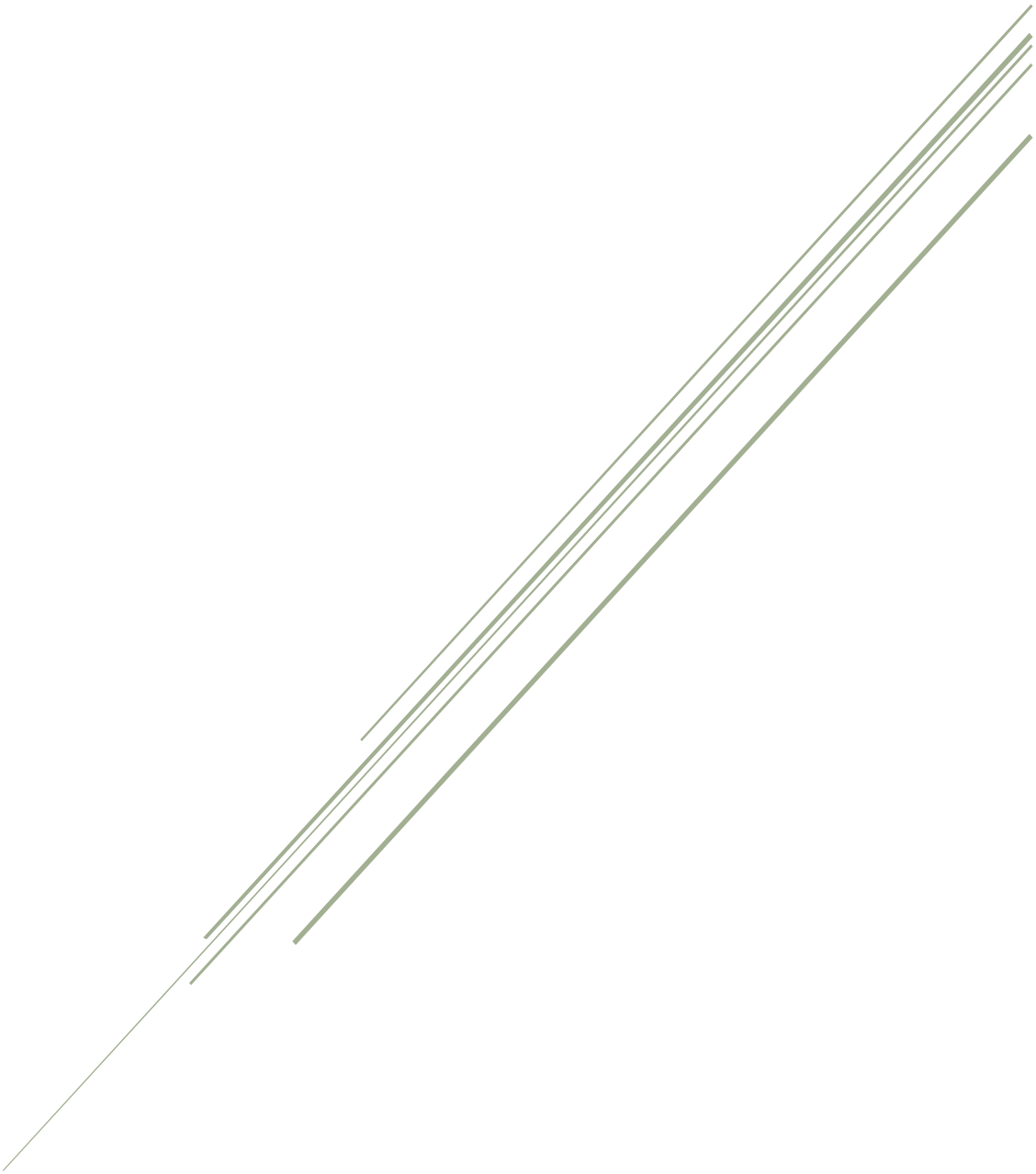
RELAZIONE

Progetto Robotica Mobile



Giaimo Natale
Mat. 209424

Introduzione

Il progetto è stato realizzato in MatLab, è costituito da uno script principale, *Mobile.m*, all'interno del quale sono presenti due variabili, *planner* e *controller*, da modificare in base alle istruzioni riportate nei commenti per richiamare i vari algoritmi implementati. Tutte le misure sono espresse in metri e gli angoli in radianti.
Ogni algoritmo è una funzione memorizzata all'interno delle rispettive sottocartelle.
Il modello di robot utilizzato è un uniciclo con lato di dimensione dBot (default $20[cm]$).

Robot Model

Il modello di robot utilizzato è l'uniciclo; questo impone i seguenti vincoli:

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

dai quali segue il vincolo anolonomo su θ:

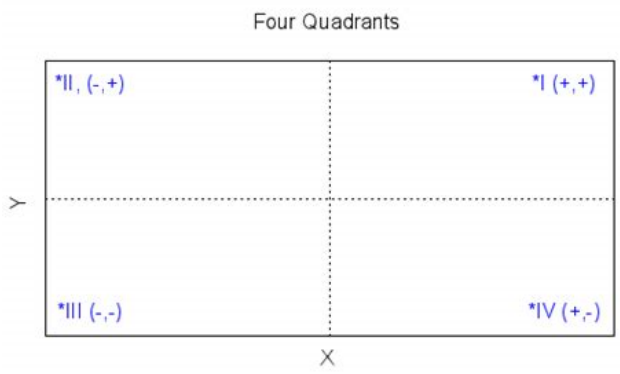
$$\theta(t) = \tan_2^{-1}\left(\frac{\dot{y}(t)}{\dot{x}(t)}\right)$$

Il modello descritto verrà introdotto nel blocco di controllo per la risoluzione del problema di trajectory tracking. Il rispetto del vincolo anolonomo viene imposto andando a calcolare l'angolo θ proprio dalla formula sopra espressa.
Per questioni di semplicità il robot avrà velocità nulla all'istante temporale $t_0 = 0$:

$$[v(t_0), \omega(t_0)] = [0,0]$$

Ostacoli ed Ambiente

Ostacoli ed ambiente sono espressi nei termini dei loro vertici. Immaginando una divisione in quadranti con origine nel centro dell'oggetto, i vertici sono memorizzati in senso orario dal vertice che risiede nel I quadrante.



Un qualsiasi ostacolo presente nell'ambiente viene inscritto in un rettangolo, questo semplifica notevolmente la rappresentazione e permette ai vari algoritmi di lavorare su un'unica struttura dati che contiene tutti gli ostacoli, senza dover analizzare casi specifici per forme più complesse come circonferenze o ellissi. Il vantaggio principale derivato dall'aver una descrizione in base ai vertici è il poter utilizzare la funzione *inpolygon* di MatLab; avere un modo immediato e affidabile di controllare se un punto appartiene all'interno o alla frontiera di un ostacolo è fondamentale per il funzionamento di tutti gli algoritmi di Path Planning implementati.
All'interno della funzione *Obstacles* vengono dichiarati gli oggetti e restituiti già ingrassati; così facendo abbiamo sicurezza che, indipendentemente dall'algoritmo utilizzato, una navigazione sui bordi è sempre possibile e priva di collisioni. Questo comporta che i vertici di un ostacolo possano risiedere all'esterno dell'ambiente; questa eventualità viene trattata in dettaglio in ogni algoritmo. In molte occasioni servirà lavorare su una discretizzazione dei lati, la funzione *ObstacleToShape* ritorna un vettore $Nx2$ con le coppie $[x, y]$ che rappresentano tutti i punti della discretizzazione di fattore *disc*.

