



SAPIENZA
UNIVERSITÀ DI ROMA

Online Motion Planning con il robot mobile MARRtino

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Roberto Sorice
Matricola 1617825

Relatore

Prof. Roberto Capobianco

Anno Accademico 2018/2019

Online Motion Planning con il robot mobile MARRtino

Tesi di Laurea. Sapienza – Università di Roma

© 2019 Roberto Sorce. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: sorce.1617825@studenti.uniroma1.it

Ringraziamenti

Ringrazio mio zio, il Dr. Francesco Ippolito, per l'essenziale e fondamentale contributo all'ideazione del design e alla realizzazione del robot MARRtino.

Un ringraziamento speciale va al mio amico, il Dr. Ivan Bergonzani, per avermi dato preziosi consigli, per avermi supportato e dedicato il suo tempo durante lo sviluppo del progetto.

Indice

1	Introduzione	1
1.1	Robotica	1
1.2	Motion planning	2
2	Stato dell'arte	5
2.1	Problema canonico della pianificazione	5
2.2	Artificial Potential Fields	6
2.2.1	Potenziale attrattivo	7
2.2.2	Potenziale repulsivo	8
2.2.3	Potenziale totale	9
2.2.4	Minimi locali	9
2.3	Vortex Fields	10
3	Background	13
3.1	MARRtino	13
3.1.1	Firmware	15
3.2	ROS	15
3.3	Percezione con sensori RGBD	16
3.4	Librerie per lo sviluppo del progetto	18
3.4.1	Eigen	18
3.4.2	Point Cloud Library	18
3.4.3	OpenCV	19
4	Implementazione algoritmo Artificial Potential Fields	21
4.1	Progettazione	21
4.2	Obstacles mapper 2D	23
4.3	APF Planner	23
5	Risultati sperimentali	27
6	Conclusioni	33

Capitolo 1

Introduzione

1.1 Robotica

La Robotica è comunemente definita come la scienza che studia le correlazioni intelligenti tra percezioni e azioni in un sistema robotico[1]. Il termine *robotica* è stato introdotto da Isaac Asimov nei suoi numerosi racconti che hanno come protagonisti i robot.

La definizione di *robot* è stata introdotta per la prima volta dallo scrittore Ceco Karel Čapek, autore del libro utopico fantascientifico *Rossumovi univerzální roboti* del 1920, il quale ha coniato il termine, derivante dalla parola ceca *robo*, traducibile con lavoro forzato.

La costante e veloce crescita dell'utilizzo di robot in diversi aspetti della vita quotidiana giustifica la ricerca e l'interesse personale nei confronti di questo campo.

Un *sistema robotico* consta di molti sottosistemi atti a svolgere un determinato compito e garantire l'integrità del sistema complessivo. Nello specifico un robot è costituito dai sottosistemi che gli permettono di avere le capacità necessarie ad interagire con l'ambiente: un *sistema sensoriale* che permette di percepire dati sia che essi provengano dall'esterno, in questo caso sensori *esteroceettivi*, sia che si tratti di dati provenienti dall'interno del sistema, sensori *proprioceettivi*, un *sistema di controllo* per correlare le corrette azioni alle percezioni acquisite rispetto all'obiettivo prefissato, infine un sistema di attuatori che permettono al robot di esercitare azioni.

I robot mobili, al centro di questa trattazione, sono caratterizzati da una base mobile che consente al robot di muoversi nell'ambiente, sono utilizzati specialmente in contesti che richiedono un'estensiva capacità autonoma di movimento. Si differenziano a loro volta, per la tipologia di base mobile utilizzata per effettuare lo spostamento.

I *wheeled mobile robots* presentano un apparato di locomozione caratterizzato da un sistema di ruote fissate allo chassis che provvedono al movimento sul piano, soggetto comunque a vincoli cinematici che limitano i movimenti. Il veicolo considerato può essere modellato con delle ruote convenzionali per ottenere uno spostamento analogo a quello di un uniciclo con due ruote fisse a dei motori che gestiscono l'intensità della trazione e una *caster wheel* con due assi di rotazione che provvede al supporto della base e all'allineamento della struttura in direzione del movimento desiderato.

Le applicazioni in cui sono richiesti robot mobili autonomi sono molteplici, ad oggi si annoverano impieghi domestici dedicati alla pulizia, applicazioni di servizio negli ospedali, trasporto logistico e industriale, monitoraggio ed esplorazione di ambienti, navigazione a guida autonoma, salvataggio, intrattenimento e persino, purtroppo, impieghi nel campo militare.

L'esecuzione di qualsiasi compito da parte di un robot prevede uno specifico movimento, che deve essere eseguito rispettando i vincoli posti dall'ambiente e dalla struttura stessa del sistema.

Per un robot mobile il problema principale è costituito da vincoli presenti nell'ambiente in cui esso opera, che si presentano come ostacoli che ne impediscono il movimento, occorre dunque *pianificare* una traiettoria, una sequenza di configurazioni valide in modo da trovare un percorso sicuro da intraprendere ed evitare collisioni, da un'origine fino alla destinazione.

Come descritto nella prossima sezione, questo problema prende il nome di *motion planning*, noto anche come *problema della navigazione*, le cui tecniche risolutive sono essenzialmente di natura algoritmica, usando metodi probabilistici o euristici al fine di ottenere una soluzione ottimale [1].

1.2 Motion planning

La trattazione di questa tesi è incentrata sul problema della *pianificazione* e vengono fornite delle soluzioni valide a quest'ultimo. Un sistema robotico si prevede debba svolgere operazioni in un ambiente popolato da oggetti fisici che rappresentano vincoli al libero movimento. Pianificare un movimento significa, quindi, trovare un percorso che il robot deve seguire gradualmente per effettuare uno spostamento da una posizione iniziale ad una posizione finale senza collidere con gli ostacoli, è inoltre necessaria la conoscenza della geometria del robot e del mondo in cui esso si muove.

Si parla di *pianificazione fuori linea* (*offline planning*) nel caso in cui le informazioni riguardanti la geometria dell'ambiente siano fornite preliminarmente, ad esempio mediante l'utilizzo di una mappa, permettendo al robot di navigare un ambiente con piena cognizione dell'intero spazio all'interno del quale può muoversi.

Un robot, però, deve essere in grado di pianificare *in linea* (*online*) il moto, cioè deve poter orientarsi in ambienti *non strutturati* (non progettati appositamente per accogliere i robot), usando informazioni parziali acquisite durante la navigazione per mezzo dei sensori. Questa pianificazione prende il nome di *online planning*. Anche in questo caso può essere utile raccogliere i dati acquisiti in una mappa in maniera continua attraverso i sensori, in alternativa, le informazioni possono essere usate per pianificare azioni seguendo un paradigma di navigazione *memoryless* (navigazione *reattiva*) [1].

Trovare una soluzione al problema della pianificazione presenta difficoltà notevoli, sebbene possa sembrare si tratti di procedure semplici. Un umano che istintivamente effettua degli spostamenti sicuri in modo da non sbattere contro ostacoli o cadere dalle scale, effettua le medesime operazioni di un robot, utilizzando il ragionamento spaziale per avere coscienza dello spazio in cui si muove e al contempo evitare le collisioni anche in presenza di ostacoli mobili; Ciò è dovuto alla capacità del cervello umano di poter acquisire e processare una quantità ingente di informazioni con

l'ausilio dei sensi per acquisire le percezioni, ma codificare un algoritmo che può essere eseguito da un robot per svolgere le medesime azioni è ancora oggi un compito arduo. La pianificazione del moto costituisce un settore attivo di ricerca, al quale contribuiscono diverse discipline come la teoria degli algoritmi, controlli automatici, geometria computazionale[1].

Formalmente si esprime il problema del motion planning attraverso il *problema canonico della pianificazione*.

Il progetto presentato si propone di descrivere il problema del motion planning online e dei metodi risolutivi basati sui potenziali artificiali descritti nel prossimo capitolo. L'organizzazione del progetto è strutturata schematicamente, con l'introduzione del robot mobile MARRtino, realizzato ed utilizzato nell'ambito del progetto e le descrizioni approfondite in merito alle tecnologie utilizzate. La trattazione prosegue descrivendo il lavoro svolto in tutte le fasi, dalla realizzazione del robot, all'implementazione dell'algoritmo nel software che viene utilizzato per effettuare la pianificazione del moto. Successivamente, si pone maggiore attenzione sulla fase di test del caso di studio proposto, riportando risultati sperimentali realizzati dapprima in un ambiente simulato e poi in un ambiente reale, in cui il robot si muove in completa autonomia mediante il metodo dei potenziali artificiali.

Capitolo 2

Stato dell'arte

2.1 Problema canonico della pianificazione

Si consideri un robot mobile \mathcal{B} , costituito da un unico corpo rigido, si assuma inoltre che il robot non sia soggetto a nessun vincolo. Il robot si muove in uno spazio Euclideo $\mathcal{W} = \mathbb{R}^N$, con $N = 2$ o 3 , chiamato *workspace*; si considerino $\mathcal{O}_1, \dots, \mathcal{O}_p$, gli ostacoli fissati in \mathcal{W} , si assume che si conoscano le geometrie di \mathcal{B} e degli ostacoli $\mathcal{O}_1, \dots, \mathcal{O}_p$ e la loro posizione all'interno di \mathcal{W} . Il problema della pianificazione è il seguente: data una configurazione iniziale \mathcal{S} ed una finale \mathcal{G} di \mathcal{B} in \mathcal{W} , trovare se esiste un *percorso*, una sequenza continua di posizioni che guidano il robot attraverso le due configurazioni evitando le collisioni tra \mathcal{B} e $\mathcal{O}_1, \dots, \mathcal{O}_p$; in caso di percorso non esistente si riporta un errore[1].

Nel caso particolare in cui il robot sia un corpo che si muove sul piano, quindi in \mathbb{R}^2 , ci si riferisce al problema canonico come *piano movers' problem*, poichè esplica le difficoltà che affrontano i traslocatori che devono spostare senza sollevare un pianoforte in presenza di ostacoli. Per un corpo rigido che si muove in \mathbb{R}^3 ci si riferisce al problema canonico come *generalized movers' problem*.

Il problema canonico serve a semplificare e ridurre il problema del motion planning al più generale problema geometrico di generare un percorso privo di collisioni. L'assunzione che il robot sia in moto libero non è verificata in sistemi meccanici soggetti a vincoli *anolonomi*, cioè a vincoli che limitano il robot a non seguire in generale cammini arbitrari. Tuttavia, è essenziale introdurre il concetto di **Configuration space** (spazio delle configurazioni), in modo da ottenere una formulazione ottimale del problema canonico [1].

