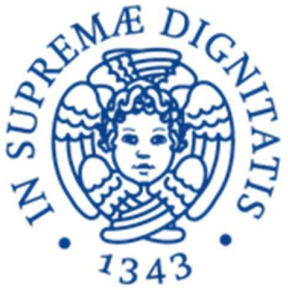


UNIVERSITÀ DI PISA

“SHELL” ROBOT

Simulazione, cinematica e aspetti computazionali

Luca ricci

INDICE

1. Shell robot
2. Descrizione mediante urdf
3. Calcolo dinematica diretta
    1. Parte superiore
    2. Parte inferiore
    3. Costruzione cinematica
4. Cinematica inversa
5. Cinematica del parallelo
6. Cinematica computazionale
    1. Classe joint
    2. Classe serial
    3. Classe parallel
    4. Libreria
7. Algoritmo L-M
8. Speed up del codice
9. Generazione traiettorie
10. Assegnamento posizioni
11. Aspetti simulativi e di controllo
    a. Simulazione
    b. Controllo
12. Ros
13. Interfaccia grafica e comandi
14. Struttura del catkin
15. riferimenti

"Shell" robot

"Shell"

Il robot da noi simulato, denominato "shell" per la forma a guscio che lo contraddistingue è un robot parallelo, in particolare caratterizzato come un esapode, in grado non solo di camminare come un esapode ma con una modalità di movimento aggiuntiva, esso è infatti in grado di rotolare e avanzare come una palla.

Descrizione mediante URDF

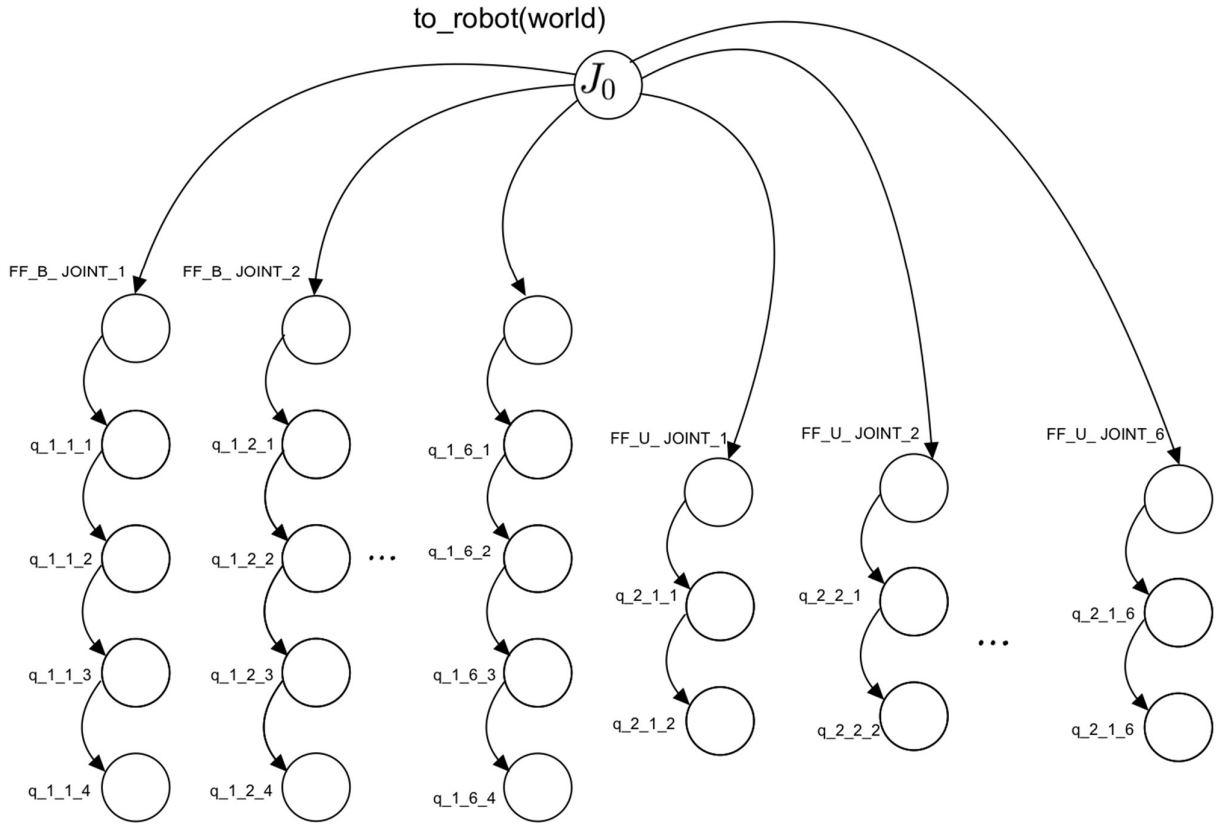
La descrizione cinematica, dinamica, visiva del robot è effettuata mediante un il file urdf *"shell_robot.urdf.xacro"* che contiene al suo interno l'applicazione dei files:

- *variables.urdf.xacro*
- *shell_robot_bottom_leg.urdf.xacro*
- *shell_robot_upper_leg.urdf.xacro*
- *chassis.urdf.xacro*

Questi file descrivono completamente tutto il robot , sia a livello cinematico che dinamico, ma anche come parte visiva.

La struttura del robot è organizzata in un diagramma ad albero dove ogni giunto "joint" collega l'elemento "*link*" precedente(*parent*) all'elemento "*link*" successivo (*child*).

Internamente alla struttura joint è possibile imporre limiti di coppia, velocità e definire le caratteristiche dinamiche dell'attuatore.
Internamente ai link è possibile invece osservare le caratteristiche dinamiche, visive, collisioni e prorpietà del materiale.
La struttura dei joint creata nel caso del robot shell è qui riportata:



La numerazione è in questo caso significativa: il primo numero dopo q indica il parallelo a cui ci si riferisce , ossia superiore(2) e inferiore(1), il secondo numero indica il numero relativo alla catena seriale che stiamo considerando,quindi è il numero associato alla gamba, il terzo numero è invece relativo al giunto.

