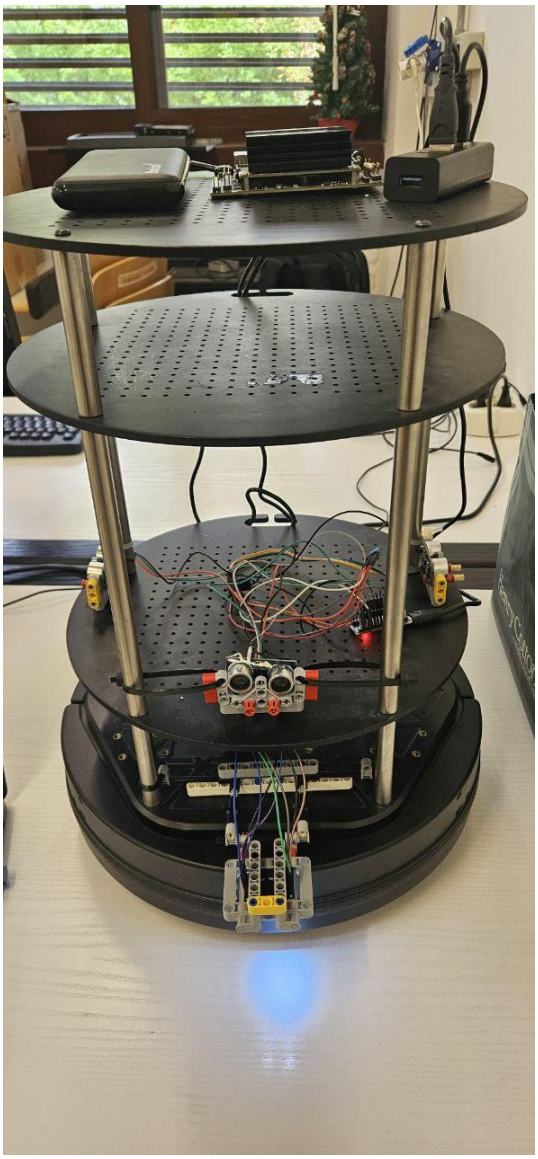
UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Corso di Laurea Magistrale in Ingegneria Informatica
Intelligent Systems and Robotics Laboratory
Prof. Giovanni De Gasperis

KOBUKI ESCAPE



Di Lazzaro Fabio, Ruocco Giulia, Sciarra Davide, Sicchio Giovanni

Sommario

Membri del gruppo e ruoli ... 3
Introduzione ... 4
Robot Virtuale ... 4
Tecnologie ... 4
CoppeliaSim ... 4
Python ... 4
Docker ... 6
MQTT ... 6
Architettura Logica ... 7
Struttura del codice ... 8
Setup ambiente Docker ... 10
Setup topic MQTT ... 12
Simulazione Virtuale ... 13
Agent ... 13
Scena di simulazione ... 16
Simulazione ... 18
Robot Fisico ... 22
Tecnologie Software ... 22
Python ... 22
Docker ... 22
MQTT ... 22
Tecnologie Hardware ... 23
Architettura Hardware ... 27
Architettura Logica ... 29
Struttura del Codice ... 30
GUI ... 36
Simulazione Fisica ... 37
Setup dell'ambiente ... 37
Setup robot ... 38
Avvio tramite GUI ... 38
Simulazione ... 39
Conclusioni ... 42

Membri del gruppo e ruoli

Di Lazzaro Fabio

- Implementazione Python
- Hardware & Setting
- Testing
- Linux
- Documentazione

Ruocco Giulia

- Documentazione
- Implementazione Python
- Hardware & Setting
- Tarature
- Testing

Sciarra Davide

- Debugging
- Implementazione Python
- Hardware & Setting
- Tarature
- Testing

Sicchio Giovanni

- Arduino
- Implementazione Python
- Testing
- Docker
- Debugging

Introduzione

Il progetto ha come obiettivo quello di realizzare un agente intelligente in grado di muoversi autonomamente all'interno di un labirinto. La realizzazione ha previsto prima una implementazione virtuale e successivamente una fisica. Il robot deve muoversi all'interno del labirinto con l'obiettivo di trovare un traguardo di colore verde. Per raggiungere questo obiettivo e per rendere l'agente intelligente, il robot è in grado di ricordare gli incroci sul quale è già passato e le direzioni intraprese sugli stessi, in modo da evitare di percorrere sezioni del labirinto già esplorate. Al raggiungimento del traguardo il robot si fermerà.

Robot Virtuale

In questa sezione verrà presentata la componente virtuale del progetto Kobuki Escape e verrà illustrato come si è cercato di riprodurre, nella simulazione virtuale, le condizioni fisiche previste, al fine di poter passare dalla simulazione virtuale a quella fisica nel modo più agevole possibile, riutilizzando quanto più possibile i componenti già progettati e utilizzati. In particolare, esamineremo le tecnologie impiegate nel progetto, l'architettura logica e fisica adottata, e tutti i dati pertinenti alla simulazione stessa. Infine, la sezione si concluderà con alcune considerazioni finali.

Tecnologie

Per la realizzazione della parte virtuale sono state utilizzate le seguenti tecnologie:

CoppeliaSim

Versione utilizzata: *CoppeliaSim Edu, Version 4.6.0 (rev.8)*

CoppeliaSim è un software avanzato di simulazione di robotica sviluppato da Coppelia Robotics. È ampiamente utilizzato per la ricerca e lo sviluppo in robotica, automazione e meccatronica. Questo software permette di simulare in tempo reale robot, macchine e ambienti complessi supportando una vasta gamma di robot e attuatori, permettendo di simulare robot mobili, bracci robotici, droni, veicoli a guida autonoma e molto altro. CoppeliaSim è altamente modulare, consentendo agli utenti di personalizzare ed estendere le funzionalità attraverso script e plugin fornendo diverse API di programmazione (Python, C/C++, Java, Lua, Matlab).

Python

Versione utilizzata: *Python 3.12*

Per la realizzazione del progetto si è scelto di utilizzare come linguaggio di programmazione Python, per la sua semplicità e versatilità. È stato inoltre necessario l'utilizzo delle seguenti librerie:

- ***coppeliasim_zmqremoteapi_client***: utilizzata per consentire la comunicazione con il simulatore CoppeliaSim;

- ***paho.mqtt.client***: consente di utilizzare il Message Broker Mqtt e permettere la comunicazione tra i diversi moduli;
- ***numpy***: per lavorare con array multidimensionali e matrici, offrendo una vasta gamma di funzioni matematiche per operare su questi array;
- ***opencv-python-headless (cv2)***: versione di OpenCV (Open Source Computer Vision Library) ottimizzata per l'uso in ambienti server o contesti in cui non è necessaria una visualizzazione grafica, come applicazioni di elaborazione delle immagini e visione artificiale eseguite su server o in contenitori Docker. Nello specifico, il modulo cv2 in opencv-python-headless fornisce tutte le funzionalità necessarie per eseguire operazioni di visione artificiale;
- ***math***: libreria standard che fornisce funzioni matematiche basilari. È progettata per eseguire operazioni matematiche comuni con numeri in virgola mobile e interi.
- ***time***: permette la gestione del tempo in Python e fornisce funzioni per gestire temporizzazioni, misurare il tempo e convertirlo tra diverse rappresentazioni;
- ***random***: libreria standard utilizzata per generare numeri pseudo-casuali e per eseguire operazioni che richiedono la casualità, in questo caso, la scelta di una direzione tra quelle disponibili.

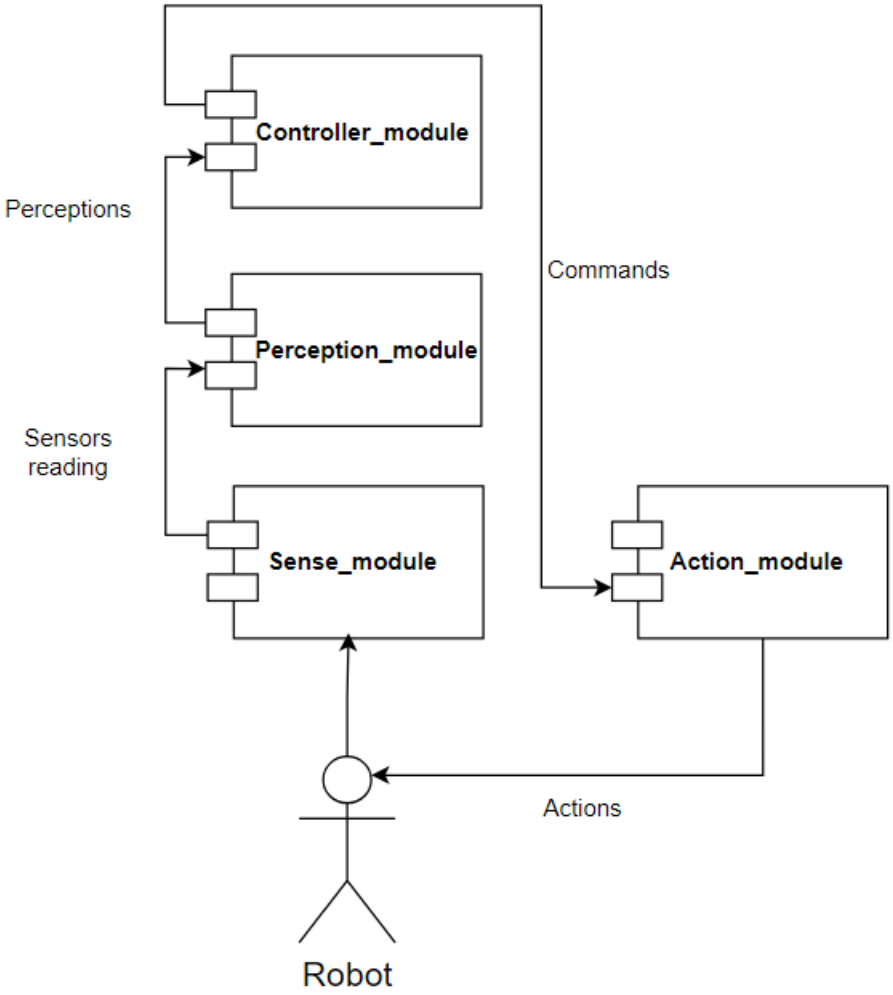
Docker

Versione utilizzata: *Docker 25.0.3*
Docker è una piattaforma open source che automatizza la distribuzione di applicazioni all'interno di container software. I container sono leggeri, portabili e autosufficienti, rendendo più facile sviluppare, testare e distribuire applicazioni in modo consistente su diversi ambienti. Docker utilizza container per isolare le applicazioni e le loro dipendenze. I container condividono il kernel del sistema operativo ma sono isolati a livello di processo e di rete, garantendo portabilità e sicurezza. Ogni container è basato su una immagine Docker, ovvero, un template immutabile che definisce tutto ciò di cui un container ha bisogno per funzionare: codice, runtime, librerie, variabili di ambiente, ecc. Le immagini sono create da un file chiamato Dockerfile.

MQTT

Versione utilizzata: *paho-mqtt 2.0.0*
MQTT (Message Queuing Telemetry Transport) è un protocollo di messaggistica leggero progettato per connessioni machine-to-machine (M2M) e Internet of Things (IoT). È particolarmente adatto per dispositivi che hanno risorse limitate e reti con larghezza di banda ridotta. Il cuore di MQTT è il concetto di "message broker", che gestisce la trasmissione dei messaggi tra i dispositivi. Questo protocollo utilizza un modello di pubblicazione/sottoscrizione (pub/sub), in cui i client possono pubblicare messaggi su un "topic" o iscriversi a un "topic" per ricevere messaggi e, grazie al modello pub/sub, MQTT è altamente scalabile e può supportare un gran numero di dispositivi con facilità.

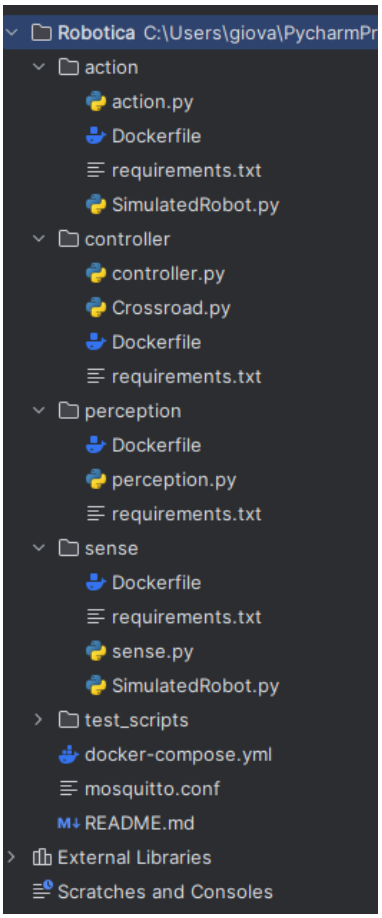
Architettura Logica



- **sense_module**: modulo che riceve le misuazioni dai sensori ad ultrasuoni, i frame dalla videocamera, l'orientazione dal robot e le invia al *perception_module*.
- **perception_module**: modulo che fa da tramite tra il *sense_module* e il controller_module. Il suo compito è quello di elaborare le informazioni ricevute dal *sense_module* e "trasformarle" in informazioni a livello più alto che verranno successivamente inviate al *controller_module*. Nello specifico, il *perception_module* trasforma le misurazioni dei sensori ad ultrasuoni in booleani che indicano se è presente o meno un muro nella direzione in cui puntano, i frame della fotocamera in un booleano che mostra se è presente o meno un oggetto verde nell'immagine e lascia invariata l'orientazione del robot.
- **controller_module**: modulo che corrisponde al cervello vero e proprio del robot
- **action_module**: è il modulo che rappresenta il corpo simulato del robot ed esegue le azioni impartite dal controller_module.

