UNIVERSITÀ DELLA CALABRIA

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica

Tesi di Laurea in Ingegneria Informatica

Path planning locale per robot mobili basato su switching tra potenziali artificiali

Relatore:
Prof. Luigi D'Alfonso

Candidato:
Flavio Maiorana
Mat. 182611

ANNO ACCADEMICO 2021/2022

Contents

Abstract

Il presente lavoro di tesi ha come obiettivo lo sviluppo, in ambiente Matlab, di un modulo software capace di risolvere un problema di path planning per robot mobili. La soluzione implementata si basa principalmente su quanto descritto in [1]. L'obiettivo é guidare un robot mobile all'interno di un ambiente non noto, cercando di fargli raggiungere un punto prestabilito, senza collisioni con eventuali ostacoli. La tecnica utilizzata é quella dei potenziali artificiali, ma con un meccanismo di switching a stati. Particolare attenzione é stata posta sullo sviluppo di un'architettura software di supporto alla soluzione implementata. È stato adottato il paradigma della programmazione orientata agli oggetti, sfruttando le funzionalità fornite dal linguaggio Matlab, al fine di rendere il software modulare e riutilizzabile per sviluppi futuri.

1 Introduzione

È innegabile che in questi anni si stia assistendo ad un aumento vertiginoso di sviluppo ed uso della robotica. È importante però evidenziare una distinzione tra due concetti apparentemente simili, ma per certi versi opposti, che caratterizzano due macro-categorie della robotica: automazione e autonomia.
L'automazione generalmente riguarda quei robot, tipicamente industriali, che operano in ambienti noti a priori ed eseguono in loop un compito predefinito; automatizzare, perciò, vuol dire sostituire l'essere umano in compiti ripetitivi e solitamente privi di eventi inaspettati.
L'autonomia, invece, ben più complessa da realizzare, é caratteristica di quei sistemi che hanno un certo grado di inconsapevolezza sul proprio futuro e l'ambiente circostante. Il robot, quindi, é definito autonomo se é un agente intelligente situato nello spazio fisico, dove un agente intelligente si definisce come un'entità che **osserva** l'ambiente e prende delle **azioni** per ottimizzare il raggiungimento del suo **obiettivo**[4]. Nel caso specifico di questo elaborato, l'ambiente del robot mobile autonomo é lo spazio bidimensionale (una superficie), e il suo obiettivo é un punto in questo spazio. Ottimizzare il raggiungimento di questo punto vuol dire arrivarci nel minor tempo possibile, senza collidere con eventuali ostacoli. Quindi, il robot autonomo deve compiere una serie di azioni, non note a priori e definite da un algoritmo di pianificazione che si basa sui dati osservati dai sensori, per spostare la sua traiettoria, al fine di evitare collisioni e raggiungere comunque l'obiettivo.
La tipica architettura di navigazione di un robot autonomo é data perciò da quattro moduli (o primitive):

Percezione Prende in input le informazioni derivanti dai sensori, le processa e le restituisce

Localizzazione e Mapping Con le informazioni sensoriali, il robot costruisce una rappresentazione del proprio intorno basandosi sulla propria posizione

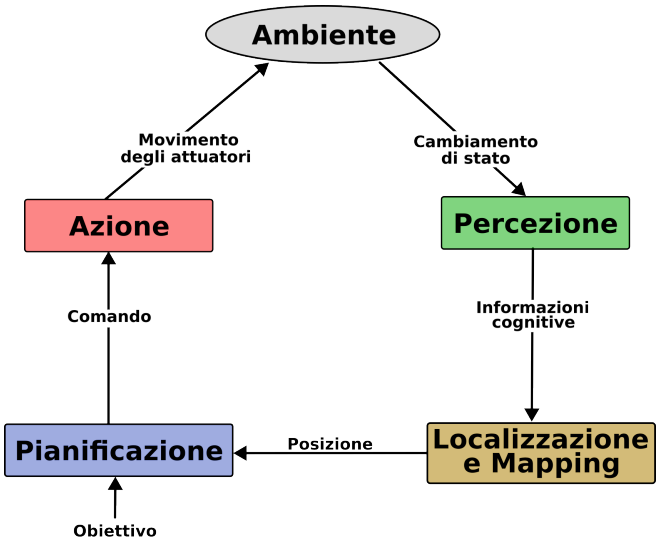
e su ciò che osserva. Il risultato finale, dopo aver fatto varie osservazioni di intorni diversi, sarà una mappa dell'ambiente, rispetto alla quale il robot può localizzarsi.

> Per scopi esemplificativi, questo modulo verrà tralasciato nell'algoritmo oggetto del presente elaborato: si userà descrivere la posizione del robot con coordinate assolute e non rispetto ad una mappa

Pianificazione In base alle informazioni sensoriali e cognitive in possesso, produce in output delle decisioni ad un livello di astrazione ancora alto. Nel caso del motion planning, la direttiva da produrre é il percorso da seguire. Più in generale, il modulo di pianificazione potrebbe generare delle azioni del tipo "gira a sinistra", oppure "fermati".

Azione Prende in input le direttive del modulo di pianificazione e produce dei comandi a basso livello per gli attuatori del robot.

Figure 1.1: Architettura di navigazione



L'architettura utilizzata in questa tesi é di tipo gerarchico: le quattro primitive vengono eseguite in ordine e in loop. È particolarmente indicata per problemi in cui l'obiettivo finale é ben definito a priori. In altre parole, non vi é nessun meccanismo di apprendimento nel robot, ma semplicemente pianificazione deterministica orientata al goal. In figura 1.1 é visivamente sintetizzato quanto appena detto. Nel presente elaborato viene affrontato un problema che rientra nel terzo modulo: il path planning, un problema di grande importanza e oggetto di interesse nel mondo della ricerca.

1.1 Path planning

Una sua rapida formulazione potrebbe essere la seguente: data la posizione iniziale (del robot) A e una posizione finale B, imposta da chi fa uso del robot, il path planning consiste nel calcolare un percorso fisicamente realizzabile e ottimale per arrivare da A a B.
Tra i metodi esistenti (e non), ci sono due importanti distinzioni da fare: sulla formulazione del problema e sulla soluzione al problema.

La prima é tra online e offline path planning, o anche locale e globale. Il path planning globale riguarda quelle situazioni in cui l'ambiente considerato é interamente noto a priori, per cui é possibile calcolare il percorso da seguire ancor prima che il robot inizi a muoversi; quello locale é inerente ai casi in cui il robot debba fare i conti lungo il suo percorso con eventi inaspettati, ad esempio ostacoli in movimento, caso in cui é necessario reagire localmente, aggiornando ripetutamente le informazioni derivanti dai sensori e aggiustando la traiettoria al fine di evitare l'ostacolo e poter raggiungere in tempi ottimali l'obiettivo. Chiaramente, la quasi totalità dei problemi di robotica autonoma deve fare i conti con una situazione del secondo tipo.

La seconda distinzione é tra soluzioni basate su tecniche di intelligenza artificiale - la cui trattazione esula dagli scopi di questa tesi - e soluzioni classiche. Queste ultime possono ulteriormente essere suddivise in [6]:

- Subgoal (o anche roadmap), la cui realizzazione più nota é il metodo che sfrutta i diagrammi di Voronoi.
- Decomposizione in celle
- Sampling based (sfrutta un approccio probabilistico)
- Potenziali artificiali, che verranno ampiamente in trattati nel prossimo capitolo

Di seguito seguirà una breve descrizione di esempi legati ai metodi appena elencati, con l'intento di motivare la scelta dei potenziali artificiali come base per lo sviluppo dell'algoritmo di navigazione, visto che la totalità dei metodi rientranti nelle prime tre categorie risolve adeguatamente il problema del path planning globale, ma risulta inefficiente nel caso del path planning locale.

Voronoi L'idea alla base di questo metodo é rappresentare lo spazio libero delle configurazioni C_{free} - l'insieme di quei punti che per certo non fanno collidere il robot con un ostacolo - come un grafo, ovvero un insieme di nodi (rappresentanti appunto la roadmap) connessi da archi. La posizione dei nodi

é definita tramite il concetto di clearance

$$\gamma(q) = \min_{s \in \partial C_{free}} \|q - s\|$$

dove q é una generica configurazione in C_{free}. La clearence é perciò una funzione che ha come funzione della configurazione q la distanza minima di q da un qualunque punto di un ostacolo. Infatti, ∂C_{free} sarebbe la frontiera dello spazio di configurazione, ovvero il bordo degli ostacoli. C_{free} é formato da quelle configurazioni q tali per cui esiste più di un punto sulla frontiera ∂C_{free} con lo stesso valore $\gamma(q)$; in altre parole, quei punti equidistanti (considerando la distanza minima) da più di un ostacolo. Il risultato é quello mostrato in figura 1.2a. Una volta calcolato il grafo, é sufficiente ritrarre i punti di start e goal sul grafo stesso e calcolare il percorso ottimale tra questi due attraverso un algoritmo di ricerca, ad esempio Djikstra.

Decomposizione esatta in celle In questa tecnica lo spazio delle configurazioni viene suddiviso in celle, come mostrato in figura 1.2b. Ogni cella é delimitata inferiormente e superiormente da un ostacolo e ogni cella ha le seguenti due caratteristiche:

- Tra ogni coppia di configurazioni nella stessa cella esiste sempre un cammino senza collisioni
- Tra ogni coppia di celle adiacenti esiste sempre un cammino senza collisioni (che conduce da una cella all'altra)

In base a questi due principi, viene costruito il cosiddetto grafo di connettività, che ha come archi le connessioni tra celle adiacenti. Come nel metodo precedente, il calcolo del percorso ottimale per raggiungere il goal consiste nell'applicare un algoritmo di ricerca su grafo.

Sampling based Il terzo metodo si basa su un approccio probabilistico. Brevemente, l'idea é di scegliere a ogni iterazione una configurazione "di prova" e fare un test di collisione su quest'ultima, cercando poi di costruire un percorso idoneo in base alla vicinanza alle configurazioni già appartenenti al grafo che si sta costruendo.

(a) Diagramma di Voronoi (b) Decomposizione in celle

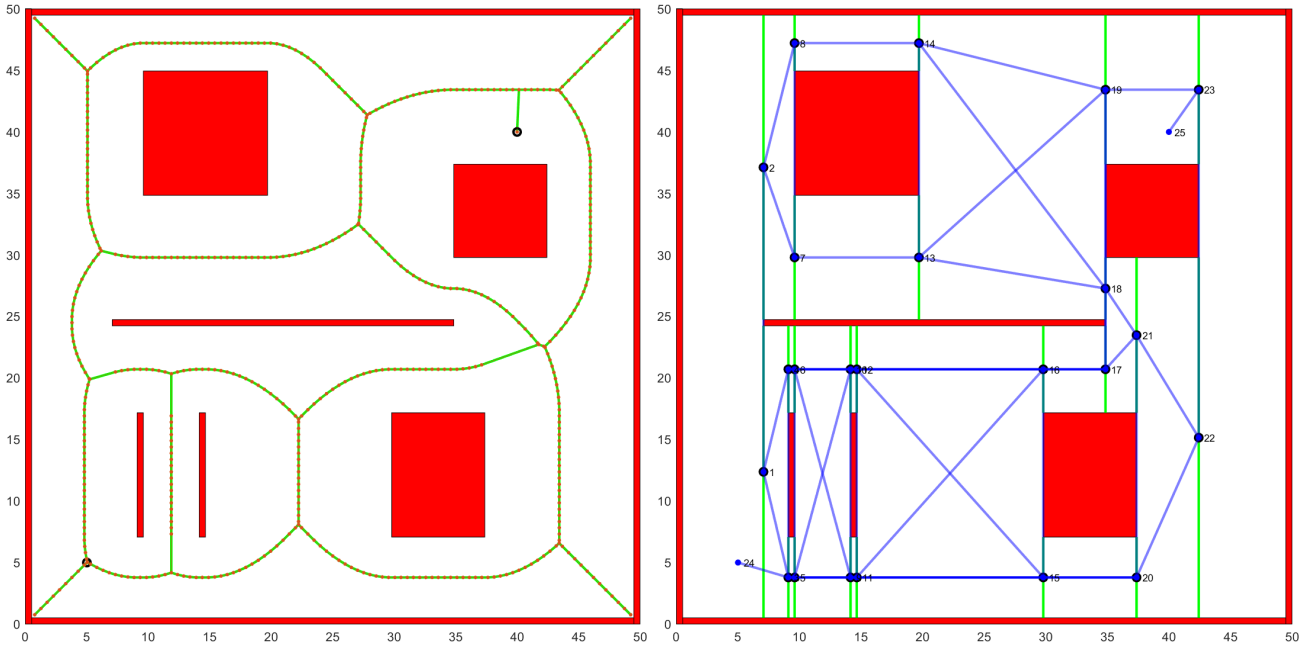


Figure 1.2: Metodi classici basati su grafo

Dunque, come appena visto, il path planning globale é ampiamente risolto grazie ai metodi appena descritti, tuttavia, sono particolarmente efficienti nel caso in cui ci si trovi in un ambiente noto a priori, poco incline al cambiamento e con l'esigenza di fare query ripetute. Il costo computazionale é quasi tutto contenuto nel calcolo del grafo, e la singola query ha un costo legato al numero di archi.
Come da titolo, il presente elaborato ha come obiettivo specifico quello di esporre un lavoro di progettazione, implementazione e simulazione di un algoritmo di path planning **locale** basato su **switching** tra **potenziali artificiali**.

1.2 Formalizzazione del problema

Sia $r(t) = [x_r(t), y_r(t), \theta_r(t)]^T$ la posizione del robot nell'istante t (per semplicità di notazione, ove necessario, si ometterà la dipendenza dal tempo nelle formule) e $O_i(t) = [x_{O,i}(t), y_{O,i}(t)]^T, i = 1...N$ la posizione degli N ostacoli mobili.

Ipotesi semplificative senza perdita di generalità

- Si assume che $t \geq 0$ e che l'istante di inizio sia sempre 0 visto che il sistema é assunto essere tempo invariante nelle sue caratteristiche principali (non vi é usura)
- Si assume che gli ostacoli mobili siano di forma circolare (tutti con raggio R_i), poiché per un ostacolo di forma generica si può considerare la sua circonferenza circoscritta

Esisterà inoltre un punto $G = [G_x, G_y]$ che indica l'obiettivo del robot. Il problema che ci si pone, con riferimento al path planning, consiste nel voler raggiungere il punto G dalla posizione iniziale $r(0)$, tenendo conto degli N ostacoli in movimento. Il robot é dotato di un raggio di visione di r_v metri, entro il quale é capace di rilevare un ostacolo. Inoltre, con l'ipotesi di rilevare gli ostacoli tramite il loro centro, si presuppone che valga la seguente condizione

$$\left\| \begin{bmatrix} x_r(t) \\ y_r(t) \end{bmatrix} - O_j(t) \right\| \leq r_v$$

ovvero $R_i \leq r_v, \forall i$. Ciò vuol dire che nel momento in cui il robot incontra un ostacolo, il centro di quest'ultimo é incluso in R_v e non vi è rischio di collisione ancor prima che l'ostacolo venga rilevato.

1.3 Struttura dell'elaborato

Nella sezione subito dopo si parlerà del metodo dei potenziali artificiali e si motiverà la scelta di utilizzarli per il path planning.
Successivamente verrà affrontato l'algoritmo vero e proprio, per il quale é stato usato un approccio top-down nella progettazione e quindi anche nell'esposizione: verrà prima discusso, in una modalità "bird-view", come é organizzato il software, affrontando come i diversi moduli interagiscono tra di loro per ottenere l'obiettivo finale, senza scendere in dettagli implementativi.
Dopo di che, sempre in una logica top-down, si affrontano (con riferimento alla figura 1.1) i singoli moduli : per primo l'ambiente e come viene rappresentato, dopo di che il modulo di azione, poi pianificazione e infine percezione. Ogni modulo, quindi, viene trattato ipotizzando di avere già accesso al modulo "inferiore" che gli fornisce dati in input.
I moduli, infatti, sono stati progettati e sviluppati indipendentemente gli uni dagli altri. Come si vedrà nella sezione 4, non vi é correlazione tra un modulo e l'altro, ma interagiscono tra di loro guidati da un oggetto - nel caso del presente elaborato sarà una classe che modella il Robot - che li incapsula tutti e che coordina le interazioni, senza curarsi della specifica implementazione sottostante. In questo modo é possibile sviluppare delle gerarchie indipendenti dei singoli moduli, con una enorme possibilità di riutilizzo e di adoperare diverse implementazioni, dalle quali la struttura del funzionamento "ad alto livello" - per essere più specifici, il livello di astrazione a cui si trova la classe rappresentante il robot - non viene influenzata.

2 Potenziali artificiali

Metodo introdotto per la prima volta negli anni 90 da Oussama Khatib, é tanto semplice quanto efficace per risolvere il problema del path planning. Si differenzia dai metodi precedentemente menzionati per il fatto che la traiettoria non viene costruita "attivamente", nel senso che non vengono definiti dei punti di passaggio che formano una traiettoria ottimale. Piuttosto quello che si fa é, mediante l'interazione con i cosiddetti potenziali artificiali, cercare una **configurazione ottimale**. Quindi, la traiettoria viene costruita mentre il robot si muove, motivo per cui é un metodo adatto al path planning locale.

