SAPIENZA
UNIVERSITÀ DI ROMA

Implementazione della collision avoidance per una base mobile in ROS

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Stefano Bonetti
Matricola 1764624

Relatore

Prof. Giorgio Grisetti

Anno Accademico 2019/2020

Implementazione della collision avoidance per una base mobile in ROS
Tesi di Laurea. Sapienza – Università di Roma

© 2020 Stefano Bonetti. Tutti i diritti riservati

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: bonetti.1764624@studenti.uniroma1.it

A coloro che sono
sempre rimasti con me

Sommario

Uno dei temi ricorrenti nel Laboratorio di Intelligenza Artificiale e Grafica Interattiva sono stati gli agenti autonomi, ovvero dispositivi in grado di portare a termine un compito autonomamente. Riguardo ad essi, un argomento di cui parleremo in questa relazione è la navigazione, cioè in breve la capacità di un robot di sapersi muovere e localizzare da solo in un certo ambiente. È chiaro che analizzeremo meglio questi concetti, ma per il momento ci basta sapere che nella navigazione esistono tre procedure chiamate mapping, localizzazione e path planning. Durante il Laboratorio abbiamo visto principalmente i primi due, pertanto in questa relazione si è cercato di andare oltre e fare un tentativo di esplorare almeno in parte il campo del path planning.

Indice

Introduzione

Obiettivo di questa relazione è l'implementazione di un algoritmo di collision avoidance, ovvero una procedura che introduca un certo livello di sicurezza per un robot mobile controllando la presenza di ostacoli nei suoi dintorni e prendendo misure adeguate per evitarli. Per tale scopo, dobbiamo prima affrontare alcuni argomenti inerenti la robotica:

- nel Capitolo 1 parleremo della tipologia di robot utilizzata e di alcuni sensori tipici;
- nel Capitolo 2 analizzeremo ROS, una nota infrastruttura su cui creare programmi per la robotica;
- nel Capitolo 3 si vedrà in che modo i robot possono muoversi autonomamente nell'ambiente;
- nel Capitolo 4 ci sarà il programma vero e proprio e l'implementazione dell'algoritmo di prevenzione delle collisioni;
- infine nel Capitolo 5 vedremo alcuni test.

Capitolo 1

Robot mobili e sensori per la navigazione

Prima di tutto, analizziamo il tipo di robot che useremo in questo progetto. Ciò che ci serve è un dispositivo che possa muoversi nell'ambiente circostante e che sia in grado di ricavarne la configurazione.

1.1 Le basi mobili

Dei robot con caratteristiche piuttosto semplici sono le basi mobili; esse sono dei dispositivi in grado di muoversi, che sia ricevendo dei comandi dall'utente o in modo autonomo, per portare a termine un obiettivo. In genere hanno con sé un certo carico, tipicamente attuatori (dei meccanismi per interagire o modificare l'esterno, per esempio i motori delle ruote o bracci meccanici) e sensori (dei dispositivi per percepire l'ambiente); serve inoltre uno strato di software per implementare il comportamento della base, ovvero per interpretare le letture dei sensori e far agire gli attuatori di conseguenza.
Questi modelli di robot possono essere controllati dando loro dei comandi di velocità in input, che specificano una velocità lineare e una angolare combinate insieme; esse vengono interpretate dalla base mobile, che regolerà opportunamente le velocità delle ruote o dei singoli giunti. Il robot, analizzando lo stato, la velocità di tali giunti, è invece in grado di fornire al programma la sua odometria, ovvero le sue coordinate rispetto a un punto iniziale fisso.

1.2 Sensori propriocettivi

Oltre a potersi muovere, è importante anche che una base mobile possa osservare e comprendere l'ambiente esterno. Ciò è possibile con l'utilizzo di sensori montati direttamente su di essa, che possono essere di due tipi: propriocettivi o esterocettivi. I sensori propriocettivi sono in grado di misurare grandezze "proprie" del robot, cioè relative al suo interno: gli encoder, per esempio, possono misurare la velocità delle ruote.

1.2.1 Encoder

Gli encoder sono dei sensori che, in parole povere, misurano lo spostamento lineare o la rotazione di un meccanismo rispetto a un altro. Un possibile funzionamento dell'encoder consiste nel posizionarlo in prossimità di una ruota che presenta una serie di tacche bianche e nere, e fargliela osservare in un punto fisso; se l'encoder è un rilevatore di intensità luminosa, allora è in condizione di calcolare, a seconda di quanto rapidamente si alternino il bianco e il nero, la velocità angolare della ruota. Confrontando i valori dei diversi encoder, il robot può capire se si stia muovendo in linea retta o se stia sterzando, e con quale velocità complessiva.

1.2.2 Unità di misura inerziali

Un altro tipo di sensori propriocettivi sono le unità inerziali, in breve IMU, in grado di calcolare grandezze quali la velocità o l'orientazione del robot. Solitamente sono dispositivi che utilizzano giroscopi (capaci di calcolare le velocità angolari) e accelerometri (che chiaramente misurano delle accelerazioni).
Così come per gli encoder, anche le IMU possono essere fondamentali per il funzionamento delle piattaforme mobili, ma per calcolare le coordinate nello spazio del robot è meglio utilizzare dispositivi più adatti. Infatti, per ottenere una posizione a partire da velocità o accelerazioni, si rende necessario fare delle integrazioni: è inevitabile però che accadano degli errori, e nel caso di integrazioni continue essi possono crescere molto in fretta.

1.3 Sensori esterocettivi

Questa seconda tipologia di sensori svolge la funzione di misurare grandezze relative all'esterno, come per esempio la distanza della piattaforma mobile da un ostacolo.
I sensori esterocettivi possono essere attivi o passivi. I primi agiscono direttamente sull'ambiente o lo perturbano in qualche modo, come i sensori ad infrarossi, ad ultrasuoni o al laser; questi ultimi due in particolare utilizzano un principio simile: un emettitore invia un impulso, che torna indietro nel momento in cui colpisce un ostacolo, e misurando il tempo impiegato per l'andata e il ritorno si può dedurre la distanza dell'oggetto. I sensori passivi invece si limitano a raccogliere le informazioni che giungono a loro, come le telecamere RGB.

1.3.1 Telecamere RGB

Come è noto, le telecamere sfruttano il principio della camera oscura: la luce proveniente dall'esterno, se fatta passare attraverso un piccolo foro, proietta un'immagine capovolta dell'ambiente quando raggiunge una apposita superficie ricevente. Nelle telecamere la luce viene fatta passare attraverso un obiettivo, che impressionerà l'immagine su una pellicola fotografica.
Le telecamere possono essere piuttosto versatili, poiché consentono di creare fotografie di diverso tipo o evidenziare dettagli differenti per uno stesso ambiente, a seconda di come siano regolate tre grandezze:

- **sensibilità della pellicola (gain)** - una pellicola con sensibilità più alta ha bisogno di meno luce per essere impressionata, ma ciò può comportare una peggiore nitidezza del prodotto finale;
- **tempo di esposizione** - dato che una foto di qualità richiede una nitidezza alta, si può compensare il basso gain che ne consegue lasciando entrare la luce nella telecamera per un tempo maggiore; di contro, questo vorrà dire dover fotografare ambienti completamente fermi;
- **apertura della lente** - invece di dare più tempo alla luce di impressionare l'immagine sulla pellicola, si può pensare di farne entrare una quantità maggiore "aprendo" il foro d'ingresso, ma ad un singolo punto della fotografia potrebbero arrivare più raggi di luce, creando un effetto di offuscamento.

Le telecamere a colori hanno dunque bisogno di essere regolate correttamente a seconda delle esigenze, ma possono fornire potenzialmente molte informazioni sull'ambiente, e le fotografie o i video sono immediatamente comprensibili per un essere umano.
Il problema che si manifesta utilizzando questi dispositivi è che le fotocamere sono praticamente dei "misuratori" di intensità luminosa del mondo circostante tridimensionale, ma proiettano queste misurazioni su una superficie bidimensionale; ciò che si perde è quindi la percezione della profondità dello scenario ripreso. In parte, la questione può essere risolta scattando tante immagini 2D di uno stesso luogo da punti diversi, e triangolando la posizione delle telecamere si può ricostruire la scena a tre dimensioni.

1.3.2 Telecamere stereoscopiche

Un altro modo per ricostruire un'immagine 3D utilizzando le telecamere RGB è attraverso le fotocamere stereoscopiche: due normali camere sono cioè posizionate ad una distanza relativa fissa (di solito una a sinistra e l'altra vicina a destra), mentre riprendono il medesimo luogo. Trovando i pixel in comune nelle due immagini, è possibile calcolare la distanza del punto dalle due telecamere, e quindi ottenere una ricostruzione tridimensionale, dato che questa comprende anche la profondità.
Chiaramente, anche tale metodologia non è esente da problemi: dovendo effettuare il confronto tra due immagini, le camere stereo hanno bisogno di contrasti e "texture" ben riconoscibili. Un'area totalmente bianca, per fare un esempio, difficilmente sarà ricostruibile in 3D senza errori.