Sebbene il robot debba compiere azioni nel mondo reale, è fondamentale rappresentare il robot come un punto mobile in uno spazio appropriato, le cui coordinate rappresentano la configurazione del robot. Lo *spazio delle configurazioni* \mathcal{C} rappresenta l'insieme di tutte le possibili trasformazioni che il robot può assumere. Le coordinate generalizzate del robot nello spazio sono essenzialmente coordinate cartesiane che esprimono la posizione dei punti nello spazio euclideo e coordinate angolari che rappresentano l'orientamento del robot. Indipendentemente dalla rappresentazione adottata, con matrici di rotazione o quaternioni, la rappresentazione assume valori nel *gruppo speciale ortonormale* $SO(m)$ (con $m = 2, 3$) di matrici reali $m \times m$ con colonne ortonormali il cui determinante è pari a 1.

In generale lo spazio delle configurazioni del robot è quindi ottenuto dal prodotto cartesiano di tali insiemi.

Per un robot mobile modellato come un uniciclo, lo spazio delle configurazioni in \mathbb{R}^2 è un sottoinsieme di $(\mathbb{R}^2 \times SO(2)) \times (\mathbb{R}^2 \times SO(2))$, la dimensione di C è quindi $n = 4$. Se n è la dimensione dello spazio delle configurazioni, una configurazione del robot in C può essere rappresentata come un vettore $\mathbf{q} \in \mathbb{R}^n$.

Per costruire un percorso privo di collisioni occorre definire l'immagine degli ostacoli nello spazio delle configurazioni. Dato un ostacolo \mathcal{O}_i ($i = 1, \dots, p$) in \mathcal{W} , la sua immagine nello spazio delle configurazioni \mathcal{C} è chiamato *C-obstacle*, definito come segue:

$$\mathcal{CO}_i = \{\mathbf{q} \in \mathcal{C} : \mathcal{B}(\mathbf{q}) \cap \mathcal{O}_i \neq \emptyset\}. \quad (2.1)$$

\mathcal{CO}_i rappresenta il sottoinsieme delle configurazioni che provoca una collisione tra \mathcal{B} e l'ostacolo i -esimo \mathcal{O}_i in \mathcal{W} . L'unione di tutti gli ostacoli nello spazio delle configurazioni *C-obstacles* definisce la *regione C-obstacle*:

$$\mathcal{CO} = \bigcup_{i=1}^p \mathcal{CO}_i \quad (2.2)$$

mentre la sua regione complementare

$$\mathcal{C}_{free} = \mathcal{C} - \mathcal{CO} = \left\{ \mathbf{q} \in \mathcal{C} : \mathcal{B}(\mathbf{q}) \cap \left(\bigcup_{i=1}^p \mathcal{O}_i \right) = \emptyset \right\} \quad (2.3)$$

rappresenta lo *spazio libero delle configurazioni*, il sottoinsieme delle configurazioni che non causano collisioni con gli ostacoli. \mathcal{C} è uno spazio连通的, infatti, date due configurazioni arbitrarie esiste un percorso che le collega, tuttavia, lo spazio \mathcal{C}_{free} potrebbe non essere连通的 a causa delle occlusioni causate dagli ostacoli *C-obstacles*. Si noti che l'assenza di vincoli nel problema canonico implica che il robot può intraprendere qualsiasi percorso all'interno di questa regione dello spazio; un percorso nello spazio delle configurazioni è detto *libero* se è interamente contenuto in \mathcal{C}_{free} .

Adesso, alla luce di queste considerazioni, è possibile riformulare il problema canonico della pianificazione. Si assume che le pose iniziali e finali del robot \mathcal{B} in \mathcal{W} siano mappate nelle corrispondenti configurazioni in \mathcal{C} , la configurazione d'origine *start* \mathbf{q}_s e la configurazione finale *goal* \mathbf{q}_g . Pianificare un movimento privo di collisioni per il robot significa generare un percorso sicuro tra \mathbf{q}_s e \mathbf{q}_g se essi appartengono alla stessa componente连通的 della regione \mathcal{C}_{free} , altrimenti si riporta un errore[1].

2.2 Artificial Potential Fields

Un approccio di natura euristica molto efficace per la pianificazione del moto *online* è quello basato sull'uso di *campi di potenziali artificiali*. La prima formulazione dell'approccio che si basa sui potenziali è apparso in[2]. Si tratta di un metodo di pianificazione *reattiva*, in cui il punto che rappresenta il robot nello spazio delle configurazioni \mathcal{C} , viene fatto muovere sotto l'azione di un campo potenziale U , ottenuto

come somma di un potenziale *attrattivo* che spinge il robot verso la destinazione e di un campo *repulsivo* dalla regione *C-obstacle*. La pianificazione avviene in maniera incrementale, ad ogni configurazione \mathbf{q} del robot, la forza artificiale generata dal potenziale è definita come l'antigradiente $-\nabla U(\mathbf{q})$ del potenziale, indicando la direzione di moto localmente più promettente. Questa tipologia di pianificazione appartiene alla categoria dei metodi *single-query*, in quanto la componente attrattiva del potenziale generato dipende da una destinazione prefissata \mathbf{q}_g [1].

2.2.1 Potenziale attrattivo

Il potenziale attrattivo viene impiegato nella navigazione per far giungere il robot nella configurazione del goal \mathbf{q}_g . Al fine di raggiungere questo obiettivo si può usare una funzione potenziale mediante un *paraboloido* con vertice in \mathbf{q}_g

$$U_{a1}(\mathbf{q}) = \frac{1}{2} k_a \mathbf{e}^T(\mathbf{q}) \mathbf{e}(\mathbf{q}) = \frac{1}{2} k_a \|\mathbf{e}(\mathbf{q})\|^2, \quad (2.4)$$

nella quale $k_a > 0$ e $\mathbf{e} = \mathbf{q}_g - \mathbf{q}$ rappresenta il vettore di *errore* rispetto alla configurazione finale \mathbf{q}_g . La funzione ha un minimo globale in \mathbf{q}_g , che vale 0, ed è sempre positiva. La forza attrattiva risultante si definisce

$$f_{a1}(\mathbf{q}) = k_a \mathbf{e}(\mathbf{q}). \quad (2.5)$$

La forza f_{a1} converge linearmente a zero al tendere della configurazione \mathbf{q} alla destinazione del goal \mathbf{q}_g .

Oltre alla funzione potenziale paraboloido è possibile definire una funzione potenziale conica

$$U_{a2}(\mathbf{q}) = k_a \|\mathbf{e}(\mathbf{q})\|. \quad (2.6)$$

Anche quest'ultima risulta sempre positiva e presenta un minimo globale nella configurazione finale \mathbf{q}_g . La forza attrattiva corrispondente a questa funzione è costante in modulo ed è definita come segue:

$$f_{a2}(\mathbf{q}) = -\nabla U_{a2}(\mathbf{q}) = k_a \frac{\mathbf{e}(\mathbf{q})}{\|\mathbf{e}(\mathbf{q})\|}. \quad (2.7)$$

La forza relativa alla funzione potenziale conica costituisce un vantaggio rispetto alla forza f_{a1} generata dal campo attrattivo paraboloidico, in quanto tende a crescere indefinitamente all'allontanarsi dalla configurazione \mathbf{q}_g . Si noti che f_{a2} non è definita in \mathbf{q}_g . La figura 2.1 mostra la forma che assumono i potenziali attrattivi U_{a1} e U_{a2} nel caso $\mathcal{C} = \mathbb{R}^2$, con $k_a = 1$.

Una soluzione ideale è combinare i vantaggi dei due potenziali utilizzando il potenziale attrattivo mediante la funzione conica, lontano da \mathbf{q}_g , e come un paraboloido in prossimità di \mathbf{q}_g . Collocando quindi la transizione tra i due potenziali in corrispondenza del vettore $\|\mathbf{e}(\mathbf{q})\| = 1$, cioè alla sfera di raggio unitario centrata in configurazione \mathbf{q}_g , si ottiene una forza attrattiva continua per qualsiasi valore della configurazione \mathbf{q} del robot[1].

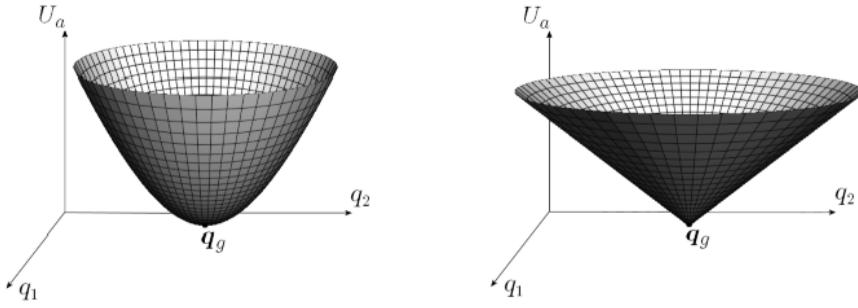


Figura 2.1. Profilo dei potenziali: attrattivo paraboloidico U_{a1} (sinistra), conico U_{a2} (destra) nel caso $\mathcal{C} = \mathbb{R}^2$, con $k_a = 1$. (Figura tratta dal libro [1]).

2.2.2 Potenziale repulsivo

Come viene definito in [2], al fine di evitare che il robot collida con ostacoli posti nell'ambiente, al potenziale attrattivo U_a viene sommato un potenziale repulsivo U_r . Questo permette di effettuare correttamente *obstacle avoidance*, quindi respingere il robot lontano dalla regione degli ostacoli *C-obstacles*, attorno alla quale viene costruito il potenziale di repulsione. Si assume per semplicità nella trattazione che gli ostacoli nella regione *C-obstacles* siano tutti poligoni convessi. Per ogni componente convessa decomposta in \mathcal{CO}_i , $i = 1, \dots, p$, si definisce un potenziale repulsivo

$$U_r = \begin{cases} \frac{k_{r,i}}{\gamma} \left(\frac{1}{\eta_i(\mathbf{q})} - \frac{1}{\eta_{0,i}} \right)^\gamma & \text{se } \eta_i(\mathbf{q}) \leq \eta_{0,i} \\ 0 & \text{se } \eta_i(\mathbf{q}) > \eta_{0,i} \end{cases}$$

nel quale $k_{r,i} > 0$, $\eta_i(\mathbf{q}) = \min_{\mathbf{q}' \in \mathcal{CO}_i} \|\mathbf{q} - \mathbf{q}'\|$ rappresenta la distanza di \mathbf{q} dalla componente \mathcal{CO}_i . $\eta_{0,i}$ rappresenta il raggio d'influenza della distanza, per il parametro γ una scelta tipica è imporlo pari a 2, ma può assumere valori maggiori. Il potenziale $U_{r,i}$ si annulla al di fuori del raggio d'influenza e tende all'infinito in prossimità dell'ostacolo \mathcal{CO}_i , tanto più è grande il valore dell'esponente γ .

