Relazione per il progetto di Misure e Strumentazione per l'Automazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Anno Accademico 2018/2019

Programmazione del Baxter ai fini della cooperazione tra i due bracci per il pick and place di un oggetto

Docente:

Prof. Freddi Alessandro

Studenti:

Cannata Giuseppe Pio

Terramani Simone

INDICE

Introduzione

ROS (*Robot Operating System*) è il framework che è stato utilizzato per lo svolgimento del lavoro di laboratorio.
Esso fornisce: librerie e tools per aiutare gli sviluppatori a creare robot applications; astrazione dell'hardware; driver dei dispositivi; librerie; strumenti di visualizzazione; comunicazione tramite scambio di messaggi tra processi (*message passing*); gestione dei pacchetti e molto altro.
ROS è rilasciato sotto una licenza open source (BSD license) ed è attualmente linux based.
ROS si propone quindi come strumento open source adatto a tutti per la realizzazione di applicazioni per il mondo della robotica. Può essere utilizzato dai professionisti della robotica in progetti ad altissimo contenuto tecnologico ma anche dagli studenti di ogni età per la robotica educativa.
La community di Robot Operating System è in crescita: si stima che negli ultimi sette anni, Ros abbia continuato a crescere fino ad includere una sempre più ampia community di utilizzatori in tutto il mondo. La maggior parte degli utilizzatori si trova nei laboratori di ricerca ma l'adozione di Ros si sta gradualmente spostando in questi anni anche ai settori commerciali, in particolare ai robot industriali e ai service robot.

Nel prosieguo di questa relazione verrà mostrata tutta la procedura (oltre al codice prodotto) per poter permettere al robot antropomorfo Baxter della Rethink Robotics, di eseguire quanto segue:

- pick di un oggetto;
- posizionamento dell'oggetto in una specifica posa nello spazio;
- scambio dell'oggetto tra un braccio e l'altro,

il tutto adottando una politica di obstacle avoidance.

Installazione di ROS e SDK del Baxter su Ubuntu 14.04

L' installazione del framework ROS si compone di 6 steps.
Per completezza inseriamo il link della guida da cui sono state ottenute le informazioni nel seguito presentate.

Link di riferimento guida: http://sdk.rethinkrobotics.com/wiki/Workstation_Setup

1. Download di Ubuntu 14.04

Per poter utilizzare il framework ROS è necessario disporre della versione 14.04 di Ubuntu.
Di seguito sono riportati i link per il download dei file iso:

32-bit (Intel x86): http://releases.ubuntu.com/trusty/ubuntu-14.04.5-desktop-i386.iso

64-bit (AMD64): http://releases.ubuntu.com/trusty/ubuntu-14.04.5-desktop-amd64.iso

2. Installazione ROS

Dopo aver scaricato ed installato Ubuntu possiamo procedere con l'installazione di ROS.
Apriamo un terminale ed eseguiamo i seguenti passaggi:

- modificare sources.list:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/sources.list.d/ros-latest.list'
```

- eseguire:

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add-
```

- verificare la disponibilità di aggiornamenti:

```
sudo apt-get update
```

- installare la versione Indigo di ROS:

```
sudo apt-get install ros-indigo-desktop-full
```

- inizializziare rosdep:

```
sudo rosdep init
rosdep update
```

- installare rosinstall:

```
sudo apt-get install python-rosinstall
```

3. Creazione del WorkSpace

Dopo aver installato ROS Indigo è necessario creare il WorkSpace. Un WorkSpace è una cartella dove è possibile compilare e creare packages utili nella programmazione con ROS.

- creare un ROS Workspace (creare una directory):

```
mkdir -p ~/ros_ws/src
```

- eseguire source ROS and Build:

```
source /opt/ros/indigo/setup.bash
```

Questo comando serve per poter attivare la compilazione. In particolare verrà utilizzato **solamente** questa volta per poter richiamare i metodi comandi di ROS.

- compilare ed installare:

```
cd ~/ros_ws
catkin_make
catkin_make install
```

4. Installazione del modello del Baxter

- installare le dipendenze:

```
sudo apt-get update
```

```
sudo apt-get install git-core python-argparse python-wstool
python-vcstools python-rosdep ros-indigo-control-msgs
ros-indigo-joystick-drivers
```

- installare l'SDK del Baxter:

```
cd ~/ros_ws/src
wstool init .
```

```
wstool merge
https://raw.githubusercontent.com/RethinkRobotics/baxter/ma
ster/baxter_sdk.rosinstall
```

```
wstool update
```

- compilare ed installare:

```
cd ~/ros_ws
catkin_make
catkin_make install
```

5. Configurazione della comunicazione Baxter-ROS

- reperire il file baxter.sh:

```
wget
https://github.com/RethinkRobotics/baxter/raw/master/baxter
.sh
chmod u+x baxter.sh
```

- modificare lo script baxter.sh:

  E' necessario modificare il file baxter.sh. Eseguire:

```
cd ~/ros_ws
gedit baxter.sh
```

Una volta aperto l'editor eseguire le seguenti modifiche:

1. Modificare il campo 'baxter_hostname' in questo modo:
   `**baxter_hostname="baxter_hostname.local"**`
2. Modificare il campo 'your_ip' con l'indirizzo ip del nostro host:
   `**your_ip="192.168.XXX.XXX"**`
3. Modificare il campo 'ros_version' inserendo la nostra versione di ROS:

```
***ros_version="indigo"***
```

Al termine salvare e chiudere l'editor.

6. Verifica corretta installazione

- Avviare l'environment:

```
cd ~/ros_ws
. baxter.sh
```

- Eseguire:

```
env | grep ROS
```

L'output desiderato è il seguente:

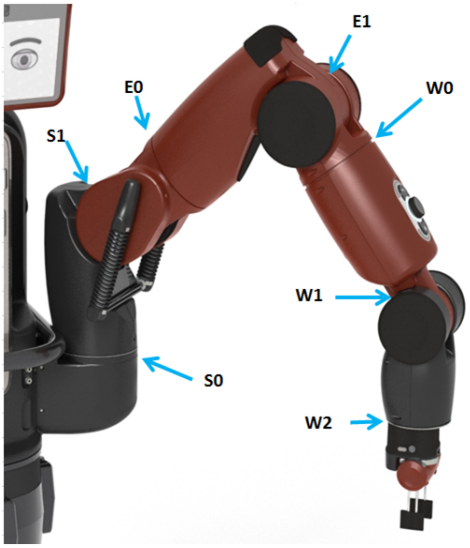
ROS_MASTER_URI - dovrebbe contenere il nome del robot.

ROS_IP - dovrebbe contenere l'ip del nostro host.

Baxter - Rethink Robotics

Baxter è il robot utilizzato all'interno dei lavori di laboratorio ed è, per l'appunto, un robot umanoide antropomorfo. Possiede due bracci con 7 d.o.f. ciascuno e tecnologie di rilevamento all'avanguardia, tra cui forza, posizione e rilevamento e controllo di coppia in ogni giunto, telecamere a supporto di applicazioni di visione artificiale, elementi di input e output integrati per l'utente come display, pulsanti, manopole e altro ancora.
Baxter è stato progettato per un funzionamento continuo e può funzionare 24 ore al giorno per 7 giorni senza il rischio di danneggiamento dell'hardware.

Nel dettaglio, ciascuno dei bracci menzionati sopra è così strutturato (la notazione sarà la stessa utilizzata nel codice sviluppato): la S sta per Shoulder (giunto di spalla), la E per Elbow (giunto di gomito) e la W per wrist (polso).



Ciascun braccio del baxter possiede quindi:

- 2 giunti di spalla (S0,S1);
- 2 giunti di gomito (E0, E1);
- 3 giunti per il polso (W0,W1,W2: il polso è di tipo RPR - Roll Pitch Roll),

per un totale di 7 gradi di libertà.

Quelle fino a qui espresse sono le uniche conoscenze riguardo le struttura del robot necessarie ai fini dello svolgimento del progetto in questione, in quanto l'interfaccia software "maschera" tutte le altre conoscenze di tipo meccanico/strutturale che sarebbero necessarie usualmente.