1.3.3 Scanner al laser

L'utilizzo di un laser non potrà certamente riprendere e fornire delle immagini del mondo intorno alla base mobile, ma ha indubbiamente diversi pregi.
Un tipico scanner al laser è formato da un emettitore e un ricevitore: come già spiegato sopra, il primo invia un raggio laser in un certo istante di tempo, che rimbalzerà sul primo ostacolo trovato sul suo cammino e verrà rilevato dal ricevitore in un istante di tempo successivo. Calcolando il tempo impiegato dal laser in quel tragitto, e conoscendo la sua velocità (ovvero quella della luce), si può dedurre la

distanza dell'oggetto dallo scanner.
Per una base mobile è necessario e utile conoscere le distanze di almeno una parte dell'ambiente, non di un singolo punto; è per tale ragione che l'emettitore può essere posizionato davanti a una superficie riflettente inclinata, facendo deflettere il raggio e ottenendo un "ventaglio" di raggi che misurano tante distanze. È possibile anche, deflettendo i laser su due piani ortogonali, avere delle ricostruzioni in 3D.
In definitiva, gli scanner al laser non solo sono molto efficaci per calcolare le distanze, ma anche estremamente precisi, e offrono un grande angolo di vista (si possono facilmente superare i 180° e arrivare anche a una visuale completa a 360°). Da ciò ne consegue che sarà proprio il laser ad essere utilizzato come sensore nel progetto di questa relazione.

Capitolo 2

Panoramica su ROS

ROS, ovvero Robot Operating System, è uno tra i più usati middleware per la robotica. È un framework completamente open source, ed è importante capire in quale contesto si sia inserito e in che modo sia strutturato.

2.1 Contesto

Nei primi sistemi robotici, ogni robot era controllato da un unico programma, il quale ne gestiva tutte le funzionalità. Esso era quindi incaricato di:

- leggere ciclicamente tutti i dati sensoriali disponibili, e sulla base di essi capire la configurazione dell'ambiente circostante;
- pianificare una serie di azioni confrontando la configurazione ottenuta con quella desiderata (cioè quella in cui è stato raggiunto l'obiettivo);
- eseguire le azioni pianificate.

È chiaro che un approccio del genere, per quanto intuitivo, non sia particolarmente efficiente o flessibile: i robot possono essere molto complessi, e un singolo malfunzionamento in un sottosistema non dovrebbe compromettere l'intera infrastruttura.
Si è iniziato allora a seguire una nuova metodologia, disaccoppiando i vari task e sottosistemi dei robot in più processi concorrenti, raggiungendo quindi una certa modularità. Per il corretto funzionamento del sistema è necessario che tutti questi programmi scambino delle informazioni l'un l'altro, e ciò può avvenire con un meccanismo di memoria condivisa o con la comunicazione attraverso messaggi. Generalmente viene utilizzata quest'ultima modalità, in quanto meno efficiente della prima ma certamente più sicura, poiché non si corrono rischi per la memoria, e scalabile.
Nel corso del tempo sono in definitiva nati diversi middleware, compreso ROS, di cui ora vedremo il funzionamento.

2.2 Caratteristiche di ROS

ROS è una sorta di strato software tra l'applicazione finale e il programma vero e proprio, cioè come una libreria fornisce un insieme di politiche e strumenti per

implementare molti servizi specifici per i robot.
La modularità è un tema centrale in ROS, e similmente ad un sistema operativo (come si deduce dal nome), esso presenta diverse funzioni, quali la definizione e gestione dei messaggi, il controllo dei processi, una struttura file system, e procedure comunemente usate già preimplementate.
ROS è caratterizzato quindi da molti punti di forza:

- è scalabile;
- è basato sulla comunicazione attraverso messaggi;
- il codice è facilmente riutilizzabile (si possono reperire pacchetti creati da altri ed implementare personalmente le funzionalità mancanti necessarie, ed eventualmente ricondividere il proprio codice);
- è indipendente dal linguaggio (generalmente sono utilizzati C++, Python e MATLAB, ma il linguaggio usato non influisce sul funzionamento di ROS; nel nostro caso useremo C++);
- supporta tool esterni (per esempio i simulatori);
- presenta una comunità attiva e collaborativa;
- il testing è semplice e immediato.

Passiamo ora a vedere come sono gestiti i programmi all'interno di ROS, premettendo che i vari progetti creati sono raggruppati in package con nomi diversi.

2.3 Nodi

Un singolo processo eseguito in ROS viene chiamato nodo, e ognuno di essi ha un nome che lo identifica univocamente. Ogni nodo si occupa di solito di una singola attività, per esempio in una base mobile potrebbe esserci un nodo per gestire lo scanner al laser, uno per i motori delle ruote, uno per la localizzazione del robot, uno per i comandi in input; pertanto ogni robot implica in genere l'avvio di più di un nodo.
I nodi possono ricevere messaggi da altri nodi e inviarne essi stessi, ma solitamente non sono a conoscenza esattamente dei loro interlocutori: semplicemente ricevono dei messaggi dall'esterno e ne inviano a chiunque sia in ascolto in quel momento.
Per avviare un nodo da terminale è disponibile il comando rosrun. Se vogliamo, ad esempio, far partire il nodo per la lettura dei comandi di un joystick, fornito dal professor Grisetti, possiamo scrivere:

```
rosrun srrg_joystick_teleop joy_teleop_node
```

Il primo parametro è il package che contiene il nodo, il secondo è il nome del nodo stesso.
Una procedura interessante e che utilizzeremo in seguito è il poter cambiare il valore dei parametri interni al nodo da linea di comando. Sempre considerando joy_teleop_node, esso presenta diversi parametri, tra cui joy_device (il file da dove

viene letto il joystick), di default con valore /dev/input/js0. Se necessario, possiamo rimappare questo parametro molto semplicemente:

```
rosrun srrg_joystick_teleop joy_teleop_node _joy_device:=/dev/input/js1
```

nel caso per esempio in cui volessimo leggere un joystick da js1.

2.4 Messaggi

I messaggi sono delle semplici strutture dati, popolate da alcuni campi tipati; essi sono definiti in una cartella chiamata msg, sempre dentro il package del progetto, in dei file .msg, dei file di testo che forniscono anche una breve descrizione dei dati presenti nei messaggi stessi. I messaggi vengono scambiati continuamente tra i nodi, e una volta che se ne è ricevuto uno è possibile accedere ai diversi campi presenti al suo interno.
In ROS sono già implementati molti tipi di messaggi comunemente utilizzati, ma se necessario è possibile definirne di nuovi, popolandoli con i dati che servono.
Similmente ai nodi, anche i messaggi hanno dei nomi che li identificano; un esempio sono quelli per i comandi di velocità, geometry_msgs/Twist: il loro nome ci dice che sono memorizzati nel file Twist.msg, nel package geometry_msgs.
Per ispezionare i dati contenuti in un tipo di messaggio, possiamo usare il seguente comando da terminale:

```
rosmsg show geometry_msgs/Twist
```

In tal caso ci verrà restituito:

```
geometry_msgs/Vector3 linear
float64 x
float64 y
float64 z
geometry_msgs/Vector3 angular
float64 x
float64 y
float64 z
```

Ne deduciamo allora che Twist è un messaggio che contiene due vettori di tre coordinate nello spazio (dei float), uno per la velocità lineare e uno per la velocità angolare.

2.5 Topic

I topic sono dei canali di comunicazione dove vengono trasmessi i messaggi. Come detto in precedenza, i nodi non sono a conoscenza dei processi con cui stanno comunicando, e questa politica di anonimità, la quale separa completamente chi riceve l'informazione da come viene prodotta, viene garantita proprio dai topic. I nodi, infatti, conoscendo unicamente il nome di un topic, potranno pubblicare su di

esso se vogliono utilizzarlo per inviare dei messaggi, mentre potranno effettuarne l'iscrizione se vogliono riceverli. Si ha quindi un meccanismo publisher-subscriber, in cui i processi alle estremità del topic si limitano ad interfacciarsi con esso, senza sapere chi ci sia dall'altra parte.
I topic hanno anche un tipo: questo dipende dal tipo del messaggio che viene trasmesso, per esempio un topic dedicato ai comandi di velocità appena visti sarà un topic di tipo geometry_msgs/Twist.

2.6 ROS core

Il roscore è un insieme di processi che sono indispensabili per la comunicazione tra nodi, e più in generale per il funzionamento dell'infrastruttura ROS. Quando i nodi vogliono pubblicare o iscriversi ad un topic, essi avvertiranno il roscore, che provvederà a connetterli l'un l'altro. A connessione stabilita, i processi comunicano direttamente (sempre interfacciandosi con i topic), senza l'ausilio del core.
Il roscore è dunque una sorta di server per tutto il sistema ROS, ed è necessario avviarlo prima di qualsiasi nodo con il comando roscore.
Da questo momento possiamo tranquillamente svolgere qualsiasi azione consentita dall'infrastruttura ROS. Dobbiamo però anche sapere che il core è in realtà un "aggregato" di tre importanti servizi:

- **il ROS Master**: è la parte che tiene traccia dei nodi publisher e subscriber e li mette in comunicazione sui topic;
- **il ROS Parameter Server**: è un server accessibile a tutti i nodi che memorizza eventuali parametri da loro usati o modificati;
- **il nodo di logging rosout**: si tratta di un nodo il cui compito è pubblicare messaggi di tipo rosgraph_msgs/Log sul topic /rosout, da cui ottenere informazioni di log.

2.7 Il filesystem ROS

Abbiamo quindi appreso che, dopo aver avviato roscore, i programmi ROS, chiamati nodi, comunicano tra loro con dei messaggi attraverso dei topic.
Prima si è accennato al fatto che i progetti ROS siano raggruppati in dei package. Nello specifico, abbiamo innanzitutto una cartella di workspace, in cui effettuare tutte le operazioni ROS e memorizzare i pacchetti; più precisamente, nel workspace troviamo almeno le seguenti tre directory:

- **build** - contiene i prodotti intermedi delle compilazioni (per esempio i file oggetto .o);
- **devel** - racchiude i file eseguibili veri e propri;
- **src** - contiene i codici sorgente suddivisi per package.

