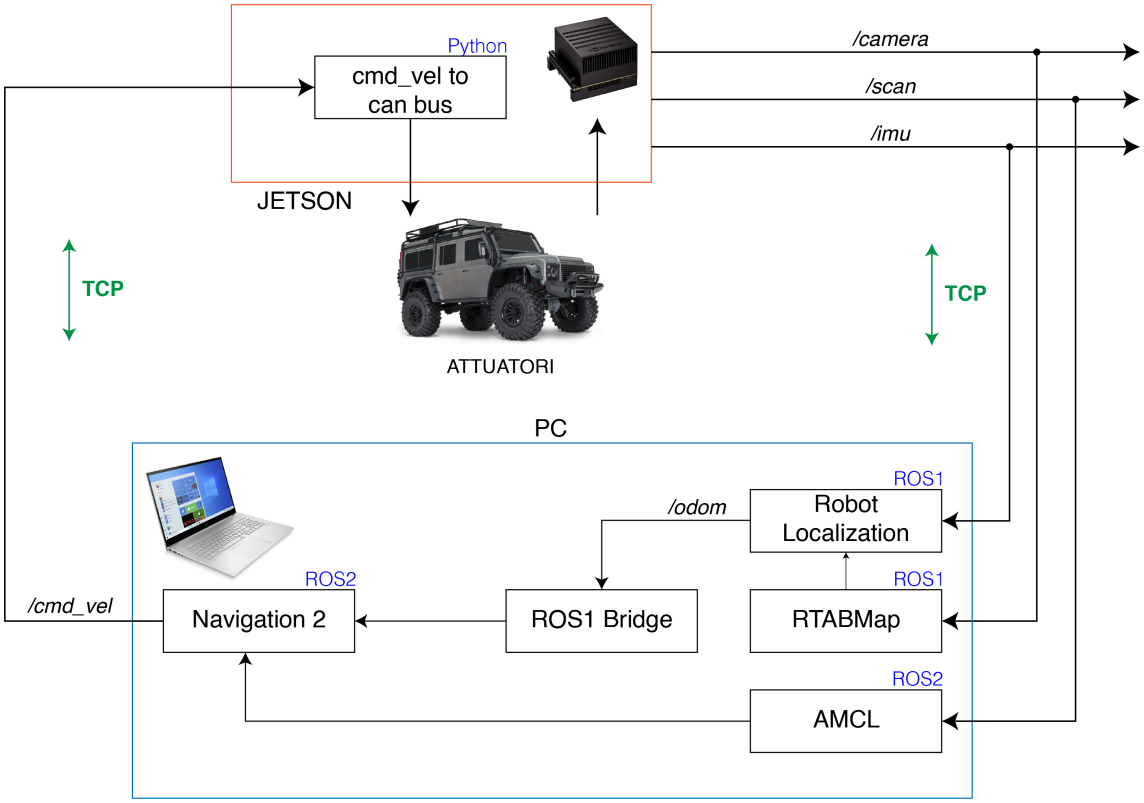


Figura 4.1: Schema della configurazione del sistema

La configurazione degli algoritmi di navigazione, così come quelli di Visual Odometry e Sensor Fusion, è la medesima rispetto a quella utilizzata in simulazione. I due sistemi dialogano tramite porta TCP, attraverso la rete locale creata dal router a bordo del veicolo. Il topic `cmd_vel`, che contiene i comandi in velocità da fornire al veicolo, viene letto dal nodo lanciato dallo script Python "cmd_vel to can bus", che si occupa di leggere il contenuto del topic, convertire le informazioni in un angolo di sterzata e un comando in velocità ed inviare il risultato tramite CAN-bus al controllore a basso livello degli attuatori del veicolo. Tale script è illustrato in dettaglio in Appendice B.5.
La scheda NVIDIA Jetson AGX Xavier acquisisce i dati provenienti dai sensori IMU, LIDAR e depth camera e attraverso appositi nodi ROS pubblica all'interno dei topic `/scan`, `/camera`, `/imu` tutte le acquisizioni. I topic sopra descritti vengono trasferiti al sistema ROS del PC tramite protocollo TCP e i nodi relativi a RTAB-Map, Robot Localization ed AMCL possono lavorare con le nuove rilevazioni.

4.2 Applicazione di un Algoritmo SLAM per l'Acquisizione di una Mappa

SLAM, acronimo di Simultaneous Localization And Mapping, è una tecnica che permette la localizzazione del veicolo all'interno di una mappa creata in tempo reale. Questo permette di costruire una mappa di un ambiente in partenza sconosciuto tramite l'utilizzo, in questo caso, di un sensore LIDAR. Il pacchetto ROS utilizzato per realizzare la mappa dell'ambiente tramite SLAM è `hector_mapping`. Esso riceve in input i dati provenienti dal sensore LIDAR e li sfrutta per ricavare in tempo reale una mappa dell'ambiente e per localizzare il veicolo all'interno di essa. In output, l'algoritmo pubblica i dati relativi alla mappa all'interno del topic `/map`. Tramite il nodo `map_server` è possibile salvare la mappa acquisita lanciando il seguente comando:

```
1 rosrun map_server map_saver
```

La mappa ottenuta tramite SLAM è mostrata nella seguente figura:

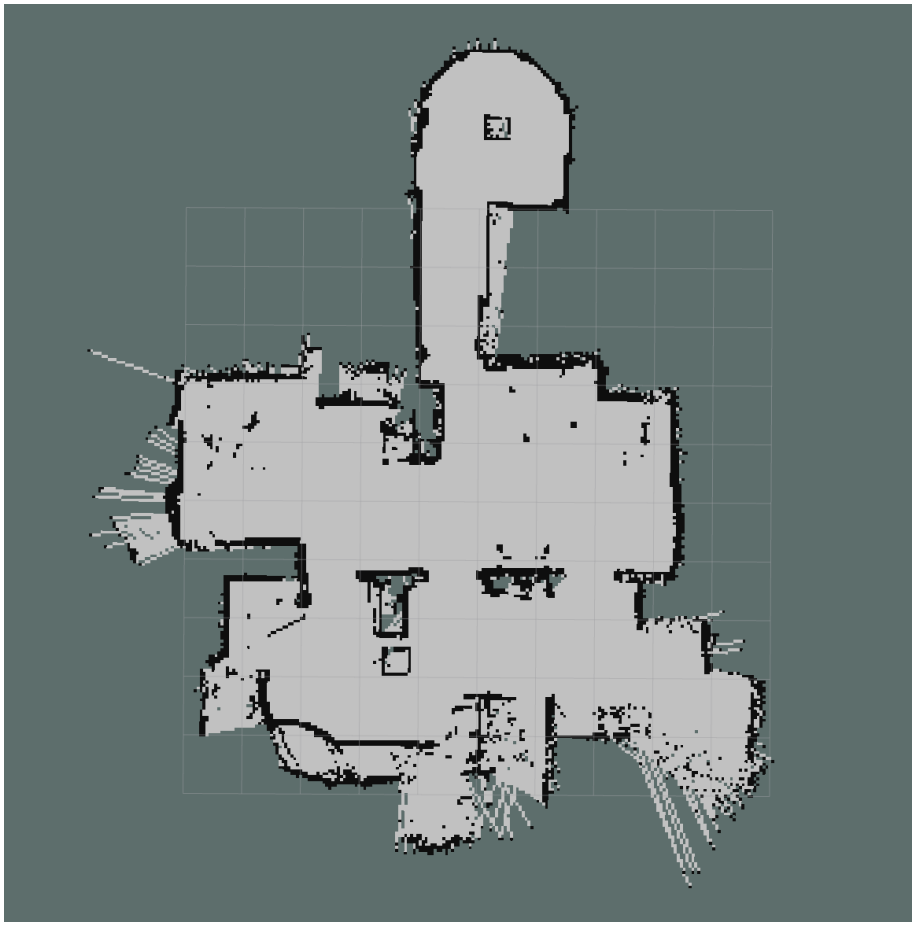


Figura 4.2: Mappa dell'ambiente ottenuta tramite SLAM

I pixel di colore nero rappresentano in questo caso le pareti e gli oggetti presenti nell'ambiente indoor in cui è stata fatta l'acquisizione, mentre i pixel grigi rappresentano lo spazio libero. A partire da questa mappa l'algoritmo di navigazione ricaverà la global costmap, che sarà parte integrante del processo di decisione del percorso da seguire.

4.3 Test Finali sul Veicolo Reale e Risultati Ottenuti

In questa sezione saranno illustrati tutti i test di navigazione svolti sul veicolo per verificare l'efficacia degli algoritmi implementati. In primo luogo è stata effettuata una verifica di efficacia del sistema, navigando da un punto A verso un punto B della mappa sopra descritta. Successivamente è stata alzata la difficoltà di navigazione per far intervenire il Recovery Server e testare il replanning del percorso.