2.1 Metodo classico

Tradizionalmente, nel path planning basato su potenziali artificiali il robot viene fatto muovere mediante una funzione in due variabili, ovvero un potenziale scalare, che nasce dalla somma di due potenziali: attrattivo e repulsivo. Questi due potenziali sono chiamati artificiali perché generano una forza che guida il robot in ogni sua configurazione r, nonostante nella realtà non vi sia nessuna sorgente a generare quella forza. Nello specifico, la forza generata dal potenziale é il suo antigradiente, ovvero il gradiente cambiato di segno, che indica al robot la direzione di moto localmente più promettente[3], cioè verso il punto di minimo della funzione. Di conseguenza il potenziale attrattivo assume una forma tale da avere un unico punto di minimo posizionato proprio nel punto di arrivo, mentre il repulsivo ha un unico punto di massimo corrispondente alla posizione dell'ostacolo. Nello specifico, ciò é di solito realizzato grazie alla funzione della configurazione del robot

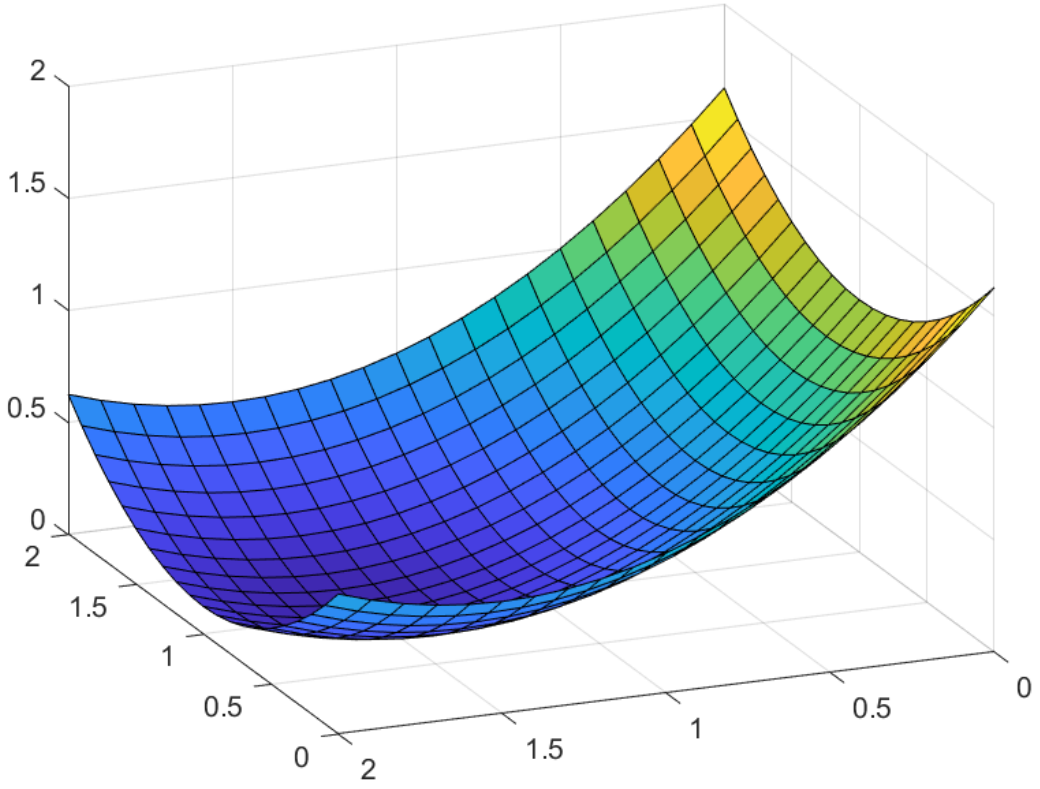
$$e(r) = G - \begin{bmatrix} x_r(t) \\ y_r(t) \end{bmatrix}$$

che misura l'errore tra la posizione del robot e il goal. L'idea dietro al potenziale attrattivo é di generare una funzione che sia direttamente proporzionale a $e(r)$, e quindi assuma valori elevati lontano dal goal e valori bassi vicino al goal, creando così un punto di minimo nel goal stesso.

Il potenziale attrattivo ha di solito la forma mostrata in figura 2.1, in cui il punto di minimo, ovvero il goal, ha coordinate $[1, 1.5]$. La funzione corrispondente é

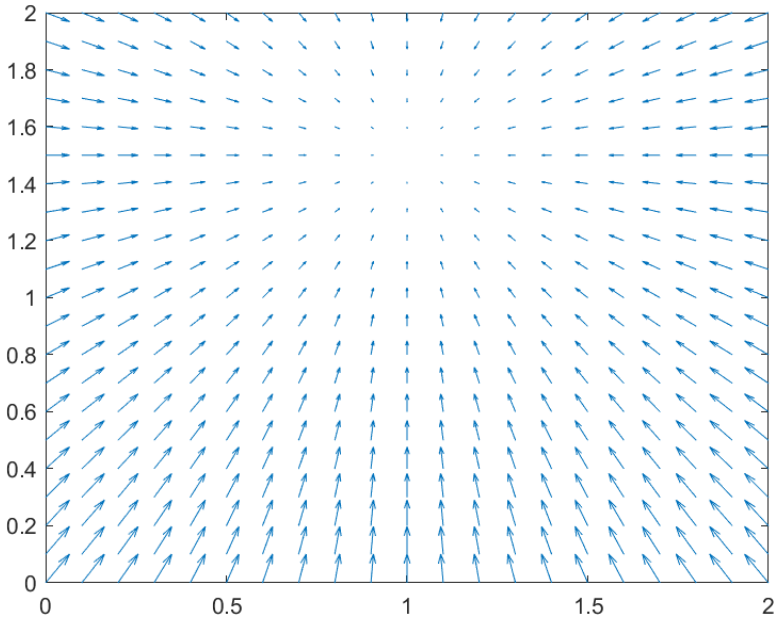
$$U_{a1}(r, G) = \frac{1}{2} \cdot k_1 ||e(r)||^2$$

Figure 2.1: Potenziale attrattivo di forma paraboloide



Il suo antigradiente di conseguenza é formato da tanti vettori che, con un'intensità proporzionale alla distanza dal goal, puntano verso quest'ultimo.

Figure 2.2: Antigradiente del potenziale attrattivo



Matematicamente si esprime come il vettore delle derivate parziali (del potenziale) cambiato di segno, ovvero

$$-\nabla U_{a1}(r, G) = k_1 e(r) \tag{2.1}$$

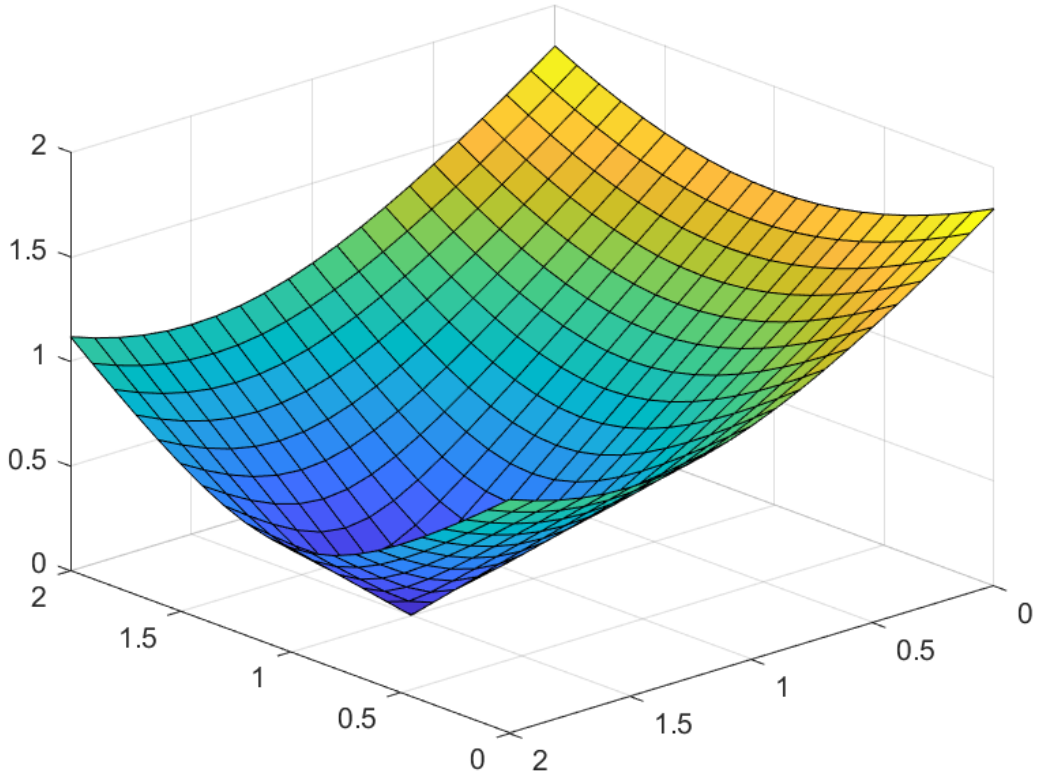
Perciò, la forza esercitata dal potenziale sul robot punta verso il goal e converge a zero quando la configurazione $r(t)$ tende alla destinazione G, esprimendo difatti l'errore tra goal e posizione del robot.

Oltre al potenziale paraboloide si può optare anche per un potenziale conico

$$U_{a2}(r, G) = k_2||e(r)|| \tag{2.2}$$

e, come si può anche vedere in figura 2.3, la riduzione dell'errore all'avvicinarsi del robot al goal non é più quadratica, ma lineare. Da ciò scaturisce un comportamento più "smooth" lontano dal goal (la velocità é meno elevata rispetto al potenziale paraboloide). Tuttavia, vicino al goal é conveniente usare quest'ultimo, per raggiungere la posizione esatta con minore velocità, il che conferisce più precisione al movimento del robot.

Figure 2.3: Potenziale attrattivo di forma conica



L'antigradiente del potenziale conico ha la seguente espressione

$$-\nabla U_a(r, G) = k_2 \frac{e(r)}{||e(r)||} \tag{2.3}$$

L'idea é quella di stabilire una soglia per la funzione errore oltre la quale, avvicinandosi al goal, il potenziale attrattivo sarà di forma paraboloide.

Dunque il potenziale attrattivo finale sarà del tipo

$$U_a(r) = \begin{cases} \frac{1}{2} \cdot k_1 ||e(r)||^2 & ||e(r)|| \leq \rho \\ k_2 ||e(r)|| & ||e(r)|| > \rho \end{cases} \tag{2.4}$$

dove ρ é la soglia. Gli scalari k e la soglia andranno scelti in maniera tale da garantire continuità nel passaggio da un potenziale attrattivo all'altro. In particolare deve valere che la velocità imposta dall'antigradiente al robot proprio nel punto di soglia deve essere la stessa per entrambi i potenziali, ovvero

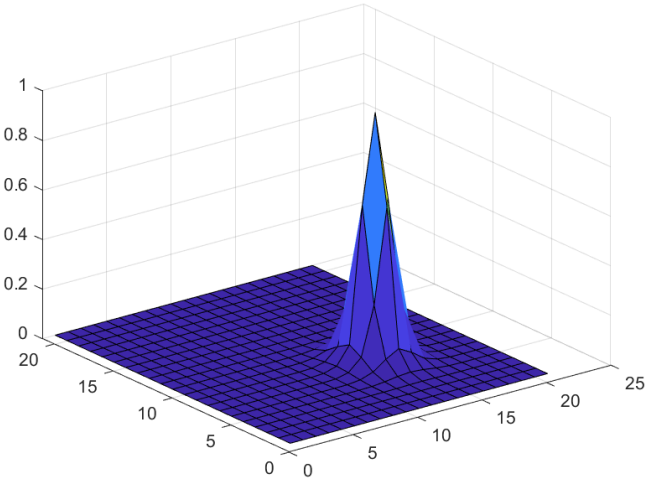
$$k_1 \cdot e(r) = k_2 \cdot \frac{e(r)}{||e(r)||} \Leftarrow ||e(r)|| = \rho$$

da cui segue che

$$k_1 \cdot \rho = k_2 \tag{2.5}$$

Il potenziale repulsivo ha una forma duale a quello attrattivo, come mostrato in figura 2.4a. Esso impone che entro uno distanza η dall'ostacolo per il quale lo si sta calcolando, la sua funzione avrà valore inversamente proporzionale alla distanza dall'ostacolo stesso. Praticamente, più ci si avvicina all'ostacolo entro una certa soglia η, più l'intensità del potenziale artificiale, perciò anche della forza generata da esso, aumenta.

(a) Potenziale repulsivo



(b) Antigradiente del potenziale repulsivo

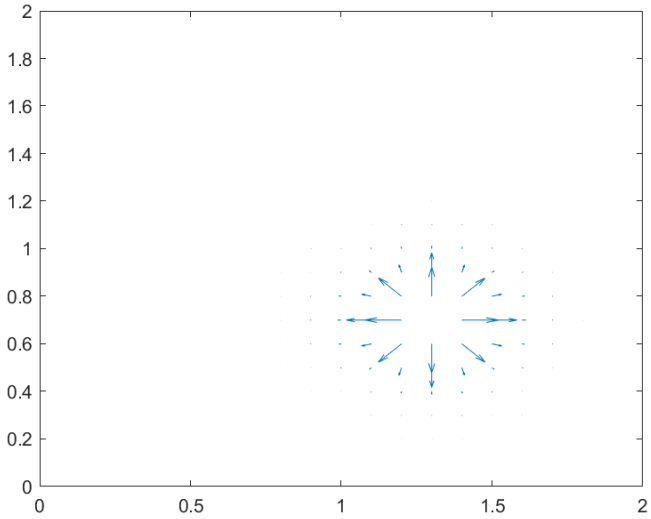


Figure 2.4: Potenziale repulsivo

Al potenziale repulsivo corrisponde la seguente funzione a tratti

$$U_r(r(t), O_j(t)) = \begin{cases} \frac{1}{2} \left(\frac{1}{d(r(t), O_j(t)} - \frac{1}{\eta} \right) & d(r(t), O_j(t)) \leq \eta \\ 0 & altrimenti \end{cases} \tag{2.6}$$

dove

$$d(r(t), O_j(t) = \left\| O_j(t) - \begin{bmatrix} x_r(t) \\ y_r(t) \end{bmatrix} \right\|$$

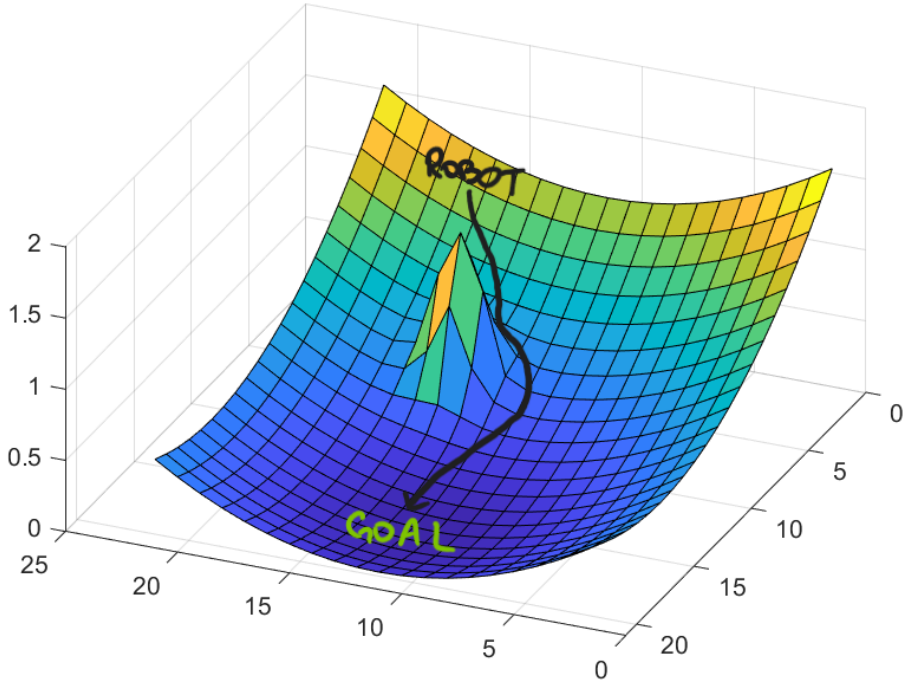
Dunque, il compito della forza generata é quello di spingere via il robot dalla posizione dell'ostacolo tanto più che il robot si avvicina entro la soglia η a quest'ultimo. In figura 2.4b si vedono le linee di campo dell'antigradiente che puntano radialmente verso l'esterno rispetto alla posizione dell'ostacolo, qui con coordinate $[1.5, 1]$.

Potenziale totale A questo punto arrivati, una volta calcolato il potenziale repulsivo per ogni singolo ostacolo e il potenziale attrattivo legato al goal, si ottiene il potenziale totale:

$$U(r(t)) = U_a(r(t), G) + \sum_{j=1}^{n} U_r(r(t), O_j(t)) \tag{2.7}$$

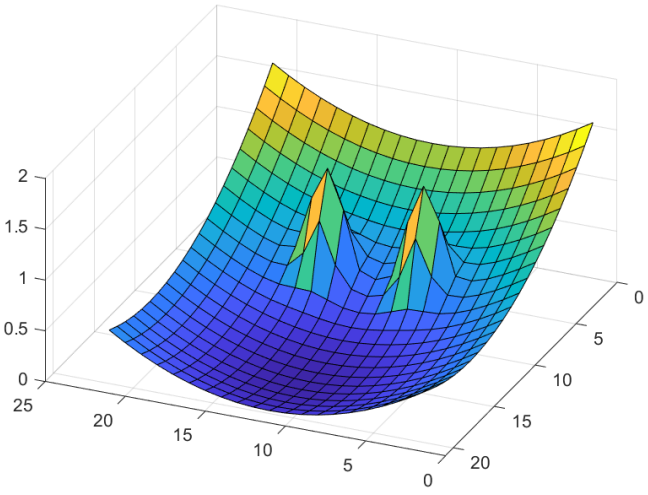
In figura 2.5 si vede un possibile percorso del robot per arrivare dal punto di partenza al goal.
Dalla (2.7) si vede che l'antigradiente è una funzione di $r(t)$: vuol dire che in ogni sua configurazione, siccome l'antigradiente si suppone già calcolato prima ancora che il robot si metta in movimento, esso può ottenere informazioni sull'antigradiente riferendosi soltanto alla sua stessa posizione. Quindi, il vettore velocità del robot $[v_x(t), v_y(t)]$ ha come riferimento in ogni istante t il valore dell'antigradiente (che é un vettore) in $[r_x(t), r_y(t)]$.
Lontano dagli ostacoli, il vettore velocità ha una direzione che punta al goal e un'intensità che diminuisce man mano che ci si avvicina al goal: più il robot si avvicina ad un ostacolo, più la direzione del vettore velocità vira verso il verso opposto rispetto a $\theta = \tan\left(\frac{y_{O,j}(t) - y_r(t)}{x_{O,j}(t) - x_r(t)}\right)$ (la direzione del vettore che collega il robot all'ostacolo), spostando così temporaneamente la traiettoria desiderata e aumentando "l'intensità della virata" lontano dall'ostacolo man mano che si avvicina ad esso. Il robot dunque, seguendo l'antigradiente, viene in ogni sua configurazione $r(t)$ - usando l'analogia con il campo gravitazionale - attratto dal goal e **contemporaneamente** respinto dagli ostacoli.
Chiaramente, il metodo é idoneo sia al path planning globale che a quello locale (basta ricalcolare il potenziale totale in presenza di ostacoli). Volendo, d'altra parte, dare un'etichetta dal punto di vista dell'architettura di navigazione, i potenziali artificiali tradizionali obbediscono ad una di tipo reattivo: considerando la figura 1.1, il modulo di mapping e path planning sono praticamente assenti. Gli attuatori del robot, che sono direttamente collegati ai sensori [2] e ricevono direttamente il comando derivante dal calcolo precedente del potenziale artificiale, senza alcun tipo di pianificazione algoritmica.

