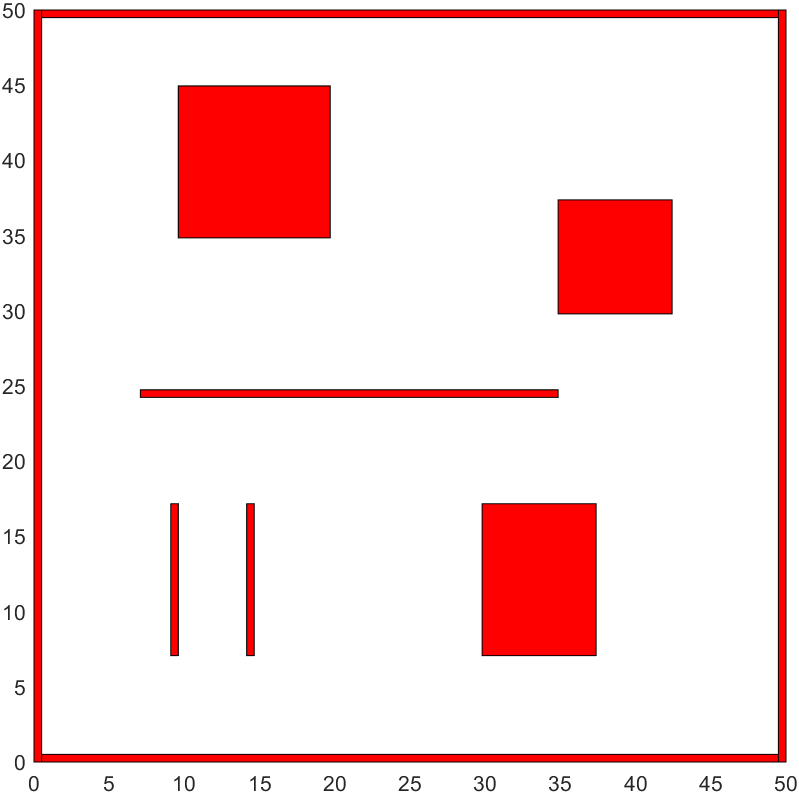
Progetto Robotica Mobile

Flavio Maiorana 182611

Tutti gli script del progetto possono essere direttamente eseguiti dal file mainScript, che contiene tutta la roadmap: dalla generazione degli ostacoli fino al controllo della traiettoria.

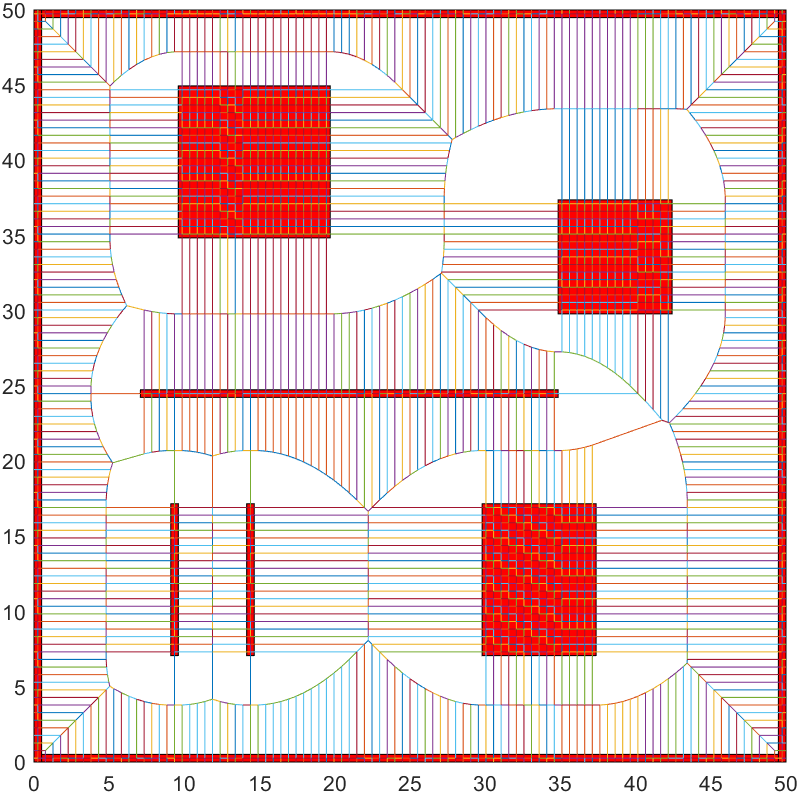
### Problema 1: Spazio di configurazione (funzione ostacoli)