4.3.1 Test di Navigazione da un Punto A verso un Punto B della Mappa

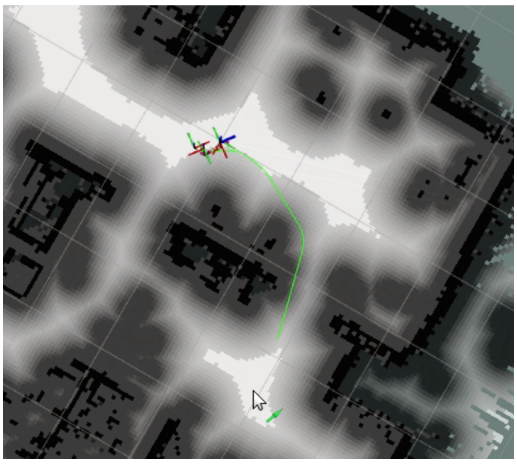
Il primo test effettuato è stato quello di navigare da un punto ad un altro della mappa, per verificare gli algoritmi ed eventualmente aggiustare parametri di navigazione, relativi ad esempio alla global costmap e alla local costmap. Nella figura seguente viene mostrato come, data la destinazione finale, il Planner Server riesce ad ottenere un percorso, basandosi sugli ostacoli presenti all'interno della mappa:

(a) Percorso elaborato dal planner server

(b) Path mostrato all'interno dell'ambiente reale

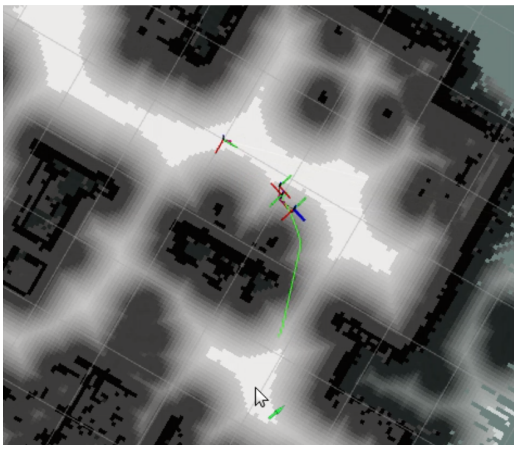
Figura 4.3: Percorso ottenuto dal Planner Server sfruttando le informazioni della mappa

Dopo aver elaborato il percorso il Controller Server, tramite l'algoritmo Regulated Pure Pursuit, inizia ad inviare comandi di velocità tramite il topic `cmd_vel` ed il veicolo inizia a muoversi. Di seguito vengono mostrate alcune istantanee del movimento del veicolo verso la destinazione:



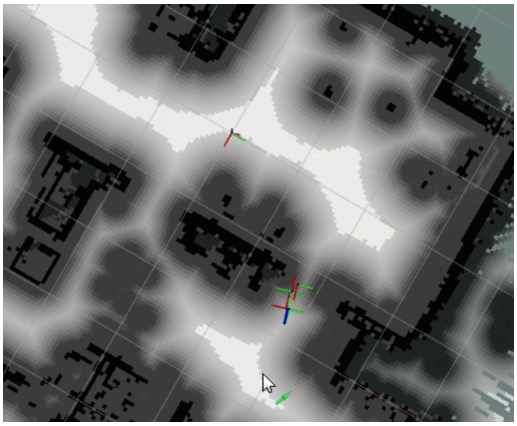
(a) Schermata RViz del test, prima parte

(b) Visuale sul veicolo, prima parte



(c) Schermata RViz del test, seconda parte

(d) Visuale sul veicolo, seconda parte



(e) Schermata RViz del test, parte finale

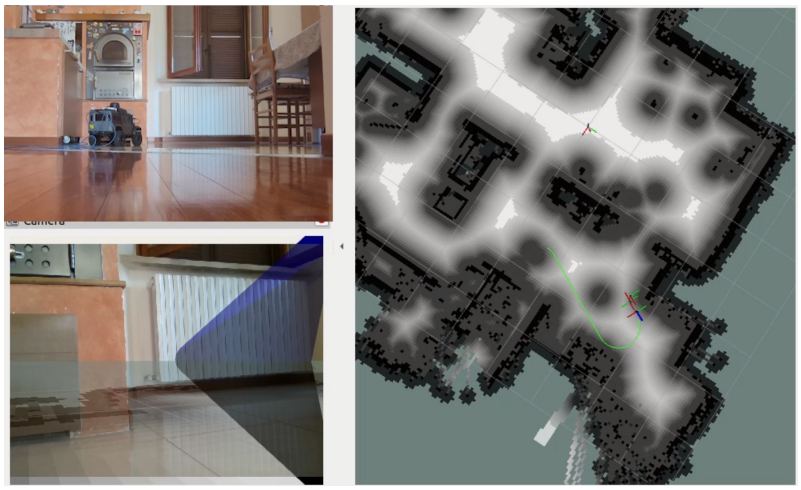
(f) Visuale sul veicolo, parte finale

Figura 4.4: Primo test effettuato sul veicolo reale

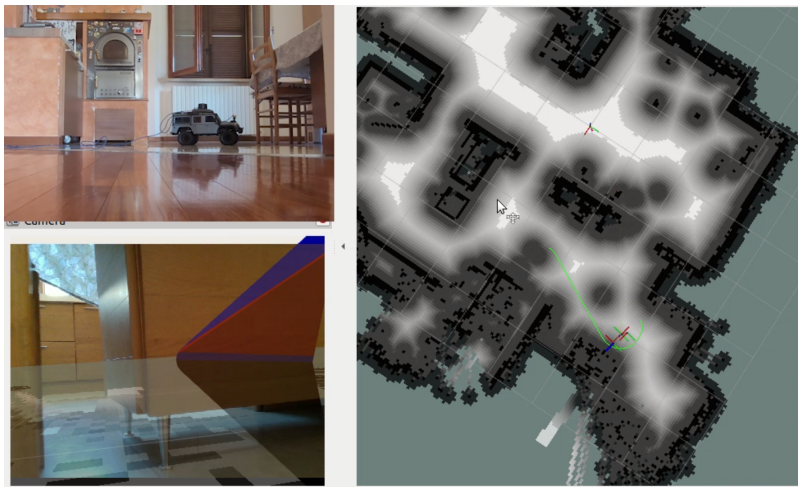
4.3.2 Test di Navigazione con Intervento del Recovery Server

La seconda prova che è stata svolta con il veicolo è la navigazione verso un punto della mappa, passando per un'area con difficoltà di manovra. Infatti, per riuscire a girarsi, il veicolo ha dovuto far intervenire il Recovery Server poiché è arrivato

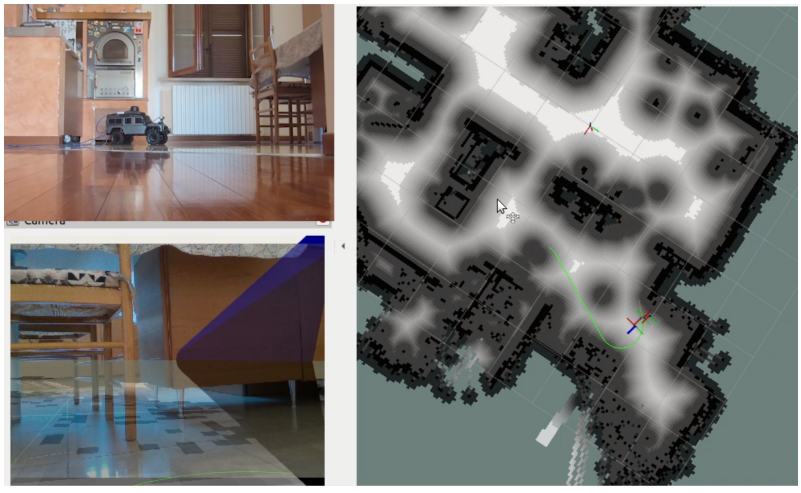
vicino ad una collisione. Successivamente il planner ha ricalcolato il percorso. Di seguito vengono mostrati i momenti salienti della prova:



(a) Parte iniziale del test



(b) Intervento del Recovery Server



(c) Ricalcolo del percorso

Figura 4.5: Test sul veicolo reale con intervento del Recovery Server

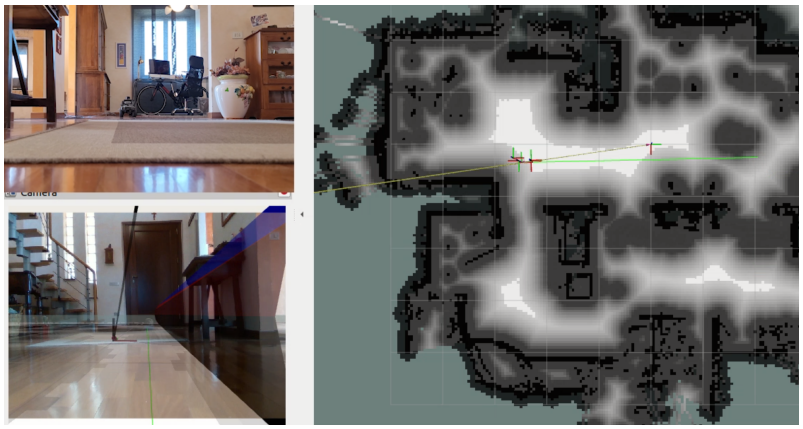
(d) Arrivo a destinazione

Figura 4.5: Test sul veicolo reale con intervento del Recovery Server

L'intervento del Recovery Server è molto importante al fine di allontanare il veicolo dall'ostacolo incontrato. Infatti, una volta arrivato vicino ad una collisione, non è più possibile per il planner elaborare un nuovo percorso. Il Recovery Server ha il compito di far eseguire al veicolo una manovra di retromarcia che gli permette di avere abbastanza spazio a disposizione davanti a sé in modo da consentire al Planner Server di elaborare una nuova traiettoria verso la destinazione.

