SAPIENZA
UNIVERSITÀ DI ROMA

Navigazione autonoma di robot in ambiente indoor

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Leandro Maglianella
Matricola 1792507

Relatore

Prof. Giorgio Grisetti

Anno Accademico 2019/2020

Tesi discussa il 30 Ottobre 2020
di fronte a una commissione esaminatrice composta da:

Prof. Bruno Ciciani (presidente)
Prof. Ioannis Chatzigiannakis
Prof. Giuseppe De Giacomo
Prof. Paolo Di Giamberardino
Prof. Riccardo Lazzeretti
Prof. Paolo Liberatore
Prof. Marco Temperini

Navigazione autonoma di robot in ambiente indoor
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Versione: 30 ottobre 2020

Email dell'autore: maglianella.1792507@studenti.uniroma1.it

A te, lettore

Sommario

Questa relazione costituisce l'elaborato scritto per la prova finale del Corso di Laurea in Ingegneria Informatica e Automatica. L'argomento qui proposto e trattato è la navigazione autonoma di un robot base mobile in un ambiente indoor, ad esso inizialmente sconosciuto. Dopo una breve descrizione delle proprietà che caratterizzano un agente intelligente e del software utilizzato, saranno trattati in dettaglio le fasi di sviluppo del progetto: mapping, auto-localizzazione del robot nella mappa e path planning per raggiungere un goal. Il progetto è stato realizzato col linguaggio di programmazione C++ sul sistema operativo Ubuntu 16.04 utilizzando il framework ROS, il quale fornisce strumenti e librerie adatte alla programmazione di robot.

Ringraziamenti

Impiego questo spazio della relazione per ringraziare tutte le persone che hanno fatto parte del mio percorso di vita finora, la famiglia, gli amici e chiunque abbia incontrato nel mio cammino. Ringrazio tutti gli insegnanti che ho avuto, fin dall'asilo ad ora, e i miei genitori per avermi guidato e sostenuto nel corso degli anni: chi sono ora deriva dal vostro esempio. Con l'augurio che troverete interessante quanto state per leggere, buona prosecuzione.

Indice

Capitolo 1

Introduzione

Questa relazione esporrà un progetto nato in collaborazione con il corso di "Laboratorio di Intelligenza Artificiale e Grafica Interattiva", da me successivamente sviluppato e ampliato. Descriverà le varie fasi affrontate nella realizzazione di un sistema robotico di navigazione autonoma e saranno trattate le tecniche da me impiegate per produrre un software funzionante. Non si entrerà in profondità nella trattazione del codice, ma se ne descriverà generalmente la costruzione e il comportamento.
Utilizzo questo primo capitolo per introdurre i successivi e rendere chiaro il processo e percorso mentale di scrittura della relazione.
Nel secondo capitolo saranno dati i fondamenti e le nozioni chiave riguardanti l'intelligenza artificiale e le caratteristiche dei sistemi robotici, necessarie per comprendere gli argomenti successivamente trattati.
Il terzo capitolo riguarderà ROS (Robot Operating System), software sul quale è stato implementato l'intero progetto: sarà rapidamente descritta la sua storia, per poi concentrarsi sulle sue features, il suo design e i suoi vari pacchetti.
Dal quarto capitolo saranno descritte le fasi impiegate nella realizzazione in simulazione del progetto, in questo capitolo in particolare si parlerà della mappatura dell'ambiente indoor in cui in robot agisce.
Il quinto capitolo conterrà il cuore del progetto: la fase di auto-localizzazione del robot nel mondo, di pianificazione del cammino da seguire e infine di creazione del controllore che modulerà la velocità del robot in base alla presenza di ostacoli, per assicurarne degli spostamenti in sicurezza.

Capitolo 2

Fondamenti di Intelligenza Artificiale

2.1 Agenti intelligenti

Lo scopo di chi lavora nel settore dell'intelligenza artificiale è quello di produrre sistemi che siano in grado di pensare ed agire in modo razionale o umano: tali sistemi prendono il nome di agenti intelligenti. Un agente intelligente (Figura 2.1) è un'entità attiva nell'ambiente in cui si trova, da esso acquisisce percezioni grazie ai sensori di cui dispone e su di esso agisce in risposta, per mezzo di attuatori, con opportune operazioni o azioni (decise per l'appunto dalla sua componente intelligente). Una delle caratteristiche più importanti degli agenti intelligenti è infatti l'autonomia, il saper decidere di per sé il proprio comportamento senza il bisogno di un operatore che le controlli.

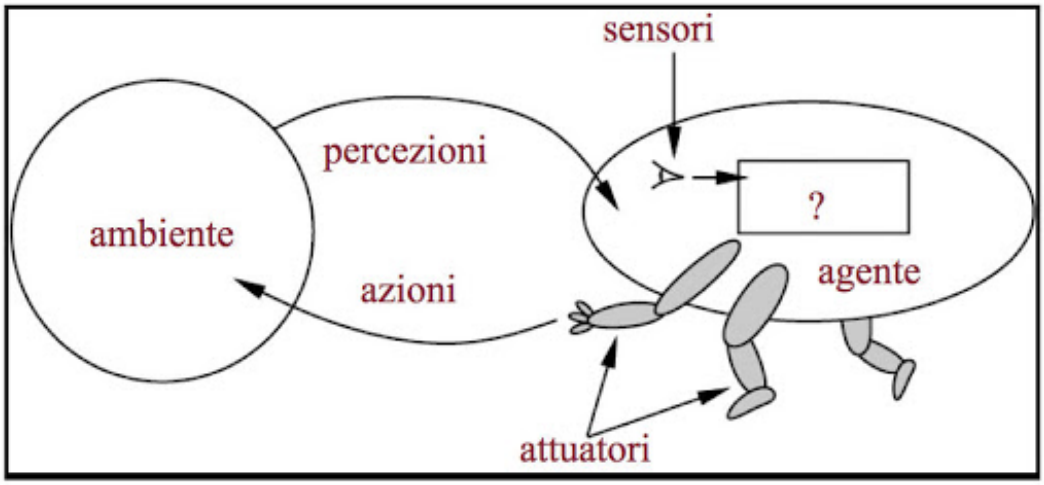


Figura 2.1. Agente intelligente

I sensori impiegati nell'ambito della robotica si dividono in diverse categorie: sensori propriocettivi o eterocettivi e attivi o passivi. I primi vengono usati per la misurazione di grandezze appartenenti all'agente stesso, come la velocità delle ruote,

l'accelerazione o il livello di batteria (un esempio di questi sensori sono gli encoders o le unità inerziali), i secondi misurano grandezze esterne all'agente come la distanza da altri oggetti, l'intensità luminosa o la temperatura. I sensori attivi effettuano misurazioni perturbando l'ambiente circostante (ad esempio i sensori a ultrasuoni o a infrarossi, i laser scanner e le camere a luce strutturata), al contrario i passivi sfruttano esclusivamente l'energia già presente nell'ambiente, non andandolo in alcun modo a modificare durante la loro attività di misurazione (ad esempio camere RGB e sensori tattili).

In particolare, l'agente intelligente che prenderemo in considerazione e su cui lavoreremo in questo progetto è una base mobile semplice, la quale ha la capacità di muoversi nell'ambiente trasportando un carico di sensori e attuatori. Come esempio di una semplice base mobile, si potrebbe pensare ad un robot commerciale come un robot aspirapolvere (Figura 2.2) che si sposta autonomamente svolgendo la propria funzione e cercando di evitare ostacoli. Altri esempi di livello più avanzato possono essere le piattaforme o le braccia meccaniche in fabbriche e centri logistici, che muovono e assemblano scaffali e merci, o anche droni elicotteri quadri-rotori per riprese o rilevazioni autonome, fin ad arrivare a modernissime tecnologie come automobili a guida autonoma e robot umanoidi. La base mobile del progetto è fornita di un semplice sensore laser scanner, dunque eterocettivo e attivo. Questo dispositivo è formato da un emettitore e un ricevitore, l'emettitore invia impulsi laser che sono ricevuti tramite riflessione dal ricevitore: attraverso la misura del tempo trascorso tra emissione e ricezione viene calcolata la distanza degli ostacoli nell'ambiente circostante. I laser scanner sono tra i sensori maggiormente utilizzati nel mondo per le sue proprietà: è infatti molto preciso e ha un ampio campo visivo. Un ulteriore impiego del laser scanner nel progetto, oltre a quello di distanziometro utile nella localizzazione e nel path planning, sarà quello di generare una scansione dell'ambiente nella fase di mapping, generando una mappa fedele delle aree circostanti percorse.