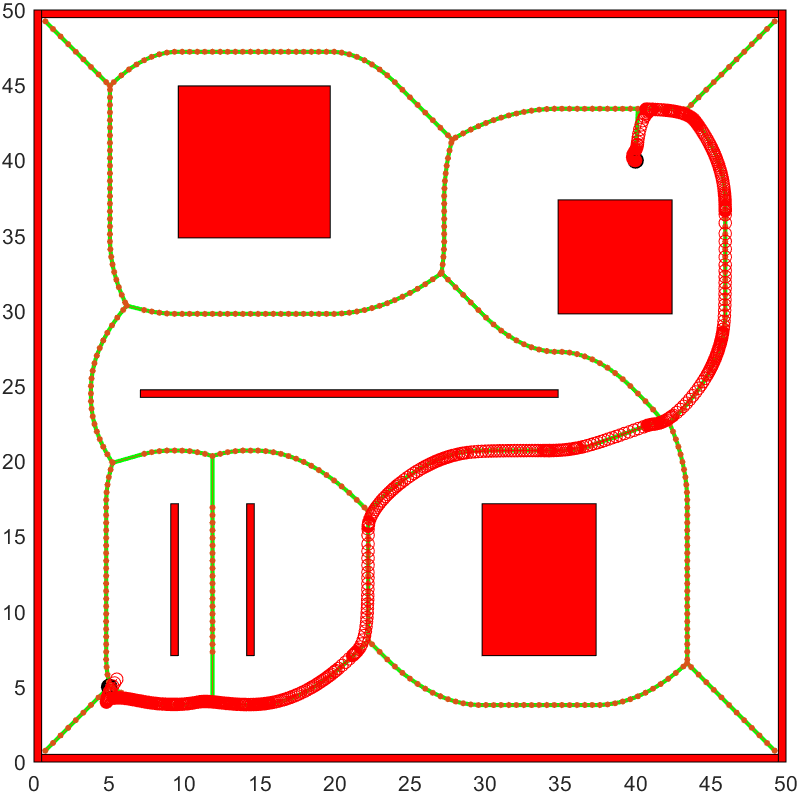
Ho scelto un cspace di dimensione 50\*50, delimitato da quattro muri che sono effettivamente considerati come degli ostacoli di forma rettangolare. Anche gli altri ostacoli interni sono rettangolari. Ho scelto di implementare lo spazio di configurazione come una griglia di occupazione binaria di dimensione 100\*100: le celle inizializzate a true sono quelle occupate da ostacoli. Inoltre, è necessaria una meshgrid di coordinate suddiviso in quadratini da 0.5 per trasportare le informazioni dalla griglia binaria al piano cartesiano.

### Problema 2: Motion Planning

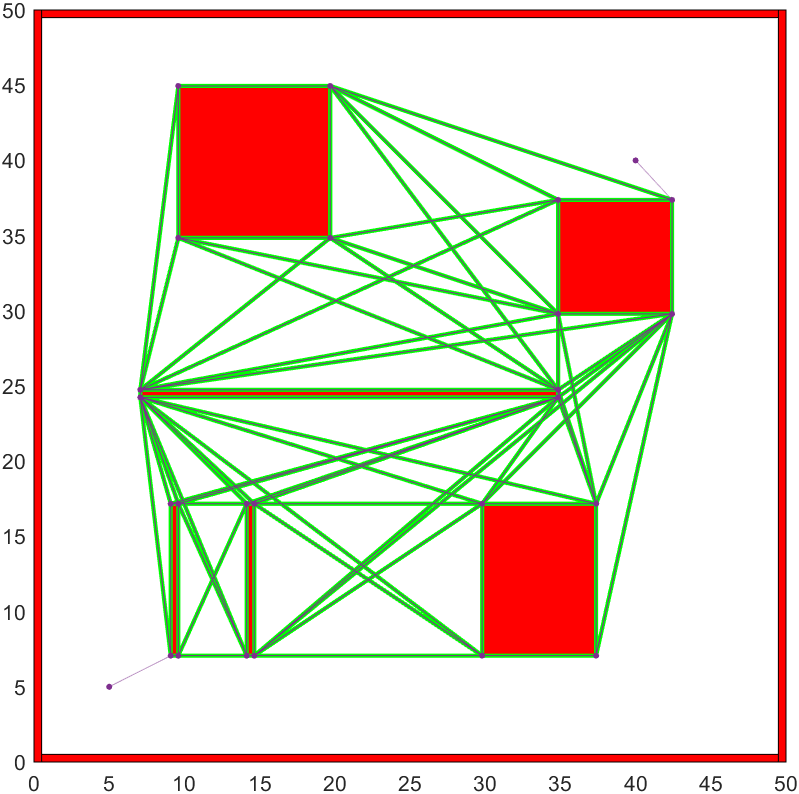
Per poter progettare traiettorie che evitino gli ostacoli ho implementato cinque metodi.

#### Ritrazione mediante diagramma di voronoi

Mediante la routine matlab voronoi ho generato il diagramma di voronoi, che ho poi pulito sfruttando lo script obstacleIntersect, per poi generare il grafo riempiendo una matrice di adiacenza. ObstacleIntersect discretizza ogni segmento del diagramma, per controllare se uno dei suoi “pezzettini” rientra in uno degli ostacoli. I nodi hanno un identificativo numerico corrispondente all’indice di un punto nell’array v (che contiene tutti i vertici generati dalla routine). Sulla sinistra il diagramma di voronoi generato da matlab, prendendo come punti di riferimento quelli della meshgrid.

