SAPIENZA
UNIVERSITÀ DI ROMA

Progetto e sviluppo di un sistema anticollisione per sistemi di navigazione robotici controllati da remoto

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti"

Corso di Laurea in Ingegneria Informatica e Automatica

Francesco Fortunato

Matricola 1848527

Relatore

Thomas Alessandro Ciarfuglia

Anno Accademico 2021/2022

Progetto e sviluppo di un sistema anticollisione per sistemi di navigazione robotici controllati da remoto
Tesi di Laurea Triennale. Sapienza Università di Roma

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

Email dell'autore: francesco.fortunato1999@gmail.com

Alla mia famiglia

Sommario

Il documento che segue è la relazione finale del progetto del *Laboratorio di Intelligenza Artificiale e Grafica Interattiva* basato sulla realizzazione di un sistema anticollisione per sistemi di navigazione robotici. Il seguente testo è frutto dello studio e del lavoro di molteplici settimane che mi ha permesso di acquisire varie conoscenze sul campo della robotica, in particolar modo dal punto di vista software.

Il sistema anticollisione per sistemi di navigazione robotici controllati da remoto si compone di una base mobile, un monitor, e un dispositivo di controllo. Tale sistema si basa, in particolare, sull'utilizzo della *forza assistita* generata dagli ostacoli posti nelle immediate vicinanze del robot. Questa forza viene applicata mentre l'utente utilizza un dispositivo di controllo per telecomandare il robot, permettendo così di evitare le collisioni, migliorando di molto le prestazioni di sicurezza.

Questa tesi rappresenta per me il coronamento tanto atteso della mia carriera universitaria triennale, e sarò legato a questo momento per tutto il resto della vita.

Ringraziamenti

Prima di procedere con la trattazione, vorrei dedicare qualche riga a tutti coloro che mi sono stati vicini in questo percorso di studi.

Vorrei ringraziare innanzitutto i miei genitori, che mi hanno permesso di arrivare a questo traguardo, per avermi sempre sostenuto senza mai darmi pressioni di alcun tipo e per essermi stati sempre accanto. Le parole non possono esprimere quanto sono grato a mia madre e mio padre per tutti i sacrifici che hanno fatto per me. Senza di loro non sarei qui, oggi, a scrivere questa tesi.

Ringrazio tutti i professori del mio corso di studi, che hanno fatto nascere grandi passioni in me.

Ringrazio tutti i miei amici e colleghi, con cui ho condiviso gioie e dolori dell'università, in particolare ai miei compagni di studi, di svago e di sofferenze Ernest, Giacomo e Gabriele.

Ringrazio la mia fidanzata Alessandra, per essere stata sempre al mio fianco, per il supporto che non è mai mancato, per tutti i momenti passati e per aver sempre riempito le nostre giornate di amore e spensieratezza.

Infine, un grazie anche me stesso e alla mia forza di volontà: ho fatto anch'io i miei sacrifici, ho sudato sui libri per anni, ed è soprattutto grazie a tutto ciò se sono arrivato fin qui.

Grazie.

Indice

Capitolo 1

Introduzione

1.1 Obiettivi della tesi

La tecnologia dei robot mobili autonomi e telecomandati è divenuta sempre più popolare per molteplici situazioni e obiettivi. I robot autonomi sono da tempo studiati per assistere nei compiti umani, sia nella vita di tutti i giorni, che in quella professionale [1, 2]. Di recente, ad esempio, in ambito sanitario, sono stati studiati e realizzati robot autonomi che hanno permesso di fronteggiare l'emergenza Covid-19 nella sanificazione degli edifici sanitari [3]; nel contesto industriale, invece, vengono studiati metodi di pianificazione del percorso (*planning path*) con prevenzione delle collisioni [4, 5]. Pertanto, possiamo dire che i robot autonomi sono stati utilizzati principalmente in ambienti ben strutturati, per il raggiungimento di obiettivi quali il rilevamento del movimento, la pulizia all'interno di un edificio (come riportato in figura 1.1), o la produzione nelle catene di montaggio [6, 7].

Figura 1.1. Immagine di un robot aspirapolvere di marca Thamtu.

Tuttavia, sono diverse le situazioni in cui i robot autonomi non funzionano correttamente, come, ad esempio, in ambienti non ben strutturati che richiedono una certa *flessibilità*. In questi casi, i robot telecomandati sono quelli che vengono maggiormente utilizzati, proprio perché i professionisti (o, comunque, chiunque ne comprenda il funzionamento) possono azionare e controllare a distanza il robot, ottenendo, per l'appunto, la sopracitata flessibilità richiesta [8, 9].

In ambito sanitario, ad esempio, sono stati studiati robot telecomandati a supporto del personale ospedaliero [10, 11], mentre in ambito industriale sono stati sviluppati algoritmi e robot telecomandati che riuscissero a perlustrare e tracciare ambienti più complessi [12]. Nella cura della salute e del benessere, essi sono stati utilizzati per assistere gli esseri umani nelle circostanze di ogni giorno; in situazioni di risposta ai disastri e in aree pericolose, questi sono stati ampiamente utilizzati per effettuare ispezioni o missioni di salvataggio [13, 14].
I robot telecomandati ci permettono, dunque, di completare una serie di attività in base al giudizio e alla competenza dell'operatore. Per il sistema telecomandato vengono solitamente utilizzati due dispositivi: un monitor, che permette la visualizzazione di informazioni visive, e un dispositivo di controllo, un controller. L'utente aziona il robot telecomandato utilizzando il dispositivo di controllo mentre visiona il monitor, che visualizza le informazioni dai vari sensori. Affinché si possa operare in modo flessibile, sono necessari operatori esperti che comprendano e siano in grado di utilizzare al meglio il robot telecomandato. Per questo motivo, il funzionamento del sistema non è semplice e, soprattutto, non è esente da rischi, il che implica la possibilità di errori operativi. Questo perché è difficile per l'operatore comprendere le situazioni ambientali reali utilizzando un monitor che mostra solo le informazioni visive. Pertanto, per avere la giusta esperienza, sono necessarie numerose fasi di addestramento affinché l'utente riconosca l'ambiente circostante dalle informazioni visive, ma non sempre questo è possibile.
Nella robotica mobile, una delle caratteristiche maggiormente cercate è la capacità da parte del robot di riuscire a navigare in modo sicuro e allo stesso tempo fluido all'interno di un ambiente sconosciuto. Il problema, però, è che esso potrebbe incontrare ostacoli che possono essere rigidi o avere forme mutevoli. La loro traiettoria e la loro velocità possono essere soggette a cambiamenti imprevedibili nel tempo. Ed è proprio in questi casi che la pianificazione e la prevenzione sono strettamente limitate, proprio poiché richiederebbero una conoscenza preliminare dell'ambiente.
Questa tesi si concentra sullo sviluppo di un sistema che previene ed evita le collisioni. In particolare, per il raggiungimento di questi obiettivi, che, di fatto, aumentano notevolmente le prestazioni di sicurezza, è stato studiato nel dettaglio il cosiddetto "*force feedback*".
Durante il funzionamento di un robot telecomandato, il force feedback viene spesso impiegato per assistere l'utente e migliorare la sua percezione degli ambienti e aumentare le sue capacità operative [15, 16, 17]. Perciò, la valutazione di un metodo combinato con l'assistenza di forze e gli aiuti visivi è uno studio molto significativo per i robot telecomandati. Per fare un parallelismo, in ambito automobilistico sono già in circolazione sistemi di prevenzione delle collisioni, come l'AEB (Autonomous Emergency Braking) o il FCW, (Forward Collision Warning), basati principalmente su una tecnologia a sensori ottici (LIDAR) che scansionano lo spazio nel contorno dell'automobile. Questi sistemi di assistenza alla guida sfruttano sensori di parcheggio, telecamera posteriore e radar per rilevare oggetti che si avvicinano all'auto durante le manovre e, in caso di rischio collisione, attiva i freni anche se il conducente non si è accorto dell'ostacolo. In un'indagine dell'HLDI (ente no profit), è stato analizzato l'impatto di questi sistemi sugli incidenti e le richieste di risarcimento. Viene mostrata di seguito in figura 1.2 una tabella rappresentante l'indagine effettuata.

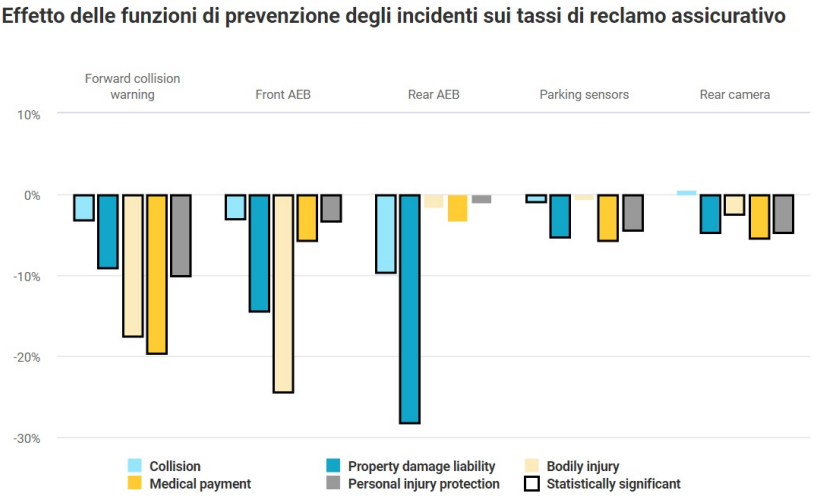

Figura 1.2. Impatto dei sistemi di assistenza anticollisioni sugli incidenti e sulle richieste di risarcimento.

Ebbene, secondo i dati elaborati dal HLDI le auto con AEB posteriore o frenata automatica in retromarcia, sono state coinvolte nel 28% di richieste di risarcimento in meno e nel 10% in meno di collisioni. Sembrerebbe scontata l'importanza di questi sistemi, eppure c'è da dire che la dotazione delle auto di massa è ancora ferma ai sensori di parcheggio e alle retrocamere.
Il sistema anticollisione per robot mobili completamente controllati da remoto proposto in questa tesi si basa non solo su laser, come i sistemi citati precedentemente per le automobili, ma anche e soprattutto sul concetto del force feedback: ciò vuol dire che il sistema non frenerà completamente il robot quando esso si avvicina agli ostacoli, ma lo farà rallentare in maniera esponenziale, modificando anche la traiettoria, in modo da evitare la collisione.
L'algoritmo proposto è stato valutato simulando un ambiente al chiuso utilizzando il meta-sistema operativo per robot **ROS**, acronimo che sta per **Robot Operating System**, e i suoi strumenti, di cui se ne parlerà meglio in seguito. I risultati sperimentali ottenuti utilizzando il metodo proposto mostrano che l'operabilità e le prestazioni di sicurezza sono particolarmente elevate. Durante lo sviluppo del progetto, è stato considerato come ambiente di simulazione l'edificio del DIAG, il Dipartimento di Ingegneria Informatica, Automatica e Gestionale "Antonio Ruberti".

