



UNIVERSITA' DEGLI STUDI ROMA TRE

Dipartimento di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

# Sperimentazione di robotica mobile in ambiente ROS

Laureando

**Flavio Lorenzi**

Matricola 487836

Relatore

**Prof.re Giovanni Ulivi**

Anno Accademico 2017-2018

Alla mia famiglia.

---

## **INDICE**

### **I**

## **PREMESSA**

### **II**

#### **1 – Turtlebot, il robot “maggiordomo”**

- 1.1 Cenni di robotica mobile
- 1.2 Il Turtlebot
- 1.3 Descrizione delle componenti
- 1.4 Considerazioni finali

#### **2 – Ambiente di lavoro: ROS**

- 2.1 ROS come piattaforma software per la robotica
  - 2.2 Architettura di sistema
  - 2.3 Perché utilizzare ROS?
  - 2.4 Descrizione delle esecuzioni
    - 2.4.1 rosnode
    - 2.4.2 rostopic
    - 2.4.3 rqt\_graph
  - 2.5 Possibile connessione con Matlab e risultati sperimentali
-

### **3 – Funzionalità di base del Turtlebot**

#### 3.1 Il Turtlebot stack

##### 3.1.1 turtlebot\_gazebo

##### 3.1.2 turtlebot\_teleop

##### 3.1.3 turtlebot\_bringup

##### 3.1.4 turtlebot\_navigation

#### 3.2 Risultati sperimentali

##### 3.2.1 find\_recharge\_dock.py

##### 3.2.2 go\_back\_bringup.py

### **4 - Trasduttori on/off board**

#### 4.1 Astra Orbbec : pregi e difetti

##### 4.1.1 il problema del tavolo

#### 4.2 Stima della posizione

##### 4.2.1 risultati sperimentali

##### 4.2.2 precisione odometrica e *motion capture*

#### 4.3 Arduino come base per introdurre nuovi sensori

##### 4.2.1 ROSserial

##### 4.2.2 risultati sperimentali

## **5 - Conclusioni e sviluppi futuri**

### **Sitografia**

### **Codice sorgente**

## I Introduzione

Negli ultimi anni la parola *robot* si è insediata sempre di più nella nostra società entrando a far parte del comune linguaggio. Tutti sanno cosa significa, ma nessuno riesce a dare una definizione specifica.

Infatti questa ha origine dalla parola ceca *robota* che può tradursi come lavoro pesante e talvolta come schiavitù; in effetti la nascita di nuove tecnologie robotiche è dovuta principalmente alla necessità di impiegarle in un lavoro che l'essere umano non riesce e/o non vuole svolgere.

Il termine *robotica* è utilizzato per descrivere quella parte di Automazione fondata sullo studio dei robot: venne usato per la prima volta (su carta stampata) nel racconto di Isaac Asimov intitolato *Liar!* (1941). In esso, egli discusse di tre regole, che in seguito divennero le Tre Leggi Della Robotica:

1. *Un robot non può recar danno a un essere umano né può permettere che, a causa del proprio mancato intervento, un essere umano riceva danno.*
2. *Un robot deve obbedire agli ordini impartiti dagli esseri umani, purché tali ordini non contravvengano alla Prima Legge.*
3. *Un robot deve proteggere la propria esistenza, purché questa autodifesa non contrasti con la Prima o con la Seconda Legge.*

Essenzialmente un *robot* è un'apparecchiatura artificiale che compie determinate azioni in base ai comandi che gli vengono dati e alle sue funzioni, sia in base ad una supervisione diretta dell'uomo, sia autonomamente basandosi su linee guida generali, magari usando processi di intelligenza artificiale; questi compiti tipicamente dovrebbero essere eseguiti al fine di sostituire o coadiuvare l'uomo, come ad esempio nella fabbricazione, costruzione, manipolazione di materiali pesanti e pericolosi ma anche in altri settori come in campo medico, o in ambienti proibitivi o non compatibili con la condizione umana.

---

In questo lavoro di tesi si approfondiscono alcuni applicativi della robotica in campo di ricerca, introducendo una piattaforma di sviluppo software che è la base su cui è costruita l'intera robotica accademica: ROS (Robotic Operating System). Verrà mostrato il suo funzionamento nel dettaglio, in particolare quali elementi fondamentali lo rendono così importante nello sviluppo di applicativi e di come essi poi vengano eseguiti su un vero robot mobile, il Turtlebot.

L'obiettivo di questo lavoro è di presentare la sperimentazione della piattaforma ROS e delle sue componenti dedicate al robot in questione.

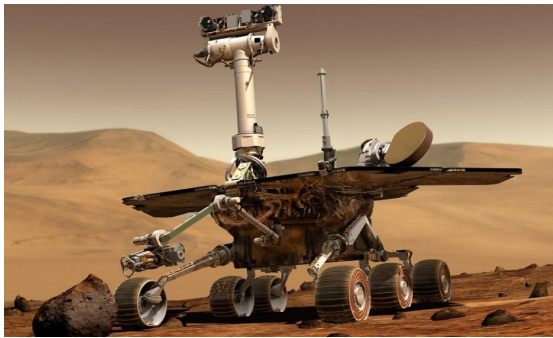
Il lavoro di tesi si articola in cinque capitoli di cui se ne fornisce di seguito una rapida descrizione:

- Capitolo 1: vengono descritte le principali caratteristiche del robot mobile utilizzato e se ne presentano le funzionalità.
- Capitolo 2: si presentano le funzionalità più rilevanti di ROS, elencando le varie componenti utili a questo studio.
- Capitolo 3: si descrivono dettagliatamente quali applicazioni della libreria Turtlebot di ROS sono state utilizzate e si espongono le applicazioni elaborate.
- Capitolo 4: ci si concentra sui sensori presenti a bordo e quelli non presenti, in modo tale da pensare a possibili sviluppi e soluzioni per colmare qualche gap di sistema.

## Capitolo 1

### **Turtlebot, il robot “maggiordomo”**

#### **1.1 Cenni di robotica mobile**



**Fig.1 : Opportunity, il secondo robot mobile utilizzato dalla Nasa su Marte**

Un robot mobile è una macchina automatica in grado di muoversi nell’ambiente che la circonda. Le applicazioni e le aree di ricerca dei robot mobili sono ormai moltissime, tra queste citiamo la sorveglianza di grandi aree, irrorazione di terreni coltivati o pulitura di ampie superfici.

I sensori sono un elemento di primaria importanza nella robotica mobile. Le tipologie di sensori installabili sono esteroceettivi, usati per misurare distanze dagli ostacoli (scanner laser), e propriocettivi, usati per misurare lo spazio percorso. Tipicamente questi sensori sono utilizzati in un processo simultaneo di mappatura e localizzazione (SLAM) per la creazione di una mappa dell’ambiente circostante. Una mappa non è altro che una struttura di dati in cui l’informazione spaziale è ricavata dalle misure sensoriali oppure fornita a priori. Per motivi computazionali l’ambiente circostante è discretizzato come una griglia composta da celle vuote od occupate da un ostacolo.

---



Le principali funzionalità richieste ad un robot mobile sono quelle di ricerca del percorso migliore da un punto di start ad uno di goal (Path planning) oppure di percorrere un percorso già pianificato nella maniera più precisa possibile (Trajectory tracking). Analizzando la prima tipologia di funzione si introduce il concetto di Obstacle Avoidance, ossia la capacità di un dispositivo di interfacciarsi con ostacoli dinamici o statici presenti sul cammino reputato ottimo. Infatti nella maggior parte delle soluzioni di robotica mobile per rendere il sistema di navigazione più robusto è presente un modulo di Obstacle Avoidance che necessita un'implementazione efficiente ed al tempo stesso molto reattiva agli stimoli percepiti dall' esterno.

## 1.2 Il Turtlebot



**Fig.2 Robot Mobile Turtlebot**

Un Turtlebot non è altro che un robot mobile composto da una base motorizzata Kobuki di fabbricazione Yujin Robotics connessa ad un Notebook interfacciato ad un sensore 3D Astra della Orbbec.

La base ricorda quella utilizzata dalla iRobot per le aspirapolveri Roomba e per questo motivo il Turtlebot ha l'appellativo di robot maggiordomo.

### 1.3 Descrizione delle componenti



Fig.3a Base Mobile Kobuki di Yujin Robotics

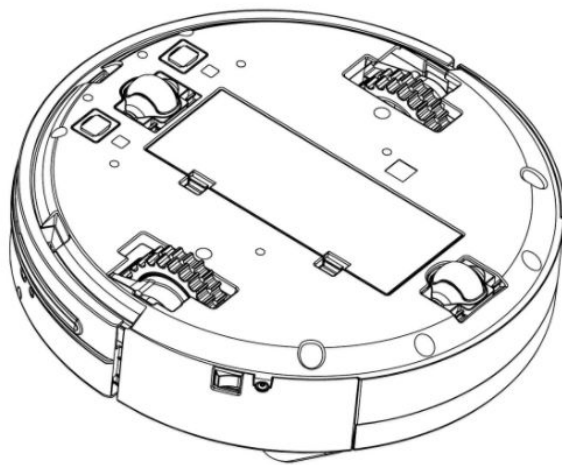


Fig.3b Vista dal basso di Kobuki

• **La base mobile Kobuki:** è il dispositivo principale che caratterizza un Turtlebot, essa è dotata di due motori elettrici Brushed controllati in corrente continua i quali permettono il raggiungimento di una velocità massima di 0.7 m/s ed una rotazionale di 180 gradi/s. La struttura cinematica è composta da due ruote differenziali su cui è montato un encoder che permette un guadagno di precisione per il controllo e due ruote sferiche di supporto.

- **Il giroscopio digitale ad un asse (angular rate gyro):** un sensore propriocettivo di fondamentale utilità per la stima della posizione del robot e del suo angolo di rotazione rispetto ad uno di riferimento. Sul Kobuki utilizzato il chip gyro è il ADXRS613 di fabbricazione Analog Devices.



**Fig.4 Sensore RGB-D Astra di Orbbec**

- **Sensore 3D Orbbec Astra:** è una potente telecamera 3D compatibile con ROS e le applicazioni robotiche, ha una portata che va dai 0.6 metri agli 8 metri, il campo visivo orizzontale è di 60° mentre quello verticale è di 49.5°. È equipaggiata con un sensore RGB ed uno infrarossi che rende questo dispositivo ideale negli scenari di ampia portata.

- **Paraurti e sensori di precipizio:** sono di fondamentale importanza durante la navigazione: i 3 bumpers frontali forniscono un feedback qualora venga urtato un oggetto ed il cliff sensor permette di salvaguardare il robot da precipitare dalle altezze come scale e simili.



**Fig.5a** funzionamento dei sensori di precipizio

- **Altri sensori:** la base Kobuki è inoltre dotata di tre sensori infrarossi posizionati nella parte frontale utili alla ricerca autonoma della base di ricarica; le ruote sono equipaggiate di un sensore di caduta delle ruote che le permette di essere sensibili ad un dislivello del terreno o alla perdita di contatto con la superficie.
- **Intel Nuc:** è il mini PC dotato di microprocessore Intel Core i7 che gestisce ed elabora gli input e gli output. In esso risiede il sistema operativo Linux Ubuntu e ROS Kinetic.



**Fig.5b** Mini Pc Intel Nuc i7

---

• **Power Bank Litionite Tanker:** si occupa di alimentare il notebook Nuc quando la base Kobuki è in operatività in quanto quest' ultima non permette di alimentare una fonte esterna come il PC. Si ricarica con alimentatore standard da presa a muro.

• **Pannello di controllo:** nella parte posteriore della base vi è un pannello di controllo dotato dei seguenti componenti:

- connettore di ricarica da 19V/2A per la ricarica del laptop (entra in funzione solo quando Kobuki è in ricarica)
- connettore di alimentazione 12V/5A per braccio manipolatore
- presa di ricarica da 12V/1.5 A per sensore 3D
- presa da 5V/1.5A generica
- presa USB di connessione fra PC e base
- Led di status indicante il livello di carica del Kobuki: verde quando in funzione e carica, giallo se la batteria è in esaurimento, verde lampeggiante se in ricarica.
- Led 1/2 e pulsanti 0/1/2 programmabili
- Pin di switch per abilitare o disabilitare la modalità di aggiornamento firmware



Fig.5c Pannello di controllo posteriore



**Fig.5d dock station di ricarica**

- **Batteria:** batteria a litio da 2200 mAh inserita nel vano sul fondo del Kobuki. Permette una autonomia media di 3 ore ed un tempo di ricarica di 1.5 ore. La ricarica della batteria può avvenire in due modi: mediante l'alimentatore di serie collegato alla presa a muro oppure utilizzando una dock di ricarica fornita nel kit.



**Fig.5e posizione della batteria**

I passaggi iniziali da seguire per una corretta configurazione del setup sono quelli riportati nella Kobuki User Guide e sono i seguenti: collegare via USB il mini PC al Kobuki e alla videocamera Astra e connettere il powerbank alla presa di alimentazione del PC. A questo punto tenere premuto il pulsante di accensione del

---

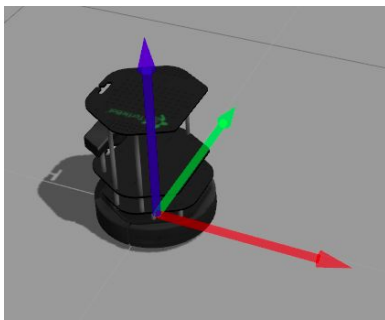
powerbank per accenderlo e premere fino a visualizzare 20V sul led (ciò indica la tensione in uscita). Accendere il Nuc e la base Kobuki. A questo punto il setup è configurato correttamente.

( <http://kobuki.yujinrobot.com/documentation/> )

I manuali utilizzati in questo studio di tesi sono stati i seguenti:

- Mastering ROS for Robotics Programming, Lentin Joseph, 2015
- Ros Robotics by Example, Fairchild, Harman, 2016
- ROS Wiki ( <http://wiki.ros.org/Robots/TurtleBot> )

#### 1.4 Considerazioni finali



**Fig.6 Assi di riferimento del robot**

Il robot è caratterizzato da tre assi di riferimento xyz: la x indica movimento in avanti, la y il movimento verso sinistra (regola della mano destra) e z indica la quota.

Inoltre come visto ha solo due ruote motrici e due di appoggio (fisse), infatti si può parlare di configurazione robotica a unicycle. Una cosa molto utile per il Dead Reckoning (stima della navigazione) sono i vincoli anolonomi (vincolo di puro rotolamento senza slittamento); tenendo solo conto del coefficiente di rotolamento  $\psi$  delle ruote basterebbe infatti rendere la velocità normale pari a zero mentre quella tangenziale pari a  $raggio * \psi$  per ottenere una stima accettabile. Sappiamo però che è impossibile ottenere una precisione esatta dagli encoder delle ruote: esistono infatti errori sistematici (per esempio : il diametro delle ruote è diverso da una ruota

all'altra; i dispositivi di rilevamento hanno una risoluzione e una frequenza di campionamento finite; ruote disallineate...) e non sistematici (percorso su pavimenti non perfettamente piani; passaggio sopra oggetti inaspettati; scivolamento delle ruote...); quindi in seguito vedremo come cercare di arginare tali gap di sistema, magari introducendo nuovi sensori alla piattaforma oppure consultando le informazioni che derivano dalla videocamera.