La creazione e configurazione del workspace è compito di Catkin, il build system ufficiale di ROS: può svolgere dunque anche altri compiti, come anche la compilazione

di tutto il workspace o dei package desiderati.
All'interno della cartella src sono quindi creati diversi pacchetti; essi contengono i codici sorgente dei nodi e altri file utili, come eventuali messaggi definiti dal programmatore e i file CMakeLists.txt, dei file generati da Catkin che specificano la configurazione del package, utili per la compilazione.

2.8 Sensori e trasformate

Vediamo ora un ultimo concetto che useremo in ROS, ovvero le trasformate. Su di una base mobile possono essere montati più sensori, e generalmente ognuno viene gestito da un nodo; dunque avviando il nodo, si avvia anche il relativo sensore. In ROS sono già definiti diversi messaggi standard per i sensori più comuni nel package sensor_msgs: nel caso dei laser si possono usare i messaggi sensor_msgs/LaserScan, tra i cui campi troviamo l'angolo minimo e massimo del sensore in radianti, la distanza massima percepibile in metri, e l'incremento in radianti dato dal singolo raggio laser (poiché abbiamo detto che gli scanner al laser sono costituiti da un "ventaglio" di raggi, e quindi ogni raggio scansionerà una piccola frazione di angolo).
Il problema che si pone è che spesso è necessario sapere in che modo siano montati i dispositivi sul robot. Sia la base mobile che i sensori hanno un loro sistema di riferimento (noto anche come reference frame), ognuno con l'origine in un punto diverso dello spazio; i punti rilevati da un laser avranno allora delle coordinate espresse rispetto al suo sistema di riferimento, ma se volessimo esplicitare questi punti rispetto al frame del robot dovremmo usare delle coordinate differenti, a seconda di come sia stato posizionato il sensore.
Ogni reference frame è rappresentato in pratica da una terna di assi (x,y,z) nello spazio, e il concetto di trasformata ci permette di calcolare come siano posti l'uno rispetto all'altro e di "spostarci" facilmente tra i sistemi di riferimento. Le trasformate vengono pubblicate di continuo su un topic /tf, e i diversi frame in una base mobile vengono strutturati in una configurazione ad albero. Esiste un nodo apposito per visualizzare graficamente l'albero delle trasformate di un robot:

```
rosrun tf view_frames
```

Se prendiamo come esempio il robot presente nel simulatore Stage (di cui parleremo a breve), esso ha una struttura molto semplice, quella in figura 2.1. base_laser_link è il nome del frame dello scanner al laser, base_link è il riferimento situato a metà fra le ruote, base_footprint rappresenta "l'ombra" del robot, mentre odom è il frame fisso dell'odometria: rimanendo fermo può facilmente calcolare le coordinate del robot nello spazio, mentre gli altri frame si muovo in modo solidale. Possiamo comprendere meglio questi concetti usando RViz, un tool di visualizzazione 3D.
Osservando l'immagine 2.2, notiamo tre diversi reference frame: questo perché nel caso in esame, base_link e base_footprint sono coincidenti. In ogni terna l'asse delle x è rappresentato dall'asse rosso, quello delle y dall'asse verde, e quello delle z dall'asse blu. Possiamo vedere che il frame del laser si trova più in alto e più avanti rispetto al riferimento del robot; in questo caso particolare il robot è anche ruotato rispetto all'odometria, che come abbiamo detto è un frame che rimane immobile, mentre gli altri sulla base mobile si muovono insieme.

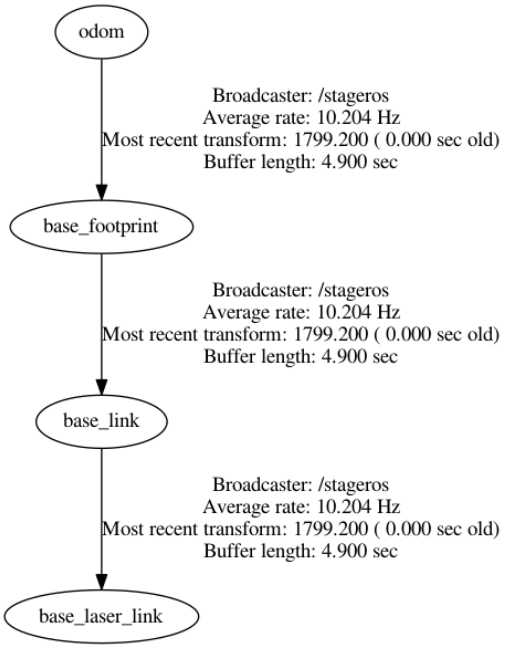

Figura 2.1. Esempio di albero delle trasformate

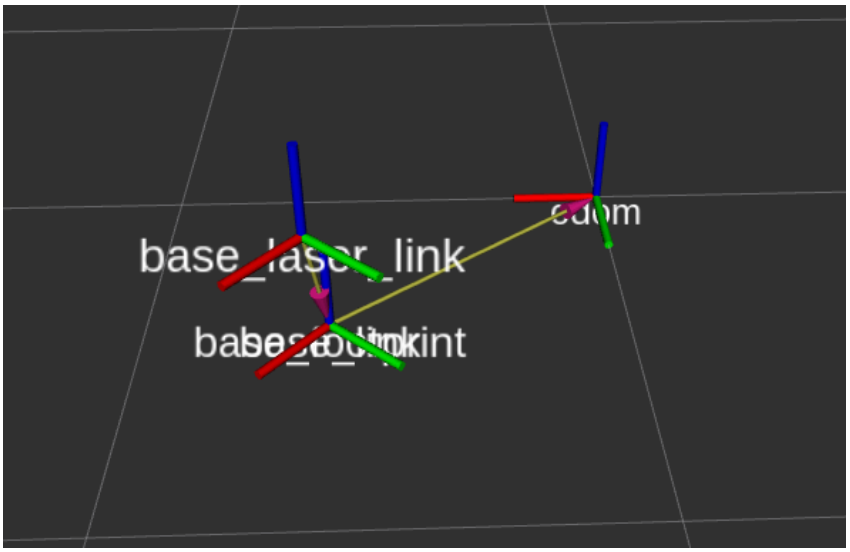

Figura 2.2. Configurazione dei reference frame in RViz

2.9 Tool utilizzati

Un aspetto di ROS nominato in precedenza è il supporto di diversi tool, specifici per molte funzioni. In questo progetto sono stati usati Stage, un simulatore, e rqt_graph, un tool per visualizzare in che modo stiano comunicando i nodi attivi.

2.9.1 Stage

Per provare i propri programmi è molto comodo avere un simulatore: in questa maniera possiamo effettuare il testing in modo semplice e sicuro, senza "scomodare" eventualmente il robot vero e proprio. A livello di programmazione ed esecuzione di nodi, non cambia molto tra l'utilizzare un simulatore o il robot: l'unica differenza sta nell'avviare il nodo del simulatore nel primo caso, o avviare i nodi del robot altrimenti (sensori, motori, ecc.).
Stage è un semplice simulatore 2D che, a partire da un file .world contenente la configurazione di un mondo, permette di visualizzarlo come una mappa con una base mobile presente in essa. Per avviarlo dobbiamo prima di tutto andare nella cartella di Stage (usando il comando roscd, il quale ci consente di spostarci tra package ROS), e poi, utilizzando ad esempio come mondo willow-erratic.world (figura 2.3), basta eseguirlo come qualsiasi nodo:

```
roscd stage_ros
rosrun stage_ros stageros world/willow-erratic.world
```

Figura 2.3. willow-erratic.world simulato in Stage

Supponiamo allora di aver avviato roscore e stageros; innanzitutto, utilizzando il comando

```
rosnode list
```

potremo avere una lista di tutti i nodi attualmente in esecuzione. È chiaro che al momento il terminale ci restituirà solo i due appena avviati:

```
/rosout
/stageros
```

Per una simulazione funzionante, è necessario sapere a quali topic sia iscritto /stageros. Possiamo, similmente ai nodi, visualizzare un elenco dei topic attivi:

```
rostopic list
```

oppure più semplicemente digitare:

```
rosnode info /stageros
```

Così facendo ci verranno restituiti, fra diverse informazioni, i topic a cui /stageros si è iscritto e su quali sta pubblicando messaggi. Ce ne sono diversi, ma per il momento ci interessa sapere che il processo si iscrive al topic /cmd_vel, sul quale riceve i comandi di velocità per il robot. Servirà quindi che l'eventuale nodo atto ad inviare le velocità al robot pubblichi sul medesimo topic.