1.2 Organizzazione della tesi

In questa sezione viene proposta un'illustrazione sulla struttura della tesi.
Nel Capitolo 2 verrà effettuato un excursus sulle tecnologie utilizzate, in particolare su ROS, sulla sua storia e sugli strumenti che mette a disposizione.
Nel Capitolo 3 verrà discusso nel dettaglio il sistema di anticollisione, l'algoritmo utilizzato, le istruzioni per eseguire il codice e testare il sistema, le scelte progettuali e le varie ragioni che hanno portato a queste scelte.

Nel Capitolo 4 verranno esposti i vari esperimenti effettuati con i relativi risultati, e verranno mostrati anche alcuni dei limiti del sistema di anticollisione, che verranno poi ripresi nell'ultimo capitolo, il Capitolo 5, in cui vengono illustrate le possibili situazioni dove è possibile sfruttare questo sistema come soluzione a molti problemi reali, e dove sono esposti possibili miglioramenti e implementazioni da aggiungere al sistema.

Capitolo 2

Tecnologie e strumenti utilizzati

In questo capitolo verranno esposte le tecnologie utilizzate. Si inizierà da alcuni cenni sulla robotica per poi approfondire il meta-sistema operativo ROS, la sua storia e gli strumenti che esso mette a disposizione.

2.1 Cenni sulla Robotica

La Robotica è quella branca dell'ingegneria che si occupa della progettazione, dello sviluppo e del funzionamento dei robot, dello studio dei sistemi informatici per l'elaborazione dei dati provenienti dai sensori e del loro utilizzo per delineare il comportamento da tenere al fine di eseguire compiti specifici. Il termine fu coniato nel 1920 dallo scrittore e scienziato ceco Karel Čapek e ha il significato di "lavoro forzato" (da *robota*, in ceco) e, in effetti, il fine ultimo di un robot è quello di riprodurre in qualche forma un tipo di lavoro umano.
Un robot deve affrontare tutte le difficoltà provenienti dall'interazione col mondo reale: una semplice salita, ad esempio, può modificarne la velocità e la percezione della distanza percorsa, o un leggero urto può condizionarne la traiettoria irrimediabilmente.
Così, pur avendola delineata come branca dell'ingegneria, sono in realtà molteplici le discipline con cui deve confrontarsi: in primis vengono richieste conoscenze dei modelli matematici e della fisica, soprattutto per quanto riguarda la parte di acquisizione ed elaborazione dei dati, essenziale per potersi rapportare con l'ambiente con cui interagisce; successivamente, l'informatica è la disciplina cardine per tutto ciò che riguarda la parte computazionale, e in primo luogo per quello che riguarda il controllo.
Come deve reagire il robot ai dati sensoriali? Nel caso debba eseguire necessariamente più azioni, quale deve essere l'ordine? Sono diverse le architetture di controllo sviluppate al fine di ricreare una sorta di "intelligenza" interna al robot, ed evidentemente il campo presenta una complessità non affrontabile dal singolo programmatore.
Nel seguito della tesi verrà esposta una breve descrizione della struttura tipica di un robot e, successivamente, si entrerà nel dettaglio del framework ROS come utile strumento per lo sviluppo software e test di applicazioni robotiche.

2.2 Robot

Un *robot* è una macchina – in particolare, programmabile da un computer – in grado di eseguire una serie di azioni complesse automaticamente [18]. Con il termine "robot", dunque, si vuole indicare un sistema autonomo operante nel mondo fisico, quindi con una sua struttura materiale, che percepisce l'ambiente che lo circonda e che interagisce con esso, attuando procedure al fine di perseguire l'obiettivo per cui è stato creato.
Un robot deve avere, perciò, una struttura fisica con cui interagire col mondo esterno in modo da poterne affrontare le sfide. Un robot che esiste solo all'interno di un computer è solo una simulazione che non affronta la vera complessità del mondo reale. È necessario, quindi, che percepisca l'ambiente in cui si ritrova ad operare e, per fare ciò, deve montare sensori di vario tipo al fine di recuperare informazioni riguardanti il mondo circostante e rispondere nel modo migliore possibile agli ostacoli catturandone la presenza nel proprio percorso. Un robot deve poter agire in risposta agli input derivanti dai sensori, montando quelli che vengono definiti attuatori e affrontare il mondo circostante nelle innumerevoli forme in cui può presentarsi.
Una macchina deve avere tutte le caratteristiche generali sopracitate al fine di essere considerata un robot, ma, per poter esistere, deve presentare quattro elementi fisici che vanno poi a rispecchiarsi nel relativo hardware: un corpo fisico, dei sensori, degli effettori/attuatori e, infine, dei controllori. Essi verranno descritti brevemente di seguito.

2.2.1 Corpo

Un robot deve obbligatoriamente essere dotato di un *corpo fisico* per esistere e per poter interagire con il mondo reale. Tuttavia, questo lo porta a dover rispettare tutte le leggi fisiche che esso impone, non potendo disporre di tutti i vantaggi presenti in ambiente di simulazione. Per potersi muovere deve sfruttare i propri attuatori; l'aumento o la diminuzione della sua velocità richiede tempo ed queste azioni possono essere influenzate dalla topologia del mondo circostante. Inoltre, disporre di un corpo implica la probabile eventualità di incorrere in ostacoli e quindi di dover essere coscienti di cosa vi è attorno. Infine, il corpo impone al robot tutta una serie di limitazioni, derivanti dalla forma e struttura del corpo stesso, che vanno a influire sui movimenti, l'interazione con oggetti o altri robot ecc.

2.2.2 Sensori

Si identificano sotto il nome di *sensori* tutti quei dispositivi fisici montati su un robot in grado di ottenere informazioni dall'ambiente circostante e permettere al robot stesso di essere "cosciente" di ciò che lo circonda. Il tipo di sensore montato dipende dal tipo di informazione che viene richiesta al fine di poter portare a compimento il proprio task. Si dividono essenzialmente in sensori passivi e attivi.
I primi si limitano a strumenti che rilevano la proprietà fisica da misurare, mentre gli ultimi generano un segnale e sfruttano la relativa interazione con l'ambiente circostante come proprietà da misurare. Pertanto, l'uso di entrambi permette al robot di essere a conoscenza del proprio stato, ossia di una descrizione di sé stesso

in un certo dato momento, nel qual caso si parla di stato interno, e dell'ambiente circostante in cui deve operare, e qui si parla di stato esterno.

2.2.3 Effettori e Attuatori

Per *effettore* si intende ogni dispositivo fisico che ha un impatto sull'ambiente reale su cui il robot deve operare. Sono essenzialmente la controparte meccanica di quello che in biologia rappresenta una gamba, un braccio, un dito, una mano. Il meccanismo che rende possibile ad un effettore di compiere un'azione viene definito *attuatore*, e tale termine include dispositivi di differenti tipi di tecnologie, come motori elettrici, dispositivi idraulici, pneumatici, materiali foto-reattivi, chimico-reattivi e altri ancora.

2.2.4 Controllori

I *controllori* sono le componenti (hardware e software) che permettono al robot di essere autonomo, processando gli input provenienti dai sensori o altre informazioni come quelle contenute in memoria, decidere quale azione intraprendere e di conseguenza quali effettori mettere in movimento.

2.3 ROS, Robot Operating System

Nel campo della robotica, le *platform* stanno assumendo un'importanza sempre maggiore. Una piattaforma può essere di tipo software o hardware. Una piattaforma software, in particolare, contiene strumenti utilizzati per creare programmi applicativi per robot come il controllo del dispositivo a basso livello, SLAM (*Simultaneous Localization And Mapping*), navigazione, manipolazione e riconoscimento di oggetti o esseri umani, rilevamento e gestione dei pacchetti, debugging e vari strumenti di sviluppo.
La piattaforma software per robot ad oggi più utilizzata è **ROS**, **Robot Operating System**. ROS è un framework open source che consente di gestire operazioni, compiti, movimenti e qualsiasi altra cosa che un robot può eseguire ed è destinato a fungere da piattaforma software non solo per chi costruisce e utilizza robot quotidianamente, ma anche per coloro che approcciano per la prima volta il mondo dello sviluppo di sistemi robotici.
Questa struttura open source della piattaforma consente l'utilizzo del codice e delle informazioni condivise da altri programmatori. Ciò implica che non è necessario scrivere tutto il codice per far funzionare un robot e questo è uno dei motivi principali per cui ROS ha riscosso un notevole successo, diventando una delle soluzioni più utilizzate nel campo dello sviluppo di sistemi robotici.

2.3.1 Storia di ROS

Possiamo collocare l'inizio dello sviluppo di ROS nel 2007: in quegli anni, due dottorandi dell'università di Stanford, Eric Berger e Keenan Wyrobek, vollero cimentarsi nella creazione di una piattaforma software che permettesse ai loro futuri e presenti colleghi accademici di interfacciarsi al mondo della robotica nella maniera

più semplice possibile.
Nei loro primi passi di costruzione di questa piattaforma "unificatrice", la prima cosa che fecero fu quella di costruire come prototipo hardware il PR1 dal programma Personal Robotics (PR) e da qui iniziarono a lavorare sul software prendendo spunto dai primi framework robotici open source, in particolare "*switchyard*", un sistema a cui Morgan Quigley, un altro dottorando di Stanford, aveva lavorato a sostegno del progetto Stanford AI Robot STAIR (STanford AI Robot) dallo Stanford Artificial Intelligence Laboratory [19].
Nell'anno successivo, durante la ricerca di finanziamenti per un ulteriore sviluppo [20], Eric Berger e Keenan Wyrobek incontrarono Scott Hassan, il fondatore di Willow Garage, un laboratorio di ricerca robotica, che all'epoca stava lavorando ad un SUV autonomo e una barca solare autonoma. Hassan condivise l'idea dei due dottorandi e li invitò a lavorare per Willow Garage. I due accettarono e iniziarono lo sviluppo della piattaforma robot ad oggi più utilizzata, alla quale parteciparono gruppi provenienti da più di venti istituzioni diverse.
La prima versione stabile di ROS, ROS 1.0, venne rilasciata il 22 gennaio del 2010 [21].
Lo sviluppo di ROS da parte di Willow Garage continuò fino al 2013, quando avvenne la transizione verso l'OSRF, l'Open Source Robot Foundation (che dal 2017 si chiama Open Robotics), che divenne il principale manutentore di ROS [22].
Da allora la popolarità di ROS è cresciuta a dismisura: il 1° settembre 2014, la NASA annunciò il primo robot con framework ROS nello spazio: *Robotnaut 2*, sulla Stazione Spaziale Internazionale [23]; anche giganti della tecnologia come Amazon e Microsoft hanno iniziato ad interessarsi a ROS durante questo periodo, con Microsoft che ha portato ROS su Windows nel settembre 2018, seguito da Amazon Web Services che ha rilasciato RoboMaker nel novembre 2018.
Tutta questa popolarità è stata favorita soprattutto dal fatto che ROS venne rilasciato sin dall'inizio sotto licenza BSD (Berkeley Software Distribution) come software gratuito sia per l'ambito di ricerca che per fini commerciali: grazie, dunque, alla natura open source, vi fu un grande contributo da parte della comunità di ricerca mondiale, che permise a ROS di diventare la piattaforma robot più utilizzata al mondo.

