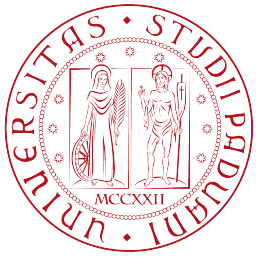


UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

COSTRUZIONE E TESTING IN GAZEBO E V-REP DI UN MODELLO PER IL ROBOT UMANOIDE NAO

RELATORE: Prof. Emanuele Menegatti

CORRELATORE: Dott. Stefano Michieletto

LAUREANDO: *Davide Zanin*

ANNO ACCADEMICO 2012/2013

Ai miei genitori, che mi hanno sostenuto in questi anni di studio.

Al mio relatore, prof. Emanuele Menegatti.

Al dott. Stefano Michieletto, per l'aiuto nella realizzazione di questo lavoro.

A tutti i compagni di corso di questi cinque anni.

Sommario

Nella *robotica* l'importanza degli ambienti di simulazione è costante in aumento. Rispetto al mondo reale, la simulazione di un robot fornisce molti vantaggi: nessuna limitazione al numero di robot per mancanza di fondi, assenza di imprecisione in sensori e attuatori, possibilità debug e nessun rischio di danneggiare il robot reale. Queste caratteristiche sono ancora più importanti quando si lavora con un *robot umanoide*. Muovere un umanoide significa tener conto di almeno 10 gradi di libertà (DoF) mantenendo nel frattempo la stabilità del robot. Qualche caduta è inevitabile e, in un caso reale, probabilmente comporterebbe un danno al robot o ai suoi sensori.

In questo lavoro è descritta la realizzazione di un modello di un robot umanoide, il *NAO*, e la sua implementazione in due simulatori: *Gazebo* e *V-REP*. Sfruttando le potenzialità messa a disposizione da *ROS*, il modello e i due simulatori vengono testati confrontando il comportamento del robot simulato con il comportamento del robot reale.

Indice

Introduzione

La robotica è la scienza che si occupa della progettazione, costruzione e del funzionamento dei robot, e dei sistemi informatici per il loro controllo e per l'elaborazione delle informazioni provenienti dai loro sensori. Queste tecnologie permettono a delle macchine automatiche di prendere il posto degli esseri umani in ambienti pericolosi o in processi di fabbricazione oppure di assomigliare agli esseri umani nell'aspetto, nel comportamento o nella cognizione. L'idea di creare delle macchine in grado di operare in modo autonomo risale all'epoca classica, ma la ricerca nelle funzionalità e potenzialità di utilizzo dei robot è cresciuta notevolmente solo a partire dagli inizi del 20° secolo. Oggi la robotica è un campo in rapida crescita con la ricerca, la progettazione e la costruzione di nuovi robot in grado di svolgere compiti pratici in ambienti domestici, industriali, commerciali e militari.

Gli ambienti di simulazione in robotica sono fondamentali: nessun rischio di danneggiare il robot reale, nessuna limitazione al numero di robot per mancanza di fondi, assenza di imprecisione dei sensori e attuatori e possibilità di debugging. Costruire un simulatore è però molto complesso. È necessario disporre di un motore grafico che renderizzi in mondo realistico il mondo simulato in modo da poter testare in simulazione anche gli algoritmi di visione, e di un motore fisico che simuli i movimenti del robot in modo accurato. Entrambi i motori richiedono conoscenze avanzate per essere sviluppati in modo efficiente, il problema principale è che il software non deve risultare troppo complessa pena una simulazione lenta o l'impossibilità di utilizzare più robot contemporaneamente. Per questo motivo spesso i simulatori si appoggiano a motori grafici e fisici sviluppati separatamente come ad esempio *OGRE* [1], *ODE* [2], *Bullet* [3] e *PhysX* [4]. Avere un simulatore completo però non è sufficiente per una simulazione realistica. Ogni robot che si intende utilizzare deve essere modellato all'interno del programma ed è proprio su questo problema che si concentra questa tesi. In particolare viene descritta la costruzione del modello del robot umanoide *NAO* [5] e la sua simulazione in *Gazebo* [6] e *V-REP* [7] sfruttando le potenzialità di *ROS* [8].

La tesi è organizzata in questo modo. Nel capitolo 1 è presente una panoramica sui robot umanoidi (NAO compreso), su ROS e sui simulatori utilizzati. Nel capitolo 2 è descritta la creazione del modello mentre nel capitolo 3 è discussa l'integrazione di Gazebo e V-REP in ROS e l'importazione nei due simulatori del modello descritto nel capitolo precedente. Nel capitolo 4 è descritto un confronto fra i due software e una comparazione tra il comportamento del robot reale e il comportamento del robot simulato dei due programmi. Le conclusioni e gli

sviluppi futuri si trovano nel capitolo 5. Infine in appendice si trovano le istruzioni per l'istallazione dei principali software utilizzati in questa tesi (Appendice A), alcuni esempi di programmazione del NAO usando ROS (Appendice B), le istruzioni usate per l'esecuzione dei test (Appendice C) e tutti i dati ottenuti nei test (Appendice D).

Capitolo 1

Robot umanoidi, ROS e simulatori

Un *robot umanoide* è un robot costruito in modo che la sua struttura assomigli a quella del corpo umano. In genere questi robot sono composti da un tronco, una testa, due braccia e due gambe ma in alcuni sono modellate solo alcune parti del corpo umano, ad esempio dalla vita in su. La differenza sostanziale con gli *androidi* è che quest'ultimi sono progettati con l'intento di assomigliare esteticamente agli esseri umani. Sono già stati prodotti alcuni androidi che risultano indistinguibili da un essere umano, come ad esempio quelli prodotti da Hiroshi Ishiguro dell'università di Osaka[1]. Questi robot sono costruiti con l'intento ingannare l'essere umano in modo che creda, per lo meno per qualche minuto, di essere davanti ad un essere umano.

I robot umanoidi sono usati in molti campi della ricerca scientifica. I ricercatori devono comprendere la struttura del corpo umano e il suo comportamento biomeccanico per costruire questi robot e questo porta ad aumentarne la comprensione del funzionamento. Inizialmente l'obiettivo di questi robot era migliorare la costruzione di ortesi e protesi, poi con il passare del tempo le conoscenze si sono trasferite dalla medicina alla robotica e viceversa.

Oltre alla ricerca, i robot umanoidi sono attualmente sviluppati con lo scopo di eseguire attività umane come l'assistenza di persone malate oppure per sostituire l'uomo in lavori pericolosi. Dotare un robot umanoide di software avanzato lo rende in grado di svolgere qualsiasi compito che un essere umano può effettuare. Tuttavia la complessità nel realizzare tutto questo è enorme, ciononostante negli ultimi anni sono stati fatti enormi passi in avanti.

Uno dei progetti più interessanti in questo senso è *Robonaut*[2]. Questo robot è costituito da un torso, una testa e due braccia, ed è pensato per lavorare assieme agli astronauti aiutandoli nel loro lavoro oppure assolvendo compiti pericolosi o banali ma dispendiosi in termini di tempo. La seconda versione di questo robot, *Robonaut 2* (*R2*), è stata inviata nel febbraio 2011 sulla *ISS* (*International Space*

[1] `http://www.geminoid.jp/en/`.
[2] `http://robonaut.jsc.nasa.gov`.

Station). Nell'ottobre dello stesso anno, R2 si sposta per la prima volta, mentre è nello spazio. Le condizioni a bordo della stazione spaziale forniscono un banco di prova per il robot permettendogli di lavorare con altre persone in condizioni di microgravità. Una volta verificato il corretto comportamento del robot, potranno essere aggiunti gli aggiornamenti software e gli arti inferiori permettendo ad R2 di muoversi all'interno della stazione ed eseguire operazioni di manutenzione, come passare l'aspirapolvere o la pulizia dei filtri.

Negli ultimi anni i robot umanoidi che più si sono fatti notare in ambito internazionale sono: *ASIMO*, *HRP-4c* e *NAO*.

Figura 1.1: ASIMO (a sinistra), HRP-4c (al centro) e NAO (a destra).

ASIMO[3] è prodotto da *Honda* e nasce nel 2000 con l'intento di aiutare persone con difficoltà motorie ma anche incoraggiare i giovani a studiare le materie scientifiche. Nell'ultima versione presentata nel 2012, il robot è alto 130 cm, pesa 48 kg e ha 57 gradi di libertà (*DoF*, *Degree of Freedom*[4]). Negli USA dal giugno 2005 è parte delle attrazioni di *Disneyland* con uno spettacolo di 15 minuti intitolato *"Say 'Hello' to Honda's ASIMO"*. Tra le capacità di ASIMO sono da segnalare: la camminata e la corsa molto simili a quelle umane, la capacità di salire e scendere le scale, calciare palloni, riconoscere oggetti in movimenti e volti umani; il riconoscimento vocale gli permette inoltre di distinguere fra voci, suoni, oggetti che cadono o in collisione. Tutte queste funzionalità permettono al robot di interagire in modo convincente con un essere umano.

HRP-4c è un robot umanoide sviluppato dal *National Institute of Advanced Industrial Science and Technology* (AIST)[5], un ente di ricerca giapponese. È alto

[3] `http://asimo.honda.com/`.

[4] In meccanica si definisce il numero di gradi di libertà di un punto materiale come il numero di variabili indipendenti necessarie per determinare univocamente la sua posizione nello spazio. Quando si parla di robot, quindi il numero di gradi di libertà indica il numero di variabili indipendenti necessarie per specificare la posizione di tutti i giunti di cui è composto; se ad esempio un robot è composto da una sbarre che può compiere una rotazione rispetto l'asse x, una rotazione rispetto l'asse y e una traslazione lungo l'asse z, allora è composto da 3 DoF.

[5] `http://www.aist.go.jp/`.

158 cm, pesa 43 kg e ha una testa e un volto molto realistici con le sembianze di una ragazza giapponese, per questo motivo è soppranominato *Miim*. Miim può rispondere alle domande che le vengono poste usando un software di riconoscimento vocale, è in grado di riconoscere i suoni ambientali, può cantare usando il sintetizzatore vocale *Vocaloid*[6] ed esprimere otto espressioni facciali.

Al robot umanoide NAO è dedicato il prossimo paragrafo.

1.1 Il robot umanoide NAO

Il robot umanoide NAO è alto 58 cm, pesa circa 5 kg ed è sviluppato dalla *Aldebaran Robotics* [5]; lo sviluppo inizia nel 2004 e dal 2007 rimpiazza il cane robot *Aibo* della Sony nella *Robot Soccer World Cup*[7].

Nell'ultima versione NAO è composto da 25 gradi di libertà, presenta una CPU Intel Atom CPU @ 1.6GHz e comunica con i sistemi esterni attraverso una connessione Wi-Fi o Ethernet. Il robot dispone inoltre di molti sensori.

Telecamere una sulla fronte e una sulla bocca, entrambe da 920p e capaci di registrare a 30 fotogrammi al secondo; sono disponibili anche degli algoritmi in grado di riconoscere volti e forme (ad esempio una palla);

Sensori tattili sulla testa e nelle mani sono presenti dei sensori tattili che permettono al robot di interagire con gli essere umani, inoltre in ciascun piede è presente un respingente in grado di rilevare eventuali collisioni con l'ambiente;

Sonar nel busto è presente un sonar che permette di stimare la distanza di eventuali ostacoli da una distanza minima di 15 cm fino a 255 cm, sotto i 15 cm l'ostacolo viene rilevato ma non è possibile stimarne la distanza;

Inertial Measurement Unit permette di misurare la velocità e l'orientamento del robot utilizzando due giroscopi e un accelerometro;

Sensori di posizione ciascun giunto dispone di un sensore in grado di misurare la sua posizione con una precisione di circa 0.1°;

Altoparlanti due altoparlanti sono posizionati nelle orecchie e grazie al sintetizzatore vocale è in grado di comunicare in otto lingue;

Microfoni nella testa sono posizionati quattro microfoni uno per ciascuna direzione, inoltre calcolando il diverso tempo di arrivo di un suono ai quattro ricevitori è in grado di calcolarne la posizione di arrivo.

Per camminare NAO utilizza un semplice modello dinamico, il pendolo lineare inverso, e la programmazione quadratica [9]. La camminata è stabilizzata

[6] `http://www.vocaloid.com/en/`.

[7] `http://www.robocup.org/`.

utilizzando i dati provenienti dai sensori rendendola robusta e resistente ai piccoli disturbi permettendo inoltre di assorbire le oscillazione frontali e laterali del busto. Il robot può camminare su molte superfici tra cui piastrelle, moquette e pavimenti in legno, inoltre è in grado di superare piccoli ostacoli (inferiori ad un centimetro) e camminare senza cadere fino a pendenze di 5°. Il modulo di gestione del movimento si basa sulla cinematica inversa generalizzata e gestisce le coordinate cartesiane, il controllo dei giunti, l'equilibrio e la gestione della priorità delle attività. Questo significa che se ad esempio viene richiesto al NAO di estendere le braccia questo potrebbe non essere eseguito completamente perché, prendendo in considerazione anche le articolazioni delle gambe, il robot decide di bloccare il movimento pur di mantenere l'equilibrio.

NAO dispone di un *Fall Manager* che protegge il robot quando cade. La sua funzione principale è quella di rilevare quando il centro di massa del robot esce dalla base di supporto individuata dai piedi, o dal piede, in contatto con il terreno. Se ciò accade tutte le attività del robot vengono sospese, le braccia assumono una posizione protettiva stabilita in base alla direzione della caduta e la rigidità del robot viene ridotta a zero.

1.2 ROS

ROS (*Robot Operating System*) [8] [10] è un framework per lo sviluppo di software per robot e include alcune funzionalità simili a quelle fornite da un sistema operativo applicate però ad un cluster eterogeneo di computer. Con il nome di *Switchyard*, nel 2007 comincia lo sviluppo di ROS presso lo *Stanford Artificial Intelligence Laboratory*[8], dal 2008 lo sviluppo continua soprattutto a *Willow Garage*[9] un'ente di ricerca e incubatore nel campo della robotica.

ROS è rilasciato sotto i termini della licenza BSD, è un software open source, gratuito per ricerca e per fini commerciali. Per massimizzare la partecipazione della comunità, ROS utilizza un modello di repository *federato*: invece di utilizzare un unico server in cui sono caricati tutti i pacchetti, gli utenti e gli sviluppatori di tutto il mondo sono invitati ad ospitare i propri pacchetti nei propri repository. In questo modo lo sviluppatore può mantenere il pieno controllo del proprio codice, mantenendone inoltre la proprietà e permettendogli di concedere il software con la licenza preferita [11].

I principali vantaggi di ROS sono:

Astrazione dell'hardware attraverso le definizione di *nodi* che possono comunicare tra loro attraverso semplici messaggi, facilita lo sviluppo e l'integrazione di funzionalità in robot diversi;

Estendibilità la struttura a *package* e *stack* (collezioni di package) spinge lo sviluppatore a sviluppare il codice in modo ordinato e ne facilità la condivisione e il riuso anche da parte della comunità.

[8] `http://ai.stanford.edu/`.

[9] `http://www.willowgarage.com/`.

Portabilità il numero di robot supportati è in costante crescita, inoltre ROS garantisce il supporto a diversi linguaggi di programmazione (C++, Python e Java);

Supporto grazie alla wiki ufficiale e alla piattaforma *ROS Answers* eventuali errori e problemi possono essere facilmente individuati e risolti;

Per comprendere l'architettura di ROS quattro sono i concetti fondamentali: i nodi (*node*), i messaggi (*message*), i topic (*topic*) e i servizi (*service*).

Nodi sono processi che eseguono una computazione. ROS è progettato per essere modulare infatti un sistema è tipicamente composto da molti nodi. In questo contesto il termine "nodo" è intercambiabile con "modulo software". L'uso del termine nodo deriva dal grafico usato per rappresentare un sistema ROS in esecuzione, i processi vengono rappresentati come dei nodi e le comunicazione instaurate fra i nodi come archi in modo simile a quanto viene fatto per i sistemi *peer-to-peer*;

Messaggi i nodi comunicano fra loro scambiandosi dei messaggi. Un messaggio è una struttura dati strettamente tipizzata che può essere un tipo di dato primitivo (un intero, un numero in virgola mobile, un booleano) ma può essere anche più complesso, come un array di dati primitivi. Un messaggio può inoltre essere composto da altri messaggi o da array di altri messaggi eventualmente anche annidati;

Topic un nodo invia un messaggio semplicemente pubblicandolo in un determinato topic che viene identificato attraverso una stringa. Un nodo che è interessato a questi messaggi si sottoscriverà al topic associato. È possibile avere più nodi che si iscrivono o pubblicano nello stesso topic e un nodo può iscriversi o pubblicare in più topic.

Servizi anche se il modello di comunicazione basato sui topic è piuttosto flessibile, il suo schema di trasmissione non è appropriato per transizioni sincrone, per questo motivo in ROS sono presenti i servizi. Un servizio viene definito da un nome (una stringa) e una coppia di messaggi tipizzati: uno per la richiesta e uno per la risposta. In questo modo un nodo può fare una richiesta ad un altro nodo e aspettare una sua risposta. Si noti che, a differenza dei topic, un solo nodo può pubblicare messaggi in un certo servizio.

Due sono i comandi fondamenti in ROS: `rosmake` [12] e `roslaunch` [13]. Il primo compila il codice sorgente di un determinato package eventualmente compilando le dipendenze associate, mentre il secondo legge un file XML in cui è descritto quali sono i nodi che devono essere avviati e gli eventuali parametri o opzioni che devono essere impostati.

ROS dispone di molti tool [14] che aiutano lo sviluppo e il debugging dei propri programmi. Particolarmente potente è *Rviz*, un visualizzatore 3D che permette di combinare i dati provenienti da sensori, modelli di robot e altri dati

3D in un'unica vista. È possibile anche inserire dei marker nel proprio software in modo che i dati vengano inviati a Rviz.

L'attuale versione stabile di ROS, rilasciata il 31 dicembre 2012, è *Groovy Galapagos* [15]. Il lavoro presentato in questa tesi fa però riferimento alla versione precedente, *Fuerte Turtle* [16], perché quando è stato iniziato nel settembre 2012, Fuerte era la versione di riferimento e nei primi mesi dal rilascio di Groovy quest'ultima risultava piuttosto instabile. Nei prossimi mesi è previsto un porting di tutti i package sviluppi.

1.2.1 NAO e robot umanoidi

In ROS si trovano degli stack che permettono di interfacciarsi al robot *NAO* e che implementano alcune funzionalità specifiche di questo robot e più in generale di tutti i robot umanoidi.

Il software è sviluppato dallo *Humanoid Robot Lab*[10] presso l'Università di Friburgo ed è composto dagli stack `nao_robot`, `nao_common`, `humanoid_msg` e `humanoid_navigation`, i primi due permettono di controllare e simulare il NAO mentre per gli altri riguardano tutti i robot umanoidi.

Robot umanoidi

All'interno dello stack `humanoid_msgs` si trova il package `humanoid_nav_msgs` contenente tutte le definizioni dei tipi di messaggi e dei servizi che possono essere usati per interagire con un robot umanoide.

Nello stack `humanoid_navigation` [17] si trovano i seguenti package:

- `footstep_planner` permette di pianificare i movimenti che un robot umanoide o bipede deve conpiere per muoversi tra due punti delle spazio evitando eventuali ostacoli;
- `gridmap_2d` implementa una mappa 2D basata sulle matrici `cv::mat` di *OpenCV*[18];
- `hrl_kinematics` è una piccola libreria per robot umanoidi basata su KDL;
- `humanoid_localization` permette la localizzazione in 6D di robot umanoidi basandosi su dati di profondità forniti da strumenti quali laser o point cloud.

Footstep planner Il planner contenuto in questo stack è basato sul package `sbpl` e i tipi di planner supportati sono A*, ARA*, AD* [19]; maggiori dettagli sull'implementazione si possono trovare in [20] e [21].

È possibile interagire con il planner tramite Rviz fornendo la mappa in cui il robot si deve muovere, il punto iniziale in cui esso si trova e il punto che deve raggiungere. Inoltre il planner interagisce continuamente con il mondo reale

[10] `http://hrl.informatik.uni-freiburg.de/`.

ripianificandondo il percorso del robot in modo da evitare eventuali ostacoli che non erano segnalati nella mappa, in questo caso è però necessario attivare anche un sistema di localizzazione in modo da permettere al robot di stimare la propria posizione.

Un esempio di utilizzo si può trovare in `http://youtu.be/o0rlrEHN1w4`.

Humanoid localization Questo package fornisce una localizzazione di Monte Carlo 6D (x, y, z, *roll*, *pitch*, *yaw*) utilizzando i dati forniti da sensori di distanza come laser 2D oppure una depth camera; inoltre i dati provenienti dall'odometria e dall'IMU (*inertial measurement unit*) vengono fusi nel *particle filter*.

Attualmente sono presenti due modelli di osservazione entrambi basati su OctoMap come mappa 3D del mondo: Ray casting e un modello "end point" cioè basato sulla mappa in cui sono riportate le distanze degli oggetti visti dal robot; maggiori dettagli si possono trovare in [22].

Un esempio di utilizzo si può vedere in `http://youtu.be/uiIi2rSKWAU`.

NAO

Nello stack `nao_robot` [23] si trovano due package fondamentali per integrare il NAO in ROS: `nao_driver` e `nao_msgs`; il primo fornisce l'accesso ai comandi per la camminata, agli angoli dei giunti e ai dati dei sensori (come odometria, IMU e telecamera) mentre il secondo definisce le tipologie di messaggi che possono essere utilizzati per comandare il robot o per leggere i valori dei sensori.

Nello stack `nao_common` [24] si trovano dei package che estendono le funzionalità di `nao_robot`, questi package sono:

- `nao_description` contiene la descrizione del NAO usando *URDF*[11];
- `nao_teleop` insieme a `nao_remote` permette di teleoperare il robot attraverso un joypad.

`nao_description` La descrizione del modello contenuta in questo stack è molto semplificata. Sebbene siano riportate tutte le parti che compongono il robot, le loro proprietà fisiche non sono tutte impostate in modo corretto, inoltre non sono presenti delle mesh realistiche infatti le varie parti del robot vengono approssimate usando dei parallelepipedi e dei cilindri (Fig. 1.2). Per questi motivi non è possibile effettuare una simulazione del robot che tenga conto di tutte le caratteristiche fisiche del mondo reale riducendo molto le possibilità di utilizzo del modello. I dettagli sono presentanti nei prossimi capitoli mentre si procederà con la descrizione del modello che è stato sviluppato.

[11] *Unified Robot Description Format* è un formato XML usato all'interno di ROS per descrivere il modello di un robot; questo formato verrà descritto nel dettaglio nel capitolo 2.

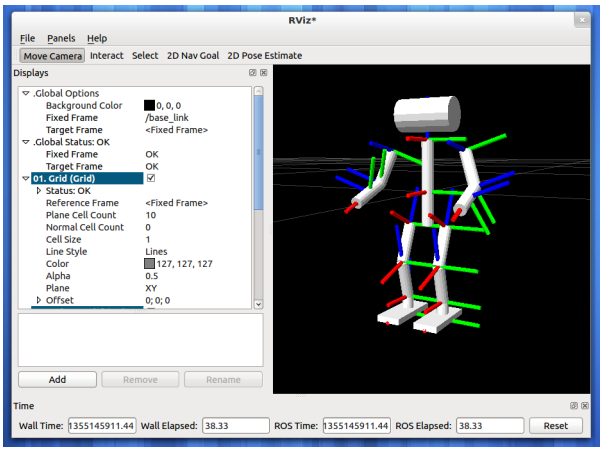


Figura 1.2: Simulazione del NAO in Rviz con il package `nao_description`.

1.3 Gazebo

Gazebo [6] è un simulatore 3D open source con la capacità di simulare gruppi robot, sensori e oggetti vari. Il progetto nasce nel 2002 presso *University of Southern California*. È progettato per aiutare i ricercatori che lavorano con veicoli robotici in ambienti esterni ma è in grado anche di gestire ambienti interni.

Gazebo utilizza il motore fisico *ODE* (*Open Dynamics Engine*) [2] ma dall'ultima versione (1.5) è stato aggiunto il supporto parziale e sperimentale al motore fisico *Bullet* [3]. In generale i parametri di questi due motori, quali quelli riguardanti l'accurattezza della simulazione, sono esposti all'utente in modo da soddisfare qualsiasi esigenza in termini di prestazioni e precisione della simulazione. Per il rendering della grafica tridimensionale, Gazebo sfrutta *OGRE* (*Object-Oriented Graphics Rendering Engine*) [1] e assicura un buon grado di realismo generando correttamente luci e ombre dell'ambiente.

All'interno del simulatore possono essere caricati vari tipo di oggetti, da semplici forme come cubi e sfere a modelli complessi come edifici o animali. Ogni oggetto ha le proprie caratteristiche: massa, velocità, frizione e numerose altre proprietà fisiche e di visualizzazione in modo da rendere la simulazione più realistica possibile. È inoltre disponibile un database di robot gestito dalla comunità che tutti possono utilizzare e modificare a proprio piacimento. Gazebo può anche generare i dati di diversi sensori: laser rangefinder, telecamere 2D e RGB-D, sensori di contatto, Inertial Measurement Units (IMU) e Radio Frequency IDentification (RFID). Connettendo uno o più sensori al modello di un robot è possibile modificare le azioni del robot sulla base dei dati generati dai sensori. Per questo motivo è possibile implementare dei plugin personalizzati che gestiscono i dati raccolti dai sensori e che controllino qualsiasi aspetto del robot.

Alcuni parametri della simulazione possono essere controllati direttamente attraverso un'interfaccia grafica basata su QT[12]. Gazebo è compatibile con diverse distribuzioni Linux e dispone di un'interfaccia nativa con ROS e *Player* [25] per-

[12] `http://qt-project.org/`.

mettendo di integrare diversi tipi di robot. In questo modo non è necessario utilizzare le API specifiche di un robot o di un sensore per sviluppare una specifica interfaccia con Gazebo infatti qualsiasi dispositivo supportato da ROS o Player può essere simulato direttamente.