Struttura del codice

Nella seguente immagine è riportata la struttura del codice realizzato per il funzionamento del robot virtuale:



- **action:**
  - **action.py:** in questo file viene definita la classe Body, un'astrazione del robot virtuale. Il compito di queso script è quello di recuperare i messaggi pubblicati sul canale di ascolto (*controller*) e, in base a quanto ricevuto, vengono eseguite delle azioni sulle ruote del robot. Le possibili azioni sono *go_straight*, *turn_left* e *turn_right* con delle velocità determinate dal messaggio ricevuto;
  - **Dockerfile:** file necessario per realizzare un'immagine Docker sfruttando gli altri tre file presenti all'interno della cartella;
  - **requirements.txt:** elenco delle librerie python da installare all'interno del container in modo da consentire il corretto funzionamento. In questo caso le librerie necessarie sono: *paho-mqtt* e *coppeliasim-zmqremoteapi-client*;
  - **SimulatedRobot.py:** definisce una classe SimulatedPioneerBody che rappresenta il corpo del robot simulato in CoppeliaSim. Questa classe fornisce metodi per eseguire azioni sul corpo del robot simulato, come impostare la velocità dei motori e avviare o fermare la simulazione;

- **controller**
  - **controller.py:** Definisce la classe Controller, ovvero, la rappresentazione del controller module e quindi del cervello del robot. Tale classe è responsabile di ricevere i messaggi dal modulo Perception, elaborarli e inviare dei messaggi al modulo Action con le azioni da eseguire. Per ogni messaggio in arrivo si

aggiorna il rispettivo parametro, in modo da avere all'interno della classe una panoramica aggiornata di tutti i valori. Ad ogni aggiornamento corrisponde un controllo in cui, si dà precedenza a quanto ottenuto dalla telecamera in modo da verificare se il robot è arrivato al traguardo o meno, altrimenti, si effettuano dei controlli per sensori ad ultrasuoni per verificare se il robot si trova in un tratto di labirinto rettilineo o in un incrocio/curva. Nel caso in cui il robot si trovi in una curva sterza nell'unica direzione disponibile. Se invece si trova in un incrocio, verifica se in quelle stesse coordinate ha già incontrato un incrocio, in caso negativo memorizza l'incrocio e le direzioni disponibili e, successivamente, ne sceglie una casuale da intraprendere. Se l'incrocio è stato precedentemente incontrato, il robot può scegliere solo tra le direzioni non ancora intraprese e, nel caso in cui le avesse scelte tutte, torna nella direzione di partenza;
    - **Crossroads.py:** classe che rappresenta l'astrazione di un incrocio. Contiene i metodi necessari per inizializzare le direzioni disponibili nel momento in cui si attraversa l'incrocio per la prima volta, per definire l'attuale direzione, aggiornare lo stato di una direzione, ottenere le direzioni disponibili, effettuare il reset delle direzioni (caso in cui sono state tutte intraprese) e aggiornare lo stato della direzione opposta a quella del senso di marcia;
    - **Dockerfile:** file necessario per realizzare una immagine Docker sfruttando gli altri tre file presenti all'interno della cartella;
    - **requirements.txt:** elenco delle librerie python da installare all'interno del container in modo da consentire il corretto funzionamento. In questo caso le librerie necessarie sono: *paho-mqtt*;
- **perception**
    - **Dockerfile:** file necessario per realizzare una immagine Docker sfruttando gli altri tre file presenti all'interno della cartella;
    - **perception.py:** definisce la classe Perceptor, il cui compito è di ricevere messaggi pubblicati sal modulo Sense e inviare messaggi che verranno ricevuto dal Controller. I messaggi ricevuti da questo modulo rappresentano le misurazioni effettuate dai vari sensori (ultrasuoni, orientazione, posizione, camera). I valori vengono elaborati a seconda della tipologia, infatti, per i dati dei sensori a ultrasuoni viene effettuato un controllo e si verifica che la distanza misurata sia maggione di una certa soglia e, in caso positivo, viene restituito il valore True, altrimenti False. Per i valori relativi a posizione e orientazioni non vengono effettuate elaborazioni ma, per i valori della telecamera, viene recuperato il frame dal canale di ascolto e lo si analizza per determinare se all'interno dell'immagine siano presenti o meno oggetti verdi. In caso di oggetti verdi viene restituito in booleano True, altrimenti False. Tutti i risultati calcolati vengono successivamente pubblicati all'interno del topic principale *perception*.
    - **requirements.txt:** elenco delle librerie python da installare all'interno del container in modo da consentire il corretto funzionamento. In questo caso le librerie necessarie sono: *paho-mqtt*, *numpy* e *opencv-python-headless*;
- **sense**
    - **Dockerfile:** file necessario per realizzare una immagine Docker sfruttando gli altri tre file presenti all'interno della cartella;

- **requirements.txt:** elenco delle librerie python da installare all'interno del container in modo da consentire il corretto funzionamento. In questo caso le librerie necessarie sono: *paho-mqtt* e *coppeliasim-zmqremoteapi-client*;
- **sense.py:** definisce una classe Body, una astrazione del robot virtuale. Il compito di questo script è di effettuare delle misurazioni tramite i sensori del robot e pubblicare i risultati nei rispettivi topic definiti nel Message Broker. Le misurazioni effettuate sono: distanze (sensori a ultrasuoni), frame (videocamera), coordinate (sensore di posizione virtuale), orientazione (sensore inerziale virtuale).
- **SimulatedRobot.py:** definisce una classe SimulatedPioneerBody che rappresenta il corpo del robot simulato in CoppeliaSim. Questa classe fornisce metodi per effettuare misurazioni tramite i sensori del robot simulato e avviare o fermare la simulazione;

Setup ambiente Docker

Per mantenere separati i quattro moduli in un ambiente virtuale, si è scelto di utilizzare la tecnologia Docker e definire quinidi un ambiente con cinque containers:

- sense_module
- perception_module
- controller_module
- action_module
- mosquitto (container utilizzato per eseguire un message broker MQTT, fondamentale per la comunicazione tra i containers)

Per poter eseguire facilmente tale architettura è stato definito un file **docker-compose.yml** all'interno del quale, seguendo la sintassi docker, vengono definite le singole immagini che saranno poi alla base dei rispettivi container, le porte relative ad ogni continer e le dipendenze tra le varie tipologie di containers. Di seguito il codice di questo file:

```yaml
version: '3'

services:

  sense_module:
    build:
      context: ./sense
      dockerfile: Dockerfile
    container_name: sense_module
    hostname: sense_module
    ports:
      - "8080:8080"
    expose:
      - 8080
    environment:
      - TZ=Europe/Rome
    depends_on:
      - mosquitto

  action_module:
    build:
      context: ./action
      dockerfile: Dockerfile
    container_name: action_module
    hostname: action_module
    ports:
      - "8090:8090"
    expose:
      - 8090
    environment:
      - TZ=Europe/Rome
    depends_on:
      - mosquitto
      - sense_module
      - perc_module
      - controller_module

  perc_module:
    build:
      context: ./perception
      dockerfile: Dockerfile
    container_name: perc_module
    hostname: perc_module
    ports:
      - "8100:8100"
    expose:
      - 8100
    environment:
      - TZ=Europe/Rome
    depends_on:
      - sense_module
      - mosquitto

  controller_module:
    build:
      context: ./controller
      dockerfile: Dockerfile
    container_name: controller_module
    hostname: controller
    ports:
      - "8110:8110"
    expose:
      - 8110
    environment:
      - TZ=Europe/Rome
    depends_on:
      - sense_module
      - perc_module
      - mosquitto

  mosquitto:
    image: eclipse-mosquitto:latest
    container_name: mosquitto_broker_container
    hostname: mosquitto_module
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf
    restart: always
    ports:
      - '1883:1883'
      - '9001:9001'
    environment:
      - TZ=Europe/Rome
```

Per poter realizzare i containers e successivamente avviare la simulazione, bisogna eseguire i seguenti comandi:

- **`docker compose build:`** comando necessario per realizzare le immagini dei containers partendo dal codice e dal Dockerfile di una certa cartella, oppure, scaricandola dall'archivio Docker online (caso del message broker);
- **`docker compose up:`** comando che consente di eseguire i container sulla base delle immagini precedentemente creat;
- **`docker compose down:`** comando da utilizzare nel momento in cui si volessero spegnere i containers.

Setup topic MQTT

I topic di comunicazione definiti nel Message Broker sono suddivisi in modo da essere relativi al modulo che pubblica sul topic dedicato. Di seguito un elenco dei topic e dei sotto-topic:

- **sense:** topic nel quale vengono pubblicati tutte le misure effettuate dal modulo sense;
    - **ultrasonicSensor[0]:** dati realtivi al sensore ad ultrasoni 0 (sinistra);
    - **ultrasonicSensor[4]:** dati realtivi al sensore ad ultrasoni 4 (centro);
    - **ultrasonicSensor[7]:** dati realtivi al sensore ad ultrasoni 7 (destra);
    - **Vison_sensor:** dati relativi alla telecamera, corrispondono a frame;
    - **orientation:** orientazione del robot espressa in radianti;
    - **position:** coppia di coordiante che indicano la posizione del robot all'interno della scena virtuale;
        - **x:** coordinata x;
        - **y:** coordinata y.

Esempio di topic: **sense/nome_sensore** oppure **sense/position/coordinata**

- **perception:** topic nel quale vengono pubblicati i risultati delle elaborazioni delle misure ottenute dal modulo perception;
    - **front:** valore booleano che indica se di fronte al robot è presente o meno un ostacolo;
    - **left:** valore booleano che indica se a sinistra del robot è presente o meno un ostacolo;
    - **right:** valore booleano che indica se a destra del robot è presente o meno un ostacolo;
    - **green:** valore booleano che indica se nell'immagine fornita dalla telecamera è presente o meno un oggetto verde (traguardo);
    - **orientation:** orientazione del robot espressa in radianti;
    - **position_x:** coordinata x della posizione del robot nella scena virtuale;
    - **position_y:** coordinata y della posizione del robot nella scena virtuale.

Esempio di topic: **perception/nome_percezione**

- **controls:** topic nel quale vengono pubblicati i risultati delle elaborazioni dei dati prodotti dal perception che, nel modulo controller, vengono convertiti in comandi da inviare al modulo action;
    - **direction:** stringa che indica la tipologia di azione che deve eseguire il robot. Le possibili azioni ci sono: *go, cross, turn_right, turn_right_slow, turn_right_more_slow, turn_left, turn_left_slow, turn_left_more_slow*;
    - **target:** booleano che indica se il traguardo è stato raggiunto o meno.

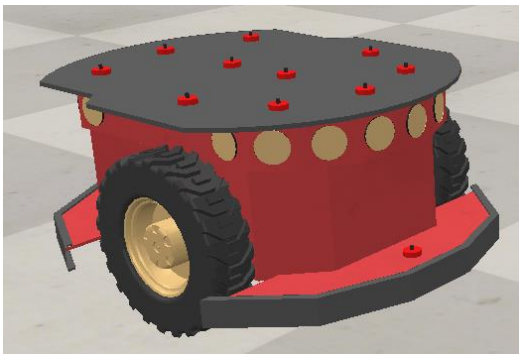
Esempio di topic: **controls/direction**

Simulazione Virtuale

Nella simulazione il robot partirà da una posizione prefissata all'interno di un labirinto per poi dirigersi, autonomamente, alla ricerca del traguardo. Il robot sarà in grado di riconoscere incroci e mantenerli nella propria memoria ricordando le coordinate e quali direzioni sono state intraprese. In questo modo, si va a evitare che il robot finisca per ripetere sempre gli stessi percorsi ed evitando quindi che si entri in un loop infinito.