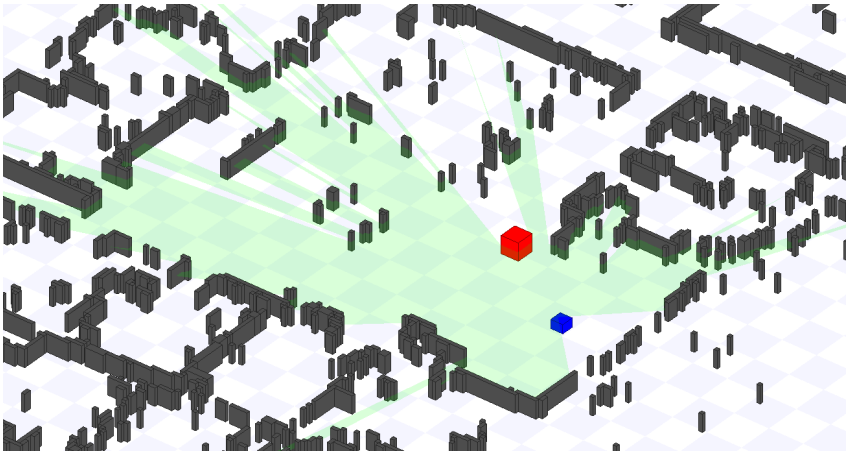

Figura 2.4. Visuale alternativa in Stage; notiamo il robot in blu, un ostacolo mobile in rosso, e lo scanner al laser con l'area in verde

2.9.2 rqt_graph

Abbiamo visto che, per comprendere quali nodi siano attivi e come stiano comunicando, sono disponibili diversi comandi da terminale, quali list e info.
A volte è molto utile, soprattutto nel caso di infrastrutture più complesse, avere una rappresentazione grafica dei processi in esecuzione e dei topic esistenti fra loro. rqt_graph è un tool che adempie proprio a tale scopo, e possiamo avviarlo con il seguente comando:

```
rosrun rqt_graph rqt_graph
```

Ipotizziamo che siano attivi due processi già visti, ovvero Stage e il programma di lettura del joystick; una volta fatto partire rqt_graph, questo analizzerà brevemente la situazione e restituirà un grafo di nodi, come in figura 2.5.
Nel grafo si possono notare i nodi rappresentati da degli ellissi, mentre i rettangoli

identificano i topic; viene anche specificata la loro direzione, potendo così capire chi sia il publisher e chi il subscriber. Il caso appena preso in esame è molto semplice, ma appare evidente come /joy_control pubblichi su /cmd_vel, mentre /stageros ci si sia iscritto: essendo entrambi i nodi interlocutori sullo stesso topic, sono correttamente in comunicazione l'un l'altro, e quindi è possibile inviare comandi di velocità con un joystick a Stage, che simulerà gli spostamenti del robot nel suo mondo.

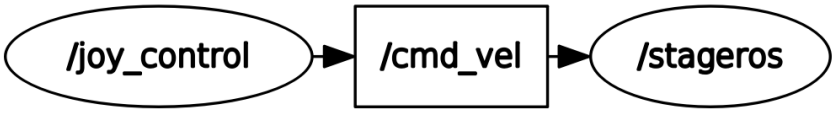

Figura 2.5. Grafo di nodi generato con rqt_graph

Capitolo 3

Navigazione

La navigazione è un insieme di procedure al fine di far muovere autonomamente una base mobile, cioè farle fare dei percorsi senza l'ausilio di un utente che fornisca comandi di velocità. È in questo contesto che si inserisce la collision avoidance.

3.1 Concetti della navigazione

Quando si tratta di navigazione, abbiamo a che fare con un caso di agente autonomo: un robot che cerca di portare a termine un compito da solo muovendosi, evitando eventuali ostacoli che possano trovarsi sul suo cammino. Per fare ciò, una piattaforma mobile autonoma ha bisogno di conoscere l'ambiente circostante, dove si trova in tale ambiente, pianificare il cammino da percorrere, ed eseguire le azioni opportune. Si rendono allora necessarie tre grandezze:

- una mappa del luogo;
- la posizione del robot sulla mappa;
- il percorso da seguire.

3.1.1 Mappa

La mappa è una rappresentazione dell'ambiente in cui deve operare il robot. Ne esistono di diverse tipologie (si possono avere mappe più "tradizionali", per esempio con aree percorribili e muri, fino ad avere un ambiente definito come insieme di nodi e archi in un grafo), dipendentemente dal tipo di sensori che sono stati montati sulla piattaforma mobile e da quali funzioni essa debba svolgere. È importante che la mappa contenga abbastanza informazioni per dedurre dove sia posizionato il robot e che percorsi possa seguire.

3.1.2 Posizione

La posa del robot è un concetto strettamente legato alla mappa che è stata costruita: uno stesso punto potrebbe apparire in due mappe diverse, ma avere coordinate differenti. Generalmente la posizione del robot è espressa con delle coordinate rispetto all'origine della mappa, eventualmente anche con la sua angolazione.

3.1.3 Percorso

Il percorso è un qualsiasi cammino che il robot può percorrere. Dato un punto di partenza e uno di arrivo, lo si può definire come una sequenza di stati, di azioni consecutive per portare a termine l'obiettivo assegnato.

3.2 Determinazione dei dati

Per la navigazione è importante quindi che il robot conosca tutti e tre i concetti appena introdotti, e nel caso così non fosse sono possibili delle procedure specifiche:

- **mapping** - costruire la mappa utilizzando i dati sensoriali del robot mentre effettua un certo percorso e conoscendo la sua posizione nel mondo;
- **localizzazione** - determinare la posa del robot facendo un riscontro continuo tra le rilevazioni dei sensori e le informazioni contenute nella mappa;
- **path planning** - determinare il percorso migliore tra due punti qualsiasi evitando gli ostacoli, conoscendo la mappa e la posizione in essa.

I procedimenti descritti suppongono che, in mancanza di una delle tre grandezze, questa possa essere determinata a patto che si sappiano esattamente le due rimanenti. Può accadere però che ne sia nota addirittura solo una, ed è chiaro che bisogna usare degli approcci diversi:

- **SLAM** (Simultaneous Localization And Mapping) - mentre la base mobile percorre un cammino certo, costruire la mappa stimando insieme la posizione;
- **localizzazione attiva** - avendo una mappa, cercare di andare nel punto di arrivo usando dei riferimenti dell'ambiente per essere localizzati il più correttamente possibile;
- **esplorazione** - conoscendo la posizione, dedurre la mappa dai sensori nel percorso migliore possibile.

3.3 Stack di navigazione e prevenzione di collisioni

È chiaro a questo punto che per la corretta navigazione di un robot autonomo necessitiamo di una mappa (costruita in caso con i metodi visti), un modulo di localizzazione che stimi di continuo la posizione usando i sensori (si potrebbe pensare di sfruttare l'odometria, ma genererà per forza degli errori che possono accumularsi), e un modulo di path planning, il quale calcolerà il passo successivo a seconda della posa stimata.
Nonostante il modello descritto possa sembrare adatto alla navigazione, in uno scenario reale non è garantito sia così. L'ambiente potrebbe infatti cambiare nel tempo, e ciò può portare il robot ad effettuare decisioni sulla base di una mappa ormai obsoleta: per esempio, esso potrebbe scontrarsi con un ostacolo che prima non era presente, e che quindi non è segnato sulla mappa. Abbiamo detto pure che la localizzazione dipende dai dati sensoriali, ma anche qui potrebbero avvenire

degli errori di stima: è abbastanza intuitivo capire che, prendendo un luogo sempre uguale e senza punti di riferimento particolari, è difficile capire in che punto ci si trovi. Si è manifestato allora il bisogno di creare un modello che utilizzi sì mappa, localizzazione e path planning, ma che garantisca anche un livello di affidabilità e sicurezza superiore: ciò è possibile in un sistema che utilizzi i sensori per rilevare cambiamenti rispetto alle informazioni date dalla mappa (nuovi ostacoli, persone in movimento, porte aperte o chiuse), e che implementi degli algoritmi di obstacle detection e avoidance per evitare scontri correggendo la traiettoria del robot.

Introduciamo dunque un modello di navigazione reattivo, che prenda delle decisioni in anticipo ma sia in grado di cambiarle a seconda dei dati sensoriali, e che possa all'eventualità fermarsi o correggere i comandi inviati ai motori della base mobile: il navigation stack, la cui struttura è schematizzata in figura 3.1.
Lo stack di navigazione è un insieme di nodi, topic e procedure, con lo scopo di far muovere in modo autonomo un robot senza farlo perdere o scontrare: esso usa diversi dati, quali locazione attuale, locazione desiderata, odometria (data da encoder o simili) e i rilevamenti dei sensori, e fornisce in output un comando di velocità da inviare alla base mobile.
Partiamo dalla mappa: essa viene fornita da un nodo che la pubblica continuamente, map_server, in modo che più robot in contemporanea possano accedervi (ricordiamo che ROS è scalabile). Abbiamo poi dei nodi dipendenti direttamente dal robot che si sta usando, come il nodo che fornisce l'odometria (in genere il nodo "centrale" della base mobile), i nodi dei sensori e le loro trasformate (dobbiamo sapere in che maniera siano orientati e posizionati). Una volta che si hanno tutti questi dati, il nodo amcl provvede alla localizzazione.