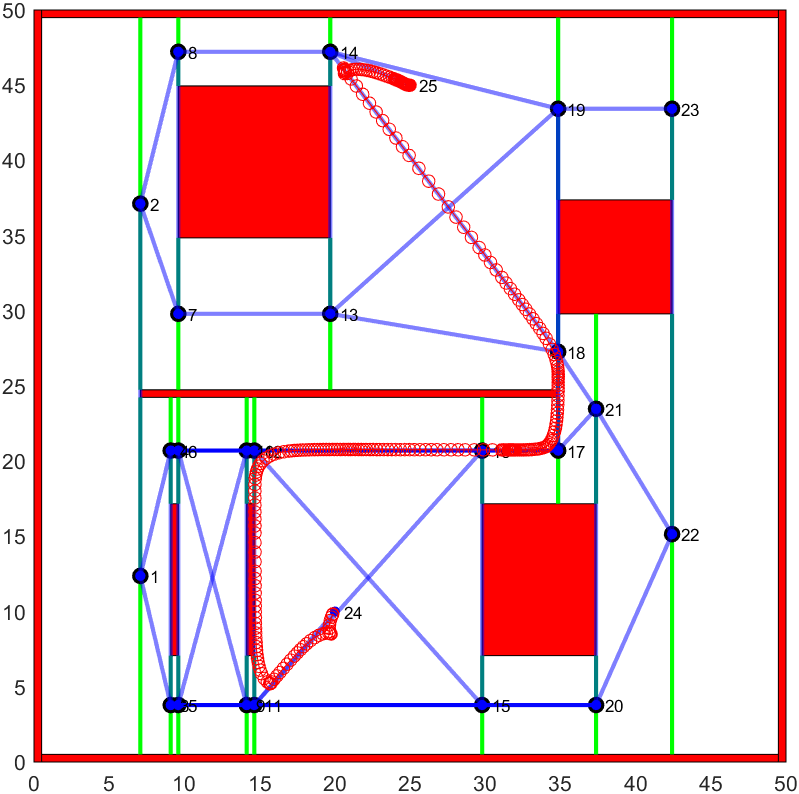
Qui invece il diagramma di Voronoi una volta terminata la pulitura da quei segmenti che intersecano gli ostacoli. Il grafo si prende tutti i vertici generati dal diagramma e li connette nella matrice di adiacenza, individuando ogni nodo come il suo indice dentro il vettore generato dalla routine voronoi. La matrice di adiacenza sará nxn, dove n è il numero di vertici del diagramma, e ogni elemento (i,j) è il collegamento tra nodo i e nodo j. Per la traiettoria, quindi, il robot seguirà i nodi del grafo, come in figura.

#### Grafo di visibilità

Il procedimento è simile alla seconda parte di voronoi. Una volta generati i collegamenti tra vertici, bisogna eliminare quelli non visibili, ovvero quei vertici il cui collegamento interseca un ostacolo, con l’aiuto di obstacleIntersect. Si puó notare che gli ostacoli non subiscono intersezioni. È ovvio che il robot camminerà per forza lungo il muro dell’ostacolo, il che non è un problema se ipotizziamo che gli ostacoli siano misurati per eccesso.

Questi metodi che sfruttano il grafo come strumento di motion planning sono computazionalmente meno costosi nel caso di ambienti noti e soprattutto statici. Questo perché, una volta calcolato il grafo, per lo stesso ambiente quel grafo non cambia mai; cambiano soltanto i punti in cui vengono agganciati start e goal. Chiaramente, ciò che va calcolato ogni volta è il cammino minimo da percorrere sul grafo per andare da un nodo all’altro.

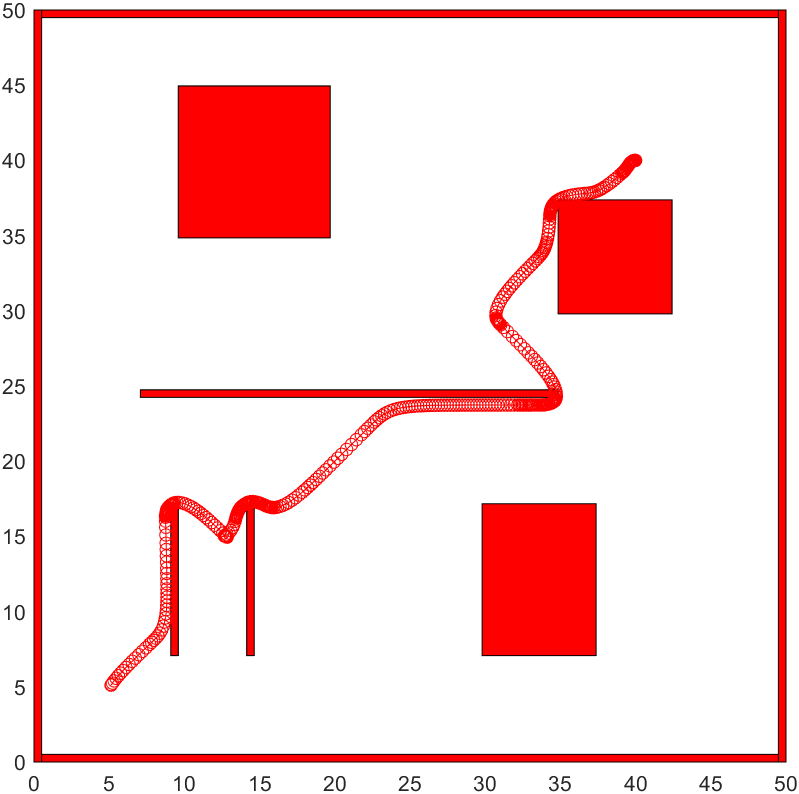
#### Decomposizione in celle con metodo trapezoidale (funzione celldecomp)