2.3.2 Il Meta-Sistema Operativo ROS: Caratteristiche e Filosofia

ROS è un meta-sistema operativo open source per il robot. Fornisce i servizi che potremmo aspettarci da un classico sistema operativo, inclusa l'astrazione dell'hardware, il controllo del dispositivo di basso livello, l'implementazione di funzionalità più comuni, lo scambio di messaggi tra i processi e la gestione dei pacchetti. Fornisce inoltre strumenti e librerie per ottenere, creare, scrivere ed eseguire codice su più computer, in modo da semplificare il compito di creare un comportamento robotico complesso e robusto su un'ampia varietà di piattaforme robotiche.
Nonostante il nome, dunque, ROS non è semplicemente un sistema operativo. Piuttosto potremmo definirlo come un SDK (*Software Developement Kit*) che fornisce gli elementi costitutivi necessari per creare applicazioni robot. In un certo senso, ROS può essere visto come l'impianto idraulico alla base dei nodi e del passaggio

dei messaggi, che fornisce un insieme ricco e maturo di strumenti e di abilità come mostrato in figura 2.1.

Figura 2.1. Un'immagine raffigurante l'equazione ROS: impianto idraulico + strumenti + capacità + ecosistema = ROS!

Le caratteristiche principali di ROS sono le seguenti:

- la riutilizzabilità del programma: ciò significa che un utente può focalizzarsi su un obiettivo dell'applicazione che vuole sviluppare e scaricare il pacchetto relativo alle funzionalità rimanenti. Allo stesso tempo, esso può condividere il programma che ha sviluppato, in modo da rendere possibile agli altri di riutilizzarlo;
- la modularità: spesso, per offrire un servizio, alcuni programmi hardware sono sviluppati in un singolo frame, ma, per raggiungere la riutilizzabilità dei software robotici, ogni programma viene suddiviso in porzioni minori, in base alla propria funzione;
- l'elevato numero di strumenti disponibili per lo sviluppo: ROS offre diversi tool per eseguire debug, strumenti per la visualizzazione 2D (come stage), 3D (come Rviz), o che permettono di visualizzare le relazioni tra nodi (come Rqt). Sono inoltre disponibili in numero sempre più crescente algoritmi robotici che permettono di mappare l'ambiente intorno il robot, rappresentare e interpretare i dati dei sensori, pianificare lo spostamento, manipolare oggetti e molte altre operazioni;
- la community attiva: esistono più di 5000 pacchetti sviluppati volontariamente e condivisi tra gli utenti, pagine Wiki che documentano molti aspetti del framework ed un sito Q&A dove si può chiedere e fornire aiuto, condividendo ciò di cui si è a conoscenza;
- l'ecosistema: come detto precedentemente, sono molteplici le piattaforme software sviluppate, ma la più popolare e utilizzata è proprio ROS, anche perché sta creando un ecosistema valido per chiunque, sviluppatori hardware del campo della Robotica, squadre di sviluppo ROS, sviluppatori di applicazioni software e utenti alle prime armi come gli studenti.

Per quanto riguarda gli aspetti filosofici di ROS, invece, possiamo descriverli con le seguenti proprietà:

- Peer-to-peer: i sistemi ROS consistono in una piccolo numero di programmi interconnessi tra loro che si scambiano messaggi in maniera continua. Questi messaggi vengono inviati direttamente da un programma all'altro, rendendo il

sistema più complesso sotto alcuni punti di vista, ma più bilanciato ed efficiente sotto altri, permettendo di avere un numero molto più alto di informazione.

- Supporto multi-linguaggio: i moduli ROS possono essere scritti in diversi linguaggi come: C++, Python, LISP, Java, JavaScript, MATLAB, ecc.
- Leggero: le convenzioni utilizzate in ROS fanno in modo che gli utenti creino delle cosiddette librerie "standalone", ossia indipendenti dalle altre. Questo strato extra viene proposto per permettere il riutilizzo dei software al di fuori di ROS e semplifica in maniera notevole la creazione dei test autorizzati usando strumenti integrati standard.
- Gratis e open source: il core di ROS viene rilasciato sotto i permessi della licenza BSD, che permette un uso sia commerciale che non-commerciale. ROS permette inoltre lo scambio dei dati tra moduli usando l'IPC (Inter Process Communication), il che significa che i processi creati usando ROS possono avere altre licenze per varie componenti.

2.4 Strumenti di ROS

ROS ha molteplici strumenti che possono essere utili quando il robot si muove, oppure quando un algoritmo è in esecuzione, e si vuole capire se il sistema funziona correttamente o meno. Esistono diversi tool ROS, inclusi quelli che gli utenti hanno rilasciato personalmente.
Gli strumenti che verranno descritti e che rappresentano quelli che sono stati maggiormente utilizzati durante gli esperimenti e i test in laboratorio, sono i seguenti: *RViz* (uno strumento di visualizzazione 3D), *Rqt* (ovvero un framework software che implementa strumenti GUI sotto forma di plug-in per visualizzare la correlazione tra nodi e messaggi) e *stageros* (un simulatore 2D).

2.4.1 Il visualizzatore 3D RViz

RViz è lo strumento di visualizzazione 3D di ROS. Lo scopo principale è quello di permettere di visualizzare messaggi e argomenti ROS in 3D, permettendoci di controllare visivamente i dati e i comportamenti del nostro sistema.
C'è la possibilità di visualizzare anche rappresentazioni in diretta dei valori dei sensori relativi ad argomenti ROS inclusi i dati della telecamera, le misurazioni della distanza a infrarossi, i dati del sonar e così via.
RViz ha varie funzioni come l'interazione, il movimento della telecamera, la selezione, il cambio della messa a fuoco della telecamera, la misurazione della distanza, la stima della posizione 2D, il punto di destinazione della navigazione 2D, punto di pubblicazione [25].
RViz è sicuramente uno degli strumenti più utili di ROS.

2.4.2 Lo strumento di sviluppo dell'interfaccia utente grafica Rqt

Oltre ad Rviz, che permette la visualizzazione 3D, ROS fornisce anche tool GUI per lo sviluppo di robot, ovvero *Rqt*.

"*Rqt graph*" è uno strumento che mostra sotto forma di diagramma la correlazione tra nodi attivi e messaggi trasmessi sulla rete ROS. Esso è molto utile per comprendere l'attuale struttura della rete ROS quando il numero di sensori, attuatori e programmi è molto alto.
"*Rqt plot*", invece, è uno strumento che fornisce un plug-in GUI che visualizza i valori numerici in un grafico 2D. Lo strumento riceve messaggi ROS e li visualizza attraverso coordinate 2D [22].
Analizziamo un esempio e studiamo i comandi "turtlesim_node" e "turtle_teleop_key" [24]. Il primo comando apre una finestra blu al cui centro c'è una tartaruga. Eseguendo invece il secondo comando in una nuova finestra, vedremo messaggi e istruzioni che danno la possibilità di muovere la tartaruga utilizzando i tasti freccia della tastiera (←, →, ↑, ↓) o, in un'altra versione, premendo le lettere a, s, w, z. La tartaruga si muoverà secondo il tasto freccia come mostrato nella figura 2.2:

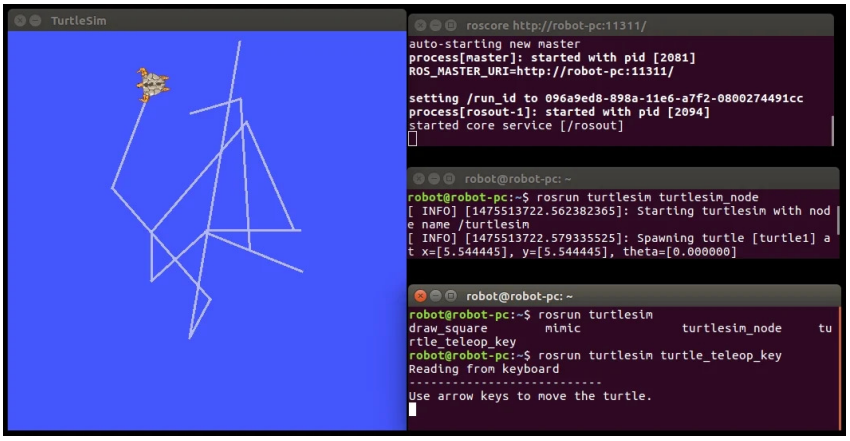

Figura 2.2. Esempio di esecuzione di ROS Turtle.

L'altra figura 2.3 mostra, invece, il grafico generato da Rqt graph, dove i cerchi rappresentano i nodi (/teleop_turtle, /turtlesim), la scritta /turtle1/cmd_vel rappresenta il topic e la freccia indica la trasmissione del messaggio.

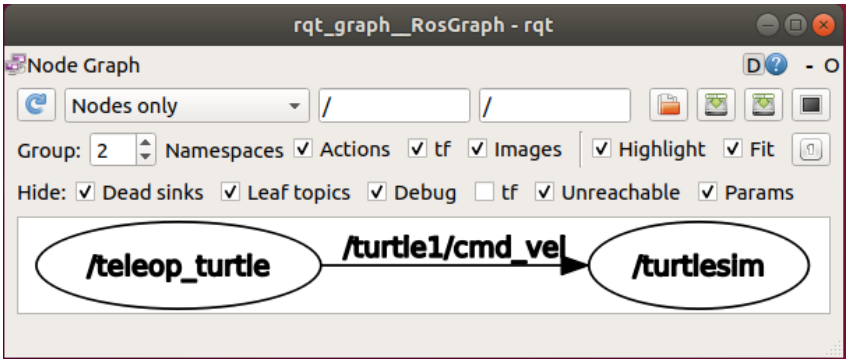

Figura 2.3. Esecuzione di Rqt Graph dell'esempio precedente.