La parametrizzazione in questo caso è di fatto un caso particolare di parametrizzazione local POE. La parametrizzazione è quindi in questo caso più generale rispetto al caso di parametrizzazione mediante DH.
La descrizione della trasformazione tra il sistema di riferimento i-esimo e il successivo avvieme infatti mediante 6 coordinate (x, y, z, y, p, r) ed esse compongono di fatto una prima traslazione del sistema di riferimento mediante le coordinate x, y, z e successivamente una rotazione sequenziale y, p, r.
In questo caso, non è necessario che vengano seguite le convenzioni di DH e non è necessario quindi che l'asse di rotazione del giunto coincida con l'asse z locale.
La specifica sull'asse di rotazione del giunto (locale) è specificata nel campo "axis" all'interno dei singoli elementi "joint".

- $(x, y, z) \rightarrow$ indica come l'origine del sistema di riferimento S_{i+1} è posizionata rispetto a S_i quando la variabile di giunto qi è nulla.
- $(y, p, r) \rightarrow$ indica come il sistema di riferimento S_{i+1} è orientato rispetto a S_i (usando una trasformazione (z, y, x) quando la variabile di giunto qi è nulla.

Viene qui riportata la tabella dei valori (x, y, z, y, p, r) che descrivono le trasformazioni relative ai sistemi di riferimento utilizzati nell'urdf nel caso di una singola gamba (si itera su $k = 1 \ldots 6$ per ottenere le altre):

Gambe inferiori:

$joint$	X	Y	Z	R	P	Y	$Axis$	$type$
$J0 \rightarrow J1$	0	0	-0.038	0	0	$\pi/3 * K$	Na	$fixed$
$J1 \rightarrow J2$	0	0	0	0	0	0	Z	$revolute$
$J1 \rightarrow J3$	0.066	0	0	0	0	0	Na	$fixed$
$J2 \rightarrow J4$	0.120	0	0	0	0	0	Z	$revolute$
$J3 \rightarrow J5$	0.038	0.025	-0.2375	0	$\pi/2 - \pi/3$	0	Y	$revolute$
$J6 \rightarrow J6$	0.07	0	0	$-\pi/2$	$\pi/3$	0	Z	$revolute$

Gambe superiori:

$joint$	X	Y	Z	R	P	Y	$Axis$	$type$
$J0 \rightarrow J1$	0	0	0.045	0	0	$\pi/3 * K$	Na	$fixed$
$J1 \rightarrow J2$	0.08	0	0.018	0	$-\pi/10$	0	Y	$revolute$
$J1 \rightarrow J3$	0.085	0	0	0	$-\pi/2$	0	Y	$revolute$

Calcolo cinematica diretta

Per la costruzione della cinematica diretta siamo partiti da una caratterizzazione mediante DH del robot, costruendo quindi una parametrizzazione generale della struttura. Il robot si caratterizza infatti in 2 parti distinte entrambe parallele:

- *Parte superiore*
- *Parte inferiore*

La parametrizzazione viene effettuata nel programma di calcolo mediante DH e quindi dalle matrici di trasformazione che vanno a definire una chiara interpretazione fisica.

La generica matrice di trasformazione dal giunto i-1 al giunto i-esimo viene definita come segue:

$$T_{i-1,i} = T_{tr_{Z_{i-1}}}(d_i) \cdot T_{rot_{Z_{i-1}}}(\theta_i) \cdot T_{tr_{x_{i-1}}}(r_i) \cdot T_{rot_{x_{i-1}}}(\alpha_i)$$

$$T_{i-1,i} = \begin{bmatrix} c\theta_i & s\theta_i * c\alpha_i & s\theta_i * s\alpha_i & r_i * c\theta_i \\ s\theta_i & c\theta_i * s\alpha_i & -c\theta_i * s\alpha_i & r_i * s\theta_i \\ 0 & s & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad matrice\ DH\ i-esima$$

Dove α , r , θ , d sono i parametri di DH relativi alla trasformazione. Nella matrice sarà presente un parametro attivo che andrà a definire la variabile di giunto, ossia la rotazione attorno all'azze Z di ogni giunto , ossia θ.

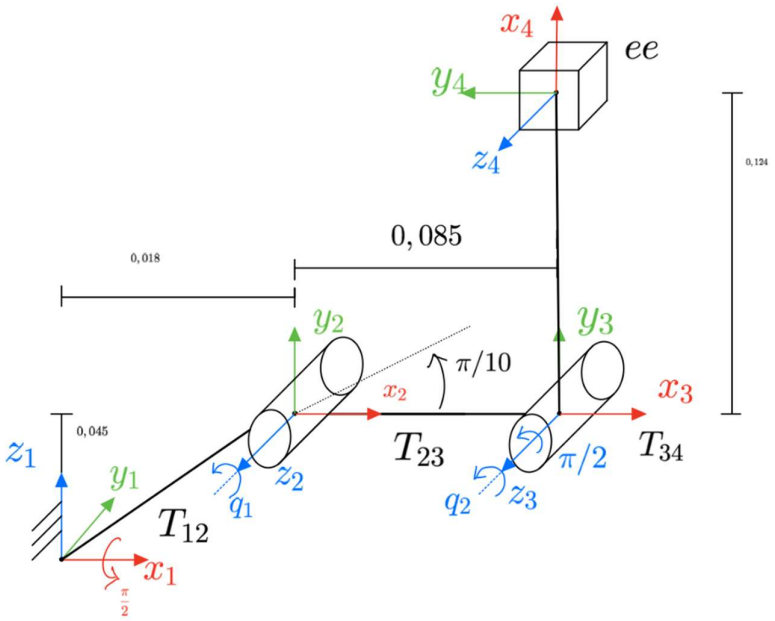
1 PARTE SUPERIORE

La parte superiore è composta da 6 seriali che definiscono il parallelo superiore , si tratta quindi di 6 RR planari distribuiti attorno al centro del robot sfasati di pi/3.

Facendo riferimento alla figura qui riportata è possibile dare una caratterizzazione mediante DH e costruire quindi una parametrizzazione del singolo seriale.