Agent

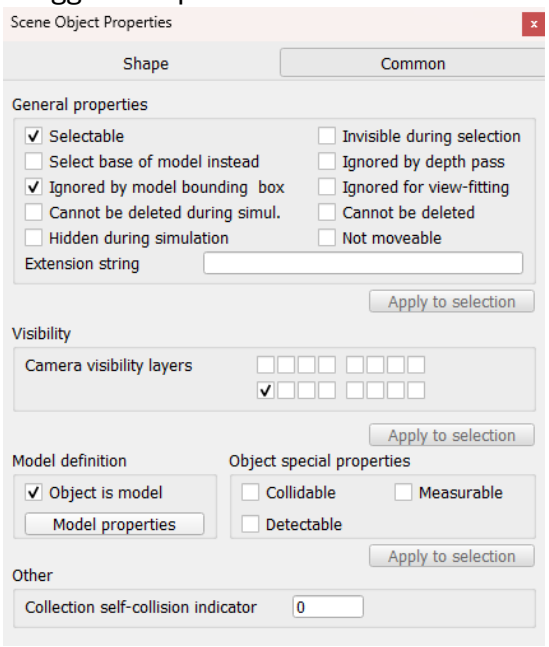
Data la grande somiglianza con il robot che verrà utilizzato per la simulazione fisica, si è scelto di utilizzare un robot PioneerP3DX.



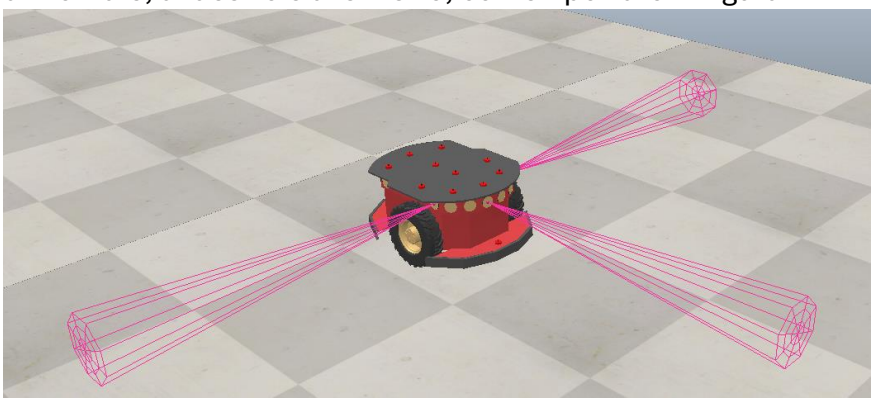
Di default, questo robot è dotato di:

- 2 ruote;
- 16 sensori ad ultrasuoni, numerati da 0 a 15;
- 11 pin di connessione per altri sensori, numerati da 0 a 10.

Di seguito sono riportati i settaggi scelti per il PioneerP3DX all'interno dell'ambiente virtuale:

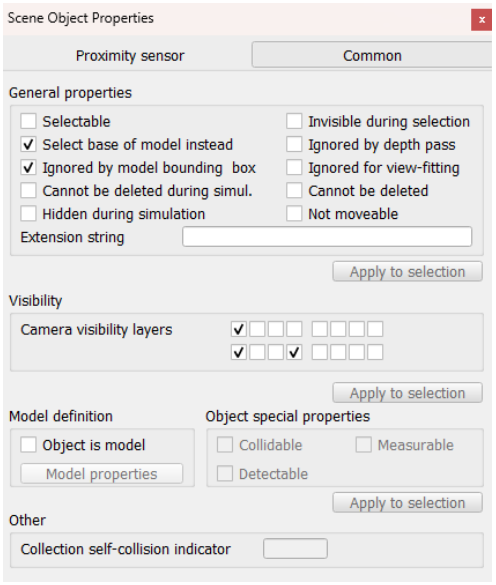


Date le esigenze, si è scelto di utilizzare solo tre dei sensori a ultrasuoni, quelli che corrispondono al frontale, al destro e al sinistro, come riportato in figura:

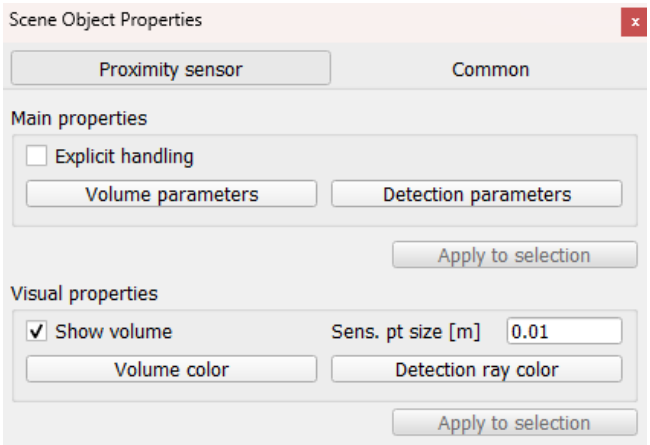


I sensori utilizzati sono: **ultrasonicSensor[0]**, **ultrasonicSensor[4]**, **ultrasonicSensor[7]**, i quali corrispondono rispettivamente a: sensore sinistro, sensore frontale e sensore destro. Di seguito i settaggi dei sensori:

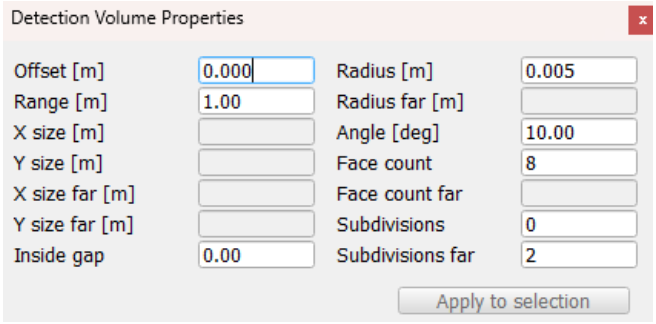
- Scene Object Properties, sezione Common:



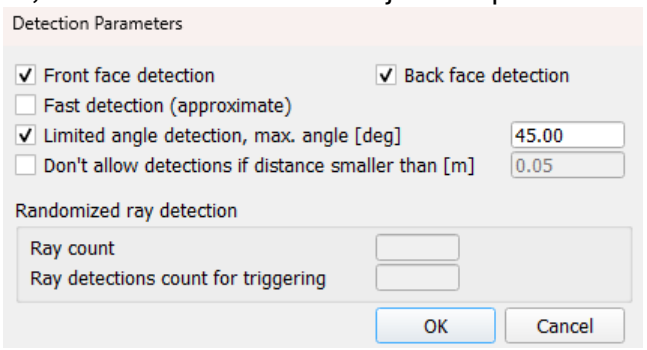
- Scene Object Properties, sezione Proximity sensor :



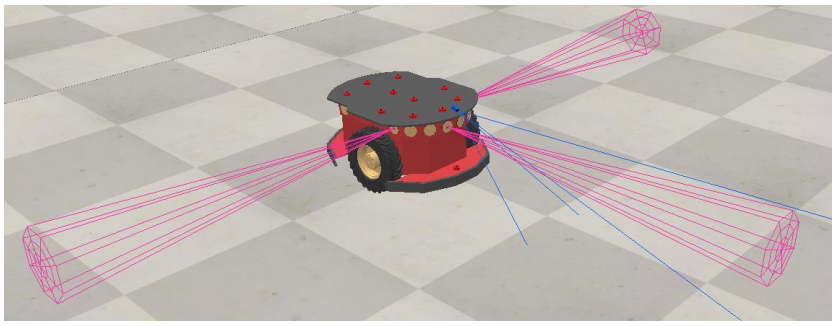
- Volume Parameters, accessibile da Scene Object Properties:



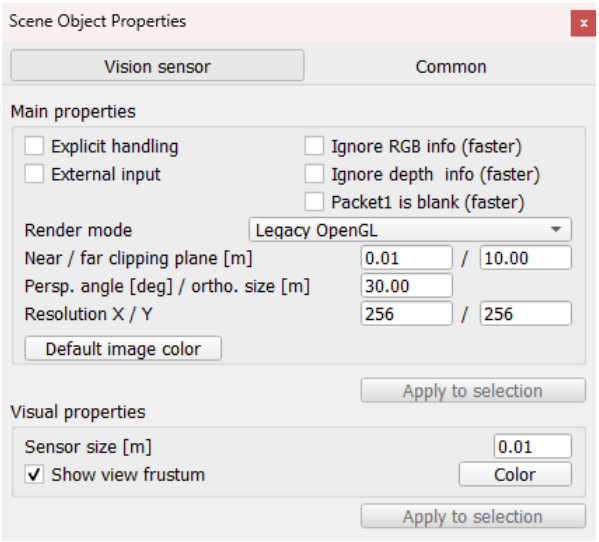
- Detection Parameters, accessibile da Scene Object Properties:

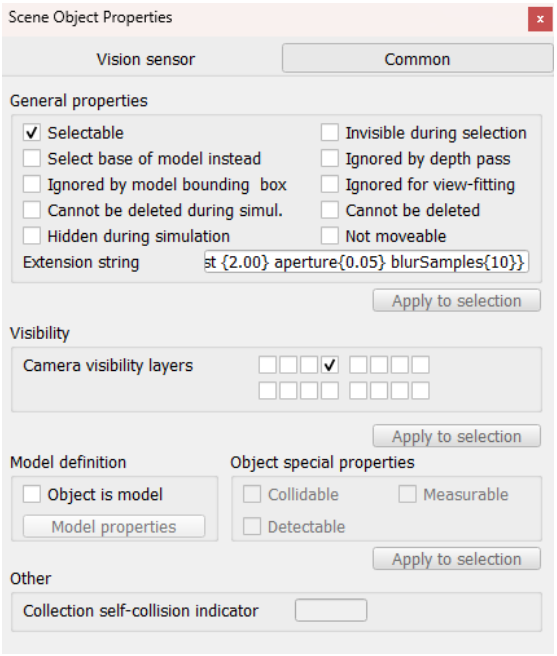


Oltre ai sensori a ultrasuoni, è stato necessario aggiungere un **Vision_sensor** collegato nella parte frontale del robot, in modo tale da avere una telecamera per verificare la presenza o meno del traguardo.



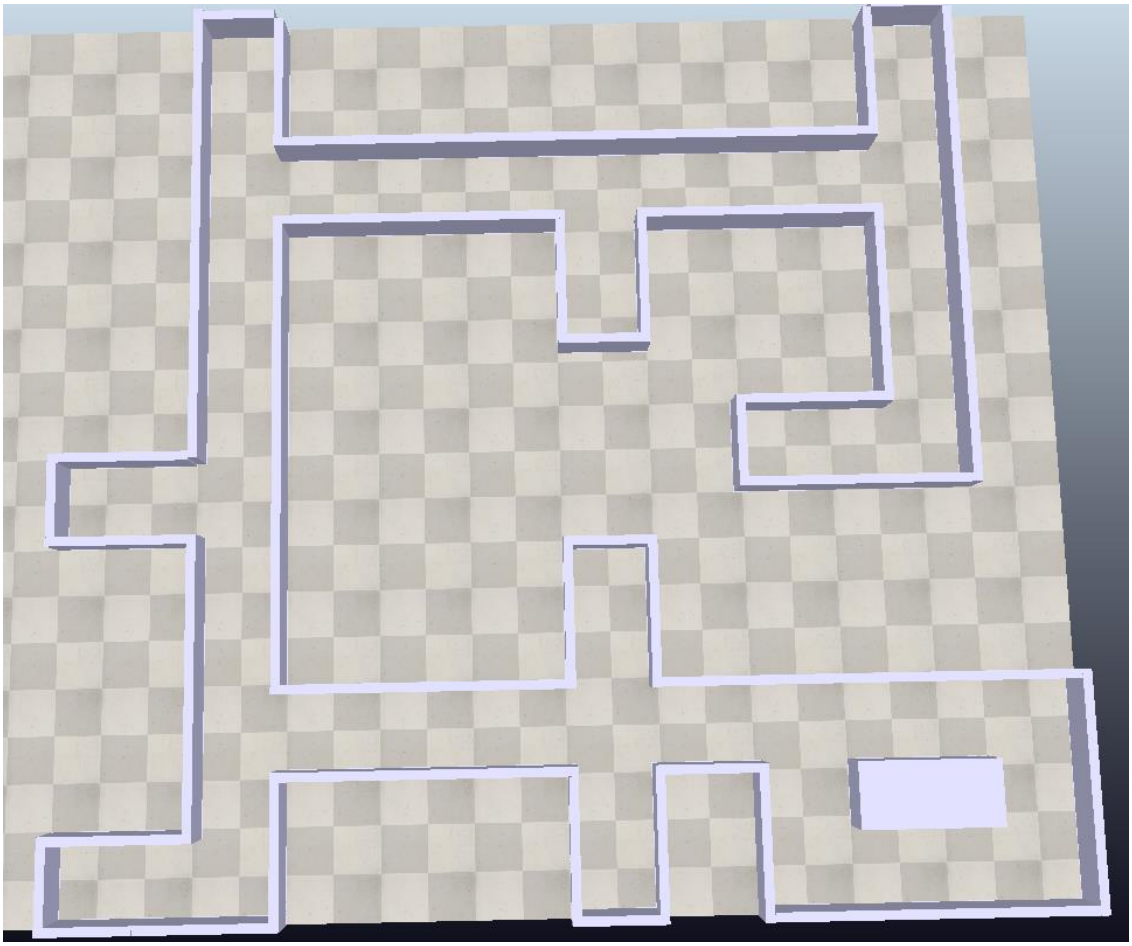
Le proprietà del sensore sono state configurate nel seguente modo:



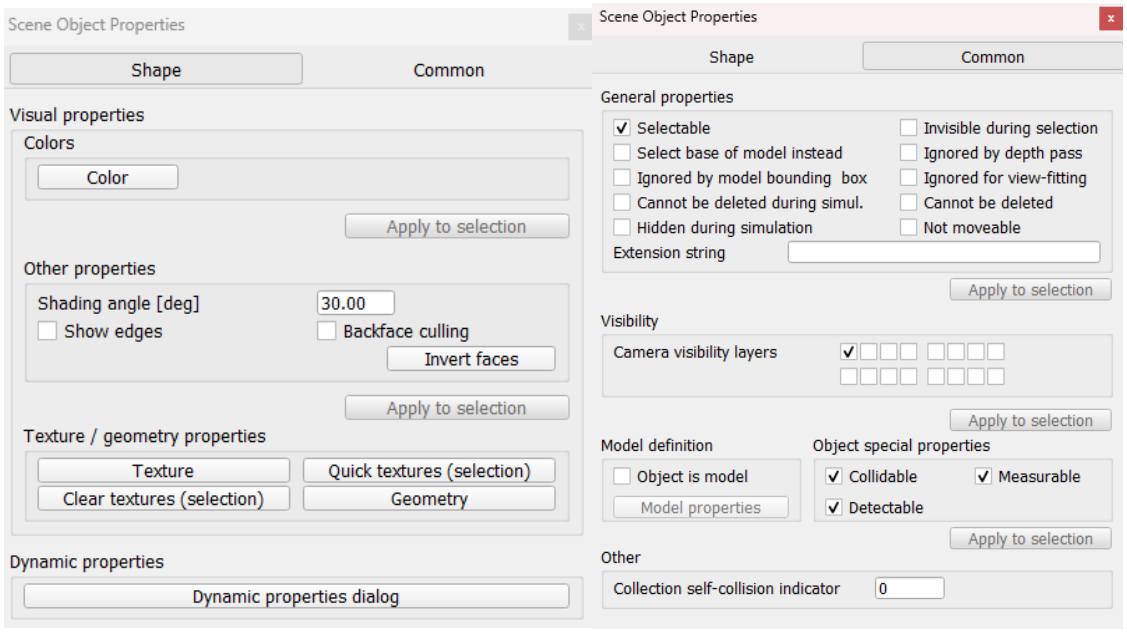


Scena di simulazione

Per realizzare il labirinto nell'ambiente di simulazione virtuale, è stato utilizzato l'editor di CoppeliaSim, ottenendo il seguente risultato:

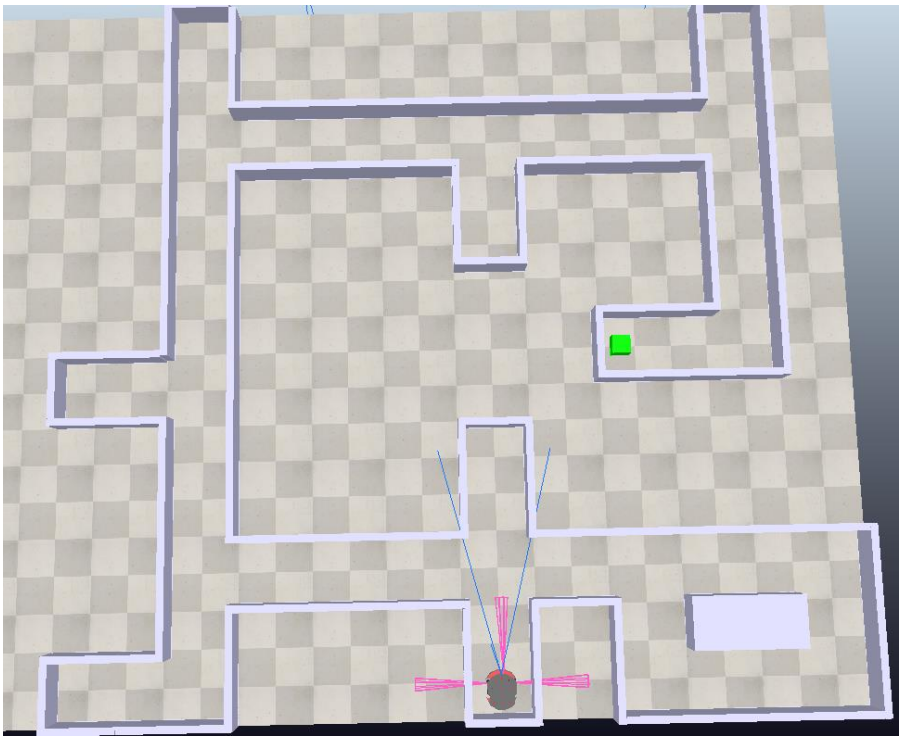


Il labirinto è stato creato componendo tra loro elementi messi a disposizione dal simulatore, in particolare 44 Cuboid, con i seguenti settaggi:



In seguito, è stato inserito un ulteriore Cuboid di colore verde che ha la funzione di traguardo:

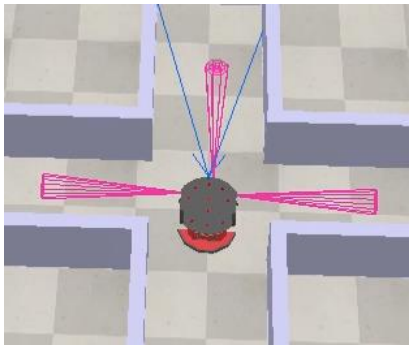
Il risultato finale dell'ambiente di simulazione è il seguente:



Simulazione

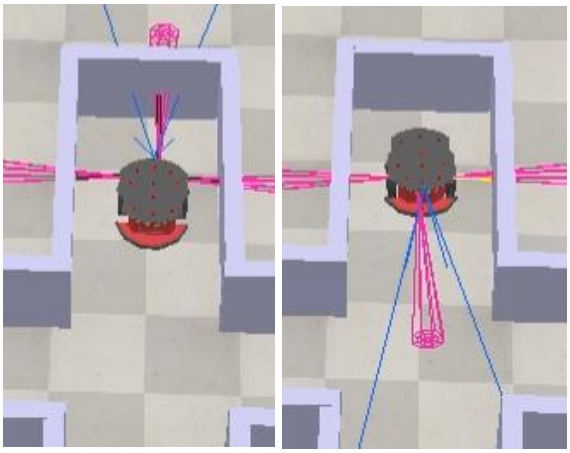
In questa sezione, analizzeremo una simulazione dell'attività del robot all'interno dell'ambiente simulato.

- **Primo incrocio**



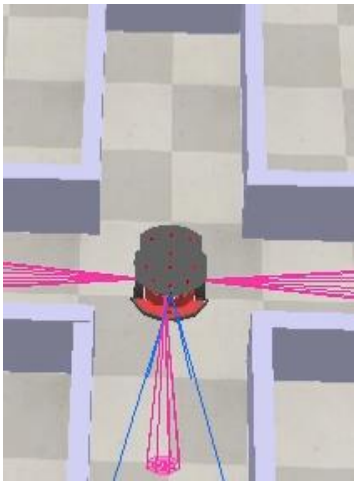
Dopo la partenza, il robot incrontra subito un incrocio a quattro uscite. Come primo step il robot controlla se alle sue attuali coordinate è presente o meno un incrocio e, dato che è appena partito e non ha incroci in memoria lo inizializza.
Nell'inizializzazione definisce la posizione dell'incrocio con le sue attuali coordinate e definisce quali sono le direzioni disponibili (in questo caso *nord, sud, est, ovest*) e segna la direzione di provenienza come percorsa (*sud*). A questo punto il robot sceglie causalmente una tra le direzioni disponibili e la intraprende. (In questa simulazione il robot sceglie *nord*);

- **Vicolo cieco**



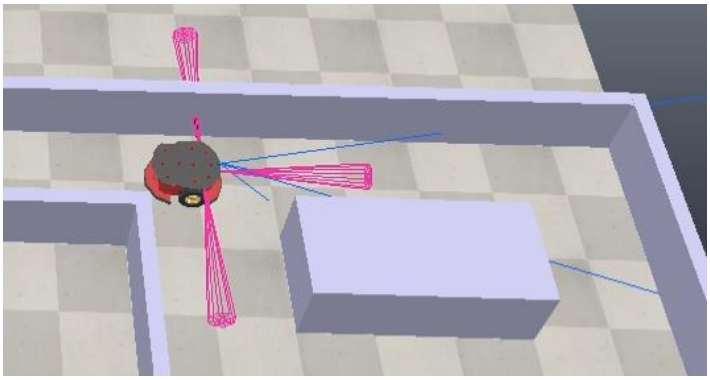
Dopo aver proseguito dritto, il robot incontra un vicolo cieco e ruota sul posto in senso orario fino ad aver compiuto una rotazione di 180°;

- **Secondo passaggio sul primo incrocio**



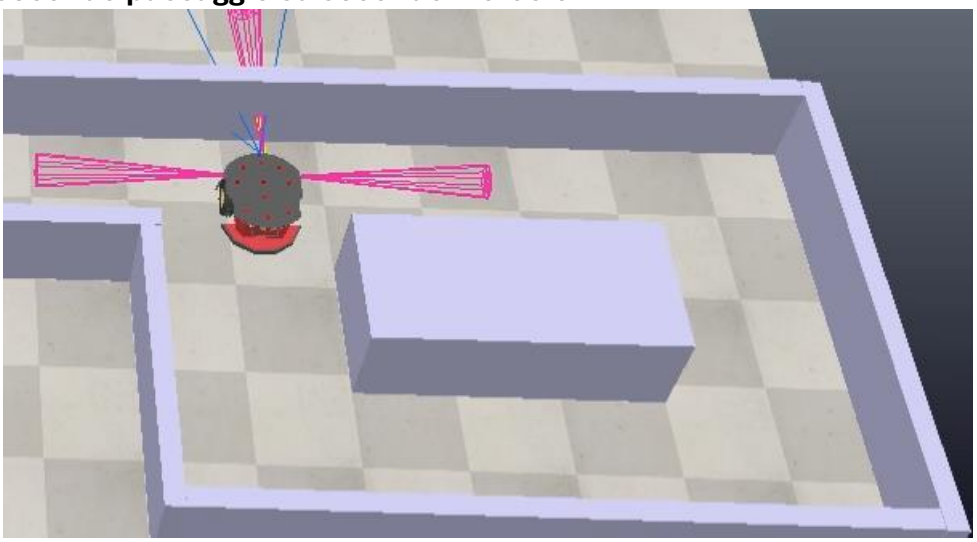
Proseguendo il suo cammino, il robot torna nuovamente sull'incrocio visto precedentemente e lo riconosce grazie ad un controllo sulle coordinate. In questo caso, dato che precedentemente è stata scelta la direzione *nord*, restano disponibili solo altre due direzioni (*est* e *ovest*) e sceglie di andare verso *est*.

- **Secondo incrocio**



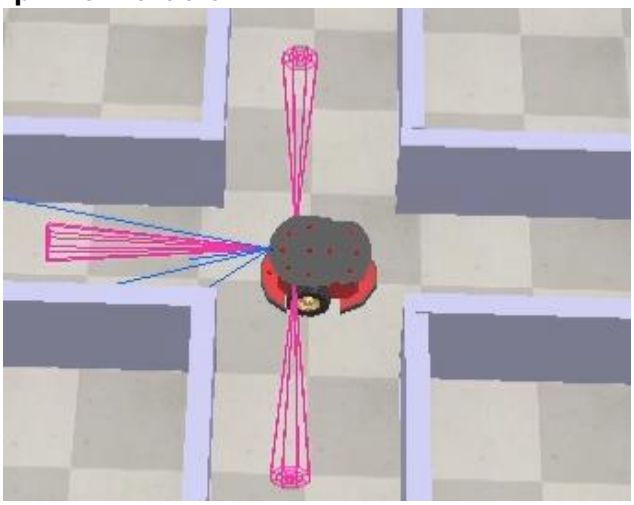
A questo punto, il robot rileva un nuovo incrocio e lo inizializza con le direzioni *est*, *ovest* (percorsa), *sud* e assegna le coordinate. A questo punto sceglie come direzione casuale *est*.

