

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

Contents

1	Introduzione	3
1.1	Path planning	4
1.2	Formalizzazione del problema	7
1.3	Organizzazione della tesi	7
2	Potenziali artificiali	9
2.1	Metodo classico	9
2.2	Potenziale bypassante	15
3	Panoramica sull'algoritmo di navigazione	18
3.1	Organizzazione del software	19
4	Rappresentazione dell'ambiente	20
5	Modulo di azione	21
5.1	Modello cinematico	21
5.2	Legge di controllo	23
6	Modulo di pianificazione	25
6.1	Stato attrattivo	26
6.2	Switching allo stato bypassante	27
6.3	Stato bypassante	31
7	Modulo di percezione	34
8	Simulazione e Risultati	36
8.1	Simulazione	36
8.2	Risultati	37

Abstract

In questo lavoro di tesi é stato sviluppato, basandosi principalmente sull'articolo [2] del Professore D'Alfonso et al, un modulo software in Matlab che affronta un problema di path planning per robot mobili. L'obiettivo é di guidare un robot mobile all'interno di un ambiente non noto, cercando di fargli raggiungere un punto prestabilito, senza collisioni con eventuali ostacoli. Lo strumento principale utilizzato é quello dei potenziali artificiali, ma con un meccanismo di switching a stati. Ci si é concentrati anche a curare l'architettura del software, sfruttando la programmazione orientata agli oggetti messa a disposizione da Matlab, al fine di rendere il software modulare e riutilizzabile per eventuali espansioni future.

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. Il primo 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 massimizzare il raggiungimento del suo **obiettivo**[5]. Nel caso specifico di questa tesi, l'ambiente del robot autonomo é lo spazio bidimensionale (una superficie), e il suo obiettivo é un punto in questo spazio. Massimizzare 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, detti anche primitive:

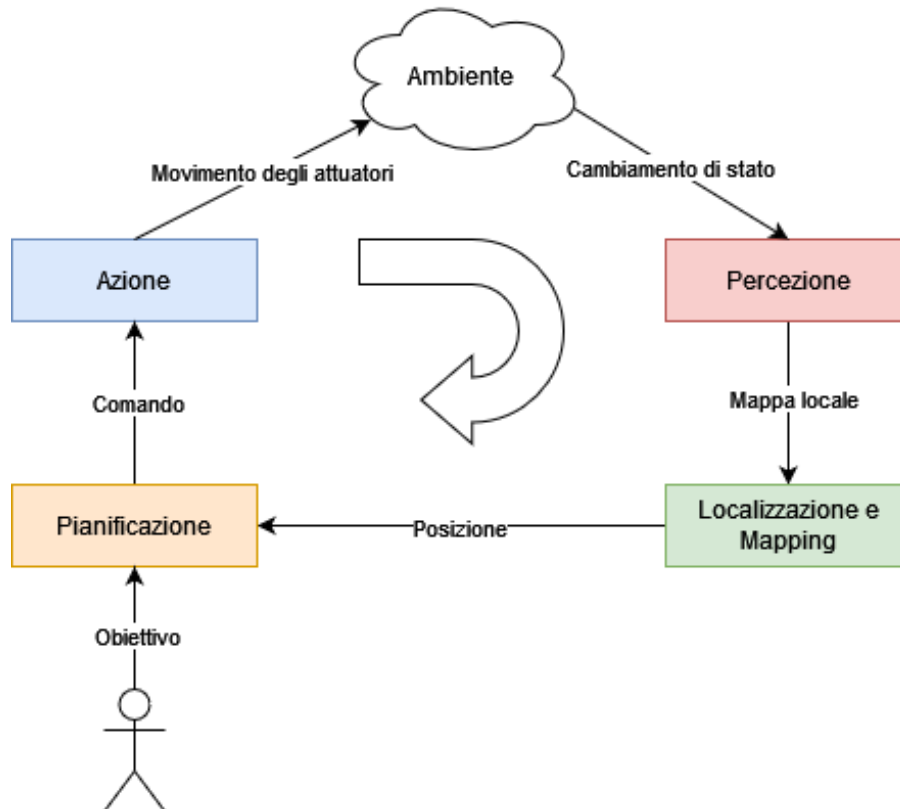
Percezione Prende in input le informazioni derivanti dai sensori, le processa e le restituisce

Localizzazione e Mapping Con le informazioni sensoriali, il robot costruisce una rappresentazione del proprio intorno basandosi sulla propria posizione e ciò che osserva. Il risultato globale, dopo aver fatto varie osservazioni di interni diversi, sarà una mappa dell'ambiente, rispetto alla quale il robot può localizzarsi. (Per scopi esemplificativi, questo modulo verrà tralasciato nell'algoritmo di questa tesi, e 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 alto. Nel caso del motion planning, la direttiva da produrre é il percorso da seguire.

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

Figure 1: Architettura di navigazione



L'architettura utilizzata in questa tesi é la cosiddetta gerarchica: 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 é visivamente sintetizzato quanto appena detto. In questa tesi viene affrontato un problema che rientra nel terzo modulo: il path planning, un problema di grande importanza e argomento di molta 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. All'interno dei 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, quali ostacoli dinamici, per 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 maggior parte dei problemi di robotica autonome 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 che fruttava un approccio probabilistico.
- Potenziali artificiali, che verranno ampiamente trattati nel prossimo capitolo

Di seguito si descriverá brevemente degli esempi legati ai metodi appena elencati.

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

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

dove q é una generica configurazione in C_{free} . La clearance é perciò una funzione che ha come valore in ogni configurazione q la distanza minima tra tutte le distanze 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 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 Dijkstra.

Decomposizione esatta in celle In questa tecnica lo spazio delle configurazioni viene suddiviso in celle, come mostrato in figura 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.

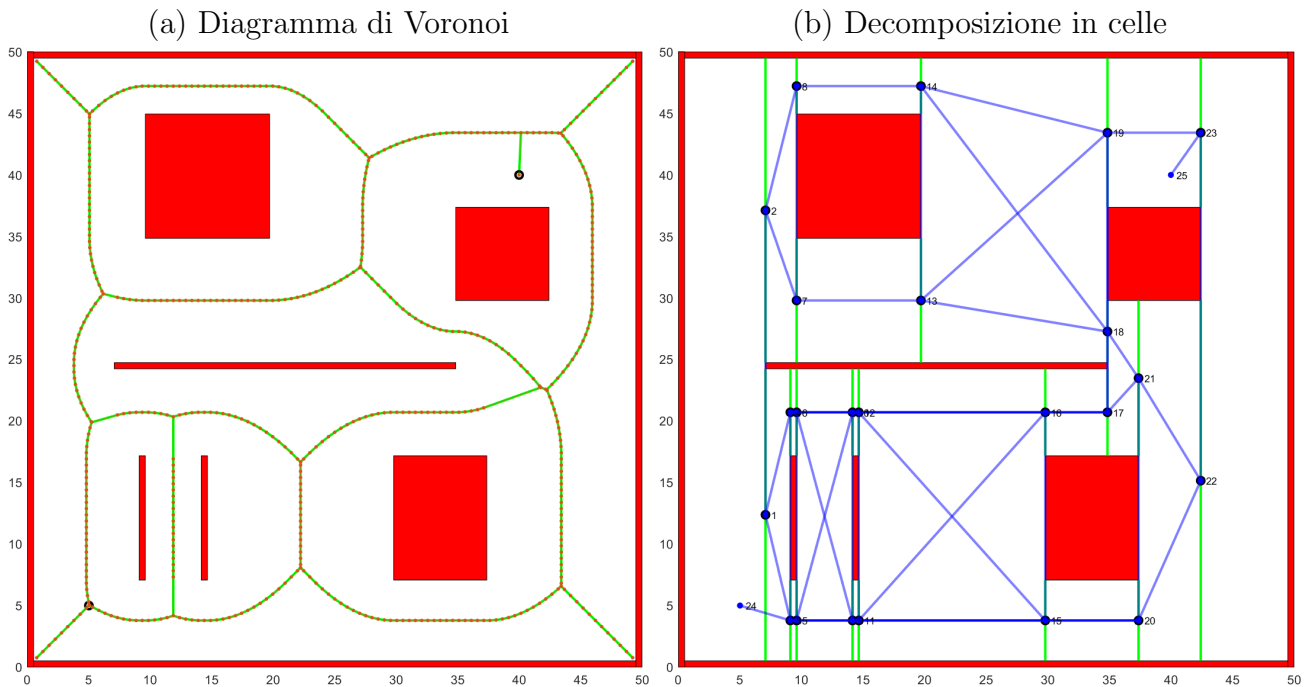


Figure 2: Metodi classici basati su grafo

Sampling based Il terzo metodo si basa su un approccio probabilistico. Brevemente, l'idea é di scegliere a ogni iterazioni una configurazione "di prova" e fare un test di collisione su quest'ultima e si cerca di collegare in basa alla

vicinanza alle configurazioni già appartenenti al grafo che si sta costruendo.