La forza repulsiva che si genera dal potenziale repulsivo $U_{r,i}$ si definisce come

$$f_{r,i}(\mathbf{q}) = -\nabla U_{r,i}(\mathbf{q}) = \begin{cases} \frac{k_{r,i}}{\eta_i^2(\mathbf{q})} \left(\frac{1}{\eta_i(\mathbf{q})} - \frac{1}{\eta_{0,i}} \right)^{\gamma-1} \nabla \eta_i(\mathbf{q}) & \text{se } \eta_i(\mathbf{q}) \leq \eta_{0,i} \\ 0 & \text{se } \eta_i(\mathbf{q}) > \eta_{0,i}. \end{cases}$$

Considerando $\mathcal{C} = \mathbb{R}^2$ e la componente convessa \mathcal{CO}_i poligonale, le curve equipotenziali generate dal potenziale repulsivo $U_{r,i}$, sono formate da rette parallele lungo i lati del poligono, legati da archi di circonferenza lungo i vertici.

La figura 2.2 raffigura nel dettaglio le curve generate da un potenziale repulsivo vicino un ostacolo nel caso appena descritto.

Il potenziale repulsivo complessivo è infine ottenuto come la somma di tutti i singoli potenziali di ogni componente convessa \mathcal{CO}

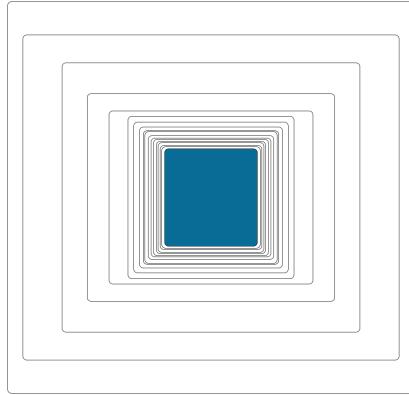


Figura 2.2. Curve equipotenziali generate dal potenziale repulsivo U_r attorno ad un ostacolo.

$$U_r(\mathbf{q}) = \sum_{i=1}^p U_{r,i}(\mathbf{q}). \quad (2.8)$$

2.2.3 Potenziale totale

Si definisce il potenziale totale ottenuto dalla sovrapposizione del potenziale attrattivo e del potenziale repulsivo

$$U_t(\mathbf{q}) = U_a(\mathbf{q}) + U_r(\mathbf{q}) \quad (2.9)$$

al quale corrisponde il campo di forza

$$\mathbf{f}_t(\mathbf{q}) = -\nabla U_t(\mathbf{q}) = \mathbf{f}_a(\mathbf{q}) + \sum_{i=1}^p \mathbf{f}_{r,i}(\mathbf{q}). \quad (2.10)$$

Il potenziale totale U_t così ricavato ha un unico punto di minimo globale nella destinazione \mathbf{q}_g . Un problema ricorrente, di cui questo metodo è affetto, è la comparsa di *minimi locali*, zone d'ombra in cui il campo di forza si annulla, ciò avviene ogni qual volta la risultante delle forze generate dal potenziale in un determinato punto sia nulla.

2.2.4 Minimi locali

Il problema dei minimi locali si presenta ogni qualvolta il potenziale repulsivo genera delle curve equipotenziali con minore curvatura, ciò si traduce in forze attrattive e repulsive, che si annullano in un determinato punto, in vicinanza di un ostacolo, impedendo di fatto che il robot raggiunga la posa di destinazione \mathbf{q}_g ; I metodi di pianificazione basati sui potenziali si dicono *non completi*, proprio perché non garantiscono il raggiungimento della destinazione a causa delle zone d'ombra. Qualunque tecnica di pianificazione adottata che si basa sui potenziali artificiali

presenta questa insidia che non è possibile eliminare del tutto, ma è possibile invece, attuare delle tecniche per far sì che il sistema robotico non entri nel bacino di attrazione di un minimo locale. In situazioni estremamente particolari, è possibile evitare l'introduzione di minimi locali, come nel caso in cui tutte le componenti convesse \mathcal{CO}_i della regione $\mathcal{C}\text{-}obstacles$ siano tutte modellate in forma sferica; in questo caso il potenziale totale U_t presenta dei punti di sella isolati in cui il campo di forza si annulla, ma non vi è la presenza di alcun minimo locale[1].

2.3 Vortex Fields

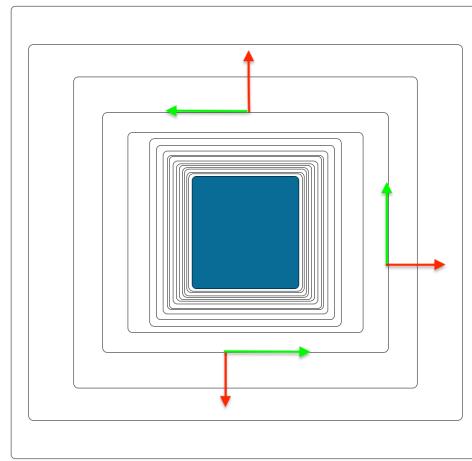


Figura 2.3. Forza repulsiva f_r (in rosso), normale alle curve equipotenziali e forza vorticosa f_v (in verde), tangente alle curve.

Esistono diversi metodi atti ad implementare un meccanismo di evasione dai minimi locali; Sebbene il metodo dei potenziali garantisca *obstacle avoidance*, può risultare insufficiente il suo utilizzo anche in casi molto semplici. L'introduzione di minimi locali è dovuto all'azione repulsiva del potenziale U_r , una soluzione efficiente a tale problema, permette di evitare l'introduzione dei minimi mediante una rivisitazione del metodo dei potenziali artificiali noto come *Vortex fields*.

La tecnica proposta per prima nell'articolo [3], consiste nell'attuare un metodo di pianificazione esente dai minimi, ponendo attorno all'ostacolo un *campo vorticoso* che forzi il robot non solo di evitarlo, ma a costeggiare il $\mathcal{C}\text{-}obstacle$ per raggiungere la sua destinazione, prevenendo del tutto la comparsa di zone d'ombra in cui il robot si blocca. Il metodo rivisitato prevede una tecnica di path planning più aggressiva rispetto al caso dei potenziali. Anche in questo caso non è necessaria la piena conoscenza dell'ambiente, trattandosi di un'ulteriore tecnica di pianificazione online.

Assumendo $\mathcal{C} = \mathbb{R}^2$, si definisce il *campo vorticoso* in \mathcal{CO}_i come un vettore tangente alle curve equipotenziali, invece che normale ad esse:

$$f_v = \pm \begin{pmatrix} \frac{\partial U_{r,i}}{\partial y} \\ -\frac{\partial U_{r,i}}{\partial x} \end{pmatrix}. \quad (2.11)$$

I campi f_r e f_v presentano stessa intensità e differiscono soltanto nel verso, la figura 2.3 mostra la differenza tra i due campi. Il vettore così definito è orientato in senso antiorario, la scelta della direzione del campo per un robot mobile influenza semplicemente la lunghezza del percorso stimato, tuttavia l'orientamento del campo vorticoso deve essere scelto accuratamente in modo che il campo totale formato dal campo attrattivo sovrapposto al campo vorticoso non introduca nessuna zona d'ombra, per far ciò è necessario scegliere in base al punto d'ingresso del robot nell'area d'influenza del *C-obstacle*. Quando viene evitato l'ostacolo è necessario effettuare un *rilassamento* del campo vorticoso per evitare che il robot orbiti indefinitamente attorno all'ostacolo e resti incastrato nel suo bacino d'attrazione. Questo problema viene risolto considerando l'angolo tra il vettore del gradiente attrattivo e il vettore vorticoso, che tende a zero durante il passaggio attorno all'ostacolo, permettendo al robot di fugare il campo vorticoso e continuare il percorso in direzione del *goal* [3].

Capitolo 3

Background

3.1 MARRtino



Figura 3.1. Robot mobile MARRtino: modello realizzato e utilizzato nell'ambito di questo progetto.

MARRtino è una piattaforma robotica open source, versatile e low-cost, messa a disposizione dal Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti (DIAG). La piattaforma, sviluppata in ambito dei corsi del MARR (Master in Artificial Intelligence and Robotics), è un *wheeled mobile robot* basato sul framework open source **ROS** (Robot Operating System).

Il sistema robotico adotta la struttura cinematica di un uniciclo, rispettando questo modello, è possibile realizzarlo in diversi materiali e personalizzarlo nell'aspetto; la configurazione del robot può essere adattata alle molte esigenze applicative.

Le componenti meccaniche ed elettroniche essenziali del robot sono elencate di seguito, in alcuni casi è possibile scegliere tra più opzioni tra quelle elencate:

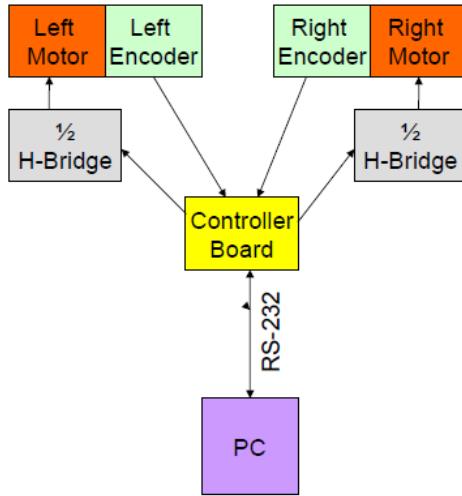


Figura 3.2. Schema logico del robot MARRtino.

- Chassis realizzabile in diverse forme e materiali (legno, plastica, alluminio)
- Scheda logica Genuino Mega 2560 Rev. 3 o SainSmart Mega2560 R3 ATmega2560-16AU
- Arduino Motor Shield R3 o Double H-Bridge L298N
- Due motori Pololu 2825 70:1 o SainSmart 29:1 Metal Gear
- Supporti motore Pololu L-Bracket
- Due ruote Pololu 3281 Scooter wheel 144mm
- Una ruota caster
- Una batteria da 12 V, 7.2 Ah

ulteriori componenti prevedono l'impiego di sensori di profondità o sonar posizionati a bordo del robot.

A bordo del robot viene posto, inoltre, un PC, che comunica con il controllore logico tramite un connettore seriale; Il PC è parte integrante del robot, si occupa dell'esecuzione del software, della gestione dei processi e del coordinamento dei dati acquisiti dai sensori e le azioni da eseguire; in alternativa è possibile montare un calcolatore single-board come il Raspberry Pi 3 Model B.

Il computer per poter comunicare con MARRtino deve necessariamente utilizzare il sistema operativo Linux Ubuntu 16.04(Xenial), in modo da poter integrare il framework ROS, versione Kinetic. Sul controllore (scheda logica Arduino Mega) deve essere installato il firmware, disponibile sul sito.

3.1.1 Firmware

Ogni volta che viene lanciato il nodo (processo) *Orazio_robot_node* che avvia il robot, il controllore esegue il firmware del MARRtino, un programma *event-driven* scritto in **C** sviluppato dal Prof. Giorgio Grisetti. Il firmware esegue un controllore PID su entrambe le ruote, integra l'odometria, implementa un *watchdog* ed ha il compito di comunicare dati con il computer. Il PC invia periodicamente al controllore le velocità desiderate (traslatorie/angolari), il controllore, invece, invia al PC pacchetti che contengono informazioni sull'odometria e sullo stato della batteria; se il controllore non riceve dati dal PC per un certo intervallo di tempo, setta la velocità a zero per arrestare il robot come misura di sicurezza.