Una volta costruita la cinematica del seriale sarà sufficiente ruotarlo per costruire il parallelo superiore in quanto il problema è caratterizzato da una forte simmetria.
La tabella di DH in questo caso è costruita come segue:

T	α	d	θ	r	$Type$
$T01$	0	0	$2k * \pi/6$	0	$Fixed$
$T12$	$\pi/2$	0.045	0	0.018	$Fixed$
$T23$	0	0	0	0.085	$Rotoidal$
$T34$	0	0	$\pi/2$	0.124	$Rotoidal$

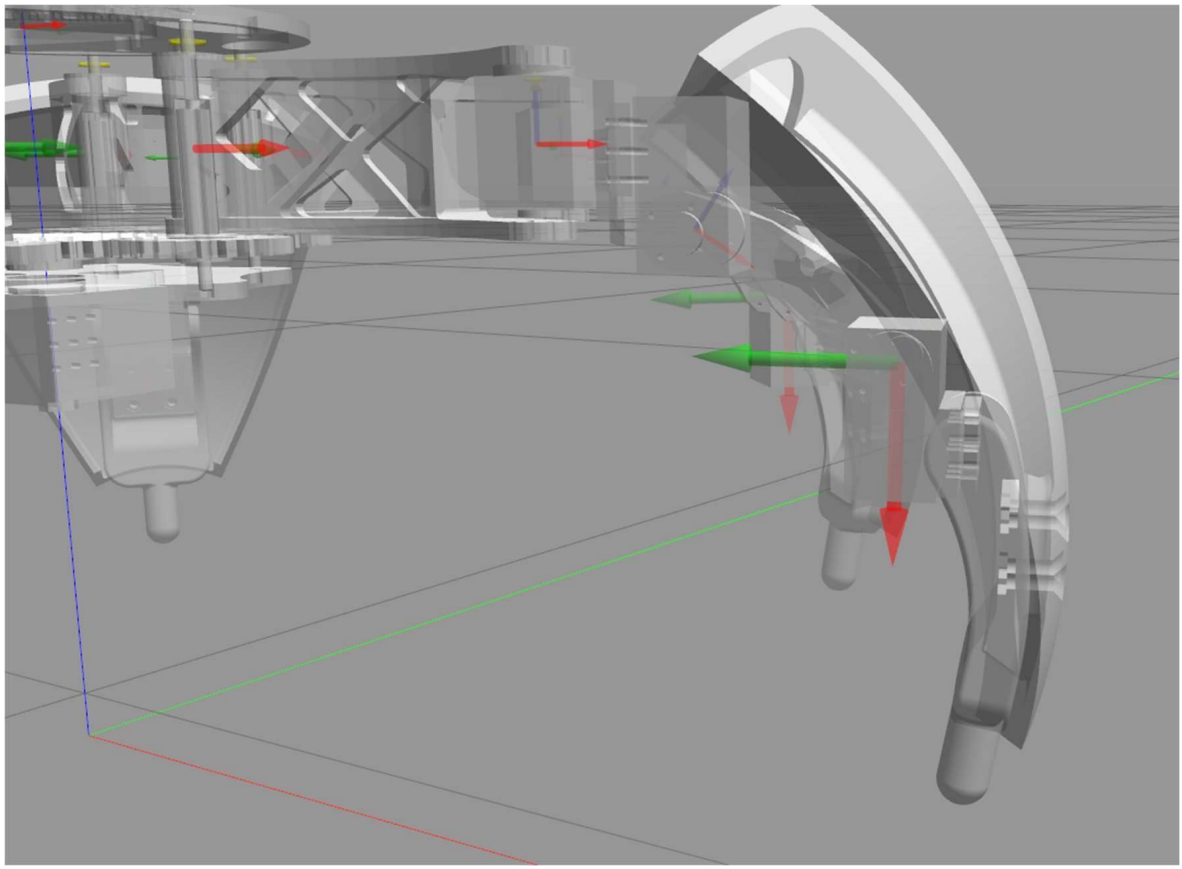


La tabella fa riferimento al singolo seriale, gli altri sono costruiti variando k per via della simmetria del problema.

La stessa strategia è applicata anche nella parte inferiore.

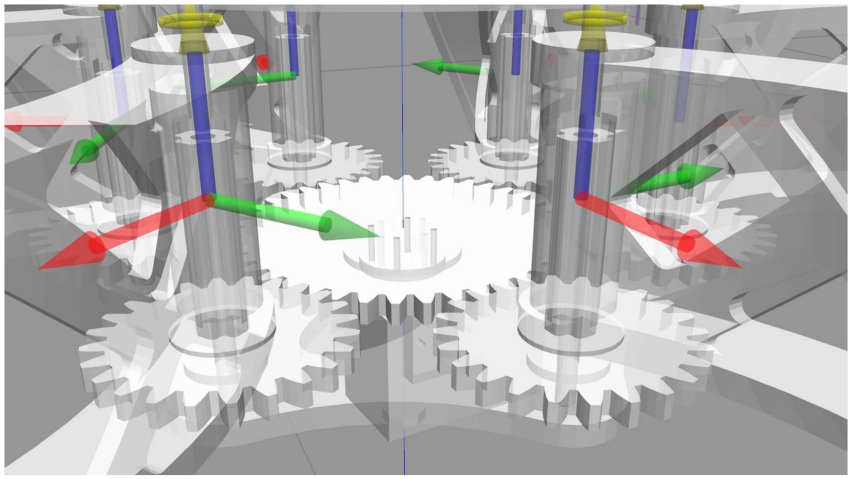
PARTE INFERIORE

La parte inferiore, essendo quella che in fase di camminata si occupa della locomozione del robot, è necessariamente più complicata. Essa è infatti composta da 6 seriali, disposti come sopra.



Il seriale inferiore è però costruito con 3 giunti indipendenti ogni gamba e un giunto mimik per gamba.

Il giunto mimik viene mosso da un meccanismo a ventaglio ed agisce in modo dipendente su tutte le gambe e allo stesso modo.

