
Università degli Studi del Sannio

Dipartimento di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica


# Analisi e Controllo di Sistemi Cyberfisici

*Project Report*

---

Progettazione ed implementazione di EFSM per il controllo di un robot mobile con guida differenziale

---


*Prof.ssa:*
Elisa Mostacciuolo

*Studenti:*
Razzano Federica, matr. 399000542
Stocchetti Federico, matr. 399000543

# Contents

# 1 Introduzione

In questa relazione viene illustrato il lavoro svolto per il corso di Analisi e Controllo di Sistemi Cyberfisici. Grazie alle conoscenze ottenute durante il corso, è stato possibile sviluppare un robot mobile con guida differenziale capace di esplorare e mappare un ambiente arbitrario con ostacoli, utilizzando gli strumenti messi a disposizione da MATLAB per la robotica e il controllo.

Il progetto si concentra sulla creazione di un sistema integrato che include sensori per la raccolta dei dati ambientali, attuatori per il movimento del robot, controllori PID per la regolazione precisa del movimento e sottosistemi Stateflow per la gestione della logica di controllo. L'ambiente di simulazione è stato realizzato utilizzando il Robotics Playground di MATLAB, che ha permesso di concentrare lo sviluppo sulla logica di controllo piuttosto che sull'implementazione di sensori e attuatori simulati.

Un aspetto cruciale del progetto è stato l'uso di Stateflow per implementare macchine a stati finiti estese e non, con il compito di gestire i vari aspetti della logica di controllo del robot. Questo ha incluso la navigazione, la pianificazione del percorso, il tracking dell'orientamento e la mappatura dell'ambiente.

Il sistema di navigazione del robot è stato sviluppato utilizzando un algoritmo di ricerca in ampiezza (Breadth-First Search, BFS), che ha permesso di trovare percorsi efficienti attraverso l'ambiente esplorato. I dati raccolti dai sensori sono stati elaborati tramite uno script esterno per generare la mappa dell'ambiente e del percorso effettuato e la rispettiva OccupancyGridMap.

# 2 Problem Statement

Il presente progetto si concentra sulla realizzazione di un robot a guida differenziale simulato in ambiente 3D, utilizzando il Robotics Playground di MATLAB e Stateflow per la logica di controllo. L'obiettivo principale è sviluppare un sistema autonomo capace di mappare in modo efficace una stanza arbitraria contenente vari ostacoli di grandezza e posizione variabile.

## 2.1 Sfide Progettuali

- **Navigazione Autonoma:**
  - Sviluppare un algoritmo di navigazione che permetta al robot di esplorare l'ambiente senza assistenza esterna.
  - Assicurarsi che il robot possa evitare ostacoli in tempo reale utilizzando i dati sensoriali raccolti.
  - Assicurarsi che l'intero ambiente sia esplorato e assicurarsi la terminazione autonoma dell'esplorazione.
- **Pianificazione del Percorso:**
  - Utilizzare un algoritmo per generare un percorso efficiente attraverso l'ambiente.
  - Mantenere coerenza tra il modello interno dell'ambiente, usato per la navigazione, e le coordinate effettive del robot.
- **Mappatura dell'Ambiente:**
  - Progettare un sistema per la raccolta e il processing dei dati dei sensori, in particolare utilizzando un sensore Lidar per rilevare ostacoli e spazi liberi.
  - Creare una mappa dettagliata dell'ambiente esplorato.
- **Controllo del Movimento:**
  - Implementare controllori per gestire la dinamica di movimento del robot.
  - Assicurare che il robot possa mantenere la traiettoria pianificata durante il movimento rettilineo e la rotazione.
- **Elaborazione dei Dati Raccolti:**
  - Sviluppare uno script esterno per pulire, integrare ed elaborare i dati raccolti dai sensori allo scopo di utilizzarli per la creazione di una mappa accurata che contenga percorso effettuato, ostacoli e spazi liberi dell'ambiente mappato.

# 3 Modello Fisico ideale

In questa sezione viene descritto il modello fisico ideale del robot mobile. Il modello ideale assume l'assenza di inerzia, il che consente di formulare le equazioni differenziali che governano la dinamica del robot in termini di posizione e orientamento angolare. Di seguito sono presentate le equazioni che rappresentano il movimento del robot basato sulla velocità delle ruote sinistra ($v_L$) e destra ($v_R$):

$$\dot{x} = \frac{v_L}{2}\cos(\theta) + \frac{v_R}{2}\cos(\theta)$$
$$\dot{y} = \frac{v_L}{2}\sin(\theta) + \frac{v_R}{2}\sin(\theta)$$
$$\dot{\theta} = \frac{1}{s}\left(-v_L + v_R\right)$$

dove $x$ e $y$ rappresentano le coordinate della posizione del robot nel piano, $\theta$ è l'angolo di orientamento, $v_L$ e $v_R$ sono le velocità delle ruote sinistra e destra, rispettivamente, e $s$ è la distanza tra le due ruote.

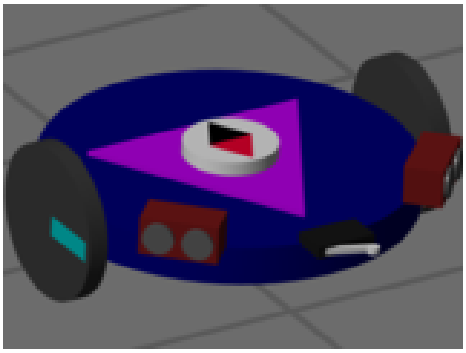


Figure 1: Rappresentazione schematica del robot mobile.

Questo modello non considera l'inerzia e l'attrito, elementi presenti nel modello finale del robot simulato, poiché prende in considerazione l'utilizzo di motori stepper e non di semplici motori DC.
Nonostante queste semplificazioni, il modello fisico ideale fornisce una base valida per descrivere la dinamica del robot, modellare il problema e illustrare il sistema robotico come una sequenza di stati composti da posizione e orientamento.

# 4 Progettazione

Il progetto sviluppa un robot mobile autonomo, concepito per esplorare efficacemente ambienti non strutturati con presenza di ostacoli. L'architettura complessiva del robot è implementata attraverso MATLAB e Simulink, integrando numerosi sottosistemi Stateflow per la gestione della logica di controllo. Questa sezione dettaglia la progettazione del sistema, la disposizione dei suoi elementi principali e la formalizzazione dei suoi componenti chiave.

## 4.1 Formalizzazione delle componenti del sistema

Il sistema è costituito da diversi componenti chiave che lavorano insieme per permettere una navigazione autonoma e una mappatura dettagliata dell'ambiente. Di seguito è presentata una descrizione di ciascuna componente principale:

- **Environment Simulato:** Un ambiente 3D creato all'interno di Simulink dove il robot viene testato, compreso di vari ostacoli che il robot deve superare e mappare.
- **Sensori:** Il robot è dotato di vari sensori per navigare e raccogliere dati sull'ambiente, inclusi:
  - Giroscopio per determinare l'angolazione.
  - Bussola per definire l'orientamento.
  - Lidar per mappare l'ambiente e rilevare gli ostacoli.
  - Encoder per calcolare la distanza percorsa.
- **Attuatori:** Motori che permettono al robot di muoversi e ruotare all'interno dell'ambiente simulato, controllati da feedback per garantire movimenti precisi.
- **Controllori PID:** Utilizzati per regolare la velocità dei motori e l'orientamento del robot in base ai dati ricevuti dai sensori, garantendo che il robot segua le traiettorie pianificate con precisione.
- **Sottosistemi Stateflow:** Il robot integra quattro sottosistemi Stateflow, ciascuno progettato per gestire un aspetto della logica di controllo. Questi includono:
  - *Navigation* - Responsabile della direzione e del movimento del robot, gestisce stati come avanti, stop, gira a sinistra e gira a destra.
  - *Path Planning* - Elabora un percorso attraverso l'ambiente, decidendo la sequenza di azioni basate sulla mappa e sulla posizione corrente.
  - *Orientation* - Mantiene traccia dell'orientamento del robot.

– *Pose&Mapping* - Determina la posizione precisa del robot nell'ambiente, essenziale per la navigazione accurata ed elabora i dati sensoriali per creare una rappresentazione dettagliata dell'ambiente circostante.

Questi sottosistemi sono realizzati come Macchine a Stati Finiti (FSM) e Macchine a Stati Finiti Estese (EFSM), che gestiscono variabili continue e discrete, fornendo una struttura robusta per il controllo dinamico e le decisioni basate su eventi.

- **Script esterno per l'elaborazione dei dati raccolti:** Uno script che processa i dati raccolti dal Lidar per creare una mappa dettagliata dell'ambiente esplorato dal robot.

Ciascuna di queste componenti è integrata in un modello Simulink che gestisce le interazioni e la sincronizzazione tra i diversi moduli, formando un sistema complesso di tipo event-triggered, dove eventi specifici generati dai sensori o dagli input interni scatenano transizioni di stato all'interno delle EFSM. Questo approccio rende il sistema intrinsecamente discreto, poiché opera in base a eventi e cambiamenti di stato ben definiti, pur gestendo dati che possono variare continuamente come quelli dei sensori.

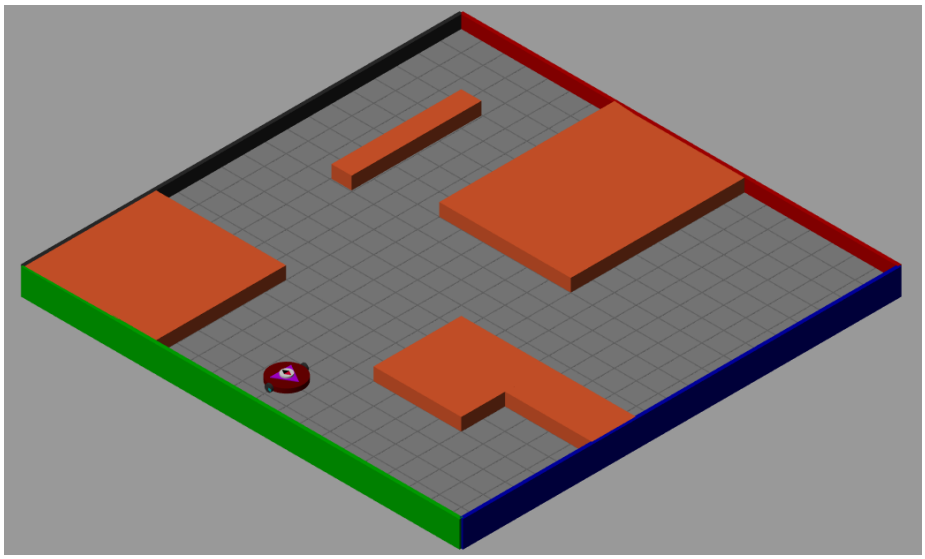


Figure 2: Ambiente della simulazione.