Dopo aver trovato i segmenti che congiungono in verticale ogni coppia vertice-ostacolo, ho trovato i nodi del grafo calcolando i punti medi di questi segmenti verticali, per poi collegare i nodi tra di loro e numerarli in base alla loro posizione nell’array dei vertici. Nel grafo finale, peró, ci vanno solo quei collegamenti che non intersecano ostacoli, né segmenti vertice-ostacolo.

In sostanza, con i tre metodi precedenti si avrà alla fine un grafo, a cui verranno aggiunti i due nodi di pstart e pgoal. Lo step successivo è quello di trovare un cammino minimo tra il nodo iniziale e finale. Come output alla funzione quindi, questi tre metodi generano una sequenza di nodi (a cui è abbinata un’informazione sulle coordinate cartesiane) che verranno dati in pasto alla funzione che genera le traiettorie. Qui il percorso a puntini rosso è la traiettoria a distanza minima con interpolazione polinomiale tra un nodo e l’altro.

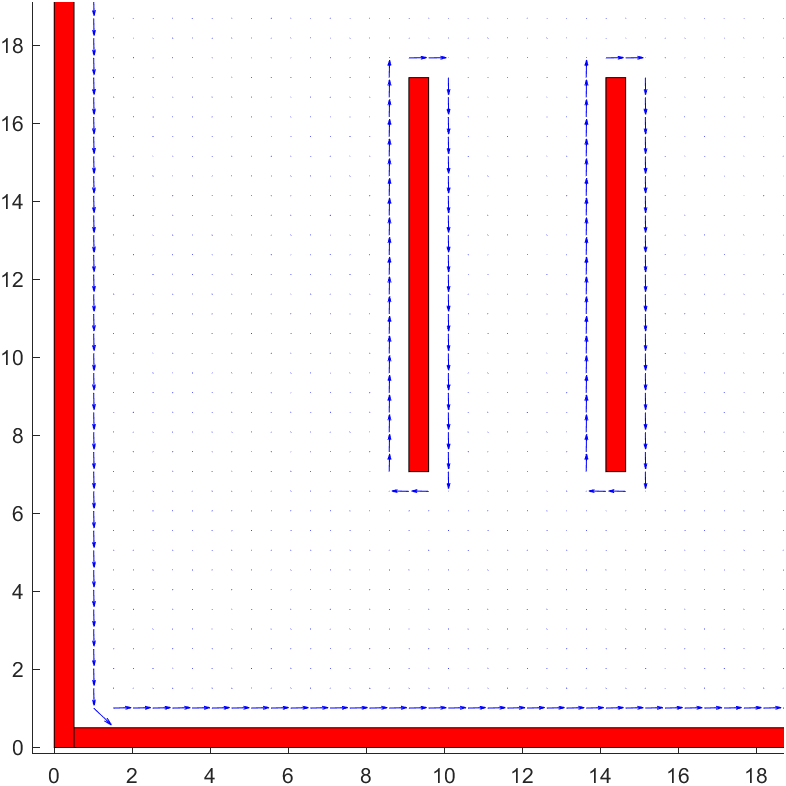
#### Potenziali artificiali discreti

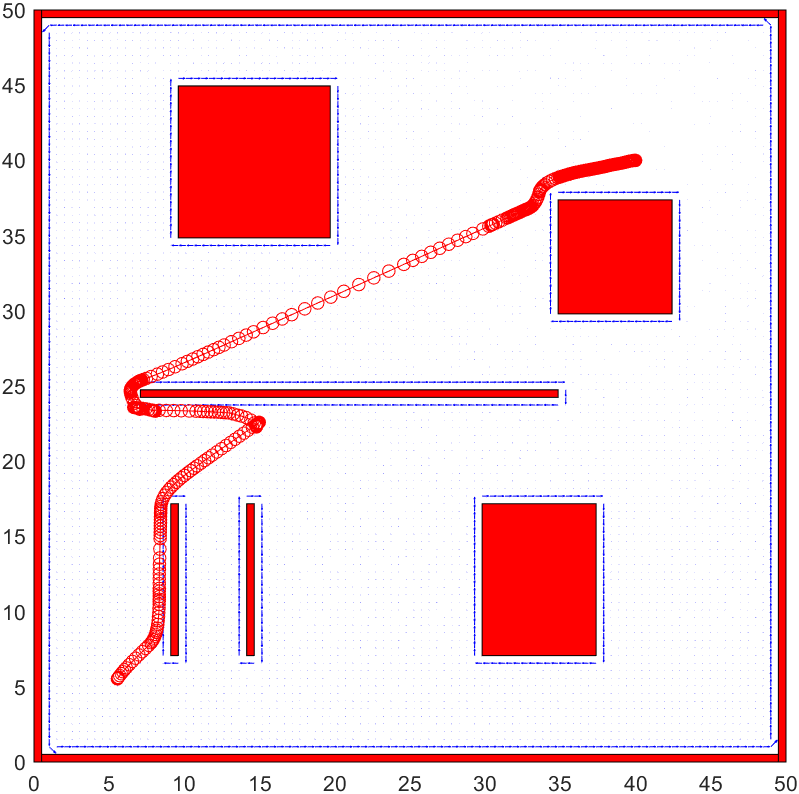
Per implementare questo metodo ho riempito una griglia apposita cspace con i rispettivi numeri che indicano la distanza della casella da quella di goal con un procedimento iterativo: scandisco lato per lato un quadrato di dimensione inizialmente 3x3 e poi via via ad aumentare, fino a 100, inserendo l’indice progressivo se la cella nella griglia binaria è libera e il valore inf se è occupata. In figura si puó vedere il campo potenziale artificiale discreto che ha un punto di minimo in (40,40): ogni punto della meshgrid ha un valore di Z pari a un numero naturale tra 0 e 100, a seconda della distanza dal punto di goal, e un valore infinito se in quella cella è presente un ostacolo.