La documentazione è di buon livello e sono presenti diversi tutorial che aiutano chi si è avvicinato da poco a questo simulatore, a volte però alcune caratteristiche non sono descritte in modo adeguato come ad esempio gli effetti in fase di simulazione di alcuni parametri e come implementare un plugin. Uno strumento potente è la piattaforma *Gazebo Answers* infatti permette agli utenti di pubblicare le proprie domande relative a problemi riscontrati o dubbi sull'utilizzo del simulatore. Grazie all'aiuto degli utenti più esperti e degli sviluppatori stessi di Gazebo i problemi vengono risolti in tempi brevi e in modo adeguato permettendo ai nuovi utenti di migliorare le proprie capacità e conoscenze velocemente.

Una demo che presenta le principali funzionalità dell'ultima versione del software si può vedere all'indirizzo `http://vimeo.com/61568492`.

1.4 V-REP

V-REP [7] è un simulatore sviluppato dalla *Coppelia Robotics* con lo scopo di simulare sistemi di automazioni industriali. V-REP è disponibile in quattro licenze: *Player* (gratuito), *Pro Edu* (gratuito per l'istruzione), *Eval Pro* (gratuito per usi non commerciali) e *Pro* (per uso commerciale), inoltre dall'inizio del 2013 è open source per usi non commerciali. Nel marzo 2010 è stata rilasciata la prima versione pubblica e dall'agosto 2012 è disponibile l'integrazione in ROS. Può essere visto come un simulatore ibrido in quanto combina cinematica e dinamica con il fine di ottenere le migliori prestazioni nei vari scenari simulati.

V-REP dispone di un ambiente di sviluppo integrato ed è basato su un sistema di controllo distribuito infatti ciascun modello può essere controllato attraverso uno script incorporato, un plugin, un nodo ROS o un'API chiamata da un client remoto. I controllori possono essere implementati in C/C++, Python, Lua, Matlab e Urbi. Il simulatore è disponibile per i sistemi Windows, Linux e MacOS. Nel caso si sfrutti l'integrazione con ROS il trasferimento del controllo da un robot simulato ad un robot reale è immediato.

La documentazione è molto buona, copre tutti gli aspetti del simulatore inoltre sono presenti alcuni tutorial che aiutano l'utente a comprendere meglio le funzionalità più complesse. Dall'inizio del 2013, nel sito ufficiale è stata aperto un forum che permette agli utenti di porre le proprie domane, segnalare eventuali bug e richiedere nuove funzionalità. Il simulatore è inoltre distribuito con molti modelli: persone, robot umanoidi, robot simili ad animali, robot con ruote e robot industriali.

V-REP dispone di molte funzionalità avanzate come il rilevamento delle collisioni e la misura di distanze minime fra oggetti in tempo reale, la simulazione di tagli e incisioni su superfici, la gestione di sensori di prossimità e di visione, path planning e la registrazione e visualizzazione di dati. Dispone inoltre di due

motori fisici: ODE [2] e Bullet [3]; l'utente può passare da uno all'altro prima di avviare la simulazione.

Una demo che illustra le principali funzionalità si può vedere all'indirizzo `http://youtu.be/bwGYUayglHY`.

Capitolo 2

Creazione del modello

In questo capitolo viene descritto il processo che ha portato alla realizzazione del modello del NAO che ha portato alla creazione del package ROS `nao_model` [26]. Per prima cosa viene analizzato il problema delle mesh, poi viene presentata la codifica URDF e infine come questa è usata per la creazione del modello.

2.1 Reperimento e conversione delle mesh

Per cercare di velocizzare la creazione del modello si è deciso di cercare delle mesh del robot che potessero essere adatte al nostro scopo. Dopo attente ricerche si è scelto di utilizzare le mesh presenti in *Webots*[1]. Webots è un simulatore professionale sviluppato dal *Politecnico Federale di Losanna (EPFL)* e ampiamente utilizzato per scopi didattici. Il software è distribuito utilizzando una licenza proprietaria ma i modelli presenti nel simulatore sono rilasciati utilizzando una licenza *BSD-style*, quindi liberamente modificabili e distribuibili.

I modelli dei robot usati in Webots sono implementati tramite una versione modificata del formato *VRML*, un linguaggio inizialmente pensato per il Web che permette la definizione di oggetti tridimensionali. Le modifiche implementate dagli sviluppatori di Webots riguardano la robotica, in particolare permettono di utilizzare la sintassi VRML anche per la definizione dei sensori, ma non è inclusa la definizione delle caratteristiche fisiche dei componenti.

Come prima cosa è stato quindi modificato il file che descrive il modello del NAO ripulendolo da tutte le caratteristiche non appartenenti allo standard VRML. Questa fase all'apparenza semplice ha richiesto un discreto lavoro perché, non avendo precedenti esperienze con questo formato, si è dovuto anche studiarne la sintassi in modo da individuare quali parti del file dovessero essere eliminate, inoltre la dimensione del file (circa 8500 righe) rendeva poco agevole l'operazione. Per facilitare il compito il file è stato suddiviso nelle varie parti che compongono il robot (la testa, il busto, le due braccia e le due gambe); durante questa ope-

[1] `http://www.cyberbotics.com/`.

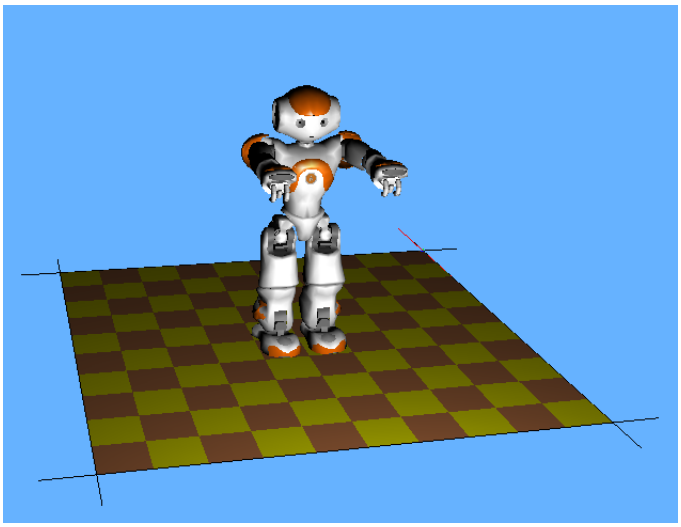


Figura 2.1: Simulazione del NAO in Webots 7.0.2.

razione due software sono stati molto utili: *Kate*[2] e *view3dscene*[3]. Il primo è un editor di testo che supporta l'evidenziazione del codice VRML mentre il secondo è un visualizzatore di oggetti tridimensionali di vari formati tra cui, appunto, il VRML; particolarmente apprezzata è la capacità di view3dscene di segnalare eventuali errori nel file caricato specificando la riga che genera l'errore.

A questo punto è stato necessario convertire le mesh in modo da essere compatibili con il formato URDF. Il primo vincolo richiesto è che ogni *link*[4] del robot deve essere rappresentato da una singola mesh. Tornava quindi utile quanto già svolto in precedenza relativamente alla divisione del file del modello in vari pezzi anche se la suddivisione richiesta in questo caso è ancora maggiore; ad esempio il braccio va suddiviso in spalla, avambraccio, bicipite, mano e dita. Il secondo requisito richiesto è relativo al formato delle mesh, in particolare sono supportati i formati *STL* (*STereoLithography*), un linguaggio nato per i software di stereolitografia CAD e utilizzato nella prototipazione rapida, e *Collada*, è un formato file XML di interscambio tra applicazioni 3D distribuito gratuitamente. Dopo alcune prove è stato scelto il secondo formato perché permette di definire delle mesh contenenti parti di colori diversi, inoltre utilizzando *Blender* 2.65[5] è possibile importare file di tipo VRML ed esportali in Collada, anche se in alcuni casi con qualche imprecisione. I difetti si verificano quando sono presenti delle texture applicate ad alcune parti della mesh (vedi Fig. 2.2), nonostante questo nessun software tra quelli considerati riesce ad importare i file VRML ed esportarli in modo migliore, anche considerando il formato STL.

Anche se non strettamente richiesto è stata applicata un'altra modifica alle

[2] `http://kate-editor.org/`.

[3] `http://castle-engine.sourceforge.net/view3dscene.php`.

[4] La definizione di link verrà data con maggior precisione in 2.2, per ora si può pensare un link come un "pezzo" del robot come ad esempio la testa, le spalle, ecc.

[5] `http://www.blender.org/`.

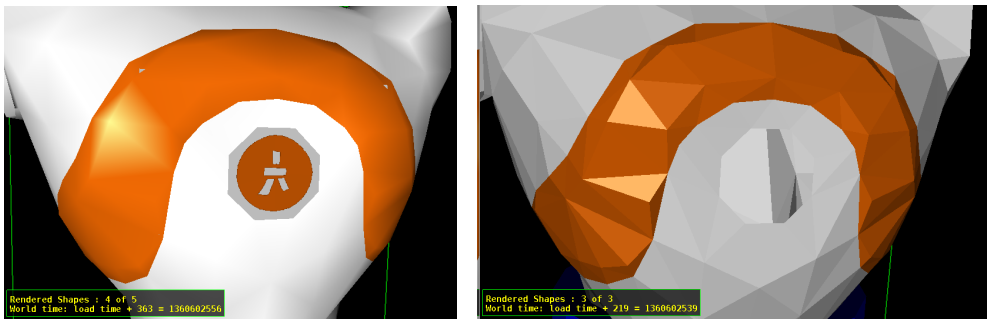


Figura 2.2: Dettaglio della mesh del busto del NAO, a sinistra in formato VRML e a destra in formato Collada ottenuto con Blender.

mesh. Il formato URDF permette di applicare delle traslazioni e rotazioni alle mesh nel momento in cui vengono associate ad un link, però per rendere più semplice la codifica del file è consigliabile fornire le mesh in modo che siano già allineate e orientate secondo la posa iniziale del robot. Fortunatamente le mesh risultano già allineate all'origine del link associato, lo stesso non vale per l'orientazione ma è stato sufficiente ruotare le mesh prima di $-\pi$ lungo l'asse y e poi di $-\frac{\pi}{2}$ lungo l'asse x per ottenere il risultato desiderato (Fig. 2.3). Le rotazioni sono state applicate modificando i file VRML.

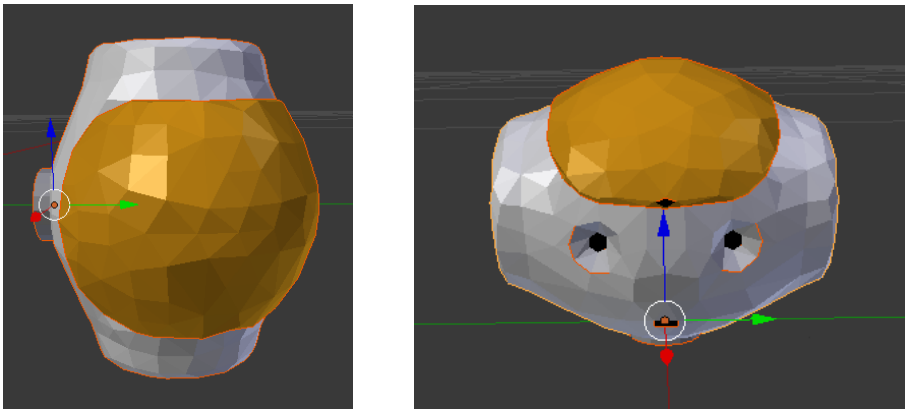


Figura 2.3: A sinistra la mesh della testa nella versione presente in Webots, a destra il risultato dell'applicazione di una rotazione di $-\pi$ lungo l'asse y e seguita da una di $-\frac{\pi}{2}$ lungo l'asse x.

2.2 Struttura del file URDF

URDF (*Unified Robot Description Format*) [27] è un formato XML usato in ROS per la rappresentazioni di modelli di robot. La descrizione del modello consiste nella definizione di due insiemi: l'insieme dei *link* e l'insieme dei *joint*. Un link descrive le caratteristiche di un corpo rigido dotato di inerzia (ad esempio un

braccio o un piede) mentre un joint descrive la cinematica e la dinamica di un giunto. I link vengono inoltre usati per collegare tra loro i vari giunti.

2.2.1 I joint

Un giunto (Fig. 2.4) viene definito specificando due attributi obbligatori e una serie di elementi, alcuni dei quali facoltativi. Gli attributi sono: *name* che specifica il nome univoco del joint e *type* che specifica il tipo di giunto; attualmente sono disponibili cinque tipi di joint.

- *revolute* definisce un giunto a cerniera che ruota lungo un'asse con un intervallo di movimento limitato specificato da un limite inferiore e da un limite superiore;
- *continuos* è come *revolute* ma senza alcun limite di movimento;
- *prismatic* specifica un giunto scorrevole che scorre lungo un'asse con un intervallo limitato di movimento specificato da un limite inferiore e da un limite superiore;
- *fixed* non definisce propriamente un giunto poiché tutti i gradi di libertà di un giunto di questo tipo sono bloccati, torna utile per connettere due link senza che questi possano muoversi uno rispetto all'altro;
- *planar* un giunto di questo tipo si muove perpendicolarmente rispetto all'asse specificato.

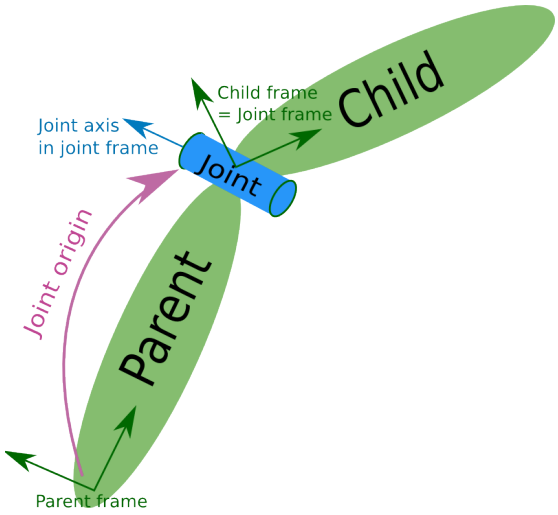


Figura 2.4: Caratteristiche di un joint in URDF.

Per completare la definizione di un giunto bisogna inoltre specificare i seguenti elementi:

- *origin* specifica, tramite traslazione e rotazione, la posizione del giunto rispetto all'origine del link padre;
- *parent* indica il link padre;
- *child* indica il link figlio;
- *axis* è l'asse lungo il quale il giunto si muove (non è necessario se il giunto è *fixed*);
- *calibration* (opzionale) usato per calibrare la posizione assoluta del joint;
- *dynamics* (opzionale) specifica il coefficiente di smorzamento e di frizione del giunto;
- *limit* stabilisce i limiti entro i quali il giunto si può muovere e la massima forza e velocità che può applicare, è obbligatorio solo per i giunti *revolute* e *prismatic*;
- *safety_controller* (opzionale) definisce alcuni limiti di sicurezza per il giunto.

2.2.2 I link

Ciascun link (Fig. 2.5) viene definito specificando l'attributo obbligatorio *name* che definisce il nome univoco del link. Per completare la definizione possono essere aggiunti gli elementi seguenti.

- *inertia* specifica le proprietà inerziali del link (massa, centro di massa e matrice di inerzia);
- *visual* in questo elemento viene specificato come deve essere rappresentato il link, ad esempio usando dei semplici solidi (cilindri, sfere o parallelepipedi) oppure una mesh; è inoltre possibile definire una traslazione e una rotazione da applicare all'oggetto da visualizzare;
- *collision* definisce le proprietà di collisione; si noti che questo elemento può essere diverso dall'elemento *visual* ad esempio usando modelli per la collisione più semplici in modo da ridurre i tempi di calcolo.

2.2.3 Le macro xacro

Per velocizzare la creazione e migliorare la leggibilità di un modello URDF è possibile utilizzare il package `xacro` [28]. Xacro è un linguaggio XML che permette di definire delle macro. Lo sviluppatore può definire le proprie macro, usarle per la descrizione del proprio modello e poi attraverso l'uso di appositi comandi il codice verrà interpretato generando il file URDF completo. Xacro permette anche di definire delle costanti o includere altri file xacro.

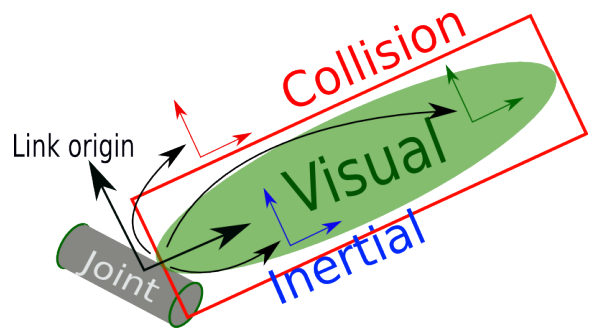


Figura 2.5: Caratteristiche di un link in URDF.

Il funzionamento delle macro viene analizzato nel dettaglio nei paragrafi 2.3.1 e 2.3.2 mentre viene descritto il modello del NAO. Per convertire un file dal formato Xacro al formato URDF viene usato il seguente comando.

```
rosrun xacro xacro.py model.xacro > model.urdf
```

2.3 Modello del NAO

In questa sezione viene descritto come è implementato il modello URDF del NAO. Tutte le caratteristiche del robot sono ricavate dalla documentazione ufficiale [29].

2.3.1 Definizione dei giunti

Mappare i giunti del NAO nella codifica URDF è semplice, tutti i giunti sono di tipo *revolute* inoltre le convenzioni usate in ROS per il sistema di coordinate e delle rotazione sono le stesse presenti nella documentazione ufficiale. L'unico dettaglio che richiede qualche sforzo in più è la posizione del giunto rispetto al link padre infatti nella documentazione sono riportate solo le dimensioni di alcuni link e le distanze di questi dal torso, comunque a partire da queste informazioni è possibile calcolare le distanze di interesse.

Come già detto nel paragrafo 2.2.3 è possibile velocizzare e semplificare la definizione del modello URDF tramite la costruzione di macro Xacro, viene quindi definita la macro `j` (Cod. 2.1) per la definizione dei joint.

La macro `j` richiede sette parametri: il nome del giunto (`name`), i nomi dei link padre e figlio (`parent` e `child`), i limiti in radianti entro il quale il giunto può muoverversi (`lower` e `upper`), l'asse rispetto al quale ruota (`axis`) e la posizione rispetto al link padre (`origin`). Gli ultimi due parametri, `axis` e `origin`, presentano un `*` perché non sono definiti come attributi ma come elementi.

Per chiarire meglio la differenza vediamo come è definito il giunto *HeadYaw*, che permette al robot di girare la testa a destra a sinistra, e come questa definizione viene trasformata nel corrispondente codice URDF (Cod. 2.2). Il primo elemento che compare (riga 2) viene assegnato alla variabile `*axis` definita nella

```
<xacro:macro name="j" params="name parent child lower upper *axis *origin">
  <joint name="${name}" type="revolute">
    <limit lower="${lower}" upper="${upper}" effort="100.0" velocity="5"/>
    <parent link="${parent}"/>
    <child link="${child}"/>
    <dynamics damping="0" friction="25"/>
    <xacro:insert_block name="axis"/>
    <xacro:insert_block name="origin"/>
  </joint>
</xacro:macro>
```

Codice 2.1: La macro `j` usata per la definizione dei joint del NAO.

```
<xacro:j name="HeadYaw" parent="Torso" child="Neck" lower="-2.0857" upper="2.0857">
  <axis xyz="0 0 1"/>
  <origin xyz="0 0 0.1265" rpy="0 0 0"/> <!-- NeckOffsetZ -->
</xacro:j>

<joint name="HeadYaw" type="revolute">
  <limit effort="100.0" lower="-2.0857" upper="2.0857" velocity="5"/>
  <parent link="Torso"/>
  <child link="Neck"/>
  <dynamics damping="0" friction="25"/>
  <axis xyz="0 0 1"/>
  <origin rpy="0 0 0" xyz="0 0 0.1265"/>
</joint>
```

Codice 2.2: Definizione del joint *HeadYaw* usando la macro `j` (righe 1-4) e il corrisponde codice URDF generato a partire da questa definizione (righe 6-12).

macro `j` e, quando la macro viene interpretata, viene sostituito alla riga 6 di Cod. 2.1. Lo stesso meccanismo si applica al secondo elemento (riga 3) che viene assegnato alla variabile `*origin` e poi sostituirà la riga 7 di Cod. 2.1. Il risultato finale ottenuto dall'interpretazione della definizione del giunto HeadYaw si può vedere in Cod. 2.2 righe 6-12.

Infine, come si può vedere nella definizione della macro `j`, a tutti i giunti è impostata una forza massima applicabile (`effort`) di 100 N e una velocità massima (`velocity`) di 5 rad/s, inoltre tramite l'elemento `dynamics` è impostato un smorzamento (`damping`) di 0 $\frac{N \cdot m \cdot s}{rad}$ e una frizione (`friction`) di 25 $N \cdot m$.

2.3.2 Definizione dei link

La definizione dei link del NAO è più problematica rispetto ai joint. Uno dei problemi riscontrati è già stato affrontato ed è quello relativo alle mesh, bisogna precisare però che a non tutti i link è stato possibile associare una mesh in quanto rappresentavano alcuni parti non presenti nel modello di Webots, in particolare questi link sono *Neck*, *RElbow* e *LElbow* cioè il collo e i due gomiti.

Il secondo problema riguarda la definizione della matrice d'inerzia di un link; le specifiche di ROS richiedono che questa matrice sia riferita rispetto al centro di massa del link mentre nella documentazione del NAO è riferita rispetto all'origine del sistema di riferimento del link. Dopo aver preso in considerazione la strada di approssimare i vari link a dei solidi più semplici di cui fosse possibile calcolare la matrici di inerzia con semplici calcoli, si è utilizzato il *teorema di Huygens-Steiner* o *teorema degli assi paralleli*.

Teorema degli assi paralleli. *Detti $R_A(A,i,j,k)$ e $R_B(B,i,j,k)$ due generici sistemi di riferimento, entrambi solidali ad un corpo b di massa m e paralleli tra loro, con diversa origine A e B, legati dalla traslazione relativa*

$$\overrightarrow{AB} = t_{AB} = \begin{bmatrix} t_x \\ t_y \\ t_y \end{bmatrix} = \begin{bmatrix} x_B - x_A \\ y_B - y_A \\ z_B - z_A \end{bmatrix}$$

e dette Γ_A e Γ_B le relative matrici di inerzia, si avrà tra queste la seguente relazione:

$$\Gamma_A = \Gamma_B + m\{\left\|t_{AB}\right\|^2 I - t_{AB} t_{AB}^T\}$$

Più semplicemente, definendo Γ e $\Gamma^{'}$ come segue

$$\Gamma = \Gamma_B = \begin{bmatrix} \Gamma_{xx} & \Gamma_{xy} & \Gamma_{xz} \\ \Gamma_{yx} & \Gamma_{yy} & \Gamma_{yz} \\ \Gamma_{zx} & \Gamma_{zy} & \Gamma_{zz} \end{bmatrix}$$

$$\Gamma^{'} = \Gamma_A = \begin{bmatrix} \Gamma^{'}_{xx} & \Gamma^{'}_{xy} & \Gamma^{'}_{xz} \\ \Gamma^{'}_{yx} & \Gamma^{'}_{yy} & \Gamma^{'}_{yz} \\ \Gamma^{'}_{zx} & \Gamma^{'}_{zy} & \Gamma^{'}_{zz} \end{bmatrix}$$

si ha che

$$\begin{aligned} \Gamma^{'}_{xx} &= \Gamma_{xx} + m(t_y^2 + t_z^2) \\ \Gamma^{'}_{yy} &= \Gamma_{yy} + m(t_x^2 + t_z^2) \\ \Gamma^{'}_{zz} &= \Gamma_{zz} + m(t_x^2 + t_y^2) \\ \Gamma^{'}_{xy} &= \Gamma^{'}_{yx} = \Gamma_{xy} + m t_x t_y \\ \Gamma^{'}_{xz} &= \Gamma^{'}_{zx} = \Gamma_{xz} + m t_x t_z \\ \Gamma^{'}_{yz} &= \Gamma^{'}_{zy} = \Gamma_{yz} + m t_y t_z \end{aligned} \tag{2.1}$$

Nel nostro caso è necessario trasformare le matrici di inerzia I_{NAO} riferite all'origine del sistema di riferimento del link cioè quelle fornite nella documentazione in modo che si riferiscano al centro di massa. Definendo le matrici da inserire nel file URDF I_{URDF} e ponendo $\overrightarrow{AB} = -CoM$ dove CoM è il centro di massa del link, $\Gamma^{'} = I_{URDF}$ e $\Gamma = I_{NAO}$ è possibile applicare le formule viste in Eq. 2.1 e calcolare le matrici di inerzia I_{URDF} da inserire nella definizione dei link nel file URDF.

```
<xacro:macro name="l" params="name xyz mass *inertia *visual *collision">
  <link name="${name}">
    <inertial>
      <origin xyz="${xyz}" rpy="0 0 0"/>
      <mass value="${mass}" />
      <xacro:insert_block name="inertia"/>
    </inertial>
    <xacro:insert_block name="visual"/>
    <xacro:insert_block name="collision"/>
  </link>
  <gazebo reference="${name}">
    <turnGravityOff> false </turnGravityOff>
  </gazebo>
</xacro:macro>
```

Codice 2.3: Definizione della macro `l` usata per la definizione dei link del NAO.