Figura 2.2

2.2 Sistemi robotici moderni ideali

I primi sistemi robotici venivano sviluppati in maniera monolitica, con un unico processo che eseguiva in parallelo su più thread tutte le operazioni del robot; tuttavia ben presto ci si rese conto che un'implementazione di questo tipo non garantiva al sistema una flessibilità e una sicurezza adeguate. Un programma sviluppato in tale maniera infatti, oltre a diventare estremamente complesso, era più delicato poiché il crash di un singolo thread comportava il crash dell'intero sistema. Per questo motivo oggi si preferisce una implementazione multi-processo dove le funzionalità sono isolate in tanti processi ed in cui ci è permesso, in caso di crash di un processo, di accorgercene e di riavviarlo senza disturbare gli altri processi contemporaneamente in esecuzione (le funzionalità sono quindi anche aggiornabili o sostituibili a runtime). La Figura 2.3 mostra un esempio di computation graph dei processi in esecuzione

parallelamente in un robot a navigazione autonoma moderno. Riassumendo i concetti finora espressi possiamo dunque dire che ogni processo scambia informazioni e offre servizi agli altri processi, ognuno avente funzionalità indipendenti e di cui lo sviluppatore deve garantirne la modularità.

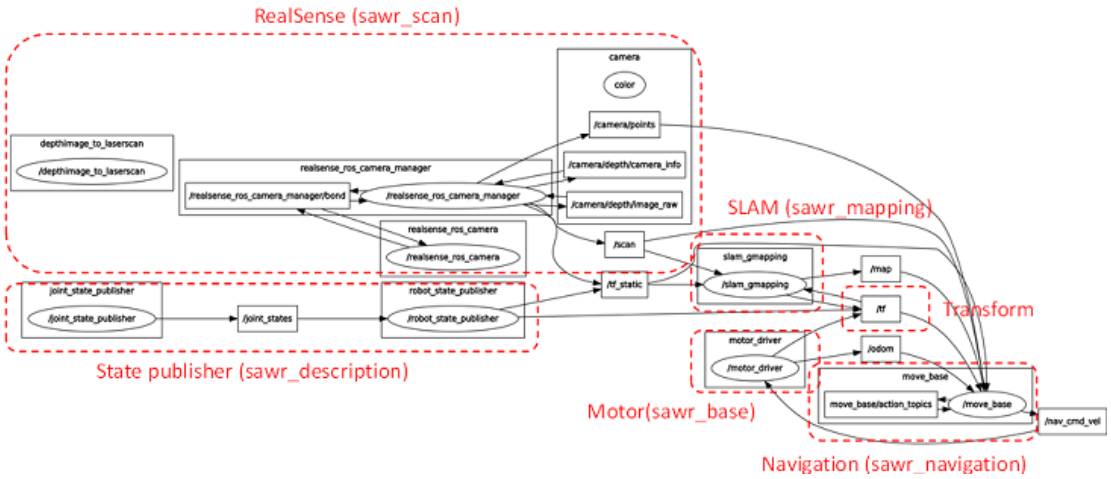


Figura 2.3. Computation graph di un sistema a navigazione autonoma

I vari processi sono disaccoppiati tramite meccanismi di inter-process communication (IPC) e per comunicare tra loro hanno due alternative: utilizzare apposite aree di memoria condivisa per scambio di dati oppure tramite messaggi. Solitamente viene usata la strategia dei messaggi che, seppur sia meno efficiente, garantisce una maggiore robustezza al sistema. Un ulteriore grande vantaggio dei messaggi è quello di poter permettere di scalare il sistema, attraverso la rete, su multipli computer che concorrono insieme all'esecuzione dell'applicazione robotica: come è infatti semplice immaginare i messaggi possono essere facilmente incapsulati in pacchetti UDP o TCP, inviabili poi via Internet.

Capitolo 3

Robot Operating System

Figura 3.1. Logo di ROS

Le funzionalità descritte nella sezione 2.2 sono implementate nei sistemi robotici per mezzo di middleware. Questi offrono gli strumenti adatti per l'implementazione e possiamo immaginarli come uno strato di programmi tra l'applicazione finale e il software, come una sorta di intermediario su cui poggiano tutti i moduli del sistema. Nel passato, alcuni tra i maggiori middleware per la robotica più popolari sono stati Carmen, Player/Stage, OpenRDK, OROCOS e Microsoft Robotic Studio, ognuno di questi differendo dagli altri per le proprie peculiarità. Successivamente poi col passare del tempo, anche a causa dell'andamento dell'evoluzione tecnologica riguardante CPU e schede di memoria e di rete, la scelta del middleware impiegato nello sviluppo di software per la robotica converse e portò alla nascita di Robot Operating System (abbreviato in ROS). Analizziamo adesso questo framework, su cui poggia il mio intero progetto.

3.1 Features di ROS

L'idea di ROS nasce nell'Università di Stanford, in California, per poi continuare ad essere sviluppato nel 2007 nel laboratorio di robotica Willow Garage dal lavoro congiunto di molti grandi talenti dell'informatica: venne costruito attorno al robot più complesso di allora, PR2 (Figura 3.2), che comprendeva moltissimi sensori e aveva le capacità di muoversi e usare braccia meccaniche. ROS consiste in un insieme di strumenti e application programming interfaces (API) che permettono di costruire sistemi, definire la struttura dati dei messaggi, usarli e inviarli e controllare i processi/nodi. ROS è fornito di un suo standard di file system, ciò va ad impor-

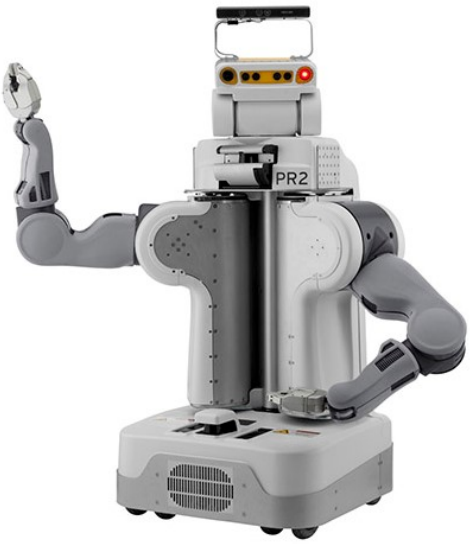
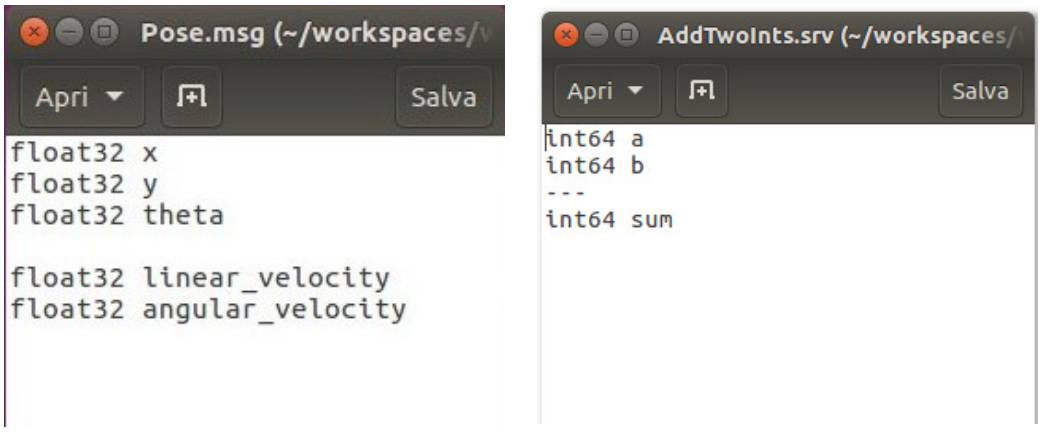


Figura 3.2. Robot PR2

re al programmatore una organizzazione coerente di cartelle che vada a sposare la sua convenzione, aumentando la comprensibilità del progetto. Vi è inoltre presente un utilissimo build system, catkin, che permette una rapida compilazione dell'intero sistema. In ROS sono compresi pacchetti di base per garantire funzionalità come la lettura dei dati sensoriali dei moltissimi tipi di sensori supportati, la navigazione, la manipolazione e il riconoscimento di oggetti; questi pacchetti possono essere utilizzati e ampliati dal programmatore nel proprio progetto. Ulteriori features che pilotarono il successo di ROS sono da ricercare nella sua riusabilità del codice, l'essere open source, la presenza di API che supportano molti linguaggi di programmazione ad ogni livello, il libero utilizzo di qualsiasi libreria senza essere legati unicamente a strumenti interni a ROS, la scalabilità via rete e la facilità di testing. Infine l'ambiente di ROS continua, durante il corso degli anni, ad essere ben manutenuto e ben disposto nei confronti dei propri utenti e ad integrare efficacemente librerie famose ed altri popolari progetti open source come Gazebo, OpenCV, MoveIt e Point Cloud Library.