2.4.3 Il simulatore 2D Stage

Il nodo *Stageros* permette di utilizzare il simulatore multi-robot Stage 2D, tramite libstage. Stage simula un mondo definito in un file *.world* che contiene tutte le informazioni su di esso: gli ostacoli contenuti, i robot presenti e tutti gli altri oggetti inclusi.
Questo nodo espone solo un sottoinsieme delle funzionalità di Stage. In particolare, trova i modelli Stage di tipo laser, telecamera, posizione e, successivamente, mappa questi modelli ai vari ROS topic. Se almeno un laser/telecamera e un modello di posizione non vengono trovati, stageros non può funzionare [27].
Ecco rappresentato nella figura 2.4, un esempio dell'esecuzione di stage ros con il file *cappero_laser_odom_diag_obstacle_2020-05-06-16-26-03.world*, che rappresenta la mappa del DIAG:

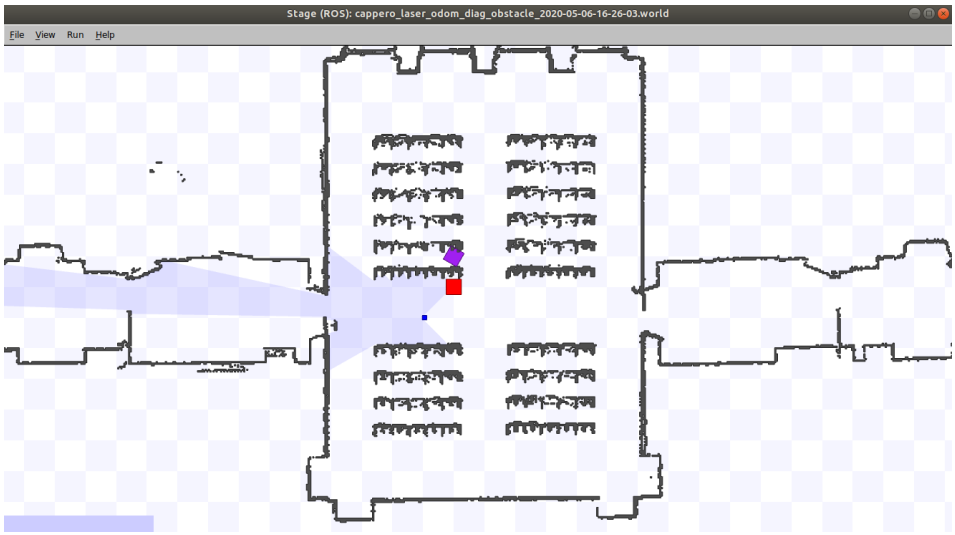

Figura 2.4. Esecuzione di stage.

Capitolo 3

Descrizione del Sistema Anticollisione

Nel seguito della tesi verrà discusso nel dettaglio il sistema anticollisione proposto, l'algoritmo utilizzato, le scelte progettuali e le varie ragioni che hanno portato a queste scelte.

3.1 Specifiche del progetto

Il progetto consiste nell'ideare e sviluppare un sistema anticollisione basato su laser. Senza la presenza di questo, la base mobile prende in input (da un nodo del sistema ROS) una velocità lineare e una angolare con cui la base mobile naviga all'interno del mondo.
Con l'integrazione del sistema anticollisione, oltre alla velocità lineare e quella angolare, viene ricevuto in input anche un array di valori che rappresentano la distanza degli ostacoli che il laser scanner rivela. Il sistema, poi, produce un comando di velocità che, utilizzando lo scanner, previene la collisione con gli oggetti rivelati nel contorno, "deflettendo" la traiettoria del robot verso uno spazio libero da collisioni.
La navigazione, come detto precedentemente, è stata simulata grazie al file .world fornitoci dal professore e all'utilizzo di *stage*.
Continuando con le specifiche del progetto, il sistema ideato doveva, dunque, garantire che la base mobile riuscisse a navigare in un particolare ambiente di simulazione non solo senza collidere con i possibili ostacoli che potevano presentarsi, ma anche modificando la traiettoria in maniera intelligente, scegliendo la strada "*migliore*", cioè quella più sgombra da ostacoli, senza interrompere la navigazione.
Gli ostacoli presenti possono essere di due tipi: statici, come un muro, i banchi delle aule, o dinamici come altri oggetti in movimento.
Il sistema realizzato possiede un'ulteriore implementazione rispetto alle specifiche richieste dal progetto, ovvero la possibilità di controllare da remoto la base mobile attraverso un joystick, migliorando di gran lunga l'operabilità, senza, dunque, l'obbligo di dover inserire manualmente da terminale i valori della velocità da inviare in input al robot.

3.2 Struttura del workspace del progetto

Lo sviluppo del progetto si basa su pacchetti forniti dal professore durante il corso. Tutte le informazioni vengono fornite all'interno di un repository GitLab (vedere la sezione 3.5). Il progetto consiste, all'atto pratico, nella creazione e nello sviluppo di un package ROS all'interno del proprio catkin workspace, che ha una struttura di questo tipo:

```
catkin_ws/       – WORKSPACE
  src/               – SOURCE SPACE
    CMakeLists.txt       – 'Toplevel' CMake file, provided by catkin
    collision_avoidance/
      CMakeLists.txt       – CMakeLists.txt file for package collision_avoidance
      package.xml          – Package manifest for package collision_avoidance
      src/
        collision_avoidance.cpp       – collision_avoidance.cpp file (node)
```

Nel package creato sono state incluse le seguenti dipendenze ROS, necessarie per la compilazione dei vari file: *geometry_msgs*, *nav_msgs*, *sensor_msgs*, *std_msgs* e *roscpp*.
Un package ROS, per essere considerato tale, deve contenere almeno i seguenti elementi:

- Un file *CMakeLists.txt*, che sarà l'input per il sistema di compilazione CMake e per la creazione di pacchetti software. Ogni package compilato con CMake deve contenere uno o più file CmakeLists.txt che descrivono come compilare il codice e dove installarlo. In esso sono definite molte informazioni come la versione del CMake, il nome del package, le dipendenze, vari generatori di messaggi, servizi, azioni e altro.

- Un file chiamato *package.xml* che deve essere presente nella root di ogni pacchetto compilato con catkin e in cui vengono definite tutte le meta-informazioni del pacchetto quali il nome, la versione, gli autori, i manutentori e le dipendenze sugli altri catkin package.

- Una directory */src* nella quale è possibile dichiarare nodi, messaggi, servizi e azioni create dall'utente. Di norma, i file dello stesso tipo vengono raggruppati in cartelle diverse per motivi di organizzazione del lavoro. Nel progetto sviluppato, non era necessario creare altri file tranne che un file nodo (in questo caso il file collision_avoidance.cpp) e, perciò, dentro la cartella /src si trova solo questo file.

3.3 Dettagli di implementazione e terminologia

Il file *collision_avoidance.cpp*, che, di fatto, è un cosiddetto "nodo" (di cui se ne parlerà in questa sezione) nel sistema ROS ed è il file nel quale è stato implementato il sistema anticollisione, è stato scritto utilizzando come linguaggio di programmazione

il linguaggio C++. La scelta è stata strettamente personale: lo stesso sistema, infatti, sarebbe stato scrivibile utilizzando anche altri linguaggi di programmazione.

3.3.1 Terminologia

Verranno date ora definizioni e terminologie usate in questo progetto:

- Un *nodo* è definito come un file eseguibile che sfrutta ROS per comunicare con altri nodi.
- Un *messaggio* è un tipo di dato ROS che viene trasmesso tra più nodi.
- Un *topic* è un canale di comunicazione che permette ai vari nodi di scambiarsi dati oppure messaggi. Un topic può essere usato per mandare/ricevere messaggi di un solo tipo ed è attivo solo quando c'è almeno un nodo che pubblica messaggi e almeno uno che li riceve.
- Un nodo che pubblica messaggi su un topic viene definito *publisher*, mentre un nodo che si sottoscrive ad un topic viene definito *subscriber*. Un nodo può essere contemporaneamente sia di tipo publisher che di tipo subscriber.

3.3.2 Dettagli di implementazione

Il nodo *collision_avoidance.cpp* comunica con altri nodi, ed ha il ruolo sia di publisher che di subscriber.
I due nodi del sistema ROS con cui il nodo /collision_avoidance comunica sono:

- /stageros: un eseguibile fornito durante il corso, che lancia il simulatore 2D Stage con una base mobile presente in una mappa descritta da un file di input. Questo nodo pubblica anche su due topic a cui il nodo /collision_avoidance è sottoscritto. Questi due topic sono:
  - /base_scan: un topic che permette lo scambio di messaggi del tipo sensor_msgs/LaserScan. Questi messaggi contengono informazioni ricevute da un laser planare che fornisce informazioni utili per la navigazione della base mobile. Tra tutte le informazioni che si trovano in questo messaggio, la più importante è quella relativa al vettore che contiene la distanza tra un ostacolo e la base mobile per ogni lettura del laser. Si noti che il laser scansiona l'ambiente con un angolo pari a 270°.
  - /odom: un altro topic che permette lo scambio di messaggi del tipo nav_msgs/Odometry. Questi messaggi contengono informazioni riguardanti la posizione e velocità nello spazio della base mobile. La posizione deve essere calcolata secondo le coordinate di un frame header presente nel messaggio, mentre la velocità secondo quelle del frame child. Il nodo /collision_avoidance ricava da questo messaggio la posizione della base mobile e il suo orientamento in modo da poter calcolare le direzioni delle forze che agiscono sulla base mobile e, successivamente, capire la direzione finale che essa deve prendere in presenza degli ostacoli.

- /joy_control: un altro eseguibile, anch'esso fornito durante il corso, che legge comandi usando un telecomando da gioco (joystick) e li converte in comandi di velocità lineare e angolare. Questi comandi di velocità sono proprio quelli che il nodo /collision_avoidance riceve in input. La trasmissione di questi messaggi (che sono di tipo nav_msgs/Odometry) avviene attraverso il topic /vel_joystick.

Nella figura 3.1 viene mostrata la relazione tra i nodi e i topic presenti nel sistema ROS proposto.

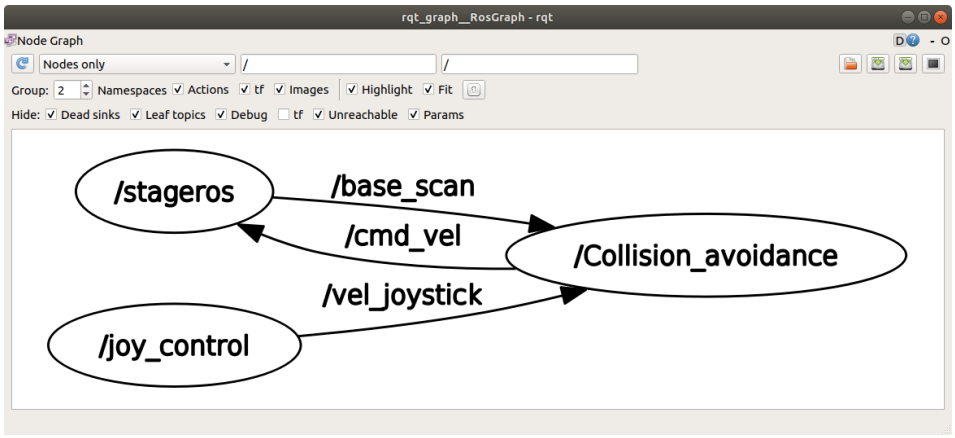

Figura 3.1. Rappresentazione Rqt Graph del sistema anticollisione.

Lo strumento che consente di visualizzare questa relazione è Rqt Graph, lo strumento fornito da ROS che è stato descritto nel capitolo precedente.

3.4 Approccio utilizzato e algoritmo

Come detto poc'anzi, il compito di questo sistema era quello di rivelare nell'intorno della base mobile tutti i possibili ostacoli che si trovassero nelle vicinanze e reagire di conseguenza nel miglior modo possibile per evitarli.
Nel seguito della relazione verrà esposto l'approccio utilizzato e uno pseudocodice relativo all'algoritmo presente nel nodo /collision_avoidance.cpp.