Path Planning

Tutte le funzioni prendono in input ambiente, ostacoli, coordinate del punto di start e del punto di goal (APF richiede anche dBot). In output forniscono un vettore *via* $Nx2$ che rappresenta le coppie delle coordinate dei punti che descrivono la path.

APF

Fortemente basato sul codice mostrato a lezione (A.A. 2021/2022), la differenza principale deriva sui punti nei quali viene calcolato il potenziale repulsivo e il calcolo effettivo del gradiente. Nello script mostrato a lezione esso veniva calcolato ad ogni iterazione della procedura di path planning nel solo punto corrente; nella mia implementazione il campo potenziale totale viene calcolato per ogni punto di una discretizzazione della griglia rappresentante l'ambiente, dividendo il calcolo dalla procedura di path finding. Questa scelta è motivata dal costo computazione del calcolo ad ogni iterazione del contributo di ogni

ostacolo; assenza di ostacoli mobili; semplicità di implementazione. Le funzioni utilizzate sono:

$$f_{attr}(x,y) = \frac{1}{2} * ((x-X)^2 + (y-Y)^2)$$

$$f_{rep}(x,y) = \frac{1}{((x-X)^2 + (y-Y)^2)}$$

Con $[x,y]$ coordinate del punto analizzato e $[X,Y]$ coordinate del punto che causa l'effetto (goal nel caso attrattivo, un punto sul lato dell'ostacolo e dell'ambiente nel caso del repulsivo). Ho provato diverse funzioni per una rappresentazione degli ostacoli, alla fine ho deciso di applicare il repulsivo per ogni punto su una discretizzazione dei lati.
Viene introdotta anche:

$$\rho(x,y) \coloneqq ((x-X)^2 - (y-Y)^2) \leq dBot^2$$

Utilizzata per applicare l'effetto repulsivo in un intorno del punto. Anche se l'ostacolo è già ingrassato si vuole evitare una sovrapposizione di effetti tra più ostacoli vicini. Grazie all'utilizzo delle costanti w_a *e* w_o il contributo del repulsivo può essere maggiormente controllato.
In fine una matrice di rotazione

$$rm(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Essa viene applicata sulle coordinate del gradiente ∇J_r per fornire una rotazione θ alla direzione del repulsivo. Importante specificare l'importanza della scelta di θ, la rotazione deve permettere al robot di circumnavigare l'ostacolo in una direzione che non vada a impattare con l'ambiente. Per ovviare a questo problema $\theta \in \left\{\frac{\pi}{3}, -\frac{\pi}{3}\right\}$; il repulsivo continuerà così a descrivere un allontanamento dall'ostacolo ma descrivendo una traiettoria circolare. La scelta di θ viene effettuata per ogni ostacolo prima del calcolo del repulsivo.
Si calcola la pendenza delle rette con estremi

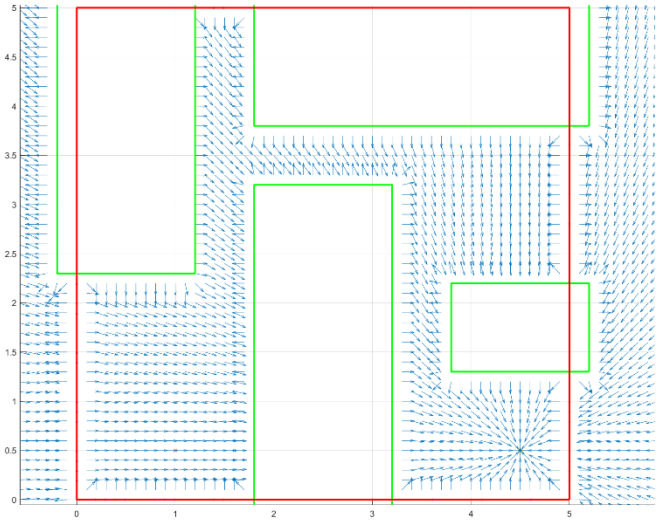
$$r_g = [centro_{obs}, goal], r_s = [centro_{obs}, start]$$

ottenuta la loro pendenza m si usufruisce della funzione $\tan^{-1}(m)$ per ottenere l'angolo rispetto all'asse delle x. Con

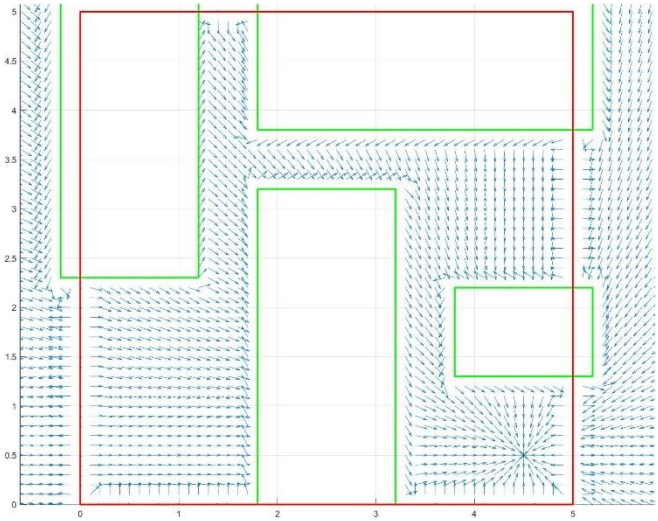
$$\theta_s = \tan^{-1}(m_s)\,; \theta_g = \tan^{-1}(m_s)$$

$$\theta_i = \begin{cases} -\frac{\pi}{3} \; se \; \theta_s > \theta_g \\ \frac{\pi}{3} \; altrimenti \end{cases}$$

$i \; indice \; ostacolo \; nella \; cell \; structure \; O$



Potenziale totale in assenza dell'applicazione della matrice di rotazione.



Potenziale totale con matrice di rotazione applicata al repulsivo.

Viene creata una *meshgrid* $[XX, YY]$ su tutti i punti descritti da un ingrassamento di 2 metri dell'ambiente (necessario per evitare che vengano forniti errori se punti di un ostacolo si trovano all'esterno dell'ambiente). Il potenziale attrattivo viene calcolato applicando ∇f_{attr} su tutti i punti della *meshgrid*.
Il discorso si complica per il repulsivo sugli ostacoli; data la decisione di applicare ∇f_{rep} sui punti della discretizzazione, viene creata una matrice *in* contenente 1 sui punti che si trovano all'interno degli ostacoli. Due *for* innestati iterano sugli indici ($\dim(XX) == \dim(in)$) e si applica il repulsivo sui punti $[i,j]\; t.c. \; in(i,j) == 1$.
Una volta calcolato il contributo dell'ostacolo corrente si procede all'applicazione della matrice di rotazione per tutte le coppie di punti influenzate dal contributo e si somma il risultato al repulsivo totale.
A questo punto siamo in possesso del repulsivo e

dell'attrattivo, possiamo quindi calcolare il campo totale (nel rispetto delle direzioni) con la formula

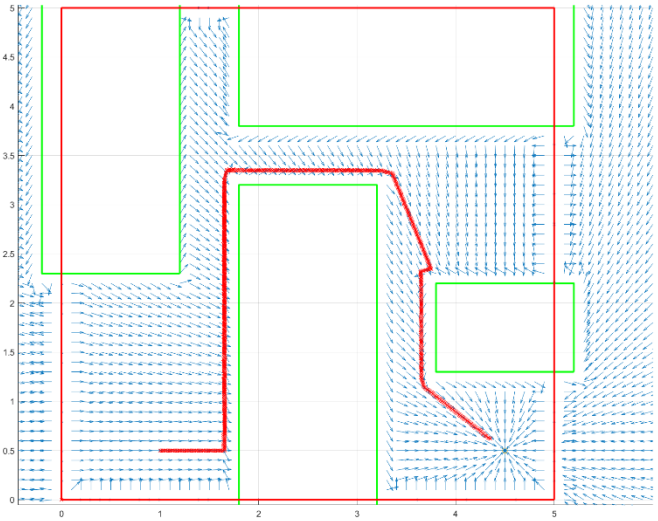
$$\nabla J_i = w_a * \nabla J_{ai} + wo * \nabla J_{ri}, i \in \{x, y\}$$

Procediamo a normalizzare la matrice per ottenere vettori unitari e poter descrivere lo spostamento nella fase di path planning.