Dopo avere creato il campo potenziale artificiale, cerco il percorso che mi porti al punto di goal spostandomi sempre all’interno di un intorno 3x3 di una casella verso il minimo di quell’intorno, simulando cosí in maniera discreta l’inseguimento di un antigradiente che mi porti verso il punto di minimo. Chiaramente ci possono essere problemi di minimo locale: possiamo arenarci su un risultato sbagliato e/o andare in deadlock. Per evitare ció, nel momento in cui il robot arriva al bordo di un ostacolo, inizia a girarci intorno, finché non riesce a trovare una casella che con piú probabilitá lo porterà verso il goal senza sbattere contro altri ostacoli. Questa modalità è implementata nella funzione circum.

#### Potenziali artificiali

In veste di funzione continua, il campo potenziale totale avrà questo aspetto. È la somma tra il potenziale repulsivo sugli ostacoli e quello attrattivo sul goal (in alto a destra in figura). L’implementazione in Matlab è vettorizzata. Sfruttando le proprietá della griglia binaria, ho usato la routine Matlab bwdist, che calcola la distanza di ogni punto della griglia dal piú vicino punto occupato. Di conseguenza, otterrò come potenziale repulsivo una matrice 100x100 che è inversamente proporzionale ai valori che si trovano in d: piú mi avvicino a un ostacolo, piú il potenziale repulsivo aumenta, fino ad arrivare al massimo possibile proprio sull’ostacolo. Il potenziale attrattivo sará anch’esso una matrice 100x100, calcolato come un paraboloide con un punto di minimo nel punto di goal. Di seguito, dopo aver sommato gli antigradienti, bisogna cercare un percorso che porti al punto di goal seguendo proprio l’antigradiente, pesando l’avanzamento con un coefficiente k.

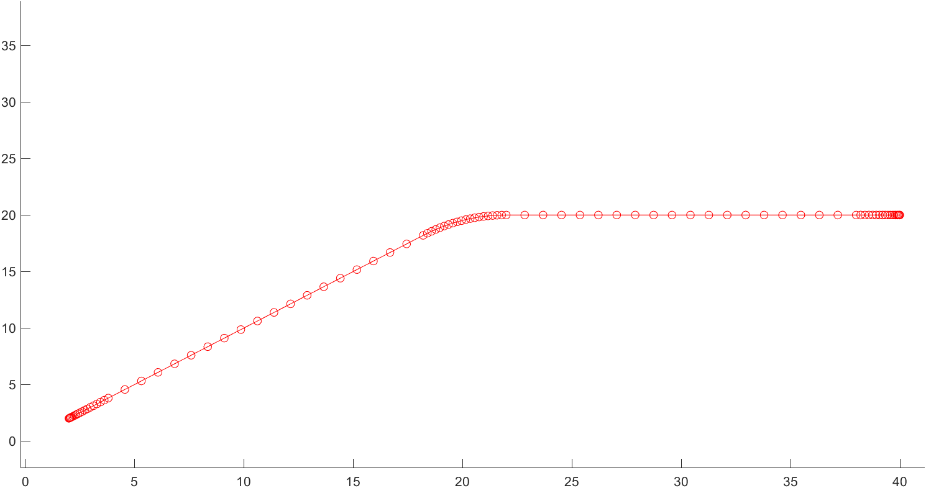
Qui uno zoom sul plot dell’antigradiente, che lontano dagli ostacoli, punta semplicemente verso il goal; qui le freccette sono impercettibili perché molto meno intense e quindi meno lunghe rispetto a quelle addosso agli ostacoli, che invece puntano in senso antiorario rispetto all’ostacolo stesso, al fine di girarci intorno. È una possibile implementazione dei potenziali vorticosi, che aggirano il problema dei minimi locali.

