Università degli studi di Palermo

Ingegneria Informatica LM-32

Robotica

CleanBot

Membri del gruppo:

Ajello Gabriele

Sciarrabba Enrico Maria

Solaro Giuliano

Sommario

1. Introduzione

Il progetto CleanBot è finalizzato alla creazione di un robot per la pulizia di ambienti interni. L'azione principale del robot è quella di muoversi in uno spazio delimitato da mura evitando gli ostacoli, fissi o mobili che siano, e riuscendo a raggiungere tutti gli spazi liberi dell'ambiente.

Il movimento è gestito tramite:

- Sensori, che consentono al robot di riconoscere gli ostacoli
- EKF-Slam, che consente al robot di mappare l'intero ambiente e di tenere traccia della posizione stimata in modo da poter correggere la traiettoria quando necessario
- Algoritmo ottimizzato di ricerca, che consente di trovare dei percorsi minimi sulla mappa per poter gestire la copertura di tutti gli spazi liberi dell'ambiente e il ritorno alla base una volta terminata la pulizia.

Per semplicità di implementazione si è deciso supporre che il robot lavori in un mondo a griglia. In questo modo è strato possibile discretizzare i movimenti e rendere più efficiente la creazione della mappa e la ricerca di percorsi minimi su questa. Tuttavia, l'algoritmo EKF-Slam è stato implementato in maniera classica, quindi utilizzabile per mappare qualsiasi tipo di ambiente. Inoltre, è stato necessario creare manualmente degli oggetti che il robot riconosce come landmark ai fini dell'implementazione dell'algoritmo Slam.

Video allegati:

- Video\cleanbot.mp4: mostra la simulazione con il robot che pulisce l'intero ambiente
- Video\mappa.mp4: mostra la simulazione e grafico della mappa
- Video\slam_cella.mp4: mostra la simulazione e il grafico dello Slam aggiornato cella per cella
- Video\slam_continuo.mp4: mostra la simulazione e il grafico delle Slam aggiornato in maniera continua (ogni 32ms)

2. Struttura Generale

Dentro la cartella principale ‘CleanBot’ si trovano 5 sottocartelle:

- controllers, contenente i file del controllore del robot
- libraries, non contiene alcun dato utile ai fini del progetto
- plugins, non contiene alcun dato utile ai fini del progetto
- protos, contenente i file relativi all’aspetto del robot (Create.proto) e dell’ambiente (Home.proto) oltre alle textures utilizzate nel progetto
- worlds, contenente il file del mondo di webots (cleanbot.wbt) dal quale viene avviata la simulazione

2.1 Protos

Il meccanismo PROTO consente di specificare dei nodi in file esterni da aggiungere, in fase di simulazione, al mondo di webots in modo da estendere i nodi già esistenti. Ai fini del progetto sono stati utilizzati per creare un albero di nodi semplice e rendere più immediata la comprensione delle parti costituenti del mondo.

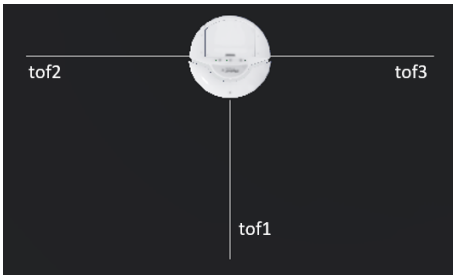
2.1.1 Create.proto

File relativo al robot utilizzato per il progetto: descrive tutti i nodi di cui questo è costituito.
La forma finale del robot è quella mostrata in figura.

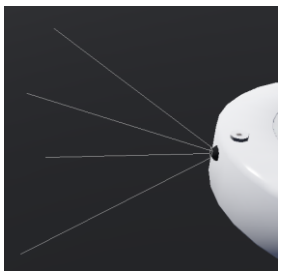
Tra i nodi principali possiamo distinguere:

- ‘right wheel motor’ e ‘left wheel motor’, ovvero i motori rotazionali rispettivamente della ruota sinistra e destra utilizzati per gestire il movimento del robot.
  I movimenti concessi sono due: andare avanti e girare di 90° a destra o a sinistra. Questo schema di movimenti è il migliore che si adatti al mondo a griglia.
- ‘tof1’, ‘tof2’ e ‘tof3’: questi sono i sensori di distanza Time Of Flight che misurano la distanza da un oggetto emettendo un raggio laser o infrarossi che viene riflesso da un oggetto e ritorna alla sorgente. In base al tempo impiegato viene calcolata la distanza. Sono inoltre sensori molto precisi: l’errore massimo misurato in simulazione è stato di circa 3cm.

Nel progetto sono stati utilizzati per riconoscere gli ostacoli e permettere al robot di evitarli. Proprio per questo sarebbe stato possibile utilizzare un qualsiasi sensore di distanza, in quanto non è stata data troppa importanza al valore di misurazione.
'tof1' è orientato in avanti e consente di rilevare ostacoli frontali, 'tof2' è orientato a destra e consente di rilevare ostacoli alla destra del robot, 'tof3' è orientato a sinistra e consente di rilevare ostacoli alla sinistra del robot.

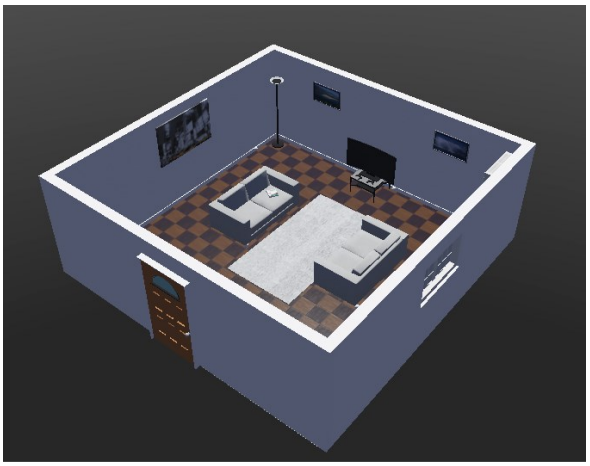


- 'camera', ovvero la telecamera che consente di riconoscere i landmark usati nell'algoritmo EKF-Slam. Il suo utilizzo è necessario anche per la misura della distanza di un landmark dal robot in modo da poter prima posizionarlo nella mappa e, successivamente, sfruttarlo per correggere l'errore sulla posizione del robot.
  L'angolo di apertura della telecamera utilizzata nel progetto è di 90° e l'immagine restituita ha una dimensione di 128x128 pixel. È stata inoltre equipaggiata di riconoscimento degli oggetti in base al colore: è in grado di riconoscere oggetti di colore bianco fino a una distanza massima di 3 metri.



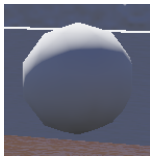
2.1.2 Home.proto

File relativo all'ambiente utilizzato per il progetto: descrive tutti i nodi di cui questo è costituito.
La forma finale dell'ambiente è quella mostrata in figura. È importante notare come il robot sia in grado di muoversi autonomamente in qualsiasi mondo a griglia. Di conseguenza anche se venisse modificato l'ambiente con aggiunzione o rimozione di ostacoli o modificata l'intera mappa, il robot riuscirebbe comunque a portare a termine il suo compito.

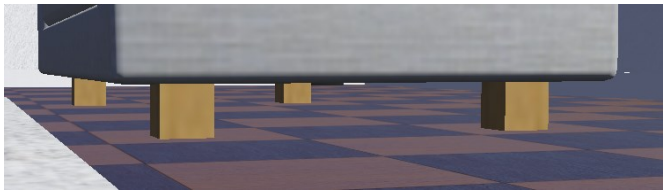



Tra i nodi più importanti possiamo distinguere:

- Landmark: elementi di forma sferica inseriti negli angoli della casa e su alcuni ostacoli. Si è deciso di inserirli in questi punti per simulare un comportamento reale in cui il robot, non avendoli a disposizione, sceglierebbe proprio di localizzarsi in base a spigoli o ostacoli particolari. Questi vengono riconosciuti e localizzati dal robot attraverso l'algoritmo EKF-Slam e successivamente utilizzati per poter correggere la sua posizione. Il colore di riconoscimento dei landmark è bianco (rgb(1,1,1)) ed è proprio per questa caratteristica che rende la camera capace di distinguerli rispetto al resto dell'ambiente



Tutti gli altri nodi sono solamente ostacoli inseriti nella mappa per mostrare il comportamento del robot nelle diverse situazioni. Possiamo infatti distinguere delle aree principalmente sgombre, come di fronte la porta d'ingresso, in cui il comportamento del robot sarà semplicemente quello di muoversi e localizzarsi tramite Slam, e aree ricche di ostacoli, come sotto i divani, in cui saranno ampiamente sfruttati i sensori.

2.2 Worlds

2.2.1 cleanbot.wbt

L'unico mondo webots utilizzato per il progetto è 'cleanbot.wbt' che unisce i nodi .proto precedentemente descritti ai nodi di base in modo da renderizzare l'ambiente e il robot e rendere possibile la simulazione.

3. Controller

Premessa

Per il corretto funzionamento dell'algoritmo è necessario installare due pacchetti di python:

- numpy
- matplotlib

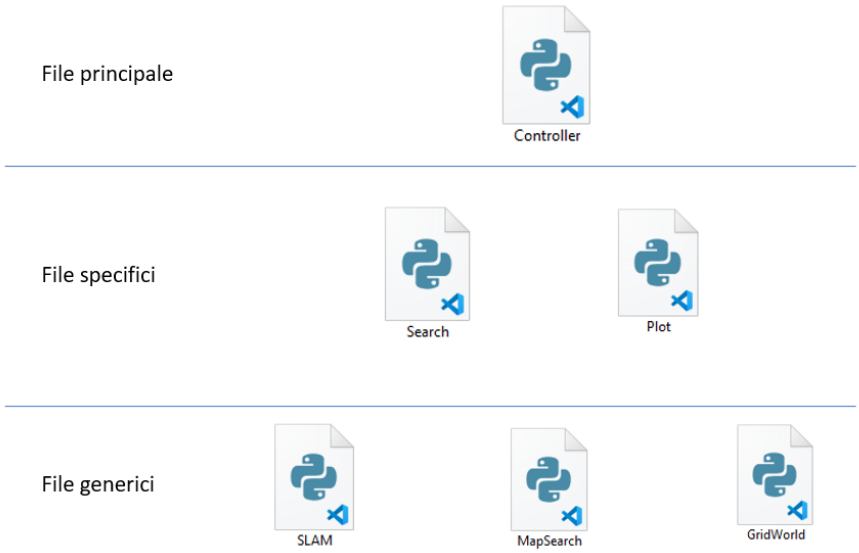
Per farlo basta inserire da terminale il comando: 'pip install <nome_pacchetto>'

Il controllore del robot è interamente scritto in python e implementa i comportamenti di movimento, localizzazione tramite EKF-Slam e ricerca di percorsi nella mappa attraverso un algoritmo ottimizzato di ricerca.