3.2 Design di ROS

Nella figura 2.3 dello scorso capitolo già avevamo visto il computation graph di un sistema robotico in azione, scendiamo ora più nel dettaglio nella sua composizione esponendo i concetti di nodo, messaggio, topic e servizio. In ROS ogni processo in esecuzione è rappresentato da un nodo del grafo. Ogni nodo comunica con gli altri attraverso strutture dati messaggio definite dal programmatore che possono contenere al proprio interno comandi, dati o informazioni di qualsiasi genere.



(a) Esempio di definizione di un messaggio **(b)** Esempio di definizione di un servizio

Figura 3.3

La definizione di un messaggio avviene in un file di testo con estensione ".msg", in Figura 3.3.(a) si può ad esempio vedere il messaggio "Pose.msg" che contiene in sé cinque variabili di tipo float che determinano la velocità e le coordinate della posizione di un oggetto. Un flusso di messaggi forma un topic, rappresentati nel grafo di computazione dagli archi; ogni topic viene identificato da un nome e può

trasportare messaggi di un unico tipo. I servizi sono delle remote procedure calls (RPCs) e rappresentano operazioni a risultato singolo, di cui è noto il comportamento, l'inizio e la conclusione. Anche un servizio, come i messaggi, viene definito in una struttura dati in un file di testo con estensione ".srv" in cui i parametri di ingresso alla chiamata sono divisi dai valori di ritorno con tre trattini "- - -". Ad esempio, nella Figura 3.3.(b) è definito il servizio "AddTwoInts.srv" che prende in input due variabili intere e restituisce la loro somma (ovviamente affinché venga realmente restituita la somma bisogna scrivere un programma sottostante che svolga tale operazione). Un nodo può pubblicizzare e pubblicare messaggi su un topic o iscrivercisi per riceverli, inoltre può anche pubblicizzare servizi o chiamare servizi offerti da altri nodi.

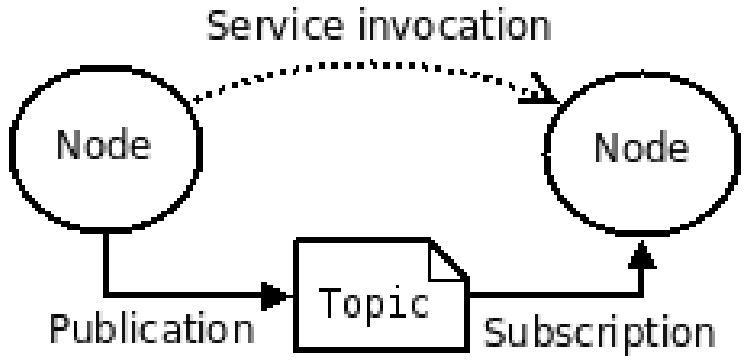


Figura 3.4. Trasmissioni tra nodi

Le subscribe e advertise di un nodo sono di norma le prime istruzioni nel suo processo. Tutte le informazioni riguardanti quali nodi pubblicizzano quali risorse sono conservate in un processo speciale chiamato "roscore", tramite esso i nodi vengono messi in comunicazione tra loro; una volta che la connessione tra due nodi viene stabilita, poi interagiranno direttamente senza aver bisogno di dover interpellare roscore. Roscore mantiene infine anche un server di parametri condivisi a tutti i nodi.

3.3 Transformation frame

Un sistema robotico è quasi sempre composto da moltissime componenti, montate in vari punti, che possono muoversi indipendentemente rispetto alle altre e che spesso si muovono in maniera non sincronizzata. Per poter effettuare azioni il robot ha bisogno di una tecnica con cui calcolare rapidamente le posizioni relative di tutte le sue parti rispetto a qualsiasi altra parte: è qui che entra in gioco il pacchetto "tf" di ROS. Ogni punto rilevante del robot viene dotato di un proprio reference frame, cioè un proprio sistema di riferimento di coordinate; conoscendo le distanze relative tra ognuno di questi punti e i recenti spostamenti tridimensionali di ognuno di essi tramite i giunti su cui sono montati, si costruisce il cosiddetto albero delle trasformate. Nell'esempio mostrato in Figura 3.5 si può vedere un piccolo robot con il corrispondente albero delle trasformate, ogni frame è collegato alla parte del robot a cui è legato da una freccia celeste. Per far in modo di trovare la posizione relativa di un punto rispetto ad un altro basterà semplicemente moltiplicare tra loro tutte le

matrici di trasformazione, ognuna rappresentata da un arco dell'albero (ricorcando bene di usare l'inversa della matrice quando un arco viene percorso nel senso opposto al senso della freccia nera). Conoscendo, attraverso l'albero, le posizioni relative delle componenti in più intervalli di tempo (poiché ovviamente le misurazioni vengono effettuate a tempo discreto) è possibile interpolarle, assumendo ad esempio che il robot si muova di moto uniforme, ed ottenere le posizioni per ogni istante. Non tratterò qui ulteriormente in profondità l'argomento: per i fini del progetto basti sapere che tutte queste funzioni sono implementate in tf e hanno rappresentato uno strumento indispensabile alla sua riuscita positiva.

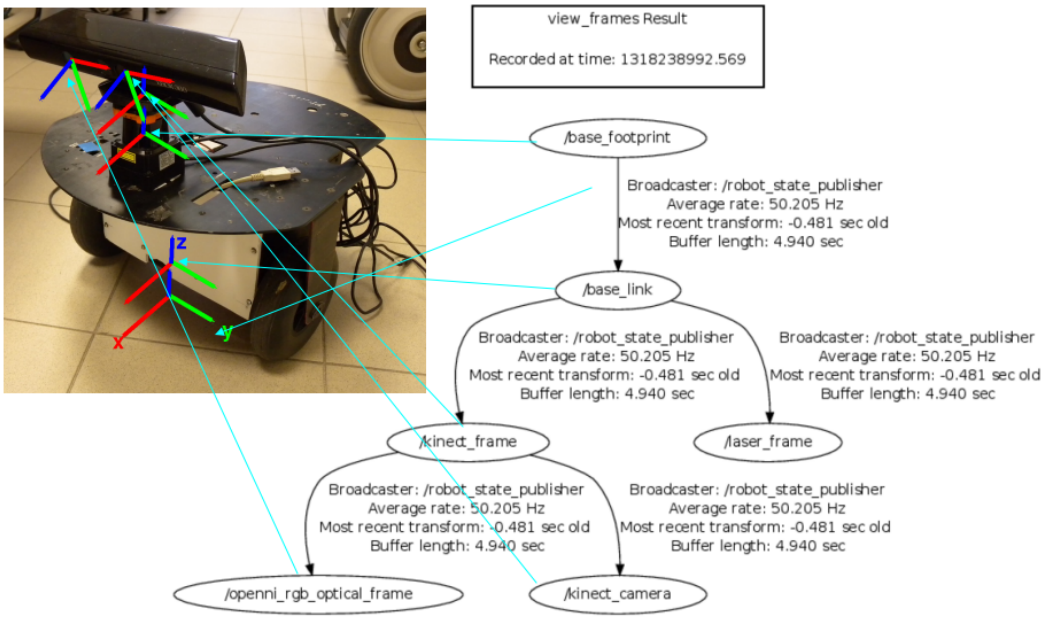


Figura 3.5. Albero delle trasformate di un robot

3.4 Ulteriori strumenti di ROS

In ROS si ha l'opportunità di usare un'enorme quantità di strumenti e pacchetti, tra cui stageros, rosbag, rviz, gmapping, move_base e actionlib; tutti questi saranno discussi nei prossimi capitoli man mano che verranno incontrati nella discussione del progetto.

Capitolo 4

Ambiente simulato e Mappatura

Entriamo ora nella vera e propria trattazione del progetto. In questo capitolo descriverò la prima fase della navigazione: la mappatura dell'ambiente in cui in robot si trova a lavorare, nel nostro caso un mondo simulato virtualmente.

