MOTION PLANNING PER UN ROBOT A MOLTI GRADI DI LIBERTA'

Relatore

Enrico Pagello

Laureanda

Elisa Tosello

23 Aprile 2012

ANNO ACCADEMICO 2011/2012

INDICE

INTRODUZIONE

Dato un oggetto mobile, il robot, data la descrizione dell'ambiente che lo circonda, ambiente solitamente caratterizzato da oggetti statici e dinamici che impediscono il libero moto del corpo in analisi, e dati infine due punti di questo spazio, un problema di *Motion Planning* vuole determinare un cammino libero da collisioni che conduca la massa dalla posizione di partenza a quella desiderata.

Per risolvere questo tipo di problema saranno necessari la geometria del robot e degli ostacoli, la kinect del robot e cioè i gradi di libertà (o DoF, *Degrees of Freedom*) che lo caratterizzano, la sua configurazione iniziale e quella obiettivo.

Nel dettaglio, la configurazione di un robot è data dalle posizioni di tutti i punti che lo compongono in un certo istante secondo uno specifico sistema di coordinate.
L'insieme di tutte le possibili configurazioni all'interno dell'ambiente in considerazione viene detto *Configuration Space* (C).
All'interno di questo spazio, allora, il robot può venir ridotto ad un punto ed ogni punto dello spazio rappresenterà una sua possibile configurazione. Ovvio che questo insieme includerà anche le porzioni relative agli ostacoli: il *Collision Space* (C_{obs}).
Il problema di trovare un cammino libero da collisioni si riduce quindi al determinare una sequenza di punti, configurazioni, validi, e cioè che non si trovano all'interno del *Collision Space*, tra loro connessi da lati validi, in altro modo non collocati o non intersecanti il *Collision Space*.
Lo spazio valido viene indicato con C_{free}.

Dato allora un ambiente sul quale si vuole risolvere un problema di *Motion Planning*, si dovrà per prima cosa fissare un approccio per delineare le configurazioni non valide presenti all'interno di C_{obs}.

Si potrebbe effettuare una rappresentazione esplicita di tutti gli ostacoli a partire dalla geometria del problema. Così facendo però si dovrebbero determinare tutti i contatti prima di iniziare l'elaborazione e quindi si raggiungerebbero tempi di esecuzione davvero ingenti.

Per evitare la costruzione precisa di C_{obs} è stato introdotto il *Sampling-based Motion Planning*: non usa alcun modello esplicito ma analizza il C-space effettuando un campionamento random dei punti che lo compongono. Una volta selezionati i campioni si determina se la configurazione è in collisione usando un modulo di collision detection. Quest'ultimo viene solitamente implementato in via separata per poi venir adattato al particolar tipo di problema. In questo modo si possono sviluppare algoritmi indipendenti dai modelli geometrici ed applicabili per la risoluzione di problemi che prevedono la presenza di corpi caratterizzati da molti DoF, come quelli di robotica. Ciò non sarebbe stato possibile usando tecniche che rappresentano esplicitamente il C_{obs}.

Con l'obiettivo di risolvere *Sample Based Motion Planning* per un robot umanoide il lavoro affrontato ha previsto l'analisi di due librerie: *I-Collide* e *OMPL*.

I-Collide è una libreria che tratta solo collision detection; *OMPL*, invece, è più completa e fornisce molti motion planners atti alla risoluzione dell'intero problema.

Notando le notevoli opportunità offerte da quest'ultima essa è stata approfondita in maniera più dettagliata e si è pensato di utilizzarla per risolvere un problema di Motion Planning per un robot umanoide in uno spazio tridimensionale.

Vale la pena notare che quello robotico è un campo in piena evoluzione e per questo, nel momento in cui viene scritto codice a riguardo, è apprezzabile che questo venga condiviso così da poter essere riutilizzato. A tal fine sono nati diversi strumenti che consentono di astrarre la programmazione dal sistema operativo ospite: sistemi di sviluppo per la robotica (RSS, o *Robotic Software System*); e si assiste alla condivisione di un numero sempre maggiore di librerie che trattano robotica.
ROS, o *Robot Operating System*, è un RSS nato proprio con questo scopo; risulta infatti essere un sistema operativo open source che mette a disposizione librerie e tools atti allo sviluppo di applicazioni robotiche.
È importante focalizzarsi su questo sistema perché uno dei vantaggi ricavati dall'utilizzo di OMPL è proprio la sua integrazione all'interno di ROS. OMPL può allora venir considerata come uno strumento di uso comune all'interno della comunità di sviluppatori e, quindi, codice che preveda l'utilizzo degli algoritmi da essa messi a disposizione risulta essere più interessante di codice che implementi la libreria I-Collide. Esso infatti potrà venir riutilizzato da altri soggetti nonché diffuso.

Dopo l'analisi teorica delle due librerie e la conseguente scelta di proseguire il lavoro con OMPL, si è passati alla vera e propria realizzazione di codice che permetta di effettuare del Motion Planning per il robot umanoide Robovie-X all'interno di uno spazio tridimensionale.

Innanzitutto è stato costruito l'ambiente.
Affinché poi il robot possa muoversi senza collidere sono state definite le parti dell'ambiente occupate e quelle libere andando di fatto a costruire un'occupancy grid tridimensionale. Il suo sviluppo è stato reso possibile grazie all'utilizzo di *Octomap*, sempre offerta da ROS, come tutte le altre librerie utilizzate nell'esperienza.

Nota la descrizione dello spazio è stato possibile far compiere un percorso valido al robot: dapprima sono stati scelti punto di partenza e di arrivo, poi è stato scelto il cammino sfruttando uno dei planner offerti da ROS ed infine è stata realizzata la camminata.

Prove sperimentali hanno fatto scegliere, per la risoluzione del problema di Path Planning, la libreria *SBPL*, ottima per la ricerca di un percorso minimo. OMPL, ottima invece per la risoluzione di Motion Planning di corpi con molti gradi di libertà, vede il suo utilizzo nella determinazione del movimento da far compiere allo stesso robot: ai suoi giunti e ai suoi link.

Si ricorda l'importanza dei sistemi di simulazione fisica e grafica in ambienti a due e tre dimensioni: essi riescono a risolvere tutte quelle problematiche che in robotica derivano dal legame tra software e hardware.
Nel caso in analisi, essendo presente un'integrazione di ROS all'interno dell'ambiente di simulazione *Gazebo*, il robot umanoide, i movimenti da esso effettuati e l'ambiente che lo circonda sono stati portati proprio all'interno di tale simulatore, tenendo presenti le dovute forze di gravità e inerzie.

Descrivendo brevemente la struttura del testo: il primo capitolo introduce la libreria I-Collide, il secondo invece quella OMPL.
Concentrandosi su quest'ultima il terzo capitolo espone la sua integrazione in ROS, oltre che una breve descrizione delle funzionalità offerte da questo sistema.
Passando poi all'analisi del lavoro implementato, il capitolo 4 espone le modalità secondo le quali è stata costruita l'occupancy grid tridimensionale; il capitolo 5 va a mappare nell'ambiente di lavoro il robot Robovie-X utilizzato nell'esperienza; il capitolo 6 descrive le metodologie secondo cui è stato risolto il problema di Motion Planning facendo compiere al robot un percorso da una posizione iniziale ad una obiettivo.

Concentrandosi infine sull'implementazione della camminata il capitolo 7 introduce la cinematica del robot e l'8 definisce lo ZMP, o *Zero Motion Point*, punto che deve venir preso in considerazione se si vuole che il robot si muova in equilibrio. Il 9 descrive l'algoritmo che consente la vera e propria realizzazione della marcia bipede. L'ultimo capitolo, il 10, tratta infine la simulazione all'interno di Gazebo.

CAPITOLO 1

I-COLLIDE

I-Collide è una libreria che si occupa del rilevamento di collisioni in ambienti composti da poliedri convessi.
La sua implementazione è stata seguita da sviluppatori ben noti all'interno della comunità robotica: Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, Brian Mirtich, Madhav K. Ponamgi e John F. Canny [1].
Come suddetto I-Collide tratta solo poliedri convessi, ma espone anche le metodologie per decomporre poliedri non convessi, quali corpi articolati, in insiemi di poliedri convessi, che potranno così venir anch'essi analizzati da questa libreria.

In breve:

- per ogni oggetto dello spazio in analisi vengono considerate le sue features;
- vengono costruite delle bounding boxes attorno a tali oggetti;
- vengono considerate tutte le possibili coppie di bounding boxes e viene testata la loro sovrapposizione: se assente i corpi al loro interno non collidono ed il metodo ritorna la più piccola distanza che separa questi corpi, cioè la distanza tra la coppia di feauters più vicine; altrimenti possono collidere. La collisione non è certa poiché le bounding boxes ricoprono una superficie maggiore rispetto a quella degli oggetti che contengono. Viene dunque effettuato un test che ritorna una distanza, in caso di mancata collisione, o un messaggio di avvenuta collisione.

Si nota come il test per la collisione non venga effettuato tra tutte le coppie di oggetti, ma solo tra quelle le cui bounding boxes si sovrappongono. Questo poiché I-Collide utilizza la coerenza, che rende il metodo flessibile e veloce.

1.1 Coerenza temporale e geometrica

I-Collide, nell'analizzare l'evolversi di un ambiente, considera time steps sufficientemente piccoli da far in modo che gli oggetti in analisi possano percorrere distanze, o effettuare movimenti, infinitesimali da un istante temporale a quello successivo.

Ciò si traduce in una coeranza temporale, proprietà secondo la quale lo stato dell'applicazione non cambia significativamente tra due istanti di tempo successivi (o tra due frames).

Se i movimenti effettuati sono minimi, allora vi sarà anche coerenza nella geometria del sistema:

la geometria degli oggetti, definita per mezzo delle coordinate dei loro vertici, subisce solo leggeri cambiamenti nel passaggio da un frame al successivo.

Usando la coerenza si riduce non solo il numero di test effettuati per rilevare le collisioni ma anche il loro tempo di esecuzione. Infatti, dati n oggetti simultaneamente in movimento, anziché considerare le $O(2^n)$ possibili loro interazioni, se ne considerano solo O(n + m) con m numero di oggetti molto vicini tra loro.

Si riconda che due oggetti sono molto vicini tra loro se le loro bounding boxes si sovrappongono.

1.2 Estensione ad oggetti non convessi

Importante è estendere l'algoritmo ai politopi non convessi di modo che esso possa venir utilizzato anche per rilevare le collisioni tra corpi articolati e quindi in ambienti popolati da umanoidi.

L'estensione viene realizzata utilizzando una rappresentazione gerarchica.
Nel dettaglio: si vanno ad utilizzare alberi i cui nodi interni possono essere sottoparti convesse o non convesse mentre tutti i nodi foglia sono politopi convessi. A partire dai nodi foglia si costruiscono delle urne convesse per ogni nodo risalendo l'albero fino ad arrivare alla radice.
Il bounding volume associato ad ogni nodo è pari al bounding volume associato all'unione dei suoi figli. Ciò significa che il bounding volume della radice racchiuderà il volume di tutta la gerarchia.
Ecco che considerando, per esempio, una mano, ad essa verrà associato un albero che avrà come figli le dita e come radice l'intera mano.

L'algoritmo di collision detection testa innanzitutto la collisione tra i due nodi genitore. Se tra essi non vi è collisione viene ritornata la più vicina coppia di features che descrive gli oggetti. Altrimenti l'algoritmo viene espanso ricorsivamente ai figli di modo da determinare dove vi è collisione.

1.3 Costruzione delle bounding boxes

Le bounding boxes vengono costruite attorno agli oggetti come dei cubi di dimensione fissa. Ogni cubo è grande abbastanza da contenere l'oggetto in ogni sua orientazione. I cubi vengono costruiti con i lati paralleli agli assi coordinanti di modo da falicilitare la successiva analisi delle sovrapposizioni.

Si sarebbero potuti costruire anche bounding boxes dinamici, cioè adattabili a seconda dell'orientazione dell'oggetto.
Secondo la nostra analisi ciò avrebbe portato ad una maggiore raffinatezza dei calcoli, ma secondo l'analisi degli autori della libreria le bounding boxes a dimensione variabile avrebbero portato solo ad un aggravamento dei tempi di esecuzione, in particolare all'aumentare:

- dei politopi presenti nell'ambiente;
- della loro complessità, definita come numero di facce;
- della loro velocità di transizione e rotazione;
- della densità, definita come la percentuale di volume dell'ambiente occupato dai politopi.

I grafici di figura 1.1 analizzano le performance; la linea in grassetto riporta l'andamento delle bounding boxes dinamiche, l'altra quello delle bounding boxes fisse.

Si fa notare che, per una buona implementazione dell'algoritmo, le bounding boxes di ogni politopo vengono estese in ogni direzione di un certo γ piccolo. In questo modo:

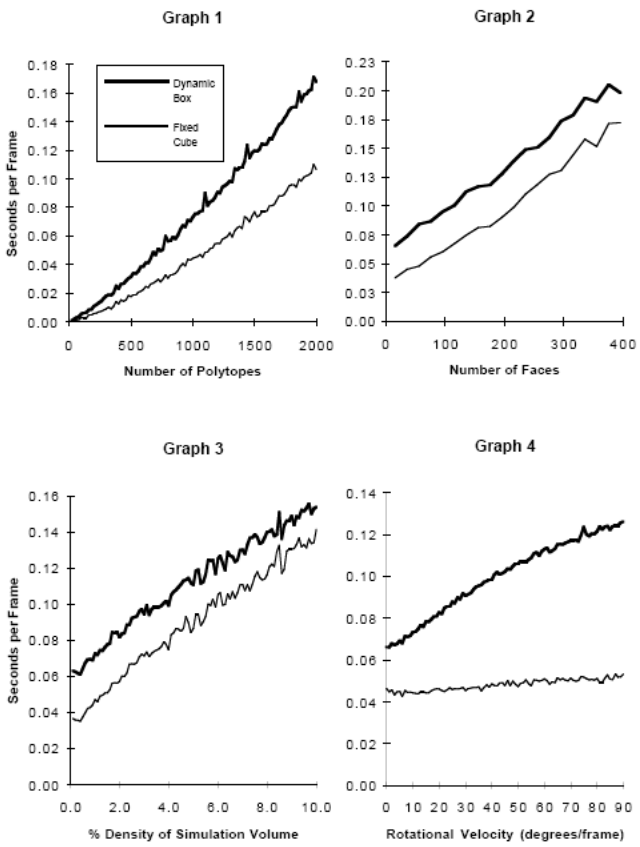

Figura 1.1: Analisi delle performance

- i test fatti per verificare la presenza di collisioni vengono isolati da eventuali errori;
- si può attivare la routine di collision detection prima che gli oggetti si compenetrino.

1.4 Sovrapposizione delle bounding boxes

Una volta create le urne convesse, gli oggetti saranno contornati da dei bounding volumes tridimensionali. Passo successivo sarà quello di determinare le coppie che si sovrappongono. A tal fine i bounding volumes vengono ordinati nello spazio tridimensionale attraverso la definizione dell'algoritmo *Sweep and prune*.

L'algoritmo proietta ogni bounding boxe tridimensionale sugli assi x, y e z. Visto che le urne sono allineate con gli assi, proiettarle su di essi significa creare degli intervalli. Focalizzandosi ora sulle sovrapposizioni, una coppia di bounding boxes si accavallerà se e solo se i relativi intervalli si sovrapporranno in tutte e tre le dimensioni.

Si costruiscono tre liste, una per ogni dimensione. Ogni lista contiene i valori delle estremità degli intervalli corrispondenti a quella dimensione. Ogni lista viene ordinata con l'algoritmo *Insertion Sort* e nel contempo viene mantenuto uno stato di overlap per ogni coppia di bounding boxes. Questo stato si compone di un flag booleano per ogni dimensione. Quando tutti e tre i flag sono *true* la corrispondente coppia di bounding boxes si sovrappone.

I flag vengono settati durante l'esecuzione dell'*Insertion Sort* e quando un flag viene attivato, lo stato di overlap può indicare una delle tre situazioni seguenti:

- tutte e tre le dimensioni della coppia di bounding boxes in analisi si sovrappongono allo stato corrente: si aggiunge la corrispondente coppia di politopi in una lista di coppie attive;

- la coppia si è sovrapposta al passo precedente: si rimuove la corrispondente coppia di politopi dalla lista di coppie attive;

- la coppia non si è sovrapposta al passo precedente e non si sovrappone allo stato corrente: non viene effettuata alcuna operazione.

Quando l'ordinamento relativo allo step temporale corrente è completato, la lista di coppie attive contiene tutte le coppie di politopi le cui bounding boxes si sovrappongono all'istante corrente. Tale lista viene data in input alla routine di collision detection che può procedere al rilevamento delle collisioni.

Da notare che dati n oggetti, il costo dell'ordinamento viene stimato come $O(n \log n)$, ma sfruttando la coerenza il costo computazionale di questo algoritmo diventa lineare: ad ogni passo vengono mantenute le liste ordinate del frame precedente cambiando solo i valori delle estremità degli intervalli. Questo perché gli oggetti effettueranno piccoli movimenti da un frame a quello successivo, e quindi le liste resteranno abbastanza ordinate.

1.5 Collision detection

Questa routine processa la lista attiva per trovare eventuali collisioni tra politopi: per ogni coppia presente in lista studia la relativa coppia di features e ritorna la minima distanza che separa gli oggetti in analisi o il loro stato di collisione.

Per determinare gli stati di esatto contatto tra i politopi convessi viene usato l'algoritmo di *Lin-Canny*.

L'algoritmo mantiene, per ogni coppia di politopi convessi in movimento nello spazio, la coppia di feautures più vicine e la memorizza in una matrice. Tale coppia viene considerata la più vicina ad ogni passo poiché si fa nuovamente uso della coerenza: le closest features correnti sono probabilmente vicine alle closest features precedenti. In questo modo il costo computazionale diventa pressoché costante.

Le coppie di closest features vengono determinate attraverso l'utilizzo delle *regioni di Voronoi*.

L'algoritmo inizia con una coppia candidata di features scelta in maniera casuale, una per ogni politopo, e controlla se i punti più vicini si trovano su queste features.
Visto che i politopi e le loro facce sono convessi, questo è un test locale che coinvolge solo le features vicine delle features candidate correnti.
Se l'algoritmo fallisce nell'analisi della feature corrente si passa alla feature vicina di uno o di entrambi i candidati e si testa nuovamente.

Le *regioni di Voronoi* formano una partizione dello spazio fuori dal politopo. Quando i politopi si interpenetrano alcune features possono non cadere all'interno di queste regioni. Per ovviare il problema si partiziona anche lo spazio interno dei politopi convessi e le regioni si ottengono collegando ogni vertice del politopo con il baricentro dello stesso.

All'arrivo della lista attiva viene analizzata, per ogni coppia di bounding boxes, la corrispondente coppia di features e viene calcolata la distanza euclidea tra tali configurazioni di modo da rilevare la presenza di collisioni: vi sarà collisione ogniqualvolta questa distanza sarà inferiore ad un certo valore soglia ϵ infinitesimale.

1.6 Implementazione

Nell'implementazione dell'algoritmo l'applicazione dapprima carica una libreria di politopi. Qualsiasi sia il formato del file che li descrive (per esempio l'output di un qualche package tridimensionale), questo viene convertito nel formato minimale di I-Collide, secondo il quale i politopi sono descritti dai seguenti parametri:

- il loro numero;
- la loro complessità, definita come numero di facce;
- la loro velocità di rotazione;
- la loro velocità di transizione;
- la densità dell'ambiente, definita come il rapporto tra il volume del politopo ed il volume dell'ambiente.

Dopo aver caricato i politopi, l'applicazione ne sceglie alcune coppie di modo da poter su di esse attivare la routine di collision detection.

In ogni istante l'applicazione informa I-Collide sulle trasformazioni del mondo per ogni politopo che si muove nell'ambiente considerato. E sempre in qualsiasi istante essa può richiamare I-Collide attivando così la routine che testa le collisioni.

Una volta richiamata, I-Collide inizia l'elaborazione e ritorna una lista contenente tutte le coppie in collisione oltre che, per ognuna di esse, la coppia di features in collisione.

L'applicazione va così a rispondere alle collisioni rilevate in modo appropriato.

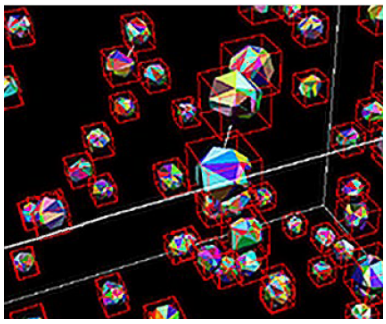

Figura 1.2: Urne

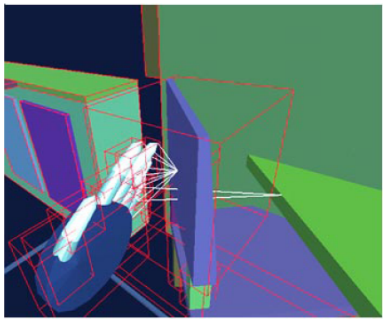

Figura 1.3: Overlap

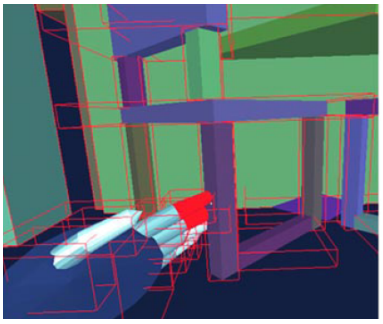

Figura 1.4: Collisione

Dopo una pre-elaborazione nella quale, attraverso l'algoritmo di *Li-Canny*, vengono determinare le closest features di tutte le coppie di politopi, il lavoro della routine di collision detection sarà composto dalle seguenti fasi:

- Creazione delle bounding boxes;
- Ordinamento di tali urne attraverso l'algoritmo *Sweep and Prune* con conseguente determinazione delle sovrapposizioni; per ogni sovrapposizione il metodo inserisce la relativa coppia di politopi in una lista attiva;
- Esame della lista attiva con calcolo delle distanze che separano le coppie di features corrispondenti alle coppie di politopi in lista. Se la distanza è inferiore ad un certo ϵ vi sarà collisione.

Per rendere più chiara la realizzazione del metodo si riporta la grafica dell'elaborazione: la figura 1.2 mostra come vengono costruite le bounding boxes attorno agli oggetti convessi; la figura 1.3, invece,

mette in luce la sovrapposizione tra due urne: si può notare come, in caso di accavallamento, entrano in gioco le coppie di closest features. Infine, la figura 1.4 presenta un caso di collisione tra politopi: i politopi in urto vengono evidenziati di rosso.

CAPITOLO 2

OMPL

OMPL, o *Open Library Motion Planning*, è una libreria, sviluppata in C++, che implementa molti algoritmi probabilistici di motion planning.
È stata sviluppata alla Rice University di Houston dalla prof. Lydia Kavraki con il contributo di Ioa Sucan, Mark Moll e Matt Maly, tre ricercatori della stessa università.
OMPL include la maggior parte degli algoritmi moderni di Motion Plannig: basic probabilistic map, alberi RRT, EST, SBL e KPIECE. Include inoltre algoritmi che consentono di effettuare del Kinodynamic Motion Planning, un modulo di interfacciamento con il simulatore di fisica ODE e un package per l'ambiente ROS.
È possibile utilizzare tale libreria semplicemente includendola nella propria applicazione C++ o utilizzando i Python binding di cui OMPL è dotata.
Importante è sottolineare che il suo utilizzo è concesso sotto licenza BSD, che ne consente la libera divulgazione e sviluppo.

È bene precisare che OMPL non implementa volutamente alcun tool di collision detection e di visualizzazione dei risultati; in questo modo la libreria può essere usata non solo per scopi inerenti la robotica mobile, ma anche per ricerche legate a medicina o industria. Infatti l'utente sarà libero di adattare e personalizzare la propria applicazione.

Per scopi didattici è stato sviluppato OMPL.app, un'applicazione front-end attraverso cui l'utente può interagire e visualizzare i risultati grafici ottenuti da OMPL.
OMPL.app si occupa di motion planning di corpi rigidi, rappresentati mediante forme di grafica vettoriale ed il modulo di collision detection è fornito dalla libreria PQP; per la rappresentazione grafica, invece, si usa un'applicazione scritta in PyQt ed OpenGL.
OpenGL, o *Open Graphics Library*, non è altro che una specifica che definisce una API per più linguaggi e per più piattaforme e che consente di scrivere applicazioni che producono computer grafica 2D e 3D. L'interfaccia consiste in circa 250 diverse chiamate di funzione che si possono usare per disegnare complesse scene tridimensionali a partire da semplici primitive.

2.1 OMPL e OMPL.app

OMPL mette a disposizione:

- classi per la rappresentazione dell'insieme degli stati (manifold), dello stato iniziale e del goal;
- algoritmi per il campionamento di stati di un manifold;

- algoritmi di planning probabilistici;
- strutture dati per la rappresentazione di problemi di tipo kynodimanico;
- classi per l'integrazione con il simulatore fisico ODE;
- un package per l'integrazione con ROS;

Non include invece:

- moduli di collision detector, che devono essere definiti dall'utente;
- tool di rappresentazione grafica dei risultati;

Come suddetto, a fini didattici è stata sviluppata OMPL.app, un'applicazione che utilizza parte delle funzioni di OMPL per poter risolvere problemi di motion planning di corpi rigidi. Essa include:

- un tool grafico in grado di caricare e visualizzare ambienti grafici e robot costituiti da forme di grafica vettoriale (modelli Collada con estensione .dae);
- un collision detector che si basa sulla geometria dell'ambiente e del robot;
- l'implementazione del manifold SE2 (roto-traslazioni nel piano) e SE3 (roto-traslazioni nello spazio);
- una GUI che facilita la configurazione di tutti i parametri del problema (limiti del manifold, stato iniziale, goal, algoritmi di planning e la sua parametrizzazione);
- l'interfacciamento tramite OMPL, attraverso cui viene risolto il problema di motion planning;

Anche OMPL.app ha dei limiti; essa infatti non consente di risolvere problemi di motion planning per robot caratterizzati da:

- uno o piu giunti;
- manifold diversi da SE2 e SE3 (adatti, ad esempio, per la modellazione di un braccio manipolatore);

In questo capitolo verranno trattate le principali funzioni che compongono OMPL; per una documentazione più completa ed esaustiva, si consiglia di visitare il sito [6].

2.2 Tipi di motion planning

OMPL consente di risolvere due categorie di problemi di pianificazione del moto: geometrici e kinodinamici.

Il planning geometrico include i soli vincoli di natura geometrica, quelli che assicurano l'assenza di collisioni durante il moto che il robot effettua nel percorrere la distanza che separa lo stato iniziale al goal; non viene fatta nessuna assunzione che vincoli il robot nei suoi movimenti.

Il planning kinodinamico, in letteratura meglio noto come *planning under differential constraints*, considera invece anche i vincoli di natura fisica che regolano il moto del robot. In questo tipo di pianificazione devono venir soddisfatti i limiti di velocità ed accelerazione caratterizzanti il robot e le sue componenti.
Ciò significa che, ad ogni passo, è necessario applicare la legge di controllo allo stato del robot per determinare il successivo stato che il robot può raggiungere.
A seconda del grado di precisione richiesto dall'applicazione, il comportamento del robot può essere determinato utilizzando metodi d'integrazione numerica del motion model, come l'integrazione nel

tempo della legge di controllo applicata, o avvalendosi di simulatori come ODE.
Quest'ultimo risulta essere uno dei simulatori più performanti di dinamica dei corpi rigidi: oltre ad essere open source è ricco di funzioni, stabile, indipendente dalle piattaforme e con una API in C/C++ intuitiva. Ha inoltre integrato il rilevamento delle collisioni con l'attrito e per questo viene comunemente usato nella simulazione di veicoli o oggetti in ambienti virtuali.

2.3 Planners disponibili per OMPL

OMPL mette a disposizione la maggior parte degli algoritmi di tipo probabilistico (*sampling based*). Essi vengono qui di seguito riportati ed analizzati.

PRM

L'algortimo PRM, o *Probabilistic Roadmap*, determina la soluzione attraverso una mappa di stati (roadmap) che approssima la connettività tra lo stato iniziale ed il goal all'interno dello spazio valido Q_{free}. Si compone di due fasi:

- learning: viene determinata la roadmap che connette lo stato iniziale e il goal (grafo non orientato);
- ricerca del percorso valido.