La breve descrizione dei metodi serviva anche a 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 [2], ma risulta inefficiente nel caso del path planning locale. 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, questa tesi 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

Chiamerò $r(t) = [x_r(t), y_r(t), \theta_r(t)]^T$ 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 che, per scopi esemplificativi in fase di prototipazione dell'algoritmo, saranno di forma circolare e con raggio R_i . Quest'ultima ipotesi non provoca una perdita di generalità, visto che 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 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, 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 .

1.3 Organizzazione della tesi

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 usato un approccio top-down nella progettazione e quindi anche nell'esposizione nella tesi: prima si affronta in una modalità "bird-view" come é organizzato nel software, affrontando come i diversi moduli interagiscono tra di loro per ottenere l'obiettivo finale, senza

scendere in dettagli implementativi. Successivamente, sempre in una logica top-down, si affrontano (con riferimento alla figura 1) i singoli moduli : per primo l'ambiente e come viene rappresentato, dopo il modulo di azione, poi pianificazione e poi percezione. Ogni modulo, quindi, viene trattato dando per scontato di avere già accesso al modulo "inferiore" che gli fornisce dati in input. I moduli, infatti, sono stati progettati indipendentemente gli uni dagli altri.

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

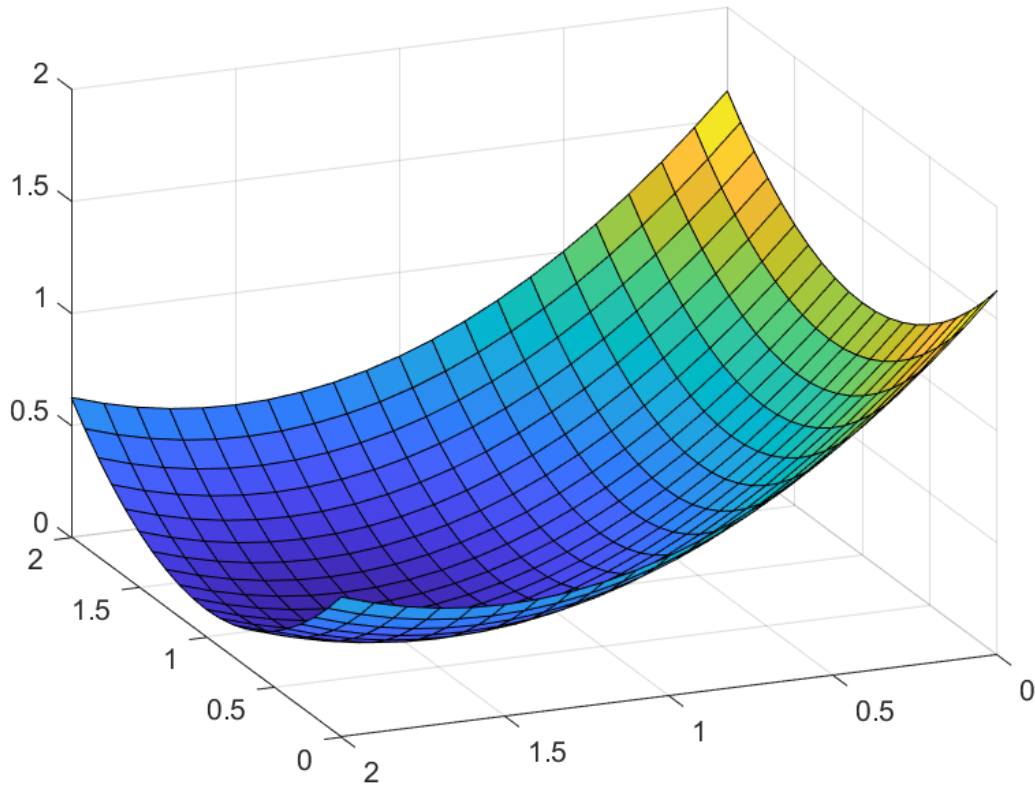
Tradizionalmente, nel path planning basato su potenziali artificiali il robot viene fatto muovere mediante una funzione in due variabili, ovvero un potenziale scalare, che nasce dalla somma di due potenziali: attrattivo e repulsivo. Questi due potenziali sono chiamati artificiali perché generano una 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[4], 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

$$e(r) = G - \begin{bmatrix} x_r(t) \\ y_r(t) \end{bmatrix} \quad (1)$$

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 5, in cui il punto di minimo, ovvero il goal, ha coordinate $[1, 1.5]$.

Figure 3: Potenziale attrattivo

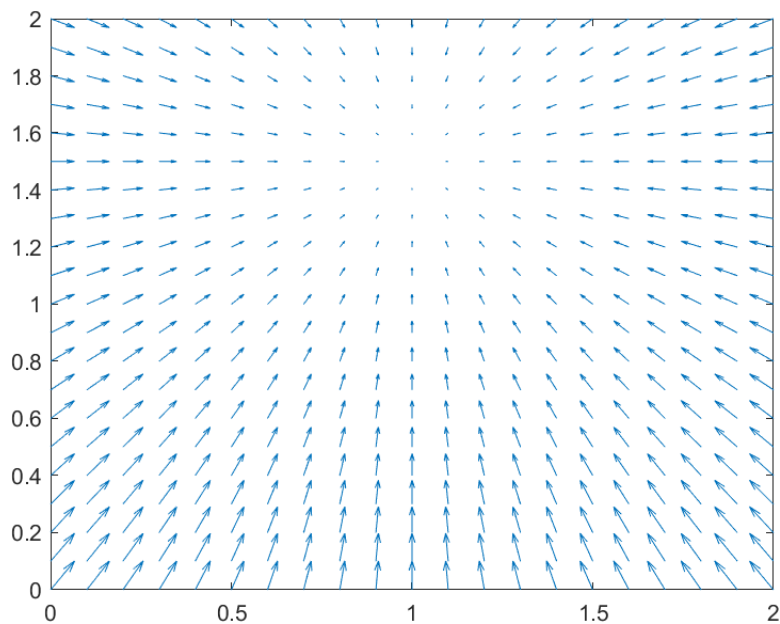


La funzione corrispondente é

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

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

Figure 4: Antigradiente del potenziale attrattivo



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

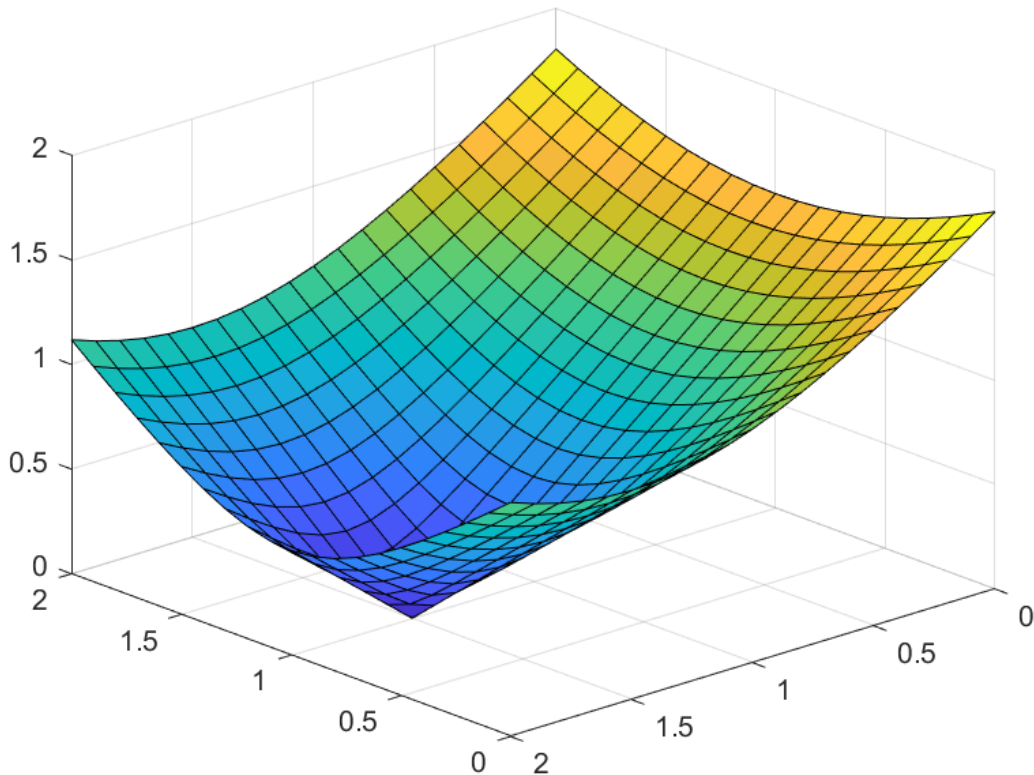
$$-\nabla U_{a1}(r, G) = k_1 e(r) \quad (3)$$