4.1 StageROS

A causa della pandemia di Covid-19 di questi ultimi mesi non è stato possibile usare le attrezzature fisiche dei laboratori informatici de La Sapienza, pertanto l'intero progetto è stato realizzato e testato in un ambiente di simulazione virtuale a 2 dimensioni di ROS chiamato stageros. Ciò nonostante dal punto di vista del programma il robot simulato si comporta esattamente come se fosse fisico, interagendo con il suo ambiente, producendo messaggi sensoriali e ricavando da essi messaggi che lo informano della velocità alla quale muoversi per raggiungere il goal. Pertanto il progetto è, ovviamente, portabile e valido a lavorare anche in ambiente reale con robot fisici. Stageros utilizza le informazioni contenute in un file con estensione ".world" per creare il mondo virtuale, qui è definita la configurazione del mondo generalmente attraverso una immagine bitmap, della quale bisogna specificare a quanti metri nel mondo corrisponda ogni pixel. Inoltre sono definiti tutti i "modelli" del mondo come il pavimento, gli ostacoli, i robot e i loro sensori (volendo infatti si può lavorare su più robot contemporaneamente). Per ogni modello sono specificate e stabilite caratteristiche come le coordinate e le dimensioni e gli altri attributi facoltativi come ad esempio il colore. Per il mio progetto ho usato uno dei mondi preinstallati su ROS: "willow-erratic.world" (Figura 4.1). Come si può vedere il mondo è grande e per il progetto non necessitiamo di un mondo così esteso; perciò ci concentreremo solo su una sezione più ristretta di esso (Figura 4.2). La stanza di riferimento che prendiamo è quella contenente il robot (il quadratino blu), il quale ha montato su di lui un laser scanner Hokuyo UTM-30LX con un campo di visuale ampissimo, di quasi 360°; qui è inoltre presente un ostacolo particolare (il quadratino rosso), il quale è un ostacolo dinamico ma che tuttavia nel progetto non verrà utilizzato in alcun modo e sarà quindi considerabile come tutti gli altri ostacoli e muri statici, colorati in nero. Nonostante stageros sia, come ho detto, un simulatore 2-D, offre inoltre anche una utile visione in prospettiva 3-D del mondo (Figura 4.3).

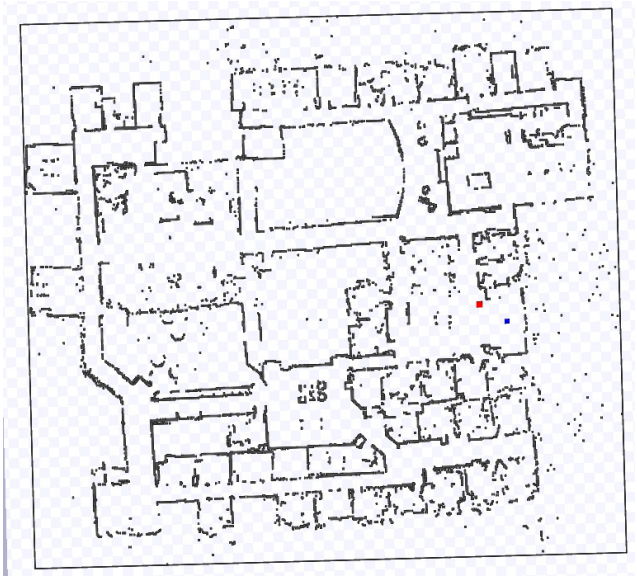


Figura 4.1. Mondo willow-erratic

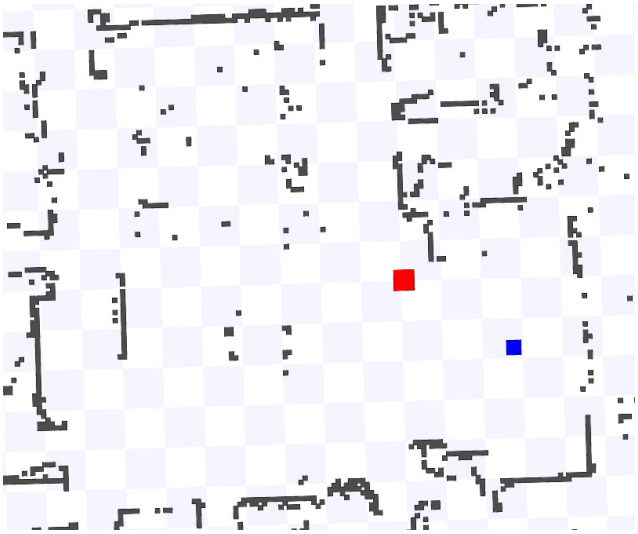


Figura 4.2. Particolare di willow-erratic

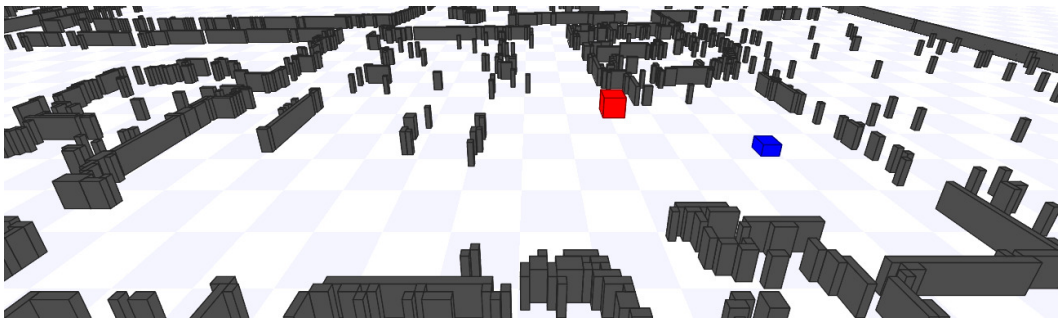


Figura 4.3. Prospettiva a 3 dimensioni

4.2 Rviz

Per il resto del progetto useremo un importantissimo strumento di ROS: rviz, un visualizzatore 3-D nel quale è possibile osservare qualsiasi dato sensoriale o topic in uso. La Figura 4.4 è rviz appena viene eseguito, usando il tasto "Add" in basso a sinistra si possono selezionare quali informazioni visualizzare. Ad esempio, aggiungendo il topic di transformation frame e del laser scanner si visualizza il contenuto della Figura 4.5. I tratti rossi sono i punti in cui il laser va a scontrare su degli ostacoli mentre al centro a destra è dove la tf localizza il robot.