3.2 ROS



ROS (The Robot Operating System) è un framework open-source progettato per semplificare lo sviluppo di software per molte piattaforme robotiche. Si tratta di un *middle-ware*, una collezione di librerie e strumenti utile ad implementare funzioni complesse e robuste[4]. Il framework fornisce una pletora di strumenti per la gestione dei processi chiamati *nodi*, per la definizione dei messaggi scambiati dal sistema, fornisce un file system integrato e un build system per la compilazione del codice chiamato Catkin.

I punti di forza di ROS sono: modularità (scalabilità), integrazione di molte librerie open-source, indipendenza dal linguaggio di programmazione, organizzazione in packages degli eseguibili (nodi) sotto un *workspace*, facilità di testing. Il framework si basa su un modello di comunicazione tra processi (*message passing*), i quali comunicano tra loro scambiandosi *messaggi*; uno stream di messaggi dello stesso tipo è organizzato in *topic*.

La comunicazione avviene principalmente mediante il paradigma *publisher/subscriber* in cui un processo sottoscrive un topic per ricevere dati e pubblica su un altro topic i dati elaborati per tradurli in output.

Le modalità di comunicazione in ROS sono gestite dal *roscore*, una collezione di nodi e istanze che garantiscono la comunicazione tra processi e va dunque avviato prima dell'esecuzione di ogni altro nodo. Una volta avviato, vengono avviati automaticamente i processi responsabili della gestione delle componenti del framework e del monitoraggio degli stessi; tramite il comando *roscore* da terminale (bash su Linux), si avviano i processi *Master*, Parameter server e *rosout*. Ogni nodo attivo è identificato dal proprio nome e deve registrarsi al ROS master prima di intraprendere qualsiasi azione. La lista di tutti i nodi e topic attivi a run-time formano un grafo; la struttura complessiva può essere rappresentata come un grafo in cui i nodi sono i processi attivi, connessi ai restanti attraverso gli archi rappresentati dai topic.

Il Master è di rilevante importanza per la rete di comunicazione interna tra processi, in quanto tiene traccia dei nodi e dei topic attivi, di publishers e subscribers ai vari topic e permette ad un nodo di essere individuato dagli altri, inoltre esso mette in relazione *peer-to-peer* i nodi comunicanti.

Un nodo in ROS può essere scritto in diversi linguaggi, tuttavia sono supportati ufficialmente i linguaggi *object-oriented* C++ e Python, per i quali ROS provvede a fornire dei pacchetti e delle librerie per interfacciare il codice al framework correttamente; rispettivamente sono forniti i pacchetti *roscpp* per il linguaggio C++ e *rospy* per Python. I pacchetti forniscono la classe *NodeHandle* per la corretta gestione del nodo, per avviarlo e fermarlo all'interno di un programma roscpp (o rospy) ed altre interfacce per definire gli scambi di messaggi, la sottoscrizione e la pubblicazione sui topic.

ROS supporta la programmazione multi-thread, ma non specifica un modello preciso di threading per l'applicazione, nel caso di un programma single-thread in ROS è possibile definire un nodo che esegue operazioni mediante delle funzioni callback, che non termineranno la loro esecuzione fin quando non muore il processo o fin quando non viene inserita da terminale la combinazione di tasti **Ctrl+c**; roscpp supporta ogni tipo di callback supportata dalla libreria C++ Boost: funzione, metodo di classe, oggetto funtore.

Per effettuare la compilazione di un programma ROS è necessario includere nella compilazione che avviene tramite il comando dal terminale **catkin_make**, due ulteriori files, un file di testo *CMakeList* che specifica tutte le librerie e dipendenze che devono essere linkate per la corretta compilazione ed esecuzione del programma, un file package.XML che provvede alla definizione di proprietà del package.

3.3 Percezione con sensori RGBD

Il MARRtino, supporta diverse tipologie di sensori per acquisire le percezioni dell'ambiente di lavoro, tramite l'integrazione del framework ROS; i sensori RGBD (componente RGB, componente di profondità Depth) si prestano egregiamente alla raccolta dati e alla successiva elaborazione.

Questa tipologia di sensori nasce come un'evoluzione della camera monoculare RGB, che misura l'intensità della luce proiettata in un sensore attraverso un sistema di lenti o specchi (come avviene ad esempio nelle fotocamere *Reflex* digitali), proiettando l'ambiente tridimensionale in un'immagine bidimensionale. Tuttavia a causa di questa trasformazione, si viene incontro ad una perdita di informazione essenziale che rende inosservabile la profondità. Le camere RGBD a tal proposito utilizzano, in aggiunta alla camera monoculare, una speciale telecamera con sistema a luce strutturata che effettua una scansione dell'ambiente. Questo permette di percepire la profondità, attraverso triangolazione stereo (due telecamere in posizioni differenti) o attraverso *time of flight* (una telecamera unica che funziona da emittente e ricevente); in questo modo, l'acquisizione dei punti sul sensore, restituisce uno spazio euclideo tridimensionale, che permette di localizzare oggetti dell'ambiente in base alla profondità rilevata dal sensore.

Per le finalità richieste dal progetto, si è optato per un sensore RGBD economico, il sensore **Microsoft Kinect** mostrato in figura 3.3, dotato di una telecamera RGB

con risoluzione 640×480 , di una telecamera a radiazione infrarossa e di uno scanner 3D a luce strutturata per il calcolo della profondità, con un frame-rate pari a 30 fps. La Kinect è un sensore sviluppato da Microsoft come accessorio di motion detection



Figura 3.3. Sensore RGBD Kinect V. 1.0

per la console videoludica Xbox 360, tuttavia, grazie alle sue caratteristiche e lo sviluppo di driver open-source è possibile utilizzarla al di là dello scopo per cui è stata originariamente concepita. Collegando il sensore al PC è possibile acquisire i dati in input attraverso driver specifici come **OpenNI** e renderli disponibili all'elaborazione in ROS attraverso i topic.

Il sensore presenta delle limitazioni notevoli che devono essere tenute in considerazione durante l'acquisizione dei dati, la figura 3.4 evidenzia il campo di visione della Kinect che risulta essere molto limitato, la limitazione più significativa è rappresentata dalla distanza minima di operatività, ovvero la minima distanza dalla quale riesce ad acquisire la percezione dell'ambiente.

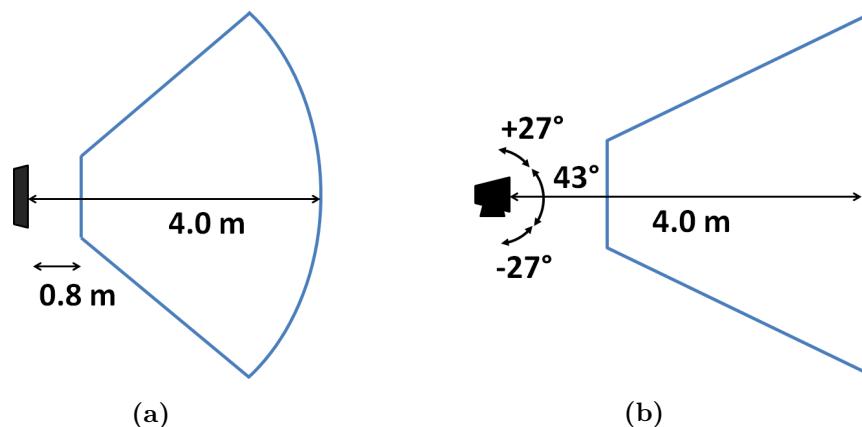


Figura 3.4. Angular field of view: Orizzontale in 3.4a, verticale in 3.4b.

I dati acquisiti dalla Kinect formano un insieme di punti multidimensionali, caratterizzati dalla loro posizione nello spazio e da metadati, come colore RGB associato ad ogni singolo punto, valore d'intensità, profondità. L'insieme di questi punti

viene definito *point cloud* (nuvola di punti) e può essere visualizzato con l'ausilio di apposite librerie e immagazzinato in ROS nell'omonima struttura dati.

3.4 Librerie per lo sviluppo del progetto

3.4.1 Eigen

Eigen è una libreria templatistica implementata in C++ anch'essa open source dedicata all'algebra lineare, utile per effettuare operazioni su vettori e matrici. Eigen è nota per essere una libreria versatile e veloce che raccoglie algoritmi efficienti e fornisce classi templatistiche per la definizione di matrici di dimensione arbitraria, matrici sparse e vettori, supporta inoltre la vettorizzazione usando il set di istruzioni della corrispondente architettura della CPU usata dal calcolatore. Ciò permette di velocizzare la computazione, soprattutto nel caso di matrici di dimensione molto grande. Eigen è una libreria che dipende esclusivamente dalla libreria standard C++, dunque per usare le sue funzioni è necessario solamente includere nel codice sorgente gli headers contenenti i templates[5]. La libreria usa come convenzione la rappresentazione di matrici in formato *column-major*.

3.4.2 Point Cloud Library

I punti che costituiscono una point cloud sono rappresentati nello spazio da coordinate cartesiane (x, y, z), è possibile acquisire una scansione tridimensionale dell'ambiente e ricostruire la superficie scansionata attraverso sensori di profondità come i laser scanner 3D, stereo camera, camera RGBD. Le point cloud rappresentano uno strumento efficace per la gestione dei dati e le molteplici applicazioni in diversi campi, in Robotica vengono impiegate per implementare le funzioni di navigazione, *obstacle avoidance*, object recognition, mapping di un ambiente, motivo per cui in molti sistemi robotici vengono installati potenti sensori in grado di acquisirle.

La gestione dei dati è effettuata mediante la Point Cloud Library (PCL), una libreria open source per il processamento di immagini 2D/3D e point cloud che offre numerosi algoritmi e funzioni. Il framework scritto in C++ è **cross-platform** ed è suddiviso in molte librerie più piccole rese disponibili separatamente, questa modularità permette la distribuzione della libreria anche su calcolatori con ridotta capacità di calcolo[6]; la PCL dipende da altre librerie, tra cui Boost, Eigen, OpenMP.

La libreria è interamente supportata da ROS, attraverso il driver OpenNI permette l'acquisizione dei dati provenienti dalla Kinect e il salvataggio diretto nella struttura dati **PointCloud** che immagazzina i dati in opportuni campi in un ordine prefissato.

PointCloud è una classe templatistica C++ che contiene campi dati adibiti a memorizzare una point cloud, la classe contiene sostanzialmente i campi:

- **width(int)** definisce la larghezza del dataset espressa in numero di punti,
- **height(int)** definisce l'altezza del dataset espressa in numero di punti
- **points(std::vector <pointT>)** contiene l'array di dati contenenti tutti i punti di tipo generico PointT.