4.3.3 Test di Navigazione con Obstacle Avoidance

L'ultima prova, che sfrutta le stesse feature testate in precedenza, si è svolta facendo calcolare un certo percorso al veicolo per arrivare a destinazione ed inserendo successivamente un ostacolo lungo tale percorso. Il veicolo rileva l'oggetto, fa intervenire il Recovery Server per allontanarsi da esso ed infine ricalcola il percorso per poter raggiungere la destinazione finale aggirandolo.



(a) Parte iniziale del test

Figura 4.6: Test sul veicolo reale con Obstacle Avoidance

(b) Intervento del Recovery Server

(c) Aggiramento dell'ostacolo

(d) Arrivo a destinazione

Figura 4.6: Test sul veicolo reale con Obstacle Avoidance

4.4 Considerazioni Finali e Sviluppi Futuri

I risultati mostrati nei precedenti paragrafi testimoniano come il veicolo riesca con successo a raggiungere le destinazioni assegnate. Con l'ausilio della RGB-D camera e del sensore LIDAR è possibile eseguire una scansione accurata dell'ambiente circostante e rilevare con precisione ostacoli presenti lungo il percorso, anche se questi non sono mostrati all'interno della mappa iniziale. Grazie alla feature di Recovery implementata ad hoc, il veicolo riesce ad allontanarsi dall'ostacolo e quindi rielaborare correttamente un nuovo percorso. La navigazione all'interno di un ambiente indoor, in cui ad esempio non è possibile utilizzare un sistema di posizionamento assoluto come il GPS, richiede l'utilizzo di algoritmi di odometria molto precisi. Tuttavia, tutti i sistemi di questo tipo sono affetti da un errore che inevitabilmente cresce con il tempo, pertanto è necessario l'utilizzo di un sistema di localizzazione che riesca a correggerli, senza ricorrere all'utilizzo di sistemi non utilizzabili in ambiente indoor. È questo il caso dell'Adaptive Monte Carlo Localization utilizzato in questa tesi, che sfruttando le informazioni di odometria, le scansioni LIDAR e la mappa riesce a localizzare con precisione il veicolo nell'ambiente permettendo una navigazione efficace.

A partire dai risultati ottenuti con questa tesi, si potrebbe sviluppare un lavoro di approfondimento sul Recovery Server ed elaborare un algoritmo di intervento più complesso. Inoltre, se si desiderasse applicare questo progetto ad ambienti outdoor, sarebbe possibile aggiungere al sistema un ricevitore GPS per riuscire ad ottenere una localizzazione precisa anche in aree molto estese e far collaborare con esso il sistema di Visual Odometry.
ROS rappresenta un framework molto flessibile, infatti si presta particolarmente all'integrazione di nuovi algoritmi su quelli già implementati, grazie alla sua struttura modulare e all'agevole comunicazione tra i vari nodi. Sarebbe possibile pertanto partire dal sistema già sviluppato ed aggiungere nuove feature senza dover intervenire pesantemente sul lavoro già svolto.

Il mondo della guida autonoma è in forte sviluppo oggi più che mai, grazie agli strumenti messi a disposizione dalla tecnologia ed al lavoro dei ricercatori che si stanno dedicando al progetto. Elon Musk, CEO della casa costruttrice Tesla, pioniera insieme a Google di questa tecnologia negli ultimi anni, ha affermato che entro la fine del 2021 le proprie auto saranno dotate del livello 5 di guida autonoma, il che significa che per guidare, l'auto non avrà bisogno della presenza a bordo del conducente. L'attuabilità effettiva di questo grado di autonomia però, passa soprattutto per i sistemi legislativi dei vari paesi, che molto difficilmente saranno pronti entro l'anno a legiferare in materia.
In circa un secolo di storia dunque, si è passati da un rudimentale veicolo radiocomandato attraverso un antenna montata sul tetto nel 1926, alla possibilità di mettere su strada veicoli completamente autonomi, capaci di gestire imprevisti lungo il percorso e di raggiungere la destinazione prefissata senza bisogno dell'intervento umano. Questo rappresenta motivo di grande ottimismo guardando verso il futuro, soprattutto in termini di sicurezza su strada, dove la maggior parte degli incidenti, oggi causati da errore umano, potranno essere evitati. Il fine principale dello sviluppo della guida autonoma infatti è, oltre che migliorare il comfort di guida, quello della

sicurezza e di diminuire di conseguenza il numero di incidenti e morti su strada. Basandosi sui dati ISTAT raccolti nel 2019 emerge come le cause principali di incidente siano distrazione, nervosismo, stress, calo d'attenzione, alterazione dello stato di coscienza. Un sistema di guida autonoma non può essere affetto da nessuno di questi disturbi, pertanto un miglioramento degli algoritmi, fino ad arrivare a sistemi di livello 5 affidabili, porterebbe un abbassamento drastico del numero di incidenti stradali ed un miglioramento complessivo della vita di tutti coloro che ogni giorno compiono un tragitto a bordo di un mezzo di trasporto.

Appendice A

A.1 Installazione e Setup dell'Ambiente ROS

A.1.1 ROS Noetic

Nel corrente paragrafo vengono illustrati l'installazione dell'ambiente ROS Noetic e il setup del `catkin_ws`. Il primo step è stato reperire tutte le risorse e le chiavi necessarie all'installazione, attraverso i seguenti comandi:

```
1 sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
    lsb_release -sc) main" > /etc/apt/sources.list.d/ros-
    latest.list'
2 curl -sSL 'http://keyserver.ubuntu.com/pks/lookup?op=get&
    search=0xC1CF6E31E6BADE8868B172B4F42ED6FBAB17C654' |
    sudo apt-key add -
```

Il passo successivo consiste nell'installazione vera e propria di ROS. Si è scelto l'installazione base dell'ambiente, poiché tutti i plugin sono già stati installati in precedenza con ROS2. Dopo aver eseguito un aggiornamento della lista dei Debian packages, è possibile lanciare il comando di installazione `apt install`

```
1 sudo apt update
2 sudo apt install ros-noetic-ros-base
3 source /opt/ros/noetic/setup.bash
```

L'ultima linea di comando listata sopra, è necessaria per eseguire il sourcing di ROS, e rendere tutti i comandi disponibili alla shell. É molto importante che, se è già presente un'altra versione di ROS installata, `∼/.bashrc` esegua solamente il sourcing del file `/setup.bash` della versione correntemente in uso. Infine si procede ad installare alcuni tools molto importanti per fare build di nuovi pacchetti:

```
1 sudo apt install python3-rosdep python3-rosinstall python3-
    rosinstall-generator python3-wstool build-essential
```

Setup del catkin workspace In primo luogo è necessario individuare la directory in cui si desidera creare il workspace. Nel caso corrente è stata scelta la directory Home "`/∼`". I comandi per creare ed eseguire il build del workspace sono mostrati di seguito:

```
1 mkdir -p ~/catkin_ws/src
2 cd ~/catkin_ws/
```

```
3         catkin_make
```

Al fine di collegare tale workspace con ROS Noetic, è necessario che l'ambiente `ROS_PACKAGE_PATH` includa anche la directory del workspace.

```
1         echo $ROS_PACKAGE_PATH /home/youruser/catkin_ws/src:/opt/
             ros/kinetic/share
```

All'interno del `catkin_ws` è possibile installare pacchetti compatibili con ROS. Nella sezione successiva sarà illustrata la procedura di installazione del pacchetto `ackermann_vehicle`, presente nella repo `gkouros/ackermann_vehicle` di GitHub.

A.1.2 ROS2 Foxy

L'ultima versione dell'ambiente ROS 2 - Foxy Fitzroy - non risulta compatibile con le versioni di Ubuntu Linux precedenti alla "20.04", pertanto l'installazione del framework ROS è stata eseguita su tale versione di Ubuntu.