Di seguito verrà mostrato il procedimento necessario all'installazione di Gazebo (software di simulazione per il Baxter) e alla configurazione dell'ambiente di lavoro.

Installazione Gazebo

Viene mostrata in questa parte della relazione la procedura seguita per l'installazione del simulatore Gazebo, necessario per testare il corretto funzionamento del software prima di un eventuale test su robot reale.
N.B.: Questa procedura segue quella presentata nella sezione **Installazione di ROS su Ubuntu 14.04**
Dopo aver quindi configurato l'ambiente di lavoro bisogna effettuare i seguenti passaggi:

1. Aprire un terminale ed eseguire il comando:

```
$ sudo apt-get install gazebo2 ros-indigo-qt-build
ros-indigo-driver-common ros-indigo-gazebo-ros-control
ros-indigo-gazebo-ros-pkgs ros-indigo-ros-control
ros-indigo-control-toolbox ros-indigo-realtime-tools
ros-indigo-ros-controllers ros-indigo-xacro python-wstool
ros-indigo-tf-conversions ros-indigo-kdl-parser
```

   Questo comando è necessario all'installazione di Gazebo e a tutti i pacchetti accessori necessari al suo funzionamento. Come si può notare leggendo il comando da eseguire, la distribuzione di ROS consigliata per lavorare con il Baxter è quella denominata **Indigo**: a questa si farà riferimento durante tutto il lavoro svolto.

2. Entrare ora nella cartella del proprio workspace ed utilizzare il comando wstool per installare quanto scaricato nel punto 1. Eseguire quindi i seguenti comandi, nella stessa shell, uno di seguito all'altro:

```
$ cd ~/ros_ws/src
$ wstool init .
$ wstool merge
https://raw.githubusercontent.com/RethinkRobotics/baxter_simulator/ma
ster/baxter_simulator.rosinstall
$ wstool update
```

3. Compilare il tutto, eseguendo i comandi:

```
$ source /opt/ros/indigo/setup.bash
$ cd ~/ros_ws
$ catkin_make
```

4. Copiare il file baxter.sh e modificarlo, come visto in Installazione di ROS su Ubuntu 14.04 inserendo l'indirizzo IP della macchina che si sta utilizzando e la versione di ROS in uso (come già detto la versione da utilizzare è Indigo):

```
$ cp src/baxter/baxter.sh .
```

5. Una volta svolto quanto fino ad ora descritto, chiudere tutti i terminali utilizzati. Aprirne ora uno nuovo ed eseguire il seguente comando:

```
$ ./baxter.sh sim
```

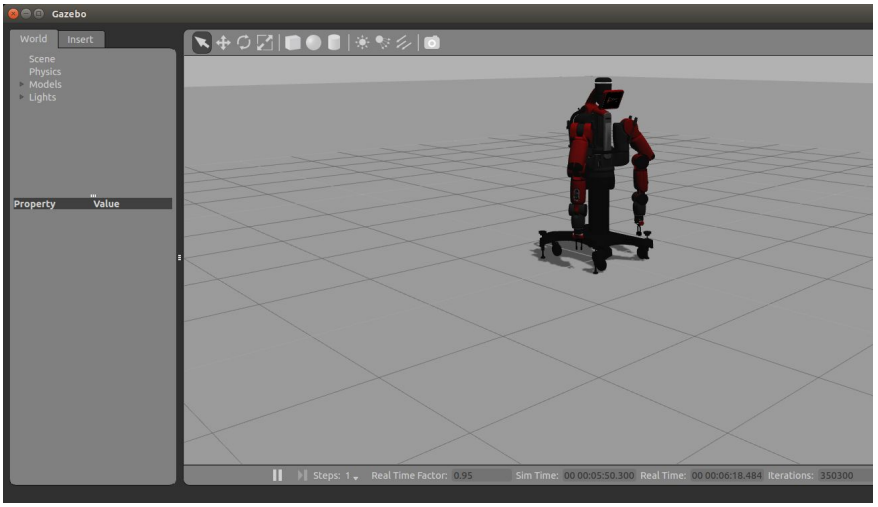
6. Nel medesimo terminale lanciare finalmente il simulatore:

```
$ roslaunch baxter_gazebo baxter_world.launch
```

Il Baxter sarà completamente caricato solo quando nel terminale relativo al simulatore (quello di cui sopra) compariranno le seguenti stringhe di log:

```
[ INFO] [1400513321.531488283, 34.216000000]: Simulator is loaded and
started successfully
[ INFO] [1400513321.535040726, 34.219000000]: Robot is disabled
[ INFO] [1400513321.535125386, 34.220000000]: Gravity compensation
was turned off
```

Il terminale in cui sono stati eseguiti gli step 5 e 6 deve essere il medesimo e questo non deve venir chiuso dall'utente in quanto contiene il processo che mantiene in run il processo di simulazione (chiusura terminale -> chiusura Gazebo).
Al termine di questa procedura l'utente avrà di fronte la seguente schermata:



E’ importante precisare quanto segue: Gazebo, quando lanciato con gli step appena mostrati, contiene una rappresentazione del robot alternativa a quella reale. In altre parole, una volta lanciato è come se nella schermata di Gazebo vi fosse il robot reale ed il collegamento con quest’ultimo fosse stabilito. Ciò viene ora precisato e verrà ripreso quando si mostrerà la procedura di installazione di Rviz e di Moveit!: di nuovo, il processo contenente il modello del Baxter in Gazebo pubblica costantemente informazioni relative alla struttura e allo stato di tutti i componenti del robot e questi possono essere proficuamente utilizzati dal programmatore.

7. Prima di procedere alla stesura del codice oppure alla prova di codici di esempio già forniti risulta necessario abilitare il robot. Aprire quindi un nuovo terminale ed eseguire i comandi:

```
$ ./baxter.sh sim
$ rosrun baxter_tools enable_robot.py -e
```

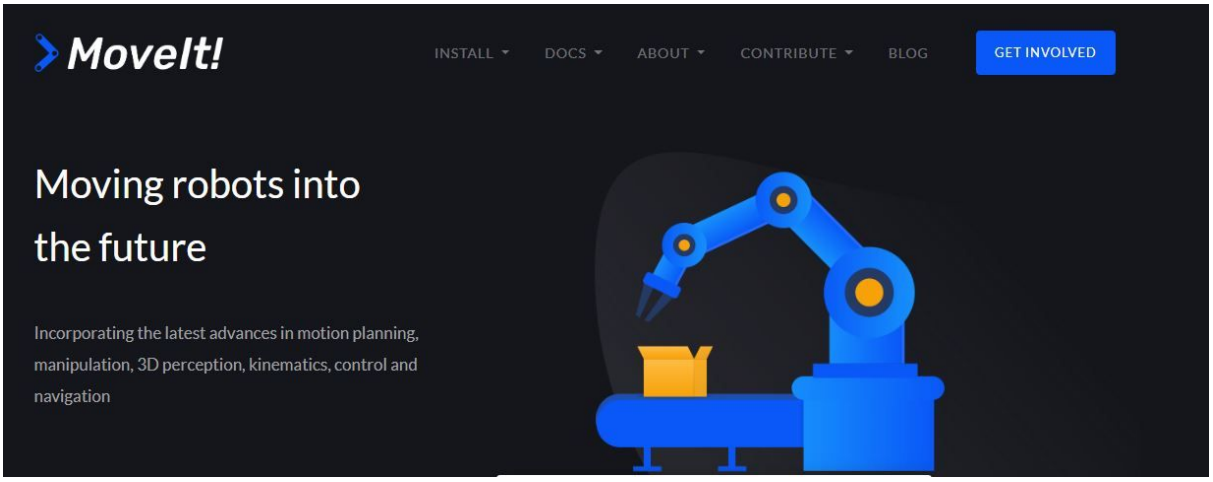
N.B.: Il comando `$ ./baxter.sh sim` va eseguito in ogni nuovo terminale aperto in quanto permette, in pratica, di collegarsi al Baxter e di poter quindi interagire con esso.