3.4.1 Approccio utilizzato

L'approccio utilizzato in questo progetto è stato quello di associare ad ogni ostacolo rivelato dallo scanner una *forza repulsiva* in base alla distanza dell'ostacolo dal robot. Anche la velocità ricevuta in input può essere vista come una forza: per distinguerla da quella repulsiva generata dagli ostacoli, chiameremo la forza generata dalla velocità ricevuta *forza attrattiva*.
La forza repulsiva deve essere abbastanza grande da riuscire a sovrastare quella attrattiva nel caso ci siano ostacoli troppo vicini, permettendo così il raggiungimento dell'obiettivo richiesto, ovvero quello di evitare gli ostacoli.
Considerando la direzione del percorso minimo dalla base mobile all'ostacolo, la

forza repulsiva dovrà avere stessa direzione, ma verso opposto.
Notiamo che la forza repulsiva è, in realtà, una *somma* di tante forze repulsive quanti sono gli ostacoli che la base mobile incontra durante la navigazione. Perciò, quando si parla di forza repulsiva della base mobile, si parla della somma di tutte le componenti delle forze repulsive associate ad ogni ostacolo.
Nella figura 3.2 seguente vengono mostrate in forma grafica tutte le forze in gioco in una possibile situazione in cui si riceve in input una certa velocità lineare positiva, una velocità angolare nulla, e agisce sulla base mobile una forza repulsiva generata dall'ostacolo *OBSTACLE* situato ad una certa distanza r, con le relative componenti x e z della forza.

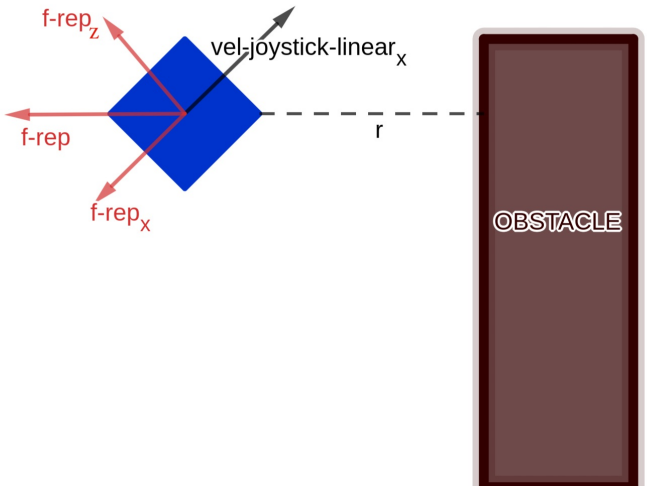

Figura 3.2. Forze in gioco presenti durante la navigazione del robot.

La forza repulsiva dovrà avere un valore maggiore quanto più l'ostacolo si trova vicino alla base mobile e minore quanto più l'ostacolo è lontano. Come si evince anche dall'immagine, il valore delle componenti della forza cambia se l'ostacolo si trova a sinistra, a destra o di fronte alla base mobile.
La forza attrattiva, d'altra parte, ha sempre la stessa direzione e verso della velocità lineare con cui naviga la base mobile. Si noti che quando la base mobile riceve in input anche una velocità angolare, questa forza comunque ha stesso verso della velocità lineare. L'effetto della velocità angolare è semplicemente quello di modificare l'orientamento della base mobile.
Il valore della forza attrattiva viene calcolato moltiplicando la velocità lineare ricevuta in input per una costante, ed essa ha valore maggiore quanto più la velocità ricevuta è alta, mentre ha valore minore quanto più la velocità ricevuta è bassa.
Nella figura 3.2, viene mostrata anche le componente della forza attrattiva secondo gli assi di orientamento (in questo caso l'unica componente è quella indicata con vel-joystick-linear$_x$). Inoltre, poiché la direzione della base mobile coincide con l'asse delle ordinate, allora la componente secondo l'asse delle ascisse sarà sempre nulla.
L'algoritmo utilizzato calcola dunque tutte queste forze per ogni istante della na-

vigazione, ne calcola le componenti secondo gli assi x e z e fa la somma di *tutte* le componenti.
La forza risultante è, dunque, *pari alla somma della forza attrattiva e di tutte le forze repulsive che vengono generate dalla presenza degli ostacoli in quell'istante.*
La componente x della forza risultante indica il valore della velocità lineare con cui la base mobile deve navigare in reazione alla presenza degli ostacoli, mentre la componente z indica il valore della nuova velocità angolare da applicare al robot.
Ecco rappresentate nella figura 3.3 le componenti delle forze risultanti dell'esempio precedente.

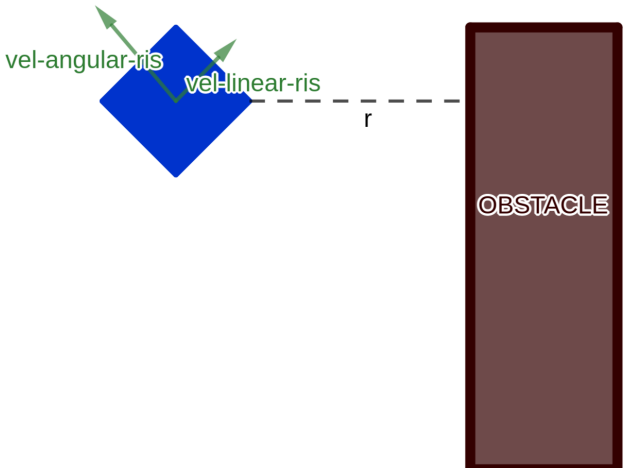

Figura 3.3. Forze risultanti dell'esempio precedente.

È importante notare, inoltre, che quando il verso della forza risultante è opposto a quello dell'orientamento della base mobile (per opposto si intende che i due vettori formano un'angolo maggiore di 90°), ci dev'essere una velocità angolare che permetta alla base mobile di roteare verso la giusta direzione. Questa velocità si sovrappone alla possibile velocità angolare che viene inviata in input alla base mobile.
Si noti che il valore della velocità angolare creata dalla forza risultante, è maggiore quanto più l'angolo tra le due direzioni è ampio, mentre risulta essere minore quanto più l'angolo è stretto. Questo valore della velocità angolare, che all'interno del codice viene definito come *vel_imposta*, determina di fatto quanto velocemente reagisce il sistema alla presenza degli ostacoli.
Altro aspetto importante da notare è che l'algoritmo descritto si "aziona" *solo* nel momento in cui si riceve in input una velocità lineare strettamente positiva. Se la velocità lineare ricevuta è pari a 0, allora il robot rimarrà fermo, mentre nel caso in cui si riceva una velocità lineare negativa, il robot andrà in retromarcia senza modificare la sua traiettoria. Questo perché il laser installato sul robot copre una visuale pari a 270°.

3.4.2 Pseudocodice

In questo paragrafo viene mostrato uno pseudocodice che mostra il funzionamento del programma:

```
MAIN
    Legge il valore della velocità ricevuto in input dal joystick
    Inizializza a zero tutte le forze: attrattiva, repulsiva e risultante
    Genera un array che considera solo i valori centrali letti dal laser
    WHILE true
        IF velocità lineare positiva
            Calcola l'orientamento in radianti della base mobile
            FOR ogni valore della distanza dagli ostacoli abbastanza piccolo
            nell'array dei valori centrali dello scanner
                Calcola le componenti della forza repulsiva generate
                dall'ostacolo
                Somma questi valori alla forza repulsiva totale
            END FOR
            Calcola la forza attrattiva data dalla velocità lineare in input
            Calcola la forza risultante come somma tra forza attrattiva e
            repulsiva
            Calcola il giusto orientamento della base mobile
            Assegna quindi la velocità angolare da trasmettere
            Diminuisci la velocità lineare se vicino agli ostacoli
            Esegui il publish della velocità
        END IF
    END WHILE
END MAIN
```

3.5 Istruzioni di esecuzione del Sistema

Nel seguito verranno date tutte le indicazioni per poter installare e testare il sistema anticollisione.
Per l'esecuzione o il test del sistema è innanzitutto richiesta l'installazione del meta-sistema operativo ROS, seguendo le istruzioni sulla pagina ufficiale all'indirizzo: *http://wiki.ros.org/melodic/Installation.*
Per questo progetto è stata utilizzata la distribuzione ROS Melodic. Il funzionamento è garantito se si utilizza questa versione e il sistema operativo Ubuntu 18.04.
Inoltre, durante il corso, è stato fornito agli studenti del corso *Laboratorio di Intelligenza Artificiale e Grafica Interattiva* un repository GitLab contenente uno script shell che installerà da sé tutti i pacchetti e i nodi necessari al corretto funzionamento del sistema anticollisione.
Più precisamente, verranno installati i package forniti dal professore che permetteranno di leggere comandi di velocità ricevuti da un joystick e di convertirli in messaggi di tipo nav_msgs/Odometry. Sono presenti inoltre i file che permettono di installare e avviare il simulatore 2D Stage e il file .world che descrive il mondo nel quale si vuole far navigare la base mobile.

La documentazione per la configurazione del workspace e l'installazione di tutti i file viene fornita sul repository GitLab del professor Grisetti Giorgio, presente all'indirizzo *https://gitlab.com/grisetti/labiagi_2020_21*.
Come specificato precedentemente, il progetto è stato sviluppato per il sistema operativo Ubuntu 18.04 e tutti i comandi per l'esecuzione vengono eseguiti da shell Linux, mentre il codice del sistema anticollisione e le relative istruzioni per l'installazione si trovano nel repository pubblico presente all'indirizzo *https://github.com/francesco-fortunato/Collision_Avoidance*.
Una volta installato ROS ed eseguite le procedure di installazione per questi due repo, è possibile avviare il sistema anticollisione.
Per prima cosa, come per ogni sistema che utilizza ROS, bisogna avviare la raccolta dei nodi e dei programmi necessari per far funzionare il sistema, ovvero *roscore*.
Su terminale basta eseguire il comando roscore.
Il secondo passo è quello di collocarsi all'interno della cartella fornita durante il corso. In particolare, poiché va utilizzato il file eseguibile che legge comandi inviati dal joystick, bisogna modificare il topic sul quale questo nodo pubblica i messaggi di velocità. Per fare ciò, basterà aprire con un qualsiasi editor il file chiamato joy_teleop_node.cpp che si trova nella directory workspaces/-srrg2_labiagi/src/srrg_joystick_teleop/src. Il nome del topic andrà modificato in /vel_joystick, lo stesso nome al quale il nodo /collision_avoidance, rappresentante il nodo principale del sistema anticollisione, deve sottoscriversi per poter leggere la velocità ricevuta in input.
A questo punto ci si ricolloca nuovamente nella root del repo e si compila il workspace. La compilazione deve avvenire almeno una volta dopo la prima configurazione del workspace e ogni volta che un suo file viene modificato. Basterà eseguire il file *setup.sh*. Successivamente, si esegue il comando *catkin build* nella cartella in cui è stato clonato il repo Collision_Avoidance.
Su un altro tab del terminale Linux viene eseguito il comando per lanciare il nodo /joy_control attraverso l'istruzione rosrun srrg_joystick_teleop joy_teleop_node.
Il comando, come si può vedere, viene composto dalla keyword *rosrun*, fondamentale per lanciare nodi nel sistema ROS, il package dove il file eseguibile si trova e il nome dell'eseguibile.
Per lanciare il nodo del simulatore Stage, ci si posiziona dentro la cartella che si trova nella directory workspaces/srrg2_labiagi/src/srrg2_navigation_2d/config.
In questa cartella si trovano diversi file, tra cui i file .world che descrive il mondo in cui si vuol far navigare il robot. Per lanciare il simulatore, dunque, si esegue il comando rosrun stage_ros stageros <file.world> su un nuovo tab del terminale.
Il comando è composto dalla parola chiave *rosrun*, che, come detto prima, è necessario per lanciare i nodi ROS, il nome del package dove il file del simulatore si trova (ovvero stage_ros), il nome dell'eseguibile e il nome del file che descrive il mondo.
Come detto precedentemente, il sistema anticollisione è stato pensato per navigare all'interno del mondo 'cappero_laser_odom_diag_obstacle_2020-05-06-16-26-03.world', ma nulla vieta di testarlo anche in altri mondi.
Come ultima cosa, si avvia il nodo collision_avoidance. Per lanciare il nodo, ci si sposta nella directory del workspace Collision_Avoidance. Si compila poi il workspace sempre con il comando 'catkin_build' (ricordiamo che tale istruzione deve essere eseguita ogni qualvolta si effettui una modifica, per rigenerare i file eseguibili