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 un errore lineare tra goal e posizione del robot. Oltre al potenziale parabolicoide si può optare anche per un potenziale conico

$$U_{a2}(r, G) = k_2 ||e(r)|| \quad (4)$$

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

Figure 5: Potenziale attrattivo



L'antigradiente del potenziale conico ha la seguente espressione

$$-\nabla U_a(r, G) = k_2 \frac{e(r)}{||e(r)||} \quad (5)$$

L'idea é quella di stabilire una soglia per la funzione errore oltre (e su) la quale, avvicinandosi al goal, il potenziale attrattivo sarà di forma paraboloidale. Dunque il potenziale attrattivo finale sarà del tipo

$$U_a(r) = \begin{cases} \frac{1}{2} \cdot k_1 ||e(r)||^2 & ||e(r)|| \leq \rho \\ k_2 ||e(r)|| & ||e(r)|| > \rho \end{cases} \quad (6)$$

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

$$k_1 \cdot \rho = k_2 \quad (7)$$

Il potenziale repulsivo ha una forma duale a quello attrattivo, come mostrato in figura 6a. Esso impone che entro una 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.

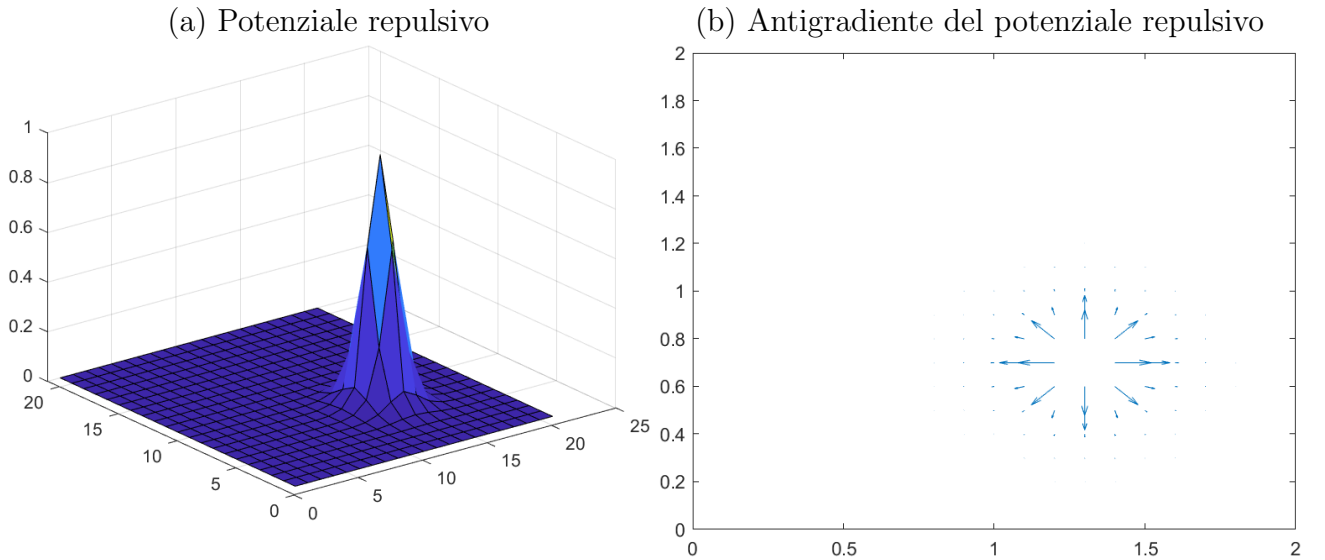


Figure 6: Potenziale repulsivo

Al potenziale repulsivo corrisponde la seguente funzione a tratti

$$U_r(r(t), O_j(t)) = \begin{cases} \frac{1}{2} \left(\frac{1}{d(r(t), O_j(t))} - \frac{1}{\eta} \right) & d(r(t), O_j(t)) \leq \eta \\ 0 & \text{altrimenti} \end{cases} \quad (8)$$

dove

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

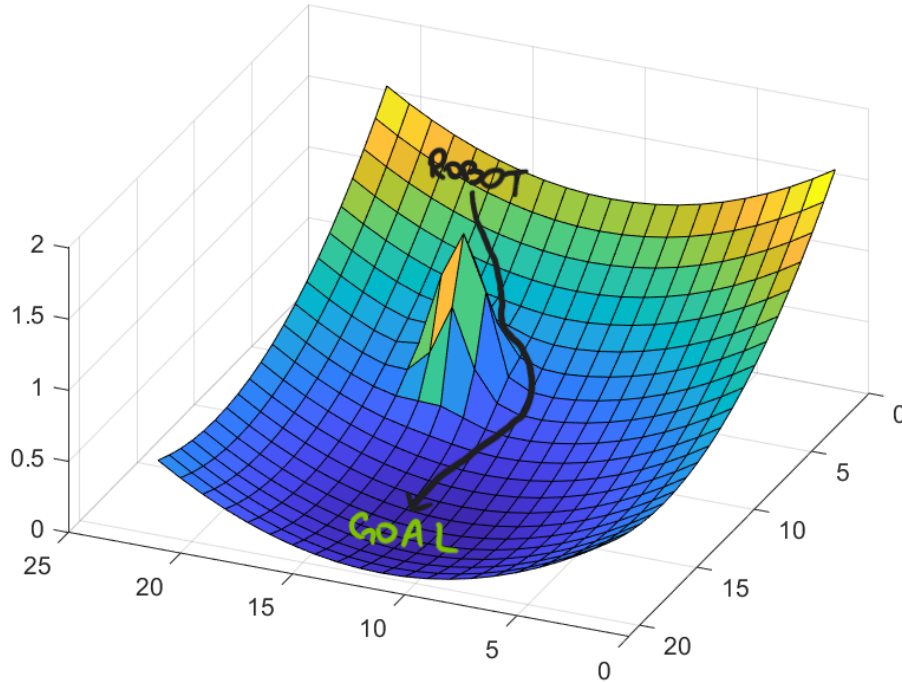
Dunque, il compito della forza generata é quello di spingere via il robot dalla posizione dell'ostacolo tanto più che il robot si avvicina entro la soglia η a quest'ultimo. In figura 6b si vedono le linee di campo dell'antigradiente che puntano radialmente verso l'esterno rispetto alla posizione dell'ostacolo, qui con coordinate $[1.5, 1]$.

Una volta calcolato il potenziale repulsivo per ogni singolo ostacolo, si ottiene il potenziale totale:

$$U(r(t)) = U_a(r(t), G) + \sum_{j=1}^n U_r(r(t), O_j(t)) \quad (9)$$

In figura 7 si vede un possibile percorso del robot per arrivare dal punto di partenza al goal. Dalla 9 si vede che l'antigradiente é una funzione di $r(t)$: vuol dire che in ogni sua configurazione, siccome l'antigradiente si suppone già calcolato, il robot può ottenere informazioni su quest'ultimo 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 $\theta = \tan \left(\frac{y_{O,j}(t) - y_r(t)}{x_{O,j}(t) - x_r(t)} \right)$ (ovvero 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, il modulo di mapping e path planning sono praticamente assenti. Gli attuatori del robot, che sono collegati ai sensori tramite una funzione di trasferimento [3], ricevono direttamente il comando derivante dal calcolo precedente del potenziale artificiale, senza alcun tipo di pianificazione algoritmica.