8. A titolo esemplificativo, per verificare che il procedimento sia andato a buon fine si possono eseguire, in un nuovo terminale, i comandi:

```
$ ./baxter.sh sim
$ rosrun baxter_examples joint_velocity_wobbler.py
```

Si dovrebbero quindi vedere i bracci del robot oscillare simultaneamente.

Installazione di RViz e framework Moveit!

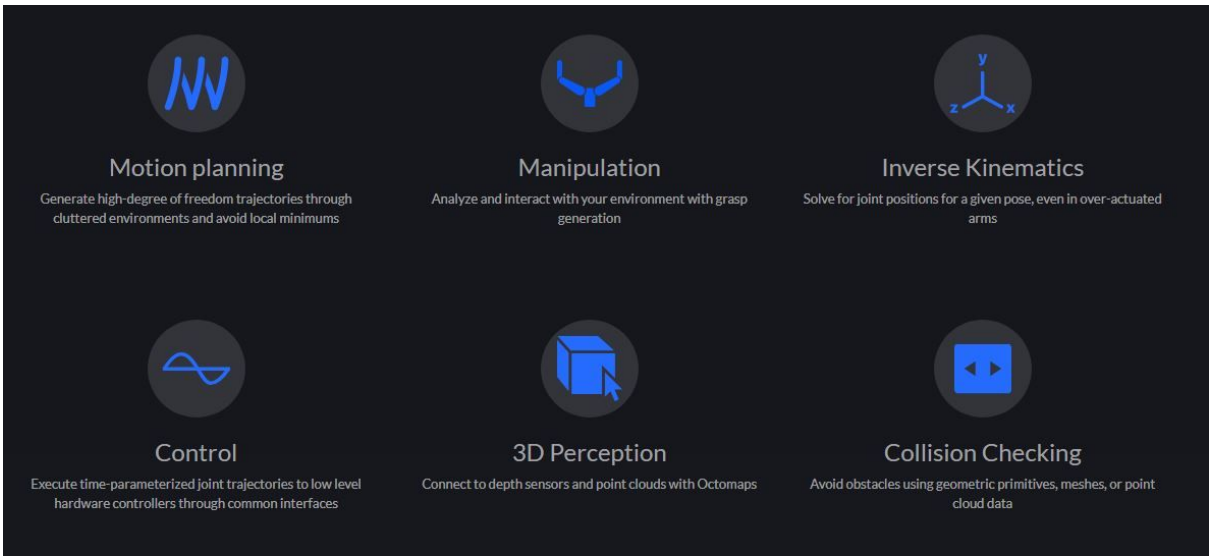


Nel presente lavoro sono stati utilizzati il framework Moveit! e la libreria OMPL (*Open Motion Planning Library*).

Moveit! è un framework di motion planning ed offre funzionalità come:

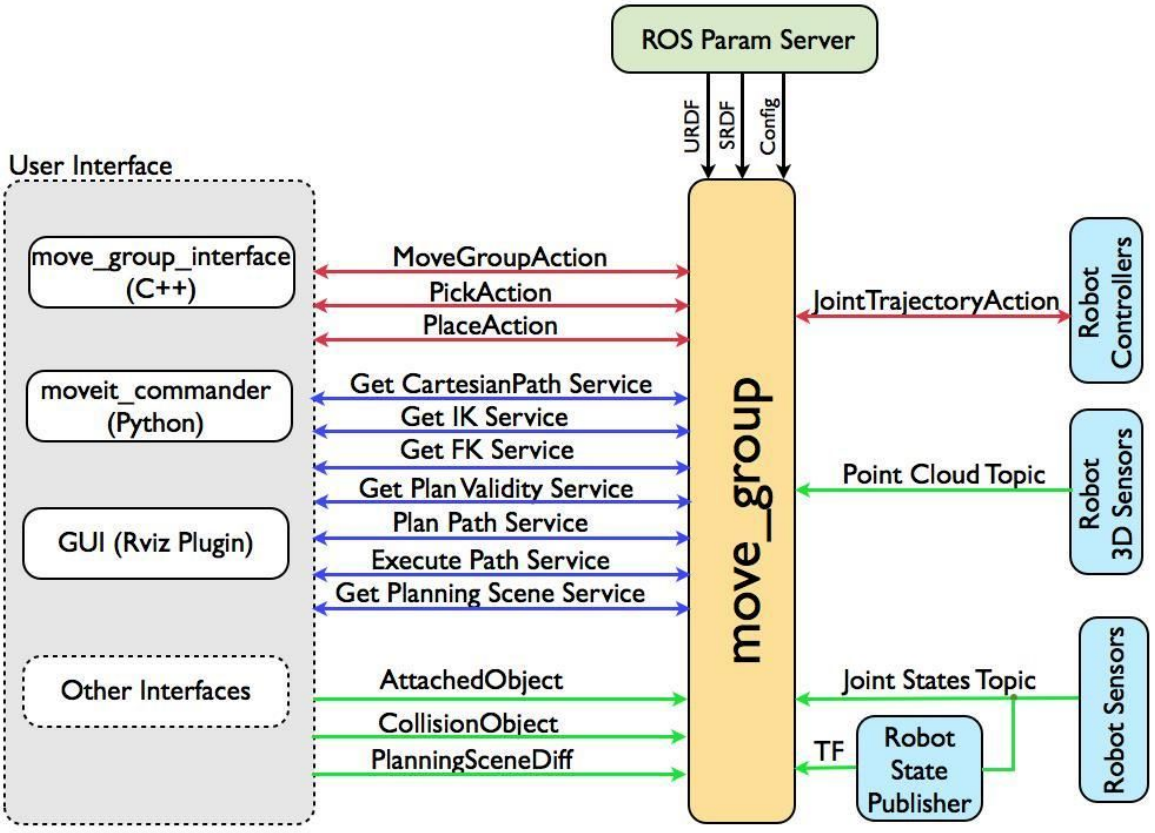
- Cinematica (IK (inversa), FK (diretta), Jacobiano) integrata come plugin di Moveit!;
- Motion Planning (OMPL, SBPL, CHOMP) integrata come plugin di MoveIt!;
- Rappresentazione dell'ambiente (rappresentazione robot, rappresentazione dell'ambiente, controllo collisioni, valutazione vincoli);
- Esecuzione tramite move_groups;
- Benchmarking;
- Database warehouse per l'archiviazione (scene, stati robot, piani di movimento);
- Un'API C ++ / Python e molto altro.

OMPL è una libreria open source di pianificazione del movimento ed è costituita da numerosi algoritmi di motion planning all'avanguardia. È inoltre integrata in Moveit! che fornisce funzionalità di pianificazione del movimento in ROS. I robot sono descritti dai file URDF, che descrivono la geometria del robot, la cinematica e ulteriori informazioni sul robot. Moveit! può caricare tali file, creare spazi di stato appropriati per gruppi di giunti definiti dall'utente (ad esempio "braccio sinistro", "gamba destra", "parte superiore del corpo", "corpo intero", ecc.) e chiamare i pianificatori OMPL per trovare percorsi ammissibili. Esiste un supporto per la cinematica inversa, che rende possibile, ad esempio, l'inclusione di vincoli di end-effector. I percorsi prodotti da OMPL sono tradotti da MoveIt! in traiettorie dinamicamente fattibili (feasible) e MoveIt! rileverà automaticamente le auto-collisioni in una fase di pre-elaborazione. L'ambiente può essere fornito sotto forma di raccolta di oggetti geometrici (triangoli, sfere, cilindri, ecc.), una nuvola di punti (ottenuta da un sensore RGBD) o una combinazione di entrambi.



Di seguito verrà mostrata in maggior dettaglio l'architettura di Moveit!, come avviene il motion planning in Moveit! e come è composta una Planning Scene in Moveit!

Architettura del sistema



Il nodo move_group

La figura sopra mostra l'architettura di sistema di alto livello per il nodo primario fornito da MoveIt! chiamato *move_group*. Questo nodo funge da integratore: riunisce tutti i singoli componenti per fornire un insieme di azioni e servizi ROS da utilizzare per gli utenti.