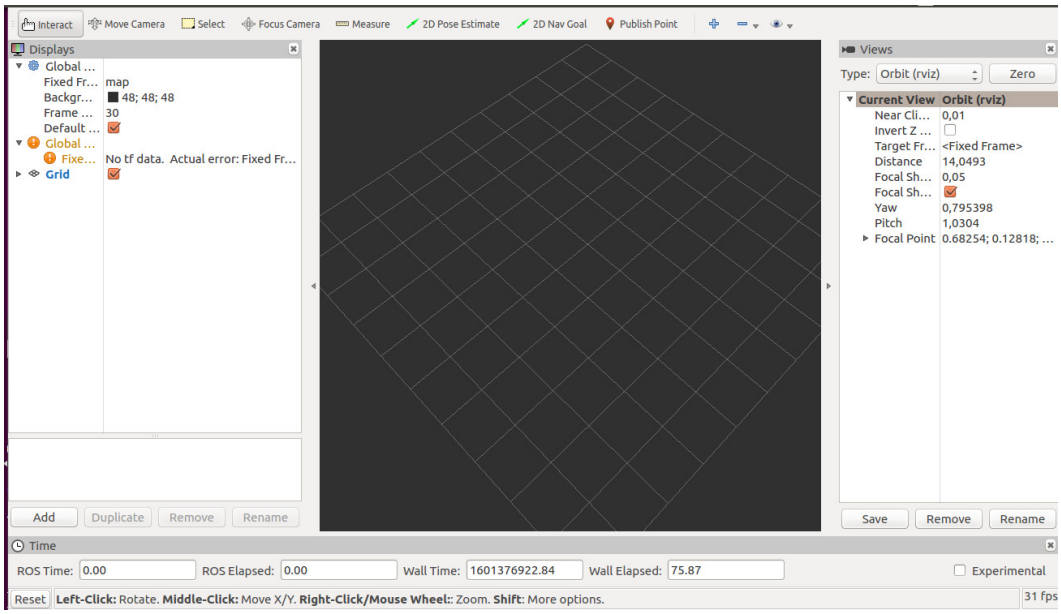


Figura 4.4. Schermata di rviz

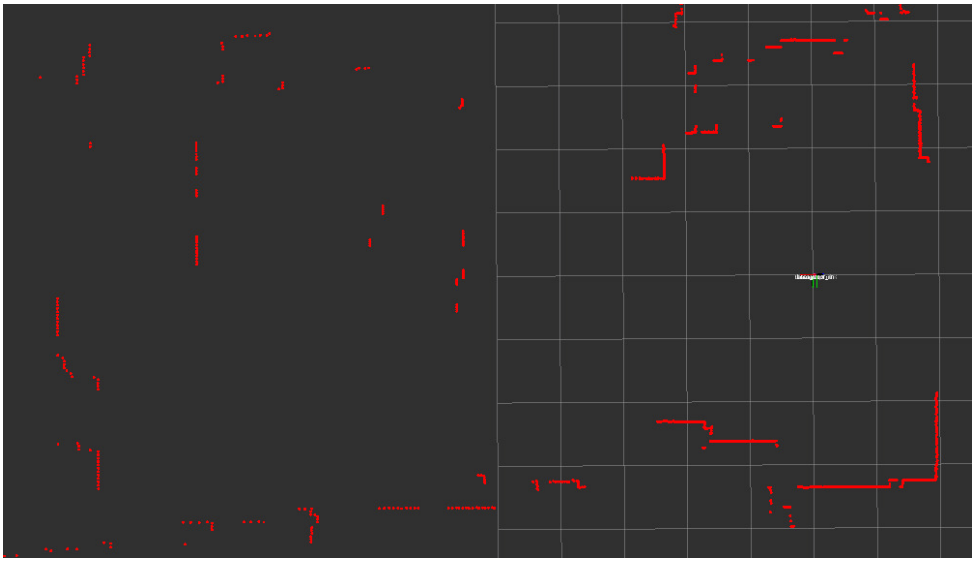


Figura 4.5. Visione del robot attraverso il laser scanner

4.3 Mappatura

Avendo ora posto le basi del mondo, veniamo alla prima fase per raggiungere l'obiettivo di una navigazione autonoma. Il robot deve essere in grado di capire l'ambiente tramite le misurazioni dei suoi sensori; per fare ciò deve prima di tutto popolare una struttura dati che rappresenterà lo stato del mondo e di sé stesso ai fini dell'esecuzione del proprio task. Trattandosi questo di un task di navigazione, per determinare lo stato saranno sufficienti le coordinate e l'orientamento del robot. Inoltre, ogni locazione non ha senso se essa non viene contestualizzata in una mappa, definita come la rappresentazione dell'ambiente in cui il robot opera. Pertanto il primo compito che ho svolto è stato fabbricarmi una mappa della stanza, per fare questo ho inizialmente costruito uno specifico file di ROS, una bag, con estensione ".bag". Le bags hanno la funzione di conservare messaggi, per poterli successivamente processare, analizzare o visualizzare e sono utili per replicare in maniera identica determinate situazioni. Ho creato la bag utilizzando il pacchetto rosbag, guidando manualmente tramite un joystick il robot nella stanza e registrando contemporaneamente tutti i messaggi che venivano trasmessi su tutti i topic. Una volta registrata la bag è possibile farne il playback, cioè si possono re-inviare nel canale di comunicazione di ROS tutti i messaggi salvati al suo interno. Questi messaggi così generati sono repliche esatte, praticamente identici a quelli originali, a meno di errori durante la registrazione della bag. Infine dunque per generare la mappa ho riprodotto la bag insieme a un nodo del pacchetto gmapping che implementava un algoritmo di simultaneous localization and mapping (SLAM): questo algoritmo consiste letteralmente nel disegnare la mappa utilizzando i dati sensoriali, in questo caso il laser scanner, mentre il robot si sposta.

Salvando il risultato dello SLAM ho ottenuto due file: map.yaml (Figura 4.6), che contiene meta-informazioni come la risoluzione (in questo caso 0.05, cioè ogni pixel della mappa rappresenta 5 centimetri del mondo) e l'origine della mappa, e map.pgm che è un'immagine della mappa creata (Figura 4.7, la vera immagine generata era molto grande, 4000×4000 pixels, qui ne mostro uno zoom di 600×400 pixels). Questa coppia di file saranno forniti ai moduli di localizzazione e pianificazione dal nodo map_server dello stesso pacchetto map_server e la mappa è, ovviamente, visualizzabile su rviz.

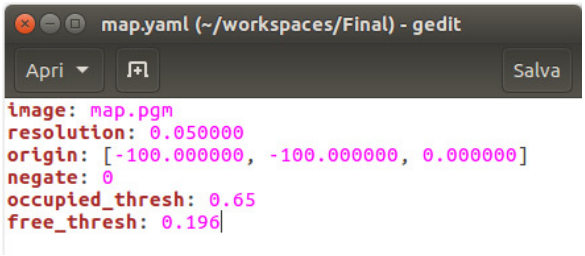


Figura 4.6. map.yaml

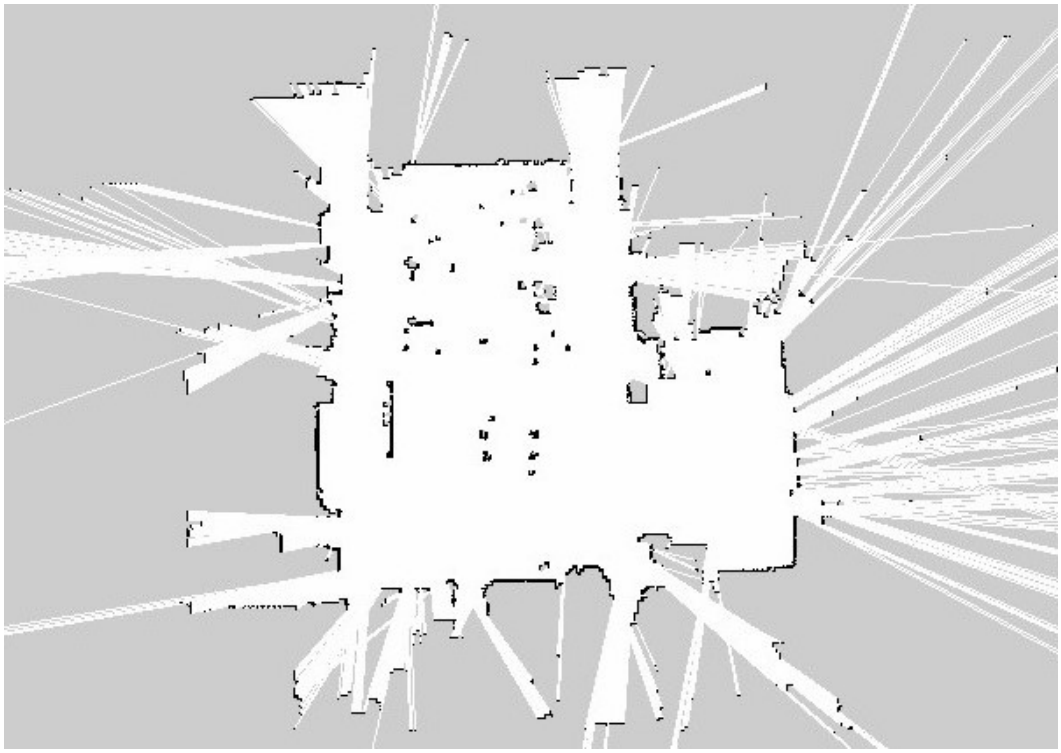


Figura 4.7. Zoom della mappa ricavata dalla bag da gmapping

Capitolo 5

Localizzazione e Pianificazione

5.1 Localizzazione

Affrontiamo adesso il modulo di localizzazione, con cui il robot determina la sua posizione in funzione delle misurazioni dai dati sensoriali e dalla conoscenza acquisita dalla mappa. È stato utilizzato per implementare questa funzionalità un algoritmo di particle filter. Senza entrare troppo in dettaglio nel codice, è stata definita una struttura dati "Particle", con attributi la posizione nel mondo e un peso (che indica la probabilità che il robot si trovi nella posizione della particella), e un vettore "ParticleVector" composto di tante Particle. Il programma va a piazzare questo insieme di particelle in tutto il mondo, ognuna di esse identificata da una freccetta la quale rappresenta un'ipotesi della posizione in cui il robot potrebbe essere.