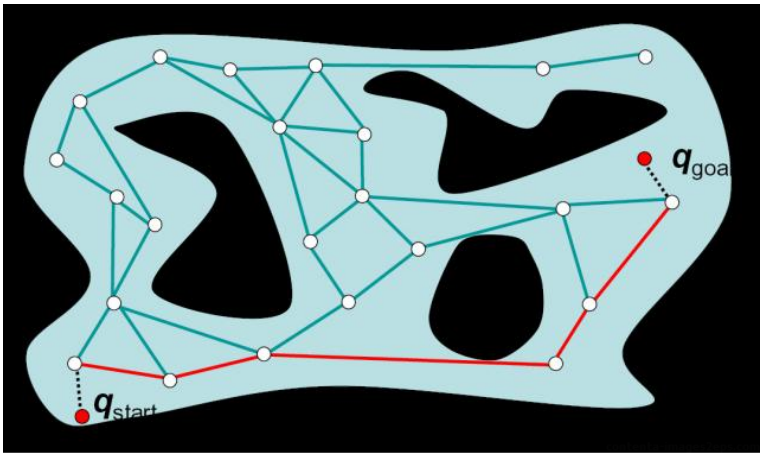

Figura 2.1: Sviluppo di una Probabilistic Roadmap

Si procede alla descrizione del relativo pseudocodice, di seguito riportato.

La fase di learning viene computata all'interno del ciclo *while*: ad ogni iterazione viene campionato un nuovo stato q dello spazio degli stati secondo una distribuzione di probabilità P. Se q appartiene all'insieme degli stati validi Q_{free}, stati liberi da collisioni nei quali è permesso il moto del robot, allora viene aggiunto ai nodi N della roadmap corrente.

Sfruttando gli ora $N \cup q$ nodi della mappa, si procede con la costruzione di un insieme di n nodi vicini di q, si uniscono poi questi nodi a q con degli archi andando così ad aggiungere nuovi lati al grafo. Anche il percorso individuato da questi ultimi lati deve essere valido.

All'uscita del ciclo *while* si procede con la seconda fase, quella di ricerca: si determina il percorso di costo minimo tra gli stati *start* e *goal*, che saranno ora due stati appartenenti ad un grafo connesso, attraverso l'algoritmo di Dijkstra.

Algorithm 1: PRM

begin
- Inizializzazione della roadmap: grafo non orientato R = (N, E) con N = { start, goal} ed E = ∅
- **while** *(start e goal non appartengono alla stessa componente connessa di R) and (time < TIME_MAX)* **do**
  - campiona un nuovo stato q secondo una distribuzione di probabilitàP
  - **if** $q \in Q_{free}$ **then** $N = N \cup \{q\}$
  - costruisci un insieme $V \subset N$ di n nodi vicini di q
  - $\forall v \in V$
  - **if** *(q, v) è valido* **then** $E \leftarrow EU\{(q, v)\}$
- **if** $start$ e $goal$ *appartengono alla stessa componente connessa di R* **then**
  - determina il percorso di costo minimo tra $start$ e $goal$
- **else**
  - ``Nessuna soluzione trovata!´´

RRT

L'RRT, o *Rapidly exploring Random Tree*, ricava la soluzione costruendo un albero che connette lo stato iniziale al goal.

Di seguito viene presentato lo pseudocodice d'interesse: dapprima si campiona, secondo una distribuzione di probabilità P, uno stato q_{rand} all'interno dello spazio degli stati. Poi questo stato viene dato in input alla funzione *EXTEND* che provvede alla costruzione dell'albero RRT.

Algorithm 2: RRT

begin
- Inizializza un albero T = (N, E) con N = {start} ed $E = \emptyset$
- **while** *time < TIME_MAX* **do**
  - campiona un nuovo stato q_{rand} secondo una distribuzione di probabilità P
  - EXTEND(q_{rand}, T)

EXTEND ricerca il nodo dell'albero q_{near} più vicino a q_{rand}.

Questi due nodi devono essere vicini tra loro, in particolare la distanza che li separa non deve superare un certo ϵ piccolissimo.

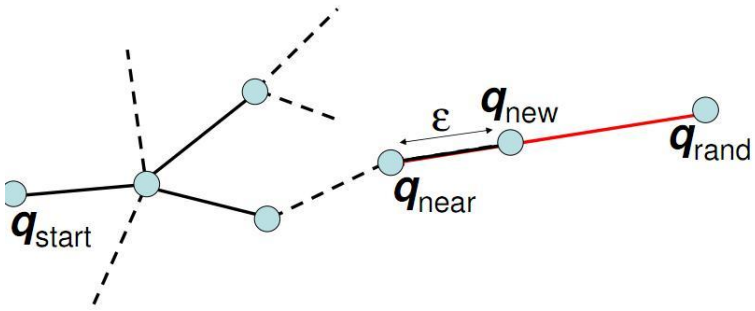

Figura 2.2: Sviluppo di un RRT

Algorithm 3: EXTEND(q_{rand}, T)

begin
- trova il nodo $q_{near} \in N$ più vicino a q_{rand}
- **if** $distance(q_{near}, q_{rand}) > \epsilon$ **then**
  - trova un nuovo nodo q_{new} in direzione di q_{rand} t.c. $distance(q_{near}, q_{rand}) \leq \epsilon$
- **else**
  - $q_{new} \longleftarrow q_{rand}$
- **if** (q_{near}, q_{new}) *è valido* **then**
  - $N \longleftarrow N \cup \{q_{new}\}$
  - $E \longleftarrow E \cup \{(q_{near}, q_{new})\}$
  - **if** $q_{new} = q_{rand}$ **then**
    - return REACH
  - **else**
    - return ADVANCED
- **else**
  - return TRAPPED

In caso negativo viene ricercato un altro nodo q_{new} che si interponga tra q_{near} e q_{rand} (vedi figura 2.2). Altrimenti sarà q_{rand} lo stesso q_{new}.
Infine si costruisce un arco che collega q_{near} a q_{new}.
Si deve garantire la validità del cammino tra i due nodi; infatti solo in caso di validità l'albero RRT viene espanso con il nuovo nodo ed il nuovo arco.

Il procedimento continua iterativamente fino a quando il nuovo stato q_{new} coincide con lo stato *goal*.

RRT Connect

L'algoritmo costruisce due alberi RRT, uno a partire dallo stato iniziale e l'altro a partire dal goal. I due alberi vengono connessi per poi determinare una soluzione.

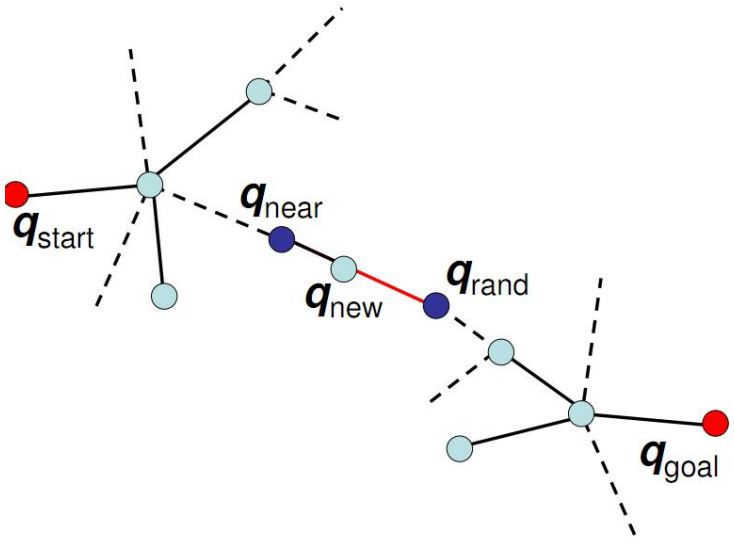

Figura 2.3: Sviluppo contemporaneo di due alberi

Algorithm 4: RRT Connect

```
begin
    Inizializza un albero T_1 = (N_1, E_1) con N_1 = {start} ed E_1 = ∅
    Inizializza un albero T_2 = (N_2, E_2) con N_2 = {goal} ed E_2 = ∅
    while time < TIME_MAX do
        campiona un nuovo stato q_rand secondo una distribuzione di probabilità P
        if EXTEND(q_rand, T_1) ≠ TRAPPED then
            if CONNECT(q_new, T_2) = REACHED then
                calcola soluzione
        inverti(T_1, T_2)
```

Algorithm 5: CONNECT(q_{new}, T_2)

```
begin
    repeat
        S ⟵ EXTEND(q_new, T_2)
    until S = ADVANCED
    return S
```

Lo peseudocodice presentato si compone di un ciclo *while* all'interno del quale viene espanso uno dei due alberi tramite la funzione *EXTEND*; i due alberi vengono invertiti alla fine di ogni iterazione attraverso inverti(T_1, T_2) di modo da garantire una corretta alternanza delle espansioni.

Ad ogni espansione di un albero: viene campionato un nuovo stato q_{rand}, secondo una distribuzione di probabilità P, e dato in ingresso alla funzione *EXTEND*. Essa avvia l'elaborazione; se l'output prodotto è diverso da *TRAPPED* allora un nuovo nodo è stato aggiunto all'albero corrente e si procede con il tentativo di connettere i due alberi. A tal scopo viene richiamata la funzione *CONNECT*, che espande l'altro albero in direzione del nuovo stato aggiunto all'albero in analisi, come illustrato in figura 2.3.

L'espansione termina quando viene raggiuto q_{rand} o quando viene raggiunto un lato non valido.

Questa strategia limita il calcolo del nodo q_{near} ricavando lunghi percorsi rettilinei.

Parallel RRT

L'unica peculiarità che caratterizza questo algoritmo è l'avvio contemporaneo di più thread per la costruzione dell'albero RRT. Non vi è niente che differisce la costruzione delle strutture dati da quanto presentato nelle sezioni precedenti. L'accesso agli RRT, infatti, avviene in mutua esclusione.

Lazy RRT

L'algoritmo costruisce dapprima un albero senza eseguire alcun controllo sulla validità degli stati; l'obiettivo è quello di produrre in tempo breve una possibile soluzione. Solo dopo aver trovato un percorso vengono ispezionati i nodi e gli archi che lo compongono; se si trova che un lato non è valido: esso viene rimosso, l'albero diviso in due parti e si espande nuovamente l'albero aggiungendo nuovi lati.

L'elaborazione termina al ritrovamento di un percorso valido o all'esaurimento del tempo riservato alla computazione.

KPIECE

KPIECE, o *Kynodinamic motion Planning by Interior Exterior Cell Exploration*, costruisce un albero che connette stato iniziale e finale privilegiando le zone meno esplorate dello spazio degli stati.

Pseudocodice:

Algorithm 6: KPIECE

```
begin
    crea una griglia G = {Cell(start)} composta da una sola cella
    crea una lista Cell(start).list = {start} contenente gli stati visitati dalla Cell(start)
    while time < TIME_MAX do
        cell ⟵ seleziona una cella EXTERIOR o INTERIOR
        q_old ⟵ seleziona una stato di cell già visitato
        campiona un nuovo stato q_rand secondo una distribuzione di probabilità P t.c.
        distance(q_old, q_rand) ≤ ε
        if (q_old, q_rand) è valido then
            aggiungi Cell(q_rand) alla griglia G
            aggiungi q_rand alla lista Cell(start).list
            salva il percorso tra q_old e q_rand nell'albero della soluzione
            if (q_rand = q_old) then
                return il percorso start e goal
```

KPIECE, infatti, discretizza l'intero spazio degli stati per mezzo di una griglia di celle aventi tutte la stessa dimensione. Inizialmente la griglia è popolata da un'unica cella collocata in corrispondenza dello stato iniziale e poi, di volta in volta, vengono allocate le celle necessarie.

Ogni cella contiene una lista di stati, inizialmente vuota, nella quale vengono inseriti tutti gli stati visitati appartenenti alla cella. In questo modo è possibile individuare le aree altamente e scarsamente esplorate.

In particolare le celle della griglia possono essere divise in due tipologie: INTERIOR ed EXTERIOR. Le INTERIOR, data n la dimensione dello spazio di discretizzazione, sono le celle che hanno $2n$ celle vicine; esse individuano aree in cui l'esplorazione è già avvenuta. Le rimanenti celle invece sono quelle EXTERIOR.

L'algoritmo predilige l'espansione di aree meno esplorate quindi, all'interno del ciclo *while*, al momento della scelta della cella da espandere preferirà avviare l'espansione a partire dalle celle EXTERIOR.

Una volta determinata la tipologia di cella da espandere, l'effettiva cella i viene selezionata secondo:

$$Importance_i = \frac{\log_{10}(I) * score}{S * N * C}$$

con:

- I: numero dell'iterazione in cui la cella è stata creata;
- $score$: valore che stima la distanza dal goal;
- S: numero di volte in cui la cella è stata selezionata per l'espansione;
- N: numero di vicini della cella;

- C: indice di copertura della cella, cioè la somma dei suoi stati visitati.

Data la cella, da essa si estrae uno stato q_{old} già visitato, e quindi memorizzato all'interno della relativa lista, secondo una *half normal distribution*.

Da q_{old} si inizia ad espandere l'albero: con probabilità P si ricava un nuovo stato q_{rand} posto a distanza minore di ϵ da q_{old}. Si traccia il lato che li congiunge e lo si valuta; se valido:

- si aggiunge, se necessario, la cella contenente q_{rand} alla griglia;
- si inserisce q_{rand} alla nuova cella appena istanziata;
- si aggiorna l'albero di ricerca aggiungendo il nuovo lato.

Altrimenti:

- si estrae l'ultimo stato q_{new} valido appartenente al lato;
- si controlla che il suo valore sia maggiore di una certa soglia;
- in caso affermativo si imposta $q_{rand} = q_{new}$.

Si può dunque notare come l'albero venga espanso ad ogni passo senza perdere il lavoro svolto (vedi figura 2.4).

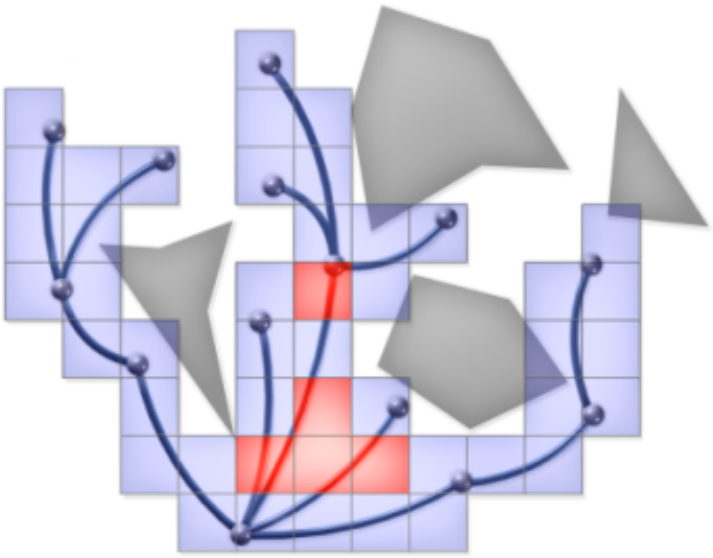

Figura 2.4: Sviluppo di KPIECE

Una volta aggiunto il lato si controlla se lo stato goal è stato raggiunto, in caso negativo si itera il procedimento.

Si può notare come la componente probabilistica incida molto sul corretto funzionamento dell'algoritmo. Essa risulta essere presente in:

- selezione di una tipologia di cella: viene usata una distribuzione discreta che privilegia le celle EXTERIOR;
- selezione di uno stato appartenente a una cella: viene usata la distribuzione *half normal gaussian*;

- selezione di un nuovo stato q_{rand}: viene usata una distribuzione di probabilità P

Si ricorda inoltre che, senza perdita di generalità, OMPL prevede la possibilità di definire, per spazi caratterizzati da un alto numero di dimensioni, proiezioni in altri spazi degli stati di minore dimensione dove poter poi analizzare la griglia (vedi figura 2.5).

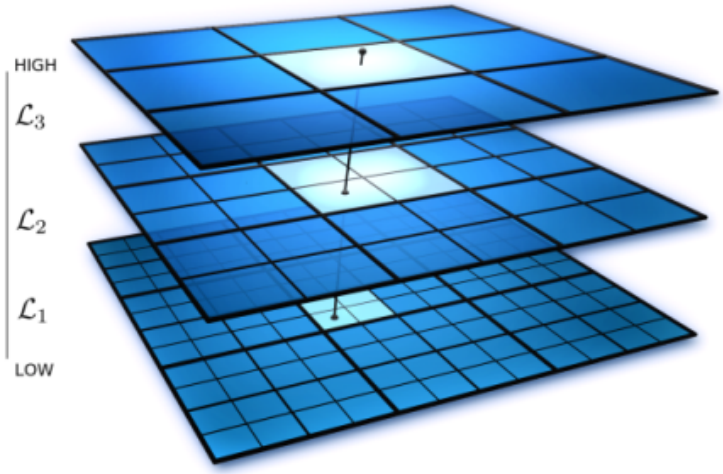

Figura 2.5: Proiezione di KPIECE

BKPIECE

BKPIECE, o *Bi-directional KPIECE*, costruisce due alberi, uno a partire dallo stato iniziale e uno a partire dal goal con relativa costruzione di due griglie, una per albero. I due alberi vengono estesi alternativamente.

Quando viene inserito un nuovo stato q_{rand1} all'interno della lista degli stati visitati da una cella della prima griglia si controlla se nelle stesse coordinate della seconda griglia è presente una cella con almeno uno stato visitato. In caso affermativo si estrae da quest'ultima uno stato q_{old2} e si verifica se il lato (q_{rand1}, q_{old2}) è valido. In caso affermativo i due alberi sono connessi ed è possibile determinare una soluzione.

LBKPIECE

LBKPIECE, o *Lazy bi-directional KPIECE*, costruisce due alberi senza verificare che i nuovi lati aggiunti siano validi. Solo dopo aver connesso i due alberi viene testata la soluzione trovata: si percorre il cammino dallo stato iniziale al goal e si eliminano dalla lista degli stati visitati di una cella le componenti che risultano essere non valide.

Il procedimento si ripete espandendo nuovamente gli alberi fino alla loro connessione.

EST

EST, o *Expansive space tree*, costruisce un albero dallo stato iniziale al goal privilegiando l'espansione delle aree meno esplorate. Anche qui, come in KPIECE, lo spazio viene discretizzato per mezzo di una griglia di celle così da tener traccia delle zone ad alta densità di campioni.

In EST, data una cella X, essa viene selezionata con probabilità $P(X) = 1/|x.list|$; visto che tale probabilità è inversamente proporzionale alla cardinalità della lista di nodi componenti una cella, verranno privilegiate le celle meno esplorate.

Algorithm 7: EST

```
begin
    crea una griglia G = {Cell(start)} composta da una sola cella
    crea una lista Cell(start).list = {start} contenente gli stati visitati dalla Cell(start)
    while time < TIME_MAX do
        cell ⟵ seleziona una cella della griglia G
        q_old ⟵ seleziona una stato di cell già visitato
        campiona un nuovo stato q_rand secondo una distribuzione di probabilità P t.c.
        distance(q_old, q_rand) ≤ ε
        if (q_old, q_rand) è valido then
            aggiungi Cell(q_rand) alla griglia G
            aggiungi q_rand alla lista Cell(start).list
            salva il percorso tra q_old e q_rand nell'albero della soluzione
            if (q_rand = q_old) then
                return il percorso start e goal
```

Una volta selezionata la cella si estraggono q_{old}, dalla cella stessa, e q_{rand}, entrambi secondo una distribuzione di probabilità P e, come negli algoritmi esposti in precedenza, sempre sotto la condizione che q_{rand} si trovi ad una distanza minore di ϵ da q_{old}.

Si costruisce il lato (q_{old}, q_{rand}), lo si analizza e in caso di validità si aggiorna la griglia.

Si studia se è stata trovata una soluzione ed in caso negativo si itera nuovamente il procedimento.

SBL

SBL, o *Single query Bi-Directional Probabilistic Roadmap with Lazy collision checking*, è un'estensione di EST. Anche questo algoritmo costruisce due alberi, uno dallo stato iniziale e uno dallo stato finale e, alternativamente, li estende. Ma ogniqualvolta viene aggiunto un nuovo lato all'albero non se ne controlla la validità (strategia lazy).

Ad ogni passo dell'iterazione si verifica se è possibile connettere i due alberi e quando questi sono connessi si determina una possibile soluzione. Solo dopo viene scandito il cammino proposto per valutarne la validità; se si rileva che un lato non è valido, lo si elimina dalla lista degli stati della cella che lo contiene e si effettua una nuova espansione (come LBKPIECE). L'intero procedimento si ripete finché non viene determinata una soluzione.

2.4 OMPL e I-Collide a confronto

Quanto descritto in questo capitolo fa notare le elevate potenzialità caratterizzanti la libreria OMPL, soprattutto se messa a confronto con I-Collide.

La grande varietà di algoritmi proposti ed il fatto che essi siano ben utilizzabili per la risoluzione di Motion Planning per robot a molti gradi di libertà, fa capire come i risultati ottenuti con OMPL risultino essere più precisi rispetto a quelli che si ottengono con I-Collide.
OMPL, infatti, consente di implementare il manifold SE2 (roto-traslazioni nel piano) e SE3 (roto-traslazioni nello spazio), possibilità non offerta da I-Collide, ancora ferma a bounding box statiche e di dimensione fissa.

Inoltre, OMPL viene usata per pianificare il movimento dei robot anche all'interno di ROS. In par-

ticolare, questa libreria è stata intensamente sfruttata per la navigazione del PR2, robot offerto dalla Willow Garage e su cui si sta concentrando la maggior parte della sperimentazione robotica ROS.

Scelto quindi di continuare l'esperienza con OMPL, nel capitolo seguente viene trattata con maggiore accortezza l'integrazione di OMPL in ROS.

CAPITOLO 3

OMPL E ROS

L'ultimo release di OMPL prevede l'integrazione con l'interfaccia ROS.

Figura 3.1: PR2

ROS, o *Robot Operating System*, è un sistema operativo open source, diffuso con licenza BSD, che mette a disposizione librerie e tools che facilitano gli sviluppatori di software a creare applicazioni robotiche. Fornisce astrazione dall'hardware, device drivers, librerie, visualizzatori, messaggistica, gestione dei pacchetti [7].
Si può quindi notare come l'integrazione di OMPL, libreria di pianificazione del moto, con ROS, sistema software che offre l'implementazione di molte funzionalità necessarie al robot come percezione, controllo, prese, permetta di creare oltre che una pianificazione completa dei movimenti anche una pipeline di esecuzione.
Con l'integrazione vengono dunque offerti algoritmi di motion planning molto accurati attraverso OMPL seguiti da una possibilità di facile implementazione su robot reali attraverso ROS.
L'integrazione è presente nella versione *Electric* di ROS, quella più recente, ed è costituita dal pacchetto *ompl_ros_interface*, che permette appunto di configurare motion planners per robot. Anche questo pacchetto è stato rilasciato e può essere liberamente utilizzato.

Figura 3.2: Robovie-X

È importante notare che con l'obiettivo di insegnare la realizzazione del motion planning su robot reali e quindi di esporre le funzionalità del pacchetto appena citato, all'International Conference on Intelligent Robots and Systems (IROS) del 2011 è stato presentato un tutorial. Il suo sviluppo è stato seguito da S. Chitta e E. Gil Jones della WillowGarage Inc. con il supporto di J. Sucan, M. Moll e L.E. Kavraki della Rice University [8].

Questo tutorial è meritevole di citazione poiché permette di rapportarsi con le componenti sensoriali e cinematiche necessarie per la pianificazione del moto di un robot con molti gradi di libertà nonché di concentrarsi sulla ripianificazione dei task.
Nel dettaglio il pacchetto è stato implementato sia su di un simulatore sia su di un robot fisico, il PR2 della Willow Garage, robot dotato di due bracci mobili ed una base con ruote (vedi figura 3.1).

Le competenze apprese da tale tutorial sono state fonte d'ispirazione per il lavoro svolto: motion planning del robot umanoide Robovie-X (vedi figura 3.2).

3.1 Installazione

ROS

È necessario installare la versione *ROS Electric* ed in particolare la configurazione *Desktop-Full*:

```
sudo apt-get install ros-electric-desktop-full
```

Questa configurazione installa la versione completa di ROS: non solo il sistema ma anche le librerie dei robot, i simulatori 2D e 3D, la navigazione, la percezione 2D e 3D ed Rviz.
Si ricorda che *Rviz* è l'ambiente usato da ROS per la visualizzazione 3D dei robot e dell'ambiente.

Conviene far sì che le variabili dell'ambiente ROS vengano automaticamente aggiunte alla propria sezione bash ogniqualvolta viene lanciata una nuova shell:

```
echo "source /opt/ros/electric/setup.bash" >> ~/.bashrc
. ~/.bashrc
```

OMPL

La versione Electric di ROS installa automaticamente OMPL ed il pacchetto *ompl_ros_interface* che fornisce l'interfaccia ROS necessaria alla libreria stessa.

3.2 Creazione di un workspace ROS

Viene creata la cartella *ros_workspace* all'interno della directory *home*, cartella che conterrà tutto il lavoro:

```
mkdir ~/ros_workspace
```

Poi viene creato uno script bash per configurare il proprio ambiente di lavoro ROS. Si crea dunque un file di setup *setup.sh*:

```
#!/bin/sh
 source /opt/ros/electric/setup.bash
 export ROS_ROOT=/opt/ros/electric/ros
 export PATH=$ROS_ROOT/bin:$PATH
 export PYTHONPATH=$ROS_ROOT/core/roslib/src:$PYTHONPATH
 export ROS_PACKAGE_PATH=~/ros_workspace:/opt/ros/electric/stacks:$ROS_PACKAGE_PATH
```

Questo file aggiunge *ros_workspace* al ROS_PACKAGE_PATH. Si digita:

```
. setup.sh
```

E per confermare che è stato impostato il percorso del package:

```
echo $ROS_PACKAGE_PATH
```

Per far sì che il cambiamento apportato sia permanente, assumendo che il file *setup.sh* sia nella directory *home*, si aggiunge alla fine del file *.bashrc*:

```
source  ~/setup.sh
```

Agendo in tal modo l'ambiente di lavoro è ora impostato.

3.3 Creazione di un package ROS con relative dipendenze

Durante il lavoro verranno creati dei pacchetti contenenti le funzioni necessarie all'elaborazione dei task previsti.
Ci si sofferma allora sul comando *roscreate_pkg* che permette di creare un nuovo package ROS.
roscreate_pkg elimina molte delle fasi che vengono eseguite quando si costruisce a mano un nuovo package e riduce la probabilità d'errore che nasce nel momento in cui si digitano a mano i build files e i manifesti.

Per creare un nuovo package nella directory corrente:

```
roscreate-pkg [package_name]
```

Si possono anche specificare le dipendenze di quel pacchetto:

```
roscreate-pkg [package_name] [depend1] [depend2] [depend3]
```

Se per esempio dovessimo ideare il package *planning* dipendente da *ompl_ros_interface* e *roscpp* all'interno di *ros_workspace*, ci si posiziona nella cartella e si crea il package:

```
$ cd ~/ros_workspace
$ roscreate-pkg planning ompl_ros_interface roscpp
```

Tutti i pacchetti ROS sono composti dai file:

- manifest.xml;
- CMakeLists.txt;
- mainpage.dox;
- Makefiles.

Esaminiamo nel dettaglio alcuni di questi file:

- manifest.xml: definisce com'è stato creato il package, come funziona e com'è documentato. Il manifesto è una specifica del pacchetto e supporta una vasta gamma di strumenti ROS, dalla compilazione alla documentazione e distribuzione. Oltre a fornire una minima specifica sui metadati del pacchetto, dichiara le dipendenze in maniera indipendente dal linguaggio e dal sistema operativo. La presenza di un manifesto all'interno di una cartella è significativa: qualsiasi directory all'interno del package ROS contenente un *manifest.xml* viene considerata pari ad un pacchetto (si nota che un package non può contenere altri package).

  Il manifesto è simile ad un file readme: indica da chi è stato scritto il pacchetto e la licenza. La licenza è importante perché i pacchetti sono i mezzi attraverso i quali viene distribuito ROS (solitamente tutti i pacchetti vengono rilasciati sotto licenza BSD).

  Vengono poi inclusi tag come <depend> ed <export> che aiutano a gestire l'installazione e l'uso del pacchetto. <depend> punta ad un altro package ROS che deve essere installato; <export> descrive language-specific build e runtime flags che devono essere usate da ogni pacchetto che dipende da questo pacchetto. Per esempio, per un package contenente codice roscpp, un tag di tipo <export> può dichiarare i file header e le librerie che devono venir prese in considerazione da tutti i pacchetti che dipendono da questo pacchetto;

- CMakelists.txt: dice a CMake come costruire il package.

Dopo avere creato il package bisogna avere la sicurezza che ROS riesca a trovarlo. È utile richiamare la routine *rospack profile* dopo aver fatto cambiamenti al proprio path di modo che le nuove directory possano venir trovate:

```
$ rospack profile
$ rospack find planning
```

che restituirà:

```
YOUR_PACKAGE_PATH/planning
```

Se questo fallisce significa che ROS non è in grado di trovare il nuovo package e quindi che il proprio ROS_PACKAGE_PATH è afflitto da errori.

