Abstract

L'elaborato si occupa di sviluppare e simulare un controllore per robot mobile terrestre mediante ROS 2. In particolare, dopo aver selezionato un robot rappresentante il modello dell'uniciclo verranno spiegati i fondamenti basilari per la comprensione di ROS 2 e sarà scelta una legge di controllo che permetta la navigazione da un punto nello spazio ad un punto target con un assetto specifico. Invece che cercare unicamente di portare nel minor tempo possibile il robot nella posizione verrà avvalorata la qualità del percorso attraverso delle traiettorie efficienti e stabili. In seguito sarà sviluppato il controllore tramite ROS 2. Esso è composto da due unità fondamentali di cui una si occupa della pianificazione dei punti target e la seconda di implementare la legge di controllo. Come dice lo stesso nome, il pianificatore si occuperà di scegliere un punto nello spazio e l'assetto finale del robot mentre il controllore garantirà il raggiungimento della posizione modificando la velocità angolare rispetto all'asse z del robot in funzione dei risultati ottenuti dalla legge di controllo. Inoltre per garantire stabilità durante la fase di arrivo, il robot decellererà maggiormente all'avvicinamento della posizione target. Questo controllore risolve il cosidetto *parking problem* in quanto permette il posizionamento del robot nell'intorno di un punto con un preciso orientamento nello spazio. L'elaborato è inoltre oggetto di sviluppo in quanto dalla legge di controllo iniziale, che permette lo spostamento tra due punti, si potrebbe raggiungere la definizione di una traiettoria composta da più punti, in cui il robot riesca a muoversi con facilità e senza interruzioni nel percorso prestabilito.

Indice

Elenco delle figure

Introduzione

Motivazione

Sin dall'avvento dei primi calcolatori moderni il concetto di robot come macchina artificiale in grado di svolgere compiti assegnategli dall'uomo, per supervisione diretta o tramite il perseguimento di regole precise specificate a priori, è da molto oggetto di ricerche e di studi che hanno portato a solide basi sulle quali poter implementare sistemi particolarmente evoluti e con capacità articolate. La rapida evoluzione dei microprocessori ed il conseguente aumento della potenza di calcolo ha reso possibile la concretizzazione di un un'idea già nota alla società e già teorizzata nella letteratura matematica ed ingegneristica. Per garantire la progettazione di robot efficienti e per poter concretizzarli entrano in gioco materie fondamentali come l'analisi matematica per la realizzazione di modelli o la fisica che ne permette la messa in pratica nel mondo reale. L'informatica, invece, è la componente fondamentale dal punto di vista computazionale, in quanto si occupa della trasmissione dei dati, del loro immagazzinamento e della loro elaborazione dando forma alle cosiddette *intelligenze artificiali*. Nella pratica, però, questi robot che sempre più frequentemente tendono anche alla guida autonoma necessitano di un'entità che li guidi e che gli permetta di muoversi nello spazio in maniera uniforme e controllata. Per questo molte applicazioni robotiche hanno bisogno di componenti che implementino leggi di controllo efficienti e sicure.

Contributi

Il suddetto elaborato si propone di fornire un controllore con un'architettura modulare attraverso la piattaforma ROS 2 per un robot mobile terrestre. La legge di controllo garantirà il raggiungimento di una posizione nello spazio in un determinato assetto attraverso una traiettoria intuitiva e stabile.

Organizzazione

Il primo capitolo si impone di spiegare i principali componenti di ROS 2 e di Gazebo. In particolare è diviso in tre sezioni. La prima si occupa di spiegare i principi base di comunicazione nelle applicazioni sviluppate con ROS 2. La seconda di illustrare brevemente un ambiente di sviluppo di ROS 2, descrivendo l'organizzazione di workspace e package e file di lancio, mentre l'ultima fornisce le informazioni basilari per poter comprendere ed utilizzare Gazebo. Il secondo capitolo descrive il modello matematico preso come riferimento. La prima parte descrive il modello matematico dell'uniciclo e conclude con la definizione della legge di controllo utilizzata per l'elaborato. Infine il terzo e ultimo capitolo, inizialmente spiega dettagliatamente come effettivamente il controllore è stato implementato su ROS 2, mentre la seconda parte mostra i risultati ottenuti e li analizza. Inoltre viene brevemente descritto il robot utilizzato come modello nell'elaborato.

Capitolo 1

Strumenti di sviluppo: ROS 2 e Gazebo

ROS 2 è un meta-sistema operativo open-source per robot, erede ed estensione di ROS (Robot Operating System). ROS 2 mette a disposizione tutti i servizi tipici di un sistema operativo come l'astrazione dei componenti fisici, il controllo di componenti hardware di basso livello, la comunicazione tra processi di un sistema e la gestione dei pacchetti software coinvolti. Oltre ad avere le più comuni funzioni di un sistema operativo tradizionale, mette a disposizione una serie di librerie per gestire l'intero processo di vita di un robot, dalla progettazione fisica alle sperimentazioni in simulazioni virtuali e non. Come verrà approfondito nel capitolo, queste librerie gestiscono la compilazione di *package*, la creazione di *workspace* e di altre operazioni di "alto livello" che ne conferiscono caratteristiche tipiche di un framework. I software nel panorama di ROS 2 possono essere divisi in tre gruppi principali:

1. linguaggi e tools, indipendenti da ROS 2, utilizzati per il supporto su qualsiasi piattaforma
2. librerie per lo svliluppo lato client come per esempio rclcpp e rclpyp per lo sviluppo in Python e C++ di applicazioni robotiche con due noti linguaggi di programmazione all'interno della comunità degli sviluppatori
3. una serie di packages application-related che usano una o più librerie client di ROS 2 utili per gestire lo sviluppo di progetti.

1.1 Comunicazione in ROS 2

Il concetto di comunicazione è fondamentale per un'applicazione di ROS 2. Infatti ogni sistema robotico è composto da molti componenti sia hardware, sia software che lavorano in maniera

indipendente tra loro, ma che allo stesso tempo necessitano di una forte coesione e per questo devono comunicare tra loro in maniera efficiente e con un basso dispendio di energie. L'unità elementare di un componente di ROS 2 e del suo predecessore ROS 1 è il nodo ovvero un processo che svolge un task preciso all'interno di un progetto di ROS 2. Un robot è composto da un insieme di nodi che comunicano tra di loro scambiandosi messaggi dalle funzionalità più disparate, dal messaggio per indicare la velocità del robot, al messaggio sulla sua posizione. Dal principio di singola responsabilità si intuisce che comporre il sistema attraverso molti nodi, invece che scrivere un unico applicativo monolitico porta a molti vantaggi:

1. Un errore fatale di un nodo non porterebbe al completo malfunzionamento del sistema come in un unico blocco monolitico, ma solamente all'arresto di uno specifico componente che potrebbe essere o superfluo o comunque non decisivo per il sistema generale.

2. Il codice viene considerevolmente alleggerito permettendo una più semplice gestione di bug ed errori semantici.

Ogni nodo è quindi un modulo di un sistema molto complesso che comunica con gli altri nodi per permettere una comunicazione efficace e soprattutto efficiente. Questi nodi fanno parte di un package e sono associati ad un file eseguibile presente nel file system di ROS 2, installato sul proprio calcolatore.
Per comunicare, hanno bisogno di un canale di comunicazione che possa unire ogni nodo del sistema ai restanti secondo una logica basata su un argomento di conversazione, o per meglio dire, un aspetto del progetto. Nel seguente elaborato, si darà attenzione in particolare al concetto di *topic*, ovvero il principale bus di comunicazione per lo scambio di messaggi tra nodi. In generale ogni nodo non può instaurare una comunicazione stabile con i restanti $n-1$ nodi del sistema, in quanto se da un lato questo aumenterebbe l'efficienza del sistema, dall'altro causerebbe un'ingente perdita a livello di risorse all'aumentare di nodi e file di configurazione. Per questo è stato deciso di implementare un meccanismo che permettesse una comunicazione efficace ma a bassi costi e molto rapida. Da qui è nata l'idea di utilizzare il paradigma di *publish/subscribe* o modello ad eventi come rappresentato in figura 1.2. Il meccanismo publish/subscribe consiste sull'idea di base che un nodo produce informazioni e che queste informazioni possano risultare utili, se non fondamentali, ad altri molteplici nodi del sistema. Per questo non crea un canale di comunicazione specifico tra le entità del sistema, ma implementa un sistema di comunicazione di basso livello molto eterogeneo con un forte disaccopiamento tra le entità in gioco. Quindi in generale i nodi non sono consapevoli con quali altri nodi stanno comunicando ma pubblicano messaggi predefiniti su topic all'interno del sistema e si iscivono ad altri topic da cui leggono i messaggi, permettendo diverse tipologie di

