Abstract

Indice

Elenco delle figure

Introduzione

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 vista in letteratura(da modificare). Semplificando ai minimi termini,un robot per compiere l'obiettivo stabilito deve affrontare diverse sfide presenti nel mondo reale, una salita o un terreno imperverso, per esempio, potrebbero portare ad una perturtbazione della traiettoria di un robot terrestre che si sta muovendo da un punto x ad un punto y. Maggiore sarà la perturbazione, maggiore sarà l'intervallo massimo di errore del robot per raggiungere il punto y. Per garantire la progettazione di robot efficienti e il più possibile precisi ci sono alla base materie fondamentali per uno studio preciso dei modelli di caso d'uso reali come l'analisi matematica per la realizzazione di un modello funzionale e la fisica che ne permette la concretizzazione nel mondo reale. L'informatica invece è la componente fondamentale dal punto di vista computazionale in quanto si occupa della trasmissione dei dati, dell'immagazzinamento delle informazioni e della loro elaborazione dando forma all' "intelligenza artificiale" capace di saper svolgere autonomamente innumerevoli compiti. Per la progettazione lato software il modo migliore per approcciare il problema è di utilizzare frameworks che permettono la fruizione di strumenti e soluzioni a problemi comuni in maniera semplice ed esaustiva evitando la continua reinvenzione di soluzioni a problemi noti. Da questo lato il framework di riferimento è ROS. ROS è una collezione di librerie software open source (Python, C++) e tools progettate al fine di aiutare gli sviluppatori nella realizzazione di applicazioni robot. E' considerato un sistema operativo anche se in realtà dispone sia di molte peculiarità tipiche dei comuni sistemi operativi come l'astrazione dell'hardware, sia di caratteristiche tipiche dei middleware e caratteristiche tipiche di un framework software. Il controllore è stato progettato basandosi sulla legge di navigazione dei robot a ruote

differenziali. E' implementato utilizzando i concetti basilari di ROS 2, utilizzando tramite lo scambio di messaggi tra "nodi" con i topics. Il controllore riceverà le coordinate nello spazio dal pianificatore, ovvero l'entità che programma il percorso, seguendo il pattern pubblish/subscribe che permetterà il raggiungimento al punto desiderato con un margine di errore molto basso.

Motivazioni

Letteratura

Contributi

Relatore: Giuseppe Notarstefano Correlatore: Andrea Testa

Capitolo 1

STRUMENTI DI SVILUPPO

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 comune 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. Le librerie permettono la fase di building dei packages, la creazione di workspaces 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 linguaggi di programmazione noti 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 ROS2

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 di loro, ma che necessitano di una forte coesione e per questo devono comunicare tra di loro in maniera efficiente e poco dispendiosa di risorse. L'unità elementare di

un componente di ROS 2 e ROS 1 è il nodo. Un nodo è 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 attraverso topics pubblici, i quali si scambiano messaggi dalle funzionalità più disparate, dal messaggio per indicare la velocità nello spazio del robot, al messaggio sulla sua posizione(si possono usare anche parametri nelle funzioni). 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 fatale 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 ROS2 installato sul proprio calcolatore.
I nodi per comunicare hanno bisogno di un canale di comunicazione che possa unire ogni nodo del sistema ai restanti secondo una logica che segue un argomento. L'argomento in questione è il topic, ovvero il 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 files. Per questo è stato deciso di implementare un meccanismo che permetesse una comunicazione efficace ma a bassi costi e molto rapida, ed è da qui che è nata l'idea di utilizzare un paradigma di publish/subscriber o modello ad eventi. 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 esso non crea una 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 pubblici all'interno del sistema e si iscivono ad altri topic da cui leggono i messaggi permettendo diverse tipologie di scambio come per esempio 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 nel topic. Questo per evitare confusione tra le diverse parti in gioco e per dividere la comunicazione fortemente a livello semantico. Esistono topic fondamentali che sono già implementati su ROS 2 e che quindi utilizzano un messaggio già definito a priori. 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 del nodo l'istanziazione di un subscriber non è una condizione sincrona bloccante ma asincrona e non bloccante. 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 la funzione 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. 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 del di variabili di moderni linguaggi di programmazione. Definito il messaggio (.msg) il DDS (riferimento sezione successiva) si occuperà della conversione dal valore utilizzato dal compilatore o interprete del linguaggio di programmazione utilizzato per la creazione del nodo al tipo di dato trasportato e riconosciuto dal DDS come mostrato in figura **??**. 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. Un esempio è il seguente: $float64 a float64 b float64 c$ Nell'esempio 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.