Anche in questo caso per velocizzare la scrittura del modello si è definita una macro Xacro, in particolare questa macro si chiama `l` (Cod. 2.3). La macro `l` viene usata per definire i link a cui è associata una mesh, infatti richiede tre attributi, `name`, `xyz` e `mass`, che specificano rispettivamente il nome, il centro di massa e la massa del link, e tre elementi che indicano la matrice di inerzia, come il link deve essere visualizzato e l'oggetto da utilizzare per la gestione delle collisioni. Le righe 11-13 riguardano solo Gazebo e vengono analizzate nel paragrafo 3.1.1.

```
<xacro:l name="Head" xyz="-0.00112 3e-05 0.05258" mass="0.60533">
  <inertia ixx="0.00430483" ixy="8.7678e-06" ixz="5.33702e-06"
           iyy="0.00416541" iyz="-2.90031e-05" izz="0.000986496" />
  <xacro:insert_visualization name="Head" xyz="0 0 0" />
  <xacro:insert_collision name="Head" xyz="0.002 0 0.054" size="0.119 0.133 0.116" />
</xacro:l>
```

Codice 2.4: Definizione del link *Head* usando la macro `l`.

Un esempio di link che è definito tramite questa macro è *Head*, la testa, (Cod. 2.4). Per definire questo link, oltre alla macro `l`, vengono usate anche le macro `insert_visualization` e `insert_collision` (Cod. 2.5).

La macro `insert_visualization` inserisce un elemento che specifica la mesh che deve essere usata per visualizzare il link. Questa macro richiede due attributi, il primo (`name`) indica il nome del link mentre il secondo (`xyz`) è la posizione della mesh rispetto al centro del link[6]. Tutte le mesh sono salvate nella cartella `mesh/dae` all'interno del package `nao_model` e il nome corrisponde al nome del link[7].

La gestione delle collisioni avviene approssimando ciascun link ad un parallelepipedo; il calcolo dei volumi che approssimano i link sono stati effettuati partendo

[6]Per tutti i link questo valore è sempre posto a `0 0 0` perché tutte le mesh sono già allineate al centro del link.

[7]Per maggiori dettagli si veda il paragrafo B.1.1.

```
1  <xacro:macro name="insert_visualization" params="name xyz">
2    <visual>
3      <origin xyz="${xyz}" rpy="0 0 0"/>
4      <geometry name="${name}_visual_geom">
5        <mesh filename="package://nao_model/mesh/dae/${name}.dae"/>
6      </geometry>
7    </visual>
8  </xacro:macro>
9
10 <xacro:macro name="insert_collision" params="name xyz size">
11   <collision>
12     <origin xyz="${xyz}" rpy="0 0 0" />
13     <geometry name="${name}_collision_geom">
14       <box size="${size}"/>
15     </geometry>
16   </collision>
17 </xacro:macro>
```

Codice 2.5: Definizione delle macro `insert_visualization` (righe 1-8) e `insert_collision` (righe 10-17).

dalle mesh e usando la funzione *bounding box* di Blender. Per la definizione dei parallelepipedi viene definita la macro `insert_collision`. Questa macro richiede tre paramatri: il nome del link (`name`), la posizione del parallelepipedo rispetto al centro del link (`xyz`) e la sua dimensione (`size`).

In Cod. 2.6 viene mostrato il codice URDF riguardante il link Head che si ottiene dall'espansione delle macro presenti in Cod. 2.4.

```
1  <link name="Head">
2    <inertial>
3      <origin rpy="0 0 0" xyz="-0.00112 3e-05 0.05258"/>
4      <mass value="0.60533"/>
5      <inertia ixx="0.00430483" ixy="8.7678e-06" ixz="5.33702e-06" iyy="0.00416541" iyz="
           -2.90031e-05" izz="0.000986496"/>
6    </inertial>
7    <visual>
8      <origin rpy="0 0 0" xyz="0 0 0"/>
9      <geometry name="Head_visual_geom">
10       <mesh filename="package://nao_model/mesh/dae/Head.dae"/>
11     </geometry>
12   </visual>
13   <collision>
14     <origin rpy="0 0 0" xyz="0.002 0 0.054"/>
15     <geometry name="Head_collision_geom">
16       <box size="0.119 0.133 0.116"/>
17     </geometry>
18   </collision>
19 </link>
```

Codice 2.6: Codice URDF del link *Head* ottenuto dall'espansione della definizione di Cod. 2.4.

```
1  <xacro:macro name="l2" params="name xyz mass *inertia">
2    <link name="${name}">
3      <inertial>
4        <origin xyz="${xyz}" rpy="0 0 0" />
5        <mass value="${mass}" />
6        <xacro:insert_block name="inertia" />
7      </inertial>
8    </link>
9    <gazebo reference="${name}">
10     <turnGravityOff> false </turnGravityOff>
11   </gazebo>
12 </xacro:macro>
13
14 <xacro:l2 name="Neck" xyz="-1e-05 0.00014 -0.02742" mass="0.06442">
15   <inertia ixx="0.000123429" ixy="1.47981e-09" ixz="-6.76036e-10"
16            iyy="0.000124435" iyz="-3.00245e-07" izz="5.535e-06" />
17 </xacro:l2>
```

Codice 2.7: Definizione della macro `l2` (righe 1-12) usata per i link di cui non è disponibile una mesh ed esempio di utilizzo con il link Neck (14-17).

Per i link che a cui non è associata nessuna mesh è definita un apposita macro: `l2` (Cod. 2.7 righe 1-12). La definizione è molto simile alla macro `l` solo che in questo caso non sono richiesti gli elementi `visual` e `collision`. Un esempio di link che utilizza questa macro è *Neck*, il collo, (Cod. 2.7 righe 14-17).

Per facilitare lo sviluppo del modello il codice è stato suddiviso in tre file.

- `nao_robot_v4.xacro` in questo file sono riportate tutte le macro usate per la definizione dei giunti e dei link, e la definizione delle costanti `pi` e `pi_2` di valore rispettivamente π e $\frac{\pi}{2}$, inoltre da questo file vengono inclusi gli altri due;
- `nao_robot_v4_joint.xacro` qui si trovano le definizioni dei giunti;
- `nao_robot_v4_link.xacro` contiene la descrizioni dei link.

Per quanto riguarda il calcolo delle matrici di inerzia, è stata scritto un piccolo programma in cui sono caricate le informazioni che servono per caratterizzare ciascun link. Una volta eseguito, il programma calcola le matrici di inerzia corrette e crea il file `nao_robot_v4_link.xacro` contenente, appunto, la definizione di tutti i link utilizzando le macro `l` e `l2`.

Capitolo 3

Integrazione in ROS

In questo capitolo viene descritto come è possibile utilizzare Gazebo e V-REP in ROS. Nella prima parte viene illustrato il procedimento da seguire per importare il modello URDF, nella seconda è descritto lo sviluppo dei plugin per controllare i modelli e infine nell'ultima parte sono riportati i risultati ottenuti. A questo scopo sono stati implementati due package: `nao_gazebo_plugin` [30] e `nao_v_rep` [31].

3.1 Importazione del modello URDF

Per poter utilizzare i due simulatori in ROS bisogna prima di tutto importare il modello URDF del NAO. Come si vedrà questo è piuttosto semplice in entrambi i simulatori anche se per V-REP è richiesta qualche operazione in più.

3.1.1 Gazebo

Per poter lanciare Gazebo e caricare il modello URDF del NAO presente nel package `nao_model` è sufficiente eseguire il seguente comando:

roslaunch nao_model gazebo_fuerte.launch

Il file `gazebo_fuerte.launch` (Cod. 3.1) contiene le istruzioni in grado di avviare tutti i nodi e impostare tutti i parametri necessari per avviare il simulatore e caricare il modello. In particolare nelle righe 2-3 viene avviato Gazebo caricando un mondo "vuoto" che contiene solo una sorta di pavimento e la GUI. Per avviare un nodo vengono specificati diversi attributi.

- `pkg` specifica il nome del package in cui è contenuto l'eseguibile da avviare;
- `type` è il nome dell'eseguibile;
- `name` indica il nome del nodo;
- `args` permette di passare dei parametri all'eseguibile;

- `required` se impostato a `true` fa si che quando il nodo termina eventuali altri nodi avviati da roslaunch vengono terminati;

- `output` specifica dove devono essere visualizzate o salvate le informazioni stampate a video dal nodo.

Nella riga 5 viene creata una variabile chiamata `robot_description` che contiene il codice URDF generato a partire dal file `nao_robot_v4.xacro`, utilizzando un comando simile a quello già visto nel paragrafo 2.2.3. Questa variabile viene poi utilizzata nella riga 6 per caricare il modello all'interno di Gazebo, in particolare con il parametro `-z 0.334` viene specificata l'altezza alla quale il modello deve essere collocato[1].

```
1 <launch>
2   <node pkg="gazebo" type="gazebo" name="gazebo" args="$(find gazebo_worlds)/worlds/empty.
        world" required="true" output="screen"/>
3   <node pkg="gazebo" type="gui" name="gazebo_gui" output="screen" required="true"/>
4
5   <param name="robot_description" command="$(find xacro)/xacro.py '$(find nao_model)
        /urdf/nao_robot_v4.xacro'" />
6   <node pkg="gazebo" type="spawn_model" name="spawn_nao" args="-urdf -param
        robot_description -x 0 -y 0 -z 0.334 -model nao_v4" output="screen"/>
7 </launch>
```

Codice 3.1: Il file `gazebo_fuerte.launch` del package `nao_model` che permette di avviare Gazebo e caricare il modello del NAO.

Affinché il modello funzioni correttamente in Gazebo è necessario aggiungere delle definizioni al file Xacro. Una prima modifica riguarda la macro `l` che come già visto nel paragrafo 2.3.2 comprende il Cod. 3.2; compito di questa parte di codice è abilitare l'effetto della forza di gravità al link specificato.

```
1   ...
2   <gazebo reference="${name}">
3     <turnGravityOff> false </turnGravityOff>
4   </gazebo>
5   ...
```

Codice 3.2: Porzione di codice della macro `l` (Cod. 2.3) che abilita la forza di gravità al link specificato.

La seconda modifica riguarda i coefficienti di frizione dei piedi. Per entrambi i piedi i coefficienti sono stati impostati a 0.8, come ad esempio si può vedere per il piede destro in Cod. 3.3. Questi valori vengono passati al motore fisico ODE usato da Gazebo.

[1] Il centro di massa del robot si trova all'incirca al centro del busto a 0.334 m di altezza.

```
<gazebo reference="RFeet">
  <mu1 value="0.8" />
  <mu2 value="0.8" />
</gazebo>
```

Codice 3.3: Definizione dei coefficienti di frizione del link *RFeet*.

3.1.2 V-REP

Integrare V-REP in ROS non è semplice come per Gazebo. Prima di tutto occorre installarlo in modo che che possa integrarsi con ROS, questo comporta l'istallazione e la compilazione di alcuni package[2].

L'importare il modello URDF risulta immediato infatti V-REP dispone di un comodo plugin che permette di importare dei modelli URDF. Una volta importato, il modello può essere modificato e salvato nel formato utilizzato da V-REP per memorizzare le scene. Nel nostro caso sono stati modificati i valori dei PID che controllano la posizione dei giunti, portando l'azione proporzionale da 0.1 a 0.5 mentre le azioni derivative e integrative sono rimaste a 0, inoltre sono stati modificati i coefficienti di frizione dei piedi per i due motori fisici, impostandoli come mostrato in Fig. 3.1.

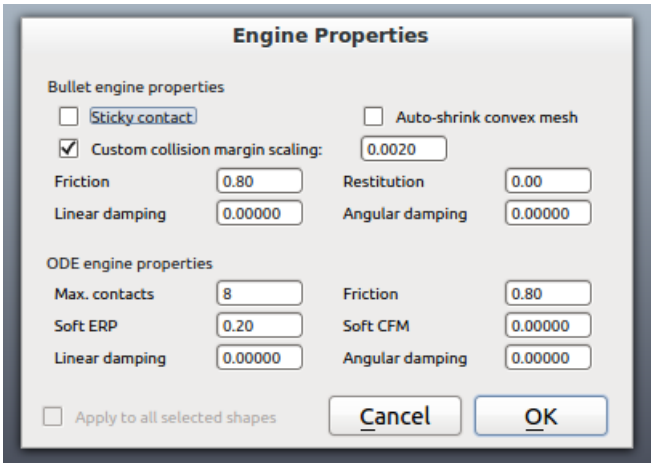


Figura 3.1: Coefficienti di frizioni dei piedi del modello in V-REP.

Avviare il simulatore in ROS e caricare il modello avviene in modo diverso rispetto a Gazebo, in questo caso infatti viene avviato il programma specificando come parametro la scena contenente il modello del NAO precedentemente importato. Il codice del file `v_rep.launch` del package `nao_model` è riportato in Cod. 3.4, il suo compito è avviare l'eseguibile `vrep` contenuto nel package `nao_v_rep`. Questo eseguibile non è altro che uno script bash che avvia V-REP passando come parametro la scena contenente il modello del NAO.

[2]Le istruzioni complete per installare e configurare V-REP in modo che possa integrarsi con ROS sono descritte nell'appendice A.4.

```
<launch>
  <node pkg="nao_v_rep" type="vrep" name="vrep" output="screen" required="true"/>
</launch>
```

Codice 3.4: Il file `v_rep.launch` del package `nao_model`.

3.2 Plugin ROS

Finora si è visto come importare il modello URDF in Gazebo e V-REP, ma affinché i simulatori risultino perfettamente integrati in ROS è necessario sviluppare dei plugin (implementati come package) per controllare i link del robot durante la simulazione.

3.2.1 Gazebo

Gazebo simula la dinamica dei movimenti dei link del modello, è quindi richiesto di fornire al simulatore la velocità e la forza con cui ciascun giunto si sta muovendo. Nel caso del NAO, ROS pubblica nel topic `joint_states` la posizione dei link del robot, il compito del plugin è quello di fornire a Gazebo le velocità e le forze da applicare ai giunti a partire da queste posizioni.

Il plugin è implementato a partire dalla versione sviluppata per il *Robovie-X* in [32]. Purtroppo questa versione non è compatibile con ROS Fuerte in quanto sviluppata per la versione precedente, per questo motivo il plugin è stato completamente riscritto anche se il funzionamento è principalmente lo stesso.

È possibile fare in modo che il plugin venga avviato automaticamente quando il modello viene caricato in Gazebo aggiungendo le istruzioni riportate in Cod. 3.5 al file del modello. In particolare nella riga 2 viene specificato il nome del package contenente il plugin (in questo caso `nao_gazebo_plugin`) e il file contente il codice eseguibile (`libnaoGazeboPlugin.so`).

```
<gazebo>
  <controller:nao_gazebo_plugin name="nao_gazebo_plugin" plugin="libnaoGazeboPlugin.so">
    <alwaysOn> true </alwaysOn>
    <updateRate> 1000.0 </updateRate>
    <timeout> 5 </timeout>
  </controller:nao_gazebo_plugin>
</gazebo>
```

Codice 3.5: Istruzioni per abilitare l'avvio del nodo `nao_gazebo_plugin` quando il modello URDF viene caricato in Gazebo.

Il plugin è implementato estendendo la classe `gazebo::ModelPlugin` e implementando il metodo `Load`. Di seguito vengono analizzate le parti più rilevanti del codice.

In Cod. 3.6 è riportato un estratto del metodo `Load`. Questo metodo viene richiamato in fase di caricamento del plugin; la prima cosa che compie è caricare

```
1  void NaoGazeboPlugin::Load(physics::ModelPtr parent, sdf::ElementPtr sdfElem){
2
3    //caricamento in joint_map dei giunti del robot
4    //...
5
6    cmd_angles_sub_ = rosnode_->subscribe<sensor_msgs::JointState>("joint_states", 1, &
         NaoGazeboPlugin::cmdCallback, this);
7    cmd_pgain_sub_ = rosnode_->subscribe<std_msgs::Float64>("nao/pgain", 1, &
         NaoGazeboPlugin::cmdCallbackPgain, this);
8    cmd_dgain_sub_ = rosnode_->subscribe<std_msgs::Float64>("nao/dgain", 1, &
         NaoGazeboPlugin::cmdCallbackDgain, this);
9
10   conn = event::Events::ConnectWorldUpdateStart(boost::bind( &NaoGazeboPlugin::
         OnUpdate, this));
11 }
```

Codice 3.6: Estratto del metodo `Load`.

nella hash map `joint_map` tutti i giunti di cui è composto il robot. Questa hash map usa come chiave il nome del giunto e come valore una coppia composta da un oggetto `physics::JointPtr` che permette di controllare i giunti in Gazebo, e un valore di tipo `double` che indica la posizione che il giunto deve assumere.

Nelle righe 6-8 vengono registrate le callback necessarie al plugin. La prima aggiorna i valori che i giunti devono assumere (ascoltando i messaggi del topic `joint_states`) mentre la seconda e la terza permettono di aggiornarne in tempo reale i valori del PID usato per controllare i giunti (tramite i topic `nao/pgain` e `nao/dgain`).

Infine alla riga 10 viene registrato il metodo `OnUpdate` in modo che venga invocato ogni volta che Gazebo aggiorna lo stato del robot.

```
1  void NaoGazeboPlugin::OnUpdate(){
2    for (HashMap::iterator it = joint_map.begin(); it != joint_map.end(); it++){
3      physics::JointPtr joint = it->second.first;
4      if (joint){
5        double current_vel = joint->GetVelocity(0);
6        double current_angle = joint->GetAngle(0).GetAsRadian();
7        double damping_force = dgain_ * (0 - current_vel);
8        double diff_force = pgain_ * (it->second.second - current_angle);
9        double effort_cmd = diff_force + damping_force;
10
11       joint->SetForce(0, effort_cmd);
12       joint->SetVelocity(0, effort_cmd);
13     }
14   }
15 }
```

Codice 3.7: Il metodo `OnUpdate`.

Il metodo `OnUpdate` (vedi Cod. 3.7) non fa altro che scorrere tutti i giunti e imprimere a questi una forza e una velocità calcolata sulla base della loro posizione e velocità attuale e sulla posizione che il giunto deve assumere. I valori vengono calcolati utilizzando un PID che usa un'azione proporzionale pari

a `pgain_` e un'azione derivativa pari a `dgain_`; questi valori inizialmente sono posti rispettivamente a 16 e 0.2 ma possono essere cambiati durante la simulazione pubblicando un messaggio contenente il nuovo valore nei topic `nao/pgain` e `nao/dgain`[3].

3.2.2 V-REP

L'integrazione di V-REP avviene scrivendo un package in ROS (`nao_v_rep`) e uno script in V-REP. Il compito dello script è lanciare l'esecuzione del package `nao_v_rep` quando la simulazione viene avviata; questo package ha invece il compito di leggere i messaggi dal topic `/joint_states` e, dopo averli opportunamente codificati, inviarli a V-REP.

L'invio dei messaggi a V-REP avviene definendo e abilitando un topic all'interno del simulatore e specificando quale tipo di messaggi verranno inviati. È stato quindi creato un topic chiamato `/NaoVRepPlugin/joints` in cui V-REP si aspetta vengano pubblicate le posizioni in cui i giunti si devono spostare, di tipo `simros_strmcmd_set_joint_state`. A differenza di Gazebo, non è necessario implementare un PID nel nodo ROS: V-REP applicherà in automatico un controllore PID ai valori pubblicati nel topic appena descritto. I valori del PID sono impostati per ciascun giunto.

È importante far notare una piccola differenza tra i messaggi che specificano le posizioni dei giunti in ROS e in V-REP. Entrambi i simulatori utilizzano due array, nel primo vengono inseriti gli identificativi dei giunti mentre nel secondo vengono specificate le loro posizioni. La differenza sta nel modo in cui vengono definiti gli identificatori dei giunti, in ROS è usato il nome del giunto mentre in V-REP viene associato a ciascun giunto un numero univoco. All'avvio del nodo `nao_v_rep` è quindi necessario richiedere a V-REP gli identificativi che sono stati stati assegnati ai giunti e questo può essere fatto utilizzando il servizio `/vrep/simRosGetObjectHandle`. Come ultima nota per verificare quando la simulazione in V-REP viene terminata è sufficiente controllare i messaggi del topic `/vrep/info` in cui vengono pubblicate le informazione relative allo stato della simulazione.

Vengono ora descritte le parti più importanti del nodo `nao_v_rep`.

In Cod. 3.8 è mostrato il costruttore della classe `NaoVRepPlugin`; la prima cosa che viene fatta è caricare nella hash map `joint_map` tutti i giunti non fixed che compongono il modello (per brevità il codice non è riportato). A differenza di quanto implementato nel plugin per Gazebo, in questo caso l'hash map usa come chiave il nome del giunto e come valore l'identificativo usato da V-REP per riferirsi a quel specifico giunto. Per ottenere questo identificativo è stato implementato il metodo `getVRepHandle` che verrà analizzato in seguito. Nelle righe 5-6 vengono effettuate le iscrizioni ai topic `/vrep/info`, per ottenere le informazioni sullo stato della simulazione, e `/joint_states` per reperire la posizione che il giunto deve assumere. Successivamente nelle righe 11-15 viene creato un topic

[3]Per fare ciò si po' ad esempio usare il commando `rostopic` [33].

```
1  NaoVRepPlugin::NaoVRepPlugin(){
2    //caricamento joint in joint_map
3    //...
4
5    vrep_info_sub = rosnode->subscribe("/vrep/info",1, &NaoVRepPlugin::vrepinfoCallback,
         this);
6    joint_state_sub = rosnode->subscribe<sensor_msgs::JointState>("/joint_states", 1, &
         NaoVRepPlugin::jointStateCallback, this);
7
8    client_enableSubscriber = rosnode->serviceClient<vrep_common::simRosEnableSubscriber>(
         "/vrep/simRosEnableSubscriber");
9
10   srv_enableSubscriber.request.topicName = "/NaoVRepPlugin/joints"; // the topic name
11   srv_enableSubscriber.request.queueSize = 1; // the subscriber queue size (on V-REP side)
12   srv_enableSubscriber.request.streamCmd = simros_strmcmd_set_joint_state; // the
         subscriber type
13
14   if (!client_enableSubscriber.call(srv_enableSubscriber)
15       &&(srv_enableSubscriber.response.subscriberID==-1)) ros::shutdown();
16
17   vrep_joint_state_pub = rosnode->advertise<vrep_common::JointSetStateData>("joints", 1)
         ;
18 }
```

Codice 3.8: Il costruttore della classe `NaoVRepPlugin`.

all'interno di V-REP specificandone il tipo (`simros_strmcmd_set_joint_state`) cioè indicando che verranno specificate le posizioni dei giunti. Infine nella riga 17 viene creato il publisher che invierà i messaggi nel topic `/NaoVRepPlugin/joints` che è appena stato creato.

```
1  void NaoVRepPlugin::jointStateCallback(const sensor_msgs::JointState::ConstPtr&
       jointState){
2    vrep_common::JointSetStateData joint_data;
3    for (unsigned int i=0; i < jointState->position.size(); i++){
4      joint_data.handles.data.push_back(joint_map[jointState->name[i]]);
5      joint_data.values.data.push_back(jointState->position[i]);
6    }
7    vrep_joint_state_pub.publish(joint_data);
8  }
```

Codice 3.9: La callback `jointStateCallback`.

In Cod. 3.9 è mostrato il codice del metodo `jointStateCallback`. Questo metodo viene invocato ogni volta che un messaggio viene pubblicato nel topic `/joint_states` e semplicemente scorre tutti i giunti contenuti nel messaggio appena ricevuto e ne costruisce uno di tipo `JointSetStateData` da pubblicare all'interno di V-REP. Il nuovo messaggio contiene gli identificativi dei link, ottenuti dall'hash map `joint_map`, e le nuove posizioni che i giunti devono assumere.

Infine in Cod. 3.10 è presente il codice del metodo `getVRepHandle` che, dato un link, ritorna l'identificativo numerico che lo identifica in V-Rep. L'implementazione è piuttosto semplice, infatti è necessario creare un `ServiceClient` che si colleghi al servizio `/vrep/simRosGetObjectHandle` e poi inviare una richiesta in

```
int NaoVRepPlugin::getVRepHandle(boost::shared_ptr<urdf::Joint> joint){
  ros::ServiceClient client = rosnode->serviceClient<vrep_common::simRosGetObjectHandle
      >("/vrep/simRosGetObjectHandle");
  vrep_common::simRosGetObjectHandle srv;
  srv.request.objectName = joint->name;
  client.call(srv);
  if (srv.response.handle == -1)
    ROS_ERROR("Invalid handle for joint %s", joint->name.c_str());
  return srv.response.handle;
}
```

Codice 3.10: Il metodo `getVRepHandle`.

cui è inserito il nome del giunto. Il servizio risponderà con l'identificativo oppure con -1 se il link non è valido.

Come ultima cosa è necessario associare al modello uno script che avvi il nodo `nao_v_rep`. Questo script (di tipo threaded ed associato al link `base_joint`) è molto semplice, sostanzialmente verifica se ROS è in esecuzione e esegue il nodo `nao_v_rep`; la parte essenziale è riportata in Cod. 3.11.

```
simSetThreadSwitchTiming(2)
simDelegateChildScriptExecution()

-- Check if the required ROS plugin is there:
moduleName=0
moduleVersion=0
index=0
pluginNotFound=true
while moduleName do
   moduleName,moduleVersion=simGetModuleName(index)
   if (moduleName=='Ros') then
      pluginNotFound=false
   end
   index=index+1
end

-- Ok now launch the ROS client application:
if (not pluginNotFound) then
   -- Now we start the client application:
   result = simLaunchExecutable('nao_v_rep',"",0)
end
```

Codice 3.11: Script V-REP per avviare l'esecuzione del nodo `nao_v_rep`.

3.3 Risultati ottenuti

In questa sezione vengono presentate alcune immagini e video relativi al modello in Gazebo e V-REP.

3.3.1 Gazebo

In Fig. 3.2 si può vedere il modello caricato in Gazebo. Risultano di interesse anche i video [4] e [5] in cui viene rispettivamente mostrata una prova dei giunti del modello e la simulazione della camminata implementata dai driver ufficiali del NAO; maggiori dettagli si possono trovare nei paragrafi B.1.1 e 4.2.