Figure 7: 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 è "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 trattata nel capitolo riguardante l'algoritmo di navigazione.

Altro problema da considerare è 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, è 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 (i gradienti 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 8 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.

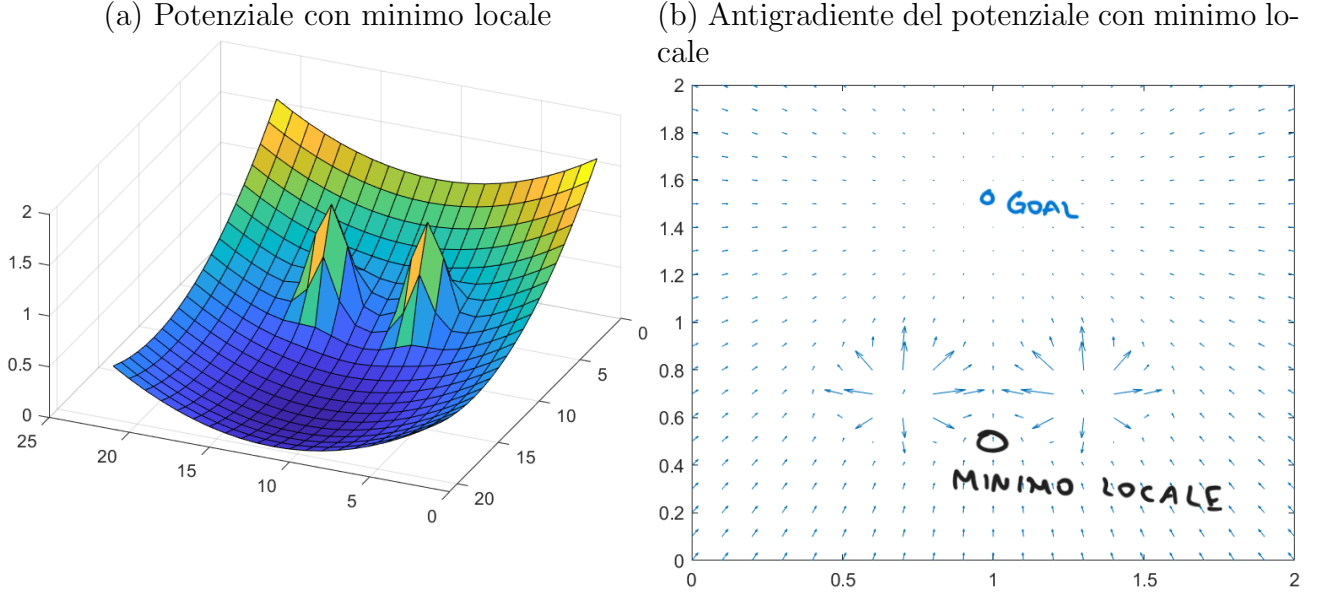


Figure 8: Minimo locale

2.2 Potenziale bypassante

Al fine di evitare le non-idealità dovute all'utilizzo dei potenziali sommati, l'idea implementata in questa tesi é quella di sfruttare al posto di quello repulsivo un altro tipo di potenziale che chiameremo 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 dalla posizione, ad esempio un elicoide:

$$\begin{cases} x = x_0 + r \cos(\xi) \\ y = y_0 + r \sin(\xi) \\ z = c\xi \end{cases} \quad (10)$$

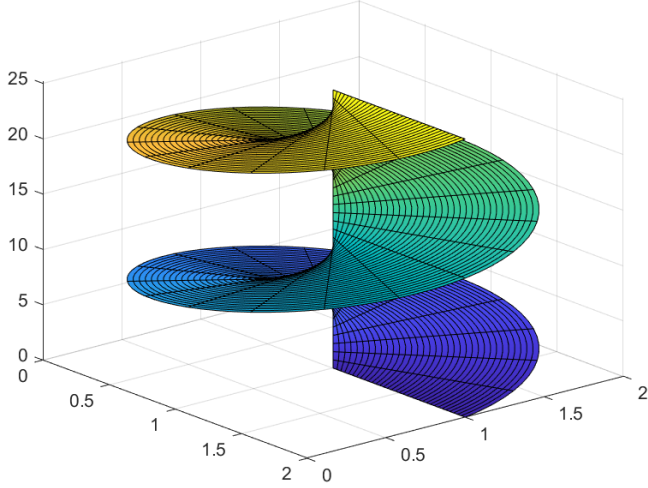
Applicando nella terza equazione la tangente da entrambe le parti, otteniamo $\tan(\xi) = \tan(\frac{z}{c})$ che, confrontando con le prime due equazioni, diventa

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

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, ride-nominato bypassante, in senso orario rispetto all'ostacolo e centrato in esso:

$$\Gamma(x, y, x_0, y_0) = c \tan^{-1} \left(\frac{y - y_0}{x - x_0} \right) \quad (11)$$

(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

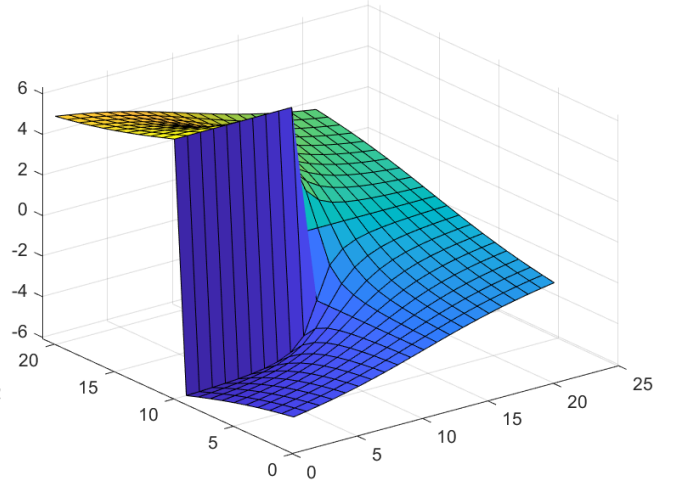


Figure 9: Potenziale bypassante

Da notare che il potenziale in figura 9b ha una discontinuità per $x = x_0$, per cui la funzione 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 11 può essere visto come una funzione della posizione del robot e dell'ostacolo da aggirare

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

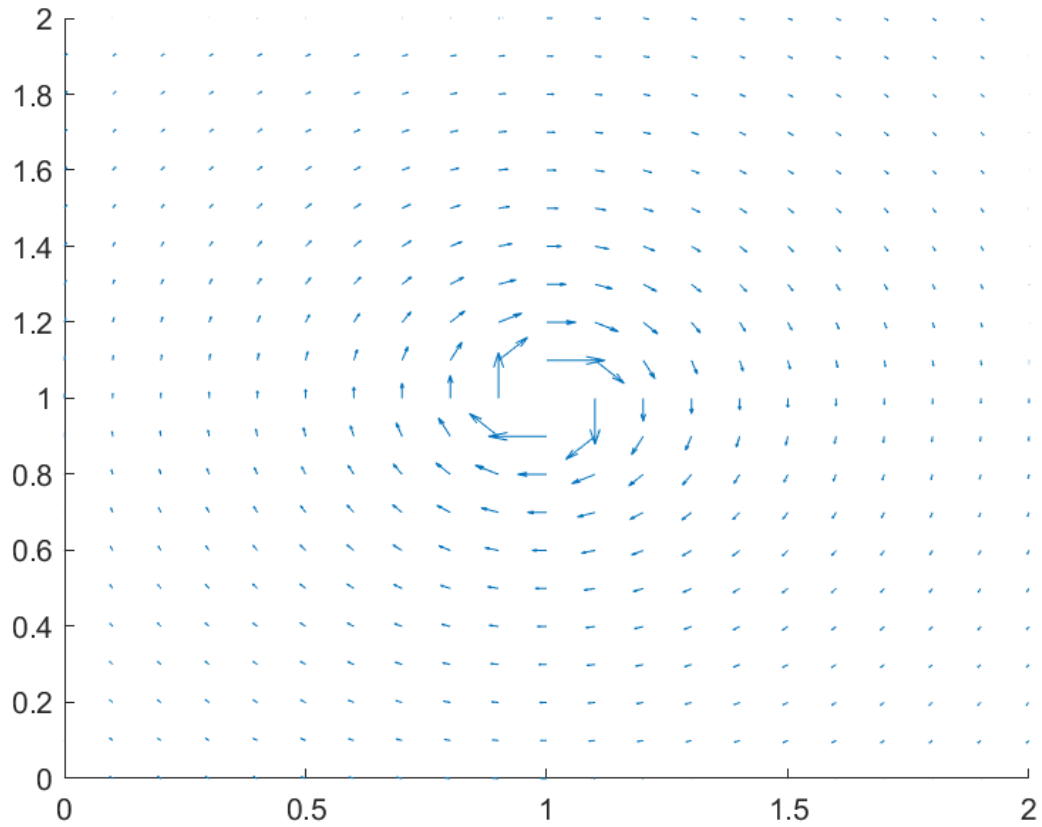
Dunque, la forza generata dal potenziale bypassante é esprimibile come il suo antigradiente

$$-\nabla U_b(r(t), O_j(t)) = \begin{bmatrix} \frac{c(y_r(t) - y_{O,j}(t))}{(x_r(t) - x_{O,j}(t))^2 + (y_r(t) - y_{O,j}(t))^2} \\ \frac{c(x_{O,j}(t) - x_r(t))}{(x_r(t) - x_{O,j}(t))^2 + (y_r(t) - y_{O,j}(t))^2} \end{bmatrix} \quad (12)$$