- **Loop – secondo passaggio su secondo incrocio**



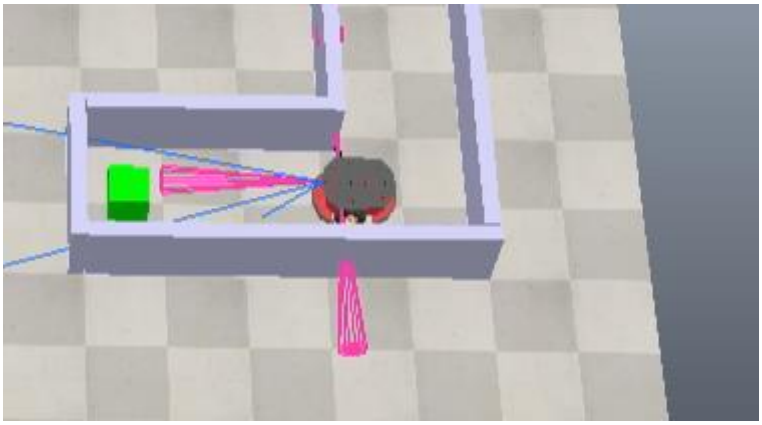
Il robot completa la rotazione intorno al rettangolo centrale e torna nuovamente sul secondo incrocio. In questo momento l'agente capisce che ha già attraversato questo incrocio e che quindi la direzione da cui proviene è percorsa (*sud*) quindi, dato che tutte e tre le direzioni disponibili sono state percorse viene abilitata solo la direzione da cui è stato scoperto l'incrocio (*ovest*) e il robot torna nella direzione di provenienza, evitando un loop infinito.

- **Terzo passaggio sul primo incrocio**



Come nel precedente step, il robot riconosce l'incrocio e, in questo caso, intraprende l'unica direzione non ancora esplorata: *ovest*.

- **Traguardo**



Dopo aver continuato il suo percorso all'interno del labirinto (seguendo la stessa logica degli step precedenti), il robot arriva al traguardo. Nel momento in cui il robot non sta ruotando, è uscito da una curva/incrocio e rileva un oggetto di colore verde (che corrisponde al traguardo), i motori vengono spenti e il robot si ferma completando con successo il percorso all'interno del labirinto.

Al seguente link è possibile visionare la simulazione completa: link al video
Per vedere una simulazione alternativa nello stesso labirinto (senza loop), aprire il seguente link: link secondo video

Robot Fisico

In questa sezione verrà presentata la componente fisica del progetto Kobuki Escape e verrà illustrato come dalla simulazione virtuale si è cercato di produrre fedelmente la realizzazione fisica del robot. In particolare, esamineremo le tecnologie impiegate, l'architettura logica e fisica, la taratura dei parametri, la GUI realizzata e la simulazione fisica vera e propria. Infine, la sezione si concluderà con alcune considerazioni finali.

Tecnologie Software

Python

Versione utilizzata: *Python 3.12*
Per la realizzazione del progetto si è scelto di utilizzare come linguaggio di programmazione Python, per la sua semplicità e versatilità. È stato inoltre necessario l'utilizzo delle seguenti librerie:

- ***paho.mqtt.client***: consente di utilizzare il Message Broker Mqtt e permettere la comunicazione tra i diversi moduli;
- ***time***: permette la gestione del tempo in Python e fornisce funzioni per gestire temporizzazioni, misurare il tempo e convertirlo tra diverse rappresentazioni;
- ***random***: libreria standard utilizzata per generare numeri pseudo-casuali e per eseguire operazioni che richiedono la casualità, in questo caso, la scelta di una direzione tra quelle disponibili;
- ***tkinter*:** utilizzata per creare l'interfaccia grafica per l'avvio del sistema.
- ***flask*:** consente di realizzare framework web leggeri e flessibili. In questo caso è utilizzato per lanciare il server che permette di ricevere le richieste di avvio e stop della simulazione dalla interfaccia grafica.

Docker

Versione utilizzata: *Docker 25.0.3*
Docker è una piattaforma open source che automatizza la distribuzione di applicazioni all'interno di container software. I container sono leggeri, portabili e autosufficienti, rendendo più facile sviluppare, testare e distribuire applicazioni in modo consistente su diversi ambienti. Docker utilizza container per isolare le applicazioni e le loro dipendenze. I container condividono il kernel del sistema operativo ma sono isolati a livello di processo e di rete, garantendo portabilità e sicurezza. Ogni container è basato su una immagine Docker, ovvero, un template immutabile che definisce tutto ciò di cui un container ha bisogno per funzionare: codice, runtime, librerie, variabili di ambiente, ecc. Le immagini sono create da un file chiamato Dockerfile.

MQTT

Versione utilizzata: *paho-mqtt 2.0.0*

MQTT (Message Queuing Telemetry Transport) è un protocollo di messaggistica leggero progettato per connessioni machine-to-machine (M2M) e Internet of Things (IoT). È particolarmente adatto per dispositivi che hanno risorse limitate e reti con larghezza di banda ridotta. Il cuore di MQTT è il concetto di "message broker", che gestisce la trasmissione dei messaggi tra i dispositivi. Questo protocollo utilizza un modello di pubblicazione/sottoscrizione (pub/sub), in cui i client possono pubblicare messaggi su un "topic" o iscriversi a un "topic" per ricevere messaggi e, grazie al modello pub/sub, MQTT è altamente scalabile e può supportare un gran numero di dispositivi con facilità.

Tecnologie Hardware

Per la realizzazione fisica del robot sono state usate le seguenti componenti:

- **Kobuki**: Il Kobuki è un robot mobile sviluppato da Yujin's Robots, progettato principalmente per applicazioni di ricerca e sviluppo nella robotica mobile. Questo robot rappresenta il nucleo del nostro progetto grazie alla sua robustezza e semplicità d'integrazione con altri sistemi, come la piattaforma di calcolo Jetson.
  È dotato di due motori a trazione differenziale che gli permettono di muoversi agilmente in diverse direzioni, con controllo preciso di velocità e rotazione.
  Sensori:
  Il robot è equipaggiato con diversi sensori che gli consentono di interagire con l'ambiente:

  - Encoder: Utilizzati per tracciare la distanza percorsa e monitorare la velocità delle ruote, permettendo una navigazione accurata.
  - Sensori di prossimità a infrarossi: Questi sensori rilevano ostacoli ravvicinati, aiutando il robot a evitare collisioni.
  - Sensori a urto: Situati sulla parte frontale, questi sensori registrano impatti fisici diretti, permettendo al robot di riconoscere quando si scontra con un oggetto.
  - Giroscopio: Fornisce informazioni sull'orientamento e permette di monitorare i cambiamenti angolari, aiutando a mantenere una navigazione stabile.

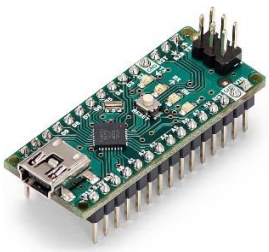
  Nel nostro progetto, il **Kobuki** viene utilizzato come piattaforma mobile per eseguire i comandi di navigazione e interagire con l'ambiente circostante, grazie alla sua capacità di rilevare ostacoli e rispondere agli input del controller. Grazie ai suoi sensori e motori di precisione, è in grado di navigare in modo autonomo all'interno del labirinto.

- **Nvidia Jetson:** il Jetson è una serie di board di computer embedded sviluppata dalla Nvidia.
  Nel nostro progetto, la scheda **Nvidia Jetson** funge da cervello centrale per l'elaborazione dei dati e il controllo del robot ed ospita il modulo principale per il suo funzionamento:
    - **Sense_Action_Module**: Questo modulo unificato riceve i dati dai sensori del robot Kobuki e dagli Arduino Nano, li elabora in tempo reale grazie alla potenza di calcolo della Jetson, e li pubblica nel canale di canale di comunicazione. Inoltre, gestisce i comandi di movimento e trasmette istruzioni precise al Kobuki per eseguire azioni come avanzare, ruotare o fermarsi.

- **Arduino Nano**: l'Arduino Nano è una microcontroller board sviluppata da arduino.cc che contiene 30 header I/O e può essere caricata tramite un cavo mini-USB type-B o con una batteria da 9 V.
  Nel nostro progetto, **due** schede **Arduino Nano** giocano un ruolo cruciale nella gestione dei sensori collegati al sistema. Ad una delle schede sono collegati tre sensori ad ultrasuoni **HC-SR04**, mentre l'altra gestisce il sensore di colore **TCS230**. Entrambi gli Arduino acquisiscono e inviano i dati al modulo **Sense_Action**, che poi elabora queste informazioni per permettere al robot di interagire con l'ambiente.

  1. **Arduino 1 con sensori ad ultrasuoni HC-SR04**: I tre sensori ad ultrasuoni sono disposti strategicamente per misurare la distanza dagli ostacoli circostanti. Le misurazioni vengono elaborate dal primo Arduino e inviate al modulo Sense_Action, dove vengono tradotte in informazioni utili per il movimento del robot, come l'evitamento degli ostacoli.
  2. **Arduino 2 con sensore di colore TCS230**: Il secondo Arduino è dedicato al sensore di colore TCS230, che rileva la presenza di oggetti colorati, in particolare l'obiettivo verde all'interno del labirinto. Questo Arduino invia i dati cromatici al modulo Sense_Action.

  La separazione tra i due Arduino permette una gestione più efficiente dei sensori, ottimizzando la raccolta e l'elaborazione delle informazioni. Il modulo **Sense_Action** riceve e processa i dati provenienti da entrambi i microcontrollori per poi inviarli sul canale di comunicazione.



- **Modulo WiFi WNA1100 – Adattatore wirelesss USB N150**: questo modulo è in grado di fornire un supporto per il WiFi nei dispositivi sprovvisti di esso.
  Nel nostro progetto viene utilizzato connettendolo al Nvidia Jetson per fornire un supporto WiFi in grado di hostare un server web locale.

- **Sensori ad ultrasuoni HC-SR04**: questo sensore ad ultrasuoni è in grado di rilevare la presenza di oggetti nelle immediate vicinanze e calcolarne anche la distanza.
  Nel nostro progetto vengono utilizzati 3 sensori di questo tipo posti: 1 davanti e 2 ai lati del Kobuki in modo da poter individuare eventuali muri/ostacoli e/o strada libera.
  Possiede 4 pin:
  - VCC: utilizzato per la tensione di alimentazione (da 3 a 5,5V).
  - GND: utilizzato per la messa a terra.
  - TRIG: utilizzato per inviare il segnale ad ultrasuoni quando viene impostato ad alto.
  - ECHO: utilizzato per produrre un impulso che verrà poi interrotto quando viene ricevuto il segnale riflesso dall'ostacolo.

- **Sensore di colore TCS230**: questo sensore è composto da una matrice 8x8 di fotodiodi. Questi 64 fotodiodi sono suddivisi in 4 gruppi. 16 fotodiodi hanno un filtro rosso, 16 fotodiodi hanno un filtro verde, 16 fotodiodi hanno un filtro blu e i restanti 16 fotodiodi non presentano filtro.
  Nel contesto del nostro progetto, il sensore è posizionato nella parte anteriore del robot **Kobuki** e ha la funzione di identificare se il traguardo, rappresentato da un oggetto verde, è stato raggiunto. Una volta rilevato il colore verde, il sistema riconosce che l'obiettivo è stato trovato e il robot può fermarsi.

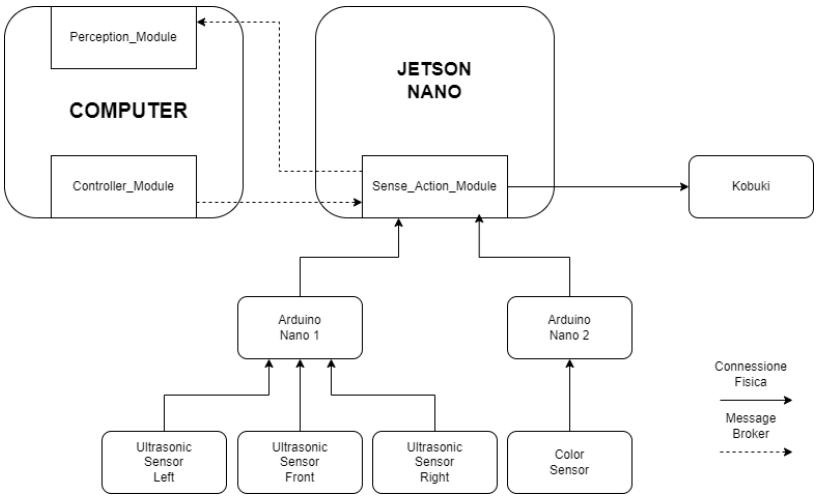
  Il **TCS230** è dotato di 8 pin:

  - **VCC**: Pin di alimentazione che fornisce la tensione necessaria per il funzionamento del sensore.
  - **GND**: Pin per la messa a terra.
  - **S0 - S3**: Pin di input che permettono di configurare la sensibilità e la frequenza del sensore.
  - **OUT**: Pin di output che trasmette il segnale digitale corrispondente al colore rilevato.