È costituito da 6 file:

- Controller.py
- GridWorld.py
- MapSearch.py
- Plot.py
- Search.py
- SLAM.py

Possiamo suddividere tutti questi file in tre gruppi:



3.1 File generici

I File generici implementano delle classi e sono quindi da vedere come delle librerie che mettono a disposizione metodi di utilità. Questi file potrebbero quindi essere riutilizzati, da soli, per altri scopi poiché risultano indipendenti da tutti file gli altri codici del progetto.

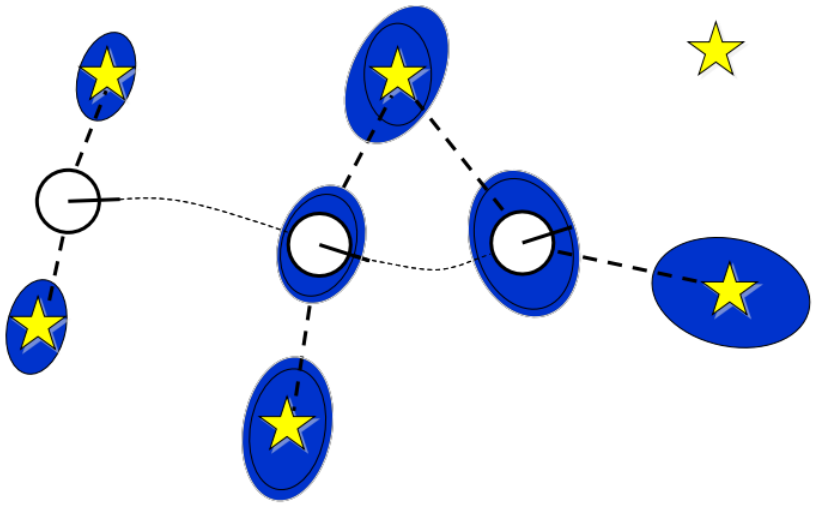
3.1.1 Slam.py -- Simultaneous Localization and Mapping (SLAM)

La localizzazione e mappatura simultanea (SLAM) si pone il problema di stimare, in tempo reale, la struttura del mondo circostante al robot e permettergli di localizzarsi in questo.
SLAM coinvolge un agente in movimento (ad esempio un robot) che ha almeno un sensore in grado di raccogliere informazioni sull'ambiente circostante. Si compone di tre operazioni di base, che vengono reiterate a ogni intervallo di tempo:

1. Il robot si muove e raggiunge un nuovo punto di vista della scena. A causa di rumore ed errori, questi movimenti aumentano l'incertezza sulla sua posizione. Una soluzione automatizzata richiede un modello matematico per modellare questo movimento: si parla di **modello del movimento**.
2. Il robot scopre caratteristiche interessanti nell'ambiente chiamate landmark, che devono essere incorporate nella mappa. A causa di errori nei sensori, la posizione di questi punti di riferimento sarà incerta. Inoltre, poiché la posizione del robot è già incerta, queste due incertezze si combinano tra loro. Una soluzione automatizzata richiede un modello

matematico per determinare la posizione dei punti di riferimento nella scena utilizzando i dati ottenuti dai sensori: si parla di **modello inverso dell'osservazione**.

3. Il robot osserva punti di riferimento che erano stati precedentemente mappati, e li usa per correggere sia la sua posizione che quella di tutti i punti di riferimento nello spazio. In questo caso, quindi, entrambe le incertezze diminuiscono. Una soluzione automatizzata richiede un modello matematico per prevedere i valori della misurazione dal punto di riferimento previsto e dalla localizzazione del robot: si parla di **modello diretto dell'osservazione**.



Con questi tre modelli siamo in grado di costruire un algoritmo per SLAM. È inoltre necessario effettuare delle stime in modo da propagare correttamente le incertezze ogni volta che si verifica una delle tre situazioni di cui sopra. Ai fini del progetto è stato utilizzato il **filtro di Kalman esteso (EKF)**.

Riportiamo sotto i modelli precedentemente citati.

- Modello del movimento: $R = g(R, u, n)$
- Modello di osservazione diretta: $z_i = h(R, S, L_i)$
- Modello inverso: $L_i = k(R, S, z_i)$

Nelle definizioni, *R* identifica il vettore che descrive lo stato del robot, mentre *u* ed *n* identificano rispettivamente il segnale di controllo e l'errore sul modello.
Riguardo ai modelli di osservazione, L_i identifica il vettore che descrive l'i-esimo landmark e y_i la sua misura. Tipicamente le funzioni *h* e *k* sono una l'inversa dell'altra.

EKF-SLAM

Nell'EKF-SLAM, la mappa è un grande vettore che impila posizioni del robot e dei landmark, ed è modellata attraverso gaussiane. Questa mappa, solitamente chiamata mappa stocastica, è mantenuta aggiornata dall'EKF attraverso i processi di previsione (il robot si muove) e di correzione (i sensori osservano i punti di riferimento nell'ambiente che erano stati

precedentemente mappati). Per ottenere una vera esplorazione, EKF è arricchito con un ulteriore passaggio di inizializzazione dei punti di riferimento, in cui quelli appena scoperti vengono aggiunti alla mappa. L'inizializzazione di un punto di riferimento viene eseguita invertendo la funzione di osservazione e utilizzando questa e i suoi Jacobiani per calcolare, dalla posizione del sensore e dalle misurazioni, lo stato del punto di riferimento osservato e le sue co-varianze con il resto dei landmark della mappa. Queste relazioni vengono quindi aggiunte al vettore di stato e alla matrice delle covarianze.

La mappa è un grande vettore che memorizza gli stati (posizioni) del robot e dei landmark.

$$x = \begin{bmatrix} R \\ M \end{bmatrix} = \begin{bmatrix} R \\ L_1 \\ \vdots \\ L_n \end{bmatrix}$$

Dove *R* è l'attuale stato del robot ed $M = (L_1, L_2, \ldots, L_n)$ è lo stato dei landmark visti fino a quell'istante.

Nell'EKF, questa mappa è modellata da una variabile gaussiana utilizzando la media e la matrice delle covarianze del vettore di stato, che denotiamo rispettivamente come μ e $\sum$. Dati N landmark, μ ha una lunghezza di 2N+3 e $\sum$ una dimensione di (2N+3)x(2N+3)

$$\mu = \begin{bmatrix} \bar{R} \\ \bar{M} \end{bmatrix} = \begin{bmatrix} \bar{R} \\ \overline{L_1} \\ \vdots \\ \overline{L_n} \end{bmatrix} \qquad \sum = \begin{bmatrix} \sum_{RR} & \sum_{RM} \\ \sum_{MR} & \sum_{MM} \end{bmatrix}$$

Inizializzazione EKF-SLAM

La mappa, all'inizio, non contiene alcun landmark. Inoltre, la posizione iniziale del robot è solitamente considerata l'origine della mappa che verrà costruita (punto [0,0]). Dunque:

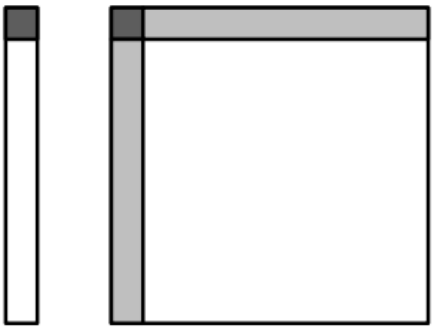
$$\mu = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \sum = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Predizione

Il passo di previsione EKF è normalmente scritto come:

$$\mu = g(\mu, u, 0) \qquad \sum = G \sum G^T + R$$

Sono mostrate sotto le parti aggiornate della mappa in base al movimento del robot: la media è rappresentata dalla barra a sinistra. La matrice delle covarianze dal quadrato a destra.



La matrice $G = \frac{dg(\mu,u)}{d\mu}$ è la matrice Jacobiana del modello di movimento e ha dimensione 3x3.
Nello SLAM solo una parte dello stato è variabile nel tempo: il robot, che si muove. Quindi, poiché la maggior parte della mappa è invariante rispetto al movimento del robot, nella fase di previsione si creano matrici giacobiane sparse con la seguente struttura

$$G = \begin{bmatrix} \frac{dg_R}{dR} & 0 \\ 0 & I \end{bmatrix}$$

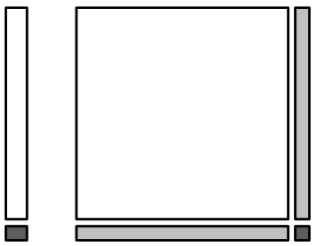
Dove $\frac{dg_R}{dR}$ è la matrice 3x3 sopra indicata e *I* è una matrice identità di dimensione 2Nx2N. La dimensione totale di G è quindi (2N+3)x(2N+3)

Inizializzazione dei landmark

L'inizializzazione di un landmark avviene quando il robot scopre punti di riferimento non ancora mappati e decide di incorporarli nella mappa. In quanto tale, questa operazione comporta un aumento della dimensione del vettore di stato. Questa è una delle principali differenze tra il semplice EKF e l'EKF-SLAM.
L'inizializzazione del punto di riferimento è semplice nei casi in cui il sensore fornisce informazioni su tutti i gradi di libertà del nuovo punto di riferimento. Quando ciò accade, basta invertire la funzione di osservazione *h* per calcolare lo stato del nuovo landmark L_{n+1} a partire dallo stato del robot *R*, da quello del sensore *S* e l'osservazione y_{n+1} che costituisce il modello di osservazione inversa di un punto di riferimento.

$$L_{i+1} = k(R, S, z_{i+1})$$

L'immagine sotto mostra la nuova mappa dopo l'inizializzazione di un nuovo punto di riferimento. Le parti in grigio corrispondono alla media e alla covarianza del punto di riferimento (grigio scuro) e alle varianze incrociate tra il punto di riferimento e il resto della mappa (grigio chiaro).



Dal punto di vista delle formule utilizzate, per prima cosa si calcola la posizione stimata del landmark e le matrici Jacobiane della funzione

$$L_{i+1} = k(R, S, z_{i+1}) \qquad k_R = \frac{dk}{dR} \qquad k_{z_{i+1}} = \frac{dk}{dz_{i+1}}$$

Quindi si calcola la covarianza del punto di riferimento e la sua varianza incrociata con il resto della mappa

$$\sum_{LL} = k_R \sum_{RR} k_R^T + k_{z_{i+1}} R k_{z_{i+1}}^T \qquad \sum_{Lx} = k_R \sum_{Rx} \qquad \sum_{Rx} = [\sum_{RR} \ \sum_{RM}]$$

Osservazione dei punti di riferimento mappati

La fase di correzione EKF è descritta dalle seguenti formule:

$$K = \sum H^T (H \sum H^T + Q)^{-1}$$

$$\mu = \mu + K(z - h(\mu))$$

$$\sum = (I - KH) \sum$$

Dove $H = \frac{dh(\mu)}{d\mu}$ è la matrice Jacobiana, $\sum$ è la matrice delle covarianze della misura.

La prima formula consente di calcolare la matrice *K* detta il **guadagno di Kalman**. Le altre due comportano invece l'aggiornamento di μ e $\sum$, quindi della mappa del filtro.

Nello SLAM, le osservazioni si verificano quando un punto di riferimento già osservato dal robot viene rivisto. Il risultato della sua elaborazione è una parametrizzazione geometrica del landmark nello spazio di misura: il vettore y_i.