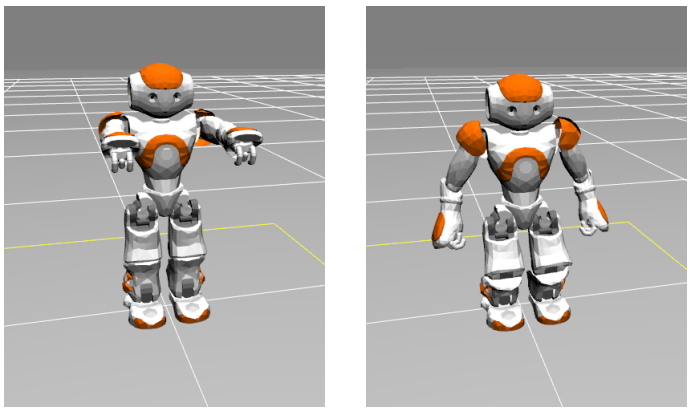


Figura 3.2: Modello del NAO importato in Gazebo.

3.3.2 V-REP

In Fig. 3.3 si può vedere il modello caricato in V-REP. Nel video [6] viene mostrata una prova dei giunti del modello mentre nel video [7] il robot muove i propri giunti in modo da assumere due posture diverse; per maggiori dettagli si vedano i paragrafi B.1.1 e B.1.2.

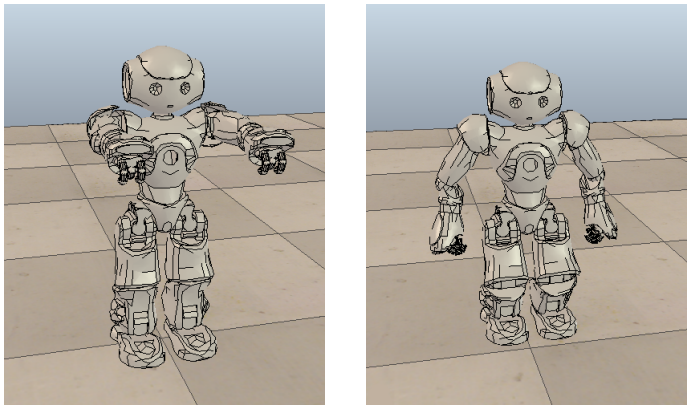


Figura 3.3: Modello del NAO importato in V-REP.

[4] http://youtu.be/lZv3M3eM8Kc.
[5] http://youtu.be/pLk4NyAhujU.
[6] http://youtu.be/BBWo0Dcve9o.
[7] http://youtu.be/FEIwuolKR_g.

Capitolo 4

Confronto fra Gazebo e V-REP

In questa capitolo Gazebo e V-REP vengono messi a confronto. Nella prima parte vengono prese in considerazione le caratteristiche tecniche dei due simulatori mentre nella seconda parte sono presentati i risultati dei test svolti con l'obiettivo di mettere a confronto il comportamento del robot reale e del robot simulato, e i dati forniti dall'odometria.

4.1 Caratteristiche dei simulatori

In tabella 4.1 sono riportate le caratteristiche principali dei due simulatori; le funzionalità prese in considerazione sono le seguenti:

- **Licenza** in particolare se il simulatore è open source;
- **OS** descrive i sistemi operativi supportati dal software;
- **Linguaggi di programmazione** indica i linguaggi di programmazione con cui è possibile interfacciarsi al programma;
- **Anno di inizio** specifica l'anno in cui è iniziato lo sviluppo;
- **Collision detection** indica se è presente o meno un sistema per rilevare le collisioni;
- **Sensori** elenca i sensori supportati;
- **GUI** descrive la possibilità di creare o modificare il modello o la scena direttamente dall'interfaccia grafica del programma; "Buona" significa che è possibile creare delle scene e gestire alcuni parametri della simulazione ma che non è possibile creare un modello completo di un robot, "Ottima" indica che è possibile gestire qualsiasi aspetto della simulazione, costruzione del modello inclusa, dalla GUI;
- **Portabilità** "Sì" indica che il codice sorgente scritto per la simulazione può essere direttamente trasferito al robot reale;

Caratteristiche	Gazebo	V-Rep
Licenza	Open Source	Open Source per uso non commerciale
OS	Linux	Linux, Win, MacOSX
Linguaggi di programmazione	C++, Python, Java	C/C++, Python, Java, Matlab, Urbi
Anno di inizio	2002	2010
Collision detection	Sì	Sì
Sensori	Laser rangefinder, telecamere RGB-D e 2D, sensori di contatto, IMU, RFID	Sensori di prossimità, visione e forza
GUI	Sufficiente	Ottima
Portabilità	Sì	Sì
Scalabità	Buona	Ottima
Real time	Buono	Buono
Interfacce	ROS (Ottima), Player (Ottima)	ROS (Buona)
Documentazione	Buona	Ottima
Tutorial	Sì	Sì
Debugging	Sì	Sì

Tabella 4.1: Principali caratteristiche di Gazebo e V-REP.

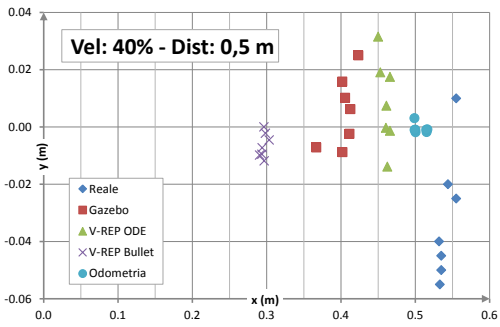
- **Scalabilità** "Ottima" significa che il simulatore non richiede troppe risorse per una simulazione complessa e che può simulare più robot contemporaneamente, "Buona" indica che in alcuni casi la simulazione non è sufficientemente veloce ma che comunque è possibile utilizzare più robot insieme;
- **Real time** specifica il numero di operazioni che possono essere svolte durante la simulazione, "Buono" indica che è possibile muovere la telecamera e modificare alcuni dei parametri della simulazione;
- **Interfacce** indica la facilità di integrazione del simulatore in altri sistemi;
- **Documentazione** descrive il livello della documentazione del simulatore;
- **Tutorial** indica se sono disponibili dei tutorial che aiutano a famigliarizzare con il simulatore;
- **Debugging** specifica se il simulatore dispone di funzionalità che permettono di facilitare il debugging.

4.2 Test 1: camminata rettilinea

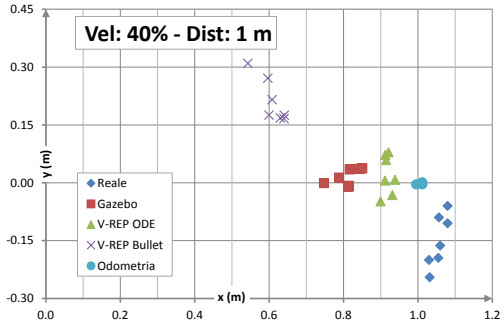
Nel primo test è stata usata la camminata fornita dai driver NAO per far camminare in linea retta il robot per tre diverse distanze (0.5, 1 e 3 m) a tre diverse

velocità (40%, 80% e 100% della massima velocità prodotta dai motori del NAO). Ciascuna prova è stata ripetuta per 7 volte con il robot reale, con V-REP (con entrambi i motori fisici) e in Gazebo. Durante l'esecuzione del test con il robot reale viene registrata l'odometria[1]. D'ora in avanti con il termine *distanza percorsa* si indicherà la distanza che il robot ha percorso lungo la coordinata x cioè in avanti, mentre indicherà con *deviazione* la distanza in y cioè la deviazione rispetto alla linea retta della camminata ideale. Tutti i risultati ottenuti sono riportati nella sezione D.1.

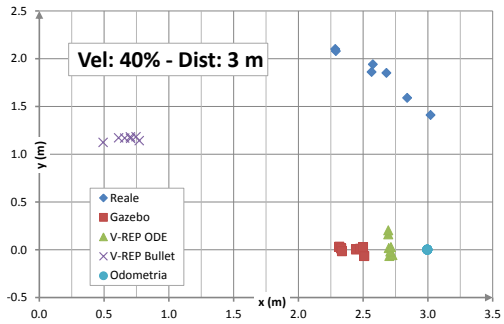
4.2.1 Velocità 40%



(a) Distanza 0.5 m.



(b) Distanza 1 m.



(c) Distanza 3 m.

Figura 4.1: Grafici del test camminata rettilinea con velocità 40%.

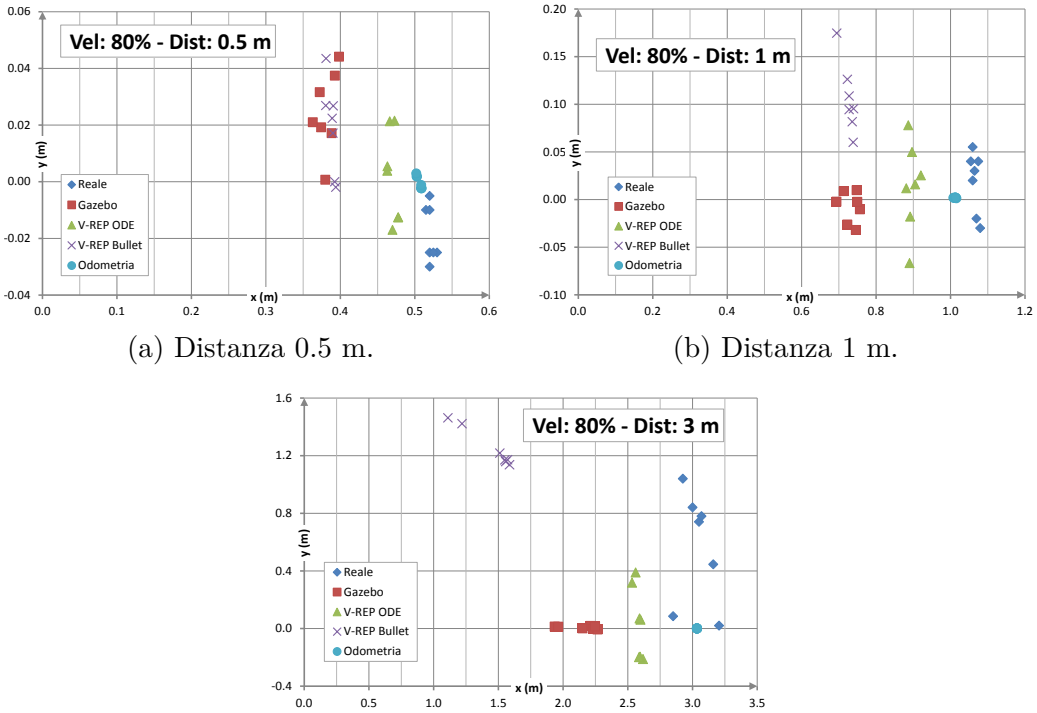
Dai grafici in Fig. 4.1 si può notare come l'odometria tenda a sottostimare la distanza percorsa dal robot reale. Quest'ultimo tende a deviare verso destra di 2-6 cm quando la distanza da percorrere è 0.5 m (Fig. 4.1a), di 3-30 cm quando la distanza è 1 m (Fig. 4.1b) e di ben 1.5-2.0 m in 3 m di distanza però in quest'ultimo caso verso sinistra (Fig. 4.1c).

Tutti i simulatori tendono a percorrere una distanza inferiore rispetto a quella calcolata dall'odometria. In Gazebo il robot percorre 0.4 m nel primo caso (Fig.

[1]In questo caso l'uso del termine odometria è forzato in quanto solitamente viene usato per indicare la stima della posizione di un veicolo su ruote basandosi su dati provenienti da sensori. In questo caso le informazioni provengono dai driver ufficiali che, sapendo com'è implementata la camminata, offrono una stima della posizione del robot.

4.1a), 0.8 m nel secondo (Fig. 4.1b) e 2.5 m nel terzo (Fig. 4.1c) mantenendo delle deviazione molto basse, quasi sempre inferiori ai 2 cm. V-REP ODE si comporta meglio percorrendo rispettivamente nei tre test 0.45 m, 0.85 m e 2.75 m, anche se con delle deviazioni, tendenzialmente verso sinistra, leggermente più accentuate rispetto a Gazebo. V-REP Bullet a questa velocità si comporta piuttosto male, infatti il robot percorre solo 0.3 m, 0.6 m e 0.75 m con deviazioni verso sinistra molto elevate, negli ultimi due test di 15-30 cm e 1 m circa rispettivamente.

4.2.2 Velocità 80%



(a) Distanza 0.5 m.

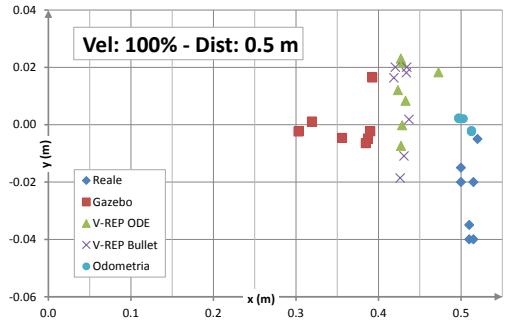
(b) Distanza 1 m.

(c) Distanza 3 m.

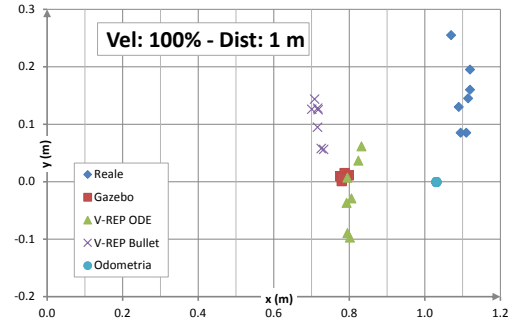
Figura 4.2: Grafici del test camminata rettilinea con velocità 80%.

Rispetto a quando visto nel paragrafo 4.2.1, l'aumento della velocità all'80% della velocità massima (Fig. 4.2) porta sostanzialmente a tre effetti. Il primo riguarda il robot reale che tende a camminare maggiormente in linea retta avvicinandosi a quanto calcolato dall'odometria, il secondo è che le deviazioni nei simulatori aumentano, tendenzialmente raddoppiano, tranne in V-REP Bullet che, terzo effetto, migliora decisamente. Il comportamento di Gazebo e V-REP ODE, deviazioni a parte, è sostanzialmente lo stesso già visto con velocità pari al 40% mentre in questo caso il robot in V-REP percorre 0.4 m (Fig. 4.2a), 0.7 m (Fig. 4.2b) e 1.0-1.5 m (Fig. 4.2c) ma con delle deviazioni verso sinistra ancora piuttosto elevate nell'ultimo test.

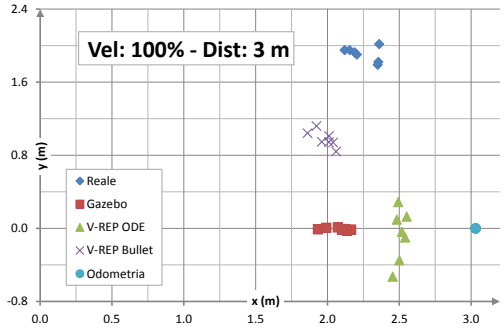
4.2.3 Velocità 100%



(a) Distanza 0.5 m.



(b) Distanza 1 m.



(c) Distanza 3 m.

Figura 4.3: Grafici del test camminata rettilinea con velocità 100%.

Spostare il robot impostando la velocità della camminata al massimo produce sopprattutto un elevato aumento delle deviazioni del robot reale e un aumento della variabilità delle singole prove nei simulatori (Fig. 4.3). In particolare il robot reale arriva a deviare verso sinistra di ben 2 m nell'ultimo test (Fig. 4.3c) mentre negli altri due peggiora leggermente rispetto al test a velocità 40%.

Tutti i simulatori, tranne V-REP ODE, peggiorano percorrendo meno strada e aumentando variabilità e deviazione ma l'entità della variazione è comunque molto piccola. V-REP Bullet, migliora decisamente arrivando a risultati molto simili agli altri simulatori percorrrendo nelle tre prove 0.45 m (Fig. 4.3a), 0.75 m (Fig. 4.3b) e 2.0 m (Fig. 4.3c) anche se nell'ultimo test la deviazione verso destra risulta ancora elevata a circa 0.8 m.

4.2.4 Distanze medie e deviazioni medie

Nei grafici di Fig. 4.4 sono riportate le distanze medie percorse e le deviazioni medie nella camminata del robot reale e simulato al variare della velocità.

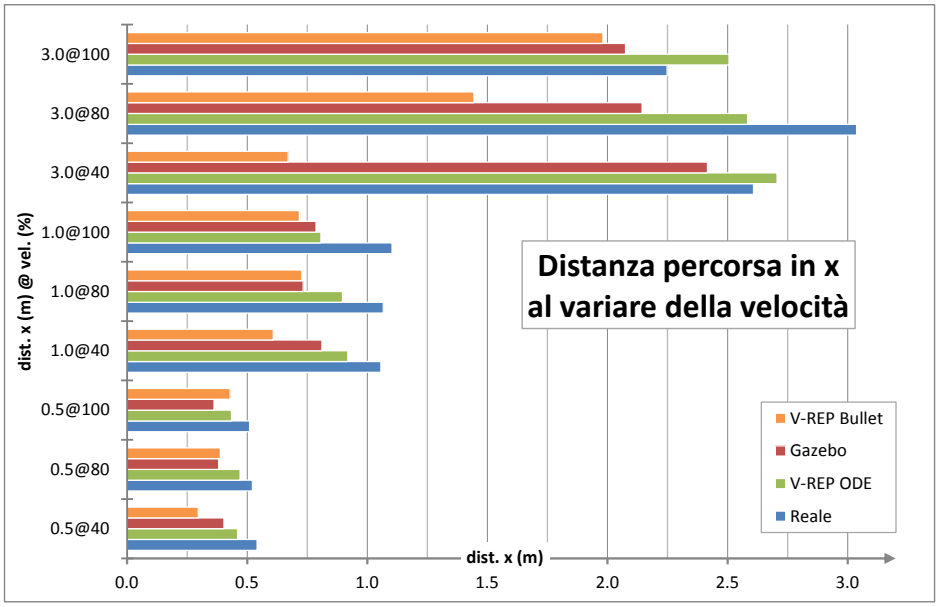
Dal grafico relativo alle distanze medie percorse (Fig. 4.4a) si nota subito che il robot reale è quello che percorre più strada tranne nei test a 3 m alle velocità 40% e 100% in cui le elevate deviazioni fanno percorre al robot meno strada lungo l'asse x. Una seconda osservazione è che V-REP ODE è sicuramente il

simulatore che si comporta meglio essendo quello che più si avvicina alla distanza misurata dall'odometria, anche se peggiora leggermente all'aumentare della velocità. Il robot in Gazebo si comporta in modo leggermente peggiore rispetto a V-REP ODE, anche in questo caso però i valori sono piuttosto buoni e tendono a peggiorare con l'aumento della velocità. V-REP Bullet invece si comporta in modo non soddisfacente a basse velocità, migliora a velocità 80% e raggiunge gli altri simulatore alla massima velocità riusciendo a fare anche meglio di Gazebo in alcuni casi.

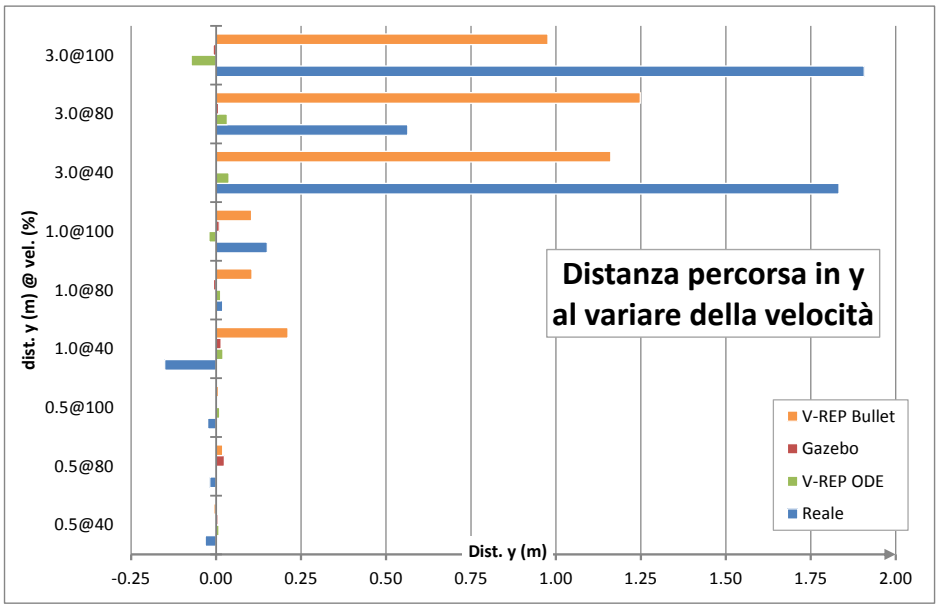
Nel grafico di Fig. 4.4b sono riportarte le deviazioni verso sinistra (positive) e verso destra (negative) rispetto alla linea retta che il robot dovrebbe percorrere. Per distanze basse, 0.5 m e 1 m, le deviazioni sono molto basse tranne nel caso del robot reale a velocità 40% e 100% e V-REP Bullet per distanze di 1m. Per il test di distanza sui 3 m, incominciano a farsi vedere delle deviazioni in V-REP ODE, il robot reale mantiene invece delle deviazioni a sinistra maggiori di 1.75 m tranne nel caso con velocità è 40% in cui scende a 0.5 m, il robot in V-Bullet devia tra circa un metro e 1.25 m mentre la camminata simulata in Gazebo risulta decisamente rettilinea.

Riassumendo:

- la miglior velocità, fra quelle testate, per muovere il robot reale in modo rettilineo è all'80% di quella massima;
- Gazebo e V-REP ODE hanno comportamenti molto simili, in V-REP il robot percorrre leggermente più strada mentre la camminata in Gazebo risulta più rettilinea;
- i risultati peggiorano in Gazebo e V-REP ODE all'aumentare della velocità viceversa migliorano in V-REP Bullet;
- Gazebo ha una ripetibilità delle prove maggiore;
- V-REP Bullet ha il comportamento peggiore, a basse velocità il robot percorre poca strada e devia decisamente, anche più del robot reale, mentre aumentando la velocità la distanza percorsa migliora ma rimangono delle deviazioni rilevanti.



(a) Distanza frontali medie.



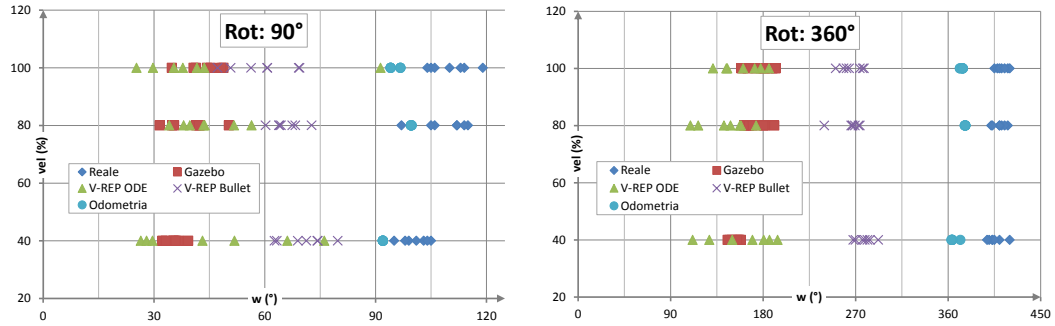
(b) Deviazioni medie.

Figura 4.4: Grafici delle deviazioni medie e delle distanze frontali medie nella camminata del robot reale e simulato al variare della velocità.

4.3 Test 2: rotazione sul posto

Il secondo test utilizza, come nel primo, le funzionalità implementate dai driver ufficiali per eseguire una rotazione sul posto, in particolare il robot ruota di quattro angoli diversi: 90° e 360° in senso antiorario e 180° e 270° in senso antiorario. Ciascuna rotazione è eseguita a tre velocità diverse, le stesse del primo test, 40%, 80% e 100% della massima velocità possibile, e ciascun test è ripetuto sette volte. Infine l'odometria viene registrata mentre si sta muovendo il robot reale. Tutti i risultati ottenuti sono riportati nella sezione D.2.

4.3.1 Rotazioni antiorarie



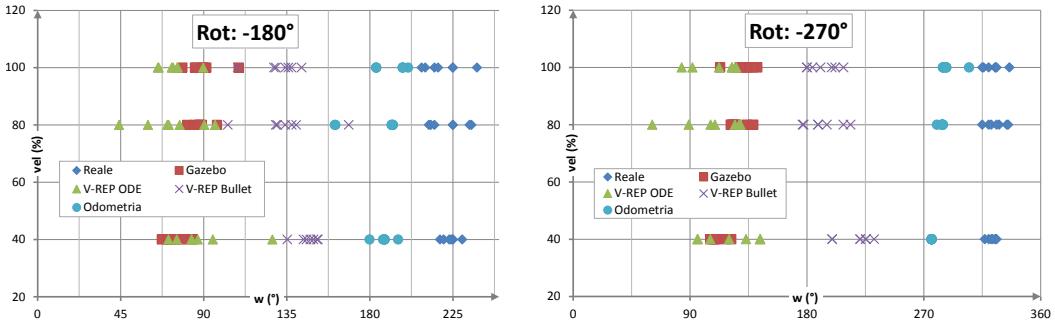
(a) Rotazione di 90°. (b) Rotazione di 360°.

Figura 4.5: Grafici del test rotazione sul posto in senso antiorario.

In Fig. 4.5 sono riportati i risultati ottenuti per le rotazioni di 90° e 360° in senso antiorario. Lungo l'asse delle ascisse sono riportati gli angoli percorsi dal robot mentre nelle ordinate sono riportate le velocità con cui è stato eseguito il test. Come per il primo test, anche in questo caso l'odometria sottostima la distanza percorsa dal robot reale e i simulatori eseguono una rotazione ancora minore. In questo caso il peggiore risultato si ottiene in Gazebo, nel primo test compie una rotazione di circa 30–40° (Fig. 4.5a) mentre nel secondo di 135°–210° (Fig. 4.5b). I risultati migliorano leggermente all'aumentare della velocità. V-REP ODE si comporta in modo piuttosto variabile, con risultati a volte migliori e a volte peggiori, con differenze di anche 45° nel primo test e di quasi 90° nel secondo. Il robot simulato in V-REP Bullet è quello che si avvicina più al comportamento calcolato dall'odometria arrivando a circa 75° nel primo test e 270° nel secondo, con una variabilità di circa 30°. In questo caso però i dati peggiorano leggermente all'aumentare della velocità.