Grazie alla sua alta precisione nel rilevamento dei colori, il **TCS230** rappresenta un componente chiave per il riconoscimento del traguardo nel labirinto, garantendo che il robot possa completare la sua missione in modo autonomo ed efficace.

- **Cavi Jumper**: questi cavi sono dei minuscoli ponticelli metallici utilizzati per collegare le componenti in un circuito digitale.
  Nel nostro progetto sono stati utilizzati per collegare i vari sensori all'Arduino Nano.

Architettura Hardware

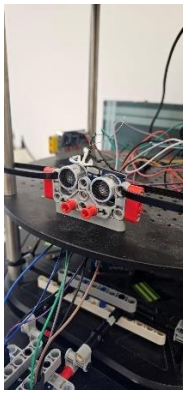


Nel nostro progetto, abbiamo sviluppato un sistema robotico basato sull'integrazione di diversi componenti hardware, tra cui sensori, microcontrollori e unità di elaborazione, al fine di garantire la navigazione autonoma del robot Kobuki all'interno di un ambiente strutturato come un labirinto.
L'architettura hardware del sistema prevede il collegamento di due **Arduino Nano** al **Jetson Nano**, ciascuno dei quali ha compiti specifici. Un Arduino è dedicato alla gestione dei tre sensori ad ultrasuoni, mentre l'altro si occupa del sensore di colore. Entrambi i microcontrollori inviano i dati raccolti direttamente al Jetson, che li elabora e li utilizza per controllare le azioni del robot Kobuki.

Un Arduino è collegato ai tre sensori ad ultrasuoni, con la seguente piedinatura:

Piedinatura Arduino Nano – sensori ad ultrasuoni:

Pin Arduino Nano	Piede sensore ad ultrasuoni
D2	**TRIG** sensore **frontale**
D3	**ECHO** sensore **frontale**
D4	**TRIG** sensore di **sinistra**
D5	**ECHO** sensore di **sinistra**
D6	**TRIG** sensore di **destra**
D7	**ECHO** sensore di **destra**
D8 – D12	-



L'altro Arduino è invece connesso al **sensore di colore TCS230** attraverso la seguente piedinatura:

Piedinatura Arduino Nano – sensore di colore:

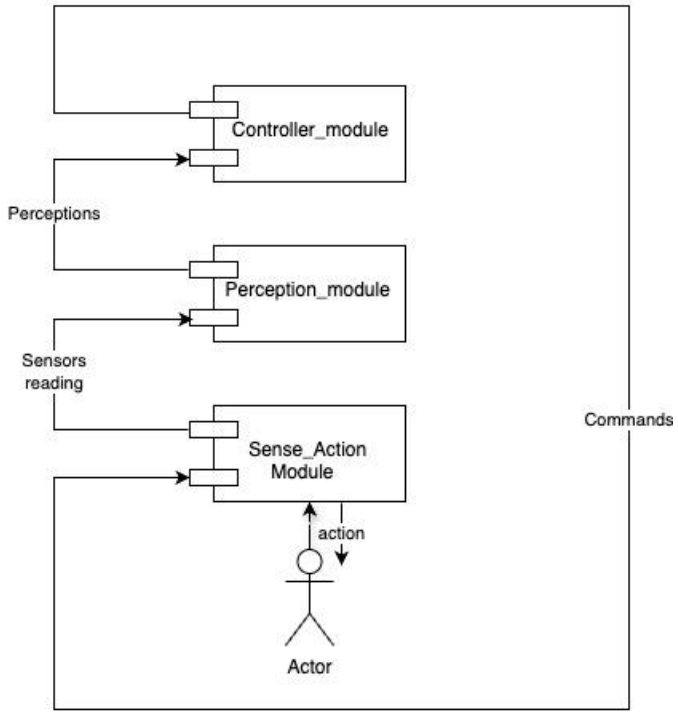
Pin Arduino Nano	Piede sensore di colore
D2	S0
D3	S1
D4	S2
D5	S3
D6	OUT
D7 – D12	-

Nel nostro progetto, la comunicazione tra le varie componenti del sistema (moduli di senso/azione, percezione e controllo) avviene tramite il protocollo **MQTT** (Message Queuing Telemetry Transport) che permette ai vari moduli di inviare e ricevere informazioni tramite specifici **topic**.
I topic di comunicazione sono organizzati in base al modulo che pubblica le informazioni. I dati vengono raccolti, elaborati e trasmessi attraverso questi canali strutturati, garantendo che ogni componente del sistema riceva le informazioni necessarie per eseguire le proprie operazioni. Di seguito viene riportata la struttura dei principali topic utilizzati nel sistema.

- **Sense**
  Questo topic raccoglie e pubblica tutti i dati provenienti dai sensori collegati al robot. I sotto-topic sono i seguenti:
  - **Posizione del robot**: Pubblica le coordinate della posizione del robot (x, y) per tracciare la sua posizione e riconoscere eventuali incroci.
  - **Colore**: Pubblica il colore rilevato dal sensore di colore, utile per determinare se il robot ha raggiunto un obiettivo specifico (ad esempio, un oggetto verde).
  - **Sensori ad ultrasuoni**: Pubblica le misurazioni dei tre sensori ad ultrasuoni per indicare la distanza dagli ostacoli nelle tre direzioni: frontale (*front*), sinistra (*left*) e destra (*right*).
  - **Rotazione**: Pubblica un booleano (*rotating*) che indica se il robot sta attualmente ruotando o meno.

    - **Orientazione:** Pubblica l'attuale orientazione del robot (*orientation*) espressa in punti cardinali relativi.
- **Perception**
  Questo topic riceve i dati pubblicati nel topic *sense* e li elabora per trasformarli in informazioni più ad alto livello, utili per il controllo del robot. I sotto-topic includono:
    - **Percezione degli ostacoli**: Pubblica un'informazione booleana che indica se il robot è troppo vicino a un ostacolo o meno, basandosi sui dati degli ultrasuoni.
    - **Verde rilevato**: Pubblica un valore booleano che indica se il robot ha individuato un oggetto di colore verde nel suo percorso.
    - **Rotazione**: Pubblica un booleano (*rotating*) che indica se il robot sta attualmente ruotando o meno.
    - **Orientazione:** Pubblica l'attuale orientazione del robot (*orientation*) espressa in punti cardinali relativi.
- **Controller**
  Questo topic riceve i dati elaborati dal *perception_module* e invia i comandi decisionali per controllare il robot. I principali sotto-topic sono:
    - **Comandi di controllo**: Pubblica i comandi necessari per il movimento del robot, come girare, avanzare, fermarsi, ecc., che saranno eseguiti dal modulo di azione.

Architettura Logica



- **Sense_Action_Module**: Questo modulo combina le funzionalità di **lettura dei sensori** e **l'esecuzione delle azioni**. La funzionalità di questa componente è quella di leggere i dati dai sensori ad ultrasuoni, dal sensore di colore e gestire l'orientamento e la posizione del robot, inviando ogni informazione sul canale di comunicazione. In aggiunta, esegue le azioni necessarie (come muoversi, girare, fermarsi, ecc.) sulla base dei comandi ricevuti.

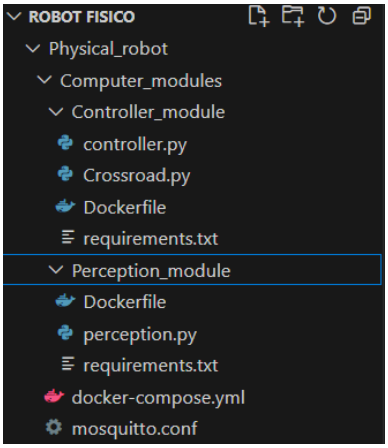
- **Perception_Module**: Questo modulo funge da interfaccia tra il **Sense_Action_Module** e il **Controller_Module**. Il suo compito principale è elaborare i dati grezzi ricevuti dai sensori e trasformarli in informazioni ad un livello più alto. Ad esempio, le misurazioni dei sensori ad ultrasuoni vengono convertite in valori booleani per indicare la presenza o meno di un ostacolo in determinate direzioni, mentre il valore rilevato dal sensore di colore viene tradotto in un booleano che segnala la presenza o meno di un oggetto verde nell'immagine. L'orientamento del robot, invece, viene mantenuto invariato e inoltrato al controller.
- **Controller_Module**: Il **controller_module** rappresenta il "cervello" del sistema robotico, dove avviene tutta la logica decisionale. Riceve le informazioni elaborate dal **Perception_Module** e le utilizza per pianificare le azioni appropriate. Nello specifico, il **controller_module** prende decisioni come evitare ostacoli, avanzare verso il traguardo o fermarsi una volta raggiunto l'obiettivo. Implementa algoritmi di controllo autonomo che consentono al robot di navigare all'interno del labirinto in maniera efficiente e autonoma, evitando zone già esplorate e riconoscendo il traguardo verde. Il modulo è responsabile di inviare comandi precisi al **Sense_Action_Module** per far sì che il robot esegua le operazioni necessarie (come spostarsi, ruotare o fermarsi) in base alla situazione ambientale e agli obiettivi programmati.

Struttura del Codice

Nel progetto fisico, a differenza della versione virtuale, la struttura del codice è organizzata in tre principali directory:

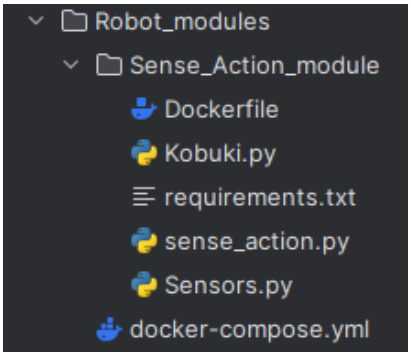
- **Computer_Modules:**
  Questa cartella contiene i moduli **Perception e Controller**, i quali richiedono una maggiore capacità di calcolo e vengono eseguiti direttamente su un computer. Il **Perception Module** elabora i dati provenienti dai sensori e li trasforma in informazioni di livello superiore, mentre il **Controller Module** gestisce la logica di navigazione e presa di decisioni.
- **Robot_Modules:**
  Contiene il modulo **Sense_Action**, il quale viene eseguito sulla piattaforma **Jetson**. Questi moduli richiedono una capacità di calcolo inferiore poiché si occupano principalmente della raccolta dei dati dai sensori e dell'esecuzione dei comandi inviati dal controller.
- **Project_Start:**
  Include il codice responsabile dell'avvio della **GUI web**, che consente l'interazione con il robot e la visualizzazione dello stato e delle operazioni.

Computer_modules:



- Cartella Controller_module:
    - **controller.py:** definisce la classe Controller, ovvero, la rappresentazione del controller module e quindi del cervello del robot. Tale classe è responsabile di ricevere i messaggi dal modulo Perception, elaborarli e inviare dei messaggi al modulo Action con le azioni da eseguire. Per ogni messaggio in arrivo si aggiorna il rispettivo parametro, in modo da avere all'interno della classe una panoramica aggiornata di tutti i valori. Ad ogni aggiornamento corrisponde un controllo in cui, si dà precedenza a quanto ottenuto dal sensore di colore in modo da verificare se il robot è arrivato al traguardo o meno, altrimenti, si effettuano dei controlli per sensori ad ultrasuoni per verificare se il robot si trova in un tratto di labirinto rettilineo, in un incrocio o in una curva. Nel caso in cui il robot si trovi in una curva, sterza nell'unica direzione disponibile. Se invece si trova in un incrocio, verifica se in quelle stesse coordinate ha già incontrato un incrocio, in caso negativo memorizza l'incrocio e le direzioni disponibili e, successivamente, ne sceglie una casuale da intraprendere. Se l'incrocio è stato precedentemente incontrato, il robot può scegliere solo tra le direzioni non ancora intraprese e, nel caso in cui le avesse scelte tutte, torna nella direzione di partenza;
    - **Crossroads.py:** classe che rappresenta l'astrazione di un incrocio. Contiene i metodi necessari per inizializzare le direzioni disponibili nel momento in cui si attraversa l'incrocio per la prima volta, per definire l'attuale direzione, aggiornare lo stato di una direzione, ottenere le direzioni disponibili, effettuare il reset delle direzioni (caso in cui sono state tutte intraprese) e aggiornare lo stato della direzione opposta a quella del senso di marcia;
    - **Dockerfile:** file necessario per realizzare una immagine Docker sfruttando gli altri tre file presenti all'interno della cartella;
    - **requirements.txt:** elenco delle librerie python da installare all'interno del container in modo da consentire il corretto funzionamento. In questo caso l'unica libreria necessaria è: *paho-mqtt*;
- Cartella Perception_module:
    - **Dockerfile:** file necessario per realizzare una immagine Docker sfruttando gli altri tre file presenti all'interno della cartella;
    - **perception.py:** definisce la classe Perceptor, il cui compito è di ricevere messaggi pubblicati dal modulo Sense e inviare messaggi che verranno ricevuti dal Controller. I messaggi ricevuti da questo modulo rappresentano le misurazioni effettuate dai vari sensori. I valori vengono elaborati a seconda della tipologia, infatti, per i dati dei sensori a ultrasuoni viene effettuato un controllo e si verifica che la distanza misurata sia maggione di una certa soglia e, in caso positivo, viene restituito il valore True, altrimenti False. Per i valori relativi alla posizione non vengono effettuate elaborazioni ma, per i valori del sensore di colore, viene recuperato il colore ottenuto dal relativo sensore e si verifica se tale colore è *Verde* o meno. In caso di colore verde viene restituito un booleano True, altrimenti False. Tutti i risultati calcolati vengono successivamente pubblicati all'interno del topic principale *perception*.
    - **requirements.txt:** elenco delle librerie python da installare all'interno del container in modo da consentire il corretto funzionamento. In questo caso l'unica libreria necessaria è: *paho-mqtt*;