In primo luogo, per poter installare ROS 2, è necessario aggiungere le repository apt di ROS 2 nel sistema. Tale operazione è stata realizzata eseguendo i comandi seguenti:

```
sudo apt update && sudo apt install curl gnupg2 lsb-release

sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/
    master/ros.key  -o /usr/share/keyrings/ros-archive-keyring.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/
    keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/
    ubuntu $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list
    .d/ros2.list > /dev/null
```

Una volta acquisite le repository, si procede all'installazione degli strumenti di sviluppo e degli strumenti di ROS:

```
sudo apt update && sudo apt install -y \
build-essential \
cmake \
git \
libbullet-dev \
python3-colcon-common-extensions \
python3-flake8 \
python3-pip \
python3-pytest-cov \
python3-rosdep \
python3-setuptools \
python3-vcstool \
wget

# install some pip packages needed for testing
python3 -m pip install -U \
argcomplete \
flake8-blind-except \
flake8-builtins \
flake8-class-newline \
flake8-comprehensions \
flake8-deprecated \
flake8-docstrings \
flake8-import-order \
flake8-quotes \
pytest-repeat \
pytest-rerunfailures \
pytest

# install Fast-RTPS dependencies
sudo apt install --no-install-recommends -y \
libasio-dev \
libtinyxml2-dev
# install Cyclone DDS dependencies
sudo apt install --no-install-recommends -y \
```

```
36 libcunit1-dev
```

Successivamente viene creato un workspace e vi vengono clonate tutte le repo all'interno:

```
1 mkdir -p ~/ros2_foxy/src
2 cd ~/ros2_foxy
3 wget https://raw.githubusercontent.com/ros2/ros2/foxy/ros2.repos
4 vcs import src < ros2.repos
```

Vengono installate le dipendenze utilizzando "rosdep":

```
1 sudo rosdep init
2 rosdep update
3 rosdep install --from-paths src --ignore-src --rosdistro foxy -y --
    skip-keys "console_bridge fastcdr fastrtps rti-connext-dds-5.3.1
     urdfdom_headers"
```

Infine, viene fatto un build di tutto il codice nel workspace: il workspace consiste in una directory in cui sono presenti tutti i packages di ROS. É necessario eseguire una build del workspace per rendere il codice grezzo al suo interno e i pacchetti di ROS utilizzabili dall'utente.

```
1 cd ~/ros2_foxy/
2 colcon build --symlink-install
```

Inoltre, è necessario eseguire il "sourcing" del file "local_setup.bash" per il setup dell'ambiente:

```
1 . ~/ros2_foxy/install/local_setup.bash
```

Per terminare la configurazione di ROS, è necessario che la shell di Linux abbia accesso ai comandi del framework. Il seguente comando si occupa di questo.

```
1 source /opt/ros/foxy/setup.bash
```

Per evitare di eseguire tale comando all'apertura di ogni terminale, è possibile aggiungere l'istruzione direttamente nello script di startup della shell:

```
1 echo "source /opt/ros/foxy/setup.bash" >> ~/.bashrc
```

A.2 Installazione del Software Gazebo

In questa sezione si descriverà il processo di installazione del software di simulazione Gazebo. L'ambiente ROS 2 Foxy Fitzroy è compatibile con Gazebo 11 o versioni successive. Si è pertanto optato per l'installazione di Gazebo 11.

I comandi da eseguire tramite terminale necessari all'installazione di Gazebo sono di seguito riportati:

```
# repo setup
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/
    ubuntu-stable `lsb_release -cs` main" > /etc/apt/sources.list.d/
    gazebo-stable.list'
cat /etc/apt/sources.list.d/gazebo-stable.list

# setup keys
wget https://packages.osrfoundation.org/gazebo.key -O - | sudo apt-
    key add -

# update debian database
sudo apt-get update

# install Gazebo 11
sudo apt-get install gazebo11

# check installation by launching Gazebo
gazebo
```

Una volta installato Gazebo, il passo successivo consiste nell'installazione dei pacchetti di ROS necessari a dialogare con Gazebo:

```
sudo apt install ros-foxy-gazebo-ros-pkgs
```

É possibile testare l'interfacciamento di Gazebo con ROS 2 con gli esempi forniti nei "Gazebo ROS packages". Una dimostrazione viene di seguito riportata:

- Aprire un nuovo terminale
- Eseguire il sourcing di ROS 2 (se l'istruzione è stata inserita nello startup della shell questa operazione non è necessaria);
- Se non presenti, installare i seguenti core tools:

```
        sudo apt install ros-foxy-ros-core ros-foxy-geometry2
```

- Caricare il "differential drive world" con Gazebo:

```
        gazebo --verbose /opt/ros/foxy/share/gazebo_plugins/
            worlds/gazebo_ros_diff_drive_demo.world
```

- In un altro terminale, è possibile eseguire un comando per far muovere il veicolo, pubblicando un'istruzione nel topic relativo, in questo caso "geometry_msgs/Twist":

```
        ros2 topic pub /demo/cmd_demo geometry_msgs/Twist '{
            linear: {x: 1.0}}' -1
```

Quando si termina Gazebo, potrebbe verificarsi che il processo ”gzserver” non termini e venga mostrato il seguente messaggio di errore all'avvio della successiva simulazione:

```
1 [Err] [Master.cc:95] EXCEPTION: Unable to start server[bind:  Address already in
      use].  There is probably another Gazebo process running.
```

Per risolvere il problema è necessario eseguire un ”kill” di tale processo con il seguente comando da shell:

```
1 killall -9 gzserver
```

A.3 Codice per l'Implementazione in Simulazione di un Sensore LIDAR

In questa appendice viene mostrato il codice utilizzato per implementare in simulazione un sensore LIDAR, necessario per il corretto funzionamento dell'algoritmo di Adaptive Monte Carlo Localization. Per effettuare questa operazione, è sufficiente richiamare il giusto plugin di Gazebo, che si occupa di simulare il sensore e di conseguenza tutte le sue uscite. Ciò che è necessario definire quindi, è un link che descrive il sensore, un giunto che colleghi il link al robot e il plugin di Gazebo associato al link. Di seguito viene mostrato il codice utilizzato:

```
<link name="base_scan">
        <inertial>
                <mass value="0.001"/>
                <origin rpy="0 0 0" xyz="0 0 0"/>
                <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.000001
                   " iyz="0" izz="0.0001"/>
        </inertial>
        <visual>
                <origin rpy="0 0 0" xyz="0.1 0 0.2"/>
                <geometry>
                        <box size="0.001 0.001 0.001"/>
                </geometry>
        </visual>
        <collision>
                <origin rpy="0 0 0" xyz="0.1 0 0.2"/>
                <geometry>
                        <box size=".001 .001 .001"/>
                </geometry>
        </collision>
</link>

<gazebo reference="base_scan">
        <material>Gazebo/FlatBlack</material>
        <sensor type="ray" name="lds_lfcd_sensor">
                <pose>0 0 0 0 0 0</pose>
                <update_rate>5</update_rate>
                <ray>
                        <scan>
                                <horizontal>
                                        <samples>360</samples>
                                        <resolution>1</resolution>
                                        <min_angle>0.0</min_angle>
                                        <max_angle>6.28319</
                                           max_angle>
                                </horizontal>
                        </scan>
                        <range>
                                <min>0.120</min>
                                <max>3.5</max>
                                <resolution>0.015</resolution>
                        </range>
                        <noise>
                                <type>gaussian</type>
                                <mean>0.0</mean>
                                <stddev>0.01</stddev>
```

```
44                          </noise>
45                  </ray>
46                  <plugin name="gazebo_ros_lds_lfcd_controller"
                       filename="libgazebo_ros_laser.so">
47                          <topicName>/scan</topicName>
48                          <frameName>base_scan</frameName>
49                  </plugin>
50          </sensor>
51  </gazebo>
```

A.4 Installazione ed Utilizzo del Pacchetto "ackermann_vehicle"

Dopo aver eseguito il setup di tutto l'ambiente ROS Noetic e del workspace, si è passato all'installazione di un veicolo con geometria ackermann. Tale veicolo è disponibile nella repo `gkouros/ackermann_vehicle` di GitHub. Per attuare il salvataggio e il building del pacchetto nel catkin workspace sono stati eseguiti i seguenti comandi da shell, posizionandosi nella cartella principale del ws "∼`/catkin_ws`:

```
1        cd ~/catkin_ws/src
2        git clone https://github.com/gkouros/ackermann_vehicle.git
3        cd ~/catkin_ws
4        catkin_make
```

Il pacchetto `ackermann_vehicle` contiene due principali directory. Il file `.launch` eseguito nella sezione corrente carica il veicolo, il programma Gazebo per la visualizzazione grafica e il controllore. Il comando necessario per lanciare tale file è il seguente:

```
1        roslaunch ackermann_vehicle_gazebo ackermann_vehicle.launch
```

Lanciando tale comando sono stati riscontrati vari errori, dovuti principalmente a pacchetti ROS e Python mancanti, che rendevano ineseguibile la simulazione. I pacchetti ROS installati al fine di risolvere tali mancanze sono elencati di seguito:

- ros-noetic-xacro
- ros-noetic-stereo-image-proc
- ros-noetic-ackermann-msgs
- ros-noetic-ackermann-steering-controller
- ros-noetic-std-msgs
- ros-noetic-controller-manager-msgs
- ros-noetic-joint-state-controller
- ros-noetic-joint-state-publisher
- ros-noetic-joint-trajectory-controller
- ros-noetic-effort-controllers

Mentre i pacchetti Python necessari al funzionamento del pacchetto sono:

- pip3
- python-math
- numpy
- python3-rospy

- rospkg
- yaml

Un ulteriore errore riscontrato durante l'esecuzione del pacchetto, è nello script python relativo al controllore. Lanciando il file `ackermann_vehicle.launch` viene mostrato il seguente errore:

```
Traceback (most recent call last):
  File "/home/damiano/catkin_ws/src/ackermann_vehicle/ackermann_vehicle_gazebo/nodes/ackermann_controller.py", line 120, in <module>
    import rospy
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/__init__.py", line 49, in <module>
    from .client import spin, myargv, init_node, \
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/client.py", line 60, in <module>
    import rospy.impl.init
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/impl/init.py", line 54, in <module>
    from .tcpros import init_tcpros
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/impl/tcpros.py", line 45, in <module>
    import rospy.impl.tcpros_service
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/impl/tcpros_service.py", line 54, in <module>
    from rospy.impl.tcpros_base import TCPROSTransport, TCPROSTransportProtocol, \
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/impl/tcpros_base.py", line 160
    (e_errno, msg, *_) = e.args
                   ^
SyntaxError: invalid syntax
```

Figura A.1: Errore di sintassi dovuto allo script `ackermann_controller.py`

Questo errore è dovuto al fatto che il pacchetto `rospy` richiede python3 per poter funzionare correttamente. Nello script `ackermann_controller.py` viene invece richiamata la versione 2 di python e il pacchetto rospy porta all'errore sopra mostrato. Per risolvere è necessario modificare la prima riga dello script come segue:

```
1 #!/usr/bin/env python3
```

Lo script python sopra mostrato è di fondamentale importanza per permettere il movimento del veicolo. Esso lancia un nodo, chiamato `ackermann_controller`, legge i messaggi pubblicati all'interno del topic `ackermann_cmd`, che contiene informazioni di velocità, accelerazione, sterzata ecc., ed attua tali comandi, muovendo il veicolo. Nell'appendice B.2 sarà mostrato lo script python utilizzato per convertire i comandi inviati da Navigation2 attraverso il topic `cmd_vel` in messaggi compatibili con il nodo `ackermann_controller`.