corretti). Successivamente ci si colloca nella directory in cui si trova il file C++ che descrive il sistema, seguendo il percorso src/collision_avoidance/src.
Il comando per lanciare il nodo è il seguente: rosrun collision_avoidance collision_avoidance.
Dopo aver eseguito queste istruzioni, sarà possibile testare il funzionamento del sistema, controllare la base mobile attraverso il proprio joystick e vedere come vengono evitate le collisioni.

3.6 Strumenti utili per il debug

In questa sezione sono illustrati i principali tool per eseguire il debug:

- il primo comando, già citato nella tesi, è *Rqt_Graph*, che è possibile eseguire lanciando da terminale la seguente istruzione:
  rosrun rqt_graph rqt_graph;

- un altro comando utile è il seguente:
  rostopic pub [-r] [frequenza(Hz)] <nome_topic> <tipo_messaggio>
  che permette la pubblicazione su un determinato topic del sistema, specificandolo in <nome_topic>, messaggi del tipo <tipo_messaggio>, specificando se deve essere inviato un solo messaggio oppure più messaggi con una certa frequenza;

- infine, se l'utente volesse mettersi in ascolto su un certo topic, in modo da visualizzarne i messaggi, è possibile utilizzare il comando:
  rostopic echo <nome_topic> [-nNUM]
  dove si specifica il nome del topic in <nome_topic> e il numero degli ultimi messaggi che si vogliono leggere in [-nNUM].

Capitolo 4

Esperimenti e Risultati

In questo capitolo verranno esposti i problemi affrontati e gli esperimenti effettuati con i relativi risultati. Verranno elencati, infine, i limiti del sistema anticollisione proposto.

4.1 Esperimenti e problemi affrontati

Nella seguente sezione verranno mostrati i vari problemi affrontati durante gli esperimenti nell'ambiente di simulazione.

4.1.1 Problema dei valori da considerare e del valore delle forze

Come spiegato anche nel precedente capitolo, il laser montato sulla base mobile scansiona l'ambiente con un'ampiezza frontale di 270°. Avere una vista più ampia del normale (solitamente, l'angolo massimo che l'occhio umano può vedere ha un'ampiezza di circa 180°) non causa alcun nessun effetto negativo. Al contrario, aumenta il livello di controllo del sistema.
L'algoritmo implementato, permette di rivelare ed evitare nel modo migliore possibile solo ed esclusivamente gli ostacoli che si trovano di fronte alla base mobile ed entro una certa ampiezza, e non anche gli ostacoli laterali (che, di fatto, non ostruiscono il passaggio del robot). Ciò comporta che l'algoritmo deve usare solo il ventaglio dei valori frontali del laser. Per fare ciò, poiché il range dell'array del laser contiene 1081 valori per 270°, si è deciso di optare per una soluzione che considerasse due ulteriori array che rappresentassero i valori centrali e frontali: uno di 321 valori e uno più piccolo di 181 valori. In questo modo è stato possibile rendere la navigazione della base mobile molto più efficiente e fluida nel caso in cui fosse entrata all'interno di spazi molto stretti: in queste circostanze, infatti, il controllo viene fatto su due diversi array, in modo da permettere al robot di entrare nello spazio stretto e, allo stesso tempo, di capire se, una volta entrato, avesse avuto lo spazio per girare su sé stesso.
Come conseguenza, le forze repulsive considerate *non* sono tutte quelle rivelate nell'intorno di 270°, ma *solo* quelle considerate "frontali". L'angolo che determina l'ampiezza dei valori centrali considerati è stato scelto, dunque, in modo tale da permettere alla base mobile di passare anche nei tratti molto stretti, ovvero percorsi

contenenti ostacoli da entrambi i lati, ma con una distanza tra loro che fosse più grande della larghezza della base mobile, ovvero abbastanza grande da permetterne il passaggio. Uno di questi spazi è quello tra i banchi dell'Aula Magna del DIAG. Dopo vari esperimenti svolti, l'ampiezza ottima è stata scelta pari a 50°.
Nel codice, però, come riportato sopra, viene considerato anche un secondo array con un range di valori relativi ad un'ampiezza maggiore (circa 90°). Quest'ultimo è servito, in particolare, per permettere alla base mobile di diminuire la propria velocità quando questa si avvicinava troppo agli ostacoli, rendendo la navigazione molto più fluida. Nei primi esperimenti, infatti, senza questo decremento, poteva succedere che la velocità fosse talmente alta da non permettere alla base mobile di riuscire a fermarsi, andando a collidere contro gli ostacoli, anche se la situazione era stata analizzata correttamente. In poche parole, considerando solo il primo range di valori, la base mobile sarebbe anche riuscita a reagire positivamente evitando la collisione, ma senza avere abbastanza spazio per modificare la propria traiettoria, fallendo dunque nel tentativo. Il principale motivo della considerazione di questo range più ampio di valori, quindi, è dovuto proprio al necessario decremento della velocità lineare, in modo da far sì che la base mobile avesse il tempo e lo spazio necessario per reagire correttamente. In realtà, grazie a questo secondo array considerato, si è ottenuto non solo un miglioramento nel comportamento della base mobile, ma anche una elusione degli ostacoli e una navigazione molto più fluida (o, come direbbero gli anglo-americani, più *smooth*). Successivamente, si è deciso di confermare e implementare questa scelta, finché dopo vari esperimenti non si è ottenuto l'ottimo valore tra navigazione fluida e soddisfacimento dei requisiti.

Nelle prime fasi dello sviluppo dell'algoritmo, il valore della forza repulsiva causato da un ostacolo i veniva calcolato come:

$$forza\_repulsiva[i] = k_1 \cdot distanza\_ostacolo[i]. \tag{4.1}$$

La variabile *distanza_ostacolo[i]* nell'equazione 4.1, calcolava inizialmente il complemento a 1 della distanza tra l'ostacolo e la base mobile, cioè:

$$distanza\_ostacolo[i] = 1 - laser[i], \tag{4.2}$$

dove con *laser[i]* ci si riferisce al valore della distanza che il laser legge per l'ostacolo i. Durante gli esperimenti si è notato, però, che in alcune situazioni il comportamento non era quello previsto. Più precisamente, vi erano casi in cui la base mobile non riusciva ad evitare gli ostacoli che si trovavano davanti ad essa. Questo malfunzionamento era evidente quando dall'altra parte vi erano molti ostacoli che portavano la forza repulsiva totale ad avere una direzione che spingeva la base mobile ad andare verso l'ostacolo meno influente in termini di distanza.
La soluzione a questo problema è stata quella di calcolare il valore della forza repulsiva per ogni ostacolo secondo una *legge esponenziale*. Questo ha portato ad avere sì un valore della forza repulsiva quasi uguale a quello precedente per gli ostacoli più distanti, ma, allo stesso tempo, ha portato anche ad avere un aumento considerevole del valore della forza repulsiva per gli ostacoli più vicini. Dopo molteplici di esperimenti, il grado dell'esponente che è sembrato permettere alla base mobile di avere un comportamento il più adeguato possibile è stato fissato pari a 5. Perciò, la formula con cui si calcola la forza repulsiva nel codice è stata modificata in

$$forza\_repulsiva[i] = k_1 \cdot (distanza\_ostacolo[i])^5 \tag{4.3}$$

La formula per calcolare invece la forza attrattiva è:

$$forza\_attrattiva = k_2 \cdot velocita\_lineare\_joystick_x \tag{4.4}$$

I coefficienti k_1 e k_2 sono stati scelti tali per cui la forza repulsiva avesse un valore quanto più simile possibile alla forza attrattiva.

4.1.2 Problema della situazione di "trappola"

Un altro problema comparso durante lo sviluppo è stato quello della situazione di "*trappola*" per la base mobile. Il nome è stato scelto proprio perché la situazione sembra inizialmente del tutto normale, con il robot che riesce a passare nello spazio stretto, salvo però trovarsi poi in una posizione in cui la collisione risulta inevitabile. Questa situazione si verifica in particolar modo quando la base mobile entra in percorsi stretti, senza via d'uscita, con ostacoli simmetrici o quasi e in cui risulta impossibile evitare la collisione. Analizzando la forza repulsiva totale generata in questi casi, si è notato che questa aveva verso contrario allo spostamento della base mobile e della velocità lineare. Il suo valore, però, era comunque più piccolo di quello della forza che abbiamo definito attrattiva, non influenzando dunque né la direzione di navigazione della base mobile, né tanto meno l'efficace riduzione della velocità. Quando ci si trovava in questa situazione, la base mobile continuava ad andare verso gli ostacoli frontali, anche se con velocità decrescente, fino, però, ad arrivare al momento della collisione, senza riuscire a fermarsi.
La soluzione a questo problema è stata quella di considerare tre possibili situazioni in cui la base mobile si sarebbe potuta trovare: una situazione di completa sicurezza, **SAFE**, una situazione di particolare attenzione, **WARNING**, e una situazione estrema, **DANGER**, che poteva risolversi in **TRAP** se non si fosse trovata alcuna via d'uscita. Quando il robot si trova nello stato SAFE, l'algoritmo non viene azionato. Il robot può eseguire qualsiasi comando ricevuto dall'utente. Nella figura 4.1 viene mostrata una situazione in cui la base mobile si trova in uno stato SAFE.