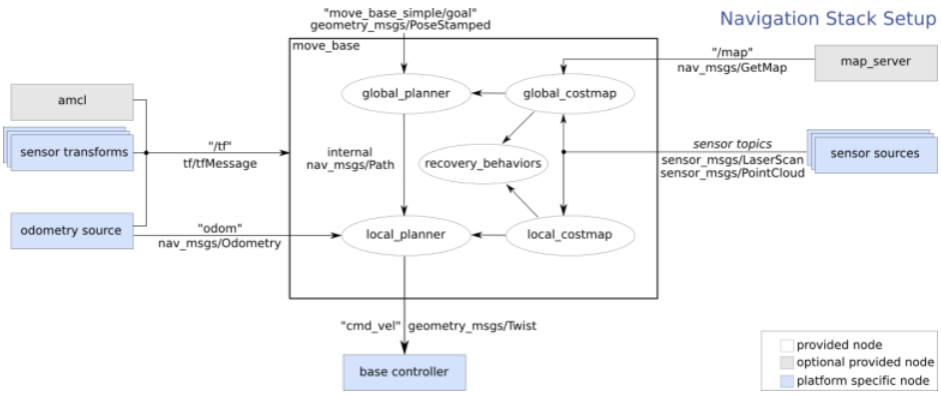

Figura 3.1. Struttura dello stack di navigazione

Ci manca solo il path planning, compito del nodo move_base, che prende in input tutte le informazioni descritte, insieme alle coordinate da raggiungere (il goal). Questo nodo è composto da diversi moduli, di cui quello per noi di maggiore interesse è il planner, suddiviso in global planner e local planner.
Il compito del global planner è eseguire un algoritmo per trovare il percorso a "costo" minore dalle coordinate attuali a quelle di arrivo. Il nodo si occupa solamente di trovare la sequenza di posizioni più efficiente per raggiungere il goal, ma si basa solo sulle informazioni della mappa (cioè non tiene conto di nuovi ostacoli) e ignora i

vincoli di movimento della base mobile (parliamo proprio della struttura sua e degli attuatori: non è detto ad esempio che il dispositivo sappia spostarsi lateralmente, e quindi potrebbe aver bisogno di sterzare o fermarsi del tutto e girare). Il local planner sopperisce a questa mancanza, facendo correzioni locali della traiettoria globale tenendo conto delle possibili azioni eseguibili dal robot.
Si capisce quindi che è qui che si inseriscono gli algoritmi di collision avoidance. Una volta deciso un path globale, è fondamentale pianificare continue azioni e correzioni locali, per evitare collisioni impreviste e tenere in considerazione i possibili movimenti della base mobile.

Capitolo 4

Approccio di collision avoidance a basse risorse computazionali

Abbiamo dunque compreso cosa siano mapping, localizzazione e path planning. Cerchiamo allora di addentrarci in questo ultimo ambito con una possibile implementazione di un algoritmo di collision avoidance, il quale cerca di mantenere una certa robustezza ma senza utilizzare troppe risorse.

4.1 Struttura del nodo ROS

Partiamo innanzitutto da una situazione già analizzata in precedenza. Poiché testeremo il programma con Stage, e per farlo vorremo controllare il robot con un joystick, possiamo usare i nodi /joy_control e /stageros: come già visto, il primo invierà al secondo delle velocità, ovvero messaggi di tipo geometry_msgs/Twist, sul topic cmd_vel. La base mobile si muoverà di conseguenza.
Ciò che vogliamo realizzare è una sorta di nodo "controllore", ovvero un programma che:

- riceva dei comandi di velocità in ingresso;
- analizzi le letture dello scanner al laser del robot;
- sulla base dell'analisi fatta invii alla base mobile dei nuovi comandi di velocità, ottenuti come correzioni di quelli ricevuti in input.

Abbiamo allora un nodo che in maniera simile a un local planner sfrutta i dati sensoriali per effettuare degli aggiustamenti di traiettoria, e otteniamo un modello di comunicazione tra nodi come quello in figura 4.1. In particolare, dato che il

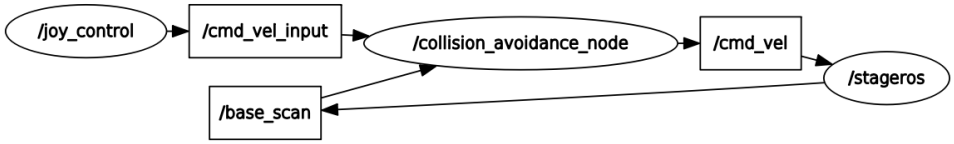

Figura 4.1. Modello di comunicazione tra /collision_avoidance_node e gli altri nodi

robot riceve i comandi dal topic /cmd_vel, è necessario che il nostro nodo, chiamato /collision_avoidance_node, pubblichi su esso. Al programma servono però i dati sensoriali, che vengono pubblicati continuamente sul topic /base_scan, su cui circolano dei messaggi di tipo sensor_msgs/LaserScan: pertanto, /collision_avoidance_node ne effettua l'iscrizione. Mancano infine le velocità in input, dunque il nodo di obstacle avoidance creerà un topic apposito (/cmd_vel_input) al quale si iscrive; se eseguiamo /joy_control (o in generale un nodo che invii delle velocità) rinominando da linea di comando il suo topic di pubblicazione (con la procedura che abbiamo visto nella Sezione 2.3) proprio in /cmd_vel_input, ecco che avremo la corretta comunicazione tra tutti i processi.
L'implementazione delle funzionalità del nodo è nel file collision_avoidance_node.cpp, interno al package collision_avoidance, e questa prima parte di gestione dei topic avviene all'interno del main:

```
int main(int argc, char** argv) {

  ros::init(argc, argv, "collision_avoidance_node");

  ros::NodeHandle nh("~");
  string input_vel_topic;
  string laser_topic;
  string output_vel_topic;

  nh.param("input_vel_topic", input_vel_topic,
    string("/cmd_vel_input"));
  nh.param("laser_topic", laser_topic, string("/base_scan"));
  nh.param("output_vel_topic", output_vel_topic,
    string("/cmd_vel"));

  ...

  ros::Subscriber input_vel_sub = nh.subscribe(input_vel_topic,
    10, inputVelCallback);
  ros::Subscriber laser_sub = nh.subscribe(laser_topic,
    10, laserCallback);

  output_vel_pub = nh.advertise<geometry_msgs::Twist>
    (output_vel_topic, 10);

  ...

}
```

La funzione ros::init inizializza il nodo con il nome specificato, e si può notare poi che i nomi dei tre topic vengono inizializzati come parametri (che saranno salvati nel parameter server) dalla funzione param, così da poterli eventualmente cambiare da linea di comando a seconda delle esigenze: non è per esempio detto che tutte le basi mobili pubblichino i dati sensoriali su un topic chiamato /base_scan, e quindi

il poter rinominare i canali di comunicazione ci lascia più libertà.
La funzione subscribe effettua l'iscrizione al topic specificato, e prende in input una funzione di callback, implementata nel programma, che viene chiamata automaticamente quando si riceve un messaggio; questo messaggio viene poi preso in input dalla callback stessa. Dato che il nodo si iscrive a due topic abbiamo bisogno di due callback, chiamate inputVelCallback e laserCallback. Infine advertise comunica al ROS core che il nodo ha iniziato a fare da publisher.

Sempre nel main vengono inizializzate anche due variabili globali:

```
int main(int argc, char** argv) {
  ...
  twist.linear.x = 0;
  twist.linear.y = 0;
  twist.linear.z = 0;
  twist.angular.x = 0;
  twist.angular.y = 0;
  twist.angular.z = 0;

  listener = new tf::TransformListener;
  ...
}
```

Esse sono state dichiarate come variabili globali perché dovranno essere accessibili anche alle funzioni di callback. La variabile twist, un messaggio di tipo geometry_msgs/Twist, servirà per salvare il comando ricevuto dal joystick, azione eseguita da inputVelCallback, e sarà poi inviato al robot con le dovute correzioni da laserCallback; listener invece è un TransformListener, un tipo di dato che si mette in ascolto delle trasformate pubblicate sul topic /tf; ciò vuol dire che la struttura completa di comunicazione è quella in figura 4.2. Il TransformListener servirà alla callback del laser, perché essa avrà bisogno di calcolare i punti rilevati dallo scan rispetto al reference frame del robot, quando normalmente sono espressi rispetto al riferimento del sensore.

Vediamo allora inputVelCallback, che appunto salva nella variabile twist le velocità ricevute in ingresso:

```
void inputVelCallback(const geometry_msgs::Twist::ConstPtr& msg)
{
  float x = msg->linear.x;
  float z = msg->angular.z;
  ...
  twist.linear.x = x;
  twist.angular.z = z;
}
```

Ricordiamo che i messaggi di tipo Twist sono formati da due vettori nelle tre dimensioni (x,y,z), uno per la velocità lineare e uno per la velocità angolare. In realtà, per la velocità lineare ci interessa solo la componente x, mentre per quella angolare solo la z: per come sono configurate le ruote, possiamo pensare alla base

mobile come a un'automobile, cioè un dispositivo che è in grado solo di andare avanti e indietro (spostamenti rispetto le x) e girarsi verso destra e sinistra (velocità angolare espressa rispetto le z).
L'altra funzione di callback, laserCallback, si occupa invece della prevenzione delle collisioni.