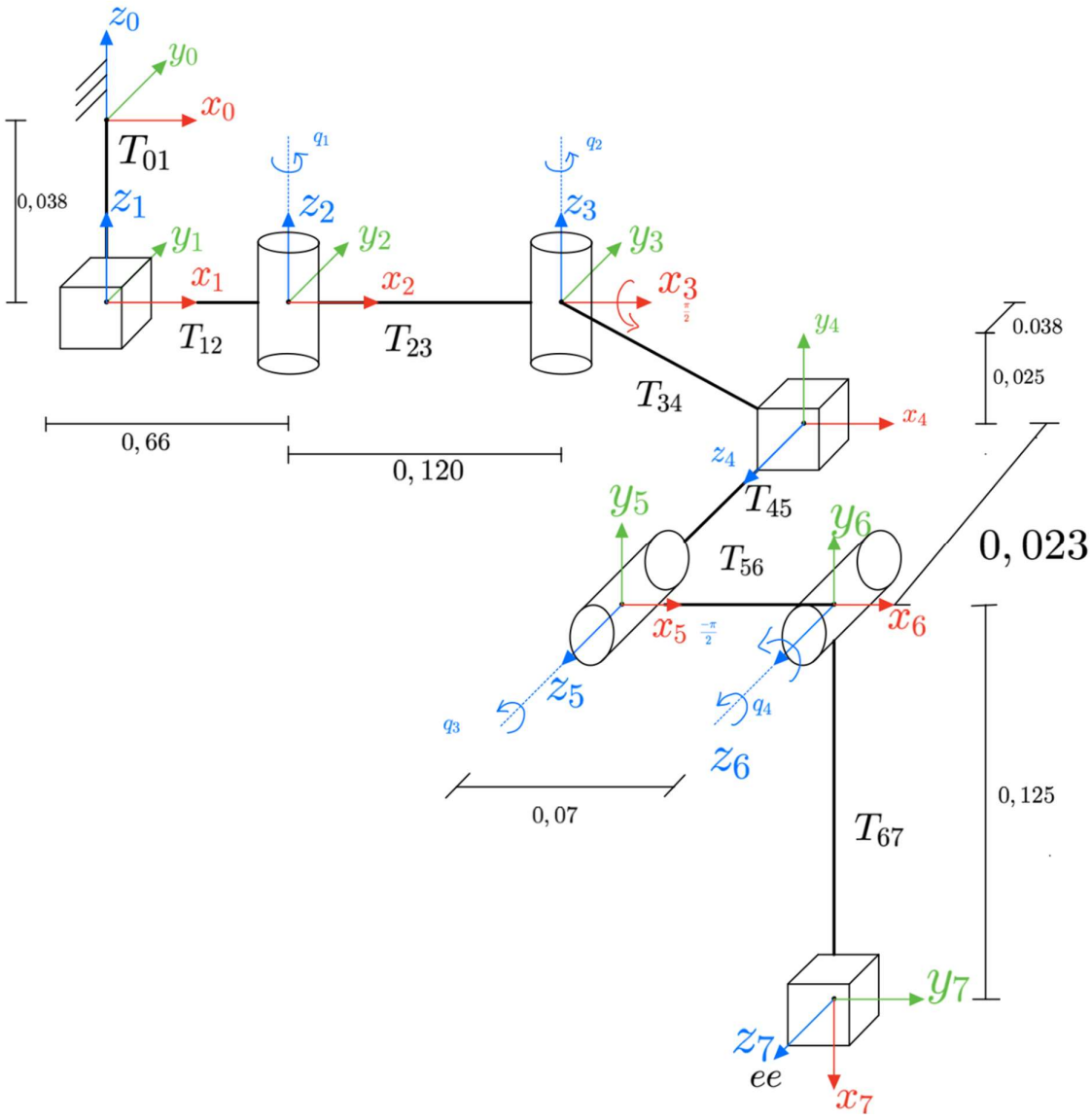


Questo approccio è stato deciso per limitare il numero di motori (già nel robot molto elevato), il peso e in generale diminuire la complessità del problema.

Facendo riferimento alla figura possiamo anche in questo caso caratterizzare la catena cinematica mediante DH e costruire quindi una parametrizzazione.

In questo caso la tabella DH risulta la seguente:

T	α	d	θ	r	$Type$
$T01$	0	-0.03825	$2k * \pi/6$	0	$Fixed$
$T12$	0	0	0	0.066	$Fixed$
$T23$	0	0	0	0.120	$Rotoidal$
$T34$	$\pi/2$	-0.025	0	0.038	$Rotoidal$
$T45$	0	0.02375	0	0	$Fixed$
$T56$	0	0	0	0.07	$Rotoidal$
$T67$	0	0	$-\pi/2$	0.12475	$Rotoidal$


z_0
y_0
x_0
T_{01}
0,038
z_1
y_1
x_1
T_{12}
q_1
z_2
y_2
x_2
T_{23}
q_2
z_3
y_3
x_3
T_{34}
y_4
x_4
z_4
T_{45}
0.038
0,025
0,66
0,120
y_5
T_{56}
y_6
0,023
x_5
x_6
z_5
q_3
q_4
z_6
0,07
T_{67}
0,125
y_7
z_7
ee
x_7

2 COSTRUZIONE CINEMATICA

Una volta definita la cinematica per entrambe le parti possiamo scrivere una relazione tra le coordinate finali dell'end effector e gli angoli di giunto, ossia $\mathrm{T}_{0-\mathrm{n}}(\mathrm{q})$.

Ciò ci permette quindi di definire anche il sistema di riferimento finale e quindi al variare di q possiamo sapere il punto dell'end effector e gli angoli rispetto al punto di origine.

Dalla matrice $T_{0-7,l}(qi,k)$ e $T_{0-4,u}(qi,k)$ possiamo estrarre le coordinate finali e gli angoli di eulero come segue:

$$T_{0,n} = \prod_{i=1}^{n} T_{i-1,i}$$

Le coordinate spazili del punto coincidono con i primi 3 elementi dell'ultima colonna delle matrici, mentre gli angoli di eulero vengono estratti dal blocco 3x3 corrispondente alla matrice di rotazione.

Le coordinate del punto si ottenono come :

$$p_n = [x\ y\ \ z]^T = T_{0,n}[3,0:3]$$

Mentre esplicitando la matrice di rotazione nella seguente forma:

$$R = R_z(\varphi) \cdot R'_y(\theta) \cdot R''_z(\psi)$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Gli angoli di eulero si ottengono dalle note formule :

$$\Phi = Atan2(r_{23}, r_{13})$$
$$\theta = Atan2\left(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}\right)$$
$$\psi = Atan2(r_{32}, -r_{31})$$

Cinematica inversa

Dopo aver costruito la cinematica diretta siamo passati alla cinematica inversa. Il problema ammette 2 metodi risolutivi :

- Risoluzione analitica del problema
- Risoluzione algoritmica

La strada che è stata decisa per arrivare alla soluzione è la seconda. La scelta è giustificata dal fatto che è stato cenessario costruire un metodo di cinematica inversa del tutto generale così da poter incorporare sia la composizione superiore che inferiore delle gambe in un unico metodo risolutivo.