Figura 4.1. Possibile situazione in cui la base mobile si trova in uno stato SAFE.

Nel momento in cui la base mobile si avvicina ad un ostacolo, quando si supera una certa distanza da esso, che definiamo *distanza di sicurezza* si entra nella modalità WARNING. In questo caso l'algoritmo entrerà in azione modificando considerevolmente la velocità angolare e lineare del robot, deflettendone la traiettoria. Ecco un esempio, nella figura 4.2, in cui la base mobile si trova in uno stato WARNING.

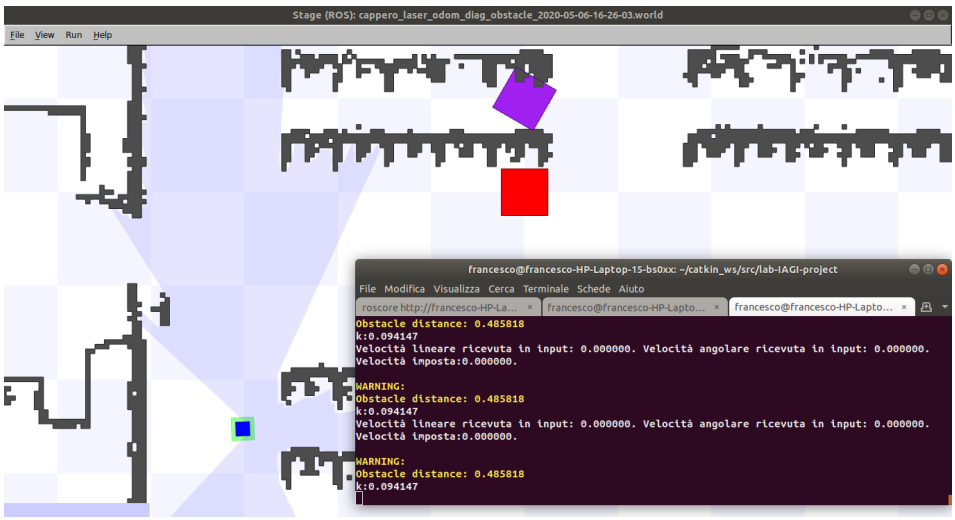

Figura 4.2. Possibile situazione di WARNING per la base mobile.

Se, invece, il robot si fosse trovato ad una distanza minore della distanza di sicurezza dall'ostacolo, diciamo *qualche valore in più* della *distanza di pericolo* (ovvero quella che la base mobile deve mantenere dagli ostacoli in modo da potersi arrestare e girare senza che ci siano collisioni), allora questo sarebbe entrato in modalità DANGER. Nella figura 4.3 viene riportato un esempio.

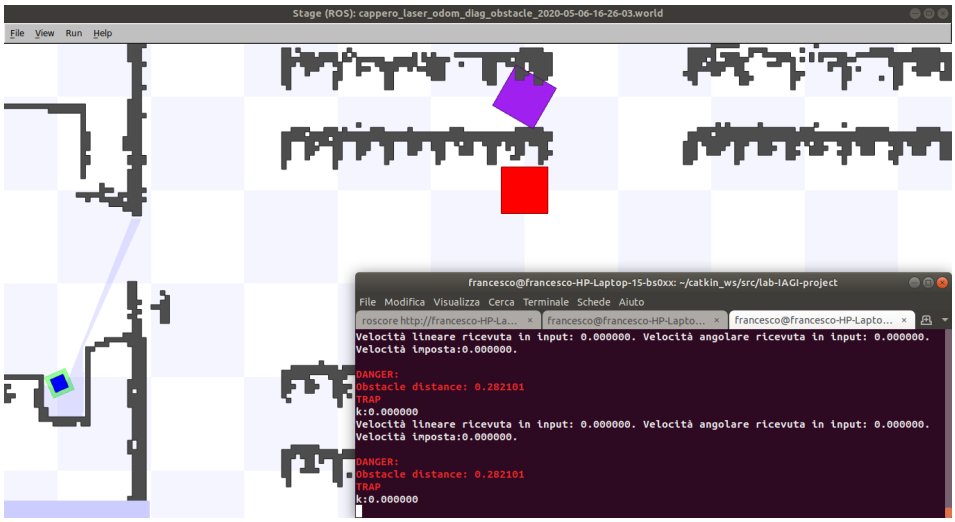

Figura 4.3. Base mobile in modalità DANGER/TRAP.

Anche in questo caso, per evitare la collisione, il comportamento dell'algoritmo cambia. Appena la base mobile entra in questa zona, l'algoritmo cerca di capire da

quale lato si trova l'ostacolo più lontano analizzando l'array relativo allo scanner laser. Se si rileva un ostacolo più lontano dagli altri, il robot inizia a girare su sé stesso con una velocità angolare costante e lineare nulla verso la zona con meno ostacoli finché si ritrova di nuovo nella zona di sicurezza (SAFE) o di minor pericolo (WARNING) dove riprende il suo comportamento a seconda del nuovo stato. Nel caso in cui non dovesse trovare una zona libera da ostacoli, la base mobile capisce di trovarsi nella situazione di TRAP. Ecco un esempio nella figura 4.4

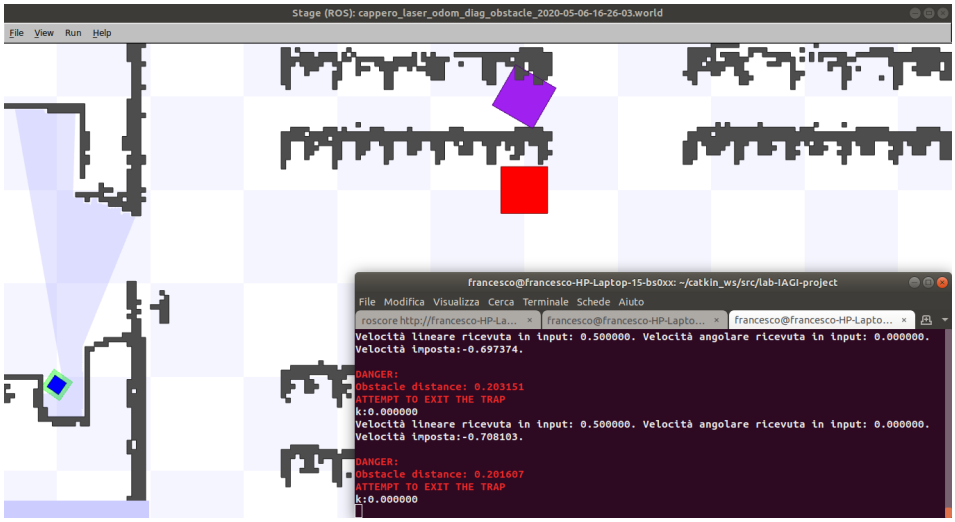

Figura 4.4. Situazione di TRAP dalla quale la base mobile tenta di uscire.

In questa situazione, la base mobile non va né avanti, né indietro, ma gira su sé stessa (di default verso sinistra, ma se si manda il comando di velocità angolare verso destra, il robot cambierà direzione) finché non trova uno spazio libero da ostacoli, uscendo dalla zona di TRAP (figura 4.5).

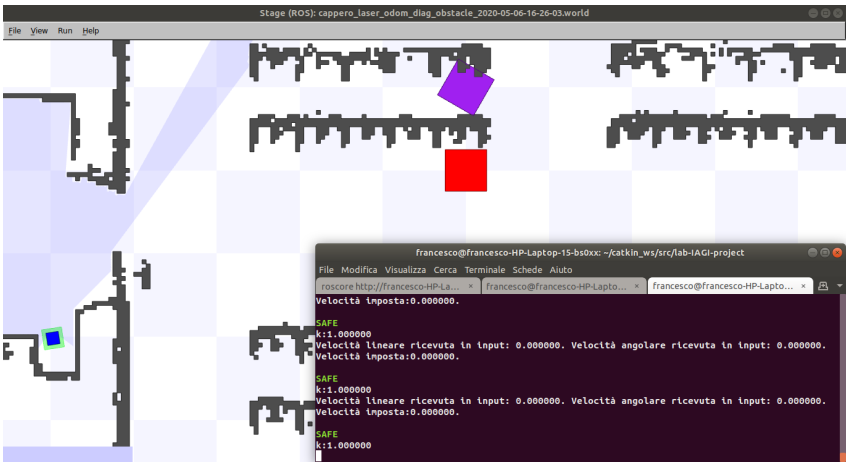

Figura 4.5. La base mobile è uscita dalla situazione di TRAP.

4.2 Limiti del Sistema

Supponendo che l'ostacolo non sia così grande da violare i vincoli cinematici e dinamici del robot e che il sistema sperimentale funzioni correttamente, l'algoritmo di evasione degli ostacoli dovrebbe funzionare in tutti i casi. In questa sezione, verranno discussi i limiti dell'algoritmo per evitare gli ostacoli.

4.2.1 Presenza di ostacoli in movimento

Prendiamo come esempio il caso in cui ci sia un ostacolo in movimento.
L'algoritmo è in tempo reale, i movimenti degli ostacoli non sono noti (supponiamo che i limiti di movimento degli ostacoli siano simili a quelli del robot) ed essi possono cambiare velocità durante il movimento.
Con l'algoritmo sviluppato, è possibile che il robot non riesca ad evitare le collisioni se la velocità dell'ostacolo è superiore a quella del robot. Ipotizziamo che la base mobile stia andando verso un determinato punto B, nelle cui vicinanze si trova un ostacolo. Quando il robot arriva al punto B, se l'ostacolo iniziasse ad aumentare la sua velocità verso il robot con una velocità massima maggiore di quella del robot, quest'ultimo non sarebbe in grado di evitare una collisione con l'ostacolo.
Da quanto detto, concludiamo che per evitare il singolo ostacolo, la velocità massima dell'ostacolo deve essere inferiore (o, al massimo, uguale) a quella del robot.

4.2.2 Presenza di ostacoli multipli

Un altro caso può essere quello dell'evitare ostacoli multipli. Se la velocità dell'ostacolo è limitata come detto sopra, l'algoritmo sviluppato può effettivamente evitare collisioni con singoli ostacoli. Tuttavia, per più ostacoli, ci sono ancora alcuni casi in cui il robot non può evitare collisioni con l'ostacolo. Un caso è che il robot aumenta la sua velocità angolare per evitare collisioni con un ostacolo. Quando il robot è alla sua velocità massima nella direzione, un secondo ostacolo appare nel raggio di collisione. In questo caso il robot non riesce a fermarsi e ciò si traduce nel fallimento della prevenzione delle collisioni con il secondo ostacolo.
Queste limitazioni sono dovute al fatto che i movimenti degli ostacoli non sono preconosciuti e il movimento del robot non è istantaneo, ma progressivo.