A destra il percorso compiuto dal punto (5,5) di start al punto (40,40) di goal. Da notare come il la traiettoria scorra lungo il muro degli ostacoli una volta che si avvicina oltre una certa soglia di distanza. Inoltre, degno di nota è il fatto che il percorso con i potenziali discreti sembra piú “denso”, nonostante entrambi abbiano la stessa “capillarità” (entrambi i potenziali sono matrici 100x100). Ma, chiaramente, l’avanzamento nei potenziali discreti è unitario, non si possono “saltare” delle celle, cosa che invece avviene nel secondo metodo, in cui interviene il peso k, che velocizza o rallenta opportunamente la traiettoria a seconda di dove ci troviamo rispetto a un qualsiasi ostacolo.

### Problema 3: Generazione Traiettoria

È responsabile per questo compito lo script trajectoryGen. Definita una sequenza di n punti e una sequenza di n istanti temporali di visita o passaggio nei punti, verranno generati dei vettori di dimensione 2\*n: per ogni punto ci sará un polinomio interpolante e un tratto rettilineo al punto successivo per quanto la riguarda la traiettoria continua, e per la traiettoria a spezzate ci sará un tratto di sola rotazione e un tratto di sola traslazione.

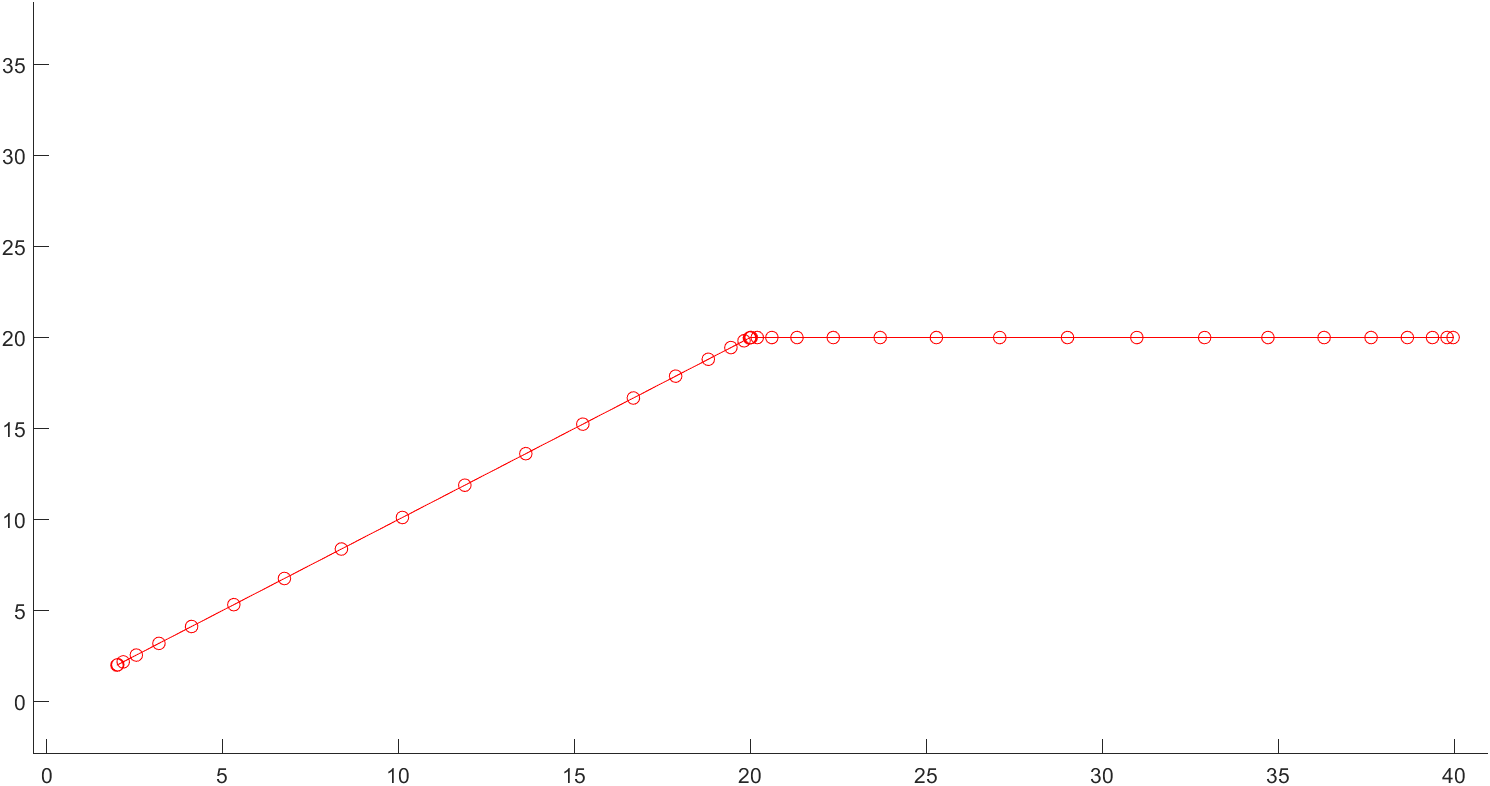
#### Traiettoria per spezzate interpolate da polinomi cubici