Ci si muove alla directory del proprio package con:

```
$ roscd planning
$ pwd
```

che restituisce:

```
YOUR_PACKAGE_PATH/planning
```

Le dipendenze di primo ordine possono essere rivisionate con il tool rospack:

```
rospack depends1 planning
```

che nel nostro caso restituirà:

```
ompl_ros_interface roscpp
```

rospack lista le stesse dipendenze che sono state usate come argomento in roscreate_pkg. Queste dipendenze sono memorizzate nel file manifest. Per visionare il file manifest:

```
$ roscd planning
$ cat manifest.xml
```

che nel nostro caso restituirà:

```
<package>

...

  <depend package="ompl_ros_interface"/>
  <depend package="roscpp"/>

</package>
```

3.4 Rviz

Rviz è l'ambiente dedito alla visualizzazione tridimensionale messo a disposizione da ROS per rappresentare i robot ed il loro moto. Anch'esso, come l'intera libreria a cui appartiene, è stato rilasciato sotto licenza BSD.

Installazione

Dapprima si soddisfano tutte le dipendenze del sistema

```
rosdep install rviz
```

poi si costruisce il visualizzatore

```
rosmake rviz
```

ed infine si avvia il visualizzatore

```
rosrun rviz rviz
```

La prima volta che viene lanciato Rviz, si apre una finestra a sfondo nero rappresentante la vista tridimensionale (vuota perché non vi è ancora alcun oggetto da rendere visibile). Descrivendo la finestra si può notare sulla sinistra una lista dei display attivi, e quindi caricati all'istante corrente. Tale lista al primo lancio mostrerà solo le opzioni globali. Altri pannelli, descritti in seguito, compaiono invece alla destra. Tra essi si ricorda *Views* (vedi figura 3.3).

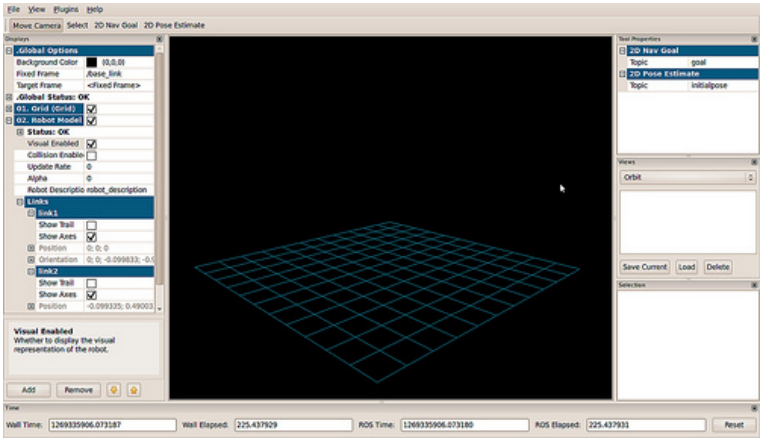

Figura 3.3: Rviz

Display

Un display è un *qualcosa che disegna qualcosa* nel mondo tridimensionale ed è dotato di opzioni liberamente fissabili all'interno della lista di parametri che lo caratterizza. Esempi sono il display che mette in luce lo stato del robot o quello che mostra la griglia rappresentante il suolo.

Per aggiungere un nuovo display basta premere il pulsante *Add* in basso (si faccia riferimento sempre alla figura 3.3), esso apre una finestra di dialogo che elenca tutti i tipi di display disponibili. A seconda del tipo selezionato vengono visualizzati dati diversi. Per comunicare il corretto caricamento o meno dei dati che caratterizzato il display selezionato, esso è dotato di uno stato (*OK, Warning, Error e Disabled*), visibile sia attraverso una stringa sia un colore.

Si elencano alcuni dei display messi a disposizione (di seguito descritti nel dettaglio):

- Grid;
- Map;
- Markers;
- Odometry;
- RobotModel;
- TF.

Più display caricati contemporaneamente vanno a formare una configurazione.
Visto che spesso si ha la necessità di avere più configurazioni di modo da poter utilizzare in più modi il visualizzatore, Rviz permette di caricare e salvare configurazioni diverse.

Views

Prima di andare ad esporre alcuni dei display ci si sofferma sul sopracitato pannello *Views*, alla destra della finestra di visualizzazione. Esso permette di selezionare la telecamera con la quale osservare gli oggetti esposti. Le diverse opzioni fornite permettono di scegliere il tipo di proiezione (ortografico o prospettivo):

- Fotocamera orbitale (di default): semplicemente permette di ruotare attorno ad un punto focale, solitamente situato al centro della scena.
  Controlli attraverso il mouse:
  - Tasto sinistro: tenendolo premuto e trascinando il puntantore si può ruotare attorno al punto;
  - Scroll: permette di zoomare.
- Telecamera FPS (first-person): agisce allo stesso modo in cui un umano ruota la testa per cercare qualcosa.
- Telecamera ortografica Top-Down: si lavora lungo l'asse Z (nel frame del robot) e facendo uso di una vista ortografica, che significa che le cose non diventano più piccole man mano che ci si allontana dal centro.

Frame delle coordinate

Rviz usa il sistema di trasformate TF per trasformare i dati dal sistema di coordinate di partenza al sistama di riferimento globale (ci si soffermerà su TF nel capitolo 5). Il visualizzatore offre due sistemi di coordinate importanti:

- Fixed Frame: è il sistema di riferimento più importante. Viene usato per denotare il frame del mondo e per questo fa solitamente riferimento alla mappa (*map*) o appunto al mondo (*world*), ma può anche ricollegarsi, per esempio, al frame dell'odometria.
  Se il Fixed Frame viene erroneamente settato a, per esempio, la base del robot, allora tutti gli oggetti che il corpo vede o vedrà gli appariranno di fronte, nella posizione, rispetto alla sua, in cui sono stati rilevati. Per risultati corretti, il sistema fisso non dovrebbe muoversi rispetto al mondo.
  Da notare che ogniqualvolta si modifica il sistema di riferimento fisso, tutti i dati attualmente riportati vengono cancellati e poi nuovamente trasformati.
- Target Frame: è il frame che si riferisce alla visuale della telecamera. Per esempio, se il Target Frame è la mappa, si vedrà il robot muoversi all'interno della mappa. Se il target frame è la base del robot, il robot rimarrà nello stesso punto mentre tutto il resto si muoverà rispetto le sue coordinate.

2D Nav Goal

Questo strumento consente di impostare una posizione obiettivo, che viene inviata al topic *goal* di ROS.
Per fissarla basta premere l'omonimo tasto situato nella parte alta della finestra di Rviz e poi cliccare su di una posizione del piano e, tenendo premuto, trascinare il puntatore per selezionare l'orientamento.
Il robot dovrà compiere un moto fino a raggiungere quelle coordinate e quell'orientazione.

2D Pose Estimate

Consente di inizializzare il sistema di localizzazione impostando una posizione di partenza che verrà poi inviata al topic *initialpose* di ROS.

Come per *2D Nav Goal* si clicca su di una posizione del piano e si trascina fino ad impostare l'orientamento.

Si espongono ora alcuni tra i più importanti display forniti da Rviz.

Grid

Grid mostra una griglia di linee bidimensionale o tridimensionale lungo il piano. Essa sarà centrata all'origine del Target Frame.
Nella tabella 3.1 si elencano le proprietà caratterizzanti questo display.

Nome	Descrizione	Valori validi	Default
Plane Cell Count	Numero di celle da disegnare nel piano della griglia	1+	10
Normal Cell Count	Numero di celle da disegnare lungo la normale della griglia	0+	0
Cell size	Lunghezza, in metri, del lato di ogni cella	0.0001+	1
Line style	Stile della linea della cella	Linea, tratteggio	Linea
Line Width	Spessore della linea	0.0001+	0.03
Color	Colore delle linee	([0-255], [0-255], [0-255])	(127, 127, 127)
Alpha	Trasparenza delle linee	[0-1]	0.5
Plane	Piano su cui tracciare le linee	XY, XZ, YZ	XY

Tabella 3.1: Grid

Map

Map mostra una mappa sul piano, cioè la descrizione di un ambiente che potrebbe essere popolato da oggetti.
Per ogni punto del piano Map va allora a definire se esso è occupato o meno attraverso un messaggio di tipo *nav_msgs/OccupancyGrid*: l'occupacy grid viene trattata associando un valore pari a 100 all'occupato, 100 va infatti a rappresentare il colore nero, ed il valore 0, il bianco, allo spazio libero. Tutto ciò che non si riesce a decodificare e quindi resta sconosciuto viene rappresentato dal colore grigio.
La tabella 3.2 descrive Map.

Nome	Descrizione	Valori validi	Default
Topic	Argomento da sottoscrivere	Graph Resource Name valido	Stringa vuota
Alpha	Trasparenza da applicare alla mappa	[0-1]	1
Request frequency	Quanto spesso chiedere nuovamente la mappa (secondi); con 0 non si richiede più la mappa	0+	0

Tabella 3.2: Map

Markers

Il display Markers permette al programmatore di visualizzare forme primitive arbitrarie come frecce, linee, cubi, sfere, cilindri, triangoli, punti e mesh.

Odometry

Odometry accumula un messaggio *nav_msgs/Odometry* nel corso del tempo. Questi messaggi vengono poi mostrati sotto forma di frecce.

Si veda la tabella 3.3 per uno studio dettagliato delle proprietà del display.

Si ricorda che l'odometria non è altro che l'uso di dati da sensori in movimento per stimare cambiamenti della posizione durante il corso del tempo.
Il termine deriva da due parole greche: *hodos*, viaggio, cammino, percorso, e *metron*, misura.
L'odometria viene usata dai robot, sia dotati di gambe sia di ruote, per stimare (non determinare) la loro posizione in relazione ad una locazione iniziale.
Questo metodo è sensibile agli errori dovuti all'integrazione delle misure della velocità nel tempo. Per usare effettivamente l'odometria sono allora necessari: una rapida ed accurata raccolta dei dati, una taratura delle attrezzature, un'accurata elaborazione.

Nome	Descrizione	Valori validi	Default
Color	Colore della linea	([0-255], [0-255], [0-255])	(255, 25, 0)
Topic	Argomento da sottoscrivere	Graph Resource Name valido	Stringa vuota
Position tolerance	Distanza lineare, in metri, alla quale l'odometria deve cambiare e creare una nuova linea	0.0001+	0.1
Angle tolerance	Distanza angolare alla quale viene pubblicata una nuova linea	[0.0001, 1]	0.1
Keep	Numero di linee da tenere prima che una nuova linea faccia sparire una vecchia linea	0+	100

Tabella 3.3: Odometry

RobotModel

RobotModel carica i link del robot, definiti dal modello URDF del robot stesso (per l'analisi del modello si rimanda il lettore al capitolo 5), nelle loro corrette posizioni, descritte dall'albero delle trasformate TF.
Le proprietà del display vengono riportate nella tabella 3.4.

Nome	Descrizione	Valori validi	Default
Visual enabled	Abilita la possibilità di disegnare il robot	True, False	True
Collision enabled	Abilita la possibilità di disegnare lo spazio delle collisioni dei link del robot	True, False	False
Update rate	Velocità alla quale aggiornare la posizione di ogni link, in secondi	0.01+	0.1
Alpha	Trasparenza da applicare ai link	[0-1]	1
Robot description	Parametro da cui ricavare l'URDF	Graph Resource Name valido	robot_description

Tabella 3.4: RobotModel

TF

TF mostra la gerarchia, e quindi l'albero, delle trasformate TF.
Vi sono tre porzioni di dati che possono essere visualizzate: il nome del frame, gli assi del frame e le frecce che congiungono un frame col suo genitore.
Se vengono caricati gli assi, l'asse X viene indicato col colore rosso, l'asse Y con il verde e l'asse Z con

il blu (nel capitolo 5 verrà mostrato l'albero del robot Robovie-X).
Le peculiarità del display vengono elencate nella tabella 3.5 di pagina seguente.

Nome	Descrizione	Valori validi	Default
Show names	Decide se mostrare o meno i nomi dei frame	True, False	True
Show axes	Decide se mostrare o meno gli assi dei frame	True, False	True
Show arrows	Decide se mostrare o meno le frecce tra padri e figli	True, False	True
All enabled	Decide se mostrare tutti i frame	True, False	True

Tabella 3.5: TF

TF permette anche di settare le proprietà di ciascun singolo frame. Si veda la tabella 3.6 per ulteriori dettagli.

Nome	Descrizione	Valori validi	Default
Enabled	Decide se mostrare il frame	True, False	Valore di *All Enabled*
Parent	Mostra il nome del genitore di questo frame		
Position	Mostra la posizione del frame, nel sistema di riferimento fisso		
Orientation	Mostra l'orientazione del frame, nel sistema di riferimento fisso		

Tabella 3.6: TF per il singolo frame

CAPITOLO 4

OCCUPANCY GRID TRIDIMENSIONALE

L'occupancy grid tridimensionale viene costruita per mezzo dei tool forniti dalla libreria OctoMap di ROS, libreria rilasciata sotto licenza BSD che mette a disposizione strutture dati ed algoritmi di mapping sviluppati in C++ .

Si citano i suoi ideatori: Kai M. Wurm, Armin Harnung, Maren Bennewitz, Cyrill Stachniss e Wolfram Burgard [2].

L'implementazione della mappa si basa sull'*octree*, struttura dati ad albero in cui ogni nodo interno ha esattamente otto figli (da qui il nome *octree* = *oct* + *tree*). Come nel nostro caso, gli octree vengono usati principalmente per partizionare uno spazio tridimensionale andando a suddividerlo ricorsivamente in ottanti come mostrato in figura 4.1.

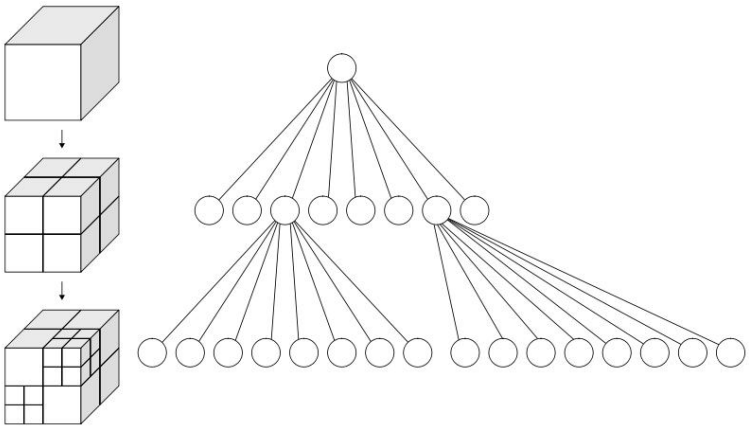

Figura 4.1: Octree

Il modello 3D viene creato di modo da rispettare i seguenti requisiti:

- completezza: la mappa deve essere in grado di modellare ambienti arbitrari nella loro totalità e senza che vengano effettuate assunzioni a priori. Il modello rappresentativo di uno spazio dovrà quindi comporsi di aree occupate e spazi liberi, ma verrà codificata anche la non disponibilità di informazioni su di un'area (*unknown areas*). Mentre la distinzione tra spazi occupati e liberi è essenziale per una navigazione sicura del robot, l'informazione sulle aree non conosciute è importante, per esempio, in caso di esplorazione autonoma dell'ambiente;

- possibilità di aggiornamento: deve esserci la possibilità di aggiungere in qualsiasi istante nuova informazione, reperita per esempio dai sensori del robot durante il moto. Questo implica la presenza di una mappa inizializzata a priori ma estendibile nel momento in cui vengono esplorate nuove aree. La possibilità di aggiornamento è importante per poter affrontare ambienti dinamici, in cui gli ostacoli possono spostarsi liberamente, oltre che ambienti multi-agente, in cui più robot possono contribuire alla formazione della stessa mappa con conseguente risparmio di tempo e maggiore completezza;

- flessibilità: l'estensione della mappa non deve essere nota a priori ma la mappa deve poter venire dinamicamente espansa ogniqualvolta ve ne sia di bisogno. Un'altra caratteristica importante che permette il raggiungimento di questo requisito è la multi-risoluzione: ogni mappa deve essere dotata di più gradi di accuratezza di modo che, per esempio, un pianificatore ad alto livello possa usare una mappa grossolana, mentre un pianificatore locale possa operare usando una risoluzione più fine.

OctoMap è stata rilasciata in ROS e per questo può venir direttamente inclusa all'interno di un qualsiasi nodo in esecuzione su questo sistema operativo.
ROS fornisce lo stack *octomap_mapping* che contiene i seguenti pacchetti:

- octomap, dedito all'estrazione e alla compilazione del codice SVN di SourceForge così da rendere OctoMap disponibile in ROS. Esso infatti contiene l'ultimo release proveniente dalla repository di OctoMap e fornisce un modo conveniente per scaricare e compilare la libreria in modo che possa essere gestita dal sistema di dipendenza ROS;

- *octomap_ros*, che mette a disposizione funzioni di conversione tra ROS e OctoMap;

- *octomap_server*, che fornisce il nodo ROS octomap_server_node; quest'ultimo permette di caricare file di tipo OctoMap, creare visualizzazioni in *RViz* ed inviare la mappa ad altri nodi.
  La mappa caricata dal nodo può essere statica (in questo caso viene direttamente caricata la mappa in formato .bt creata per mezzo di OctoMap ponendo il file come argomento della linea di comando del nodo) oppure può venir incrementalmente costruita dai dati in entrata (attraverso la funzione *PointCloud2*).
  Si nota che *octomap_server_node* inizia la sua esecuzione con una mappa vuota se da linea di comando non gli viene dato in ingresso alcun argomento.
  La distribuzione agli altri nodi avviene per mezzo di stream binari attraverso il topic *octomap_binary*.
  La visualizzazione delle celle occupate è resa possibile in *Rviz* inviando a questo sistema la mappa come *MarkerArra*y.

Si analizza nel dettaglio il procedimento seguito per la formazione della mappa tridimensionale.

Attraverso i tool *binvox*, *viewvox* e *binvox2bt*, messi a disposizione dalla libreria, ed il pacchetto *octomap*, di *octomap_mapping*, si crea il file OctoMap desiderato.
Questo file viene poi caricato in ROS, e visualizzato in *Rviz*, per mezzo del nodo *octomap_server*.

4.1 Passo 1: binvox

Innanzitutto si va a creare con *binvox*, programma a riga di comando che trasforma file 3D in .binvox sviluppato da Patrick Min del dipartimento di CS di Princeton [9], la versione voxelized della mesh 3D a disposizione, mesh che contiene un disegno tridimensionale dell'ambiente di cui si vuole costruire l'occupancy grid.
La *voxelization* non è altro che il processo di conversione di un insieme di poligoni tridimensionali nei blocchi equivalenti.
Binvox può essere eseguito su di una grande varietà di dati 3D: Wavefront OBJ, VRML 2.0, UG, OFF, DXF, XGL, POV, BREP, PLY, JOT. La maggior parte di essi è supportata solo per i poligoni, a parte VRML

che ricopre qualsiasi tipo di oggetto,ma OBJ è nella realtà il formato supportato nel modo migliore. Per questo nell'elaborazione è stato usato un modello 3D di tipo OBJ.
Supponendo che questo modello sia noto col nome di *filename.obj* si va allora ad effettuarne la voxelization.
Si esegue:

```
./binvox -d 32 -c filename.obj
```

Se il modello è costituito da particolari sottili, per non perderli bisogna impostare una risoluzione più alta, per esempio con:

```
./binvox -d 256 -c filename.obj
```

Viene così creato il file filename .binvox.
Questo output può essere visualizzato dal programma interattivo *viewvox OpenGL*, messo a disposizione dagli stessi autori, basta digitare

```
./viewvox filename.binvox
```

In *viewvox* ogni asse cartesiano è caratterizzato da un colore: l'asse x è rosso, l'asse y è verde e l'asse z è blu. Alcuni modelli usano l'asse z come l'asse verticale al suolo mentre in *viewvox* è l'asse y ad avere questa caratteristica. Si può però provvedere ad una rotazione impostando il parametro -rotx in *binvox*, che fa ruotare l'oggetto di 90 gradi in senso antiorario attorno all'asse x, andando effettivamente a cambiare l'asse verticale da y a z. Anche nel caso in analisi l'asse è stato cambiato e quindi il comando effettivamente eseguito è stato:

```
./binvox -d 400 - rotx -c filename.obj
```

e la successiva visualizzazione è stata molto dettagliata ed ha avuto assi correttamente orientati.

Di default *viewvox* rimuove i voxel interni creando modelli cavi. Se si vogliono mantenere i voxels interni, si esegue:

```
./viewvox -ki filename.binvox
```

I comandi da mouse e tastiera disponibili in *viewvox* vengono stampati nella finestra del terminale in cui è stato eseguito il comando.
Di seguito se ne analizzano alcuni:

- *s* per far vedere una slice del modello voxel;
- *k* e *j* per far vedere la slice successiva situata rispettivamente sopra o sotto a quella corrente (l'indice della slice corrente viene visualizzato all'interno della finestra del terminale);
- *a* per vedere i voxels a colori alternati;
- *n* per vedere contemporaneamente le slice sopra e sotto a quella corrente;
- *t* per far sì che queste slice diventino trasparenti.

È inoltre possibile:

- mostrare i valori delle coordinate nell'immagine (premendo 1);
- passare dalla proiezione ortografica a quella prospettica (premendo p);
- capovolgere verso il basso gli assi X, Y o Z (premendo x, y, or z) (X, Y, o Z per vederli invece nell'altro verso);

- memorizzare e rimemorizzare le impostazioni (per esempio quando si inizia nuovamente *viewvox* con lo stesso modello di voxel, esso usa la stessa telecamera ed altre impostazioni);
- usare i tasti per far fare alla telecamera un passo (sinistra, destra, ecc.) pari ad un voxel per volta;
- la griglia corrente ha una cella per ogni voxel; essa può venir visualizzata completamente oppure un livello alla volta, una fetta per volta (premendo g);
- sia *binvox* sia *viewvox* mostreranno la dimensione dell'area riempita dal modello voxel e la sua bounding box. Si può rifilare il modello voxel attraverso questa bounding box (parametro della linea di commando -fit);
- veduta di una singola fetta munita del numero che contraddistingue quella fetta.

Si sta invece lavorando verso il raggiungimento dei seguenti obiettivi:

- aggiungere un'opzione che permetta una rotazione di 45 gradi;
- mostrare le coordinate (x, y) sui voxel;
- segnalibri e possibilità di salto alle posizioni della telecamera;
- mesh colorate da convertire in voxel colorati;
- supporto per leggere file .3ds e scrivere file .vxl;
- centrare una slice nella veduta;
- mostrare il numero di voxel in una slice;

4.2 Passo 2: binvox2bt

A partire dal file .binvox si crea con *binvox2bt* un octree binario che mette in evidenza spazi occupati, liberi e aree non note.
In particolare l'opzione –mark-free va a marcare tutti gli spazi liberi lasciando il resto *unknown*.
È anche possibile impostare il formato della bounding box con -bb xmin ymin zmin xmax ymax zmax che ridimensiona l'albero binario risultante.
Nella nostra elaborazione si esegue:

```
./binvox2bt --mark-free filename.binvox
```

e si ottiene *filename.bt* ove .bt è l'acronimo di bonsai tree, il formato dell'octree binario.

4.3 Passo 3: octomap_server_node

Da terminale si apre *Rviz* con

```
rosrun rviz rviz
```

Se sorgono problemi nel contattare il server si digita, in un nuovo terminale, il comando

```
roscore
```

per poi nuovamente lanciare *Rviz* dal terminale di partenza. *roscore* avvia l'omonima collezione di nodi e programmi, prerequisito essenziale di un sistema che si basa su ROS. È necessario che essa sia in esecuzione per far sì che i nodi ROS possano comunicare tra loro (si ricorda che il comando roslaunch avvia automaticamente roscore se rileva che non è ancora in esecuzione).

A questo punto il file .bt ottenuto al passo precedente può essere visualizzato:

```
octomap_server_node filename.bt
```

Con l'esecuzione di questo comando il nodo ROS invia la mappa ad *Rviz* come *MarkerArray*. Si aggiunge quindi il display MarkerArray in Rviz ed in esso si imposta il topic *occupied_cells_vis*; il frame di default deve essere *map*.
Si ricorda che un display in *Rviz* non è altro che un qualcosa che disegna qualcosa nel mondo tridimensionale e che è dotato di opzioni liberamente fissabili all'interno della lista di parametri che lo caratterizza.

Per ovviare la possibilità di visione non ottimale in *Rviz*, ROS mette a disposizione il pacchetto *octovis* all'interno dello stack *octomap_visualization*. *octovis* è un tool di visualizzazione basato su Qt e libQGLViewer creato appositamente per OctoMap e rilasciato sotto licenza GNU-GPL. Test hanno dimostrato che la visualizzazione in *octovis* è migliore e più efficiente rispetto a quella fornita da *Rviz*.

4.4 Lancio in Rviz

Per caricare la mappa all'interno del proprio ambiente di lavoro basta aggiungere, all'interno del file di lancio che si sta usando, il nodo *octomap_server_node* del pacchetto *octomap_server*, passandogli come argomento il file binario che mappa l'ambiente in analisi, il file avente estensione .bt, che nello specifico è *map.binvox.bt*.

Supponendo che questo file si trovi all'interno della cartella *bin* dello stesso pacchetto *octomap_server*, il file .launch risultante avrà la seguente struttura:

```
<launch>
...
<node name="octomap_server" pkg="octomap_server" type="octomap_server_node"
args="$(find octomap_server)/bin/map.binvox.bt" />
...
<\launch>
```

CAPITOLO 5

MAPPATURA DEL ROBOT

In ROS il modello di un robot viene rappresentato per mezzo di un formato XML noto come URDF, o *Unified Robot Description Format*.
URDF permette non solo di dare una rappresentazione visuale del robot, ma anche del suo collision model e della cinematica e dinamica che lo caratterizzano. Questo viene effettuato andando a mappare il robot come una lista di *link*, le parti rigide del robot, e *joint*, i giunti che connettono tra loro i link (si veda figura 5.1).

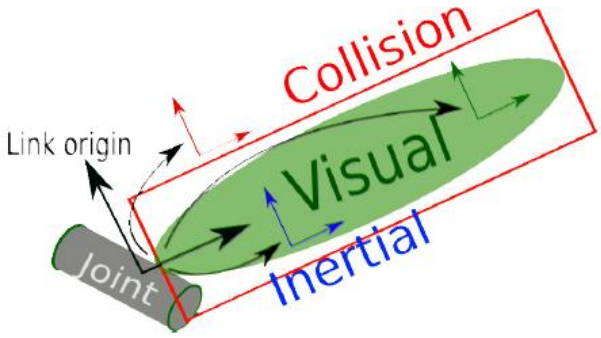

Figura 5.1: Costruzione di un modello URDF

Il processo svolto da URDF per descrivere un robot mette in luce l'impossibilità di rappresentare ciò che non è formato da link rigidi connessi da giunti di modo da formare strutture ad albero; non verranno allora supportati elementi flessibili o robot paralleli.

Nel nostro caso è stato utilizzato il modello URDF del Robovie-X costruito dallo studente Ordan [3]. Tale modello è stato riadattato di modo da renderlo fruibile anche in Gazebo, cosicchè anche la simulazione nell'ambiente 3D fosse ottimale.

5.1 Il modello

Di seguito si introduce la struttura del sopracitato file URDF.

Si considera il robot strutturato come un albero.
In esso la radice viene rappresentata attraverso il tag <robot>, che consente di specificare il nome del robot attraverso il suo unico attributo *name*.
A partire dalla radice si vanno ad elencare i diversi <link> che descrivono le componenti del corpo

stesso: bracci, gambe, torso e testa.
I link vengono tra loro uniti dai giunti, elencati nel modello per mezzo dei tag <joint>.

Si procede ad una descrizione più esaustiva di <link> e <joint>.

<link>

<link> consente di descrivere un corpo rigido. Ad esso è possibile dare un nome, attraverso l'attributo *name*, ed è possibile descriverne le proprietà visuali, di inerzia e collisione. Queste ultime sono implementate come elementi dotati di un proprio sistema di riferimento, che può coincidere o meno con quello del link, come da figura 5.1.
Nel dettaglio:

- <visual>: definisce le proprietà visuali del link: forma, materiale, ... Esso si compone di:
  - <origin>: rappresenta la trasformazione del sistema di riferimento dell'elemento di visualizzazione rispetto quello del link. Viene usato l'attributo *xyz* per definire l'offset x,y e z dell'origine; *rpy* per specificare invece gli angoli in radianti di roll, pitch e yaw;
  - <geometry>: descrive la forma dell'elemento. Si usa <box> se si vuole ottenere un parallelepipedo, <cylinder> per un cilindro, <sphere> per una sfera e <mesh>, attributo usato nella descrizione del robot in analisi, per descrivere una mesh;
  - <material>: viene usato se si vuole specificare il materiale che deve essere usato nella visualizzazione. Esso può essere caratterizzato da un particolare colore, <color>, ma anche da una particolare consistenza, <texture>.
- <collision>: descrive le proprietà di collisione del link. È possibile impostare proprietà di collisione uguali a quelle dell'elemento visual, tuttavia, per ragioni di efficienza, solitamente si utilizza un modello di collisione più semplice, basato su primitive geometriche. Supporta due attributi importanti:
  - <origin>: come in <visual> consente di specificare l'origine del sistema di riferimento del modello delle collisioni;
  - <geometry>: descrive il modello geometrico da utilizzare per il controllo delle collisioni. La struttura dell'elemento è la stessa del corrispondente elemento visual.
- <inertial>: consente di descrivere l'inerzia del modello. Si compone di:
  - <origin>: come già specificato;
  - <mass>: consente di specificare la massa del link. Questo attributo è essenziale per il corretto calcolo delle forze d'inerzia;
  - <inertia>: la matrice d'inerzia rispetto il sistema di riferimento dell'elemento d'inerzia. Si ricorda che questa matrice è simmetrica e quindi basta specificare solo sei degli attributi che la costituiscono: ixx, ixy, ixz, iyy, iyz e izz.
    Per speficificare le inerzie è stata utilizzata la tabella presente in [10].
    Si ricorda che queste forze devono essere correttamente impostate se si vuole che, all'interno del simulatore 3D usato, il modello resti in piedi in equilibrio anche se soggetto alla forza di gravità.