Interfaccia utente

Gli utenti possono accedere alle azioni e ai servizi forniti da move_group in uno dei tre modi seguenti:

- In **C ++** - utilizzando il pacchetto *move_group_interface* che fornisce un'interfaccia C ++ facile da configurare per *move_group*
- In **Python** - utilizzando il pacchetto moveit_commander (*approccio seguito in questo lavoro*)
- **Attraverso una GUI** - utilizzando il plug-in Motion Planning per Rviz (*il visualizzatore ROS*)

Robot Interface

move_group comunica con il robot per ottenere informazioni sullo stato corrente (posizioni dei giunti, ecc.), per ottenere nuvole di punti e altri dati sensoriali dai sensori del robot e per comunicare con i controller del robot.

Planning Scene

move_group utilizza il *Planning Scene Monitor* per creare una scena di pianificazione, che è una rappresentazione del mondo e dello stato attuale del robot. Lo stato del robot può includere qualsiasi oggetto trasportato dal robot che è considerato rigidamente collegato al robot.

Motion Planning

Il motion planning plugin

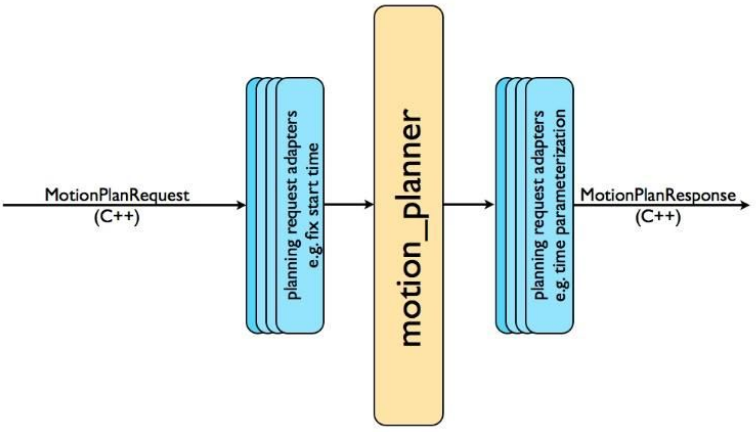
MOVEit! lavora con i pianificatori del movimento attraverso un'interfaccia a plugin. Ciò consente a MoveIt! di utilizzare diversi motion planner da più librerie e di essere facilmente estensibile. L'interfaccia con i pianificatori del movimento è realizzata attraverso un servizio ROS (offerto dal nodo *move_group*). I pianificatori di movimento predefiniti per move_group sono configurati usando OMPL..

La motion plan request

La motion plan request specifica chiaramente ciò che il pianificatore di movimento deve realizzare. In genere, si chiederà al pianificatore di spostare un braccio in una posizione diversa (nello spazio comune) o l'end effector in una nuova posa. Le collisioni vengono controllate per impostazione predefinita (comprese le auto collisioni). È anche possibile specificare dei vincoli per il motion planner: i vincoli incorporati forniti da MoveIt! sono vincoli cinematici:

- **Vincoli di posizione**: limitare la posizione di un giunto all'interno di una regione di spazio
- **Vincoli di orientamento:** limitano l'orientamento di un giunto che si trova all'interno dei limiti di rotazione, inclinazione o imbardata specificati
- **Vincoli di visibilità**: limitare un punto su un giunto che si trova all'interno del cono di visibilità per un particolare sensore
- **Vincoli comuni**: limitare un giunto tra due valori
- **Vincoli specificati dall'utente:** è inoltre possibile specificare i propri vincoli con un callback definito dall'utente.

Il risultato del motion plan

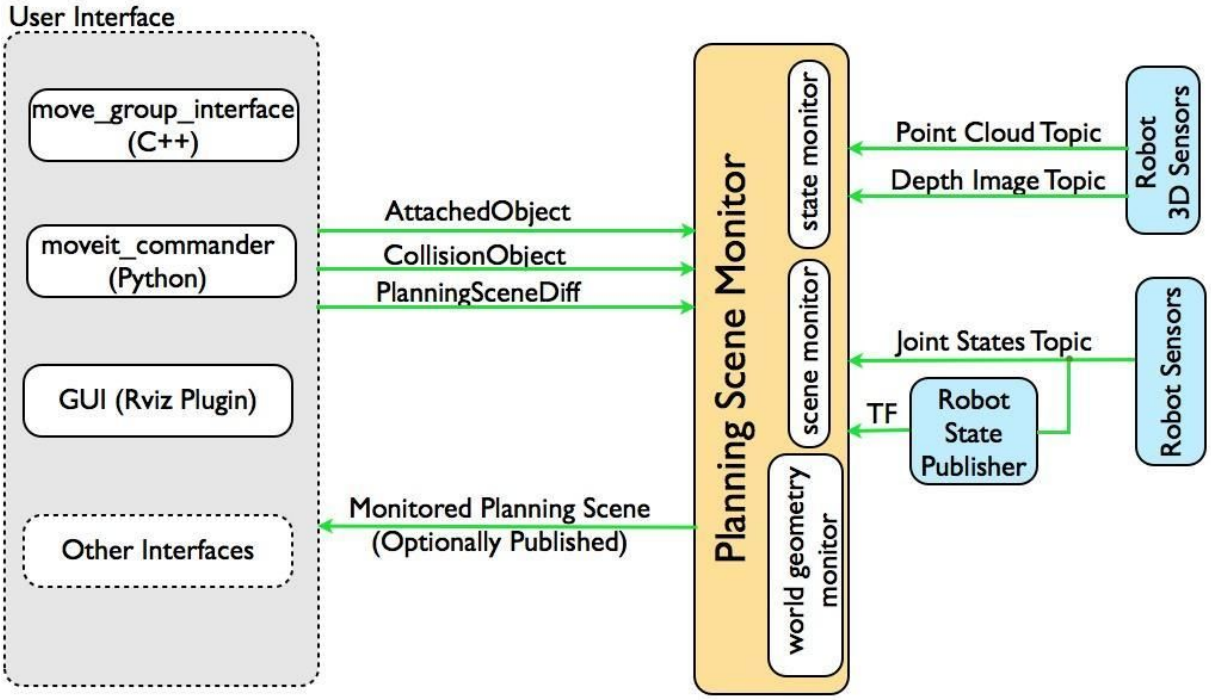


Viene a questo punto generata una traiettoria desiderata in risposta alla richiesta del motion plan. Questa traiettoria sposta il braccio (o qualsiasi altro gruppo di giunti) nella posizione desiderata. Si noti che il risultato uscente da *move_group* è una traiettoria e non solo un percorso: *move_group* utilizzerà le velocità e le accelerazioni massime desiderate (se specificate) per generare una traiettoria che obbedisce ai vincoli di velocità e accelerazione a livello di giunto.

OMPL

OMPL (Open Motion Planning Library) è una libreria di pianificazione del movimento open source che implementa principalmente pianificatori di movimento randomizzati. MOVEit! si integra direttamente con OMPL e utilizza i pianificatori del movimento da quella libreria come set primario / predefinito di pianificatori. I pianificatori di OMPL sono astratti; cioè OMPL non ha l'idea di robot. Invece, MoveIt! configura OMPL e fornisce il back-end per OMPL per lavorare con problemi in Robotica.