scambio come per esempio punto a punto, uno a molti, molti ad uno e più comunemente molti a molti. Ogni topic però è fortemente tipizzato in quanto può pubblicare o ricevere solamente messaggi della tipologia del topic. Questo per evitare confusione tra le diverse parti in gioco e per dividere la comunicazione fortemente a livello semantico. Diviene quindi importante in ROS 2 la possibilità di definire messaggi predefiniti con un significato preciso ed estremamente specifici per il topic di competenza. Un messaggio è sostanzialmente una struttura dati contenente più campi definiti anch'essi da un nome. Essi possono essere di diverse tipologie, da valori stringa o booleani a in particolare valori numerici, con un'ampia gamma di scelta comprensiva dei tipi di variabili dei moderni linguaggi di programmazione. Definito il messaggio (.msg) il DDS (spiegato più precisamente nella sezione successiva) si occuperà della conversione dal valore utilizzato dal compilatore o interprete del linguaggio di programmazione utilizzato al tipo di dato trasportato e riconosciuto dal DDS come mostrato in figura 1.3. Ogni topic è identificato con una stringa, che deve essere univoca all'interno della rete di processi. ROS 2 fornisce una serie di messaggi preimpostati, ma molti progetti, al fine di aumentare la compatibilità con altri pacchetti software, implementano topic con messaggi specifici e stringhe identificative comuni. Più precisamente se un nodo produce informazioni utili ad altri nodi crea un *publisher*, un'entità in grado di scrivere in un topic e pubblica il messaggio specifico per quel topic, mentre viceversa se necessita di informazioni su un determinato topic, crea un *subscriber* e aspetta la pubblicazione. Come accennto sopra, i messaggi sono asincroni e dipendono dal risultato di altri nodi che pubblicano in quel topic, quindi per mantenere la massima efficienza dal nodo il *subscriber* deve garantire la lettura di messaggi asincroni. Da qui deriva la necessità di definire all'interno della dichiarazione del subscriber una funzione di callback asincrona che venga eseguita ogni qual volta un nodo pubblica sul topic a cui si è iscritti. Ricevuta la notifica di pubblicazione su un topic il nodo interromperà il normale flusso di operazione ed eseguirà una funzione detta di *callback*, la quale avrà come parametro nella firma il messaggio del topic. La funzione di callback iscritta ad un topic solitamente è incaricata di analizzare il messaggio e trarne informazioni significative o di formulare una risposta da pubblicare su altri topic con determinati messaggi significativi. Ogni messaggio può essere composto da molteplici campi, obbligatoriamente inseriti singolarmente per riga e può anche contenere array di messaggi. I nomi scelti sono solitamente significativi e rendono più semplice l'invio e la ricezione dei dati.

```
float64 a
float64 b
float64 c
```

Nell'esempio soprastante è rappresentato un messaggio contenente tre float a 64 bit di

precisione(un'eventuale posizione nello spazio cartesiano (x, y, z)). Inoltre è estendibile in quanto può contenere al suo interno altri messaggi più specifici secondo una gerarchia ad albero. Per mantenere una divisione ordinata tra le risorse i messaggi vengono gestiti e mantenuti in package a sè stanti. E' quindi normale avere diversi package per lo svliuppo dei nodi ed altri package unicamente per la definizione di messaggi. Nell'immagine sottostante 1.1 si può notare come lo strumento *rqt-graph* permetta la visione dei nodi e dei topic attivi nel sistema durante la sua attivazione.

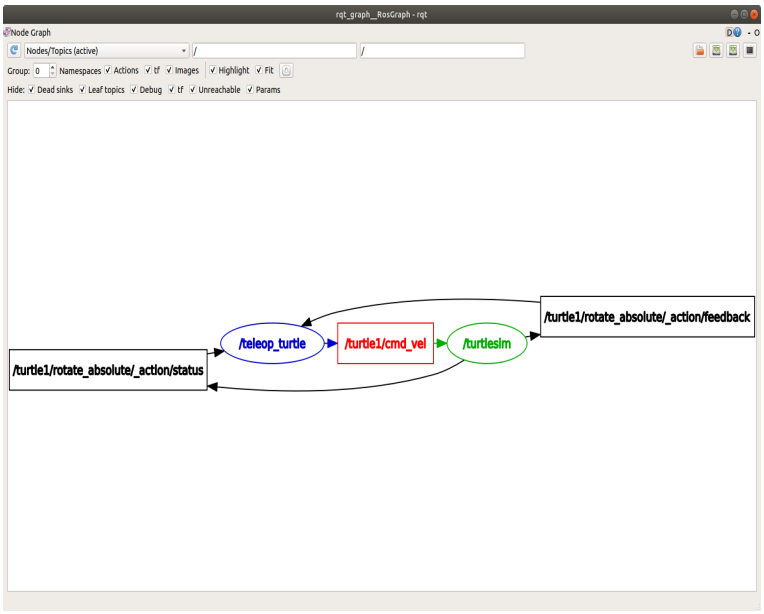


Figura 1.1: Esempio concreto di nodi che comunicano attraverso topic grazie allo strumento di ROS 2 rqt_graph

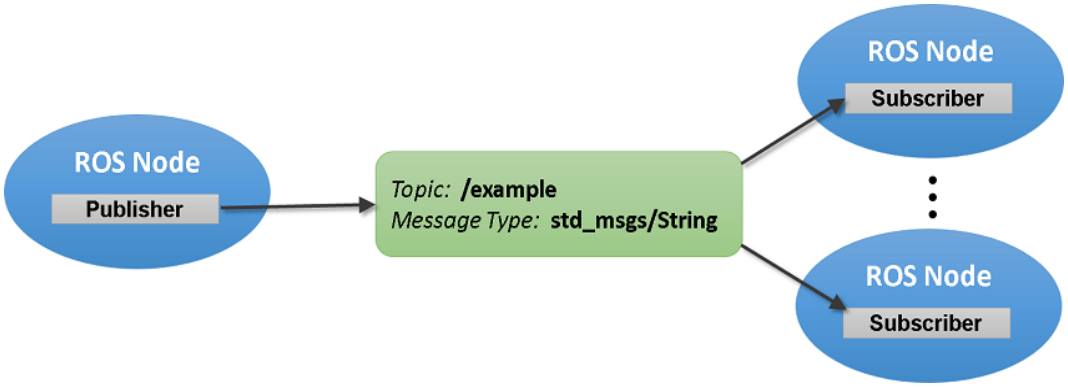


Figura 1.2: Grafico meccanismo di publish/subscribe

1.2 Struttura di un ambiente di sviluppo in ROS 2

L'ambiente di sviluppo per applicazioni di ROS 2 è fondamentale. Infatti, queste applicazioni devono tenere conto di innumerevoli aspetti e devono poter gestire un'alta quantità di file e risorse. Per questo hanno una struttura fortemente gerarchica e facilmente intercambiabile. L'unità base è il workspace, ovvero una directory contenente un insieme di package di ROS 2. É possibile avere più workspaces contenuti fra di loro, denominati meta workspace. Devono avere almeno tutte le dipendenze del workspace che li contiene e possono sovrascrivere le funzioni del workspace principale. Ogni workspace invece è composto da molti package. Il package è sostanzialmente il contenitore del codice. É fondamentale per organizzare ordinatamente il sistema che si sta sviluppando e per permettere il building separato di parti del progetto. Inoltre a differenza di altri sistemi software in cui ogni sviluppatore lavora ad un package alla volta, sia su ROS che su ROS 2 è normale distribuire il codice in un alto numero di package, secondo il principio di riutilizzo/rilascio. Tra i linguaggi di programmazione a disposizione per svliuppare un package, i due principali sono C++ e Python. Nel contesto dell'elaborato, uniformenente con la scelta di C++ come linguaggio di programmazione, è stato scelto di utilizzare il sistema di building *ament_cmake*. Conseguentemente i due file fondamentali e sempre presenti di un package di ROS 2 sono:

1. package.xml: contiene informazioni generali sul progetto e in particolare le dipendenze esterne del tuo package.