## 4.2 Automi a stati finiti

Un automa a stati finiti (FSM) è definito da una quintupla:

$$(Stati, Ingressi, Uscite, update, statiIniziali)$$

dove:

- **Stati**: rappresenta l'insieme finito degli stati possibili della FSM.
- **Ingressi**: rappresenta l'insieme delle valutazioni degli ingressi. Matematicamente:
$$i : P \to V_{p_1} \cup \ldots \cup V_{p_M} \cup \{absent\}$$
con $P = \{p_1, p_2, \ldots, p_M\}$ porte di ingresso.
- **Uscite**: rappresenta l'insieme delle valutazioni delle uscite. Matematicamente:
$$u : Q \to V_{q_1} \cup \ldots \cup V_{q_N} \cup \{absent\}$$
con $Q = \{q_1, q_2, \ldots, q_N\}$ porte di uscita.
- **update**: è una funzione
$$f : Stati \times Ingressi \to Stati \times Uscite$$
che mappa lo stato e la valutazione degli ingressi nello stato successivo e nella valutazione delle uscite.
- **statiIniziali**: specifica gli stati iniziali da cui parte la FSM.

### 4.2.1 Robot Controller

RobotController è una macchina a stati risultante dalla composizione di tre macchine a stati: Orientation, Navigation e PathPlanning. Queste macchine a stati funzionano simultaneamente, ognuna con i propri stati, ingressi e uscite, interagendo tramite una parte delle loro uscite in un funzionamento *cascade-like*. Orientation e Navigation sono FSM, PathPlanning è una EFSM.
RobotPose infine è una EFSM in cascata con RobotController.

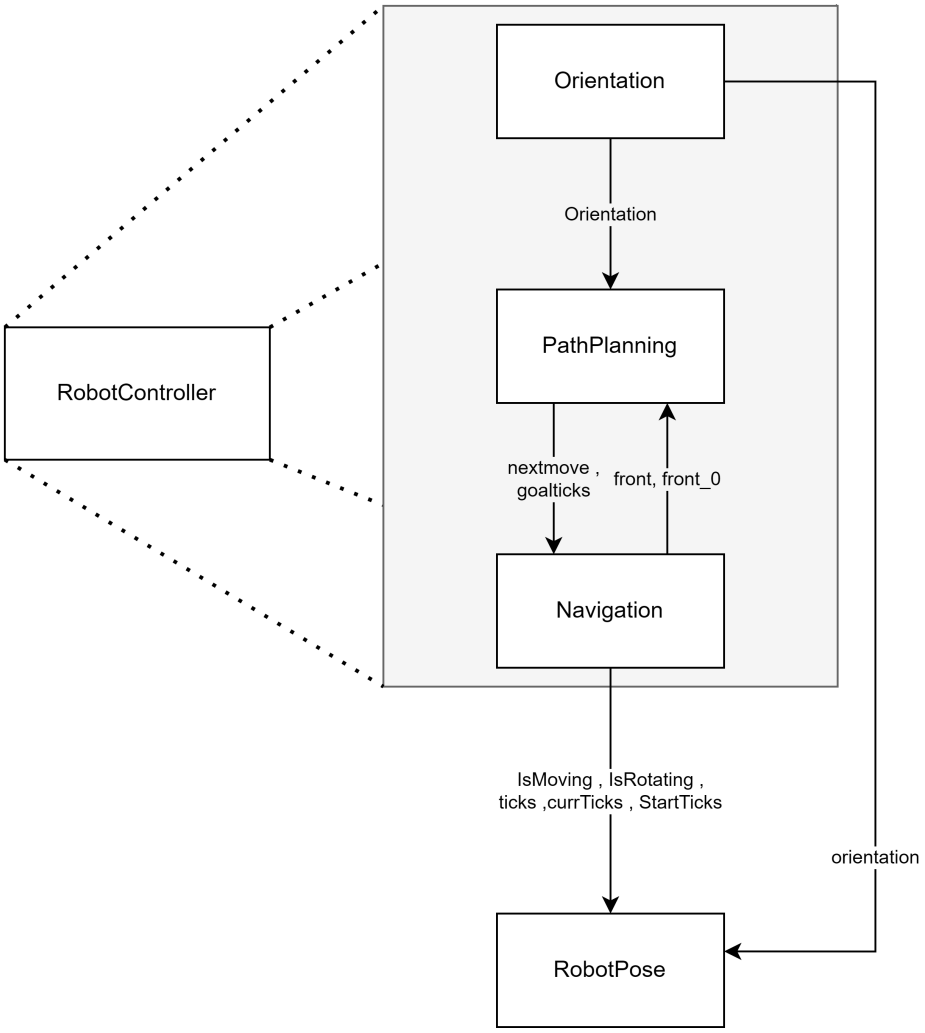



Figure 3: Composizione delle macchine a stati

#### 4.2.2 FSM Orientation

- **Stati**: {NORD, OVEST, EST, SUD}
- **Stato Iniziale**: {EST}
- **Ingressi**:

$$Inputs = \{compass : real[0, 180], navigation : \{turnleft, turnright\}\}$$

- **Uscite**:

$$Outputs = \{orientation : int[0, 270]\}$$

- **Funzione di Update**:

$$update(s,i) = \begin{cases} NORD, 90 & \text{se } s = EST \wedge i(\text{navigation}) = \text{turnleft} \wedge i(\text{compass}) > 45 \\ SUD, 270 & \text{se } s = EST \wedge i(\text{navigation}) = \text{turnright} \wedge i(\text{compass}) < 45 \\ OVEST, 180 & \text{se } s = NORD \wedge i(\text{navigation}) = \text{turnleft} \wedge i(\text{compass}) > 135 \\ EST, 0 & \text{se } s = NORD \wedge i(\text{navigation}) = \text{turnright} \wedge i(\text{compass}) < 135 \\ SUD, 270 & \text{se } s = OVEST \wedge i(\text{navigation}) = \text{turnleft} \wedge i(\text{compass}) > 135 \\ NORD, 90 & \text{se } s = OVEST \wedge i(\text{navigation}) = \text{turnright} \wedge i(\text{compass}) < 135 \\ EST, 0 & \text{se } s = SUD \wedge i(\text{navigation}) = \text{turnleft} \wedge i(\text{compass}) > 45 \\ OVEST, 180 & \text{se } s = SUD \wedge i(\text{navigation}) = \text{turnright} \wedge i(\text{compass}) < 45 \\ s, \text{orientation} & \end{cases}$$

#### 4.2.3 FSM Navigation

- **Stati**: {idle, forward, Stop, TurnRight, TurnLeft}
- **Stato Iniziale**: {idle}
- **Ingressi**:

$$\begin{aligned} Inputs = \{&ticks : int[0, \infty), \\ &nextmove : int\{1, 2, 3, 5\}, \\ &goalticks : int[0, \infty), \\ &compass : real[0, 180]\} \end{aligned}$$

- **Uscite**:

$$\begin{aligned} Outputs = \{&front : int[0, \infty), \\ &front\_0 : int[0, \infty), \\ &IsMoving : bool, \\ &IsRotating : bool, \\ &degrees : int, \\ &DONE : bool\} \end{aligned}$$

- **Funzione di Update**:

$$
update(s,i) = \begin{cases}
idle, \{front = 0, front_0 = 0, isMoving = false, isRotating = false, \\
degrees = 0, DONE = false\} \\
\text{se } s = idle \wedge i(ticks) = \{absent\} \wedge i(nextmove) = \{absent\} \\
\wedge i(goalticks) = \{absent\} \wedge i(compass) = \{absent\}; \\
\\
Stop, \{front_0 = ticks, isMoving = false, isRotating = false\} \\
\text{se } s = idle \wedge i(ticks) = \{absent\} \wedge i(nextmove) = \{absent\} \\
\wedge i(goalticks) = \{absent\} \wedge i(compass) = \{absent\}; \\
\\
Stop, \{front_0 = ticks, isMoving = false, isRotating = false\} \\
\text{se } s = forward \wedge i(goalticks) = \{front\}; \\
\\
Stop, \{front_0 = ticks, isMoving = false, isRotating = false\} \\
\text{se } s = TurnRight \wedge \\
[(i(compass) > 89.9 \wedge i(compass) < 90.1) \vee i(compass) > 179.9 \vee i(compass) < 0.1]; \\
\\
Stop, \{front_0 = ticks, isMoving = false, isRotating = false\} \\
\text{se } s = TurnLeft \wedge \\
[(i(compass) > 89.9 \wedge i(compass) < 90.1) \vee i(compass) > 179.9 \vee i(compass) < 0.1]; \\
\\
Stop, \{front_0 = ticks, isMoving = false, isRotating = false, DONE = true\} \\
\text{se } s = Stop \wedge i(nextmove) = 5; \\
\\
forward, \{front = ticks, isMoving = true\} \\
\text{se } s = Stop \wedge i(nextmove) = 1; \\
\\
TurnLeft, \{degrees+ = 90, isRotating = true\} \\
\text{se } s = Stop \wedge i(nextmove) = 2; \\
\\
TurnRight, \{degrees- = 90, isRotating = true\} \\
\text{se } s = Stop \wedge i(nextmove) = 3;
\end{cases}
$$

#### 4.2.4 EFSM PathPlanning

- **Stati**: {init, getnextmove, moveforward, lookahead, stillforward}
- **Stato Iniziale**: {init}