<joint>

I diversi link vengono tra loro uniti dai giunti: un giunto connette tra loro il link padre al figlio aggiungendo un elemento all'albero cinematico.

<joint> consente di descrivere la cinematica, la dinamica e i limiti del giunto. Inoltre consente di dare un nome ad ogni giunto, attraverso l'attributo *name*, e fissarne il tipo, con *type*. I tipi di giunto supportati sono:

- revolute: giunto a cerniera che ruota attorno ai suoi assi in un range limitato, come quelli definiti nel caso in analisi;
- continuous: giunto che può ruotare senza limiti attorno ai suoi assi;
- prismatic: giunto traslazionale che può traslare lungo gli assi in un range limitato;
- fixed: giunto bloccato in tutti i gradi di libertà;
- foating: giunto libero in tutti i suoi gradi di libertà;
- planar: giunto che vincola il movimento in un piano.

Come per <link>, anche <joint> è caratterizzato da attributi che ne consentono la completa descrizione. Tra quelli usati si ricordano:

- <origin>: rappresenta la trasformazione dal parent link al child link. Il giunto si trova all'origine del child link;
- <parent>: consente di indicare il nome del parent link attraverso l'attributo *name*;
- <child>: rappresenta il child link, specificato anche in questo caso per mezzo dell'attributo *name*;
- <axis>: è l'asse del giunto rispetto il sistema di riferimento del giunto. Asse di rotazione per giunti rotazionali, di traslazione per giunti prismatici, la normale del piano di scivolamento per giunti planari. L'attributo xyz rappresenta le componenti del vettore normalizzato.
  Per esempio, per un giunto rotazionale, il valore (1, 0, 0) sta ad indicare che la rotazione avverrà lungo l'asse x;
- <limit>: consente di specificar i valori limite entro i quali il giunto può muoversi. Sarà contraddistinto da limiti superiori, *upper*, e inferiori, *lower*. Sarà inoltre possibile specificare il massimo sforzo applicabile al giunto, con *effort*, e la massima velocità del giunto, con *velocity*.

Questi elementi hanno reso possibile la creazione di un modello avente la struttura specificata in figura 5.2.

Sempre all'interno del modello URDF sono stati inseriti attributi utili alla descrizione del robot all'interno del simulatore utilizzato, Gazebo. Per una descrizione più esaustiva di queste estensioni si rimanda il lettore al capitolo 10.

Utile è ricordare, al lettore che voglia costruire il proprio modello, che ROS fornisce *xacro*, un linguaggio XML che consente di costruire file XML più corti e semplici rispetto ad URDF.
Esso mette infatti a disposizione la possibilità di dichiarare delle macro.
Nel caso in analisi sono state create macro che definiscono le strutture di link e giunti. Queste macro sono state poi richiamate all'interno del codice alla definizione di ogni link e giunto del robot. In questo modo, anzichè andare a definire ad ogni richiamo la loro struttura, questa viene definita una volta soltanto.
Il codice diventa meno prolisso e più semplice.

Importante è sottolineare che il formato URDF viene utilizzato da molti package ROS, tra cui *robot_state_publisher*, così da ottenere alberi cinematici per TF a partire dallo stato dei giunti.

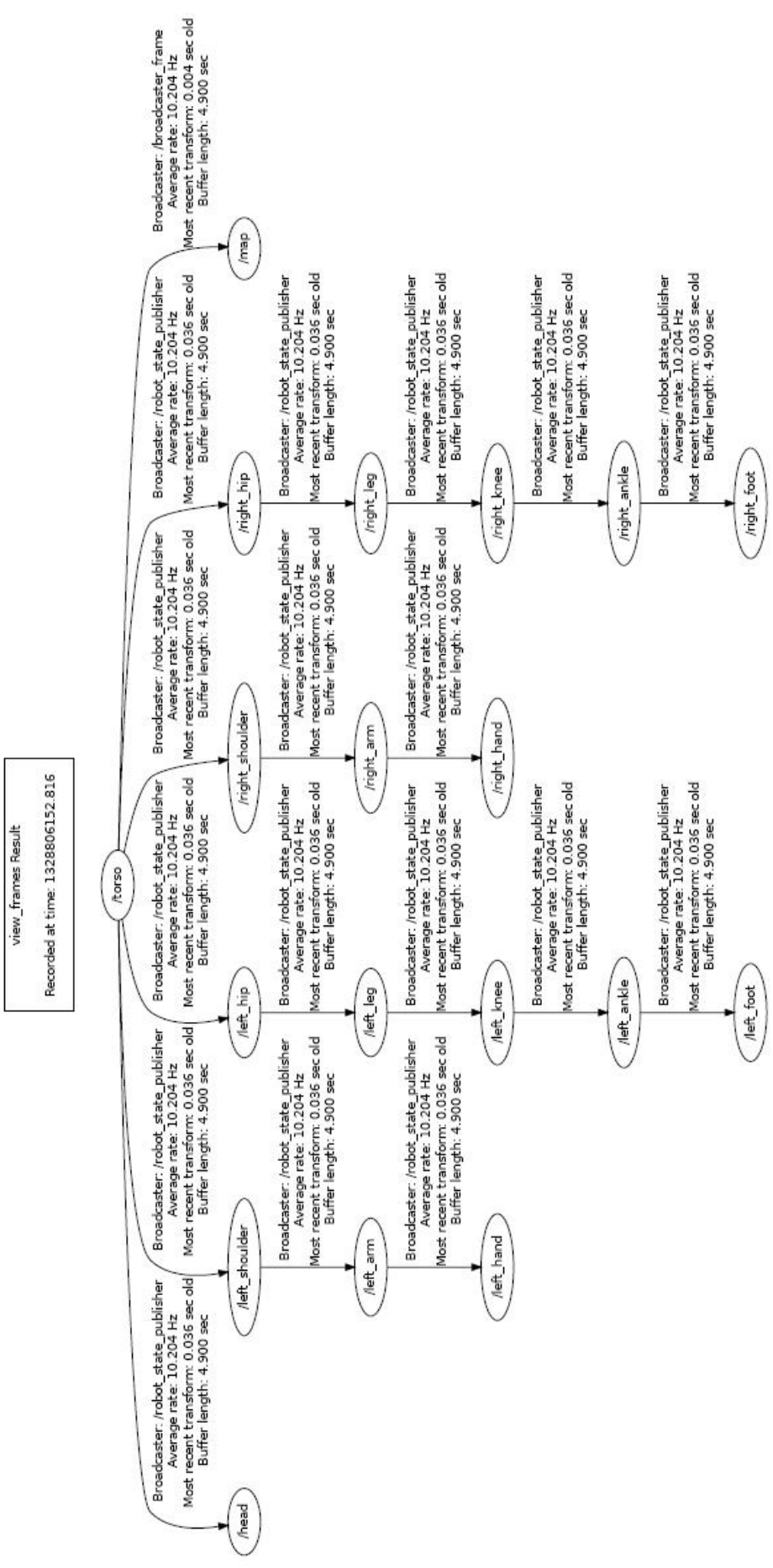

Figura 5.2: Gerarchia dei giunti del Robovie-X utilizzato

5.2 TF

Ci si sofferma allora sul pacchetto *TF*.
Esso permette all'utente di tener traccia nel tempo dei molteplici frame delle coordinate; le relazioni tra i frame delle coordinate vengono mantenute all'interno di una struttura ad albero così da dare la possibilità all'utente di trasformare punti o vettori tra due sistemi di coordinate in qualsiasi momento.
Un sistema robotico, infatti, ha tipicamente molti frame delle coordinate tridimensionali che cambiano nel corso del tempo, come il frame del mondo, il frame di base, il frame della testa o quello che si occupa delle prese. *TF* tiene traccia di tutti questi frame nel tempo permettendo così di rispondere a domande come:

- Dov'era il frame della testa in relazione al frame del mondo 5 secondi fa?
- Dov'è situato l'oggetto nel gripper in relazione alla base del robot?
- Qual'è la posizione corrente del frame di base nel frame del mondo?

TF opera in un sistema distribuito. Questo significa che tutte le informazioni sui sistemi delle coordinate di un robot sono disponibili a tutti i componenti ROS in qualsiasi computer del sistema. Non vi è alcun server centrale dedito alla memorizzazione delle informazioni sulle trasformazioni.

5.3 robot_state_publisher

Il pacchetto è di fondamentale importanza poiché permette di pubblicare lo stato del robot a TF.
Una volta che lo stato è stato pubblicato, esso è disponibile a tutti i componenti del sistema che usano TF.
robot_state_publisher prende gli angoli dei giunti del robot come input e pubblica delle posizioni tridimensionali dei link del robot, usando il modello dell'albero cinematico del robot.

Nel nostro caso i due pacchetti sopracitati hanno permesso di mettere in collegamento l'URDF del Robovie-X con la mappa creata al capitolo precedente.
Dapprima è stato usato *robot_state_publisher* per rendere disponibile a *TF* l'URDF utilizzato; è stato poi usato *TF* stesso per aggiungere il frame \map, frame di base per la mappa, ed impostarlo come figlio del frame \tronco_link1, frame di base del robot.
L'albero delle dipendenze risultante viene mostrato in figura 5.2.

5.4 map e tronco

Ci si sofferma sull'importanza del collegamento effettuato andando a descrivere nel dettaglio i frame \map e \tronco_link1.

Si ricorda che il sistema di coordinate \map è quello che va a rappresentare il sistema di coordinate fisso del mondo, con il suo asse Z rivolto verso l'alto.
Questo frame non è continuo, nel senso che in esso, in qualsiasi istante, la posizione di una piattaforma mobile può variare in modo discreto attraverso dei salti.
In una tipica configurazione, la componente di localizzazione costantemente ricalcola la posizione del robot in \map a seconda dell'informazione data dai sensori del robot stesso; in questo modo si eliminano i salti dovuti al movimento, non si toglie però la discretizzazione dovuta all'arrivo di nuova informazione da parte di un nuovo sensore.
Si può così notare che il frame \map è utile come riferimento globale a lungo termine, ma i salti discreti lo trasformano in un quadro di riferimento povero per il rilevamento e l'elaborazione locale.

Il sistema delle coordinate di base di un robot, nel nostro caso il frame \tronco_link1, è rigidamente fissato alla base del robot mobile.

Il punto di fissaggio può venir dislocato in qualsiasi posizione arbitraria del corpo e con qualsiasi orientazione, è vero infatti che a seconda della piattaforma hardware selezionata vi sarà un posto diverso sulla base che fornisce un punto di riferimento evidente.

Affinché più frame possano comunicare tra loro è necessario che essi vengano messi in collegamento attraverso delle dipendenze. È stata scelta una rappresentazione ad albero per attaccare tra loro tutti i frame delle coordinate nel sistema robotico. Ogni frame allora ha un frame delle coordinate padre ed un certo numero di figli.

Nel nostro caso è importante che il frame di base del robot comunichi con quello del mondo perché, solo con il loro collegamento, sarà possibile dislocare e far compiere del movimento al robot in un ambiente descritto da una mappa.
È stato quindi deciso di attaccare i frame descritti ponendo \tronco_link1 come genitore (esso è genitore anche del resto dei frame che rappresentano le componenti del robot) e \map come figlio.

5.5 Aggiungere un frame

Per aggiungere il frame \map all'albero dei sistemi di coordinate del robot si va innanzitutto a creare un nuovo package *add_frame* dipendente da *tf* e *roscpp*. In esso, all'interno della cartella *src*, si crea il file *frame_tf_broadcaster.cpp* contenente il seguente codice:

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "my_tf_broadcaster");
  ros::NodeHandle node;

  tf::TransformBroadcaster br;
  tf::Transform transform;

  ros::Rate rate(10.0);
  while (node.ok()){
    transform.setOrigin( tf::Vector3(0, 0, 0) );
    transform.setRotation( tf::Quaternion(0, 0, 0) );
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
    "torso_link1", "map"));
    rate.sleep();
  }
  return 0;
};
```

In esso, all'interno del ciclo *while*, viene creata una nuova trasformata dal padre *torso_link1* al nuovo figlio *map*, che si trova esattamente nella stessa posizione del padre senza alcuna translazione lungo gli assi.

Una volta creato il codice lo si può compilare. Dapprima si apre il file *CMakeLists.txt* e gli si aggiunge in fondo la seguente linea:

```
rosbuild_add_executable(frame_tf_broadcaster src/frame_tf_broadcaster.cpp)
```

poi si passa alla costruzione del pacchetto:

```
make
```

Se il processo viene eseguito correttamente si trova il file binario *frame_tf_broadcaster* all'interno della cartella *bin* di *add_frame*. Per aggiungere il nuovo frame appena creato in fase di lancio basta aggiungere nel file .launch il nuovo nodo:

```
<launch>
   ...
   <node pkg="add_frame" type="frame_tf_broadcaster"
         name="broadcaster_frame" />
 </launch>
```

Una volta aggiunto anche il frame map al robot, il modello risultante è quello di figura 5.3.

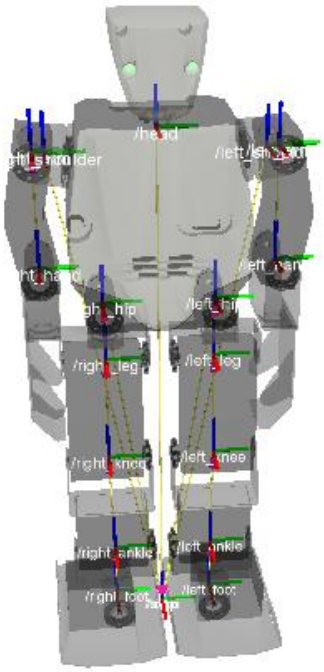

Figura 5.3: tf

5.6 Visionare l'albero dei frame

Potrebbe essere utile visionare l'intero albero delle trasformate delle coordinate, come in figura 5.2. A tal scopo basta usare il tool *view_frames*, che crea un file PDF contenente il grafo corrente:

```
rosrun tf view_frames
```

Se oltre a creare il grafo lo si vuole da subito visionare una volta pronto, basta digitare in Ubuntu:

```
rosrun tf view_frames
evince frames.pdf
```

5.7 Collision Model

Importante, per il planning, è poter visionare il modello delle collisioni del robot, a tal scopo ROS mette a disposizione il pacchetto *planning_environment*.
Questo pacchetto dipende dai parametri che vengono caricati nella configurazione YAML del robot ed in particolare da quelli pubblicati sotto i topic *robot_description_planning/groups* e *robot_descrip-*

tion_collision/default_collision_operations.

Il gruppo di parametri *robot_description_planning/groups* esiste per definire i gruppi di giunti e link che costituiscono il robot. Questi gruppi permettono di controllare la pianificazione e la collisione; i planner infatti si possono riferire a questi gruppi e questi gruppi possono essere usati per abilitare o disabilitare i controlli di collisione su opportuni insiemi di link.
Ad ogni gruppo viene dato un nome che identifica l'insieme di link e giunti che vi appartengono. Giunti e link possono comparire in più di un gruppo ed un gruppo non deve per forza contenere sia giunti sia link. L'unione dei link di tutti i gruppi verrà usata nel collision checking, quindi tutti i link per i quali deve venir fatto un controllo di collisione devono essere inclusi in qualche gruppo. Si nota che è permesso formare anche un gruppo costituito da un solo link.

L'altro topic importante del file YAML corrisponde invece alla parametrizzazione delle operazioni di collisione di default per il controllo dell'auto-collisione ed è contenuto in *robot_description_collision*. Secondo l'impostazione predefinita, tutti i link del robot inclusi nei gruppi di *robot_description_planning* vengono aggiunti all'interno dello spazio delle collisioni dell'ambiente di pianificazione, ma il controllo delle auto-collisioni non viene abilitato di default. Questo significa che la configurazione di partenza dello spazio delle collisioni non riporta che un dato stato è in collisione anche se entrambe le braccia del robot stanno tentando di occupare lo stesso spazio.
Tuttavia, se le operazioni di collisione vengono specificate in *default_collision_operations*, un utente dell'ambiente di pianificazione può definire una configurazione di default sicura, che rileva auto-collisioni dannose anche quando un qualsiasi utente del servizio di planning environment non include alcuna specifica richiesta di operazioni di collisione.

La richiesta di operazione di collisione ha il seguente formato:

```
-object1: <planning_group or link name>
 object2: <planning_group or link name>
 operation: <enable or disable>
```

Tutte le richieste vengono tra loro unite di modo da creare una lista e tutte le operazioni contenute in questa lista verranno applicate allo spazio delle collisioni.
Ogni oggetto può specificare un singolo link o il nome di un planning group definito in *robot_description_planning*. L'operazione *enable* abilita il controllo di auto-collisione tra due oggetti, *disable* invece disabilita questo controllo.

È vero che, nel momento in cui si controllano le collisioni, è spesso necessario prendere in considerazione le auto-collisioni tra i link del robot. Ma a causa della geometria del robot alcuni link non saranno mai in collisione con altri.
Diventa allora vantaggioso indicare tra quali link verificare le collisioni così da ridurre il numero di controlli. Questo viene appunto specificato usando le collision operations, che consentono all'utente di abilitare o disabilitare i controlli di collisione tra certi insiemi di link. Se i controlli di collisione sono disabilitati, il sistema ignorerà una collisione tra due link.
Di default tutte le collisioni sono disabilitate e quindi qualsiasi configurazione del robot viene considerata valida. Qualsiasi possibilità di collisione tra due link deve così venir completamente specificata nel file di configurazione.

Si ricorda che alcuni giunti con molti gradi di libertà non sono definiti all'interno dell'URDF, per esempio manca la mappatura del giunto che connette il robot al mondo; questi devono comunque venir specificati per un corretto funziomento del corpo nel mondo in cui viene inserito.
La definizione viene fatta in questa sede sotto il topic *multi_dof_joints*. Qui si va a mappare il frame del mondo come padre del frame base del robot.

Figura 5.4: Arm Navigation Wizard

Sono stati mappati i gruppi di giunti e link del robot Robovie-X utilizzato nell'esperimento e sono state definite le relazioni esistenti tra essi attraverso una GUI nota come *Arm Navigation Wizard*. Questa ha permesso la creazione di tutti i tool necessari al funzionamento di *planning_environment*. Nella sezione seguente se ne descrive il funzionamento.

5.8 Arm Navigation Wizard

Arm Navigation Wizard (in figura 5.4 viene presentata l'interfaccia grafica) è un programma grafico che guida l'utente attraverso la generazione automatica dell'insieme delle configurazioni del robot.

Viene lanciata da terminale specificando l'URDF del robot, nel nostro caso con

```
roslaunch planning_environment planning_description_configuration_wizard.launch \
urdf_package:=robovie_x_model urdf_path:=xacro/roboviex.urdf
```

Questo comando fa apparire, in due finestre separate, il visualizzatore ROS (*Rviz*) e *Wizard*. Il primo mostra il modello URDF del robot su di una griglia rappresentativa dell'ambiente, il secondo interagisce col primo mostrando le configurazioni dei giunti e dei bracci che sono state sviluppate durante la procedura guidata.

Le configurazioni create vengono generate per mezzo di un campionamento automatico degli stati dei giunti (campionamento uniforme randomizzato) seguito da un controllo volto a fare in modo che in questi stati non vi siano autocollisioni.

Il livello di campionamento può essere scelto dall'utente: più elevata è la densità del campionamento maggiore è l'accuratezza del risultato, ma maggiore è anche il tempo riservato alla computazione, che può raggiungere l'ordine delle ore. Bisogna quindi scegliere un giusto equilibrio tra accuratezza e costo.

Le configurazioni vengono raggruppate in planning groups: i gruppi di parti del robot citati nella sezione precedente. Un gruppo potrà, ad esempio, essere costituito da un braccio ed il suo end-effector. Vengono considerate non solo le catene cinematiche (nell'esempio precedente tronco-braccio-mano),

ma anche l'insieme di giunti che si interpongono tra le diverse componenti.

Quanto creato viene posizionato all'interno del package *roboviex_arm_navigation* suddiviso in due cartelle: *config* e *launch*. La prima contiene i file di configurazione, si citano i più importanti:

- *joint_limits.yaml*: definisce i limiti di velocità ed accelerazione dei giunti del robot. Si può editare per aumentare le performance;
- *roboviex_planning_description.yaml*: definisce i multi-dof joints che sono nell'URDF. Contiene anche la lista di operazioni che vengono eseguite per realizzare la funzione di collision checking correttamente ed efficientemente;
- *ompl_planning.yaml*: si compone dei file di configurazione per i planner di OMPL.

La directory *launch*, invece, include tra gli altri:

- *roboviex_planning_environment.launch*: nodo che mantiene le informazioni sull'ambiente;
- *constraint_aware_kinematics.launch*: nodo che lancia la cinematica per i due bracci;
- *ompl_planning.launch*: nodo dedito alla pianificazione;
- *move_right_arm.launch*, *move_left_arm.launch*, *move_right_leg.launch* e *move_left_leg.launch*: nodi che lanciano lo stato delle macchine per poi procedere con la pianificazione e l'esecuzione;
- *trajectory_filter_server.launch*: nodo che filtra le traiettorie e le smussa.

Il file generato, utile per il *planning_environment*, sarà *roboviex_planning_description.yaml*, caratterizzato dalla seguente forma:

```
multi_dof_joints:
  - name: world_joint
    type: Floating
    parent_frame_id: map
    child_frame_id: torso
groups:
  - name: head
    base_link: torso
    tip_link: head
  - name: head_joint
    joints:
      - hj1
  - name: left_arm
    base_link: torso
    tip_link: left_hand
  - name: left_arm_joints
    joints:
      - laj1
      - laj2
      - laj3
  - name: left_leg
    base_link: torso
    tip_link: left_foot
  - name: left_leg_joints
    joints:
      - llj1
```

```
      - llj2
      - llj3
      - llj4
      - llj5
  - name: right_arm
    base_link: torso
    tip_link: right_hand
  - name: right_arm_joints
    joints:
      - raj1
      - raj2
      - raj3
  - name: right_leg
    base_link: torso
    tip_link: right_foot
  - name: right_leg_joints
    joints:
      - rlj1
      - rlj2
      - rlj3
      - rlj4
      - rlj5
default_collision_operations:
  - object1: torso
    object2: head
    operation: disable  # Adjacent in collision
  - object1: torso
    object2: left_shoulder
    operation: disable  # Adjacent in collision
  - object1: left_shoulder
    object2: left_arm
    operation: disable  # Adjacent in collision
  - ...
  - ...
  - object1: right_knee
    object2: right_shoulder
    operation: disable  # Never in collision
  - object1: right_leg
    object2: right_shoulder
    operation: disable  # Never in collision
```

5.9 Visualizzazione del modello delle collisioni del robot

Il *planning_environment* ha reso possibile la visualizzazione del modello di collisione del robot ogniqualvolta necessario: è stato creato un tool che permette di vedere se la posizione in cui si vuole dislocare il robot è buona o porta il corpo in uno stato di collisione facendo visualizzare il collision model del robot con colore verde nel primo caso e con colore rosso nel secondo.

Per far ciò, dato il file cpp in cui si sta lavorando ed il suo header, si è proceduto nel seguente modo: nell'header è stato incluso l'header che definisce il modello delle collisioni con il comando

```
#include <planning_environment/models/collision_models.h>
```

e sono stati inizializzati

```
ros::Publisher marker_pub;
planning_environment::CollisionModels collision_models;
planning_models::KinematicState* state;
```

Il primo è un Publisher che consente la pubblicazione dei marcatori e quindi la visualizzazione delle forme, nel nostro caso la sagoma degli arti e dei giunti che costituiscono il robot.
Il secondo inizializza una classe di tipo *CollisionModels*, classe capace appunto di caricare il modello del robot da un server dei parametri e che nel nostro caso servirà per riprodurre il modello delle collisioni del robot stesso.
L'ultimo è invece un oggetto di tipo *KinematicState*, che definisce uno stato cinematico andando a descrivere quantitativamente il moto del robot. Si ricorda la defizione di cinematica come geometria del movimento: la cinematica di un punto si può pensare come geometria dello spazio vettoriale quadridimensionale formato dalle tre coordinate spaziali e della coordinata temporale.

Una volta inizializzato l'ambiente è stata poi inizializzata la funzione che permette di impostare lo stato cinematico del robot alle coordinate (x, y, θ), coordinate della mappa 2D che rappresenta il suolo:

```
void updateRobotPosition(float x, float y, float theta);
```

Passando quindi alla descrizione dei contenuti del file cpp atti alla visualizzazione del robot rosso o verde a seconda della bontà della posizione stimata, si è proceduto in questo modo:

```
...
collision_models("robot_description")
...
state = new planning_models::KinematicState(collision_models.getKinematicModel());
marker_pub = nh_private.advertise<visualization_msgs::MarkerArray>
     ("visualization_marker", 10);
...
...
good_color.a = collision_color.a = .8;
good_color.g = 1.0;
collision_color.r = 1.0;
...
void FootstepPlanner::updateRobotPosition(float x, float y, float theta){
   btTransform cur(tf::createQuaternionFromYaw(theta), btVector3(x, y, 0.0));
   state->getJointStateVector()[0]->setJointStateValues(cur);
   state->updateKinematicLinks();
}
....
void setPosition(float x, float y, float theta){
...
   visualization_msgs::MarkerArray robotMarker;
   updateRobotPosition(x, y, theta);
   collision_models.getRobotMarkersGivenState(*state,
                                              robotMarker,
                                              collision_color,
                                              "Goal pose in collision",
                                              ros::Duration(10.0));
   marker_pub.publish(robotMarker);
}
...
```

Prima tra tutte viene svolta l'operazione che crea la classe *CollisionModels*: viene presa la scena di pianificazione, associata a *robot_description* perché questo parametro contiene l'URDF del robot, ed in base a questa si setta la classe *collision_models*; verranno impostate tutte le informazioni pertinenti ed alla fine si otterrà una ricostruzione esatta dello stato del mondo esistente all'interno dell' *environment_server* nel momento a cui si riferisce la scena di pianificazione.
La chiamata successiva restituisce l'oggetto KinematicState che sarà fissato allo stato attuale robot.
Si avvia poi un array di marcatori per la visualizzazione delle forme.
Si impostano i colori verde come *good_color*, colore che identifica una buona posizione per il robot, e rosso come *collision_color*, il colore della collisione.
Si definisce infine la funzione che aggiorna la posizione del modello cinematico del robot e la si inserisce all'interno della funzione che va a prendere stati della mappa di modo da poter comunicare all'utente il buon fine o meno della sua scelta appunto con robot rosso e verde a seconda del caso.

5.10 Riepilogo

Aggiungendo tutti i nodi necessari ad ultimare il lavoro, robot e mappa, il file di lancio avrà struttura finale

```
<launch>

<!--includo la mappa-->

<node name="octomap_server" pkg="octomap_server" type="octomap_server_node"
 args="$(find octomap_server)/bin/map.binvox.bt" />

<!-- imposto la risoluzione della griglia -->
<param name="resolution" value="0.01" />

<!-- carico il modello urdf del robot -->
<arg name="model" default="$(find robovie_x_model)/xacro/robovie-x.xacro"/>
<param name="robot_description" textfile="$(find
robovie_x_model)/xacro/roboviex.urdf" />

<!-- aggiungo il collegamento alla mappa -->
<node pkg="add_frame" type="frame_tf_broadcaster" name="broadcaster_frame" />

<!-- aggiungo la mappatura dei giunti fatta in Wizard con le relative
   collisioni-->
<rosparam command="load" ns="robot_description_planning" file="$(find
 roboviex_arm_navigation)/config/roboviex_planning_description.yaml" />

<!-- pubblico le trasformate-->
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" ></node>
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />

<!--pacchetto che serve per far muovere il robot -->
<node name="environment_server" pkg="planning_environment"
type="environment_server" />

<!--lancio rviz con la configurazione adatta a mostrare mappa tridimensionale
    con le celle occupate, robot, tf, ..-->
<include file="$(find footstep_planner)/launch/rviz_footstep_planning.launch" />
```