Planning Scene



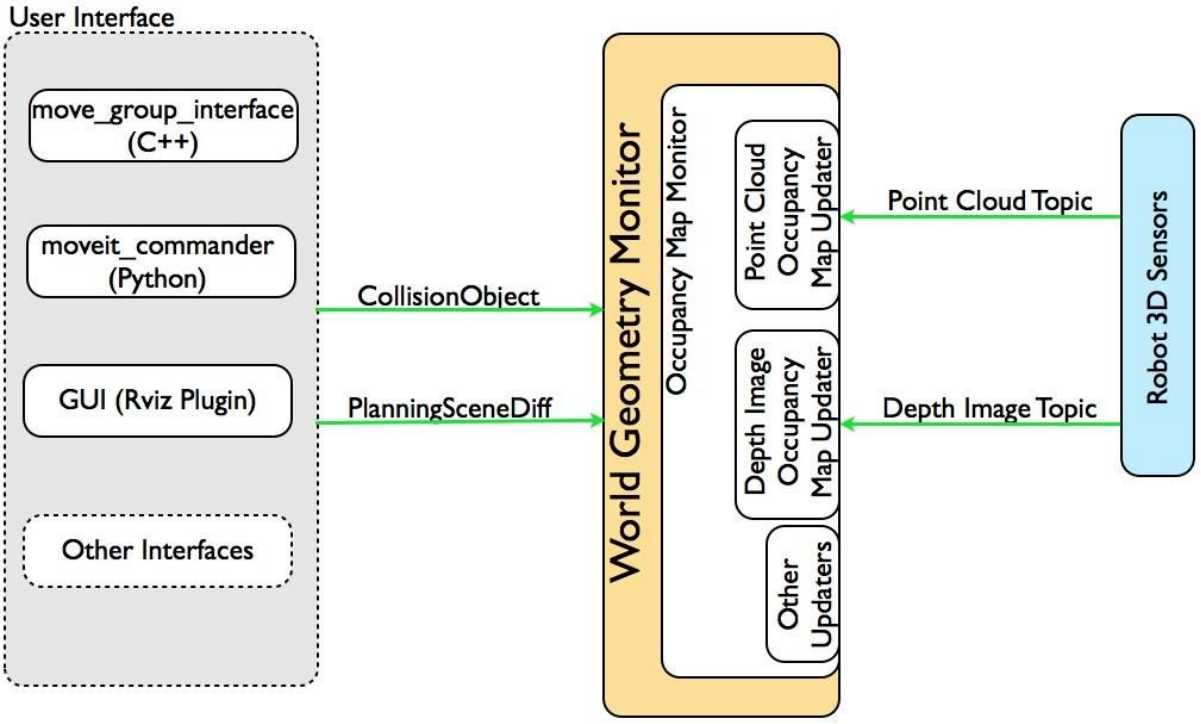
La planning scene, gestita dallo *scene monitor*, viene utilizzata per rappresentare il mondo intorno al robot e memorizza anche lo stato del robot stesso. Il *planning scene monitor* riceve:

- **Informazioni di stato**: nell'argomento joint_states
- **Informazioni sul sensore**: utilizzando il *world geometry monitor* (descritto di seguito)
- **Informazioni sulla geometria del mondo:** dall'input dell'utente sull'argomento planning_scene (*PlanningSceneDiff*).

World Geometry Monitor

Il world geometry monitor ricostruisce la geometria del mondo usando le informazioni provenienti dai sensori sul robot e dall'input dell'utente. Usa la *Occupancy Map* descritta sotto per costruire una rappresentazione 3D dell'ambiente attorno al robot e la aggiorna con le informazioni di *planning_scene* sugli oggetti.

3D perception



La 3D perception in MoveIt! è gestita dall' *Occupancy Map Monitor* che utilizza un'architettura a plugin per gestire diversi tipi di input del sensore, come mostrato nella figura di cui sopra. In particolare, MoveIt! ha un supporto integrato per la gestione di due tipi di input:

- **Nuvole di punti**: gestite dal plugin *Point Cloud Occupancy Map Updater*
- **Immagini di profondità**: gestite dal plugin *Depth Image Occupancy Map Updater*

Collision checking

Il Collision checking in MoveIt! è configurato all'interno di una *planning scene* utilizzando l'oggetto *CollisionWorld*: MoveIt! è configurato in modo che gli utenti non debbano mai preoccuparsi di come avviene il controllo delle collisioni. Il collision checking in MoveIt! viene realizzato principalmente utilizzando il pacchetto FCL (a Flexible Collision Library): la libreria principale di MoveIt! riguardo il collision checking.

Collision Objects

MOVEit! supporta il controllo delle collisioni per diversi tipi di oggetti, tra cui:

- **mesh**
- **Forme primitive**: ad es. scatole, cilindri, coni, sfere e piani
- **Octomap**: l'oggetto Octomap può essere utilizzato direttamente per il controllo delle collisioni

Allowed Collision Matrix (ACM)

Il controllo delle collisioni è un'operazione computazionalmente molto onerosa che spesso rappresenta circa il 90% del carico computazionale durante la pianificazione del movimento. La *Allowed Collision Matrix* (ACM) codifica un valore binario corrispondente alla necessità di verificare la collisione tra coppie di corpi (che potrebbero essere sul robot o nel mondo). Se il valore corrispondente a due corpi è impostato su 1 nell'ACM, questo specifica che non è necessario un controllo di collisione tra i due corpi. Ciò accadrebbe se, ad esempio, i due corpi fossero sempre così lontani da non entrare mai in collisione tra loro.

Installazione

Verranno ora descritti i passaggi per poter installare e rendere operativo l'ambiente di lavoro costituito da Moveit! (appena descritto) e da Rviz (Ros-visualization), che è analogo a Gazebo essendo un simulatore, e che deve essere utilizzato per poter usufruire delle funzionalità che Moveit! mette a disposizione.

1. Il primo passo da effettuare è quello di installare e buildare Moveit!. Eseguire in sequenza, in una shell:

```
$ cd ~/ros_ws/src
$ git clone https://github.com/ros-planning/moveit_robots.git
$ sudo apt-get update
$ sudo apt-get install ros-indigo-moveit-full
$ ./baxter.sh
$ catkin_make
```

2. Una volta scaricato Moveit! bisogna avviare Gazebo, quindi aprire un terminale ed eseguire i comandi:

```
$ cd ~/ros_ws
$ ./baxter.sh sim
$ roslaunch baxter_gazebo baxter_world.launch
```

   Ciò equivale, in sequenza, ad entrare nella cartella relativa al nostro workspace di ROS, a "dichiarare" di voler lavorare con il baxter in simulazione ed infine a caricare, tramite Gazebo, il modello del robot. Questa operazione è necessaria in quanto, per poter funzionare, Moveit! e Rviz hanno bisogno del modello del robot e delle info relative ad esso: sarà proprio Gazebo a pubblicarle e renderle disponibili.

3. Aprire ora un nuovo terminale, lasciando aperto quello del punto 2, ed eseguire i comandi:

```
$ cd ~/ros_ws
$ ./baxter.sh sim
$ rosrun baxter_tools enable_robot.py -e
```

   I primi due comandi sono quelli di default da eseguire in ogni nuovo terminale aperto mentre il terzo serve ad abilitare il robot (abilitare la sua capacità di muoversi).

4. Nello stesso terminale del punto 3, una volta abilitato il robot, avviare il joint trajectory action server. Eseguire quindi:

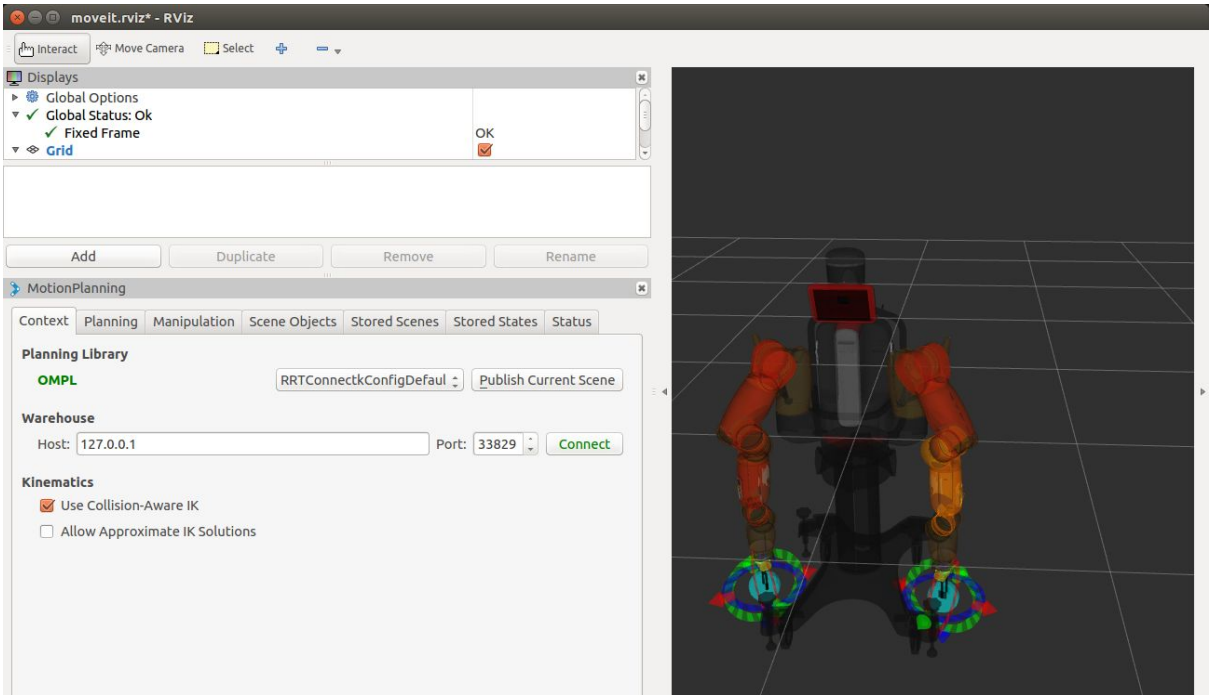
```
$ rosrun baxter_interface joint_trajectory_action_server.py
```

Questo è un processo che deve rimanere attivo per tutto il tempo in cui si utilizza Moveit!/Rviz e quindi anche questo terminale dovrà rimanere aperto, allo stesso modo di quello contenente Gazebo.

5. Aprire un nuovo terminale (ricordandosi di lasciare aperti ed in run gli altri due) ed eseguire i comandi:

```
$ cd ~/ros_ws
$ ./baxter.sh sim
$ roslaunch baxter_moveit_config baxter_grippers.launch
```

Fatto ciò dovrebbe avviarsi Rviz (e conseguentemente Moveit!) e l'utente si troverà i fronte ad una schermata analoga alla seguente, dove il Baxter in Rviz si troverà nella stessa posizione del Baxter in Gazebo. Anche in questo caso, questo terminale dovrà rimanere aperto ed in esecuzione per tutta la durata dei lavori.



Sebbene possa sembrare “strano” avere due modelli del robot eseguiti in simultanea in simulazione, ciò è dovuto ai seguenti fattori:

- Non si dispone del Baxter “reale”, per cui è necessario averne una versione simulata, attraverso Gazebo;
- Gazebo non offre alcuna funzionalità (perlomeno di default) per quanto riguarda il motion planning;
- Moveit! per poter funzionare ha bisogno di Rviz;
- Rviz deve ricevere il modello del Baxter da qualche fonte: in questo caso la fonte pubblicante tali dati è proprio il processo di Gazebo;
- Moveit! deve essere utilizzato essendo uno strumento molto potente per quanto riguarda il motion planning e, nello specifico caso di questo laboratorio, ha permesso la risoluzione di un problema di obstacle avoidance.

Ora, se si sono eseguiti correttamente tutti i passaggi, si può iniziare ad utilizzare Moveit! e le sue feature tramite interfaccia grafica in Rviz: si può muovere il robot; cambiare libreria per il motion planning; inserire e modificare ostacoli e molto altro. Per un utilizzo programmatico però, bisognerà ricorrere obbligatoriamente alla stesura di codice.

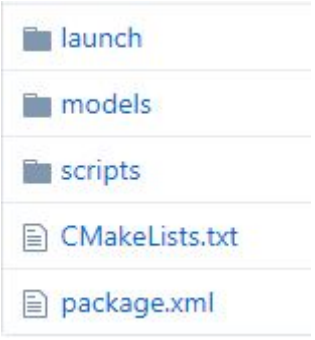
Presentazione della Robot Application

In questa sezione verrà illustrato l'applicativo realizzato durante il progetto.
Di seguito è presente il link alla repository Github dov'è presente il codice sorgente.

Link: https://github.com/GiuseppeCannata/Baxter

PACKAGE

il package si presenta in questo modo:



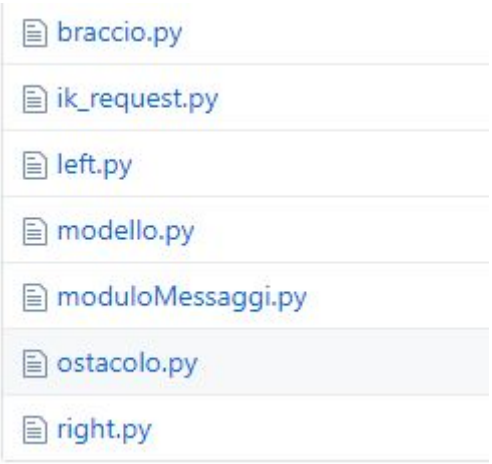
Le directory sono quindi le seguenti:

- **launch** contiene i file *.launch*. Per il nostro progetto non sono stati utilizzati per cui la cartella è vuota;
- **models** contiene i file per la rappresentazione in gazebo degli oggetti da noi creati (i.e. il tavolo ed il cubetto rosso per il pick and place);
- **scripts** contiene i nodi sviluppati per il progetto.

Oltre alle directory di cui sopra, sono presenti i file *CMakeLists.txt* e *package.xml*. Questi permettono, in estrema sintesi, a ROS di identificare tutti questi elementi (directories + *CMakeLists.txt* + *package.xml*) come un package e vengono generati automaticamente alla creazione dello stesso.

SCRIPTS

Il contenuto della cartella scripts è riportato nel seguito:



I seguenti file *.py* sono i nodi del nostro applicativo,

Spieghiamo ora in dettaglio ognuno di essi:

- **braccio.py**

  E' una classe la cui istanza rappresenta un braccio del Baxter. In particolare i metodi che espone come interfaccia sono:

move_with_joints	Consente di effettuare lo spostamento del braccio considerando gli angoli di giunto.
move_with_pose	Consente di effettuare lo spostamento del braccio considerando la pose (*coordinate nello spazio cartesiano*). Questo metodo richiamerà *ik_request* per poter trasformare le coordinate di pose in coordinate nello spazio di giunti.
get_endpoint_pose	Restituisce la posizione in coordinate cartesiane del gripper.
gripper_open	Consente di aprire il gripper del braccio
gripper_close	Consente di chiudere il gripper del braccio

  Le istanze di questa classe verranno create nei nodi *left.py* e *right.py* ad indicare rispettivamente il braccio sinistro e destro del baxter.

- **ik_request.py**

  È un modulo python al cui interno è contenuta la funzione ik_request.
  Quest'ultima permette di invocare il servizio di cinematica inversa (*del Baxter*).
  Se il calcolo della cinematica inversa andrà a buon fine allora ik_request restituirà al chiamante gli angoli di giunto, altrimenti avremo un log di errore.

- **left.py**

  È uno nodo ROS che:

  1. Permette al braccio sinistro di effettuare il pick del blocchetto presente sul tavolo e di cui è specificata la posizione;
  2. Comunica al braccio destro di aver effettuato il pick;
  3. Al momento opportuno apre il gripper e rilascia l oggetto.

  Nota: Nel sorgente troviamo dei waypoints. Quest'ultime sono rappresentate nella sezione: *posizioni bracci*

- **right.py**

  È uno nodo ROS che:

  1. Attende che il braccio sinistro effettui il pick
  2. Una volta effettuato il pick, permette al braccio sinistro di spostarsi verso la posizione ricevuta dal braccio sinistro
  3. Effettua il pick e lo comunica al braccio sinistro

  Nota: Nel sorgente troviamo dei waypoints. Quest'ultime sono rappresentate nella sezione: *posizioni bracci*

- **modello.py**

  È uno script python che consente di caricare in gazebo i modelli sia del tavolo che del blocchetto.