Le osservazioni dei punti di riferimento vengono elaborate nell'EKF una per una. Ognuna di queste dipende solo dalla posizione o dallo stato del robot *R*, dallo stato del sensore *S* e, assumendo di riosservare il punto di riferimento i-esimo, dallo stato precedente del punto di riferimento L_i. Proprio per questo, per ogni landmark osservato, si ottiene una funzione di osservazione individuale (modello di osservazione) che non dipende da nessun altro punto di riferimento se non da L_i stesso. Quindi anche il Jacobiano H è una matrice sparsa:

$$H = [H_R \ \ 0 \ \ \dots \ \ 0 \ \ H_{L_i} \ \ 0 \ \ \dots \ \ 0]$$

con

$$H_R = \frac{dh_i(\bar{R}, S, \bar{L_\iota})}{dR} \qquad H_{L_i} = \frac{dh_i(\bar{R}, S, \bar{L_\iota})}{dL_i}$$

Implementazione EKF-SLAM

Nel progetto lo Slam è stato implementato come una classe python che consente di istanziare un oggetto e di richiamare tutti i metodi per necessari al suo funzionamento.

Come già visto dalla teoria descritta precedentemente, le operazioni principali da effettuare sono quattro:

- Inizializzazione del vettore di stato e della matrice di covarianza;
- Predizione;
- Aggiunzione di un landmark precedentemente non osservato;
- Correzione.

Sono stati quindi creati quattro metodi che svolgono tali compiti. In aggiunta è stato implementato un metodo di controllo che verifica se un landmark è già stato osservato.

Nell'implementazione si comincia definendo i modelli di movimento, misura e misura inversa.

Per quanto riguarda il modello di movimento, è stato utilizzato un semplice moto rettilineo uniforme, sia per il moto lineare, per aggiornare le posizioni x e y, che per il moto angolare, per aggiornare l'angolo θ.

Per l'individuazione dei landmark è stata invece utilizzata una telecamera, che, come implementato nel simulatore webots utilizzato nella realizzazione del progetto, restituisce la posizione dell'oggetto osservato. È stata sfruttata questa caratterista per ottenere le coordinate del landmark relative al robot e, da queste, calcolare la misura della distanza (**range**) e dell'angolo di inclinazione (**bearing**) dell'oggetto osservato dal robot.

Inizializzazione dell'algoritmo

Nella fase di inizializzazione vengono creati il vettore di stato *R* e la matrice di covarianza ∑. Questo è effettuato dal costruttore **__init__** che viene chiamato automaticamente quando viene istanziato un oggetto di tipo SLAM.

$$\mu = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \sum = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Il metodo, inoltre, inizializza le matrici d'errore sul modello di movimento e sul modello di misura. Nel progetto gli errori sono stati utilizzati i seguenti valori di errore:

$$errore\ sul\ movimento = \begin{bmatrix} err\_mov_x & 0 & 0 \\ 0 & err\_mov_y & 0 \\ 0 & 0 & err\_mov_\theta \end{bmatrix} = \begin{bmatrix} 0.001 & 0 & 0 \\ 0 & 0.001 & 0 \\ 0 & 1 & 0.0001 \end{bmatrix}$$

$$errore\ sulla\ misura = \begin{bmatrix} ere\_mes_{dist} & 0 \\ 0 & err\_mes_\theta \end{bmatrix} = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$$

Dove i valori sono indicati in metri e si riferiscono all'errore che del sistema per ogni intervallo di tempo dt di 32ms.

Codice:

```
11      def __init__(self):
12          self.mu = np.zeros((3, 1))
13          self.sigma = np.zeros((3,3))
14
15          self.errorMovementX = 0.001
16          self.errorMovementY = 0.001
17          self.errorMovementTeta = 0.0001
18
19          self.errorMeasurementD = 0.1
20          self.errorMeasurementTeta = 0.1
21
22          self.R = [self.errorMovementX**2, self.errorMovementY**2, self.errorMovementTeta**2]
23          self.Q = np.array([[self.errorMeasurementD**2, 0],
24                             [0, self.errorMeasurementTeta**2]])
```

Predizione

Per effettuare predizione è stato creato il metodo **predictionMove**. Questo prende in input un vettore contenente le informazioni riguardanti il movimento. Nello specifico le informazioni sono rappresentate tramite un vettore di tre elementi:

$$[linearVelocity\ (v), \qquad angularVelocity\ (w), \qquad timeInterval\ (t)]$$

Per la particolare implementazione del progetto, il robot può, in un determinato istante di tempo, solamente ruotare o andare avanti. Di conseguenza i moti traslatorio e rotatorio sono stati separati: ci si aspetta di avere una velocità angolare nulla in caso di velocità lineare non nulla. Il metodo, inoltre, inizializza le matrici d'errore sul modello di movimento e sul modello di misura. Riportiamo di seguito le espressioni delle matrici per la descrizione del moto

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} + \begin{bmatrix} t * v * \cos(\theta_t) \\ t * v * \sin(\theta_t) \\ t * w \end{bmatrix}$$

Riguardo l'errore da aggiungere alla matrice ∑, data la previsione, bisogna calcolare lo Jacobiano della matrice sopra descritta:

$$G = \begin{bmatrix} 1 & 0 & -t * v * \sin(\theta) \\ 0 & 1 & t * v * \cos(\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

Codice:

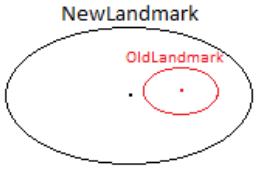
```
35      def predictionMove(self, motion):
36          self.mu[0,0] += motion[0] * motion[2] * np.cos(self.mu[2,0])
37          self.mu[1,0] += motion[0] * motion[2] * np.sin(self.mu[2,0])
38          self.mu[2,0] += motion[1] * motion[2]
39          while self.mu[2,0] >  np.pi: self.mu[2,0] -= 2*np.pi
40          while self.mu[2,0] < -np.pi: self.mu[2,0] += 2*np.pi
41
42          g = np.array([ [1, 0, -motion[0]*motion[2]*np.sin(self.mu[2,0])], [0, 1, motion[0]*motion[2]*np.cos(self.mu[2,0])], [0, 0, 1] ])
43          G = np.vstack(( np.hstack(( g , np.zeros((3,len(self.mu)-3)) )) , np.hstack(( np.zeros((len(self.mu)-3,3)) , np.eye(len(self.mu)-3) )) ))
44
45          self.sigma = np.dot( np.dot( G , self.sigma ) , np.transpose(G) )
46          self.sigma[0,0] += self.R[0]
47          self.sigma[1,1] += self.R[1]
48          self.sigma[2,2] += self.R[2]
```

Controllo landmark

```
59      def controlLandmark(self, measures):
60          for measure in measures:
61              exist = False
62
63              t = np.array([ [-measure[0,0]*np.sin(measure[1,0]+self.mu[2,0])] , [measure[0,0]*np.cos(measure[1,0]+self.mu[2,0])] ])
64              Tmu = np.hstack(( np.eye(2) , t ))
65              Tz = np.hstack(( np.array([ [np.cos(measure[1,0]+self.mu[2,0])] , [np.sin(measure[1,0]+self.mu[2,0])] ]) , t ))
66
67              sigmaLL = np.dot( np.dot( Tmu , self.sigma[0:3,0:3] ) , np.transpose( Tmu ) )
68              sigmaLL += np.dot( np.dot( Tz , self.Q ) , np.transpose( Tz ) )
69              sigmaLL = 3 * np.array([np.sqrt(sigmaLL[0,0]), np.sqrt(sigmaLL[1, 1])])
70
71              landmark = np.array([ self.mu[0] + measure[0,0]*np.cos(measure[1,0]+self.mu[2,0]) , self.mu[1] + measure[0,0]*np.sin(measure[1,0]+self.mu[2,0]) ])
72
73              for i in range(2, int( ((len(self.mu)-1) / 2) + 1)):
74                  if -sigmaLL[0] <= landmark[0] - self.mu[2*i-1,0] <= sigmaLL[0] and -sigmaLL[1] <= landmark[1] - self.mu[2*i,0] <= sigmaLL[1]:
75                      t = [i, measure[0,0], measure[1,0]]
76                      self.correctionLandmark(t)
77                      exist = True
78                      break
79
80              if not exist:
81                  self.addLandmark(landmark, measure)
```

Quando viene osservato un landmark, la prima cosa da capire è se questo sia già stato visto in precedenza oppure è un punto di riferimento nuovo. Per questo motivo è stato implementato un metodo che, date le misure su un landmark, capisce se è già presente o meno nel vettore di stato μ. Tale metodo è **controlLandmark**.

Per prima cosa si ricava la posizione e l'errore del landmark. Immaginando adesso la sua gaussiana, si controlla se esiste un landmark, già in μ, la cui gaussiana si trova all'interno di quella del nuovo landmark. Se si verifica questa condizione i due landmark sono gli stessi, ma ovviamente la nuova osservazione ha un errore maggiore.
Viceversa, nel caso in cui nessun landmark si trovi all'interno della gaussiana, significa che il nuovo landmark non è mai stato osservato; verrà quindi aggiunto al vettore di stato μ.



Aggiunzione landmark

Per inizializzare un nuovo landmark basta calcolare lo Jacobiano della funzione *k* prima rispetto allo stato del robot e poi rispetto alla misura del landmark. Nel nostro caso la funzione *k* è definita come

$$\begin{bmatrix} L_x \\ L_y \end{bmatrix} = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix} + \begin{bmatrix} d * \cos(\phi + \mu_\theta) \\ d * \sin(\phi + \mu_\theta) \end{bmatrix}$$

Con *d* e ϕ rispettivamente la distanza (range) e l'angolo (bearing) misurati dal robot al landmark.