4.3.2 Rotazioni orarie

In Fig. 4.6 sono riportati i grafici dei risultati ottenuti nelle rotazioni di 180° e 270° in senso orario. La situazione non cambia molto rispetto alle rotazioni in senso antiorario, V-REP Bullet risulta ancora il simulatore migliore anche se

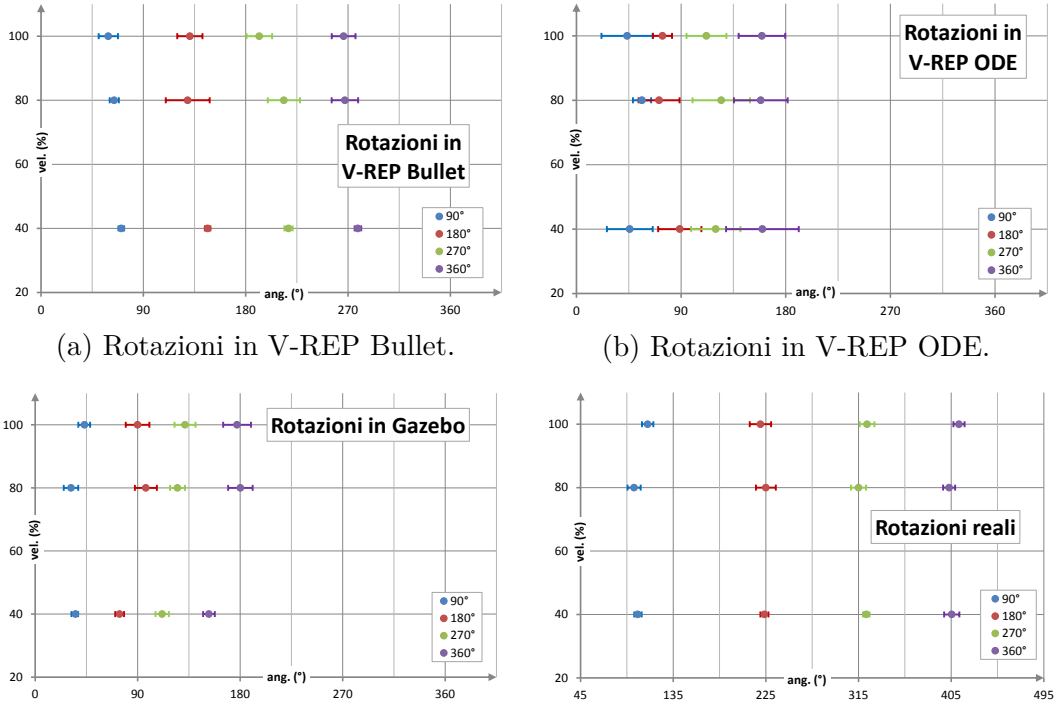


(a) Rotazione di 180°. (b) Rotazione di 270°.

Figura 4.6: Grafici del test rotazione sul posto in senso orario.

nel primo test arriva a 135°–150° e nel secondo a 180°–225°. Gazebo ha una variabilità piuttosto bassa ma gli angoli percorsi sono decisamente bassi 75°–90° e 90°–135° mentre V-REP ODE ha una variabilità decisamente alta, di 45° nel primo caso e di quasi 90° nel secondo, con risultati simili a quelli di Gazebo.

4.3.3 Rotazioni medie e scarti quadratici medi



(a) Rotazioni in V-REP Bullet. (b) Rotazioni in V-REP ODE.

(c) Rotazioni in Gazebo. (d) Rotazioni reali.

Figura 4.7: Grafici delle rotazioni medie e delle deviazioni standard del robot reale e simulato al variare della velocità

Nei grafici di Fig. 4.7 sono rappresentate le rotazioni medie percorse dai robot nei simulatori e dal robot reale e i rispetti intervalli di variabilità usando la

deviazione standard. In tutti i grafici è usata la stessa scala per l'asse delle ascisse in modo che gli intervalli siano confrontabili. Si può notare come in V-REP ODE le variazioni risultino piuttosto ampie anche a velocità molto basse (Fig. 4.7b), in V-REP Bullet gli intervalli sono bassi per velocità basse e più elevati all'umentare della velocità (Fig. 4.7a) mentre in Gazebo rimangono bassi a tutte le velocità (Fig. 4.7c). Infine il robot reale si comporta piuttosto bene con basse variabilità (Fig. 4.7d).

4.3.4 Confronto fra le rotazioni medie

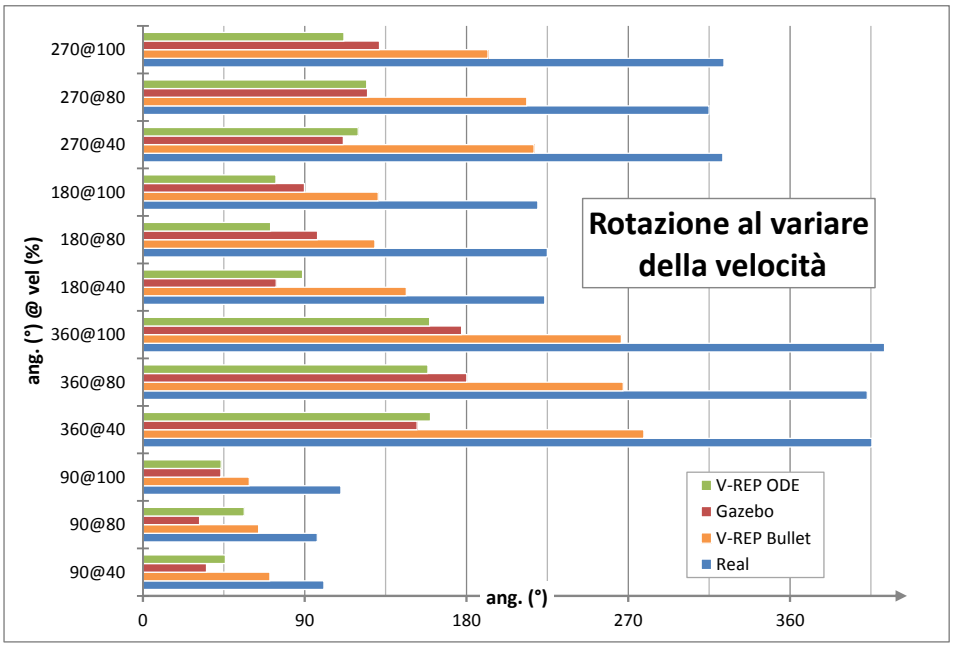


Figura 4.8: Cofronto fra le rotazioni medie eseguite dal robot reale e simulato.

In Fig. 4.8 è riportato un grafico che confronta le rotazioni percorse in tutti i test. Si nota subito come la distanza percorsa dal robot reale sia molto lontana dall'angolo percorso dal miglior simulatore, V-REP Bullet. Considerando gli angoli medi percorsi, Gazebo e V-REP ODE sostanzialmente si equivalgono, in alcuni casi è migliore il primo in altri il secondo ma le differenze sono piuttosto piccole tuttavia, come già evidenziato nel paragrafo 4.3.3. Gazebo presenta una variabilità basse mentre in V-REP ODE è piuttosto alta.

4.4 Vantaggi e svantaggi di Gazebo e V-REP

Nelle tabelle 4.2 e 4.3 sono riportati i principali vantaggi e svantaggi dei due simulatori. Anche considerando i risultati dei due test risulta difficile stabilire quale simulatore sia migliore; V-REP è leggermente in vantaggio grazie ai due motori fisici di cui dispone e al gran numero di tipologie di simulazioni supportate ma

Gazebo	**V-REP**
Perfettamente integrato in ROS	Documentazione e tutorial di ottima qualità
Supporto a molti tipi di sensori	Dalla GUI è possibile modificare completamente il modello e i parametri della simulazione
Possibilità di estendere le funzionalità tramite plugin	Disponibilità di due motori fisici
Ottima piattaforma di Q&A	Requisiti hardware modesti

Tabella 4.2: Principali vantaggi di Gazebo e V-REP.

anche in Gazebo il supporto al motore fisico Bullet sarà disponibile in futuro[2]. Infine bisogna considerare che attualmente tra i due, Gazebo risulta maggiormente integrato in ROS quindi nel caso la gestione dei robot avvenga utilizzando ROS è preferibile quest'ultimo simulatore piuttosto che V-REP.

Gazebo	**V-REP**
Su hardware meno potente a volte instabile e lento	Integrazione in ROS non immediata
Interfaccia grafica in alcuni casi non all'altezza	Distribuito con diverse licenze
Documentazione e tutorial a non sempre approffonditi a sufficienza	Creare un modello completo è più complesso

Tabella 4.3: Principali svantaggi di Gazebo e V-REP.

[2]Attualmente è ancora in sviluppo.

Capitolo 5

Conclusioni e sviluppi futuri

In questa tesi si è mostrato come sia possibile creare un modello di un robot reale complesso, quali sono le difficoltà che si incontrano e quanto valido possa essere un ambiente simulato se confrontato con il mondo reale. I risultati ottenuti dimostrano come la simulazione non tenga giustamente conto di tutte le imprecisioni o disturbi che caratterizzano il mondo reale ma anche come il comportamento del robot simulato sia vicino a quello reale. In alcuni casi i risultati sono sotto le aspettative ma sicuramente possono migliorare aumentando la precisione delle caratteristiche fisiche del modello. Il processo di modellazione non può quindi essere considerato concluso. In primo luogo deve essere completato con l'aggiunta dei sensori di cui il NAO dispone e implementando i plugin per la gestione dei dati che questi generano, permettendone quindi l'utilizzo in ROS. In secondo luogo è necessario verificare con maggior precisione quali siano i corretti valori per i coefficienti di frizione dei piedi del robot e le massime velocità e forze che ciascun giunto può applicare. Infine rimangono da correggere alcuni dettagli delle mesh ed effettuare il porting del modello e dei package sviluppati ad una versione più recente di ROS. Grazie alla comunità e alla filosofia open source che hanno reso possibile la realizzazione di questa tesi, il modello migliorerà costantemente permettendo anche di implementare le modifiche presenti nelle nuove versioni del NAO che sicuramente verranno prodotte.

Appendice A

Installazione

In questa sezione sono riportate le istruzioni per installare tutto il software necessario per poter utilizzare il modello del NAO in Gazebo e V-REP. La versione utilizzata di ROS è la Fuerte ed è stata installata in Ubuntu 12.04 a 32 bit. Per l'installazione si vedano le istruzioni riportate in [34]. La directory `~/ros_workspace` verrà usata come directory di lavoro supponendo che tale directory sia configurata come *overlays* [35].

A.1 Gli stack `nao_robot` e `nao_common`

In questo capitolo si trovano le istruzioni per installare gli stack `humanoid_msgs`, `nao_robot` e `nao_common`. Questi stack sono sviluppati dall'Università di Friburgo e offrono le funzionalità relative ai robot umanoidi e in particolare al NAO descritte nella sezione 1.2.1.

1. creare una directory chiamata `nao` in `~/ros_workspace` e spostarsi dentro a questa:

```
$ mkdir ~/ros_workspace/nao
$ cd ~/ros_workspace/nao
```

2. creare un file di testo `rosinstall.txt` contenente le seguenti righe[1]:

```
-svn:
  uri: https://alufr-ros-
       pkg.googlecode.com/svn/trunk/humanoid_stacks/humanoid_msgs
  local-name: stacks/humanoid_msgs

-svn:
  uri: https://alufr-ros-
       pkg.googlecode.com/svn/trunk/humanoid_stacks/nao_robot
```

[1] Prestare attenzione alle tabulazioni, inoltre il link che segue la stinga `uri:` deve essere completamente specificato in un'unica riga.

```
  local-name: stacks/nao_robot

-svn:
  uri: https://alufr-ros-
       pkg.googlecode.com/svn/trunk/humanoid_stacks/nao_common
  local-name: stacks/nao_common
```

3. eseguire il seguente comando:

```
$ rosinstall . /opt/ros/fuerte rosinstall.txt
```

4. aggiungere in fondo al file `~/.bashrc` la seguente riga:

```
ROS_PACKAGE_PATH=~/ros_workspace/nao:$ROS_PACKAGE_PATH
```

   dopodiché chiudere e riaprire il terminale per applicare le modifiche.

5. compilare gli stack

```
$ rosmake humanoid_msgs nao_robot nao_common
```

A.2 I driver NAOqi

Per poter utilizzare il NAO è necessario installare i driver NAOqi[2].

1. creare una nuova cartella in un percorso facilmente raggiungibile

```
$ mkdir ~/naoqi
```

2. estrarre la versione del NAOqi SDK compatibile con il sistema operativo in uso, nella directory `~/naoqi`. Ad esempio per Ubuntu a 32 bit l'SDK viene estratto in `~/naoqi/naoqi-sdk-1.12-linux32`;

3. aggiungere le seguenti righe in fondo al file `~/.bashrc`

```
export NAOQIPATH=~/naoqi/naoqi-sdk-1.12-linux32
export PYTHONPATH=$NAOQIPATH/lib:$PYTHONPATH
```

   dopodiché chiudere e riaprire il terminale per applicare le modifiche;

4. testare il funzionamento dei driver eseguendo il seguente comando:

```
$ $NAOQIPATH/naoqi
```

   il risultato dovrebbe essere simile a quello di Fig. A.1; premere CTRL+C per uscire.

[2]Si possono scaricare da `http://users.aldebaran-robotics.com/`, è necessario essere utenti registrati per poterli scaricare.

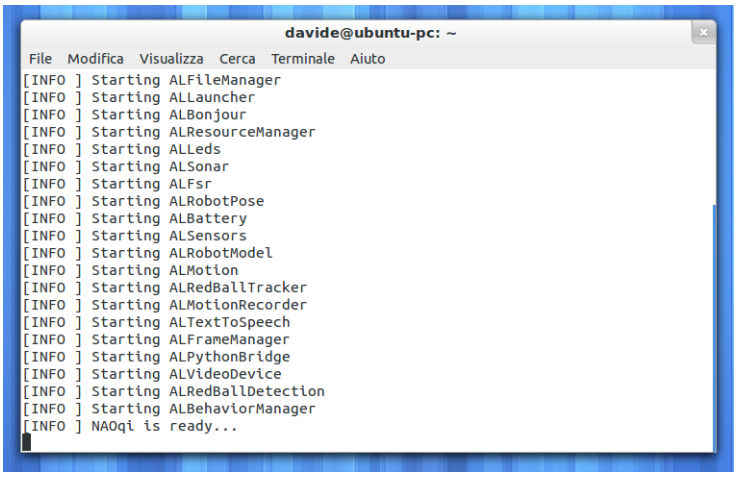


Figura A.1: Driver NAOqi in esecuzione.

A.3 Lo stack `robot_NAO`

Nello stack `robot_NAO` si trovano tutti gli stack che sono stati sviluppati per estendere le funzionalità già presenti in ROS.

- `nao_model`: in questo package si trova il modello del NAO;
- `nao_gazebo_plugin`: permette di simulare il NAO all'interno di Gazebo;
- `nao_example`: in questo package si trovano alcuni esempio che utilizzano il NAO;
- `nao_v_rep`: permette di simulare il NAO all'interno di V-REP.

Le istruzioni per scaricare e compilare i package sono riportate di seguito.

1. Scaricare i package nella propria home directory[3]:

   ```
   $ cd ~
   $ git clone gitolite@robotics.dei.unipd.it:robot_NAO
   ```

2. Aggiungere la directory `~/robot_NAO` alla variabile `ROS_PACKAGE_PATH` aggiungendo alla fine del file `~/.bashrc` la seguente riga:

   ```
   ROS_PACKAGE_PATH=~/robot_NAO:$ROS_PACKAGE_PATH
   ```

   dopodiché chiudere e riaprire il terminale per applicare le modifiche.

3. Spostarsi nella directory di ciascun package e creare il progetto per Eclipse. Il package `nao_v_rep` richiede più lavoro, verrà quindi trattato per ultimo.

[3]Il software verrà reso pubblico a breve `http://www.ros.org/wiki/robot_NAO`.

```
$ roscd nao_model
$ cmake -G“Eclipse CDT4 -Unix Makefiles”

$ roscd nao_gazebo_plugin
$ cmake -G“Eclipse CDT4 -Unix Makefiles”

$ roscd nao_example
$ cmake -G“Eclipse CDT4 -Unix Makefiles”
```

4. Compilare i package con il seguente commando:

```
$ rosmake nao_model nao_gazebo_plugin nao_example
```

5. Abilitare la possibilità di eseguire[4] i file `nao_example/bin/nao_driver` e `nao_example/bin/nao_driver_sim` e `nao_example/bin/NAOqi`

Nel caso si volesse simulare il NAO anche in V-REP è neccessario prima installare il programma seguento le istruzioni riportate in A.4 e poi procedere con la configurazione e la compilazione del package `nao_v_rep`, come descritto di seguito.

1. Spostarsi nella cartella del package e creare i seguenti link simbolici

```
$ roscd nao_v_rep

$ ln -s $V_REP_PATH/programming/include/v_repConst.h
    include/v_repConst.h
$ ln -s $V_REP_PATH/programming/include/v_repLib.h
    include/v_repLib.h
$ ln -s $V_REP_PATH/programming/common/v_repLib.cpp
    src/v_repLib.cpp
```

2. Creare il progetto per Eclipse

```
$ cmake -G“Eclipse CDT4 -Unix Makefiles”
```

3. Compilare il package

```
$ rosmake nao_v_rep
```

4. Abilitare la possibilità di eseguire[5] il file `nao_v_rep/bin/nao_v_rep`.

[4]Dalla scheda `Permessi` della finestra proprietà del file abilitare `Consentire l'esecuzione del file come programma`.

[5]Dalla scheda `Permessi` della finestra proprietà del file abilitare `Consentire l'esecuzione del file come programma`.

A.4 Installazione e configurazione di V-REP

Per poter usare V-REP e integrarlo in ROS è necessario seguire le seguenti istruzioni.

1. Scaricare da `http://coppeliarobotics.com/downloads.html` una versione di V-REP compatibile con il proprio sistema operativo e estrarre l'archivio in una directory, ad esempio in `~/V-REP`;

2. Aggiungere alla fine del file `~/bashrc` la seguente riga:

```
export V_REP_PATH=~/V-REP
```

3. Copiare la directory `programming/ros_stacks/vrep` presente all'interno all'archivio di V-REP all'interno del proprio workspace di ROS, ad esempio

```
$ cp -R ~/V-REP/programming/ros_stacks/vrep ~/ros_workspace
```

4. Compilare il package `vrep_common`

```
$ roscd vrep_common
$ make
```

5. Spostarsi all'interno del package `vrep_plugin`, creare i seguenti link simbolici e compilare il package

```
$ roscd vrep_plugin

$ ln -s $V_REP_PATH/programming/include/v_repConst.h
    include/v_repConst.h
$ ln -s $V_REP_PATH/programming/include/v_repLib.h
    include/v_repLib.h
$ ln -s $V_REP_PATH/programming/common/v_repLib.cpp
    src/v_repLib.cpp

$ make
```

6. Copiare il file `libv_repExtRos.so` all'interno della cartella di V-REP

```
$ cp lib/libv_repExtRos.so $V_REP_PATH
```

7. Per testare la corretta installazione avviare in un terminale il master di ROS con il commando

```
$ roscore
```

e in un altro terminale spostarsi nella cartella di installazione di V-REP e avviarlo, il risultato dovrebbe essere simile a questo:

```
$ ./vrep.sh
License file 'v_rep':
---> ok
Simulator launched.
Plugin 'BubbleRob': loading...
Plugin 'BubbleRob': load succeeded.
Plugin 'K3': loading...
Plugin 'K3': load succeeded.
Plugin 'RemoteApi': loading...
Plugin 'RemoteApi': load succeeded.
Plugin 'Ros': loading...
Plugin 'Ros': load succeeded.
```

in particolare verificare che il plugin di ROS sia stato caricato con successo.

Maggiori dettagli sulla procedura di installazione si possono trovare in [36].

A.5 Installazione e configurazione di Rviz

In questo paragrafo è descritto come installare e configurare rviz per simulare il NAO.

1. Installare i pacchetti `ros-fuerte-rospack` e `ros-fuerte-visualization`.

   ```
   $ sudo apt-get install ros-fuerte-rospack ros-fuerte-visualization
   ```

2. Avviare i driver NAOqi e il pacchetto `nao_driver`, eseguendo in terminali diversi i seguenti comandi:

   ```
   $ $NAOQIPATH/naoqi --verbose --broker-ip 127.0.0.1

   $ $LD_LIBRARY_PATH=$NAOQIPATH/lib:$LD_LIBRARY_PATH
       NAO_IP=127.0.0.1 roslaunch nao_driver nao_driver_sim.launch
   ```

3. Avviare `robot_state_publisher` e caricare il modello URDF del NAO

   ```
   $ roslaunch nao_description nao_state_publisher.launch
   ```

4. Avviare rviz

   ```
   $ rosrun rviz rviz
   ```

se tutto è andato a buon fine, verrà caricata la GUI[6].

5. In alto a sinistra nella finestra `Dispalys`, cambiare il valore da `Fixed Frame` a `/base_link`. Il valore in `Target Frame` deve essere `<Fixed Frame>`. A questo punto il valore di `Global status` dovrebbe essere `OK`. Premere il bottone `add` e aggiungere la griglia (`grid`) e poi premere di nuovo `add` per aggiungere il modello del robot (`RobotModel`). Per abilitare la visualizzazione dei sistemi di riferimento di ciascun giunto è sufficiente abilitare l'opzione `Show Axes` di ciascun giunto (`Links`) del modello. Il risultato dovrebbe essere simile a quello di Fig. 1.2.

6. Uscire da RViz e premere il bottone `Save` per salvare le configurazioni.

[6]Se è la prima volta che RViz viene avviato potrebbe essere visualizzato un errore relativo alla mancanza del file di configurazione. Al termine di questa procedura il file di configurazione verrà creato e questo errore non si presenterà più.

Appendice B

Esecuzione di alcuni esempi

In questo capitolo si trovano alcuni esempi che descrivono come utilizzare le funzionalità descritte nei precedenti capitoli.

B.1 Lo stack `robot_NAO`

Per poter eseguire tutti gli esempi descritti in questo capitolo è necessario aver installato lo stack `robot_NAO` (vedi A.3), i driver NAOqi (vedi A.2) e aver installato e configurato V-REP (vedi A.4).

B.1.1 Il package `nao_model`

In questo package è contenuto il nuovo modello del NAO; l'organizzazione delle directory è riporatata di seguito.

- `mesh` contiene tutte le mesh del robot, per comodità in tutti i formati in cui sono state convertite;
- `launch` contiene i file con estensione `launch` usati per caricare il modello nei simulatori;
- `urdf` contiene i file URDF e Xacro che descrivono il modello.

Per convertire il modello Xacro, contenuto nel file `urdf/nao_robot_v4.xacro` sono sufficienti i seguenti comandi:

```
$ roscd nao_model/urdf
$ rosrun xacro xacro.py nao_robot_v4.xacro > nao_robot_v4.urdf
```

che non fanno altro che spostarsi nella directory `urdf` del package `nao_model` e usare il package `xacro` per interpretare le macro presenti nel file `nao_robot_v4.xacro` e scriverle nel file `nao_robot_v4.urdf` (vedi 2.2.3).

Il file `launch/robot_state_publisher.launch`, il cui codice è riportato in Cod. B.1, avvia il nodo `robot_state_publisher` che pubblica lo stato dei giunti del robot.

```
1 <launch>
2   <param name="robot_description" command="$(find xacro)/xacro.py '$(find nao_model)
        /urdf/nao_robot_v4.xacro'" />
3   <node pkg="robot_state_publisher" type="state_publisher" name="robot_state_publisher"/>
4 </launch>
```

Codice B.1: Il file `robot_state_publisher.launch`.

Il file `launch/joint_state_publisher.lauch` (Cod. B.2) include il codice appena visto (riga 2) e avvia il nodo `joint_state_publisher` (righe 5-7) che, attraverso una comoda GUI, permette di cambiare la posizione dei giunti del robot.

```
1 <launch>
2   <include file="$(find nao_model)/launch/robot_state_publisher.launch"/>
3
4   <arg name="gui" default="true" />
5   <param name="use_gui" value="$(arg gui)"/>
6   <node pkg="joint_state_publisher" type="joint_state_publisher" name="state_publisher"/>
7 </launch>
```

Codice B.2: Il file `launch/joint_state_publisher.launch`.

In questo stack si trovano anche i file per avviare i simulatori supportati in ROS. Il file `launch/rviz.launch` avvia rviz.

```
1 <launch>
2   <!-- open Rviz-->
3   <node name="rviz" pkg="rviz" type="rviz" required="true"/>
4 </launch>
```

Codice B.3: Il file `rviz.launch`.

I file `launch/v_rep.launch` e `launch/gazebo_fuerte.launch` avviano rispettivamente V-REP e Gazebo come descritto nei paragrafi 3.1.1 e 3.1.2. Una volta che il modello è stato caricato in un simulatore è possibile muovere i giunti del robot avviando con il commando `roslaunch` il file `joint_state_publisher.launch`. Ad esempio per eseguire la simulazione in Gazebo sono sufficienti i seguenti due comandi (da eseguire in terminali diversi):

```
$ roslaunch nao_model joint_state_publisher.launch
$ roslaunch nao_model gazebo_fuerte.launch
```

il risultato ottenuto si può vedere in `http://youtu.be/lZv3M3eM8Kc`.

B.1.2 Il package `nao_example`

Il package `nao_example` contiene quattro piccolo esempi che mostrato come implementare dei nodi che utilizzino le funzionalità descritte nei capitoli precedenti.