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 10: Antigradiente del potenziale bypassante

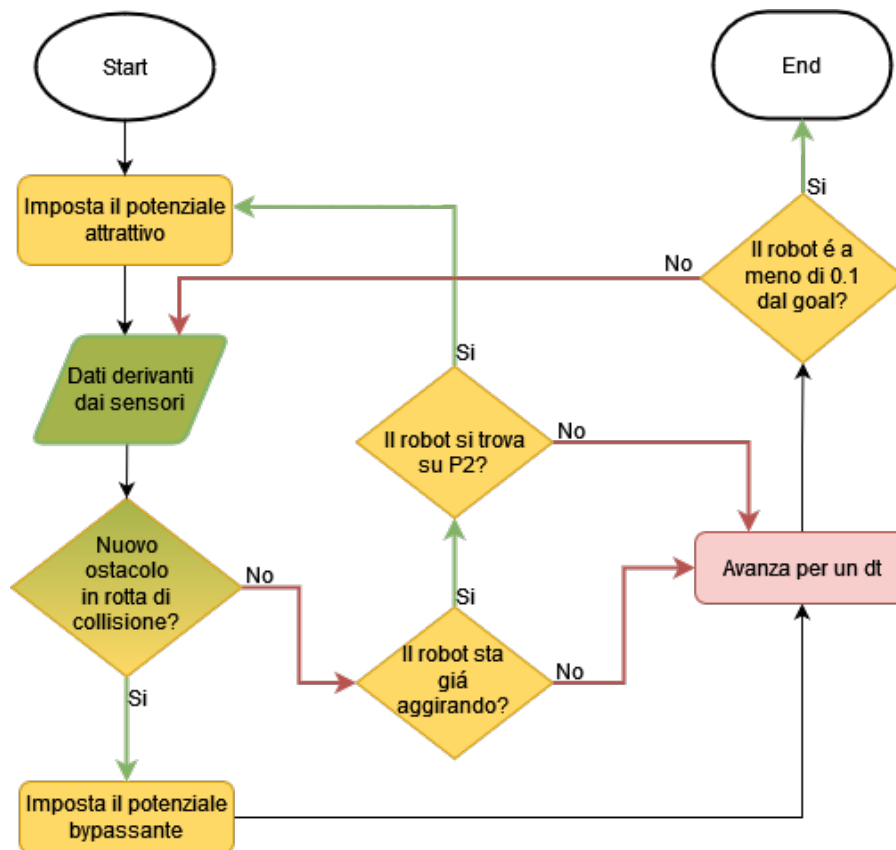


Questo tipo di potenziale, chiaramente, é intrinsecamente adatto ad ostacoli di forma circolare, motivo per cui nel lavoro di tesi si é considerato soltanto questo tipo di ostacoli.

3 Panoramica sull'algoritmo di navigazione

La strategia é semplice: nella posizione iniziale, il robot sonda l'ambiente attorno a sé: se non vi sono presenti ostacoli a impedirne l'avanzamento verso il goal, la traiettoria da seguire é quella imposta dal potenziale attrattivo; altrimenti 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 11 si vede, orientativamente, come funziona l'algoritmo.

Figure 11: Flowchart Diagram

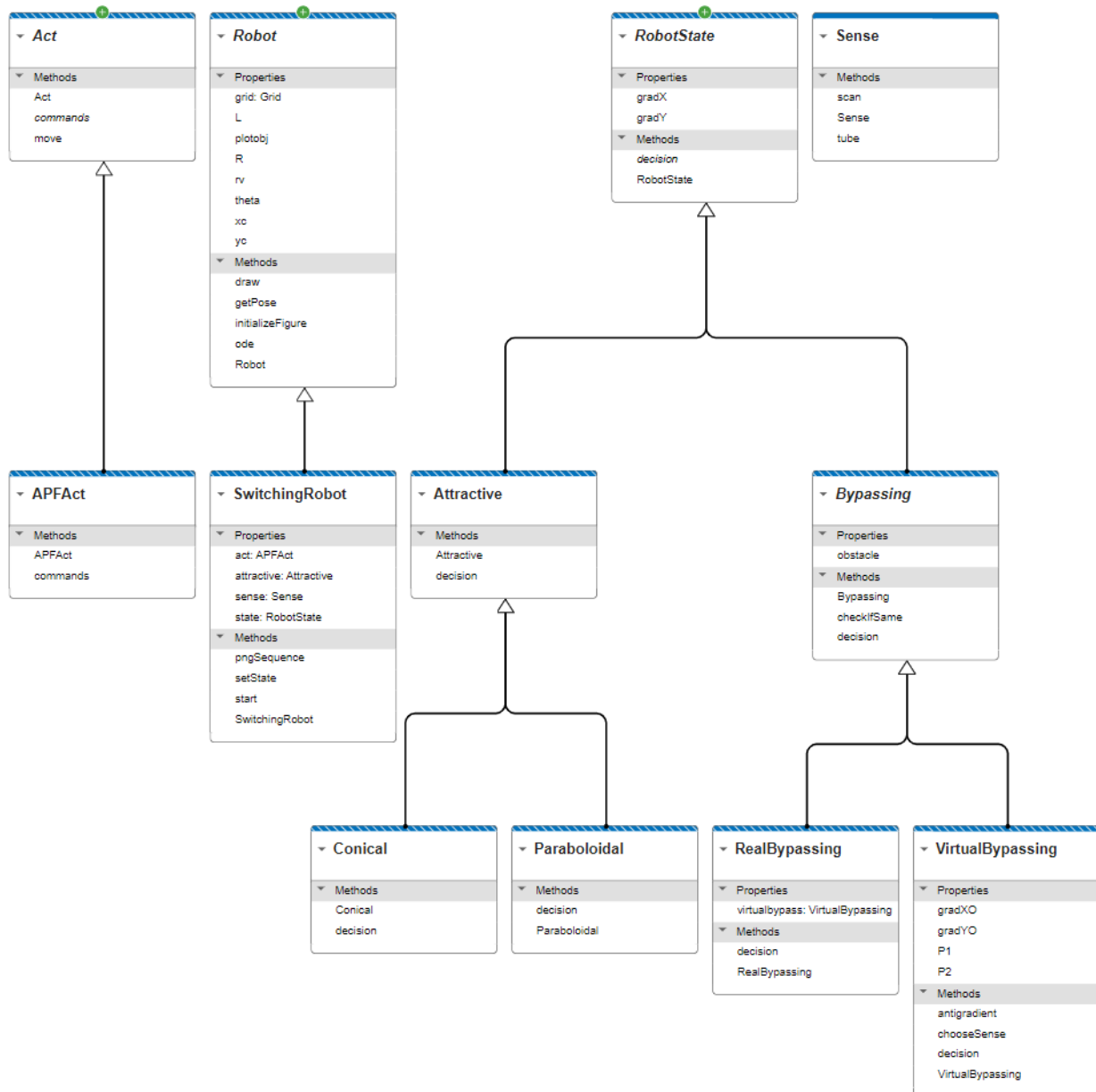


Le pseudo-istruzioni in giallo sono quelle relative al modulo di pianificazione e le verdi al modulo di visione. Naturalmente verranno approfonditi nelle rispettive sezioni. Una considerazione sull'istruzione rossa, relativa al modulo di azione: avanzare per un dt é indice del fatto che é stato usato nel progetto un approccio completamente discretizzato (questo aspetto viene approfondito nel paragrafo successivo).