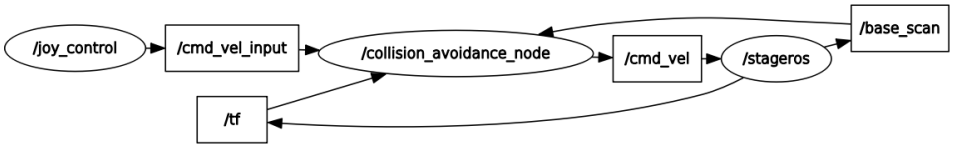

Figura 4.2. Modello di comunicazione includendo /tf

4.2 Costruzione dell'algoritmo

Vediamo come è stato pensato l'algoritmo di collision avoidance.
L'informazione principale che abbiamo a disposizione sono i diversi punti rilevati dal laser, dei quali possiamo ricavare l'angolo e la distanza rispetto al robot; l'angolo è calcolato a partire dall'asse x del reference frame della base mobile, quindi un punto esattamente davanti al robot avrà un angolo di 0°, i punti a destra avranno un angolo negativo, e quelli a sinistra uno positivo.
Innanzitutto possiamo definire una distanza di soglia oltre la quale gli ostacoli rilevati non vengono considerati: un punto molto lontano non costituisce infatti una minaccia per il robot, e non vogliamo che ne influenzi il comportamento; inoltre il programma sarà più efficiente se deve occuparsi solo di un sottoinsieme dei punti rilevati. Consideriamo dunque solo le rilevazioni sotto una distanza chiamata THRESH_RANGE.
A questo punto l'approccio che si è deciso di seguire è quello di trattare il robot come se fosse un punto nello spazio, e pensare ai singoli punti degli ostacoli rilevati sotto la distanza di soglia come se applicassero una forza repulsiva verso di esso. I contributi dei diversi punti andranno considerati come applicati direttamente sulla base mobile, scomposti sulle x e sulle y del suo reference frame, e le componenti sommate, dandoci così una forza repulsiva totale frontale (sulle x) e una laterale (sulle y) applicate al robot; in tal modo il robot cambierà traiettoria e verrà allontanato da qualsiasi ostacolo. La forza repulsiva dovrà dipendere da quanto siano vicini l'ostacolo e la base mobile, perché un punto in prossimità dovrebbe respingere via il robot maggiormente di uno che si trova più lontano.
Per il calcolo di queste forze F_x e F_y si è deciso di usare le seguenti formule:

$$F_x = -\frac{v}{SCALE} \sum_i \left(\frac{1}{x_i^2} \cos(\vartheta_i)\right)$$

$$F_y = -\frac{v}{SCALE} \sum_i \left(\frac{1}{x_i^2} \sin(\vartheta_i)\right)$$

- x_i è la distanza fra robot e punto i-esimo tra quelli rilevati sotto la soglia THRESH_RANGE: il nostro scopo è che non avvengano mai collisioni, e con

la frazione $\frac{1}{x_i^2}$ otteniamo una forza che cresce in fretta per distanze sempre più brevi fino a tendere idealmente ad infinito, facendo evitare qualsiasi contatto tra ostacolo e base mobile; i singoli contributi vanno poi sommati per ottenere la forza risultante;

- ϑ_i è l'angolo tra robot e punto i-esimo;
- v è la velocità lineare del robot, poiché all'aumentare di essa anche la forza repulsiva deve aumentare proporzionalmente;
- $SCALE$ è una costante con lo scopo di diminuire il modulo della forza totale e renderla di dimensioni comparabili alla velocità lineare usuale della base mobile, perché la somma avviene fra tanti punti e c'è il rischio di ottenere una forza molto elevata che non rende facilmente manovrabile il robot. Una $SCALE$ bassa garantisce una forza maggiore e quindi una migliore prevenzione delle collisioni, ma il dispositivo potrebbe non passare attraverso le porte o in luoghi stretti; al contrario con una $SCALE$ alta il robot può passare agilmente in più luoghi, ma non si ha lo stesso livello di sicurezza.

Le forze inoltre sono negative perché repulsive.

Si pone però un problema: il metodo visto suppone che la base mobile sia un punto nello spazio in grado di muoversi liberamente nel piano delle coordinate (x,y), ma fra i nostri requisiti abbiamo imposto quello di considerare i vincoli cinematici del dispositivo. L'algoritmo funzionerebbe tranquillamente nel caso ideale analizzato, nel quale la base mobile si muove leggendo i campi linear.x e linear.y dei comandi di velocità Twist, e in cui per effettuare la correzione di traiettoria basterebbe che il programma sommasse ad essi i valori rispettivamente di F_x e F_y.
Nello scenario reale abbiamo detto di dover trattare la base mobile similmente ad un'automobile, che quindi non può eseguire spostamenti laterali, cioè sull'asse delle y. Dato che, come visto in inputVelCallback, i campi che ci interessano delle velocità sono linear.x e angular.z, si potrebbe pensare semplicemente di leggerli dai comandi in ingresso e correggerli sommando il valore di F_x a linear.x e di F_y a angular.z. Questo procedimento è stato testato con il programma, e mentre è esatto per la prima velocità, perché è come per il caso ideale, non lo è per la seconda: lo spostamento verso i lati dato da una sterzata non è "immediato" come quello causato da una velocità laterale, e infatti non è stato raro che la base mobile si scontrasse con degli ostacoli posti davanti perché non riusciva a girare abbastanza in fretta.
Un primo tentativo di risolvere questo problema è stato, invece di utilizzare una sola costante di ridimensionamento $SCALE$ per entrambe le formule, di usarne due diverse, una per la correzione lineare e una per la correzione angolare, così da dare più "peso" alla seconda e far sterzare più velocemente il robot. Ciò non è bastato per due motivi:

- gli ostacoli particolarmente vicini ai lati della base mobile generavano sterzate troppo brusche: questo succedeva perché i punti rilevati ai lati hanno un angolo in modulo vicino a 90°, cioè una componente della forza F_y dipendente dal seno dell'angolo elevata;

- il robot non riusciva ad evitare ostacoli posti davanti in mancanza di rilevazioni di punti ai lati: ciò è spiegato da un motivo simile, cioè che gli oggetti di fronte hanno un angolo prossimo agli 0°, ovvero una correzione di sterzata dipendente sempre dal seno quasi nulla, per cui il robot rallentava ma iniziava a girare comunque troppo tardi, scontrandosi.

Ad ogni modo, in gran parte dei casi questo approccio di far respingere il robot dall'ambiente circostante sembrava avere buone premesse, poiché veniva allontanato automaticamente dagli oggetti troppo vicini con un semplice calcolo, i punti rilevati ai lati lo facevano deviare bene, e la velocità lineare rimaneva abbastanza costante, evitando brusche decelerazioni e facendo quasi fermare del tutto la piattaforma mobile solo quando strettamente necessario.
Il problema principale è dunque riuscire a far girare il robot abbastanza presto quando vengono trovati pochi punti di riferimento con angoli piccoli. Alla fine, dopo diversi test per cercare di dare più "potere" alla forza di sterzata, si è deciso di usare una formula leggermente diversa per la correzione di velocità rotazionale:

$$F_x = -\frac{v}{SCALE} \sum_i \left(\frac{1}{x_i^2} \cos(\vartheta_i)\right)$$

$$F_y = -\frac{v}{SCALE} \sum_i \left(\frac{1}{x_i^4} \sin(\vartheta_i)\right)$$

Analizziamo meglio la situazione con il grafico in figura 4.3: in esso è rappresentata la correzione di velocità angolare data da un singolo punto rilevato dal laser, dipendente dalla distanza e scalata da $SCALE$ (che è piuttosto alta, nel programma è stato usato un valore di 1500); ignoriamo per il momento la velocità del robot e l'angolo del punto. Si può osservare che per entrambe le formule la forza tende ovviamente a infinito al diminuire della distanza, ma nel caso della nuova formula con $\frac{1}{x^4}$ il grafico

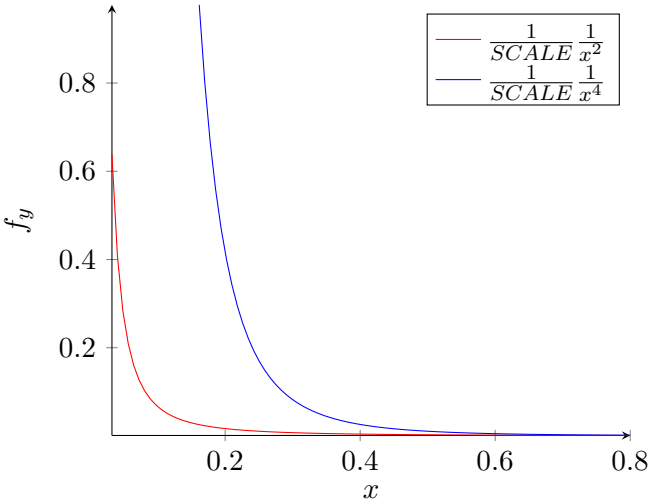

Figura 4.3. Confronto tra formule per la correzione angolare in funzione della distanza

"si alza" prima e più in fretta. Punti che prima non erano abbastanza vicini da dare un contributo significativo alla rotazione del robot (soprattutto quelli di fronte)

adesso hanno la possibilità di farlo ruotare, facendolo deviare in modo sicuro; i punti che invece sono più lontani continuano ad avere un'influenza bassa. Interessante notare che con questo approccio si è riuscito a mantenere la stessa costante $SCALE$ sia per F_x che F_y.