```
<!-- parametri del piede del robot e parametri di planning-->
<include file="$(find footstep_planner)/launch/footstep_planner.launch" />

</launch>
```

CAPITOLO 6

MOTION PLANNING

Pronti l'occupancy grid tridimensionale ed il robot su di essa posizionato, si può risolvere il problema di Motion Planning.

Ambiente e corpo vengono visualizzati in Rviz e, attraverso *2D Nav Goal* e *2D Initial Pose*, si impostano posizione iniziale e obiettivo.
Una volta cliccato su di un punto della mappa per impostare la posizione, la sua bontà viene comunicata all'utente con l'apparizione di una sagoma rossa o verde a seconda della presenza o meno di collisioni (come illustrato nel capitolo precedente).
Si ricorda che un robot deve sempre trovarsi in punti dello spazio in cui non urta contro gli oggetti che lo popolano; per questo verranno considerate valide solo le posizoni contraddistinte da sagome verdi.

Fissati partenza ed arrivo si passa al calcolo di un percorso valido che colleghi i due punti: questo percorso deve tener conto delle dimensioni del robot, dei passi che esso andrà a compiere e deve ovviamente considerare il fatto che ad ogni passo il robot non deve entrare in collisione.

Il percorso valido viene mappato sul suolo come una sequenza di punti. Questi dovranno essere ricoperti dalle suole dei piedi dell'automa e per questo sarà necessario un ulteriore controllo di collisione lungo tutta la superficie ricoperta dal piede attorno al punto.

Se la posizione trovata è valida, viene visualizzata l'orma del piede. Ovviamente un cammino sarà composto da un'alternanza di piedi sinistro e destro. Questo è possibile attivando i flag *RIGHT* e *LEFT*, che ad ogni passo vengono alternati.

6.1 Mappatura dell'ambiente per il controllo delle collisioni

Affinché possa essere effettuato un controllo delle collisioni è necessario che venga mappato tutto l'ambiente di modo da sapere se i punti che lo compongono sono o meno occupati.
A tal scopo è stata costruita l'occupancy grid tridimensionale; ma per velocizzare la creazione del percorso sul suolo quest'ultima è stata dapprima traslata in una occupancy grid a due dimensioni.

La mappa a due dimensioni è una griglia di dimensione XxY che viene suddivisa in XxY punti, ciascuno dei quali contraddistinto da un flag di punto libero o occupato ed inserito all'interno di una lista.

Una volta che il planner va a selezionare il punto da inserire tra la sequenza di quelli che compongono il cammino, lo controlla secondo la lista creata: se libero farà parte della sequenza, altrimenti verrà scartato e si effettuerà una nuova ricerca.

6.2 La suola del piede

Importante è soffermasi sul controllo di collisione della suola del piede.
Si applica un controllo efficiente nella mappa delle distanze simile al metodo suggerito da Sprunk et al. per il controllo delle collisioni di un robot rettangolare dotato di ruote [4]: la suola del piede viene ricorsivamente suddivisa secondo le sue circonferenze circoscritta ed inscritta (vedi figura 6.1).

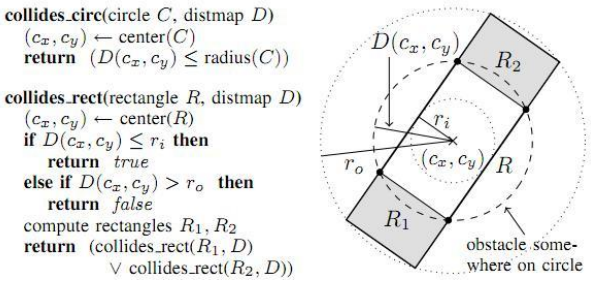

Figura 6.1: Algoritmo per il controllo della collisione della suola del piede

In particolare, l'andare a controllare le traiettorie, affinché queste non siano afflitte da collisioni, domina il costo computazionale durante l'ottimizzazione; per un controllo efficiente l'algoritmo va allora ad approssimare la suola del robot ed il suo carico utile attraverso una serie di cerchi e rettangoli. Usando una mappa delle distanze dell'ambiente dinamicamente aggiornabile, $D(x; y)$, questi elementi possono venir efficientemente controllati da collisioni con i metodi esposti in figura 6.1: il disegno mostra un elemento rettangolare R con diametri interno ed esterno rispettivamente pari a r_i ed r_o. Solo se la distanza $D(c_x, c_y)$, che separa il centro di R dall'ostacolo, è tra r_i ed r_o, il risultato dipende da ricorsivi controlli di collisione delle sottoparti R_1 ed R_2 (porzioni importanti in quanto garantiscono l'esistenza di regioni di sicurezza).
Si nota che la necessità di un numero superiore di ricorsioni può determinare l'assunzione di collisione.

6.3 Planner utilizzato

Viene utilizzato un algortimo fornito dalla libreria SBPL, o *Search-Based PLanning*, fornita da ROS e sviluppata da Maxim Likhachev all'University of Pennsylvania in collaborazione con la Willow Garage.

L'algoritmo prevede la generazione di un grafo che vada a mappare i punti dell'ambiente e, a grafo completato, che ricerchi un collegamento tra i due punti desiderati.

L'algoritmo utilizzato è ARA*, la *anytime version* di A*.

Inizialmente si pensava di utilizzare l'algoritmo KPIECE offerto da OMPL, ma dopo prove sperimentali è stato riscontrato che tale algoritmo non raggiunge l'efficienza di quello usato.
I sampling based planner, infatti, sono ottimi per ottenere percorsi praticabili in spazi ad elevate dimensioni, ma i cammini generati possono essere anche di qualità arbitrariamente scarsa se giudicati secondo metriche quali la distanza. Questo significa che si può finire a navigare dall'altra parte della stanza e poi di nuovo in punti vicini all'obiettivo, che generalmente non è quanto desiderato da un progettista.
Questi problemi potrebbero essere affrontati andando a *smussare* il percorso, ma si potrebbe finire

col trovare ancora una volta percorsi che nemmeno si avvicinano a quello ottimale.
La ricerca effettuata da pianificatori come A*, d'altra parte, garantisce la produzione di percorsi ottimali o percorsi che si avvicinano a quello ottimo di un qualche fattore.
Ciò spiega l'utilizzo di ARA* nella risoluzione del problema di Motion Planning, sarà invece utile usare KPIECE per far muovere gli arti, qui infatti non si lavora più su di una mappa a due dimensioni ma all'interno di uno spazio.

Di seguito si descrive più in profondità l'algoritmo utilizzato.

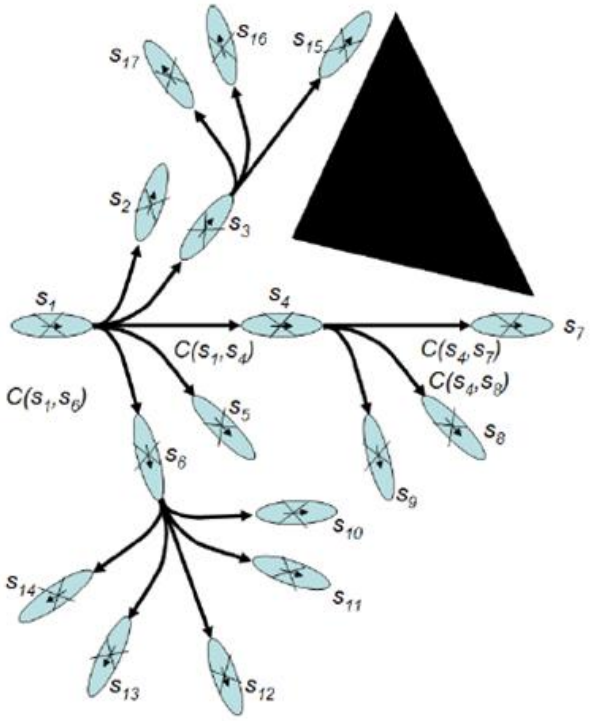

Figura 6.2: Costruzione della mappa tramite SBPL

Innanzitutto l'ambiente viene discretizzato per mezzo di una griglia di celle della stessa dimensione. Poi, come si può notare dalla figura 6.2, date in ingresso tutte le primitive del moto, pre-calcolate per ogni orientazione del robot, si crea la mappa dei possibili punti validi.

Si fa notare che ogni stato in uscita sarà pari al centro della corrispondente cella (x, y, θ); che esso sarà uno stato in cui il robot non è in collisione con l'ambiente che lo circonda e, infine, che si riferirà ad una transizione possibile, cioè che a tutti gli effetti può venir attuata dal robot.

Ogni nuovo stato viene legato agli stati precedenti per mezzo di lati validi.
Si dovrà allora garantire la scelta del percorso meno costoso che congiunge $Start$ a $Goal$. A tal scopo, ad ogni lato viene associato un costo, per esempio il numero di passi che devono essere effettuati dal robot per ricoprire quel tragitto.

Per determinare, ad ogni passo, qual è il miglior percorso da seguire dati i costi imputati ai lati mappati, si usa la ricerca A^*: gli stati vengono espansi secondo la funzione funzione $f(s) = g(s) + h(s)$ con s lo stato corrente.
Come si vede in figura 6.3, $g(s)$ si riferisce al costo del più corto percorso che congiunge lo $Start$ allo stato corrente, esso verrà determinato sommando i costi imputati ad ogni lato secondo i vincoli imposti nell'SBPL; $h(s)$, invece, va a stimare il costo del percorso più corto che congiunge s al $Goal$.

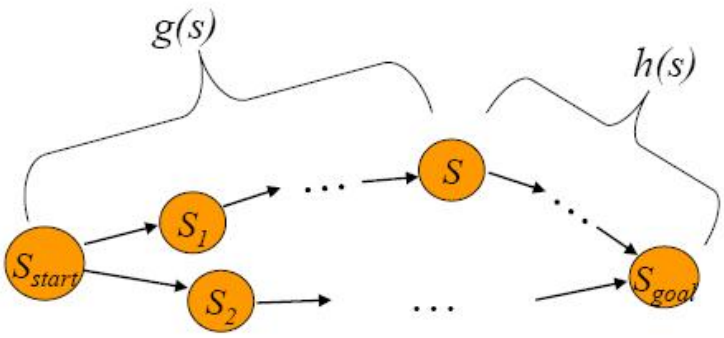

Figura 6.3: Funzionamento di A^*

In questo caso sarà necessario utilizzare una funzione euristica, come la distanza euclidea.

Il costo totale del cammino che connette $Start$ a $Goal$ sarà pari alla somma delle due funzioni, ovviamente si sceglierà il costo minore se si vuol far compiere al robot il percorso ottimo.

CAPITOLO 7

CINEMATICA DI UN CORPO UMANOIDE

Per far realmente camminare il robot lungo il percorso trovato al capitolo precedente, è utile focalizzarsi sulle caratteristiche strutturali che definiscono un corpo di questo tipo e sulle leggi fisiche necessarie per la generazione di movimenti che gli permettano di muoversi nello spazio in condizioni di sicurezza: senza perdere l'equilibrio, senza collidere con l'ambiente che lo circonda e senza auto-collidere, condizioni che potrebbero provocare anche danni ingenti alla sua struttura.

È noto che un robot può essere definito come un sistema di corpi, gli arti, legati ad un certo numero di articolazioni o giunti, lo studio della posizione e dell'orientamento di questi punti è essenziale per poter poi pianificare movimenti che non danneggino la struttura del robot.

La cinematica è quella teoria che analizza la relazione tra la configurazione (posizione ed orientamento) di un segmento e gli angoli delle articolazioni [5]:

- cinematica diretta: fa riferimento al metodo che permette di calcolare la posizione e l'orientamento dei segmenti in funzione dei valori presi dagli angoli delle articolazioni (es.: per calcolare il centro di gravità, conoscere la configurazione attuale dei segmenti, individuare collisioni con l'ambiente). È il calcolo della posizione a partire dagli angoli delle articolazioni;

- cinematica inversa: ricerca degli angoli articolari che corrispondono alla posizione e all'orientazione di un segmento dello spazio (es: se si usa un sistema di visione, esso dà una descrizione della geometria delle scale che il robot deve salire, allora vi sarà il bisogno di una cinematica inversa per determinare la configurazione da adottare per posizionare correttamente i piedi durante la marcia).

Nel caso in analisi, a partire da dei punti in cui devono situarsi i piedi del robot, si va a calcolare il movimento da far fare al corpo di modo che, dato il piede in quel punto, si mantenga la stabilità. È quindi un problema di cinematica inversa: data una posizione si impongono gli angoli che i giunti devono avere per raggiungerla.

Per capire come far muovere correttamente il robot, in questo capitolo si andranno ad esaminare nel dettaglio sia la cinematica diretta sia quella inversa.

Verranno esposti gli algoritmi necessari al calcolo delle posizioni degli arti, ma anche delle orientazioni dei giunti. Si farà infine chiarezza sul metodo scelto per trattare il caso in analisi.

7.1 Creazione di un modello

Una rappresentazione strutturale di un robot umanoide è pari ad una decomposizione del sistema secondo un numero di corpi legati ad un numero di articolazioni.

Si considera un sistema con 12 gradi di libertà che rappresenta due gambe. Si definiscono i nomi e gli ID di ogni segmento di questa figura (vedi figura 7.1).

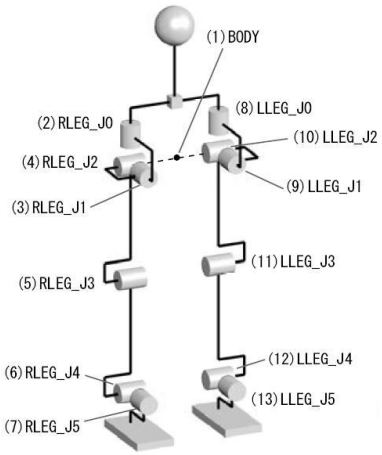

Figura 7.1: Struttura di un robot bipede a 12 DOF

Si definiscono le coordinate locali che permettono di reperire le posizioni e le orientazioni di ogni corpo. Le origini dei sistemi di riferimento locali possono essere posizionate sui rispettivi assi di rotazione. Si va anche a determinare l'orientazione di questi sistemi di modo che essi coincidano con il riferimento del mondo nel momento in cui il robot si trova nella configurazione iniziale (vedi figura 7.2):

$$R_1 = R_2 = ... = R_{13} = E$$

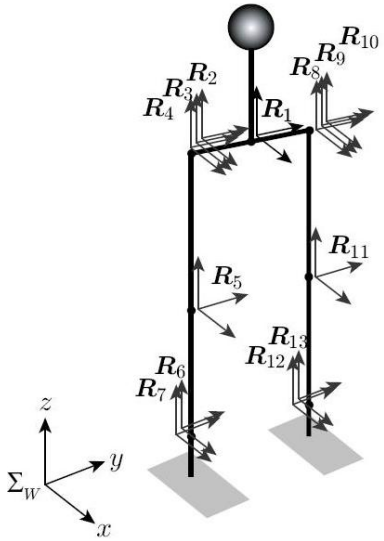

Figura 7.2: Matrice delle rotazioni che descrive l'orientamento relativo di ogni corpo

Il passo successivo consiste nel definire i vettori a_i, che hanno origine negli assi di rotazione associati ad ogni articolazione, e i vettori delle posizioni relative b_j di ogni corpo j (figura 7.3).

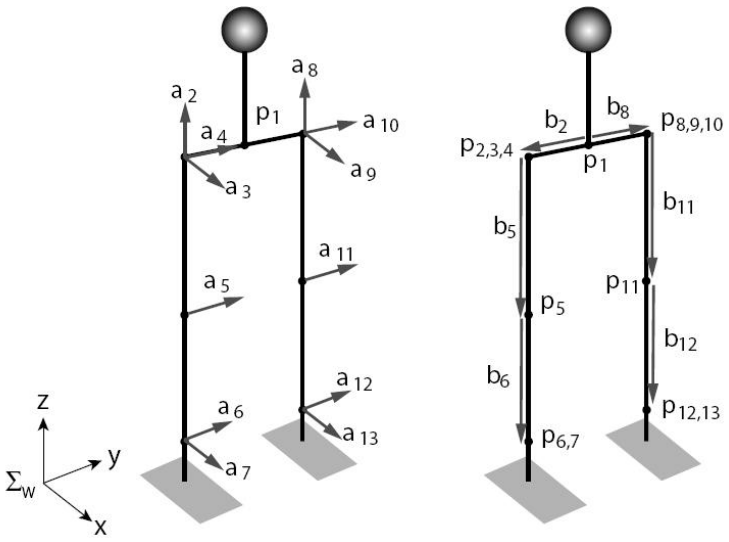

Figura 7.3: a_i e b_j

I vettori che partono dagli assi di rotazione sono dei vettori unitari che precisano la direzione degli assi di rotazione. Sono espressi nelle coordinate del padre del segmento considerato ed il verso positivo di rotazione è il verso trigonometrico indiretto.
Prendendo per esempio l'articolazione del ginocchio come riferimento, i vettori d'asse sono $a_5, a_{11} = \begin{bmatrix}0 & 1 & 0\end{bmatrix}^T$. Girando questa articolazione nel verso positivo, il ginocchio si va a piegare allo stesso modo di quello umano.

Il vettore posizione relativa b_j indica la posizione di un punto di riferimento locale nel riferimento padre. Questo vettore è nullo se i due punti di riferimento si sovrappongono. Un esempio può essere quello delle caviglie in cui $b_7, b_{13} = 0$.

La rappresentazione descritta verrà usata nel calcolo della cinematica diretta, dello Jacobiano delle velocità e della cinematica inversa.

7.2 Cinematica diretta

Come suddetto, la cinematica diretta si riferisce al metodo che permette di calcolare la posizione e l'orientamento dei segmenti in funzione dei valori degli angoli delle articolazioni.
È importante soffermarsi su di essa poiché viene da sempre considerata la base di tutta la simulazione robotica.

Può essere calcolata per mezzo delle regole delle catene di trasformazione omogenee; si comincia allora col calcolare la trasformazione omogenea del segmento di figura 7.4.
Vi è il bisogno di definire un riferimento locale Σ_j la cui origine è situata sull'asse di rotazione dell'articolazione.
Il vettore d'asse, espresso nel riferimento del segmento padre, è a_j e l'origine di Σ_j si trova a distanza b_j dall'asse di riferimento padre.
La variabile articolare viene denotata con q_j e l'orientamento del segmento, quando l'angolo articolare è nullo, vale quanto la matrice identità E.
La trasformazione omogenea, espressa nelle coordinate del segmento padre Σ_i, è:

$$ {}^{i}T_j = \begin{bmatrix} e^{\hat{a}_j q_j} & b_j \\ 000 & 1 \end{bmatrix} \tag{7.1} $$

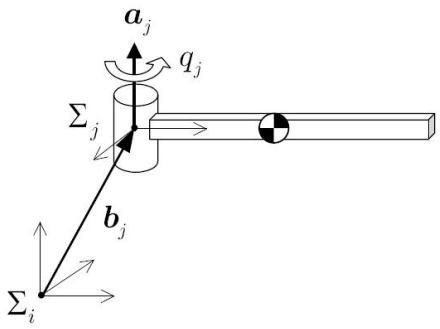

Figura 7.4: Posizione e orientamento di un segmento

dove $\hat{a}_j$ sta ad indicare l'antisimmetrica di a_j.

Si suppone ora che il riferimento Σ_i sia attaccato a un segmento in movimento nel riferimento del mondo (vedi figura 7.5).

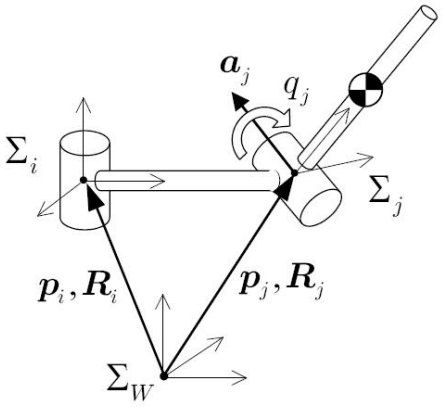

Figura 7.5: Posizione e orientamento di più segmenti

Siano noti la posizione e l'orientamento assoluto (p_i, R_i) del padre.
Allora la trasformazione omogenea Σ_i in rapporto al riferimento fisso del mondo vale

$$T_i = \begin{bmatrix} R_i & p_i \\ 000 & 1 \end{bmatrix} \tag{7.2}$$

A partire dalle regole di calcolo delle trasformazioni in catena si può esprimere la trasformazione omogenea T_j del segmento j in rapporto con il riferimento del mondo:

$$T_j = T_i^i T_j \tag{7.3}$$

Le equazioni (7.1), (7.2) e (7.3) ci permettono di conoscere la posizione assoluta p_j e l'orientazione assoluta R_j di Σ_j:

$$\begin{cases} p_j = p_i + R_i b_j \\ R_j = R_i e^{\hat{a}_j q_j} \end{cases} \tag{7.4}$$

Così si può calcolare la posizione e l'orientamento di tutti i segmenti.

Di seguito si presenta il codice necessario per effettuare il calcolo. In esso *uLINK* è la classe che definisce tutti i link del robot; nello specifico uLINK(i) andrà ad indicare il link di ID i (i = 1 per la base e così via per gli altri), uLINK(i).p indicherà il campo posizione e uLINK(i).R l'orientazione.
Si riporta il codice:

```
function ForwardKinematics(j)
global uLINK

if j == 0 return; end
if j ~= 1
i = uLINK(j).mother;
uLINK(j).p = uLINK(i).R*uLINK(j).b + uLINK(i).p;
uLINK(j).R = uLINK(i).R*Rodrigues(uLINK(j).a, uLINK(j).q);
end
ForwardKinematics(uLINK(j).sister);
ForwardKinematics(uLINK(j).child);
```

7.3 Cinematica inversa

Ci si trova di fronte ad un problema di cinemtica inversa ogniqualvolta si vogliono trovare gli angoli articolari che corrispondono alla posizione e all'orientamento di un segmento nello spazio.

Esistono due metodologie che permettono di risolvere questo tipo di cinematica: approccio analitico o approccio numerico. Di seguito vengono descritti entrambi i metodi con conseguenti vantaggi e svantaggi legati al loro utilizzo.

Soluzione analitica

Ci si concentra sulla gamba destra del modello esposto al paragrafo precedente. Siano le posizioni e le orientazioni di tronco e gamba destra rispettivamente (p_1, R_1) e (p_7, R_7). Per facilitare la comprensione dei calcoli, viene introdotto il vettore D che collega il riferimento del tronco (la base) e l'estremità superiore della gamba (l'anca). Si indica poi con A la lunghezza della coscia e con B quella della tibia (si veda figura 7.6).

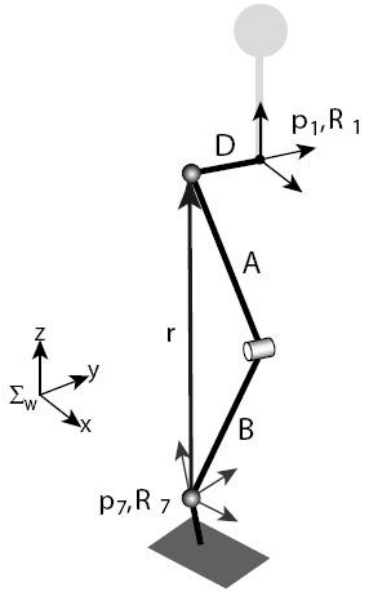

Figura 7.6: Geometria della gamba

Data questa notazione, la posizione assoluta dell'anca viene descritta come

$$p_2 = p_1 + R_1 \begin{bmatrix} 0 \\ D \\ 0 \end{bmatrix} \tag{7.5}$$

Si calcola il vettore posizione dell'anca nelle coordinate delle caviglie:

$$r = R_7^T (p_2 - p_7) \equiv [r_x r_y r_z]^T \tag{7.6}$$

Sia C la distanza tra la caviglia e l'anca

$$C = \sqrt{r_x^2 + r_y^2 + r_z^2} \tag{7.7}$$

si espone dunque la figura 7.7 di modo da mettere in evidenza la possibilità di ottenere il valore dell'angolo del ginocchio q_5 dal triangolo ABC.

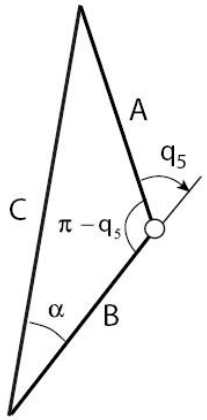

Figura 7.7: Angolo del ginocchio

La *regola del coseno* dice che

$$C^2 = A^2 + B^2 - 2ABcos(\pi - q_5) \tag{7.8}$$

Da cui si può ricavare il valore dell'angolo del ginocchio

$$q_5 = -\cos^{-1}(\frac{A^2 + B^2 - C^2}{2AB}) + \pi \tag{7.9}$$

Sia α l'angolo inferiore del triangolo ABC. La *regola del seno* afferma che

$$\frac{C}{\sin(\pi - q_5)} = \frac{A}{\sin\alpha} \tag{7.10}$$

Da cui si può ricavare il valore di α:

$$\alpha = \sin^{-1}(\frac{A\sin(\pi - q_5)}{C}) \tag{7.11}$$

Ci si concentri ora sulle coordinate locali della caviglia.
Il vettore r descritto in figura 7.8 permette di calcolare le diverse rotazioni possibili a livello della caviglia:

$$\begin{cases} q_7 = atan2(r_y, r_z) \\ q_6 = atan2(r_x, sign(r_z)\sqrt{r_y^2 + r_z^2}) - \alpha \end{cases} \tag{7.12}$$

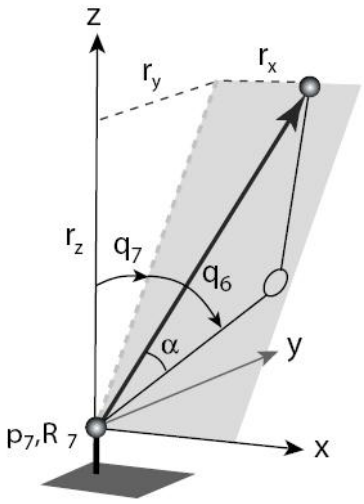

Figura 7.8: Angoli della caviglia

dove la funzione *atan2(x, y)* fornisce l'angolo tra il vettore (x, y) e l'asse z. La funzione *sign(x)*, invece, ritorna +1 se x ha un valore positivo e -1 nel caso contrario.
Non resta che calcolare *roll*, *yaw* e *pitch* dall'alto della gamba. La manipolazione delle equazioni che definiscono ogni valore articolare permette di ottenere questi valori mancanti. Per esempio,

$$R_7 = R_1 R_z(q_2) R_x(q_3) R_y(q_4) R_y(q_5 + q_6) R_x(q_7) \tag{7.13}$$

permette di ottenere

$$R_z(q_2) R_x(q_3) R_y(q_4) = R_1^T R_7 R_x(q_7) R_y(q_5 + q_6) \tag{7.14}$$

Sviluppando i calcoli matriciali per far apparire gli angoli che ci interessano, si ottiene l'ugualianza matriciale seguente:

$$\begin{bmatrix} c_2c_4 - s_2s_3s_4 & -s_2c_3 & c_2s_4 + s_2s_3c_4 \\ s_2c_4 + c_2s_3s_4 & c_2c_3 & s_2s_4 - c_2s_3c_4 \\ -c_3s_4 & s_3 & c_3c_4 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{7.15}$$

dove $c_2 \equiv \cos q_2$ e $s_2 \equiv \sin q_2$. L'identificazione termine a termine fornisce i valori ricercati:

$$\begin{cases} q_2 = atan2(-R_{12}, R_{22}) \\ q_3 = atan2(R_{32}, -R_{12}s_2 + R_{22}c_2) \\ q_4 = atan2(-R_{31}, R_{33}) \end{cases} \tag{7.16}$$

Questa stessa tecnica può essere utilizzata anche per la gamba sinistra, basta cambiare il segno di D, ma vale solo per il modello di robot utilizzato. Nel caso, per esempio, in cui il sistema non possieda tre assi concorrenti all'anca, l'algoritmo è totalmente differente.
Esistono molti algoritmi di cinematica inversa, ma le formule analitiche richiedono in generale troppi calcoli troppo costosi. Per questo motivo di solito viene usato, per la risoluzione della cinematica inversa, non l'approccio analitico, ma quello numerico.

Soluzione numerica

- *Passo 1*: Definire la posizione e l'orientamento (p_B, R_B) della base (tronco);

- *Passo 2*: Definire la posizione e l'orientamento (p^{ref}, R^{ref}) del segmento obiettivo (per esempio il piede destro visto che il nostro scopo è quello di andare ad effettuare una marcia bipede);
- *Passo 3*: Definire il vettore q che contiene gli angoli articolari tra la base e il piede;
- *Passo 4*: Calcolare la cinematica diretta per ottenere la posizione e l'orientamento (p, R) del piede;
- *Passo 5*: Calcolare la differenza tra posizione e orientamento ottenuti e desiderati del piede $(\Delta p, \Delta R) = (p^{ref} - p, R^T R^{ref})$;
- *Passo 6*: Se gli errori $(\Delta p, \Delta R)$ sono abbastanza piccoli fermare i calcoli;
- *Passo 7*: Se invece gli errori $(\Delta p, \Delta R)$ sono troppo grandi, calcolare Δq per ridurre l'errore;
- *Passo 8*: Calcolare $q = q + \Delta q$ e ritornare al passo 3.

Cosa significa $(\Delta p, \Delta R)$ abbastanza piccoli

Uno stato contenente errore nullo viene descritto dalle equazioni:

$$\begin{cases} \Delta p = 0 \\ \Delta R = E \end{cases} \tag{7.17}$$

Usando questo stato come 0, un esempio di funzione che valuta l'errore può essere:

$$err(\Delta p, \Delta R) = \|\Delta p\|^2 + \|\Delta \omega\|^2 \text{ con } \Delta \omega = (\ln R)^v \tag{7.18}$$

Ed i calcoli si possono fermare quando $err(\Delta p, \Delta R) < 10^{-6}$.

Come calcolare concretamente Δq per ridurre l'errore

Viene usato il metodo di Newton-Raphson (rapido e preciso). Esso va a considerare come cambiano posizione e orientazione $(\delta p, \delta \omega)$ nel momento in cui il valore degli angoli viene modificato di δq:

$$\begin{cases} \delta p = X_p(q, \delta q) \\ \delta \omega = X_\omega(q, \delta q) \end{cases} \tag{7.19}$$

X_p e X_ω non sono noti, ma finchè δq è piccolo si può considerare che essi siano descrivibili per mezzo di semplici addizioni e moltiplicazioni.
Se si usano le matrici si ottiene:

$$\begin{bmatrix} \delta p \\ \delta \omega \end{bmatrix} = \begin{bmatrix} J_{11} J_{12} J_{13} J_{14} J_{15} J_{16} \\ J_{21} J_{22} J_{23} J_{24} J_{25} J_{26} \\ J_{31} J_{32} J_{33} J_{34} J_{35} J_{36} \\ J_{41} J_{42} J_{43} J_{44} J_{45} J_{46} \\ J_{51} J_{52} J_{53} J_{54} J_{55} J_{56} \\ J_{61} J_{62} J_{63} J_{64} J_{65} J_{66} \end{bmatrix} \delta q \tag{7.20}$$

con J_{ij} (i, j = 1, ..., 6) constanti che definiscono posizione ed orientazione attuali dei segmenti del robot. Si ricorda che il vettore q possiede sei componenti se la gamba possiede sei segmenti.
Introducendo la matrice J quale contenitore di queste costanti, si ottiene:

$$\begin{bmatrix} \delta p \\ \delta \omega \end{bmatrix} = J \delta q \tag{7.21}$$

La matrice J viene detta *Jacobiana delle velocità*.
Si può così calcolare l'aggiustamento δq da apportare invertendo la matrice J:

$$\delta q = \lambda J^{-1} \begin{bmatrix} \delta p \\ \delta \omega \end{bmatrix} \tag{7.22}$$

Questa è appunto l'equazione che viene usata per calcolare i riaggiustamenti da apportare alle posizioni articolari.
Il parametro $\lambda \in [0,1]$ è un coefficiente usato per rendere stabile il cacolo numerico.

Quanto detto viene riassunto nella funzione seguente:

```
function InverseKinematics(to, target)
global uLINK

lambda = 0.5;
ForwardKinematics(1);
idx = FindRoute(to);
for n = 1:10
j = CalcJacobian(idx);
err = CalcVWerr(Target, uLINK(to));
if norm(err) < 1E-6 return; end;
dq = lambda*(J\err);
for nn = 1:lenght(idx)
j = idx(nn);
uLINK(j).q = uLINK(j).q + dq(nn);
end
ForwardKinematicx(1);
end
```

Programmando in C/C++ il calcolo viene fatto in 0.3 ms. Questa rapidità permette di usarlo su robot complessi in tempi reali.
Questa funzione usa *FindRoute()* e *CalcVWerr()*, funzioni che vengono ora esplicitate.

Come si vede dal codice, la prima ritorna i segmenti da attraversare per andare dal torso al piede:

```
function idx = FindRoute(to)
global uLINK

i = uLINK(to).mother;
if i == 1
idx = [to];
else
idx = [findRoute(i) to];
end
```

La seconda calcola invece le differenze in posizione e orientazione:

```
function err = CalcVWerr(Cref, Cnow)

perr = Cref.p - Cnow.p;
Rerr = Cnow.R^-1*Cref.R;
werr = Cnow.R*rot2omega(Rerr);
err = [perr; werr];
```

Inoltre si utilizza *CalcJacobian()* per il calcolo della matrice jacobiana, funzione che viene descritta nel paragrafo successivo.

Jacobiana delle velocità

La Jacobiana è l'elemento che lega le piccole variazioni articolari ai loro effetti nello spazio cartesiano. Attraverso questa matrice è possibile accedere ai valori delle coppie articolari necessarie per generare uno sforzo esterno di mani e piedi.
Correntemente il metodo viene molto usato per comandare i robot.

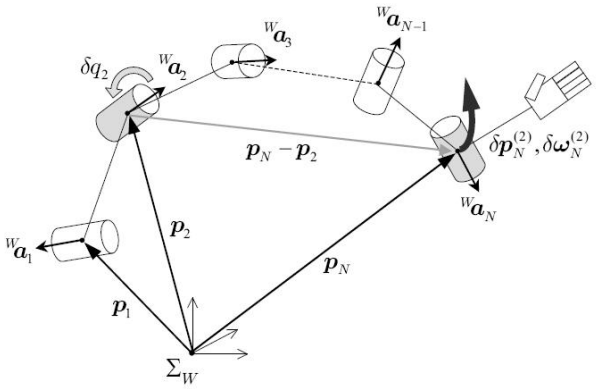

Figura 7.9: Calcolo della Jacobiana delle velocità

Si va a spiegare la procedura di calcolo della matrice Jacobiana.
Sia data una catena di N corpi flottanti nello spazio (vedi figura 7.9). Siano questi corpi numerati dalla base (corpo 1) all'estremità (corpo N). Sia l'effettore terminale del robot (la mano o il piede) il corpo N-esimo. Si supponga infine che la cinematica diretta sia già stata calcolata e che la posizione e l'orientamento di ogni corpo sia noto (il j-esimo corpo è rappresentato da p_j, R_j).

In questa catena di corpi si immobilizzano tutte le articolazioni tranne la seconda, che invece si fa variare di un piccolo angolo δq_2. La posizione dell'effettore finale si trova modificata di $\delta p_N^{(2)}$ e la sua orientazione di $\delta\omega_N^{(2)}$. Questi due valori possono essere calcolati come:

$$\begin{cases} \delta p_N^{(2)} = W_{a_2} \times (p_N - p_2)\delta q_2 \\ \delta\omega_N^{(2)} = W_{a_2}\delta q_2 \end{cases} \tag{7.23}$$

con W_{a_2} il vettore unitario portato attraverso l'asse della seconda articolazione espresso nel riferimento del mondo: $W_{a_2} = R_2 a_2$.
Si compie la stessa procedura su tutti i corpi da 1 a N e la somma di tutti questi valori permette di ottenere gli spostamenti risultanti delle piccole modifiche per tutte le articolazioni:

$$\begin{cases} \delta p_N = \sum_{j=1}^{N} \delta p_N^{(j)} \\ \delta\omega_N^{(2)} = \sum_{j=1}^{N} \delta\omega_N^{(j)} \end{cases} \tag{7.24}$$

Si può riscrivere in forma matriciale l'equazione appena esposta e si ottiene il risultato seguente:

$$\begin{bmatrix} \delta p_N \\ \delta\omega_N \end{bmatrix} = \begin{bmatrix} W_{a_1} \times (p_N - p_1) & \cdots & W_{a_{N-1}} \times (p_N - p_{N-1}) & 0 \\ W_{a_1} & \cdots & W_{a_{N-1}} & W_{a_N} \end{bmatrix} \begin{bmatrix} \delta q_1 \\ \delta q_2 \\ \vdots \\ \delta q_N \end{bmatrix} \tag{7.25}$$

In altri termini la Jacobiana J può essere scritta nella forma

$$J \equiv \begin{bmatrix} W_{a_1} \times (p_N - p_1) & \cdots & W_{a_{N-1}} \times (p_N - p_{N-1}) & 0 \\ W_{a_1} & \cdots & W_{a_{N-1}} & W_{a_N} \end{bmatrix} \tag{7.26}$$

Presentiamo il codice necessario per calcolare la matrice:

```
function J = CalcJacobian(idx)
global uLINK

jsize = length(idx);
target = uLINK(idx(end)).p;    % posizione assoluta del segmento obbiettivo
J = zeros(6, jsize);
for n = 1:jsize
j = idx(n);
a = uLINK(j).R*uLINK(j).a;  %vettore d'asse (coordinate assolute)
J(:,n) = [cross(a, target - uLINK(j).p); a];
end
```

Calcolo delle velocità

Si possono calcolare i valori articolari delle velocità lineari ed angolari dei segmenti usando i tool che sono stati ideati.
Si suppone che le posizioni e le orientazioni dei corpi siano già state calcolate.
Sia nota la velocità (v_B, ω_B) della base. Il nostro obbiettivo è quello di determinare le velocità (v_t, ω_t) del segmento obbiettivo. Per far ciò bisogna conoscere la forma delle velocità angolari necessarie per generare la velocità desiderata dall'organo terminale, conoscendo la velocità di base.
Se si interpretano le piccole variazioni dell'equazione (7.10) come delle velocità, si ottiene l'espressione

$$\dot{q} = J^{-1} \begin{bmatrix} v_d \\ \omega_d \end{bmatrix} \tag{7.27}$$

dove $\dot{q}$ è il vettore che raggruppa le velocità articolari di tutti i corpi tra la base e l'obbiettivo. (v_d, ω_d) raggruppa le velocità relative del segmento obbiettivo in riferimento alla base.
Si può vedere che:

$$\begin{cases} v_t = v_B + v_d + \omega_B \times (p_t - p_B) \\ \omega_t = \omega_B + \omega_d \end{cases} \tag{7.28}$$

I vettori p_B e p_t disegnano le posizioni rispettivamente della base e dell'obbiettivo.
A partire da questa equazione si trova:

$$\begin{cases} v_d = v_t - v_B - \omega_B \times (p_t - p_B) \\ \omega_d = \omega_t - \omega_B \end{cases} \tag{7.29}$$

Diventa così possibile esprimere le velocità articolari $\dot{q}$ a partire dalle equazioni (7.15) e (7.17).
Adesso abbiamo i vettori delle velocità articolari e, a partire da questi valori, si calcolano le velocità lineari e angolari di tutti i segmenti che compongono il robot. Conoscere le sue velocità è necessario per calcolare lo ZMP del prossimo capitolo.
Nello stesso modo in cui sono state calcolate la posizione e l'orientamento dei corpi, si possono calcolare le velocità relative tra due corpi adiacenti.
Si parte dal fatto che le velocità lineari e angolari del segmento padre i siano note e che si possa accedere alla velocità articolare $\dot{q}_j$ del corpo j dal vettore $\dot{q}$.
Le velocità lineari e angolari del segmento j si ottengono per mezzo delle equazioni seguenti:

$$\begin{cases} v_j = v_i + \omega_i \times R_i b_j \\ \omega_j = \omega_i + R_i a_j \dot{q}_j \end{cases} \tag{7.30}$$

Andiamo anche in questo caso a riportare il codice necessario al calcolo della velocità di tutti i corpi:

```
function ForwardVelocity(j)
global uLINK
```

```
if j == 0 return; end
if J ~= 1
i = uLINK(j).mother;
uLINK(j).v = uLINK(i).v + cross(uLINK(i).w, uLINK(i).R * uLINK(i).b);
uLINK(j).w = uLINK(i).w + uLINK(j).R*(uLINK(j.a)*uLINK(j).dq);
end
ForwardVelocity(uLINK(j).sister);
ForwardVelocity(uLINK(j).child);
```

CAPITOLO 8

ZMP

Nel capitolo precedente è stato trattato il problema di cinematica, diretta e inversa, di un robot; ma nel momento in cui un robot è in movimento bisogna tener conto di altri fenomeni provenienti dalla dinamica del robot. Questi fenomeni possono essere trascurati se i movimenti realizzati sono sufficientemente piccoli; in questo caso si parla di spostamenti quasi statici e il movimento è noto come una successione di configurazioni fisse.
Non si può invece ignorare la dinamica del sistema nel momento in cui si parla di marcia.
Questo capitolo ha per obiettivo quello di esporre una quantità fisica importante in robotica umanoide: lo ZMP.
Visto che un robot umanoide non è fissato al suolo bisogna andare a studiare se è o meno possibile mantenere il contatto tre le suole dei suoi piedi ed il terreno. Nel secondo caso infatti il robot perderà l'equilibrio e cadrà. Questo studio viene fatto attraverso lo ZMP.

8.1 Definizione

Lo ZMP, o *Zero Moment Point*, è stato introdotto nel campo robotico nel 1972 da Miomir Vukobratović e Yu. Stepanenko.
Si prenda una forza distribuita sul piede, caratterizzante il contatto tra il piede e il suolo, come quella di figura 8.1. Visto che il carico ripartito è lo stesso su tutta la superficie, allora questa forza si può scrivere sotto forma di una forza risultante R il cui punto di applicazione è situato dentro i limiti della superficie del piede. Questo punto è lo ZMP.

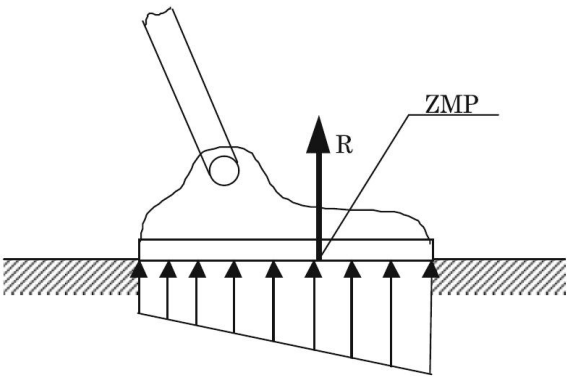

Figura 8.1: Definizione dello ZMP

Lo ZMP può quindi essere definito come il centro di pressione.
Nello specifico lo ZMP indica il punto rispetto al quale la forza di reazione dinamica al contatto del piede con il suolo non produce alcuna componente orizzontale del momento risultante della forza di reazione del suolo; per esempio il punto in cui il totale delle forze di gravità e d'inerzia è zero.

8.2 ZMP e poligono di sostentamento

Un concetto importante per la definizione dello ZMP è il poligono di sostentamento.
Se si tende un elastico attorno ai piedi del robot, a livello della superficie di contatto, si ottiene una superficie detta poligono di sostentamento. Essa include tutti i punti di contatto.
Si sottolinea il fatto che lo ZMP si trova sempre all'interno del poligono di sostentamento.

Importante è anche capire la differenza tra ZMP e CoM, o centro di massa, punto nello spazio dove si concentra l'intera massa del corpo, la posizione media della distribuzione di massa.
A tal scopo viene introdotta la figura 8.2 e si ricorda che la proiezione al suolo del CoM è il punto situato nell'intersezione tra la linea di gravità passante per il centro di massa ed il suolo.
Si nota che in caso di immobilità ZMP e proiezione del CoM coincidono altrimenti si dislocano in punti differenti.

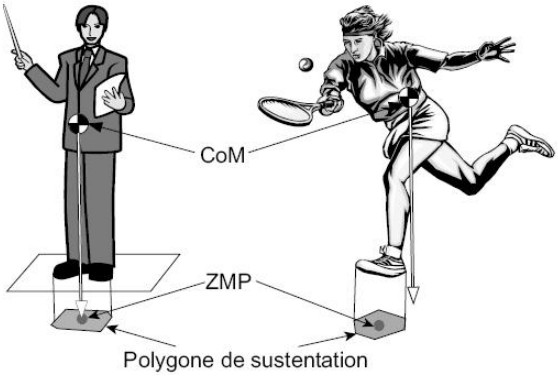

Figura 8.2: Differenza tra ZMP e CoM

Detto ciò nell'implementazione della camminata ci si concentrerà sul mantenere lo ZMP all'interno del poligono di sostentamento, solo in questo modo infatti il robot potrà marciare senza perdere l'equilibrio e cadere.

Si va quindi ad esporre come calcolare questo ZMP, per farlo ci si sofferma dapprima sull'analisi delle forze che caratterizzano un robot in movimento.

8.3 Movimenti di un robot e forze di reazione

Si considera un robot umanoide di struttura qualunque e si suppone che esso possa venir identificato con precisione nel suo ambiente. Si definiscono:

- *massa*: massa totale del robot in kilogrammi, denotata con M;
- *centro di massa*: posizione del centro di massa del robot in metri, denotata con $c \equiv \begin{bmatrix} x & y & z \end{bmatrix}^T$;
- *quantità di movimento*: misura dei movimenti del robot in traslazioni in newton-secondi, denotata con $P \equiv \begin{bmatrix} P_x & P_y & P_z \end{bmatrix}^T$;
- *momento cinetico*: misura dei movimenti del robot in rotazioni attorno all'origine in newton-metri-secondi, denotata con $L \equiv \begin{bmatrix} L_x & L_y & L_z \end{bmatrix}^T$;

Si va quindi a precisare ciò che rappresentano la quantità di movimento P e il momento cinetico L. La *dinamica del sistema* permette di stabilire le relazioni tra i diversi parametri fisici qui sopra definiti. Queste relazioni possono essere raggruppate nelle tre equazioni seguenti:

$$\dot{c} = P/M \tag{8.1}$$
$$\dot{P} = \mathbf{f}_{all} \tag{8.2}$$
$$\dot{L} = \tau_{all} \tag{8.3}$$

Si va quindi a precisare il senso di ciascuna di questa quantità fisiche.

Dinamica di un movimento di traslazione

La prima equazione fornisce la relazione tra la velocità del centro di massa e la quantità di movimento nel caso di un movimento di traslazione:

$$\dot{c} = P/M \tag{8.4}$$

Questa equazione mostra che la quantità di movimento è uguale al prodotto tra la massa totale e la velocità del centro di massa.
La seconda equazione si riferisce sempre al caso di traslazione ed indica che la quantità di movimento dipende dalle forze esterne:

$$\dot{P} = \mathbf{f}_{all} \tag{8.5}$$

dove $\mathbf{f}_{all}$ definisce la risultante delle forze esterne applicate al robot.
Si considerano le forze esterne che vengono applicate al robot. Agisce una forza di gravità su ciascuno dei componenti del robot; la somma di queste forze equivale alla forza M**g** applicata al centro di massa **c** del robot. **g** può essere definito come il vettore dell'accelerazione gravitazionale, vettore che si traduce nell'attrazione che subiscono i corpi sul pianeta; esso varrà $\begin{bmatrix}0 & 0 & -9.8\end{bmatrix}^T$ sulla Terra.
Dal punto di vista che le forze gravitazionali non dipendono dal movimento effettuato dal robot, esse vengono generalmente trattate separatamente da tutte le altre forze esterne:

$$\mathbf{f}_{all} = M\mathbf{g} + \mathbf{f} \tag{8.6}$$

dove **f** raggruppa tutte le forze esterne agenti sul robot, salvo la gravità, cioè tutte le forze di contatto con l'ambiente. Nel nostro caso, si tratta delle forze di reazione del suolo. Si può quindi scrivere l'equazione del movimento in traslazione come

$$\dot{P} = M\mathbf{g} + \mathbf{f} \tag{8.7}$$

Quando un robot è immobile in piedi, la variazione di quantità del movimento è nulla e le forze di gravità si equilibriano con le forze di reazione del suolo. Se le forze di reazione scompaiono, la quantità di movimento aumenta rapidamente verso il basso a causa della forza di gravità; è il caso della *caduta libera.*

Dinamica di un movimento di rotazione

La terza equazione traduce un movimento di rotazione:

$$\dot{L} = \tau_{all} \tag{8.8}$$

Questa equazione indica che il momento cinetico dipende dal momento risultante delle forze esterne τ_{all}.
Tra i momenti delle forze esterne applicate al robot, quello derivante dalle forze gravitazionali è dato dalla relazione

$$\tau_g = c \times Mg \tag{8.9}$$

Sia τ il momento delle forze esterne tolte quelle dovute alla gravità. Il momento risultante delle forze esterne applicate al robot vale

$$\tau_{all} = c \times Mg + \tau \tag{8.10}$$

Quindi, l'equazione di un movimento di rotazione attorno all'origine può essere scritta come

$$\dot{L} = c \times Mg + \tau \tag{8.11}$$

Nel caso in analisi τ sarà il momento di reazione del suolo applicato al livello del suolo stesso.
Dato un robot immobile, questo momento si dovrà equilibrare con quello generato dalla forza peso. Nel caso in cui questo equilibrio non avvenga, il momento di reazione del suolo aumenta rapidamente: è il caso della *caduta*.

8.4 Calcolo dello ZMP basato sui movimenti del robot

Secondo quanto descritto nel paragrafo precedente, si può procedere al calcolo della posizione dello ZMP in funzione dei movimenti effettuati dal robot (si veda figura 8.3).

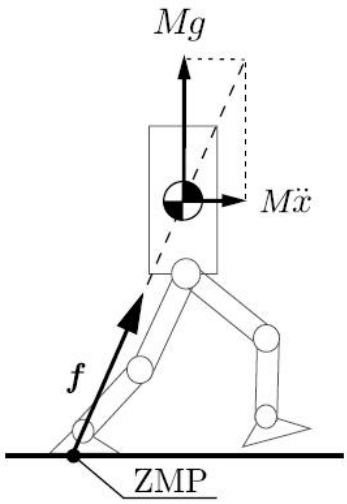

Figura 8.3: Posizione dello ZMP

A tal scopo si ricorda che le forze di reazione del suolo possono essere rappresentate al ZMP, p, per mezzo di una forza risultante equivalente, f, e un momento risultante, τ_p, attorno alla linea verticale passante per lo ZMP. Si può calcolare il momento risultante τ espresso all'origine del sistema di coordinate di riferimento:

$$\tau = p \times f + \tau_p \tag{8.12}$$

Si vanno poi ad utilizzare le relazioni esistenti tra le forze di reazione del suolo e la quantità di movimento del robot da una parte, e tra il momento risultante delle forze di reazione del suolo e il momento cinetico dall'altra:

$$\dot{P} = Mg + f \tag{8.13}$$

$$\dot{L} = c \times Mg + \tau \tag{8.14}$$

Manipolando queste tre equazioni si ottiene l'espressione di τ_p in funzione delle componenti cinetiche e statiche che caratterizzano i movimenti del robot:

$$\tau_p = \dot{L} - c \times Mg + (\dot{P} - Mg) \times p \tag{8.15}$$

τ_p è un vettore a tre componenti in cui le prime due sono nulle: le componenti del momento risultante dovuto alle forze di reazione del suolo, espresso allo ZMP, sono nulle sul piano orizzontale.
In modo più concreto, queste componenti si scrivono come:

$$\tau_{px} = \dot{L}_x + Mgy + \dot{P}_y p_z - (\dot{P}_z + Mg) p_y = 0 \tag{8.16}$$

$$\tau_{py} = \dot{L}_y + Mgx + \dot{P}_x p_z - (\dot{P}_z + Mg) p_x = 0 \tag{8.17}$$

con

$$P = \begin{bmatrix} P_x & P_y & P_z \end{bmatrix}^T \tag{8.18}$$

$$L = \begin{bmatrix} L_x & L_y & L_z \end{bmatrix}^T \tag{8.19}$$

$$c = \begin{bmatrix} x & y & z \end{bmatrix}^T \tag{8.20}$$

$$g = \begin{bmatrix} 0 & 0 & -g \end{bmatrix}^T \tag{8.21}$$

La risoluzione di questo sistema di equazioni permette di ricavare i valori di p_x e p_y, che descrivono la posizione dello ZMP sul piano orizzontale:

$$p_x = \frac{Mgx + p_z \dot{P}_x - \dot{L}_y}{Mg + \dot{P}_z} \tag{8.22}$$

$$p_y = \frac{Mgy + p_z \dot{P}_y + \dot{L}_x}{Mg + \dot{P}_z} \tag{8.23}$$

dove p_z definisce l'altezza dal suolo. Nel caso di movimento su suolo piano si avrà allora $p_z = 0$.

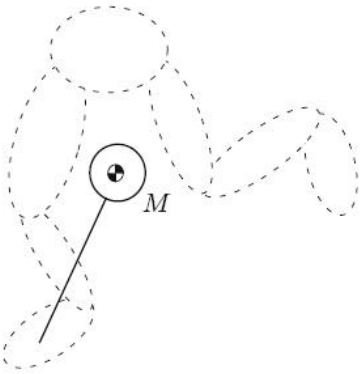

Figura 8.4: Approssimazione ad un'unica massa puntiforme

Lo ZMP può essere calcolato anche facendo ricorso a delle approssimazioni, in particolare andando a rappresentare il robot in tutto il suo insieme come un'unica massa puntiforme (si veda figura 8.4). In questo caso le espressioni della quantità di movimento e del momento cinetico all'origine sono:

$$P = M\dot{c} \tag{8.24}$$

$$L = C \times M\dot{c} \tag{8.25}$$

Si citano le corrisponendenti derivate:

$$\begin{bmatrix} \dot{P}_x \\ \dot{P}_y \\ \dot{P}_z \end{bmatrix} = \begin{bmatrix} M\ddot{x} \\ M\ddot{y} \\ M\ddot{z} \end{bmatrix} \tag{8.26}$$

$$\begin{bmatrix} \dot{L}_x \\ \dot{L}_y \\ \dot{L}_z \end{bmatrix} = \begin{bmatrix} M(y\ddot{z} - z\ddot{y}) \\ M(z\ddot{x} - x\ddot{z}) \\ M(x\ddot{y} - y\ddot{x}) \end{bmatrix} \tag{8.27}$$

Si riportano i valori trovati nelle equazioni (8.11) e (8.12) e si ottiene così la posizione dello ZMP:

$$p_x = x - \frac{(z - p_z)\ddot{x}}{\ddot{z} + g} \tag{8.28}$$

$$p_y = y - \frac{(z - p_z)\ddot{y}}{\ddot{z} + g} \tag{8.29}$$

Si ricorda che, come definito precedentemente, lo ZMP è definito sotto la direzione delle forze di reazione del suolo e le forze inerziali. I valori di queste ultime si ottengono ponendo $\ddot{z} = 0$ e $p_z = 0$:

$$p_x = x - \frac{z}{g}\ddot{x} \tag{8.30}$$

$$p_y = y - \frac{z}{g}\ddot{y} \tag{8.31}$$

Saranno queste le equazioni che verranno utilizzate per generare la marcia bipede.

8.5 Calcolo della posizione del CoM

Come detto all'inizio del capitolo, lo ZMP si trova generalmente in una posizione diversa da quella del CoM.
Per ottenere una marcia priva di cadute, è utile dislocare correttamente non solo lo ZMP ma anche il CoM. Infatti, affinché un robot non si sbilanci, non solo è necessario mantenere lo ZMP all'interno del poligono di sostentamento, ma si ritiene oppurtuno mantenere il CoM sempre alla stessa altezza durante l'esecuzione dei movimenti. In questo modo si assicura che non avvengano cadute in avanti o all'indietro.

Si propone un metodo per calcolare la posizione del centro di massa del robot.
Conoscendo la posizione e l'orientamento di ciascuno dei corpi:

- *Passo 1*: si definisce il CoM di ogni corpo nelle coordinate di riferimento;
- *Passo 2*: si calcola il momento risultante, espresso all'origine, generato dalla massa di ogni corpo;
- *Passo 3*: La posizione del centro di massa del sistema si ottiene andando a dividere il momento risultante per la massa totale del robot.

Nel dettaglio, se la posizione $\bar{c}_j$ del centro di massa di ogni corpo j è nota nel riferimento locale, la sua espressione nelle coordinate di riferimento è pari a:

$$c_j = p_j + R_j \bar{c}_j \tag{8.32}$$

dove (p_j, R_j) definisce la posizione del centro e l'orientamento del riferimento legato al j-esimo corpo nel sistema di riferimento. La posizione del centro di massa globale del robot si ottiene allora nel sistema di riferimento dividendo il momento risultante espresso all'origine per la massa totale, cioè

$$c = \sum_{j=1}^{N} m_j c_j / M \tag{8.33}$$

Si presenta il codice necessario per calcolare il momento c_j del j-esimo corpo, espresso all'origine del sistema di riferimento del robot:

```
function mc = calcMC(j)
global uLINK

if j == 0
mc = 0;
else
mc = uLINK(j).m*(uLINK(j).p + uLINK(j).R + uLINK(j).c);
mc = mc + calcMC(uLINK(j).brother) + calcMC(uLINK(j).child);
end
```

Così facendo è possibile determinare la posizione del centro di massa del robot c nelle coordinate di riferimento:

```
function com = calcCoM()
global uLINK

M = TotalMass(1);
MC = calcMC(1);
com = MC/M;
```

Come nel codice presentato al capitolo precedente, si usa la classe uLINK per rappresentare tutti i link che costituiscono il robot.
Si ricorda infatti che si sta supponendo che la struttura robotica sia stata suddivisa in segmenti, a ciascuno dei quali è stato associato un ID. Inoltre si sottilinea l'utilizzo di una struttura gerarchica particolare, detta *struttura arborescente*. I corpi costituenti l'umanoide vengono infatti organizzati secondo una struttura ad albero ed avranno genitori, fratelli e figli.
Andando allora ad analizzare la notazione utilizzata all'interno del codice, si avrà:

- uLINK(j).brother: ID del fratello del corpo avente ID j;
- uLINK(j).child: ID del figlio del corpo avente ID j;
- uLINK(j).m: massa del corpo avente ID j;
- uLINK(j).p: la sua posizione assoluta;
- uLINK(j).R: la sua orientazione assoluta;
- uLINK(j).c: il suo centro di gravità espresso nelle coordinate locali;

Ora che sono state approfondite le metodologie che permettono il calcolo della posizione di ZMP e CoM, e che è stata analizzata in maniera esaustiva la cinematica di un corpo a molti gradi di libertà, è possibile proseguire il lavoro verso la vera e propria realizzazione della marcia bipede.

CAPITOLO 9

LA MARCIA BIPEDE

Marciare significa spostarsi per mezzo di movimenti successivi di gambe e piedi senza lasciare il suolo.
In robotica si possono distinguere due tipi di marcia: quasi statica e dinamica.
Nel caso quasi-statico la proiezione del CoM sul suolo rimane permanentemente nel poligono di sostentamento. Il robot resta dunque in equilibrio se si ferma bruscamente.
Nel caso dinamico la proiezione del CoM sul suolo può trovarsi al di fuori del poligono di sostentamento. Questo si traduce nella stessa sequenza di movimenti compiuti dagli uomini nell'effettuare una camminata. Il robot non ha alcuna possibilità di conservare il suo equilibrio se si ferma bruscamente, per esempio appena prima di cambiare il piede atto al supporto.
Un robot umanoide dovrà compiere della marcia dinamica per simulare il comportamento umano.
In questo capitolo si va proprio a descrivere un metodo per generare un modello di marcia dinamica per un robot bipede umanoide; si delineano cioè tutte le traiettorie articolati, definite su intervalli di tempo che si succedono, necessarie per far camminare il robot da un punto di partenza ad uno obiettivo.

9.1 LIP-3D

Si ricorda che, affinché gli spostamenti vengano effettuati in condizioni di equilibrio, è necessario mantenere lo ZMP all'interno del poligono di sostentamento.
Inoltre, è conveniente mantenere il CoM sempre alla stessa altezza. Facendogli compiere una traiettoria rettilinea si assicura, infatti, l'assenza di sblilanciamenti.

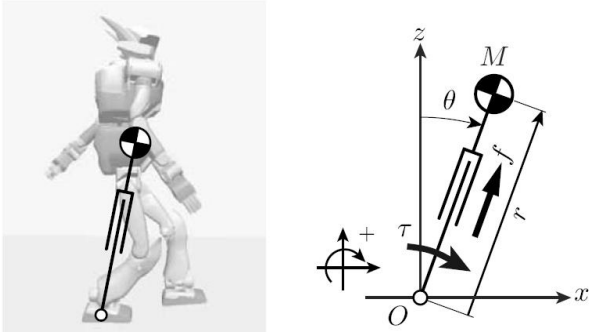

Figura 9.1: Modello 2D di un pendolo inverso

Al fine di raggiungere questi obiettivi, il robot bipede viene modellato, nello spazio tridimensionale, come un pendolo inverso a tre dimensioni: LIP-3D (LIP: *Linear Inverted Pendulum*). Si veda figura 9.1, per una descrizione del pendolo in due dimensioni, e figura 9.2, per la visualizzazione del modello a tre dimensioni.

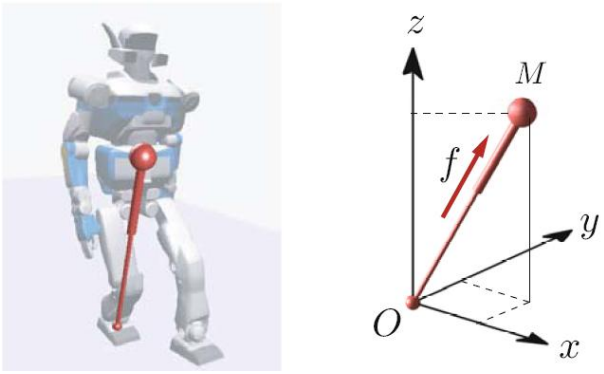

Figura 9.2: LIP-3D di un robot in marcia

Nel dettaglio, il robot viene rappresentato come una massa puntiforme: l'insieme delle masse delle diverse parti che lo costituiscono vengono ridotte al suo CoM. Si suppone poi che il robot possieda gambe senza massa, che stabiliscono la connessione tra la massa puntiforme ed il suolo per mezzo di un nesso avente forma arrotondata.
Il pendolo può girare liberamente attorno al suo punto di supporto e la lunghezza della gamba può venir modificata per mezzo della forza di propulsione f.
Esprimendo f secondo le tre componenti x, y e z:

$$f_x = (x/r)f \tag{9.1}$$
$$f_y = (y/r)f \tag{9.2}$$
$$f_z = (z/r)f \tag{9.3}$$

dove r rappresenta la distranza tra il punto di supporto e il centro di massa (si veda figura 9.3).

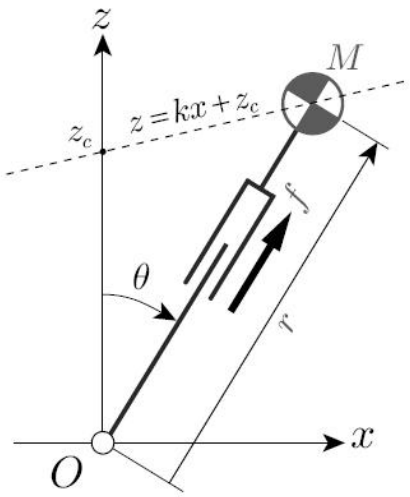

Figura 9.3: Il centro di massa viene controllato per mezzo della forza f

Le sole forze agenti sul centro di massa sono la forza di propulsione e la forza di gravità. Si possono quindi stabilire le equazioni dei movimenti che governano gli spostamenti del CoM:

$$M\ddot{x} = (x/r)f \tag{9.4}$$
$$M\ddot{y} = (y/r)f \tag{9.5}$$
$$M\ddot{z} = (z/r)f - Mg \tag{9.6}$$

Si dovrà stabilire un vincolo legato alla posizione del CoM durante il moto di modo che, durante il moto, si mantenga equilibrio.
Ricordando che stiamo lavorando su di un ambiente tridimensionale, il vincolo sarà pari ad un piano:

$$z = k_x x + k_y y + z_c \tag{9.7}$$

dove k_x e k_y caratterizzano la pendenza, mentre z_c è l'altezza alla quale il piano taglia l'asse z (figura 9.4).

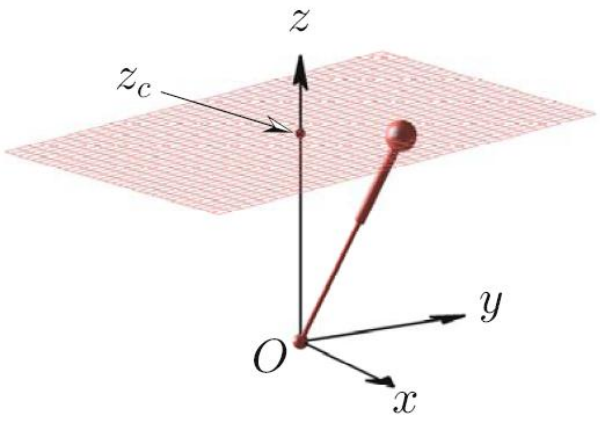

Figura 9.4: Piano posto come vincolo

Al fine di assicurare che il movimento del CoM resti nel piano delineato, l'accelerazione del CoM deve restare ortogonale alla normale del piano:

$$\begin{bmatrix} f(\frac{x}{r}) & f(\frac{y}{r}) & f(\frac{z}{r}) - Mg \end{bmatrix} \begin{bmatrix} -k_x \\ -k_y \\ 1 \end{bmatrix} = 0 \tag{9.8}$$

Risolvendo questa equazione vettoriale si riesce a ricavare f che, una volta sostituita l'equazione (9.7), sarà pari a:

$$f = \frac{Mgr}{z_c} \tag{9.9}$$

Si ricava che il centro di massa si sposta lungo il piano posto come vincolo applicando una forza propulsiva f proporzionale alla lunghezza r della gamba.
A partire dalle equazioni (9.4) e (9.5), e sostituendo in esse il valore della forza di propulsione trovata, si ottieme la dinamica orizzontale del CoM:

$$\ddot{x} = \frac{g}{z_c} x \tag{9.10}$$
$$\ddot{y} = \frac{g}{z_c} y \tag{9.11}$$

Questo è un sistema di equazioni lineari aventi z_c come unico parametro.
Si nota come i parametri d'inclinazione k_x e k_y del piano non influenzino lo spostamento orizzontale del CoM.

In particolare, visto che l'altezza z_c del pendolo inverso rimane costante nel corso del compimento dei movimenti, l'equazione (9.10) origina:

$$x(t) = x(0)\cosh(t/T_c) + T_c\dot{x}(0)\sinh(t/T_c) \tag{9.12}$$

$$\dot{x}(t) = x(0)/T_c\sinh(t/T_c) + \dot{x}(0)\cosh(t/T_c) \tag{9.13}$$

dove T_c è una costante temporale che dipende solo dall'altezza del CoM e dall'accelerazione di gravità:

$$T_c \equiv \sqrt{\frac{z_c}{g}} \tag{9.14}$$

$x(0)$ e $\dot{x}(0)$ sono posizione e velocità iniziali. Questi valori formano le condizioni iniziali.

9.2 Generazione della traiettoria di marcia

Data la descrizione di un LIP-3D e della dinamica orizzontale che caratterizza il centro di massa, si può procedere con la generazione delle traiettorie di marcia.

Si ricorda che una traiettoria di marcia viene definita per mezzo di porzioni tra loro concatenate, dette primitive di marcia. Una primitiva è infatti un elemento base che viene usato per costruzioni complesse.

Nella defizione sarà innanzittutto necessario, di passo in passo, cambiare la gamba di supporto.
Visto poi che si lavora in tre dimensioni, sarà importante garantire che il cambio di supporto sia simultaneo nelle direzioni x e y.
Inoltre, nella marcia generata, si considera una lunghezza di passo costante e si ipotizza che T_{sup} sia il tempo di supporto di ogni passo.

Allora una traiettoria primitiva sarà una porzione d'iperbole, come quella di figura 9.5, simmetrica rispetto l'asse y e definita all'interno dell'intervallo di tempo $[0\ T_{sup}]$.

Siano dati T_{sup} e z_c, una primitiva di marcia viene determinata unicamente per mezzo delle sue coordinate estreme $(\bar{x}, \bar{y})$. In questo modo si può calcolare la velocità finale $(\bar{v}_x, \bar{v}_y)$.

Si sottolinea il fatto che le primitive di marcia fanno sempre riferimento a posizioni e velocità del CoM.

Se la condizione iniziale lungo l'asse x è data da $(-\bar{x}, \bar{v}_x)$ e la posizione finale è $\bar{x}$, allora, a partire dalla soluzione analitica del LIP-3D, si trova:

$$\bar{x} = -\bar{x}C + T_c\,\bar{v}_x S \tag{9.15}$$

dove

$$T_c \equiv \sqrt{\frac{z_c}{g}};\ C \equiv \cosh\frac{T_{sup}}{T};\ S \equiv \sinh\frac{T_{sup}}{T} \tag{9.16}$$

In questo modo si ottiene la velocità finale $\bar{v}_x$ lungo l'asse x:

$$\bar{v}_x = \bar{x}(C+1)/(T_c S) \tag{9.17}$$

A partire poi dalla condizione iniziale $(\bar{y}, -\bar{v}_y)$ lungo l'asse y e volendo raggiungere la posizione finale $\bar{y}$, si trova:

$$\begin{cases} \bar{y} = \bar{y}C + T_c(-\bar{v}_y)S \\ \bar{v}_y = \bar{y}(C-1)/(T_c S) \end{cases} \tag{9.18}$$

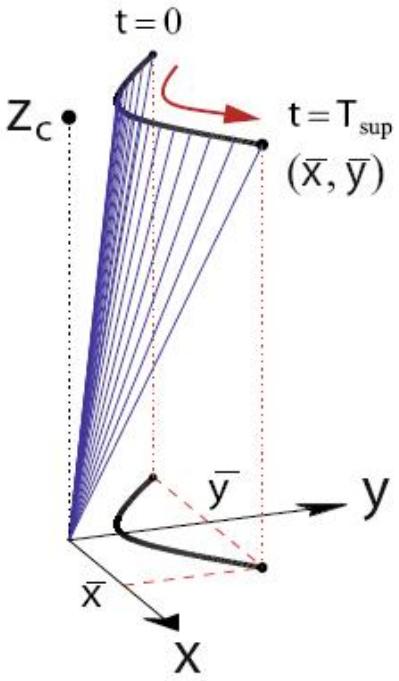

Figura 9.5: Primitiva di marcia

La concatenazione delle primitive di marcia permette di realizzare molto facilmente una traiettoria di marcia.

Si nota ora che, nel caso in analisi, si fissano dapprima le posizioni successive dei piedi, solo dopo aver determinato queste posizioni si determina la traiettoria del corrispondente spostamento.

Siano dati i seguenti parametri di marcia (come mostrato in figura 9.6):

- s_x: lunghezza del passo nel senso di marcia;
- s_y: larghezza del passo secondo la normale dello spostamento;
- s_θ: direzione del passo

Se n è l'indice dell'n-esimo passo, allora le posizioni $(p_x^{(n)}, p_y^{(n)})$ del piede sono date da

$$\begin{bmatrix} p_x^{(n)} \\ p_y^{(n)} \end{bmatrix} = \begin{bmatrix} p_x^{(n-1)} \\ p_y^{(n-1)} \end{bmatrix} + \begin{bmatrix} \cos s_\theta^{(n)} & -\sin s_\theta^{(n)} \\ \sin s_\theta^{(n)} & \cos s_\theta^{(n)} \end{bmatrix} \begin{bmatrix} s_x^{(n)} \\ -(-1)^n s_y^{(n)} \end{bmatrix} \tag{9.19}$$

In questa equazione è stato considerato il piede destro come supporto di partenza e $(p_x^{(0)}, p_y^{(0)})$ sarà la posizione del primo punto di supporto.
Le primitive di marcia corrispondenti all'n-esimo passo saranno:

$$\begin{bmatrix} \bar{x}^{(n)} \\ \bar{y}^{(n)} \end{bmatrix} = \begin{bmatrix} \cos s_\theta^{(n+1)} & -\sin s_\theta^{(n+1)} \\ \sin s_\theta^{(n+1)} & \cos s_\theta^{(n+1)} \end{bmatrix} \begin{bmatrix} s_x^{(n+1)}/2 \\ -(-1)^n s_y^{(n+1)}/2 \end{bmatrix} \tag{9.20}$$

Le caratteristiche dell'n-esima primitiva vengono dunque determinate in base agli (n+1)-esimi parametri di marcia. Questa è una condizione necessaria per poter effettuare un buon coordinamento tra gli spostamenti dei piedi e il movimento di marcia.
Le componenti in x e y della velocità finale di una primitiva di marcia si ottengono come

$$\begin{bmatrix} \bar{v}_x^{(n)} \\ \bar{v}_y^{(n)} \end{bmatrix} = \begin{bmatrix} \cos s_\theta^{(n+1)} & -\sin s_\theta^{(n+1)} \\ \sin s_\theta^{(n+1)} & \cos s_\theta^{(n+1)} \end{bmatrix} \begin{bmatrix} (1+C)/(T_c S)\bar{x}^{(n)} \\ (1c-1)/(T_c S)\bar{y}^{(n)} \end{bmatrix} \tag{9.21}$$

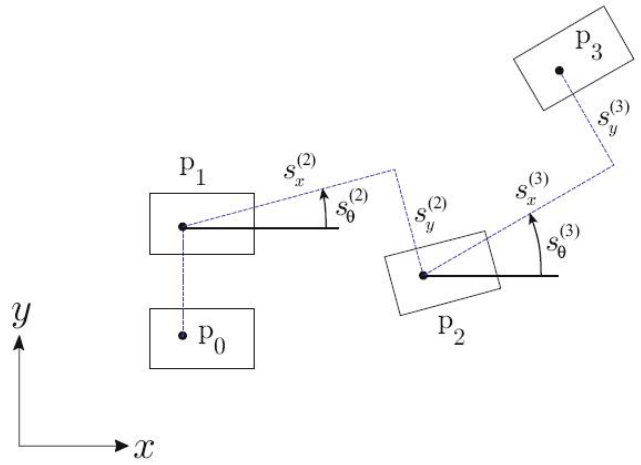

Figura 9.6: Parametri di marcia

Le primitive usate qui sopra per generare la marcia sono tutte identiche, tranne la prima e l'ultima, che corrispondono agli stati di avvio e di arresto.

Di seguito si descrive un metodo che permette di generare un modello di marcia continuo e realizzabile.

Sia p_x^* la posizione modificata del piede secondo la direzione di marcia. Si studiano le variazioni dinamiche dovute a questo cambiamento: la figura 9.7 rappresenta un pendolo inverso lineare nelle coordinate di riferimento del suolo. L'equazione della dinamica secondo l'asse x si scrive come

$$\ddot{x} = \frac{g}{z_c}(x - p_x^*) \tag{9.22}$$

e la soluzione analitica dell'equazione della dinamica è

$$x(t) = (x_i^{(n)} - p_x^*)\cosh(t/T_c) + T_c\dot{x}_i^{(n)}\sinh(t/T_c) + p_x^* \tag{9.23}$$

$$\dot{x}(t) = \frac{x_i^{(n)} - p_x^*}{T_c}\sinh(t/T_c) + \dot{x}_i^{(n)}\cosh(t/T_c) \tag{9.24}$$

dove $(x_i^{(n)}, \dot{x}_i^{(n)})$ è la condizione iniziale dell'n-esimo passo. Si può dunque estrarre una relazione tra la posizione modificata del piede p_x^* e lo stato finale $(x_f^{(n)}, \dot{x}_f^{(n)})$ dell'n-esimo passo:

$$\begin{bmatrix} x_f^{(n)} \\ \dot{x}_f^{(n)} \end{bmatrix} = \begin{bmatrix} C & T_c S \\ S/T_c & C \end{bmatrix} \begin{bmatrix} x_i^{(n)} \\ \dot{x}_i^{(n)} \end{bmatrix} + \begin{bmatrix} 1-C \\ -S/T_c \end{bmatrix} p_x^* \tag{9.25}$$

Viene scelto come stato obiettivo lo stato finale della primitiva di marcia nelle coordinate legate al suolo:

$$\begin{bmatrix} x^d \\ \dot{x}^d \end{bmatrix} = \begin{bmatrix} p_x^{(n)} + \bar{x}^{(n)} \\ \bar{v}_x^{(n)} \end{bmatrix} \tag{9.26}$$

Si calcola quindi la posizione del piede corrispondente allo stato finale più vicino allo stato desiderato $(x^d, \dot{x}^d)$. La funzione di valutazione può essere definita come

$$N \equiv a(x^d - x_f^{(n)})^2 + b(\dot{x}^d - \dot{x}_f^{(n)})^2 \tag{9.27}$$

dove a e b sono dei pesi positivi.
Sostituendo i valori trovati per x_f^n e $\dot{x}_f^n$ nella funzione di valutazione e applicando la condizione $\delta N/\delta p_x^* = 0$, si ottiene la posizione del piede che minimizza N:

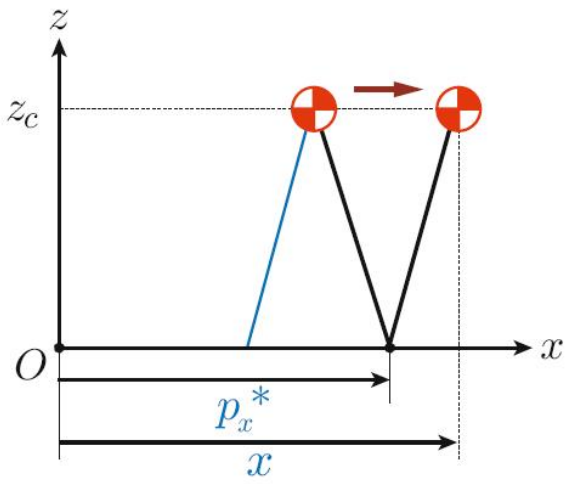

Figura 9.7: Rappresentazione di un LIP nelle coordinate di riferimento

$$\begin{cases} p_x^* = -\frac{a(C-1)}{D}(x^d - Cx_i^{(n)} - T_c S\dot{x}_i^{(n)}) - \frac{bS}{T_c D}(\dot{x}^d - \frac{S}{T_c}x_i^{(n)} - C\dot{x}_i^{(n)}) \\ D \equiv a(C-1)^2 + b(S/T_c)^2 \end{cases} \tag{9.28}$$

Il metodo che genera il modello di marcia si compone di nove passi successivi, di seguito se ne riporta l'algoritmo:

- *Passo 1*: Determinare il periodo di supporto T_{sup} e i parametri di marcia s_x, s_y e s_θ. Indicare poi la posizione iniziale del CoM(x, y) e del piede $(p_x^*, p_y^*) = (p_x^{(0)}, p_y^{(0)})$.
- *Passo 2*: T = 0, n = 0.
- *Passo 3*: Integrare l'equazione (9.22) del pendolo inverso.
- *Passo 4*: T = T + T_{sup}, n = n + 1.
- *Passo 5*: Calcolare la posizione successiva (p_x^n, p_y^n) del piede usando l'equazione (9.19).
- *Passo 6*: Determinare le caratteristiche della primitiva di marcia successiva, $(\bar{x}^{(n)}, \bar{y}^{(n)})$, usando le equazioni (9.20) e (9.21).
- *Passo 7*: Calcolare lo stato obiettivo $(x^d, \dot{x}^d)$ per mezzo della (9.26) e lo stato obiettivo $(y^d, \dot{y}^d)$ per mezzo dell'equazione corrispondente.
- *Passo 8*: Calcolare la posizione modificata del piede (p_x^+, p_y^*) per mezzo della (9.28).
- *Passo 9*: Ritornare al passo 3.

9.3 Dal pendolo lineare inverso al modello multicorpo

Quello appena esposto sarà l'algoritmo che verrà utilizzato durante la marcia.

Si nota come l'equazione del pendolo inverso corrisponda a quella dello ZMP. Questo significa che, riassumendo, dapprima si determinerà la posizione dello ZMP, in base a questa si posizionerà il piede di supporto (in questo modo si assicura che lo ZMP sia all'interno del poligono di sostentamento), e di conseguenza si ricaveranno la posizione e la velocità da imprimere al CoM.

A partire dalla posizione del CoM si determinerà la posizione dell'anca.
E una volta fissate le traiettorie dell'anca e dei due piedi, si ricaveranno gli angoli articolari del robot attraverso la cinematica inversa.

CAPITOLO 10

GAZEBO

I simulatori robotici consentono la proiezione di robot all'interno di un ambiente dotato di gravità, popolabile con oggetti e facilmente equiparabile al mondo reale.
Permettono così di creare applicazioni indipendenti dalla piattaforma hardware, con conseguente risparmio di denaro e tempo.

Nel caso in analisi è stato usato *Gazebo*. Tra gli altri, è stato scelto questo simulatore perché già integrato in ROS e largamente utilizzato tra la comunità si sviluppatori software.

Indispensabile è stato il suo utilizzo poiché ha consentito la verifica visiva della stabilità del robot usato nell'esperienza.
In breve: è stato simulato l'ambiente in uso, in esso è stato inserito il Robovie-X, è stata testata la conservazione dell'equilibrio del robot durante il movimento dei suoi giunti.

In questo capitolo si andrà ad introdurre brevemente il simulatore usato per poi procedere all'analisi del lavoro effettuato per usarlo nell'esperienza.

10.1 Il simulatore

Gazebo è un'interfaccia grafica utilizzata per la simulazione di robot in ambienti tridimensionali: è in grado di simulare popolazioni di robot, oggetti e sensori.
Si nota che è disegnato per un piccolo numero di robot, non è in grado di rappresentarne un centinaio, ma questo non influenza i fini dell'esperienza.

Esso fornisce meccanismi che consentono il compimento di movimenti realistici: ODE e OGRE.

ODE

ODE, o *Open Dynamics Engine*, è un simulatore di fisica, ideato da Russel Smith, che offre all'utente una grande varietà di funzioni, stabile, indipendente dalla piattaforma e con una API C/C++ di facile utilizzo.
Esso consente di riprodurre la dinamica di corpi rigidi, nonché la collision detection che li caratterizza. Permette inoltre di tener traccia di situazioni in cui è presente attrito.
Per questo è utile nella simulazione di veicoli o, in altro modo, oggetti collocabili in realtà virtuali. Infatti, attualmente, viene utilizzato in molti giochi per pc, strumenti di authoring 3D e di simulazione.

In generale i simulatori di corpi rigidi risolvono:

- vincoli di cinematica;
- vincoli di collisione e contatto;
- dinamica del corpo rigido: f = Ma.

Si fonda sul solver LCP, o *Link Control Protocol*, un pacchetto a cui viene prefitto il compito di controllare l'identità dei dispositivi connessi, esso va ad accettare o rifiutare tali dispositivi, determina la dimensione accettabile del pacchetto da trasmettere, ricerca errori nelle configurazioni, può anche terminare un link se i requisiti eccedono i parametri che lo caratterizzano.
In particolare viene usato il *Successive Over-Relaxation* (SOR) *Projected Gauss-Seidel* (PGS) LCP solver, essenzialmente un algoritmo che si fonda sul metodo Gauss-Seidel per la risoluzione delle matrici e che va a proiettare, ad ogni update, il vettore delle soluzioni nello spazio delle soluzioni ammissibili.

Il meccanismo elimina le deviazioni dei vincoli della posizione dei giunti imponendo un *Error Reduction Parameter* (ERP).

Quando un giunto mette in collegamento due corpi, questi corpi devono avere certe posizioni ed orientazioni relative l'uno con l'altro. In ogni modo, è possibile che i corpi si trovino in posizioni in cui i vincoli dei giunti non vengono rispettati. Questo errore può comparire in due situazioni: se l'utente setta la posizione/orientazione di un corpo senza settare correttamente la posizione/orientazione dell'altro corpo; o durante la simulazione, quando i corpi si possono scostare dalle posizioni richieste.
Esiste un meccanismo atto alla riduzione di questo errore: durante ogni passo di simulazione ogni giunto applica una particolare forza per far ritornare il corpo al giusto allineamento. Questa forza viene controllata dall'ERP, che ha un valore tra 0 e 1. Esso specifica quale proporzione di errore del giunto verrà corretta durante il successivo passo di simulazione. Se ERP=0 non viene applicata alcuna forza correttiva e i corpi potranno anche muoversi in via separata a man mano che la simulazione avanza. Se ERP=1, invece, al passo successivo verrà corretto interamente il joint error. Comunque, settare ERP a 1 non è raccomandato, infatti non si riuscirà mai a correggere completamente un errore poichè vi saranno sempre delle approssimazioni interne. Solitamente si usa un valore tra 0.1 e 0.8.

ODE va poi a migliorare la stabilità numerica complessiva attraverso la matrice *Constraint Force Mixing* (CFM).

La maggior parte dei vincoli rappresenta condizioni *hard*, che non devono venir mai violate. Ad esempio, due parti di una cerniera devono essere sempre allineate. Nella pratica questi vincoli possono venir violati a causa di una introduzione non intenzionale di errori all'interno del sistema.
Altri vincoli sono *soft* e possono venir violati, per esempio il vincolo che costringe due corpi a non penetrarsi è hard, ma diventa soft se si vanno a simulare oggetti morbidi.
Ci sono due parametri che permettono di distinguere *soft* da *hard*: l'ERP, sopra descritto, ed il CFM. Quest'ultimo va semplicemente ad aggiungere dei valori correttivi alla diagonale della jacobiana delle velocità dei corpi collegati del giunto preso in considerazione. La correzione è arbitraria ma si ricorda che, ogniqualvolta viene aggiunto un valore positivo, il sistema viene allontanato da singolarità e si va ad aumentare l'acccuratezza della computazione.

OGRE

OGRE, o *Object-Oriented Graphics Rendering Engine*, è un motore di rendering 3D flessibile ed orientato alla scena.
È software libero, divulgato sotto licenza MIT, ed ha una comunità di utilizzatori molto attiva (si ricorda il suo uso all'interno di alcuni videogiochi commerciali).

Come dice il suo nome, OGRE è solo un motore di rendering. Come tale, il suo scopo principale è quello di fornire soluzioni generali per il rendering grafico. Nonostante questo ci sono alcune strutture come vettori, matrici e gestione della memoria che hanno solo uno scopo d'aiuto. Chi cerca una soluzione *tutto in uno* per sviluppare un videogioco o una simulazione deve fare attenzione, poiché per esempio non fornisce supporto per l'audio e per la fisica. In generale, si pensa che questo sia il maggiore svantaggio per OGRE, ma può essere anche visto come una caratteristica del motore. La scelta di OGRE di essere solo un motore grafico lascia agli sviluppatori la libertà di usare le librerie che desiderano per l'audio, la fisica, gli input...

Ritornando all'esperienza in analisi, per andare ad usare il robot sotto Gazebo è stato indispensabile innanzitutto integrarne l'URDF con le caratteristiche che descrivono il corpo all'interno del simulatore. È stato poi riportato l'ambiente in uso sotto questo simulatore.

10.2 .world

In Gazebo, l'ambiente viene caricato attraverso un file avente estensione .world, un file XML che contiene la descrizione dell'ambiente.
Il file permette di personalizzare caratteristiche base del simulatore come i parametri fisici, la gravità, il materiale di cui dovrà costituirsi il suolo, la luce presente nell'ambiente, gli oggetti che lo popolano, e così via.

Di seguito si riporta il codice necessario per avviare un ambiente costituito solo da suolo e cielo (si trova in ROS sotto gazebo_worlds/launch/empty_world.launch,):

