UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze
Matematiche, Informatiche e Fisiche

TESI DI LAUREA IN
INTERNET OF THINGS, BIG DATA E WEB

Robot per l'esplorazione di ambienti

CANDIDATO
Davide Bassan

RELATORE
Prof. Ivan Scagnetto

Anno accademico 2020-2021

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

A tutti coloro che mi hanno accompagnato nel mio percorso di crescita.

Sommario

L'espansione dell'utilizzo della robotica in ambito industriale e domestico è in forte crescita.

L'obiettivo della robotica è rendere più efficiente, sicuro e preciso ciò che l'uomo fa lentamente, in maniera imprecisa o gli è pericoloso.

In questo progetto viene trattata la progettazione e l'implementazione di un robot in grado di muoversi autonomamente in maniera casuale in un ambiente sconosciuto, mapparlo tramite un apposito sensore laser, stimare la propria posizione tramite l'utilizzo di diversi algoritmi e raggiungere determinati punti d'interesse in completa autonomia.

Verrà quindi implementato un pannello di controllo dove poter monitorare gli spostamenti dell'agente che, nel caso di errori potenzialmente fatali, ammetterà l'intervento umano tramite il controllo remoto.

Verranno inoltre analizzati alcuni approcci alternativi a quelli implementati in questo progetto, che possono rendere questo robot uno strumento più efficace sia per uso domestico, come nel caso dei robot aspirapolvere e sia per usi militari come nel caso dell' esplorazione di un ambiente ostile.

Indice

Elenco delle figure

1

Ambiente di lavoro

In questo capitolo verranno trattati i software e le librerie utilizzati per la realizzazione di questo progetto, ovvero ROS (Robot Operating System), Gazebo, RViz, rospy e Robot Web Tools.

1.1 ROS e ROS Noetic Ninjemys su Linux Ubuntu

Figura 1.1: Copertina di ROS Noetic Ninjemys, tratta da http://wiki.ros.org/

ROS[1], acronimo di Robot Operating System, è un insieme di strumenti e di librerie open source[2] che rendono lo sviluppo di applicazioni robotiche più agevole.

La prima versione di ROS risale al 2007, da due dottorandi dell'Università di Stanford. I due studenti decisero di creare un sistema di base che fornisse un punto di partenza per tutti coloro che volessero sviluppare applicazioni robotiche.

[1]Per consultare il sito ufficiale di ROS, visitare https://www.ros.org/.
[2]Tipo di software libero da vincoli di copyright.

Il suo funzionamento è basato su un meccanismo di pubblicazione e sottoscrizione anonimo (publish/subscribe) che consente il passaggio di messaggi tra diversi processi.

Per questo progetto si è deciso di utilizzare la versione ROS Noetic Ninjemys[3] rilasciata il 23 Maggio 2020, e supportata fino a Maggio 2025 sul sistema operativo Linux Ubuntu 20.04. Tale scelta è stata presa per avere una maggiore compatibilità e stabilità dei pacchetti utilizzati.

La chiave del successo di ROS, oltre allo sviluppo costante di nuove librerie compatibili e alla facilità d'implementazione di esse, sta nella sua vasta comunità presente in rete.

Sono presenti infatti due pagine web dedicate alla comunità di ROS, ROS Answers[4] e ROS Discourse[5].

1.1.1 Modello a grafo

I processi ROS sono rappresentati come nodi in una struttura a grafo, collegati da archi chiamati topic. Ciascun nodo ROS può scambiare messaggi, fornire servizi o modificare dati condivisi. Tutto ciò è gestito da un nodo chiamato ROS Master.

All'avvio del software ciascun nodo deve dichiarare al nodo ROS Master se agirà in veste di Publisher (pubblicatore di messaggi) o di Subscriber (ricevitore di messaggi) comunicandone anche il tipo di messaggio che verrà scambiato.

Il ruolo del ROS Master è quindi quello d'impostare la comunicazione peer-to-peer[6] tra tutti i nodi.

Nodo	Il nodo rappresenta un processo ROS all'interno del modello a grafo.
Topic	Il topic si definisce come il bus di comunicazione su cui i nodi inviano e ricevono messaggi.
Servizio	Il servizio rappresenta un'azione che esegue un nodo e che porta a un unico risultato.

1.1.2 ROS 2

Nel 2017 comparve in rete la prima release[7] della versione due di ROS: ROS 2 - Ardent Apalone, la quale rimuoveva il concetto di nodo master ponendo tutti i nodi allo stesso livello.

Tale implementazione tuttavia non è ancora stabile e data la sua bassa diffusione molti pacchetti disponibili nella versione uno non sono ancora stati rilasciati, pertanto non è stata utilizzata per la realizzazione di questo progetto.

1.1.3 RViz

Tra gli strumenti messi a disposizione da ROS è disponibile RViz[8], un visualizzatore 3D. A differenza di Gazebo che, come verrà spiegato nella sottosezione 1.1.4, permette di simulare l'ambiente, RViz, dopo una semplice configurazione iniziale dei topic (sottosezione 1.1.1), rende visualizzabili le elaborazioni fatte dal robot e gli impulsi ricevuti dai sensori in tempo reale.

Si è deciso, al fine di agevolare la fase di sviluppo dell'applicazione, di utilizzare questo strumento.

[3] Per la documentazione completa riguardante ROS Noetic Ninjemys consultare `http://wiki.ros.org/noetic`.
[4] Per consultare il sito ROS Answer, visitare `https://answers.ros.org/questions/`.
[5] Per consultare il sito ROS Discourse, visitare `https://discourse.ros.org/`.
[6] Rete informatica nella quale i computer fungono nello stesso tempo da client e da server.
[7] Nuova versione di un software.
[8] Per consultare la pagina dedicata ad RViz, visitare `http://wiki.ros.org/rviz`.

1.1.4 Gazebo

Gazebo è un simulatore di robotica 3D open source: permette, infatti, di renderizzare realisticamente gli ambienti includendo illuminazione, ombre e texture di alta qualità come mostrato nella figura 1.2.

Fu utilizzato per alcune competizioni tra cui quelle organizzate dalla DARPA[9] (Defense Advanced Research Projects Agency), dalla NASA[10] (National Aeronautics and Space Administration) e da Toyota.

Data la sua affidabilità si è deciso di utilizzare Gazebo come strumento di simulazione per il progetto in questione.