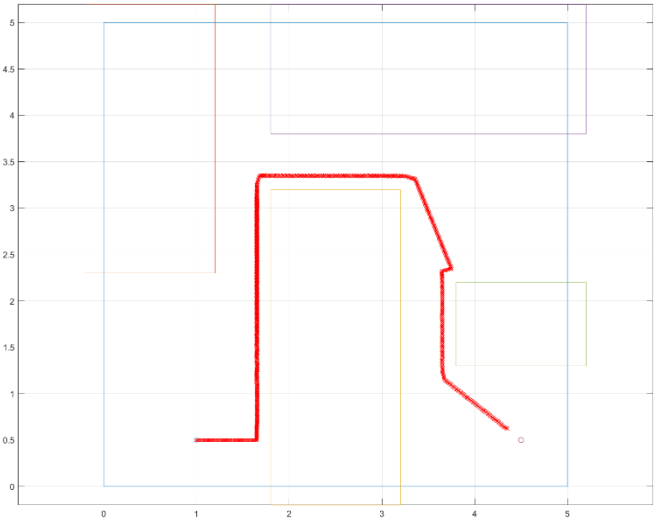
La fase di path planning effettivo si basa fortemente sulla soluzione fornita a lezione: una struttura *for* itera per un massimo di *iter* passi (limite necessario per evitare il blocco dell'algoritmo in punti di minimo) oppure finché non si verifica $norm(Xcurr, Goal) \leq dBot$, ovvero il punto corrente si trova in un intorno di raggio *dBot* dal goal.
Le coordinate saranno memorizzate in un vettore X di dimensione $iter\ x\ 2$.
Un problema riscontrato durante la progettazione consiste nell'avere una corrispondenza tra la rappresentazione discretizzata dell'ambiente e valori della posizione forniti dal passo precedente; la soluzione adottata comporta costi computazionali elevati ma, a seguito di altre tecniche testate, questa risulta la più affidabile nel contesto corrente.
Ogni iterazione si apre andando a calcolare la distanza euclidea (al quadrato) tra il punto corrente e tutti i punti della *meshgrid*. Il minimo valore fornisce l'indice del punto discretizzato $X_{disc} = [x, y]$ nei termini della *meshgrid*; ricaviamo gli indici tramite la funzione *find* di MatLab.
Il valore del gradiente viene trovato interpolando $\nabla J_{nx}\ e\ \nabla J_{ny}$ nel punto corrente. Trovato il valore possiamo applicare la funzione

$$X_{succ} = X_{curr} - \alpha * \nabla J_n(X_{disc})$$

Memorizziamo X_{succ} nella posizione del vettore X corrispondente al passo iterativo corrente.
Concluso il ciclo, viene effettuato un plot dei dati ottenuti finora e la path trovata viene salvata all'interno di un vettore via di dimensione $kx2$.
Questo vettore sarà l'oggetto restituito a fine svolgimento della funzione.



Ambiente sovrapposto al potenziale e ai punti appartenenti alla path identificata.



In rosso i punti appartenenti alla path identificata dalla tecnica APF.

DPF

Per la tecnica dei Discrete Potential Fields si lavora strettamente su una discretizzazione dell'ambiente di passo Δ. Viene creata una *meshgrid* $[XX, YY]$ in base a questa discretizzazione, traduciamo i punti dal sistema di riferimento cartesiano al sistema discretizzato tramite la funzione *pointToGrid*.
Viene inizializzata la matrice *envMatrix* che verrà popolata con i valori dei DPF.

Come primo passo poniamo il valore *Inf* su tutti i punti appartenenti agli ostacoli; vengono sovrapposti gli output della funzione *inpolygon*, applicata su tutti i punti della *meshgrid* per ogni ostacolo, in una matrice *inp*. Iterando su tutti i punti della discretizzazione poniamo

$$envMatrix(i, j) = Inf \leftrightarrow inp(i, j) == 1$$

Possiamo, adesso, richiamare la funzione *FillMatrix* per popolare la matrice *envMatrix* coi valori K partendo dal goal. Nella soluzione

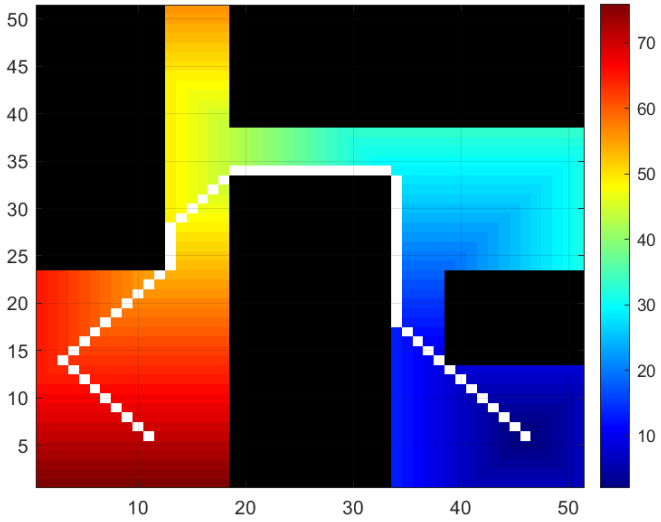
adottata il valore K al goal è pari a 1. Poiché *envMatrix* è una matrice di 0 e su questa proprietà si basa la tecnica di popolazione implementata, uno 0 al goal sarebbe stato trattato come un punto non ancora visitato.

La funzione **FillMatrix** prende in input la matrice e il punto di goal, restituisce la matrice presa in input popolata secondo la tecnica dei DPF. Per facilitare la procedura si implementa una struttura FIFO descritta da un vettore *queue* e le funzioni *Pop* e *Push*. Essa contiene una tupla di valori $[i, j, val]$ che rappresentano le coordinate di una cella e il valore da memorizzare in essa. La funzione *Neighbours* restituisce un vettore contenente le tuple dei vicini della cella corrente nella forma sopra descritta; la sua descrizione è riportata più avanti.
Un ciclo while itera finché la queue non sarà vuota, ovvero finché tutte le celle raggiungibili dal goal non saranno popolate. Ad ogni iterazione si prende l'elemento in cima alla queue, si verifica se esso è Inf o ha un valore diverso da 0 (è ostacolo o è già popolato); gli si assegna il suo valore e si aggiungono in coda alla queue i suoi vicini. La struttura FIFO della queue garantisce che la popolazione della matrice avvenga in modo ordinato, garantendo la consistenza del segnale artificiale diramato dal goal.