La risoluzione analitica , pur esseno ovviamente più veloce rispetto alla soluzione algoritmica ha diversi svantaggi da un punto di vista implementativo:

1) Non è generica
2) Non permette di gestire le eccezioni (ad esempio se vengono assegnati punti impossibili da raggiungere)
3) È possibile assegnare solo le coordinate scelte in fase di scruttura di equazioni

Il metodo algoritmico implementato da noi prevede invece un elevato grado di generalità , è stato usato infatti lo stesso algoritmo sia per la parte inferiore che per la parte superiore (ma può essere potenzialmente esteso a qualunque tipo di robot (seriale/parallelo).

Le eccezioni vengono gestite in questo caso minimizzando l'errore quadraticomedio: assegnando posizioni non raggiungibili l'algoritmo prevedere di convergere a una soluzione che minimizza lo scarto quadratico medio degli errori e quindi farà del suo meglio per portare il seriale pcon end effector più vicino al punto assegnato.

Il terzo punto prevede una modifica dell'algoritmo standard utilizzato nella maggior parte dei casi che permette di assgnare nel caso delle gambe superiori due delle coordinate tra x,y,z,yaw,pitch,roll, e per le gambe inferiori 3 delle coordinate (o 4 se viene considerato anche il meccanismo di folding).

Le coordinate possono essere scelte a piacimento, senza essere vincolati a dover assegnare coordinate specifiche.

Il componente forndamentale che è utile all'inversione cinematica (ma anche per valutazioni a livello di comporamento dinamico) è il jacobiano, esso può essere ottenuto direttamente in maniera sequenzale dalla cinematica diretta, data infatti

una certa configurazione otteniamo il jacobiano geometrico come segue:

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix}$$

$$J_{g_i} = \begin{pmatrix} J_p \\ J_o \end{pmatrix}_i = \begin{cases} \begin{pmatrix} z_{i-1} \\ 0 \end{pmatrix} \; per\ giunti\ prismatici \\ \begin{pmatrix} z_{i-1} \times (p_{ee} - p_{i-1}) \\ z_{i-1} \end{pmatrix} \; per\ giunti\ rotoidali \end{cases}$$

J_{g_i} rappresenta la i-esima colonna dello jacobiano e iterando sulle colonne possiamo comporre il jacobiano completo affiancando le colonne :

$$J_g = \begin{pmatrix} J_{g_1} & J_{g_2} & \cdots & J_{g_n} \end{pmatrix}$$

Una volta ottenuto il jacobiano geometrico è sufficiente effettuare una trasfomazione di eulero sulla parte di orientazione per ottenere il jacobiano analitico J_a.

$$J_{a_i} = \begin{pmatrix} j_{a_p} \\ J_{a_0} \end{pmatrix}_i = \begin{pmatrix} j_p \\ J_o \end{pmatrix}_i \begin{bmatrix} 1 & 0 \\ 0 & R \end{bmatrix}$$

Componiamo quindi il jacobiano completo come in maniera analoga al jacobiano geometrico affiancando le varie colonne :

$$J_a = \begin{pmatrix} J_{a_1} & J_{a_2} & \cdots & J_{a_n} \end{pmatrix}$$

Dove $R(\varphi, \vartheta, \psi)$ corrisponde a una rotazione ZYZ mediante gli angoli di eulero.

Attraverso il jacobiano analitico possiamo scrivere :

$$\dot{\xi} = J_a(q) \cdot \dot{q}$$

Dove :

- $\xi = (x \quad y \quad z \quad \psi \quad \theta \quad \varphi)$ $configurazione spaziale$
- $q = (q_1 \quad \cdots \quad q_n)$ $variabili di giunto$
- $\dot{\xi} = \begin{pmatrix} \dot{x} & \dot{y} & \dot{z} & \dot{\psi} & \dot{\theta} & \dot{\varphi} \end{pmatrix}$ $variazione della configuraione spaziale$

In termini di calcolo numerico l'operazione viene eseguita in termini discreti e quindi:

$$\Delta\xi = J_a \cdot \Delta q$$

La soluzione viene quindi tovata in maniera algoritmica :

$$\Delta q \ = \ J_a^{-1} \cdot \Delta\xi$$

Una soluzione di questo tipo porta notevoli complicazioni a livello di calcolo in quanto il jacobiano potrebbe non essere quadrato e quindi non direttamente inversibile, inoltre sussiste il problema delle singolarità cinematiche.

Sono questi aspetti da tenere in considerazine in fase di progetto.