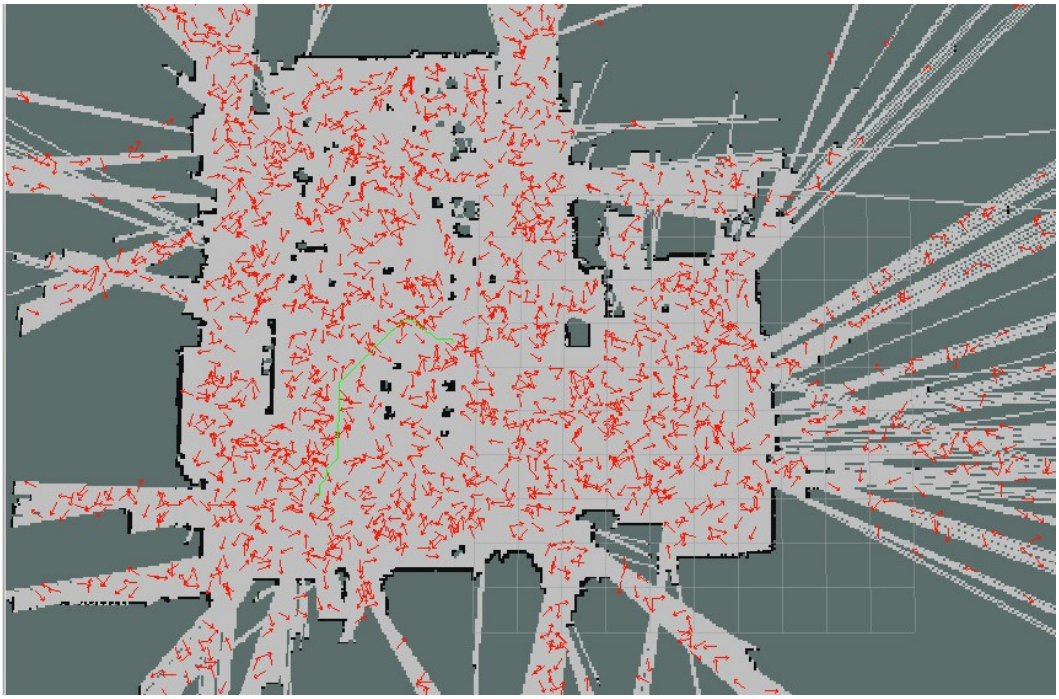


Figura 5.1. Inizio algoritmo di particle filter: robot non localizzato

La Figura 5.1 mostra la distribuzione statistica iniziale delle particelle, come si può vedere sono inizialmente tutte posizionate omogeneamente ovunque nella mappa e orientate in maniera randomica. A questo punto, attraverso i dati sensoriali del laser e le loro variazioni a seguito di piccoli spostamenti del robot, le posizioni e i pesi della distribuzione di particelle vengono statisticamente aggiornati. In poco tempo, man mano che le particelle con peso minore vengono eliminate e sostituite da particelle vicine a quelle con il peso maggiore, tutte le particelle convergono alla posizione effettiva del robot, con le frecce correttamente orientate nella direzione puntata dal robot (Figura 5.2).

Figura 5.2. Conclusione algoritmo di particle filter: robot localizzato

Il nodo di localizzazione continuerà la sua esecuzione anche durante l'esecuzione del modulo di pianificazione, come assicurazione che il robot rimanga sempre ben localizzato.

5.2 Pianificazione

5.2.1 Actionlib

Introduciamo actionlib, un altro pacchetto di ROS indispensabile nella fase di pianificazione. A differenza dei servizi, che rappresentano funzioni istantanee del robot, le azioni vengono usate quando il compito da svolgere è duraturo nel tempo e si ha bisogno della possibilità di poter ricevere feedback sull'avanzamento dell'azione e, se necessario, di poter annullare il proseguimento dell'azione nel mentre della sua esecuzione. La definizione delle azioni avviene in un file di testo con estensione ".action" e contiene al suo interno le definizioni, separate da tre trattini "- - -", del goal, del risultato da raggiungere e del feedback. La Figura 5.3 riporta un esempio di definizione di un'azione, in cui sono stabilite variabili necessarie al posizionamento del goal e alla definizione del result e del feedback.

Il goal descrive la configurazione finale del mondo che vogliamo ottenere, nel nostro task di pianificazione conterrà informazioni sulla posizione che il robot dovrà raggiungere (quindi le coordinate e l'orientamento). Il feedback riporterà invece continuativamente la posizione attuale del robot durante il suo spostamento sul cammino calcolato, infine il risultato è il segnale che dovrà essere inviato non appena il compito terminerà, andando a informare della riuscita o del fallimento dell'azione. Dalla definizione dell'action, ROS genera automaticamente durante la compilazione del codice dei file ".msg" contenenti le definizioni di messaggi con cui verranno svolti tutti gli scambi di informazioni tra i nodi interessati all'azione, andando a creare una sorta di rapporto client/server tra loro. Grazie a questo pacchetto siamo costantemente al corrente della configurazione del robot rispetto al goal e, grazie alla conoscenza degli effetti delle sue azioni sul suo stesso stato, potremo determinare quale sequenza di azioni eseguire per arrivare ad una configurazione del mondo desiderata.

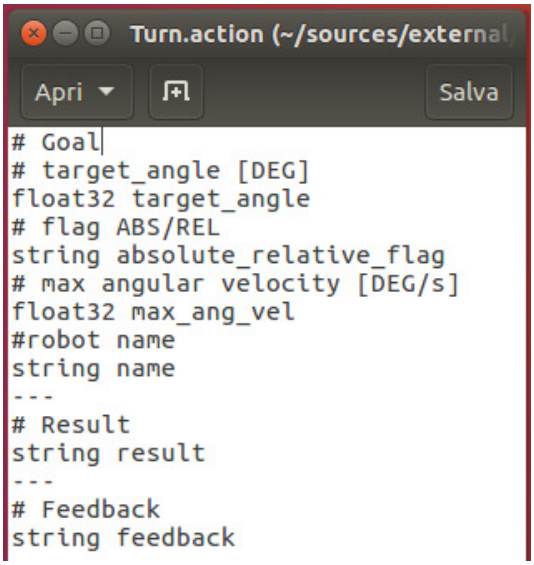


Figura 5.3. Esempio di definizione di un'azione

5.2.2 Distance map

A questo punto non ci resta che trovare il modo di determinare, se esiste, un cammino per raggiungere il goal dato. Sfruttando la mappa generata dalla fase di mappatura e fornita da map_server, viene definita e computata una struttura dati "distance_map". Questa mappa è formata da tante "DistanceMapCell", ognuna rappresentante un pixel della mappa e avente come attributi "distance" (la distanza dall'ostacolo più vicino), "parent" (puntatore alla DistanceMapCell ostacolo più vicina) e le proprie coordinate. Per prima cosa, notando che la mappa (Figura 4.7) rappresenta in

bianco le regioni libere del mondo, in nero gli ostacoli e in grigio le aree sconosciute, viene generata una "int_map": una mappa di semplici numeri interi, in cui vengono indicati con numeri interi non negativi progressivi i pixel neri e grigi (il numero così attribuito ad un ostacolo va a rappresentare la cella come suo id) mentre invece tutti i pixel bianchi sono indicati con -1. Agendo in questo modo i pixel sconosciuti sono considerati come se fossero ostacoli, ciò è stato fatto per sicurezza per non permettere al robot di addentrarsi in aree di cui non possegga informazioni. La Figura 5.4 riporta semplicemente un esempio rappresentativo di una int_map, tale immagine non è stata ricavata all'interno del mio progetto. Con la int_map viene computata la distance_map: tutte le celle libere esplorano ricorsivamente le celle adiacenti alla ricerca dell'ostacolo più vicino che, quando viene trovato, diviene il parent della cella libera di partenza e la cui distanza viene salvata in distance. Ottenuto il risultato finale siamo in possesso di una mappa iterabile per ogni sua cella di cui conosciamo tutte le regioni di traversabilità al raggiungimento del goal e le distanze per impedire al robot di scontrarsi con ostacoli.