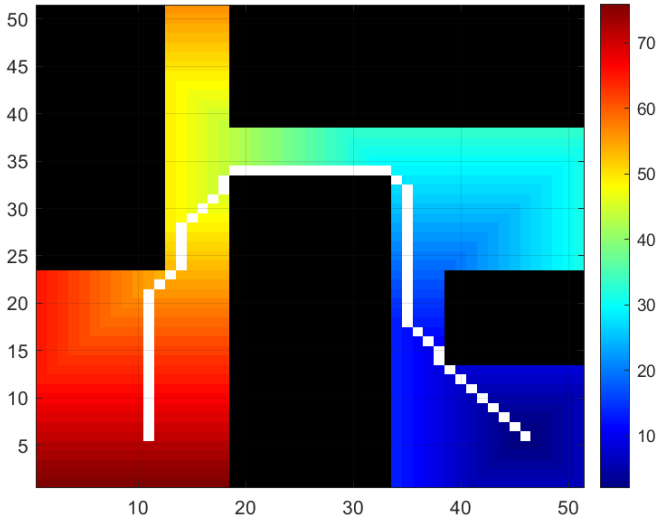
Entriamo in una descrizione più dettagliata della funzione **Neighbours**, essa prende in input le coordinate della cella corrente c_{curr}, il valore $k \coloneqq envMatrix(c_{curr}) + 1$ e la dimensione della matrice (descritta dal vettore *lim*); restituisce in output un vettore $Nx3$ contenente tuple analoghe a quelle nella queue. Due for innestati effettuano un ciclo che visita tutte le possibili celle adiacenti a c_{curr}; degli if controllano se gli indici sono ammissibili o se stiamo analizzando c_{curr}, saltando l'iterazione se una di queste condizioni si avvera. Se la cella risulta "legale" allora viene aggiunta in coda alla struttura n assieme al valore che andrà in esso memorizzato.

Tornando nella funzione *DPF*, siamo adesso in possesso della matrice *envMatrix* popolata, procediamo adesso con il trovare la path tramite la funzione *FindPath*.

FindPath prende in input la *envMatrix* e gli indici delle celle corrispondenti al punto di start e al punto di goal; restituisce, in output, un vettore $Nx2$ rappresentante le coppie di coordinate delle celle appartenenti alla path.
Viene "riciclata" la funzione Neighbours, ignorando il terzo valore della tupla; viene implementata una funzione per calcolare la distanza euclidea al quadrato tra le coordinate del punto corrente e del goal.
L'algoritmo si basa sul percorrere a ritroso il segnale generato dalla funzione *FillMatrix*. Nella queue vengono memorizzati i vicini dell'ultimo valore inserito nella path, si procede quindi a iterare finché la queue non è vuota; questo comporta che non esistono vicini con valore inferiore a quello dell'ultima cella salvato nella path, ovvero l'ultima cella visitata è proprio la cella goal.
Sia C_{last} l'ultima cella visitata e C_{curr} la cella in analisi, se $K_{C_{last}} > K_{C_{curr}}$ con K valore contenuto nella cella, allora procediamo ad analizzare il resto della queue per trovare, se esiste, un punto più vicino al goal rispetto a quello corrente. Questo passaggio di confronto di distanze è necessario per garantire che la path descriva un percorso più breve. Di seguito due immagini che mettono in risalto l'importanza dell'uso delle distanze; in nero gli ostacoli, in bianco la path identificata.



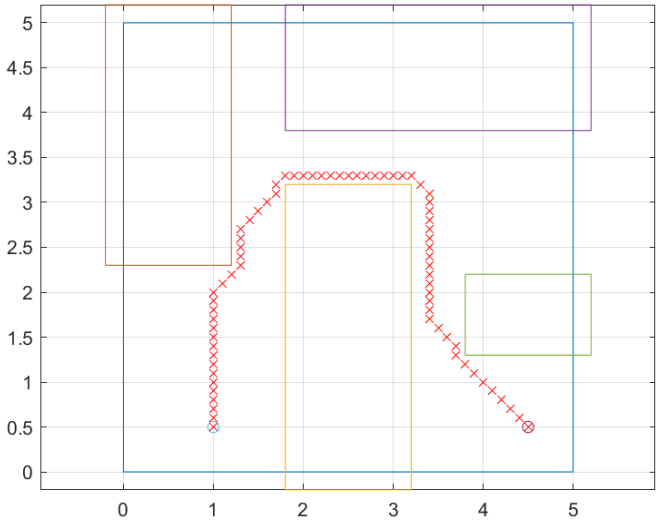
La path identificata considera il primo punto che rispetta la condizione $K_{C_{last}} > K_{C_{curr}}$ come prossimo passo.



Notiamo come l'introduzione della condizione di minima distanza migliora notevolmente la traiettoria identificata.

Una volta trovato il prossimo punto, viene salvato in coda al vettore path e si procede con una nuova iterazione.

Una volta terminata la funzione *FindPath* siamo in possesso delle coordinate nei termini della griglia della via, traduciamole nuovamente nel sistema di riferimento cartesiano e richiamiamo il plot dell'ambiente discretizzato. Il codice per una colormap adatta al contesto è stato fornito da *Martin Grunnill* al seguente link.



In rosso i punti appartenenti alla path identificata dalla tecnica DPF.

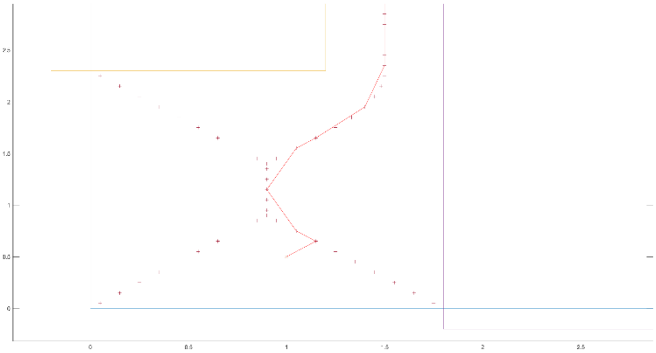
Voronoi Diagrams

Forniamo una descrizione adatta al contesto per i punti generatori; salviamo in una *cell structure* la discretizzazione degli ostacoli e dell'ambiente tramite la funzione *ObstacleToShape* come precedentemente descritto. Procediamo a salvare tutte le coppie dei punti presenti nella struttura in due vettori x, y. Richiamiamo la funzione di MatLab *voronoin* ottenendo le coordinate di tutti i vertici ottenuti dai generatori forniti in una matrice v di dimensione $Nx2$. Data la natura della rappresentazione, all'interno di v saranno presenti anche vertici interni agli ostacoli ed esterni al perimetro; tramite la funzione *inpolygon* rimuoviamo questi vertici, lasciando in v solamente le coppie che descrivono posizioni ammissibili nell'ambiente.

Possiamo, quindi, creare una matrice di adiacenza per poter popolare un grafo pesato e applicare un algoritmo noto per la risoluzione del problema.

La funzione **adjMatrix** prende in input i vertici v e restituisce la matrice di adiacenza. Un ciclo *for* itera su tutti gli elementi di v e, ad ogni iterazione, inserisce nella matrice la distanza tra il vertice corrente e tutti i vertici in un suo intorno di raggio $0.5[m]$. Questo raggio determinerà quali vertici(nodi) saranno connessi nel grafo.

È doveroso considerare che questa soluzione introduce imprecisioni per quanto riguarda la risoluzione della path. In intorni dove sono presenti tanti vertici la path "taglierà" l'agglomerato, prediligendo il vertice più vicino al goal. A seguito di vari test ho considerato il raggio 0.5 come un valore che mette in equilibrio questo fenomeno di "taglio" e il rispetto del percorso descritto dai vertici.