Il framework mette a disposizione anche un’ulteriore classe **PointCloud2**, contenente una definizione generale della struttura point cloud per salvare, caricare e inviare il dataset come messaggio in ROS.

Le nuvole di punti vengono gestite in C++ mediante gli *smart pointers*, mentre la conversione tra i due frameworks PCL e ROS è possibile grazie alle funzioni della libreria `pcl::fromROSMsg` e `pcl::toROSMsg`.

3.4.3 OpenCV

OpenCV (Open Source Computer Vision Library) è una libreria open source di computer vision e machine learning che fornisce algoritmi ottimizzati e strutture per l’elaborazione in tempo reale. Anche questa libreria sviluppata in C++ è multi-piattaforma, presenta una struttura modulare, supporta molti sistemi operativi e fornisce le API per le interfacce di diversi linguaggi come C, C++, Java, Python e MATLAB [7]. Gli algoritmi forniti dalla libreria possono essere usati per effettuare il processamento di immagini, trasformazione, filtraggio, face recognition, object identification, tracciamento di movimenti, segmentation, Human-Computer Interaction e si prestano anche a produrre point cloud 3D. OpenCV è inoltre la libreria usata come package primario in ROS per la gestione delle immagini e delle point cloud. Le immagini in OpenCV sono rappresentate da matrici contenenti tutti i punti dell’immagine, supporta la gestione di immagini in scala di grigi e a colori, con la precisazione che l’ordine dei canali delle componenti RGB è invertito (BGR). L’acquisizione dei dati RGBD provenienti dalla Kinect è supportata per mezzo di un bridge implementato nella libreria, che permette di visualizzare e salvare i dati in input.



(a) Eigen



(b) PCL



(c) OpenCV

Figura 3.5. Librerie open source utilizzate per l’implementazione dell’algoritmo.

Capitolo 4

Implementazione algoritmo Artificial Potential Fields

4.1 Progettazione

L'implementazione dell'algoritmo dei potenziali artificiali è stato organizzato secondo uno schema modulare, permettendo facilmente l'introduzione di modifiche durante la fase di progettazione e di velocizzare la fase di testing. Il progetto è stato sviluppato interamente in C++, con l'ausilio di software libero, delle librerie integrate nel framework ROS, della libreria indispensabile Boost e delle librerie open source appena descritte. L'impiego di queste risorse ha permesso di tradurre l'algoritmo in codice, acquisire, elaborare i dati e restituirli in tempo reale agli encoders dei motori del robot per effettuare uno spostamento, il tutto in totale autonomia.

I potenziali artificiali per i wheeled mobile robot sono generati e tradotti in input di velocità e infine inviati ai motori delle ruote, raccogliendo costantemente dati dal sensore di profondità e generando la percezione dello spazio di lavoro, eseguendo la corretta suddivisione delle regioni $\mathcal{C}_{obstacle}$ e \mathcal{C}_{free} .

Il software implementato nel framework ROS consta di due nodi, entrambi publisher e subscriber, distribuiti in sei files, rispettivamente:

- Due files header che comprendono l'importazione di tutte le librerie, definizione delle classi pubbliche e protected, dichiarazione di funzioni e variabili,
- due files contenenti librerie di funzioni create appositamente per l'elaborazione dell'input dal sensore e l'implementazione dell'algoritmo dei potenziali, inizializzazione lista attributi delle classi,
- due files contenenti l'inizializzazione dei nodi e le funzioni *main* che si occupano dell'esecuzione dell'algoritmo.

Il diagramma in figura 4.1 mostra lo schema dei nodi e ne esalta i principi del funzionamento secondo le convenzioni di comunicazione in ROS attraverso i topics.

Il nodo *Obstacles_mapper_2D* sottoscrive il topic `"/camera/depth/points"` per acquisire la point cloud dal sensore Kinect, esegue l'elaborazione ed il filtraggio dei punti, genera una matrice bidimensionale contenente la descrizione degli ostacoli

sul piano e pubblica i dati in output su un nuovo topic `"/camera/obstacles2D"`. Il nodo `APF_planner` sottoscrive il topic `"/camera/obstacles2D"`, sul quale vengono pubblicati costantemente i messaggi contenenti informazioni sull'ambiente, esegue l'algoritmo dei potenziali artificiali e pubblica le velocità in output sul topic `"/cmd_vel"`, responsabile del trasporto dei messaggi fino ai motori del robot.

Nel sistema robotico per poter usare correttamente tutti i sensori e attuatori è necessario coordinarli e specificare la configurazione di ognuno di essi. Questo permette di rappresentare correttamente attraverso un sistema di riferimento (frame) globale la posizione dei sensori e degli attuatori a bordo del robot. Nello specifico ogni dispositivo montato sul robot ha una sua configurazione rispetto all'ambiente e al robot stesso. ROS permette di gestire attraverso una struttura ad albero tutte le configurazioni delle componenti, ponendo per convenzione la radice dell'albero su un punto della base mobile; ogni altro dispositivo è collegato attraverso dei *link*, che rappresentano il sistema di riferimento del dispositivo, e dai *giunti* che definiscono la trasformazione che mappa il link del figlio a quello del genitore.

In particolare è rilevante puntualizzare che la nuvola di punti acquisita dalla Kinect utilizza un determinato sistema di riferimento, che deve essere adattato al frame della base dell'uniciclo, in modo da produrre movimenti consoni all'ambiente e alla collocazione degli ostacoli. Si tratta di effettuare una trasformazione (rotazione) di punti di un sistema di coordinate tridimensionale, dal sistema left-handed della Kinect (`"/camera_depth_frame"`) al sistema right-handed della base del robot (`base_link`), questa trasformazione è possibile grazie alla libreria ROS `tf` che fornisce le funzioni atte ad effettuare le suddette trasformazioni.

Se l'unica parte mobile del robot è la base, come nel caso del MARRtino, per tutti i dispositivi a bordo si utilizza il nodo `static_transform_publisher` per effettuare la trasformazione da un frame sorgente a un frame destinazione.

Si osservano di seguito nel dettaglio le operazioni effettuate dai nodi che costituiscono il cuore del progetto.

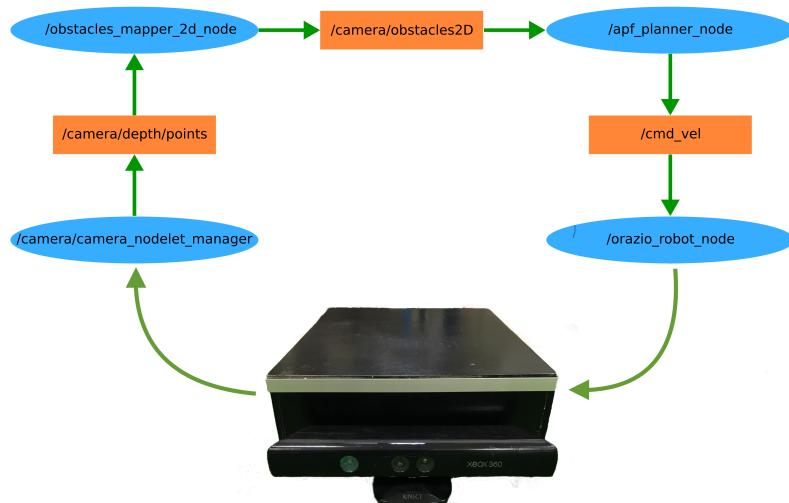


Figura 4.1. Schema organizzativo del progetto: flusso dei dati attraverso i topics.

4.2 Obstacles mapper 2D

Il nodo *Obstacles_mapper2D* descritto dallo pseudocodice 1 è deputato all'acquisizione della nuvola di punti, l'acquisizione avviene alla stessa velocità (frame-rate) con cui il sensore acquisisce le informazioni, mediante una funzione callback. Successivamente, per ogni frame acquisito dal sensore, il nodo effettua le trasformazioni avvalendosi della libreria ROS *tf*, se la trasformazione è effettuabile allora il fotogramma viene ruotato e filtrato lungo l'asse *z* (lungo l'asse dell'altezza). Il filtraggio dei punti è necessario al fine di far percepire al robot gli ostacoli invalidabili, siano essi degli oggetti sull'asse positivo del piano, sia che essi siano buche, scale, al di sotto del piano; conseguentemente il robot deve essere in grado di poter intraprendere un percorso in cui si presentano oggetti che non devono essere considerati impedimenti al passaggio, come piccoli rialzamenti sul piano o lievi discese.

Una volta effettuato il filtraggio dei punti, si procede con la creazione di una matrice di dimensione prefissata che rappresenta la visione dall'alto dell'ambiente, offrendo la conoscenza delle configurazioni libere e di quelle occupate da ostacoli. La suddivisione delle regioni $C_{obstacle}$ e C_{free} nella matrice è effettuata settando le celle della matrice corrispondenti agli ostacoli pari a uno, la celle restanti pari a zero.

La matrice generata attraverso le funzioni della libreria Eigen viene pubblicata sul topic `"/camera/obstacles2D"` sotto forma di messaggio.

4.3 APF Planner

Il nodo *APF_planner* si occupa della pianificazione del percorso osservando i dati provenienti dal nodo *Obstacles_mapper2D*. L'algoritmo dei potenziali sfrutta i dati della matrice per costruire attorno agli ostacoli il potenziale repulsivo, il potenziale attrattivo viene invece costruito in un punto della regione libera. Il goal da raggiungere può essere scelto arbitrariamente osservando la matrice e scegliendo un luogo privo di ostacoli, in questo caso il robot dovrà scegliere il punto libero da raggiungere e fin quando non avrà raggiunto quella configurazione desiderata non terminerà il moto; in alternativa è possibile scegliere un goal a distanza prefissata, per semplicità si è scelta una distanza molto prossima al robot (50 cm) in avanti. La somma delle forze dei campi attrattivo e repulsivo fornisce il percorso più promettente per raggiungere il goal senza collidere con gli ostacoli; inoltre il robot, grazie al filtraggio dei punti della point cloud, è in grado di evitare ostacoli come scale e buche. Si riporta la descrizione dell'algoritmo in pseudocodice 2.

I parametri dell'algoritmo vengono tutti inizializzati con valori prefissati all'interno della lista attributi del costruttore, è importante scegliere correttamente i valori da assegnare ad ogni parametro, in modo da produrre un moto quanto più soddisfacente per raggiungere il goal.

I potenziali sono calcolati separatamente in funzioni dedicate, ognuna delle quali restituisce le velocità lineari e angolari con il tipo di dato della libreria ROS **geometry_msgs::Twist**. Le velocità totali vengono pubblicate infine sul topic `"/cmd_vel"` che a sua volta viene sottoscritto dal nodo *orazio_robot_node*, che si occuperà di trasportare gli impulsi al sistema robotico, producendo gli spostamenti dettati dalle velocità.