2. CmakeLists.txt: si occupa di trovare le dipendenze richieste dal package e di installare correttamente il package.

Essi sono fondamentali in quanto non solo descrivono il pacchetto e contengono informazioni importanti riguardo a licenza, mantenitori e proprietario ma sono necessari per avere una fase di building dei pacchetti corretta. Infatti dichiarano e definiscono il buildtool utilizzato, dipendenze di package esterni ed informazioni necessarie per poter installare e rendere fruibile il package all'interno di ROS 2. Oltre ai due file elencati qui sopra ogni package può disporre di una cartella *src* che contiene i nodi del sistema, una cartella *include* che contiene i rispettivi file header e altre cartelle contenenti file di lancio, file *.world*, che descrivono il "mondo" della simulazione o file *.urdf* o *.sdf*. Nell'ultima sezione del capitolo sarà data una spiegazione più accurata di questi ultimi file. Come specificato nella sezione precedente è consono decidere di organizzare i progetti di ROS 2 utilizzando molti packages. Questo comporta la necessità di avere un sistema di building efficiente ed elastico che permetta di lavorare con singoli package o molti package contemporaneamente. Su ROS erano disponibili molti sistemi di building che

possedevano funzionalità diverse e molto specifiche ma che non garantivano completezza. Da qui è scaturità la necessità di disporre di un unico meccanismo di building completo ed autosufficiente per ridurre i costi di mantenimento dei progetti ed aumentarne la praticità. In questa direzione è stato sviluppato *colcon*, un sistema di building universale in grado di compilare workspaces di ROS e ROS 2 senza esserne intrinsicamente legato. Esso è infatti adibito all'accensione di Gazebo e alla gestione delle dipendenze dei package necessari per il suo funzionamento. Per compilare l'intero workspace è necessario lanciare il comando

nell'origine del workspace, utilizzando la shell del proprio sistema operativo. Terminato il building dei package all'interno del workspace saranno state create da *colcon* tre *directory* fondamentali:

1. la directory *build* contenente i file responsabili del building di ogni package (infatti ci sarà una sottocartella per ogni package all'interno del workspace)
2. la directory *install*, dove saranno effettivamente installati i package
3. la directory *log* responsabile di diversi file di log riguardanti la fase di building svolta da *colcon*.

É possibile eseguire singolarmente ogni nodo del sistema, ma l'aumento dei nodi creerebbe una situazione particolarmente difficile da gestire se affidata unicamente al lancio di ogni nodo singolarmente. È quindi stato sviluppato un sistema di file di lancio con il quale diversi nodi possono essere lanciati su ROS 2 con specifici parametri di inizializzazione e con la possibilità di richiamare altri file di lancio. Il launch file di ROS 2 è anche responsabile del monitoraggio dello stato dei processi lanciati, riportando o reagendo a cambienti nello stato del processi. Inoltre è utilizzabile per lanciare Gazebo e caricare il modello *URDF* del robot all'interno della simulazione. Su ROS 2 è realizzato tramite *Python*. Il package che si occupa della fase di lancio è l*launch_ros*. Quindi, dopo il building del progetto con *concon build*, è possibile lanciare l'intero progetto composto da diversi nodi con un semplice comando di cui ne viene riportato un semplice esempio:

```
ros2 launch myPackage script.launch.py
```

La comunicazione all'interno di ROS 2 si basa su un *DDS* (Data Distributed Service). Il DDS è un middleware, che gestisce un complesso sistema di scambio messaggi basato sul paradigma publish/subscribe. Erede del server "Master" di ROS 1, sulla scia di un server RPC, il DDS permette la definizione di messaggi, la loro serializzazione e deserelizzazione garantendo il corretto funzionamento del meccanismo di publish/subscribe. Per massimizzare l'esperienza

degli utenti e per facilitare l'ingresso nella comunità, ROS 2 provvede a fornire delle interfacce simili a ROS che nasconderanno la maggior parte della complessità, ma che contemporaneamente metteranno a disposizione API per l'accesso al DDS per utenti con necessità molto stringenti o per intergrazioni con altri sistemi DDS, come è visionabile dalla figura1.3. Il DDS si occupa anche di mantenere il formato *.msg* nella comunicazione tra nodi. Per poter essere utilizzato però deve essere convertito nel formato *.idl*, un formato a campi utilizzato dal DDS per trasportare più valori in un singolo messaggio. E' stato verificato tramite esperimenti empirici che la conversione e riconversione attuata dal DDS per trasportare i messaggi è più efficace della serializzazione e invio di ciascun campo.

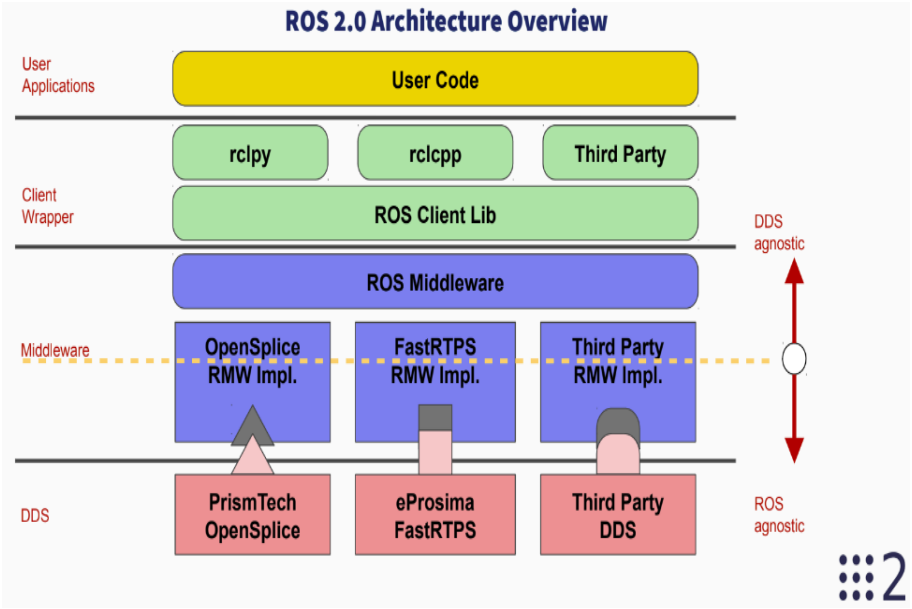


Figura 1.3: Relazione tra stack di ROS 2 e il DDS sottostante

Strettamente legata al DDS e ai servizi offerti la QualityofService (QoS) è un parametro fondamentale per gestire il livello di qualità di trasmissione dati dei nodi e quindi del sistema generale. Calibrando il livelo del QoS possibile diverse tipologie di connesione che possono passare da un TPC a diverse gradazioni di un UDP best effort. Diversamente da ROS, permette di scegliere il livello di affidabilità della comunicazione in base alla tipologia di situazione che si sta affrontando: da comunicazioni UDP con un minor grado di responsabilità ad ambienti real-time con tempistiche più stringenti. I parametri fondamentali che caratterizzano la QoS sono:

1. history: rappresenta la tipologia di salvataggio dei dati che può comprendere tutti i samples scambiati o solamente di un numero N

2. depth: Se i campioni vengono salvati in un numero finito allora il parametro rappresenta l'effettivo numero.(Nel caso il salvataggio sia totale allora è un parametro superfluo).

3. affidabilità: può essere "best effort" o "Reliable" e indica se verrà applicato un protocollo basato su UDP o su TCP

4. durabilità: indica la possibilità da parte del publisher di salvare i dati per eventuali "late-joining" iscrizioni.

5. deadline: il limite massimo di ogni dato per essere pubblicato.

6. livilness: la durata per cui un nodo è considerato ancora "vivo", cioè che può pubblicare dati in un topic.

1.3 GAZEBO E URDF

Gazebo è un simulatore dinamico 3D con la capacità di simulare efficientemente robot in complessi ambienti indoor o outdoor. E' solitamente usato per testare algoritmi su robot, per progettare il design e testarne l'affidabilità in ambienti realistici. Ciò è reso possibile da un ampio set di librerie di modelli e ambienti, un'importante disponibilità di sensori e soprattutto interfacce grafiche molto convenienti gestiti da Gazebo Server, il processo che si occupa di utilizzare i file di configurazione di Gazebo per produrre la simulazione. Il primo componente fondamentale è il file *world*, che contiene la descrizione del mondo e degli oggetti presenti in una simulazione. Incorpora la descrizione dei robot, degli scenari, eventuali luci e un'ampia gamma di sensori concreti di ampia diffusione nel mercato. Successivamente è importante definire il robot che viene utilizzato nella simulazione ed è necessario comporlo in maniera realistica, per garantire una simulazione concreta ed efficiente. Da questa necessità è nato URDF(Universal Robotic Description Format), un formato XML usato su ROS 2 e ROS per descrivere gli elementi costituivi di un robot, come essi sono collegati tra di loro e le loro proprietà fisiche. Per esempio, un corpo rigido è definito dal tag *link* e al suo interno può definire molte proprietà geometriche, sul materiale o che permettono di definire una gerarchia tra i componenti seguendo una struttura ad albero. Ma il parser che analizza il file URDF non sa la precisazione ubicazione del componente. Per definire il collegamento con altri link è necessario inserire il tag $< joint >$, il quale permette di definire l'unione concreta tra due componenti. Utilizzando questi tag principali è facile costruire le basi di qualsiasi robot come è visibile in figura 1.5. All'aumentare del numero di link e di joint del sistema, la complessità del file aumenta consideravolmente e per alleggerire il codice e per evitarne la ripetizione è stato ideato un sistema di scripting per file URDF denominato XACRO. Esso permette la definizione di costanti, ovvero di definire un componente noto del robot sotto uno specifico nome, in modo da non doverlo definire più volte. Ma può anche intervenire a livello matematico, permettendo il calcolo di espressioni che rendono più elastico il design del robot. Infine è molto pratico in quanto permette la definizione di macro, che definiscono componenti attraverso parametri. Se per esempio dovessero servire due gambe speculari, sarebbe possibile crearne due tramite una semplice dichiarazione, riempendo un campo con *left* o *right*. Per poter integrare le descrizioni dei robot del linguaggio URDF è fondamentale l'elemento *gazebo* che permette di aggiungere specifiche proprietà al robot necessarie per il corretto funzionamento all'interno della simulazione . Infatti è possibile integrare al file URDF, di descrizione fisica del robot, una serie di plugin base di Gazebo o una loro estensione creata dall'utente. Un plugin è un pezzo di codice che è compilato ed inserito all'interno della simulazione che ha accesso a tutte le