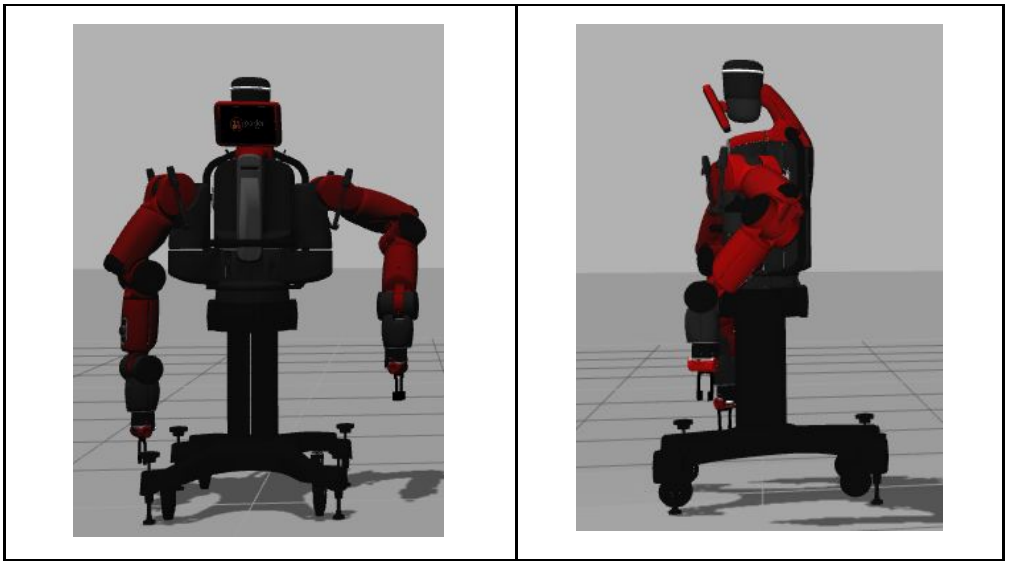
- **ostacolo.py**

  È uno script python che consente di caricare in RViz i vari ostacoli definit

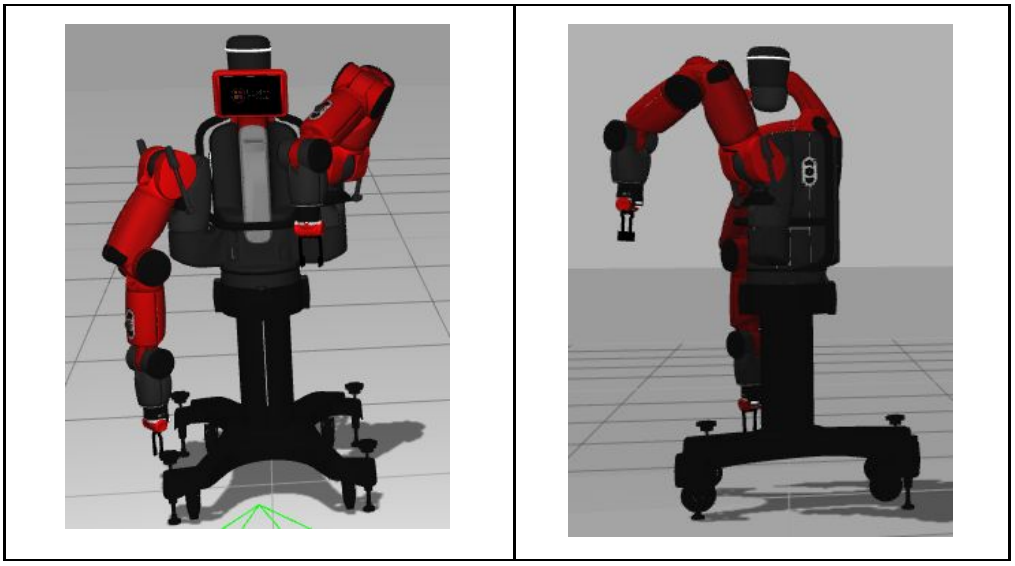
Posizioni dei bracci

- **Braccio sinistro**

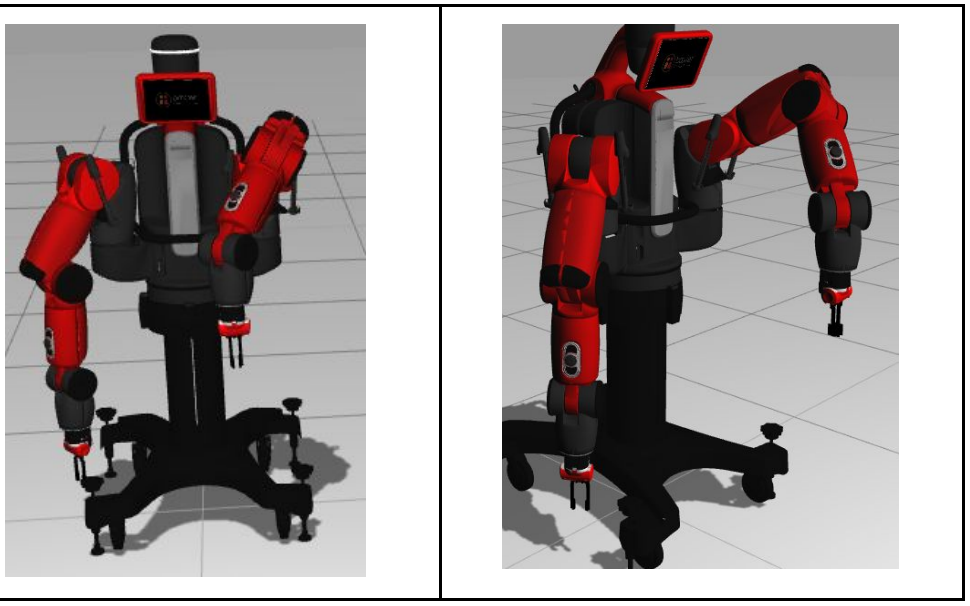
- Primo punto di arrivo



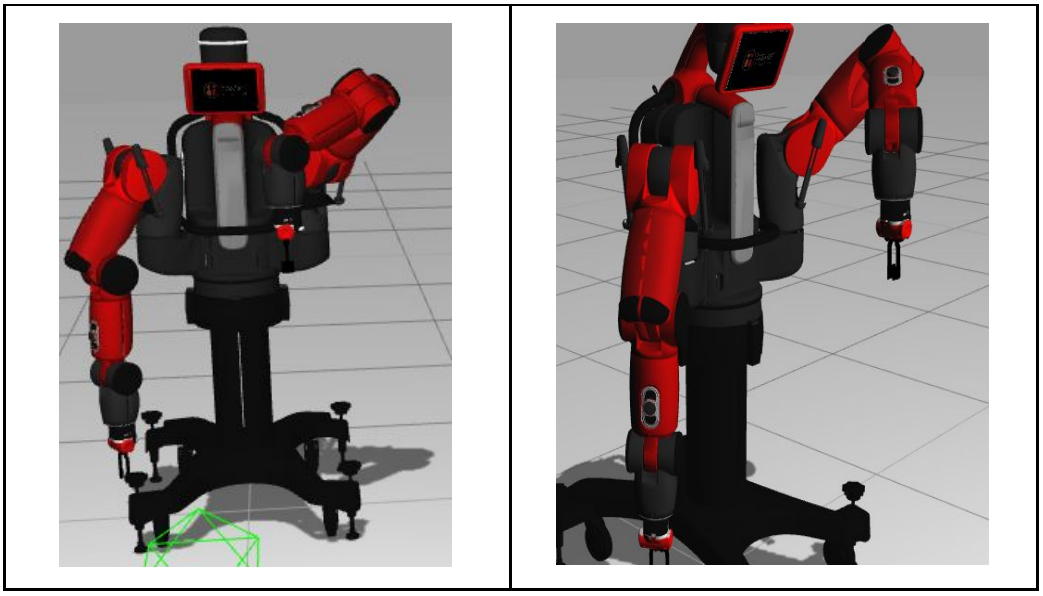
- Secondo punto di arrivo



- Terzo punto di arrivo

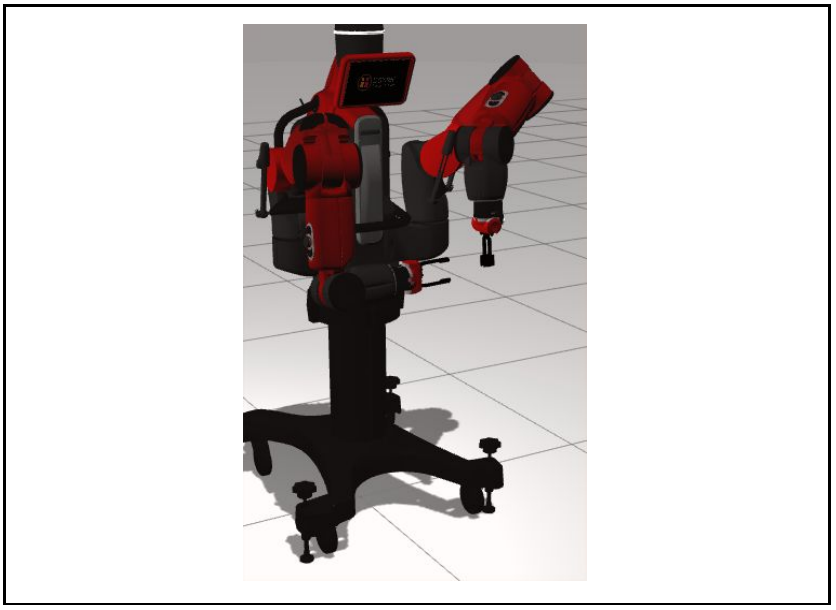


- Quarto punto di arrivo

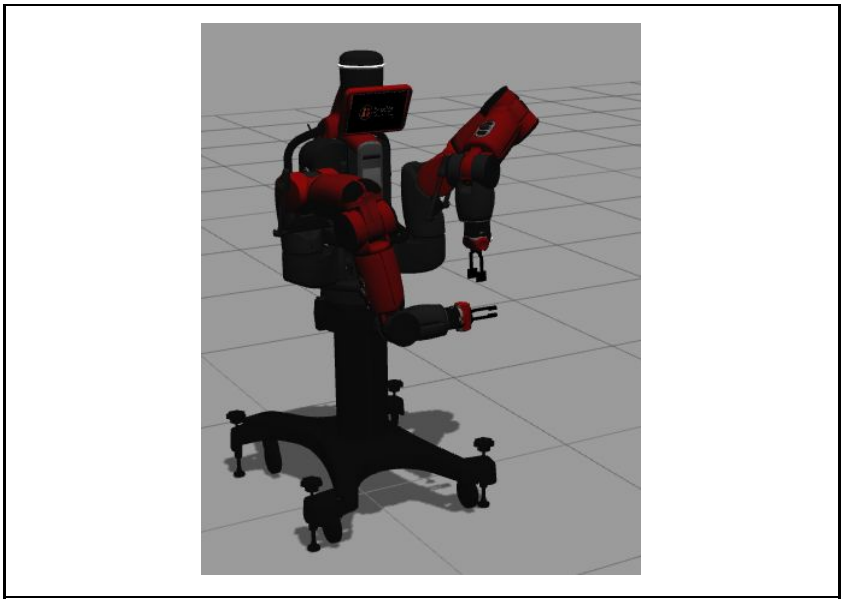


- **Braccio destro**

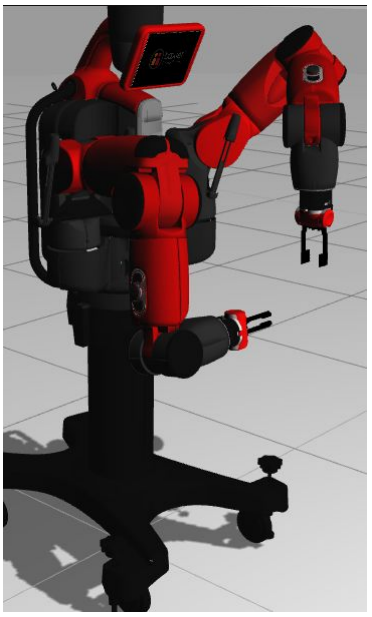
- Primo punto di arrivo



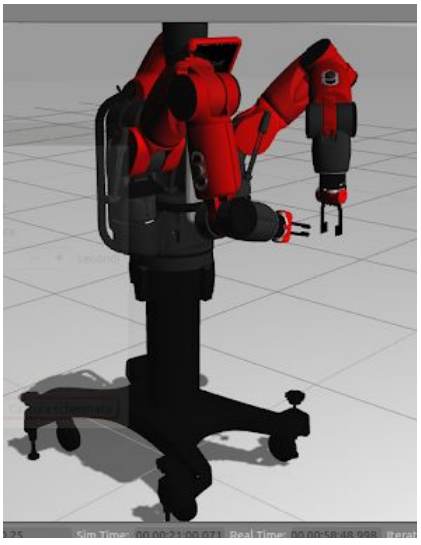
- Secondo punto di arrivo



- Terzo punto di arrivo



- Quarto punto di arrivo



Esecuzione del package

In questa sezione verrà spiegato come avviare la robot application.
In ogni terminale che verrà aperto, per potersi interfacciare con il il ROSmaster, è bene avviare il WorkSpace.
Per farlo digitare in ogni terminale:

```
cd  ~/ros_ws
./baxter.sh sim
```

1. Scaricare la cartella al link: https://github.com/GiuseppeCannata/Baxter.
   Una volta scaricata spostarla nella cartella src del proprio WorkSpace.
   Dopodichè ricompilare il Workspace facendo catkin_make.
2. Rendere i file eseguibili;
   Per farlo eseguire:
   a. roscd baxter_project
   b. cd scripts
   c. sudo chmod +x ./*.py
3. Aprire un terminale, avviare Gazebo ed attendere il termine del caricamento

```
roslaunch baxter_gazebo baxter_world.launch
```

4. Aprire un ulteriore terminale e attivare il robot eseguendo il seguente comando:

```
rosrun baxter_tools enable_robot.py -e
```

N.B.: Attendere il log del programma: Robot Enabled.
Nello stesso terminale eseguire il comando:

```
rosrun baxter_interface joint_trajectory_action_server.py
```

5. Aprire un ulteriore terminale ed avviare RViz

```
roslaunch baxter_moveit_config baxter_grippers.launch
```

6. Aprire un ulteriore terminale ed eseguire:

```
rosrun baxter_project right.py
```

7. Aprire un ulteriore terminale ed eseguire:

```
rosrun baxter_project left.py
```

Conclusioni e sviluppi futuri

Riassumendo, in questa attività di laboratorio:

1. è stato conosciuto il framework ROS, la sua struttura ed il suo funzionamento;
2. si è installato il simulatore Gazebo e ci si è interfacciati con il Baxter con dei primi programmi di esempio;
3. si è eseguito lo scambio dell'oggetto (i.e. il cubetto rosso) lavorando esclusivamente con il framework ROS e con Gazebo;
4. si è introdotto il vincolo di obstacle avoidance durante i movimenti del robot;
5. ci si è affidati, per la risoluzione del problema di cui sopra, al framework Moveit!;
6. si sono utilizzate ed integrate nel programma di partenza tutte le risorse software che Moveit! (ed Rviz) mettono a disposizione;
7. si è completato il codice e si sono svolte diverse simulazioni per il testing di quanto prodotto.

Per quanto riguarda gli sviluppi futuri, quelli reputati i più rilevanti sono i seguenti:

- testare la robot application prodotta su robot reale;
- continuare ad esplorare il framework Moveit!, per riuscire a comprendere fino in fondo la validità e la potenza dello strumento;
- implementare nel codice già prodotto un meccanismo per l'aggiunta degli ostacoli non al momento della scrittura del programma ma dinamicamente, riuscendo ad interfacciare il framework Moveit! con il sistema di visione del robot Baxter.