Muovere i giunti Il primo esempio che verrà analizzato consiste nel muovere i giunti del robot; il file sorgente associato a questo esempio è `src/move_joint.cpp` ed è composto dal solo `main`, di seguito sono analizzate le parti fondamentali.

```
1  //creo il publisher e imposto i nomi dei giunti che voglio spostare
2  ros::NodeHandle n;
3  ros::Publisher pub = n.advertise<nao_msgs::JointAnglesWithSpeed>("/joint_angles", 1,
       true);
4  std::vector<std::string> name;
5  name.push_back("RShoulderRoll");
6  name.push_back("LShoulderRoll");
7  name.push_back("HeadYaw");
8  std::vector<float> angles;
```

Codice B.4: Il file `src/move_joint.cpp`(1).

In riga 1 viene creato un `ros::Publisher`, cioè un oggetto che ha il compito di pubblicare dei messaggi; la creazione avviane utilizzando il metodo `advertise` della clsse `ros::NodeHandle` specificando il topic in cui pubblicare i messaggi (`/joint_angles`), la lunghezza della coda di messaggi (1) e un flag che specifica se l'utilmo messaggio deve essere salvato e inviato ad eventuali nuovi subscriber che si iscrivono al topic dopo esso è stato pubblicato. Nelle righe 4-8 viene creato un vettore `name` in cui vengono inseriti i nomi dei giunti da muovere e un vettore `angles` in cui vengono inseriti gli angoli che i giunti devono assumere.

```
1 nao_msgs::JointAnglesWithSpeed msg;
2 msg.joint_names = name;
3 msg.speed = 0.1;
```

Codice B.5: Il file `src/move_joint.cpp`(2).

Successivamente (righe 1-3 di Cod. B.5) viene creato il messaggio da inviare.

Infine il ciclo `while` di Cod. B.6 alterna ogni quattro secondi la pubblicazione di un messaggio contenente due posizioni diverse per i giunti *RShoulderRoll*, *LShoulderRoll* e *HeadYaw*.

Il file per eseguire questo esempio è `launch/move_joint.launch` (vedi Cod. B.7). Questo file semplicemente avvia i driver del NAO (riga 2), avvia il nodo `robot_state_publisher` (riga 3) già visto in B.1.1 e avvia l'eseguibile ottenuto dalla compilazione del sorgente appena visto (riga 4).

Anche in questo caso è possibile visualizzare l'esempio nel simulatore preferito avviando il corrispondente file `launch` contenuto nel package `nao_model` (vedi B.1.1). Ad esempio per avere la simulazione in V-REP è sufficiente lanciare i seguenti comandi in terminali diversi:

```
$ roslaunch nao_example move_joint.launch
$ roslaunch nao_model v_rep.launch
```

e il risultato che si ottiene si può vedere in `http://youtu.be/FEIwuolKR_g`.

```
while(n.ok()){
  angles.clear();
  angles.push_back(−1.3);
  angles.push_back(1.3);
  angles.push_back(−1.15);
  msg.joint_angles = angles;
  pub.publish(msg);
  sleep(4);

  angles.clear();
  angles.push_back(0);
  angles.push_back(0);
  angles.push_back(1.15);
  msg.joint_angles = angles;
  pub.publish(msg);
  sleep(4);
}
```

Codice B.6: Il file `src/move_joint.cpp`(3).

```
<launch>
  <include file="$(find nao_example)/launch/driver.launch"/>
  <include file="$(find nao_model)/launch/robot_state_publisher.launch"/>
  <node pkg="nao_example" type="move_joint" name="move_joint" output="screen"/>
</launch>
```

Codice B.7: Il file `move_joint.launch`.

Gestione delle pose Il secondo esempio prevede la gestione delle posture di un robot. Usando la classe `actionlib::SimpleActionClient` è possibile fare assumere al robot delle posture predefinite o create dall'utente. In modo automatico verrà calcolato il percorso migliore che i giunti devono compiere per passare dalla postura attuale a quella richiesta. In questo esempio si può vedere il robot che passa dalla postura dritta in piedi (`body_pose`) ad una accovacciata (`crouch`).

Il codice sorgente di questo esempio si trova nel file `src/nao_pose.cpp`.

```
ros::NodeHandle n;
actionlib::SimpleActionClient<nao_msgs::BodyPoseAction> client("body_pose", true);
if (!client.waitForServer(ros::Duration(10.0)))
  ROS_ERROR("client.waitServer failed");
```

Codice B.8: Il file `src/nao_pose.cpp`(1).

Per prima cosa viene creato un `SimpleActionClient` che permette l'invio un messaggio contenente la posa che il robot deve essumere (Cod. B.8).

Il ciclo `while` alterna l'invio di un messaggio contenente la postura `init` e la postura `crounch` (Cod. B.9).

Per avviare l'esempio è sufficiente il seguente comando:

```
$ roslaunch nao_example nao_pose.launch
```

```
nao_msgs::BodyPoseGoal goal;
while(n.ok()){
  sleep(3);
  goal.pose_name = "init";
  client.sendGoalAndWait(goal, ros::Duration(5.0));

  sleep(3);
  goal.pose_name = "crouch";
  client.sendGoalAndWait(goal, ros::Duration(5.0));
}
```

Codice B.9: Il file `src/nao_pose.cpp`(2).

Anche in questo caso è possibile utilizzare il simulatore preferito per la simulazione utilizzando i file `launch` presenti nel package `nao_model`.

```
<launch>
  <include file="$(find nao_example)/launch/driver.launch"/>
  <include file="$(find nao_model)/launch/robot_state_publisher.launch"/>

  <!-- nao_remote e pose_manager -->
  <node pkg="nao_remote" type="remap_odometry" name="remap_odometry" output="screen"/>
  <node pkg="nao_remote" type="pose_manager.py" name="pose_manager" >
    <rosparam file="$(find nao_remote)/config/basic_poses.yaml" command="load" ns="poses"/>
  </node>

  <node pkg="nao_example" type="nao_pose" name="nao_pose" output="screen"></node>
</launch>
```

Codice B.10: Il file `nao_pose.launch`

In `http://youtu.be/5ZrUAztrtU4` si può vedere l'esempio simulato in Rviz.

Registrare e riprodurre dei movimenti In questo esempio vengono registrati i movimenti del robot usando il comando `rosbag` [37] e visualizzate sulla console le posizioni attuali dei giunti.

Per prima cosa è necessario avviare uno degli esempi appena visti dopodiché lanciare la registrazione con il comando

```
$ roslaunch nao_example rec_joint.launch
```

```
<launch>
   <node pkg="nao_example" type="rec_joint" name="rec_joint" output="screen"></node>
   <node pkg="rosbag" type="record" name="record"
         output="screen" args="/joint_states -o $(find nao_example)/nao"></node>
</launch>
```

Codice B.11: Il file `rec_joint.launch`.

Verrà creato nella directory `nao_example` un file `nao_*.bag` dove `*` rappresenta la data e l'ora attuale. I file con estensione `.bag` possono essere riprodotti

con `rosbag`, per fare ciò è necessario modificare la riga 9 del file `nao_example/launch/rosbag_play.launch` (Cod. B.12) sostituendo la stringa `filename` con il nome del file che è stato creato eseguendo la registrazione.

```
1 <launch>
2   <include file="$(find nao_model)/launch/robot_state_publisher.launch"/>
3   <param name="/use_sim_time" value="true" />
4   <node pkg="rosbag" type="play" name="play" output="screen" args="--clock -d 3 $(find
        nao_example)/filename.bag" required="true"/>
5   <node pkg="nao_example" type="rec_joint" name="rec_joint" output="screen" />
6 </launch>
```

Codice B.12: Il file `rosbag_play.launch`.

Per avviare la riproduzione della registrazione usare:

```
$ roslaunch nao_example rosbag_play.launch
```

Anche in questo caso è possibile visualizzare la simulazione con il programma preferito, si otterà un risultato simile quanto visto negli esempi precedenti solo che questa volta i movimenti sono quelli registrati nel file `.bag` usato come input.

Appendice C

Esecuzione dei test

In questo appendice è descritto come sono implementati nel package `nao_test` i due test descritti nel capitolo 4. In particolare ciascun test è implementato con una classe, `WalkAndRec` per il primo e `RotAndRec` per il secondo, che ha il compito di muovere il robot, bloccarlo quando è stato raggiunto l'obiettivo stabilito e registrare i dati di interesse. Il `main` dei due test si trova nei file `test1.cpp` e `test2.cpp`. Infine per ciascun test sono presenti tre file `.launch`: uno per la simulazioni in Gazebo, uno per la simulazione in V-REP e uno per controllare il robot reale.

C.1 Test 1: camminata rettilinea

In Cod. C.1 è riportata la definizione del costruttore della classe `WalkAndRec`.

```
1 WalkAndRec(geometry_msgs::Point dist, geometry_msgs::Twist vel, WalkAndRec::Type type
      , std::string suffix, ros::NodeHandle* node_handle = 0);
```

Codice C.1: Dichiarazione del costruttore della classe `WalkAndRec`.

Sono richiesti i seguenti parametri:

- `dist` specifica la distanza che il robot deve percorrere nelle tre direzioni, affinché il robot si fermi devono essere soddisfatte tutte le tre condizioni;
- `vel` indica la velocità con cui il robot si muove nelle tre direzioni e nella rotazione su sé stesso;
- `type` è un enum che specifica se si sta lavorando con il robot reale, in Gazebo oppure in V-REP;
- `suffix` è il suffisso che viene applicato al nome del file in cui vengono registrati i dati;

- `node_handle` è un oggetto che viene usato per registrare le callback per pubblicare o registrare dei messaggi, se non specificato ne viene creato uno dal costruttore.

Il costruttore esegue principalmente quattro operazioni:

1. Prepara il file in cui vengono registrate le posizioni del robot al passare del tempo, se si sta usando il robot reale vengono registrati i dati provenienti dall'odometria altrimenti sono registrati i dati provenienti dal simulatore;

2. Sottoscrive le callback che registrano i dati e la callback che verifica se il robot ha raggiunto l'obiettivo stabilito;

3. Avvia la simulazione in V-REP (se il test utilzza V-REP);

4. Sottoscrive il topic in cui viene inviato il messaggio che farà muovere il robot.

Per avviare la simulazione è implementato il metodo `start` che pubblica un messaggio nel topic `/cmd_vel` per mettere il robot in movimento e memorizza l'istante temporale in cui la camminata è cominciata in modo da poter registrare le posizioni del robot utilizzando il tempo relativo. Nel caso del robot reale prima di usare il metodo `start` è necessario invocare il metodo `startWalkPose` che fa assumere al NAO la posizione di inizio camminata, in modo da poterlo posizionare correttamente nel pavimento e facilitare la misura della posizione finale.

Sono presenti le tre callback che registrano i dati: `torsoOdometryCallback`, `gazeboLinkStateCallback` e `vrepTorsoCallback` (Cod. C.2).

```
1 void torsoOdometryCallback(const nao_msgs::TorsoOdometry msg);
2 void gazeboLinkStateCallback(const gazebo_msgs::LinkStates msg);
3 void vrepTorsoCallback(const geometry_msgs::PoseStamped msg);
```

Codice C.2: Callback della classe `WalkAndRec`.

La callback che legge i dati dall'odometria è attiva anche nel caso il test riguardi una simulazione perché verifica quando il robot debba essere fermato ma non memorizza su file alcun valore. Quando il robot raggiunge la posizione stabilita la registrazione continua per qualche secondo in modo da registrare il punto esatto in cui il robot si ferma. Le callback dei due simulatori sono molto simili, l'unica differenza riguarda il diverso modo con cui i due software pubblicano la posizione dei link del robot.

Il `main` caricare i parametri dal test, crea un oggetto della classe `WalkAndRec` e avvia la simulazione; maggiori dettagli relativi al passaggio dei parametri sono descritti nel paragrafo C.3.

C.2 Test 2: rotazione sul posto

Il costruttore della classe `RotAndRec` (Cod. C.3) è molto simile a quello già visto della classe `WalkAndRec` (Cod. C.1). I parametri sono li stessi tranne per il primo, `target`, che specifica l'angolo che il robot deve compiere. Anche le operazioni eseguite da questo costruttore sono praticamente le stesse del costruttore della classe `WalkAndRec`.

```
1 RotAndRec(double target, geometry_msgs::Twist vel, RotAndRec::Type type, std::string suffix
      , ros::NodeHandle* node_handel = 0);
```

Codice C.3: Dichiarazione del costruttore della classe `RotAndRec`.

Anche in questo caso sono disponibili i metodi `start`, per avvia la simulazione, `startWalkPose`, per far assumere al robot reale la posizione di inizio camminata, e le tre callback per la registrazione dei dati provenienti da Gazebo e V-REP e la gestione dei dati provenienti dall'odometria.

In questo caso la gestione dei dati è più complessa rispetto al primo test in quanto è necessario gestire degli angoli, inoltre ciascun simulatore e l'odometria usano convenzioni diverse per stabilire l'orientazione del robot. Infine bisogna convertire un quaternione alle corrispondenti rotazioni di Eulero.

```
1 double getRad(const double& angle_degree);
2 double getDegree(const double& angle_rad);
3
4 double getVREPRot(const geometry_msgs::Quaternion& orient);
5 double getGazeboRot(const geometry_msgs::Quaternion& orient);
6
7 double getDistance(const double& from_rad, const double& to_rad);
```

Codice C.4: Metodi della classe `RotAndRec` per la gestione delle rotazioni.

Per questi motivi sono stati implementati alcuni metodi privati per facilitare (Cod. C.4). Come suggerisce il nome, i metodi `getRad` e `getDegree` convertono rispettivamente un angolo da gradi a radianti e viceversa. I metodi `getVREPRot` e `getGazeboRot` restituiscono l'angolo in radianti che indica la rotazione compiuta dal robot a partire da un quaternione, il primo per V-REP il secondo per Gazebo. Sono necessari due metodi diversi per ciascun simulatore perché l'angolo di partenza del robot è diverso. Infine il metodo `getDistance` calcola l'angolo percorso da `from_rad` e `to_rad`, entrambi in radianti, tenendo conto dalla senso della rotazione con cui il robot si sta muovendo.

Il `main` è molto simile a quello del primo test.

C.3 Esecuzione dei test

Per l'esecuzione dei test, per lo meno per quelli riguardanti la simulazione, si è cercato di rendere il processo il più automatizzato possibile. Per prima cosa in

tutti i file `.launch` sono definite delle variabili che specificano le opzioni del test (Cod. C.5). Una variabile viene definita utilizzando il tag `arg`, specificandone il nome (`name`) ed indicandone eventualmente il valore di default che la variabile deve assumere nel caso non venga definita da riga di comando (`default`).

```
<launch>
  <arg name="pos_x" default="1.0" />
  <arg name="pos_y" />
  <arg name="pos_z" />

  <arg name="vel_lin_x" default="1.0" />
  <arg name="vel_lin_y" default="0.0" />
  <arg name="vel_lin_z" default="0.0" />
  <arg name="vel_ang_x" default="0.0" />
  <arg name="vel_ang_y" default="0.0" />
  <arg name="vel_ang_z" default="0.0" />

  <arg name="type" default="gazebo" />
  <arg name="suffix" default="1-1-1" />
   ...
</launch>
```

Codice C.5: Dichiarazione delle opzioni del test nei file `.launch`.

Dopo aver dichiarato le variabile è necessario specificare a quali nodi debbano essere passate come parametri (Cod. C.6). Il passaggio dei parametri ad un nodo avviene utilizzando il tag `param` e specificando il nome del parametro (`name`) e il valore di default (`default`) che deve assumere se la variabile non è ancora stata definita. È importante notare che queste definizioni devono essere fatte all'interno del tag `node` affinché il nodo possa accedere ai parametri definiti.

```
<launch>
  ...
  <node pkg="nao_test" type="test1" name="test1" output="screen" required="true">
    <param name="pos_x" value="$(arg pos_x)" />
    <param name="pos_y" value="$(arg pos_y)" />
    <param name="pos_z" value="$(arg pos_z)" />
    <param name="vel_lin_x" value="$(arg vel_lin_x)" />
    <param name="vel_lin_y" value="$(arg vel_lin_y)" />
    <param name="vel_lin_z" value="$(arg vel_lin_z)" />
    <param name="vel_ang_x" value="$(arg vel_ang_x)" />
    <param name="vel_ang_y" value="$(arg vel_ang_y)" />
    <param name="vel_ang_z" value="$(arg vel_ang_z)" />
    <param name="type" value="$(arg type)" />
    <param name="suffix" value="$(arg suffix)" />
  </node>
</launch>
```

Codice C.6: Passaggio dei parametri al nodo `nao_test`.

In questo modo è possibile scrivere degli script bash in cui vengono impostate tutte le variabili necessarie per il test e avviare il corrispondente `.launch`. Se ad

esempio si volesse eseguire le 7 ripetizioni del test camminata rettilinea di 0.5 m a velocità 80% è sufficiente il seguente script:

```
pos_x=0.5
vel_lin_x=0.8
type="gazebo"

for number in $(seq 1 7);
do
    roslaunch nao_test test1_gazebo.launch pos_x:=$pos_x vel_lin_x:=
        $vel_lin_x type:="$type" suffix:="$pos_x-$vel_lin_x-$number"
done
```

La variabile `suffix` viene utilizzata per definire un suffisso da applicare al nome del file in cui vengono salvati i dati registrati, ad esempio in questo caso si otterrà il file `torso_gazebo_0.4-_0.8-x.txt` con `x` che varia da 1 a 7. Usando in modo adeguato questi script è quindi possibile automatizzare l'esecuzione dei test. Se si utilizza V-REP è necessario avviare prima il simulatore e caricare il modello (vedi B.1.1), impostare il motore fisico desiderato e avviare i test.

Caricare i parametri da un package è piuttosto semplice, ad esempio in Cod. C.7 è riportato il codice per caricare i parametri del primo test. Viene usato il metodo `param` della classe `ros::NodeHadle`, questo metodo vuole tre parametri: il nome del parametro, la variabile in cui memorizzare il valore e il valore di default da assegnare alla variabile nel caso il parametro non sia definito. Nelle righe 3-4 vengono caricati il tipo di test (reale, Gazebo o V-REP) e il suffisso da applicare al file con i dati, nelle righe 6-9 vengono caricate le distanze che il robot deve percorrere e infine nelle righe 11-17 vengono caricate le velocità con cui si deve muovere.

```
1  ros::NodeHandle* nodeHandle = new ros::NodeHandle("~");
2
3  nodeHandle->param("type", type, std::string("real"));
4  nodeHandle->param("suffix", suffix, std::string("1-1-1"));
5
6  geometry_msgs::Point dist;
7  nodeHandle->param("pos_x", dist.x, 3.0);
8  nodeHandle->param("pos_y", dist.y, -std::numeric_limits<double>::infinity());
9  nodeHandle->param("pos_z", dist.z, -std::numeric_limits<double>::infinity());
10
11 geometry_msgs::Twist vel;
12 nodeHandle->param("vel_lin_x", vel.linear.x, 0.4);
13 nodeHandle->param("vel_lin_y", vel.linear.y, 0.0);
14 nodeHandle->param("vel_lin_z", vel.linear.z, 0.0);
15 nodeHandle->param("vel_ang_x", vel.angular.x, 0.0);
16 nodeHandle->param("vel_ang_y", vel.angular.y, 0.0);
17 nodeHandle->param("vel_ang_z", vel.angular.z, 0.0);
```

Codice C.7: Istruzioni per il caricamento dei parametri nel file `test1.cpp`.

Appendice D

Risultati dei test

D.1 Test 1: camminata rettilinea

In questa sezione si trovano i risultati ottenuti nel primo test, camminata rettilinea, descritto in 4.2. Per ciascun simulatore e per l'odometria sono riportati i dati di ciascuna delle sette prove, in particolare è specificata la distanza percorsa (x), la deviazione (y) e l'altezza finale (z) del centro di massa del robot la cui posizione iniziale è (0; 0; 0.333), tutte le distanze sono in metri. La variabile t misura il tempo in secondi necessario per completare il test. Nel caso del robot reale sono state misurate solo la distanza frontale percorsa e la deviazione. Sono anche riportate le medie e le deviazioni standard delle tre prove. Infine in tabella D.6 sono riepilogate le medie e in tabella D.7 le deviazioni standard di tutti i simulatori e del robot reale in ciascun test.

TEST A: vel. = 40% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	-	0.535	-0.050	-
2	-	0.533	-0.055	-
3	-	0.544	-0.020	-
4	-	0.555	-0.025	-
5	-	0.555	0.010	-
6	-	0.532	-0.040	-
7	-	0.535	-0.045	-
E[]	-	0.541	-0.032	-
σ()	-	0.010	0.023	-

TEST B: vel. = 40% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	-	1.030	-0.200	-
2	-	1.032	-0.245	-
3	-	1.055	-0.195	-
4	-	1.057	-0.090	-
5	-	1.060	-0.163	-
6	-	1.080	-0.060	-
7	-	1.080	-0.105	-
E[]	-	1.056	-0.151	-
σ()	-	0.020	0.068	-

TEST C: vel. = 40% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	-	2.575	1.940	-
2	-	3.020	1.410	-
3	-	2.285	2.100	-
4	-	2.290	2.080	-
5	-	2.840	1.590	-
6	-	2.680	1.850	-
7	-	2.565	1.860	-
E[]	-	2.608	1.833	-
σ()	-	0.270	0.253	-

TEST D: vel. = 80% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	-	0.520	-0.025	-
2	-	0.515	-0.010	-
3	-	0.520	-0.005	-
4	-	0.520	-0.030	-
5	-	0.530	-0.025	-
6	-	0.525	-0.025	-
7	-	0.520	-0.010	-
E[]	-	0.521	-0.019	-
σ()	-	0.005	0.010	-

TEST E: vel. = 80% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	-	1.080	-0.030	-
2	-	1.060	0.055	-
3	-	1.065	0.030	-
4	-	1.055	0.040	-
5	-	1.075	0.040	-
6	-	1.060	0.020	-
7	-	1.070	-0.020	-
E[]	-	1.066	0.019	-
σ()	-	0.009	0.032	-

TEST F: vel. = 80% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	-	3.160	0.445	-
2	-	3.000	0.840	-
3	-	3.050	0.740	-
4	-	3.205	0.020	-
5	-	2.850	0.085	-
6	-	2.925	1.040	-
7	-	3.070	0.780	-
E[]	-	3.037	0.564	-
σ()	-	0.125	0.392	-

TEST G: vel. = 100% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	-	0.515	-0.020	-
2	-	0.520	-0.005	-
3	-	0.510	-0.035	-
4	-	0.500	-0.020	-
5	-	0.500	-0.015	-
6	-	0.510	-0.040	-
7	-	0.515	-0.040	-
E[]	-	0.510	-0.025	-
σ()	-	0.008	0.014	-

TEST H: vel. = 100% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	-	1.090	0.130	-
2	-	1.115	0.145	-
3	-	1.120	0.160	-
4	-	1.110	0.085	-
5	-	1.095	0.085	-
6	-	1.120	0.195	-
7	-	1.070	0.255	-
E[]	-	1.103	0.151	-
σ()	-	0.019	0.061	-

TEST I: vel. = 100% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	-	2.120	1.950	-
2	-	2.155	1.950	-
3	-	2.190	1.925	-
4	-	2.350	1.790	-
5	-	2.360	2.015	-
6	-	2.205	1.900	-
7	-	2.355	1.820	-
E[]	-	2.248	1.907	-
σ()	-	0.104	0.079	-

Tabella D.1: Test camminata rettilinea: distanze percorse dal robot reale.

TEST A: vel. = 40% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	17.897	0.402	0.016	0.310
2	18.439	0.367	-0.007	0.310
3	19.048	0.402	-0.009	0.310
4	17.484	0.411	-0.002	0.310
5	17.591	0.413	0.006	0.310
6	17.780	0.423	0.025	0.310
7	17.428	0.406	0.010	0.310
E[]	17.952	0.403	0.006	0.310
σ()	0.591	0.018	0.012	0.000

TEST B: vel. = 40% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	38.814	0.843	0.036	0.310
2	34.996	0.814	-0.008	0.310
3	34.208	0.788	0.013	0.310
4	34.489	0.817	0.035	0.310
5	35.686	0.851	0.037	0.310
6	35.656	0.813	-0.010	0.310
7	35.970	0.748	-0.001	0.310
E[]	35.688	0.811	0.015	0.310
σ()	1.525	0.035	0.021	0.000

TEST C: vel. = 40% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	95.485	2.445	0.002	0.310
2	95.429	2.316	0.032	0.310
3	95.488	2.331	0.014	0.310
4	95.447	2.477	0.010	0.310
5	95.478	2.336	-0.016	0.310
6	95.542	2.509	-0.067	0.310
7	95.398	2.500	0.027	0.310
E[]	95.467	2.416	0.000	0.310
σ()	0.047	0.086	0.034	0.000

TEST D: vel. = 80% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	9.593	0.372	0.032	0.308
2	9.981	0.388	0.017	0.309
3	9.586	0.393	0.037	0.308
4	9.329	0.374	0.019	0.303
5	9.411	0.398	0.044	0.307
6	9.778	0.363	0.021	0.309
7	9.700	0.380	0.001	0.309
E[]	9.625	0.381	0.024	0.307
σ()	0.221	0.012	0.015	0.002

TEST E: vel. = 80% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	16.949	0.723	-0.026	0.308
2	18.421	0.748	0.010	0.309
3	18.818	0.713	0.009	0.309
4	18.094	0.757	-0.010	0.309
5	17.893	0.750	-0.002	0.309
6	18.499	0.746	-0.032	0.309
7	17.701	0.694	-0.002	0.309
E[]	18.054	0.733	-0.008	0.308
σ()	0.618	0.023	0.016	0.000

TEST F: vel. = 80% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	47.784	2.212	0.017	0.304
2	49.410	2.150	0.002	0.309
3	49.333	2.234	-0.002	0.309
4	49.366	2.267	-0.006	0.309
5	49.415	1.936	0.014	0.309
6	49.491	1.962	0.012	0.309
7	49.426	2.247	0.014	0.309
E[]	49.175	2.144	0.007	0.308
σ()	0.615	0.139	0.009	0.002

TEST G: vel. = 100% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	9.121	0.387	-0.005	0.308
2	9.171	0.356	-0.005	0.308
3	9.185	0.385	-0.006	0.308
4	9.209	0.393	0.016	0.308
5	9.320	0.319	0.001	0.308
6	8.916	0.303	-0.002	0.308
7	8.879	0.390	-0.002	0.308
E[]	9.114	0.362	0.000	0.308
σ()	0.161	0.037	0.008	0.000

TEST H: vel. = 100% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	13.191	0.780	0.001	0.308
2	13.091	0.788	0.015	0.308
3	13.198	0.777	0.009	0.308
4	13.116	0.788	0.015	0.308
5	13.187	0.799	0.011	0.308
6	13.158	0.794	0.008	0.308
7	13.126	0.780	0.006	0.308
E[]	13.152	0.787	0.009	0.308
σ()	0.042	0.008	0.005	0.000

TEST I: vel. = 100% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	40.150	2.165	-0.013	0.308
2	40.191	2.072	0.009	0.308
3	40.121	1.933	-0.011	0.308
4	40.211	2.124	-0.008	0.308
5	40.097	2.136	-0.026	0.308
6	40.138	2.101	-0.017	0.308
7	40.176	1.992	0.005	0.308
E[]	40.155	2.075	-0.009	0.308
σ()	0.040	0.083	0.012	0.000

Tabella D.2: Test camminata rettilinea: distanze percorse in Gazebo.