funzioni di Gazebo. I plugins sono particolarmente utili per:

- lasciare ad ogni sviluppatore pieno controllo della simulazione
- perchè sono inseribili o disinseribili dal sistema in stato di running
- estremamente flessibili
- rappresentano una comoda interfaccia per Gazebo che non produce un eccessivo overhead per la serializzazione e deserializzazione dei messaggi

Ci sono 6 tipologie di plugins disponibili:

1. World
2. Model
3. Sensor
4. System
5. Visual
6. GUI

Ogni plugin è gestito da un differente componente di Gazebo. Un WorldPLugin sarà legato ad un world di Gazebo, mentre un ModelPlugin sarà coinvolto da un file modello che descrive un modello. Gli esempi fondamentali utilizzati nel nostro caso di interesse sono *differential_drive_controller* che fornisce un controllo basico di un robot a ruote differenziali da parte di Gazebo. Garantisce alcune funzionalità fondamentali:

1. può definire il topic da cui leggere la velocità da riprodurre nella simulazione.
2. scegliere il riferimento in base al quale pubblicare odom(ovvero la posizione nel sistema di riferimento assoluto della simulazione).

In conclusione è citato il Gazebo Master, parte fondamentale su cui si basa Gazebo. Provvede a gestire i topic, garantire la corretta traduzione dalla tabella di lookup e si occupa delle librerie per la comunicazione tra nodi. Provvede anche a definire comportamenti fisici che garantiscono realisticità e concretezza degli oggetti e dei loro comportamenti durante l'interazione con altri elementi della simulazione. Inoltre dispone di librerie per il renderning 3D delle scene nella GUI e della loro riproduzione nel simulatore.

Gazebo risulta un'applicazione stand-alone ma con una forte dipendenza con ROS e ROS 2 grazie ad un set di packages chiamati *gazebo_ros_pkgs* nei quali si crea un ponte tra le API di Gazebo e il meccanismo dei messaggi di ROS 2. I packages più importanti da citare e maggiormente fondamentali per lo sviluppo di un'applicazione robotica sono:

1. gazebo_ros_pkgs: un metapackage contenente diversi package di utilità. Si rivelano molto importanti gazebo_dev che fornisce una configurazione di default di Gazebo per le distribuzioni di ROS.

2. gazebo_msgs: predispone strutture dati utili alla comunicazione tra Gazebo e ROS 2.

3. gazebo_ros: convenienti funzioni utilizzabili da plugin.

4. gazebo_plugins: ovvero una serie di plugings di Gazebo che emulano sensori per ROS 2.

Le differenze fondamentali nell'utilizzo di Gazebo tra ROS a ROS 2 sono:

1. lo sfruttamento pieno delle nuove ricche specifiche di ROS 2, come l'assenza del server master.

2. la standardizzazione di funzionalità comuni, come il namespace di ROS, parametri e rimappatura di topic

3. la modernizzazione del codice, grazie alla possibilità di riscrivere il codice URDF esisten con l'ultimo formato *SDF*

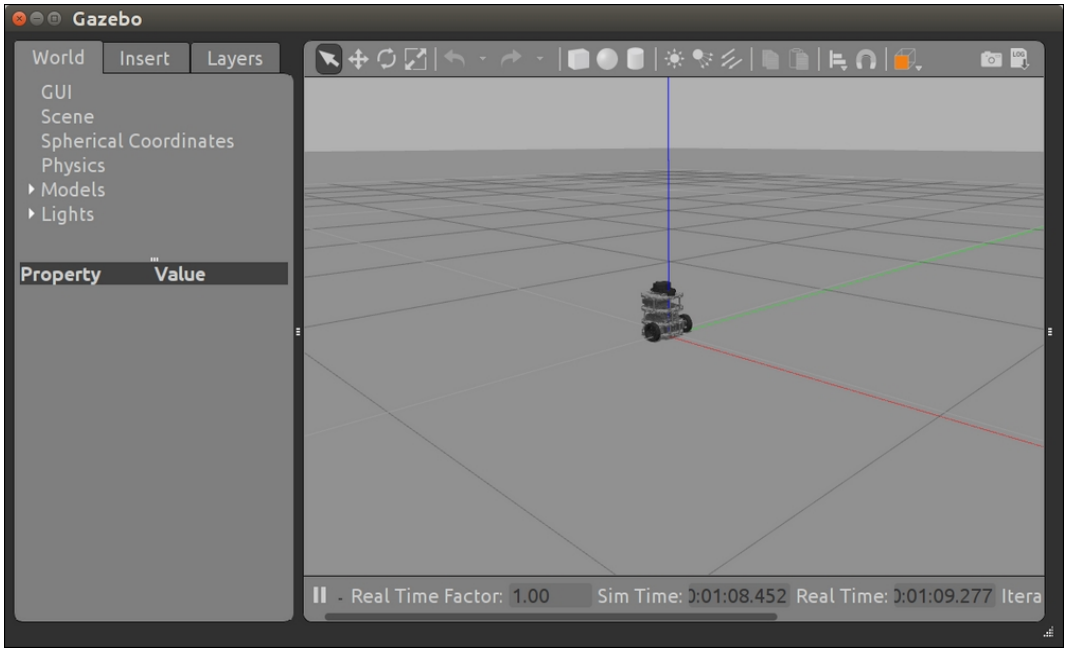


Figura 1.4: Esempio simulazione robot in un mondo vuoto con Gazebo

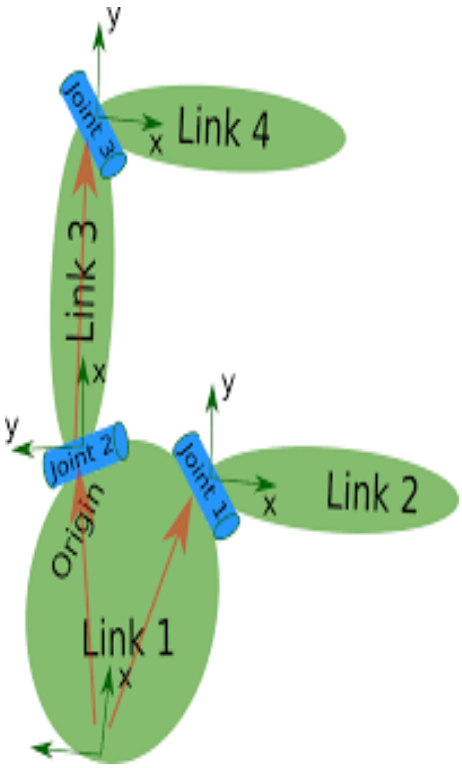


Figura 1.5: Esempio concreto della composizione di un robot tramite link e joint

Capitolo 2

Modellazione e controllo di un uniciclo

Un uniciclo è un veicolo avente una sola ruota orientabile, che generalmente si muove in solo due dimensioni dello spazio con una velocità di spostamento laterale nulla. Nonostante il nome "uniciclo" nella realtà i sistemi con una sola ruota sono particolarmente instabili, di conseguenza in molte situazioni, questo modello può descrivere, seppur in maniera approssimata, modelli di robot a due ruote o modelli semplici di automobili.