## Capitolo 2



### **Ambiente di lavoro : ROS**

#### **2.1 ROS come piattaforma software per la robotica**

Robot Operating System è un insieme di librerie software e strumenti utili nell'implementazione di applicazioni nel campo della robotica tra cui quelle di simulazione, navigazione, SLAM (Simultaneous Localization and Mapping) e percezione sensoriale. Fornisce le stesse funzioni standard di un sistema operativo tra cui: astrazione dell'hardware, controllo dei dispositivi tramite driver, comunicazione tra processi, gestione dei processi, dei pacchetti e delle loro



dipendenze. Nonostante presenti alcune similitudini con un sistema operativo convenzionale, ROS non va considerato come tale per il semplice fatto che il suo funzionamento dipende da Ubuntu.

Ciò che contraddistingue ROS è l'essere un set di strumenti open source aperti ad una comunità di sviluppatori che possono modificare o creare nuovi progetti usufruibili da chiunque in rete; questo sistema si è ampliato a tal punto da contare migliaia di moduli utili nella robotica educativa, per lo sviluppo di prodotti commerciali e per la robotica industriale.

Prima di approfondire le sezioni applicative di ROS è doveroso introdurre l'architettura generale del framework.

## **2.2 Architettura di sistema**

Strutturalmente ROS si sviluppa su 3 livelli concettuali:

Filesystem Level, Computational Level e Community Level, i cui ruoli ed elementi costitutivi verranno ora analizzati.

### **2.2.1 Filesystem**

Il ROS Filesystem comprende le risorse fondamentali presenti sul disco quali:

**Packages:** i Packages sono l'unità atomica dell'organizzazione del software in ROS. Un pacchetto è rappresentato dal programma scritto dall'utente ed ha un suo scopo. Esso può contenere un nodo ROS, un dataset, file di configurazione ed una o più librerie. Un package è quindi l'elemento base che si può progettare e poi pubblicare nella community.

**Metapackages:** i Metapackage sono delle particolari forme di Package principalmente utilizzati per rappresentare un gruppo di altri package legati tra loro.

**Package Manifest:** il Manifest è un documento XML (eXtensible Markup Language) che contiene informazioni riguardanti il package tra cui il nome, versione, breve descrizione e altre informazioni necessarie alla manutenzione (email dello sviluppatore).

---

**Repositories:** una Repository (o Stack) è una collezione di pacchetti ROS che condividono la stessa etichetta di versione ed hanno uno scopo in comune, detto task. I pacchetti di una stessa repository possono essere rilasciati insieme utilizzando il “catkin release automation tool”.

**Message (msg) types:** indica il tipo di dato e la struttura del messaggio scambiato fra due nodi in ROS.

**Service (srv) types:** indica il tipo di dato e la struttura del messaggio scambiato tra due servizi in ROS. Essi seguono un modello request/response e per questo motivo il service data type è composto da due macro sezioni, una dedicata alla request ed una alla response.

### 2.2.2 Computation Level

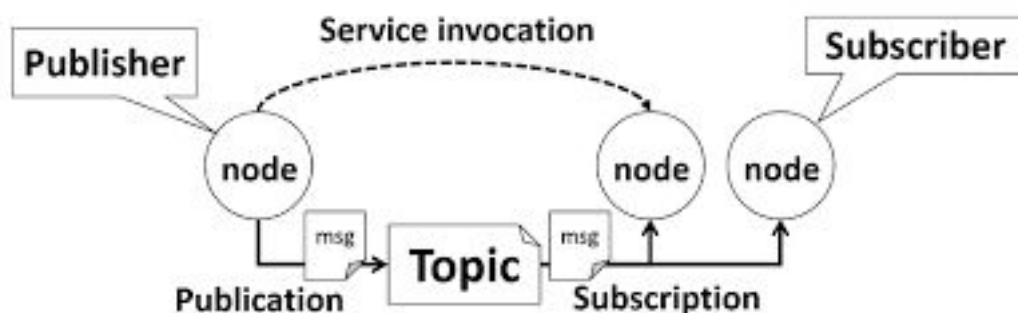


Fig.7 esempio di rete distribuita ROS

Il Computation Graph Level è la rete peer-to-peer dei processi ROS tra loro comunicanti. I principali elementi di questo livello sono i nodi, il nodo Master, il Server Parametrico, messaggi, servizi, topic, e che trattano la distribuzione di dati nella rete in diversi modi.

---

**Nodi:** In ROS un nodo è un processo che compie una qualsiasi attività di interesse. Tipicamente un sistema è composto da molteplici nodi di cui, per esempio, uno è il nodo che controlla un laser scan o i motori delle ruote, un altro si occupa della localizzazione del robot ed uno ancora che si occupa della rappresentazione grafica del sistema e così via.

**Master:** Il ROS Master si occupa registrazione dei nodi e della gestione della rete intera, senza di esso gli altri nodi presenti nel network non potrebbero comunicare tra loro. In ROS il comando base per la creazione di un master è

```
$ roscore
```

È fondamentale generare questo nodo prima di avviare ogni software perché, come sarà esposto in seguito, l'intero network di ROS necessita la presenza di un Master. Il Server parametrico è una parte del master e si occupa della memorizzazione centralizzata dei dati.

**Messages:** due o più nodi comunicano attraverso lo scambio di messaggi, la loro struttura è fortemente tipizzata e sono supportati tutti gli standard primitivi come integer, floating point, boolean così come gli array di costanti o primitive. La struttura del messaggio è definita nel file .msg all'interno di ogni package. Un rosmmsg ha una struttura bipartita in fields (campi), dati contenuti nel messaggio, e constants (costanti numeriche) utilizzate per l'interpretazione del contenuto del messaggio.

La parte del field è composta a sua volta da tipo di dato (int32, bool, string) e nome della variabile.

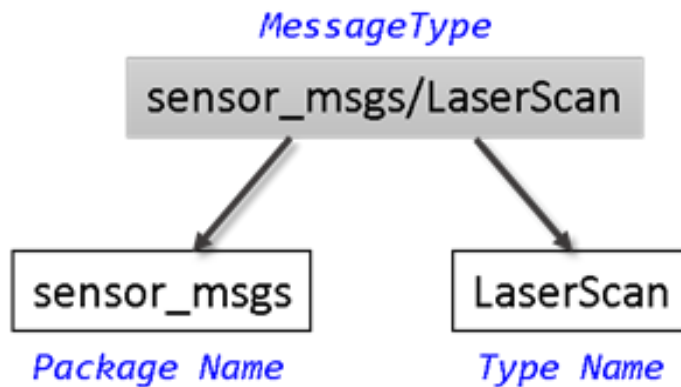


Fig.8a esempio di messaggio in ROS

Per esempio i messaggi utilizzati per azionare i motori del robot sono i messaggi tipati `geometry_msgs/Twist`. Utilizzando il comando `rosmmsg` da riga di comando si può analizzarne la struttura:

```
$ rosmmsg show geometry_msgs/Twist
```

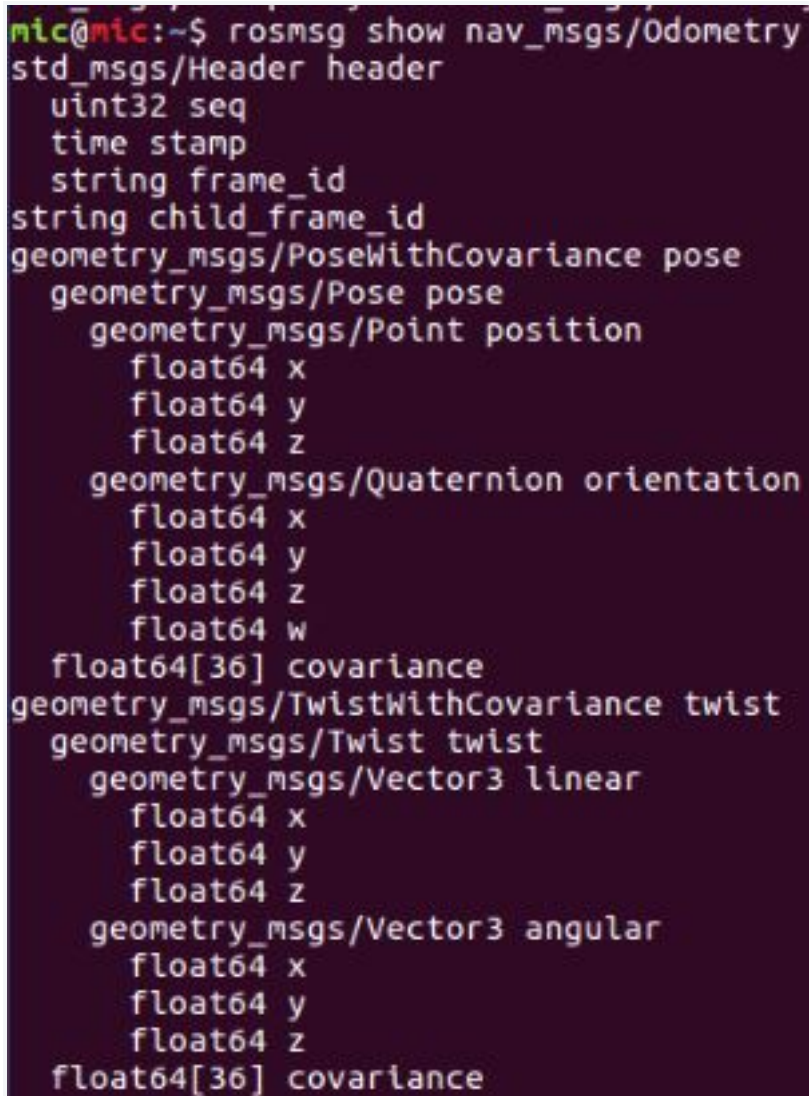
```
mic@mic:~$ rosmmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Fig.8b composizione dei messaggi `geometry_msgs/Twist`

Si evidenzia infatti come la struttura dei messaggi utilizzati per muovere il robot sono composti da due Field, `linear` e `angular`, composti a loro volta da 3 costanti `[x,y,z]`.

Una successiva analisi può essere quella relativa ai messaggi che trasportano informazioni riguardanti l'Odometria del robot.

```
$ rosmmsg show nav_msgs/Odometry
```



```
mic@mic:~$ rosmmsg show nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
      float64[36] covariance
  geometry_msgs/TwistWithCovariance twist
    geometry_msgs/Twist twist
      geometry_msgs/Vector3 linear
        float64 x
        float64 y
        float64 z
      geometry_msgs/Vector3 angular
        float64 x
        float64 y
        float64 z
      float64[36] covariance
```

Fig.8c composizione dei messaggi nav\_msgs/Odometry

Il messaggio odometrico contiene informazioni riguardanti posizione e velocità di un oggetto nello spazio. È composto da quattro campi principali:

- Header, il frame che si occupa di trasmettere le informazioni riguardanti la posizione Pose. Esso è composto dalle costanti *stamp*, che indica i secondi passati dalla creazione del frame, *seq* e *frame\_id* che lo identificano nella rete.
- Child\_frame\_id, il frame che comunica i dati di Twist identificato univocamente da un *frame\_id*.
- Pose, i dati tipati *geometry\_msgs/Pose* composti dai campi Point Position [x,y,z] e Quaternion Orientation [x,y,z] che identificano rispettivamente posizione ed orientamento di un oggetto nello spazio.
- Twist, come visto in precedenza, mostrano informazioni di velocità lineare ed angolare dell' oggetto.

**Topics:** il sistema di trasporto dei messaggi in ROS è di tipo asincrono unidirezionale e segue una semantica del tipo publish / subscribe: un nodo che vuole spedire un messaggio deve necessariamente pubblicare tali informazioni attraverso un topic il cui nome è usato per identificare il contenuto del messaggio. Un altro nodo che sia interessato a leggere tali messaggi deve sottoscrivere a quel topic per poterne leggere i messaggi spediti. Questa struttura consente inoltre che più nodi possano pubblicare o sottoscrivere ad uno stesso topic ed allo stesso modo un nodo può pubblicare o sottoscrivere a più topic. Diviene immediato pensare ai topic come a dei BUS (Binary Unit System) fortemente tipizzati in base al messaggio trasportato, a cui tutti possono collegarsi per spedire e ricevere informazioni. Nel sistema dei topics ogni nodo presenta un suo URI (Uniform Resource Identifier) cioè una sequenza di caratteri che identifica univocamente un nodo nella rete.

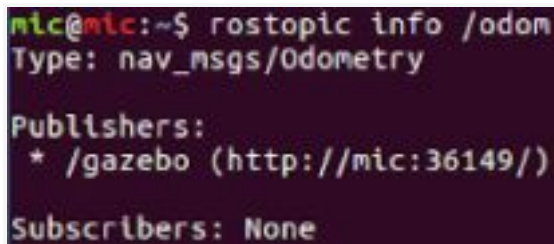
Di seguito è descritto come si stabilisce una connessione fra due nodi in una rete ROS.

---

---

Anche in questo caso prendiamo in esempio il topic su cui vengono scambiate informazioni relative l'odometria del robot: `/odom`. ROS offre vari strumenti per l'analisi dei topic, essi sono mostrati digitando `rostopic -h`. Utilizzando per esempio

```
$ rostopic info /odom
```



```
mic@mic:~$ rostopic info /odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://mic:36149/)

Subscribers: None
```

**Fig.8d** analisi del topic `/odom`

Verrà mostrato a schermo il tipo di rosmg scambiato ed il nodo che pubblica informazioni su di esso (in questo caso il nodo di nome `/gazebo`)

Affinché due nodi possano scambiarsi messaggi è necessario che il publisher si registri presso il Master inviando dettagli riguardanti il tipo di messaggio che pubblica, il nome del topic su cui intende spedire tali messaggi ed il suo URI identificativo. Allo stesso modo un subscriber si registra presso il Master inviando le sue informazioni che lo identificano. Il Master aggiorna costantemente i dati dei publisher e dei subscriber all'interno di una tabella ed informa il subscriber della presenza di un nuovo publisher. A questo punto il subscriber invia un messaggio al publisher richiedendo la connessione al topic e negoziando il tipo di protocollo che può essere TCP (Transmission Control Protocol), molto utilizzato vista la sua affidabilità, o UDP (User Datagram Protocol), utilizzato maggiormente nel broadcasting. Ora il publisher invia al subscriber la corretta configurazione in base al tipo di protocollo scelto, nel caso di TCP indirizzo IP (Internet Protocol address) e numero di porta.

Dopo che il subscriber si è connesso al publisher, ogni messaggio inviato da quest'ultimo verrà letto dal subscriber. Si noti che il Master non è coinvolto nello scambio di messaggi tra nodi.

**Services:** un'alternativa al modello publisher/subscriber è quello server/client offerto dai servizi ROS. Questo paradigma risulta più efficiente nello scambio frequente e maggiore di dati dei sistemi distribuiti. Nella rete si viene a creare un nodo service provider il cui servizio è utilizzato da un nodo client per inviare messaggi di tipo request/reply.

ROS mette a disposizione dei pratici servizi di analisi anche per i services, essi sono invocati digitando `rossrv -h`. Fra i tanti si prende in analisi il comando *show* per mostrare la composizione del servizio GetMap.