A.5 Installazione del Pacchetto "ros1_bridge"

Il pacchetto `ros1_bridge` è presente anche all'interno delle repo apt di Linux, pertanto è possibile reperirlo ed installarlo direttamente con il comando dedicato `apt install`. Facendo così però, il sistema installa un software precompilato, non scaricando il codice sorgente e non lasciando spazio ad alcun tipo di personalizzazione, che specialmente con questo pacchetto è necessaria per avere la piena compatibilità con tutto il sistema.

Infatti, nel caso sia necessario inserire dei file `yaml` con mappature personalizzate nell'installazione, non è consigliabile utilizzare `apt install`, poichè eseguendo un'installazione di questo tipo, il bridge non può conoscere la struttura dei messaggi e dei pacchetti personalizzati che si vuole inserire. Pertanto, per poter funzionare al meglio, è necessario compilare il pacchetto da codice sorgente, dopo aver eseguito le modifiche desiderate, come ad esempio la modifica delle mapping rules e l'aggiornamento di tutte le dipendenze.

In primo luogo è necessario eseguire il download di tutti i file per l'installazione del pacchetto `ros1_bridge`. Si clonano quindi tutti i file necessari all'interno della directory `src` del workspace. Ipotizzando che il root del workspace sia ∼`/dev_ws` il codice è il seguente:

```
1        cd ~/dev_ws/src
2        git clone https://github.com/ros2/ros1_bridge.git
```

Successivamente, se si possiedono tipi di messaggi personalizzati non standard, si aggiungono o modificano i file necessari, poi si lancia un build di tutti i pacchetti eccetto `ros1_bridge`, poiché effettuando il source dell'ambiente ROS1 potrebbero essere aggiunte librerie indesiderate al path:

```
1        colcon build --symlink-install --packages-skip ros1_bridge
```

Si procede poi ad eseguire il sourcing della versione corrente di ROS1, in questo caso ROS Noetic:

```
1        source /opt/ros/noetic/setup.bash
```

Se sono presenti mappature personalizzate è necessario inserire l'apposito file `yaml` e modificare `package.xml` e `CMakeLists.txt` come spiegato nel paragrafo precedente.

In ultimo, si esegue solamente il build del pacchetto `ros1_bridge`:

```
1        colcon build --symlink-install --packages-select
            ros1_bridge --cmake-force-configure
```

Per verificare che tutti i pacchetti e tutti i messaggi siano stati mappati correttamente, è possibile lanciare il seguente comando:

```
1        ros2 run ros1_bridge dynamic_bridge --print-pairs
```

A.5.1 Esempio di Comunicazione tra ROS1 e ROS2 tramite Bridge

In questa prima esecuzione del `ros1_bridge` si è caricato un veicolo ackermann tramite ROS Noetic e sono stati pubblicati messaggi di controllo dal message publisher di ROS2, direttamente nel topic di controllo del veicolo in ROS1, tramite il bridge.

Per caricare tutti gli strumenti sono necessari quattro terminali differenti nei quali vengono eseguiti i seguenti comandi:

- Terminale 1: caricamento del roscore di ROS Noetic

```
source /opt/ros/noetic/setup.bash
roscore
```

- Terminale 2: caricamento di ROS Noetic e del modello `ackermann_vehicle`.

```
source /opt/ros/noetic/setup.bash
roslaunch ackermann_vehicle_gazebo ackermann_vehicle.
    launch
```

Il veicolo caricato è mostrato nella seguente figura:

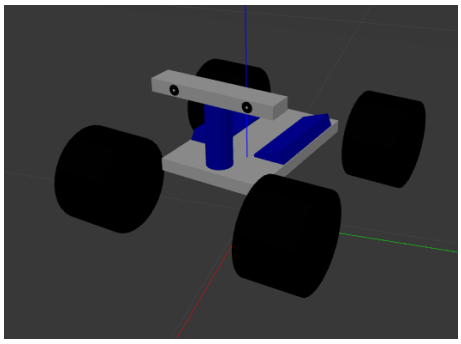


Figura A.2: Modello ackermann_vehicle mostrato in Gazebo

- Terminale 3: caricamento di entrambi gli ambienti ROS, del workspace in cui è situato il bridge ed esecuzione del bridge

```
source /opt/ros/noetic/setup.bash
source /opt/ros/foxy/setup.bash
. ~/dev_ws/install/setup.bash

ros2 run ros1_bridge dynamic_bridge
```

- Terminale 4: sourcing dell'ambiente ROS2 e pubblicazione di messaggi nel topic di comando `/ackermann_vehicle/ackermann_cmd`

```
ros2 topic pub /ackermann_vehicle/ackermann_cmd
    ackermann_msgs/AckermannDriveStamped "{drive: {
    steering_angle: 0.5}}"
```

Questo messaggio ha l'effetto di sterzare le ruote anteriori del veicolo di un angolo θ pari a $\frac{pi}{6} rad$. Nella figura seguente viene mostrato il veicolo prima e dopo l'invio del comando di sterzata:

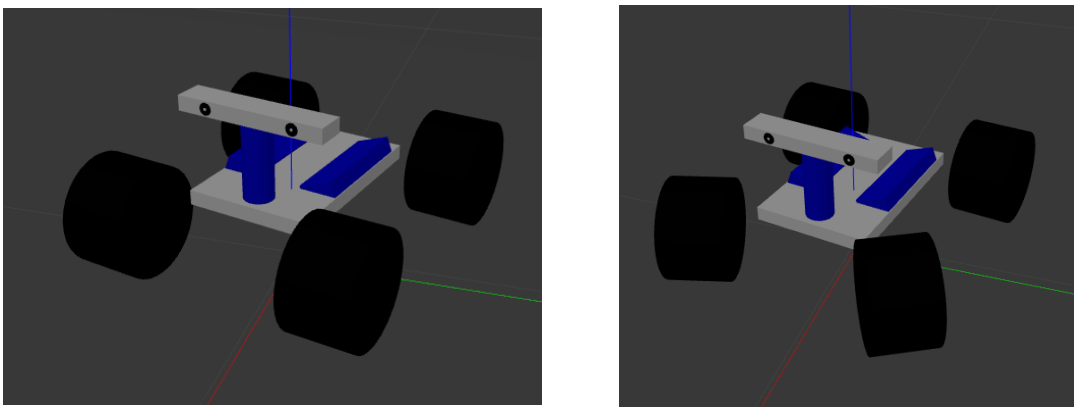


Figura A.3: Visualizzazione grafica del modello sterzante ackermann_vehicle

Lo scopo di questo esempio è semplicemente mostrare come il bridge riesca a trasmettere i messaggi pubblicati da un topic di ROS2 a un altro topic di ROS1. Ovviamente quest'esempio specifico, ovvero l'invio di comandi al controller del veicolo, sarebbe stato realizzabile anche senza l'aiuto del bridge, pubblicando i messaggi all'interno del topic `/ackermann_cmd` direttamente tramite l'apposito comando di ROS1 `rostopic pub`. Tuttavia, l'obiettivo finale di questa tesi è controllare un veicolo lanciato in ROS1 tramite il software Navigation2 installato in ROS2, pertanto è stata portata una prima dimostrazione di funzionamento del bridge.

A.6 Installazione del Pacchetto RTAB-Map

Il pacchetto RTAB-Map è stato installato seguendo le istruzioni presenti nella relativa repo GitHub[1]. Di seguito sono mostrati i passi principali.

Il pacchetto è stato installato eseguendo un operazione di "build from source", all'interno del workspace di ROS Noetic `catkin_ws` situato nella cartella "Home". Per poter procedere all'installazione è necessario che la versione di PCL installata sia >= 1.7, che corrisponde alla versione installata di default con ROS Noetic.