Algorithm 1 Obstacles mapper 2D node

```

1: function SUBSCRIBE
2:   Subscribe to Kinect topic "camera/depth/points"
3: end function
4: function TRANSFORM(base_link, camera_depth_optical)
5:   target frame "base_link"  $\leftarrow$  source Kinect frame "camera_depth_optical"
6: end function
7: function POINTCLOUDCALLBACK
8:   function ACQUIREPOINTCLOUD(point_cloud_input)
9:     point_cloud  $\leftarrow$  point_cloud_input
10:  end function
11:  function FILTERPOINTCLOUD(point_cloud)
12:    ApplyTransform()
13:    ApplyFiltering(point_cloud, Transformed_cloud)
14:    return Transformed_cloud
15:  end function
16:  function CREATEMATRIX(Transformed_cloud)
17:    matrix(x, y)  $\leftarrow$  0
18:    for all points in Transformed_cloud do
19:      matrix(x)  $\leftarrow$  Transformed_cloud(point.x)
20:      matrix(y)  $\leftarrow$  Transformed_cloud(point.y)
21:    end for
22:    return matrix(x, y)  $\leftarrow$  1
23:  end function
24:  function CONVERTEIGENTOMESSAGE(matrix)
25:    Applyconversion(cast)
26:    return Message  $\leftarrow$  matrix
27:  end function
28:  function PUBLISH
29:    Publish matrix on topic "camera/Obstacles2D"
30:  end function
31: end function

```

Algorithm 2 APF Planner node

```

1: function SUBSCRIBE
2:   Subscribe to "camera/obstacles2D"
3: end function
4: function APF_CALLBACK(matrix(x, y))
5:   function ATTRACTIVE_POTENTIAL(robot_x, robot_y, goal_x, goal_y)
6:     vector robot_to_goal  $\leftarrow$  (goal_x - robot_x, goal_y - robot_y)
7:     e  $\leftarrow$  robot_to_goal.ApplyNorm
8:     if  $e \leq \rho$  then                                 $\triangleright$  Paraboloidal
9:       attractivex,y  $\leftarrow k_{att} \times robot\_to\_goal$ 
10:    else                                          $\triangleright$  Conical
11:      attractivex,y  $\leftarrow k_{att} \times robot\_to\_goal/e$ 
12:    end if
13:    attractive_θ  $\leftarrow k_\theta \times \arctan(attractive_{x,y})$ 
14:    return attractive_velocities
15:  end function
16:  function REPULSIVE_POTENTIAL(robot_x, robot_y, obs_x, obs_y)
17:    vector robot_to_obstacle  $\leftarrow$  (obs_x - robot_x, obs_y - robot_y)
18:    ηi  $\leftarrow$  robot_to_obstacle.ApplyNorm
19:    if  $\eta_i \leq \eta_{0,i}$  then
20:      repulsivex,y  $\leftarrow k_{r,i}/\eta_i^2 \times (1/\eta_i - 1/\eta_{0,i})^{\gamma-1}$ 
21:    else
22:      repulsivex,y  $\leftarrow 0$ 
23:    end if
24:    repulsive_θ  $\leftarrow k_\theta \times \arctan(repulsive_{x,y})$ 
25:    return repulsive_velocities
26:  end function
27:  function VORTEX_POTENTIAL(robot_x, robot_y, obs_x, obs_y)
28:    vector robot_to_obstacle  $\leftarrow$  (obs_x - robot_x, obs_y - robot_y)
29:    ηi  $\leftarrow$  robot_to_obstacle.ApplyNorm
30:    if  $\eta_i \leq \eta_{0,i}$  then
31:      vortexx,y  $\leftarrow k_{v,i}/\eta_i^2 \times (1/\eta_i - 1/\eta_{0,i})^{\gamma-1}$ 
32:    else
33:      vortexx,y  $\leftarrow 0$ 
34:    end if
35:    vortex_θ  $\leftarrow k_\theta \times \arctan(vortex\_potential_y, -vortex\_potential_x)$ 
36:    return vortex_velocities
37:  end function
38:  function TOTAL_POTENTIAL
39:    total_velocities  $\leftarrow$  attractive_velocities + repulsive_velocities
40:    return total_velocities
41:  end function
42: function PUBLISH
43:   Publish total velocities on topic "cmd_vel"
44: end function

```

Oltre alla funzione *Repulsive_potential* è stata implementata un’ulteriore funzione *Vortex_potential* per sostituire le forze repulsive e garantire che il robot non sia soggetto ad arresto in punti di minimi locali. La selezione tra le due funzioni deve essere effettuata prima di avviare il software, grazie ad un parametro speciale che permette di decidere quale funzione eseguire per la generazione del potenziale repulsivo o vorticoso.

Un’ulteriore considerazione riguardo la suddivisione delle regioni nella matrice dell’ambiente, sta di fatto nel raggruppamento degli ostacoli in **blobs**, ovvero nel raggruppare tutte le celle della matrice che rappresentano un ostacolo, in modo da calcolare una volta soltanto il potenziale, piuttosto che su tutti i punti dell’ostacolo, ciò si traduce in un’implementazione dell’algoritmo leggermente più efficiente.

Il codice sorgente è rilasciato con licenza open source MIT ed è possibile scaricarlo dalla repository del progetto su GitHub [8].

Capitolo 5

Risultati sperimentali

I risultati delle esperienze sono stati raccolti effettuando numerosi test in un ambiente *non strutturato*, ottenendo tuttavia risultati promettenti, esattamente in linea con le aspettative previste dalla soluzione al problema del *motion planning* con tale algoritmo. In particolare si evince che l'algoritmo dei potenziali introduce inevitabilmente punti di minimi locali, in cui la risultante delle forze generate dai campi di potenziale è pari a zero. In questa situazione, seppur prevista dall'algoritmo, non c'è alcuna possibilità di uscire fuori dalla zona d'ombra in maniera autonoma; nel caso del *Vortex Field* questa problematica è elusa dall'algoritmo, che a differenza del primo, non introduce alcun punto di minimo, il robot è sempre in grado di aggirare l'ostacolo senza fermarsi, raggiungendo il goal prefissato.

Gli ambienti in cui sono stati raccolti i dati sono sia reali che virtuali, avvalendosi di un simulatore offerto dal framework ROS, **Stage**, è stato possibile simulare un ambiente strutturato in cui il robot esegue il moto ed effettua la *pianificazione* del percorso con riferimento al goal da raggiungere. Una volta effettuati i test in simulazione, è stato altresì possibile testare il robot negli ambienti messi a disposizione dal dipartimento, in particolare il laboratorio di Robotica, in cui si è simulato un percorso con ostacoli di vario genere per testare l'algoritmo e le probabilità di fallimento in prossimità di un ostacolo. Sebbene sia stato possibile effettuare questo caso di studio, il robot è stato posto dinanzi a delle sfide ancora più ardue, come riconoscere ostacoli al di sotto del piano in cui opera reagendo egregiamente. A tal proposito si è posto il robot MARRtino in prossimità di scale, correttamente riconosciute come un impedimento al proseguimento della pianificazione in quella direzione, ma anche di fronte a cavità sul piano di forma irregolare, altrettanto trattate come ostacoli, dimostrando così che il robot è in grado di pianificare un percorso sicuro.

Le variazioni di velocità, pubblicate sul topic `"/cmd_vel"`, sono raccolte e visualizzate attraverso il tool ROS **rqt_plot**, che genera un grafico spazio/tempo in cui viene riportato il tempo lungo l'asse delle ascisse e l'intensità della velocità, espressa in *m/s*, lungo l'asse delle ordinate.

Come si apprende dalle figure 5.1 e 5.2 che mostrano il grafico delle velocità, il moto del robot durante la navigazione presenta delle variazioni di velocità lineari e angolari al di sopra dello zero.

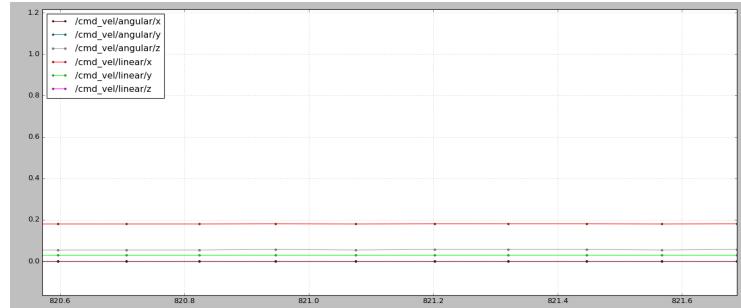
La figura 5.3 e la figura 5.4, invece mostrano il caso in cui il robot sia soggetto

ad una forza contraria rispetto all'ostacolo, in cui è sottoposto ad una variazione di velocità in direzione opposta, essendo le forze normali alle curve di equipotenziale. Le velocità corrispondenti scendono al di sotto dello zero per poter permettere al MARRtino di tornare indietro e successivamente intraprendere un percorso più promettente, lontano dall'ostacolo 5.5 .

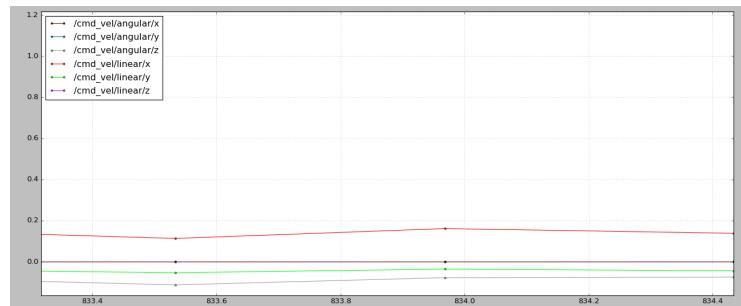
Il caso di studio prevede anche la dimostrazione della presenza di minimi locali, in questo caso la figura 5.6 mostra come le velocità siano prossime allo zero, non permettendo di fatto alcun movimento: le forze attrattive e repulsive sommate, si annullano, cadendo in un minimo locale come si è giunti nella trattazione nel capitolo 2, si riporta in figura 5.7 la visione dell'ambiente con le relative forze di potenziale che si annullano, impedendo il movimento in qualsiasi direzione.

Si riporta infine per semplicità la visione dall'alto dell'ambiente in figura 5.8a utilizzando il metodo Vortex Fields, nella matrice generata si denotano le forze del potenziale che sono dirette in direzione antioraria attorno all'ostacolo. Le forze così definite costringono il robot ad effettuare un moto lungo le curve di equipotenziale senza mai incontrare punti di minimi locali attorno al *C-obstacle*, le variazioni di velocità sono irrilevanti, in quanto simili ai casi già descritti. L'immagine 5.8b mostra la point cloud associata alla visione dall'alto già vista nella matrice, i falsi colori indicano la profondità dell'oggetto: le tonalità calde rappresentano punti vicini al robot, quelle fredde punti distanti.

I video che raccolgono gli esperimenti descritti sono raccolti in [9].



(a) Variazione velocità durante il moto.



(b) Variazione di velocità durante il moto 2.

Figura 5.1. Grafico delle velocità durante il moto in un determinato intervallo di tempo.