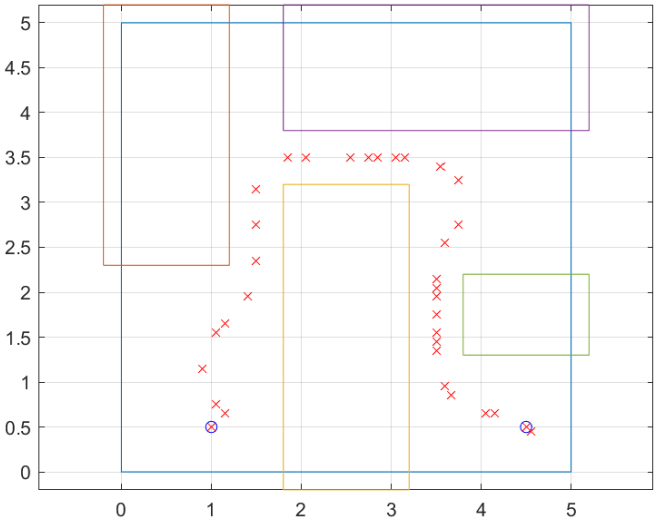


Notiamo il "fenomeno di taglio" sopra descritto. Con + sono identificati i vertici di Voronoi, la linea rappresenta il percorso identificato dalla risoluzione del grafo.

Il peso degli archi è la distanza euclidea al quadrato tra i nodi che l'arco stesso collega.

Una volta ottenuta la matrice di adiacenza utilizziamo la funzione di MatLab *graph* per ottenere, appunto, un grafo. Identifichiamo i vertici (e, di conseguenza, i nodi) più vicini al punto di start e al punto di goal; richiamiamo la funzione di MatLab *shortestpath* per ottenere il

percorso a costo minimo tra i nodi precedentemente identificati. Memorizziamo nella variabile di ritorno *via* la path ottenuta, forzando il primo e l'ultimo valore a, rispettivamente, start e goal.



In rosso i punti appartenenti alla path identificata dalla tecnica dei diagrammi di Voronoi.

Visibility Graphs

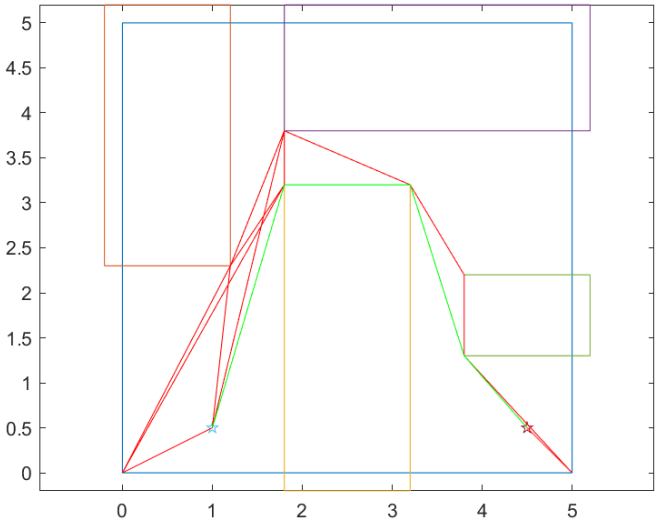
Nel progetto è stata implementata la versione semplice dei grafi di visibilità.
Memorizziamo in due vettori x_{vect}, y_{vect} le coordinate dei vertici di ostacoli e ambiente. Tuttavia, vogliamo analizzare solo vertici che possono essere raggiunti dal robot, quindi procediamo a utilizzare la funzione *inpolygon* per rimuovere da x, y tutti i punti che sono all'interno di ostacoli o all'esterno dell'ambiente.

Ottenuti i vertici, procediamo a considerare le linee che li connettono, memorizzando solo i segmenti ammissibili.
I segmenti vengono memorizzati in una cell structure *lines*. Ogni cella contiene una matrice $Nx2$ tali che $lines\{k\}(i,:) = [x_i, y_i]$;
k rappresenta l'indice di un punto $[x_{vect}(k), y_{vect}(k)]$, il contenuto di $lines\{k\}$ sono le coordinate dei vertici che descrivono segmenti ammissibili nell'ambiente corrente con il vertice k.
Tramite due for innestati controlliamo ogni coppia di punti per validare il segmento da essi descritto.
Per controllare se un segmento è ammissibile, si utilizza la funzione di MatLab *polyxpoly* per controllare se esistono intersezioni con gli ostacoli. La funzione considera intersezione se un vertice è estremo del segmento; per rimuovere questa eventualità utilizziamo la funzione *setdiff* per "pulire" le coordinate di intersezione dai vertici. Importante sottolineare che, per le verifiche svolte finora, un segmento che descrive una diagonale nell'ostacolo viene considerato ammissibile, poiché non interseca un ostacolo e i suoi estremi non si trovano all'interno dell'ostacolo stesso; utilizziamo la funzione *isDiagonal* per controllare questa eventualità (descrizione fornita in seguito). Una volta effettuati questi controlli, il segmento viene aggiunto alla lista se e solo se non esistono intersezioni.

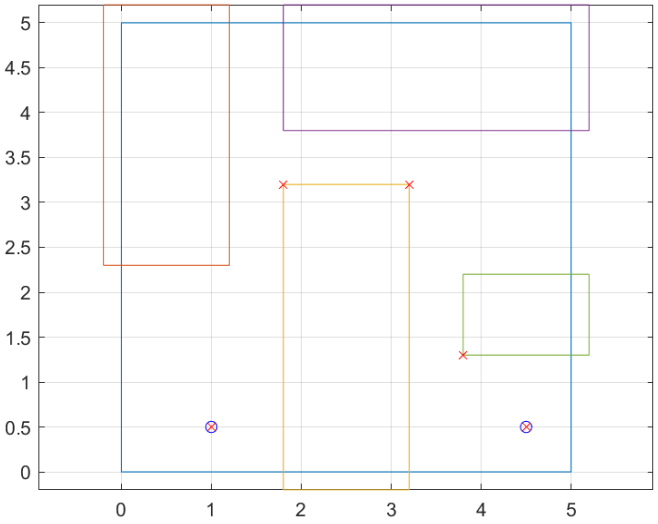
La funzione **isDiagonal** riceve in input i vertici di un ostacolo e una matrice $2x2$ rappresentante gli estremi del segmento in analisi. Si calcola il punto medio di questo segmento e, se esso risulta interno all'ostacolo, allora l'output della funzione sarà 1 e il segmento verrà ignorato nel popolamento della structure *lines*.

Una volta ottenuta la lista dei segmenti ammissibili, procediamo a creare la matrice di adiacenza tramite la funzione **adjMatrix** (differente dalla funzione utilizzata nello script Voronoi). In input richiede i vettori x_{vect}, y_{vect} e la struttura *lines*, fornendo in output la matrice. Per ogni segmento descritto dalla struttura *lines* calcola la distanza tra gli estremi e utilizza questa quantità come peso.

Ottenuta la matrice di adiacenza richiamiamo la funzione *graph* e *shortestpath* per popolare un grafo pesato e risolvere il problema di percorso a costo minimo dal nodo di start al nodo di goal, memorizzando il risultato nella variabile di output della funzione *via*.



In rosso i segmenti ammissibili, in verde il percorso identificato dalla soluzione del problema sul grafo.



Indicati con X rosse gli estremi dei segmenti che compongono il percorso identificato dalla tecnica dei grafi di visibilità.

Trajectory Tracking

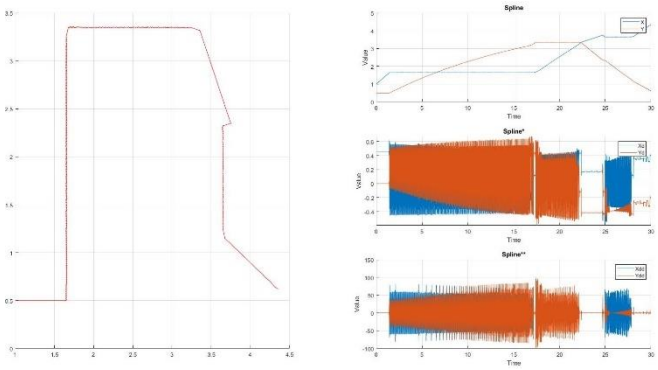
La funzione *TrakTracking*, richiamata all'interno dello script *mobile*, si occupa di ricavare una funzione rappresentate i punti ricevuti dalla fase di path planning e seguente implementazione del blocco *controller* all'interno del progetto.