Nel terminale, se non sono presenti all'interno del file `.bashrc` è necessario lanciare i seguenti comandi:

```
source /opt/ros/noetic/setup.bash
source ~/catkin_ws/devel/setup.bash
```

Successivamente si procede ad installare tutte le dipendenze necessarie al pacchetto `rtabmap_ros`. Il metodo più veloce per acquisire tali dipendenze è lanciare i seguenti comandi:

```
sudo apt install ros-noetic-rtabmap ros-noetic-rtabmap-ros
sudo apt remove ros-noetic-rtabmap ros-noetic-rtabmap-ros
```

É possibile installare dipendenze opzionali, a seconda delle necessità di progetto, consultabili nel sito del pacchetto. Per il lavoro corrente sono state installate la dipendenza `libpointmatcher`, raccomandata se si andranno ad utilizzare dispositivi LIDAR e la dipendenza `g2o`. Le istruzioni di installazione del pacchetto `libpointmatcher` sono consultabili nel sito[2], mentre per installare il pacchetto `g2o` è possibile lanciare il seguente comando:

```
sudo apt install ros-noetic-libg2o
```

Il passo successivo è installare le standalone libraries di RTAB-Map. É molto importante non clonare tali librerie all'interno del `catkin_ws`:

```
cd ~
git clone https://github.com/introlab/rtabmap.git rtabmap
cd rtabmap/build
cmake ..
make
sudo make install
```

Infine si procede ad installare il ros-pkg RTAB-Map nella cartella `/src` del workspace:

```
cd ~/catkin_ws
git clone https://github.com/introlab/rtabmap_ros.git src/
    rtabmap_ros
catkin_make
```

Usare il comando `catkin_make -j1` se la compilazione richiede più RAM di quella disponibile.

[1] link al sito: `https://github.com/introlab/rtabmap_ros`

[2] link a libpointmatcher: `https://github.com/ethz-asl/libpointmatcher#quick-start`

Appendice B

B.1 Modifica del Backup Plugin

Il ”backup plugin” originale fornito all'interno del pacchetto Navigation2, consiste nello spostamento del veicolo di una distanza prefissata, al fine di posizionarsi in un punto più conveniente per poter elaborare un percorso ed arrivare a destinazione. Nel caso si rilevi una collisione imminente, il processo di backup viene arrestato.

Il principio su cui si basa il recovery desiderato in questo elaborato è leggermente diverso: se si rileva una collisione, ed è quindi necessario l'intervento del recovery server, interviene il recovery plugin, che fa indietreggiare il veicolo di una certa distanza al fine di allontanarlo dall'ostacolo, per poi poter rielaborare un nuovo percorso. L'ipotesi fatta è che la traiettoria del veicolo, essendo sempre in avanti, fornisce spazio di manovra libero dietro di esso. Pertanto, nel caso si incontri un ostacolo di fronte, viene messa in atto una manovra di retromarcia e viene ignorata la collisione imminente con oggetti di fronte al veicolo. La struttura di un Recovery Plugin è rappresentata da una funzione onConfigure() per la configurazione del nodo, da una funzione onRun(), che contiene il codice per far partire l'azione e da una funzione onCycleUpdate() che viene eseguita durante lo svolgimento del task. Di seguito viene mostrato il codice in linguaggio C++ presente all'interno delle funzioni onRun() e onCycleUpdate(), mentre il codice di configurazione è standard e non è stato pertanto modificato:

```
1  Status BackUp::onRun(const std::shared_ptr<const BackUpAction::Goal
       > command)
2  {
3          if (command->target.y != 0.0 || command->target.z != 0.0) {
4                  RCLCPP_INFO(
5                  node_->get_logger(), "Backing up in Y and Z not
                       supported, "
6                  "will only move in X.");
7          }
8
9          command_x_ = -0.3;
10         command_speed_ = command->speed;
11
12         if (!nav2_util::getCurrentPose(
13         initial_pose_, *tf_, global_frame_, robot_base_frame_,
14         transform_tolerance_))
15         {
```

```
16                  RCLCPP_ERROR(node_->get_logger(), "Initial robot
                        pose is not available.");
17                  return Status::FAILED;
18          }
19
20          return Status::SUCCEEDED;
21  }
22
23
24  Status BackUp::onCycleUpdate()
25  {
26          geometry_msgs::msg::PoseStamped current_pose;
27          if (!nav2_util::getCurrentPose(
28          current_pose, *tf_, global_frame_, robot_base_frame_,
29          transform_tolerance_))
30          {
31                  RCLCPP_ERROR(node_->get_logger(), "Current robot
                        pose is not available.");
32                  return Status::FAILED;
33          }
34
35          double diff_x = initial_pose_.pose.position.x -
                current_pose.pose.position.x;
36          double diff_y = initial_pose_.pose.position.y -
                current_pose.pose.position.y;
37          double distance = sqrt(diff_x * diff_x + diff_y * diff_y);
38
39          feedback_->distance_traveled = distance;
40          action_server_->publish_feedback(feedback_);
41
42          if (distance >= abs(command_x_)) {
43                  stopRobot();
44                  return Status::SUCCEEDED;
45          }
46
47          auto cmd_vel = std::make_unique<geometry_msgs::msg::Twist
                >();
48          cmd_vel->linear.y = 0.0;
49          cmd_vel->angular.z = 0.0;
50          command_x_ < 0 ? cmd_vel->linear.x = -command_speed_ :
                cmd_vel->linear.x = command_speed_;
51
52          geometry_msgs::msg::Pose2D pose2d;
53          pose2d.x = current_pose.pose.position.x;
54          pose2d.y = current_pose.pose.position.y;
55          pose2d.theta = tf2::getYaw(current_pose.pose.orientation);
56
57          vel_pub_->publish(std::move(cmd_vel));
58
59          return Status::RUNNING;
60  }
```

Dopo aver modificato il codice, è stato aggiornato anche il behavior tree. Di default, il behavior tree utilizzato da Navigation2, invoca l'algoritmo spin nel ramo del recovery. Non essendo un veicolo di tipo differenziale, questo algoritmo è stato sostituito dal backup recovery modificato:

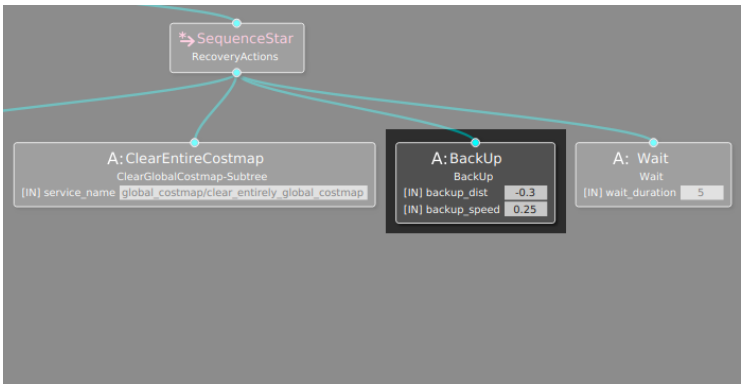


Figura B.1: Nodo del Behavior Tree sostituito

Com'è possibile vedere in figura, i parametri di distanza e di velocità possono essere forniti direttamente dal behavior tree, senza il bisogno di modificare ogni volta il codice dell'algoritmo.

B.2 Script Python per il Controllo a Basso Livello del Veicolo Ackermann

Lo studio del modello sterzante Ackermann è stato di fondamentale importanza per poter convertire i comandi provenienti da Navigation2 in input compatibili con il modello sterzante. Infatti tali comandi, pubblicati all'interno del topic `cmd_vel`, sono del tipo `geometry_msgs/msg/Twist` e contengono comandi di velocità lineare e angolare lungo i 3 assi cartesiani. L'obiettivo è quello di convertire tali messaggi in un comando di velocità lineare e uno di angolo di sterzata, introducendo le formule studiate nel Paragrafo 1.5.1, relativo al modello sterzante Ackermann. Di seguito viene mostrato il codice utilizzato per implementare lo script Python, che si occupa di creare un nodo capace di leggere i messaggi pubblicati nel topic `cmd_vel`, rielaborarli e pubblicare a sua volta messaggi di tipo `AckermannDriveStamped` contenenti informazioni di velocità ed angolo di sterzata all'interno del topic `ackermann_cmd`:

```
1  import rospy, math
2  from geometry_msgs.msg import Twist
3  from ackermann_msgs.msg import AckermannDriveStamped
4  from ackermann_msgs.msg import AckermannDrive
5
6
7  def convert_trans_rot_vel_to_steering_angle(v, omega, wheelbase):
8          if omega == 0 or v == 0:
9                  return 0
10
11         radius = v / omega
12         return math.atan(wheelbase / radius)
13
14
15 def cmd_callback(data):
16         global wheelbase
17         global ackermann_cmd_topic
18         global frame_id
19         global pub
20         global message_type
21
22         if message_type == 'ackermann_drive':
23                 v = data.linear.x
24                 steering = convert_trans_rot_vel_to_steering_angle(
                       v, data.angular.z, wheelbase)
25
26                 msg = AckermannDrive()
27                 msg.steering_angle = steering
28                 msg.speed = v
29
30                 pub.publish(msg)
31
32         else:
33                 v = data.linear.x
34                 steering = convert_trans_rot_vel_to_steering_angle(
                       v, data.angular.z, wheelbase)
35
36                 msg = AckermannDriveStamped()
37                 msg.header.stamp = rospy.Time.now()
38                 msg.header.frame_id = frame_id
```

```
39                  msg.drive.steering_angle = steering
40                  msg.drive.speed = v
41
42                  pub.publish(msg)
43
44  def main():
45          rospy.init_node('cmd_vel_to_ackermann_drive')
46
47          twist_cmd_topic = rospy.get_param('~twist_cmd_topic', '/
                cmd_vel')
48          ackermann_cmd_topic = rospy.get_param('~ackermann_cmd_topic
                ', '/ackermann_cmd')
49          wheelbase = rospy.get_param('~wheelbase', 0.335)
50          frame_id = rospy.get_param('~frame_id', 'odom')
51          message_type = rospy.get_param('~message_type', '
                ackermann_drive_stamped')
52
53          rospy.Subscriber(twist_cmd_topic, Twist, cmd_callback,
                queue_size=1)
54          if message_type == 'ackermann_drive':
55                  pub = rospy.Publisher(ackermann_cmd_topic,
                        AckermannDrive, queue_size=1)
56          else:
57                  pub = rospy.Publisher(ackermann_cmd_topic,
                        AckermannDriveStamped, queue_size=1)
58
59          rospy.loginfo("Node 'cmd_vel_to_ackermann_drive' started.\
                nListening to %s, publishing to %s. Frame id: %s,
                wheelbase: %f", "/cmd_vel", ackermann_cmd_topic,
                frame_id, wheelbase)
60
61          rospy.spin()
62
63
64  if __name__ == '__main__':
65          try:
66                  main()
67
68  except rospy.ROSInterruptException:
69          pass
```