4.3 Implementazione dell'algoritmo

Non ci rimane altro da fare che analizzare la parte fondamentale del programma, ovvero l'altra funzione di callback, lasercallback, in cui viene implementata l'obstacle avoidance.
Per prima cosa possiamo notare che, nell'eventualità in cui il robot sia fermo, cioè non venga dato nessun comando, non c'è bisogno di fare nulla, perché siamo certi che non potranno esserci collisioni. In tale situazione, laserCallback si limita semplicemente ad inoltrare al robot le velocità ricevute e salvate in twist (i cui campi saranno tutti zeri), per continuare a garantire una continua comunicazione tra i nodi.
Se invece ci si sta muovendo, è necessario analizzare i dati sensoriali. Intanto vengono inizializzate due variabili, le correzioni di velocità F_x e F_y viste sopra:

```
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  ...
  float linear_correction = 0, angular_correction = 0;
  ...
}
```

Dobbiamo prepararci a scansionare i punti rilevati e calcolare le loro coordinate rispetto al reference frame del robot, quindi ci mettiamo in ascolto delle trasformate usando la variabile listener:

```
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  ...
  std::string error_msg;
  if (listener->canTransform(robot_frame_id,
    msg->header.frame_id, msg->header.stamp, &error_msg))
  listener->lookupTransform(robot_frame_id,
    msg->header.frame_id, msg->header.stamp, transform);
  ...
}
```

Il primo argomento di entrambe le funzioni contiene il sistema di riferimento della base mobile, il secondo quello del sensore che ha generato il messaggio ricevuto, ovvero il laser scan. La trasformata viene salvata nella variabile transform.

Dobbiamo ora analizzare i punti rilevati: il messaggio ricevuto dallo scanner, di tipo sensor_msgs/LaserScan, contiene un vettore ranges, in cui sono salvate le misurazioni di distanza di tutti i raggi del laser, e angle_increment, l'incremento in angolo portato da ognuno di essi. Possiamo allora scrivere un ciclo for che scansioni

ogni valore in ranges e lo usi per calcolare distanza e angolo (per il momento ancora rispetto al sensore):

```
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  ...
  for (int i=0; i<msg->ranges.size(); ++i) {
    float range = msg->ranges[i];
    float angle = msg->angle_min + i*msg->angle_increment;
    ...
  }
  ...
}
```

Adesso possiamo finalmente usare le trasformate e calcolare le coordinate del punto attualmente in esame dal ciclo for rispetto al frame del robot. Le coordinate saranno salvate in delle variabili x e y, mentre range ed angle conterranno i nuovi valori della distanza e dell'angolo del punto sempre in riferimento alla base mobile.

Fatto ciò, ancora nel ciclo for calcoliamo le due correzioni di velocità relative al singolo punto con le formule viste:

```
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  ...
  for (int i=0; i<msg->ranges.size(); ++i) {
    ...
    if (range < THRESH_RANGE && (x * twist.linear.x > 0)) {
      ...
      float force_x = (1/pow(range,2))*cos(angle);
      float force_y = (1/pow(range,4))*sin(angle);
      linear_correction += force_x;
      angular_correction += force_y;
    }
  }
}
```

Parliamo innanzitutto dell'if. Il primo controllo che effettua è che il punto trovato sia ad una distanza inferiore a THRESH_RANGE, e tale controllo si è potuto fare solo ora perché abbiamo prima dovuto esprimere la distanza rispetto al robot. La seconda condizione impone invece che la coordinata x del punto sia in segno concorde alla velocità lineare del robot: vale a dire che l'operazione di calcolo delle forze viene fatta solo per i punti nei 180° frontali (x positive) quando la piattaforma mobile va avanti, mentre se indietreggia il calcolo si fa solo per i punti nei 180° dietro (x negative). Generalmente i robot tendono solo ad avanzare, e con questo if possiamo risparmiarci di effettuare calcoli per ostacoli che sono stati passati e ormai non costituiscono più un pericolo; si è pensato però in questo nodo ROS di contemplare un carattere il più generale possibile, quindi l'algoritmo può funzionare anche indietreggiando, e si è voluto in parte "estendere" il carattere dei local planner,

nodi che spesso fanno solo avanzare i robot. Il sensore al laser di Stage ha un angolo di circa 270°, quindi c'è un'area di 90° dietro il robot in cui ovviamente non può sapere se ci siano ostacoli, ma con l'approccio implementato il programma potrebbe tranquillamente funzionare per sensori anche a 360° e trattare l'indietreggiare con la stessa facilità dell'andare avanti (anzi, non si noterebbe la differenza tra i due). Comunque, possiamo notare il calcolo delle forze applicate dal singolo punto con le formule viste, dipendenti da distanza e angolo. Esse vengono poi sommate alla correzione lineare e angolare totali.

Finito il ciclo for e avendo trovato il contributo di ogni punto, possiamo ultimare il calcolo delle correzioni totali:

```
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  ...
  linear_correction *= (-1/SCALE) * twist.linear.x;
  angular_correction *= (-1/SCALE) * twist.linear.x;
  ...
  twist.linear.x += linear_correction;
  twist.angular.z += angular_correction;
  ...
}
```

Esse vengono sommate direttamente alla velocità lineare e angolare ricevute in input, introducendo così un rallentamento e una velocità di sterzata, e a questo punto possiamo inviare il comando corretto:

```
void laserCallback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
  ...
  output_vel_pub.publish(twist);
}
```

Capitolo 5

Fase di testing

Prima di vedere alcuni esempi di test, riassumiamo brevemente i passi importanti dell'algoritmo di collision avoidance:

- se il robot è fermo, nessuna azione di correzione della traiettoria viene presa;
- se il robot è in movimento, si calcola l'angolo e la distanza di ogni punto rilevato dal laser rispetto alla base mobile;
- per ogni punto che sia entro la distanza di soglia e che si trovi o nei 180° davanti mentre il robot avanza o nei 180° dietro mentre indietreggia, se ne calcola il contributo nella correzione di velocità lineare e angolare;
- i singoli contributi vengono sommati l'un l'altro e con le formule viste vengono sommati ai comandi di velocità ricevuti, inviandone di nuovi che fanno evitare collisioni.

Durante la scrittura del programma sono stati fatti molti test, poiché essi si sono rivelati fondamentali per capire l'approccio da seguire (quello delle forze repulsive), in che modo calcolare le forze e far respingere il robot, e per calibrare i parametri. Ad ogni cambiamento nel nodo ROS è corrisposto una serie di test, per riuscire a trovare in quali casi non funzionasse correttamente. In questo capitolo sono riportate alcune prove dell'efficacia del programma.

Stage ci lascia la possibilità di tracciare il percorso effettuato dalla base mobile, ed è uno strumento utile per vedere in che modo questo si allontani dagli ostacoli. Partiamo dal test più semplice: velocità lineare costante, nessun ostacolo (figura 5.1). Chiaramente, il robot non prenderà deviazioni, perché l'algoritmo si affida ai dati dei sensori, che entro THRESH_RANGE non hanno trovato nulla. Lo stesso test viene ripetuto con un ostacolo davanti, e il robot, dopo aver rallentato, riesce a deviare (figura 5.2). Si può vedere come in questo caso la base mobile abbia bisogno di avvicinarsi e sterzare quasi sul posto, ma ciò deriva dal fatto che i punti all'interno di angoli prossimi agli 0° forniscono una buona forza di frenata, non di rotazione. Ad ogni modo, con l'aumento dell'esponente che abbiamo visto nella formula, possiamo star certi che il robot riuscirà ad evitare anche gli ostacoli posti esattamente di fronte, senza scontrarsi.

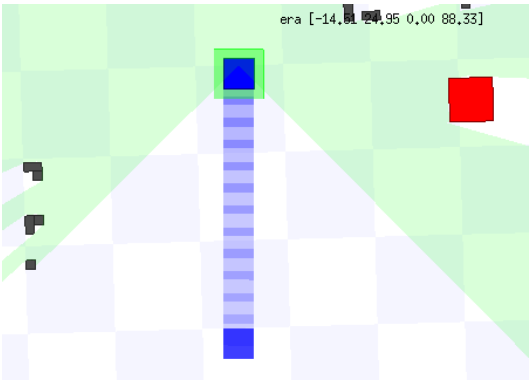

Figura 5.1. Test senza ostacoli, velocità lineare costante

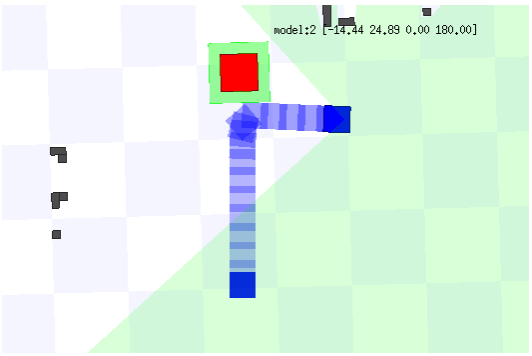

Figura 5.2. Test con ostacolo di fronte, velocità lineare costante

Proviamo poi a far sterzare il robot verso destra con un ostacolo a sinistra. Vediamo ciò in figura 5.3 e 5.4, e possiamo notare che nella seconda immagine, poiché il muro è più vicino, il robot sterzerà maggiormente. Ovviamente se l'ostacolo è a destra la base mobile andrà a sinistra (figura 5.5).
Uno dei problemi maggiori di cui abbiamo parlato era riuscire a far capire al robot di sterzare in tempo avendo pochi punti di riferimento davanti. Come si vede in figura 5.6, ciò è stato risolto. Il robot passa anche tranquillamente nelle porte (figura 5.7).
Dobbiamo a questo punto testare anche l'influenza della velocità angolare, e possiamo vedere che se camminiamo vicino ad un muro e cerchiamo di girare verso di esso, questo continuerà a respingerci dalla parte opposta (figura 5.8). Il risultato sarà un cammino più o meno parallelo al muro, in cui la sterzata da un lato verrà compensata da quella di correzione nell'altra direzione.
Ciò che ci manca è la velocità lineare negativa (figura 5.9). Come ci aspetteremmo, il robot funziona allo stesso modo e devia correttamente, anche se è chiaro che nell'angolo cieco di 90° esso non vede nulla, e non possiamo pensare che riesca ad evitare oggetti che si trovino proprio lì. Basi mobili con sensori ad angoli maggiori potranno destreggiarsi meglio tra gli ostacoli posti nella parte posteriore.
Per finire, poiché il programma deve rispondere adeguatamente a qualsiasi velocità inviata al robot, un test nel quale si è semplicemente cercato di fare diverse collisioni variando più volte sia la velocità lineare che angolare (figura 5.10). Si può osservare che l'algoritmo riesce ad adattarsi a diverse combinazioni di velocità.