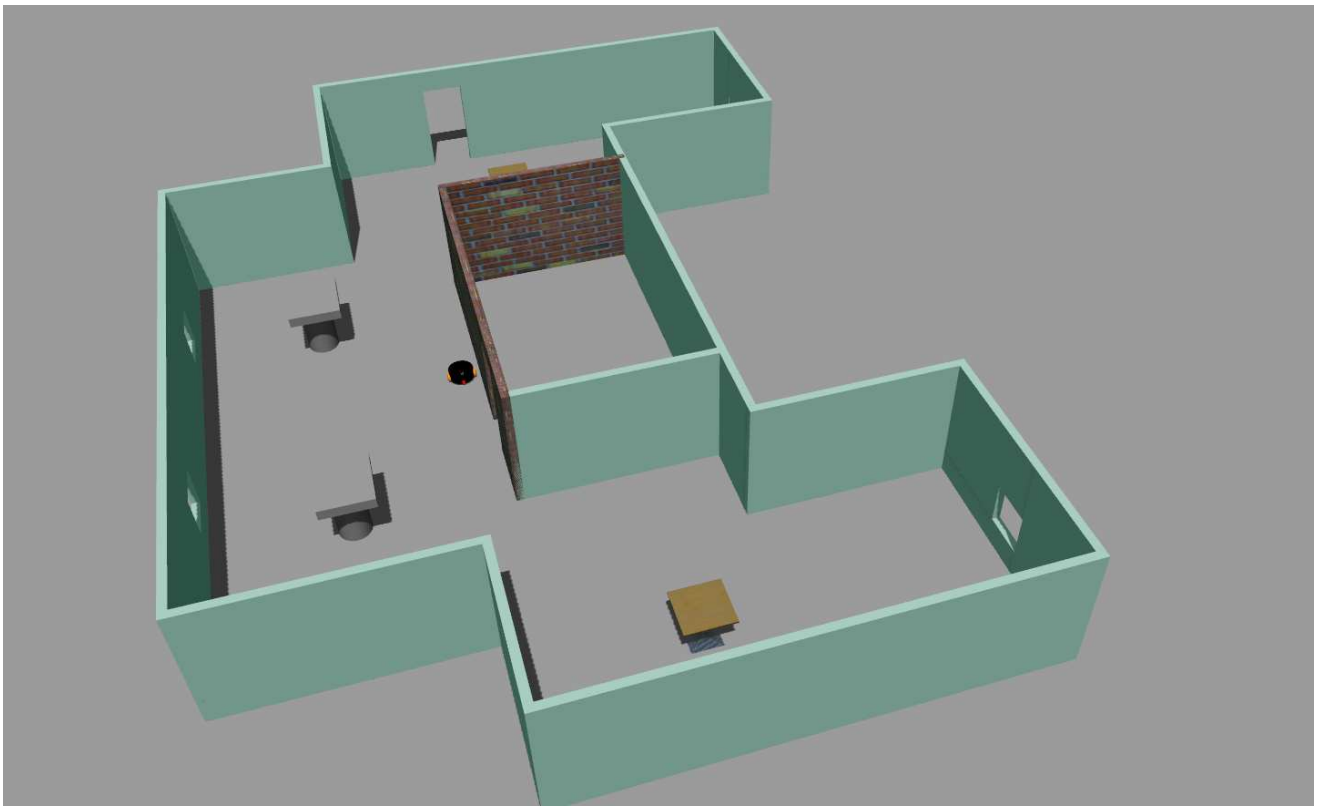

Figura 1.2: Visualizzazione dell'ambiente tramite Gazebo.

1.2 rospy

Come accennato nella sezione 1.1, ROS include un insieme di librerie per agevolare ulteriormente lo sviluppo di applicazioni robotiche, alcune di queste ancora in fase sperimentale[11].

Per questo progetto è stata utilizzata la libreria rospy[12] che grazie alla sua velocità d'implementazione e alla sua forte comunità presente nei forum permette di sviluppare applicazioni robotiche tramite il linguaggio Python[13] in maniera efficace.

[9] Per informazioni dettagliate riguardanti la competizione DARPA, consultare `https://en.wikipedia.org/wiki/DARPA_Robotics_Challenge`.

[10] Per informazioni dettagliate riguardanti la competizione organizzata dalla NASA, consultare `https://github.com/osrf/srcsim`.

[11] Per consultare la lista completa di librerie disponibili, visitare `http://wiki.ros.org/Client%20Libraries`.

[12] Per consultare la documentazione completa riguardante rospy, visitare `http://wiki.ros.org/rospy`.

[13] Per consultare il sito ufficiale di python, visitare `https://www.python.org/`.

Il progetto è stato strutturato come segue.

```
sensors/
    camera.py
    laser_scan.py
actuators/
    actuators.py
exploration/
    exploration.py
web/
    web.py
main.py
```

1.3 Robot Web Tools

Oltre agli strumenti illustrati in precedenza, ROS mette a disposizione i cosiddetti Robot Web Tools[14], una collezione di librerie adibite allo sviluppo web di applicazioni ROS.

Per la realizzazione di questo progetto sono state utilizzate le librerie JavaScript roslibjs[15], che permette d'interagire con ROS tramite browser grazie a rosbridge[16] che fa da intermediario tra la webapp e ROS, ros3djs[17], che permette la visualizzazione di elementi tridimensionali ed mjpegcanvas[18], che permette di gestire i flussi video di ROS affidandosi a web video server[19] che funge da tramite tra il browser e ROS.

[14] Per consultare gli strumenti e le librerie messe a disposizione da Robot Web Tools, visitare `http://robotwebtools.org`.
[15] Per consultare la documentazione completa riguardante roslibjs, visitare `http://wiki.ros.org/roslibjs`.
[16] Per consultare la documentazione completa di rosbridge, visitare `http://wiki.ros.org/rosbridge_suite`.
[17] Per consultare la documentazione completa di ros3djs, visitare `http://wiki.ros.org/ros3djs`.
[18] Per consultare il codice sorgente della libreria mjpegcanvasjs, visitare `https://github.com/rctoris/mjpegcanvasjs`.
[19] Per consultare la documentazione completa riguardante web video server, visitare `http://wiki.ros.org/web_video_server`.

