Control of a redundant simulated robotic arm by using a ROS interface

Gabriele Sisinna

Tutor: Egidio Falotico, Francesco Iori

8 gennaio 2021

Indice

Capitolo 1

Introduzione: NRP

Durante questo lab training è stato approfondito il controllo di un manipolatore robotico a 7 GDL (Kuka iiwa) utilizzando un'interfaccia basata su ROS e Python. Il tutto è stato implementato e testato con il simulatore (Gazebo) integrato nella Neurorobotics Platform, un framework per la simulazione di esperimenti di neurorobotica.

1.1 Ubuntu

La distribuzione linux utilizzata per l'installazione della NRP in locale è "Ubuntu Linux 18.04 (Bionic Beaver)". Inizialmente la distribuzione è stata installata in una macchina virtuale, ma a causa del carico computazionale richiesto dalle simulazioni è stato scelto di installare la distribuzione come OS principale su una macchina con SSD e processore Intel Core i7.

1.2 Download files e installazione

Per l'installazione della Neurorobotics Platform viene seguita la guida ufficiale. L'installazione richiede diverse ore di tempo in quanto è necessario compilare dai sorgenti ogni singola componente della piattaforma. Diversi errori sono stati incontrati durante questa fase, soprattutto durante la fase di build e per la compatibilità con alcune librerie (deprecated). Il tutto sembra imputabile alla migrazione della NRP verso Ubuntu 20.04 e ad alcuni disservizi dei server di BitBucket durante il download.

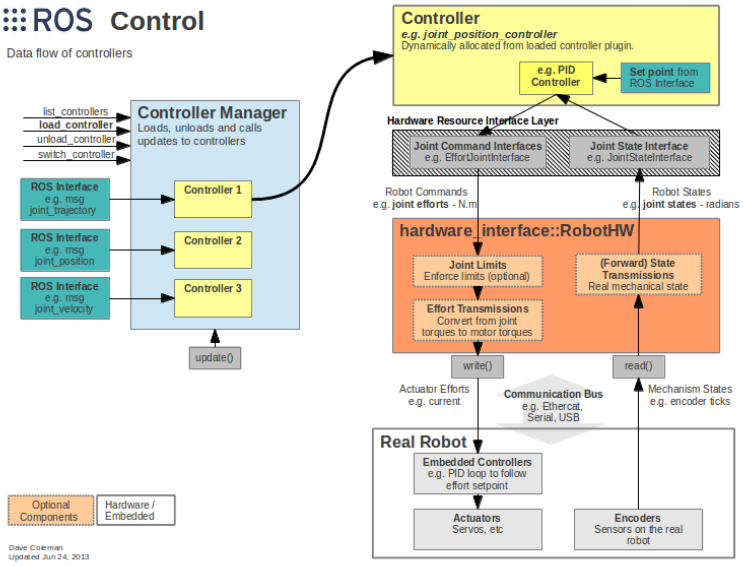
Capitolo 2

ROS

Qualsiasi comunicazione con il robot simulato e il controllo della simulazione stessa avviene attraverso il Robot Operating System (ROS), che è nativamente integrato con Gazebo. ROS è un middleware ampiamente utilizzato nella comunità della robotica e fornisce all'utente API C++ e Python.

2.1 ROS Control

I pacchetti ros control sono una riscrittura dei pacchetti pr2 mechanism per rendere i controller generici per tutti i robot oltre al PR2. Di seguito uno schema del funzionamento generale:



Per la generazione della traiettoria è stato utilizzato un controllore che accetta in ingresso la posizione desiderata per il giunto. Questa viene inviata al controllore PID che si occupa di calcolare il momento necessario per il raggiungimento della posizione desiderata. Tutti i parametri PID sono inseriti all'interno del file di configurazione "torque_controllers.yaml". Questo tipo di controllore accetta in ingresso messaggi del tipo "JointTrajectory", all'interno dei quali è possibile specificare una serie di waypoints per la traiettoria desiderata.

2.2 Rospy

Rospy è una libreria client Python per ROS. L'API client rospy consente ai programmatori Python di interfacciarsi rapidamente con topic, servizi e parametri ROS.