TEST A: vel. = 40% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	17.612	0.453	0.019	0.329
2	17.619	0.460	0.000	0.330
3	17.627	0.466	-0.001	0.330
4	17.592	0.450	0.032	0.330
5	17.633	0.461	0.007	0.330
6	17.620	0.466	0.017	0.330
7	17.625	0.462	-0.014	0.329
E[]	17.618	0.460	0.009	0.330
σ()	0.014	0.006	0.015	0.000

TEST B: vel. = 40% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	35.398	0.913	0.072	0.330
2	33.276	0.931	-0.032	0.330
3	33.247	0.921	0.079	0.330
4	33.267	0.900	-0.048	0.330
5	33.233	0.915	0.059	0.329
6	33.280	0.939	0.008	0.330
7	33.247	0.912	0.006	0.330
E[]	33.564	0.919	0.020	0.330
σ()	0.809	0.013	0.051	0.000

TEST C: vel. = 40% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	93.595	2.703	-0.067	0.328
2	93.541	2.726	-0.057	0.328
3	93.565	2.694	0.161	0.329
4	93.514	2.698	0.017	0.329
5	93.563	2.710	-0.018	0.329
6	93.552	2.715	0.027	0.328
7	93.605	2.695	0.203	0.328
E[]	93.562	2.706	0.038	0.328
σ()	0.031	0.012	0.105	0.000

TEST D: vel. = 80% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	10.417	0.478	-0.013	0.328
2	10.397	0.466	0.021	0.328
3	10.440	0.463	0.004	0.328
4	10.108	0.470	-0.017	0.328
5	10.417	0.478	-0.013	0.328
6	10.390	0.463	0.005	0.328
7	10.443	0.473	0.022	0.328
E[]	10.373	0.470	0.001	0.328
σ()	0.119	0.006	0.016	0.000

TEST E: vel. = 80% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	18.171	0.897	0.050	0.328
2	18.162	0.905	0.016	0.328
3	18.175	0.892	-0.018	0.328
4	18.147	0.890	-0.067	0.328
5	17.740	0.881	0.012	0.328
6	18.192	0.921	0.025	0.328
7	18.164	0.887	0.078	0.328
E[]	18.107	0.896	0.014	0.328
σ()	0.163	0.013	0.047	0.000

TEST F: vel. = 80% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	47.528	2.617	-0.212	0.328
2	47.529	2.599	0.061	0.327
3	47.483	2.595	-0.196	0.327
4	47.441	2.588	-0.199	0.327
5	47.460	2.561	0.389	0.328
6	47.475	2.533	0.318	0.327
7	47.553	2.591	0.070	0.328
E[]	47.495	2.583	0.033	0.328
σ()	0.041	0.028	0.250	0.000

TEST G: vel. = 100% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	8.901	0.427	-0.007	0.331
2	8.915	0.427	0.023	0.331
3	8.921	0.473	0.018	0.328
4	8.597	0.429	0.021	0.332
5	8.752	0.423	0.012	0.331
6	8.916	0.433	0.008	0.331
7	8.920	0.429	0.000	0.331
E[]	8.846	0.434	0.011	0.331
σ()	0.126	0.017	0.011	0.001

TEST H: vel. = 100% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	10.210	0.802	-0.097	0.328
2	10.180	0.806	-0.029	0.328
3	10.160	0.795	-0.090	0.330
4	10.200	0.793	-0.037	0.331
5	10.260	0.795	0.007	0.328
6	10.120	0.833	0.061	0.327
7	9.950	0.824	0.037	0.328
E[]	10.154	0.807	-0.021	0.328
σ()	0.100	0.015	0.060	0.001

TEST I: vel. = 100% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	38.256	2.483	0.094	0.329
2	38.225	2.518	-0.040	0.328
3	38.231	2.454	-0.528	0.328
4	38.200	2.551	0.127	0.328
5	38.227	2.501	-0.350	0.328
6	38.236	2.540	-0.101	0.329
7	38.226	2.493	0.286	0.331
E[]	38.229	2.506	-0.073	0.329
σ()	0.017	0.034	0.284	0.001

Tabella D.3: Test camminata rettilinea: distanze percorse in V-REP con il motore fisico ODE.

TEST A: vel. = 40% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	17.594	0.297	-0.012	0.334
2	17.582	0.296	0.000	0.334
3	17.621	0.304	-0.005	0.334
4	17.627	0.294	-0.010	0.334
5	17.639	0.294	-0.007	0.334
6	17.622	0.290	-0.010	0.334
7	17.587	0.298	-0.002	0.334
E[]	17.610	0.296	-0.006	0.334
σ()	0.022	0.004	0.004	0.000

TEST B: vel. = 40% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	33.229	0.640	0.166	0.333
2	33.259	0.600	0.175	0.333
3	33.263	0.641	0.175	0.333
4	33.275	0.630	0.168	0.333
5	33.267	0.543	0.309	0.333
6	33.240	0.609	0.215	0.333
7	33.274	0.597	0.271	0.333
E[]	33.258	0.609	0.211	0.333
σ()	0.017	0.034	0.057	0.000

TEST C: vel. = 40% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	95.513	0.612	1.171	0.334
2	95.600	0.704	1.181	0.334
3	95.537	0.705	1.164	0.334
4	95.517	0.492	1.123	0.334
5	95.566	0.659	1.168	0.334
6	95.540	0.748	1.182	0.334
7	95.607	0.772	1.142	0.334
E[]	95.554	0.670	1.162	0.334
σ()	0.038	0.095	0.022	0.000

TEST D: vel. = 80% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	10.436	0.381	0.027	0.332
2	10.426	0.390	0.017	0.332
3	10.381	0.394	-0.002	0.332
4	10.397	0.391	0.027	0.332
5	10.434	0.392	0.000	0.332
6	10.268	0.381	0.043	0.332
7	10.407	0.389	0.022	0.332
E[]	10.393	0.388	0.019	0.332
σ()	0.059	0.005	0.016	0.000

TEST E: vel. = 80% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	18.103	0.728	0.109	0.332
2	18.157	0.739	0.060	0.332
3	18.001	0.740	0.095	0.332
4	18.175	0.736	0.082	0.332
5	17.804	0.728	0.094	0.332
6	17.674	0.695	0.175	0.332
7	18.156	0.724	0.126	0.332
E[]	18.010	0.727	0.106	0.332
σ()	0.198	0.016	0.037	0.000

TEST F: vel. = 80% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	49.504	1.557	1.159	0.332
2	49.482	1.571	1.172	0.332
3	49.509	1.111	1.462	0.332
4	49.480	1.512	1.217	0.332
5	49.541	1.220	1.422	0.332
6	49.553	1.551	1.171	0.332
7	49.420	1.587	1.136	0.332
E[]	49.498	1.444	1.248	0.332
σ()	0.044	0.194	0.135	0.000

TEST G: vel. = 100% dist. = 0,5 m

#	t (s)	X (m)	Y (m)	Z (m)
1	8.922	0.434	0.018	0.331
2	8.796	0.437	0.002	0.331
3	8.714	0.431	-0.011	0.331
4	8.705	0.434	0.020	0.331
5	8.440	0.419	0.016	0.332
6	8.894	0.420	0.020	0.331
7	8.924	0.426	-0.019	0.331
E[]	8.771	0.429	0.007	0.331
σ()	0.173	0.007	0.016	0.000

TEST H: vel. = 100% dist. = 1 m

#	t (s)	X (m)	Y (m)	Z (m)
1	9.460	0.708	0.144	0.331
2	9.580	0.717	0.128	0.331
3	9.410	0.732	0.056	0.331
4	9.470	0.718	0.125	0.331
5	9.400	0.726	0.058	0.332
6	9.550	0.716	0.095	0.331
7	9.580	0.701	0.126	0.331
E[]	9.493	0.717	0.104	0.331
σ()	0.077	0.010	0.036	0.000

TEST I: vel. = 100% dist. = 3 m

#	t (s)	X (m)	Y (m)	Z (m)
1	40.182	1.960	0.946	0.331
2	40.164	1.924	1.119	0.331
3	40.245	2.010	0.942	0.331
4	40.201	1.861	1.040	0.331
5	40.231	2.061	0.840	0.331
6	40.271	2.011	1.011	0.331
7	40.247	2.039	0.940	0.331
E[]	40.220	1.981	0.977	0.331
σ()	0.039	0.070	0.089	0.000

Tabella D.4: Test camminata rettilinea: distanze percorse in V-REP con il motore fisico Bullet.

TEST A: vel. = 40% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	17.530	0.500	-0.002	0.311
2	17.587	0.500	-0.001	0.311
3	17.529	0.499	-0.001	0.311
4	17.791	0.515	-0.002	0.311
5	17.801	0.516	-0.001	0.311
6	17.720	0.499	0.003	0.311
7	17.697	0.515	-0.001	0.311
E[]	17.665	0.506	-0.001	0.311
σ()	0.116	0.008	0.002	0.000

TEST B: vel. = 40% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	33.227	1.012	0.001	0.311
2	33.336	1.011	-0.004	0.311
3	33.328	1.012	-0.003	0.311
4	32.654	0.995	-0.004	0.311
5	33.293	1.011	-0.002	0.311
6	33.310	1.012	-0.003	0.311
7	33.326	1.012	-0.003	0.311
E[]	33.211	1.009	-0.003	0.311
σ()	0.248	0.006	0.002	0.000

TEST C: vel. = 40% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	95.472	2.995	0.001	0.311
2	95.555	2.995	0.001	0.311
3	95.491	2.995	0.001	0.311
4	95.467	2.995	0.001	0.311
5	95.523	2.995	0.001	0.311
6	95.488	2.995	0.001	0.311
7	95.567	2.995	0.001	0.311
E[]	95.509	2.995	0.001	0.311
σ()	0.040	0.000	0.000	0.000

TEST D: vel. = 80% dist. = 0,5 m

#	t (s)	x (m)	y (m)	z (m)
1	9.873	0.502	0.002	0.310
2	9.913	0.503	0.002	0.310
3	9.966	0.509	-0.002	0.310
4	9.950	0.502	0.003	0.310
5	9.942	0.509	-0.002	0.310
6	9.964	0.509	-0.002	0.310
7	9.981	0.509	-0.001	0.310
E[]	9.941	0.506	0.000	0.310
σ()	0.037	0.003	0.002	0.000

TEST E: vel. = 80% dist. = 1 m

#	t (s)	x (m)	y (m)	z (m)
1	17.747	1.008	0.002	0.311
2	17.753	1.012	0.002	0.310
3	18.439	1.016	0.001	0.310
4	17.794	1.013	0.002	0.310
5	17.751	1.012	0.002	0.311
6	17.794	1.014	0.002	0.310
7	17.779	1.013	0.002	0.310
E[]	17.865	1.013	0.002	0.310
σ()	0.254	0.002	0.000	0.000

TEST F: vel. = 80% dist. = 3 m

#	t (s)	x (m)	y (m)	z (m)
1	49.453	3.035	0.001	0.310
2	49.428	3.035	0.001	0.310
3	49.468	3.035	0.001	0.310
4	49.427	3.035	0.001	0.310
5	49.490	3.035	0.001	0.310
6	49.502	3.035	0.001	0.310
7	49.373	3.035	0.001	0.310
E[]	49.449	3.035	0.001	0.310
σ()	0.044	0.000	0.000	0.000

TEST G: vel. = 100% dist. = 0,5 m

#	t (s)	X (m)	Y (m)	Z (m)
1	8.349	0.503	0.002	0.310
2	8.418	0.513	-0.002	0.309
3	8.408	0.513	-0.002	0.309
4	8.364	0.497	0.002	0.311
5	8.319	0.497	0.002	0.311
6	8.454	0.513	-0.002	0.309
7	8.343	0.513	-0.002	0.309
E[]	8.379	0.507	0.000	0.310
σ()	0.048	0.008	0.002	0.001

TEST H: vel. = 100% dist. = 1 m

#	t (s)	X (m)	Y (m)	Z (m)
1	15.177	1.031	-0.001	0.309
2	15.135	1.031	-0.001	0.309
3	15.181	1.031	-0.001	0.309
4	15.179	1.031	-0.001	0.309
5	15.190	1.031	-0.001	0.309
6	15.191	1.031	-0.001	0.309
7	15.194	1.031	-0.001	0.309
E[]	15.178	1.031	-0.001	0.309
σ()	0.020	0.000	0.000	0.000

TEST I: vel. = 100% dist. = 3 m

#	t (s)	X (m)	Y (m)	Z (m)
1	40.137	3.031	-0.001	0.309
2	40.113	3.031	-0.001	0.309
3	40.194	3.031	-0.001	0.309
4	40.153	3.031	-0.001	0.309
5	40.183	3.031	-0.001	0.309
6	40.221	3.031	-0.001	0.309
7	40.198	3.031	-0.001	0.309
E[]	40.171	3.031	-0.001	0.309
σ()	0.038	0.000	0.000	0.000

Tabella D.5: Test camminata rettilinea: distanze percorse secondo l'odometria.

TEST A: vel. = 40% dist. = 0,5 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale		0.541	-0.032	
Gazebo	17.952	0.403	0.006	0.310
V-REP ODE	17.618	0.460	0.009	0.330
V-REP Bullet	17.610	0.296	-0.006	0.334
Odometria	17.665	0.506	-0.001	0.311

TEST B: vel. = 40% dist. = 1 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale		1.056	-0.151	
Gazebo	35.688	0.811	0.015	0.310
V-REP ODE	33.564	0.919	0.020	0.330
V-REP Bullet	33.258	0.609	0.211	0.333
Odometria	33.211	1.009	-0.003	0.311

TEST C: vel. = 40% dist. = 3 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale		2.608	1.833	
Gazebo	95.467	2.416	0.000	0.310
V-REP ODE	93.562	2.706	0.038	0.328
V-REP Bullet	95.554	0.670	1.162	0.334
Odometria	95.509	2.995	0.001	0.311

TEST D: vel. = 80% dist. = 0,5 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale	-	0.521	-0.019	-
Gazebo	9.625	0.381	0.024	0.307
V-REP ODE	10.373	0.470	0.001	0.328
V-REP Bullet	10.393	0.388	0.019	0.332
Odometria	9.941	0.506	0.000	0.310

TEST E: vel. = 80% dist. = 1 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale	-	1.066	0.019	-
Gazebo	18.054	0.733	-0.008	0.308
V-REP ODE	18.107	0.896	0.014	0.328
V-REP Bullet	18.010	0.727	0.106	0.332
Odometria	17.865	1.013	0.002	0.310

TEST F: vel. = 80% dist. = 3 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale	-	3.037	0.564	-
Gazebo	49.175	2.144	0.007	0.308
V-REP ODE	47.495	2.583	0.033	0.328
V-REP Bullet	49.498	1.444	1.248	0.332
Odometria	49.449	3.035	0.001	0.310

TEST G: vel. = 100% dist. = 0,5 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale	-	0.510	-0.025	-
Gazebo	9.114	0.362	0.000	0.308
V-REP ODE	8.846	0.434	0.011	0.331
V-REP Bullet	8.771	0.429	0.007	0.331
Odometria	8.379	0.507	0.000	0.310

TEST H: vel. = 100% dist. = 1 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale	-	1.103	0.151	-
Gazebo	13.152	0.787	0.009	0.308
V-REP ODE	10.154	0.807	-0.021	0.328
V-REP Bullet	9.493	0.717	0.104	0.331
Odometria	15.178	1.031	-0.001	0.309

TEST I: vel. = 100% dist. = 3 m

	E[t] (s)	E[x] (m)	E[y] (m)	E[Z] (m)
Reale	-	2.248	1.907	-
Gazebo	40.155	2.075	-0.009	0.308
V-REP ODE	38.229	2.506	-0.073	0.329
V-REP Bullet	40.220	1.981	0.977	0.331
Odometria	40.171	3.031	-0.001	0.309

Tabella D.6: Test camminata rettilinea: distanze percorse medie.

TEST B: vel. = 40% dist. = 0,5 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.010	0.023	-
Gazebo	0.591	0.018	0.012	0.000
V-REP ODE	0.014	0.006	0.015	0.000
V-REP Bullet	0.022	0.004	0.004	0.000
Odometria	0.116	0.008	0.002	0.000

TEST B: vel. = 40% dist. = 1 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.020	0.068	-
Gazebo	1.525	0.035	0.021	0.000
V-REP ODE	0.809	0.013	0.051	0.000
V-REP Bullet	0.017	0.034	0.057	0.000
Odometria	0.248	0.006	0.002	0.000

TEST C: vel. = 40% dist. = 3 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.270	0.253	-
Gazebo	0.047	0.086	0.034	0.000
V-REP ODE	0.031	0.012	0.105	0.000
V-REP Bullet	0.038	0.095	0.022	0.000
Odometria	0.040	0.000	0.000	0.000

TEST D: vel. = 80% dist. = 0,5 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.005	0.010	-
Gazebo	0.221	0.012	0.015	0.002
V-REP ODE	0.119	0.006	0.016	0.000
V-REP Bullet	0.059	0.005	0.016	0.000
Odometria	0.037	0.003	0.002	0.000

TEST E: vel. = 80% dist. = 1 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.009	0.032	-
Gazebo	0.618	0.023	0.016	0.000
V-REP ODE	0.163	0.013	0.047	0.000
V-REP Bullet	0.198	0.016	0.037	0.000
Odometria	0.254	0.002	0.000	0.000

TEST F: vel. = 80% dist. = 3 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.125	0.392	-
Gazebo	0.615	0.139	0.009	0.002
V-REP ODE	0.041	0.028	0.250	0.000
V-REP Bullet	0.044	0.194	0.135	0.000
Odometria	0.044	0.000	0.000	0.000

TEST G: vel. = 100% dist. = 0,5 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.008	0.014	-
Gazebo	0.161	0.037	0.008	0.000
V-REP ODE	0.126	0.017	0.011	0.001
V-REP Bullet	0.173	0.007	0.016	0.000
Odometria	0.048	0.008	0.002	0.001

TEST H: vel. = 100% dist. = 1 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.019	0.061	-
Gazebo	0.042	0.008	0.005	0.000
V-REP ODE	0.100	0.015	0.060	0.001
V-REP Bullet	0.077	0.010	0.036	0.000
Odometria	0.020	0.000	0.000	0.000

TEST I: vel. = 100% dist. = 3 m

	σ(t) (s)	σ(x) (m)	σ(y) (m)	σ(z) (m)
Reale	-	0.104	0.079	-
Gazebo	0.040	0.083	0.012	0.000
V-REP ODE	0.017	0.034	0.284	0.001
V-REP Bullet	0.039	0.070	0.089	0.000
Odometria	0.038	0.000	0.000	0.000

Tabella D.7: Test camminata rettilinea: deviazioni standard delle distanze percorse.

D.2 Test 2: rotazione sul posto

In questa sezione si trovano i risultati del secondo test, rotazione sul posto, descritto in 4.3. Per ciascun simulatore e per l'odometria sono riportati i dati di ciascuna delle sette prove, in particolare è specificato l'angolo percorso (Ang.) in gradi e in radianti, e il tempo necessario per completare la prova. Nel caso reale il tempo non è stato misurato. È anche riportata la media e la deviazione standard delle sette prove. Infine in Tab. D.13 sono riepilogate le medie e in Tab. D.14 le deviazioni standard di tutti i simulatori e del robot reale in ciascuna prova.

TEST L: vel. = 40% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	-	105	1.833
2	-	103	1.798
3	-	95	1.658
4	-	104	1.815
5	-	99	1.728
6	-	98	1.710
7	-	101	1.763
E[]	-	100.714	1.758
σ()	-	3.592	0.063

TEST M: vel. = 40% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	-	400	6.981
2	-	410	7.156
3	-	403	7.034
4	-	403	7.034
5	-	405	7.069
6	-	420	7.330
7	-	398	6.946
E[]	-	405.571	7.079
σ()	-	7.413	0.129

TEST N: vel. = 40% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	-	225	3.927
2	-	225	3.927
3	-	218	3.805
4	-	230	4.014
5	-	220	3.840
6	-	224	3.910
7	-	223	3.892
E[]	-	223.571	3.902
σ()	-	3.867	0.067

TEST O: vel. = 40% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	-	325	5.672
2	-	317	5.533
3	-	323	5.637
4	-	325	5.672
5	-	326	5.690
6	-	320	5.585
7	-	322	5.620
E[]	-	322.571	5.630
σ()	-	3.207	0.056

TEST P: vel. = 80% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	-	115	2.007
2	-	112	1.955
3	-	106	1.850
4	-	105	1.833
5	-	112	1.955
6	-	114	1.990
7	-	97	1.693
E[]	-	108.714	1.897
σ()	-	6.422	0.112

TEST Q: vel. = 80% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	-	418	7.295
2	-	412	7.191
3	-	415	7.243
4	-	410	7.156
5	-	402	7.016
6	-	410	7.156
7	-	403	7.034
E[]	-	410.000	7.156
σ()	-	5.859	0.102

TEST R: vel. = 80% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	-	212	3.700
2	-	225	3.927
3	-	213	3.718
4	-	235	4.102
5	-	234	4.084
6	-	215	3.752
7	-	225	3.927
E[]	-	222.714	3.887
σ()	-	9.639	0.168

TEST S: vel. = 80% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	-	320	5.585
2	-	328	5.725
3	-	334	5.829
4	-	322	5.620
5	-	335	5.847
6	-	326	5.690
7	-	315	5.498
E[]	-	325.714	5.685
σ()	-	7.319	0.128

TEST T: vel. = 100% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	-	119	2.077
2	-	105	1.833
3	-	114	1.990
4	-	106	1.850
5	-	113	1.972
6	-	110	1.920
7	-	104	1.815
E[]	-	110.143	1.922
σ()	-	5.521	0.096

TEST U: vel. = 100% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	-	405	7.069
2	-	412	7.191
3	-	418	7.295
4	-	420	7.330
5	-	408	7.121
6	-	410	7.156
7	-	415	7.243
E[]	-	412.571	7.201
σ()	-	5.412	0.094

TEST V: vel. = 100% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	-	225	3.927
2	-	210	3.665
3	-	215	3.752
4	-	217	3.787
5	-	238	4.154
6	-	208	3.630
7	-	225	3.927
E[]	-	219.714	3.835
σ()	-	10.420	0.182

TEST Z: vel. = 100% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	-	320	5.585
2	-	336	5.864
3	-	325	5.672
4	-	325	5.672
5	-	326	5.690
6	-	316	5.515
7	-	315	5.498
E[]	-	323.286	5.642
σ()	-	7.158	0.125

Tabella D.8: Test rotazione sul posto: angolo percorso dal robot reale.