Questo messaggio di esempio composto da 3 numeri di nome a,b e c(int64 rappresenta un intero composto da 64 bit).

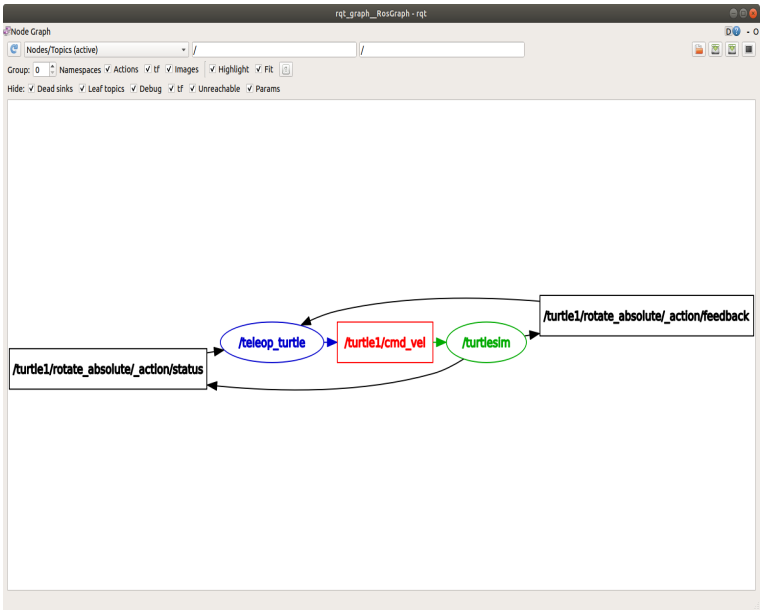

Figura 1.1: esempio concreto di nodi che comunicano attraverso topic grazie allo strumento di ROS 2 rqt_graph

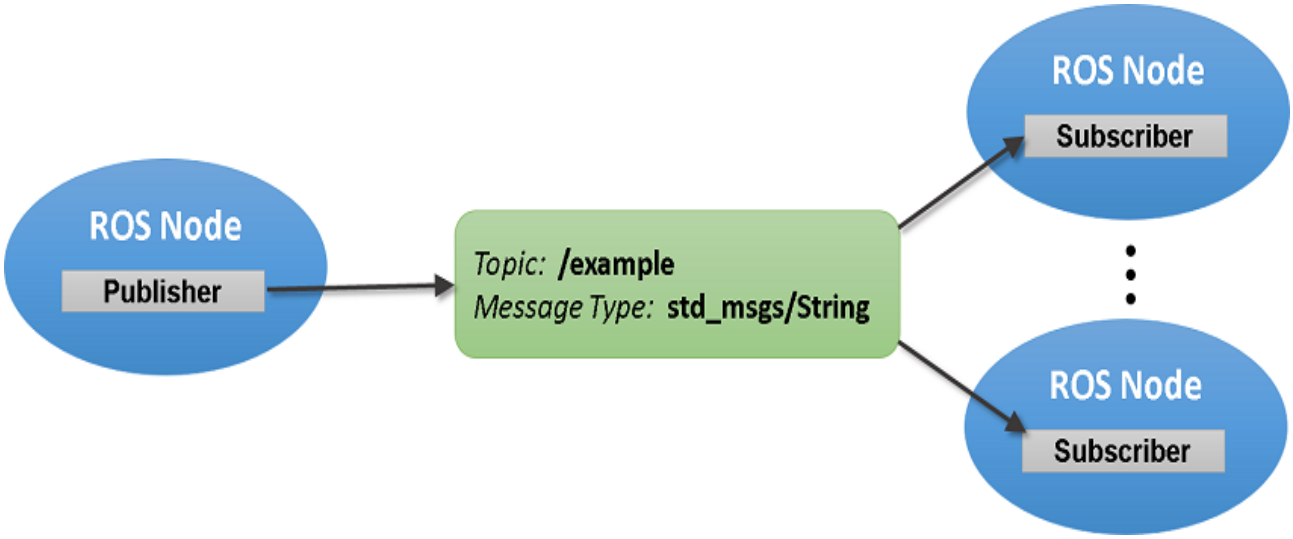

Figura 1.2: grafico meccanismo di publish/subscribe

1.2 STRUTTURA DELL'AMBIENTE 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 che contiene un insieme di package di ROS2. E' possibile avere più workspaces contenuti fra di loro, denominati meta workspace. Questi però devono avere almeno tutte le dipendenze del workspace che li contiene e inoltre possono sovrascrivere le funzioni del workspace principale. Ogni workspace invece è composto da molti package.
Esso è sostanzialmente il contenitore del codice. E' 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. Per fare il building del package sono disponibili due soluzioni: Python o CMake.
I due file fondamentali e sempre presenti di un package di ROS2 sono:

1. package.xml contiene informazioni generali sul progetto e in partico- lare le dipendenze esterne del tuo package.

2. CmakeLists.txt che 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 senza errori. 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 ROS2. 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 file header dei progetti e possibilmente altre cartelle che contengono i file di lancio, i file .world che descrivono il "mondo" della simulazione o per esempio file .urdf o .sdf specifici sulla descrizione fisica del robot.
Come specificato nella sezione precedente i progetti di ROS 2 sono separati all'interno di molti packages. Questo comporta la necessità di avere un sistema di building efficiente ed elastico che permetta di lavorare con singoli packages o molti package contemporaneamente e che sia in grado di gestire la fase di building dei package di Gazebo. Su ROS 1 erano disponibili molti sistemi

di building che possedevano funzionalità diverse e molto specifiche ma le quali non garantivano completezza. Da qui è scaturità la necessità di disporre di un unico meccanismo di building completo ed autosufficiente r ridurre i costi di mantenimento dei progetti. In questa direzione è stato sviluppato colcon, un sistema di building universale che è in grado di compilare workspaces di ROS1, ROS2 senza esserne estremamente legato, infatti esso è adibito anche all'accensione di Gazebo e alla gestione delle dipendenze dei suoi package. Per compilare l'intero workspace è necessario lanciare il comando *colconbuild* nell'origine del workspace. Al termine del building dei package all'interno del workspace saranno state create da colcon 3 cartelle fondamentali all'interno della cartella src del workspace principale:

1. la cartella build che contiene i file responsabili del building di oogni package(infatti ci sarà una sottocartella per ogni package all'interno del workspace)
2. la cartella install, dove saranno effettivamente installati i packages
3. la cartella log che contiene diversi file di log sull'intera fase di building svolta da colcon.

1.3 Strumenti e composizione

ROS2 si basa su un DDS. Il DDS è un middleware, che gestisce un complesso sistema di scambio messaggi basato publish/subscribe. Erede del server "Master" di ROS 1 sulla scia di un server RPC, il DD utilizza il linguaggio IDL per la serializzazione di messaggio, rendendo la comunicazione più flessibile e tollerante a guasti. 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 provvederà a fornire delle interfacce simili a ROS 1 che nasconderanno la maggior parte della complessità, ma che contemporaneamente metterà a disposizione API per l'accesso al DDS per utenti con necessità molto stringenti o per intergrazioni con altri sistemi DDS, **??**. 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 concreti che la conversione e riconversione attuata dal DDS per trasportare i messaggi è più efficace della serializzazione e invio di ciascun campo.
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 ROS1,

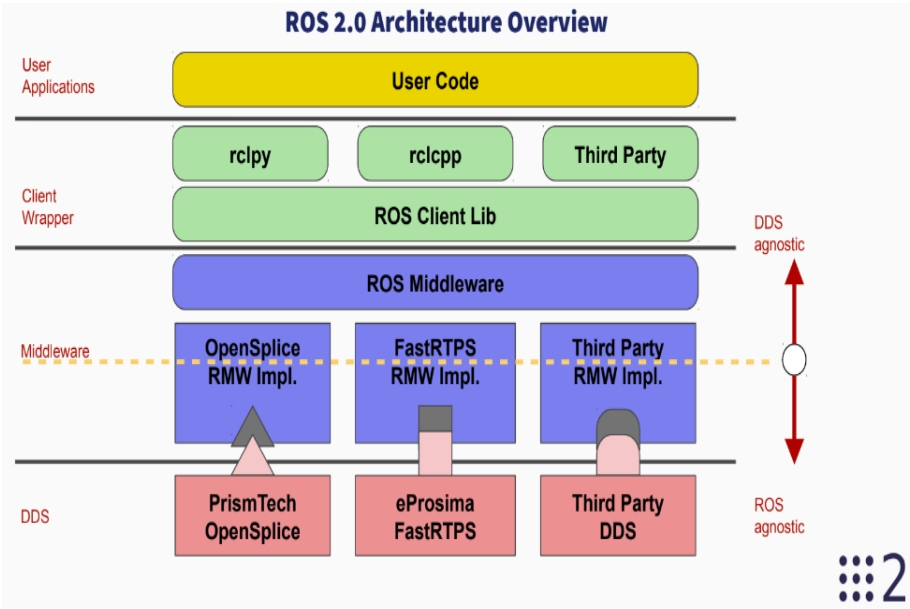

Figura 1.3: DDS

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.