- **Ingressi**:

$$\begin{aligned} Inputs = \{&in(Navigation.Stop) : bool, \\ &cellsize : [0, \infty), \\ &front, front\_0 : real[0, \infty), \\ &orientation : int[0, 270]\} \end{aligned}$$

- **Uscite**:

$$\begin{aligned} Outputs = \{&nextmove : int\{1, 2, 3, 5\}, \\ &goalticks : int[0, \infty)\} \end{aligned}$$

- **Variabili Locali**:

$$\begin{aligned} LocalVars = \{&i : int[0, \infty), \\ &nextmove2 : int\{1, 2, 3, 5\}, \\ &position : int[2], \\ &ahead : int \\ &moves2, moves : int[], \\ &map2, map : int[20][20]\} \end{aligned}$$

- **Macchina a Stati**:

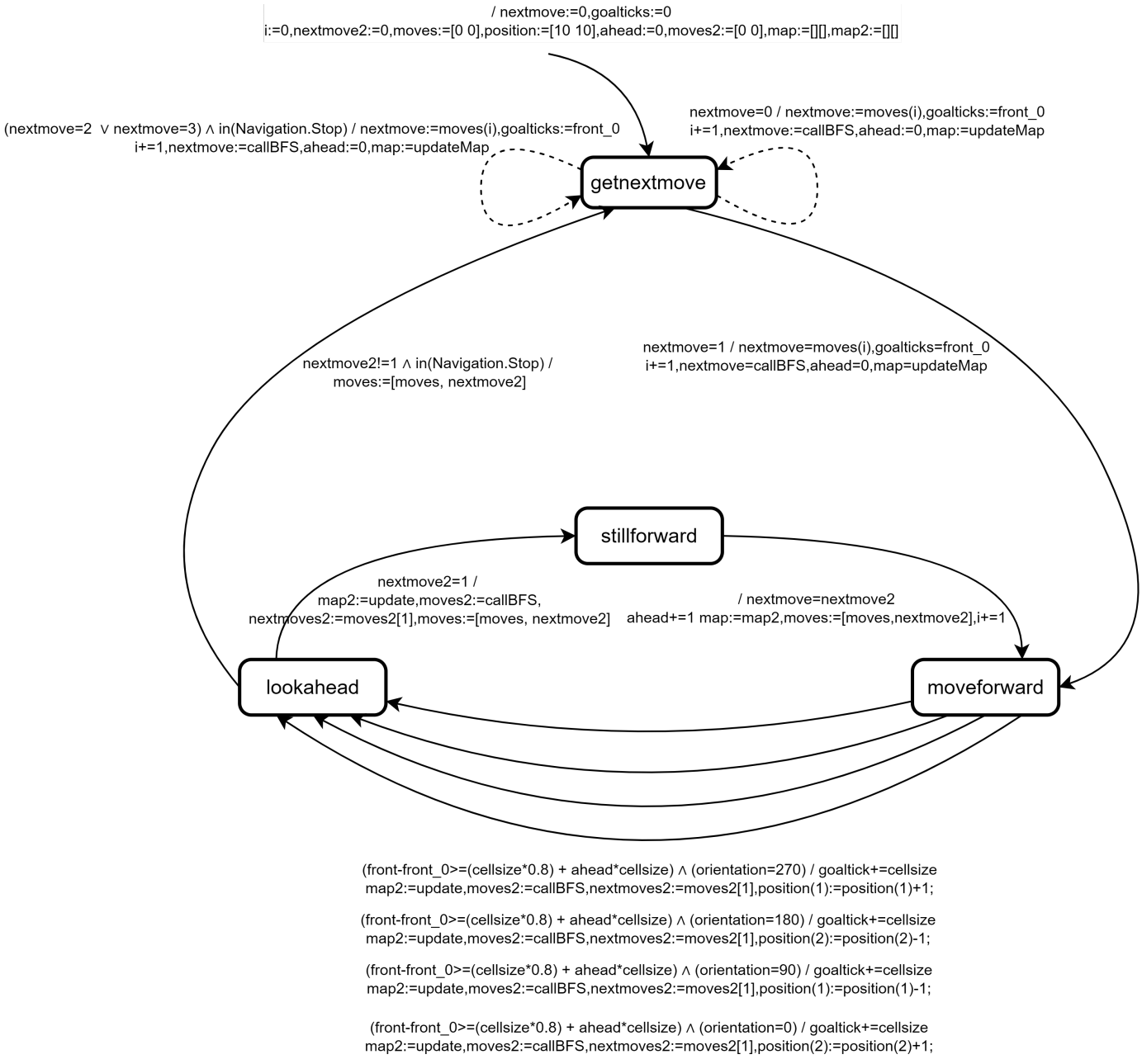



Figure 4: Macchina a stati per PathPlanning

#### 4.2.5 EFSM RobotPose

- **Stati**:

$$RobotPose = \{Case90, Init, Case0, Case180, Stop, Case270\}$$

- **Ingressi**:

$$\begin{aligned} Inputs = \{&IsMoving : bool, \\ &Orientation : int[0, 270], \\ &CurrTicks, StartTicks : int[0, \infty), \\ &DistTicks : const(0.5), \\ &l01, l02, l03, r03, r02, r01 : real[0, 1]\} \end{aligned}$$

- **Uscite**:

$$Outputs = \{X_out, Y_out, O_l01, O_l02, O_l03, O_r03, O_r02, O_r01 : real\}$$

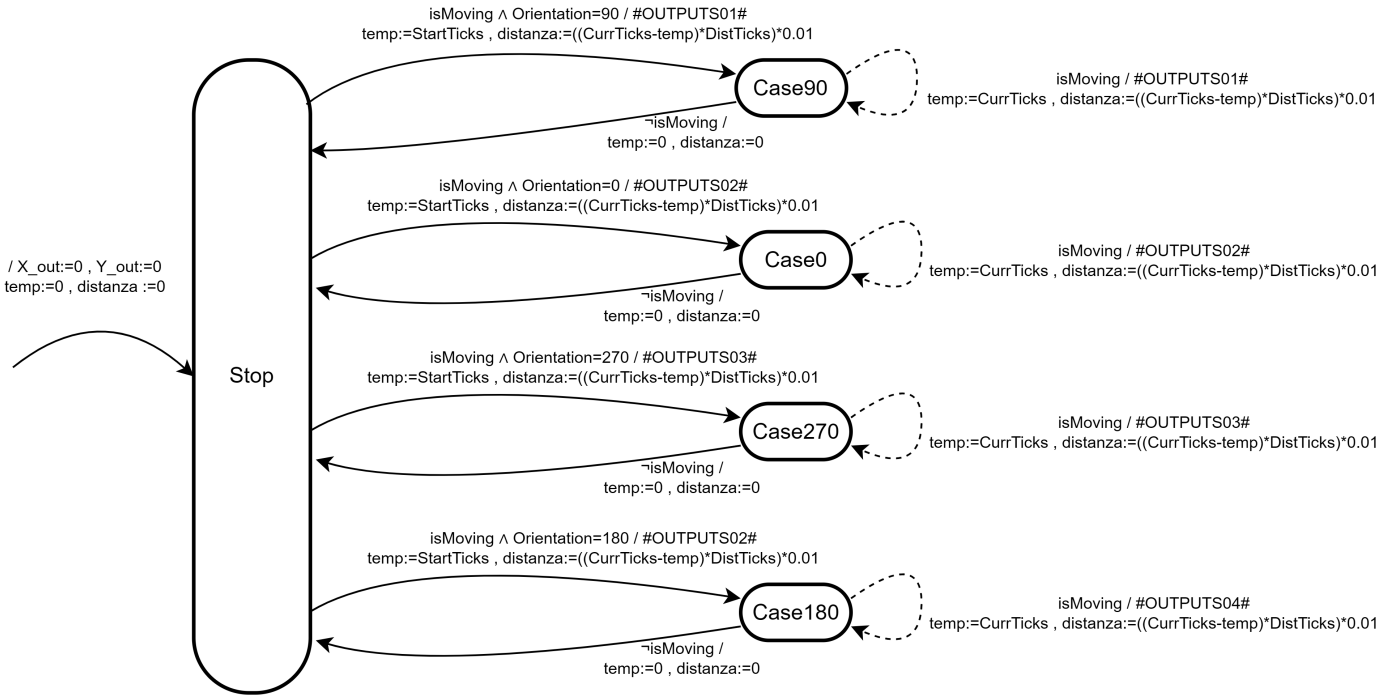
- **Stato Iniziale**:

$$statoIniziale = \{Init\}$$

- **Variabili Locali**:

$$LocalVars = \{distanza : real, temp : int\}$$

- **Macchina a Stati**:




| | |
|---|---|
| **OUTPUTS01** | Y_out = Y_out + distanza;<br>O_l01 = [X_out - l01*sin(1*(pi/8)) , Y_out + l01*cos(1*(pi/8))];<br>O_l02 = [X_out - l02/sqrt(2) , Y_out + l02/sqrt(2)];<br>O_l03 = [X_out - l03*sin(3*(pi/8)) , Y_out + l03*cos(3*(pi/8))];<br>O_r01 = [X_out + r01*sin(1*(pi/8)) , Y_out + r01*cos(1*(pi/8))];<br>O_r02 = [X_out + r02/sqrt(2) , Y_out + r02/sqrt(2)];<br>O_r03 = [X_out + r03*sin(3*(pi/8)) , Y_out + r03*cos(3*(pi/8))]; |
| **OUTPUTS02** | X_out = X_out + distanza;<br>O_l01 = [X_out + l01*cos(1*(pi/8)) , Y_out + l01*sin(1*(pi/8))];<br>O_l02 = [X_out + l02/sqrt(2) , Y_out + l02/sqrt(2)];<br>O_l03 = [X_out + l03*cos(3*(pi/8)) , Y_out + l03*sin(3*(pi/8))];<br>O_r01 = [X_out + r01*cos(1*(pi/8)) , Y_out - r01*sin(1*(pi/8))];<br>O_r02 = [X_out + r02/sqrt(2) , Y_out - r02/sqrt(2)];<br>O_r03 = [X_out + r03*cos(3*(pi/8)) , Y_out - r03*sin(3*(pi/8))]; |
| **OUTPUTS03** | Y_out = Y_out - distanza;<br>O_l01 = [X_out + l01*sin(1*(pi/8)) , Y_out - l01*cos(1*(pi/8))];<br>O_l02 = [X_out + l02/sqrt(2) , Y_out - l02/sqrt(2)];<br>O_l03 = [X_out + l03*sin(3*(pi/8)) , Y_out - l03*cos(3*(pi/8))];<br>O_r01 = [X_out - r01*sin(1*(pi/8)) , Y_out - r01*cos(1*(pi/8))];<br>O_r02 = [X_out - r02/sqrt(2) , Y_out - r02/sqrt(2)];<br>O_r03 = [X_out - r03*sin(3*(pi/8)) , Y_out - r03*cos(3*(pi/8))]; |
| **OUTPUTS01** | X_out = X_out - distanza;<br>O_l01 = [X_out - l01*cos(1*(pi/8)) , Y_out - l01*sin(1*(pi/8))];<br>O_l02 = [X_out - l02/sqrt(2) , Y_out - l02/sqrt(2)];<br>O_l03 = [X_out - l03*cos(3*(pi/8)) , Y_out - l03*sin(3*(pi/8))];<br>O_r01 = [X_out - r01*cos(1*(pi/8)) , Y_out + r01*sin(1*(pi/8))];<br>O_r02 = [X_out - r02/sqrt(2) , Y_out + r02/sqrt(2)];<br>O_r03 = [X_out - r03*cos(3*(pi/8)) , Y_out + r03*sin(3*(pi/8))]; |