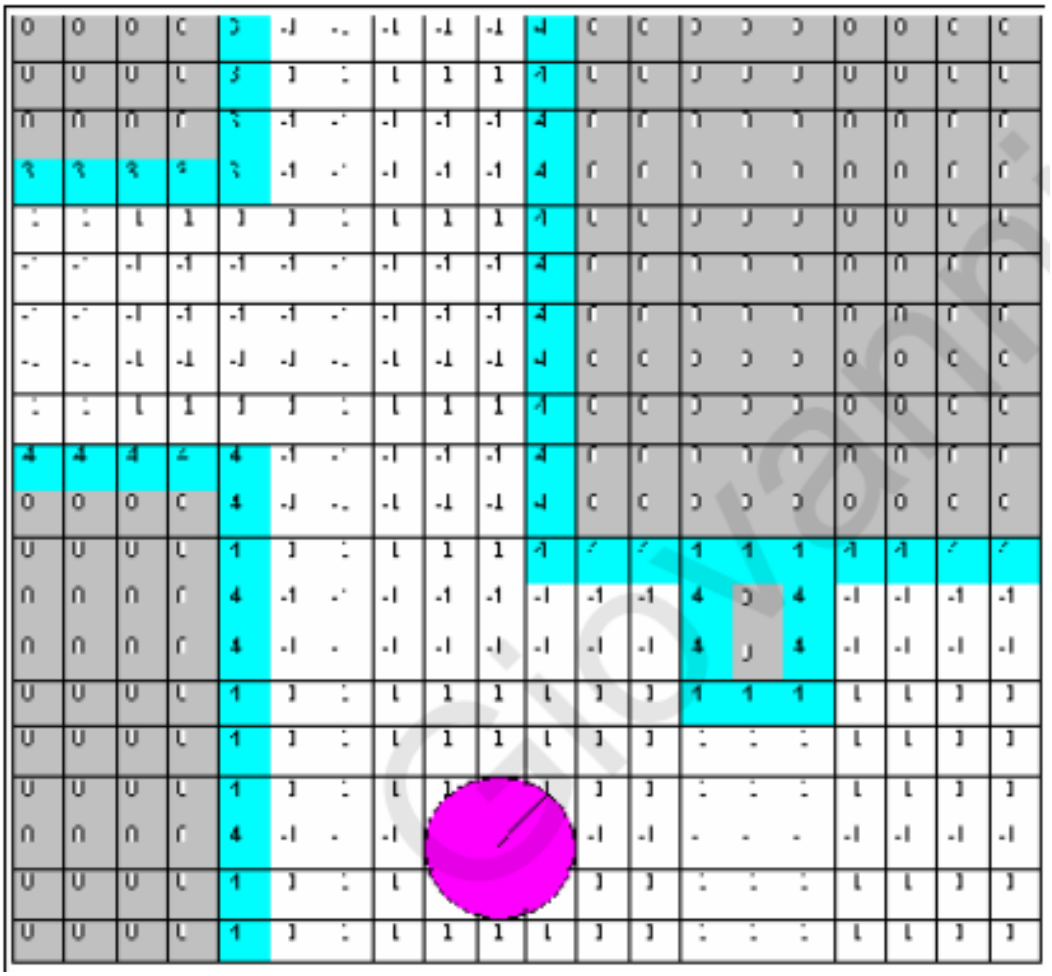


Figura 5.4. Esempio di int_map

5.2.3 Path planning

La fase di path planning, insieme col successivo controllore, compongono la parte conclusiva del progetto ed il suo cuore. Qui viene programmato il path, il percorso da seguire per raggiungere il goal, formalmente definito come la sequenza di posti da visitare per andare da un punto A a un punto B. La distance_map è convertita in un grafo non orientato in cui ogni nodo rappresenta una cella della mappa: tale nodo è collegato ai nodi rappresentanti le celle adiacenti alla cella che rappresenta se, e solamente se, sono celle libere da ostacoli (pertanto ogni nodo avrà al massimo 8 archi). A questo punto, ricavato il grafo, basterà applicargli un algoritmo di ricerca del percorso minimo, come Dijkstra o l'equivalente A*, tra il nodo con coordinate la posizione di origine del robot e il nodo con coordinate il goal impostato. Trovato il percorso risultante da questa fase e pubblicato nel suo topic, esso sarà ora visualizzabile su rviz; in questo caso era già osservabile in Figura 5.2, dove il percorso minimo trovato è descritto dalla linea verde. Il path continua ad essere continuamente ricalcolato mentre il robot si avvicina all'obiettivo, per assicurarne la correttezza.

5.2.4 Controllore

Infine, necessitiamo di un'ultima parte che comunichi al robot i movimenti e le rotazioni da effettuare per garantire che le velocità delle ruote siano coerenti con quelle necessarie per seguire il path calcolato. È stato pertanto realizzato in parte un controllore PID (Proporzionale-Integrale-Derivativo) nel quale l'errore è indicato dalla vicinanza del robot ad un ostacolo (Figura 5.5); nel codice il controllore è realizzato soltanto in parte poiché per il controllo della semplice base mobile del progetto non necessitiamo di un controllore estesissimo e completo. Il controllore imposta banalmente una velocità d'andatura per il robot e se, sempre studiando la mappa, avverte che nelle vicinanze ci sono ostacoli molto vicini rallenta il robot per fornirgli più tempo di accertarsi che il percorso che sta seguendo sia sicuro. Allo stesso identico modo, se il controllore vede che gli ostacoli più vicini sono lontani, comanderà al robot di accelerare.

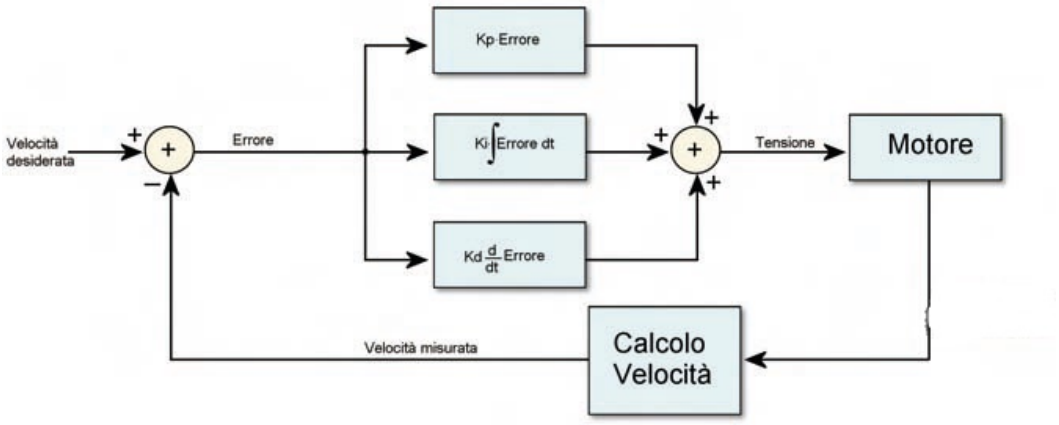


Figura 5.5. Azione completa di un controllore PID

5.3 Risultati

Nelle seguenti tre figure sottostanti (Figure 5.6, 5.7 e 5.8) viene mostrato il robot mentre autonomamente si avvicina all'obiettivo: dall'ultima figura si può vedere che il robot non riesce a posizionarsi esattamente sul goal, nonostante ci arrivi molto vicino; ciò è banalmente dovuto al fatto che il progetto non è completamente perfetto e infallibile, ma tuttavia riesce comunque a raggiungere un risultato più che soddisfacente. Come detto nella scorsa sezione, il progetto termina qui, senza la necessità di sviluppare un vero e proprio robusto controllore: ciò è dovuto principalmente al fatto che il mondo è statico, conseguentemente la mappa lo rispecchia perfettamente e, teoricamente, non contiene alcun errore di stima. Un controllore si rende infatti estremamente necessario soltanto in mappe variabili in cui, ad esempio, esistono ostacoli dinamici come oggetti o persone in movimento o porte che possono essere rispettivamente chiuse o aperte. Tali elementi non sono stati compresi nel progetto col fine di semplificarlo: in quel caso altrimenti sarebbe necessario implementare il controllore con dei moduli aggiuntivi di obstacle detection ed avoidance.