3.1 Organizzazione del software

In figura ?? é riportato il class diagram del software di navigazione. La classe Act e Sense rappresentano rispettivamente il modulo di azione e pianificazione. Il modulo di pianificazione é costituito dalla gerarchia discendente dalla classe astratta 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.

Figure 12: Class diagram



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.

```

1 function obj = start(obj)
2     %Simulation data
3     e = norm([obj.xc,obj.yc]-obj.grid.goal);
4     tspan = 0.05; tsim = 0; samples = 0;
5     %Starting simulation
6     while(e > 0.1 && tsim < 20)
7         %Sensed obstacle (empty array if nothing was sensed)
8         dObstacle = obj.sense.scan(obj);
9         %New directive
10        obj.state.decision(obj,dObstacle);
11        %Giving the command to the actuators
12        [obj.xc,obj.yc,obj.theta] = obj.act.move(obj,tspan);
13        %Refreshing the error
14        e = norm([obj.xc,obj.yc]-obj.grid.goal);
15        tsim = tsim + tspan; pause(0);
16    end
17 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. Altra variabile importante é $tspan$ 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 viene richiamato sul campo state, a cui si può assegnare l'intera gerarchia erede di RobotState, per cui, a seconda dell'oggetto assegnato a state, il risultato restituito dal metodo decision sarà diverso (é qui che avviene il meccanismo di switching tra potenziali artificiali che verrà approfondito nella sezione relativa al modulo di pianificazione). 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 approfondirà questo metodo in merito alla sezione su simulazione e risultati). Questo ciclo si ripeterà naturalmente fino a che il robot non sarà arrivato nel punto di goal.

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 in questa tesi 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 $[x, y]$, bisogna convertire queste in indici della meshgrid. Ciò viene fatto dal metodo `coord2index` della classe `Grid`.

```

1 function index = coord2index(obj, point)
2     index = zeros(1,2);
3     % x coordinate goes into the column index
4     floorx = floor(point(1));
5     index(2) = (floorx + obj.dx*floor((point(1) - floorx)/
6         obj.dx))*obj.resolution + 1;
7     % y coordinate goes into the row index
8     floory = floor(point(2));
9     index(1) = (floory + obj.dy*floor((point(2) - floory)/
10         obj.dy))*obj.resolution + 1;
11     index = floor(index);
12 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 é senza 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

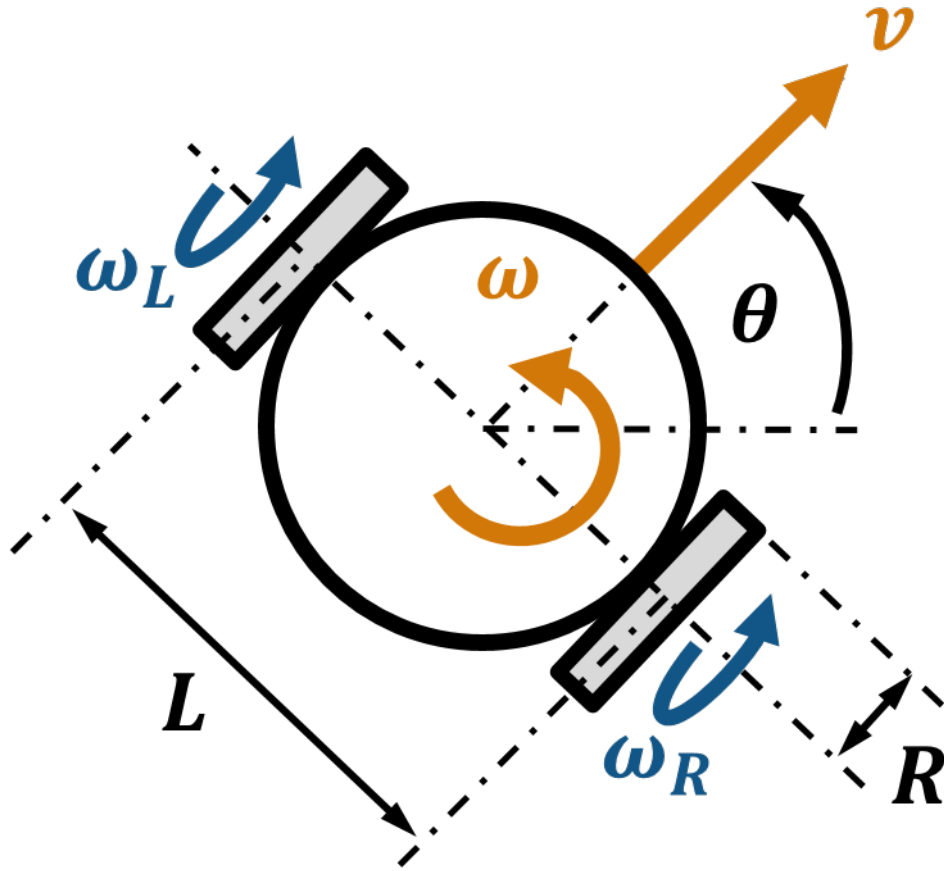
Il modulo responsabile per il movimento del robot riceve in input un comando dal modulo di pianificazione, ovvero l'antigradiente da seguire, e restituisce in output una "traduzione" del comando in una velocità da imporre agli attuatori (tramite una legge di controllo), tenendo conto dello specifico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

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 velocità e non sulla posizione, ovvero

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

Figure 13: Modello Differential-Drive [1]



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

$$\begin{cases} \dot{x}(t) = v(t) \cos(\theta(t)) \\ \dot{y}(t) = v(t) \sin(\theta(t)) \\ \dot{\theta}(t) = \omega(t) \end{cases} \quad (13)$$

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 ??, 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 scrivere 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

$$\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ \frac{R}{L} & -\frac{R}{L} \end{bmatrix} \begin{bmatrix} \omega_R(t) \\ \omega_L(t) \end{bmatrix} \quad (14)$$

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

$$\begin{bmatrix} \omega_R(t) \\ \omega_L(t) \end{bmatrix} = \begin{bmatrix} \frac{R}{2} & \frac{R}{2} \\ \frac{R}{L} & -\frac{R}{L} \end{bmatrix}^{-1} \begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} \quad (15)$$

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 13 diventerá

$$\begin{cases} \dot{x}(t) = \frac{R}{2} (\omega_R(t) + \omega_L(t)) \cos(\theta(t)) \\ \dot{y}(t) = \frac{R}{2} (\omega_R(t) + \omega_L(t)) \sin(\theta(t)) \\ \dot{\theta}(t) = \frac{R}{L} (\omega_R(t) - \omega_L(t)) \end{cases} \quad (16)$$

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

```

1 function [Xdot,wRwL] = ode(obj,vr,wr)
2     K = [obj.R/2 obj.R/2 ; obj.R/obj.L -obj.R/obj.L];
3     wRwL = K \ [vr ; wr];
4     Xdot =
5         ([cos(obj.theta) 0 ; sin(obj.theta) 0 ; 0 1] * K * wRwL);
6 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

$$v(t) = M_v \cos(\theta_{\nabla}(t) - \theta_r(t)) \quad (17)$$

$$\omega(t) = K_\omega(\theta_\nabla(t) - \theta_r(t)) \quad (18)$$

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

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

Una volta calcolate le velocità di riferimento $v(t)$ e $\omega(t)$, basta applicare la relazione 15 per avere le velocità di riferimento relative alle due ruote del differential drive, ottenendo così i comandi da impartire agli attuatori del robot. La legge di controllo, specifica per questo metodo di navigazione, è 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` è 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