Figure 5: Macchina a stati per Pose&Mapping

## 4.3 Formalizzazione dei requisiti LTL del sistema

La Linear Temporal Logic (LTL) è una specifica utilizzata per descrivere il comportamento desiderato di un sistema nel tempo. Utilizza operatori temporali per esprimere proprietà che devono essere soddisfatte da una sequenza di stati. Gli operatori temporali usati sono:

- $G(\phi)$: Globalmente, $\phi$ deve essere vero in tutti i futuri stati.
- $F(\phi)$: Prima o poi $\phi$ deve essere vero in qualche futuro stato.
- $X(\phi)$: Nel prossimo stato, $\phi$ deve essere vero.
- $\phi U \psi$: $\phi$ deve essere vero fino a quando $\psi$ diventa vero.

Per il sistema in esame, i requisiti LTL sono stati formulati come segue:

- **Nella macchina a stati Orientation lo stato Sud non deve mai seguire lo stato Nord e viceversa, così come deve avvenire per gli stati Est e Ovest**:

$$\begin{aligned} G\Big( & (NORD \implies \neg X(SUD)) \wedge \\ & (SUD \implies \neg X(NORD)) \wedge \\ & (EST \implies \neg X(OVEST)) \wedge \\ & (OVEST \implies \neg X(EST)) \Big) \end{aligned}$$

- **Il robot passa sempre allo stato Stop dopo ogni rotazione**:

$$G((TurnLeft \vee TurnRight) \implies X(Stop))$$

- **Il robot non può avanzare e ruotare contemporaneamente**:

$$G((forward \implies \neg isRotating) \wedge ((TurnLeft \vee TurnRight) \implies \neg isMoving))$$

- **isMoving e isRotating non sono mai 1 contemporaneamente**:

$$G(\neg(isMoving = 1 \wedge isRotating = 1))$$

- **Se isMoving=0 non si è mai in Case0, Case90, Case180 e Case270**:

$$G((isMoving = 0) \implies \neg(Case0 \vee Case90 \vee Case180 \vee Case270))$$

- **Dopo Case0, Case90, Case180 e Case270 il prossimo stato è sempre Stop**:

$$\begin{aligned} & G(Case0 \implies X(Stop)) \\ & G(Case90 \implies X(Stop)) \\ & G(Case180 \implies X(Stop)) \\ & G(Case270 \implies X(Stop)) \end{aligned}$$

# 5 Implementazione

Il progetto è sviluppato in MATLAB utilizzando Simulink per la modellazione e Stateflow per implementare le Extended Finite State Machines (EFSM) e le Finite State Machines (FSM). L'ambiente di sviluppo include il Robotics Playground, che fornisce strumenti avanzati per la simulazione di robot mobili, basati su Simscape Multibody, per una simulazione accurata della fisica dei sistemi coinvolti. Questi strumenti permettono di concentrare lo sviluppo sulla logica di controllo piuttosto che sui dettagli del sistema fisico, facilitando così la creazione di un sistema robusto e testabile.

## 5.1 Sviluppo del sistema

Il sistema è stato sviluppato integrando le componenti chiave in un modello Simulink.

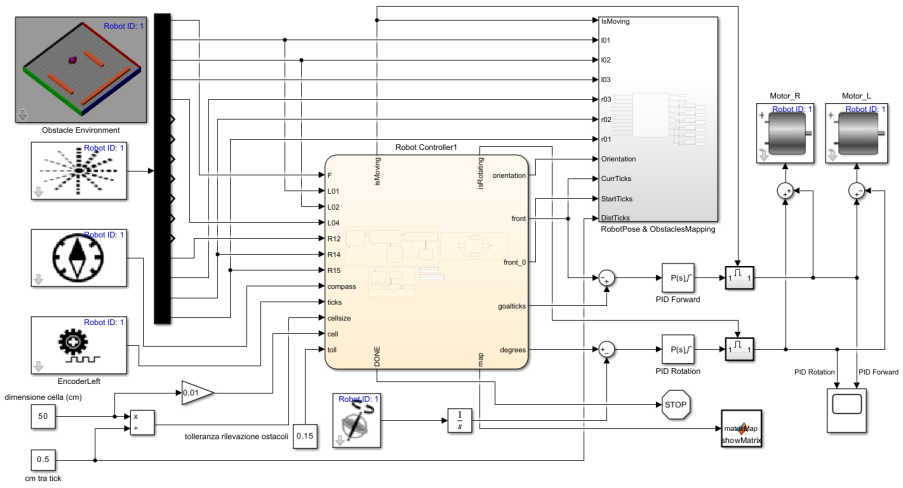



Figure 6: Modello Simulink del sistema robotico.

La struttura del sistema e il suo funzionamento sono descritti di seguito.

- **Robot controller:** Questo sottosistema implementa la logica di navigazione del robot, utilizzando gli algoritmi di pianificazione del percorso e di controllo dell'orientamento, della direzione e del movimento del robot. Viene gestito anche il modello dell'ambiente esterno che si aggiorna a tempo di esecuzione in base alle rilevazioni dei sensori e permette la navigazione di spazi arbitrari.
- **Pose & Mapping:** Questo sottosistema raccoglie e processa i dati dei sensori per determinare la posizione del robot e per mappare l'ambiente circostante.

- **Sensori e attuatori:** Include tutti i sensori utilizzati per raccogliere dati ambientali (giroscopio, bussola, Lidar, encoder) e gli attuatori che permettono il movimento del robot.
- **PID:** Controllori PID utilizzati per regolare la velocità dei motori e l'orientamento del robot, garantendo che il robot segua con precisione le traiettorie pianificate.
- **Elaborazione dei dati:** uno script esterno elabora i dati raccolti dai sensori per generare la mappa dettagliata dell'ambiente esplorato.

Il sistema opera nel seguente modo:

1. Il robot inizia la navigazione ricevendo i dati iniziali dai sensori.
2. RobotController elabora questi dati utilizzando gli algoritmi di navigazione e pianificazione del percorso.
3. I controllori PID regolano il movimento del robot per seguire le traiettorie pianificate.
4. RobotPose raccoglie continuamente dati per aggiornare la posizione del robot e mappare l'ambiente.
5. Gli script di elaborazione dati processano le informazioni raccolte per generare una mappa dettagliata dell'ambiente esplorato.

### 5.1.1 Robot controller

Il sottosistema Stateflow RobotController è responsabile della logica di navigazione, utilizzando algoritmi di pianificazione del percorso e di controllo dell'orientamento, della direzione e del movimento del robot. Il controller gestisce anche il modello dell'ambiente esterno, che si aggiorna in tempo reale in base alle rilevazioni dei sensori, permettendo così la navigazione di spazi arbitrari.

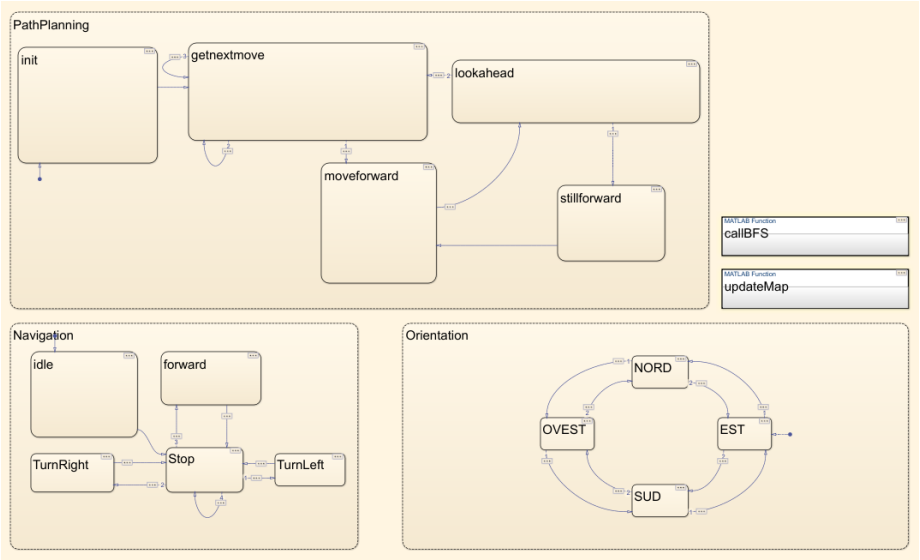



Figure 7: Schermata del Robot Controller in Stateflow.

Il funzionamento del robot controller si basa sulla creazione di una mappa (matrice) che funge da modello del mondo esterno per la navigazione. Questa mappa ha celle di grandezza *cellsize* (attualmente impostata uguale alla dimensione del robot), ognuna delle quali può assumere quattro valori:

- **0**: cella non valutata.
- **1**: cella libera (il Lidar non rileva ostacoli).
- **2**: cella visitata.
- **3**: cella occupata.