Figure 2.5: Potenziale totale



Tuttavia, nonostante la sua semplicità ed efficacia, vi sono delle non-idealità legate a questo approccio.
Ad esempio, la traiettoria potrebbe non essere continua. Il potenziale repulsivo non é "coordinato" con quello attrattivo, perciò il robot potrebbe ricevere comandi che causerebbero un cambio di direzione o di velocità troppo repentino che andrebbe gestito dalla legge di controllo o un modulo di pianificazione apposito. Questa osservazione verrà approfondita meglio successivamente.
Altro problema da considerare é il fatto che il robot viene "passivamente" spinto via dagli ostacoli senza la certezza che venga portato in una posa in cui può effettivamente evitare con successo l'ostacolo.
Ma il più importante, sicuramente, é il problema dei minimi locali: il robot non ha alcuna informazione su come uscirne, né può prevederli in anticipo. I minimi locali sono dei punti ad antigradiente nullo (il gradiente del potenziale attrattivo e repulsivo possono annullarsi a vicenda), dove il robot non ha nessuna forza a guidarlo verso il goal, situazione che dovrebbe verificarsi soltanto nel punto di goal stesso. In figura 2.6 viene mostrato il percorso simulato di un robot in presenza di un minimo locale nel potenziale artificiale. Il robot parte dalla posizione $r(0) = [5, 0]$ mentre il goal si trova in posizione $G = [5, 10]$. Gli ostacoli sono posizionati tra il robot e il goal ad una distanza tale da causare un annullamento del gradiente (quindi il minimo locale) proprio a metà tra i due, lì dove il robot cerca di passare. La traiettoria del robot, infatti, si ferma nel minimo locale: non viene spinto da nessuna forza e "crede" di essere arrivato nel punto di goal.

(a) Potenziale con minimo locale



(b) Antigradiente del potenziale con minimo locale

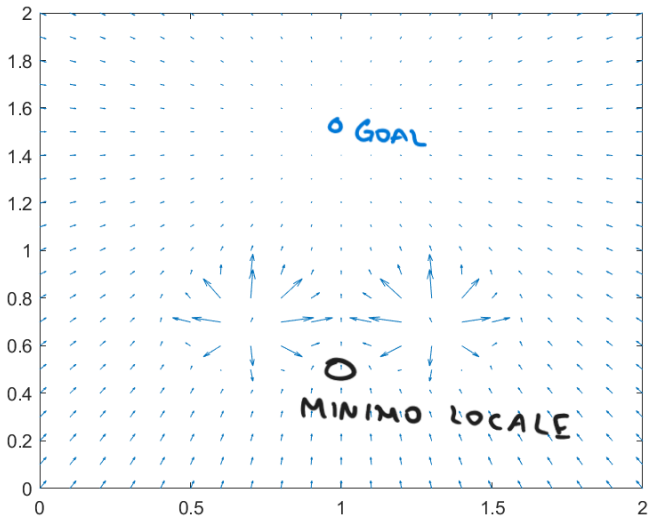


Figure 2.6: Minimo locale

2.2 Potenziale bypassante

Al fine di evitare le non-idealità dovute all'utilizzo dei potenziali sommati, l'idea implementata nel presente elaborato é quella di sfruttare, per evitare gli ostacoli, al posto di quello repulsivo, un altro tipo di potenziale che verrà chiamato bypassante. Fondamentalmente, si tratta di un potenziale che invece di spingere via il robot dall'ostacolo, lo porta a circumnavigarlo. Infatti, le linee di campo dell'antigradiente di questo potenziale sono concentriche, al contrario di quelle del potenziale repulsivo che sono radiali. L'idea quindi é di basarsi su una superficie che ruoti attorno a un centro identificato da una certa coppia di coordinate, ad esempio un elicoide:

$$\begin{cases} x = x_0 + rcos(\xi) \\ y = y_0 + rsin(\xi) \\ z = c\xi \end{cases}$$

Applicando nella terza equazione la tangente da entrambe le parti, otteniamo $\tan(\xi) = \tan(\frac{z}{c})$ che, confrontando con le prime due equazioni, diventa

$$\tan\left(\frac{z}{c}\right) = \frac{y - y_0}{x - x_0}$$

dove $[x_0, y_0]$ sarebbe la posizione dell'ostacolo. Invertendo questa funzione per esprimerla come z in funzione di x e y, otteniamo il potenziale elicoidale, ridenominato bypassante, in senso orario rispetto all'ostacolo e centrato in esso:

$$\Gamma(x, y, x_0, y_0) = c\tan^{-1}\left(\frac{y - y_0}{x - x_0}\right) \tag{2.8}$$

(a) Elicoide centrato in $[1, 1]$ con $c = 2$, $0 \leq r \leq 1$ e $0 \leq \xi \leq 4\pi$

(b) Potenziale bypassante centrato in $[1, 1]$ con $c = 2$ e in senso orario

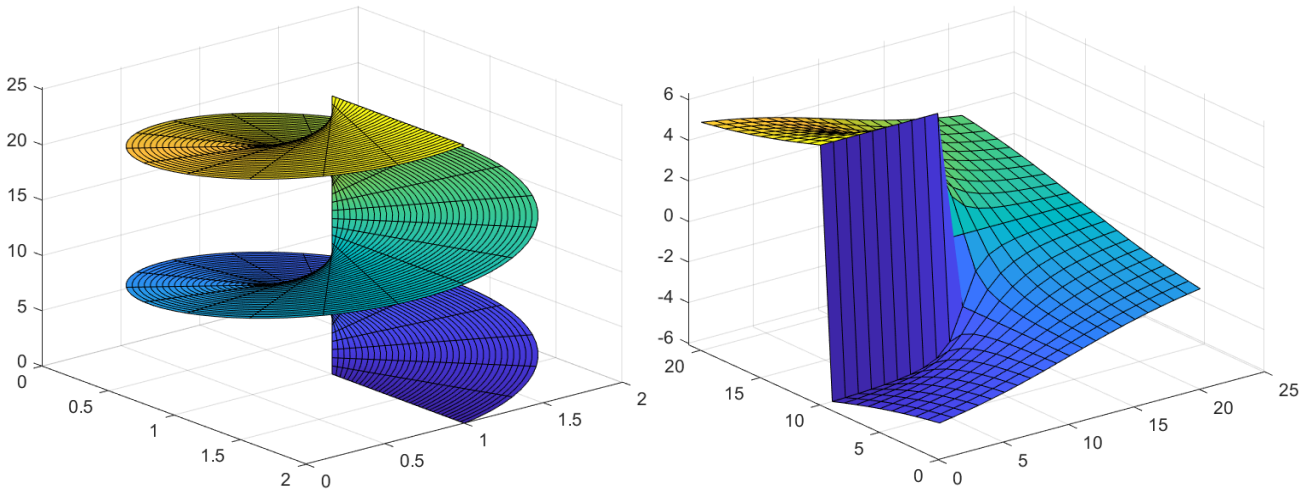


Figure 2.7: Potenziale bypassante

Da notare che il potenziale in figura 2.7b ha una discontinuità per $x = x_0$, ma la funzione in (2.9) é continua per $(x, y) \neq (x_0, y_0)$. Siccome il robot non può mai andare esattamente nella stessa posizione dell'ostacolo, non c'è perdita di generalità nell'ignorare la discontinuità e di conseguenza il robot sarà sempre in grado di seguire la forza generata dal potenziale. Il potenziale con equazione 2.8 può essere visto come una funzione della posizione del robot e dell'ostacolo da aggirare

$$\Gamma(x, y, x_0, y_0) = \Gamma(x_r(t), y_r(t), x_{O,j}(t), y_{O,j}(t))$$

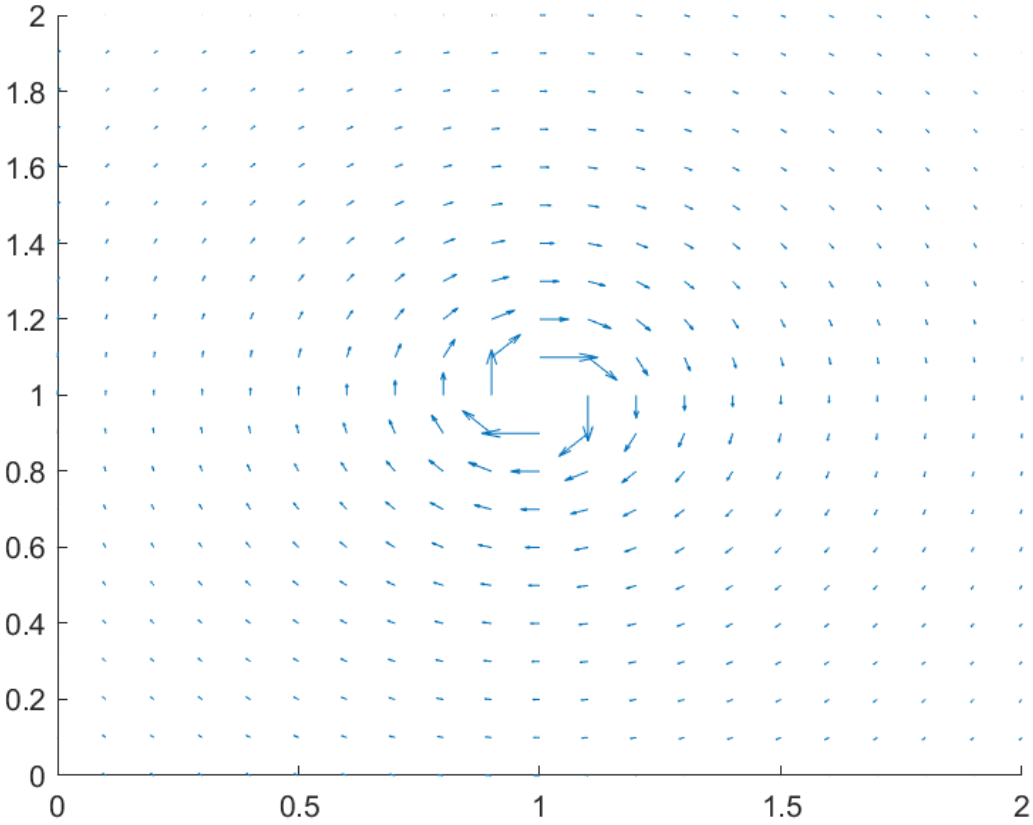
Dunque, la forza generata dal potenziale bypassante é esprimibile come il suo antigradiente

$$-\nabla U_b(r(t), O_j(t)) = \begin{bmatrix} \frac{c(y_r(t)-y_{O,j}(t))}{(x_r(t)-x_{O,j}(t))^2+(y_r(t)-y_{O,j}(t))^2} \\ \\ \frac{c(x_{O,j}(t)-x_r(t))}{(x_r(t)-x_{O,j}(t))^2+(y_r(t)-y_{O,j}(t))^2} \end{bmatrix} \tag{2.9}$$

Quindi, l'antigradiente é formato da tanti vettori che

- Indicano una velocità desiderata nella posa $r(t)$ del robot, al fine di aggirare l'ostacolo preso in considerazione
- Aumentano di intensità man mano che il robot si avvicina all'ostacolo
- Hanno forma concentrica

Figure 2.8: Antigradiente del potenziale bypassante



Questo tipo di potenziale, chiaramente, é intrinsecamente adatto ad ostacoli di forma circolare, motivo per cui nel lavoro di tesi si é considerato soltanto questo tipo di ostacoli.

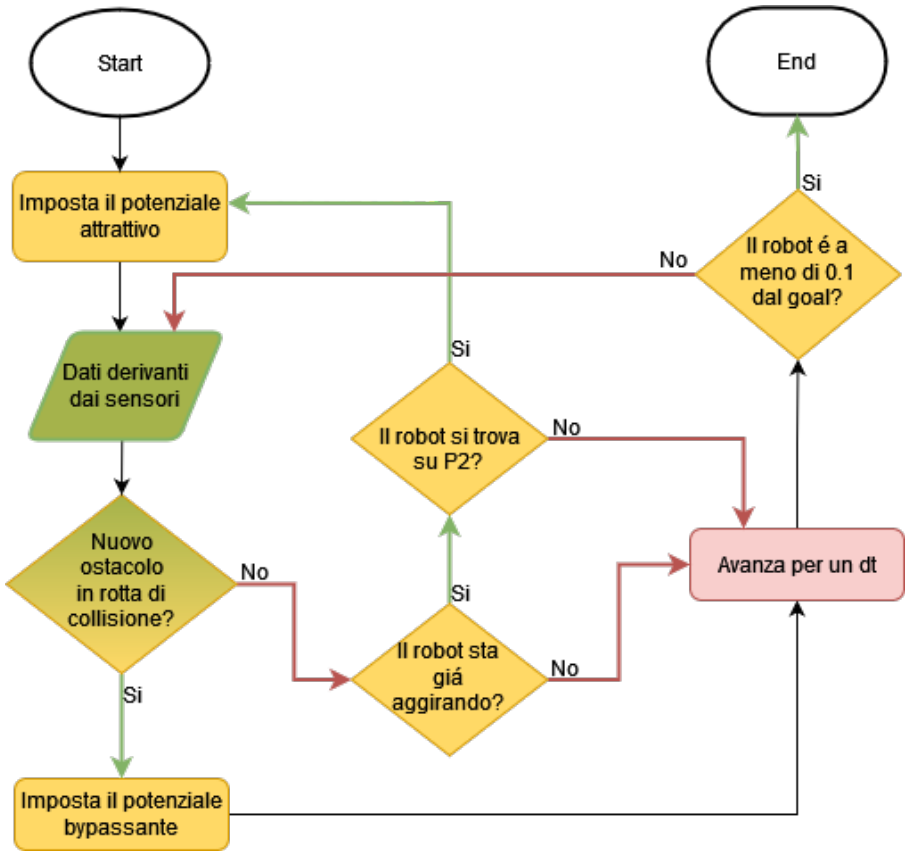
3 Panoramica sull'algoritmo di navigazione

La strategia é semplice: nella posizione iniziale, il robot sonda l'ambiente attorno a sé. Le possibilità ora sono due:

- Non vi sono presenti ostacoli a impedirne l'avanzamento verso il goal, per cui la traiettoria da seguire é quella imposta dal potenziale attrattivo
- Un ostacolo impedisce l'avanzamento del robot ed é necessario **"switchare"** dal potenziale attrattivo a quello bypassante, al fine di aggirare l'ostacolo, per poi tornare a seguire la traiettoria dell'antigradiente attrattivo

Perciò, la peculiarità di questo algoritmo é che in ogni istante di tempo il robot seguirà **un solo potenziale** alla volta, evitando così il problema dei minimi locali discusso in precedenza. Inoltre, le informazioni necessarie a pianificare lo switch non richiedono informazioni globali, ma solo di tipo **locale** relative all'ostacolo da aggirare. Nella figura 3.1 si vede, orientativamente, come funziona l'algoritmo.

Figure 3.1: Flowchart Diagram

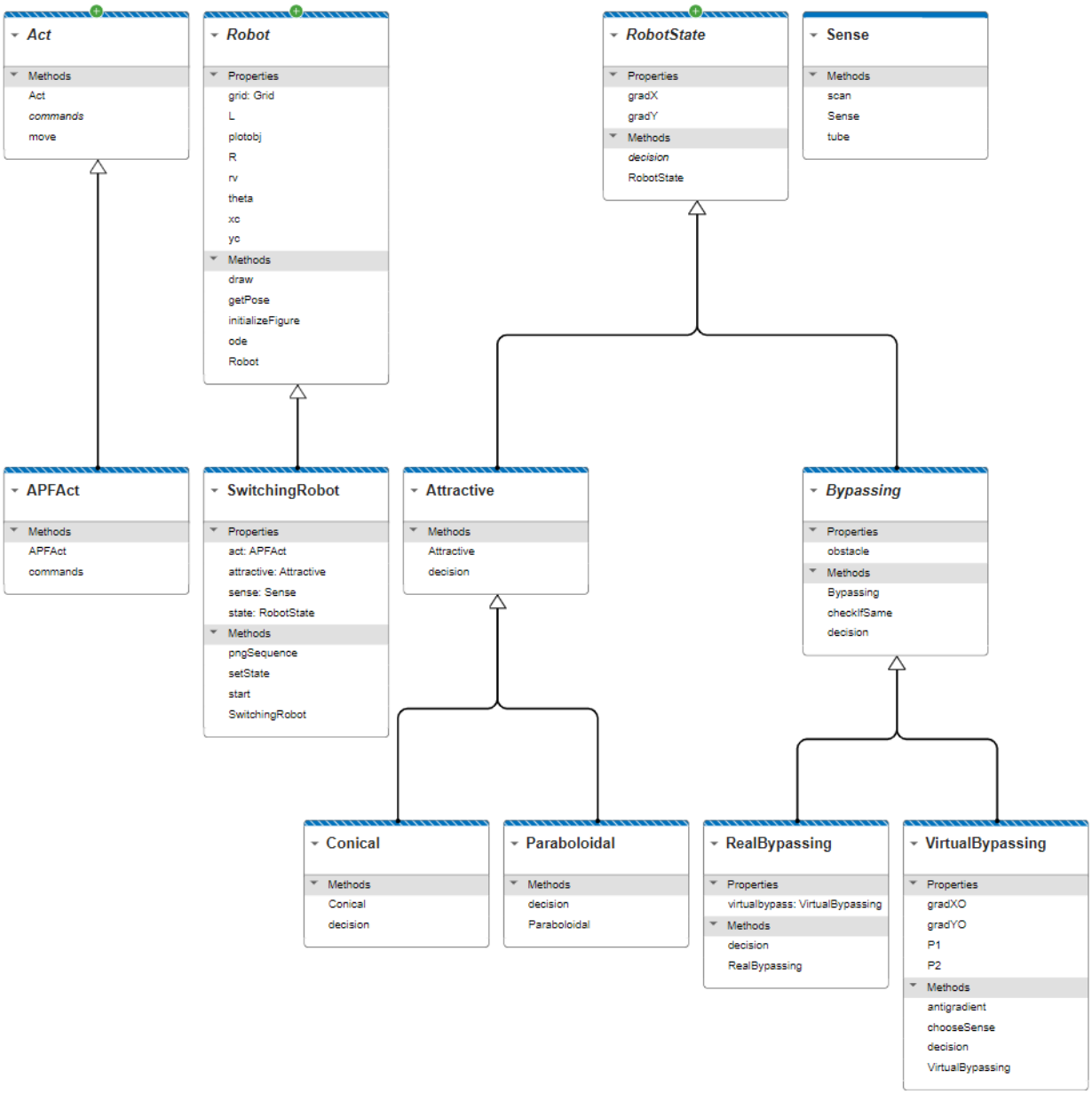


Le pseudo-istruzioni in giallo sono quelle relative al modulo di pianificazione e le verdi al modulo di visione. Naturalmente verranno approfonditi nelle rispettive sezioni (si spiegherà anche il significato del punto P2). Si vede qui ancora una volta la sequenza di azioni: prima interviene il modulo di percezione, che

restituisce al modulo di pianificazione le informazioni visive e cognitive raccolte. Quest'ultimo a sua volta decide se impostare il potenziale attrattivo o bypassante in base ai due controlli al centro. Infine, il robot viene fatto muovere per un dt, il che suggerisce che l'approccio alla simulazione é discretizzato. Ciò vuol dire il robot eseguirà in loop tutte le azioni del flowchart diagram ogni dt.