E'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. E'quindi stato sviluppato un sistema di file di lancio con il quale diversi nodi possono essere lanciati su ROS2 con specifici parametri di inizializzazione e con la possibilità di richiamare altri file di lancio. Il launch file di ROS2 è anche responsabile del monitoraggio dello stato dei processi lanciati, riportando o reagendo a cambienti nello stato del processi.Su ROS2 viene creato utilizzando Python e al suo interno specificherà il package e il nome dei nodi da eseguire durante la fase di avvio. Il package che si occupa della fase di lancio è launch_ros, il quale utilizza il framework launch(framework generale e non specifico per ROS). Dopo il completamento della fase di build con concon build è possibile lanciare l'intero "progetto" composti da diversi nodi con un semplice comando:
$ros2launchmy_packagescript.launch.py$

1.4 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, ovvero il processo che si occupa di leggere i file di configurazione e che simula il "mondo" della simulazione utilizzando un set di librerie che leggi fisiche all'interno della simulazione. Il primo componente fondamentale è il file world che contiene tutti gli oggetti della simulazione. Contiene la struttura dei robot e degli scenari, eventuali luci e un'ampia gamma di sensori concreti di ampia diffusione realmente. 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 ROS2 e ROS per descrivere gli elementi costituivi di un robot, come essi sono collegati tra di loro e le loro reali 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 $< join >$, 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 **??**. All'aumentare del numero di link e di join del sistema, la complessità del file aumenta considerevolmente 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. Così facendo se dovessero servire due gambe speculari sarebbe possibile crearne due tramite una semplice dichiarazione, definendo qual è la destra e quale la sinistra. 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 plugins 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 ed ha accesso a tutte le funzioni di Gazebo I plugins sono particolarmente utili per:

- lasciare ad ogni sviluppatore pieno controllo della simulazione
- possono essere inseriti e e disinseriti dal sistema in runnning.
- sono molto flessibili
- sono una facile interfaccia per Gazebo senza 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 Gazeb, mentre un plugin sul modello sarà coinvolto da un file 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 viene citato il Gazebo Master,la 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, per comportamenti fisici che garantiscono realisticità e concretezza negli oggetti e nei loro comportamenti durante l'interazione con altri oggetti. Inoltre disponde anche di librerie per il renderning 3D delle scene nella GUI e della loro riproduzione nel simulatore.

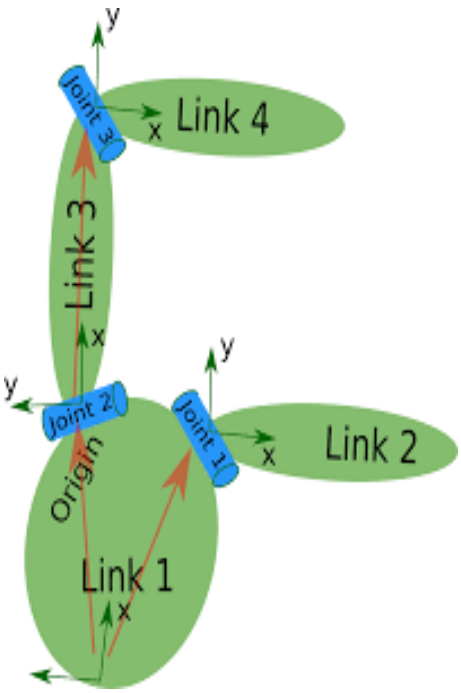

Figura 1.4: urdf

1.4.1 Packages fondamentali e progressi da ROS1 a ROS2

E' quindi un'applicazione stand-alone ma con una forte dipendenza con ROS e ROS2 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 ROS2. I packages più importanti da citare e maggiormente fondamentali per lo sviluppo di un'applicazione robotica:

1. gazebo_ros_pkgs, è 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, che predispone strutture dati utili alla comunicazione tra Gazebo e ROS2.