2

Progettazione del robot

In questo capitolo verranno trattati i linguaggi utilizzati per la modellazione grafica dell'ambiente e del robot, oltre che alla implementazione vera e propria di tutti i componenti.

2.1 Linguaggi di modellazione

2.1.1 URDF

URDF[1] acronimo di Unified Robot Description Format è un formato di file XML utilizzato in ROS per descrivere tutti gli elementi di un robot. La sua sintassi è semplice, e di seguito ne è riportato un esempio

```
<robot>
  <link>
    <pose>...</pose>
    <inertial>...</inertial>
    <collision>...</collision>
    <visual>...</visual>
  </link>
  ...
</robot>
```

Per modellare graficamente un oggetto è necessario descriverne la posizione iniziale indicata dal tag `pose`, l'inerzia contraddistinta dal tag `inertial`, ovvero la massa (definita in chilogrammi) e una matrice 3x3 che ne definisce l'inerzia rotazionale, la fisica delle collisioni identificata dal tag `collision` e la rappresentazione visuale descritta nel tag `visual`.

Dopo aver modellato ogni singolo elemento del robot si rende essenziale unire le varie parti con delle giunzioni che possono essere continue, ad esempio nel caso delle ruote che necessitano di essere in movimento, oppure fissate, ne sono un esempio i sensori fissi posti sul robot.

Nonostante possa sembrare efficiente, a causa della sua forte ridondanza, ne viene sconsigliato l'utilizzo per modelli complessi.

[1] Per consultare la documentazione completa riguardante il linguaggio URDF, visitare `http://wiki.ros.org/urdf`.

2.1.2 Xacro

Xacro[2], acronimo di XML Macro, venne introdotto per risolvere i problemi di ridondanza spiegati nella sezione precedente, infatti, è un macro linguaggio basato su XML che aggiunge all'URDF (sottosezione 2.1.1) tre elementi

- Costanti.
- Espressioni matematiche semplici.
- Macro, ovvero interi tag che possono essere riutilizzati.

Al fine di progettare in maniera efficiente il robot e l'ambiente è stato utilizzato sia URDF che Xacro.

2.2 Componenti del robot

Il robot è stato composto dalle seguenti parti:

- Uno chassis circolare di raggio venticinque centimetri, di altezza venti centimetri e del peso di quindici chilogrammi.
- Due ruote motrici ognuna pesante cinque chilogrammi, di raggio dieci centimetri e di spessore cinque centimetri.
- Una videocamera del peso di cento grammi.
- Un sensore lidar[3] hokuyo[4]

Poiché il robot presenta solo due ruote si è reso necessario aggiungere due sfere incastonate al di sotto dello chassis che permettano di mantenere l'equilibrio durante il movimento.

Per rendere possibile il funzionamento di tutte le componenti è stato necessario implementare dei driver[5].

2.2.1 Camera

Come spiegato nella sottosezione 1.1.1 ogni nodo comunica i dati attraverso dei topic.

Si è reso necessario implementare un driver con lo scopo di rendere idonee le informazioni catturate dalla videocamera per poter essere scambiate tramite i topic su ROS.

Per l'implementazione della camera nel progetto si è utilizzato il driver fornito da Gazebo (sottosezione 1.1.4), ovvero `libgazebo_ros_camera`[6].

Si sono applicate le impostazioni riguardanti formato, risoluzione e frame rate seguendo i criteri indicati di seguito.

[2]Per consultare la documentazione completa riguardante il linguaggio Xacro, visitare `http://wiki.ros.org/xacro`.

[3]Tecnica di telerilevamento che permette di determinare la distanza di un oggetto o di una superficie utilizzando un impulso laser.

[4]Produttore globale di sensori attui alla tecnologia dell'automazione, con sede in Giappone. Per maggiori informazioni consultare `https://www.hokuyo-aut.jp/`.

[5]Insieme di procedure software che permettono il controllo di una periferica hardware.

[6]Consultare `http://docs.ros.org/en/diamondback/api/gazebo_plugins/html/classgazebo_1_1GazeboRosCamera.html` per la documentazione completa.

- Risoluzione dell'immagine di 800px per 800px.
- Formato R8G8B8, ovvero a 24bit, 8bit per ciascun canale.
- Frame rate di 30 fps.

Si sono infine definiti i topic `rgb/camera_info`, per visualizzare le impostazioni della videocamera, e `rgb/image_raw`, per visualizzare l'immagine catturata dalla videocamera.

Di seguito viene riportato un estratto della sua implementazione

```
<gazebo reference="camera">
  <sensor type="camera" ...>
    <update_rate>30.0</update_rate>
    <camera ...>
      <image>
        <width>800</width>
        <height>800</height>
        <format>R8G8B8</format>
      </image>
          </camera>
    <plugin ... filename="libgazebo_ros_camera.so">
      <imageTopicName>rgb/image_raw</imageTopicName>
      <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
    </plugin>
  </sensor>
</gazebo>
```

2.2.2 Lidar

Poiché anche il sensore lidar deve poter inviare dati agli altri nodi, per favorirne il suo funzionamento si è reso indispensabile utilizzare un driver dedicato.

Come nel caso spiegato alla sottosezione 2.2.1 anche per il sensore lidar è stato utilizzato il driver fornito da Gazebo, ovvero `libgazebo_ros_laser`[7].

Si sono definiti per emulare il comportamento di un sensore hokuyo:

- La portata del sensore, da 10 centimetri a 30 metri con una precisione di un centimetro.
- L'ampiezza del sensore, di 180 gradi.
- L'accuratezza delle rilevazioni, circa del 99,7 percento.

Si è inoltre impostato come canale per la trasmissione dei valori rilevati dal sensore il topic `scan`.

Di seguito ne è consultabile un riassunto dell'implementazione.

[7]Per consultarne la documentazione visitare `http://docs.ros.org/en/diamondback/api/gazebo_plugins/html/group__GazeboRosLaser.html`.

```
<gazebo reference="hokuyo">
  <sensor type="ray" ...>
    <horizontal>
      <min_angle>-1.570796</min_angle>
      <max_angle>1.570796</max_angle>
    </horizontal>
    <range>
      <min>0.10</min>
      <max>30.0</max>
      <resolution>0.01</resolution>
    </range>
    <noise>
    </noise>
    <plugin ... filename="libgazebo_ros_laser.so">
      <topicName>/scan</topicName>
    </plugin>
  </sensor>
</gazebo>
```

2.2.3 Motori

Come accennato nella sezione 2.2 il robot presenta due ruote motrici, comandate da due attuatori. Per controllarne il funzionamento si è implementato il driver fornito da Gazebo `libgazebo_ros_diff_drive`[8].