3.1 Architettura software

Figure 3.2: Class diagram



In figura 3.2 é riportato il class diagram del software di navigazione. Le classi Act e Sense rappresentano rispettivamente il modulo di azione e pianificazione. Il modulo di pianificazione é costituito dalla gerarchia "figlia" della classe as-

tratta RobotState. SwitchingRobot é la classe che incapsula i tre moduli e ne fa uso all'interno di un suo "main" che é il metodo start. Si può quindi notare che i tre moduli sono completamente disaccoppiati tra di loro e interagiscono soltanto tramite la classe SwitchingRobot.
Quindi, per capire meglio come le classi sono collegate tra di loro, nel seguente codice, che si trova appunto nella classe SwitchingRobot, é illustrato il funzionamento dell'algoritmo dal punto di vista del robot, che, come é facile vedere, é completamente disaccoppiato dalle implementazioni concrete dei singoli moduli di navigazione.

Listing 3.1: Entry point del modulo software

```
function obj = start(obj)
   %Simulation data
   e = norm([obj.xc,obj.yc]-obj.grid.goal);
   tspan = 0.05; tsim = 0; samples = 0;
   %Starting simulation
   while(e > 0.1 && tsim < 20)
      %Sensed obstacle (empty array if nothing was sensed)
      dObstacle = obj.sense.scan(obj);
      %New directive
      obj.state.decision(obj,dObstacle);
      %Giving the command to the actuators
      [obj.xc,obj.yc,obj.theta] = obj.act.move(obj,tspan);
      %Refreshing the error
      e = norm([obj.xc,obj.yc]-obj.grid.goal);
      tsim = tsim + tspan; pause(0);
   end
end
```

L'idea é che il robot proceda in maniera discreta, ovvero che faccia dei campionamenti in istanti equidistanti e si muova di conseguenza. Fa da "guida" il ciclo while, il cui "indice di scorrimento" é l'errore e tra robot e goal. Questo ciclo si ripeterà naturalmente fino a che il robot non sarà arrivato nel punto di goal. Il tempo del ciclo é scandito dalla variabile *tspan*, che indica il tempo tra un istante di campionamento e il successivo, dove con campionamento si intende, volgarmente, uno "snapshot" dell'intorno da parte del modulo di visione.
Come si vede nel codice, il meccanismo attuato ad ogni iterazione é molto semplice: dapprima si controlla se é stato rilevato un ostacolo attraverso il metodo scan della classe Sense. Successivamente la variabile dObstacle, che é vuota se non viene rilevato nessun ostacolo, viene passata al metodo decision.
Quest'ultimo é richiamato su un oggetto di tipo RobotState. A seconda del sottotipo assegnato a RobotState, le azioni eseguite dal metodo decision saranno diverse (é proprio nel passaggio da uno un sottotipo di RobotState all'altro che avviene lo switch di potenziali).
Infine, una volta che il robot ha preso una decisione, passa il comando al modulo di azione per potersi effettivamente muovere attraverso il metodo move della

classe Act (si parlerà meglio di questo metodo nella sezione sulla simulazione). Si approfondirà nelle successive sezioni ognuna di queste tre "macro-istruzioni" nel dettaglio, approfondendo cosa succede al loro interno.

4 Rappresentazione dell'ambiente

Prima di affrontare i moduli che guidano il robot nell'ambiente circostante, ha senso chiedersi, a questo punto, come questo possa esser rappresentato, soprattutto in relazione ai potenziali artificiali. La scelta fatta nel presente elaborato é stata quella di rappresentare l'ambiente in maniera completamente discretizzata. Ciò che lo circonda é per il robot una matrice (nel linguaggio di Matlab una meshgrid) in cui ogni cella corrisponde a un $dx \cdot dy$ del mondo reale. Di conseguenza, il potenziale artificiale sarà una funzione che si concretizza in una matrice della stessa dimensione della meshgrid (il calcolo dell'antigradiente sarà affrontato nel modulo di pianificazione). Ogniqualvolta il robot voglia calcolare il valore dell'antigradiente in un punto di coordinate $[x, y]$, é necessario convertire queste in indici della meshgrid. Ciò viene fatto dal metodo coord2index della classe Grid.

Listing 4.1: Metodo che converte coordinate cartesiane in indici della meshgrid

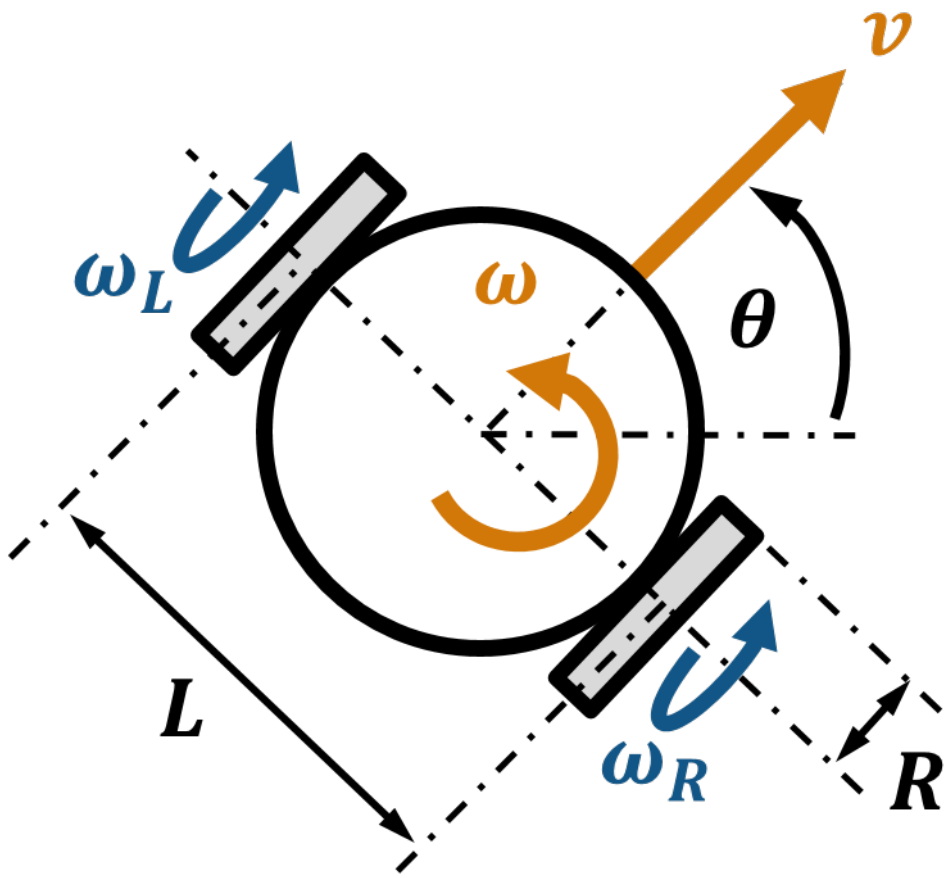
```
function index = coord2index(obj,point)
    index = zeros(1,2);
    % x coordinate goes into the column index
    floorx = floor(point(1));
    index(2) = (floorx + obj.dx*floor((point(1) - floorx)/
        obj.dx))*obj.resolution + 1;
    % y coordinate goes into the row index
    floory = floor(point(2));
    index(1) = (floory + obj.dx*floor((point(2) - floory)/
        obj.dx))*obj.resolution + 1;
    index = floor(index);
end
```

Nota
Si é scelto, in merito alla rappresentazione dell'ambiente, di non sfruttare una griglia di occupazione binaria per gli ostacoli. Questo rende il modulo di visione un po' "fittizio", nel senso che esso fa lo "snapshot" semplicemente andando a vedere la posizione degli ostacoli nella classe Grid, cosa che nella realtà non avviene. Questa semplificazione non provoca perdita di generalità, poichè basta, appunto, aggiungere una griglia di occupazione binaria accessibile dal modulo di visione tramite un sensore lidar. In ogni caso, ciò che é importante dell'implementazione discretizzata é il metodo per passare da coordinate reali a coordinate della meshgrid.

5 Modulo di azione

Il modulo responsabile per il movimento del robot riceve in input un comando dal modulo di pianificazione, ovvero l'antigradiente da seguire, e restituisce in output una "traduzione" del comando in una velocità da imporre agli attuatori (tramite una legge di controllo), tenendo conto dello specifico modello cinematico del robot.

Figure 5.1: Modello Differential-Drive



5.1 Modello cinematico

Si é ipotizzato in questa tesi di usare un modello cinematico anolonomo di tipo differential drive. Il fatto che sia anolonomo vuol dire che obbedisce ad un vincolo sulla velocitá e non sulla posizione, ovvero

$$\theta(t) = \arctan\left(\frac{\dot{y}(t)}{\dot{x}(t)}\right)$$

Praticamente, al robot é impossibile muoversi in ogni direzione con la stessa velocità; per mantenerla, deve continuare a mantenere il suo orientamento invariato. Le equazioni differenziali che modellano un generale robot che obbedisce

a vincolo anolonomo é

$$\begin{cases} \dot{x}(t) = v(t)\cos(\theta(t)) \\ \dot{y}(t) = v(t)\sin(\theta(t)) \\ \dot{\theta}(t) = \omega(t) \end{cases} \tag{5.1}$$

Quindi, secondo questo modello, un robot può essere completamente caratterizzato mediante la sua posizione $[x, y, \theta]$ e la sua velocità $[v, \omega]$, dove v indica la velocità lineare (di traslazione) e ω quella angolare (di rotazione).
In particolare, nel modello differential drive, il robot, come mostrato in figura 5.1, viene mosso da due ruote laterali (ogni ruota ha il suo motore) di raggio R, posizionate sullo stesso asse e distanti tra di loro L.
Detto ciò, si può esprimere la velocità lineare come la media tra le velocità lineari delle due ruote

$$v(t) = \frac{R \cdot \omega_R(t) + R \cdot \omega_L(t)}{2}$$

e quella angolare come la differenza tra le due velocità lineari normalizzata per la distanza tra le ruote

$$\omega(t) = \frac{R \cdot \omega_R(t) - R \cdot \omega_L(t)}{L}$$

Riunite in forma matriciale, queste due relazioni danno la seguente

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ \frac{R}{L} & -\frac{R}{L} \end{bmatrix} \begin{bmatrix} \omega_R(t) \\ \omega_L(t) \end{bmatrix} \tag{5.2}$$

Invertendo questa relazione, si può ottenere la velocitá delle due ruote a partire dalla velocità lineare e angolare di riferimento.

$$\begin{bmatrix} \omega_R(t) \\ \omega_L(t) \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ \frac{R}{L} & -\frac{R}{L} \end{bmatrix}^{-1} \begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} \tag{5.3}$$

Questa inversione é sempre possibile, visto che

$$det \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ \frac{R}{L} & -\frac{R}{L} \end{bmatrix} = \frac{-R^2}{L}$$

che é sempre non nullo, per cui la matrice é invertibile.
Dunque, il modello in equazione 5.1 diventerá

$$\begin{cases} \dot{x}(t) = \frac{R}{2}\left(\omega_R(t) + \omega_L(t)\right)\cos(\theta(t)) \\ \dot{y}(t) = \frac{R}{2}\left(\omega_R(t) + \omega_L(t)\right)\sin(\theta(t)) \\ \dot{\theta}(t) = \frac{R}{L}\left(\omega_R(t) - \omega_L(t)\right) \end{cases}$$

La relazione appena ottenuta é implementata nella classe astratta Robot in quanto rappresentante del modello cinematico

Listing 5.1: Metodo che esprime lo specifico modello cinematico

```
function [Xdot,wRwL] = ode(obj,vr,wr)
    K = [obj.R/2 obj.R/2 ; obj.R/obj.L -obj.R/obj.L];
    wRwL = K \ [vr ; wr];
    Xdot =
      ([cos(obj.theta) 0 ; sin(obj.theta) 0 ; 0 1] * K * wRwL);
end
```

5.2 Legge di controllo

Nel caso specifico del path planning con potenziali artificiali, la velocità di riferimento é data dall'antigradiente relativo alla posa del robot, cioè $v_\nabla(t) = -\nabla U(r(t))$. Con la legge di controllo viene stabilita una velocità lineare e angolare da imporre al robot e viene stabilito che

$$v(t) = M_v \cos(\theta_\nabla(t) - \theta_r(t)) \tag{5.4}$$

$$\omega(t) = K_\omega(\theta_\nabla(t) - \theta_r(t)) \tag{5.5}$$

dove $M_v = \left\|v_\nabla(t)\right\|$, $\theta_\nabla = \angle v_\nabla(t)$ e

$$K_\omega(t) = \begin{cases} \frac{\dot{\theta}_\nabla(t) + K_c|\theta_\nabla(t) - \theta_r(t)|^\nu \cdot sign(\theta_\nabla(t) - \theta_r(t))}{\theta_\nabla(t) - \theta_r(t)} & |\theta_\nabla(t) - \theta_r(t)| \geq \xi \\ 0 & altrimenti \end{cases}$$

Una volta calcolate le velocità di riferimento $v(t)$ e $\omega(t)$, basta applicare la relazione 5.3 per avere le velocitá di riferimento relative alle due ruote del differential drive, ottenendo cosí i comandi da impartire agli attuatori del robot. La legge di controllo, specifica per questo metodo di navigazione, é implementata nel metodo commands. Quest'ultimo si trova nella classe APFAct (erede di Act) dove APF sta per artificial potential fields, il che vuol dire APFAct é idoneo a mobilitare tutti quei robot che sfruttano potenziali artificiali, indipendentemente dal modello cinematico. In output, il metodo commands restituisce le due velocità (lineare e angolare), che vengono poi convertite in altre due velocità ($(w_R$ e $w_L)$ che sono gli effettivi comandi da attribuire ai motori che fanno girare le due ruote. In particolare, si possono notare due aspetti importanti:

- Le velocità desiderate (vGrad) sono, dopo aver prelevato gli antigradienti dallo stato attualmente in funzione del robot, calcolate nella posizione del robot tramite la funzione coord2index
- $\dot{\theta}_\nabla(t)$ viene calcolato per via numerica da riga 11 a 16

Listing 5.2: Metodo che genera i comandi di velocità

```
function [vr,wr] = commands(~,rx,ry,rtheta,tspan,robot)
    grid = robot.grid;
```

```
    gradX = robot.state.gradX; gradY = robot.state.gradY;
    i = grid.coord2index([rx,ry]);
    thetaN = atan2(gradY(i(1),i(2)),gradX(i(1),i(2)));
    thetaDiff = atan2(sin(thetaN-rtheta),cos(thetaN-rtheta));
    vgrad = [gradX(i(1),i(2)) gradY(i(1),i(2))];
    Mv = norm(vgrad);
    vr = (Mv * cos(thetaDiff));

    rx1 = rx + vgrad(1)*tspan;
    ry1 = ry + vgrad(2)*tspan;
    j = grid.coord2index([rx1,ry1]);
    thetaN1 = atan2(gradY(j(1),j(2)),gradX(j(1),j(2)));
    thetaDdiff = atan2(sin(thetaN1-thetaN),cos(thetaN1-thetaN));
    thetaDdot = thetaDdiff/tspan;

    Kc = 10; eps = 0.001; v = 1;
    Kw = (thetaDdot + Kc * abs(thetaDiff)^v * ...
    sign(thetaDiff))/(thetaDiff+eps);
    wr = (abs(thetaDiff) >= eps) * Kw * (thetaDiff);
end
```

La legge di controllo, così implementata, assicura che il robot converga alla velocità di riferimento $v_\nabla(t)$ in un tempo finito [1].