3. gazebo_ros, provvede convenienti funzioni utilizzabili da plugin.

4. gazebo_plugins, ovvero una serie di plugings di Gazebo che rispecchiano sensori per ROS2. Un esempio noto è gazebo_ros_diff:drive che fornisce una interfaccia per controllare la velocità differenziale del robot all'interno della simulazione.

Le differenze fondamentali che ha subito il package da ROS1 a ROS2 sono:

1. sfruttare a pieno le nuove ricche specifiche di ROS2, come l'assenza del server master.

2. sono state standardizzate funzionalità comuni, come il namespace di ROS, parametri e rimappatura di topic

3. modernizzare il codice, utilizzando l'ultimo formato SDFormat

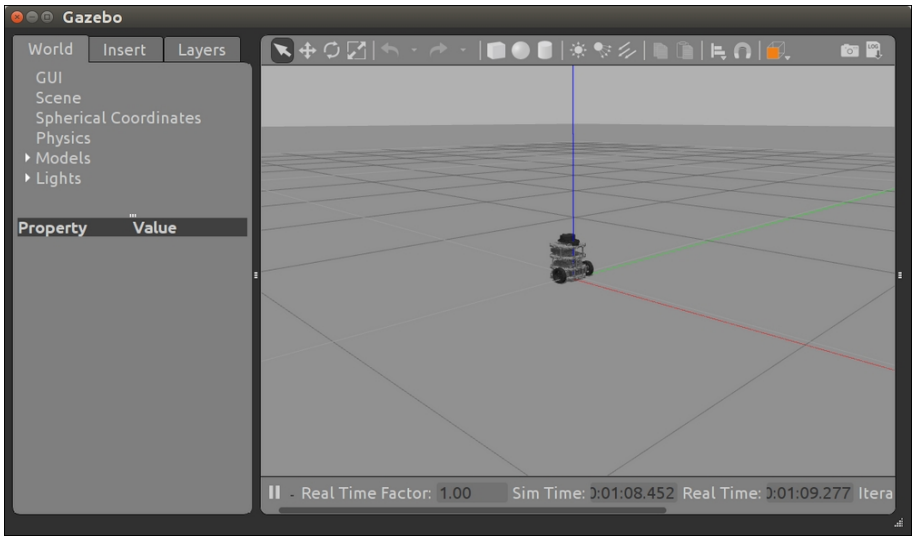

Figura 1.5: esempio simulazione robot in un mondo vuoto con Gazebo

Capitolo 2

Modellazione e controllo di un uniciclo

2.1 Modello matematico

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.
Il modello matematico si impone di costruire un percorso sicuro, veloce ed intuitivo. Per ottenere questi requisiti la guida deve avere velocità accellerazione limitate e selezionate in base alla posizione nel tragitto.
In questa sezione dunque, considereremo il modello matematico di un robot che si muove nel piano $\{x, y\}$. In questo scenario, è possibile descrivere il suo orientamento tramite un solo angolo, d'ora in avanti denominato ψ. Il modello matematico risulta dunque essere:

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

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 Figure— 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 Figure– per una rappresentazione grafica.

DA SPOSTARE SUCCESSIVAMENTE NEL CONTROLLORE.

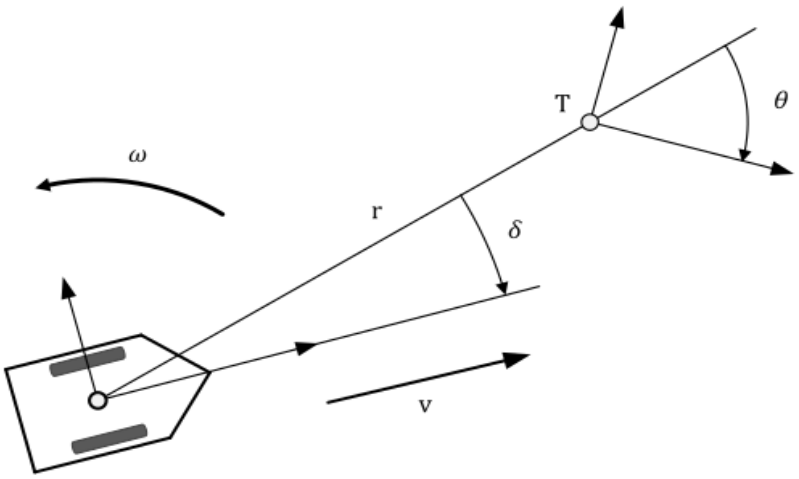

Figura 2.1: modello dell'uniciclo

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

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

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 orientamento.
Per quanto riguarda il sottosistema lento è stato dimostrato da (citazione dell'articolo) che prendendo una semplice funzione Lyapunov:

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

e considerando il controllo virtuale con $k1 > 0$ 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 $(r, \theta, \delta)^T$ che 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 da come è stato specificato nella sezione precedente la convergenza asintotica del robot al punto previsto dipende dalla scelta della velocità intorno all'origine. E' 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 desiderato e con un certo assetto. Inoltre è estendibile nella costruzione di una traiettoria, in quanto potrebbe esser ideata come un insieme di punti nello spazio con un determinato orientamento. L'insieme dei punti che comporrebbero la traiettoria sarebbero asintoticamente stabili garantendo un percorso affidabile, veloce ed intuitivo.

Capitolo 3

IMPLEMENTAZIONE E RISULTATI

Lo scopo di questa sezione è descrivere la legge di controllo per l'uniciclo che sarà poi implementata in un'architettura in ROS2. In questo contesto, 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 [citazione], permette di implementare una legge di controllo senza discontinuità e che si avvicina al punto di vista di un pitola umano.

3.1 Modellazione

3.1.1 Controllore e pianificatore

Nella maggior parte delle applicazioni robotiche classiche la struttura generale del sistema è composto da due unità fondamentali, la cui cooperazione risulta neecssaria per poter guidare in maniera intelligente un robot in un ambiente reale. Le due entità in questione sono pianificatore e controllore. Il controllore è l'unità che implementa la legge di controllo e che quindi gestisce la correttezza del percorso tra due punti dello spazio in cui il robot si deve muovere. Invece il pianificatore decide in che posizione e con che orientamento il robot si deve spostare e riceve feedback riguardo il successo dell'obiettivo imposto e spesso anche informazioni durante il tragitto stesso.

Come prima operazione è stato dunque creato un ROS2 package chiamato $position_controller$ nel quale inserire controllore e pianificatore.(in seguito verrà spiegato meglio) Nel caso preso in

esame, il meccanismo "pianificatore-controllore" è stato implementato utilizzando due distinti nodi di ROS2 e per ottenere la massima efficienza è stato scelto il linguaggio C++ per la loro implementazione. Sono stati dunque creati due file per controllore e pianificatore chiamati rispettivamente: $position_controll.cpp$ e $pianificatore.cpp$ inserite nella cartella *src* del package come si può vedere nella figura 3.1. Inoltre, come è di buona pratica in applicazioni create con C++ è normale dividere il codice vero e proprio dalla parte di richiamo delle librerie, per questo sono stati creati altri due file denominati $position_controll.hpp$ e $pianificatore.hpp$ dove sono state inserite le dichiarazioni di tutte le variabili e dei metodi utilizzati nel corrispettivo file .cpp. In via generale sono stati creati due topic per garantire la comunicazione. Il primo topic, creato dal pianificatore è stato chiamato "position" ed è il canale dove viene pubblicata la terna che specifica la posizione da raggiungere per il robot. Il messaggio Position(spiegato successivamente nella sezione messaggi) viene letto dal controllore e da qui elabora la strategia. Quando il controllore raggiunge il punto desiderato pubblica sul topic da lui creato chiamato "response" l'esito del tragitto che in maniera approssimata corrisponderà a fallimento o successo dell'operazione.(possibile estensione?). La QoS 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. Per questo il protocollo di comunicazione utilizzato è TCP/IP, il quale garantisce l'arrivo dell'intero messaggio.