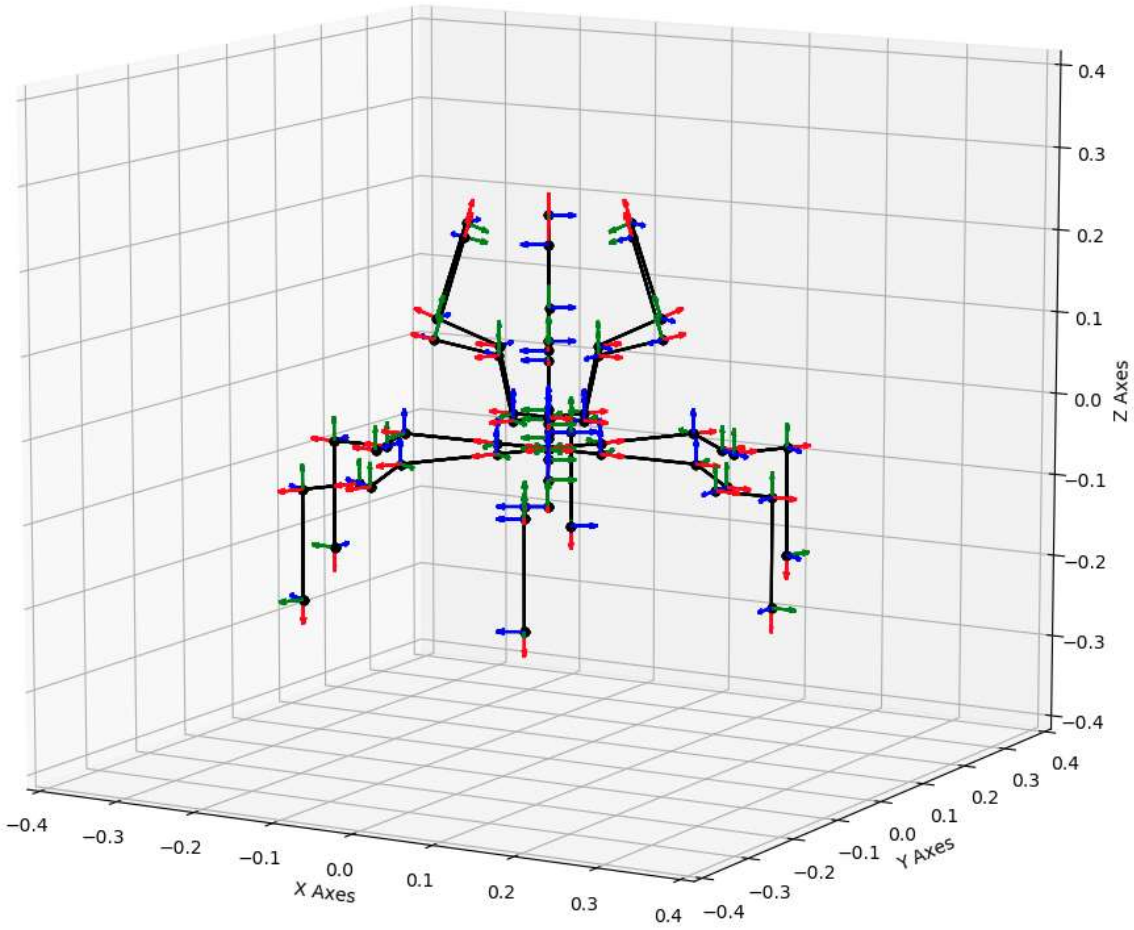
Cinematica del parallelo

Come gia detto precedentemente il robot è stato costruito basandosi sulla simmetria del problema, infatti i 2 paralleli, quello superiore quello inferiore sono costruiti ruotando di $k \cdot \pi$ con $k = 1 \dots 6$ le gambe del robot, sia superiori che inferiori, con al più un offset tra i paralleli superiori e inferiori per fare in modo chei sistemi di riferimento degli end effector superiori e inferiori del robot siano speculari rispetto al piano medio del robot una volta che viene effettuata la chiusura a sfera.

I sistemi di riferimento di origine di tutte le gambe partono dallo stesso punto, ossia il centro della sfera del robot.

In questo modo abbiamo un riferimento preciso per tutte le gambe.

La risoluzione del parallelo è trattata in maniera completamente algoritmica come nel caso precedente sfruttando i risultati ottenuti dalla cinematica inversa dei seriali.

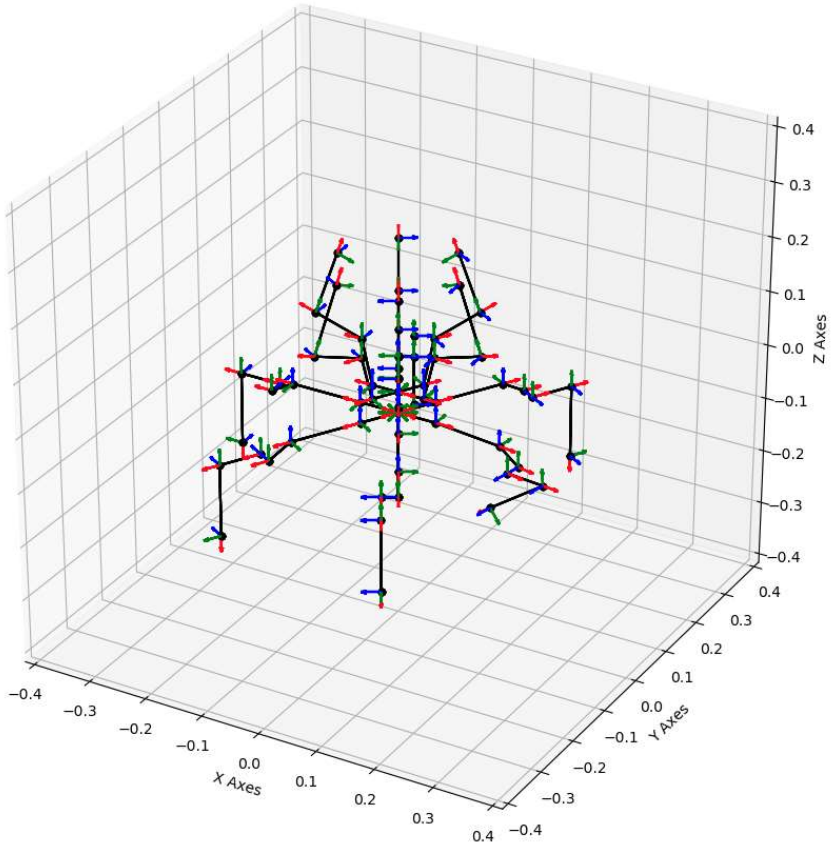


In questa figura possiamo vedere la caratterizzazione cinematica completa del robot (con degli offset sulle gambe in modo da far coincidere il modello con la simulazione).

Mediante una variazione dei parametri attivi del robot possiamo eseguire movimenti sia sulle gambe superiori che inferiri in configurazione seriale o muovendo tutto il parallelo.

E' importante notare che per come è stato costruito il robot, il primo giunto delle gambe inferiori è dipendete dal primo giunto delle altre bambe inferiori a causa del meccanismo di folding.

Come possiamo infatti vedere nella figura , modificando i valori del primo giunto si modificano in automatico tutti i giunti del parallelo mentre muovendo glu ultimi 3 giunti di una gamba, le altre gambe non vengono influenzate.



Per visualizzare in maniera dinamica le soluzioni della cinematica inversa è stato costruito un file jupiter situato nella cartella *shell_robot_kinematichs/src*. Esso permette la visualizzazione delle soluzioni della cinematica sua per il seriale che per il parallelo, la posizione e orientazione degli SDR e la struttura del robot.

Cinematica computazionale

Per questo riguarda gli aspetti computazionali e il calcolo della cinematica inversa, come detto precedentemente è stato creato un programma del tutto generale che può gestire non solo il robot specifico ma tutti i possibili robot paralleli e seriali siano essi attuati, sottoattuati o ridondanti.