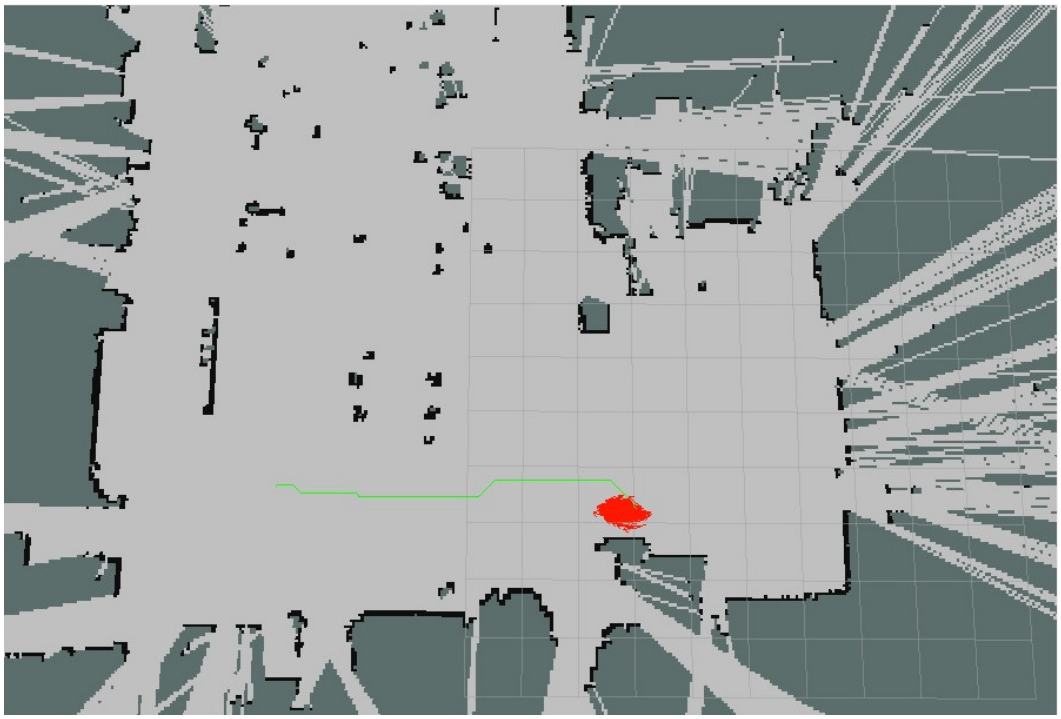


Figura 5.6

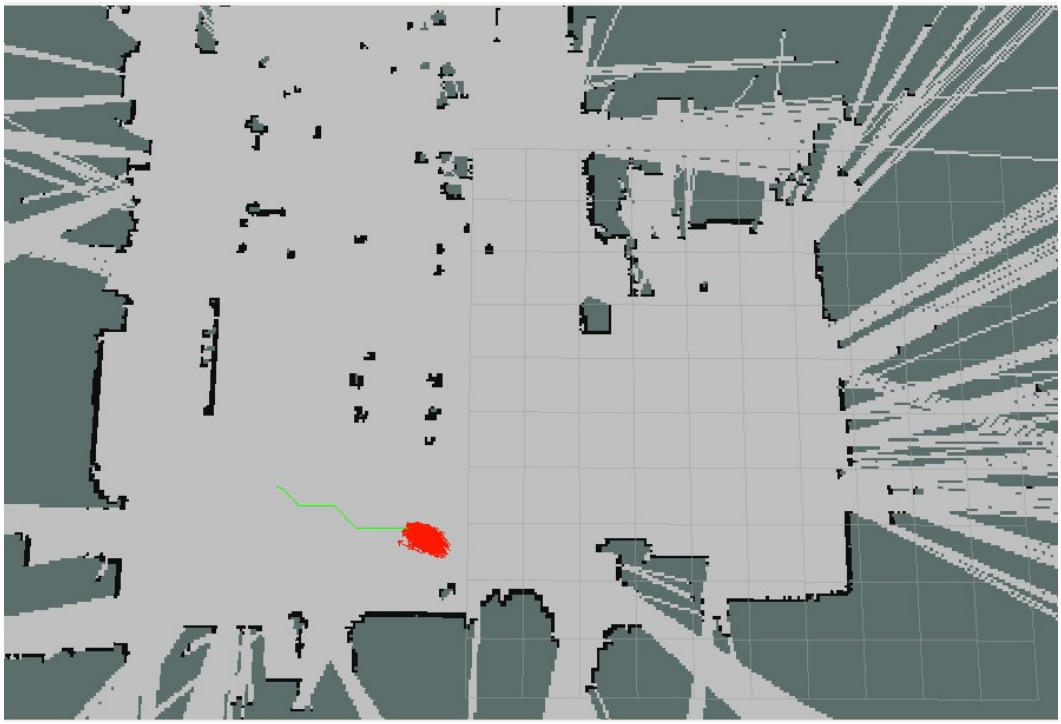


Figura 5.7

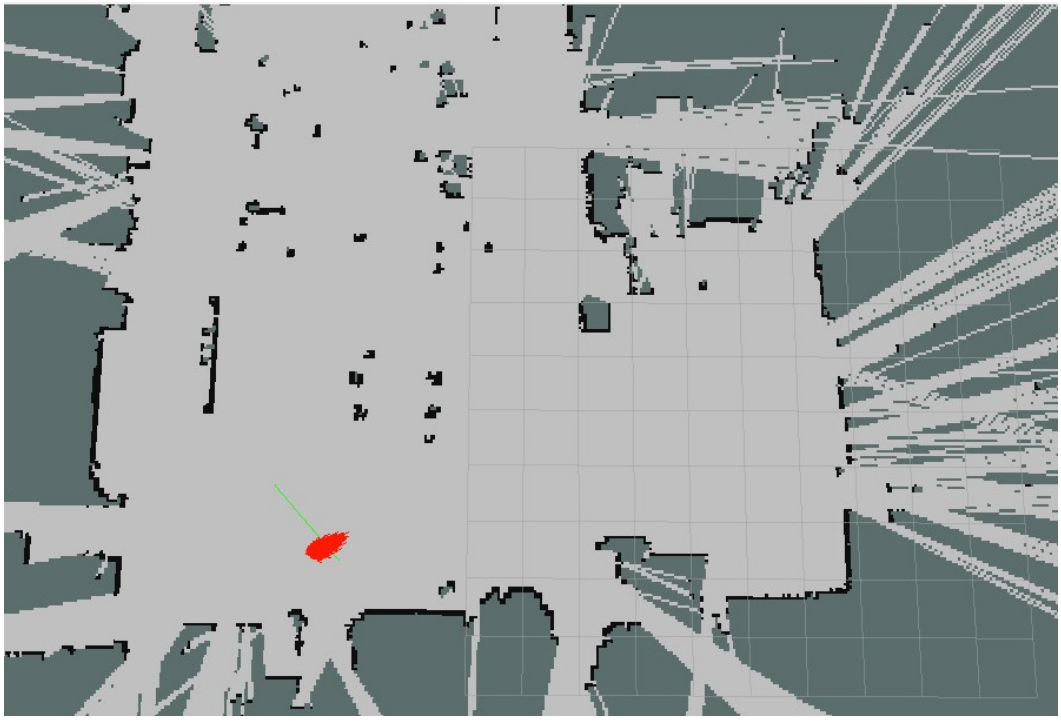


Figura 5.8

Capitolo 6

Conclusioni

Concludiamo infine la relazione, dopo aver discusso di robot intelligenti e i loro maggiori software, di ROS, delle sue caratteristiche e dei suoi indispensabili pacchetti, e di tecniche di mappatura, di localizzazione e di pianificazione. Come affermato nella sezione "Risultati", questo progetto si pone unicamente come base ad un percorso potenzialmente ampliabile infinitamente in futuro: ostacoli dinamici e un ambiente più difficilmente navigabile sono solo la punta dell'iceberg di quanto il problema di navigazione di robot sia complicabile. Inoltre, provando a riflettere ed ad aggiungere mentalmente compiti che portino il robot ad assomigliare sempre più ad un umano, pensiamo ad esempio a compiti di manipolazione, di percezione o che comportino un maggior decision making, ci accorgeremmo di quanto sia sconfinato questo ambito di ricerca, quanto ancora ci sia da scoprire e quanto l'essere umano sia esso stesso una macchina quasi-perfetta.

Capitolo 7

References

- Actionlib (2018) Accessibile su: http://wiki.ros.org/actionlib
- Agenti intelligenti. (no data) Accessibile su: https://www.intelligenzaartificiale.it/agenti-intelligenti/
- Concepts (2014) Accessibile su: http://wiki.ros.org/ROS/Concepts
- Gmapping (2019) Accessibile su: http://wiki.ros.org/gmapping
- Grisetti Giorgio, Guadagnino Tiziano & Nardi Daniele (2020) Slides del corso "Laboratorio di Intelligenza Artificiale e Grafica Interattiva 2020". Accessibili su: https://elearning.uniroma1.it/course/view.php?id=8491
- MATLAB (2018) Understanding PID Control, Part 1: What is PID Control? Accessibile su: https://www.youtube.com/watch?v=wkfEZmsQqiA
- Messages (2016) Accessibile su: http://wiki.ros.org/Messages
- Minini Andrea (no data) L'algoritmo di Dijkstra. Accessibile su: http://www.andreaminini.com/informatica/teoria-dei-grafi/algoritmo-di-dijkstra
- Nodes (2018) Accessibile su: http://wiki.ros.org/Nodes
- Rosbag (2020) Accessibile su: http://wiki.ros.org/rosbag
- Roscore (2019) Accessibile su: http://wiki.ros.org/roscore

- rviz (2018) Accessibile su: http://wiki.ros.org/rviz

- Sakemoto Mikio (2017) Build an Autonomous Mobile Robot with the Intel® RealSense™ Camera, ROS*, and SAWR. Accessibile su: https://software.intel.com/content/www/us/en/develop/articles/build-an-autonomous-mobile-robot-with-the-intel-realsense-camera-ros-and-sawr.html

- Services (2019) Accessibile su: http://wiki.ros.org/Services

- Svensson Andreas (2013) Particle Filter Explained without Equations. Accessibile su: https://www.youtube.com/watch?v=aUkBa1zMKv4

- tf (2019) Accessibile su: http://wiki.ros.org/tf

- Topics (2019) Accessibile su: http://wiki.ros.org/Topics

- Tools (2020) Accessibile su: http://wiki.ros.org/Tools