2.1 Modello matematico

In questa sezione, considereremo il modello matematico di un robot che si muove nel piano $\{x, y\}$. Nel seguente scenario, è possibile descrivere il suo orientamento utilizzando un solo angolo denominato δ. Il modello matematico risulta dunque:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ \omega \end{bmatrix},$$

dove v è la velocità lineare del robot mentre ω è la velocità angolare del robot lungo l'asse uscente dal piano $\{x, y\}$. Si rimanda il lettore a 2.1 per una rappresentazione grafica.

Nel seguito, si rappresenta come $\delta \in (-\pi, \pi]$ l'orientamento che ha il robot rispetto alla linea di congiunzione tra il robot stesso e il punto di arrivo desiderato. Inoltre, si definisce $\theta \in (-\pi, \pi]$ l'orientamento desiderato del robot quando si trova nella posizione T. Infine, si definisce $r \in \mathbb{R}_{\geq 0}$ la distanza tra il robot e il target nel piano $\{x, y\}$. Si rimanda il lettore alla 2.1 per una

rappresentazione grafica.

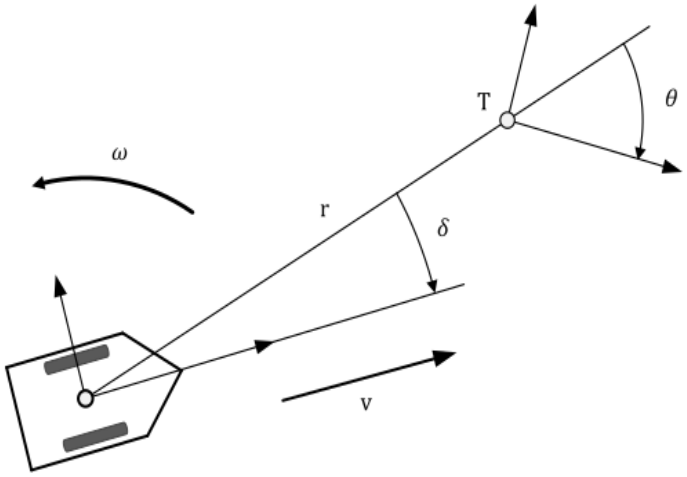


Figura 2.1: Modello dell'uniciclo

Si può quindi scrivere la legge cinematica del veicolo rispetto al robot come:

$$\begin{bmatrix} \dot{r} \\ \dot{\theta} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} -v\cos\delta \\ \frac{v}{r}\cdot\sin(\delta) \\ \frac{v}{r}\cdot\sin(\delta)+\omega \end{bmatrix} \tag{2.1}$$

2.2 Controllore

Lo scopo di questa sezione è descrivere la legge di controllo per l'uniciclo che sarà poi implementata in un'architettura tramite ROS 2. Nel seguente elaborato, l'obiettivo è far raggiungere al robot un punto T. La legge di controllo è basata su una rappresentazione in coordinate polari della posizione del punto target rispetto al veicolo. Per questo motivo, la legge di controllo è anche detta *egocentrica*. Questo cambio di coordinate, come motivato anche in [1], permette di implementare una legge di controllo senza discontinuità e che si avvicina all'esperienza di guida di un pilota umano. L'obiettivo della legge di controllo è ridurre la distanza r tra il robot e il punto di arrivo T e portare il robot a orientarsi ad un angolo θ. Partendo dall'osservare l'equazione (2.1) si può notare che, considerando v strettamente maggiore di zero e ω l'unica variabile di controllo, allora ω controlla unicamente lo stato δ, mentre $(r,\theta)^T$ sono determinate da δ. Inoltre la sola coppia $(r,\theta)^T$ descrive la

posizione del veicolo, mentre δ corrisponde al suo sterzo. Per cui è sensato dividere in due il sistema ottenendo:

$$\begin{bmatrix} \dot{r} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} -v\cos\delta \\ \frac{v}{r}\cdot\sin\delta \end{bmatrix} \tag{2.2}$$

$$\dot{\delta} = \frac{v}{r}\cdot\sin(\delta) + \omega \tag{2.3}$$

Si può quindi notare che ci sono due variabili di controllo, la prima corrisponde a δ, che guida il sotto sistema dell'equazione (2.2) verso l'origine e un controllore effettivo ω che rende la dinamica del sottosistema (2.3) più veloce rispetto al sistema (2.2) e stabilizza δ come un controllo virtuale. Per cui (2.2) diviene un sostto sistema lento e (2.3) diventa un sottosistema veloce a singola perturbazione. La divisione di questi due sistemi rende facile lo spostamento del robot nella posizione-target predefinita con il giusto assetto.
Per quanto riguarda il sottosistema lento, come dimostrato in [1], prendendo una semplice funzione Lyapunov:

$$V = 1/2\cdot(r^2+\theta^2) \tag{2.4}$$

e considerando il controllo virtuale con $k_1 > 0$ e costante

$$\delta = \arctan(-k_1\cdot\theta) \tag{2.5}$$

derivando la funzione di Lyupanov si ottiene che il controllo virtuale δ sterza il sistema dalla posizione iniziale verso l'origine e che è negativa in tutti punti tranne l'origine, quindi è considerabile stabile. Considerando che l'arcotangente è una funzione liscia e che $\texttt{arctan}(0) = 0$ allora avremo che se $\theta \to 0$ allora $\delta \to 0$ e che conseguentemente $(r,\theta,\delta)^T$ si dirige verso l'origine. Inoltre scegliendo v in modo tale da eliminare il punto di singolarità in r si ottiene che l'origine è globalmente stabile.

In seguito è da sviluppare una legge di controllo per lo sterzo del veicolo. Supponiamo z la differenza tra lo stato attuale δ e la proprietà desiderata $\arctan(-k_1\theta)$, tale che:

$$z \equiv \delta - \arctan(-k_1\theta) \tag{2.6}$$

Derivando rispetto al tempo, è possibile dimostrare che la seguente scelta di ω stabilizza l'errore z a 0.

$$\omega = \frac{-v}{r}\left[k_2\cdot+(1+\frac{k_1}{1+(k_1\theta)^2}k_1+(k_1\theta)^2)\cdot\sin(z+\arctan(-k_1\theta))\right] \tag{2.7}$$

che nelle coordinate originali, esplicitando l'input ovvero ω risulta la legge di controllo per ω:

$$\omega = \frac{-v}{r}\left[k_2(\delta - \arctan(-k_1\omega)) + (1 + \frac{k_1}{1 + (k_1\omega)^2})\sin\delta)\right] \tag{2.8}$$

Si evince che abbiamo v come variabile libera e che la forma della traiettoria non dipende da essa.

Ma come è stato poco prima la convergenza asintotica del robot al punto previsto dipende dalla scelta della velocità intorno all'origine. È globalmente asintotica solo se $v \to 0$ per $r \to 0$. Questa scelta di ω e v rende quindi possibile controllare un uniciclo verso un punto T desiderato e con un certo assetto.

Capitolo 3

Implementazione e risultati

L'ultimo capitolo si occupa di fornire spiegazioni dettagliate riguardo l'implementazione del controllore definito nel capitolo precedente, attraverso ROS 2 e i relativi strumenti spiegati nel primo capitolo. Vengono inoltre descritti e commentati i risultati ottenuti dall'implementazione attraverso un insieme di plot e immagini dimostrative. Per l'implementazione è stato utilizzato *ROS 2 Eloquent* per *Ubuntu*. Come meglio approfondito nel capitolo il robot utilizzato per effettuare gli esperimenti è il *turtlebot3*.

3.1 Implementazione

Per l'implementazione del controllore è stato deciso di seguire la struttura classica delle applicazioni robotiche. Generalmente, è una struttura formata da due unità fondamentali, la cui cooperazione risulta necessaria per una guida efficente. Le due entità sono pianificatore e controllore. Il controllore è l'unità che implementa la legge di controllo e che quindi gestisce la correttezza della traiettoria tra il punto iniziale fino alla destinazione T e il giusto assetto finale. Invece il pianificatore sceglie il suddetto punto T e l'orientamento per una navigazione ottimizzata. Esso comunica le coordinate al controllore e riceve feedback riguardo il successo dell'operazione, e molto spesso anche informazioni durante il tragitto stesso. Questa struttura a moduli distinti è estremamente efficace, in quanto una modifica del controllore o la sua sosituzione con una legge di controllo differente, non comprometterebbe la riscrittura del pianificatore. Intuivamente anche se il pianificatore fosse scambiato o modificato, il controllore rimarrebbe immutato.