2.3 Msg Custom

Per comunicare la posizione e l'orientamento dell'end-effector attraverso i vari topic è stato creato un messaggio custom "end_effector.msg". Quest'ultimo è costituito da 3 campi Float64 (vettoriali):

- position/orientation
- velocity
- time

2.4 Launch File

Per avviare contemporaneamente il nodo della cinematica inversa (listener.py) e quello che trasmette la posizione desiderata per l'end-effector (talker.py) è stato creato un unico launch file (listener_talker.launch).

Capitolo 3

Librerie aggiuntive

3.1 RBDL

RBDL è una libreria C ++ altamente efficiente che contiene alcuni algoritmi di dinamica del corpo rigido essenziali. Contiene inoltre codice per Jacobiani, cinematica diretta e inversa, gestione di vincoli esterni come contatti e collisioni e modelli a closed loop. Dopo aver importato il file URDF, è possibile calcolare lo Jacobiano analitico mediante la funzione:

rbdl.CalcPointJacobian6D(model, q, 7, LOCAL_OFFSET, J)

Per maggiori informazioni si rimanda alla documentazione ufficiale della libreria RBDL (vedi bibliografia).

3.2 Quaternion

Questo pacchetto crea un tipo quaternione in python e consente inoltre a numpy di creare e manipolare array di quaternioni. Sono disponibili le normali operazioni algebriche (addizione e moltiplicazione), insieme a numerose proprietà come norma e vari tipi di misure di distanza tra due quaternioni. Ci sono anche funzioni aggiuntive come l'interpolazione "squad" e "slerp" e conversioni da e verso rappresentazioni di rotazioni con angolo dell'asse, matrice e angolo di Eulero. Esempio di utilizzo dei quaternioni per il calcolo della SLERP relativa all'orientamento dell'end-effector:

```python
import numpy as np
from numpy.random import random
from numpy import sin
from numpy import cos
from numpy import pi
import quaternion

def mySlerp(t,q0,qf):
    teta = np.linalg.norm(quaternion.as_rotation_vector( (q0.conjugate() * qf )))
    t0 = 0
    tf = 10
    dt = (t-t0)/(tf-t0)
    quat_slerp = (sin(teta/2*(1-dt))*q0 + sin(teta/2*dt)*qf)/sin(teta/2)
    return quat_slerp

q0=np.quaternion(1,0,0,0)
qf=np.quaternion(0,0,0,1)

q = mySlerp(10,q0,qf)
print(q)
```

$$\boldsymbol{q}_{des}(t,\boldsymbol{q}_0,\boldsymbol{q}_f) = \frac{sin(\frac{\theta}{2}(1-\tilde{t}))\boldsymbol{q}_0 + sin(\frac{\theta}{2}\tilde{t})\boldsymbol{q}_f}{sin(\frac{\theta}{2})}$$

$$\theta = \texttt{np.linalg.norm(as\_rotation\_vector(} (\boldsymbol{q}_0^* \otimes \boldsymbol{q}_f)\texttt{))}$$

$$\tilde{t} = \frac{t-t_0}{t_f-t_0} \in [0,1]$$

$$\dot{\boldsymbol{q}}_{des}(t) = \frac{q_{des}(t_k) - q_{des}(t_{k-1})}{dt}$$

$$\boldsymbol{\omega}_{des}(t) = -2\,\boldsymbol{q}^*_{des}(t)\,\dot{\boldsymbol{q}}_{des}(t)$$

Per fornire la corretta orientazione all'end-effector istante per istante è stata utilizzata un'interpolazione SLERP fra il quaternione relativo alla posa iniziale e il quaternione per la posa finale desiderata:

Capitolo 4

Closed Loop Inverse Kinematics (CLIK)

4.1 Algoritmo

A causa della ridondanza del manipolatore è stato necessario utilizzare algoritmi numerici per trovare delle soluzioni alla cinematica inversa. Di seguito verranno riportate le relazioni principali utilizzate per l'implementazione dell'algoritmo CLIK:

$$\dot{x} = J_A(q)\dot{q} \tag{4.1}$$

Inversione open loop:

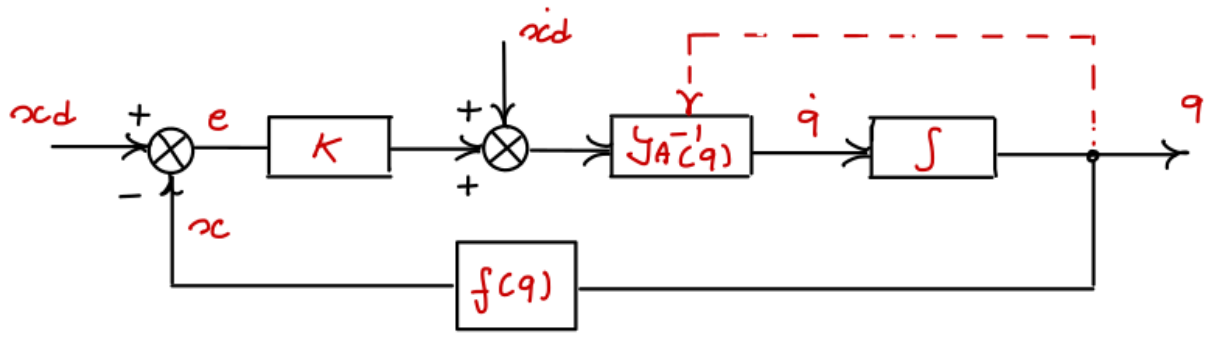
$$\dot{q} = J_A(q)^{-1}\dot{x} \tag{4.2}$$

Legge di inversione closed loop:

$$\dot{q} = J_A^{-1}(q)(\dot{x_d} + Ke) \tag{4.3}$$

Eulero in avanti per via numerica:

$$q_{k+1} = q_k + hJ_A^{-1}(q_k)(\dot{x_d}(t_k) + Ke_k) \tag{4.4}$$



4.2 Stabilità

Per avere stabilità nel passaggio al tempo discreto è necessario che sia valida la seguente condizione:

$$K_i < 2/h \tag{4.5}$$

Ciò assicura la convergenza dell'errore a zero all'aumentare del tempo t (dinamica esponenziale asintoticamente stabile).

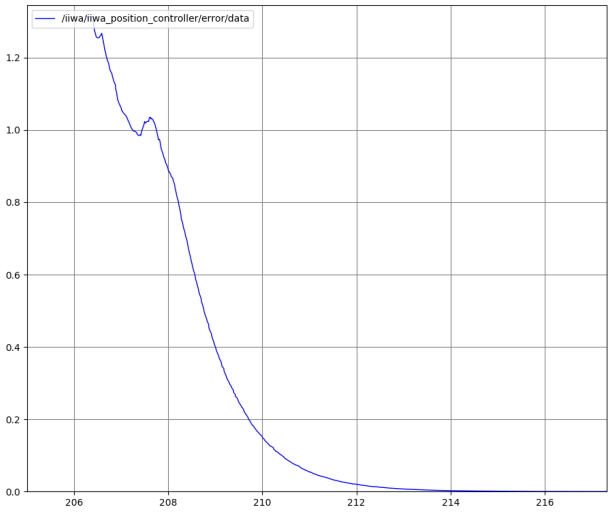
4.3 Errore

L'errore di feedback è costituito da una componente relativa all'orientamento e una componente lineare. L'ordine angular-linear verrà sempre mantenuto in quanto RBDL calcola lo Jacobiano inserendo per prime le componenti relative all'orientamento. Inizialmente sono stati utilizzati gli angoli di eulero per indicare l'orientazione dell'end-effector, successivamente sono stati implementati i quaternioni con le relative trasformazioni per calcolare l'errore di orientamento:

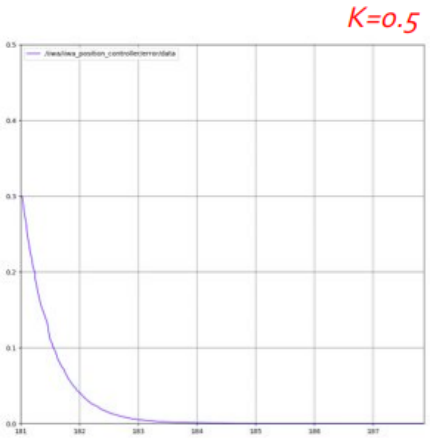
- q : quaternione di orientazione dell'end-effector
- q_{des} : quaternione di orientazione desiderata per l'end-effector