```
$ rossrv show nav_msgs/GetMap
```

Quando un nodo client richiede informazioni ad un nodo server riguardanti la composizione della mappa salvata durante un'operazione di SLAM, il messaggio che viene inviato contiene appunto tutte le informazioni che definiscono una mappa in ROS. Come si vede dalla Fig.8e una mappa è definita da due campi primari, l'Header e l'Info. Quest'ultimo a sua volta è composto da 5 metadati ognuno dei quali racchiude un'informazione specifica come il tempo di caricamento della mappa, la sua risoluzione e dimensione in pixel, la posizione iniziale in cui si trova il robot al momento del caricamento.



```
mlc@mlc:~$ rossrv show nav_msgs/GetMap
---
nav_msgs/OccupancyGrid map
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  nav_msgs/MapMetaData info
    time map_load_time
    float32 resolution
    uint32 width
    uint32 height
    geometry_msgs/Pose origin
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
    int8[] data
```

Fig.8e informazioni sul rossrv GetMap

**Bags:** i file .bag sono utilizzati per memorizzare dati di tipo rosmg, come quelli provenienti da un sensore, che sono necessari per lo sviluppo di algoritmi. Per memorizzare tutti i messaggi scambiati da un dato nodo è possibile usufruire dello strumento *record* di rosbag

```
$ rosbag record --node=/nomenodo
```

Esso memorizzerà all'interno di file .bag tutte le informazioni relative ai messaggi inviati e ricevuti da un nodo durante il periodo di osservazione.

---

### 2.2.2 Community Level



Fig.9 distribuzione ROS Kinetic

ROS è una piattaforma open source e per questo il livello Community di ROS ha la funzione di favorire lo scambio di software e conoscenza tra individui. Ciò avviene attraverso i seguenti strumenti:

**Distribuzioni:** una distribuzione è un insieme di pacchetti ROS installabile ed ha lo stesso ruolo delle distribuzioni Linux: far sì che gli sviluppatori lavorino ad una versione stabile del codice prima di avanzare a versioni successive. Si può pensare alla distribuzione ROS come ad una versione del sistema operativo stabile installabile sul proprio sistema Linux. La *rostdistro* su cui si basa questa tesi è Kinetic Kame LTS (Long Term Support) installata su Linux Ubuntu.

**Repositories:** una repository è un deposito di software e codice di appartenenza di un gruppo di ricercatori che ne detengono la proprietà e la manutenzione. ROS infatti non centralizza la presenza del codice ma lascia che ogni individuo possa dare il suo contributo alla community detenendo i diritti di applicazioni e framework. Questa metodologia di condivisione del codice ha enfatizzato e massimizzato la partecipazione della comunità internazionale al progetto, rendendo ROS tra i framework più utilizzati a livello mondiali nell'ambito dello sviluppo di applicazioni per robot di sistemi complessi.

**ROS Wiki:** Wiki ROS è il forum principale in cui reperire informazioni su ROS e chiunque può contribuire con dei contenuti o delle correzioni alle repository esistenti (<http://wiki.ros.org/>).

---

### **2.3 Perché utilizzare ROS?**

Vi sono varie motivazioni che dovrebbero spingere chiunque si approcci al mondo della robotica a considerare ROS. In primis la maggior parte delle applicazioni robotiche sfruttano il sistema operativo Linux su cui ROS è costruito e di cui ne sfrutta l'interfaccia grafica e il sistema di gestione dei processi. Di non minore importanza è la compatibilità con un'ampia gamma di software e hardware in commercio, come i controller Sony PS3 e il sensore 3D Microsoft Kinect, che rendono ROS aperto ad interagire con molti dispositivi.

La scelta delle componenti per assemblare un robot può presentare delle difficoltà se essi non sono pienamente compatibili e se lo sono, non è detto che questo basti a far funzionare a dovere l'intero sistema. In ROS ogni sensore o attuatore è identificato da un nodo che scambia il suo tipo di messaggi all'interno della rete. Il più delle volte ogni dispositivo ha un suo protocollo di trasmissione e supporta un linguaggio di programmazione diverso. In questi casi ROS rende l'intero sistema distribuito compatibile grazie all'utilizzo del topic. Un sistema il cui carico computazionale viene ripartito tra i vari nodi ha innanzitutto il vantaggio di una maggiore tolleranza agli errori, potendo gestire il crash del singolo nodo, tranne il Master la cui caduta compromette l'intera rete. Per esempio se un sensore di misurazione della distanza di un robot smette di funzionare, ROS gestisce l'interruzione mantenendo attivo il sistema.

Negli ultimi anni la condivisione di package è in forte crescita: la community alla base di ROS favorisce la creazione di applicativi stabili ed usufruibili da chiunque si avvicini al mondo della robotica di servizio senza dover implementare da capo il codice, per questo motivo ROS ha molto seguito nei campus accademici e non solo.

---

## 2.4 Descrizione delle esecuzioni

In questa sezione verrà presentato in maniera più concreta come approcciarsi ad un sistema ROS ed in particolare verrà mostrato come creare un workspace e due comandi di analisi delle reti: *roscnode info* e *rostopic echo*.

Il passo primario per l'elaborazione in ROS è la creazione di un workspace, una cartella di lavoro che conterrà tutti i package relativi al progetto. Il processo di creazione di un workspace di nome "turtlebot\_ws" inizia con il seguente comando:

```
$ mkdir ~/turtlebot_ws/src
```

In ROS la creazione di package viene affidato al sistema catkin build che genera eseguibili a partire da file grezzi. Catkin si basa su uno strumento modulari di gestione e creazione e del codice: CMake (Cross Platform Make). CMake gestisce il processo produttivo che parte dalle sorgenti e sfocia nella creazione degli artefatti, cioè i programmi compilati e pronti all'uso [1]. La costruzione di software applicativo avviene in loco, ossia nella directory in cui risiede il codice stesso evitando di mischiare codice. La sintassi di CMake è espressa in un file sempre presente in un workspace, il file CMakeLists.txt.

Un workspace contiene tipicamente tre sottocartelle (/src, /build e /devel). Il ruolo di /src è quello di contenere il codice sorgente ed i package; /build costruisce un package nel source space; /devel definisce una posizione di memoria dove vengono posizionate le risorse prima di essere installate. Con il comando

```
$ cd turtlebot_ws/src
```

ci si sposta all'interno del /src dove va avviato il comando di inizializzazione del workspace

```
$ catkin_init_workspace
```

è necessario ora ritornare alla cartella precedente digitando

```
$ cd turtlebot_ws
```

---

---

Ed infine si avvia la compilazione del workspace che completerà la costruzione dello stesso secondo il protocollo CMake sopra introdotto.

```
$ catkin make
```

L'ultimo passaggio è quello di rendere visibile il workspace all'interno del sistema ROS, ciò avviene aggiungendo il percorso del setup del workspace al file `.bashrc` (il file di configurazione Linux che viene caricato all'apertura di ogni nuova Shell).

```
$ echo "source ~/turtlebot_ws/devel/setup.bash" >> ~/.bashrc
```

```
$ source ~/.bashrc
```

### 2.4.1 rosnode

Come detto i nodi di ROS rappresentano i processi atomici che formano l'intero network ed in presenza di un Master attivo il comando per conoscere quali nodi compongono attualmente la rete è

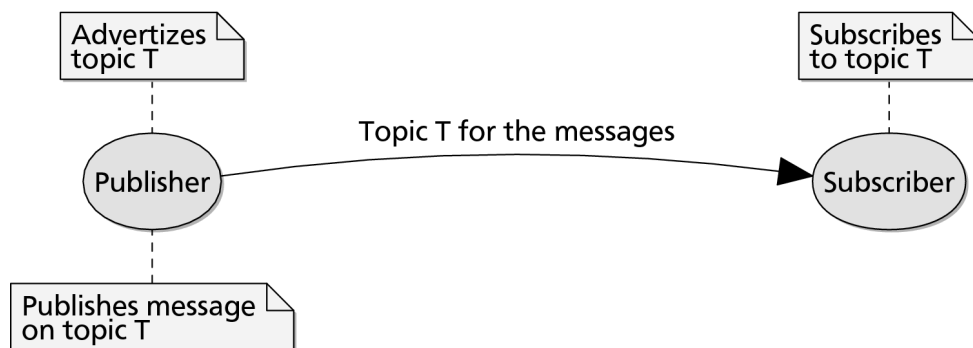
```
$ rosnode list
```

In presenza del solo Master attivo si visualizzerà il risultato `/rosout`, un nodo creato dal Master che si occupa di collezionare gli output di debugging di altri nodi. Le informazioni riguardanti i topic su cui pubblica, a cui è iscritto o i servizi offerti si leggono con il comando `roslaunch info /nomenodo`, in questo caso:

```
$ roslaunch info /rosout
```

---

### 2.4.2 rostopic



**Fig.10** scambio di informazione via topic

Così come sono presenti comandi per l'analisi dei nodi, ve ne sono alcuni per l'analisi dei topic. I comandi disponibili per l'analisi dei topic sono indicati digitando: \$ rostopic -h.

La visualizzazione della lista dei topic presenti sulla rete avviene col comando \$ rostopic list, ma è il comando \$ rostopic echo /nometopic che permette di analizzare le informazioni scambiate sul quel topic.

Prima di fare ciò è utile avviare il nodo /gazebo attraverso la chiamata di un file. launch presente nel package di turtlebot\_gazebo effettuata col comando *roslaunch*.

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

La ricostruzione di un ambiente in grafica 3D è resa possibile grazie al software Gazebo che permette una simulazione della dinamica e della raccolta dati dei sensori per prevedere ciò che potrebbe accadere nella realtà.

---



**Fig.12 Turtlebot simulato in Gazebo**

È ora possibile presentare *rostopic* per analizzare le informazioni scambiate su un topic di fondamentale importanza per questo studio di tesi: `/odom`, ossia il topic che si occupa di inviare dati odometrici del robot mobile.

```
$ rostopic echo /odom
```

```

header:
  seq: 6887
  stamp:
    secs: 68
    nsecs: 890000000
  frame_id: odom
child_frame_id: base_footprint
pose:
  pose:
    position:
      x: -9.61500260056e-07
      y: 8.16702634447e-07
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.00500403444929
      w: 0.999987479741
    covariance: [0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.05]
twist:
  twist:
    linear:
      x: -6.28405363711e-06
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: -5.43416635627e-05
    covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

```

Fig.13 Rappresentazione a schermo di rostopic echo /odom

Dalla fig.13 risulta più chiara la composizione del messaggio odometrico scambiato sul topic in esame. I valori del campo header subiscono un continuo incremento in relazione al tempo trascorso dalla creazione del topic. Durante un movimento su un piano che non presenti dislivelli le coordinate che variano in funzione della posizione del robot nella mappa di riferimento sono le position.x e position.y e quelle di orientation.x, orientation.y, e orientation.z qualora si effettui anche una rotazione. Le coordinate di Twist linear.x e angular.z variano invece in funzione della velocità lineare o rotazionale impartita ai motori.

Gli elementi che si sono dimostrati più rilevanti per questo studio sono stati *frame id* che indica il nome del topic all'interno della rete e le sezioni *pose* e *twist*: nella prima sono presenti informazioni relative alla posizione del robot nello spazio in termini di [x,y,z] e di orientamento relativo espresso attraverso una notazione matematica che rappresenta orientamento e rotazione di oggetti in un spazio 3D: il quaternione [roll, pitch, yaw, 1].



La sezione relativa al *twist* rappresenta invece il comportamento del robot in termini di spostamento lineare (velocità lineare) ed angolare (velocità angolare). Si noti che solo i valori di Linear.x e Angular.z sono gli unici a variare in un ambiente tridimensionale a quota fissa.

### 2.4.3 rqt\_graph

Lo strumento *rqt\_graph*, parte del package *rqt* utile per lo studio delle reti, si occupa di fornire un'interfaccia grafica di visualizzazione dinamica della rete ROS attiva al momento della chiamata.

Digitando:

```
$ rosrun rqt_graph rqt_graph
```

si lasci il segno di spunto solo su *debug*, *unreachable* e *dead sink* in modo tale da escludere dalla visualizzazione quei nodi che si occupano del debugging, irraggiungibili o che non hanno nodi in sottoscrizione.

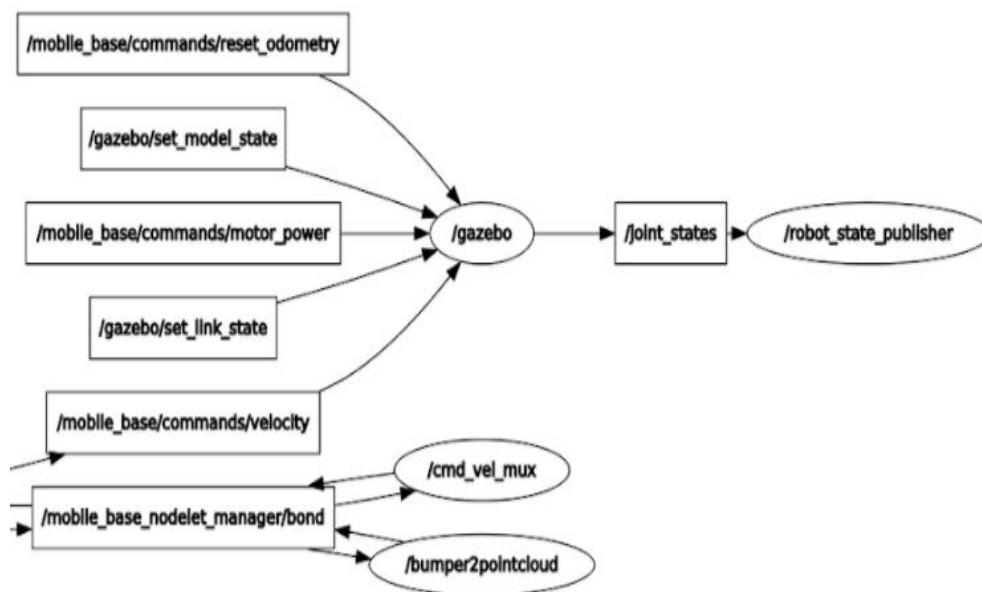


Fig.14 Rappresentazione a schermo di *rqt\_graph*

Si veda come il nodo `/gazebo` sia un nodo che pubblica informazioni su vari topic quali `/rosout` e `/clock` ed è sottoscritto ai topic `/mobile_base/commands/velocity` e `motopower` che trattano il movimento vero e proprio del Turtlebot nel simulatore.

*View\_frames* è un altro strumento utile per visualizzare il funzionamento della rete, esso fa parte del package `tf` (transform frames) che permette di tener traccia dei frames (transform frames e coordinate frames) scambiati fra due nodi. Leggendo il campo “frame identifier” del campo header di ogni `rosmmsg`, `Tf` riesce a comporre efficacemente una struttura logica del sistema in base ai frame scambiati sulla rete. Le tipologie di dati trattati sono riferimenti su puntatori, vettori, posizione e orientamento.