La costruzione di questo programma si avvale di classi, in particolare :

- *Classe giunto(Joint)*
- *Classe catena cinematica(Serial Chain)*
- *Classe parallelo(Parallel Chain)*

CLASSE JOINT

Mediante la classe giunto possiamo costruire una parametrizzazione specificando:

- i parametri di DH (α, r, θ, d)
- il parametro cinematico variabile nel giunto

Il tipo di giunto, (traslazione le o rotazione) viene setatto immediatamente , una volta scelta la variabile di giunto in maniera completamente automatica.

La classe giunto in se serve non solo a definire il singolo giunto (e quindi la matrice cambiamento di coordinate di DH) ma , mediante la composizione di oggetti giunto possiamo inserirli all'interno della classe catena cinematica seriale che crea in automatico la struttura del robot seriale a partire dalle matrici definite all'interno delle classi joint in maniera sequinziale, definendo le varie matrici $T_{0,i}$ e i relativi jacobiani dei seriali (J_g, J_a).

CLASSE SERIAL

La classe cinematica seriale viene infatti inizializzata passando ad essa una lista di oggetti Joint in sequenza dal quale tutta la struttura del seriale prende forma.

Questa classe in particolare contiene al suo interno tutta la cinematica del singolo seriale ed è quindi possibile richiedere alla classe di effettuare qualunque operazione riguardante la cinematica come :

- muovere giunti singoli
- muovere tutto il seriale
- effettuare la cinematica inversa del robot

quest'ultima viene eseguita assegnando con estrema semplicità le coordinate finali del punto che si intende raggiungere, ossia $x, y, a, yaw, pitch, roll$. Il programma effettuerà il calcolo degli angoli di giunto da assegnare e li restituirà come risultato aggiornando al contempo il seriale (e quindi le singole sottoclassi Joint).

CLASSE PARALLEL

Una volta composto il seriale è possibile costruire in maniera analoga alla precedente il robot parallelo mediante la classe Parallel chain. Ciò è stato fatto per poter effettuare il movimento del robot parallelo, in particolare della floating base.

Per la risoluzione del parallelo si è adottato lo stesso principio di cinematica inversa , ossia viene calolato il punto che l'end effector dovrebbe avere rispetto alla floating base quando quest'ultima raggiunge il punto desiderato, in questa maniera possiamo risolvere il problema di cinematica inversa in maniera analoga al caso di un seriale ma con SR finale calcolato rispetto a variazioni di posizione angoli della floating base

Grazie a questo approccio è possibile non solo risolvere la cinematica inversa del robot specifico ma viene risolta tutta la categoria di problemi di robot paralleli.

4 LIBRERIA

La libreria completa si compone sia delle 3 classi specificate sopra ma anche di un modulo di supporto composto da :

- una parte matriciale (che definisce tutte le trasformazioni matriciali utili in robotica
- un modulo su supporto per il calcolo di jacobiani, geometrici e analitici.
- un modulo per la generazione di NURBS e curve di Bezier
- algoritmi di inversione cinematica

La libreria costruita è chiamata *Robopy* ed è a disposizione su github su licenza MIT.

Algoritmo L-M modificato

Per il calcolo della cinematica inversa è stato ultilizzato un algoritmo di Levemberg Marquart con alcune modifiche che consentono all'algoritmo di trattare efficaciemente robot sotto attuati , facendo in modo di poter imporre un numero di coordinate ammissibili e di poter decidere quali imporre.

La scelta di questo algoritmo è giustificata dal fatto che è possibile dimostrare che l'algoritmo quando l'errore è grande converge in maniera polinomiale , mentre quando siamo vicini al punto converge in maniera esponenziale, cosi facendo si garantisce un convergenza veloce e con minori probabilità di incontrare singolarità cinematiche.

La modifica consente di utilizzare un algoritmo di Levemberg-Marquart con cancellazione di righe e colonne a seconda del numero di coordinate massime da poter imporre, riducendo la matrice jacobiana una matrice quadrata.

Le coordinate che non possono essere imposte vengono contrassegnate con un identificativo np.nan che corrisponde a Not A Number. In questo caso l'algoritmo va a cacellare tutte le righe della matrice jacobiana (che in condizione standard sono 6 per un seriale) e tutte le coordinate contrassegnate con NAN.

Queste operazioni vengono di fatto escluse dall'algoritmo con la possibilità di estendere facilmente l'inversione cimenatiche per risolvere anche problemi di seriali sottoattuati.

ALGORITMO

L'algoritmo è iterativo e assegnato ξ_d (configurazione desiderata) si compone come segue:

1. calcolo la cinematica diretta :

$$T_{0,n}(q^j) = \begin{bmatrix} R_n & p_n \\ 0 & 1 \end{bmatrix} = \prod_{i=1}^{n} T_{i-1,i}$$

2. calcolo il jacobiano associato alla configurazione corrente :

$$J_a(q^j) = \begin{pmatrix} j_{a_1} & j_{a_2} & \cdots & J_{a_n} \end{pmatrix}$$

3. calcolo l'errore :

$$e = \xi - \xi_d$$

inverto la cinematica e calcolo la nuova configurazione q come seque:

$$M = pinv\left(J^T \cdot J + K \cdot diag(J^T \cdot J)\right) \cdot J^T$$
$$\Delta q = M \cdot e$$

$$K\ scelto\ empiricamente$$

4. aggiorno gli angoli di giunto e ritorno al punto 1:

l'iterazione termina in 2 casi :

a. l'errore risulta sotto la soglia minima scelta dal progettista
b. si raggiunge il numero massimo di iterazioni previsto

$$q^{j+1} = q^j + \Delta q$$

La modifica dell'algoritmo sta nella possibilità di assegnare posizioni desiderate come segue :

$$\xi_d = [x \quad y \quad z \quad \psi \quad \theta \quad \varphi]$$

Dove è possibile contrassegnare le variabili che non sono di interesse con *np.nan*, in

questo modo l'identificativo applicato dice al programma quali varibili di configurazione desiderata andare a ignorare, in quanto se rientrassero nel calcolo l'algoritmo andremme a considerare una minimizzazione anche su quelle variabili (ma comunque non tutte possono essere assegnate in modo indipendete e quindi le soluzioni sarebbero inconsistenti).

$$\xi_d = [x \quad y \quad z \quad \psi \quad \theta \quad \varphi] \rightarrow \xi_{d_{rid}}(\text{subset di } \xi\text{d})$$

$\xi_{d_{rid}}$ è di fatto un vettore in cui vengono scelte g variabili da assegnare ,dove g è in questo caso è il numero di gradi di liberà del manipolatore seriale.