Tale plugin, oltre permettere la configurazione delle impostazioni di base atte al funzionamento dei motori, definisce anche i topic:

- `odom`, dove viene pubblicata l'odometria[9].
- `cmd_vel`, dove vengono pubblicate le istruzioni per movimentare gli attuatori.
- `tf`, dove viene pubblicata la trasformazione dell'odometria dalle ruote allo chassis.

Si è resa necessaria la trasformazione dell'odometria, poiché come visionabile alla Figura 2.1 non sarebbe stato accurato assumere che ruote e chassis si trovino allo stesso punto.

[8] Per consultarne la documentazione completa, visitare `http://docs.ros.org/en/noetic/api/gazebo_plugins/html/classgazebo_1_1GazeboRosDiffDrive.html`.

[9] Tecnica per stimare la posizione di un veicolo su ruote basato sulle informazioni provenienti da sensori che misurano lo spazio percorso.

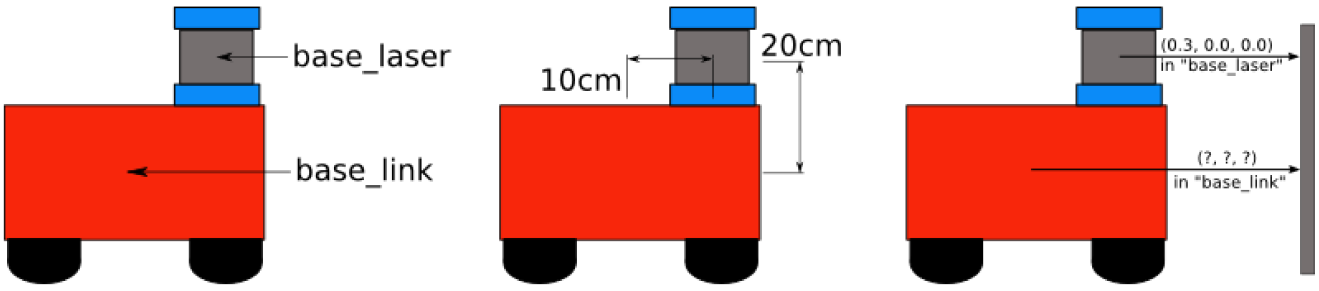

Figura 2.1: Trasformazione dal sensore laser allo chassis, immagine tratta da `http://wiki.ros.org`.

3

Stack di navigazione

In ROS lo stack di navigazione viene utilizzato per la navigazione in mappe 2D. Riceve come input l'odometria, le rilevazioni del sensore lidar e una posizione che si vuole raggiungere trasformandole in movimenti del robot.

In questo capitolo viene spiegato l'approccio SLAM e le tecniche utilizzate per implementare lo stack di navigazione all'agente.

3.1 SLAM

SLAM, acronimo di Simultaneous Localization and Mapping è una tecnica computazionale che permette la costruzione o l'aggiornamento di una mappa tenendo traccia della posizione di un agente al suo interno.

Per rendere funzionante la navigazione è quindi necessario implementare la mappatura, la localizzazione e la pianificazione del percorso (path planning); tuttavia se l'ambiente cambia (ne sono un esempio le comuni case, dove si possono aprire e chiudere porte) o le stime sono inaccurate si rende necessario l'ausilio di algoritmi per la rilevazione e l'evitamento degli ostacoli e del perfezionamento della mappa.

3.1.1 Mappatura

Per la creazione della mappa si è utilizzato il pacchetto ROS GMapping[1], che fornisce un nodo che riceve periodicamente informazioni dal sensore lidar (sottosezione 2.2.2) e ne pubblica la mappa nel topic `map`. Il pacchetto, inoltre, dalla sua versione `1.1.0` rende disponibile l'entropia sulla posizione del robot. Tale funzione, vista la sua instabilità non è stata utilizzata.

Si è reso necessario inoltre l'ausilio di un altro pacchetto, ROS Map Server[2], che oltre a permettere di salvare una mappa, la rende disponibile come servizio (sottosezione 1.1.1), cosa che si renderà essenziale successivamente (sottosezione 3.1.3).

3.1.2 Localizzazione

Per avere una localizzazione abbastanza fedele dell'agente nell'ambiente si è deciso di non utilizzare né l'entropia (sottosezione 3.1.1) né l'odometria (sottosezione 2.2.3).

[1] Per consultarne la documentazione completa, visitare `http://wiki.ros.org/gmapping`.
[2] Per consultarne la documentazione completa visitare `http://wiki.ros.org/map_server`.