Innanzitutto, fisso , che indica la quantitá di tempo che separa l’inizio del tratto interpolante dall’istante in cui effettivamente andrebbe visitato il punto su cui sto interpolando le due spezzate. Nel tratto rettilineo da a impongo una velocitá costante sia su x che su y, e di conseguenza ne deriva che la posizione varia come una retta. In sostanza, si tratta di un profilo di velocitá trapezoidale, ma con un occhio di riguardo al fatto che mi voglio fermare solo all’inizio e alla fine. Dopo di che, da a c’è il polinomio interpolante, che ha derivata iniziale uguale alla velocitá del tratto rettilineo precedente (il primo e l’ultimo tratto rettilineo sono fittizi e hanno velocitá nulla) e derivata finale uguale alla velocitá di quello successivo.

In figura è rappresentato una traiettoria che passi per i punti (2,2), (20,20), (40,20). Si puó ben notare che il secondo punto non viene effettivamente toccato, ma bypassato dal polinomio interpolante. Inoltre, si vede che all’inizio e alla fine il robot è a velocitá nulla, motivo per cui i samples si infittiscono.

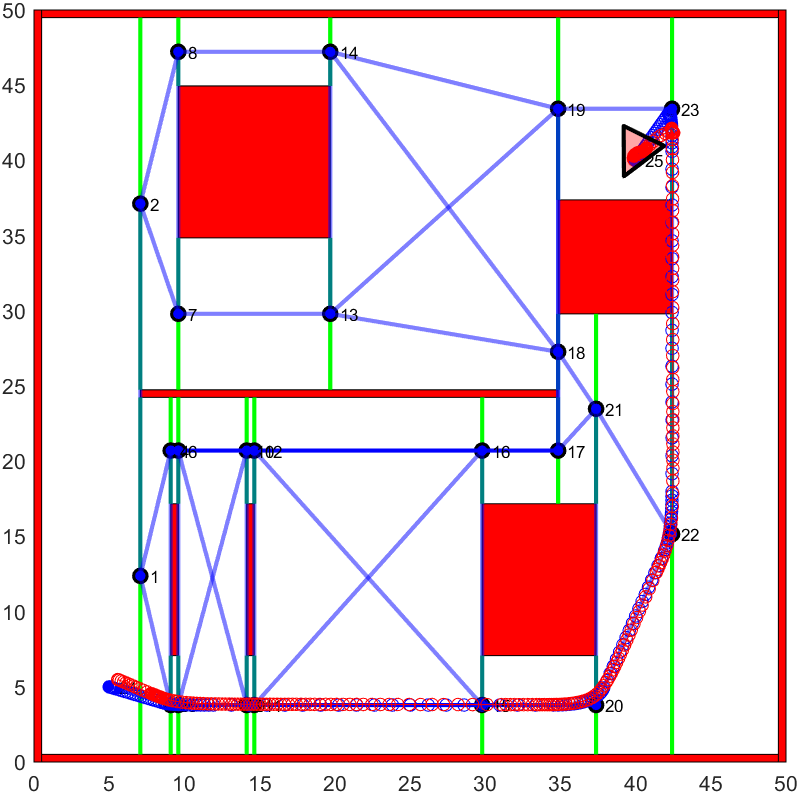
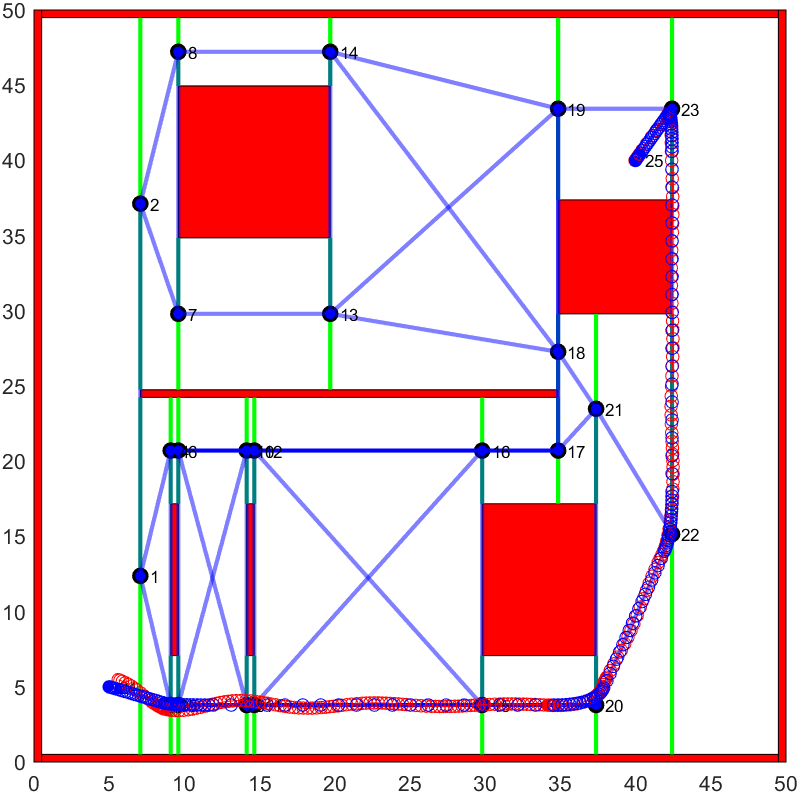
#### Traiettoria per spezzate