```
$ rosrun tf view_frames
```

## 2.5 Possibile collegamento a Matlab

Come abbiamo detto ROS è molto comodo da usare, poichè consente di collegarci ad una vasta gamma di software e trasduttori hardware. In laboratorio è stato testato infatti che è possibile connettersi con Matlab (ambiente per il calcolo numerico e l'analisi statistica) e raggiungere un controllo ad alto livello del Turtlebot, ovvero comunicare con il robot senza dover passare per i comandi base di ROS guadagnando fluidità di manovra.

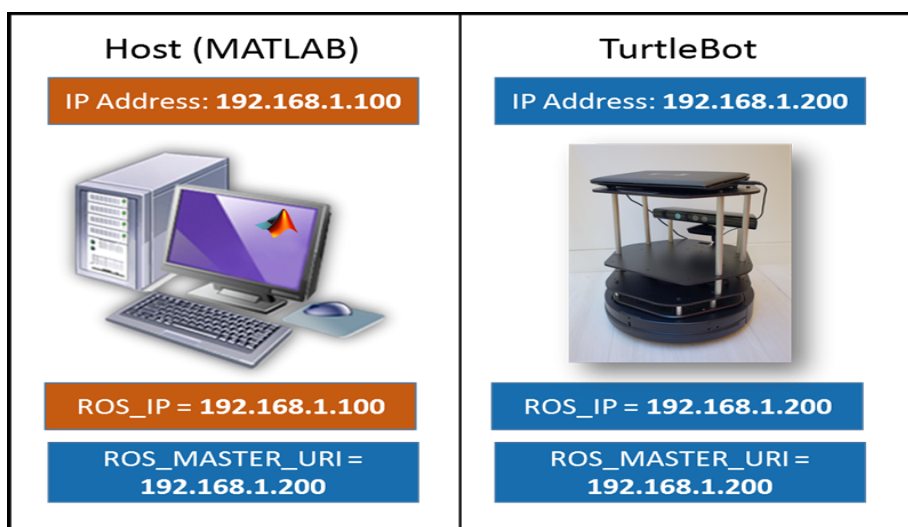


Fig.15 Setting

Impostato il `.bashrc` come descritto sopra, aprire Matlab e digitare **rosinit**: in tal modo svolgo due operazioni, inizializzo il nodo ROS e mi connetto al robot.

Ora con *rostopic list* sarà possibile vedere tutti i nodi connessi.

E' possibile controllare il movimento del TB pubblicando un messaggio sul topic */mobile\_base/commands/velocity*. Il messaggio deve essere di tipo *geometry\_msgs/Twist* e contiene dati che specificano le velocità lineari e angolari desiderate. I movimenti del TB possono essere controllati attraverso due diversi valori: la velocità lineare lungo l'asse X, che controlla il movimento in avanti e indietro e la velocità angolare attorno all'asse Z, che controlla la velocità di rotazione della base del robot.

### Risultati sperimentali

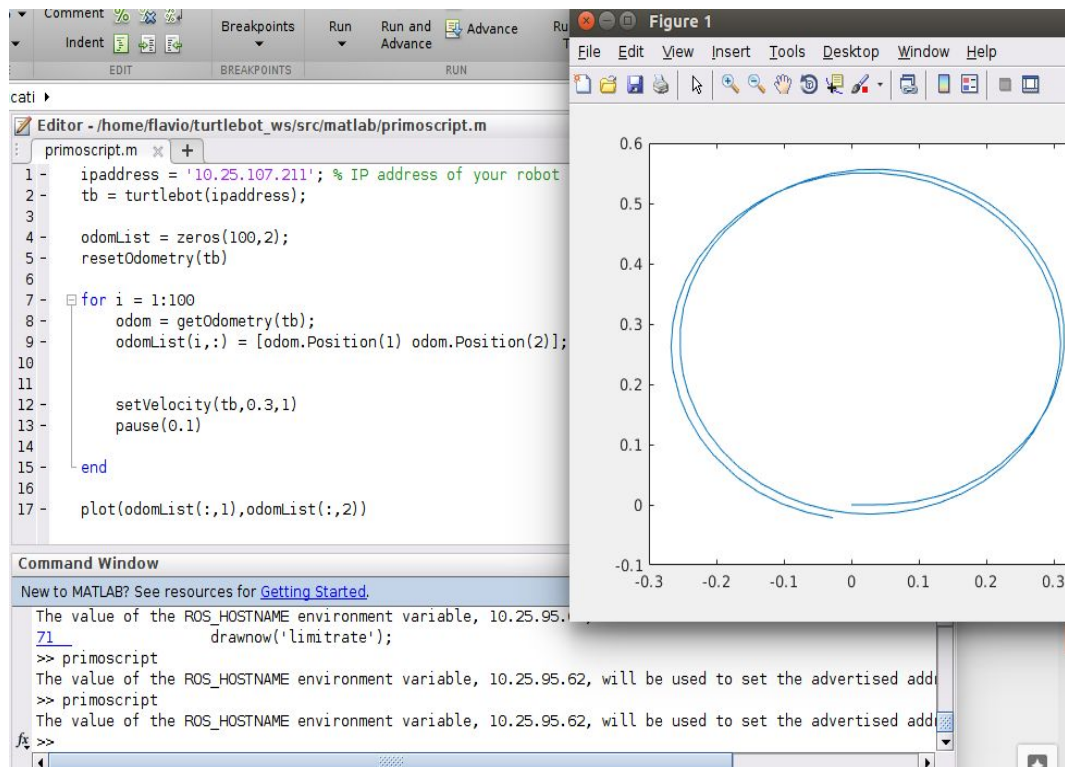
Come abbiamo detto TurtleBot utilizza l'argomento `/odom` per pubblicare la sua posizione corrente e l'orientamento. Anche in Matlab è possibile vedere questo tipo di output:

```
odom = rossubscriber('/odom') %mi sottoscrivo al topic
odomdata = receive(odom,3);
pose = odomdata.Pose.Pose;
x = pose.Position.X;
y = pose.Position.Y;
z = pose.Position.Z;
[x,y,z] %mostra le coordinate
```

**Fig.16 Codice in Matlab**

L'orientamento del TurtleBot viene memorizzato come quaternione nella variabile *pose.Orientation*. E' possibile usare la funzione **quat2eul** per convertire i quaternioni nella rappresentazione più comoda degli angoli di Eulero.

Questi dati possono essere sempre utili in fase di controllo, in quanto per un robot è di primaria importanza avere una stima ottimale della propria posizione. Dunque è stato facile, una volta in possesso di questi dati, inserirli in un semplice algoritmo “Draw A Circle” per il tracciamento della traiettoria e quindi per la gestione dell’errore.



**Fig.17 Draw a Circle in Matlab e stampa della traiettoria dopo due giri.**

**NB:** Per l'algoritmo è stato utilizzato il Turtlebot Toolbox di Matlab, una libreria apposita e scaricabile dal sito, con funzioni già pre-inizializzate come *GetFunzione* o *SetFunzione* per ottenere in maniera veloce e fluida il risultato voluto.

## Capitolo 3

### Funzionalità di base del Turtlebot

#### 3.1 Il Turtlebot stack

Il Turtlebot stack fornisce tutti i driver di base validi all'utilizzo di un Turtlebot in ambiente ROS, nei paragrafi successivi verranno analizzati nel dettaglio quelli di maggior importanza per questo studio di tesi:

- turtlebot\_gazebo
- turtlebot\_teleop
- turtlebot\_bringup
- turtlebot\_navigation

##### 3.1.1 turtlebot\_gazebo

Il package turtlebot\_gazebo offre una specializzazione dell'applicativo Gazebo per simulazioni in ROS. All'interno della cartella /worlds sono presenti degli scenari già costruiti in cui poter testare il proprio Turtlebot; è comunque possibile crearne uno da capo digitando:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch  
world_file:=/opt/ros/kinetic/share/turtlebot_gazebo/worlds/empty.world
```

All'avvio di un file .launch da terminale viene mostrato un sommario delle operazioni di caricamento effettuate dal sistema operativo, tra queste vi sono i nodi inizializzati (NODES) ed i parametri di ROS passati al software, tra questi il topic che si occupa di convertire i dati dal sensore di profondità in dati LaserScan

---

(*/depthimage\_to\_laserscan/output\_frame\_id*), la distribuzione (*/roscdistro*) e versione di ROS corrente (*/rosversion*).

Trattandosi di un simulatore, Gazebo si occupa anche di ricreare nodi e topic che ricreino la presenza di reali sensori e attuatori. Nei paragrafi successivi verranno analizzati quelli di maggior utilizzo.

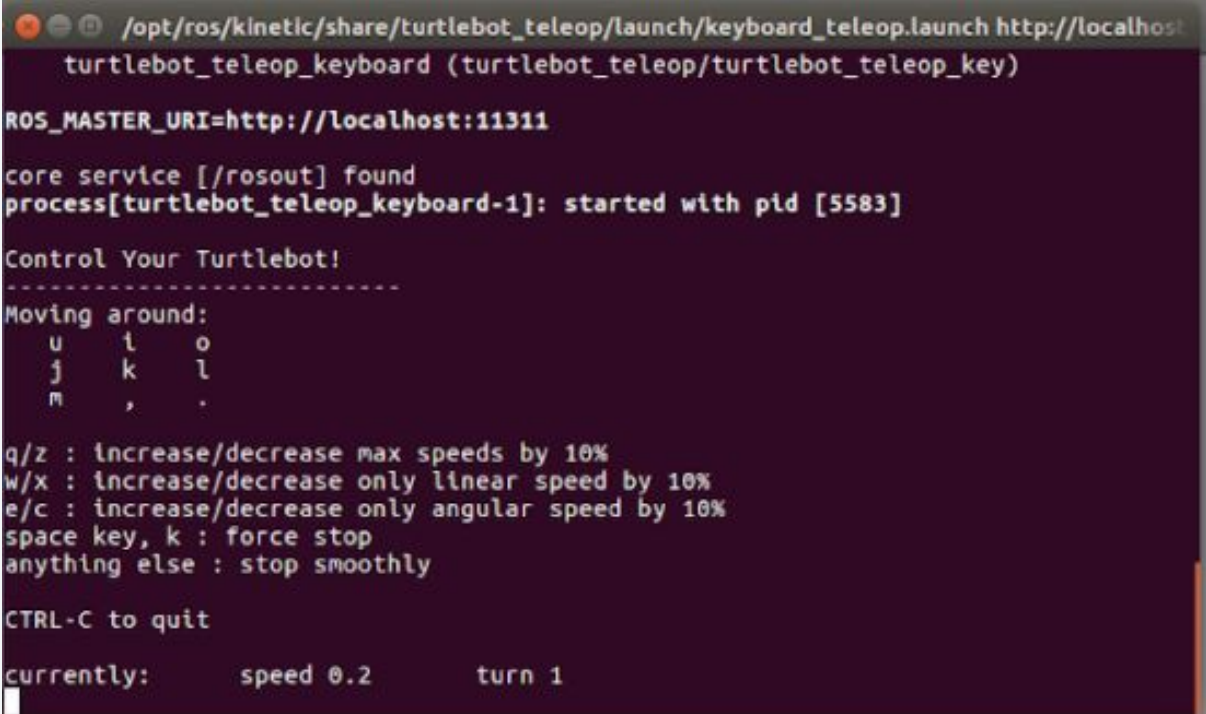
### 3.1.2 turtlebot\_teleop

Il package `turtlebot_teleop` permette la creazione di un nodo per tele-operare da un'interfaccia grafica il Turtlebot all'interno di un ambiente simulato o nella realtà attraverso tre dispositivi di default quali la tastiera o i controller Sony PS3 e Microsoft xbox 360.

Per avviare un nodo *teleop*:

```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

Fig.18 Interfaccia grafica teleop



```
/opt/ros/kinetic/share/turtlebot_teleop/launch/keyboard_teleop.launch http://localhost:11311
turtlebot_teleop_keyboard (turtlebot_teleop/turtlebot_teleop_key)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[turtlebot_teleop_keyboard-1]: started with pid [5583]

Control Your Turtlebot!
-----
Moving around:
  u   t   o
  j   k   l
  m   ,   .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.2      turn 1
```

Come si evince dalla fig.18 il file `keyboard.launch` crea un nodo di nome `turtlebot_teleop_keyboard` che si occuperà di pubblicare messaggi di tipo `geometry_msgs/Twist` che verranno interpretati da `/gazebo` simulando il movimento desiderato. Analizzando il topic su cui il nodo teleop pubblica messaggi (comando rostopic `echo /cmd_vel_mux/input/teleop`) si evidenzia che la struttura di questi è composta dalla sezione linear `[x,y,z]` e angular `[x,y,z]` e che i soli campi `Linear.x` e `Angular.z` variano al movimento del robot.

### 3.1.3 turtlebot\_bringup

Il package `turtlebot_bringup` fornisce scripts per avviare le funzionalità base del Turtlebot, in particolare all'interno della cartella `launch` sono presenti 4 file `.launch`, due dei quali sono di fondamentale importanza per questo studio: `minimal.launch` e `3dsensor.launch`. Il primo file si occupa di avviare tutti i processi necessari all'attivazione della base Kobuki, mentre il secondo sarà utilizzato per attivare le funzionalità offerte dal 3D montato sul Turtlebot, il sensore Astra in questo caso. Si noterà come i nodi ed i topic generati dall'attivazione di questi due launchers sono gli stessi che `/gazebo` genera nel suo ambiente simulato. Per questo motivo c'è piena compatibilità fra gli script testati in un ambiente simulato e quelli testati su un vero Turtlebot.

Una volta accesa la base mobile e collegata al PC digitando quanto segue, la Kobuki emette un suono che indica l'avvenuto collegamento.

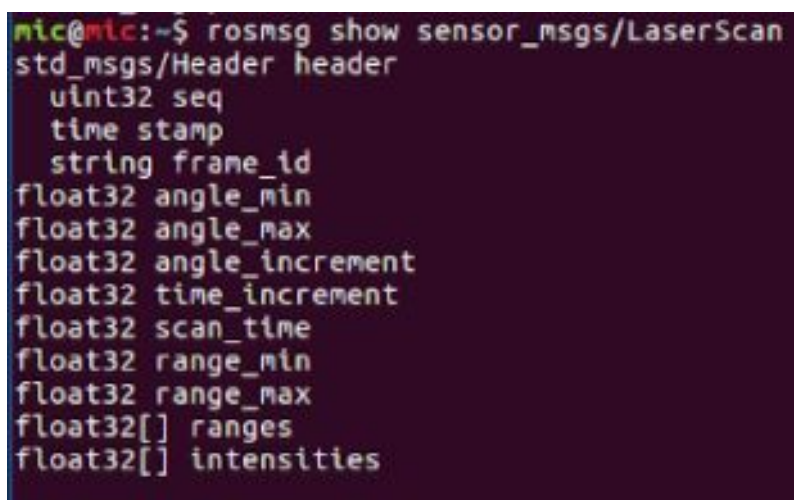
|   |
|---|
| <pre>\$ roslaunch turtlebot_bringup minimal.launch</pre>  |
| <pre>\$ roslaunch turtlebot_bringup 3dsensor.launch</pre> |

Così come per *minimal.launch*, anche *3dsensor.launch* si occupa di creare tutti i processi fondamentale all'acquisizione dei dati provenienti dal sensore 3D. *minimal.launch* e *3dsensor.launch* sono le due componenti fondamentali per gli applicativi di navigazione introdotti nel seguente paragrafo.

### 3.1.4 turtlebot\_navigation

Veniamo ora alla presentazione del navigation package. Esso è composto da un set di algoritmi che sfruttano i sensori del robot e l'odometria per controllarne i movimenti attraverso messaggi standard di velocità lineare ed angolare del tipo *geometry\_msgs/Twist*. Gli obiettivi della navigazione sono tipicamente quelli della creazione di una mappa dello spazio circostante sfruttando i sensori laser o il raggiungimento di una posizione di goal evitando ostacoli statici e dinamici percorrendo il percorso a costo minore.

Risulta quindi essenziale la presenza di un sensore visivo per “osservare” le immediate vicinanze del robot. Il Turtlebot in questione non presenta un sensore Laser vero e proprio, ma riesce a ricavare dati 2D del tipo *sensor\_msgs/LaserScan* convertendo le informazioni lette sul topic */camera/depth/image\_raw* generate dal sensore di profondità.



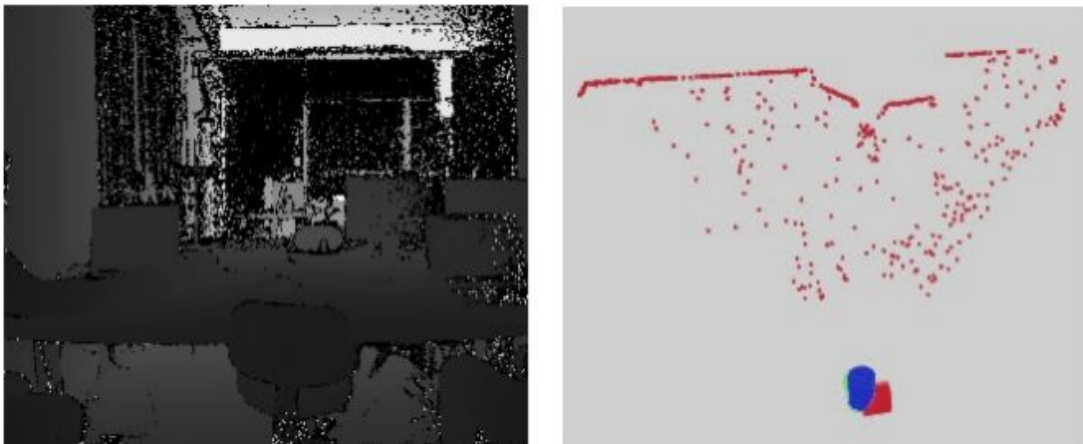
```
mic@mic:~$ rosmmsg show sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Fig.18a rappresentazione del messaggio LaserScan