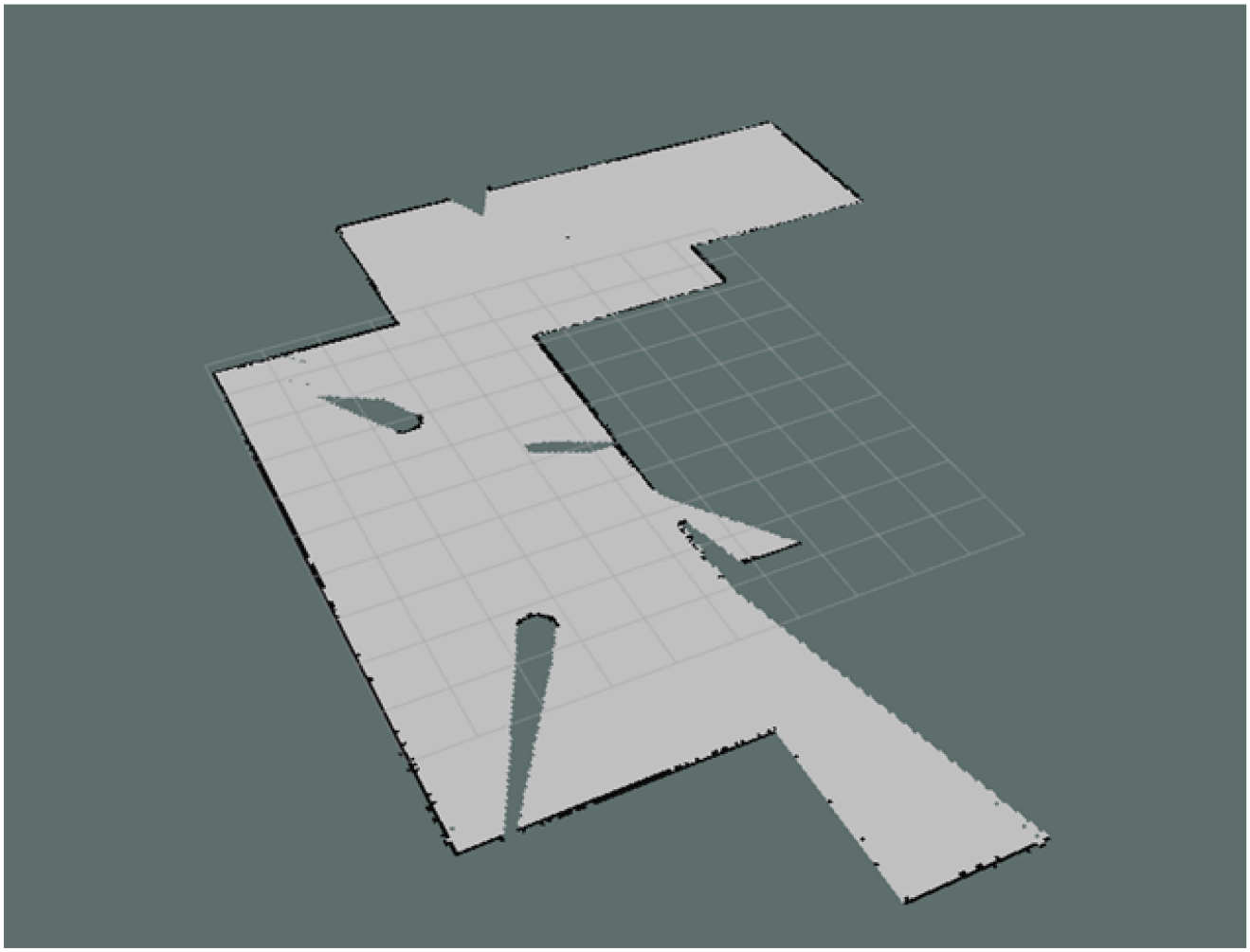

Figura 3.1: Mappatura dell'ambiente tramite GMapping

Si è implementato il pacchetto AMCL[3], acronimo di Adaptive Monte Carlo Localization, che utilizza un particle filter[4] per stimare la posizione e l'orientamento del robot.

Al suo avvio la distribuzione delle probabilità della posizione del robot sono uguali in ciascun punto della mappa, questo perché è ignota la posizione iniziale del robot, ogni volta che il robot si muove, tramite una stima Bayesiana ricorsiva[5], si rende la distribuzione delle probabilità convergente in un unico punto, ovvero la posizione del robot.

Tale pacchetto offre quindi un nodo che pubblica la posizione dell'agente nel topic `particlecloud`.

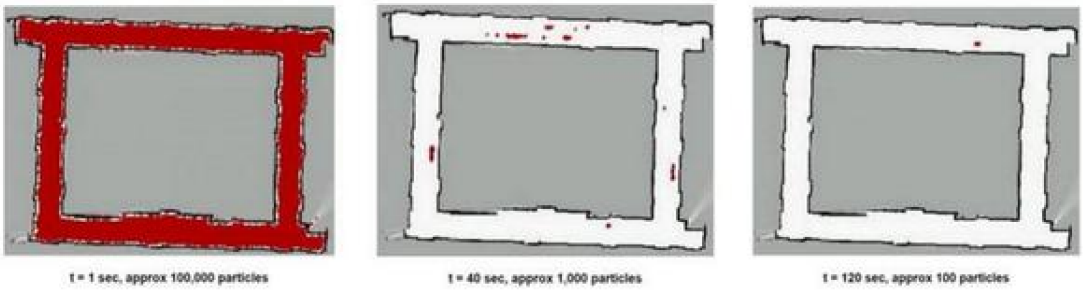

Figura 3.2: Esemplificazione dell'AMCL, tratto da `https://roboticsknowledgebase.com/`

3.1.3 Path planning

Dopo aver implementato la mappatura (sottosezione 3.1.1) e la localizzazione (sottosezione 3.1.2) si è proceduto al Path Planning[6].

In letteratura sono disponibili diversi algoritmi per la risoluzione di questo problema computazionale, quello utilizzato per la realizzazione di questo progetto è `base local planner`[7], che fornisce un'implementazione dell'algoritmo Trajectory Rollout e del Dynamic Window Approach.

Il suo funzionamento consiste nel campionare tutte le traiettorie che il robot potrebbe compiere e per ciascuna di esse eseguire una simulazione dallo stato attuale allo stato in cui si troverebbe se avesse compiuto quella traiettoria, assegnandogli un punteggio in base a diversi parametri prefissati, quali la distanza che ha dagli ostacoli e la vicinanza all'obiettivo. Esclude quindi le traiettorie illegali, ovvero quelle che andrebbero a scontrarsi con un ostacolo, e sceglie la traiettoria con il punteggio più alto.

Ripete questi passaggi fino a che non si è raggiunto l'obiettivo.

3.2 Movimento autonomo

Nella sezione 3.1 viene presentato come poter raggiungere un obiettivo fissato, per la realizzazione di questo progetto però si è reso necessario l'utilizzo di un algoritmo esplorativo. In questa sezione viene presentato, quindi, l'algoritmo utilizzato per questo progetto e alcuni algoritmi che possono essere implementati per migliorarne l'efficienza.

[3] Per consultarne la documentazione completa, visitare `http://wiki.ros.org/amcl`.
[4] Per una spiegazione completa consultare riguardante il particle filter, visitare `https://en.wikipedia.org/wiki/Particle_filter`.
[5] Per una spiegazione completa consultare `https://en.wikipedia.org/wiki/Recursive_Bayesian_estimation`.
[6] Il path planning è un problema computazionale per trovare una sequenza di configurazioni valide che sposta un oggetto dalla sorgente alla destinazione.
[7] Per consultarne la documentazione completa visitare `http://wiki.ros.org/base_local_planner`.