Data *k* è possibile calcolare le matrici k_R e $k_{z_{i+1}}$:

$$k_R = \begin{bmatrix} 1 & 0 & -d * \sin(\phi + \mu_\theta) \\ 0 & 1 & d * \cos(\phi + \mu_\theta) \end{bmatrix} \qquad k_{z_{i+1}} = \begin{bmatrix} \cos(\phi + \mu_\theta) & -d * \sin(\phi + \mu_\theta) \\ \sin(\phi + \mu_\theta) & d * \cos(\phi + \mu_\theta) \end{bmatrix}$$

Codice:

```
91      def addLandmark(self, landmark, measure):
92          self.mu = np.vstack(( self.mu , landmark ))
93
94          t = np.array([ [-measure[0,0]*np.sin(measure[1,0]+self.mu[2,0])] , [measure[0,0]*np.cos(measure[1,0]+self.mu[2,0])] ])
95          Kr = np.hstack(( np.eye(2) , t ))
96          Kz = np.hstack(( np.array([ [np.cos(measure[1,0]+self.mu[2,0])] , [np.sin(measure[1,0]+self.mu[2,0])] ]) , t ))
97
98          sigmaLL = np.dot( np.dot( Kr , self.sigma[0:3,0:3] ) , np.transpose( Kr ) )
99          sigmaLL += np.dot( np.dot( Kz , self.Q ) , np.transpose( Kz ) )
100         sigmaLX = np.dot( Kr , self.sigma[0:3,:] )
101
102         self.sigma = np.hstack((self.sigma , np.transpose(sigmaLX) ))
103         self.sigma = np.vstack((self.sigma , np.hstack(( sigmaLX , sigmaLL ))))
```

Correzione landmark

Il metodo **correctionLandmark** effettua la correzione della posizione e dell'errore dei landmark già visti. Come spiegato precedentemente è necessario calcolare la matrice Jacobiana della funzione *h*. Questa è definita come:

$$\delta = \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \begin{bmatrix} L_x - \mu_x \\ L_y - \mu_y \end{bmatrix} \qquad q = \delta^T \delta \qquad h = \begin{bmatrix} \sqrt{q} \\ atan2(\delta_y, \delta_x) - \mu_\theta \end{bmatrix}$$

Da questo si ottiene:

$$H = \frac{1}{q} \begin{bmatrix} -\delta_x\sqrt{q} & -\delta_y\sqrt{q} & 0 & \cdots & \delta_x\sqrt{q} & \delta_y\sqrt{q} & \cdots \\ \delta_y & -\delta_x & -q & \cdots & -\delta_y & \delta_x & \cdots \end{bmatrix}$$

Codice:

```
114     def correctionLandmark(self, landmark):
115         delta = np.array([[ self.mu[2*landmark[0]-1,0] - self.mu[0,0] , self.mu[2*landmark[0],0] - self.mu[1,0] ]]).T
116         q = np.sqrt( delta.T.dot(delta) )[0,0]
117         h = np.array([[ q , mt.atan2( delta[1,0], delta[0,0] ) - self.mu[2,0] ]]).T
118         while h[1,0] >  np.pi: h[1,0] -= 2*np.pi
119         while h[1,0] < -np.pi: h[1,0] += 2*np.pi
120 
121         H_t1 = np.hstack(( np.array([-q*delta[0,0] , -q*delta[1,0] , 0]) , np.arange(0,2*landmark[0]-4,1)*0 , np.array([q*delta[0,0] , q*delta[1,0]]) , np.arange(0,len(self.mu)-2*landmark[0]-1,1)*0 ))
122         H_t2 = np.hstack(( np.array([delta[1,0] , -delta[0,0] , -(q**2)]) , np.arange(0,2*landmark[0]-4,1)*0 , np.array([-delta[1,0] , delta[0,0]]) , np.arange(0,len(self.mu)-2*landmark[0]-1,1)*0 ))
123         H  = (1 / (q**2))*np.vstack(( H_t1, H_t2 ))
124 
125         K = np.dot( np.dot(self.sigma, H.T) , np.linalg.inv( np.dot( np.dot( H, self.sigma ) , H.T ) + self.Q ))
126 
127         T = -np.dot( K , H )
128         for i in range(0, len(self.mu)): T[i,i] += 1
129 
130         self.mu += np.dot( K , np.array([[landmark[1]], [landmark[2]]]) - h )
131         self.sigma = np.dot( T , self.sigma )
```

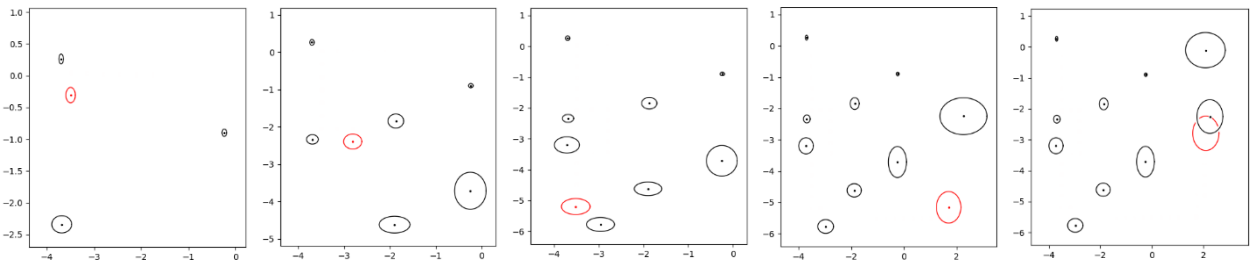
Risultati

Nello SLAM è fondamentale che vi sia la chiusura dell'anello o del loop, ovvero tornare a vedere i landmark visti nella fase iniziale, quando gli errori erano molto bassi. Come visto dalla teoria, quando il robot si muove accumula un errore che tende ad aumentare. Questo, oltre a gravare sulla posizione dello stesso, inciderà sulla posizione dei nuovi landmark avvistati.
Grazie alla chiusura del loop l'errore verrà drasticamente ridotto.

In generale le varie posizioni degli elementi in gioco (robot e punti di riferimento) non saranno mai precise, ma vi sarà sempre una soglia di errore sotto la quale non si è in grado di scendere. Questa coincide spesso con l'errore della misura (vedere i grafici dei risultati sotto).

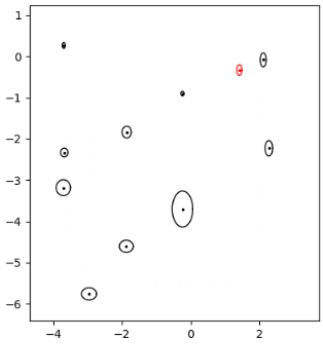
Sotto sono mostrati alcuni grafici risultanti della simulazione del movimento del robot nel progetto in cui possiamo notare come, durante la prima ricognizione, l'errore del robot, descritto attraverso l'ellissi rossa, aumenti sempre di più. Inoltre, l'errore sui landmark, descritto dalle ellissi nere, tende ad aumentare in quanto influenzato da quello sul movimento del robot.

Durante il primo giro, fase di misurazione:

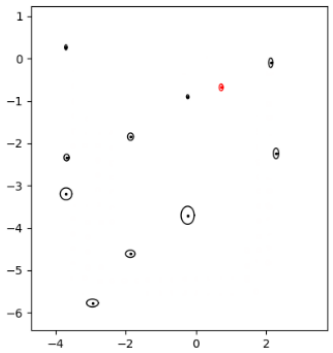


Una volta rivisti i landmark (sotto) di partenza si assiste a una drastica diminuzione dell'errore sulla posizione e sui landmark, testimoniata dalla riduzione della dimensione delle ellissi. Più il robot si muove nella mappa, più gli errori decrescono.

Nella figura sotto è riportato il grafico nel momento in cui il robot ha rivisto i landmark inziali. Si può notare come, rispetto all'ultima figura precedente, tutte le ellissi siano più piccole testimoniando una riduzione dell'errore sulle loro posizioni.

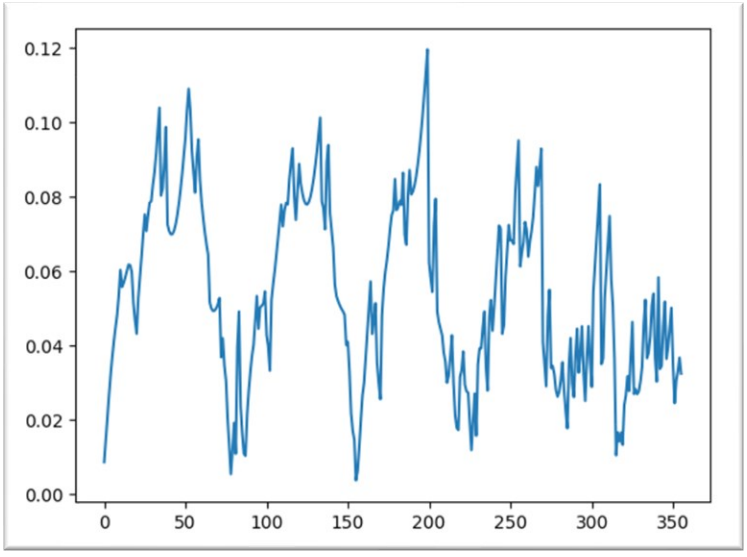


Al termine del secondo giro della stanza il grafico assume la seguente forma.



È possibile osservare l'evoluzione completa del grafico durante il movimento nei video Video\slam_continuo.mp4 oppure Video\slam_cella.mp4.

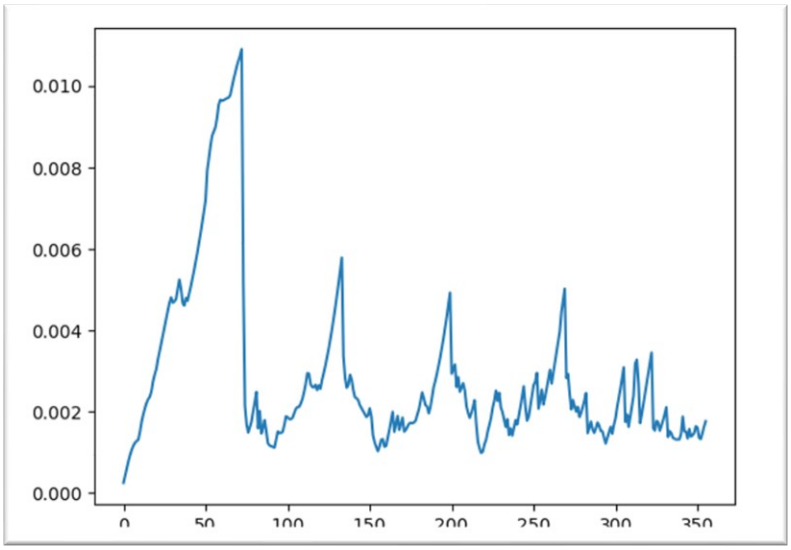
Per una maggiore comprensione dei risultati ottenuti riportiamo di seguito una serie di grafici ottenuti in fase di simulazione.



Nel grafico è rappresentata la differenza tra la posizione reale del robot, ottenuta tramite un supervisor di webots, e la posizione stimata dallo SLAM (ordinata). Notiamo in generale un andamento oscillatorio

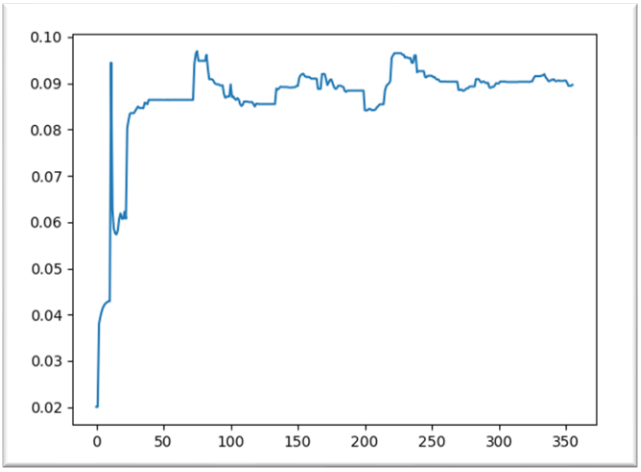
causato dal movimento del robot nella stanza. Infatti, durante la prima visita (dell'ambiente) la posizione stimata si allontana sempre di più da quella reale, causa le continue predizioni. Quando vengono riosservati i landmark di partenza, i quali hanno un errore molto basso, notiamo come il robot corregge la stima sulla propria posizione riducendo l'errore sotto i 2 cm. Ricominciando il giro della stanza questo continua ad accumulare errore fin quando non si avrà una nuova chiusura del loop.
In generale notiamo che i picchi di errore tendono a diminuire.