Ricavo della funzione

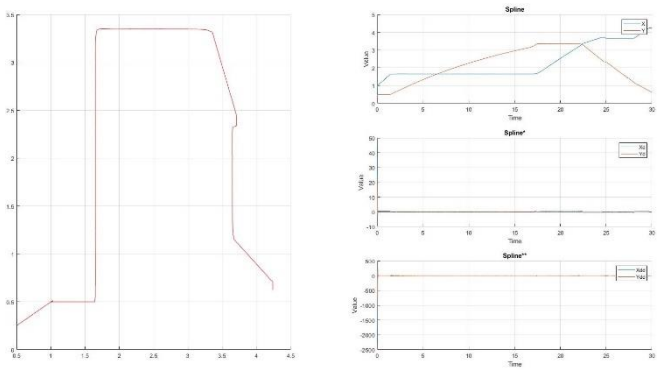
Poiché i punti ottenuti dalla fase di path planning non sono affetti da rumore, ho scelto di utilizzare la funzione *csapi* di MatLab per ricavare una funzione polinomiale definita a tratti, avendo a tutti gli effetti una tecnica di interpolazione per il ricavo della funzione. Una volta ricavate le funzioni x_{fun} e y_{fun} procediamo a ricavate derivate prime e seconde tramite la funzione *fnder* di MatLab. Questa è una rappresentazione *ppform* ma, per garantire il funzionamento di *ode45* dobbiamo poter ricavare il loro valore in funzione del tempo. Dichiariamo, quindi, una nuova funzione per ognuna delle *ppform* per permettere di ottenere valori finiti ad un istante di tempo t tramite la funzione *ppval* di MatLab. Otteniamo, inoltre, l'angolo θ forzando il rispetto del vincolo anolonomo per il modello uniciclo:

$$\theta(t) = \tan_2^{-1}\left(\frac{\dot{y}(t)}{\dot{x}(t)}\right)$$

Avendo difatti ottenuto le derivate della funzione polinomiale, è diretto il ricavo del valore di θ ad ogni istante temporale.
Nel contesto di una path generata dalla tecnica APF i punti risultano sparsi e, quindi, sarebbe preferibile ricavare una funzione tramite una tecnica di approssimazione; tuttavia, non sono riuscito a produrre del codice che portasse a una funzione approssimante consona. Ho quindi deciso di applicare un filtro mediano tramite la funzione *medfilt1* di MatLab applicata sui valori $[x, y]$ forniti dalla fase di path planning APF. Il filtro sostituisce ogni entry con la mediana di una finestra di dimensione $N = 30$ sui suoi vicini; questo ci permette di avere un segnale più dolce ma che presenta ancora molti cambi repentini di direzione in prossimità degli ostacoli. Di seguito l'esempio della spline ottenuta con e senza filtro sui punti passati.



Spline ricavata dalle coordinate non filtrate.



Spline ottenuta dalle coordinate filtrate.

Control

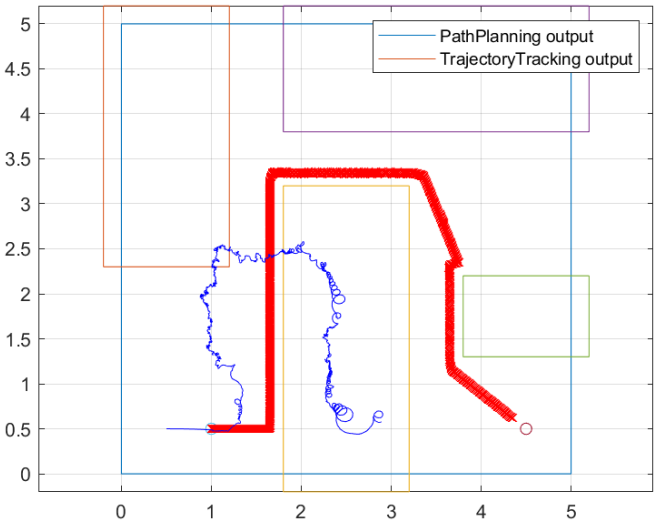
La progettazione di leggi di controllo è stata basata fortemente sulle formule ed esempi forniti a lezione (AA 2021/2022). Da un punto di vista generale le tre leggi di controllo implementate (approssimazione lineare, controllo non lineare e I/O Linearization) sono tre funzioni diverse situate all'interno della cartella Trajectory Tracking; esse vengono richiamate all'interno di uno switch come funzioni di tempo t e posizione del robot $X = [x, y, \theta]$, questo permette l'utilizzo di *ode45* per la risoluzione del sistema fornito dalle stesse funzioni.
In base al tipo di planner utilizzato, è necessario

definire diverse costanti α e δ (per il controllo basato sulla linearizzazione dell'errore), e le funzioni k_1, k_2, k_3 (controllo non lineare). I valori inseriti nel codice sono stati ottenuti a seguito di vari test, prendendo quelli che più fanno avvicinare il percorso descritto dal robot alla path desiderata. Necessario specificare che i valori all'interno del codice sono stati tarati in base agli input di default arbitrari per *goal, start,* X e T_F, variare uno solo di questi valori porterà a risultati scorretti per le tecniche di controllo basate su approssimazione lineare e controllo non lineare.

Di seguito i grafici ottenuti dalle tre leggi di controllo applicate. In tutti i grafici in rosso sono segnati i punti forniti in output dalla fase di Path Planning; in blu la traiettoria descritta dalla fase di Trajectory Tracking. Tutti questi grafici possono essere ottenuti (per una migliore lettura dei dati) modificando opportunatamente le variabili *planner* e *controller* nello script *mobile*.