All'inizio, tutte le celle sono impostate a 0. Durante l'esecuzione, il sistema controlla quali celle sono libere utilizzando la funzione *updateMap* e genera la sequenza di movimenti necessari per raggiungere la cella libera più vicina (valore 1) tramite la funzione *callBFS*. I movimenti consentiti sono: girare di 90 gradi a destra, girare di 90 gradi a sinistra, e muoversi avanti di una cella.

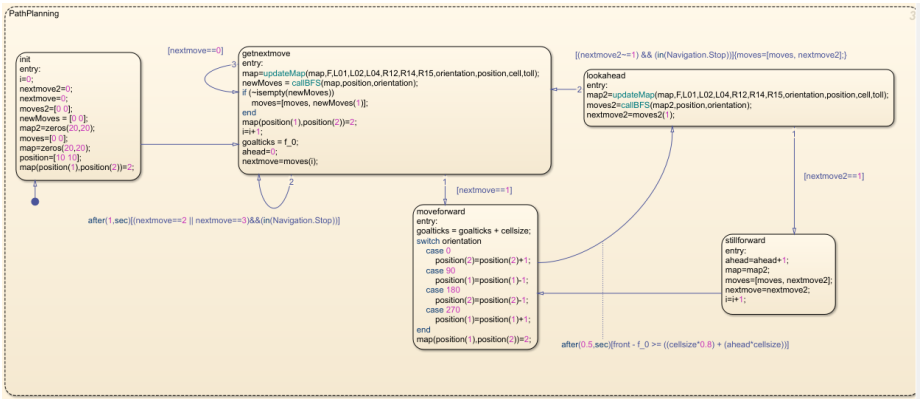



Figure 8: Sottosistema PathPlanning in Stateflow.

Il sottosistema **PathPlanning** è responsabile della generazione della sequenza di movimenti. Include gli stati:

- *init*: inizializzazione del sistema.
- *getnextmove*: calcolo della prossima mossa.
- *moveforward*: aggiornamento della posizione sulla mappa interna di navigazione, a seguito del movimento in avanti.
- *lookahead*: valutazione preventiva della prossima mossa, per un controllo senza interruzioni nel caso il prossimo movimento sia un ulteriore forward.
- *stillforward*: aggiornamento del goal per il movimento in avanti nel caso il lookahead abbia dato esito positivo.

Gli stati *moveforward*, *lookahead* e *stillforward* servono per aggiornare in maniera predittiva il percorso del robot nell'ambiente, per evitare le interruzioni date da una valutazione effettuata dopo il compimento del movimento. Grazie a questi stati quindi, l'aggiornamento dei goalticks, variabile che funge da riferimento al PID Controller, avviene in maniera dinamica durante il movimento stesso.

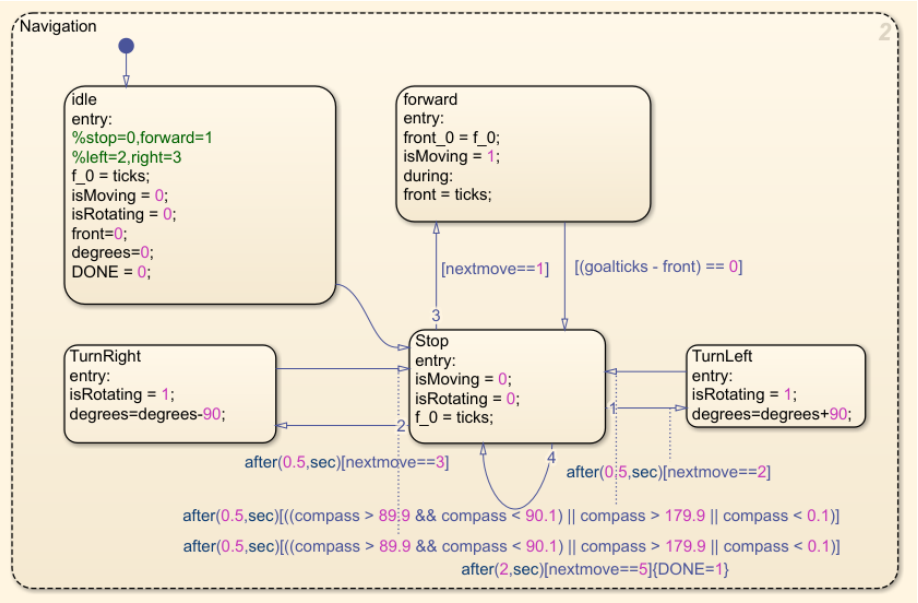



Figure 9: Sottosistema Navigation in Stateflow.

Il sottosistema **Navigation** aggiorna i riferimenti di *degrees* e *goalticks* (dell'encoder, quindi distanza) da fornire ai controllori PID. Include gli stati:

- *idle*: stato di inattività, inizializzazione.
- *forward*: movimento in avanti.
- *Stop*: arresto del movimento.
- *TurnRight*: svolta a destra.
- *TurnLeft*: svolta a sinistra.

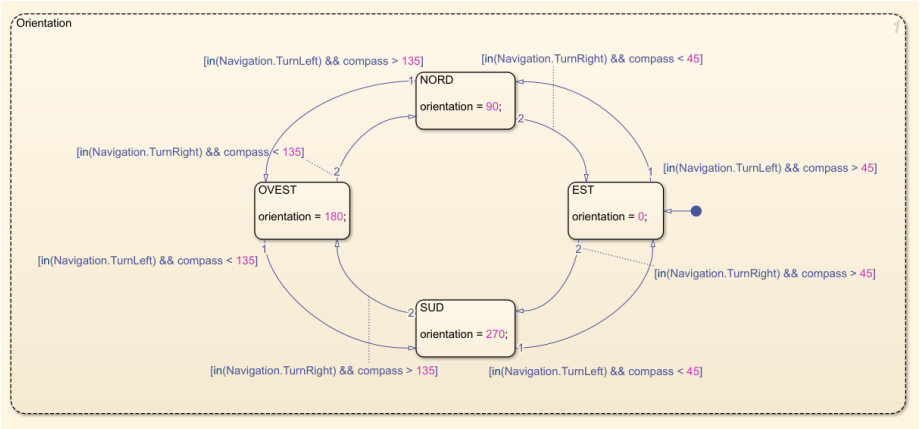



Figure 10: Sottosistema Orientation in Stateflow.

Il sottosistema **Orientation** tiene traccia dell'orientamento del robot in base ai movimenti effettuati e alla bussola. Questo sottosistema è necessario in quanto il dominio dei dati ottenuti dalla bussola è [0;180], quindi non sufficiente per sapere l'effettivo orientamento del robot. Gli stati sono:

- *NORD*: orientamento a nord.
- *OVEST*: orientamento a ovest.
- *EST*: orientamento a est.
- *SUD*: orientamento a sud.

### 5.1.2 Algoritmo di Navigazione BFS

L'algoritmo di navigazione utilizzato per determinare il percorso del robot è basato su una ricerca in ampiezza (Breadth-First Search, BFS). Questo algoritmo esplora i possibili movimenti del robot per trovare la cella libera più vicina e genera una sequenza di azioni necessarie per raggiungerla.

La funzione principale che implementa l'algoritmo di navigazione è *bfs_find_closest*. Questa funzione prende in ingresso la griglia di navigazione, la posizione iniziale del robot (*startPos*) e l'orientamento iniziale (*startOrientation*).

**Inizializzazione** La funzione inizia controllando se la posizione e l'orientamento attuali del robot sono uguali a quelli della chiamata precedente. Se sono uguali, la funzione restituisce un percorso vuoto, indicando che non è necessario alcun movimento. Altrimenti, aggiorna le posizioni memorizzate.

```
function path = bfs_find_closest(grid, startPos, startOrientation)
    persistent vecchiaPosizione vecchiaRotazione
    if(isempty(vecchiaPosizione))
```

```
        vecchiaPosizione = [37 73];
        vecchiaRotazione = 420;
    end
    if (isequal(startPos, vecchiaPosizione) &&
    ↪  isequal(startOrientation, vecchiaRotazione))
        path = [];
        return;
    end
    vecchiaPosizione = startPos;
    vecchiaRotazione = startOrientation;
```

**Definizione delle direzioni** Viene creata una mappa delle direzioni possibili basata sull'orientamento del robot (0, 90, 180, 270 gradi).

```
    directions = containers.Map({0, 90, 180, 270}, {[0, 1]; [-1,
    ↪  0]; [0, -1]; [1, 0]});
    queue = {{startPos, startOrientation, []}};
    visited = {{startPos, startOrientation}};
```

**Coda e Visitate** La funzione utilizza una coda (*queue*) per gestire le posizioni da esplorare e un insieme (*visited*) per tracciare le posizioni già visitate.

```
    while ~isempty(queue)
        front = queue{1};
        queue(1) = [];
        currentPos = front{1};
        currentOrientation = front{2};
        pathCode = front{3};

        % Try to advance
        movement = directions(currentOrientation);
        newPos = [currentPos(1) + movement(1), currentPos(2) +
        ↪  movement(2)];
```

**Esplorazione** La funzione entra in un ciclo *while* che continua fino a quando la coda non è vuota. Per ogni posizione nella coda, prova ad avanzare nella direzione corrente:

```
        if newPos(1) >= 1 && newPos(1) <= size(grid, 1) &&
        ↪  newPos(2) >= 1 && newPos(2) <= size(grid, 2)
            if grid(newPos(1), newPos(2)) == 1
                path = [pathCode, 1]; % Advance is 1
                return;
            elseif grid(newPos(1), newPos(2)) ~= 0 &&
            ↪  grid(newPos(1), newPos(2)) ~= 3
```

```
                if all(cellfun(@(x) ~isequal(x{1}, newPos) || x{2}
                ↪  ~= currentOrientation, visited))
                    visited{end+1} = {newPos, currentOrientation};
                    queue{end+1} = {newPos, currentOrientation,
                    ↪  [pathCode, 1]}; % 1 for advance
                end
            end
        end
```