Il secondo grafico mostra dell'errore sulla posizione stimata contenuto nella matrice sigma (ordinata).



In questo grafico è stata calcolata l'area dell'ellisse generata dai valori delle covarianze della posizione. Anche in questo grafico notiamo un andamento oscillatorio. Come visto prima quando il robot si allontana per troppo tempo dai landmark iniziali, l'errore stimato aumenta sempre di più, fino a quando questi non vengono rivisti. In questo caso il filtro entra in azione diminuendo l'errore sulla posizione. Da notare che ogni volta che il robot si allontana da questi landmark l'errore aumenta per poi ridimensionarsi nella chiusura del loop. Anche qui osserviamo una convergenza dei valori d'errore al trascorrere del tempo.

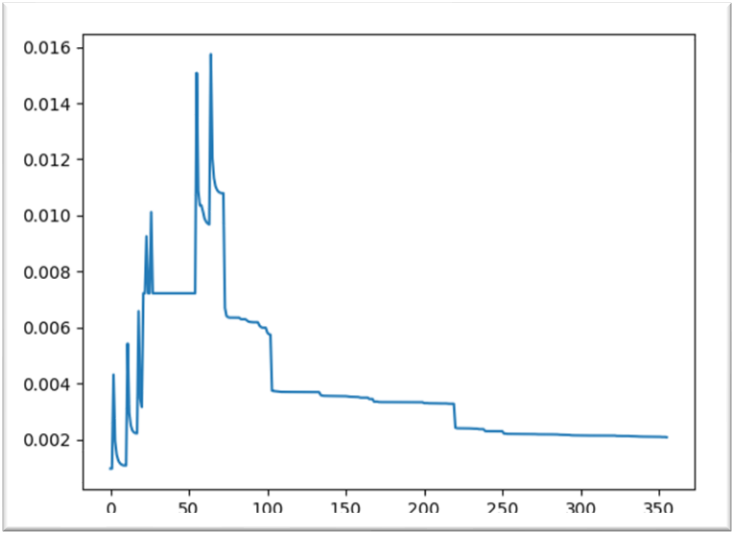
Sotto invece vengono analizzati i risultati ottenuti sui landmark. Nel primo grafico è riportata la distanza tra la posizione del landmark reale e quella stimata (in ordinata). Avendo 10 landmark nella simulazione, ogni volta è stato selezionato, per il plot, quello con errore maggiore.



In questo caso salta subito all'occhio come la posizione dei landmark dopo aver effettuato il primo giro della stanza raggiungono un valore d'errore che oscilla tra gli 8 e i 10 centimetri rispetto a quella reale.

Questo è dovuto all'errore di misura che nel nostro caso è di 9 cm. Infatti, l'errore sulla posizione dei vari landmark non potrà mai scendere sotto tale valore poiché questo risulta essere un errore intrinseco al dispositivo di misura (fotocamera).

Infine, viene effettuata una valutazione della variazione dell'area dell'ellisse generata dalle covarianze dei landmark (in ordinata). Anche in questo caso è stata considerata l'area maggiore tra tutti i landmark istante per istante.



Come descritto precedentemente, le varie ellissi accrescono la propria area fin quando non avviene la chiusura del loop. Da questo momento in poi l'area potrà solo diminuire.

In conclusione, riosservando i vari grafici, si può notare come l'andamento dell'errore del robot è di natura oscillatoria a causa dalla continua predizione sul movimento, che incrementa l'errore, e la successiva correzione data dalla chiusura del loop. I landmark invece presentano un andamento più regolare poiché elementi statici sui quali non effettuiamo alcuna predizione; in altre parole, sui landmark fissi l'errore potrà soltanto diminuire.

3.1.2 MapSearch.py

La classe consente di creare una struttura dati di supporto, chiamata **Mappa di Ricerca**, per effettuare la ricerca di un percorso minimo.
La mappa di ricerca è da vedere come un **multi-albero** ovvero un grafo aciclico diretto (DAG) in cui esiste un solo percorso che ci porta da un nodo a un altro. Ogni nodo rappresenta una cella libera della mappa su cui si muove il robot. Gli archi, data la struttura a multi-albero del grafo, collegano un genitore ai figli e viceversa. La particolarità rispetto a un normale grafo è che, per ogni nodo, vengono mantenuti soltanto i collegamenti ai figli per i quali è stato trovato un percorso ottimo

passante per il nodo stesso. In questo modo siamo sicuri che la struttura a multi-albero venga mantenuta.

Per maggiori dettagli sulla ricerca vedere la parte su 'Search.py'.

La struttura dati che implementa questo grafo è una matrice del tipo:

$$\begin{bmatrix} Nodo_{1,1} & \cdots & Nodo_{1,n} \\ \vdots & \ddots & \vdots \\ Nodo_{m,1} & \cdots & Nodo_{m,n} \end{bmatrix}$$

Il nodo (i,j) si riferisce alla cella di coordinate (i,j) nella mappa in cui robot si muove. Di conseguenza anche la mappa su cui si muove il robot ha dimensione m×n.
Ogni nodo è un array della forma:

$$Nodo_{i,j} = [posizione\ padre,\ \ posizione\ figli,\ \ costo\ totale,\ \ direzione]$$

Dove:

- **Posizione padre** è, per ogni nodo, il puntatore al padre. È rappresentato come un array contenente gli indici di riga e colonna del padre, equivalenti alla posizione della sua cella nella mappa. La sua utilità si rivela quando, una volta trovato il nodo obbiettivo, si deve ricostruire il percorso: basterà allora seguire la sequenza di genitori a partire dall'obbiettivo fino alla radice.
- **Posizione figli** è, per ogni nodo, il puntatore ai figli. In realtà non si inseriscono tutti successori come figli, ma solamente quelli per cui il percorso migliore trovato fino a quel momento passa per il nodo in esame.
  È rappresentato come una lista contenente gli indici di riga e colonna di tutti i figli, equivalenti alle posizioni delle loro celle nella mappa. La sua utilità si rivela quando troviamo un percorso migliore (a costo minore rispetto a quelli precedente conosciuti) per arrivare in un determinato nodo: in questo caso è necessario ricontrollarne i figli e, nel caso in cui il percorso migliore per questi passa dal suddetto nodo, aggiornarne il costo.
- **Costo totale** è, per ogni nodo, il costo del percorso trovato fino a quell'istante di esecuzione che ci porta al nodo partendo dalla radice. Viene aggiornato ogni volta che si trova un percorso migliore per il nodo stesso o per uno dei suoi antenati.
- **Direzione** indica la direzione che avrebbe il robot quando se passasse per questo nodo seguendo il percorso trovato fino a quell'istante. È necessario in quanto nell'algoritmo di ricerca (Search.py) vengono dati costi differenti alle operazioni del robot: se questo va dritto per passare alla cella successiva l'operazione ha costo 1, se invece deve girarsi l'operazione ha costo 2. Si tiene conto di questo nel calcolo dei costi totali dei nodi figli.

<u>Variabili:</u>

- **map**: contiene la mappa di ricerca

Metodi:

- **root**: consente di inserire la radice nel grafo. Questa viene inserita nella matrice di ricerca in posizione [0,0]
- **getNode**: dati gli indici di riga e colonna di un nodo restituisce l'intero nodo
- **setNode**: aggiunge un nodo nella posizione corretta della matrice di ricerca
- **isEmpty**: controlla se un nodo è stato già inserito nel grafo, quindi, se già abbiamo trovato un percorso per la cella a cui si riferisce il nodo
- **getParent**: dato un nodo restituisce le coordinate del genitore
- **setParent**: dato un nodo modifica il riferimento al genitore. Questo capita quando viene trovato un percorso migliore, rispetto a quello precedentemente conosciuto, per quel nodo passando per un altro genitore. Allora è necessario modificare il riferimento.
- **getChildren**: dato un nodo restituisce i riferimenti a tutti i nodi figli
- **addChild**: dato un nodo aggiunge il riferimento a un figlio
- **removeChild**: dato un nodo genitore e il figlio rimuove il riferimento del figlio dal genitore
- **getCost**: dato un nodo ne restituisce il costo del percorso migliore trovato fino a quel momento partendo dalla radice.
- **setCost**: modifica il costo di un nodo. Questo capita quando viene trovato un percorso migliore per arrivare al nodo o a uno dei suoi antenati. Allora è necessario modificare il costo di questo nodo
- **getDirection**: dato un nodo ne restituisce la direzione
- **setDirection**: dato un nodo ne modifica la direzione. Questo capita quando viene trovato un percorso migliore per arrivare al nodo. Potrebbe infatti capitare che seguendo questo nuovo percorso la direzione che avrà il robot passando per questo nodo sarà differente rispetto a quella precedentemente memorizzata.
- **updateParentAndChildren**: dato un nodo il metodo si occupa di aggiornare i puntatori al genitore, i costi e le direzioni di un nodo e di tutti i suoi discendenti. Inoltre, dopo aver aggiornato il puntatore al padre in un nodo, è necessario rimuovere, per il vecchio padre, il puntatore a questo nodo. In questo modo vengono mantenuti i vincoli descritti precedentemente su posizione del padre e posizione dei figli.
- **getPath**: dato un nodo restituisce il percorso memorizzato nel grafo che arriva al nodo partendo dalla radice.

3.1.3 GridWorld.py

La classe viene usata dal controllore per poter creare e gestire la rappresentazione iconica dell'ambiente in cui il robot si sta muovendo. Dalla classe può essere infatti istanziato un oggetto contenente la rappresentazione della mappa, oltre a delle variabili, come ad esempio la cella di partenza, la posizione e la direzione, che consentono al robot di tener traccia dei suoi movimenti e potersi localizzare. In realtà questi valori, soprattutto quello della posizione, sono solamente delle

previsioni su dove il robot possa trovarsi e vengono fatte man mano che questo si muove: la reale posizione è sempre calcolata attraverso l'algoritmo Slam.

Tutti i metodi della classe sono quindi metodi di utilità che consentono di poter aggiornare la forma della mappa, la posizione prevista del robot ed effettuare ricerche di percorsi (vedi Search.py).