```

1 function [vr,wr] = commands(~,rx,ry,rtheta,tspan,robot)
2     grid = robot.grid;
3     gradX = robot.state.gradX; gradY = robot.state.gradY;
4     i = grid.coord2index([rx,ry]);
5     thetaN = atan2(gradY(i(1),i(2)),gradX(i(1),i(2)));
6     thetaDiff = atan2(sin(thetaN-rtheta),cos(thetaN-rtheta));
7     vgrad = [gradX(i(1),i(2)) gradY(i(1),i(2))];
8     Mv = norm(vgrad);
9     vr = (Mv * cos(thetaDiff));
10
11     rx1 = rx + vgrad(1)*tspan;
12     ry1 = ry + vgrad(2)*tspan;
13     j = grid.coord2index([rx1,ry1]);
14     thetaN1 = atan2(gradY(j(1),j(2)),gradX(j(1),j(2)));
15     thetaDdiff = atan2(sin(thetaN1-thetaN),cos(thetaN1-thetaN));
16     thetaDdot = thetaDdiff/tspan;
17
18     Kc = 10; eps = 0.001; v = 1;
19     Kw = (thetaDdot + Kc * abs(thetaDiff)^v * ...
20         sign(thetaDiff))/(thetaDiff+eps);
21     wr = (abs(thetaDiff) >= eps) * Kw * (thetaDiff);
22 end

```


6 Modulo di pianificazione

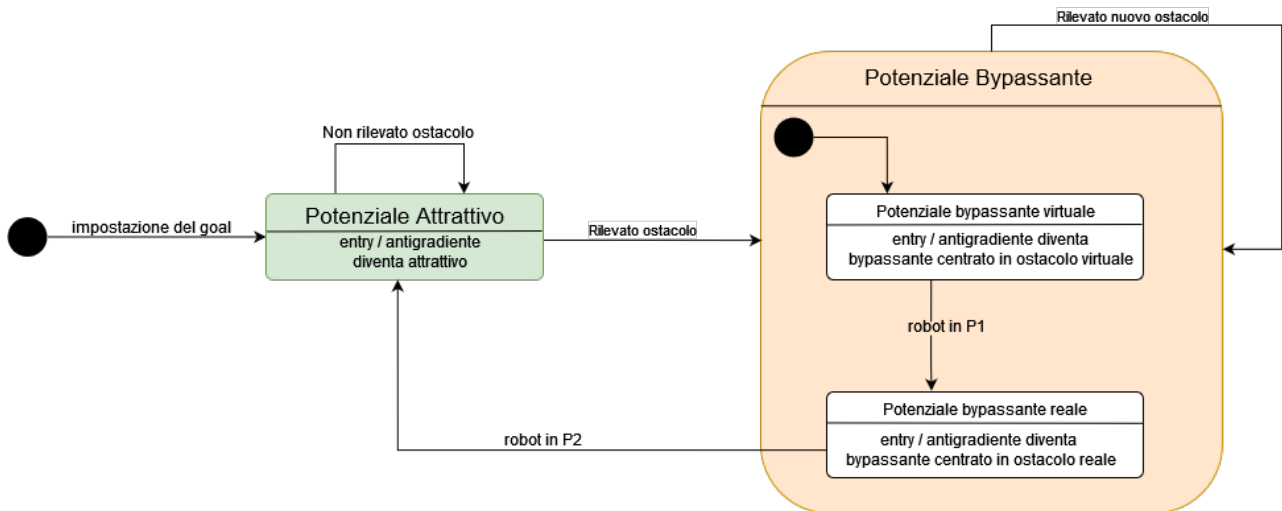
Il modulo di pianificazione riceve ad ogni iterazione in input i dati processati dai sensori e in base a ciò decide a quale potenziale il robot deve fare riferimento, inoltrando successivamente la decisione al modulo di azione.

Nota implementativa

Il modulo di pianificazione, argomento principe della tesi, è 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 è 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 è sicuramente uno statechart diagram, come riportato in figura 14. Il robot può trovarsi, in una logica aut aut, nello stato corrispondente al potenziale attrattivo oppure in quello relativo al potenziale bypassante.

Figure 14: Statechart Diagram



La strategia più naturale, sul piano dell'ingegneria del software, è 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 listing ??), il quale prende in input il robot e l'ostacolo rilevato. RobotState contiene anche l'antigradiente come variabile d'istanza: com'è naturale aspet-

tarsi, ogni stato (attrattivo, bypassante ecc) possiede un specifico antigradiente.

```
1 classdef (Abstract) RobotState < handle
2     properties
3         gradX; gradY;
4     end
5
6     methods (Abstract)
7         obj = decision(obj,robot,dObstacle);
8     end
9 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).

```
1 properties
2     act APFAct = APFAct();
3     sense Sense = Sense();
4     state RobotState = DefaultRobotState();
5     attractive Attractive;
6 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.

```
1 classdef Attractive < RobotState
2     methods
3         function obj = decision(obj,robot,dObstacle)
4             if ~isempty(dObstacle)
5                 bypassing = VirtualBypassing(robot,dObstacle);
6                 bypassing.obstacle = [dObstacle.xc dObstacle.yc];
7                 robot.setState(bypassing);
8             end
9         end
10    end
11 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.

```

1 function obj = SwitchingRobot(R,L,grid)
2     obj@Robot(R,L,grid);
3     obj.state = Conical(grid);
4     obj.attractive = obj.state;
5 end

```

Nel costruttore di Conical é implementata l'equazione 6 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.

```

1 function obj = Conical(grid)
2     di = sqrt((grid.goal(1)-grid.X).^2 +
3             (grid.goal(2)-grid.Y).^2);
4     obj.gradX = (grid.goal(1)-grid.X)./di;
5     obj.gradY = (grid.goal(2)-grid.Y)./di;
6 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 paraboloidale. La classe Paraboloidal si limiterà ad eseguire il metodo decision così come é stato ridefinito in Attractive.

```

1 function obj = decision(obj,robot,dObstacle)
2     decision@Attractive(obj,robot,dObstacle);
3     rx = robot.xc; ry = robot.yc; grid = robot.grid;
4     if norm([rx,ry]-grid.goal) < 1
5         newattractive = Paraboloidal(grid);
6         robot.attractive = newattractive;
7         robot.setState(newattractive);
8     end
9 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 14. L'obiettivo ultimo del metodo, perciò, 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.

```

1 function sense = chooseSense(~, obstacle, pose)
2     xr = pose(1); yr = pose(2); thetar = pose(3);
3     phi = atan2(obstacle.yc - yr, obstacle.xc - xr);
4     alphav = atan2(obstacle.v(2), obstacle.v(1));
5     vphi = pi/2 - phi + alphav
6     vphi = atan2(sin(vphi), cos(vphi));
7     if cos(vphi) > 0
8         sense = "clock";
9     else
10        sense = "counterclock";
11    end
12 end

```

Nel listing é riportato il metodo chooseSense - si trova nella classe VirtualBypassing - che riceve in input un ostacolo e la posa del robot e restituisce in output una stringa indicante il verso di percorrenza di quell'ostacolo. Si estrae in phi l'angolo tra robot e ostacolo rispetto all'orizzontale e in alphav l'angolo corrispondente alla direzione con cui viaggia al momento l'ostacolo. Successivamente, tramite l'angolo vphi si controlla se il vettore velocità dell'ostacolo punta verso il semipiano sinistro o destro del sistema di riferimento fittizio in figura: nel primo caso il verso di percorrenza sarà quello antiorario, orario nel secondo caso.

Aggiungere figura su scelta verso

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 velocità 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 10 - 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

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

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.

Figura circonferenze

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

1. Posizione dell'ostacolo virtuale
2. Punto in cui switchare dall'ostacolo virtuale al reale
3. 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

$$\begin{cases} x_\Omega = -m \cdot y_\Omega \\ (x_\Omega - x_o)^2 + (y_\Omega - y_o)^2 = (\sqrt{x_\Omega^2 + y_\Omega^2} + h)^2 \end{cases} \quad (19)$$

Risolvendo questo sistema per y_Ω si ottengono due possibili soluzioni

$$\begin{cases} y_{\Omega,1} = \frac{(-h^2 + x_o^2 + y_o^2)(mx_o - y_o + h\sqrt{m^2 + 1})}{2h^2m^2 + 2h^2 - 2m^2x_o^2 + 4mx_oy_o - 2y_o^2} \\ y_{\Omega,2} = \frac{-(-h^2 + x_o^2 + y_o^2)(mx_o - y_o + h\sqrt{m^2 + 1})}{2h^2m^2 + 2h^2 - 2m^2x_o^2 + 4mx_oy_o - 2y_o^2} \end{cases} \quad (20)$$

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 due "ingredienti", manca il terzo, 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 $-\nabla U_b(P1, \Omega(t)) = -\nabla U_b(P1, O(t))$ e anche $-\nabla U_b(P2, O(t)) = -\nabla U_a(P2)$. 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 12

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

dove $[x, y]$ é un punto incognito e $[x_o, y_o]$ la posizione dell'ostacolo. Da finire