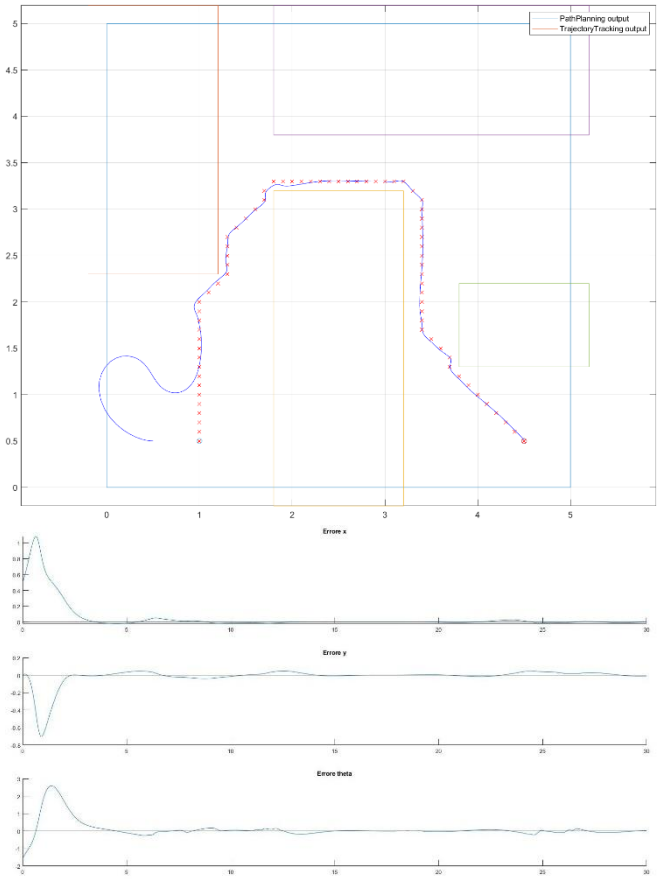
Linearization

APF



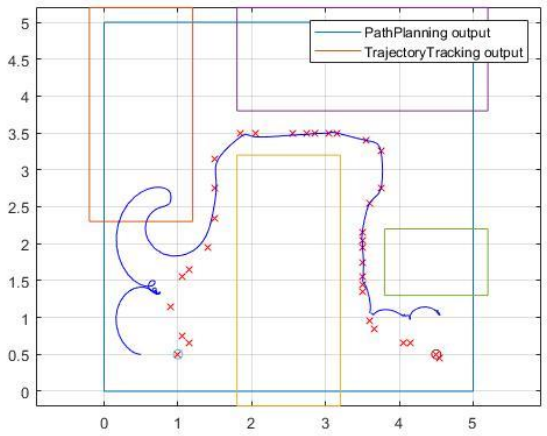
Indipendentemente dalla modifica delle costanti α, δ non sono riuscito a ottenere un risultato considerabile corretto per la path APF.

DPF



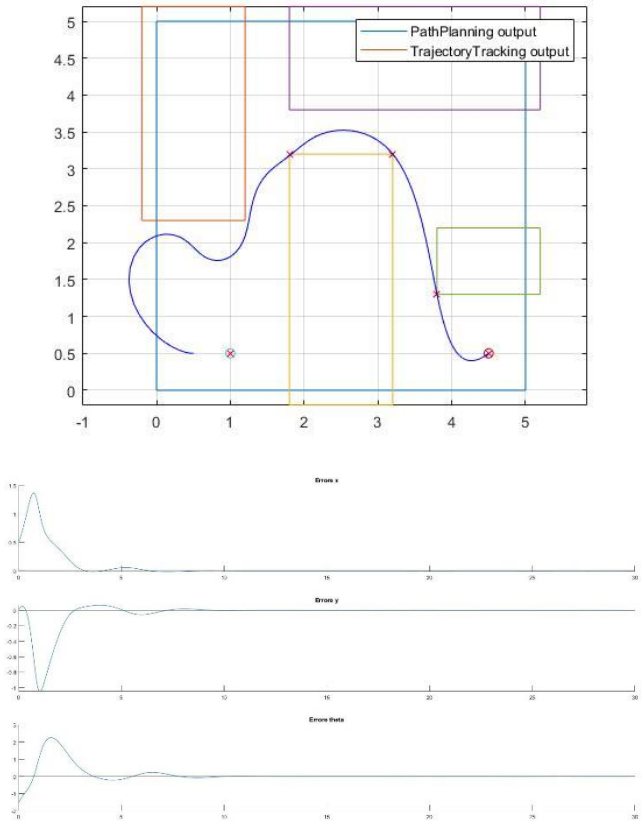
Ad eccezione della curva all'inizio della traiettoria, notiamo come gli errori, a seguito dell'agganciamento, rimangano in un intorno di 0. Questa curva iniziale deriva dal fatto che lo stato iniziale del robot $\theta_{bot}(T_0) = \pi$, esso ha quindi bisogno di agganciarsi alla traiettoria prima di poterla seguire.

Voronoi Diagrams



Situazione analoga alla path generata dagli APF.

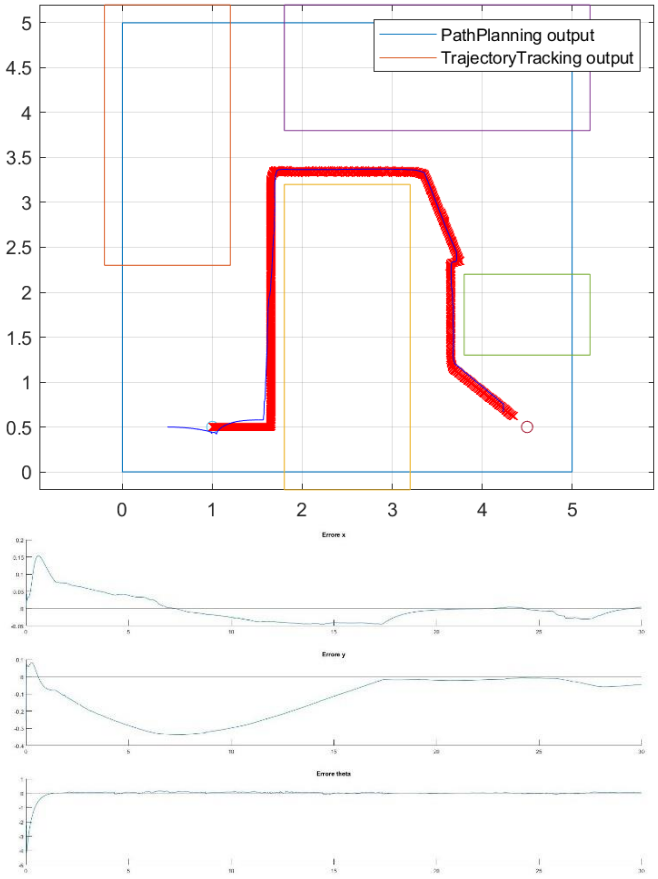
Visibility Diagrams



Analogamente alla situazione riscontrata con la traiettoria DPF, ad eccezione di una fluttuazione iniziale, gli errori rimangono in un intorno di 0 a seguito dell'agganciamento del robot alla path virtuale.

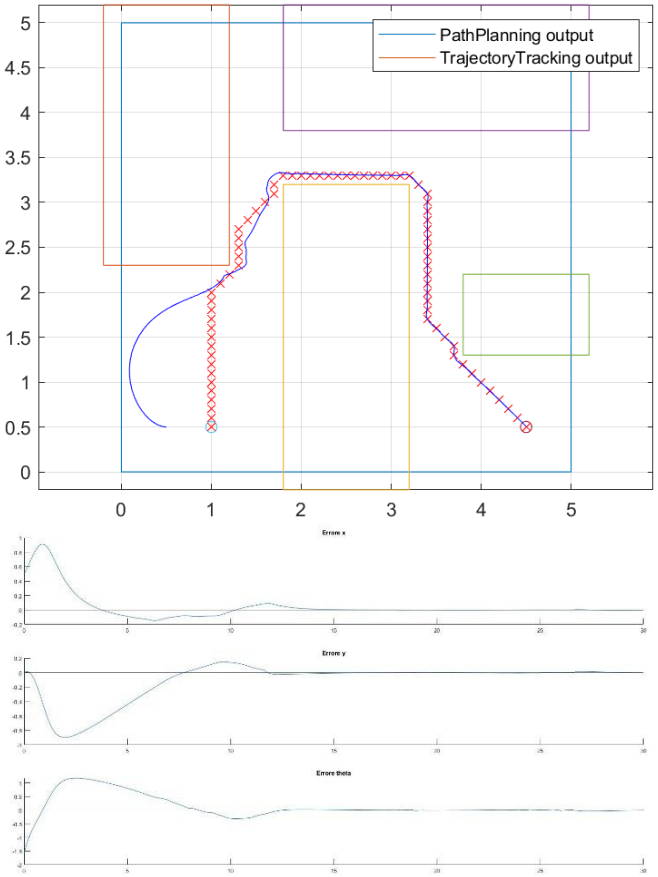
NonLinear control

APF



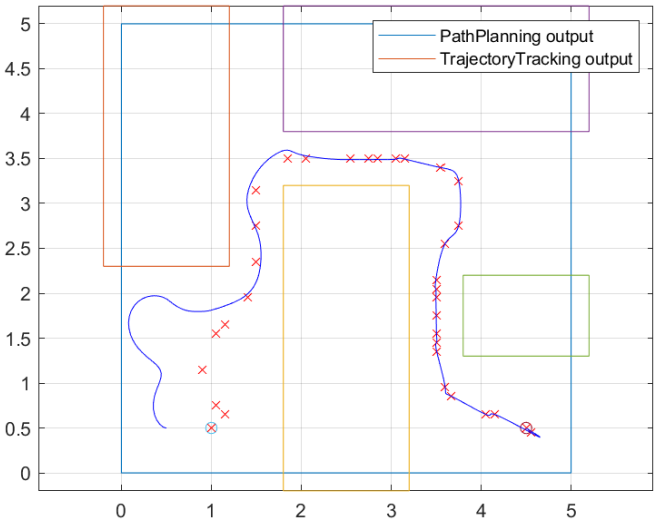
Gli errori rimangono in un intorno di 0 durante la percorrenza della traiettoria.