Robot_modules:



Cartella Sense_Action_module:

- **Dockerfile**: file necessario per realizzare un'immagine Docker sfruttando il file sense_action.py presente all'interno della cartella.
- **Kobuki.py**: definisice la classe Kobuki contenente vari attributi e metodi utilizzati all'interno del file sense_action.py.
- **requirements.txt**: elenco delle librerie python da installare all'interno del container in modo da consentire il corretto funzionamento. In questo caso l'unica libreria necessaria è: *paho-mqtt*.
- **sense_action.py**: definisce la classe Body e 2 client del message broker: uno per il sotto-modulo Sense e l'altro per il sotto-modulo Action.
  Nel modulo Sense leggiamo i dati dai sensori grazie al metodo *read_sensor_data* definito nel file Sensors.py presente all'interno della cartella e andiamo poi a pubblicare i dati ricevuti sul topic *sense/** che sarà inviato al modulo Perception.
  Nel modulo Action, otteniamo l'azione da eseguire dal modulo Controller; questa azione verrà poi eseguita dal robot grazie al metodo *move* definito nella classe Kobuki.

Project_start:

- Cartella Project_start/Server_Jetson:
  - **Flask server (server.py)**
    Questo codice gestisce le richieste HTTP GET su /start e /stop, avviando o fermando la simulazione tramite subprocess che esegue gli script start.sh e stop.sh.
  - **Script Bash (start.sh, stop.sh)**
    Script semplici che avviano e fermano il container Docker utilizzando docker-compose.

- Cartella Project_start:
  - **docker_control.py**
    Script che permette di controllare Docker tramite comandi up e down, utilizzando subprocess e cambiando directory per eseguire il comando Docker correttamente.
  - **read_Config.py**
    Fornisce un'interfaccia per leggere valori dal file config.json, molto utile per accedere dinamicamente a configurazioni come l'IP di Jetson.
  - **GUI in Tkinter (**web_page.py**)**
    Questo file contiene il codice per una GUI che permette di avviare e fermare la simulazione. È una finestra Tkinter che visualizza l'immagine di Kobuki e due bottoni: "Start" e "Stop", che eseguono richieste HTTP a un server Flask per avviare/fermare la simulazione e gestiscono Docker tramite docker_control.py.

Taratura parametri

Rotazione

Poiché i dati dei sensori fisici, come il giroscopio, possono essere imprecisi, il controllo della rotazione, nello sviluppo fisico del robot, si basa su un metodo più robusto e semplice che non dipende dall'angolo target preciso. Questo approccio evita problemi legati all'accuratezza dei sensori e assicura che il robot possa completare le rotazioni in modo affidabile.
Nel simulatore virtuale, la rotazione del robot viene gestita attraverso il controllo preciso degli angoli e dell'orientamento, in cui le funzioni `turn_right` e `turn_left` sono progettate per impostare un angolo target specifico per la rotazione. Queste funzioni calcolano la rotazione necessaria e poi invocano `set_robot_orientation` la quale è responsabile della determinazione della direzione (sinistra o destra) e la velocità di rotazione necessaria per raggiungere l'angolo target. Utilizza una tolleranza angolare, definita come `ANGLE_TOLERANCE`, per decidere se il robot ha raggiunto l'orientamento desiderato o se deve continuare a ruotare.
Nel contesto fisico, invece, la gestione della rotazione è stata adattata in questo modo:
La rotazione viene gestita attraverso una serie di chiamate alla funzione `move`, che controlla la velocità e l'angolo di sterzo del robot. Ad esempio, `turn_left e turn_right` fanno ruotare il robot tramite un numero preciso di chiamate alla funzione move (43 chiamate per la rotazione di 90 gradi) piuttosto che basarsi su un angolo target specifico.

Sensore di colore

Per la realizzazione del nostro progetto, abbiamo dovuto tarare alcuni parametri, uno di questi riguarda il sensore di colore; per fare in modo che funzioni in maniera pulita, abbiamo eseguito i seguenti passaggi:

1. Sviluppato uno script di test caricato poi sull'Arduino Nano nel quale andiamo a stampare a schermo i valori di "Rosso", "Verde", "Blu" e della "Luce".
2. Successivamente, abbiamo preso un foglio bianco e lo abbiamo posto davanti al sensore di colore andando ad annotare il valore più alto della Luce.
3. Analogamente, abbiamo preso un foglio nero, lo abbiamo posto davanti al sensore di colore e abbiamo annotato il valore più basso della Luce.

4. Questi due valori ci saranno utili poi per lo script finale (VALORE1, VALORE2). Difatti, saranno poi definiti come costanti “WHITE_THRESHOLD” e “BLACK_THRESHOLD” e ci serviranno per distinguere i rispettivi colori.

Sensore a ultrasuoni

I sensori ultrasonici sono stati tarati posizionando gli stessi a distanze differenti, di interesse progettuale, da un ostacolo e rilevando le distanze misurate con i seguenti risultati:

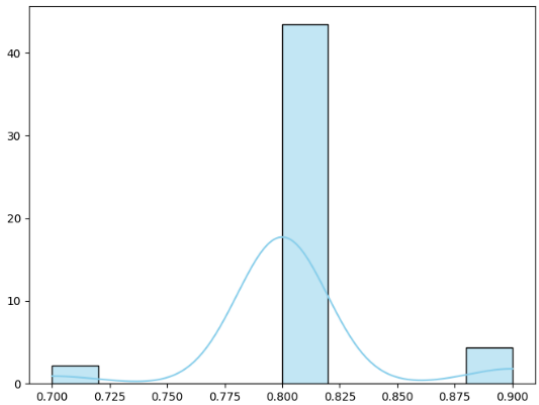
DISTANZA OSTACOLO	DISTANZA MISURATA
10 cm	9 cm
20 cm	18 cm
30 cm	27 cm
40 cm	35 cm
50 cm	45 cm

I risultati ottenuti, considerando le distanze di movimento rettilineo e le dimensioni del labirinto, sono ritenuti accettabili per gli scopi del progetto.

Movimento rettilineo

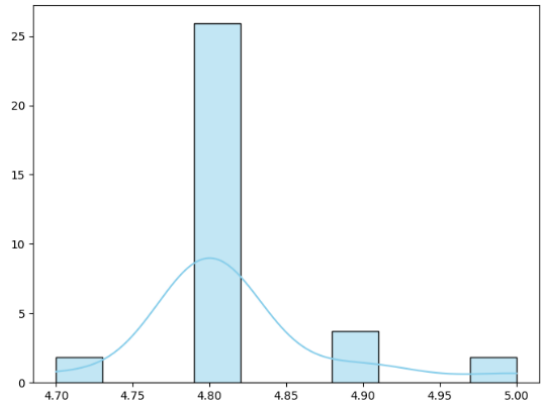
In questa sezione analizzeremo il comportamento del nostro robot durante l’avanzamento rettilineo, in risposta a comandi di spostamento (move) predefiniti, e come questi influenzano la distanza totale percorsa. In particolare, il numero di comandi “move” presi in considerazione sarà: 1, 5 e 10. L'obiettivo è analizzare come il robot percorre lo spazio in funzione di questi comandi e valutare la distribuzione statistica dei suoi spostamenti attraverso l'utilizzo di funzioni gaussiane.
L'analisi include il calcolo preciso dello spazio percorso in risposta al numero di comandi dati, con particolare attenzione alla variabilità del movimento descritta dalle curve gaussiane.

Con 1 comando *move*:



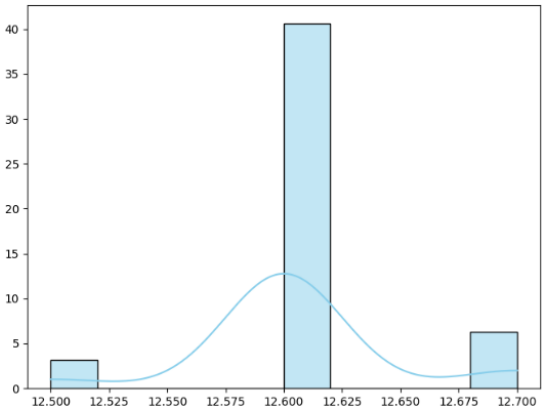
Con un singolo comando di movimento si può notare come il robot percorre sempre la stessa distanza (0.8 cm) con variazioni nell’ordine dei millimetri.

Con 5 comandi *move*:



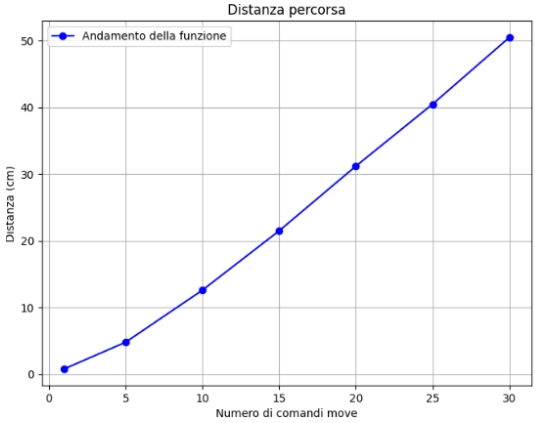
Con cinque comandi di movimento il robot percorre quasi sempre la stessa distanza (4.8 cm) con variazioni nell'ordine dei millimetri.

Con 10 comandi *move*:



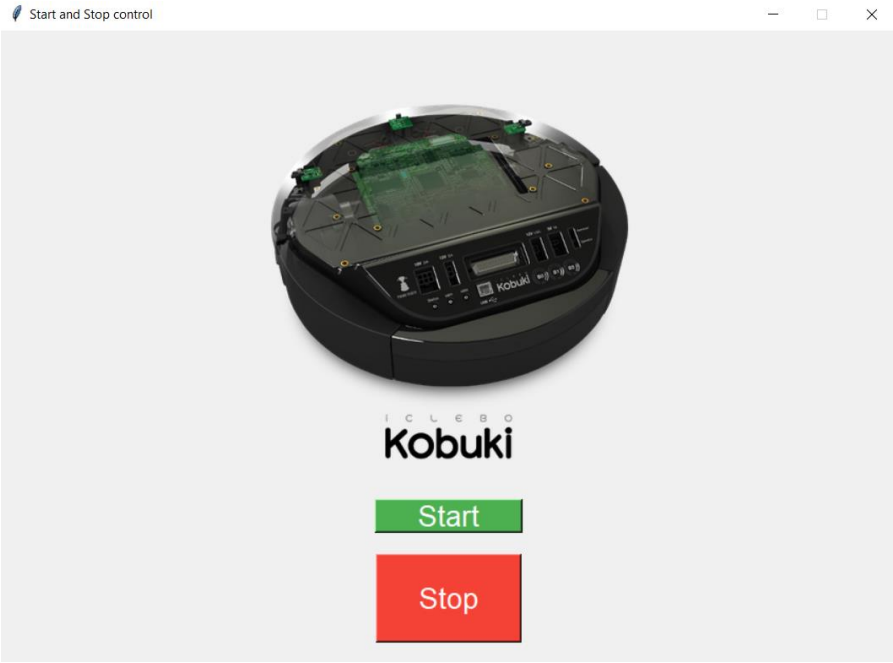
Con dieci comandi di movimento ha lo stesso comportamento dei casi precedenti (in questo caso distanza percorsa: 12.6 cm) con variazioni nell'ordine dei millimetri.

Andatura in relazione al numero di comandi:



Da questo grafico si evince che il robot impiega un certo numero di chiamate prima di stabilizzare il proprio movimento, infatti, si nota che l'andamento fino a venticinque comandi *move* non è uguale in ogni intervallo ma cambia di angolazione, mentre, tra le 25 e le 30 chiamate si ha un andamento più lineare e stabile.