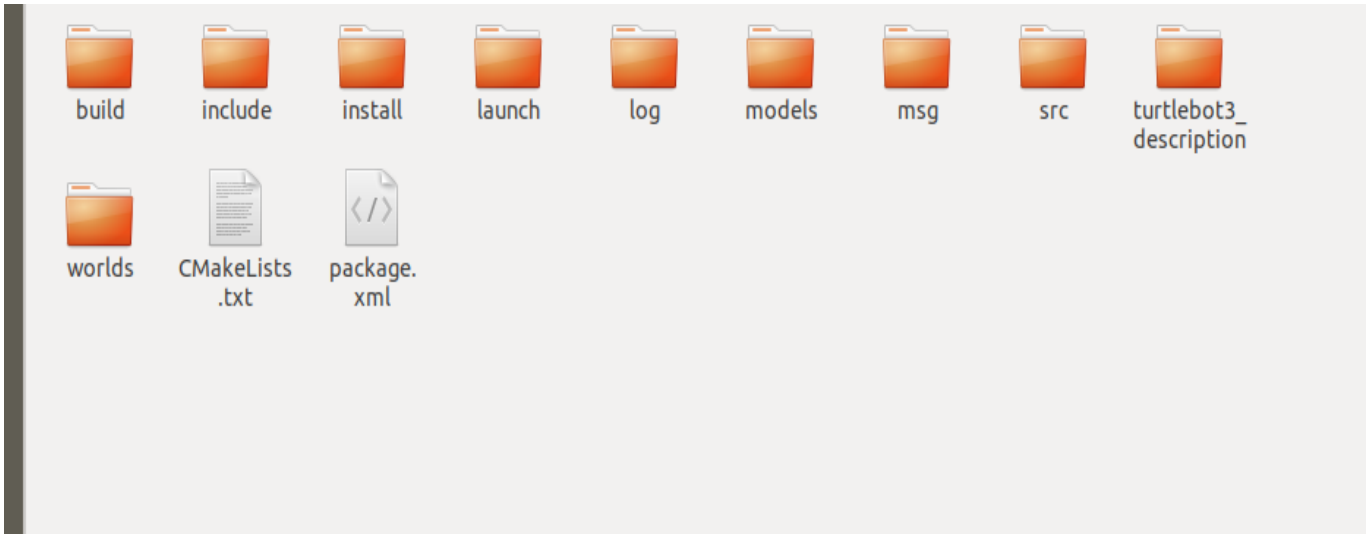

Figura 3.1: modello dell'uniciclo

3.2 Controllo

3.2.1 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, in particolare rclcpp, che è necessaria per la creazione dei nodi, dei topic e per poter iscriversi ad un topic, il file header per i due messaggi response e position. In particolare è stata dichiarata la classe Pianificatore che deriva $rclcpp :: Node$ e che quindi ottiene tutte le peculiarità di un nodo per ROS 2. Il file pianificatore.cpp invece è composto da un breve che si occupa di inizilaizzare gli argomenti passati come parametri dalla barra di comando, di lanciare la classe rappresentatante il nodo e di eliminare il nodo al termine della sua esecuzione liberando la memoria e le rimanenti risorse utilizzate.

Il pianificatore, nella sua fase di inizializzazione crea un 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 la posizione nello spazio del robot con il giusto orientamento. infine attraverso il publisher pubblica sul topic la terna, la quale verrà letta dal controllore. Dopo che il controllore pubblica sul topic response esso valuta il risultato ottenuto ed invia una nuova terna al controllore.

3.2.2 Controllore

Come per il pianificatore anche il controllore richiama nel file header tutte le librerie fondamentali per lo sviluppo nel quale i più rilevanti riguardano le coordinate assolute del robot, la trasformazione di queste coordinate nel sistema di riferimento del robot,la possibilità di definire la velocità e la consueta rclcpp.

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. Successivamente il controllore crea un subscriber sul topic odom. Il topic odom è un topic

fondamentale di ROS 2 e presente all'interno di rclcpp in quanto rappresenta la posizione in coordinate assolute di un punto all'interno del mondo della simulazione.
Il valore di odom è continuamente aggiornato e viene pubblicato dal nodo turtlebot3_diff_drive, che calcola la posizione del robot(da scrivere meglio). 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

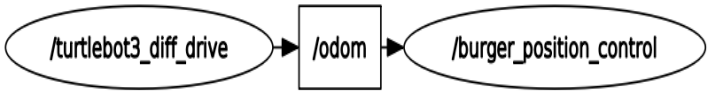

Figura 3.2: modello dell'uniciclo

che descrive la posizione di una parte del robot all'interno dello spazio, e quindi è composto dalla quaterna cartesiana (x, y, z, ω). In seguito viene utilizzata la libreria tf2 che prende in ingresso la quaterna, crea la matrice di rotazione e da essa ottiene le rotazioni del robot rispetto ai tre assi x, y e z utilizzando la funzione getRPY(roll, pitch, yaw). I due valori rolle pitch in condizioni normali sono trascurabili in quanto non è prevista nessuna rotazione rispetto all'asse x e y, viene solamente effettuato un controllo per verificare che il robot non sia caduto a terra, mentre, come mostrato in figura 3.3 yaw corrisponde al valore della rotazione intorno all'asse z del robot con riferimento al valore di odom.