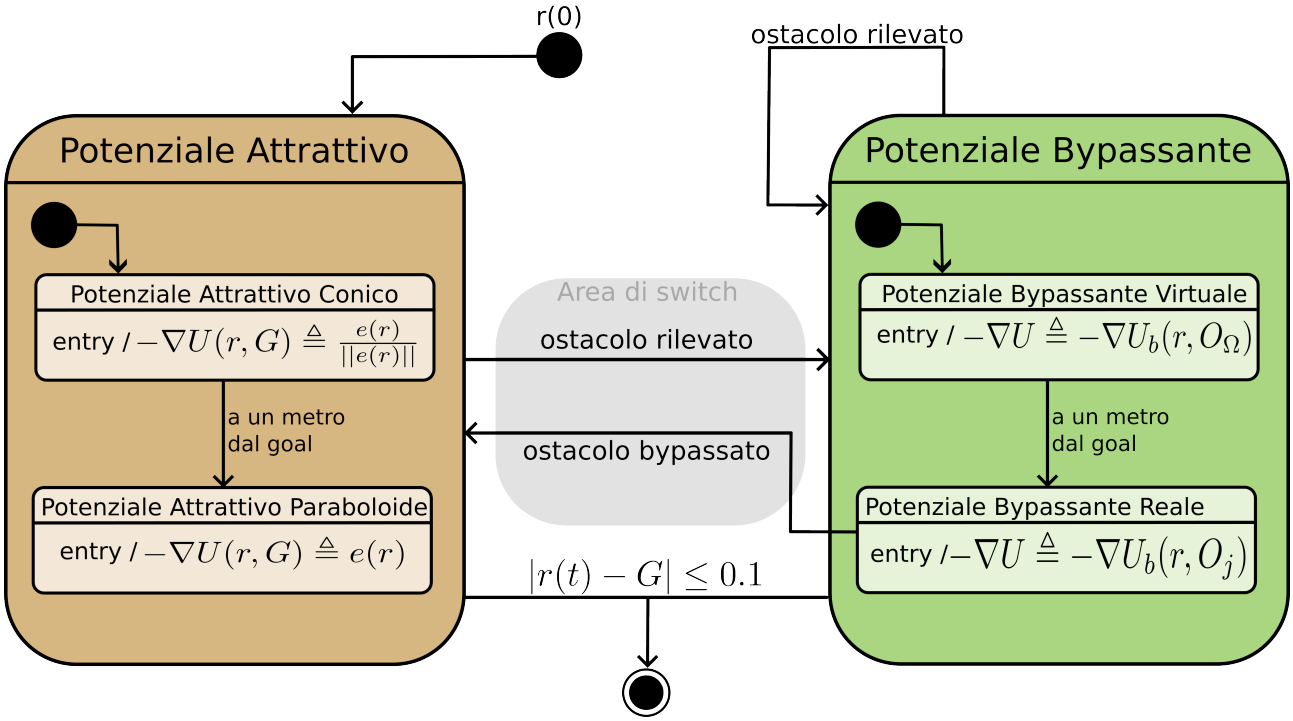
6 Modulo di pianificazione

Il modulo di pianificazione riceve ad ogni iterazione in input i dati processati dai sensori e in base a ciò decide a quale potenziale il robot deve fare riferimento, inoltrando successivamente la decisione al modulo di azione.

Nota implementativa
Il modulo di pianificazione, argomento principe della tesi, é stato sviluppato indipendentemente dagli altri. Cioè, come preannunciato nella sezione 1.2, si presuppone che il robot conosca in ogni istante la propria posizione e quella degli ostacoli all'interno del raggio di visione, come anche le loro velocità. Tale ipotesi semplificativa non provoca una perdita di generalità, grazie soprattutto al fatto che il software é stato progettato per essere modulare e con basso accoppiamento. Si parlerà meglio di questo aspetto nella sezione riguardante il modulo di visione.

Data la struttura intrinseca dell'algoritmo di navigazione che si basa su un meccanismo di switching tra due stati, per definizione mutuamente esclusivi, un modo per modellare visivamente il pianificatore é sicuramente uno statechart diagram, come riportato in figura 6.1. Il robot può trovarsi, in una logica aut aut, nello stato corrispondente al potenziale attrattivo oppure in quello relativo al potenziale bypassante.

Figure 6.1: Statechart Diagram



La strategia più naturale, sul piano dell'ingegneria del software, é di sfruttare il design pattern State per modellare correttamente il comportamento del robot. Ne consegue che gli oggetti di tipo SwitchingRobot hanno un campo RobotState che incapsula il comportamento relativo ad uno specifico antigradiente e

quindi responsabile di gestire i comportamenti **state-specific**. Perciò, RobotState si configura come un'interfaccia che offre all'esterno il metodo decision (già visto nel metodo 3.1, il quale prende in input il robot e l'ostacolo rilevato. RobotState contiene anche l'antigradiente come variabile d'istanza: com'è naturale aspettarsi, ogni stato (attrattivo, bypassante ecc) possiede un specifico antigradiente.

Listing 6.1: Classe asratta RobotState

```
classdef (Abstract) RobotState < handle
    properties
        gradX; gradY;
    end

    methods (Abstract)
        obj = decision(obj,robot,dObstacle);
    end
end
```

Il metodo decision, in pratica, decide se operare o meno uno switch del campo state (riportato di seguito tra i campi della classe SwitchingRobot).

Listing 6.2: Campi della classe SwitchingRobot

```
properties
   act APFAct = APFAct();
   sense Sense = Sense();
   state RobotState = DefaultRobotState();
   attractive Attractive;
end
```

6.1 Stato attrattivo

Lo stato iniziale del robot é quello attrattivo, conseguente all'imposizione del goal e il calcolo del potenziale attrattivo. Com'è prassi per il pattern State, ogni comportamento state-specific é incapsulato in una classe a parte. In questo caso specifico, la ridefinizione del comportamento specifico consiste nel ridefinire il metodo decision e calcolare gradX e gradY. Come riportato nel listing di seguito, se il robot si trova nello stato attrattivo, esso deve controllare innanzitutto se è stato rilevato un ostacolo, ed eventualmente prepararsi per bypassarlo. Il costruttore di VirtualBypassing é il metodo che calcola il potenziale bypassante e verrà trattato nel paragrafo successivo. Una volta calcolato il nuovo stato da assegnare al robot, viene richiamato il metodo setState che cambia il campo state dell'oggetto robot passato in input.

Listing 6.3: Classe astratta Attractive (erede di RobotState)

```
classdef Attractive < RobotState
   methods
      function obj = decision(obj,robot,dObstacle)
```

```
            if ~isempty(dObstacle)
                bypassing = VirtualBypassing(robot,dObstacle);
                bypassing.obstacle = [dObstacle.xc dObstacle.yc];
                robot.setState(bypassing);
            end
        end
    end
end
```

Come discusso nel capitolo sui potenziali artificiali, il potenziale attrattivo sarà conico fino a che $e(r) > 1$. Infatti, vediamo che nel costruttore di SwitchingRobot, il potenziale attrattivo viene inizializzato a Conical.

Listing 6.4: Costruttore di SwitchingRobot

```
function obj = SwitchingRobot(R,L,grid)
    obj@Robot(R,L,grid);
    obj.state = Conical(grid);
    obj.attractive = obj.state;
end
```

Nel costruttore di Conical é implementata l'equazione 2.4 con $k_2 = 1$. Quindi, nel momento in cui viene costruito l'oggetto SwitchingRobot, gli viene subito assegnato un potenziale attrattivo di tipo conico. L'implementazione a oggetti rende il software molto flessibile, poiché SwitchingRobot é indipendente dal potenziale che possiede.

Listing 6.5: Costruttore di Conical

```
function obj = Conical(grid)
    di = sqrt((grid.goal(1)-grid.X).^2 +
        (grid.goal(2)-grid.Y).^2);
    obj.gradX = (grid.goal(1)-grid.X)./di;
    obj.gradY = (grid.goal(2)-grid.Y)./di;
end
```

Nel caso del potenziale attrattivo conico vi sono due azioni da compiere: primo, eseguire l'implementazione di decision della classe padre di tutti i potenziali attrattivi (Attractive); secondo, controllare se il robot si trova a meno di un'unità di distanza dal goal ed eventualmente assegnare come stato quello paraboloide. La classe Paraboloidal si limiterà ad eseguire il metodo decision così come é stato ridefinito in Attractive.

Listing 6.6: Ridefinizione del metodo decision in Conical

```
function obj = decision(obj,robot,dObstacle)
    decision@Attractive(obj,robot,dObstacle);
    rx = robot.xc; ry = robot.yc; grid = robot.grid;
    if norm([rx,ry]-grid.goal) < 1
        newattractive = Paraboloidal(grid);
        robot.attractive = newattractive;
        robot.setState(newattractive);
    end
end
```

6.2 Switching allo stato bypassante

Il costruttore di VirtualBypassing (si spiegherà successivamente perché il prefisso Virtual), fulcro di tutto il progetto, ha la responsabilità di calcolare il potenziale bypassante e tutto ciò di necessario affinché il robot sia in grado di giungere in una posa in cui non collida con l'ostacolo; nel linguaggio dello statechart costituisce la entry action del macrostato "Potenziale Bypassante" in figura 6.1. L'obiettivo ultimo del metodo, perció, é di scambiare il potenziale correntemente in uso con quello bypassante. Naturalmente, il bypassing viene fatto con una ratio che si basa sulla velocità e la direzione di movimento dell'ostacolo da aggirare; in particolare, gli aspetti cardine da considerare sono:

- La scelta del verso di bypassing
- Nessuna discontinuità nella traiettoria dovuta allo switch tra i potenziali

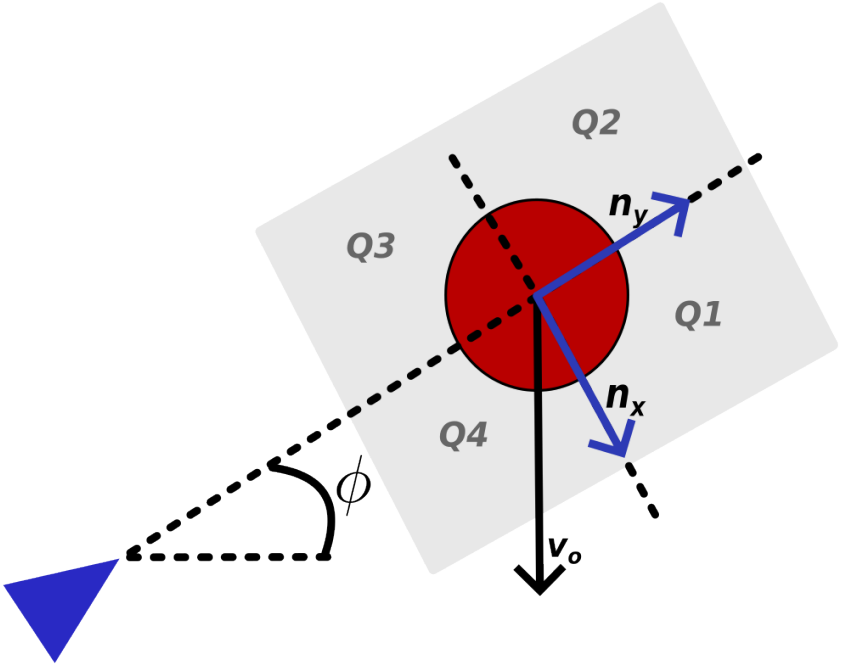
Primo aspetto La scelta del verso di bypassing é cruciale per una buona riuscita senza collisioni. Le possibilità, naturalmente, sono due: orario e antiorario.

Listing 6.7: Calcolo del verso di bypassing (orario o antiorario)

```
xr = pose(1); yr = pose(2); thetar = pose(3);
phi = atan2(obstacle.yc - yr,obstacle.xc - xr);
%Scelta verso
Rphi = [cos(phi-pi/2) cos(phi) ; sin(phi-pi/2) sin(phi)];
vphi = Rphi*obstacle.v;
if cos(atan2(vphi(2),vphi(1))) > 0
   sense = "clock";
else
   sense = "counterclock";
end
```

La variabile phi indica l'angolo tra robot e ostacolo. Rphi, invece indica una matrice di rotazione che adatta l'angolo a cui viaggia il robot al sistema di riferimento di phi.
Quindi vphi indica com'é orientata la velocità del robot rispetto all'angolo tra esso e l'ostacolo: se vphi sta nel primo o quarto quadrante, l'ostacolo va aggirato in senso orario, altrimenti antiorario. Dunque, informalmente, se l'ostacolo si muove a destra rispetto alla direzione del robot, bisogna girare alla sua sinistra per evitare una collisione.

Figure 6.2: Scelta del verso di bypassing



Secondo aspetto Lo scopo é quello di generare una traiettoria "smooth", che sia in grado di assicurare continuità al moto del robot. Ci si pone quindi il problema di calcolare, nel momento in cui un ostacolo é stato rilevato, la velocitá e direzione di movimento del robot dopo lo switch. Nella pratica questo problema consiste, in prima battuta, nel calcolare il **punto preciso** in cui eseguire lo switch e il **coefficiente moltiplicativo** c - compare nell'antigradiente del potenziale bypassante in formula 2.8 - che modula la velocità di bypassing. Tuttavia, un'ulteriore problematica é data dalla possibilità che il potenziale di bypass possa **riaggiornarsi**, conseguentemente al fatto che l'ostacolo potrebbe essere ripetutamente ri-rilevato (dopo la prima volta); in quest'ottica l'ostacolo verrà comunque bypassato, ma il movimento del robot potrebbe essere soggetto a discontinuità involute o, in generale, comportamenti non del tutto predicibili a priori.
L'idea di fondo dietro la **soluzione** é di calcolare il potenziale bypassante nell'istante τ in cui viene rilevato per la prima l'ostacolo e mantenerlo fisso per tutta la durata del bypass. Quindi

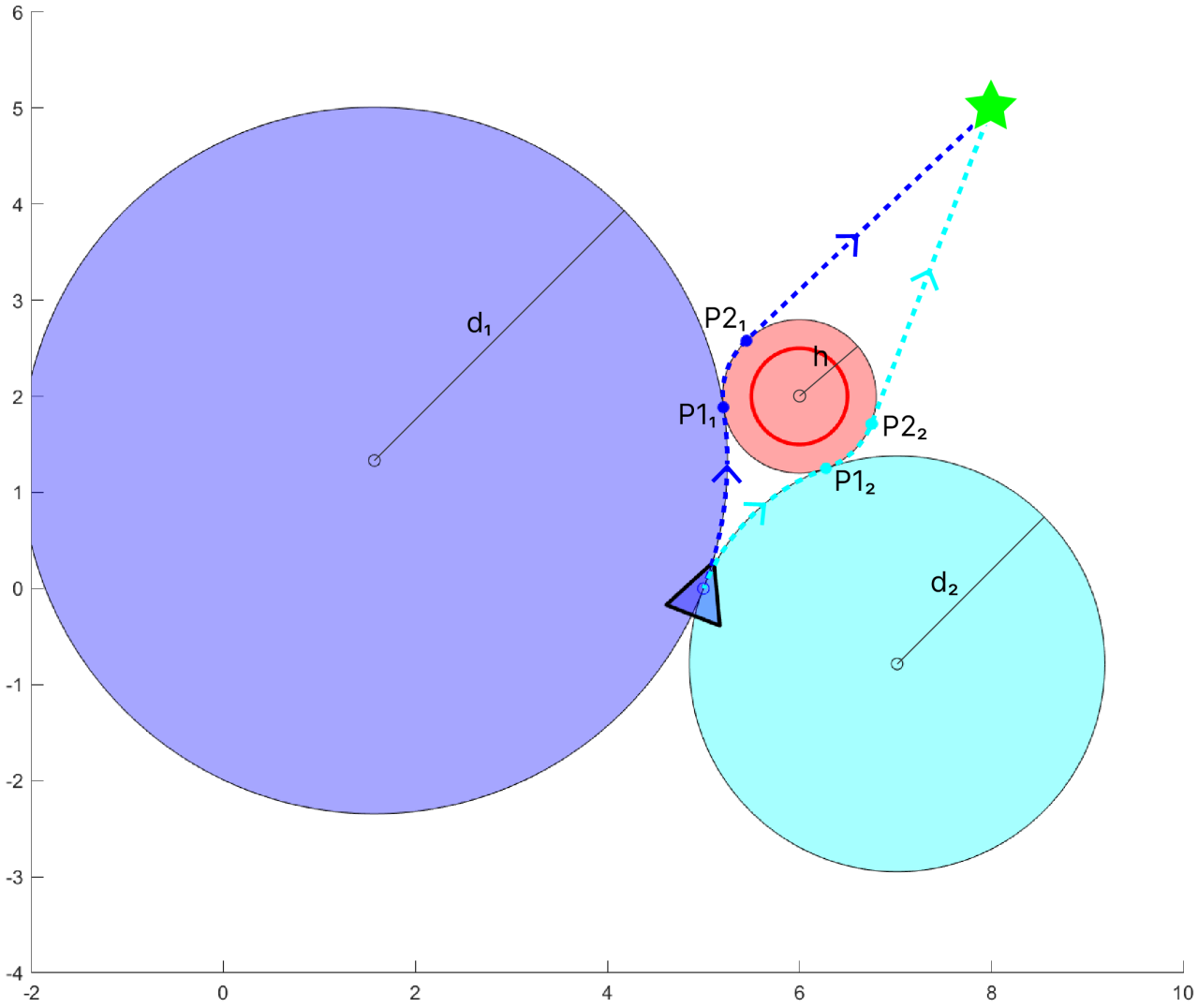
$$-\nabla U_b\left(r(t), O_j(t)\right) \Rightarrow -\nabla U_b\left(r(t), O_j(\tau)\right)$$

Nota
Considerando questa strategia, potrebbe sorgere un problema nel caso in cui un ostacolo cambi direzione durante il bypass. Tuttavia, si é deciso, per scopi esemplificativi, di tralasciare questo aspetto, anche perché può essere gestito indipendentemente dalla struttura portante del modulo di pianificazione, con l'aiuto del modulo di visione.