Per questa tipologia di traiettoria ho sfruttato un profilo di velocitá polinomiale, con partenza da velocitá nulla e arrivo a velocitá nulla. Ho calcolato simbolicamente . Si nota in figura che il robot visita, fermandosi, tutti e tre i punti.

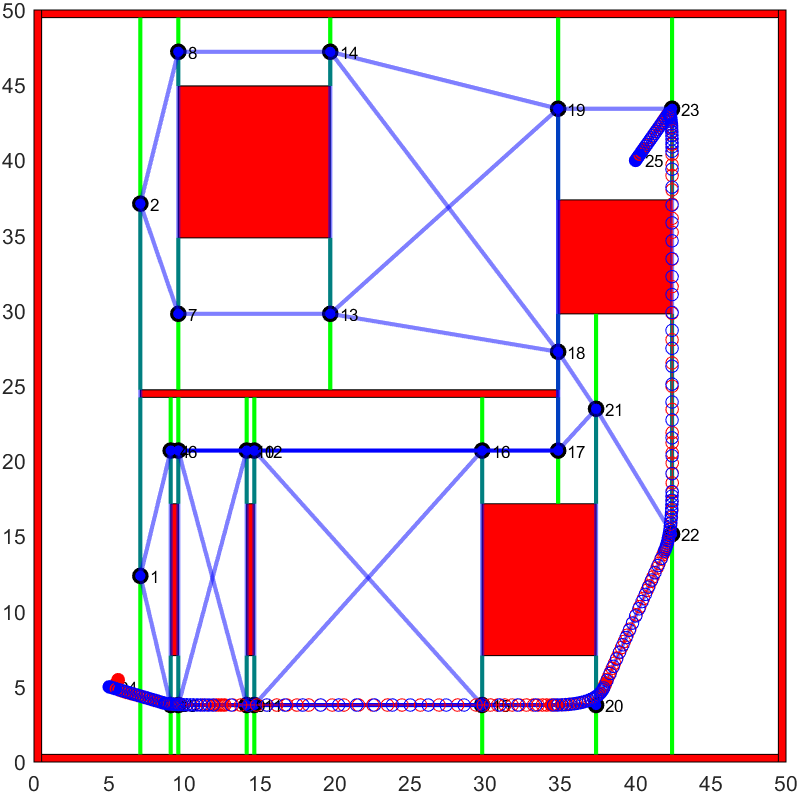


### Problema 4: Controllo della traiettoria

Lo step finale è quello di, una volta generata, controllare la traiettoria con degli algoritmi di controllo. A tal fine ho sfruttato la trajectory tracking per inseguire la traiettoria principale fino a un intorno del punto di goal, e per ultimo la posture regulation per poter esattamente arrivare nel punto desiderato.

Nella figura sulla sinistra il metodo di controllo per feedback linearization, che insegue la traiettoria con un punto b poco distante dal centro del robot. In blu la traiettoria da inseguire, in rosso il percorso effettivamente seguito dal robot. Vediamo che il transitorio non è molto lungo, ma si nota particolarmente alla fine che l’inflessione è leggermente spostata rispetto alla traiettoria reale, il che dimostra che questo metodo è piú stabile sul transitorio, ma meno preciso, appunto perché il punto del robot controllato in realtà non è esattamente il suo centro, ma un punto b poco distante dal centro. È, inoltre, l’unico metodo di controllo con cui è possibile utilizzare il percorso a minima distanza, come mostrato nelle figura di seguito. Qui la traiettoria viene seguita con piú precisione, anche perché non è presente il polinomio interpolante tra un punto e l’altro. Il robot arriva in un punto, si ferma, ruota e riparte in linea retta verso il punto successivo.

Un altro metodo di controllo è invece quello nonlineare (sulla sinistra), che è stabile globale e tende asintoticamente alla traiettoria desiderata, ma come si puó ben vedere in basso nella figura, ha un transitorio molto lungo, con un moto oscillatorio smorzato piuttosto prolungato.

La performance migliore per questo particolare percorso è del metodo di controllo linearizzato (sulla destra) che considera dominanti i termini lineari dell’espansione in serie taylor della funzione errore dell’algoritmo di controllo. Il transitorio è piuttosto breve, e segue la traiettoria con precisione.