Infine la legge di controllo è implementato nel metodo update_callback, sul quale è impostato un timer di 100 ms, ovvero il periodo ogni quanto viene eseguito il metodo. Dopo aver accertato che il nodo abbia ricevuto almeno una posizione di odom e che su position sia stato pubblicato il primo obiettivo da raggiungere sono 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) con la formula $path_theta = atan2(goalposey - lastposey, goal_posex - lastposex)$; desiderato ed infine viene calcolata la legge di controllo.

$$twist.angular.z = -((vel/distance) * ((k2 * (delta - (atan(den)))) + (sin(delta) * (1 + (k1/den2)))));$$ (3.1)

$geometry_msg :: msg :: Twist$ è un messaggio di ROS 2 che esprime la velocità di un robot. Nel nostro caso eravamo interessati alla velocità angolare del robot rispetto all'asse z, per cui è stato scelto angular.z. Come descritto nel capitolo precedente, la velocità lineare del robot v (nel caso di ROS2 twist.angular.x)è stata tenuta costante ad $1\frac{m}{s}$ fino a quando non è stata

raggiunta la distanza limite di $0.2m$, dalla quale è stata scelta come velocità lo stesso valore della distanza dalla posizione target per rallentare in maniera uniforme il robot e per annullare in (3.1) il termine asscicurandoci di non avere comportamenti pericolosi con *distance* eccessivamente piccolo. Infine il messaggio twist è stato pubblicato sul topic cmd_vel di turtlebot3_diff_drive. Con il comando rqt_graph,messo a disposizione da ROS 2, andiamo quindi a controllare durante l'esecuzione del sistema i principali nodi e topic che vengono pubblicati in figura **??**.

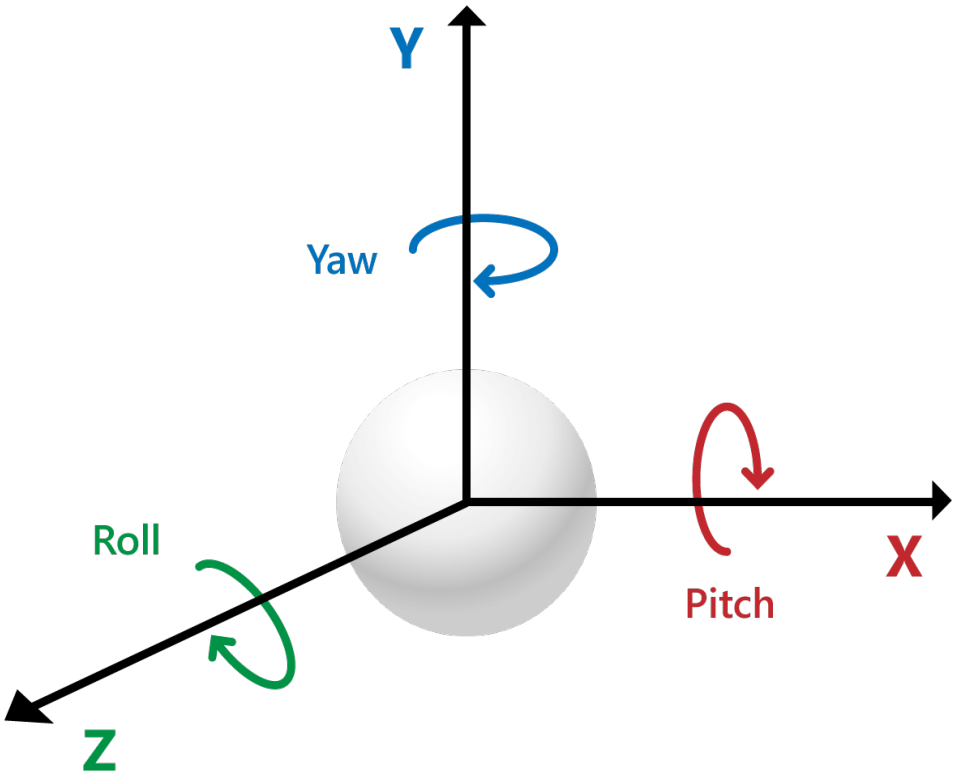

Figura 3.3: Esempio di matrice di rotazione

Bibliografia