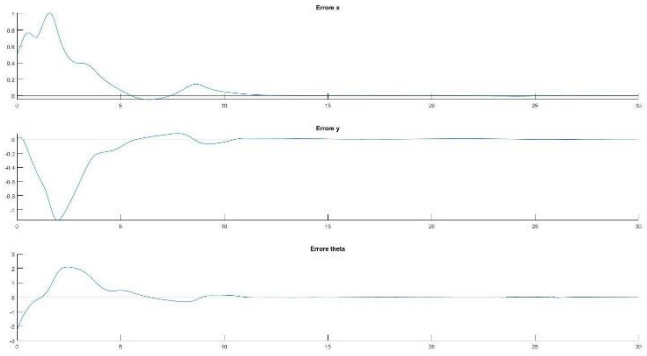
DPF



Si presenta lo stesso problema durante la fase di “agganciamento”, riconducibile allo stato iniziale del robot $\theta_{bot}(T_0) = \pi$.

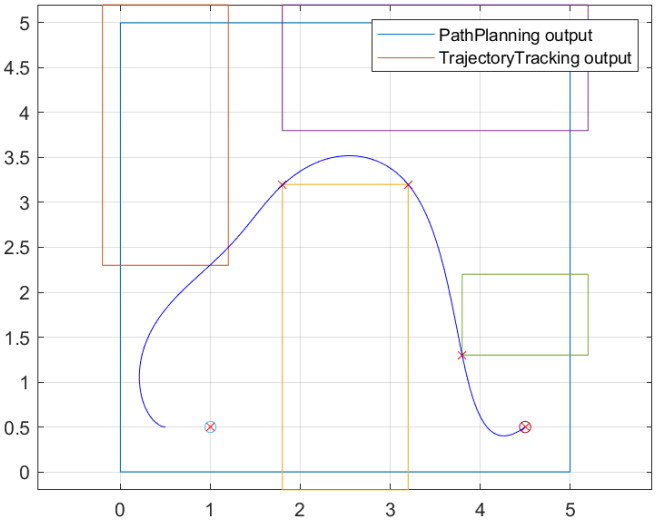
Voronoi Diagrams

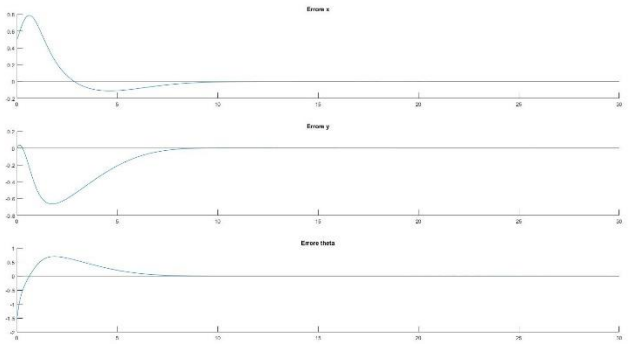




Indipendentemente dalla modifica del valore delle funzioni k_1, k_2, k_3 (costanti nel contesto corrente) la traiettoria ottenuta mantiene questo comportamento prima dell’agganciamento.

Visibility Graphs

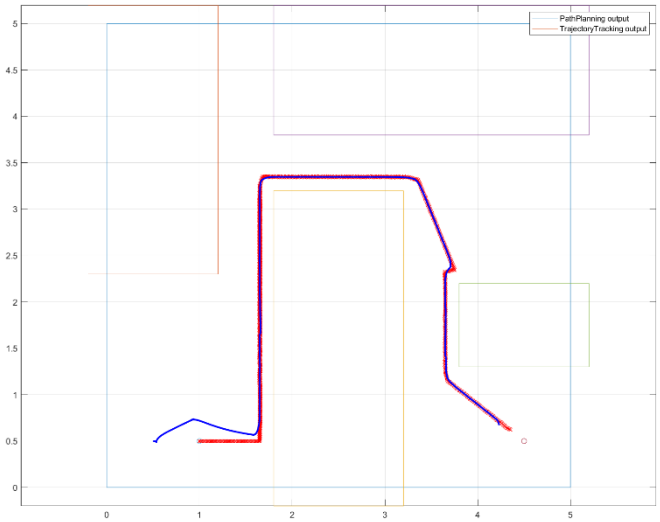




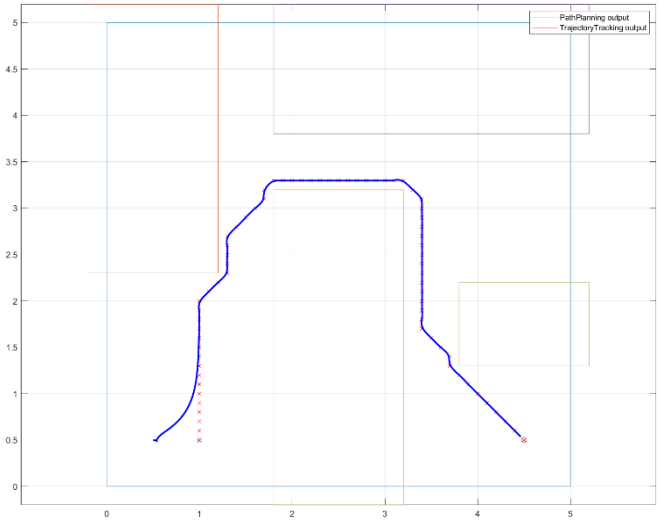
Notiamo lo stesso comportamento nella fase iniziale.

I/O Linearization

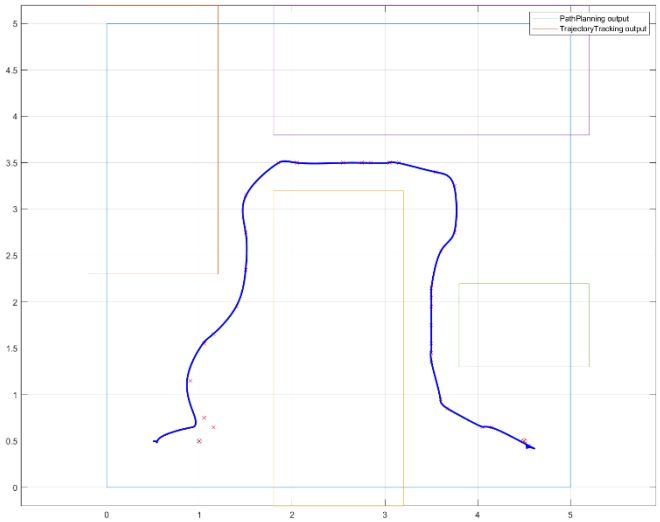
APF



DPF



Voronoi Diagrams



Visibility Diagrams