Variabili:

- **map**: è la rappresentazione che il robot sta creando dell'ambiente in cui si muove. Questa mappa è rappresentata sotto forma di una matrice in cui ogni cella corrisponde a una cella del mondo a blocchi. I valori ammessi sono:
  - 0: cella libera e non ancora visitata
  - 1: cella occupata da ostacoli fissi, quali mura o mobili.
  - -1: cella libera e visitata

  La matrice è inizializzata come:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

  Il robot si trova nel punto centrale e la cella dietro è sempre occupata dalla sua base di partenza. Quando il robot si muove la matrice viene espansa aggiungendo righe e colonne e vengono aggiornati i valori in base agli input provenienti dai sensori.
- **startCell**: è la cella da cui il robot è partito e in cui il robot tornerà dopo aver finito di pulire l'ambiente.
- **position**: contiene le coordinate in cui si trova il robot. Come già detto è solamente una predizione che il robot effettua tenendo conto dei movimenti fatti.
  Il suo valore iniziale è [1, 1]
- **front**: contiene le coordinate della cella di fronte al robot. Si è preferito mantenerle in una variabile piuttosto che calcolarle quando necessario per semplicità di implementazione. Anche queste sono una predizione fatta dal robot.
- **left**: contiene le coordinate della cella alla sinistra del robot. Si è preferito mantenerle in una variabile piuttosto che calcolarle quando necessario per semplicità di implementazione. Anche queste sono una predizione fatta dal robot.
- **right**: contiene le coordinate della cella alla destra del robot. Si è preferito mantenerle in una variabile piuttosto che calcolarle quando necessario per semplicità di implementazione. Anche queste sono una predizione fatta dal robot.
- **direction**: è la direzione che ha il robot in ogni istante. Dato che il robot si muove in un mondo a griglia, tutte le possibili direzioni sono state discretizzate, e sono (riferite alla matrice):
  - verso l'alto, indicata col valore 0 e riferita al fatto che, nella matrice, sta puntando alla riga sopra a quella corrente
  - verso destra, indicata col valore 1 e riferita al fatto che, nella matrice, sta puntando alla colonna davanti a quella corrente
  - verso sotto, indicata col valore 2 e riferita al fatto che, nella matrice, sta puntando alla riga sotto a quella corrente
  - verso sinistra, indicata col valore 3 e riferita al fatto che, nella matrice, sta puntando alla colonna dietro a quella corrente

  Il suo valore iniziale è 2.

Metodi di aggiornamento:

- **updateMap**: si occupa di aggiungere righe e/o colonne nella matrice che rappresenta la mappa basandosi sul movimento del robot.
  Se la posizione del robot si trova nelle celle esterne della matrice, ovvero nella seconda o penultima riga, si decide di aggiungere una riga rispettivamente all'inizio o alla fine; allo stesso modo se si trova nella seconda o penultima colonna, si decide di aggiungere una colonna rispettivamente all'inizio o alla fine della matrice. Si parte dalla seconda o penultima riga o colonna in modo che le variabili front, left e right abbiano sempre indici validi (all'interno della matrice). Inoltre, la scelta è rafforzata dal fatto che il robot non potrà mai trovarsi nelle righe o colonne più esterne in quanto queste celle saranno occupate dalle mura che delimitano l'ambiente.
  Attraverso questo aggiornamento la dimensione della matrice viene resa coerente con quella dell'ambiente che il robot sta esplorando. Inoltre, a causa dell'aggiunzione di elementi, tutte le coordinate di riferimento mantenute nelle variabili precedentemente descritte vengono opportunamente aggiornate.
- **addObstacle**: dato in input un riferimento su dove si trova l'ostacolo ('front', 'left' o 'right') aggiorna il valore della matrice in quel punto ponendo il valore della cella a 1.
- **doAction**: dato in input un riferimento all'azione svolta ('front', 'left' o 'right') aggiorna le variabili direction, front, left e right. Inoltre, pone a -1 la cella indicata in position in quanto è quella che il robot sta visitando al momento.

Metodi di ricerca:

- **cells2movements**: prende in input un percorso contenente una sequenza di celle e lo trasforma in una sequenza di movimenti che il robot deve attuare per arrivare all'obbiettivo.
  I movimenti sono:
  - 'front': andare dritto
  - 'left': girare a sinistra
  - 'right': girare a destra

  La trasformazione consiste nel calcolare la mossa necessaria per passare da una cella alla successiva e trasformarla in sequenze del tipo: ['front'] se la cella successiva è di fronte a quella corrente, ['right', 'front'] se la cella successiva è a destra, quindi il robot deve prima girarsi, e ['left', 'front'] se la cessa successiva è a sinistra.
- **searchPath**: dato in input l'obbiettivo, chiama il metodo search (vedi Search.py) per effettuare una ricerca del percorso migliore per arrivare all'obbiettivo partendo dalla posizione corrente (memorizzata nella variabile position). Trovato il percorso lo trasforma in una sequenza di mosse. Il metodo viene utilizzato all'interno del controllore solamente per cercare celle libere non esplorate (con valore 0).
- **goToBase**: viene chiamato quando il robot ha terminato la pulizia dell'ambiente e deve ritornare alla base (memorizzata nella variabile startCell). Chiama quindi il metodo search (vedi Search.py) per trovare il percorso migliore e lo trasforma in una sequenza di mosse.

Metodi di utilità:

- **plotMap**: plotta la mappa in un grafico python creato tramite libreria matplotlib
- **getFrontValue**: restituisce il valore della mappa nella cella di fronte al robot
- **getRightValue**: restituisce il valore della mappa nella cella alla sinistra del robot
- **getLeftValue**: restituisce il valore della mappa nella cella alla destra del robot
- **getStartCell**: restituisce le coordinate della cella da cui è partito il robot
- **getPosition**: restituisce le coordinate della posizione del robot
- **getDirection**: restituisce la direzione corrente del robot

3.2 File specifici

I File specifici contengono dei metodi che istanziano oggetti delle classi nei file generici precedentemente descritti e li utilizzano per svolgere determinate operazioni.

3.2.1 Search.py

Il file contiene tutte le funzioni necessarie per effettuare una ricerca di un percorso minimo sulla mappa attraverso l'utilizzo della classe MapSearch.py precedentemente descritta. Più nello specifico, le varie funzioni servono per istanziare un oggetto MapSearch, creare e riempire una **mappa di ricerca** (struttura dati ausiliaria), e infine restituire il percorso migliore che ci porti dalla radice all'obbiettivo.

Una ricerca viene avviata quando il robot si trova in una configurazione in cui nessuna delle celle di fronte, a destra o a sinistra è libera (quindi è occupata o è già stata visitata). Lo scopo della ricerca è quello di trovare la cella libera più vicina al robot e calcolare il percorso da seguire affinché il robot la raggiunga. Nel caso in cui non ne venga trovata nessuna nell'intera mappa significa che il robot ha concluso la sua esplorazione: deve quindi tornare alla base (cella da cui è partito). Per farlo viene avviata una ulteriore ricerca per calcolare il percorso migliore per questo nuovo obbiettivo.

Una precisazione da fare è che il seguente algoritmo di ricerca è stato proposto dagli sviluppatori del progetto e ottimizzato ai fini del progetto stesso. Al seguito sono riportati alcuni test effettuati sull'algoritmo per valutarne il costo computazionale e l'occupazione di memoria.
In una mappa con una dimensione approssimabile a 20x20, come quella usata nell'ambiente di esempio, nel caso peggiore in cui si ricerca un percorso da un angolo all'altro della mappa otteniamo i seguenti risultati:

```
------------ Dati Ricerca ------------
  Dimensione mappa: 20x20
  Punto di partenza: [0, 0]
  Obbiettivo: [19, 19]
--------------------------------------

Inizio Ricerca  ----  1644579336.003074
...
Ricerca Terminata  ----  1644579336.0160758

------------------ Risultati ------------------
Tempo impiegato: 0.013s
Memoria reale utilizzata: 27.0MB
Memoria virtuale utilizzata: 114.38MB
-----------------------------------------------
```

I risultati di altre esecuzioni, in cui vengono utilizzate mappe rispettivamente di 300x300 e 500x500 elementi, sono riportati sotto:

```
------------- Dati Ricerca -------------
  Dimensione mappa: 300x300
  Punto di partenza: [0, 0]
  Obbiettivo: [299, 299]
----------------------------------------

Inizio Ricerca  ----  1644579034.0968637
...
Ricerca Terminata  ----  1644579044.808718

------------------- Risultati --------------------
Tempo impiegato: 10.713s
Memoria reale utilizzata: 55.39MB
Memoria virtuale utilizzata: 143.24MB
--------------------------------------------------
```

```
------------- Dati Ricerca -------------
  Dimensione mappa: 500x500
  Punto di partenza: [0, 0]
  Obbiettivo: [499, 499]
----------------------------------------

Inizio Ricerca  ----  1644579076.4910812
...
Ricerca Terminata  ----  1644579125.4974818

------------------- Risultati --------------------
Tempo impiegato: 49.007s
Memoria reale utilizzata: 108.82MB
Memoria virtuale utilizzata: 196.55MB
--------------------------------------------------
```

Dai risultati sopra mostrati possiamo notare come l'algoritmo riesca a trovare il percorso migliore anche in grafi di 250.000 nodi mantenendo limitati il tempo e le risorse computazionali utilizzate.

Tornando alla descrizione dell'algoritmo, la ricerca viene effettuata partendo dalla radice che viene inserita nella **mappa di ricerca**.

```
26      map_search = Map(dim)
27      map_search.root(position, direction)
```

Ricordare che gli indici del nodo nella mappa di ricerca equivalgono alle coordinate della cella a cui il nodo si riferisce nella mappa dell'ambiente. Di conseguenza se la radice fosse la cella di coordinate (i,j) allora il nodo a essa associato verrebbe inserito nella mappa di ricerca in posizione (i,j). Sotto è riportato un esempio in cui sono rappresentate parte della mappa del mondo, a sinistra, e parte della mappa di ricerca, a destra. Si suppone che la cella da cui partire per effettuare la ricerca sia la $cella_{i,j}$, di conseguenza viene inserita la radice $nodo_{i,j}$ all'interno della mappa di ricerca (inizialmente riempita di liste vuote).

$$\begin{bmatrix} cella_{i-1,j-1} & cella_{i-1,j} & cella_{i-1,j+1} \\ cella_{i,j-1} & cella_{i,j} & cella_{i,j+1} \\ cella_{i+1,j-1} & cella_{i+1,j} & cella_{i+1,j+1} \end{bmatrix} \qquad \begin{bmatrix} [\,] & [\,] & [\,] \\ [\,] & nodo_{i,j} & [\,] \\ [\,] & [\,] & [\,] \end{bmatrix}$$

Ogni nodo che viene inserito nella mappa di ricerca è anche marcato come nodo non visitato e viene inserito in una lista chiamata **Open**.

$$Open = [Nodo_1, Nodo_2, \dots, Nodo_i, Nodo_{i+1}, \dots Nodo_k]$$

Open, inoltre, mantiene sempre un certo ordinamento: il costo totale del $nodo_i$ è sempre minore di quello del $nodo_{i+1}$.
Ricordare che il costo totale è un valore mantenuto all'interno del nodo e indica il costo del percorso migliore trovato fino a quel momento che ci porta al nodo partendo dalla radice.

```
53      open.sort(key = lambda node: map_search.getCost(node))
```

Proprio questo particolare ordinamento garantisce l'ottimalità dell'algoritmo di ricerca: esplorando sempre il percorso migliore tra tutti quelli ancora aperti, siamo sicuri del fatto che quando esploriamo per la prima volta il nodo obbiettivo, è stato già trovato il percorso migliore per poterci arrivare.