Come si può apprezzare dal codice, la funzione principale dello script è convert_trans_rot_vel_to_steering_angle(), che prende in input un messaggio letto dal topic `cmd_vel` e ricava da esso un angolo di sterzata, che restituisce al chiamante. In questo modo il valore di sterzata viene salvato all'interno della variabile "steering", che insieme a "v" viene utilizzata per costruire il nuovo messaggio di tipo "AckermannDriveStamped" da pubblicare all'interno del topic `ackermann_cmd`.

Tutti i comandi relativi alla pubblicazione e lettura di messaggi vengono gestiti grazie a delle funzioni presenti all'interno della libreria "rospy", fornita con il pacchetto ROS.

B.3 Codice per il Setup dell'Algoritmo di Visual Odometry

In questa appendice sarà mostrato il codice necessario al setup del pacchetto RTAB-Map, realizzato utilizzando un file `.launch`, all'interno del quale sono presenti anche tutte le informazioni per lanciare l'intera simulazione. I nodi lanciati per il funzionamento di RTAB-Map sono:

- rgbd_sync;
- rgbd_odometry;
- rtabmap_ros.

Di seguito viene mostrato il codice utilizzato per lanciare il primo nodo:

```
<node pkg="nodelet" type="nodelet" name="rgbd_sync" args="
    standalone rtabmap_ros/rgbd_sync" output="screen">
        <remap from="rgb/image"        to="/camera/color/image_raw"
            />
        <remap from="depth/image"      to="/camera/
            aligned_depth_to_color/image_raw"/>
        <remap from="rgb/camera_info"  to="/camera/color/
            camera_info"/>
        <remap from="rgbd_image"       to="rgbd_image"/>

        <param name="approx_sync"       value="false"/>
</node>
```

Questo nodo è necessario quando informazioni provengono da sensori diversi, per assicurarsi che esse siano sincronizzate tra loro prima di essere elaborate da RTAB-Map. In questo caso specifico, il parametro `approx_sync` è settato al valore "false", infatti le informazioni provengono tutte dalla stessa camera, pertanto risultano già sincronizzate. L'uso di questo nodo quindi, ha soltanto lo scopo di verificare la sincronizzazione dei dati in input.

```
<node pkg="rtabmap_ros" type="rgbd_odometry" name="rgbd_odometry"
    output="screen">
        <param name="subscribe_rgbd" type="bool"   value="true"/>
        <param name="subscribe_rgb"  type="bool"   value="false"/>
        <param name="frame_id"       type="string" value="
            camera_link"/>
        <param name="odom_frame_id" type="string" value="odom"/>
        <param name="publish_tf"  type="bool"   value="false"/>

        <remap from="rgbd_image" to="rgbd_image"/>

        <param name="Odom/Strategy"            type="string" value=
            "0"/>
        <param name="Odom/ResetCountdown"      type="string" value=
            "1"/>
        <param name="OdomF2M/BundleAdjustment" type="string" value=
            "1"/> <!-- should be 0 for multi-cameras -->
        <param name="Vis/EstimationType"      type="string" value="
            1"/> <!-- should be 0 for multi-cameras-->
```

```
14          <param name="Vis/FeatureType"          type="string" value="
               6"/>
15          <param name="Vis/CorGuessWinSize"      type="string" value="
               0"/>
16          <param name="Vis/CorNNType"            type="string" value="
               3"/>
17          <param name="Vis/MaxDepth"             type="string" value="
               4.0"/>
18          <param name="Vis/MinInliers"           type="string" value="
               20"/>
19          <param name="Vis/InlierDistance"       type="string" value="
               0.1"/>
20          <param name="OdomF2M/MaxSize" type="string" value="3000"/>
21          <param name="Odom/FillInfoData"         type="string" value=
               "true"/>
22  </node>
```

Il nodo `rgbd_odometry` rappresenta il cuore dell'algoritmo di Visual Odometry e ricava, leggendo i dati in input dai topic della camera, i valori di odometria che saranno forniti al sistema, attraverso il topic `/odom`. Tale topic sarà utilizzato dall'algoritmo di Sensor Fusion per ottenere un'odometria ancora migliore.

```
1   <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="
       screen" args="--delete_db_on_start">
2           <param name="frame_id"         type="string" value="
               camera_link"/>
3           <param name="subscribe_depth" type="bool"   value="false"/>
4           <param name="subscribe_rgb"  type="bool"   value="false"/>
5           <param name="subscribe_rgbd"  type="bool"   value="true"/>
6           <param name="publish_tf"  type="bool"   value="false"/>
7
8           <remap from="rgbd_image" to="rgbd_image"/>
9
10          <param name="queue_size"  type="int"  value="10"/>
11          <param name="approx_sync" type="bool" value="false"/>
12
13          <!-- RTAB-Map's parameters -->
14          <param name="RGBD/AngularUpdate"         type="string" value
               ="0.01"/>
15          <param name="RGBD/LinearUpdate"          type="string" value
               ="0.01"/>
16          <param name="RGBD/OptimizeFromGraphEnd" type="string" value
               ="false"/>
17  </node>
```

Il nodo appena presentato rappresenta la base dell'algoritmo, grazie al quale tutta l'infrastruttura della Visual Odometry riesce a funzionare.

B.4 Codice XML per la Definizione dei Sistemi di Riferimento a Bordo del Veicolo

Di seguito viene mostrato il codice utilizzato per la descrizione dei sistemi di riferimento associati al veicolo ed a tutti i sensori montati su di esso. Nella descrizione sono riportate tutte le misure del robot e le distanze relative tra i dispositivi. `traxxas.urdf.xacro`, come tutti i file di questo tipo, è scritto in linguaggio XML:

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

  <!-- includes -->

    <!-- Degree-to-radian conversions -->
    <xacro:property name="degrees_45" value="0.785398163"/>
    <xacro:property name="degrees_90" value="1.57079633"/>

    <!-- Camera position wrt base_link -->
    <xacro:property name="camera_offset_x" value="0.22"/>
    <xacro:property name="camera_offset_z" value="0.07"/>

    <!-- Base link height -->
    <xacro:property name="base_link_height" value="0.15"/>

    <!-- Laser position wrt base link -->
    <xacro:property name="laser_offset_z" value="0.18"/>


    <link name="base_link">
      <visual>
        <geometry>
          <box size="0.53 .2 .28"/>
        </geometry>
      </visual>
    </link>

    <link name="camera_link">
      <visual>
        <geometry>
          <box size="0.025 0.09 0.023"/>
        </geometry>
      </visual>
    </link>

    <link name="laser_link">
      <visual>
        <geometry>
          <cylinder length="0.045" radius="0.035"/>
        </geometry>
      </visual>
    </link>

    <link name="imu_link">
      <visual>
        <geometry>
          <box size="0.02 0.02 0.001"/>
        </geometry>
      </visual>
```

```
50      </link>
51
52      <joint name="base_link_to_camera_link" type="fixed">
53        <parent link="base_link"/>
54        <child link="camera_link"/>
55        <origin xyz="0.22 0 .225"/>
56      </joint>
57
58      <joint name="base_link_to_imu_link" type="fixed">
59        <parent link="base_link"/>
60        <child link="imu_link"/>
61        <origin xyz="-0.1 0.05 .31"/>
62      </joint>
63
64      <joint name="base_link_to_laser_link" type="fixed">
65        <parent link="base_link"/>
66        <child link="laser_link"/>
67        <origin rpy="0 0 3.14" xyz="0 0 0.31"/>
68      </joint>
69  </robot>
```

B.5 Script Python per il Controllo a Basso Livello del Veicolo in Scala 1:10

Per poter convertire i comandi inviati dal Controller di Navigation2 attraverso il topic `/cmd_vel`, è stato necessario elaborare uno script Python che leggesse i messaggi pubblicati in tale topic, li convertisse in comandi di velocità ed angolo di sterzata e li inviasse al controllore degli attuatori del veicolo tramite protocollo CAN-bus.