6.3 Stato bypassante

Una volta che é avvenuto lo switch tra potenziale attrattivo e bypassante, bisogna definire, ricollegandosi allo statechart in figura 14, 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.

```

1 classdef (Abstract) Bypassing < RobotState
2
3     properties
4         obstacle; %ostacolo che sta per essere bypassato
5     end
6
7     methods
8         function obj = decision(obj,robot,dObstacle)
9             %Controllo se l'ostacolo rilevato e' diverso
10            %da quello che sto bypassando
11            dO = obj.checkIfSame(dObstacle);
12            robot.attractive.decision(robot,dO);
13        end
14
15        function dO = checkIfSame(obj,dObstacle)
16            %%Ostacolo non rilevato
17            if isempty(dObstacle)
18                obj.obstacle = []; dO = []; return;
19            end
20            %%Ostacolo rilevato
21            %Ho ancora in vista l'ostacolo che sto bypassando
22            if ~isempty(obj.obstacle)
23                %Controllo se l'ostacolo rilevato e' lo stesso
24                distance =
25                    sqrt((obj.obstacle(1) - dObstacle.xc)^2 +
26                        (obj.obstacle(2) - dObstacle.yc)^2);
27                if abs(distance - norm(0.05*(dObstacle.v))) < 0.01
28                    dO = []; %SI
29                else
30                    dO = dObstacle; %NO
31                end
32            end
33            %Non ho piu in vista l'ostacolo che sto bypassando
34            if isempty(obj.obstacle)
35                dO = dObstacle;
36            end
37            %Aggiorno l'ostacolo corrente con quello rilevato
38            obj.obstacle = [dObstacle.xc dObstacle.yc];
39        end
40    end
41 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 precedentemente, l'ordine dei potenziali bypassanti é sempre virtuale e poi reale.


```

1 function obj = decision(obj,robot,dObstacle)
2     decision@Bypassing(obj,robot,dObstacle);
3     if norm([robot.xc robot.yc] - obj.P1) < 0.1
4         robot.setState(RealBypassing(obj));
5     end
6 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.

```

1 function obj = decision(obj,robot,dObstacle)
2     decision@Bypassing(obj,robot,dObstacle);
3     if norm([robot.xc robot.yc] - obj.virtualbypass.P2) < 0.1
4         robot.setState(robot.attractive);
5     end
6 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 r_v dal robot e dentro un tubo direzionato dal robot al goal di larghezza r_m . Figura tubo

```
1 classdef Sense
2
3     methods
4         %% Method that looks in a radius  $r_v$ 
5         %%and a tube  $T(t)$  if there are any obstacles
6         function dObstacle = scan(obj,robot)
7             rx = robot.xc; ry = robot.yc; rv = robot.rv;
8             n = length(robot.grid.obstacles);
9             distances = zeros(1,n) + inf;
10            dObstacles = cell(1,n);
11            for j = 1 : n
12                o = robot.grid.obstacles(j);
13                distance = norm([rx ry] - [o.xc o.yc]);
14                if distance <= rv && obj.tube(robot,o)
15                    dObstacles{j} = o;
16                    distances(j) = distance;
17                end
18            end
19            %%Ostacolo tra quelli rilevati a distanza minima
20            [~,pos] = min(distances);
21            dObstacle = dObstacles(pos); dObstacle = dObstacle{1};
22        end
23    end
24
25    methods(Access=private)
26        %% Method that builds the tube  $T(t)$ 
27        function result = tube(~,robot,obstacle)
28            pose = robot.getPose(); rx = pose(1); ry = pose(2);
29            grid = robot.grid;
```

```

30     G = grid.goal;
31
32     rm = robot.rv + 1;
33     angle = atan2(G(2)-ry,G(1)-rx);
34     deltaX = rm/2*sin(angle); deltaY = rm/2*cos(angle);
35     x1 = rx + deltaX; y1 = ry - deltaY;
36     x4 = rx - deltaX; y4 = ry + deltaY;
37     x2 = G(1) + deltaX; y2 = G(2) - deltaY;
38     x3 = G(1) - deltaX; y3 = G(2) + deltaY;
39
40     m14 = (y4-y1)/(x4-x1); q14 = m14*x1 - y1;
41     if abs(m14) > exp(10)
42         v14 = (obstacle.xc - x1);
43     else
44         v14 = (obstacle.yc - m14*obstacle.xc + q14);
45     end
46
47     m12 = (y2-y1)/(x2-x1); q12 = m12*x1 - y1;
48     if abs(m12) > exp(10)
49         v12 = (x1 - obstacle.xc);
50     else
51         v12 = (obstacle.yc - m12*obstacle.xc + q12);
52     end
53
54     m34 = (y4-y3)/(x4-x3); q34 = m34*x3 - y3;
55     if abs(m34) > exp(10)
56         v34 = (x4 - obstacle.xc);
57     else
58         v34 = (obstacle.yc - m34*obstacle.xc + q34);
59     end
60
61     if (angle >= 0 && angle <= pi/2) %first quadrant
62         result1 = v14 >= 0;
63         result2 = v12 >= 0; result3 = v34 <= 0;
64     elseif (angle > pi/2 && angle < pi) %second quadrant
65         result1 = v14 >= 0;
66         result2 = v12 <= 0; result3 = v34 >= 0;
67     elseif (angle >= -pi && angle <= -pi/2 || angle == pi) %third quadrant
68         result1 = v14 <= 0;
69         result2 = v12 <= 0; result3 = v34 >= 0;
70     else %fourth quadrant
71         result1 = v14 <= 0;
72         result2 = v12 >= 0; result3 = v34 <= 0;
73     end
74     result = result1 && result2 && result3;
75 end
76 end
77 end

```

8 Simulazione e Risultati

8.1 Simulazione

Non avendo la possibilità di testare l'algoritmo su robot reali, l'unica opzione è quella di simularlo in Matlab. A tal fine, serve innanzitutto il metodo `move` (già visto nel precedente listing) 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.

```
1 function [xc,yc,theta] = move(obj,robot,tspan)
2     rx = robot.xc;
3     ry = robot.yc;
4     rtheta = robot.theta;
5     [vr,wr] = obj.commands(rx,ry,rtheta,tspan,robot);
6     Xdot = robot.ode(vr,wr);
7
8     rx2 = rx + tspan/2*Xdot(1);
9     ry2 = ry + tspan/2*Xdot(2);
10    rtheta2 = rtheta + tspan/2*Xdot(3);
11    [vr,wr] = obj.commands(rx2,ry2,rtheta2,tspan,robot);
12    Xdot = robot.ode(vr,wr);
13
14    xc = rx + tspan*Xdot(1);
15    yc = ry + tspan*Xdot(2);
16    theta = rtheta + tspan*Xdot(3);
17 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 è integrare numericamente attraverso la derivata numerica `Xdot`, come si vede da riga 14 a 16. La posizione del robot tra un istante di campionamento è pari alla posizione attuale più il prodotto tra la derivata (istantanea) della posizione e l'intervallo di campionamento. Nello specifico, in questo metodo si è 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.

```
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 dapprima generata 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.

8.2 Risultati

List of Figures

1	Architettura di navigazione	4
2	Metodi classici basati su grafo	6
3	Potenziale attrattivo	10
4	Antigradiente del potenziale attrattivo	10
5	Potenziale attrattivo	11
6	Potenziale repulsivo	12
7	Potenziale totale	14
8	Minimo locale	15
9	Potenziale bypassante	16
10	Antigradiente del potenziale bypassante	17
11	Flowchart Diagram	18
12	Class diagram	19
13	Modello Differential-Drive [1]	22
14	Statechart Diagram	25

References

- [1] MathWorks Student Competitions Team (2022). Mobile Robotics Simulation Toolbox (<https://github.com/mathworks-robotics/mobile-robotics-simulation-toolbox>).
- [2] Luigi D’Alfonso et al. “Obstacles avoidance based on switching potential functions”. In: *Journal of Intelligent & Robotic Systems* ().
- [3] 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* ().
- [4] Claudio Medio and Giuseppe Oriolo. “Robot Obstacle Avoidance Using Vortex Fields”. In: *Advances in Robot Kinematics*. 1991.
- [5] Robin Murphy. *Introduction to AI Robotics*.
- [6] Bruno Siciliano et al. *Robotica. Modellistica, pianificazione e controllo*.