La parte iterativa dell'algoritmo procede nel seguente modo:

1. Si estrae il primo elemento della lista open

```
31          node = open.pop(0)
```

2. Se il nodo è un nodo obbiettivo, data l'ottimalità dell'algoritmo, siamo sicuri di aver trovato il percorso migliore per questo. Per ricostruire il percorso si segue, partendo da questo nodo obbiettivo, la sequenza di puntatori al genitore fino alla radice. La ricerca viene quindi terminata.
   Gli obbiettivi della ricerca possono essere: un intero, quando il robot deve capire se nella mappa esistono celle che ancora non ha esplorato (quindi contengono un valore pari a 0), oppure una lista contenente le coordinate [x,y] della cella obbiettivo, quando il robot deve tornare alla base.

```
33          if (type(goal) is list and node == goal) or (type(goal) is int and map[node[0], node[1]] == goal):
34              return map_search.getPath(node)
```

3. Se il nodo non è un nodo obbiettivo si calcolano i suoi successori. Dato che si lavora su un mondo a griglia i successori di un nodo sono solamente quelli sopra, sotto, a destra e a sinistra. Per ognuno di questi vengono indicate in ordine le coordinate x e y e la direzione che avrebbe il robot se passasse per quel successore

```
5       nextNodes = [[node[0]-1, node[1], 0], [node[0], node[1]+1, 1], \
6                    [node[0]+1, node[1], 2], [node[0], node[1]-1, 3]]
```

   Tra questi nodi selezioniamo solamente i successori validi, ovvero quelli che si trovano dentro la mappa e che si riferiscono a celle libere

```
11          if nextNode[0] < 0 or nextNode[1] < 0 or nextNode[0] >= dim[0] or \
12             nextNode[1] >= dim[1] or map[nextNode[0]][nextNode[1]] == 1:
13              nextNodes.pop(i)
14              continue
```

   Infine, aggiungiamo ai successori validi anche il costo dell'arco che lo collega al genitore; questo sarà utile per calcolare il costo totale di questo successore.
   Per quanto riguarda i costi degli archi si suppone che ogni singola operazione base (andare dritto o girarsi) abbia costo 1. Un arco ha quindi costo:
   - 1, se la direzione del padre è uguale a quella del successore. In questo caso il robot dovrebbe solamente andare dritto.
   - 2, se la direzione del successore si trova a destra o a sinistra di quella del padre. In questo caso il robot dovrebbe prima girarsi e successivamente procedere verso il successore.

- 3, se la direzione del successore è opposta a quella del padre. In questo caso il robot dovrebbe girarsi due volte per poi procedere verso il successore

```
14          if dir == nextNode[2]:
15              nextNodes[i].append(1)
16          elif dir == (nextNode[2]+1) % 4 or dir == (nextNode[2]-1) % 4:
17              nextNodes[i].append(2)
18          else:
19              nextNodes[i].append(3)
```

4. Per ogni successore valido ottenuto al passo precedente si controlla se questo sia già stato visitato, quindi, se si trova già nella mappa di ricerca.
   a. Se il successore non è ancora stato visitato si aggiunge come figlio del nodo in esame. Infine, si aggiunge nella lista Open

```
43                  if map_search.isEmpty(nextNode):
44                      map_search.addChild(node, nextNode)
45                      map_search.setNode(nextNode, node, cost, direction)
46                      open.append(nextNode)
```

   b. Se invece è già stato visitato, si controlla se il costo del nuovo percorso trovato (quello passante per il nodo che stiamo esaminando) è minore rispetto a quello precedente (costo memorizzato all'interno del nodo), e, in tal caso, si aggiunge questo successore come figlio del nodo in esame aggiornano il percorso e il costo totale attraverso la funzione updateParentAndChildren (vedi MapSearch.py)

```
48                  else:
49                      if map_search.getCost(nextNode) > map_search.getCost(node) + cost:
50                          map_search.addChild(node, nextNode)
51                          map_search.updateParentAndChildren(nextNode, node)
```

5. Ricomincia il ciclo tornando a punto 1

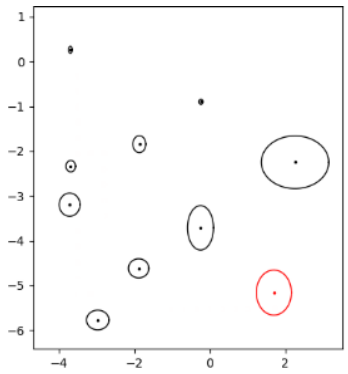
3.2.2 Plot.py

Il file si occupa della gestione dei grafici.

È costituito da due metodi:

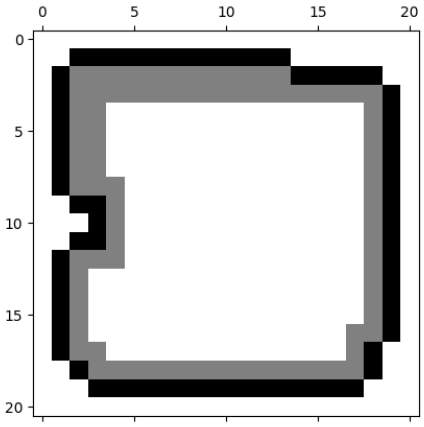
- **plot_slam**: si occupa del grafico relativo alla posizione dei landmark, in nero, e del robot, in rosso, insieme ai rispettivi errori. L'errore è indicato da un'ellisse i cui valori *a* e *b*, corrispondenti agli assi, rappresentano gli errori rispettivamente lungo l'asse x e y.

Per la stampa dell'ellisse sono stati campionati 100 punti da questa e grazie alla funzione di plot è stato possibile creare una sua rappresentazione discretizzata.



I video in cui è possibile vedere l'evolversi del grafico di pari passo alla simulazione sono Video\slam_continuo.mp4 in cui il grafico viene aggiornato in maniera continua (ogni 32ms), e Video\slam_cella.mp4 in cui il grafico viene aggiornato di cella in cella.

- **plot_map**: si occupa di mostrare la mappa generata dal robot nella fase di mappatura. In questo metodo è stato semplicemente richiamato il metodo matshow del modulo matplotlib.pyplot il quale permette di stampare di una matrice (il mondo a griglia). Nel grafico le celle in nero sono quelle occupate, in grigio quelle visitate e in bianco quelle non ancora esplorate.



Un video in cui è possibile vedere l'evoluzione della mappa di pari passo alla simulazione è Video\mappa.mp4.

3.3 File Principale

Il **File principale** è il controllore del robot che, a sua volta, sfrutta tutte le funzioni messe a disposizione dai file specifici e dai file generici per poter portare a termine il suo compito.

3.3.1 Controller.py

Questo file è quello principale che controlla l'esecuzione di tutti i processi che gestiscono il robot.

Metodi per l'attivazione dei dispositivi del robot:

- **robot.getDevice:** in base all'argomento passato restituisce i motori delle ruote del robot, i sensori di distanza e la telecamera del robot.
- **setPosition:** per selezionare la posizione che il robot deve raggiungere. In questo caso **inf** così da gestire il movimento solo mediante la velocità.
- **setVelocity:** per selezionare le velocità delle due ruote.
- **ds.enable:** per attivare i sensori di distanza.
- **camera.enable:** per attivare la telecamera.
- **camera.recognitionEnable:** metodo per far riconoscere alla telecamera gli oggetti in base al loro colore.

Variabili d'ambiente

- **cellSize:** dimensione di una cella. Nell'implementazione, una cella ha una lunghezza uguale al diametro del robot.
- **bodyRadius**: raggio del robot.
- **maxSpeed**: velocità angolare massima raggiungibile dalle route. Viene assegnata a queste quando il robot va dritto. Per il robot utilizzato è pari a 6.28 rad/s.
- **turnSpeed:** velocità angolare da assegnare alle ruote durante una rotazione del robot. Ai fini del progetto è stata posta a 3 rad/s.
- **linearVelocity**: velocità lineare del robot quando la velocità delle ruote è impostata a maxSpeed.
- **angularVelocity:** velocità angolare del robot quando la velocità delle ruote è impostata a turnSpeed.

Variabili per il movimento del robot

- **distanceBetweenWheels:** distanza tra le ruote, cioè pari al raggio moltiplicato per due.
- **rateOfRotation**: Indica quanti radianti vengono percorsi in un secondo quando il robot ruota a una velocità angolare *angularVelocity*.
- **angleOfRotation**: pari a 90 gradi, supponiamo che il robot ruoti sempre di 90 gradi durante il tragitto.
- **durationRotation:** durata in secondi della rotazione di 90 gradi.

- **durationSide:** durata in secondi del tragitto del robot per muoversi della dimensione di una cella.

Metodo iniziale

- **main**: è il metodo principale in cui ad ogni ciclo *while*, cioè ogni 32 millisecondi, vengono richiamati i metodi principali di movimento e controllo della posizione, degli ostacoli e della mappa. controlState controlla dove si trova il robot e se la traiettoria sta subendo un errore rispetto alla direzione prevista, elaborateAcion è il metodo per decidere quale azione eseguire e doAction la esegue. Al termine della mappatura completa dell'ambiente il robot ritorna al punto di partenza e si ferma settando la velocità delle ruote a zero.

```
342 def main():
343     while robot.step(timestep) != -1:
344         controlState()
345         action = elaborateAction()
346         next = doAction( action )
347 
348         if not next: break
349 
350     motorLeft.setVelocity(0)
351     motorRight.setVelocity(0)
```

Metodi principali

- **controlState:** come già detto questo metodo aggiorna la mappa mediante i metodi provenienti dalla classe GridWorld

```
126     gridWorld.updateMap()
127     measureObstacles()
```

e controlla la posizione del robot. Dalla classe Slam otteniamo i valori della posizione del robot e, nel caso discostino dalla posizione in cui il robot prevede di trovarsi di un fattore pari a un quarto della dimensione della cella, si attiverà una routine che ne correggerà la posizione. I metodi utilizzati da questa routine di correzione sono correctPosition e measureObstacles descritti più avanti. Infine, se è richiesto verrà stampato il grafico della mappa.

```
129     [x, y] = slam.getMu()[0:2, 0]
130     startCell = gridWorld.getStartCell()
131     position = gridWorld.getPosition()
132     xTarget = (position[0]-startCell[0]) * cellSize
133     yTarget = (position[1]-startCell[1]) * cellSize
134     dist_x = xTarget - x
135     dist_y = yTarget - y
136     distance = np.sqrt(dist_x**2 + dist_y**2)
137 
138     if distance > cellSize/4:
139         theta = mt.atan2(dist_y, dist_x)
140         correctPosition(distance, theta)
141         measureObstacles()
```

- **elaborateAction:** questo metodo richiama i metodi della classe GridWorld che restituiscono i valori della mappa nelle celle di fronte, a destra e a sinistra del robot. Il metodo restituisce quattro possibili valori:
    - “right”: Si predilige una rotazione verso destra se la cella è libera e non è già stata visitata.
    - “front”: se la cella di fronte è libera oppure se è stata già visitata e la a sinistra è occupata.
    - “path”: se nessuna delle due precedenti condizioni sono verificate e se le celle di fronte a destra e a sinistra sono già state visitate.
    - “left”: nel caso nessuna delle precedenti condizioni sono verificate.

```
147  def elaborateAction():
148      if gridWorld.getRightValue() == 0:
149          return "right"
150
151      if gridWorld.getFrontValue() == 0 or (gridWorld.getFrontValue() == -1 and gridWorld.getLeftValue() == 1):
152          return "front"
153
154      if gridWorld.getRightValue() == gridWorld.getFrontValue() == gridWorld.getLeftValue() == -1:
155          return "path"
156
157      else:
158          return "left"
```