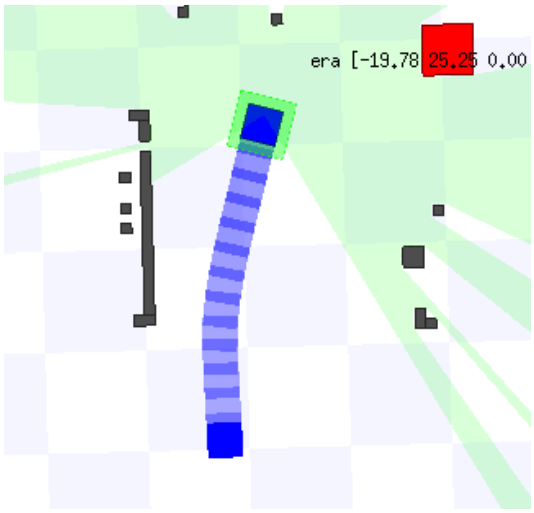

Figura 5.3. Test con ostacolo a sinistra lontano, velocità lineare costante

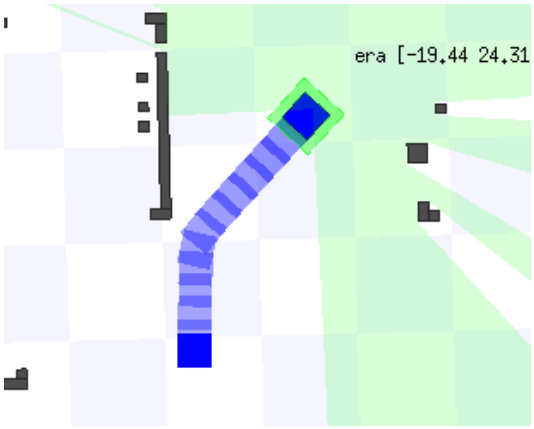

Figura 5.4. Test con ostacolo a sinistra vicino, velocità lineare costante

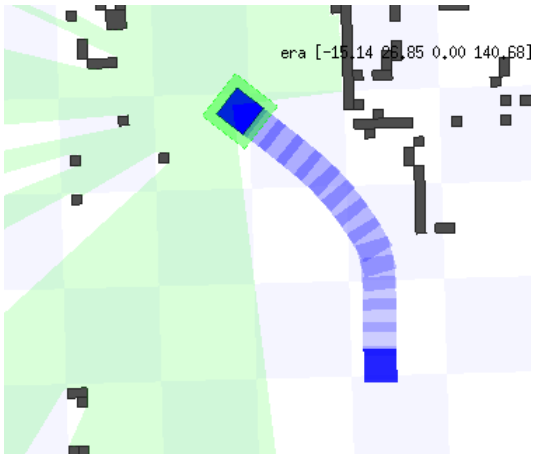

Figura 5.5. Test con ostacolo a destra, velocità lineare costante

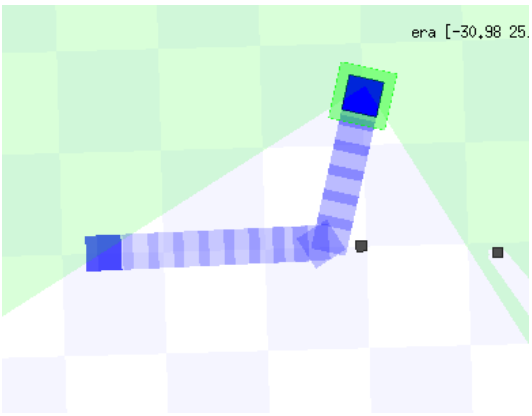

Figura 5.6. Test con ostacolo piccolo davanti, velocità lineare costante

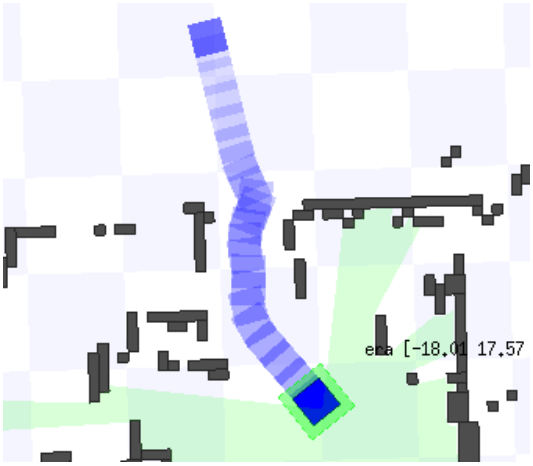

Figura 5.7. Attraversamento di una porta, velocità lineare costante

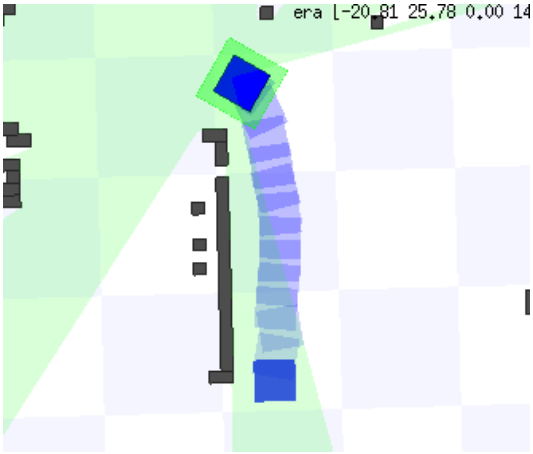

Figura 5.8. Test vicino a un muro, velocità lineare costante, velocità angolare costante positiva (senso antiorario)

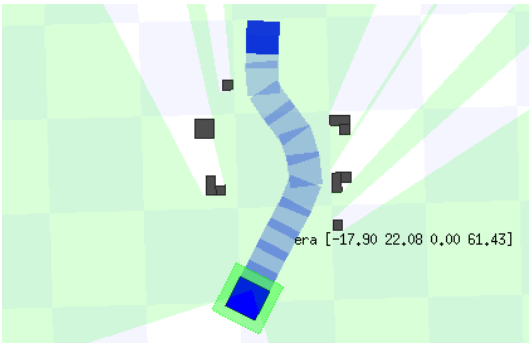

Figura 5.9. Test a velocità lineare costante negativa

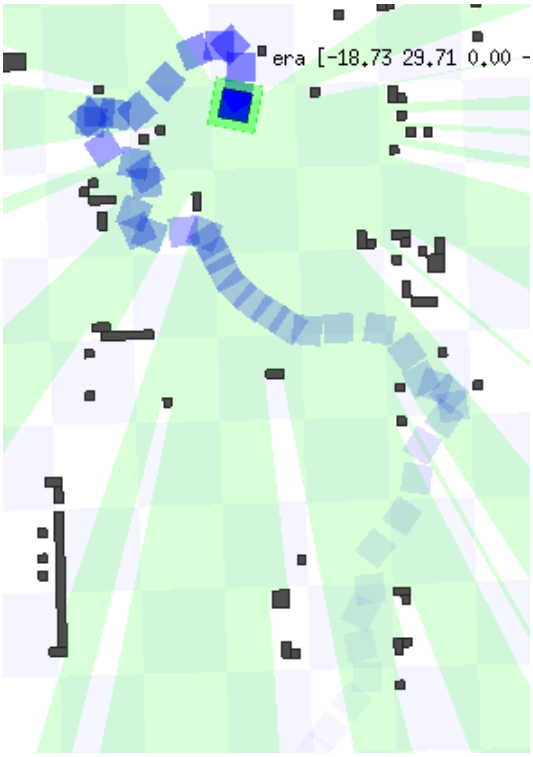

Figura 5.10. Test a velocità lineare e angolare variabili

Capitolo 6

Conclusione

Riassumendo, abbiamo creato un nodo ROS che, usando molti dei concetti che abbiamo studiato nelle sezioni precedenti, tenta di implementare un algoritmo di collision avoidance senza fare calcoli o usare risorse entrambi eccessivamente complicati, e che mantenga un carattere il più generale possibile, si veda la possibilità di rinominare i topic a seconda dei bisogni e la possibilità di usare uno scanner al laser con qualsiasi angolo, compresi 360°.
Nel complesso penso di potermi ritenere soddisfatto: il robot evita agilmente gli ostacoli, non fa brusche frenate se non quando necessario, e a volte riesce ad entrare in luoghi dove non mi aspetterei ci riuscirebbe. Si è tenuto conto dei vincoli cinematici della base mobile e si è adattato l'algoritmo di conseguenza, ed è stato esplorato almeno in parte un argomento visto poco durante il corso, il path planning.
Oltre a tutto questo, il progetto mi ha aiutato a comprendere molto meglio ROS, uno strumento estremamente interessante, e a capire meglio come risolvere un problema facendo tanti piccoli step e con tanti test alla ricerca dell'istanza in cui il programma non reagisse correttamente, migliorando di volta in volta la soluzione, sperando che tutto ciò sia stato un buon insegnamento per il futuro.