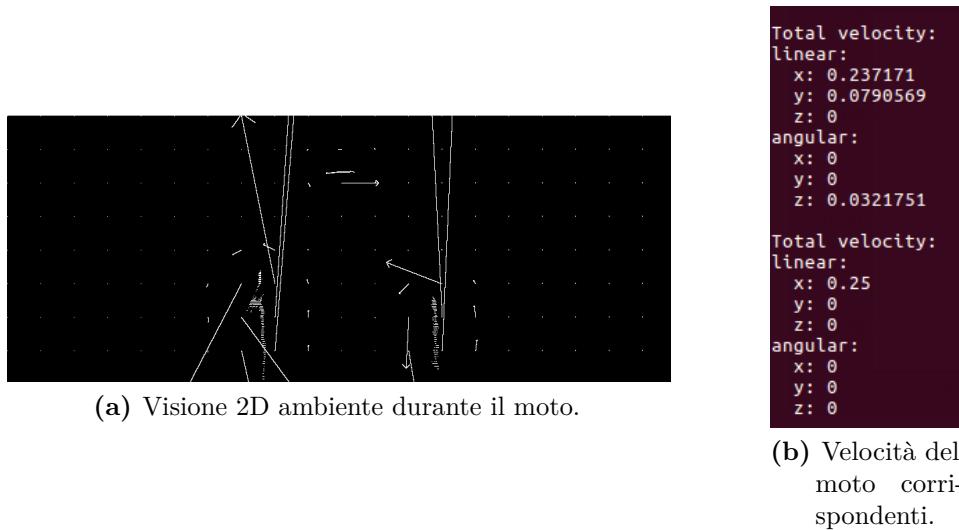


Figura 5.2. Visualizzazione ambiente dall’alto con forze del potenziale totale e rispettive velocità del moto.

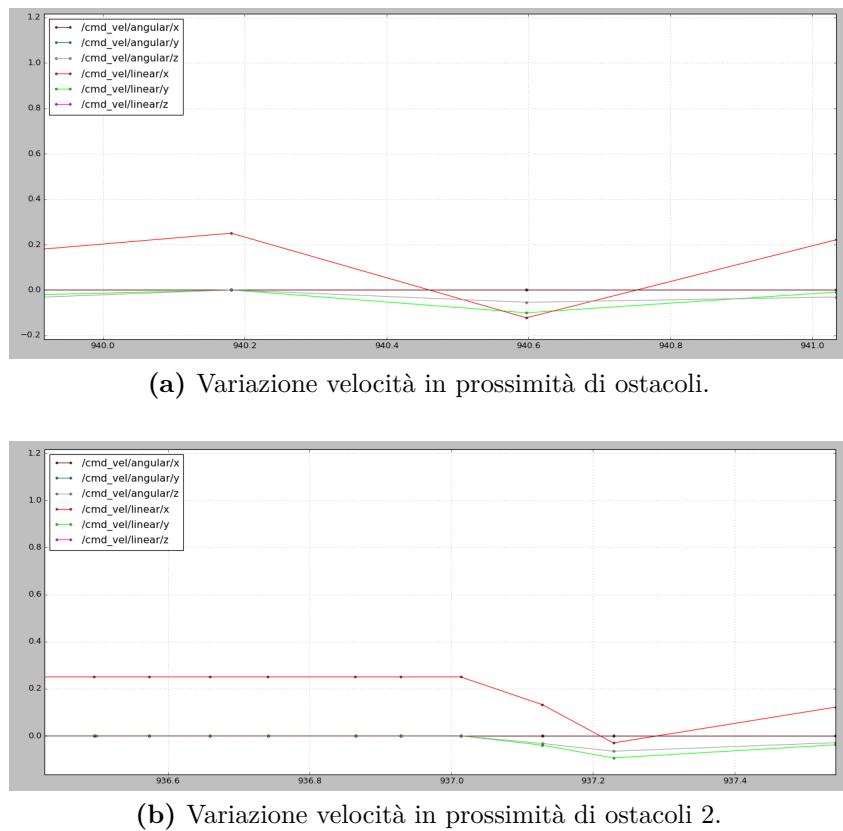


Figura 5.3. Grafico delle velocità in presenza di ostacoli in un determinato intervallo di tempo.

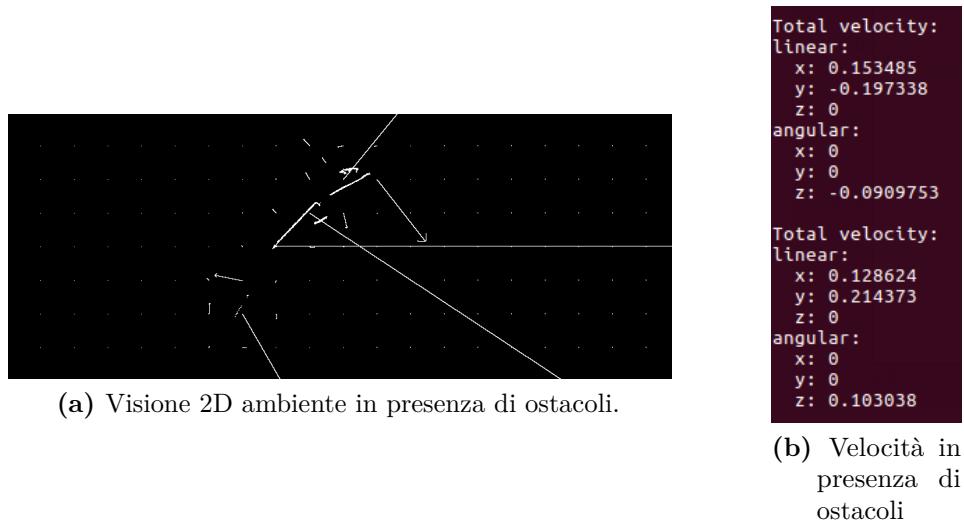


Figura 5.4. Visualizzazione ambiente dall’alto con forze del potenziale totale in prossimità di ostacoli e velocità risultante.

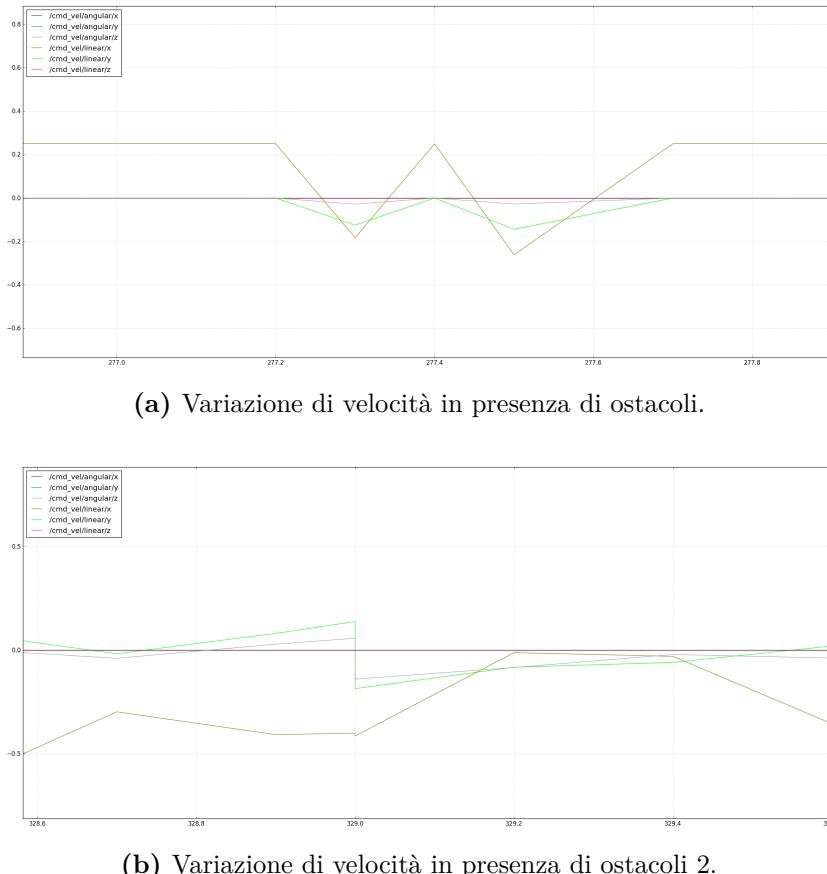
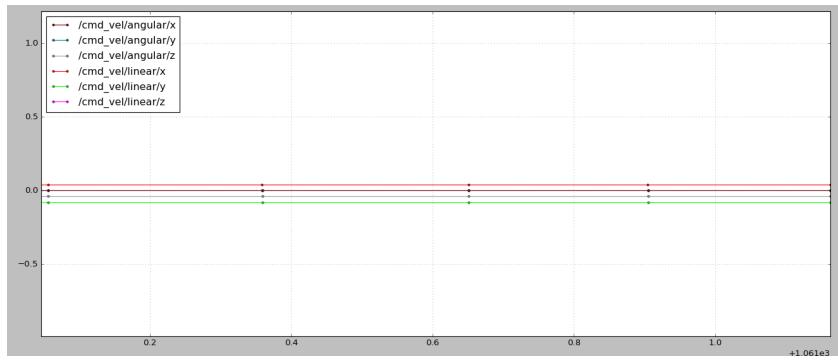
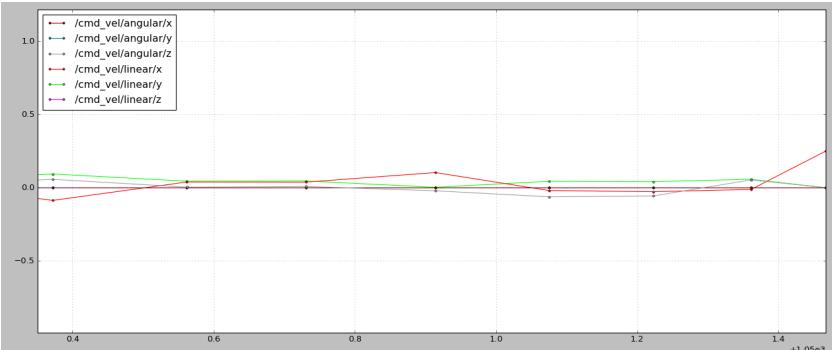


Figura 5.5. Grafico delle velocità durante la pianificazione in presenza di ostacoli in un determinato intervallo di tempo.