Se la nuova posizione è all'interno dei limiti della griglia e la cella è libera (valore 1), la funzione restituisce il percorso aggiornato con il codice di avanzamento (1). Se la cella non è occupata (valori diversi da 0 e 3) e non è stata visitata con lo stesso orientamento, viene aggiunta alla coda per ulteriori esplorazioni.

```
        % Rotate left
        newOrientation = mod(currentOrientation + 90, 360);
        if all(cellfun(@(x) ~isequal(x{1}, currentPos) || x{2} ~=
        ↪  newOrientation, visited))
            visited{end+1} = {currentPos, newOrientation};
            queue{end+1} = {currentPos, newOrientation, [pathCode,
            ↪  2]}; % 2 for rotate left
        end

        % Rotate right
        newOrientation = mod(currentOrientation - 90, 360);
        if all(cellfun(@(x) ~isequal(x{1}, currentPos) || x{2} ~=
        ↪  newOrientation, visited))
            visited{end+1} = {currentPos, newOrientation};
            queue{end+1} = {currentPos, newOrientation, [pathCode,
            ↪  3]}; % 3 for rotate right
        end
    end
```

**Nessun Percorso Trovato** Se nessun percorso verso una cella libera viene trovato, la funzione restituisce un codice di errore (5 5 5), indicando che non è stato possibile trovare un percorso.

```
    path = [5 5 5]; % Return 5 if no path is found
    clear vecchiaRotazione;
    clear vecchiaPosizione;
end
```

Questo algoritmo assicura che il robot trovi il percorso più breve verso la cella libera più vicina, tenendo conto delle sue capacità di movimento e dell'ambiente circostante.

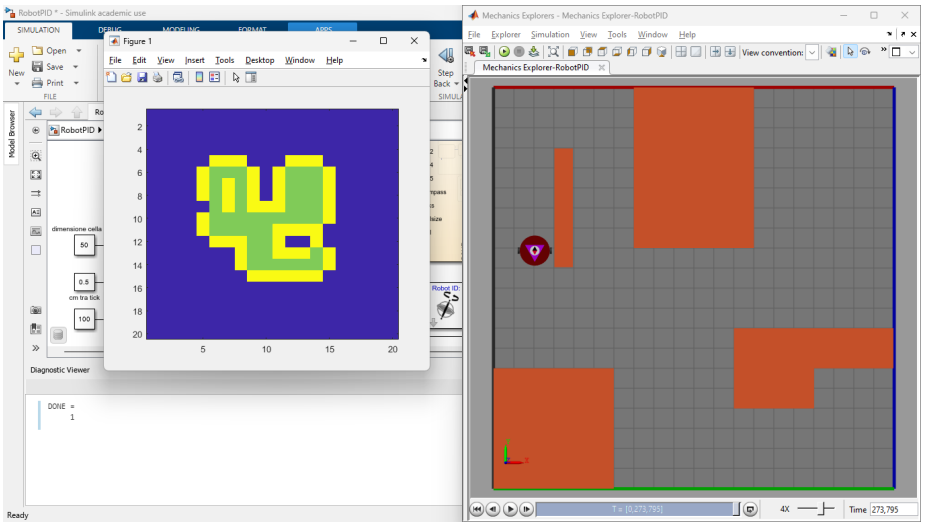


Figure 11: Utilizzo della mappa per la navigazione.

### 5.1.3 Pose & Mapping

Il sottosistema Stateflow Pose & Mapping è responsabile della creazione delle coordinate degli ostacoli presenti nella stanza in base a posizione e orientamento del robot, raccogliendo i dati dal LIDAR e utilizzando formule di geometria euclidea.

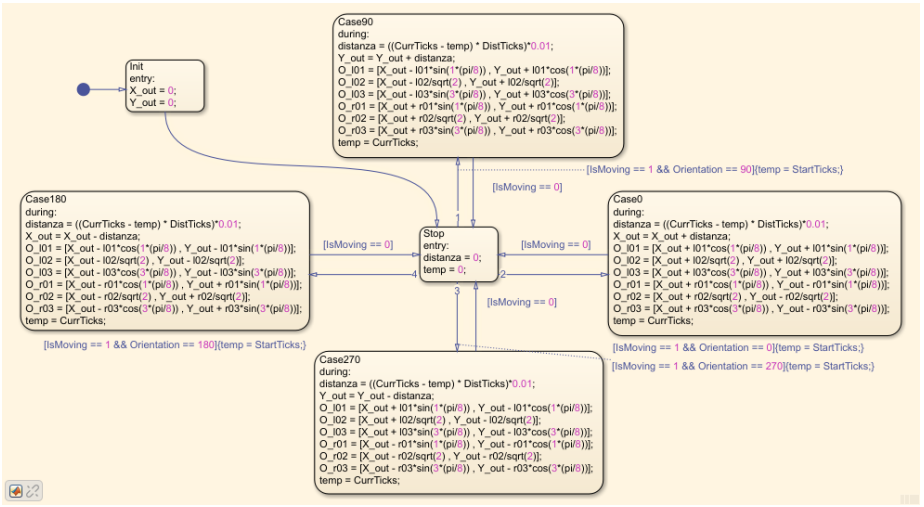



Figure 12: Sottosistema Pose&Mapping in Stateflow.

### 5.1.4 Sensori e attuatori

Il sistema utilizza una varietà di sensori e attuatori forniti dal Robotics Playground di MATLAB per raccogliere dati ambientali e controllare i movimenti del robot.

**Sensori:**

- **Gyro Sensor:** Questo sensore misura la velocità angolare in gradi al secondo (deg/s).
- **Encoder Sensor:** Questo sensore collegato ad un rispettivo motore del robot fornisce un conteggio dei tick proporzionali alla rotazione della ruota collegata al motore. I ticks possono aumentare o diminuire in base alla direzione della rotazione della ruota. I ticks per rotazione sono stati impostati al valore 50. L'encoder è fondamentale per misurare la distanza percorsa dal robot, fornendo feedback continuo al controllore PID.
- **Compass Sensor:** Questo sensore rileva la rotazione in gradi dal centro del robot rispetto al sistema di coordinate dell'ambiente simulato. Viene utilizzato per mantenere e aggiornare l'orientamento del robot durante la navigazione, ma è limitato nel range, che varia da 0 a 180, per cui è usato insieme ad una logica a stati che traccia la rotazione.
- **Lidar:** Questo sensore rileva gli ostacoli nell'ambiente circostante e mappa le distanze dagli oggetti. La distanza massima a cui è possibile rilevare un ostacolo può essere impostata manualmente essendo un sensore simulato, quindi è stato scelto un valore di 1 metro per riflettere un utilizzo reale e inserire un constraint realistico al funzionamento del sistema.

**Attuatori:**

- **Motori:** Questo blocco imposta la velocità di un motore del robot. L'input è un valore compreso tra -127 e 127, dove il segno determina la direzione della rotazione. I motori sono controllati dai segnali di uscita dei PID, che regolano la velocità e la direzione per eseguire i movimenti pianificati.

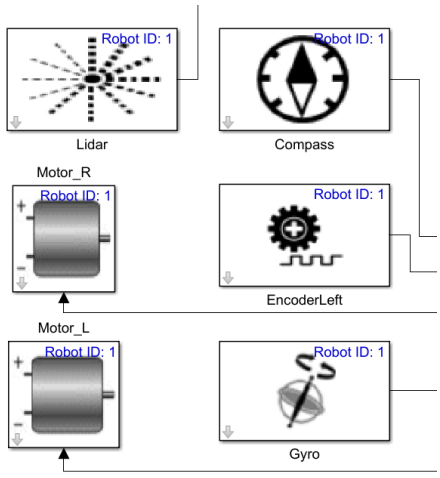



Figure 13: Blocchi Simulink dei sensori e dei motori utilizzati.

### 5.1.5 PID

I controllori PID sono utilizzati per regolare la velocità dei motori e l'orientamento del robot, garantendo che il robot segua con precisione le traiettorie pianificate. Due PID distinti sono implementati per gestire il movimento rettilineo e la rotazione angolare del robot.

**PID Forward:** Questo controllore è responsabile del movimento rettilineo del robot. Il riferimento (*setpoint*) per questo PID è fornito dalla variabile *goalticks*, calcolata nel sottosistema *PathPlanning*. La lettura dal sensore (*process variable*) proviene dall'encoder che misura i ticks cumulativi percorsi dalle ruote del robot. I parametri del PID Forward sono:

- **Proporzionale (P)**: 16

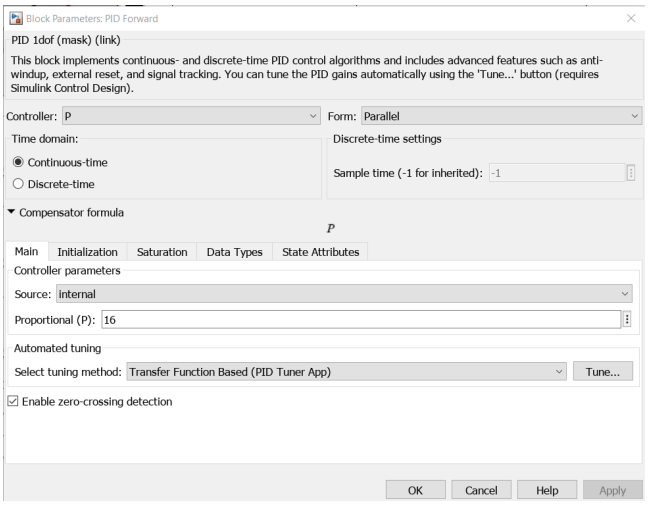



Figure 14: Parametri del PID Forward.