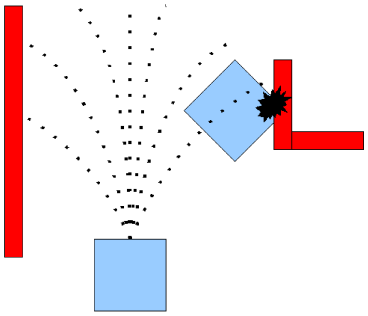

Figura 3.3: Visualizzazione grafica del funzionamento di Trajectory Rollout e del Dynamic Window Approach, immagine tratta da http://wiki.ros.org/

3.2.1 Esplorazione randomica

Grazie all'ausilio delle informazioni ricevute dal sensore lidar (come spiegato nella sottosezione 2.2.2), è stato possibile implementare un algoritmo esplorativo basato su tali dati. Ciò permette al robot di muoversi casualmente in tutto lo spazio evitando gli ostacoli, di seguito se ne fornisce uno pseudo-codice.

```
function exploration(laser):
  # Obstacle Threshold
  obst_thresh = 1
  if laser['front'] > obst_thresh and
    laser['left'] > obst_thresh and
    laser['right'] > obst_thresh:
          obstacle_state = 'No obstacle found'
  elif laser['front'] < obst_thresh and
    laser['left'] > obst_thresh and
    laser['right'] > obst_thresh:
          obstacle_state = 'Found obstacle in front'
  elif laser['front'] > obst_thresh and
    laser['left'] > obst_thresh and
    laser['right'] < obst_thresh:
          obstacle_state = 'Found obstacle in right'
  elif laser['front'] > obst_thresh and
    laser['left'] < obst_thresh and
    laser['right'] > obst_thresh:
    obstacle_state = 'Found obstacle in left'
  ...
```

Tuttavia utilizzare questo approccio può causare alcune criticità, di seguito vengono elencate le più rilevanti.

Inefficienza	L'agente può visitare più volte gli stessi spazi prima di visitare spazi nuovi.
Sicurezza	Nel caso di ambiente ostile l'esplorazione casuale può portare a movimenti irreversibili (vicolo cieco).

3.2.2 Metodo di esplorazione alternativo

In letteratura sono proposti molti algoritmi per l'esplorazione.

Uno tra i più efficienti è l'algoritmo LRTA*, acronimo di Learning Real Time A*. Tale algoritmo si basa sull'algoritmo di ricerca A* per una ricerca locale online[8]. Il suo funzionamento è basato su una funzione euristica aggiornata costantemente dai dati ottenuti dall'esplorazione. A ogni passo si sceglie il movimento che ha un costo minore, tale costo $f(x)$ è dato dalla somma del costo per raggiungere un determinato spazio $g(x)$ più il valore euristico su tale spazio $h(x)$, in caso di pari costo viene sempre scelta la cella inesplorata.

Il costo stimato viene quindi ricavato dalla formula

$$f(x) = h(x) + g(x). \tag{3.1}$$

L'algoritmo migliora la sua conoscenza dell'ambiente tramite l'apprendimento, aggiornando la stima euristica di ciascuno spazio.

Nonostante tale algoritmo sia molto più efficiente dell'esplorazione randomica (sottosezione 3.2.1) non risolve il problema della sicurezza legato all'irreversibilità di un movimento.

3.3 Controllo remoto

Al fine di ridurre la possibilità d'incorrere in errori fatali durante l'esecuzione del robot si è ammesso il controllo remoto tramite un controller.

È stata implementata tramite il pacchetto ROS Joy[9] la possibilità di utilizzare un generico controller accoppiabile al sistema operativo in uso.

Tale pacchetto fornisce un nodo che invia costantemente i dati ricevuti dal controller al topic `joy`, che, quindi, contiene informazioni riguardanti i tasti che sono stati premuti e le levette che sono state direzionate.

Per implementare il funzionamento del controller è, stato sufficiente accoppiare il direzionamento delle levette alla velocità dei motori, espressa in velocità angolare e velocità lineare.

Di seguito un estratto dal codice sorgente.

```
def joystick(self, data):
  self.coords.linear.x = 4*data.axes[1] # Up and down
  self.coords.angular.z = 4*data.axes[0] # Left and right
```

[8] Nella ricerca locale online l'agente si trova in un ambiente operativo ignoto o parzialmente conosciuto.

[9] Per consultarne la documentazione completa, visitare `http://wiki.ros.org/joy`.

4

WebApp

In questo capitolo verrà trattata l'implementazione della WebApp, che permette il controllo remoto del robot e la sua diagnostica in tempo reale.

4.1 Vue.js

Vue.js[1] è un framework[2] basato su JavaScript[3] che semplifica lo sviluppo di applicazioni Web.

Per questo progetto si è deciso di adottare Vue.js poiché rende più ordinato ed efficiente il codice sorgente.

Grazie ad alcune librerie fornite da ROS è stato possibile integrare la visualizzazione della mappa, la localizzazione del robot sulla mappa e l'immagine ricevuta dalla videocamera oltre che a interrompere l'esplorazione autonoma intervenendo utilizzando il controller, come spiegato nella sezione 3.3.

4.1.1 Stabilire una connessione con il robot

Il primo passo che si è affrontato per implementare le funzioni accennate nella sezione 4.1 è stato quello di stabilire una connessione con il robot. Poiché la webapp ammette il controllo tramite internet, è stato indispensabile implementare una cella di testo adibita all'inserimento dell'indirizzo IP del robot a cui è stata associata la porta 9090.

Grazie alla libreria fornita da ROS, roslibjs[4], si è instaurata la connessione con il robot.

Di seguito viene proposto un estratto del codice.

```
connect: function(){
  this.ros = new ROSLIB.Ros({ url: "ws://" + this.address + ":9090"});
  this.ros.on('connection', () => {console.log('Connesso!')});
},
disconnect: function(){ this.ros.close() }
```

[1] Per consultare il sito ufficiale di Vue.js, visitare `https://vuejs.org/`.
[2] Implementazione logica di supporto sul quale un software può essere sviluppato.
[3] Linguaggio di programmazione lato client utilizzato per lo sviluppo di applicazioni web.
[4] Per consultare la documentazione completa riguardante roslibjs, visitare `http://wiki.ros.org/roslibjs`.