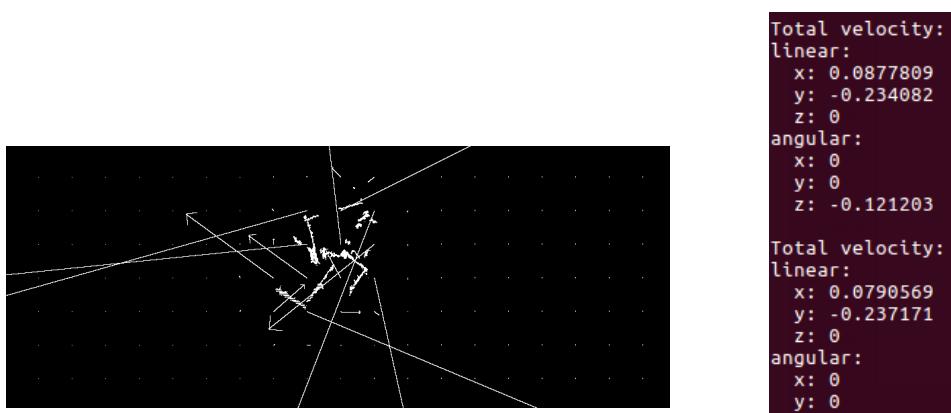


(a) Velocità in presenza di minimo locale.



(b) Velocità in presenza di minimo locale; tentativo di fuga dal minimo locale.

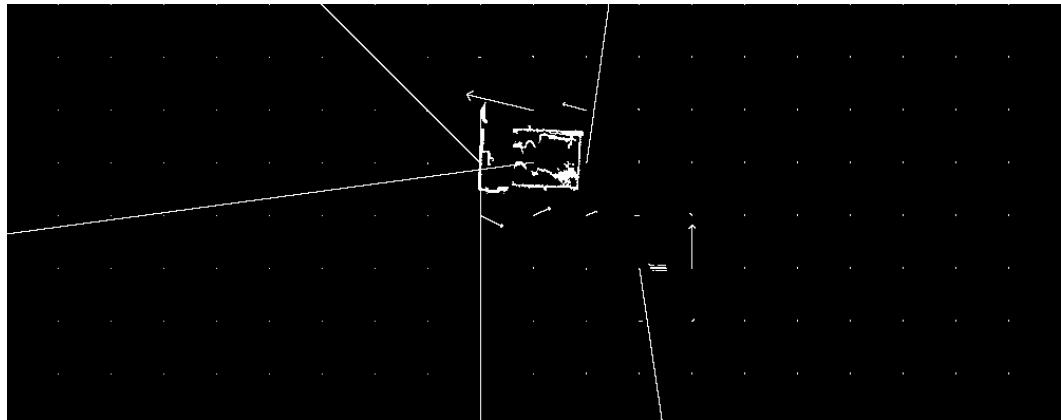
Figura 5.6. Grafico delle velocità in presenza di un minimo locale: velocità prossime allo zero.



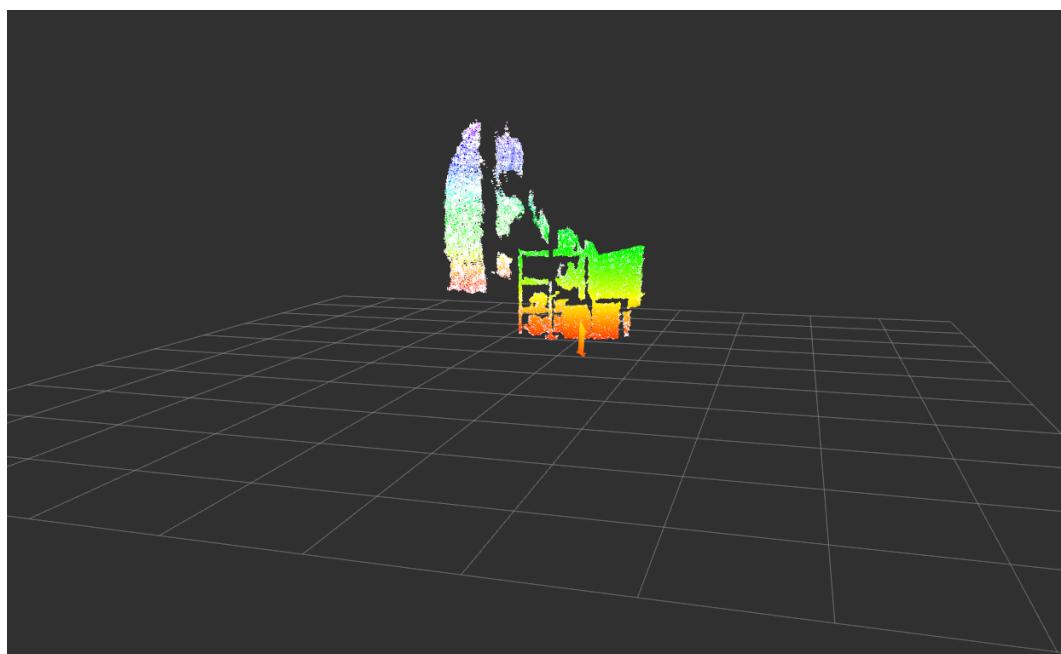
(a) Visione 2D ambiente in presenza di minimo locale.

(b) Velocità in presenza di minimo locale.

Figura 5.7. Visualizzazione ambiente dall'alto con forze del potenziale totale e velocità corrispondenti nella zona d'ombra.



(a) Visione 2D ambiente utilizzando Vortex Fields.



(b) Point cloud corrispondente in RViz.

Figura 5.8. Visualizzazione ambiente dall'alto con forze del potenziale Vortex e Point cloud risultante.

Capitolo 6

Conclusioni

Il progetto prende vita nell'ambito del corso di laboratorio di Intelligenza Artificiale e Grafica Interattiva, erogato durante l'ultimo anno di Laurea triennale in Ingegneria Informatica e Automatica. Il corso ha introdotto e permesso di acquisire le conoscenze necessarie per affrontare diversi problemi nell'ambito dell'IA e della Robotica, permettendo di costruire il robot MARRtino secondo le specifiche dei requisiti descritti nel capitolo 3. Durante il corso sono stati affrontati i temi più importanti che hanno portato alla realizzazione del progetto finale, in particolare l'introduzione ad un nuovo linguaggio (C++), l'introduzione al framework ROS e alle librerie da cui esso dipende, la gestione di dati provenienti da sensori, e la progettazione di sistemi in grado di operare autonomamente.

Per affrontare il problema del *motion planning online* è stato necessario intraprendere lo studio di alcuni elementi del corso *Autonomous and Mobile Robotics*, erogato durante il primo anno di Laurea Magistrale in Intelligenza Artificiale e Robotica (MARR), tenuto dal Prof. G. Oriolo, co-autore di [1] e [3].

La progettazione del software atto ad implementare gli algoritmi proposti, ha permesso il movimento autonomo mediante i soli dati acquisiti costantemente dal sensore, con molte limitazioni intrinseche, in quanto non ideato per questo scopo e un campo di visione strettamente ridotto. Oltre a queste limitazioni è considerevole notare che il robot MARRtino effettua la navigazione autonoma con il solo algoritmo dei potenziali, non richiedendo di fatto l'ausilio esplicito della *Navigation Stack* implementata in ROS, che fornisce metodi e funzioni per la navigazione autonoma.

Il progetto nel suo insieme si prefigge di fornire una soluzione al problema della pianificazione, attuando i metodi descritti e delineando le problematiche introdotte a partire da una soluzione euristica ed efficiente, seppur incompleta. Come si evince dal fatto che nella ricerca di un percorso ottimale, seppur esista, non è garantito il raggiungimento del goal prefissato utilizzando l'algoritmo dei potenziali artificiali.

Il problema viene ovviato solamente attuando un metodo, come descritto in precedenza, che non presenta punti di minimi locali, cioè il metodo dei Vortex Fields. Un'altra soluzione che non incorre in minimi locali è quella della *navigazione numerica* [1]. Questo algoritmo, che prevede di trovare uno *shortest path* utilizzando algoritmi di ricerca con un grafo delle frontiere, è tuttavia un metodo attuabile offline, in quanto è necessaria la conoscenza preliminare dell'ambiente.

Questi algoritmi prevedono dei costi computazionali notevoli, esistono ad ogni

modo, algoritmi differenti per la risoluzione del problema del *motion planning*. Un metodo alquanto efficiente consiste nel modellare tutti gli ostacoli come oggetti sferici, permettendo di eliminare del tutto, anche in questo caso, punti di minimi locali, costringendo come nel metodo dei Vortex Fields di aggirare l'ostacolo ruotandovi attorno. Tuttavia, in questo modo, la generazione di oggetti sferici che racchiudono il \mathcal{C} -*obstacle*, ingrandisce la regione occupata dall'ostacolo, precludendo possibili configurazioni libere appartenenti a \mathcal{C}_{free} .

Un altro approccio comunemente attuato si basa sempre sui potenziali artificiali utilizzando *funzioni armoniche*, usando quindi soluzioni legate a particolari equazioni differenziali che descrivono il processo fisico di trasmissione del calore [1].

Di notevole interesse è la soluzione dell'algoritmo *best-first*, anch'esso basato sui potenziali e simile alla funzione di navigazione numerica, ma attuabile solo in configuration space decisamente ristretti, in quanto il costo computazionale temporale è esponenziale nella dimensione di \mathcal{C} .

Un'ultima soluzione attuabile in molti casi è l'adozione di un metodo che permette al robot di intraprendere percorsi casuali per evadere dal bacino d'attrazione dei minimi locali, è possibile ad esempio imporre delle velocità angolari per far girare il robot in una determinata direzione e permettere quindi di trovare un percorso migliore.

Per concludere, l'attività di progettazione e costruzione del robot autonomo MARRtino si è rivelata interessante e ricreativa, così come la fase di progettazione del software, che è stata utile ad apprendere nuovi costrutti di programmazione orientati a risolvere problemi di questa natura. Il corso nel suo complesso è stato affascinante e stimolante, invogliando lo studente a voler approfondire sempre di più lo studio in quest'ambito.

Gli algoritmi implementati sono risultati concordi alle aspettative, tenendo conto delle problematiche introdotte dagli approcci considerati e delle possibili alternative soluzioni che è possibile adottare nei diversi casi di studio. Con riferimento alle molteplici applicazioni in cui vengono impiegati i robot mobili, il robot MARRtino, seppur costruito e progettato per scopi prettamente didattici, si presenta come un esempio valido e semplice di robot autonomo che può essere impiegato in innumerevoli applicazioni di servizio, grazie alla sua struttura modulare e la possibilità di realizzarlo in diversi modi. Il codice sviluppato è inoltre adattabile a molti altri sistemi robotici, grazie al framework ROS ed è modificabile a seconda dell'utilizzo finale.

Bibliografia

- [1] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics Modelling, Planning and Control*. Springer London, 2009.
- [2] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *The International Journal of Robotics Research*, pages 90–98, 1986.
- [3] C. De Medio, F. Nicolò, and G. Oriolo. Robot motion planning using vortex fields. In *New Trends in Systems Theory*, pages 237–244. Birkhäuser Boston, 1991.
- [4] The robot operating system. <https://www.ros.org>.
- [5] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [6] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.
- [7] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [8] Motion planning online: Artificial potential fields e vortex fields. "<https://github.com/RobSorce/Tesi>".
- [9] Video sperimentali: Marrtino, motion planning online.