- **doAction:** prende in input una delle possibili direzioni restituite dal metodo precedente e, nel caso sia una tra “right”, “front” e “left”, chiama makeMove che esegue il movimento e passa il controllo al metodo doAction della classe GridWorld che aggiorna la mappa; viceversa, se il valore in input è “path” richiama il metodo di ricerca del percorso migliore per raggiungere una cella vuota, se ce ne sono, altrimenti, se tutta la stanza è stata visitata, per tornare alla base. Trovato il percorso si usa il metodo followPath per faglielo seguire.

```
162  def doAction(action):
163      global gridWorld
164
165      if action == "path":
166          path = gridWorld.searchPath(0)
167          if path:
168              followPath(path)
169              return True
170          else:
171              path = gridWorld.goToBase()
172              followPath(path)
173              return False
174
175      makeMove(action)
176      gridWorld.doAction(action)
177      return True
```

Metodi per il movimento

- **makeMove:** In base al valore di direzione passato come parametro, si ottengono da un dizionario tutti i valori utili per compiere il movimento. Così facendo il robot ruoterà verso destra o verso sinistra di 90 gradi oppure andrà avanti di una cella.

```
moves = {
    "right": {
        "motorLeft": turnSpeed,
        "motorRight": -turnSpeed,
        "linearVelocity": 0,
        "angularVelocity": -angularVelocity,
        "duration": durationRotation
    },
    "left": {
        "motorLeft": -turnSpeed,
        "motorRight": turnSpeed,
        "linearVelocity": 0,
        "angularVelocity": angularVelocity,
        "duration": durationRotation
    },
    "front": {
        "motorLeft": maxSpeed,
        "motorRight": maxSpeed,
        "linearVelocity": linearVelocity,
        "angularVelocity": 0,
        "duration": durationSide
    }
}
```

```
192     param = moves[action]
```

Nella gestione del movimento si è deciso di lavorare con il tempo: si avvia il moto e, ciclicamente, si verifica se è già trascorso un intervallo di tempo pari alla durata del movimento. In caso positivo il ciclo viene interrotto, altrimenti si comanda al robot di procedere.

```
199     while robot.step(timestep) != -1:
200         currentTime = robot.getTime()
201 >       if currentTime < startTime + duration: ···
232
233         else:
234             motorLeft.setVelocity(0)
235             motorRight.setVelocity(0)
236             break
```

Inoltre, sempre all'interno di questo ciclo, viene effettuato un controllo sugli ostacoli improvvisi, ovvero quelli che si presentano quando il robot ha già cominciato a svolgere il movimento. Nel codice calcoliamo se lo spazio libero di fronte, ottenuto dalla lettura del sensore tof frontale, è maggiore rispetto a quello necessario per completare il movimento. In caso negativo decidiamo di fermare il robot.

```
205                 spaceDone = (currentTime - startTime) * param["linearVelocity"]
206                 if duration == 0: spaceLeft = (cellSize - spaceDone) * 1000
207                 else: spaceLeft = (duration*param["linearVelocity"] - spaceDone) * 1000
208
209                 if param["linearVelocity"] != 0 and tof[0].getValue() < spaceLeft*0.8:
210                     motorLeft.setVelocity(0)
211                     motorRight.setVelocity(0)
```

Si entra allora all'interno di un ciclo di attesa in cui si possono distinguere due casi: quando il robot sta mappando si ferma e attende che l'ostacolo sia passato. Siamo sicuri del fatto che l'ostacolo sia mobile perché, se fosse stato fisso, lo avrebbe rilevato prima di cominciare a muoversi e lo avrebbe segnato con un valore pari a 1 sulla mappa. Il secondo caso si presenta quando il robot sta seguendo un percorso. In questo caso potrebbe accadere che questo percorso porti il robot in zone della mappa esplorate molto prima e che, di conseguenza, potrebbero avere nuovi ostacoli fissi o mobili (pensare ad esempio a una porta chiusa). Una volta incontrato questo ostacolo attende per 10 secondi che il percorso si liberi, perché potrebbe essere un ostacolo mobile; passato questo intervallo, riconosce l'ostacolo come fisso e il ciclo viene interrotto riportando un fallimento.

```
213                 while robot.step(timestep) != -1:
214
215                     if tof[0].getValue() > spaceLeft + cellSize*100:
216                         robot.step(2000)
217                         break
218
219                     if not mapping and robot.getTime() - currentTime > 10:
220                         return False
```

Infine, sempre all'interno del ciclo while di partenza, richiama i metodi necessari per l'aggiornamento dell'EKF-SLAM.

```
224                 slam.predictionMove([param["linearVelocity"], param["angularVelocity"], dt])
225
226                 if camera.getRecognitionNumberOfObjects() != 0:
227                     measurement = measureLandmark()
228                     slam.controlLandmark(measurement)
```

- **followPath:** dato in input un percorso, sotto forma di sequenza di azioni, si occupa di richiamare metodo **makeMove** per eseguirle e, passo per passo, i metodi necessari per l'aggiornamento della mappa**.** Come accennato prima, nel caso in cui il robot trova un ostacolo imprevisto, **makeMove** fallisce, e followPath richiede la ricerca di un nuovo percorso.

```
241 def followPath(path):
242     global gridWorld
243
244     for action in path:
245         success = makeMove(action, mapping = False)
246
247         if not success: break
248
249         gridWorld.doAction(action)
250         controlState()
251
252     if not success:
253         doAction("path")
```

- **correctPosition:** viene richiamato quando il robot si è allontanato troppo rispetto alla posizione che aveva previsto (vedi il metodo controlState() scritto sopra). Il suo compito è quello di correggere la posizione del robot e riportarlo nel punto previsto. La correzione si articola in tre passaggi:
    - **rotazione** che porta il robot a puntare verso la posizione desiderata

```
260     angleRotation = theta - slam.getMu()[2, 0]
261     angleRotation = adjustAngle(angleRotation)
262     durationRotation = getDurationRotation(abs(angleRotation))
263
264     if angleRotation > 0: action = "left"
265     else: action = "right"
266
267     makeMove(action, durationRotation)
```

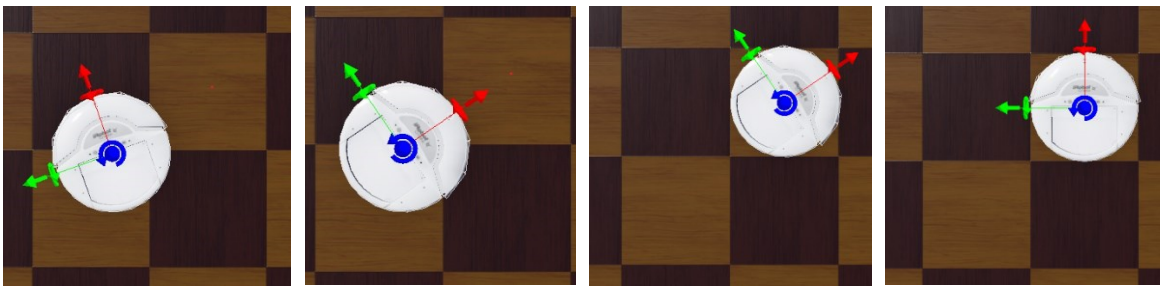
    - **movimento** che porta il robot nel punto desiderato

```
269     durationSide = getDurationSide(abs(distance))
270     makeMove("front", durationSide)
```

    - **rotazione** che aggiusta la direzione del robot per puntare verso la direzione che dovrebbe seguire.

```
272     correctAngle = nearestAngle(incInit)
273     returnAngle = correctAngle - slam.getMu()[2, 0]
274     returnAngle = adjustAngle(returnAngle)
275     durationRotation = getDurationRotation(abs(returnAngle))
276
277     if action == "left": action = "right"
278     elif action == "right": action = "left"
279
280     makeMove(action, durationRotation)
```

Sotto è riportata una sequenza di immagini rappresentanti le fasi principali di un normale movimento di correzione.



- **measureObstacles:** controlla, attraverso i sensori tof, se le celle di fronte, a destra e a sinistra contengono ostacoli e, in caso affermativo, utilizza il metodo addObstacle di GridWorld per aggiungerli.

```
285  def measureObstacles():
286      global gridWorld
287
288      if tof[0].getValue() < cellSize*900: gridWorld.addObstacle("front")
289      if tof[1].getValue() < cellSize*900: gridWorld.addObstacle("right")
290      if tof[2].getValue() < cellSize*900: gridWorld.addObstacle("left")
```

- **measureLandmark:** grazie all'oggetto recognition della telecamera, calcola le coordinate di un landmark che si trova nel campo visivo della telecamera relative alla posizione del robot. Da questi è poi possibile ricavare la distanza (range) e l'angolo di inclinazione (bearing) del robot dal landmark visto. Le misure sono necessaire affinché l'algoritmo Slam sia in grado di calcolare la posizione del landmark.

```
294  def measureLandmark():
295      measures = []
296      landmarks = camera.getRecognitionObjects()
297      for landmark in landmarks:
298          position = landmark.get_position()
299          position[0] += 0.172
300          measures.append(np.vstack(( [mt.sqrt(position[0]**2 + position[1]**2)] , \
301                                      [mt.atan2(position[1], position[0])] )))
302
303      return measures
```

<u>Metodi di utilità</u>

- **getDurationRotation:** restituisce il tempo in secondi necessario per ruotare di un angolo dato in input. Il valore dt = 32ms è stato sommato per adattare meglio il tempo di rotazione al movimento effettivo del simulatore webots. In questo, infatti, qualsiasi sia il movimento comunicato, il primo intervallo di tempo il robot resta fermo.

```
309  def getDurationRotation(angleOfRotation):
310      return angleOfRotation / rateOfRotation + dt
```

- **getDurationSide:** restituisce il tempo in secondi necessario per andare avanti della distanza data in input.

```
312  def getDurationSide(distance):
313      return distance / linearVelocity
```

- **nearestAngle**: restituisce quale degli angoli tra $[0, \frac{\pi}{2}, -\frac{\pi}{2}, \pi, -\pi]$ è più vicino all'angolo in input. Viene usato in correctPosition per correggere la direzione del robot e riportarla verso l'angolo corretto.

```
315  def nearestAngle(angle):
316      angles = [0, 1.57, 3.14]
317      distances = []
318
319      for el in angles:
320          if angle >= 0:
321              distances.append(abs(angle - el))
322          else:
323              distances.append(abs(angle + el))
324
325      if angle >= 0:
326          return angles[distances.index(min(distances))]
327      else:
328          return -angles[distances.index(min(distances))]
```

- **adjustAngle:** riporta il valore dell'angolo in input nel range [-π, π] nel caso esca fuori da questo.

```
330   def adjustAngle(angle):
331       while angle >  np.pi: angle -= 2*np.pi
332       while angle < -np.pi: angle += 2*np.pi
333       return angle
```