Una volta rilevato l'ostacolo nell'istante τ, il compito del pianificatore é di

assegnare al robot un nuovo potenziale da seguire; questo, al fine di rendere continuo lo switch, sarà posizionato in un ostacolo virtuale e non, come ci si potrebbe aspettare, in quello realmente rilevato. La motivazione dietro a questo procedimento é la seguente. Le linee di campo del potenziale bypassante sono delle circonferenze. Il robot possiede in ogni istante una velocità, data da un vettore bidimensionale. Quindi, per far "rientrare" il robot sul potenziale bypassante in maniera continua, bisogna trovare quella circonferenza che tange la traiettoria del robot nell'istante τ. La circonferenza cercata, in pratica, sarà una linea di campo del potenziale bypassante centrato nell'ostacolo virtuale. Solo successivamente il robot seguirà effettivamente il potenziale centrato nell'ostacolo reale. In figura 6.3 é rappresentato lo scenario tipico nel momento in cui va bypassato un ostacolo. Il cerchio rosso interno rappresenta l'ostacolo, mentre il rosso esterno rappresenta la circonferenza centrata nell'ostacolo reale che rappresenta una sorta di "delimitatore" oltre il quale non andare per non collidere con l'ostacolo. I cerchi blu e ciano indicano le (uniche) due alternative per il robot (come indicato anche dai numeri a pedice) e rappresentano le circonferenze centrate nell'ostacolo virtuale Ω; di conseguenza, la traiettoria che va dal robot a P1 rappresenta l'antigradiente del potenziale bypassante centrato in Ω, mentre quella da P1 a P2 l'antigradiente del potenziale bypassante centrato in $r(\tau)$. Notiamo che la traiettoria é continua, senza punti di rottura o virate improvvise. Naturalmente, la continuità andrà gestita anche in termini di velocità del robot.

Figure 6.3: Possibili percorsi dopo aver rilevato un ostacolo



Più in dettaglio, gli "ingredienti" del potenziale bypassante sono:

1. Posizione dell'ostacolo virtuale
2. Punto in cui switchare dall'ostacolo virtuale al reale (P1)
3. Punto in cui switchare dall'ostacolo reale al potenziale attrattivo (P2)
4. Coefficiente dell'antigradiente nell'ostacolo virtuale e reale

Il primo punto si configura come un sistema di due equazioni in due incognite, ovvero $[x_\Omega, y_\Omega]$. Uno dei due vincoli é dato dalla condizione di tangenza tra la circonferenza centrata in $[x_\Omega, y_\Omega]$ e la direzione di moto del robot in τ. A tal fine si può supporre, senza perdita di generalità, che il robot si trovi nell'origine all'istante τ. Da ciò ne deriva che l'equazione della circonferenza centrata in $[x_\Omega, y_\Omega]$ diventa

$$d^2 = x_\Omega^2 + y_\Omega^2$$

dove d é il raggio della circonferenza. La condizione di tangenza é soddisfatta se

$$x_\Omega = -m \cdot y_\Omega$$

dove $m = \tan(\theta_r)$, cioè la direzione di moto del robot. In pratica, questa equazione afferma che la retta, che congiunge il robot al punto $[x_\Omega, y_\Omega]$, deve essere perpendicolare a $\tan(\theta_r)$. L'altro vincolo si ricollega al secondo punto della "lista di ingredienti": bisogna imporre (con un ragionamento simile al primo vincolo) che la circonferenza "omega" sia tangente alla circonferenza attorno all'ostacolo reale nel punto di switch tra i due potenziali bypassanti. Di conseguenza si impone che

$$(x_\Omega - x_o)^2 + (y_\Omega - y_o)^2 = (d + h)^2$$

dove $[x_o, y_o]$ é il centro della circonferenza nell'ostacolo e h il suo raggio. Riunendo i due vincoli in un sistema si ottengono le due equazioni in due incognite

$$\begin{cases} x_\Omega = -m \cdot y_\Omega \\ (x_\Omega - x_o)^2 + (y_\Omega - y_o)^2 = (\sqrt{x_\Omega^2 + y_\Omega^2} + h)^2 \end{cases} \tag{6.1}$$

Risolvendo questo sistema per y_Ω si ottengono due possibili soluzioni

$$\begin{cases} y_{\Omega,1} = \frac{(-h^2+x_o^2+y_o^2)(mx_o-y_o+h\sqrt{m^2+1})}{2h^2m^2+2h^2-2m^2x_o^2+4mx_oy_o-2y_o^2} \\ \\ y_{\Omega,2} = \frac{-(-h^2+x_o^2+y_o^2)(mx_o-y_o+h\sqrt{m^2+1})}{2h^2m^2+2h^2-2m^2x_o^2+4mx_oy_o-2y_o^2} \end{cases} \tag{6.2}$$

x_Ω viene poi trovato per sostituzione. La scelta di una soluzione piuttosto che l'altra - in linea di massima, una sarà a sinistra dell'asse tra robot e ostacolo e l'altra sulla destra - dipenderà dal verso di bypassing (verrà affrontato successivamente, così come anche il calcolo di P1).
Una volta calcolata la posizione dell'ostacolo virtuale e del punto di switch tra ostacolo virtuale e reale, é necessario calcolare il punto P2 di switch tra il potenziale bypassante e quello attrattivo. L'idea per risolvere tale problema é di trovare, tra le infinitamente possibili, una retta che passi per il punto di goal e tanga la circonferenza centrata nell'ostacolo reale proprio in P2. Anche in questo caso le soluzioni possibili saranno due; quella idonea verrà scelta in base al verso di bypassing. Innanzitutto, il fascio di rette che partono dal punto di goal e passano dal punto cercato sono esprimibili come

$$y - y_g = m(x - x_g)$$

dove x e y sono le coordinate del punto P2 cercato. Inoltre, il punto P2 si troverà naturalmente sulla circonferenza centrata nell'ostacolo reale

$$(x - x_o)^2 + (y - y_o)^2 = h^2$$

e sostituendo in questa l'espressione della retta in funzione di y, si ottiene un'equazione che esprime i possibili punti di intersezione tra la generica retta e la circonferenza

$$(m^2+1)x^2 - 2(x_o + m(y_o - y_g + mx_g)x + (y_o - y_g + mx_g)^2 + x_o^2 - h^2 = 0$$

Ovviamente ci sono troppe incognite per poter risolvere questa equazione, ma noi interessa la coppia di rette tale per cui esse sono tangenti alla circonferenza. Ciò equivale a imporre che il punto di intersezione sia uno solo e quindi che quest'equazione di secondo grado abbia una sola soluzione. La conseguenza é che troviamo i due coefficienti angolari ponendo a zero il delta dell'equazione

$$(-2x_o - 2m(y_o + y_g + mx_g))^2 - 4(m+1)((y_o - y_g + mx_g)^2 + x_o^2 - h^2) = 0$$

e le due soluzioni saranno

$$\begin{cases} m_1 = \frac{(x_g - x_o)(y_o - y_g) + h\sqrt{-h^2 + (x_g - x_o)^2 + (y_g - y_o)^2}}{h^2 - (x_g - x_o)^2} \\ \\ m_2 = -\frac{(x_g - x_o)(y_o - y_g) + h\sqrt{-h^2 + (x_g - x_o)^2 + (y_g - y_o)^2}}{h^2 - (x_g - x_o)^2} \end{cases}$$

grazie alle quali é possibile trovare le due soluzioni per P2.
Infine, dopo aver trovato i primi tre "ingredienti", manca il quarto, che serve a garantire continuità nello switch tra diversi potenziali. L'obiettivo perciò é di trovare per i due potenziali un coefficiente modulante in maniera tale che

$$-\nabla U_b(P1, \Omega(t)) = -\nabla U_b(P1, O(t)) \tag{6.3}$$

e anche

$$-\nabla U_b(P2, O(t)) = -\nabla U_a(P2) \tag{6.4}$$

A parole, le velocità del robot tra antigradienti "adiacenti" devono essere uguali nei punti di switch. Quello che si fa in pratica é sfruttare il vincolo dato dal modulo del potenziale bypassante calcolato partendo dall'equazione 2.9

$$||\nabla U_b|| = \frac{c}{\sqrt{(x - x_o)^2 + (y - y_o)^2}} \tag{6.5}$$

dove $[x, y]$ é un punto incognito e $[x_o, y_o]$ la posizione dell'ostacolo.
Riassumendo il tutto, lo switch da potenziale attrattivo a potenziale bypassante avviene nei seguenti step, tutti facenti parte del costruttore di VirtualBypassing:

1. Calcolo del verso di bypass
2. Calcolo del raggio attorno all'ostacolo reale

Listing 6.8: Istruzioni per calcolare h

```
Rtheta =
    [cos(thetar-pi/2) sin(thetar-pi/2) ; cos(thetar) sin(thetar)];
vtheta= Rtheta*obstacle.v;
alphavtheta = atan2(vtheta(2),vtheta(1));
angdiff =
    abs(atan2(sin(-pi/2-alphavtheta),cos(-pi/2-alphavtheta)));
rv = robot.rv; ro = obstacle.raggio;
minlim = 0.2; maxlim = 0.2;
hmin = ro+minlim; hmax = rv - maxlim;
gradoinv = abs(1-angdiff/(pi/2));
h = ((hmax-hmin)*gradoinv + hmin);
```

Per calcolare il raggio attorno all'ostacolo reale - misura orientativamente la soglia massima di avvicinamento durante il bypass all'ostacolo rilevato in τ - si sfrutta una quantità che verrà battezzata **"grado di invasività"** dell'ostacolo. Tale quantità é misurata innanzitutto con la variabile angdiff, che contiene la differenza, in valore assoluto, tra la direzione di velocità dell'ostacolo θ_o(riportato nel sistema d'orientamento della direzione del robot, similmente alla scelta del verso di bypassing) e $\frac{-\pi}{2}$. Quest'ultimo angolo rappresenta il grado massimo di invasività: ad esempio, se il robot viaggia con un angolo di $\frac{\pi}{2}$, l'ostacolo gli sbatterà contro molto più velocemente se θ_o é opposto a θ_r. Il grado di invasività é minimo, invece, quando l'ostacolo viaggia ad un angolo retto rispetto al robot. Quindi

$$\begin{cases} |1 - \frac{angdiff}{\frac{\pi}{2}}| = 0 & grado\ minimo \\ |1 - \frac{angdiff}{\frac{\pi}{2}}| = 1 & grado\ massimo \end{cases}$$

(questo valore é assegnato alla variabile gradoinv) e si può notare che h é proporzionale a questo grado, nel senso che vale la seguente relazione

$$\begin{cases} gradoinv = 0 \Rightarrow h = R_i + minlim \\ gradoinv = 1 \Rightarrow h = r_v - maxlim \end{cases}$$

dove minlim e maxlim sono scelti arbitrariamente e indicano quindi rispettivamente la soglia massima di avvicinamento all'ostacolo (non può esssere troppo lasca) e il raggio massimo (non può essere troppo grande per non dover far fare al robot curve troppo brusche). Dunque h é una funzione lineare esprimibile come

$$h(gradoinv) : [0, 1] \rightarrow [hmin, hmax]$$

3. Calcolo soluzioni per l'ostacolo virtuale

Listing 6.9: Istruzioni per calcolare x_Ω e y_Ω

```
load yOmega; yOmega = double(subs(yOmega(:)));
xOmega = -m*yOmega;
d = double(sqrt(xOmega.^2 + yOmega.^2));
```

yOmega é un vettore di due componenti e si ottiene applicando l'equazione 6.2. Di solito una circonferenza sta alla sinistra rispetto a phi e l'altra a destra, ma può anche succedere, per eventualità legate alle condizioni di tangenza con la circonferenza, che entrambe le soluzioni consistano in circonferenze dallo stesso lato. A tal fine aiuta il prossimo passo.

4. Scelta circonferenza idonea

Listing 6.10: Istruzioni per scegliere la circonferenza idonea

```
1  bool1 = abs(norm(xOmega))>0.01&&sign(xOmega(1))==sign(xOmega(2));
2  bool2 = abs(norm(yOmega))>0.01&&sign(yOmega(1))==sign(yOmega(2));
3  if bool1 || bool2 %Ho due circonferenze dallo stesso lato
4     [~,indiceC] = min(d);
5  else %Non ho due circonferenze dallo stesso lato
6     if oSense == "clock" %voglio la circonferenza sinistra
7        indiceC = (sign(sin(angle)) == sign(xOmega(1))) + 1;
8     else %voglio quella destra
9        indiceC = ~(sign(sin(angle)) == sign(xOmega(1))) + 1;
10    end
11 end
12 xOmega = xOmega(indiceC) + xr; yOmega = yOmega(indiceC) + yr;
13 dOmega = d(indiceC); xo = xo0; yo = yo0;
```

Da riga 1 a 4 si controlla se le due circonferenze calcolate si trovano dallo stesso lato, nel qual caso ce ne sarà sicuramente una con un raggio molto grande. Ciò succede quando, da un lato o dall'altro, i vincoli di tangenza non riescono ad essere soddisfatti, per cui il risolutore simbolico di Matlab calcola una circonferenza che tange quella centrata nell'ostacolo reale, ma internamente invece che esternamente, motivo per cui avrà un raggio grande. La soluzione é semplicemente prendere la più piccola tra le due circonferenze in questo caso. Se siamo, invece, nel caso "ordinario", bisogna estrapolare la circonferenza d'interesse a seconda del verso di percorrenza dell'ostacolo. Ricordando che, nel calcolare la circonferenza dell'ostacolo virtuale, si é ipotizzato che il robot fosse nell'origine. In base a questo ragionamento, con l'ausilio della variabile angle che ribattezzerò α che indica la direzione di viaggio del robot si può fare il seguente ragionamento:

$$\begin{cases} \sin\alpha \geq 0 \cap clockwise \Rightarrow x_\Omega \leq 0 \\ \sin\alpha \geq 0 \cap anticlockwise \Rightarrow x_\Omega \geq 0 \end{cases}$$

In sostanza, se il verso di bypassing é orario, serve la circonferenza sinistra, che sarà quella per cui x_Ω é non negativo se l'orientamento del robot

"punta" verso il primo o secondo quadrante. Allo stesso modo si può ragionare per tutti gli altri casi.

5. Calcolo del punto P1

Listing 6.11: Istruzioni per calcolare P1

```
centerDir = [xo,yo]-[xOmega,yOmega];
centerDir = centerDir/norm(centerDir);
obj.P1 = [xOmega yOmega] + dOmega*centerDir;
```

Una volta calcolata la posizione dell'ostacolo virtuale, bisogna trovare il punto di tangenza tra la circonferenza trovata e quella centrata nell'ostacolo reale. Ciò si fa semplicemente sfruttando la conoscenza del fatto che, quando due circonferenza si tangono, la retta che congiunge i due centri passa per il punto di tangenza.

6. Calcolo del punto P2

Listing 6.12: Istruzioni per calcolare P2

```
obj.P2 = double(subs(solxp2(indiceP2)));
syms xp2; xp2 = obj.P2(1);
obj.P2(2) = double(subs(solyp2(indiceP2)));
```

La scelta della soluzione é basa sul verso di bypass: indiceP2 é pari a 2 se il verso é antiorario.

7. Calcolo del coefficiente modulante del potenziale bypassante reale

Listing 6.13: Istruzioni per calcolare il potenziale bypassante reale

```
agradX = robot.attractive.gradX;
agradY = robot.attractive.gradY;
j = grid.coord2index(obj.P2);
cO = norm([agradX(j(1),j(2)) agradY(j(1),j(2))])*h;
[obj.gradXO,obj.gradYO] = obj.antigradient(dO,grid,cO,oSense);
```

Per rendere continuo lo switch da potenziale bypassante (reale) a potenziale attrattivo deve valere il vincolo nell'equazione 6.4. Quindi, riprendendo l'equazione 6.5, bisogna imporre che il coefficiente modulante del potenziale bypassante nell'ostacolo reale sia

$$c = ||\nabla U_a(P2)|| \cdot h$$

.

8. Calcolo del coefficiente modulante del potenziale bypassante virtuale

Listing 6.14: Istruzioni per calcolare il potenziale bypassante virtuale

```
dr = sqrt((pose(1)-X).^2 + (pose(2)-Y).^2);
dp = sqrt((obj.P1(1)-X).^2 + (obj.P1(2)-Y).^2);
```

```
3  drp = sqrt((obj.P1(1)-pose(1))^2 + (obj.P1(2)-pose(2))^2);
4  k = grid.coord2index(obj.P1);
5  i = grid.coord2index([pose(1) pose(2)]);
6  currgradX = robot.state.gradX;
7  currgradY = robot.state.gradY;
8  cr = norm([currgradX(i(1),i(2)),currgradY(i(1),i(2))])*dOmega;
9  cp = norm([obj.gradXO(k(1),k(2)),obj.gradYO(k(1),k(2))])*dOmega;
10 cV = cr*(dp/drp) + cp*(dr/drp);
11 [obj.gradX,obj.gradY] = obj.antigradient(vObstacle,grid,cV,vSense);
```

Per quanto riguarda il potenziale bypassante nell'ostacolo virtuale, i vincoli da imporre sono due: uno é quello espresso in equazione 6.3 e l'altro é definibile come

$$c(r(t)) = ||\omega|| \cdot d \tag{6.6}$$

dove ω é il potenziale che il robot segue in $r(t)$, attrattivo o bypassante che sia, e d il raggio della circonferenza centrata nell'ostacolo virtuale. Bisogna dunque stabilire una funzione una funzione per c che rispetti entrambi i vincoli. Ciò che si é fatto nella pratica é stato basarsi sulla distanza dai punti "estremi" del potenziale bypassante (nell'ottica del movimento che fa il robot) $P1$ e $r(\tau)$. Le variabili dr e dp esprimono le distanze dai punti calcolate sulla meshgrid e cr e cp indicano i coefficienti nei due punti (i primi riferiti a $r(\tau)$, mentre drp indica la distanza tra i due punti. La funzione, al rigo 10, esprime i due vincoli richiesti:

$$\begin{cases} dr = 0 \Rightarrow c = cr \\ dp = 0 \Rightarrow c = cp \end{cases}$$

Le istruzioni appena elencate fanno tutte parte, in questa sequenza naturalmente, del costruttore di VirtualBypassing.

6.3 Stato bypassante

Una volta che é avvenuto lo switch tra potenziale attrattivo e bypassante, bisogna definire, ricollegandosi allo statechart in figura 6.1, il comportamento state-specific quando il robot sta bypassando un ostacolo.
Innanzitutto, anche mentre il robot sta bypassando, il modulo di percezione deve essere recettivo a nuovi ostacoli da, eventualmente, bypassare. Perciò, nella classe astratta Bypassing, il metodo decision é ridefinito in tal senso. Il metodo checkIfSame restituisce l'ostacolo ricevuto in ingresso invariato soltanto se questo é diverso dall'ostacolo attualmente in procinto di essere bypassato. A parole, dunque, se l'ostacolo rilevato é nuovo, decision della classe Bypassing fa uno switch (come se il robot si trovasse in uno stato attrattivo) ad un nuovo stato bypassante relativo al nuovo ostacolo.