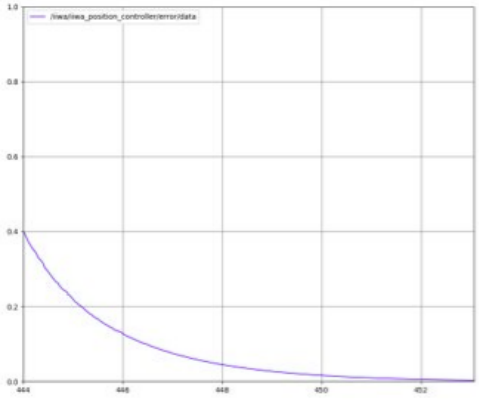
L'errore di rotazione è dato dalla parte vettoriale del quaternione: $e_q = q_{des} \circledast q\star$

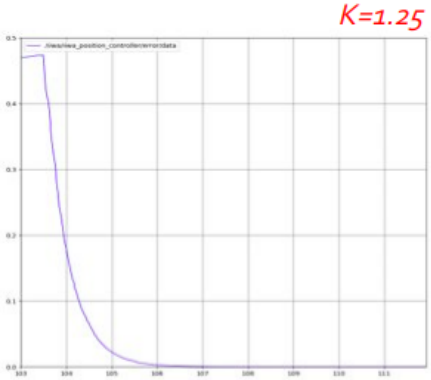
Plottando la norma dell'errore totale è possibile vedere come il controllore riesca a raggiungere la posa desiderata in funzione del tempo. I seguenti grafici sono stati generati tramite rqt-plot:



Modulando un fattore di velocità K è possibile cambiare la velocità con la quale l'end-effector dovrebbe seguire la traiettoria desiderata:







Capitolo 5

Simulazioni e test

Sono state eseguite diverse simulazioni per task di pick and place. Un controllo basato sulla dinamica del manipolatore migliorerebbe sensibilmente la stabilità del sistema nella sua interezza, mentre i risultati presentati fanno riferimento all'utilizzo di controllori separati per ogni singolo giunto. In allegato al presente report è possibile visualizzare un video dimostrativo dell'esecuzione di una traiettoria per task di pick and place.

Capitolo 6

Codice sorgente

Tutto il codice sorgente è disponibile liberamente presso il seguente indirizzo:

https://github.com/gsisinna/labtraining

6.1 Struttura

I nodi principali sono due: "listener.py" e "talker.py". Il listener è il nodo che si occupa di calcolare la cinematica inversa ogniqualvolta viene aggiornato lo stato attuale dei giunti del manipolatore e viene ricevuto un nuovo messaggio per la posizione dell'end-effector. Il talker è il nodo che invia la posizione desiderata per l'end-effector. Il rate di controllo è stato impostato a 100Hz.

Capitolo 7

Conclusioni

Sono state approfondite tematiche relative alla meccanica dei robot, controllo di manipolatori ridondanti e programmazione in Python. Le principali difficoltà riscontrate sono state dovute alla compatibilità fra librerie, utilizzo di controllori poco documentati all'interno del pacchetto ros-control e la programmazione del nodo della cinematica inversa in Python. Anche l'utilizzo dei quaternioni e le diverse convenzioni per gli angoli di Eulero hanno portato a diversi errori durante il debugging degli script relativi ai nodi. Nel complesso, l'utilizzo di algoritmi più robusti, come quelli del secondo ordine, o il controllo basato su dinamica del manipolatore, meglio si prestano ad un task del genere.

Concludo ringraziando i tutor che mi hanno seguito durante questo periodo e che hanno reso possibile per me apprendere nuovi concetti e applicarli immediatamente con un progetto così pratico.

Capitolo 8

Bibliografia

- Dispense del corso "Meccanica dei Robot" (M.Gabiccini)
- Gazebo Tutorials - http://gazebosim.org/tutorials
- Documentazione NRP - https://bitbucket.org/hbpneurorobotics/neurorobotics-platform/src/master/
- Ros-control Offical Wiki
- Numpy-quaternion - https://pypi.org/project/numpy-quaternion/
- RBDL - https://rbdl.github.io/
- Robotics (Siciliano, B., Sciavicco, L., Villani, L., Oriolo, G.)