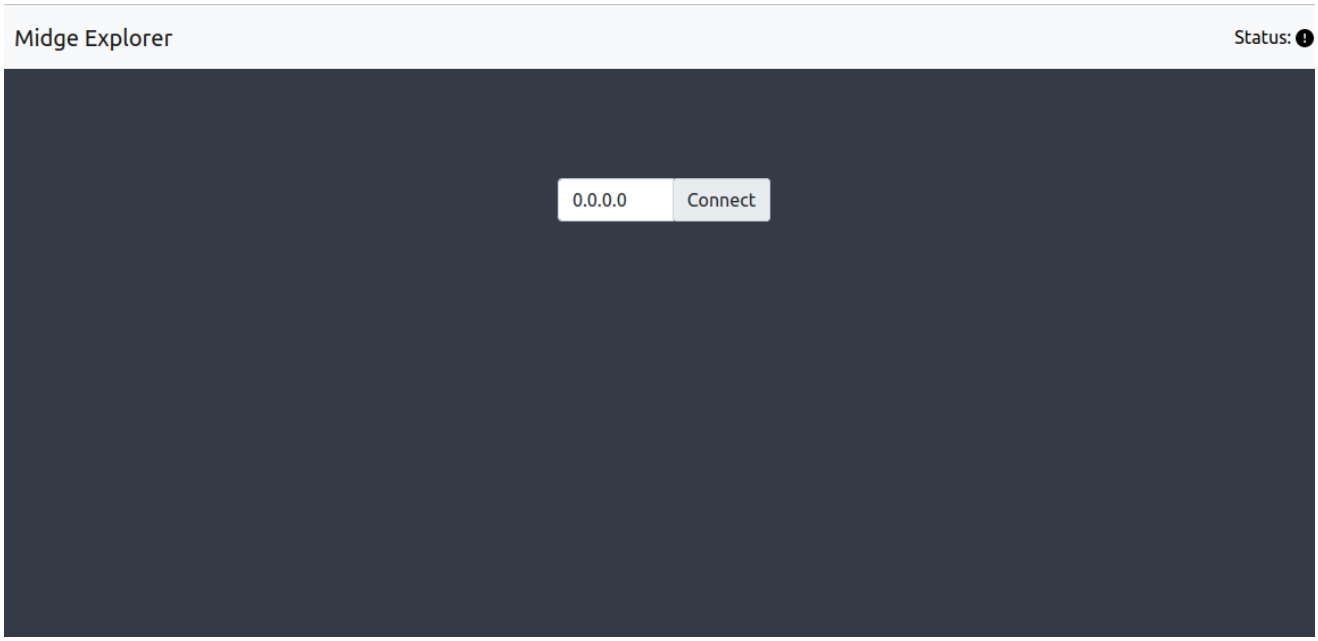

Figura 4.1: Schermata iniziale della webapp

4.1.2 Visualizzazione della mappa

Dopo aver stabilito la connessione al robot è stato necessario visualizzare la mappa e localizzare il robot al suo interno.

Per implementare queste funzioni si è fatto affidamento alle librerie mjpegcanvas e ros3djs[5]. Si è quindi stabilita una connessione sui topic `rgb/image_raw` (sottosezione 2.2.1), `map` (sottosezione 3.1.1) e `particlecloud` (sottosezione 3.1.2).

Di seguito viene proposto un estratto del codice sorgente.

```
/* Setup camera */
set_camera: function(){
  this.cameraViewer = new MJPEGCANVAS.Viewer({
    divID: 'live_camera',
    host: this.address,
    width: 720,
    height: 540,
    topic: '/camera/rgb/image_raw',
    port: 8080,
  })
},
/* Setup map */
set_navigator: function(){
  this.navigation_viewer = new ROS3D.Viewer({
    divID : 'navigator',
```

[5]Per consultare il codice sorgente della libreria ros3djs, visitare `https://github.com/RobotWebTools/ros3djs`.

```
    width : 720,
    height : 540,
  });
  this.tf_client = new ROSLIB.TFClient({
    ros : this.ros,
    rate : 10.0,
    fixedFrame : '/map'
  });
  this.occupancy_grid = new ROS3D.OccupancyGridClient({
    ros : this.ros,
    rootObject : this.navigation_viewer.scene,
  });
  this.pose_array = new ROS3D.PoseArray({
    ros: this.ros,
    topic: '/particlecloud',
    tfClient: this.tf_client,
    rootObject: this.navigation_viewer.scene
  });
}
```

4.1.3 Intervento umano durante l'esplorazione

Come spiegato nella sezione 3.3, per prevenire errori fatali durante l'esecuzione dell'agente si è ammessa la possibilità d'intervenire tramite un controller generico. Si è quindi creato un topic dedicato a modificare l'impostazione di navigazione, da esplorazione randomica a controllo remoto

Per integrarlo nella web app è stato sufficiente creare un pulsante nella webapp adibito a richiamare una funzione che inviasse il metodo di navigazione selezionato.

Di seguito un estratto del codice sorgente.

```
...
  this.autonomous = true;
...
set_navigation: function(){
    this.navigation_method = new ROSLIB.Topic({
    ros: this.ros,
    name: '/navigation_method',
    messageType: 'std_msgs/String'
  })
},


autonomous_function: function(){
```

```
  if(this.autonomous){
    this.autonomous = false
    let msg = new ROSLIB.Message({data:'joy'})
    this.navigation_method.publish(msg)
  }
  else{
    this.autonomous = true
    let msg = new ROSLIB.Message({data:'auto'})
    this.navigation_method.publish(msg)
  }
},
```