**PID Rotation:** Questo controllore gestisce la rotazione angolare del robot. Il riferimento (*setpoint*) per questo PID è dato dalla variabile *degrees*, che incrementa o decrementa di 90 gradi in base alle decisioni prese dal *Robot Controller*. La lettura dal sensore (*process variable*) proviene dall'integratore collegato alla velocità angolare del robot. I parametri del PID Rotation sono:

- **Proporzionale (P)**: 10

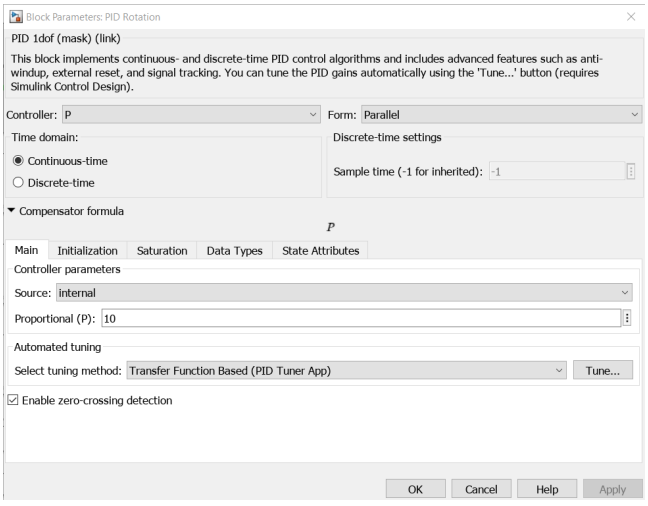



Figure 15: Parametri del PID Rotation.

La combinazione dei due controllori PID permette al robot di muoversi e

orientarsi con precisione all'interno dell'ambiente simulato, seguendo i percorsi pianificati e reagendo in tempo reale alle variazioni rilevate dai sensori. Nella figura seguente viene mostrata l'uscita del sistema tramite uno scope posto sull'integratore del giroscopio, dopo aver dato come riferimento di orientamento al PID il valore 90 gradi.

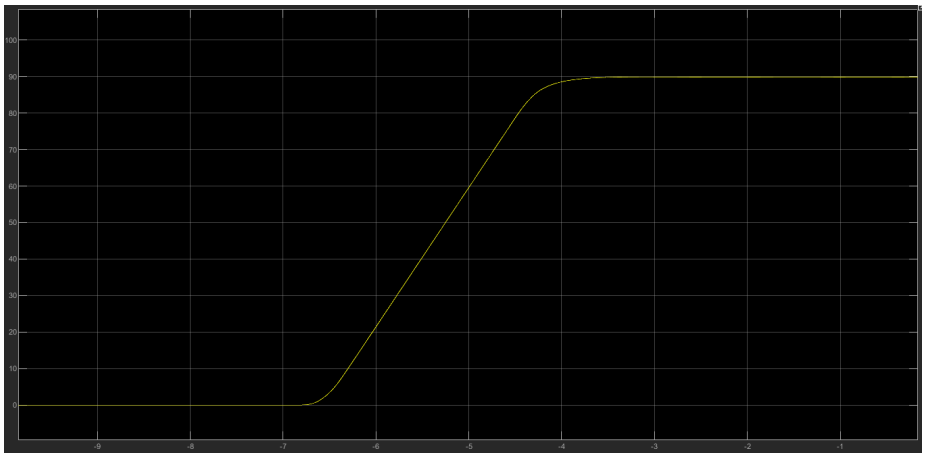


Figure 16: Risposta del sistema controllato.

### 5.1.6 Elaborazione dei dati

Lo script esterno *OccypancyMap.m* processa i dati generati dal sottosistema Stateflow Pose & Mapping per creare una mappa dettagliata dell'ambiente esplorato dal robot. Questo script esegue operazioni di filtraggio, analisi e rappresentazione grafica dei dati, permettendo una visualizzazione chiara delle aree esplorate e degli ostacoli rilevati.

**Raccolta dei dati** La prima parte dello script raccoglie i dati relativi alle coordinate X e Y rilevate dai sensori del robot. I dati vengono combinati utilizzando la funzione *cat* di MATLAB.

```
X = cat(1, out.L01_X, out.L02_X, out.L03_X, out.R01_X, out.R02_X,
↪  out.R03_X);
Y = cat(1, out.L01_Y, out.L02_Y, out.L03_Y, out.R01_Y, out.R02_Y,
↪  out.R03_Y);
robotX = out.Xrobot;
robotY = out.Yrobot;
```

**Filtraggio dei dati** I dati raccolti vengono filtrati per rimuovere i valori NaN (Not a Number), che possono rappresentare rilevazioni non valide o mancanti.

```
X = X(~isnan(X));
Y = Y(~isnan(Y));
```

**Visualizzazione preliminare dei dati** Successivamente, i dati filtrati vengono visualizzati in un grafico per una prima ispezione, insieme alla traiettoria del robot.

```
figure;
plot(X, Y, '.', robotX, robotY, "--");
grid on;
```

**Normalizzazione delle coordinate** Le coordinate X e Y vengono normalizzate per assicurare che tutte le rilevazioni siano positive, aggiustando i valori minimi se necessario.

```
[minX,~] = bounds(X);
[minY,~] = bounds(Y);
if minX < 0
    X = X + abs(minX);
    robotX = robotX + abs(minX);
end
if minY < 0
    Y = Y + abs(minY);
    robotY = robotY + abs(minY);
end
```

**Creazione e visualizzazione della mappa di occupazione** Lo script crea una mappa di occupazione binaria utilizzando la funzione *binaryOccupancyMap* di MATLAB, che permette di rappresentare le aree esplorate e gli ostacoli rilevati. Infine, la mappa di occupazione viene visualizzata.

```
[~,maxX] = bounds(X);
[~,maxY] = bounds(Y);

map = binaryOccupancyMap(maxX, maxY, 10);
setOccupancy(map, [X Y], 1);
figure;
show(map);
```

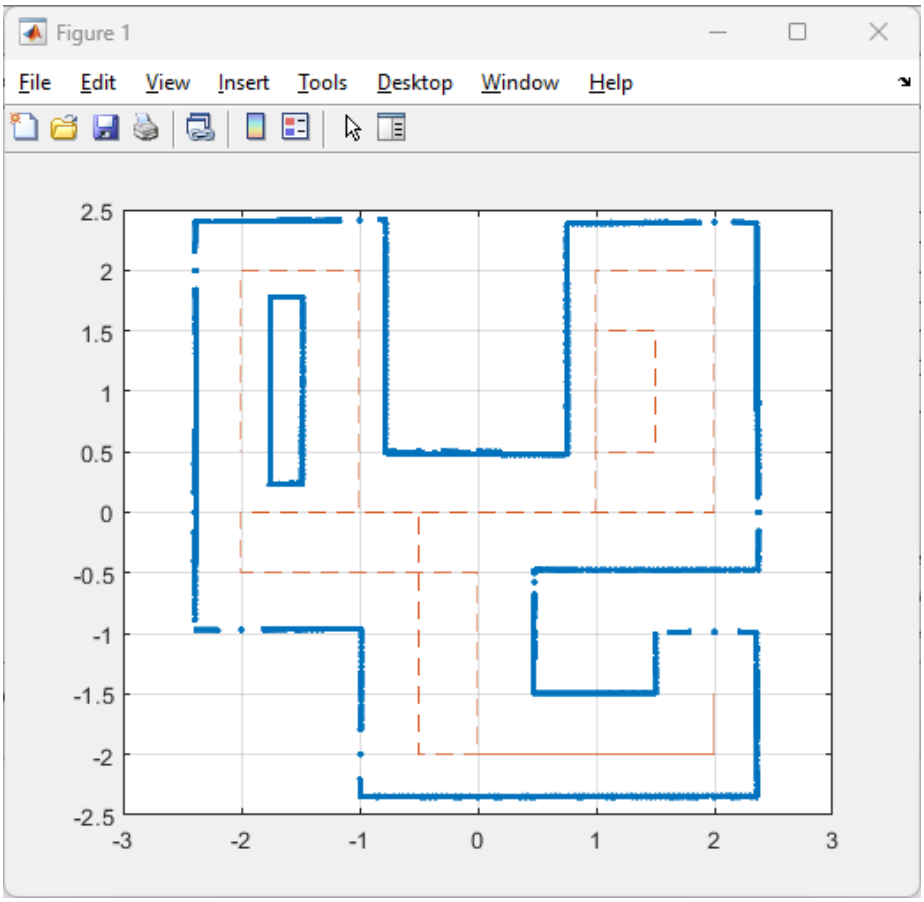



Figure 17: Mappa per prima ispezione, con traiettoria del robot.

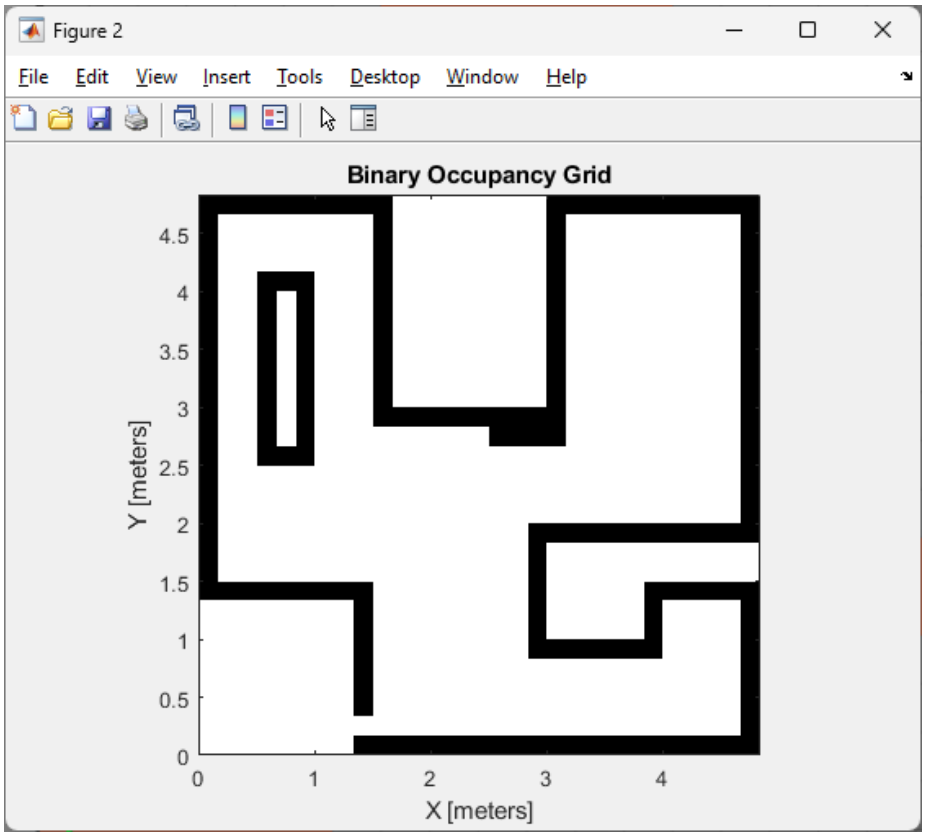



Figure 18: Mappa finale di occupazione binaria.