Listing 6.15: Classe astratta Bypassing (erede di RobotState)

```
classdef (Abstract) Bypassing < RobotState

    properties
        obstacle; %ostacolo che sta per essere bypassato
    end

    methods
        function obj = decision(obj,robot,dObstacle)
            %Controllo se l'ostacolo rilevato e' diverso
            %da quello che sto bypassando
            dO = obj.checkIfSame(dObstacle);
            robot.attractive.decision(robot,dO);
        end

        function dO = checkIfSame(obj,dObstacle)
            %%Ostacolo non rilevato
            if isempty(dObstacle)
                obj.obstacle = []; dO = []; return;
            end
            %%Ostacolo rilevato
            %Ho ancora in vista l'ostacolo che sto bypassando
            if ~isempty(obj.obstacle)
            %Controllo se l'ostacolo rilevato e' lo stesso
                distance =
                    sqrt((obj.obstacle(1) dObstacle.xc)^2 +
                    (obj.obstacle(2) - dObstacle.yc)^2);
                if abs(distance - norm(0.05*(dObstacle.v))) < 0.01
                    dO = []; %SI
                else
                    dO = dObstacle; %NO
                end
            end
            %Non ho piu in vista l'ostacolo che sto bypassando
            if isempty(obj.obstacle)
                dO = dObstacle;
            end
            %Aggiorno l'ostacolo corrente con quello rilevato
            obj.obstacle = [dObstacle.xc dObstacle.yc];
        end
    end
end
```

Da notare che, per controllare che due ostacoli siano uguali, si é fatto un controllo di tipo odometrico basandosi sulla velocità dell'ostacolo e la posizione del robot. Ciò potrebbe essere soggetto ad errori di misura poichè il robot (e gli ostacoli) nella realtà non é fermo. Tuttavia, considerando un tempo di campionamento abbastanza piccolo, si può in merito a ciò ipotizzare che il robot e gli ostacoli siano fermi tra uno "snapshot" e l'altro.

Ammesso che, invece, il robot non incontri nuovo ostacoli. Come detto prece-

dentemente, l'ordine dei potenziali bypassanti é sempre virtuale e poi reale.

Listing 6.16: Ridefinizione del metodo decision di VirtualBypassing

```
function obj = decision(obj,robot,dObstacle)
    decision@Bypassing(obj,robot,dObstacle);
    if norm([robot.xc robot.yc] - obj.P1) < 0.1
        robot.setState(RealBypassing(obj));
    end
end
```

Dunque, il metodo decision ridefinito nella classe VirtualBypassing deve controllare a ogni iterazione che il robot non sia arrivato al punto P1; se così é, avviene uno switch al potenziale centrato nell'ostacolo rilevato nell'istante τ. Successivamente, si esegue ad ogni iterazione la ridefinizione di decision della classe RealBypassing.

Listing 6.17: Ridefinizione del metodo decision di RealBypassing

```
function obj = decision(obj,robot,dObstacle)
    decision@Bypassing(obj,robot,dObstacle);
    if norm([robot.xc robot.yc] - obj.virtualbypass.P2) < 0.1
        robot.setState(robot.attractive);
    end
end
```

Similmente, si controlla ad ogni iterazione che il robot non sia già arrivato nel punto P2, nel qual caso bisogna switchare al potenziale attrattivo.

7 Modulo di percezione

L'unico scopo del modulo di percezione relativamente a questo specifico lavoro di tesi é di controllare se gli ostacoli siano d'intralcio alla traiettoria del robot verso il goal. Si é fatto dunque un'importante ipotesi semplificativa che consiste nel supporre che il robot conosca in ogni istante la posizione, velocità e identità degli oggetti che deve evitare. Ciò é stato fatto per due motivi: primo, per concentrarsi sullo sviluppo del modulo di pianificazione; secondo, perché é sufficiente rimuovere queste ipotesi aggiungendo un modulo di visione (che lavora indipendentemente dall'algoritmo di pianificazione), che sia in grado, ad esempio, di fare object-tracking come anche riconoscerli. Lo stesso ragionamento si può fare per la propriocezione del robot - la percezione della sua posizione nell'ambiente - per la quale é sufficiente nel software un modulo che integri odometria e, eventualmente, sensori esterni per migliorare la precisione delle misurazioni.
Le funzionalità del modulo di percezione sono racchiuse nella classe Sense. Il metodo scan riceve in input il robot e controlla, tra gli ostacoli presenti, quali sono quelli posizionati entro un raggio rv dal robot e dentro un tubo direzionato dal robot al goal di larghezza rm. Figura tubo

Listing 7.1: Classe per il modulo di percezione

```
classdef Sense

    methods
        %% Method that looks in a radius rv
        %%and a tube T(t) if there are any obstacles
        function dObstacle = scan(obj,robot)
            rx = robot.xc; ry = robot.yc; rv = robot.rv;
            n = length(robot.grid.obstacles);
            distances = zeros(1,n) + inf;
            dObstacles = cell(1,n);
            for j = 1 : n
                o = robot.grid.obstacles(j);
                distance = norm([rx ry] - [o.xc o.yc]);
                if distance <= rv && obj.tube(robot,o)
                    dObstacles{j} = o;
                    distances(j) = distance;
                end
            end
            %Ostacolo tra quelli rilevati a distanza minima
            [~,pos] = min(distances);
            dObstacle = dObstacles(pos); dObstacle = dObstacle{1};
        end
    end

    methods(Access=private)
        %% Method that builds the tube T(t)
        function result = tube(~,robot,obstacle)
```

```
            pose = robot.getPose(); rx = pose(1); ry = pose(2);
            grid = robot.grid;
            G = grid.goal;

            rm = robot.rv + 1;
            angle = atan2(G(2)-ry,G(1)-rx);
            deltaX = rm/2*sin(angle); deltaY = rm/2*cos(angle);
            x1 = rx + deltaX; y1 = ry - deltaY;
            x4 = rx - deltaX; y4 = ry + deltaY;
            x2 = G(1) + deltaX; y2 = G(2) - deltaY;
            x3 = G(1) - deltaX; y3 = G(2) + deltaY;

            m14 = (y4-y1)/(x4-x1); q14 = m14*x1 - y1;
            if abs(m14) > exp(10)
                v14 = (obstacle.xc - x1);
            else
                v14 = (obstacle.yc - m14*obstacle.xc + q14);
            end

            m12 = (y2-y1)/(x2-x1); q12 = m12*x1 - y1;
            if abs(m12) > exp(10)
                v12 = (x1 - obstacle.xc);
            else
                v12 = (obstacle.yc - m12*obstacle.xc + q12);
            end

            m34 = (y4-y3)/(x4-x3); q34 = m34*x3 - y3;
            if abs(m34) > exp(10)
                v34 = (x4 - obstacle.xc);
            else
                v34 = (obstacle.yc - m34*obstacle.xc + q34);
            end

            if (angle >= 0 && angle <= pi/2) %first quadrant
                result1 = v14 >= 0;
                result2 = v12 >= 0; result3 = v34 <= 0;
            elseif (angle > pi/2 && angle < pi) %second quadrant
                result1 = v14 >= 0;
                result2 = v12 <= 0; result3 = v34 >= 0;
            elseif (angle >= -pi && angle <= -pi/2 || angle == pi ) %third qu
                result1 = v14 <= 0;
                result2 = v12 <= 0; result3 = v34 >= 0;
            else %fourth quadrant
                result1 = v14 <= 0;
                result2 = v12 >= 0; result3 = v34 <= 0;
            end
            result = result1 && result2 && result3;
        end
    end
end
```

8 Simulazione

L'algoritmo é stato simulato in ambiente Matlab. A tal proposito viene qui ripreso il codice che costituisce il "main" del modulo software.

Listing 8.1: Entry point del modulo software

```
function obj = start(obj)
   %Simulation data
   e = norm([obj.xc,obj.yc]-obj.grid.goal);
   tspan = 0.05; tsim = 0; samples = 0;
   %Starting simulation
   while(e > 0.1 && tsim < 20)
      %Sensed obstacle (empty array if nothing was sensed)
      dObstacle = obj.sense.scan(obj);
      %New directive
      obj.state.decision(obj,dObstacle);
      %Giving the command to the actuators
      [obj.xc,obj.yc,obj.theta] = obj.act.move(obj,tspan);
      %Refreshing the error
      e = norm([obj.xc,obj.yc]-obj.grid.goal);
      tsim = tsim + tspan; pause(0);
   end
end
```

Dei metodo scan e sense si é parlato ampiamente, ma non del metodo move della classe Act. Tale metodo prende in ingresso il robot e l'intervallo di campionamento della simulazione, restituendo in output la nuova posizione del robot. Il metodo move sostituisce quindi gli effettivi attuatori del robot in maniera tale da poter simulare il movimento del robot in assenza di un robot reale.

Listing 8.2: Metodo responsabile di far muovere il robot in simulazione

```
function [xc,yc,theta] = move(obj,robot,tspan)
   rx = robot.xc;
   ry = robot.yc;
   rtheta = robot.theta;
   [vr,wr] = obj.commands(rx,ry,rtheta,tspan,robot);
   Xdot = robot.ode(vr,wr);

   rx2 = rx + tspan/2*Xdot(1);
   ry2 = ry + tspan/2*Xdot(2);
   rtheta2 = rtheta + tspan/2*Xdot(3);
   [vr,wr] = obj.commands(rx2,ry2,rtheta2,tspan,robot);
   Xdot = robot.ode(vr,wr);

   xc = rx + tspan*Xdot(1);
   yc = ry + tspan*Xdot(2);
   theta = rtheta + tspan*Xdot(3);
end
```

Come si vede nel listing, il metodo genera dapprima i comandi di velocità con il metodo commands (di questo metodo si parlerà meglio nella sezione riguardante

il modulo di azione), con i quali genera una variabile Xdot - derivata numerica nella posa attuale del robot - attraverso il metodo ode specifico per ogni modello cinematico. In pratica, quello che si fa é integrare numericamente attraverso la derivata numerica Xdot, come si vede da riga 14 a 16. La posizione del robot tra un istante di campionamento é pari alla posizione attuale più il prodotto tra la derivata (istantanea) della posizione e l'intervallo di campionamento. Nello specifico, in questo metodo si é scelto di adottare una doppia integrazione numerica (Eulero esplicito), denominata anche metodo "Runge-Kutta 2".
Una volta definito quali sono i componenti dell'algoritmo a livello macroscopico, si può vedere come si avvia la simulazione nel suo totale nel seguente script.

Listing 8.3: Istruzioni per avviare la simulazione

```
1  grid = Grid(50,10);
2  grid.setGoal([6,9]);
3  %Obstacles and robot initial configuration
4  grid.addObstacle(Obstacle(2,4,[1;0]));
5  grid.addObstacle(Obstacle(10,6,[-1;0]));
6  grid.addObstacle(Obstacle(1.5,10,[0.5;-0.5]));
7  n = length(grid.obstacles);
8  r = SwitchingRobot(0.15,0.3,grid);
9  %Inizio simulazione
10 r.start();
```

Si vede dunque che viene prima generato un oggetto di tipo Grid, ovvero la struttura dati che serve a rappresentare l'ambiente circostante. Successivamente vengono "immessi" gli ostacoli (i primi due parametri del costruttore indicano la posizione e il terzo la velocità). Dopo di che, viene costruito il robot (il primo parametro del costruttore indica il raggio in centimetri delle ruote, mentre il secondo la distanza tra le due ruote). Infine, viene avviato il metodo start, già trattato in precedenza e la simulazione ha inizio.

9 Risultati

Nel testare l'algoritmo di path planning si é usata la seguente configurazione di parametri:

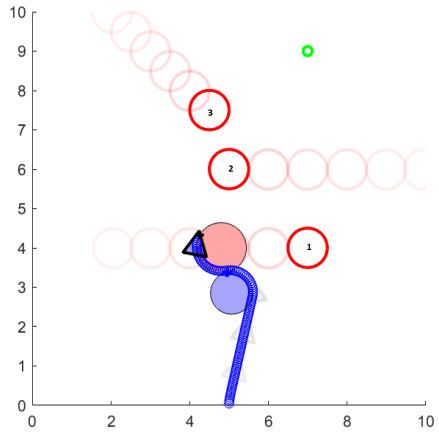
- Il raggio di visione $R_v = 1.5m$
- La dimensione degli ostacoli $R_i = 0.5m$
- Le soglie di avvicinamento $minlim = 0.2m$ e $maxlim = 0.2m$
- La larghezza del tubo direzionato $R_m = 3m$
- I parametri della legge di controllo $K_c = 10, c = 1, \xi = 0.001$

L'algoritmo é stato testato con successo su varie configurazioni di diverso tipo; a titolo esempio se ne riporteranno alcune che riassumano un po' tutti i casi "ordinari" e un'altra a parte in cui si fa un confronto con il metodo classico nell'ambito di un caso "patologico" per quest'ultimo (i minimi locali).
Da notare che il sistema di riferimento cartesiano é espresso in metri.

9.1 Tre ostacoli

La configurazione di partenza é la seguente: i tre ostacoli all'istante iniziale sono $O_1 = [2, 4]$ con velocità $v_1 = [1, 0]$, $O_2 = [10, 6]$ con velocità $v_2 = [-1, 0]$ e $O_3 = [2.5, 10]$ con velocità $v_3 = [-0.5, -0.5]$. In figura 9.6 sono riportati due punti salienti della simulazione.I cerchi pieni indicano le circonferenze di ostacolo virtuale e reale nell'istante in cui l'ostacolo rilevato, le circonferenze vuote indicano gli ostacoli veri e propri.
In figura 9.1a é riportato il movimento del robot nelle varie istantanee della simulazione: il cerchio rosso pieno rappresenta la posizione dell'ostacolo O_1 nel momento in cui viene rilevato; vediamo perciò che il robot gira a sinistra in maniera tale da bypassare l'ostacolo in senso orario per non collidere. Se, naturalmente, il verso scelto fosse stato quello antiorario, il robot avrebbe colliso. Inoltre, notiamo che il robot non si cura di informazioni sensoriali riguardanti gli altri ostacoli, che non influenzano il bypassing in atto.
In figura 9.1b notiamo che avviene il bypass dell'ostacolo O_3. Tuttavia, mentre il primo bypass avviene nella maniera "ordinaria" (switch da potenziale attrattivo a bypassante e di nuovo all'attrattivo), qui assistiamo ad uno switch da potenziale bypassante dell'ostacolo O_2 (indicato dai cerchi pieni più trasparenti) al potenziale bypassante di O_3, come voluto dall'algoritmo, che mantiene attivo il modulo di visione anche durante un bypass.

(a) Bypassing del primo ostacolo



(b) Bypassing del terzo ostacolo

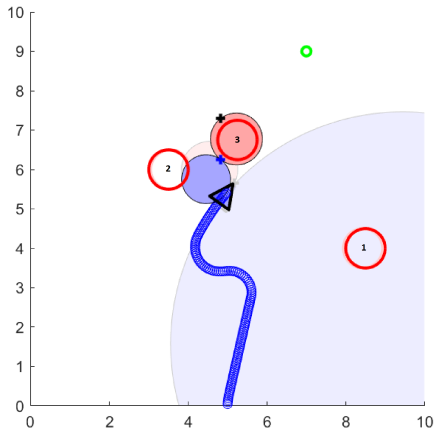


Figure 9.1: Snapshots della simulazione

In riferimento alla figura 9.6, i cerchi rossi vuoti indicano gli ostacoli in movimento. I cerchi rossi pieni indicano le circonferenze attorno all'ostacolo rilevato all'istante τ, mentre i cerchi blu pieni le circonferenze attorno all'ostacolo virtuale.
Alla fine, il robot avrà percorso la traiettoria riportata nel seguente snapshot. Il robot é arrivato con successo nel goal, evitando gli ostacoli incontrati.

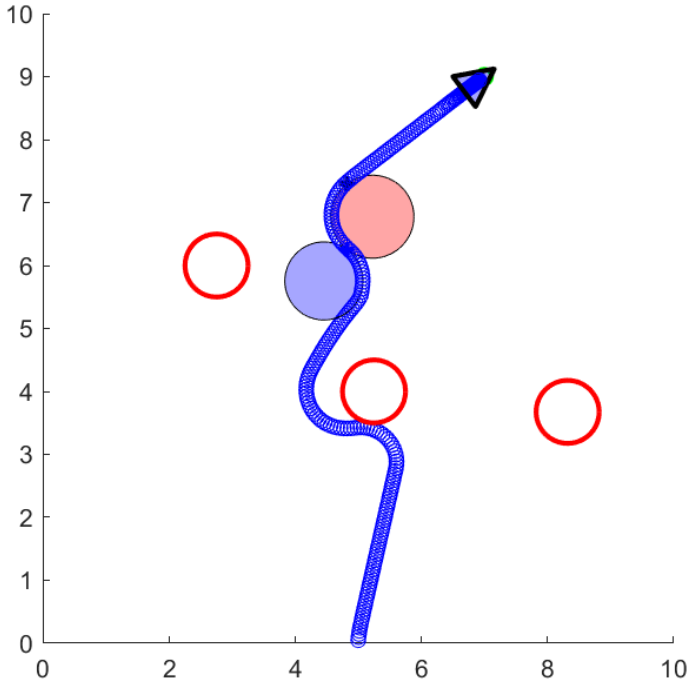


Figure 9.2: Traiettoria finale

9.2 Otto ostacoli fissi

In figura 9.3 si può notare come il robot gestisca, con una traiettoria continua e senza collidere con gli ostacoli, una situazione in cui, in proporzione alle dimensioni dell'ambiente (in questo caso 10x10[m]), risulti un'alta densità di ostacoli. In questo esempio sono stati "immessi" nell'ambiente otto ostacoli fissi. Da notare che, visto che gli ostacoli sono fissi e il robot riceve questo informazione grazie al modulo di visione, lo spostamento dalla traiettoria "principale" é minima, per mantenere l'ottimalità della traiettoria, ma sufficiente per non collidere con l'ostacolo.