In particolare sono indicati i parametri del sensore laser quali angolo di inizio e fine misurazione (`angle_min`, `angle_max`), tempo tra due misurazioni (`scan_time`), distanza minima e massima di misurazione (`range_min` e `range_max`) e gli array `ranges` e `intensities` indicanti distanza e intensità del segnale di ritorno.

Come facilmente intuibile dal nome, il nodo `/depthimage_to_laserscan` si sottoscrive al topic `/camera/depth/image_raw` e pubblica in output messaggi di tipo `sensor_msgs/LaserScan` sul topic `/scan`.



**Fig.18b** Una depth-image (sinistra) ed il rispettivo scan equivalente (destra)

Il compito svolto da questo nodo è esattamente inteso per risolvere il problema di simulare la presenza di uno scanner Laser vero, in quanto il risultato ottenuto è assimilabile a ciò che si otterrebbe proiettando verticalmente sul piano di un sensore laser tutti gli oggetti inquadrati dalla telecamera.

Una delle funzionalità di maggior rilevanza della robotica mobile è sicuramente quella di poter ricreare una mappa 2D dell'ambiente esplorato dal robot sfruttando le letture dei sensori visivi per poi effettuare una navigazione autonoma. In ROS ciò avviene attraverso un processo di SLAM (Simultaneous Localization and Mapping) in cui si costruisce una mappa posizionale ed al tempo stesso si stima la traiettoria compiuta.

---

Il vantaggio di SLAM rispetto ad una semplice percezione sensoriale dell'ambiente è la precisione con cui si costruisce la mappa poiché una combinazione fra dati visivi e dati odometrici permette una riduzione degli errori provenienti da rumore visivo o inaccuratezza dei sensori.

Vi sono molti algoritmi per la costruzione di una mappa, ma ROS ne utilizza uno definito come GMapping. Per avviare il processo di ricostruzione dell'ambiente circostante si richiama il file *gmapping.launch* del package *turtlebot\_navigation*:

```
$ roslaunch turtlebot_navigation gmapping_demo.launch
```

E' ora necessario introdurre il software RViz utile per la visualizzazione del processo di SLAM in corso:

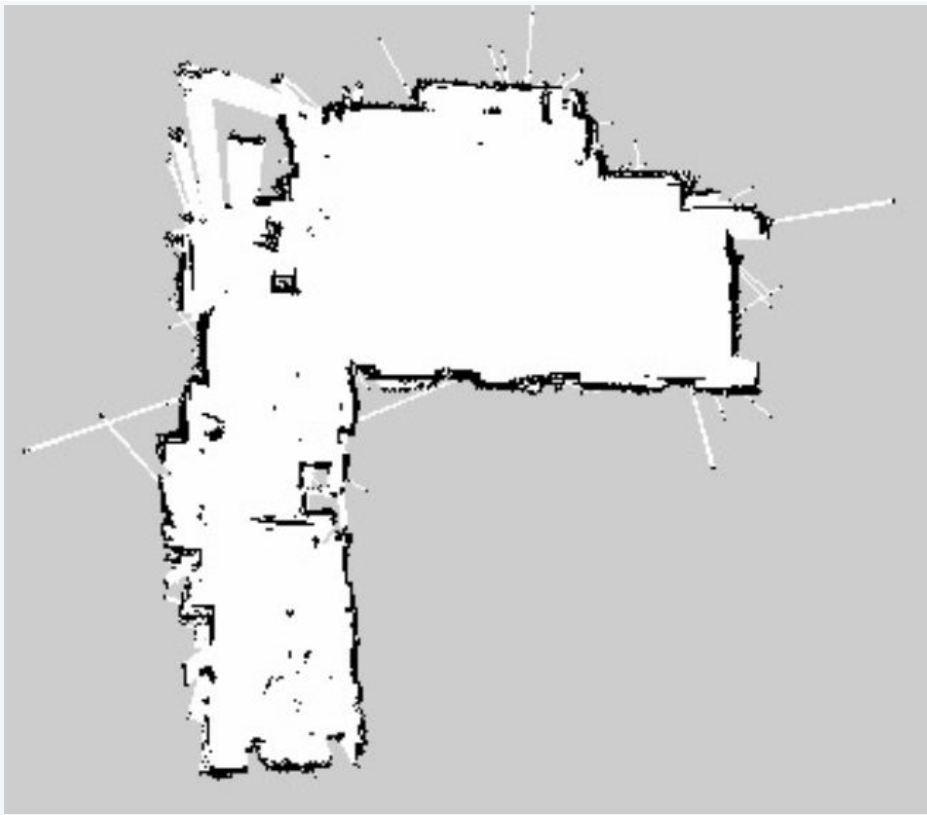
```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

Esso fa parte del package *rviz\_launchers*, specializzazione del più ampio RViz utilizzato nelle applicazioni di robotica come supporto visivo.

Il processo di costruzione della mappa necessita dell'intervento manuale di un operatore che avrà cura di far muovere il robot nell' ambiente circostante utilizzando l'interfaccia fornita dal nodo */teleop* precedentemente introdotto. Sull' interfaccia RViz si verrà a creare progressivamente una ricostruzione degli spazi esplorati con una qualità che sarà risultato della precisione dei sensori e della perizia del lavoro dell'operatore.

A lavoro terminato il comando per salvare la scansione effettuata è:

```
$ rosrun map_server map_server -f  
/home/numeutente/turtlebot_ws/src/ambiente.yaml
```



**Fig.19 risultato della scansione del laboratorio di Robotica**

Ottenuta una ricostruzione dell'ambiente circostante è ora fondamentale che il robot sia capace di effettuare una stima della sua posizione in essa, tale processo è il risultato del confronto tra la mappa e i dati provenienti dai sensori.

Nel navigation stack di ROS\_Turtlebot l'algoritmo utilizzato per la stima probabilistica della posizione durante il movimento prende il nome di AMCL (Adaptive Monte Carlo Localization), specializzazione ROS del ben più noto algoritmo MCL. Esso ad ogni ciclo stima dove dovrebbe essere il robot in base ai dati ricevuti dal sensore laser reale o simulato che sia. Ogni singola possibile posizione è rappresentata da un vettore [posizione ed orientamento] definito particella. Le operazioni di stima effettuate con AMCL non generano mai una singola particella, ma piuttosto un gruppo di esse rappresentante un'area in cui dovrebbe trovarsi il robot in base alle misurazioni laser.

---

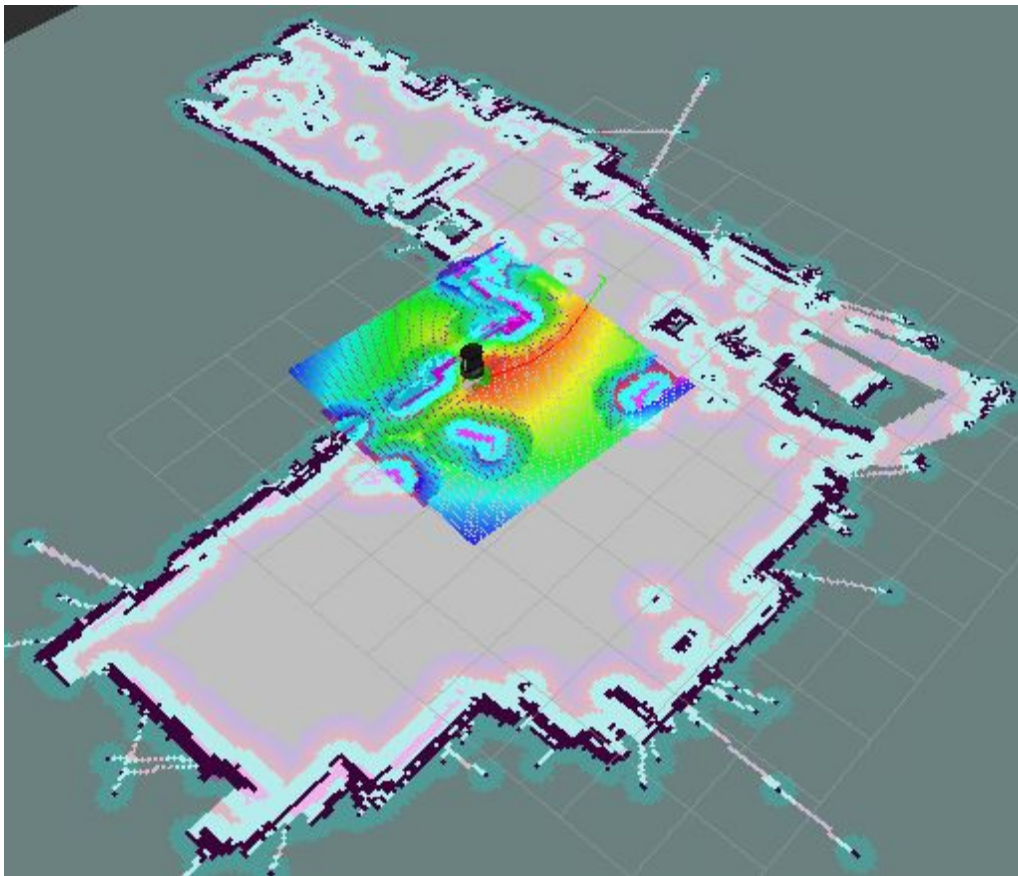
L'efficienza di questo processo si basa sul fatto che vengano scartate tutte quelle particelle le cui informazioni non corrispondono ai dati sensoriali.

Una volta salvata la mappa, si è passati alle fasi di test della qualità di navigazione autonoma del robot.