Come prima operazione è stato dunque creato un ROS 2 package chiamato *position_controller* nel quale inserire controllore e pianificatore. Il paradigma *pianificatore-controllore* è stato implementato utilizzando due distinti nodi di ROS 2 ed è stato utilizzato C++ per ottenere la

massima efficienza. Sono stati dunque creati due file per controllore e pianificatore chiamati rispettivamente:

```
position_controll.cpp
pianificatore.cpp
```

inseriti nella cartella *src* del package in figura **??**. Inoltre, come usuale in applicazioni create con C++, è stato diviso il funzionamento dalla parte di richiamo delle librerie, per questo sono stati creati altri due file denominati

```
position_controll.hpp
pianificatore.hpp
```

dove sono state inserite le dichiarazioni di tutte le variabili e dei metodi utilizzati nel corrispettivo file *.cpp*.

Sono stati creati due topic per garantire la comunicazione della posizione e la conferma del suo raggiungimento. Il primo è stato chiamato *position* ed è dove è stata pubblicata la terna di valori che specifica la posizione e l'assetto finale target. Il messaggio *position* viene letto rapidamente dal controllore eìche successivamente elabora la strategia tramite la legge di controllo. Quando il controllore raggiunge il punto desiderato pubblica sul topic da lui creato denominato *response* l'esito del tragitto, che corrisponderà a fallimento o successo dell'operazione. I messaggi *position* e *response* sono stati definiti in un package a parte specifico per la definzione di messaggi denominato come $position_controller_msgs$. Il messaggio *response* è stato definito:

```
float64 posx
float64 posy
float64 postheta
```

utilizzando dei *float64* per garantire un'accuratezza massima, mentre *position* è statosemplicemente definito come:

```
bool resp
```

dove il valore *true* corrisponderebbe al successo dell'operazione mentre *false* al fallimento dell'operazione.

La QoS di ROS 2 utilizzata per la creazione di questi due topic è stata pensata per ottenere una comunicazione estremamente sicura in quanto la perdita della posizione successiva o il mancato arrivo della conferma della posizione da parte del controllore metterebbe in stallo l'applicazione. È stata data una particolare attenzione ai Per questo il protocollo di comunicazione utilizzato è TCP/IP, che garantisce il completo arrivo del messaggio. Inoltre è stato deciso di mantenere una storia di 10 messaggi per mantenere la massima sicurezza.

3.1.1 Pianificatore

Come accennato precedentemente il pianificatore è composto da un file *.cpp* e da un file *.hpp*. Nel file header sono stati incluse le principali librerie per il funzionamento del pianificatore. Oltre alle consuete librerie di I/O, ed espressioni matematiche le fondamentali sono le librerie per i messaggi della posizione target, il messaggio di conferma per il pianificatore, l'header del messaggio odom ed in particolare rclcpp, necessaria per la creazione di nodi e iscrizione o creazione di topic. Infine è stata dichiarata la classe *Pianificatore* derivata da *rclcpp::Node* e che quindi ottiene tutte le peculiarità di un nodo di ROS 2.
Il file $pianificatore.cpp$ invece è composto da un breve *main* che si occupa di inizializzare gli argomenti passati come parametri, di lanciare la classe rappresentatante il nodo e di eliminare il nodo stesso al termine della sua esecuzione liberando la memoria e le restanti risorse utilizzate. Il pianificatore, nella sua fase di inizializzazione crea un oggetto publisher che può pubblicare sul topic *position*. Mentre si iscrive al topic subscriber che invece indica la risposta del controllore. Il subscriber ha un comportamento asincrono, in quanto un messaggio su un topic può essere pubblicato in qualsiasi momento senza una logica stringente, per questo ad esso viene associato un metodo di callback eseguito ogni volta che il subscriber legge un messaggio sul topic di interesse. Successivamente inizializza un messaggio *Position* e inserisce la terna (x, t, θ), ovvero le coordinate del piano *(x,y)* con il giusto assetto. Infine con l'oggetto publisher creato pubblica sul topic il messaggio, il quale è letto dal nodo del controllore. Successivamente il controllore pubblica sul topic response il valore che può essere 1, se è stato raggiunto l'obiettivo o 0 se l'obiettivo è fallito.

3.1.2 Controllore

Come per il pianificatore anche il controllore richiama nel file header tutte le comune librerie fondamentali di i/o e di funzioni del sistema operativo. Le più rilevanti riguardo la struttura del progetto con ROS 2 sono *odometry*, contenente il messaggio per le coordinate assolute del robot, *tf2* , per effettuare le trasformazioni dal sistema di riferimento assoluto a quelle del robot, e *twist* un messaggio che definisce le componenti della velocità nello spazio. Il controllore crea un subscriber sul topic position, per leggere dal pianificatore la terna (x, t, θ) mentre viceversa rispetto al pianificatore pubblica sul topic response per confermare l'avvenuta ricezione del punto o un eventuale errore se il procedimento non avvenisse correttamente, come se per esempio il messaggio pubblicato sul topic position non fosse composto da 3 valori, o se il robot non fosse riuscito ad arrivare a destinazione. Come seconda operazione il controllore crea un subscriber per il topic *odom*. Il topic odom è fondamentale in quanto rappresenta la posizione in coordinate

assolute di un punto all'interno alla simulazione. Seppure molto frequente l'aggiornamento di odom la pubblicazione è asincrona ed è quindi necessario associargli una funzione di callback denominata *odom_callback*.

La funzione di callback riceve un messaggio il messaggio da odom il quale è composto da una quaterna di valori (x, y, z, ω), con omega velocità angolare. In seguito viene utilizzata la libreria *tf2* che prende in ingresso i valori di odom, ottiene un quaternione, crea la matrice di rotazione e da essa ottiene le rotazioni del robot rispetto ai tre assi cartesiani utilizzando il metodo $getRPY(roll, pitch, yaw)$. I valori roll e pitch in condizioni normali sono trascurabili in quanto non è prevista nessuna rotazione rispetto all'asse x e y, viene solamente effettuato un breve controllo, intuitivo ed immediato per verificare che il robot non sia ribaltato, mentre come visibile in figura 3.1, *yaw* corrisponde al valore in radianti della rotazione intorno all'asse z del robot in riferimento ad odom.

La legge di controllo è implementato nel metodo *updatecallback*, sul quale è impostato un timer di 100 ms, ovvero il periodo ogni quanto viene eseguito il metodo. Dopo che aver ottenuto almeno una volta il valore di odom e la posizione target attravero il topic request, vengono calcolati le variabili necessarie per applicare la legge di controllo (2.5) sulla velocità angolare ω. Per prima cosa viene calcolato il valore di δ come

```
$last_pose_theta - path_theta$
```

In seguito è calcolata la distanza tra la posizione attuale e il punto (x, y) desiderato:

```
path_theta = atan2(goalposey-lastposey, goal_posex - lastposex);
```

ed infine viene calcolata la legge di controllo:

```
twist.angular.z= -((vel/distance)*((k2*(delta-(atan(den))))+(sin(delta)*(1+(k1/den2)))));
```

Come spiegato nel precedente capitolo la legge di controllo è applicata alla velocità angolare ω. Nell'implementazione essa corrisponde al messaggio *geometry_msg::msg::Twist* ed in particolare alla velocità angolare rispetto all'asse z. La velocità lineare del robot v, rappresentata da *twist.angular.x* è stata mantenuta costante a $1\frac{m}{s}$ fino a quando non è stata raggiunta la distanza margine di $0.2m$, dalla quale è stata settata lo stesso valore della distanza dalla posizione target, per ottenere un rallentamento uniforme e controllato. Da notare che in questo caso, come si può intuitivamente notare da (2.1) il numeratore annulla il denominatore assicurando sbalzi di velocità angolare improvvisi con valori di *distance* molto piccoli. Infine il messaggio twist è stato pubblicato sul topic cmd_vel di turtlebot3_diff_drive, che si occupa del movimento. Il robot inizia il tragitto e una volta raggiunta la distanza margine incomincia a deaccelerare arrivando nella posizione T.

Per quanto riguarda l'avvio dell'applicazione è stato definito un sistema di lancio all'interno del

package *position_controller* chiamato *emptyworld.launch.py* che si occupa dell'avvio di Gazebo, caricando il file *world* rappresentante il mondo vuoto e il modello URDF del robot. Infine lancia il secondo file di lancio denominato*robot_state-publisher.launch.py* che invece attiva *robot-state_ publisher* e i due nodi protagonisti *position_controller* e *pianificatore*. Per lanciare dalla barra comando della shell utilizziamo il comando

```
ros2 launch position_controller empty_wolrd.launch.py
```

Infine il comando *rqt_graph*, messo a disposizione da ROS 2, otteniamo quindi una panoramica generale di nodi e topic attivi durante l'esecuzione come visionabile dalla figura 3.2.