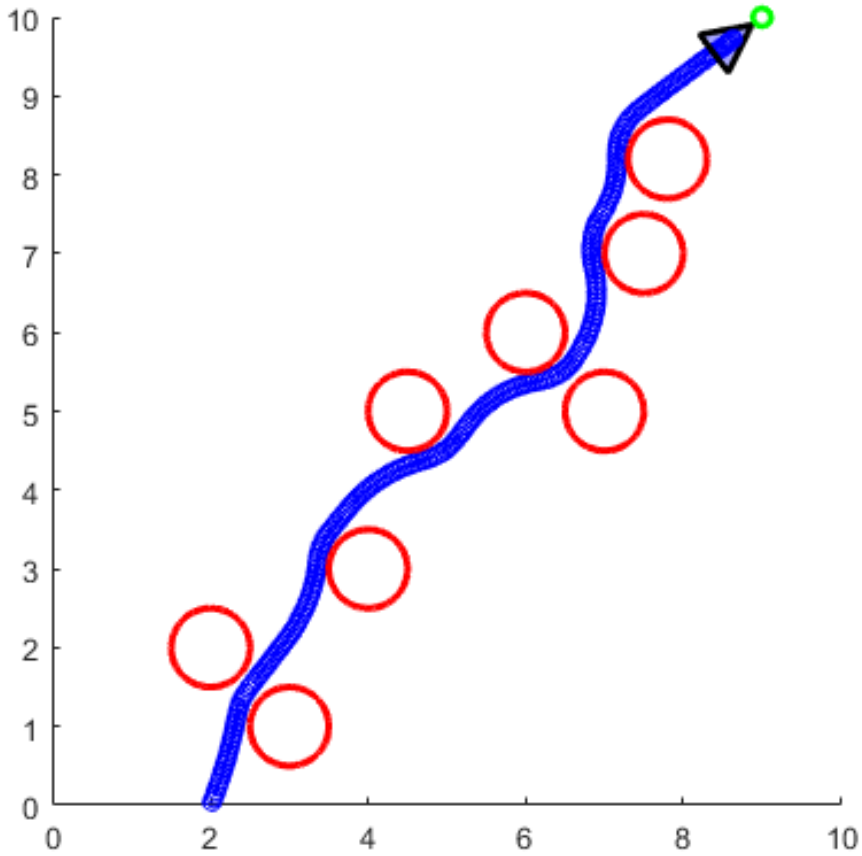
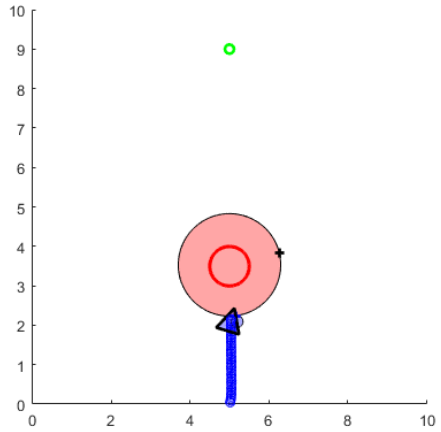


Figure 9.3: Simulazione dell'algoritmo con otto ostacoli fissi

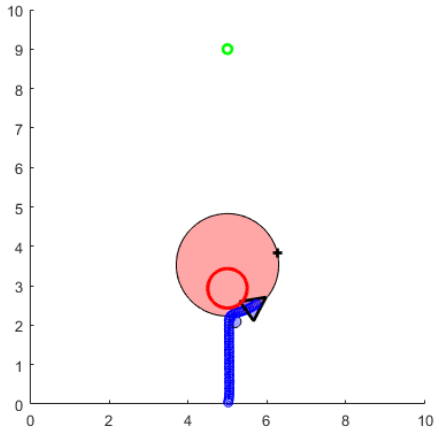
9.3 Considerazione sul calcolo di h

Un parametro sicuramente molto importante é il raggio h della circonferenza centrata nell'ostacolo da bypassare. Da esso dipende la capacità del robot di bypassare correttamente l'ostacolo, senza collidere durante il bypass.

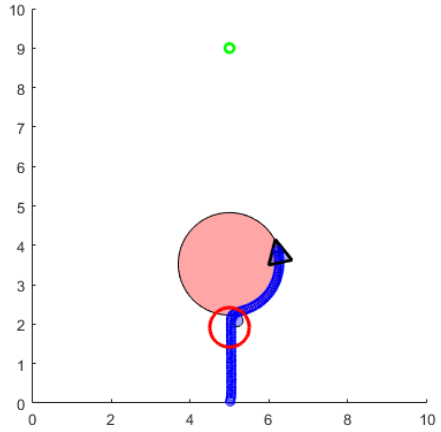
(a) Snapshot prima del bypass all'istante τ



(b) Snapshot durante il bypass



(c) Snapshot all'istante $\tau + \Delta t_{bypass}$



(d) Snapshot nella posizione finale

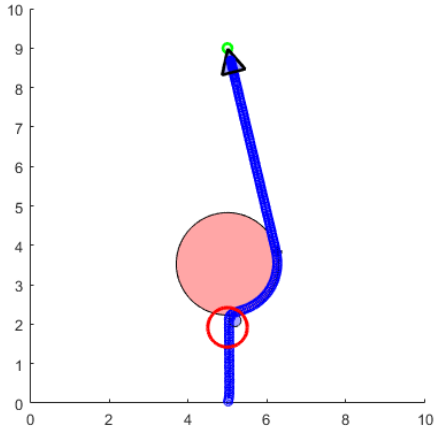


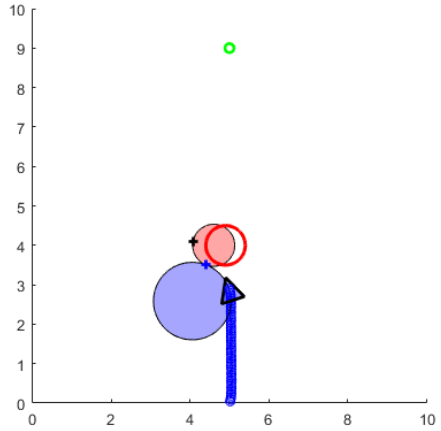
Figure 9.4: Bypassing con grado di invasività massimo

Da notare che il cerchio rosso vuoto indica l'ostacolo, mentre quello pieno indica la circonferenza attorno all'ostacolo all'istante *tau*. In figura 9.4c l'ostacolo (cerchio rosso vuoto) si trova, all'istante $t = \tau + \Delta t_{bypass}$ - dove Δt_{bypass} é una durata legata al bypass - in posizione $[5, 2]$, ovvero lì dove si trovava il robot all'istante $t = \tau$. Perciò non vi é nessuna collisione, visto che a $t = \tau + \Delta t_{bypass}$ il robot si trova, ispezionando la figura, in posizione circa $[6, 4]$.

In figura 9.4 si nota come il robot bypassi l'ostacolo tenendosi ad una distanza di sicurezza piuttosto elevata. Ciò é dovuto al fatto che il grado di invasività é massimo, per cui il robot deve spostarsi dalla sua traiettoria più in fretta possibile. Con un h più piccolo ci sarebbe stata quasi sicuramente una collisione, mentre con h più grande il robot avrebbe dovuto tenersi ad una distanza di sicurezza pari praticamente al raggio di visione, il che avrebbe comportato una curvatura troppo brusca (si nota che già in queste condizioni la curvatura non é molto "gentile").
In figura 9.5 (per semplicità si riporterà soltanto gli snapshot durante e dopo

il bypassing) invece, si può assistere ad un caso in cui il grado di invasività dell'ostacolo é minimo. Il robot, nei confronti del caso precedente si sposta molto meno dalla sua traiettoria e il tempo impiegato per bypassare é anche minore.

(a) Durante il bypassing



(b) Dopo il bypassing

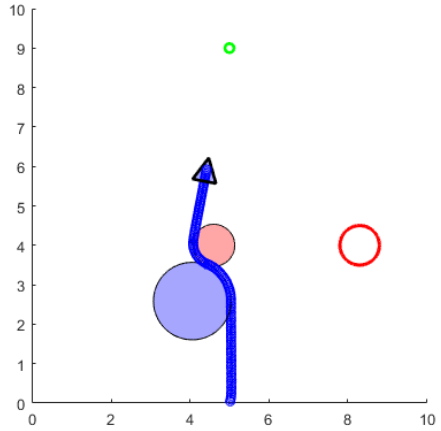
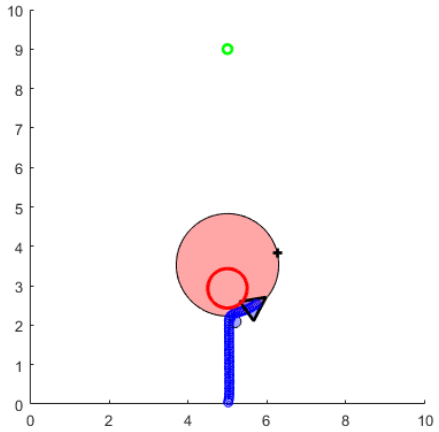


Figure 9.5: Bypassing con grado di invasività minimo

9.4 Considerazione sullo switch al potenziale attrattivo

Collegandosi alla sezione precedente, un'ulteriore considerazione che si potrebbe fare riguarda la durata dello switch: può succedere che, a fronte di una durata piuttosto breve in cui il robot potrebbe collidere con l'ostacolo, la traiettoria fa un giro più largo, allungando così il tempo rispetto a quello effettivamente necessario per eliminare l'ostacolo dal raggio di visione. Dalla figura 9.6a si può intuire che l'ostacolo aveva già percorso parecchi metri quando il robot stava ancora finendo di bypassare. Naturalmente, questo prolungamento temporale é giustificato dalla necessità di rendere la traiettoria continua durante gli switch. Si potrebbe, tuttavia, scegliere di variare l'algoritmo e fare in modo che il robot passi al potenziale attrattivo non appena l'ostacolo da bypassare non é più nel suo raggio di visione (e nel tubo direzionato). Ne consegue che la traiettoria é quella mostrata in figura 9.6b, in cui vediamo una curvatura più ripida in corrispondenza dello switch dal potenziale bypassante a quello attrattivo. Si può notare, però, che, a parità di tempo trascorso (i due ostacoli si trovano nella stessa posizione nello snapshot) il robot nel secondo caso é più vicino al goal. Dunque, a fronte di un tempo minore per arrivare nel goal, bisogna sacrificare la continuità della traiettoria.

(a) Il robot switcha in P2



(b) Il robot switcha prima

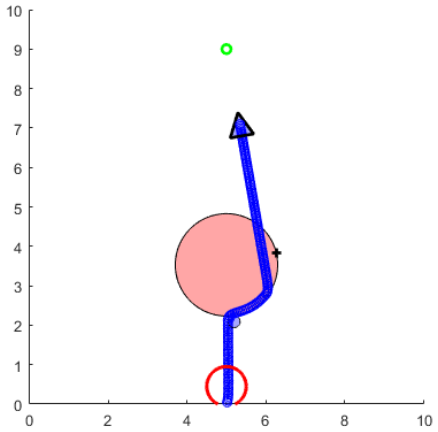


Figure 9.6: Alternativa allo switching in P2

9.5 Minimi locali

Come ultimo esempio si considera un caso in cui nel metodo classico é limitante il problema dei minimi locali. A tal fine la configurazione di partenza é $G = [6, 10]$, $O_1 = [5, 4]$ con $v_1 = [0, 0]$ e $O_2 = [7, 4]$ con $v_2 = [0, 0]$ (entrambi gli ostacoli sono fermi). Il moto del robot inizia da $[5, 0]$. In figura 9.8 si può vedere che il robot (gli scatter points rossi) compie un percorso che inizialmente va verso il goal (il minimo), ma, una volta arrivato vicino ai due ostacoli, inizia a ruotare sullo stesso punto. Ciò é dovuto al fatto che non vi é nessuna forza a spingerlo avanti, conseguentemente al fatto che i potenziali repulsivi e attrattivi si annullano a vicenda in quel punto.

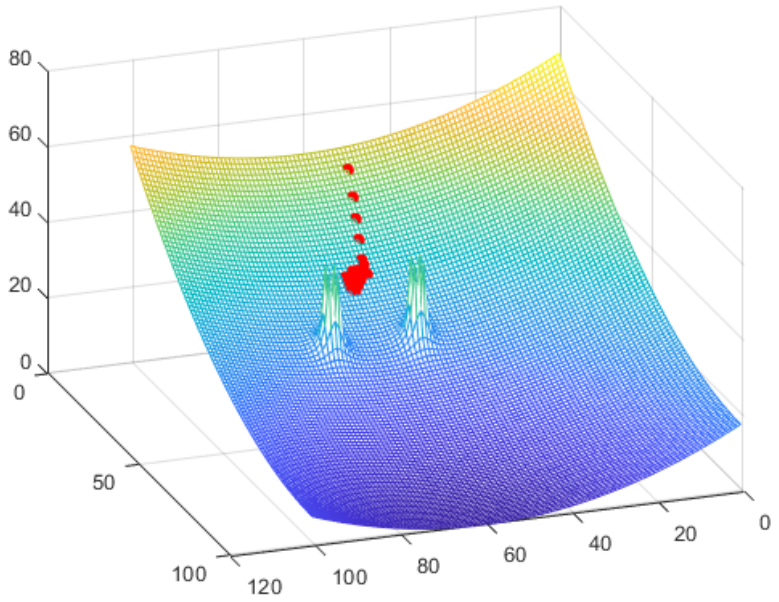


Figure 9.7: Path planning con metodo classico

Al contrario, con l'algoritmo sviluppato in questo lavoro di tesi, il problema dei minimi locali é assente. Il robot rileva l'ostacolo alla sinistra, lo evita e non si fa "disturbare" dall'ostacolo alla destra, che non intralcia minimamente l'avanzamento verso il goal.

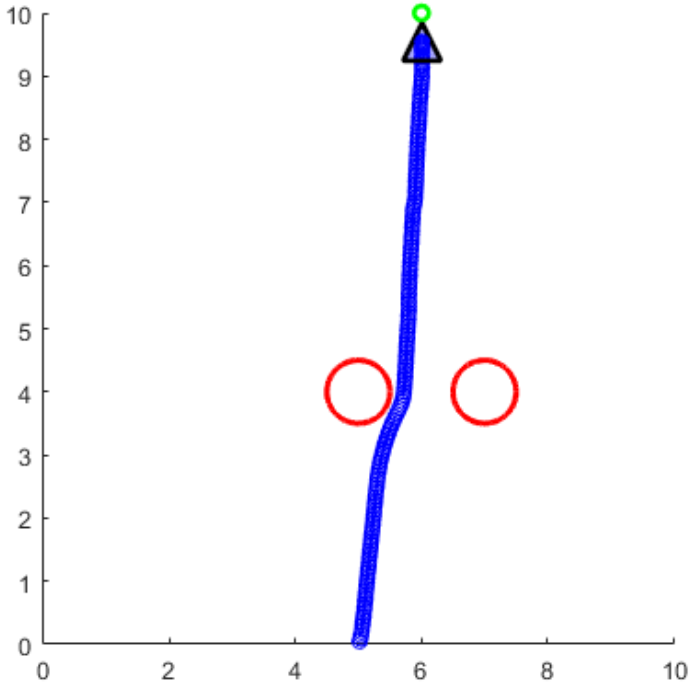


Figure 9.8: Path planning con metodo bypassante

Ciò é un esempio lampante del concetto "all'algoritmo non servono informazioni globali riguardanti gli altri ostacoli": se nel metodo classico, infatti, il moto del robot viene influenzato non solo dall'ostacolo più vicino (da aggirare), ma anche da quelli più lontani, nel metodo proposto nel presente elaborato il robot fa esclusivamente uso di ciò che succede nel suo raggio di visione, agendo di conseguenza. Quindi, come ribadito anche nel paragrafo precedente, la scelta del raggio di visione (e anche della distanza di sicurezza dagli ostacoli) é cruciale per una buona riuscita del path planning. Un raggio di visione troppo grande potrebbe risultare in un "overkill", nel senso che il robot potrebbe voler evitare ostacoli che in realtà sono molto lontani; un raggio di visione troppo piccolo sfocerebbe eventualmente in una collisione.

10 Conclusione

Le applicazioni dell'algoritmo di path planning sono ovviamente innumerevoli, come lo sono anche le espansioni future.
Come già detto precedentemente, l'algoritmo può essere facilmente integrato con un modulo di visione ad-hoc. Allo stesso modo, può essere adattato ad ambienti specifici, ad esempio mappe di ambienti chiusi, e integrato con tecniche di SLAM [5]. Brevemente, queste tecniche consistono nel localizzare il robot localmente e costruire la mappa mentre esso esplora l'ambiente.
Casi studio interessanti possono essere anche quelli in cui gli ostacoli siano di forma diversa e non circolari, come supposto in questo lavoro di tesi. Un'altra generalizzazione che si può fare in merito agli ostacoli riguarda il loro movimento: può essere studiato un movimento a "zig-zag" e possono essere analizzate possibili situazioni di stallo.
Un'ulteriore espansione che si potrebbe fare sarebbe rendere l'algoritmo di path planning distribuito, ovvero ideale per un sistema multi-agente che sia coordinato per raggiungere un obiettivo comune, come ad esempio nel robot soccer.
In conclusione, in questo lavoro di tesi si é implementato un algoritmo di path planning locale per robot mobili basato su un meccanismo di switching tra potenziali artificiali, sfruttando un diverso tipo di potenziale (bypassante). Si assicura con l'algoritmo implementato che il robot possa sempre giungere in una posa dalla quale é capace di raggiungere il goal e di evitare gli ostacoli (fissi o in movimento). Il metodo é stato testato numericamente in ambiente Matlab e si é rivelato efficace, anche con più ostacoli in movimento.

Listings

List of Figures

References

[1] Luigi D'Alfonso et al. "Obstacles avoidance based on switching potential functions". In: *Journal of Intelligent & Robotic Systems* ().

[2] Dong-Han Kim and Jong-Hwan Kim. "A real-time limit-cycle navigation method for fast mobile robots and its application to robot soccer". In: *Journal of Robotics & Autonomous Systems* ().

[3] Claudio Medio and Giuseppe Oriolo. "Robot Obstacle Avoidance Using Vortex Fields". In: *Advances in Robot Kinematics*. 1991.

[4] Robin Murphy. *Introduction to AI Robotics.*

[5] Søren Riisgaard and Morten Rufus Blas. *SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping.*

[6] Bruno Siciliano et al. *Robotica. Modellistica, pianificazione e controllo.*