```
$ roslaunch turtlebot_navigation amcl_demo.launch
```

```
map_file:=/home/numeutente/turtlebot_ws/src/lab.yaml
```

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```



**Fig.20** esplorazione autonoma dell' ambiente con AMCL

Il compito iniziale dell'operatore è quello di comunicare la posizione iniziale del robot nella mappa indicando sull' interfaccia grafica RViz un punto  $[x,y]$  ed un orientamento utilizzando il tool "2D Pose Estimate". A questo punto è possibile comunicare al Turtlebot di raggiungere una posizione a scelta nella mappa attraverso il comando "2D Nav Goal". La descrizione di come esso avvenga non è parte di questo studio di tesi.

### **3.2 Risultati sperimentali**

Nel seguente paragrafo sarà descritta la parte di lavoro fatta durante l'attività di tirocinio riguardante ciò che è stato detto finora.

#### **3.2.1 find\_recharge\_dock.py**

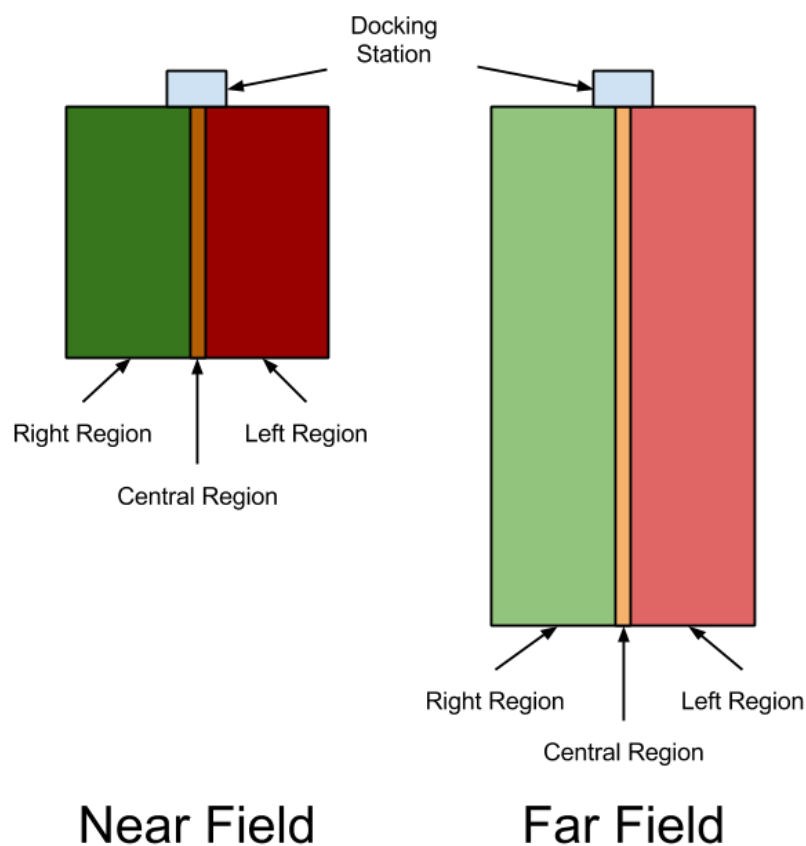


**Fig.23 Turtlebot e dock di ricarica**

La batteria della base Kobuki permette un'operatività media di circa 3 ore variabili in base all' intensità di utilizzo. Abbiamo deciso quindi di sintetizzare una procedura che permetta al robot di raggiungere autonomamente la sua postazione di ricarica posizionata all' interno di un ambiente noto.

---

Per fare ciò ci siamo basati sulla libreria `kobuki_auto_docking` che implementa la ricerca della docking station utilizzando i 3 sensori infrarossi posizionati sul frontale della base. La problematica che abbiamo voluto risolvere è stata quella di creare un algoritmo che posizioni il robot ad una distanza ideale a partire dalla quale effettuare la ricerca con gli infrarossi. Note a priori le coordinate del dock di ricarica, abbiamo inserito nell'algoritmo delle coordinate che permettessero al robot di effettuare la ricerca ad una distanza di 3 metri circa.



**Fig.24 funzionamento degli infrarossi di Kobuki**

All'avvio dell'algoritmo il sistema richiama una subroutine che riceve in input le coordinate del punto di arrivo e un quaternion di orientamento scelto di default.

All' interno della stessa è richiamata una procedura di *MoveBaseGoal()* del package *move\_base* di ROS che fornisce un' implementazione di un nodo che si occupa di pianificare il cammino verso tale posizione. Qualora la procedura di arrivo nella posizione scelta abbia successo viene richiamata la ricerca della dock station attraverso il comando *roslaunch kobuki\_auto\_docking activate.launch* della libreria di Kobuki.

### 3.2.2 go\_back\_bringup.py

Una volta che la ricarica della base mobile è terminata risulta necessario lo sviluppo di una procedura che permetta al robot di staccarsi autonomamente dal dock per essere subito operativo. Il problema nasce dal fatto che di default non è impostata una rilevazione della posizione o meno del robot sulla base di ricarica e ciò comportava che un operatore staccasse manualmente il Turtlebot. Abbiamo allora modificato lo script che viene lanciato di default all'attivazione della base mobile: il *minimal.launch* introdotto nei precedenti paragrafi.

All' avvio della procedura viene inizializzato un timer che viene utilizzato come riferimento temporale per la scelta di quale operazione effettuare tra quelle di retromarcia, rotazione e fermo. La fase di retromarcia viene avviata per i primi 3 secondi finiti i quali si passa alla fase di rotazione che dura circa 4 secondi che conclude la procedura. Alla fine il robot dovrebbe trovarsi 30 cm distante dalla base e ruotato di circa 180 gradi rispetto l'orientamento originale.

## Capitolo 4

### Trasduttori on/off board

Con questo ultimo capitolo ho voluto concentrare l'attenzione sui sensori presenti a bordo, in quanto in robotica sono dei concetti chiave per la classificazione ed importanza di un robot. Inoltre ho deciso di focalizzarmi anche su ciò che manca al Turtlebot, ovvero possibili trasduttori esterni che potrebbero essere implementati a bordo per renderlo all'avanguardia tecnologica e quindi più efficiente, più completo.

#### 4.1 Astra Orbbec : pregi e difetti



**Fig.25 Hardware della camera Astra Orbbec**

Come abbiamo visto, la camera utilizzata presenta numerose funzionalità e ben tre diversi sensori visivi che, per un'attività di sperimentazione come quella effettuata durante questo tirocinio, si sono rivelate condizioni ottimali. Per quanto riguarda la risoluzione, se si resta in un certo range di distanza (0.6 m - 5 m), si hanno risultati accettabili mentre se ci si allontana oltre lo scostamento (deviation) dell'immagine

---



aumenta all'aumentare della distanza. A 5 metri avremo una deviazione di circa 1,20% rispetto alla realtà.

La *view* è caratterizzata dall'unione di un angolo di campo orizzontale di  $60^\circ$  e da un angolo di campo verticale di circa  $50^\circ$  con un range di portata che va dagli 0,6 m agli 8 m.

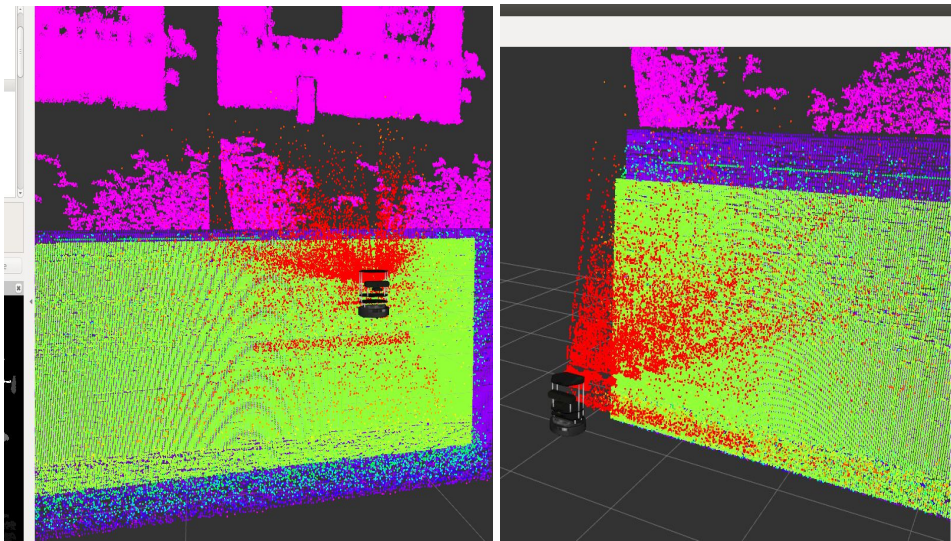
Posto un oggetto di fronte al TB è stato immediato verificare che la camera presenta numerosi punti ciechi in zone al di sotto del suo FIELD OF VIEW.

Per esempio come mostrato in figura 26 al di sotto di circa 60 cm in diagonale non riesce a vedere nulla.



**Fig.26 Lower Bound della view**

Per questo motivo prima che un oggetto arrivi a circa 20 cm (presi orizzontalmente) dal TB (*critical point*) se il robot non ha già elaborato un *obstacle avoid* e quindi il percorso migliore per aggirarlo, ci urterà sicuramente contro.



**Fig.27a e 27b Eye View della fotocamera**

Nelle figure 27a e 27b infine è mostrata la stessa immagine da angolazioni diverse: il Turtlebot in ambiente simulato riconosce il muro che ha di fronte in ambiente reale grazie alla fotocamera. Il colore rosso mostra il range di visione di cui abbiamo parlato prima; i colori verde e blu mostrano il piano orizzontale su cui il robot capisce di trovarsi (in blu il contorno); in rosa invece focalizza il piano verticale, ovvero il muro, di cui focalizza anche una finestra (in nero, in quanto ciò che c'è fuori è out dal raggio d'azione).

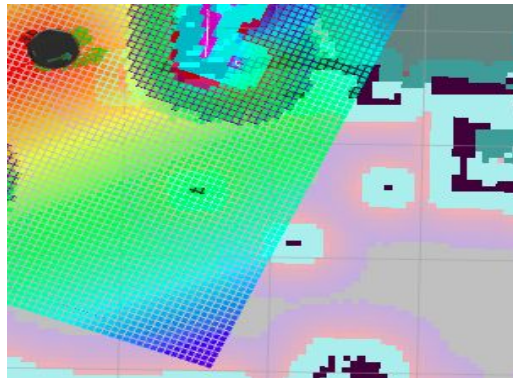
#### **4.1.1 Il problema del tavolo**

Sappiamo che RViz è un software real time che ci informa sulla posizione e progressivi spostamenti del robot in un ambiente statico (ricostruito ed elaborato tramite l'Astra camera).

Una considerazione importante è capire “come” il robot vede e riconosce gli oggetti.

Dato che ricostruisce l'ambiente grazie alla camera per ottenere su interfaccia grafica una visuale dall'alto della stanza, tende come prima cosa a segnalare come ombre i punti ciechi.

Dopodichè tende a schiacciare gli oggetti circostanti identificandoli come possibili ostacoli. Ogni oggetto piano rialzato come un tavolo crea forte ambiguità, poiché schiacciandolo è visto come fosse parte del pavimento e il robot ne riconosce solo le gambe in quanto sporgono come fossero piccoli oggetti cilindrici o addirittura segmenti (di dubbia pericolosità per il suo passaggio). Effettuando numerose prove in laboratorio è possibile osservare come il robot non sempre riesce ad evitarli e in media il 30% delle volte ci urta contro.



**Fig.28 Il tavolo visto su RViz come punti neri**

Dato che non si può arginare il problema è naturale pensare a dei metodi per ridurre l'errore: come prima cosa è importante per il robot avere una stima della propria posizione che sia il più possibile vicina a valori reali. Inoltre si potrebbero aggiungere nuovi sensori di controllo che verifichino la vicinanza a possibili ostacoli. Nel corso di questo capitolo vedremo tutto il lavoro effettuato in laboratorio per risolvere questo problema.

#### **4.2 Stima della posizione**

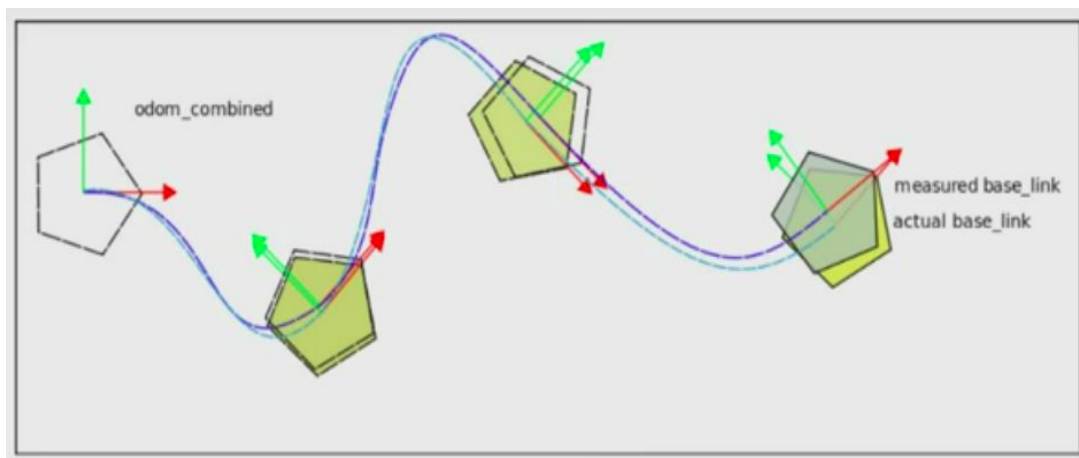
L'odometria, di cui abbiamo già parlato precedentemente, è la tecnica per stimare la posizione di un veicolo su ruote che si basa su informazioni provenienti da sensori che misurano lo spazio percorso dalle ruote.

Il Turtlebot ha un funzionamento a feedback per eseguire un calcolo (o meglio stima) della sua posizione utilizzando un continuo colloquio tra gli encoder presenti sulle ruote e il giroscopio al suo interno: si filtrano inoltre tali informazioni per ottenere risultati più vicini alla realtà.

La precisione di tale stima dipende quindi da due cose: dalla precisione del chip applicato dalla fabbrica madre (il cui part number è scritto sul lato destro del Turtlebot power/sensor board) e dal modo in cui viene gestita l'odometria.

Il chip permette di effettuare una stima della posizione del robot e del suo angolo di rotazione rispetto ad uno di riferimento.

Come detto però è fondamentale sfruttare l'odometria, ovvero l'uso dei dati del sensore di movimento (encoder delle ruote) per stimare il cambiamento di posizione ogni volta (vengono presi degli intervalli di tempo in cui effettuare la stima) rispetto a quella di partenza.

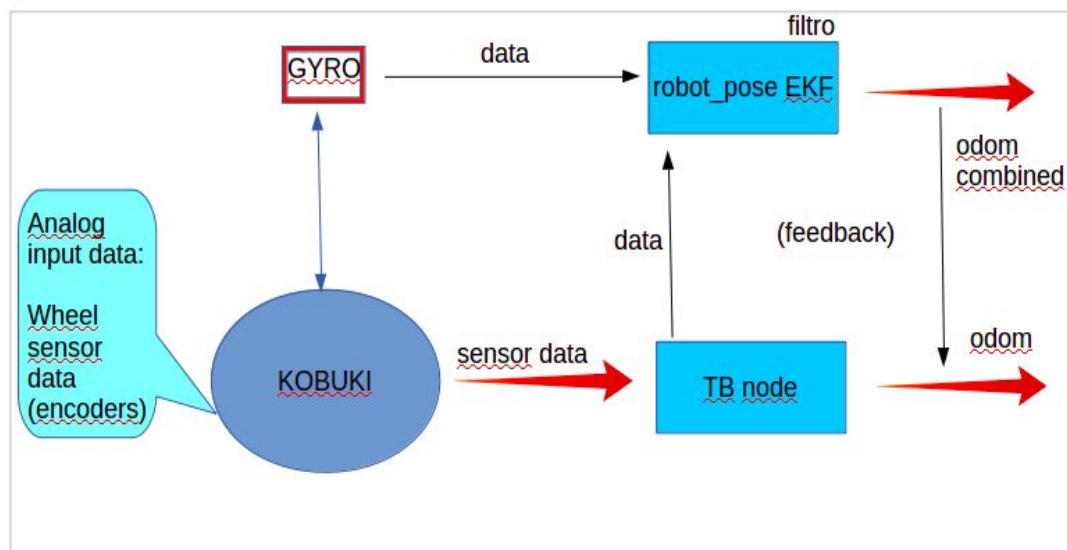


**Fig.29 Distacco tra posizione reale e posizione stimata**

Il nodo Ros riceve i dati dagli encoders ed elabora come si sta muovendo: aggiungendo quindi un giroscopio alla base si avrà una combinazione di dati più accurati. **NB:** dato che si parla di stima e non di misura nella figura 29 ci dovrebbe essere scritto `estimated base_link` invece che `measured base_link`.

Infine, per rendere i dati più vicini a quelli reali sarebbe opportuno utilizzare un filtro: in robotica è molto utilizzato l'*ekf* (extended Kalman Filter).

L'ekf (extended kalman filter) prende tutte le misurazioni osservate nel tempo provenienti sia dagli encoders che dal giroscopio, attenuando ad ogni ciclo rumore o altre inesattezze (fastidi) producendo in output valori che tendono a essere vicini a quelli reali.



**Fig.30 Sensor Fusion (integrazione delle informazioni) utilizzata dal Turtlebot**

Per concludere quindi all'interno della base Kobuki non c'è una bussola che fa da supporto per l'orientamento, ma c'è una girobussola, ovvero un sistema di navigazione non basato sul campo magnetico terrestre ma sulle proprietà giroscopiche.

#### 4.2.1 Risultati sperimentali

Per ottenere risultati soddisfacenti è importante valutare con opportuni algoritmi e tecniche quanto la precisione del robot nel riconoscere la sua posizione sia accurata. La tecnica più semplice consiste nel far percorrere al robot una traiettoria circolare mantenendo sempre lo stesso angolo di curvatura e vedere di quanto si discosta dalla traiettoria voluta al passare dei giri. Ciò lo abbiamo già visto nel secondo capitolo di questa tesi, in quanto è stato facile effettuarlo in ambiente Matlab.

Purtroppo però questo algoritmo non è del tutto efficace in quanto il movimento è solo angolare. Quindi ho pensato di utilizzare un algoritmo che prendesse degli input anche lineari. La scelta è ricaduta sul “*draw a square*” ovvero il classico test del quadrato. Il codice python è stato preso dal web, ma poi rivisitato inserendo un tempo di campionamento più elevato (20 HZ) e due diversi comandi di twist per i motori. Di conseguenza è stato facile richiamarlo in Matlab aggiungendo una funzione di plot (stampa) per l’odometria.

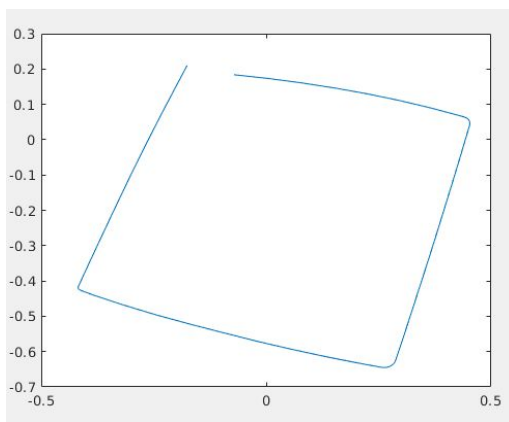


Fig.31 plot della traiettoria dopo un giro

Si intuiscono in questo modo problemi evidenti: il quadrato non è perfetto; il punto di start non coincide con quello di end; la traiettoria non sempre è lineare e le distanze sono diverse tra loro.

Sono quindi arrivato alla conclusione che tutto ciò è dovuto a due principali fattori: la forma delle ruote (anche una piccola imperfezione potrebbe far variare la traiettoria) e il pavimento del laboratorio non perfettamente liscio.

#### 4.2.2 Precisione odometrica e motion capture

Per concludere questo argomento ho pensato di ricavare alcune misure, in modo tale da avere un’idea non solo digitale ma anche pratica della traiettoria del robot.

---

Per farlo ho utilizzato sempre l'algoritmo del *draw a square*: inserendo prima di ogni ciclo una funzione di wait, il robot si è fermato per circa un secondo prima di girare e ciò mi ha consentito di attaccare a terra dei nastri di colori diversi (un colore per giro).

Per prendere le X e le Y, in modo da avere risultati digitali ed immediati ho utilizzato il sistema di telecamere Optitrack del laboratorio, mettendo il sensore di posizione in ogni punto preso dal nastro. Quindi grazie al sistema di motion capture ho raccolto i seguenti dati:

**Primo Giro (nastri gialli) :**

|            |             |
|------------|-------------|
| <b>(1)</b> | <b>(2)</b>  |
| X : 43 mm  | X : -345 mm |
| Y : 940 mm | Y : 1250 mm |

|             |             |
|-------------|-------------|
| <b>(3)</b>  | <b>(4)</b>  |
| X : -750 mm | X : -390 mm |
| Y : 918 mm  | Y : 470 mm  |

**Secondo Giro (nastri rossi) :**

|            |             |
|------------|-------------|
| <b>(5)</b> | <b>(6)</b>  |
| X : -7 mm  | X : -395 mm |
| Y : 936 mm | Y : 1207 mm |

|             |             |
|-------------|-------------|
| <b>(7)</b>  | <b>(8)</b>  |
| X : -740 mm | X : -555 mm |
| Y : 941 mm  | Y : 531 mm  |

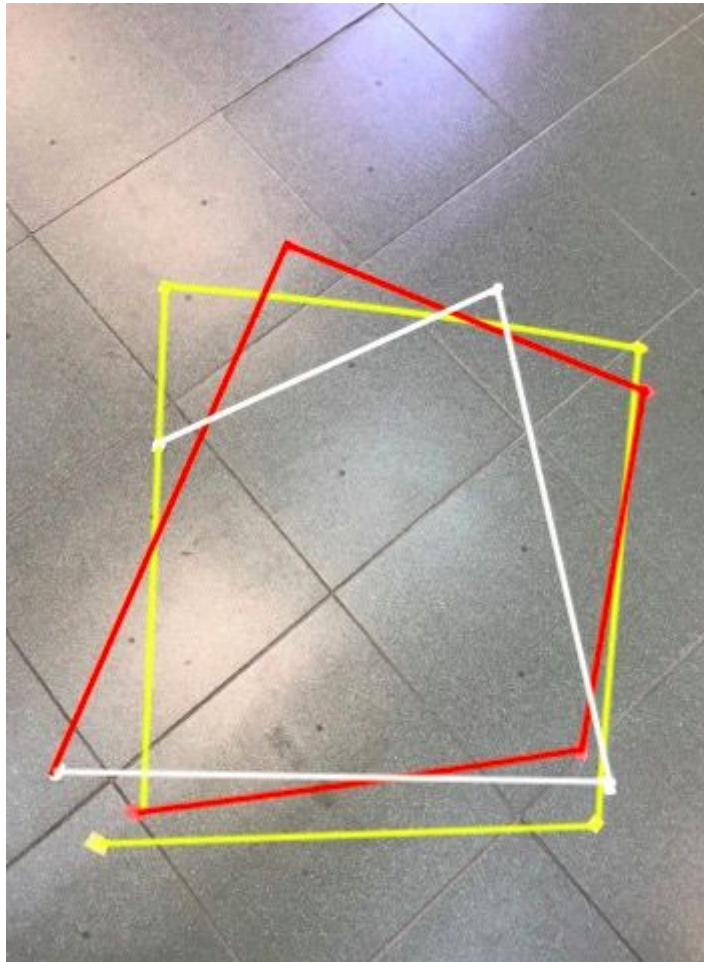
**Terzo Giro (nastri bianchi) :**

|            |             |
|------------|-------------|
| <b>(9)</b> | <b>(10)</b> |
| X : 27 mm  | X : -399 mm |
| Y : 844 mm | Y : 1299 mm |

|             |             |
|-------------|-------------|
| <b>(11)</b> | <b>(12)</b> |
| X : -750 mm | X : -599 mm |
| Y : 999 mm  | Y : 602 mm  |

---





**Fig.32 Traiettorie virtuali del robot ottenute unendo i nastri**

Per calcolare la distanza avendo le coordinate xy ora non resta che utilizzare la formula seguente:

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Dopo il I Giro avremo le seguenti distanze: 50 cm , 52 cm, 56 cm, 60 cm.

Dopo il II Giro avremo : 47 cm, 46 cm, 50 cm, 55 cm.

Dopo il III Giro avremo : 44 cm, 50 cm, 39 cm, xxx.

Notiamo, come detto prima, che le distanze risultano molto diverse tra loro, senza seguire determinate logiche (la traiettoria è quindi casuale).



### **4.3 Arduino come base per introdurre nuovi sensori**

Arduino è una piattaforma hardware programmabile composta da una serie di schede elettroniche dotate di un microcontrollore. Si possono realizzare in maniera relativamente rapida e semplice piccoli dispositivi come controllori di luci, di velocità per motori, sensori di luce, automatismi per il controllo della temperatura e dell'umidità e molti altri progetti che utilizzano sensori, attuatori e comunicazione con altri dispositivi.

La base Kobuki del Turtlebot presenta, come abbiamo visto nei precedenti capitoli, una modesta quantità di sensori che però, risulta insoddisfacente se si pensa al vasto repertorio in cui opera la branca della robotica. Sembra naturale quindi pensare a tutti i possibili trasduttori che potrebbero essere implementati a bordo. Dato che la base rende difficile tale compito in quanto non presenta molte periferiche USB, ho personalmente pensato di far comunicare ROS con l'IDE di Arduino: in questo modo il robot riesce a vedere qualunque sensore venga implementato a bordo della piattaforma.

Ma come è possibile far comunicare tutte le parti in modo fluido e senza incomprensioni? Di seguito è illustrata una rapida guida che culminerà con l'oggetto principale di questo capitolo: l'implementazione a bordo di un sensore ultrasuoni, per cercare di ridurre i punti ciechi della telecamera.

---

### 4.3.1 ROSserial

ROSSerial è un protocollo di grande utilità in quanto è in grado di far comunicare senza grosse difficoltà ROS con qualunque tipo di periferica. E' come se fosse un adattatore virtuale per ottenere un discorso fluido e far sì che i pacchetti giungano a destinazione.

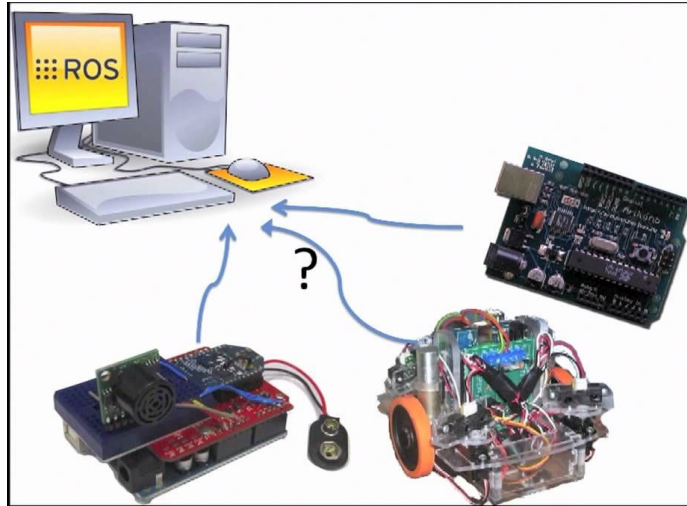


Fig.33 La necessità di un protocollo di comunicazione

ROSSerial include un insieme di librerie *client* fondamentali che consentono agli utenti di ottenere facilmente nodi ROS attivi e in esecuzione sui vari sistemi.

Tra le più importanti abbiamo:

- `rosserial_arduino`, compatibile principalmente con Arduino Uno e Leonardo;
- `rosserial_embeddedLinux`, supporto per i sistemi *embedded* Linux;
- `rosserial_windows`, per la comunicazione ROS-applicazioni Windows;

Per il nostro scopo utilizzeremo quindi il pacchetto **rosserial\_arduino**: fornisce un protocollo di comunicazione ROS che funziona sulla UART dell'Arduino. Infatti trasforma questo in un nodo ROS completo, che può pubblicare e sottoscrivere direttamente messaggi, pubblicare trasformazioni TF e ottenere una connessione con tutto il sistema.

---

Il primo passo da effettuare è chiaramente installare l'IDE Arduino su Ubuntu, che si può fare in maniera facile e veloce seguendo uno tra i tanti tutorial sul web o su YouTube.

Passiamo quindi ai passi da effettuare per installare il pacchetto roserial e ottenere un'interfaccia di comunicazione seriale.

Su terminale digitare:

```
sudo apt-get install ros-kinetic-roserial-arduino
```

```
sudo apt-get install ros-kinetic-roserial
```

```
roslaunch roserial_arduino make_libraries.py opt/arduino-1.8.5/libraries
```

L'ultimo comando crea una libreria detta `ros_lib` con un numero considerevole di codici (per lo più di base) di Arduino interfacciati già per ROS.

La cartella `ros_lib` si troverà seguendo il percorso

*`opt/ros/kinetic/share/roserial_arduino/src/ros_lib`.*

Ora sarà possibile vedere, una volta avviato l'IDE e collegato l'Arduino al pc, la cartella `ros_lib` con tutti i codici in linguaggio Arduino: per verificare il corretto funzionamento basta avviare il codice minimale *Blink* e vedere se compila ed esegue.

**NB:** ARDUINO NON E' STATO ANCORA SOTTOSCRITTO AL TOPIC

Questa è la parte più importante e difficile da capire:

-Bisogna come prima cosa digitare **`sudo usermod -aG dialout $USER`** per ottenere i diritti della presa seriale USB, in quanto senza questo comando il *launch* da sicuramente errore perchè non la riconosce.

-Come seconda cosa va creato il file di launch (avvio di sistema) , che non fa parte del roserial e che quindi va inizializzato:

il file va chiamato **`arduino.launch`** e inserito nella directory:

*`/opt/ros/kinetic/share/roserial_python/cmake`*

---

---

A bordo del file va implementato il seguente codice:

```
<launch>
<node pkg="roserial_python" type="serial_node.py" name="serial_node"
output="screen">
  <param name="~port" value="/dev/ttyACM0" />
  <param name="~baud" value="57600" />
</node>
</launch>
```

Il file è molto banale e facile da implementare in quanto basta sottoscrivere al nodo e impostare come parametri il nome della propria presa USB (*dev*) e il baud rate minimo richiesto dalla scheda.

Finalmente è possibile effettuare una prima comunicazione con ROS-ARDUINO lanciando il file HELLO WORLD.

I passi da effettuare sono:

- andare su `ros_lib` dell'IDE e aprire lo script Hello World
- compilare e caricare lo script sull'Arduino
- eseguire i seguenti comandi su terminale:

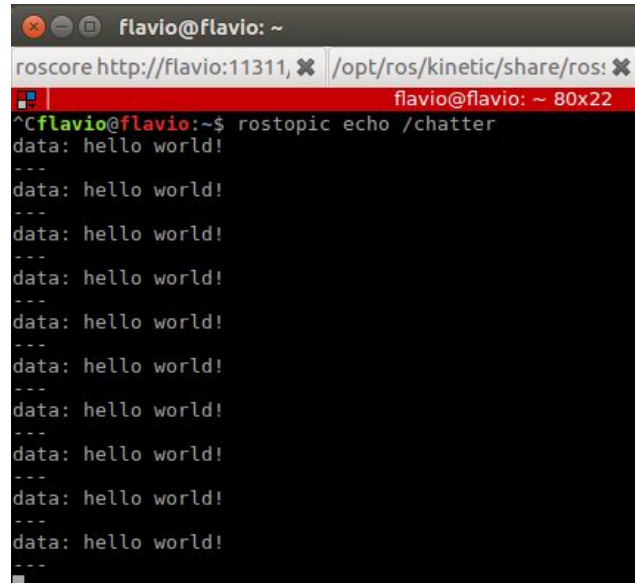
**roscore**

**roslaunch roserial\_python arduino.launch**

**roslaunch roserial\_python serial\_node.py /dev/ttyACM0**

**rostopic list**                    %è possibile vedere il nodo arduino

**rostopic echo /chatter**            %per stampare l'output desiderato

**Fig.34 output di Hello World su ROS tramite Arduino**A screenshot of a terminal window titled 'flavio@flavio: ~'. The window shows the ROS environment with the command 'rostopic echo /chatter' executed. The output consists of multiple lines of 'data: hello world!' separated by dashed lines. The terminal window has a red title bar and standard Linux window controls.

#### 4.3.2 Risultati sperimentali

Come detto sopra ho voluto concentrare il mio lavoro sull'implementazione a bordo di un sensore ultrasuoni in grado di rilevare la presenza di oggetti nelle immediate vicinanze, senza che vi sia un effettivo contatto. La distanza entro cui questi sensori rilevano oggetti è definita *portata vedente* (alcuni modelli dispongono di un sistema di regolazione per poter calibrare la lunghezza di veduta).

Il dispositivo utilizzato è l' SFR05, tra i più utilizzati per scopi didattici in quanto hanno buone prestazioni (precisione di circa 3 mm) ed economici sul mercato. Questo presenta cinque pin in uscita: VCC,GND (per alimentazione e la messa a terra), ECHO (segnale TTL positivo, di durata proporzionale alla distanza rilevata: è il segnale di ritorno con l'informazione), Trig (invio del segnale verso l'esterno), OUT(per risparmiare il pin del TRIG in quanto se abilitato, fa funzionare l'ECHO sia da input che da output).

Come primo passo ho creato un nuovo sketch nell'IDE Arduino chiamato *ultrasound*.

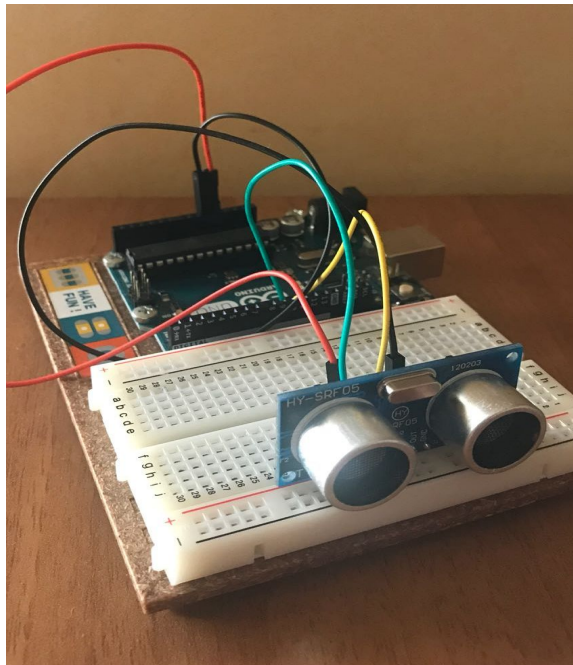


Fig.35 Collegamento del sensore tramite breadboard

Compilando e caricando lo sketch, basta passare al terminale e digitare:

```
roscore
```

```
roslaunch rosserial_python arduino.launch
```

```
roslaunch rosserial_python serial_node.py /dev/ttyAMC0
```

```
rostopic echo /chatter           %Per visualizzare la distanza su schermo
```

Posizionando il sensore a circa 23 cm da un oggetto, è possibile vedere in figura 36 come la stampa cambia di conseguenza mostrando la distanza esatta.

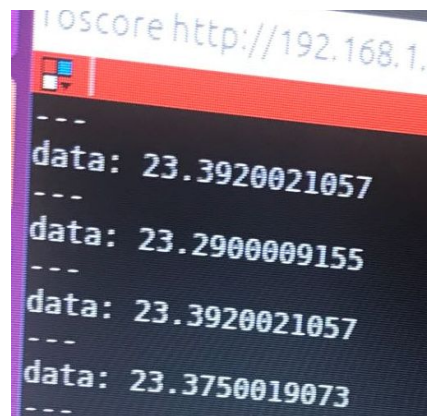


Fig.36 distanza da un oggetto

Una volta terminato il lavoro l'obiettivo prefissato è stato raggiunto: in questo modo si è risolto il problema dei punti ciechi, in quanto posizionando in basso il sensore (come si può vedere in figura 37) il robot capisce quando si sta avvicinando ad un oggetto sconosciuto, anche se tale corpo non è visto dalla telecamera. Si è ridotta notevolmente quindi la possibilità di urti improvvisi.



**Fig.37 Turtlebot più completo con a bordo il sensore ultrasuoni**

## Conclusioni e possibili sviluppi futuri

Il lavoro effettuato ha mostrato un primo personale approccio al mondo della robotica utilizzando la piattaforma software ROS per il controllo di sistema, scoprendo e implementando le funzionalità del robot mobile Turtlebot.

E' stato un lavoro impegnativo ma allo stesso tempo interessante, attraverso cui ho potuto apprendere nuove informazioni sul mondo dei robot, potendo constatare la teoria con la pratica.

Nonostante per questa tesi il lavoro sia giunto al termine, rimangono tante possibili strade future per implementare ancora il nostro "robot maggiordomo" e renderlo più completo.

Si potrebbero per esempio approfondire gli script di *obstacle avoidance* già presenti nelle librerie Turtlebot, rendendole più personali e più precise, andando ad utilizzare ulteriori strumenti : -nuovi sensori di posizione come accelerometri e magnetometri per migliorare la stima che il robot ha di sé nello spazio, magari utilizzando ancora Arduino e ROSserial per implementarli; -un algoritmo più efficiente da seguire come la *logica Fuzzy*.

Oppure introdurre si potrebbe pensare di correggere l'odometria, che abbiamo scoperto essere molto imprecisa utilizzando diversi strumenti:

- Filtri di Kalman completi e PID in matlab;

- Map Matching* in matlab, ovvero utilizzando le misure provenienti dal laser scan della fotocamera si potrebbe pensare di creare un sistema di localizzazione del robot, il quale percepisce le distanze dai vari oggetti che ha intorno, emettendo quindi un feedback per i dati che derivano invece dall'Imu (aggiornando e correggendo possibili errori).

- Utilizzo di *landmark* (particolari forme geometriche che il robot può riconoscere facilmente, magari contenenti informazioni aggiuntive come *bar-code*) le cui caratteristiche vanno però definite nel database del robot.

---



## Sitografia

Mastering ROS for Robotics Programming: Lentin Joseph

<https://drive.google.com/drive/folders/0BxwNOGVcYgmGR0lZSng1RWtiMGs>

<http://wiki.ros.org/ROS/Concepts>

[https://it.wikipedia.org/wiki/Robot\\_Operating\\_System](https://it.wikipedia.org/wiki/Robot_Operating_System)

<http://www.hotblackrobotics.com/it/blog/2017/12/11/installiamo-ros-su-raspberry-pi>

<https://it.wikipedia.org/Robot>

<https://github.com/surabhi96/Library-navigating-robot/wiki/Ultrasonic-sensor-with-ROS>

<https://it.mathworks.com/help/supportpkg/turtlebotrobot/ug/plot-turtlebot-odometry.html>

[http://wiki.ros.org/rosterial\\_arduino/Tutorials/Arduino%20IDE%20Setup](http://wiki.ros.org/rosterial_arduino/Tutorials/Arduino%20IDE%20Setup)

<https://orbbe3d.com/product-astra/>

[http://wiki.ros.org/turtlebot\\_calibration/Tutorials/Calibrate%20Odometry%20and%20Gyro](http://wiki.ros.org/turtlebot_calibration/Tutorials/Calibrate%20Odometry%20and%20Gyro)

<http://wiki.ros.org/rosterial>

[http://wiki.ros.org/rosterial\\_arduino](http://wiki.ros.org/rosterial_arduino)

## Codice Sorgente

Di seguito è riportato tutto il codice in linguaggio Matlab, Python, Arduino di cui è stato parlato finora e che è stato usato e sviluppato in laboratorio durante l'attività di tirocinio con il Turtlebot.

### **find\_recharge\_dock.py**

```
#!/usr/bin/env python
```

```
# Copyright (c) 2015, Mark Silliman
```

```
# All rights reserved.
```

```
# TurtleBot must have minimal.launch & amcl_demo.launch
```

```
# running prior to starting this script
```

```
# For simulation: launch gazebo world & amcl_demo prior to run this script
```

```
# Questo script permette al Turtlebot di raggiungere un punto della mappa scelto da  
utente vicino alla dock di ricarica ed avviare la ricerca della stessa
```

```
import os import rospy
```

```
#Vengono importati tutte le tipologie di messaggi utili allo scopo
```

```
from move_base_msgs.msg import MoveBaseAction,  
MoveBaseGoal
```

```
import actionlib
```

```
from actionlib_msgs.msg import *
```

```
from geometry_msgs.msg import Pose, Point, Quaternion
```

---

**#routine GoToPose**

```
class GoToPose():
def __init__(self):
self.goal_sent = False
```

**# What to do if shut down (e.g. Ctrl-C or failure)**

```
rospy.on_shutdown(self.shutdown)
```

**# Tell the action client that we want to spin a thread by default**

```
self.move_base =
actionlib.SimpleActionClient("move_base", MoveBaseAction)
rospy.loginfo("Wait for the action server to come up")
```

**# Allow up to 5 seconds for the action server to come up**

```
self.move_base.wait_for_server(rospy.Duration(5))
```

**#funzione goto si occupa di inviare le informazioni del punto scelto come goal**

```
def goto(self, pos, quat):
```

**# Send a goal**

```
self.goal_sent = True
goal = MoveBaseGoal()
goal.target_pose.header.frame_id = 'map'
goal.target_pose.header.stamp = rospy.Time.now()
goal.target_pose.pose = Pose(Point(pos['x'], pos['y'],
0.000),
Quaternion(quat['r1'], quat['r2'], quat['r3'],
quat['r4']))
```

**# Start moving**

```
self.move_base.send_goal(goal)
```

**# Allow TurtleBot up to 60 seconds to complete task**

```
success =
self.move_base.wait_for_result(rospy.Duration(60))

state = self.move_base.get_state()
result = False
```

---

```
if success and state == GoalStatus.SUCCEEDED:
```

```
# We made it!
```

```
result = True
else:
self.move_base.cancel_goal()
```

```
self.goal_sent = False
return result
```

```
#funzione shutdown qualora venga premuto ctrl+c
```

```
def shutdown(self):
if self.goal_sent:
self.move_base.cancel_goal()
rospy.loginfo("Stop")
rospy.sleep(1)
```

```
#ciclo principale di controllo
```

```
if __name__ == '__main__':
try:
```

```
#viene inizializzato il nodo nav_test associato a questo algoritmo
```

```
rospy.init_node('nav_test', anonymous=False)
navigator = GoToPose()
```

```
# Customize the following values so they are appropriate for your location
```

```
#queste coordinate sono un punto distante circa 3 metri da dove è posizionato la dock di  
ricarica. La scelta del punto è stata effettuata da RViz durante il processo di navigazione  
autonoma
```

```
position = {'x': 0.992, 'y' : -2.01}
quaternion = {'r1' : 0.000, 'r2' : 0.000, 'r3' : 0.000,  
'r4' : 1.000}
```

```
rospy.loginfo("Go to (%s, %s) pose", position['x'],  
position['y'])
```

**#nella variabile bool success si inserisce il risultato del processo di raggiungimento della posizione di input**

```
success = navigator.goto(position, quaternion)

if success:
    rospy.loginfo("Hooray, reached the desired pose")
```

**#qualora success = true viene richiamato uno script che si occupa di avviare la ricerca della dock**

```
os.system("roslaunch kobuki_auto_docking activate.launch
--screen")

else:
    rospy.loginfo("The base failed to reach the desired
pose")
```

**# Sleep to give the last log messages time to be sent**

```
rospy.sleep(1)

except rospy.ROSInterruptException:
    rospy.loginfo("Ctrl-C caught. Quitting")
```

### **draw\_a\_square.py**

```
import rospy
from geometry_msgs.msg import Twist
from math import radians

class DrawASquare():
    def __init__(self):

        # inizializzo il nodo ROS

        rospy.init_node('drawasquare', anonymous=False)

        # Stop se premo ctrl + c
        rospy.on_shutdown(self.shutdown)
```

**#inizializzo il nodo con 10 comandi in coda per garantire fluidità e non perdere informazioni**

```
self.cmd_vel =  
rospy.Publisher('cmd_vel_mux/input/navi', Twist,  
queue_size=10)
```

**# 20 HZ : tempo di campionamento accettabile NB: in tal modo è possibile controllare il moto del TB creando due intervalli.**

```
r = rospy.Rate(20);
```

**# creo due variabili di Twist() per i motori.**

**# vai a una velocità di 0.2 m/s**

```
move_cmd = Twist()  
move_cmd.linear.x = 0.2
```

**#gira di 90 rad/s**

```
turn_cmd = Twist()  
turn_cmd.linear.x = 0  
turn_cmd.angular.z = radians(90);
```

**#disegna un quadrato : Go forward per 2 secondi (40 x 20 HZ) poi gira per 1 sec (20 x 20 HZ).**

```
count = 0  
while not rospy.is_shutdown():
```

**# va avanti per 0.4 m**

```
rospy.loginfo("Va avanti")  
for x in range(0,40):  
    self.cmd_vel.publish(move_cmd)  
    r.sleep()
```

**# gira di 90 NB: il range qui è pari agli HZ perché ho settato la z a 90**

```
rospy.loginfo("Gira")  
for x in range(0,20):    #gira di 90 per circa un secondo  
    self.cmd_vel.publish(turn_cmd)  
    r.sleep()  
count = count + 1  
if(count == 4):  
    count = 0  
if(count == 0):
```

---

**#sono nella posizione di partenza quindi stampa :**

```
        rospy.loginfo("Ora TurtleBot sarà vicino  
alla posizione di partenza (piu o meno)")
```

```
def shutdown(self):
```

**# stop turtlebot e stampa finale**

```
        rospy.loginfo("Finiscila di disegnare quadrati")  
        self.cmd_vel.publish(Twist())  
        rospy.sleep(1)
```

```
if __name__ == '__main__':  
    try:  
        DrawASquare()  
    except:  
        rospy.loginfo("nodo terminato.")
```

### **helloWorld.ino**

```
/*  
 * rosserial Publisher Example  
 * Prints "hello world!"  
 */  
  
#include <ros.h>  
#include <std_msgs/String.h>  
  
ros::NodeHandle nh;  
  
std_msgs::String str_msg;  
ros::Publisher chatter("chatter", &str_msg);  
  
char hello[13] = "hello world!";
```

---

```
void setup()
{
    nh.initNode();
    nh.advertise(chatter);
}

void loop()
{
    str_msg.data = hello;
    chatter.publish( &str_msg );
    nh.spinOnce();
    delay(1000);
}
```

### **ultraSound.ino**

```
#include <ros.h>
#include <std_msgs/Float64.h>

ros::NodeHandle nh;
std_msgs::Float64 Distance;

//do un nome al mio nodo ROS per l'ultrasuoni, parametrico con &Distance
ros::Publisher chatter("chatter",&Distance);

// definisco i numeri dei pin digitali relativi ai due impulsi
const int trigPin = 9;
const int echoPin = 10;

// definizione variabili NB: ho utilizzato il float in quanto con int i valori alti non
// sono supportati
long durata;
float distanza;
```

---



```
void setup() {
  nh.initNode();           //inizializzo il nodo ROS
  nh.advertise(chatter);
  pinMode(trigPin, OUTPUT); // Setto il trig come Output
  pinMode(echoPin, INPUT); // Setto l'echo come Input
  Serial.begin(57600); // Inizio comunicazione seriale
}

void loop() {

  // Libero il trig per l'operazione successiva
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);

  // Infatti setto lo stato del trig ogni 10 microsecondi (spento, acceso, spento) per inviare un
  // suono che verrà captato dall'echo

  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // Lettura dell'echo e ritorno della durata del suono in microsecondi
  durata = pulseIn(echoPin, HIGH);

  // Calcolo la distanza
  distanza = durata * 0.034 / 2;
  // velocità del suono (0.034) a circa 20° moltiplicato per la durata, /2 sennò si considera
  // andata e ritorno

  // pub data

  Distance.data = distanza;
  chatter.publish(&Distance);
  nh.spinOnce();
  // Per stampare la distanza sul monitor in cm: NB non serve in quanto con rostopic
  // echo chatter posso vederla con ROS
  // Serial.print("Distanza: ");
  // Serial.println(distanza);
  delay(100);
}
```

---