TEST L: vel. = 40% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	6.916	36.696	0.640
2	6.947	35.152	0.614
3	6.942	32.289	0.564
4	6.848	35.412	0.618
5	6.912	35.863	0.626
6	6.851	33.178	0.579
7	6.921	39.259	0.685
E[]	6.905	35.407	0.618
σ()	0.040	2.290	0.040

TEST M: vel. = 40% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	23.821	147.157	2.568
2	23.806	155.977	2.722
3	23.832	156.810	2.737
4	23.877	149.531	2.610
5	23.820	158.394	2.764
6	23.891	154.695	2.700
7	23.798	145.451	2.539
E[]	23.835	152.574	2.663
σ()	0.035	5.119	0.089

TEST N: vel. = 40% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	12.610	72.589	1.267
2	12.557	74.894	1.307
3	12.591	67.275	1.174
4	12.614	70.448	1.230
5	12.600	84.047	1.467
6	12.641	78.688	1.373
7	12.568	71.101	1.241
E[]	12.597	74.149	1.294
σ()	0.029	5.654	0.099

TEST O: vel. = 40% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	18.311	105.644	1.844
2	18.298	108.451	1.893
3	18.357	110.627	1.931
4	18.339	121.756	2.125
5	18.259	107.275	1.872
6	18.266	109.499	1.911
7	18.292	117.405	2.049
E[]	18.303	111.522	1.946
σ()	0.036	5.863	0.102

TEST P: vel. = 80% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	4.271	41.433	0.723
2	4.314	34.907	0.609
3	4.455	50.285	0.878
4	4.363	42.424	0.740
5	4.258	42.169	0.736
6	4.314	35.431	0.618
7	4.321	31.520	0.550
E[]	4.328	39.738	0.694
σ()	0.066	6.280	0.110

TEST Q: vel. = 80% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	12.914	165.049	2.881
2	12.981	161.558	2.820
3	12.955	190.915	3.332
4	12.966	166.334	2.903
5	13.004	169.999	2.967
6	12.991	181.693	3.171
7	13.037	180.213	3.145
E[]	12.978	173.680	3.031
σ()	0.039	10.746	0.188

TEST R: vel. = 80% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	7.248	81.020	1.414
2	7.288	84.968	1.483
3	7.238	87.433	1.526
4	7.308	88.386	1.543
5	7.246	89.037	1.554
6	7.304	87.072	1.520
7	7.260	97.192	1.696
E[]	7.270	87.872	1.534
σ()	0.029	4.911	0.086

TEST S: vel. = 80% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	10.187	122.232	2.133
2	10.132	121.382	2.119
3	10.217	123.142	2.149
4	10.165	133.606	2.332
5	10.196	125.354	2.188
6	10.146	138.719	2.421
7	10.121	125.000	2.182
E[]	10.166	127.062	2.218
σ()	0.035	6.543	0.114

TEST T: vel. = 100% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	3.955	48.820	0.852
2	3.849	34.835	0.608
3	3.956	41.303	0.721
4	3.910	44.364	0.774
5	3.855	45.369	0.792
6	3.967	40.692	0.710
7	3.861	48.194	0.841
E[]	3.908	43.368	0.757
σ()	0.052	4.869	0.085

TEST U: vel. = 100% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	11.711	182.057	3.178
2	11.801	189.187	3.302
3	11.789	173.577	3.030
4	11.718	192.045	3.352
5	11.855	164.844	2.877
6	11.805	158.905	2.773
7	11.717	180.305	3.147
E[]	11.771	177.274	3.094
σ()	0.056	12.232	0.213

TEST V: vel. = 100% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	6.443	78.443	1.369
2	6.380	91.415	1.595
3	6.433	108.858	1.900
4	6.392	85.628	1.494
5	6.459	90.123	1.573
6	6.319	85.209	1.487
7	6.375	89.605	1.564
E[]	6.400	89.897	1.569
σ()	0.049	9.428	0.165

TEST Z: vel. = 100% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	9.051	141.772	2.474
2	9.114	131.039	2.287
3	9.066	134.533	2.348
4	9.055	138.493	2.417
5	9.132	133.691	2.333
6	9.032	113.245	1.976
7	9.089	128.371	2.240
E[]	9.077	131.592	2.297
σ()	0.036	9.234	0.161

Tabella D.9: Test rotazione sul posto: angolo percorso in Gazebo.

TEST L: vel. = 40% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	6.997	66.117	1.154
2	6.935	43.119	0.753
3	6.972	76.082	1.328
4	6.936	26.432	0.461
5	6.973	27.989	0.489
6	7.002	51.798	0.904
7	6.981	29.605	0.517
E[]	6.971	45.877	0.801
σ()	0.027	19.696	0.344

TEST M: vel. = 40% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	23.848	186.219	3.250
2	23.934	169.671	2.961
3	23.881	180.578	3.152
4	23.904	111.479	1.946
5	23.936	127.699	2.229
6	23.954	194.087	3.387
7	23.868	150.029	2.618
E[]	23.903	159.966	2.792
σ()	0.039	31.261	0.546

TEST N: vel. = 40% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	12.654	94.955	1.657
2	12.652	86.744	1.514
3	12.711	83.532	1.458
4	12.604	75.419	1.316
5	12.729	127.177	2.220
6	12.698	70.845	1.236
7	12.639	83.358	1.455
E[]	12.669	88.861	1.551
σ()	0.044	18.591	0.324

TEST O: vel. = 40% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	18.317	133.167	2.324
2	18.345	143.920	2.512
3	18.368	95.971	1.675
4	18.351	120.159	2.097
5	18.335	105.592	1.843
6	18.378	144.131	2.516
7	18.350	95.686	1.670
E[]	18.349	119.804	2.091
σ()	0.020	21.228	0.370

TEST P: vel. = 80% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	4.403	38.023	0.664
2	4.422	39.704	0.693
3	4.366	43.747	0.764
4	4.358	34.146	0.596
5	4.342	51.676	0.902
6	4.385	43.398	0.757
7	4.332	56.397	0.984
E[]	4.373	43.870	0.766
σ()	0.032	7.791	0.136

TEST Q: vel. = 80% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	13.038	109.249	1.907
2	13.008	173.210	3.023
3	13.032	148.221	2.587
4	13.063	157.436	2.748
5	13.020	116.944	2.041
6	13.068	142.023	2.479
7	13.023	158.550	2.767
E[]	13.036	143.662	2.507
σ()	0.022	23.116	0.403

TEST R: vel. = 80% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	7.345	59.785	1.043
2	7.352	96.158	1.678
3	7.279	44.150	0.771
4	7.303	90.285	1.576
5	7.270	70.283	1.227
6	7.313	76.987	1.344
7	7.253	71.078	1.241
E[]	7.302	72.675	1.268
σ()	0.037	17.646	0.308

TEST S: vel. = 80% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	10.246	109.175	1.905
2	10.245	88.997	1.553
3	10.255	128.893	2.250
4	10.210	127.927	2.233
5	10.281	105.840	1.847
6	10.243	60.993	1.065
7	10.213	124.524	2.173
E[]	10.242	106.621	1.861
σ()	0.025	24.732	0.432

TEST T: vel. = 100% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	3.979	37.822	0.660
2	4.050	29.707	0.518
3	3.989	43.608	0.761
4	3.995	41.631	0.727
5	3.944	35.397	0.618
6	3.981	91.336	1.594
7	3.936	25.257	0.441
E[]	3.982	43.537	0.760
σ()	0.038	22.033	0.385

TEST U: vel. = 100% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	11.819	145.163	2.534
2	11.872	160.630	2.804
3	11.854	185.794	3.243
4	11.814	177.705	3.102
5	11.834	131.238	2.291
6	11.857	144.284	2.518
7	11.890	171.883	3.000
E[]	11.848	159.528	2.784
σ()	0.028	20.055	0.350

TEST V: vel. = 100% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	6.445	73.375	1.281
2	6.510	72.641	1.268
3	6.499	89.692	1.565
4	6.493	65.186	1.138
5	6.488	65.665	1.146
6	6.514	75.614	1.320
7	6.403	75.902	1.325
E[]	6.479	74.011	1.292
σ()	0.040	8.184	0.143

TEST Z: vel. = 100% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	9.114	124.619	2.175
2	9.098	83.532	1.458
3	9.203	122.333	2.135
4	9.142	112.502	1.964
5	9.131	91.853	1.603
6	9.209	122.258	2.134
7	9.210	125.535	2.191
E[]	9.158	111.805	1.951
σ()	0.048	17.179	0.300

Tabella D.10: Test rotazione sul posto: angolo percorso in V-REP con il motore fisico ODE.

TEST L: vel. = 40% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	7.005	63.254	1.104
2	6.961	68.916	1.203
3	6.929	74.342	1.298
4	6.988	74.238	1.296
5	6.962	62.665	1.094
6	7.051	79.717	1.391
7	6.944	71.244	1.243
E[]	6.977	70.625	1.233
σ()	0.041	6.199	0.108

TEST M: vel. = 40% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	23.929	292.090	5.098
2	23.896	269.948	4.711
3	23.832	283.883	4.955
4	23.896	276.706	4.829
5	23.900	278.898	4.868
6	23.866	267.760	4.673
7	23.891	281.557	4.914
E[]	23.887	278.692	4.864
σ()	0.030	8.313	0.145

TEST N: vel. = 40% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	12.666	147.855	2.581
2	12.620	149.426	2.608
3	12.712	151.397	2.642
4	12.695	145.684	2.543
5	12.707	144.048	2.514
6	12.634	135.285	2.361
7	12.685	151.837	2.650
E[]	12.674	146.505	2.557
σ()	0.036	5.705	0.100

TEST O: vel. = 40% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	18.373	199.439	3.481
2	18.377	225.344	3.933
3	18.323	221.052	3.858
4	18.373	199.439	3.481
5	18.350	231.794	4.046
6	18.377	225.344	3.933
7	18.323	221.052	3.858
E[]	18.357	217.638	3.798
σ()	0.025	12.941	0.226

TEST P: vel. = 80% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	4.346	63.972	1.117
2	4.366	68.279	1.192
3	4.430	72.688	1.269
4	4.356	63.822	1.114
5	4.366	60.206	1.051
6	4.411	67.438	1.177
7	4.329	64.387	1.124
E[]	4.372	65.827	1.149
σ()	0.036	4.017	0.070

TEST Q: vel. = 80% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	13.070	239.626	4.182
2	13.043	265.822	4.639
3	13.066	274.100	4.784
4	13.036	269.324	4.701
5	13.104	272.931	4.764
6	13.067	266.798	4.657
7	12.985	267.236	4.664
E[]	13.053	265.120	4.627
σ()	0.037	11.669	0.204

TEST R: vel. = 80% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	7.416	168.569	2.942
2	7.316	103.043	1.798
3	7.290	137.667	2.403
4	7.355	140.130	2.446
5	7.315	134.138	2.341
6	7.307	129.869	2.267
7	7.341	129.038	2.252
E[]	7.334	134.636	2.350
σ()	0.042	19.336	0.337

TEST S: vel. = 80% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	10.293	195.451	3.411
2	10.229	188.595	3.292
3	10.306	177.460	3.097
4	10.230	188.753	3.294
5	10.283	208.163	3.633
6	10.219	176.749	3.085
7	10.242	213.572	3.728
E[]	10.258	192.678	3.363
σ()	0.036	14.151	0.247

TEST T: vel. = 100% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	3.976	56.277	0.982
2	3.987	69.183	1.207
3	3.996	69.388	1.211
4	3.980	60.676	1.059
5	3.947	47.202	0.824
6	3.998	50.740	0.886
7	4.001	60.639	1.058
E[]	3.984	59.158	1.032
σ()	0.019	8.483	0.148

TEST U: vel. = 100% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	11.893	276.840	4.832
2	11.847	278.150	4.855
3	11.825	258.180	4.506
4	11.874	261.358	4.562
5	11.838	274.153	4.785
6	11.880	250.619	4.374
7	11.896	263.748	4.603
E[]	11.865	266.150	4.645
σ()	0.028	10.455	0.182

TEST V: vel. = 100% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	6.458	108.479	1.893
2	6.459	142.973	2.495
3	6.461	134.266	2.343
4	6.528	137.470	2.399
5	6.518	129.082	2.253
6	6.484	128.057	2.235
7	6.428	135.852	2.371
E[]	6.476	130.883	2.284
σ()	0.036	11.097	0.194

TEST Z: vel. = 100% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	9.175	201.877	3.523
2	9.136	190.440	3.324
3	9.194	184.219	3.215
4	9.110	179.870	3.139
5	9.070	208.220	3.634
6	9.175	180.063	3.143
7	9.137	199.178	3.476
E[]	9.142	191.981	3.351
σ()	0.043	11.290	0.197

Tabella D.11: Test rotazione sul posto: angolo percorso in V-REP con il motore fisico Bullet.

TEST L: vel. = 40% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	7.837	91.974	1.605
2	8.147	91.953	1.605
3	8.160	91.970	1.605
4	7.994	91.953	1.605
5	7.955	91.971	1.605
6	8.255	91.981	1.605
7	8.015	91.990	1.606
E[]	8.052	91.970	1.605
σ()	0.143	0.014	0.000

TEST M: vel. = 40% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	24.951	363.959	6.352
2	25.403	371.971	6.492
3	24.613	363.752	6.349
4	24.920	363.955	6.352
5	24.983	363.978	6.353
6	24.915	363.991	6.353
7	24.243	363.190	6.339
E[]	24.861	364.971	6.370
σ()	0.357	3.100	0.054

TEST N: vel. = 40% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	13.792	187.785	3.277
2	13.159	187.159	3.267
3	13.853	187.954	3.280
4	13.756	195.282	3.408
5	13.803	187.802	3.278
6	13.743	187.951	3.280
7	12.572	179.855	3.139
E[]	13.525	187.684	3.276
σ()	0.483	4.462	0.078

TEST O: vel. = 40% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	19.317	275.974	4.817
2	19.369	275.981	4.817
3	19.374	275.969	4.817
4	19.243	275.942	4.816
5	19.361	275.974	4.817
6	19.378	275.970	4.817
7	19.427	275.977	4.817
E[]	19.353	275.970	4.817
σ()	0.058	0.013	0.000

TEST P: vel. = 80% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	5.473	99.600	1.738
2	5.439	99.580	1.738
3	5.496	99.587	1.738
4	5.505	99.605	1.738
5	5.403	99.606	1.738
6	5.124	99.828	1.742
7	5.480	99.591	1.738
E[]	5.417	99.628	1.739
σ()	0.134	0.089	0.002

TEST Q: vel. = 80% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	14.344	376.343	6.568
2	14.117	376.594	6.573
3	14.449	376.952	6.579
4	14.364	376.703	6.575
5	14.335	376.179	6.566
6	14.261	376.759	6.576
7	14.216	376.719	6.575
E[]	14.298	376.607	6.573
σ()	0.109	0.264	0.005

TEST R: vel. = 80% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	6.554	161.341	2.816
2	6.658	160.993	2.810
3	8.391	192.448	3.359
4	8.421	191.881	3.349
5	8.397	191.880	3.349
6	8.424	192.438	3.359
7	8.472	191.873	3.349
E[]	7.903	183.265	3.199
σ()	0.886	15.098	0.264

TEST S: vel. = 80% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	11.421	284.690	4.969
2	11.368	284.423	4.964
3	9.679	280.105	4.889
4	11.463	284.693	4.969
5	10.979	284.054	4.958
6	11.324	284.113	4.959
7	11.314	284.120	4.959
E[]	11.078	283.743	4.952
σ()	0.637	1.626	0.028

TEST T: vel. = 100% ang. = 90°

#	t (s)	ang (°)	ang (rad)
1	5.020	94.083	1.642
2	5.014	96.692	1.688
3	5.002	94.073	1.642
4	5.073	94.056	1.642
5	5.019	94.072	1.642
6	4.866	96.702	1.688
7	4.996	96.714	1.688
E[]	4.998	95.199	1.662
σ()	0.064	1.407	0.025

TEST U: vel. = 100% ang. = 360°

#	t (s)	ang (°)	ang (rad)
1	12.931	374.218	6.531
2	12.961	374.205	6.531
3	12.945	371.556	6.485
4	12.931	371.558	6.485
5	12.744	373.788	6.524
6	12.808	374.207	6.531
7	12.878	374.197	6.531
E[]	12.885	373.390	6.517
σ()	0.081	1.261	0.022

TEST V: vel. = 100% ang. = -180°

#	t (s)	ang (°)	ang (rad)
1	7.876	197.783	3.452
2	7.700	183.408	3.201
3	7.674	183.390	3.201
4	7.851	200.751	3.504
5	7.772	197.798	3.452
6	7.691	183.394	3.201
7	7.809	197.800	3.452
E[]	7.768	192.046	3.352
σ()	0.081	8.158	0.142

TEST Z: vel. = 100% ang. = -270°

#	t (s)	ang (°)	ang (rad)
1	10.383	287.457	5.017
2	10.921	304.840	5.320
3	10.269	284.818	4.971
4	10.381	284.849	4.972
5	10.455	284.853	4.972
6	10.391	287.189	5.012
7	10.323	287.184	5.012
E[]	10.446	288.741	5.039
σ()	0.217	7.203	0.126

Tabella D.12: Test rotazione sul posto: angolo percorso secondo l'odometria.

TEST L: vel. = 40% ang. 90°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	100.714	1.758
Gazebo	6.905	35.407	0.618
V-REP ODE	6.971	45.877	0.801
V-REP Bullet	6.977	70.625	1.233
Odometria	8.052	91.970	1.605

TEST M: vel. = 40% ang. 360°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	405.571	7.079
Gazebo	23.835	152.574	2.663
V-REP ODE	23.903	159.966	2.792
V-REP Bullet	23.887	278.692	4.864
Odometria	24.861	364.971	6.370

TEST N: vel. = 40% ang. -180°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	223.571	3.902
Gazebo	12.597	74.149	1.294
V-REP ODE	12.669	88.861	1.551
V-REP Bullet	12.674	146.505	2.557
Odometria	13.525	187.684	3.276

TEST O: vel. = 40% ang. -270°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	322.571	5.630
Gazebo	18.303	111.522	1.946
V-REP ODE	18.349	119.804	2.091
V-REP Bullet	18.357	217.638	3.798
Odometria	19.353	275.970	4.817

TEST P: vel. = 80% ang. 90°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	97.000	1.693
Gazebo	4.321	31.520	0.550
V-REP ODE	4.332	56.397	0.984
V-REP Bullet	4.329	64.387	1.124
Odometria	4.998	99.591	1.738

TEST Q: vel. = 80% ang. 360°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	403.000	7.034
Gazebo	13.037	180.213	3.145
V-REP ODE	13.023	158.550	2.767
V-REP Bullet	12.985	267.236	4.664
Odometria	14.216	376.719	6.575

TEST R: vel. = 80% ang. -180°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	225.000	3.927
Gazebo	7.260	97.192	1.696
V-REP ODE	7.253	71.078	1.241
V-REP Bullet	7.341	129.038	2.252
Odometria	8.472	191.873	3.349

TEST S: vel. = 80% ang. -270°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	315.000	5.498
Gazebo	10.121	125.000	2.182
V-REP ODE	10.213	124.524	2.173
V-REP Bullet	10.242	213.572	3.728
Odometria	11.314	284.120	4.959

TEST T: vel. = 100% ang. 90°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	110.143	1.922
Gazebo	3.908	43.368	0.757
V-REP ODE	3.982	43.537	0.760
V-REP Bullet	3.984	59.158	1.032
Odometria	4.998	95.199	1.662

TEST U: vel. = 100% ang. 360°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	412.571	7.201
Gazebo	11.771	177.274	3.094
V-REP ODE	11.848	159.528	2.784
V-REP Bullet	11.865	266.150	4.645
Odometria	12.885	373.390	6.517

TEST V: vel. = 100% ang. -180°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	219.714	3.835
Gazebo	6.400	89.897	1.569
V-REP ODE	6.479	74.011	1.292
V-REP Bullet	6.476	130.883	2.284
Odometria	7.768	192.046	3.352

TEST Z: vel. = 100% ang. -270°

	E[t] (s)	E[ang] (°)	E[ang] (rad)
Reale	-	323.286	5.642
Gazebo	9.077	131.592	2.297
V-REP ODE	9.158	111.805	1.951
V-REP Bullet	9.142	191.981	3.351
Odometria	10.446	288.741	5.039

Tabella D.13: Test rotazione sul posto: angolo medio percorso.

TEST L: vel. = 40% ang. 90°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		3.592	0.063
Gazebo	0.040	2.290	0.040
V-REP ODE	0.027	19.696	0.344
V-REP Bullet	0.041	2.490	0.108
Odometria	0.143	0.014	0.000

TEST M: vel. = 40% ang. 360°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		7.413	0.129
Gazebo	0.035	5.119	0.089
V-REP ODE	0.039	31.261	0.546
V-REP Bullet	0.030	2.883	0.145
Odometria	0.357	3.100	0.054

TEST N: vel. = 40% ang. -180°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		3.867	0.067
Gazebo	0.029	5.654	0.099
V-REP ODE	0.044	18.591	0.324
V-REP Bullet	0.036	2.389	0.100
Odometria	0.483	4.462	0.078

TEST O: vel. = 40% ang. -270°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		3.207	0.056
Gazebo	0.036	5.863	0.102
V-REP ODE	0.020	21.228	0.370
V-REP Bullet	0.025	3.597	0.226
Odometria	0.058	0.013	0.000

TEST P: vel. = 80% ang. 90°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		6.422	0.112
Gazebo	0.066	6.280	0.110
V-REP ODE	0.032	7.791	0.136
V-REP Bullet	0.036	4.017	0.070
Odometria	0.134	0.089	0.002

TEST Q: vel. = 80% ang. 360°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		5.859	0.102
Gazebo	0.039	10.746	0.188
V-REP ODE	0.022	23.116	0.403
V-REP Bullet	0.037	11.669	0.204
Odometria	0.109	0.264	0.005

TEST R: vel. = 80% ang. -180°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		9.639	0.168
Gazebo	0.029	4.911	0.086
V-REP ODE	0.037	17.646	0.308
V-REP Bullet	0.042	19.336	0.337
Odometria	0.886	15.098	0.264

TEST S: vel. = 80% ang. -270°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		7.319	0.128
Gazebo	0.035	6.543	0.114
V-REP ODE	0.025	24.732	0.432
V-REP Bullet	0.036	14.151	0.247
Odometria	0.637	1.626	0.028

TEST T: vel. = 100% ang. 90°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		5.521	0.096
Gazebo	0.052	4.869	0.085
V-REP ODE	0.038	22.033	0.385
V-REP Bullet	0.019	8.483	0.148
Odometria	0.064	1.407	0.025

TEST U: vel. = 100% ang. 360°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		5.412	0.094
Gazebo	0.056	12.232	0.213
V-REP ODE	0.028	20.055	0.350
V-REP Bullet	0.028	10.455	0.182
Odometria	0.081	1.261	0.022

TEST V: vel. = 100% ang. -180°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		10.420	0.182
Gazebo	0.049	9.428	0.165
V-REP ODE	0.040	8.184	0.143
V-REP Bullet	0.036	11.097	0.194
Odometria	0.081	8.158	0.142

TEST Z: vel. = 100% ang. -270°

	σ(t) (s)	σ(ang) (°)	σ(ang) (rad)
Reale		7.158	0.125
Gazebo	0.036	9.234	0.161
V-REP ODE	0.048	17.179	0.300
V-REP Bullet	0.043	11.290	0.197
Odometria	0.217	7.203	0.126

Tabella D.14: Test rotazione sul posto: deviazioni standard angolo percorso.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] "OGRE." http://www.ogre3d.org/. [Accessed: 2013-04-08].

[2] "ODE." http://www.ode.org/. [Accessed: 2013-04-08].

[3] "Bullet." http://bulletphysics.org/. [Accessed: 2013-04-08].

[4] "PhysX." https://developer.nvidia.com/physx. [Accessed: 2013-04-08].

[5] "Aldebaran Robotics." http://www.aldebaran-robotics.com. [Accessed: 2013-04-08].

[6] "Gazebo." http://gazebosim.org/. [Accessed: 2013-04-08].

[7] "V-REP." http://coppeliarobotics.com/. [Accessed: 2013-04-08].

[8] "ROS." http://www.ros.org/. [Accessed: 2013-04-08].

[9] D. Gouaillier, C. Collette, and C. Kilner, "Omni-directional closed-loop walk for nao," in *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, pp. 448–454, IEEE, 2010.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, 2009.

[11] "ROS wiki: repositories." http://www.ros.org/wiki/Repositories. [Accessed: 2013-04-08].

[12] "ROS wiki: rosmake." http://www.ros.org/wiki/rosmake. [Accessed: 2013-04-09].

[13] "ROS wiki: roslaunch." http://www.ros.org/wiki/roslaunch. [Accessed: 2013-04-09].

[14] "ROS wiki: tools." http://www.ros.org/wiki/Tools. [Accessed: 2013-04-08].

[15] "ROS Groovy Galapagos." http://ros.org/wiki/groovy. [Accessed: 2013-04-08].

[16] "ROS Fuerte Turtle." http://ros.org/wiki/fuerte. [Accessed: 2013-04-08].

[17] "ROS wiki: humanoid_navigation stack." http://www.ros.org/wiki/humanoid_navigation. [Accessed: 2013-04-09].

[18] "OpenCV." http://opencv.willowgarage.com/. [Accessed: 2013-04-09].

[19] D. Ferguson, M. Likhachev, and A. Stentz, "A guide to heuristic-based path planning," in *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)*, pp. 9–18, 2005.

[20] J. Garimort and A. Hornung, "Humanoid navigation with dynamic footstep plans," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 3982–3987, IEEE, 2011.

[21] A. Hornung, A. Dornbush, M. Likhachev, and M. Bennewitz, "Anytime search-based footstep planning with suboptimality bounds,"

[22] A. Hornung, K. M. Wurm, and M. Bennewitz, "Humanoid robot localization in complex indoor environments," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 1690–1695, IEEE, 2010.

[23] "ROS wiki: nao_robot stack." http://www.ros.org/wiki/nao_robot. [Accessed: 2013-04-08].

[24] "ROS wiki: nao_common stack." http://www.ros.org/wiki/nao_common. [Accessed: 2013-04-08].

[25] "Player." http://playerstage.sourceforge.net/. [Accessed: 2013-04-09].

[26] "ROS wiki: nao_model." http://www.ros.org/wiki/nao_model. [Accessed: 2013-04-17].

[27] "ROS wiki: urdf package." http://www.ros.org/wiki/urdf. [Accessed: 2013-04-02].

[28] "ROS wiki: xacro package." http://www.ros.org/wiki/xacro. [Accessed: 2013-04-02].

[29] "NAO official documentation." http://www.aldebaran-robotics.com/documentation. [Accessed: 2013-04-02].

[30] "ROS wiki: nao_gazebo_plugin." http://www.ros.org/wiki/nao_gazebo_plugin. [Accessed: 2013-04-17].

[31] "ROS wiki: nao_v_rep." http://www.ros.org/wiki/nao_v_rep. [Accessed: 2013-04-17].

[32] E. Tosello, "Motion planning per un robot a molti gradi di libertà," Master's thesis, Università degli Studi di Padova, 2012.

[33] "ROS wiki: rostopic." `http://www.ros.org/wiki/rostopic`. [Accessed: 2013-04-03].

[34] "ROS wiki: ROS Fuerte installation instructions." `http://www.ros.org/wiki/fuerte/Installation`. [Accessed: 2013-04-04].

[35] "ROS wiki: Overlays." `http://www.ros.org/wiki/fuerte/Installation/Overlays`. [Accessed: 2013-04-04].

[36] "V-REP User Manual: ROS tutorial." `http://www.coppeliarobotics.com/helpFiles/en/rosTutorial.htm`. [Accessed: 2013-04-04].

[37] "ROS wiki: rosbag." `http://www.ros.org/wiki/rosbag`. [Accessed: 2013-04-05].