```
<launch>
  <!-- start gazebo with an empty plane -->
  <param name="/use_sim_time" value="true" />
  <node name="gazebo" pkg="gazebo" type="gazebo" args="$(find
gazebo_worlds)/worlds/empty.world" respawn="false" output="screen"/>
</launch>
```

Importante è il parametro /use_sim_time; esso porta ROS ad usare il tempo di simulazione pubblicato da Gazebo all'interno del topic /clock, invece che usare l'orologio di sistema.
Quando un robot reale è in esecuzione, infatti, le librerie ROS useranno l'orologio di sistema, quello riportato dal proprio computer. Ma quando si sta eseguendo in simulazione, è desiderabile avere un tempo simulato di modo che si possa avere controllo sulle accelerazioni, i rallentamenti o gli stop. Per esempio, se si stanno usando dati pubblicati da dei sensori, si vorrà avere il tempo che corrisponda al timestamp dei dati dai sensori stessi.
Questo viene supportato mettendo mettendo le librerie ROS in ascolto di un /clock globale che viene usato per pubblicare il tempo simulato durante l'esecuzione di un sistema (non vengono mandati messaggi a /clock quando si sta usando l'orologio di sistema).
Il parametro /use_sim_time viene impostato a *true* nel momento in cui si sta usando il tempo simulato. Esso verrà quindi usato anche all'interno del launch della nostra esperienza, viene però lanciato un .world popolato da ostacoli che il robot dovrà evitare nel suo cammino (vedi figura 10.1).

Si procede quindi alla descrizione delle modalità che consentono la creazione di un file .world.

Innanzitutto è necessario inserire il seguente blocco in testa al file:

```
<?xml version="1.0"?>
<gz:world xmlns:gz="http://playerstage.sourceforge.net/gazebo/xmlschema/#gz"
```

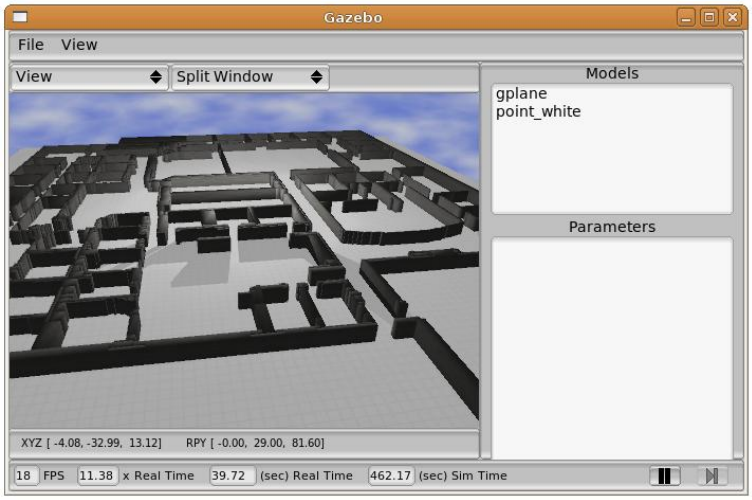

Figura 10.1: Ambiente in uso simulato in Gazebo

```
xmlns:model="http://playerstage.sourceforge.net/gazebo/xmlschema/#model"
xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
xmlns:window="http://playerstage.sourceforge.net/gazebo/xmlschema/#window"
xmlns:param="http://playerstage.sourceforge.net/gazebo/xmlschema/#params"
xmlns:ui="http://playerstage.sourceforge.net/gazebo/xmlschema/#params">
****** definition of the world is entered here ********
</gz:world>
```

e poi si elencano i modelli di tutti gli oggetti che devono essere inseriti all'interno della mappa.
Nella fattispecie, nel modello usato e caricato, oltre al cielo ed al suolo (si rimandi il lettore alla sezione */gazebo_worlds* di ROS [13] per una descrizione più dettagliata della costruzione dei modelli) viene caricata la pianta della Willow Garage. Per far ciò basta inserire all'interno del .world il codice seguente:

```
<model:physical name="willow_map">
 <xyz>-10 -5 0</xyz>
 <rpy>0 0 0</rpy>
 <static>true</static>
 <body:trimesh name="willow_map_body">
   <geom:trimesh name="willow_map_geom">
     <scale>0.01 0.01 0.01</scale>
     <mesh>willowgarage2.dae</mesh>
     <visual>
       <scale>0.01 0.01 0.01</scale>
       <mesh>willowgarage2.dae</mesh>
     </visual>
   </geom:trimesh>
 </body:trimesh>
</model:physical>
```

dove *xyz* viene usato per indicare le coordinate all'origine della mappa, *rpy* le rispettive rotazioni roll, pitch e yaw. La mappa è statica e viene inserita come un file .dae scalato rispetto al file originale.
Si ricorda che .dae, o COLLADA (acronimo di COLLAborative Design Activity), è un formato file di interscambio tra applicazioni 3D, usato per esempio da Blender, realizzato in codice XML.

10.3 URDF

Come indicato al capitolo 5, il robot viene rappresentato come un albero composto da un insieme di link uniti tra loro da un insieme di giunti.
Affinché il robot possa venir simulato in Gazebo è indispensabile estendere il file URDF precedentemente creato aggiungendo alla definizione di ogni link la mappatura dello stesso all'interno del simulatore.
Per esempio, sarà necessario completare la configurazione del link *head* integrando la specifica delle sue caratteristiche con una scrittura del tipo:

```
<gazebo reference="head">
    <turnGravityOff>false</turnGravityOff>
</gazebo>
```

che sottolinea il fatto che il link venga indicato con il nome *head* anche in Gazebo ed il fatto che per quel link venga presa in considerazione la gravità.
In questa sezione possono venir specificate anche altre proprietà, come il materiale costituente il link. Questi approfondimenti non sono però stati trattati nel lavoro proposto.

Si sottolinea che, per aggiungere queste specifiche all'interno dell'URDF, è indispensabile munire il tag *robot*, oltre che del nome dato alla struttura, anche degli xml Gazebo.
Per il Robovie-X creato si otterrà un'intestazione di questo tipo:

```
<robot name="robovie_x"
xmlns:body="http://playerstage.sourceforge.net/gazebo/xmlschema/#body"
xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#controller"
xmlns:geom="http://playerstage.sourceforge.net/gazebo/xmlschema/#geom"
xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#interface"
xmlns:joint="http://playerstage.sourceforge.net/gazebo/xmlschema/#slider"
xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
xmlns:xacro="http://ros.org/wiki/xacro">
```

Alla fine del file, invece, in una nuova sezione <gazebo>..</gazebo>, verrà indicato il controllore.
Esso andrà a caricare il robot in condizioni di stabilità ed equilibro all'interno di un ambiente gravitazionale. Nel tempo stesso, ogniqualvolta verranno impostate posizioni non ottimali, provvederà alla caduta del corpo, garantendo quindi un comportamento il può possibile vicino a quello reale:

```
<gazebo>
    <controller:robovie_x_gazebo_plugin name="robovie_x_gazebo_plugin"
             plugin="librobovieXGazeboPlugin.so">
     <interface:audio name="gazebo_ros_test_dummy_interface"/>
     <alwaysOn>true</alwaysOn>
     <updateRate>1000.0</updateRate>
       <timeout>5</timeout>
    </controller:robovie_x_gazebo_plugin>
</gazebo>
```

Questo controllore è stato creato per mezzo di un file .cpp, *ros_controller.cpp*, composto da quattro funzioni fondamentali:

- *LoadChild*: carica tutti i giunti del robot a partire dal modello URDF;
- *InitChild*: chiama per la prima volta il controllore ed inizializza tutti i giunti impostando loro velocità e forza nulle;

- *UpdateChild*: viene ciclicamente richiamata durante il funzionamento del controllore. Garantisce l'impostazione delle corrette velocità e forze ai giunti di modo che il robot simuli la realtà;
- *FiniChild*: una volta che il controllore viene fermato libera la memoria delle computazioni effettuate.

Vista l'importanza di questo componente, di seguito se ne riporta il codice:

```
#include <RobovieXGazeboPlugin.h>
#include <fstream>
#include <iostream>
#include <math.h>
#include <unistd.h>
#include <set>
#include <gazebo/Global.hh>
#include <gazebo/World.hh>
#include <gazebo/PhysicsEngine.hh>
#include <gazebo/XMLConfig.hh>
#include <gazebo/Model.hh>
#if GAZEBO_MAJOR_VERSION == 0 && GAZEBO_MINOR_VERSION >= 10
#include <gazebo/Joint.hh>
#else
#include <gazebo/HingeJoint.hh>
#include <gazebo/SliderJoint.hh>
#endif
#include <gazebo/Simulator.hh>
#include <gazebo/gazebo.h>
#include <angles/angles.h>
#include <gazebo/GazeboError.hh>
#include <gazebo/ControllerFactory.hh>
#include <urdf/model.h>
#include <map>

#define JOINT_MAX_FORCE (1.0)

namespace gazebo {

GZ_REGISTER_DYNAMIC_CONTROLLER("robovie_x_gazebo_plugin", RobovieXGazeboPlugin);

boost::bimap<int,std::string> RobovieXGazeboPlugin::_jointNameMap = initJointNameMap();

boost::bimap<int,std::string> RobovieXGazeboPlugin::initJointNameMap()
{
   boost::bimap<int,std::string> jointNameMap;
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(head_joint_yaw,"head_joint_yaw"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_shoulder_joint_pitch,"left_shoulder_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_shoulder_joint_pitch,"right_shoulder_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_shoulder_joint_roll,"left_shoulder_joint_roll"));
```

```
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_shoulder_joint_roll,"right_shoulder_joint_roll"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_arm_joint_pitch,"left_arm_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_arm_joint_pitch,"right_arm_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_hip_joint_roll,"left_hip_joint_roll"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_hip_joint_roll,"right_hip_joint_roll"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_hip_joint_pitch,"left_hip_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_hip_joint_pitch,"right_hip_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_knee_joint_pitch,"left_knee_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_knee_joint_pitch,"right_knee_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_ankle_joint_pitch,"left_ankle_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_ankle_joint_pitch,"right_ankle_joint_pitch"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(left_ankle_joint_roll,"left_ankle_joint_roll"));
   jointNameMap.insert(boost::bimap<int,std::string>::
         value_type(right_ankle_joint_roll,"right_ankle_joint_roll"));
   return jointNameMap;
}

/* Return the name corresponding to the input joint */
std::string RobovieXGazeboPlugin::getNameFromJoint(const int& joint)
{
   std::string name = "";
   if (_jointNameMap.left.count(joint))
      name = _jointNameMap.left.at(joint);
   return name;
}

/* Return the joint corresponding to the input name */
int RobovieXGazeboPlugin::getJointFromName(const std::string& name)
{
   int joint = 0;
   if (_jointNameMap.right.count(name))
         joint = _jointNameMap.right.at(name);
   return joint;
}

RobovieXGazeboPlugin::RobovieXGazeboPlugin(Entity *parent)
  : Controller(parent),
    pgain_(2.0),
    dgain_(0.0)
{
  this->parent_model_ = dynamic_cast<Model*>(this->parent);
```

```
  if (!this->parent_model_)
    gzthrow("RobovieXGazeboPlugin controller requires a Model as its parent");
  Param::Begin(&this->parameters);
  this->robotParamP = new ParamT<std::string>("robotParam", "robot_description", 0);
  this->robotNamespaceP = new ParamT<std::string>("robotNamespace", "/", 0);
  this->setModelsJointsStatesServiceNameP = new ParamT<std::string>
         ("setModelsJointsStatesServiceName","set_models_joints_states", 0);
  Param::End();
}

RobovieXGazeboPlugin::~RobovieXGazeboPlugin()
{
  delete this->robotParamP;
  delete this->robotNamespaceP;
  delete this->rosnode_;
  delete this->ros_spinner_thread_;
}

/* angle-vector */
void RobovieXGazeboPlugin::cmdCallback(const sensor_msgs::JointState::
      ConstPtr& jointState)
{
  // vector copy
  for (unsigned int i=0;i<jointState->position.size();i++)
    {
      ROS_ERROR("Nome del giunto: %s", jointState->name[i].c_str());
      cmd_[getJointFromName(jointState->name[i])] = jointState->position[i];
      ROS_ERROR("setting %f", i);
    }
  angles_pub_.publish(jointState);
  usleep(250*1000);
}

/* pgain setting */
void RobovieXGazeboPlugin::cmdCallbackPgain(const std_msgs::Float64::ConstPtr& gain)
{
  pgain_ = gain->data;
}

/* dgain setting */
void RobovieXGazeboPlugin::cmdCallbackDgain(const std_msgs::Float64::ConstPtr& gain)
{
  dgain_ = gain->data;
}

void RobovieXGazeboPlugin::LoadChild(XMLConfigNode *node)
{
  // get parameter name
  this->robotParamP->Load(node);
  this->robotParam = this->robotParamP->GetValue();
  this->robotNamespaceP->Load(node);
  this->robotNamespace = this->robotNamespaceP->GetValue();
  int argc = 0;
```

```
char** argv = NULL;

// read urdf
if (!this->urdf_model_.initFile(ros::package::getPath("robovie_x_model") +
     "/xacro/robot.urdf"))
{
   ROS_ERROR("load urdf error!");
}

std::vector <boost::shared_ptr<urdf::Link> > links;
urdf_model_.getLinks(links);

for(unsigned int i=firstRobovieXJoint;i<=lastRobovieXJoint;i++)
   joints_.push_back(NULL);

for (std::vector <boost::shared_ptr<urdf::Link> >::iterator it = links.begin();
     it != links.end(); it++)
{
    std::vector <boost::shared_ptr<urdf::Joint> > joints = (*it)->child_joints;
    for ( std::vector<boost::shared_ptr<urdf::Joint> >::iterator j =
       joints.begin(); j !=joints.end(); j++)
    {
       std::string jointName = (*j)->name;
       ROS_ERROR("name=%s", jointName.c_str());
       gazebo::Joint *joint = this->parent_model_->GetJoint(jointName);
       if (joint)
       {
         ROS_ERROR("Joint %s inserted!", jointName.c_str());
          this->joints_[getJointFromName(jointName)] = joint;
          joint->SetVelocity(0,0);
          joint->SetMaxForce(0,JOINT_MAX_FORCE);
        }
        else
        {
           this->joints_.push_back(NULL);
        }
    }
}
ROS_ERROR("Joint inserted! %i", joints_.size());

ros::init(argc,argv,"gazebo",ros::init_options::NoSigintHandler|ros::
   init_options::AnonymousName);
this->rosnode_ = new ros::NodeHandle(this->robotNamespace);
cmd_.resize(this->joints_.size());
angles_pub_ = rosnode_->advertise<std_msgs::Float64>("robovie_x/angles", 1);
cmd_angles_sub_ = rosnode_->subscribe<sensor_msgs::JointState>("joint_states",
   1, &RobovieXGazeboPlugin::cmdCallback, this);
cmd_pgain_sub_ = rosnode_->subscribe<std_msgs::Float64>("robovie_x/pgain",
   1, &RobovieXGazeboPlugin::cmdCallbackPgain, this);
cmd_dgain_sub_ = rosnode_->subscribe<std_msgs::Float64>("robovie_x/dgain",
   1, &RobovieXGazeboPlugin::cmdCallbackDgain, this);

ROS_INFO("starting gazebo_ros_test plugin in ns: %s",this->robotNamespace.c_str());
```

```
}

void RobovieXGazeboPlugin::InitChild()
{
  this->ros_spinner_thread_ = new boost::thread( boost::bind( &ros::spin ) );
}

void RobovieXGazeboPlugin::UpdateChild()
{
  // Copies the commands from the mechanism joints into the gazebo joints
  ROS_ERROR("num=%d", this->joints_.size());
  for (unsigned int i = 0; i < this->joints_.size(); ++i)
  {
   Joint *joint = this->joints_[i];
   if(!joint)
   {
      ROS_ERROR("INDICE %i ERRORE NESSUN JOINT", i );
      continue;
   }
   else
   {
      ROS_ERROR("Tutto ok!");
   }

   double
   current_velocity = joint->GetVelocity(0),
   current_angle = joint->GetAngle(0).GetAsRadian(),
   damping_force = dgain_ * (0 - current_velocity),
   diff_force = pgain_ * (cmd_[i] - current_angle),
   effort_command = diff_force + damping_force;
     ROS_ERROR("c=%f,cmd=%f, f=%f, v=%f", current_angle, cmd_[i], effort_command,
        current_velocity);
      joint->SetForce(0, effort_command);
      joint->SetVelocity(0,effort_command);
  }
}

void RobovieXGazeboPlugin::FiniChild()
{
  ROS_DEBUG("Calling FiniChild in RobovieXGazeboPlugin");
  for (unsigned int i=0; i<this->joints_.size(); i++){
    if (this->joints_[i]){
      delete this->joints_[i];
      this->joints_[i] = NULL;
    }
  }
  this->ros_spinner_thread_->join();
}
} // namespace gazebo
```

Importante è soffermarsi sul ciclo che garantisce il giusto equilibrio del robot.
Per imprimere parametri corretti è stato costruito un PID, formulando un controllo *Proporzionale-Integrale-Derivativo*.

In generale come funziona un PID? Esso acquisisce in ingresso il valore dato da un processo e lo confronta con un valore di riferimento. La differenza, il cosiddetto segnale di errore, viene usata per determinare il valore della variabile di uscita del controllore, che è la variabile manipolabile del processo.
Il PID regola l'uscita in base a:

- il valore del segnale di errore (azione proporzionale);
- i valori passati del segnale di errore (azione integrale);
- quanto velocemente il segnale di errore varia (azione derivativa).

I controllori PID sono relativamente semplici da comprendere, installare, e tarare al confronto con più complessi algoritmi di controllo basati sulla teoria del controllo ottimo e del controllo robusto. La taratura dei parametri avviene di solito attraverso semplici regole empiriche che risultano in controllori stabilizzanti di buone prestazioni per la maggior parte dei processi.

Nel caso in analisi il PID agisce sulla velocità e sulle forze da imprimere ai giunti.
Affinché esso sia sempre al corrente dei valori costituenti queste grandezze, è necessario pubblicare costantemente lo stato di link e giunti. A tal scopo esistono e vengono utilizzati *robot_state_publisher* e *joint_state_publisher*.

Il file di lancio, comprensivo non solo di visualizzazione del robot in Gazebo, ma anche in Rviz, mappa dell'ambiente in entrambi i sistemi e ricerca del percorso Start-Goal, sarà:

```
<?xml version="1.0"?>

<launch>

<!--includo la mappa-->
<node name="octomap_server" pkg="octomap_server" type="octomap_server_node"
   args="$(find footstep_planner)/map/map.binvox.bt" />

<!-- risoluzione della griglia -->
<param name="resolution" value="0.00001" />

<!-- send urdf to param server -->
<arg name="model" default="$(find robovie_x_model)/xacro/robot.urdf" />
<arg name="gui" default="true" />
<param name="robot_description" textfile="$(arg model)" />
<param name="use_gui" value="$(arg gui)"/>

<!-- aggiungo il collegamento alla mappa -->
<node pkg="add_frame" type="frame_tf_broadcaster" name="broadcaster_frame" />

<!-- start robot state publisher -->
<node pkg="robot_state_publisher" type="state_publisher"
   name="robot_state_publisher" output="screen"/>

<node name="joint_state_publisher" pkg="joint_state_publisher"
   type="joint_state_publisher"/>

<!-- aggiungo la mappatura dei giunti fatta in Wizard con le relative collisioni-->
<rosparam command="load" ns="robot_description_planning" file="$(find
```

```
    robovie_x_arm_navigation)/config/robovie_x_planning_description.yaml" />

<!--pacchetto che serve per far muovere il robot -->
<node name="environment_server" pkg="planning_environment" type="environment_server"/>

<!-- start gazebo with an empty plane -->
<node pkg="gazebo" type="gazebo" name="gazebo" args="-u $(find
   gazebo_map)/worlds/map_collada.world"/>

<param name="/use_sim_time" value="true"/>

<!-- spawn the model in gazebo -->
<node name="spawn_robovie_x" pkg="gazebo" type="spawn_model" args="-z 0.1 -urdf
   -param robot_description -model robot_description" respawn="false" output="screen"/>

<!--lancio rviz con la configurazione adatta a mostrare mappa tridimensionale con
   le celle occupate, robot, tf, ..-->
<include file="$(find footstep_planner)/launch/rviz_footstep_planning.launch"/>

<!--node name="rviz" pkg="rviz" type="rviz"/-->

<!-- parametri del piede del robot e parametri di planning-->
<include file="$(find footstep_planner)/launch/footstep_planner.launch" />

</launch>
```

che metterà in luce una situazione come quella riportata nelle figure 10.2 e 10.3.

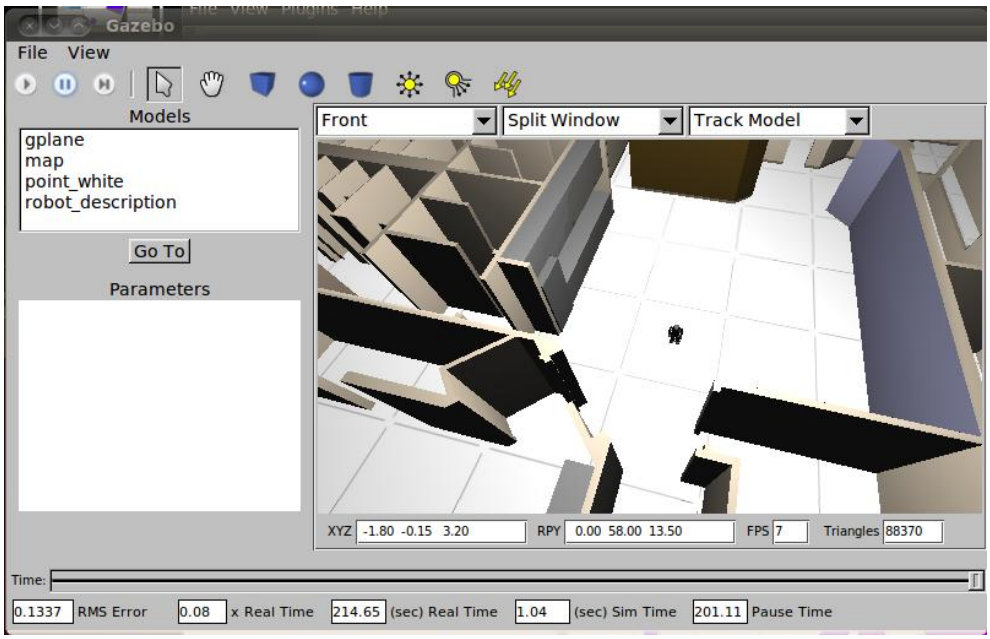

Figura 10.2: Robovie-X in Gazebo

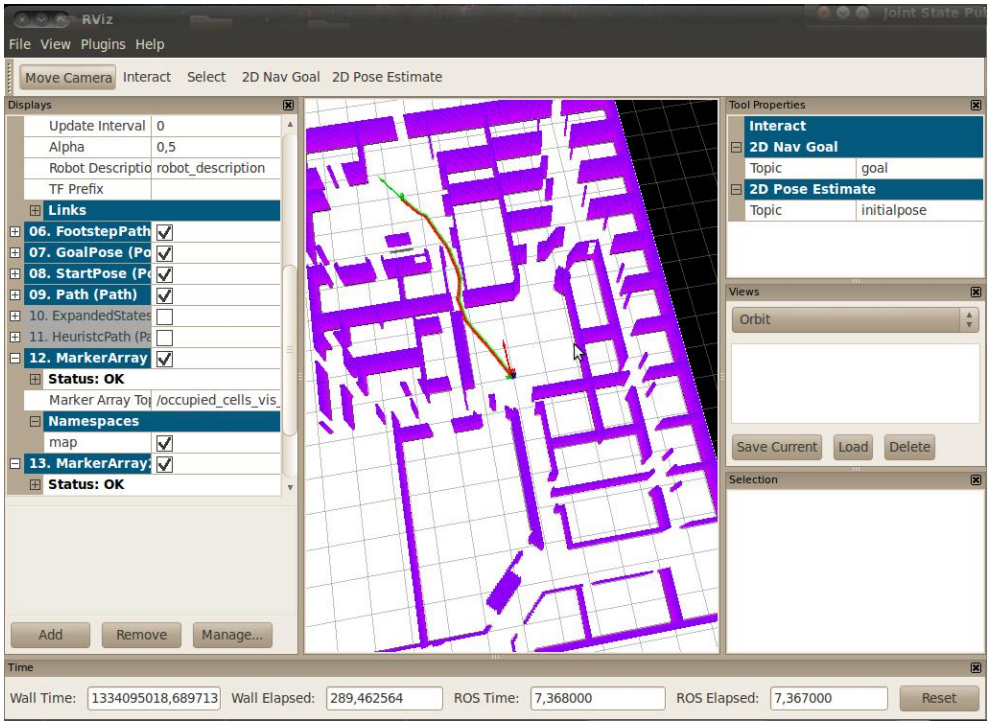

Figura 10.3: Robovie-X in Rviz

Conclusioni

Il lavoro svolto ha visto la risoluzione di un problema di Motion Planning in un ambiente tridimensionale e sotto l'utilizzo del robot Robovie-X.

Si fa notare al lettore che il codice creato può essere riutilizzato per l'elaborazione del moto di un qualsiasi robot in un qualsiasi spazio popolato da ostacoli statici.
Per poter riadattare il lavoro basta, per quel che riguarda l'occupancy grid tridimensionale, dare in ingresso il modello .obj desiderato; per quel che riguarda il robot, modificare il modello URDF a seconda delle nuove forme e dei nuovi giunti e modificare le dimensioni della suola del piede prima di iniziare la costruzione del cammino. Per Gazebo, infine, basta modificare il file .world ed i controllori che permettono al robot di rimanere in equilibrio durante la simulazione.

BIBLIOGRAFIA

[1] **Jonathan D. Cohen, Ming C. Ling, Dinesh Manocha, Madhav K. Ponamgi**: *I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments*, 14 Novembre 2000.

[2] **Kai M. Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, Wolfram Burgard**: *OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems*, in Proc. of the ICRA 2010 Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation, 2010.

[3] **Alessandro Ordan**: *Integrazione e test del simulatore SimSpark in ROS; Limiti e possibilità di utilizzo di ROS in ambiente industriale*, 12 Dicembre 2011.

[4] **C. Sprunk, B.Lau, P.Pfaff, W. Burgard**: *Online Generation of Kinodynamic Trajectories for Non-Circular Omnidirectional Robots*, in Proc. Of the IEEE Int. Conf. on Robotics & Automation (ICRA), 2011.

[5] **Shuuji Kajita, Hirohisa Hirukawa, Kensuke Harada, Kazuhito Yokoi**: *Introduction à la commande des robots humanoïdes*, 2009.

SITOGRAFIA

[6] **OMPL**: *http://ompl.kavrakilab.org/index.html*.

[7] **ROS**: *http://www.ros.org/wiki/*.

[8] **IROS**: *http://kavrakilab.org/OMPLtutorial*.

[9] **Binvox**: *http://www.patrickmin.com/minecraft/*.

[10] **Inerzia**: *http://en.wikipedia.org/wiki/List_of_moments_of_inertia/*.

[11] **Ode**: *http://www.ode.org/*.

[12] **Ogre**: *http://www.ogre3d.org/*.

[13] **Gazebo_worlds**: *http://www.ros.org/wiki/gazebo_worlds*.