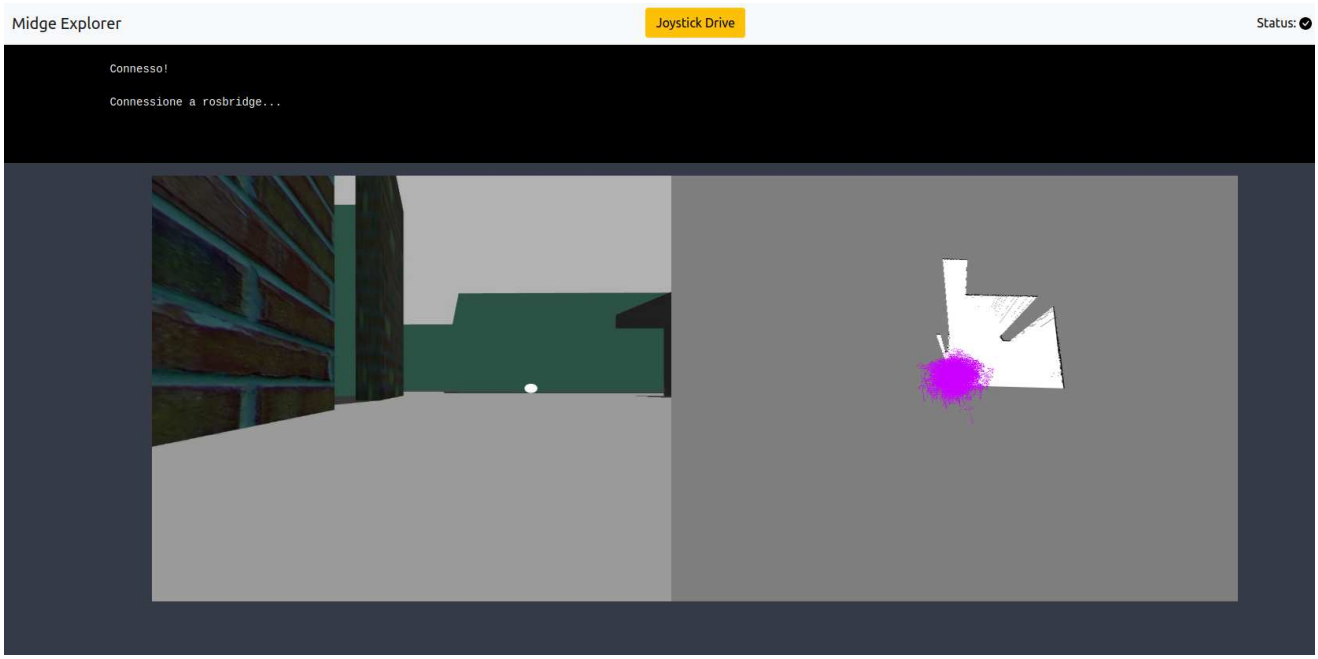

Figura 4.2: Pannello di controllo della webapp

5

Conclusioni

In conclusione, la robotica può rendere più sicuro ed efficiente ciò che all'uomo risulta pericoloso e difficile. Con le tecnologie attuali l'uomo e i robot sono due anelli imprescindibili della stessa catena. Con questo progetto si è voluto realizzare un robot che possa aiutare l'uomo a compiere ciò che da solo esegue con difficoltà e imprecisione.

Questo progetto è stato ideato per essere open source, e quindi utilizzabile e modificabile da chiunque per qualsiasi scopo.

Il progetto completo è reso disponibile all'indirizzo

`https://github.com/davidebassan/Midge-Explorer`.

5.1 Sviluppi futuri

5.1.1 Diagnostica dell'ambiente

Una primo sviluppo già avviato è quello di aggiungere sensori ambientali atti al rilevamento della temperatura, dell'anidride carbonica, dell'ossigeno e dell'umidità e rendere il robot autonomo nel verificare se un area è sicura o meno per la presenza l'uomo.

Questa implementazione troverebbe una collocazione ad esempio, in ambito militare, dove l'agente dovrebbe esplorare ambienti ostili.

5.1.2 Esplorazione collaborativa

Uno sviluppo permesso da ROS è quello di rendere cooperativa l'esplorazione di un ambiente, che oltre a renderne più rapida l'esecuzione, implementando lo sviluppo illustrato nella sottosezione 5.1.1 consentirebbe rilevazioni più accurate e una mappatura più fedele alla realtà.

Per tale sviluppo si propone la consultazione del pacchetto ROS rtt exploration[1], che propone una soluzione basata sull'algoritmo Rapidly-exploring random tree.

[1] per la documentazione completa riguardante rtt exploration consultare `http://wiki.ros.org/rrt_exploration`.

5.1.3 Rilevamento di soggetti dispersi

Uno sviluppo che si ritene essenziale nel caso di un utilizzo in ambito militare è rendere il robot capace di rilevare eventuali dispersi durante la sua esplorazione.

Tale sviluppo necessita dell'addestramento di una rete neurale addestrata a riconoscere volti e persone e ciò potrebbe non essere così semplice da attuare data la posizione della videocamera.

Per tale sviluppo si propone comunque la consultazione di OpenCV[2]

5.2 Ringraziamenti

Mi è doveroso dedicare questo spazio del mio elaborato alle persone che hanno contribuito, con il loro instancabile supporto, alla realizzazione di questo progetto.

In primis, un ringraziamento speciale al mio relatore Scagnetto Ivan, per i suoi indispensabili consigli e la sua immensa pazienza.

In secundis, un ringraziamento ai miei fantastici genitori che mi hanno supportato e sopportato in ogni istante del mio percorso di studi.

Infine, un ringraziamento particolare va a Francesca, Francesco, Giuseppe, Paolo, Michele, Matteo, Giovanni, Rachele e Agnese per il loro sostegno e la loro presenza durante questi anni che mi hanno permesso di esprimere al meglio le mie capacità.

Grazie a tutti, senza di voi non ce l'avrei mai fatta.

[2]Per la consultazione al sito ufficiale di OpenCV, visitare `https://opencv.org/`.

Bibliografia

[1] Sito ufficiale di ROS, `https://www.ros.org/`.

[2] Documentazione di URDF, `http://wiki.ros.org/urdf`.

[3] Documentazione di Xacro, `http://wiki.ros.org/xacro`.

[4] Simulatore Gazebo `https://en.wikipedia.org/wiki/Gazebo_simulator`.

[5] Documentazione di rospy `http://wiki.ros.org/rospy`.

[6] Robot Web Tools `http://robotwebtools.org/`.

[7] Sito ufficiale di ROS Wiki `http://wiki.ros.org/`.

[8] M. Csorba; J.K. Uhlmann; H.F. Durrant-Whyte (1997). *A sub-optimal algorithm for automatic map building*

[9] Documentazione di Gmapping `http://wiki.ros.org/gmapping`.

[10] Robotics Knowledgebase (2020). *Adaptive Monte Carlo Localization*

[11] D. Fox; W. Burgard; S. Thrun (1997). *The dynamic window approach to collision avoidance*

[12] Okpedia. *Ricerca LRTA**