4.2.3 Limiti nel mondo reale

Le limitazioni del sistema arrivano anche e soprattutto nel momento in cui usciamo fuori dall'ambiente di simulazione.
Nel mondo reale, infatti, potrebbero apparire parecchie difficoltà a causa delle forze esterne, ma non solo.
Pensiamo ad esempio se volessimo abbellire il nostro robot con dell'attrezzatura extra, magari con un gagliardetto, o delle luci ausiliarie. Esse potrebbero ostruire il laser scanner. Oppure, si pensi ad un fondo stradale scivoloso, in cui lo spazio di frenata viene esteso: ridurrebbe la capacità della funzione di evitare una collisione.
Essendo il robot multidirezionale, inoltre, potrebbe succedere che un oggetto si posizioni nelle immediate vicinanze del robot, e un eventuale aumento della velocità angolare potrebbe non permettere il soddisfacimento dell'evitamento dell'ostacolo.

In queste situazioni impegnative la funzione di anticollisione può incontrare difficoltà, ma, poiché il robot è controllato da remoto, l'abilità dell'utente può comunque essere d'aiuto nel prevenire le collisioni.

Capitolo 5

Conclusioni

In questa tesi ho voluto proporre una possibile soluzione al problema di anticollisione per robot controllati da remoto. Il sistema proposto permette ad una base mobile su cui è montato uno scanner laser di navigare in un ambiente chiuso e sconosciuto evitando i possibili ostacoli statici e dinamici. Ovviamente, questa soluzione, che si applica solo su sistemi 2D e attrezzati con scanner laser, non copre tutti i possibili campi destinati all'utilizzo, ma può essere implementato per rendere possibile l'applicazione anche in altri casi e sono molti i perfezionamenti possibili: gli sviluppi più sofisticati nell'evitare le collisioni, infatti, vengono raggiunti combinando le informazioni provenienti da più sensori e sistemi.

Ad esempio, l'installazione di una telecamera frontale, e quindi l'aggiunta di un'ulteriore assistenza visiva, può essere utilizzata per migliorare l'operabilità. Per citare un altro esempio, in quasi tutti i sistemi di navigazione, vi è la possibilità che la base mobile possa muoversi a marcia indietro con una certa velocità. Anche nel sistema presentato è possibile far andare in retromarcia il robot, ma senza prestazioni di sicurezza, ovvero senza che il sistema anticollisione funzioni correttamente. L'aggiunta del sistema anticollisione anche per la situazione della retromarcia può avvenire montando altri laser o aumentando l'ampiezza del laser scanner fino a 360°. Facendo così, viene reso possibile il controllo di tutti gli ostacoli, includendo anche quelli che si trovano dietro la base mobile.

Una delle possibili applicazioni del sistema anticollisione proposto, oltre quella dei settori industriali e della cura della salute, è quella del settore automobilistico. Infatti, basti pensare al numero e tipo di incidenti stradali commessi nel 2021, come mostrato nella seguente immagine (figura 5.1). Come si può vedere, tra le principali cause dei sinistri stradali ci sono: guida distratta o andamento indeciso (23.802 sinistri; il 15,7% del totale), mancato rispetto della distanza di sicurezza (13.148 sinistri, l'8,7% del totale), mancata precedenza al pedone (4.838 sinistri, il 3,2% del totale), ecc. Questi tipi di incidenti potrebbero sicuramente essere evitati se si montasse sulle automobili un sistema di anticollisione, portando ad un enorme risparmio di denaro anche sulla sanità pubblica.

Figura 5.1. Fotografia degli incidenti stradali [28].

Si provi, infine, a guardare al traffico e alle collisioni in orbita: un'altra applicazione potrebbe essere proprio quella relativa al settore aerospaziale. Si veda, in particolare, la figura 5.2, nella quale viene rappresentata la massa di oggetti in orbita terrestre dal 1958 ad oggi.

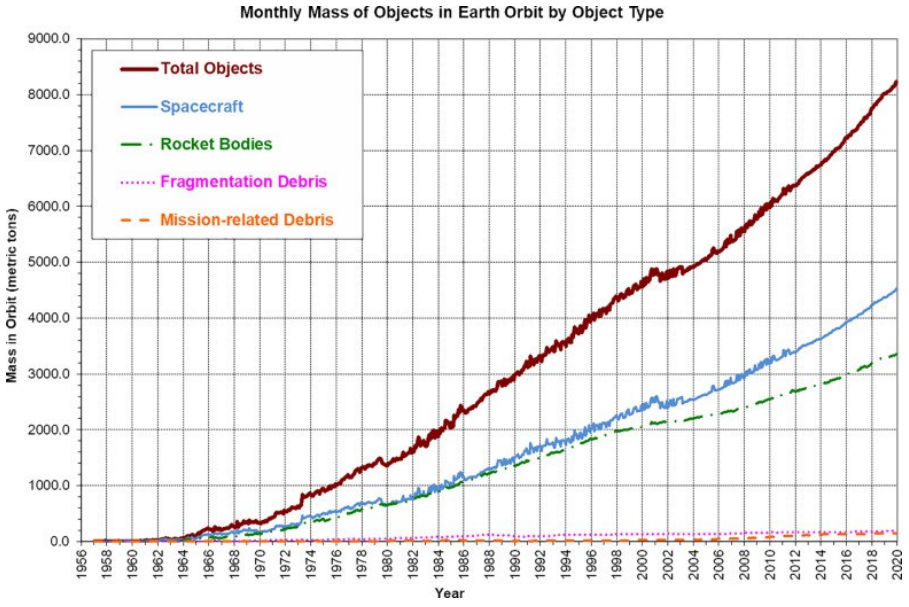

Figura 5.2. Evoluzione del numero di satelliti e detriti in orbita terrestre dal 1958 ad oggi. Non è considerato l'incremento di oltre 1000 detriti provocato dal test ASAT russo del 2021 [29].

Una collisione in orbita rappresenta un evento disastroso, non solo per il danno economico relativo alla distruzione di un satellite, ma soprattutto per la quantità di detriti che tale impatto può generare, i quali a loro volta diventano potenziali

inneschi per altre collisioni, dando origine ad un catastrofico effetto domino che, se non controllato, può portare alla saturazione delle regioni orbitali d'interesse scientifico e commerciale. Il problema è, in realtà, ben noto già da diversi anni con il nome di Sindrome di Kessler e la soluzione potrebbe essere proprio quella dell'installazione di un sistema anticollisione su ogni satellite, in modo da prevenire urti anche nello spazio.

In conclusione, il sistema proposto ha davvero svariati campi di applicazione che spaziano dall'ambito automobilistico, a quello sanitario, a quello per la cura della vita, alla risposta di disastri o all'esplorazione di aree pericolose, fino ad arrivare al settore aerospaziale e tale sistema può realmente rendere più semplice e sicura la vita e il lavoro delle persone.

Bibliografia

[1] Yi J.-B., Kang T., Song D., Yi S.-J, *Unified Software Platform for Intelligent Home Service Robots*, Appl. Sc., 2020.

[2] Joon A., Kowalczyk W., *W. Design of Autonomous Mobile Robot for Cleaning in the Environment with Obstacles,* 2021.

[3] Ruan K., Wu Z., Xu Q., *Smart Cleaner: A New Autonomous Indoor Disinfection Robot for Combating the COVID-19 Pandemic*, 2021, mdpi.com/2218-6581/10/3/87.

[4] Viacheslav Pshikhopov, *Path Planning for Vehicles Operating in Uncertain 2D Environments*, Butterworth-Heinemann, 2017.

[5] Biagiotti L., Melchiorri C., *Trajectory Planning for Automatic Machines and Robots*, Spring, 2008.

[6] Zivkovic, Z., Kröse, B, *People Detection Using Multiple Sensors on a Mobile Robot*, Spring, 2008.

[7] Vijayalakshmi M., *ICT for Competitive Strategies*, CRC Press, 2020, pp.81-90.

[8] Sayers C., *Remote Control Robotics*, Spring, 1999.

[9] Cook G., *Mobile Robots: Navigation, Control and Remote Sensing*, Wiley-IEEE Press, 2011.

[10] Bing Z., Kaige W., *A New Remote Health-Care System Based on Moving Robot Intended for the Elderly at Home*, Denise Ferebee, 2017.

[11] Junqiang Z., Ziyang M., Jia M., *A Remote-Controlled Robotic System with Safety Protection Strategy Based on Force-Sensing and Bending Feedback for Transcatheter Arterial Chemoembolization*, 2020, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7569875/.

[12] Farabee K., Itmamul Haque A., *Night Patrolling Robot*, 2021, 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST) https://ieeexplore.ieee.org/document/9331198.

[13] Shashank G., Pierre L., Keshav C., Jeremi G., Mario Nunez J., *Command and Control Systems for Search and Rescue Robots*, IntechOpen, 2017.

[14] Sadia S., Saiful Islam S.; Kazi Ragib Ishraq S., *A Low-Cost Urban Search and Rescue Robot for Developing Countries*, 2019, 2019 IEEE International Conference on Robotics, Automation, Artificial-intelligence and Internet-of-Things (RAAICON) https://ieeexplore.ieee.org/document/9087495.

[15] R. Masaki and N. Motoi, *Remote Control Method With Force Assist Based on Time to Collision for Mobile Robot*, in IEEE Open Journal of the Industrial Electronics Society, vol. 1, pp. 157-165, 2020, doi: 10.1109/OJIES.2020.3013249.

[16] R. Masaki and N. Motoi, *Remote Control Method With Force Assist Based on Time to Collision for Mobile Robot*, in IEEE Open Journal of the Industrial Electronics Society, vol. 1, pp. 157-165, 2020, doi: 10.1109/OJIES.2020.3013249.

[17] Khurshid RP, Fitter NT, Fedalei EA, Kuchenbecker KJ, *Effects of Grip-Force, Contact, and Acceleration Feedback on a Teleoperated Pick-and-Place Task*, IEEE Trans Haptics. 2017 Jan-Mar;10(1):40-53. doi: 10.1109/TOH.2016.2573301. Epub 2016 May 26. PMID: 27249838.

[18] Definition of 'robot', *Oxford English Dictionary*.

[19] Ng Andrew, Gould Stephen, Quigley Morgan, Berger, Eric. *STAIR: The STanford Artificial Intelligence Robot project*, Snowbird Workshop, 2008.

[20] Keenan Wyrobek, *Personal Robotics Program Fund Fundraising Deck from 2006*, 2017.

[21] *ROS 1.0 – ROS robotics news*, Open Robotics. Retrieved 29 April 2019. ROS.org

[22] *Osrf – Ros @ Osrf*, 11 February 2013. Retrieved 12 July 2014, Osrfoundation.org

[23] *ROS running on ISS – ROS robotics news*, Open Robotics. Retrieved 12 December 2017, ROS.org

[24] Sito Ufficiale ROS, ROS.org

[25] RViz Official Documentation, http://wiki.ros.org/rviz

[26] Rqt Official Documentation, http://wiki.ros.org/rqt

[27] Stage Ros Official Documentation, http://wiki.ros.org/stage

[28] Sicurauto, sicurauto.it/news

[29] *Orbital Debris, Quarterly news*, National Aeronautics and Space Administration, vol. 23, Issue 1 & 2, May 2019.