# 6 Verifica dei requisiti LTL del sistema

Di seguito vengono presentate le tracce di esempio per ogni requisito LTL formulato per il sistema.

## 6.1 Requisito 1: Nella macchina a stati Orientation lo stato Sud non deve mai seguire lo stato Nord e viceversa, così come deve avvenire per gli stati Est e Ovest

**Formula**:

$$G\Big((NORD \implies \neg X(SUD)) \wedge (SUD \implies \neg X(NORD)) \wedge (EST \implies \neg X(OVEST)) \wedge (OVEST \implies \neg X(EST))\Big) \quad (1)$$

**Traccia di esempio**:

$$EST \rightarrow NORD \rightarrow OVEST \rightarrow SUD \rightarrow EST$$

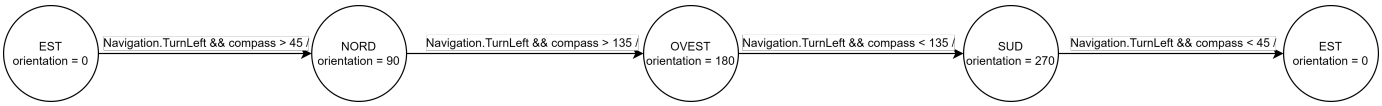



Figure 19: Specifica 1

Ogni transizione rispetta il requisito perché non ci sono transizioni dirette tra NORD e SUD, né tra EST e OVEST.

## 6.2 Requisito 2: Il robot passa sempre allo stato Stop dopo ogni rotazione

**Formula**:

$$G\Big((TurnLeft \vee TurnRight) \implies X(Stop)\Big) \quad (2)$$

**Traccia di esempio**:

$$TurnLeft \rightarrow Stop \rightarrow TurnRight \rightarrow Stop \rightarrow forward$$

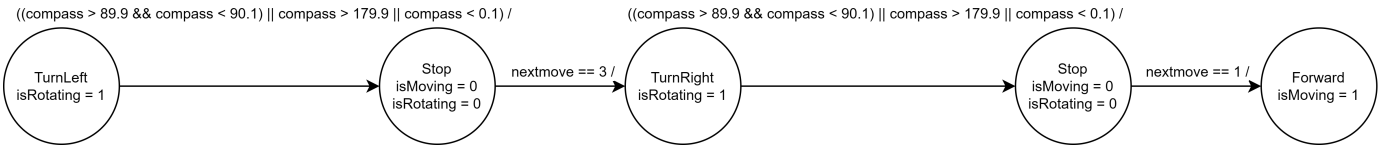



Figure 20: Specifica 2

Ogni rotazione (TurnLeft e TurnRight) è seguita dallo stato Stop.

## 6.3 Requisito 3: Il robot non può avanzare e ruotare contemporaneamente

**Formula**:

$$G((forward \implies \neg isRotating) \wedge ((TurnLeft \vee TurnRight) \implies \neg isMoving)) \tag{3}$$

**Traccia di esempio**:

$$forward(isMoving = 1, isRotating = 0) \rightarrow Stop(isMoving = 0, isRotating = 0) \rightarrow TurnLeft(isMoving = 0, isRotating = 1)$$

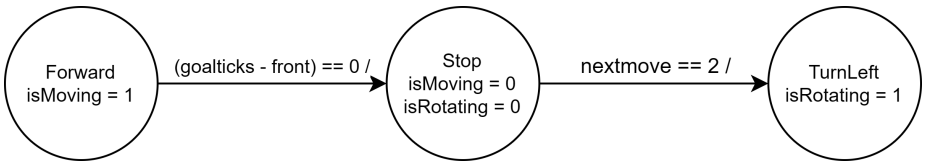



Figure 21: Specifica 3

In ogni stato forward, `isRotating` è 0.

## 6.4 Requisito 4: isMoving e isRotating non sono mai 1 contemporaneamente

**Formula**:

$$G\Big(\neg(isMoving = 1 \wedge isRotating = 1)\Big) \tag{4}$$

**Traccia di esempio**:

$$forward(isMoving = 1, isRotating = 0) \rightarrow Stop(isMoving = 0, isRotating = 0) \rightarrow TurnLeft(isMoving = 0, isRotating = 1) \rightarrow Stop(isMoving = 0, isRotating = 0) \rightarrow forward(isMoving = 1, isRotating = 0)$$

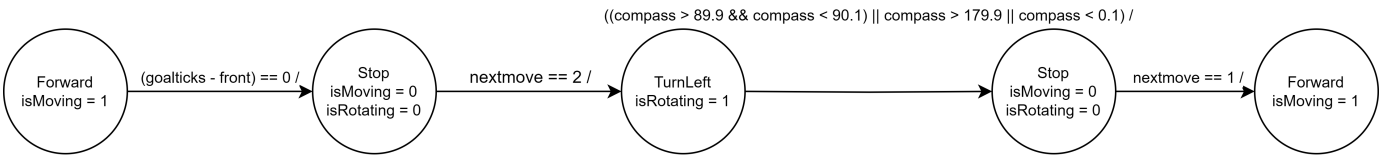



Figure 22: Specifica 4

`isMoving` e `isRotating` non sono mai 1 contemporaneamente in nessuno stato.

## 6.5 Requisito 5: Se isMoving=0 non si è mai in Case0, Case90, Case180 e Case270

**Formula**:

$$G((isMoving = 0) \implies \neg(Case0 \vee Case90 \vee Case180 \vee Case270)) \quad (5)$$

**Traccia di esempio**:

$$Case0 \rightarrow Stop \rightarrow Case90 \rightarrow Stop$$

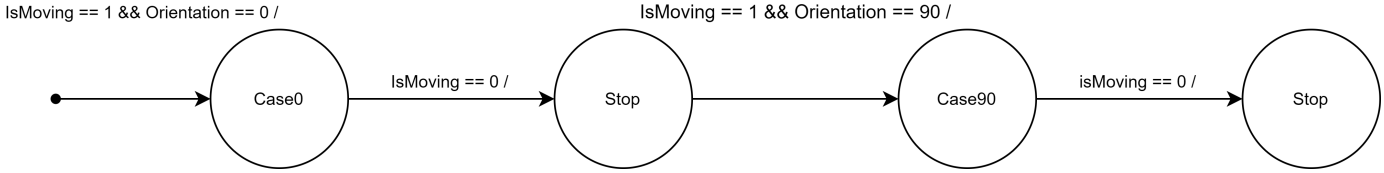



Figure 23: Specifica 5

Quando`isMoving = 0` si passa sempre allo stato `Stop`

## 6.6 Requisito 6: Dopo Case0, Case90, Case180 e Case270 il prossimo stato è sempre Stop

**Formula**:

$$G\Big((isMoving = 0) \implies \neg(Case90 \vee Case0 \vee Case180 \vee Case270)\Big) \quad (6)$$

$$G\Big(Case90 \implies X(Stop)\Big) \quad (7)$$

$$G\Big(Case0 \implies X(Stop)\Big) \quad (8)$$

$$G\Big(Case180 \implies X(Stop)\Big) \quad (9)$$

$$G\Big(Case270 \implies X(Stop)\Big) \quad (10)$$

**Traccia di esempio**:

$$Case90 \rightarrow Stop \rightarrow Case0 \rightarrow Stop \rightarrow Case270 \rightarrow Stop \rightarrow Case180$$

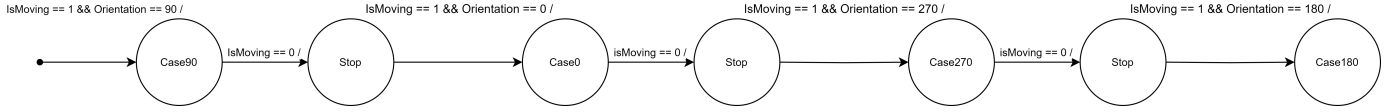



Figure 24: Specifica 6

Ogni stato `Case90`, `Case0`, `Case180` e `Case270` è sempre seguito dallo stato `Stop`.

# 7 Conclusione e possibili sviluppi futuri

## 7.1 Conclusione

Grazie all'utilizzo di MATLAB e Simulink, è stato possibile integrare componenti chiave come sensori, attuatori, controllori PID e sottosistemi Stateflow per creare un robot a guida differenziale autonomo, per una mappatura efficiente. Le conoscenze acquisite durante il corso, in particolare riguardo all'uso di Stateflow per la gestione delle macchine a stati, sono state fondamentali per sviluppare un sistema che combina efficacemente i vari moduli di controllo e sensoriali.

## 7.2 Sviluppi futuri

Nonostante i risultati ottenuti, ci sono diversi ambiti in cui il sistema potrebbe essere ulteriormente migliorato:

- **Controllori PID:** Attualmente, i controllori PID utilizzano solo il termine proporzionale. L'inclusione dei termini integrale e derivativo potrebbero portare ad un sostanziale miglioramento dei tempi necessari per raggiungere in maniera stabile i riferimenti.
- **Algoritmo di pianificazione del percorso:** L'algoritmo BFS, sebbene efficace, potrebbe essere sostituito da algoritmi più *compute efficient* come A* (A-star), che offre performance comparabili ad un minor costo computazionale. Questo permetterebbe di alleggerire situazioni limite per cui BFS debba valutare troppi percorsi per via della grandezza della mappa.
- **Implementazione in ambienti reali:** Infine, la transizione dal modello simulato a un robot fisico reale rappresenta un passo significativo. Ciò richiederebbe l'adattamento e la calibrazione dei controllori e dei sensori per funzionare in un ambiente reale, affrontando sfide aggiuntive come rumore dei sensori e variazioni ambientali.