```
import rclpy
import math
import can
import time

from rclpy.node import Node

from geometry_msgs.msg import Twist

bus = can.Bus(interface='socketcan',
channel='can0',
receive_own_messages=False)
wheelbase = 0.316
cmdAcc = 0
cmdStr = 0
flag = False


def convert_trans_rot_vel_to_steering_angle(v, omega, wheelbase):
  if omega == 0 or v == 0:
    return 0

  radius = v / omega
  return -math.atan(wheelbase / radius) * 180 / math.pi


def set_backwards():
  print("--------setBackwards--------")
  message = can.Message(arbitration_id=123, is_extended_id=True,
                        data=[int(cmdAcc) + 45, int(cmdStr) + 45, 0
                           x00, 0x00, 0x00, 0x00, 0x00, 0x00])
  bus.send(message, timeout=0.01)
  time.sleep(0.1)
  message = can.Message(arbitration_id=123, is_extended_id=True,
                        data=[int(0) + 45, int(cmdStr) + 45, 0x00,
                           0x00, 0x00, 0x00, 0x00, 0x00])
  bus.send(message, timeout=0.01)
  time.sleep(0.1)
  message = can.Message(arbitration_id=123, is_extended_id=True,
                        data=[int(cmdAcc) + 45, int(cmdStr) + 45, 0
                           x00, 0x00, 0x00, 0x00, 0x00, 0x00])
  bus.send(message, timeout=0.01)
  time.sleep(0.1)


class MinimalSubscriber(Node):

  def __init__(self):
    super().__init__('minimal_subscriber')
```

```
47      self.subscription = self.create_subscription(
48        Twist,
49        'cmd_vel',
50        self.listener_callback,
51        10)
52      self.subscription  # prevent unused variable warning
53
54
55    def listener_callback(self, msg):
56      self.get_logger().info('I heard: "%s"' % msg.linear.x)
57      global flag
58      cmdAcc = msg.linear.x * 40
59      if cmdAcc < 0 & flag is False:
60        flag = True
61        setBackwards()
62      elif flag is True & cmdAcc >= 0:
63        flag = False
64
65      if cmdAcc > 15:
66        cmdAcc = 15
67      if cmdAcc < -15:
68        cmdAcc = -15
69
70      cmdStr = convert_trans_rot_vel_to_steering_angle(msg.linear.x,
           msg.angular.z, wheelbase)
71      if cmdStr > 45:
72        cmdStr = 45
73      elif cmdStr < -45:
74        cmdStr = -45
75      # send a message
76      print("------------------ " + str(cmdAcc) + " " + str(cmdStr))
77      message = can.Message(arbitration_id=123, is_extended_id=True,
78                            data=[int(cmdAcc) + 45, int(cmdStr) + 45,
                                  0x00, 0x00, 0x00, 0x00, 0x00, 0x00])
79      bus.send(message, timeout=0.01)
80      time.sleep(0.01)
81
82
83  def main(args=None):
84    rclpy.init(args=args)
85
86    minimal_subscriber = MinimalSubscriber()
87
88    rclpy.spin(minimal_subscriber)
89
90    # Destroy the node explicitly
91    # (optional - otherwise it will be done automatically
92    # when the garbage collector destroys the node object)
93    minimal_subscriber.destroy_node()
94    rclpy.shutdown()
95  if __name__ == '__main__':
96    main()
```

In primo luogo viene inizializzato il nodo ROS, che diventa un subscriber del topic `/cmd_vel`. Il callback si occupa di ottenere un comando in velocità ed uno in sterzata compatibili con le specifiche del veicolo, ad esempio `cmdStr` non può avere un valore assoluto maggiore di 45 gradi, e successivamente i comandi vengono inviati

attraverso CAN-bus al controllore degli attuatori. É stata aggiunta anche la funzione `set_backwards()` che è necessaria per far passare il veicolo alla retromarcia, quando richiesto, mentre la funzione `convert_trans_rot_to_steering_angle` è utilizzata per convertire l'input in velocità lineare e angolare in un angolo di sterzata.

Bibliografia

[1] automobile.it. *Self driving cars, il futuro della guida è l'autonomia*. 2021. URL: https://www.automobile.it/magazine/mobilita-sostenibile/self-driving-cars-1677.

[2] Linda Capecci. *Guida autonoma, un secolo di storia.* 2020. URL: https://www.lautomobile.aci.it/articoli/2020/08/06/guida-autonoma-un-secolo-di-storia.html.

[3] Intel Corporation. *Intel Realsense D400 Series Datasheet*. 2019. URL: https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/Intel-RealSense-D400-Series-Datasheet.pdf.

[4] R. Craig Coulter. *Implementation of the Pure Pursuit Path Tracking Algorithm*. A cura di Carnegie Mellon. 1992. URL: http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/MRIS/coulter_r_craig_1992_1.pdf.

[5] Gianluca Covini. *Angolo di Ackermann: storia e funzione*. 2020. URL: https://www.autotecnica.org/angolo-di-ackermann-storia-e-funzione/.

[6] Amanda Dattalo. *Introduction to ROS*. 2018. URL: http://wiki.ros.org/ROS/Introduction.

[7] Dmitri Dolgov et al. *Practical Search Techniques in Path Planning for Autonomous Driving*. 2007. URL: https://ai.stanford.edu/~ddolgov/papers/dolgov_gpp_stair08.pdf.

[8] Felipe Espinosa et al. *Odometry and Laser Scanner Fusion Based on a Discrete Extended Kalman Filter for Robotic Platooning Guidance*. 2011. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3231507/#app1.

[9] Open Source Robotics Foundation. *Bridge communication between ROS 1 and ROS 2*. 2016. URL: https://github.com/ros2/ros1_bridge.

[10] Open Source Robotics Foundation. *Gazebo Documentation*. 2020. URL: http://gazebosim.org/.

[11] Open Source Robotics Foundation. *ROS2 Design*. 2020. URL: http://design.ros2.org/.

[12] Dustin Franklin. *NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics*. 2018. URL: https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/.

[13] Bosch Sensortec GmbH. *BNO055 Intelligent 9-axis absolute orientation sensor*. Ver. 1.7. 2020. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf.

[14] Bo Hu e He Huang. *Visual Odometry Implementation and Accuracy Evaluation Based on Real-time Appearance-based Mapping*. 2020. URL: https://myukk.org/SM2017/sm_pdf/SM2256.pdf.

[15] Mathieu Labbe. *Real-Time Appearance-Based Mapping*. 2021. URL: http://introlab.github.io/rtabmap/.

[16] Wunderkammer Laboratory. *Ackermann Vehicle Package*. 2013. URL: https://github.com/jbpassot/ackermann_vehicle.

[17] Steve Macenski. *SMAC Planner*. 2021. URL: https://github.com/ros-planning/navigation2/tree/main/nav2_smac_planner.

[18] Steven Macenski et al. «The Marathon 2: A Navigation System». In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020.

[19] Tom Moore. *Robot Localization Package*. 2021. URL: http://wiki.ros.org/robot_localization.

[20] Tom Moore. *Robot Localization Wiki*. 2016. URL: http://docs.ros.org/en/melodic/api/robot_localization/html/index.html.

[21] OpenRobotics. *ROS2 Documentation*. 2021. URL: https://docs.ros.org/en/foxy/index.html.

[22] Rosaria. *Incidenti stradali, quali sono le cause più frequenti*. 2020. URL: http://motori.quotidiano.net/comefare/incidenti-stradali-quali-sono-le-cause-piu-frequenti.htm.

[23] Alvise-Marco Seno. *Guida autonoma, un sogno partito quasi cent'anni fa*. 2017. URL: https://ruoteclassiche.quattroruote.it/guida-autonoma-un-sogno-partito-quasi-centanni-fa/.

[24] Ltd. Shanghai Slamtec Co. *RPLIDAR A2, Introduction and Datasheet*. 2016. URL: https://www.generationrobots.com/media/robopeak_2d_lidar_brief_en_A2M4.pdf.

[25] Shrijit Singh Steve Macenski. *Nav2 Regulated Pure Pursuit Controller*. 2021. URL: https://github.com/ros-planning/navigation2/tree/main/nav2_regulated_pure_pursuit_controller.

Ringraziamenti

Vorrei innanzitutto ringraziare il Professor Gianluca Ippoliti, relatore di questa tesi di laurea, per la sua disponibilità, cordialità e competenza, dimostrate non soltanto in questi ultimi mesi conclusivi, ma durante tutto il percorso di laurea. Inoltre, desidero ringraziarlo per avermi proposto quest'opportunità di tirocinio in cui ho potuto accrescere le mie conoscenze e competenze, all'interno un settore, quello dell'automotive, di cui sono appassionato.

Colgo altresì l'occasione per ringraziare il mio correlatore Gianluca Toscano che, insieme all'Ing. Alessandro Serrapica, ha fornito sempre un pronto supporto durante l'attività di tirocinio.