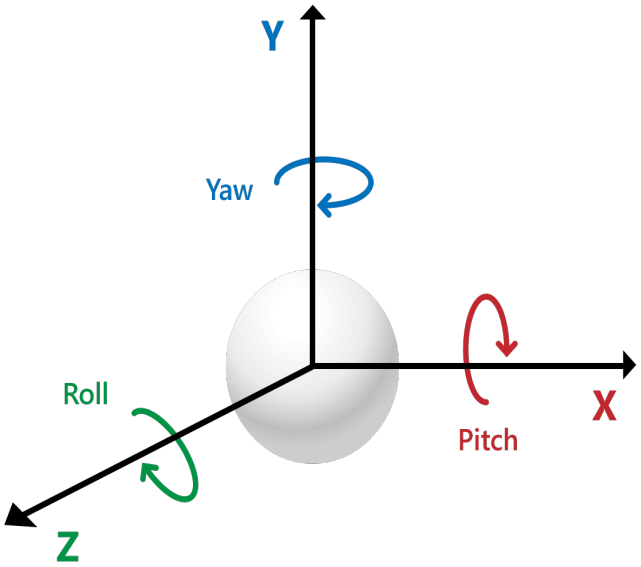


Figura 3.1: Esempio di rotazione intorno agli assi cartesiani

3.1.3 TurtleBot 3

Il robot terrestre utilizzato per eseguire l'esperimento è il TurtleBot 3 modello Burger, uno dei tre modelli appartenenti alla terza serie dei TurtleBot prodotti da Robotis(è open source?). È una piattaforma completamente basata su ROS 1 e su ROS 2 e viene utilizzata per scopi

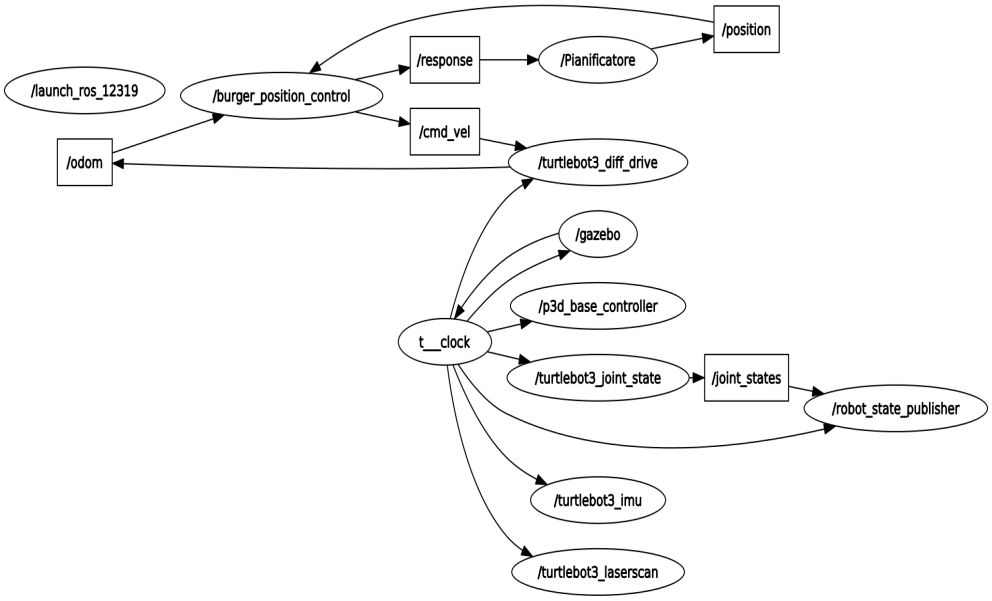


Figura 3.2: Esempio di matrice di rotazione

educativi, ma anche commerciali e di ricerca. È un robot mobile semplice, facilmente estendibile, il cui obiettivo pricipale è di compattare le risorse e di ridurre i costi all'essenziale senza perdere di efficienza e produttività. Come è visibile nell'immagine sottostante, dispone di una telecamera LiDAR per la navigazione SLAM e come computer di bordo dispone di un Rasberry Pi 3 affiancato dal modulo di controllo per i sistemi embedded che usano ROS 1 o ROS 2 denominato OpenCR (OpensourceControlforROS). Uno dei maggiori punti di forza e di ampia diffusione nella comunità di sviluppatori è dovuta alla completà libertà di utilizzo del firmware e del software già prodotto. Come è visionabile su [3], il sito di Robotis fornisce una guida completa per il download delle principali funzionalità del robot, grazie alla quale si ottiene un pacchetto già completo contenente tutti gli strumenti necessari per la simulazione con Gazebo e gli strumenti basilari per la navigazione del robot, facilitando di fatto l'esperienza dell'utente.

3.2 Risultati

Nell'ultima sezione del capitolo saranno analizzati i risultati delle simulazioni effettuate con Gazebo.

Per verificare la correttezza della traiettoria saranno riportati e spiegati dei plot MatLab

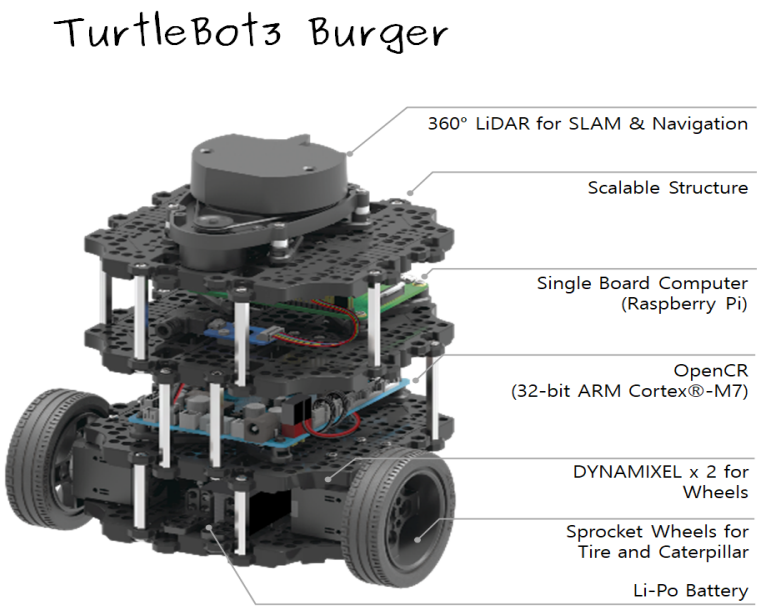
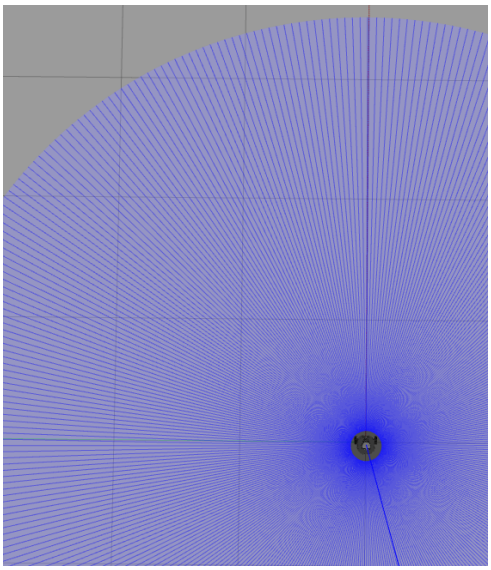
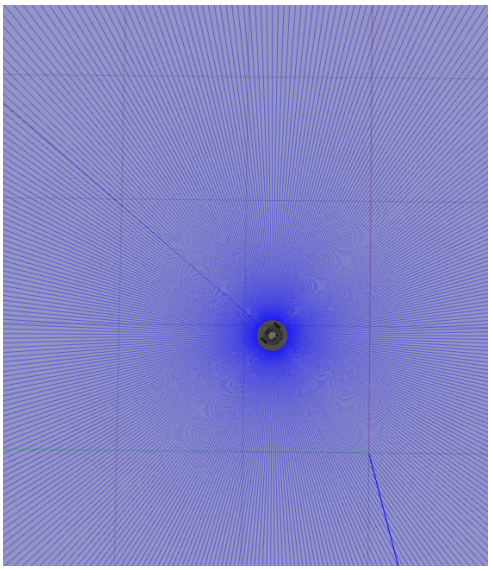


Figura 3.3: scheda componenti del Turtlebot3 modello Burger





rappresentanti la posizione iniziale e finale del robot, selezionando un insieme di punti casuali. Successivamente riproveremo gli stessi percorsi variando i valori di k_1 e k_2 nella legge di controllo, fornendo adeguati commenti riguardo alla precisione e al cambiamento della forma della curva in base al cambiamento dei due valori. Nella prima immagine, il pianificatore ha selezionato qualche punto esempio. ~~Per non creare eccessiva confusione sono stati raggruppati in un'unica immagine~~. I punti selezionati sono *(3.0,3.0,0.82)* con la traiettoria colorata in rosso e *(3.0,2.0,0.26)* con la traiettoria colorata in verde. In particolare le traiettorie hanno come valori significativi $k_1 = 1$ e $k_2 = 3$.