GUI



La GUI messa a disposizione è molto semplice ed intuitiva. All'interno di essa possiamo trovare la foto del Kobuki assieme a 2 bottoni: uno verde utilizzato per far partire il robot ed uno rosso per farlo fermare.
Il bottone verde è responsabile di far partire sia i moduli docker all'interno del robot, sia i moduli docker all'interno del computer che invia la richiesta.
Il bottone rosso è invece responsabile di fermare tutti i moduli docker e conseguentemente anche il robot.
La Classe GUI definita nel file *web_page.py* possiede 5 funzioni:

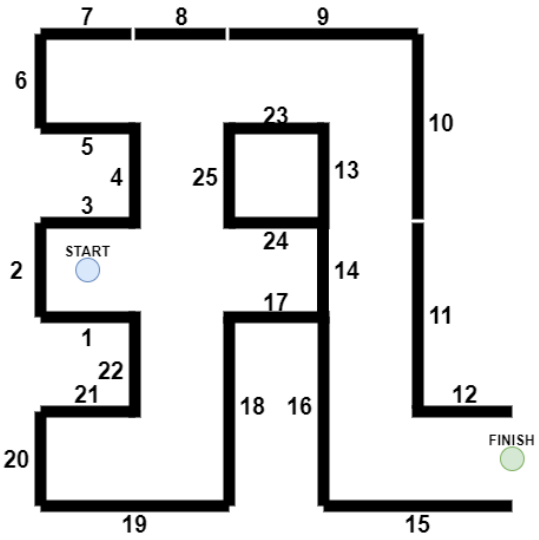
1. ***__init__***: funzione utilizzata per l'inizializzazione di alcuni parametri come, per esempio, il titolo della pagina web, le dimensioni, ecc.
2. ***start***: funzione utilizzata per avviare la funzione *send_request* con argomento "start" e la funzione *control_docker_compose* con argomento "up".
   Questa funzione viene attivata una volta premuto il bottone verde "Start", non appena il bottone verrà premuto, apparirà un messaggio che ci informerà che la simulazione inizierà a breve.
3. ***send_request*:** prende in input il parametro "action". Legge l'IP del Jetson tramite un file di configurazione e manda una richiesta all'URL composto dall'IP e dal parametro "action". Dopodichè, tenta di fare una richiesta GET all'URL e in caso di successo apparirà un messaggio che ci informa di ciò; altrimenti apparirà un messaggio che ci informa del fallimento della richiesta con annesso codice di errore.
4. ***control_docker_compose***: prende in input il parametro "action". Questa funzione esegue il comando "python3 ./docker_control.py {action}", il quale andrà ad eseguire il comando "docker compose {action}". Questo codice è racchiuso in un costrutto try catch nel quale è possibile un'eccezione: CalledProcessError. In caso invece di errori non previsti è possibile che venga alzata un'eccezione generica.
5. ***stop***: funzione analoga alla funzione *start*, cambiano però i parametri inviati alle funzioni *send_request* e *control_docker_compose*, rispettivamente: "stop" e "down".

Simulazione Fisica

Setup dell'ambiente

Per la simulazione del robot, abbiamo optato nel realizzare un labirinto fatto di polistirolo in quanto è un materiale che riesce a far rimbalzare su di sé i segnali ad ultrasuoni e quindi riesce a far funzionare bene i sensori ad ultrasuoni e quindi la capacità del robot di rilevare muri.

Il labirinto è stato progettato in modo da essere il più completo possibile, ovvero, contenendo curve a destra e sinistra, incroci a quattro uscite, incroci a T e vicoli ciechi.



La realizzazione fisica, come detto precedentemente, è stata fatta attrverso l'utilizzo di pannelli di polistirolo di due diverse dimensioni:

- 1000 x 250 (7 pannelli)
- 500 x 250 (18 pannelli)

Ogni singolo pezzo è stato successivamente numerato per rispettare quanto definito nel precedente schema.
Il risultato finale è il seguente:

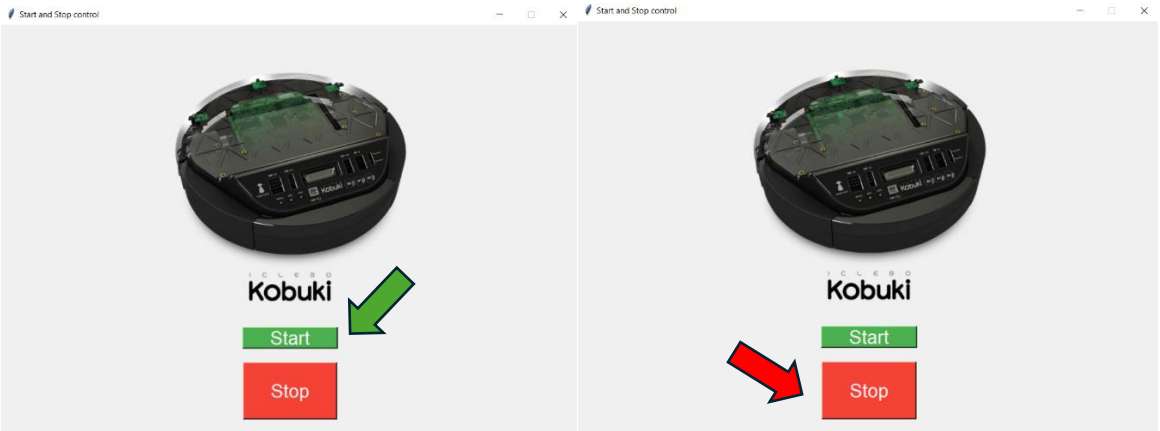
Setup robot

Prima di avviare la simulazione, bisogna effettuare alcuni passaggi di setup, ovvero:

- Collegare il Jetson Nano ad un powerbank per alimentarlo;
- Sul Jetson Nano lanciare lo script di avvio del server Python per la recezione delle richieste di Start e Stop. Eseguire tale operazione entrando nella cartella PATH_CARTELLA e poi digitando il comando:
  python3 server.py;
- Accendere il robot Kobuki;
- Posizionare il robot nella posizone di partenza.

Avvio tramite GUI

Come detto nella precedente sezione, è necessario avviare il server Python presente sul Jetson Nano per poter effettuare lo start e lo stop della simulazione tramite la GUI.
Dopo aver eseguito tale operazione, è sufficiente aprire l'interfaccia (eseguendo lo script web_page.py), avviare il Docker Engine sul dispositivo che avrà il ruolo di eseguire i moduli Perception e Controller e successivamente premere i bottoni Start e Stop per eseguire rispettivamente l'avvio e la fine della simulazione.



Simulazione

- **Primo incrocio**

Dopo la partenza, il robot incontra subito un incrocio a 4 uscite. Come primo step il robot controlla se nelle attuali coordinate ha già incontrato un incrocio e, dato che la simulazione è appena iniziata, memorizza questo primo incrocio e lo inizializza, definendo coordinate e direzioni presenti e disponibili (in questo caso *nord, sud, est e ovest*) e segna da direzione di provenienza come già percorsa (*sud*). A questo punto il robot sceglie casualmente una direzione tra le tre disponibili e la intraprende. (In questa simulazione sceglie *est*);

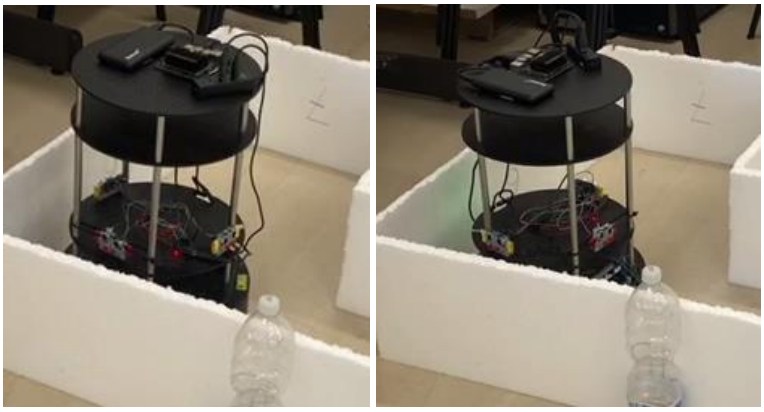
- **Curva a destra**

Il robot incontra una curva verso destra. La riconosce come tale grazie ai controlli effettuati sui sensori a ultrasuoni e quindi procede con una rotazione di 90 gradi verso destra;

- **Vicolo cieco**

Dopo aver proseguito dritto, il robot incontra un vicolo cieco e ruota sul posto in senso orario eseguendo una rotazione di 180 gradi;

- **Curva a sinistra**

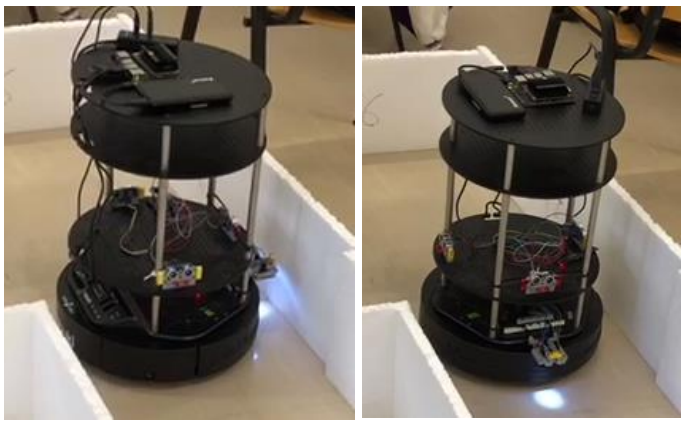


Tornando indietro, il robot incontra la curva precedente che in questo senso di marcia è verso sinistra e quindi procede con una rotazione di 90 gradi verso sinistra;

- **Secondo passaggio sul primo incrocio**

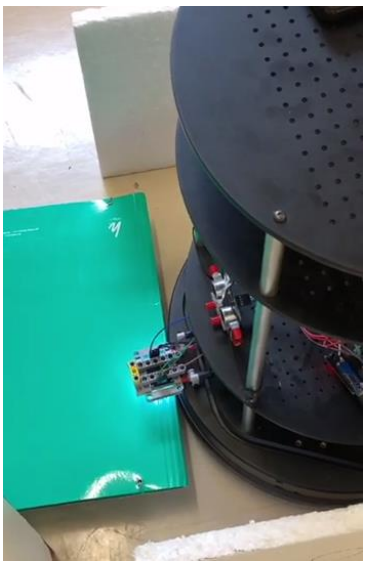
Proseguendo il cammino, il robot torna nuovamente sull'incrocio visto precedentemente e lo riconosce grazie a un controllo sulle coordinate. In questo caso, dato che precendemente è stata scelta la direzione *est*, restano disponibili solo le direzioni *nord* e *ovest* e sceglie di intraprendere la seconda;

- **Secondo incrocio**



Successivamente, il robot rileva un nuovo incrocio (a T) e lo inizializza con le direzioni *sud* e *nord* disponibili e *est* già percorsa (direzione di provenienza). A questo punto sceglie come direzione causale *nord*;

- **Traguardo**



Dopo aver effettuato una curva a destra e una sinistra, il robot raggiunge il traguardo. Nel momento in cui il sensore di colore rileva il colore verde, i motori vengono disattivati e il robot si ferma completando con successo il percorso all'interno del labirinto.

Ai seguenti link è possibile visionare questa simulazione e altre due con variazioni del labirinto.

Simulazione 1: link al video
Simulazione 2: link al video

Conclusioni

Il nostro progetto riesce a raggiungere l'obiettivo preposto, ovvero navigare in un labirinto riconoscendo le direzioni degli incroci già intraprese finchè non individua un oggetto di colore verde nel suo cammino che designa il traguardo. Tale comportamento è stato confermato anche con variazioni del labirinto e della posizione di partenza del robot e del traguardo.
Sono tuttavia possibili dei miglioramenti a questo progetto:

- **Monitoraggio del livello della batteria**: in un contesto reale non possiamo essere sicuri che la batteria del robot sia sufficiente per viaggiare dall'inizio alla fine del labirinto.
  Potrebbe essere quindi utile aggiungere uno schermo led che possa mostrare il livello attuale della batteria del robot.
- **Utilizzo del bump sensor**: come già anticipato, il Kobuki possiede un bump sensor, il quale, permette di controllare se il robot colpisce un ostacolo o meno.
  Questo sensore potrebbe essere utile come backup: nel caso in cui il sensore ad ultrasuoni posto davanti fallisca, il bump sensor potrebbe essere un'alternativa valida ad esso.
- **Ostacoli dinamici**: il nostro progetto si basa sul fatto che gli ostacoli siano fissi, ma nella realtà, ciò potrebbe non essere per forza vero, se, ad esempio, questo robot venisse posto in una strada trafficata, un pedone o un animale potrebbero attraversare all'improvviso e, in quel caso, sarebbe necessario poter gestire questa eventualità.