Ecco i due plot disegnati con MATLAB della posizione:

Nella seconda immagine sono stati lasciati inalterati gli obiettivi ma è stato modificato il valore

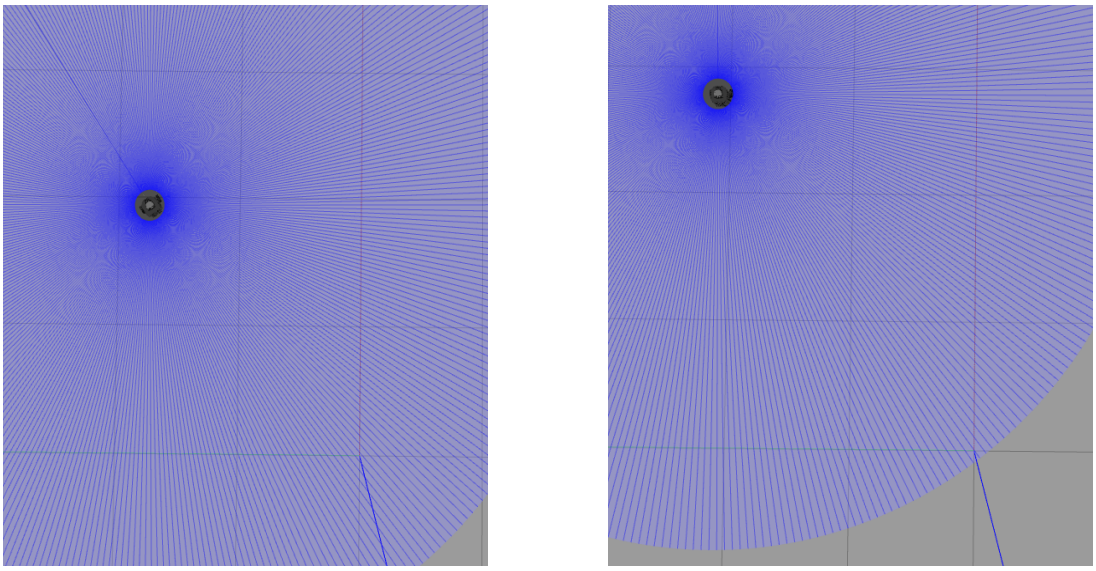


Figura 3.4: Esempio di simulazione con Gazebo

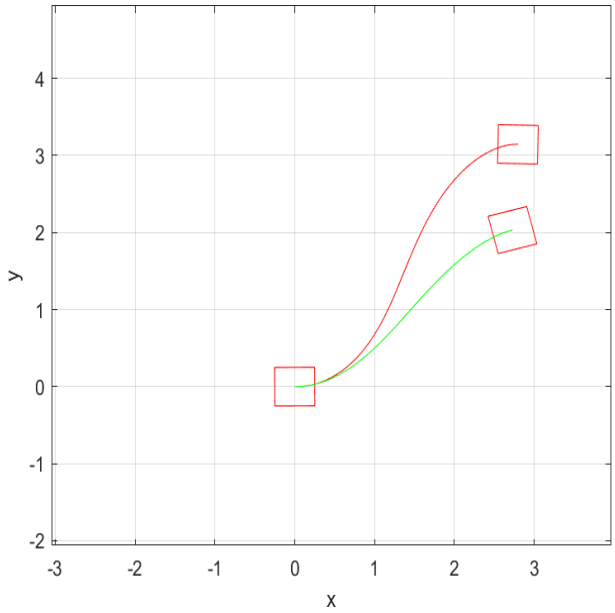


Figura 3.5: Le due traiettorie hanno $k_2 = 3$

di k_2 in $k_2 = 5$. Ed ecco le nuove traiettorie:

Infine, le posizioni sono rimaste immutate ma con $k_2 = 10$:

Dalle immagini qui sopra si può notare come il cambiamento di k_2 modifichi in maniera piuttosto accentuata la forma della traiettoria verso un determinato punto. All'aumentare di k_2 le curve diventano sempre più strette e che quindi la convergenza alla traiettoria finale desiderata è molto

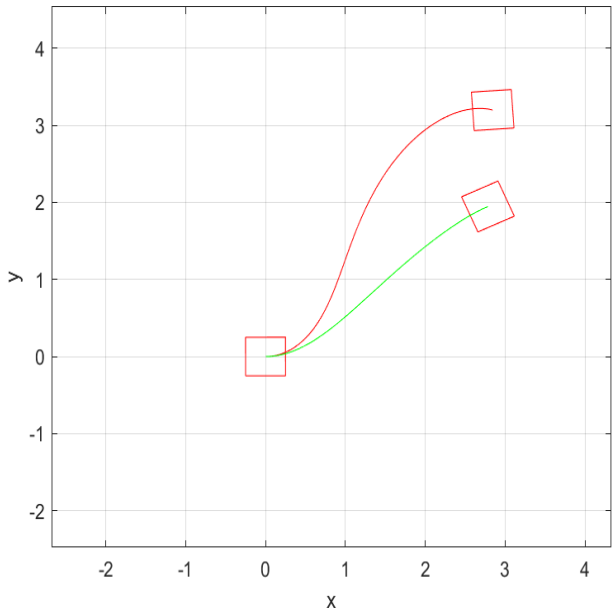


Figura 3.6: Le due traiettorie hanno $k_2 = 5$

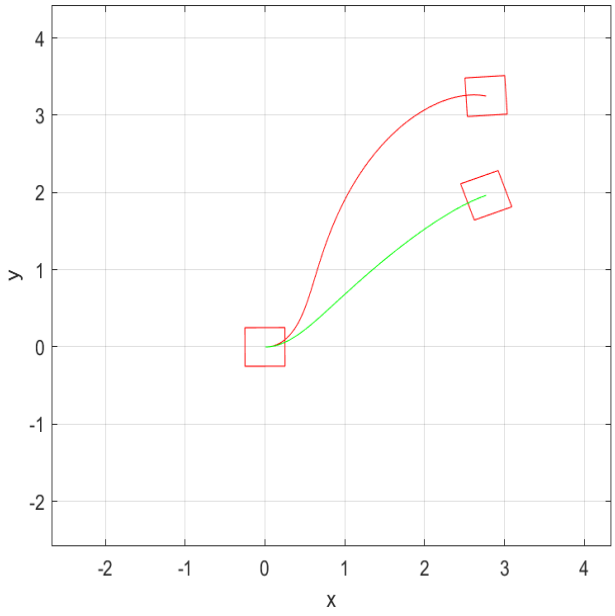


Figura 3.7: Le due traiettorie hanno $k_2 = 10$

più veloce e performante rispetto a dei valori bassi di k_1. Se infatti viene considerata la formula (2.1) è immediato notare che il termine k_2 elevato porta ad un maggiore aumento della velocità angolare e quindi ad un repentino spostamento verso la traiettoria desiderata. In definitiva il controllore implementato nell'elaborato garantisce un'elevata elasticità rispetto a molteplici situazioni, dalla traiettoria più morbida ma sicura con k_1 basso, ad una traiettoria più decisa con un valore di k_2 molto alto. [**?**]

Capitolo 4

Conclusioni

Il controllore implementato permette quindi la navigazione di robot mobili terrestri con un alto grado di fiducia e di efficienza attraverso una piattaforma open-source, che sempre di più negli ultimi anni ha guadagnato popolarità e fiducia nella comunità di sviluppatori in ambito di robotica. Inoltre la struttura molto modulare dell'applicazione consente future estensioni di ciascun componente. Il pianificatore potrebbe essere di fatto esteso in numerosi modi che andrebbero ad arricchire particolarmente la sua complessità ed efficacia tramite algoritmi sempre più evoluti e influenzate da altre discipline come il machine-learning o la computer vision. Invece nel caso del controllore, il primo passo da svolgere sarebbe quello di implementare il meccanismo delle actions di ROS 2 per ottenere una maggiore consapevolezza del tragitto percorso e un meccanismo di feedback più corpulento. Ma successivamente sarebbe possibile decidere di cambiare il robot utilizzato o la legge di controllo senza particolari modifiche nel codice evitand eccessivi costi in termini di tempo e altrettante risorse.

Bibliografia

[1] B. K. J. J. Park, "A smooth control law for graceful motion of differential wheeled mobile robots in 2d environment," 2011.