Di fatto ciò che viene effettuato è un taglio della matrice Jacobiana e di ξ_d potendo quindi costruire $\xi_{d_{rid}}$, ossia un vettore di configurazione desiderata ridotto con variabili desiderate epurate dalle variabili di configurazione del quale non siamo interessati.

Ciò è utile principalmente per i robot sottoattuati per i quali non è possibile assegnare tutta la configurazione in modo indipendete.

La convergenza dell'algoritmo si ha in media dopo 2-3 iterazioni con un errore massimo consentito dell' 1% , la soluzione è quindi abbastanza buona.

Speed up del codice

Per il programma è stato scelto di utilizzare il linguaggio di prorgammazione Python per un veloce sviluppo del codice e di testing.

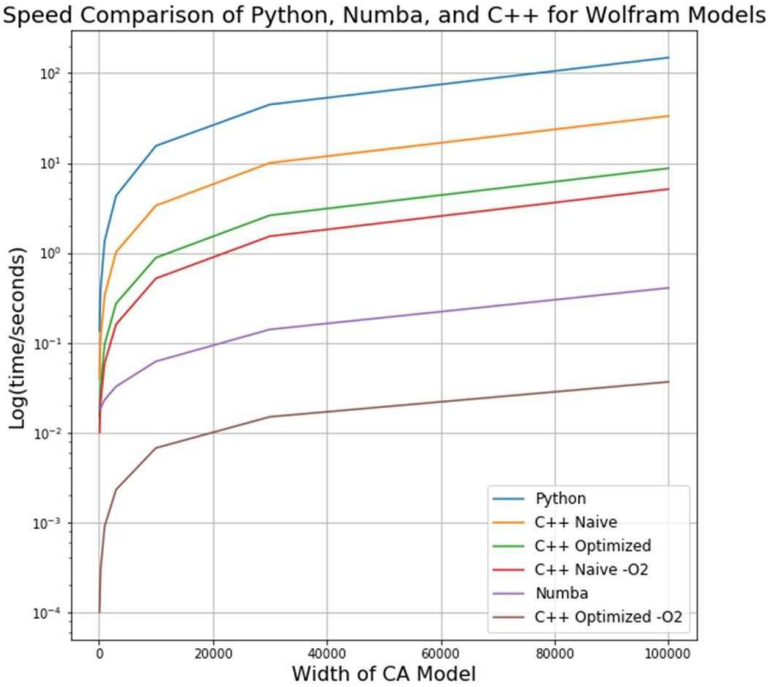
Come noto in ambiente informatico, python è un linguaggio intepretato e non viene compilato (come accade ad esempio per C e $C++$) perciò la compilazione effettiva del codice avviene in runtime mano a mano che il codice viene eseguito.

Siccome il programma non lavora offline ma online, devono eessere effettuati calcoli a ogni ciclo ,ciò crea notevili problemi a livello computazionale poichè l codice risulta molto lento (il programma deve essere infatti compilato tutte le volte che viene letto dall'interpreter) .

Nel programma è però necessario effettuare una serie di pesanti operaizoni computazionali , deve essere infatti eseguito nel caso di un movimento la generazione della traiettoria e la cinematica inversa per tutti i punti generati in manira sequenzale e per tutte le gambe.

Vista la mole di calcoli è quindi fondamentale la velocià di esecuzione del codice.

Per fare ciò sono stati inseriti all'interno del programma blocchi di codice in C e forran (gia compilati) in grado di eseguire operazioni in modo notevolmente più veloce , specialemente per la parte di calcoli effettiva. Inoltre il codice viene parallelizzato su tutti i core della CPU in modo da diluire il carico computazionale e non bloccare nessun task.



Mediante il grafico possiamo infatti vedere una comparazione su generici algoritmi di wolfram ,dei vari tempi di calcolo da parte di vari linguaggi di programmazione, in particolare è da notare la grossa differenza tra python e il codice ottimizzato in C++ (simile comunque al C a livello di velocità).

Questo giustifica la tecnica di programmazione ibrida adottata.

L'obbiettivo stato raggiunto mediante 2 principali librerie esterne :

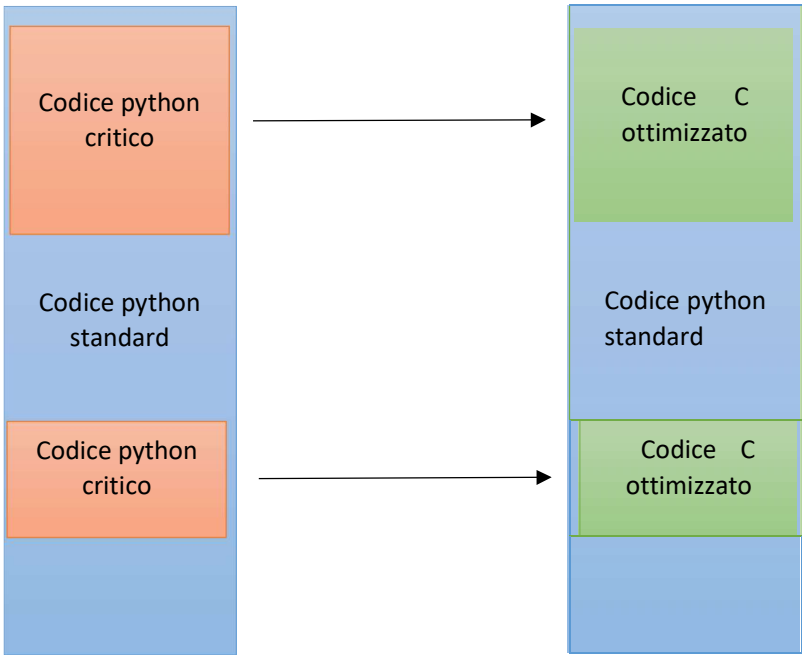
- *Numpy*
- *Numba*

Il primo fornisce infatti delle primitive di algebra lineare (scritte anch'esse in c) in grado di eseguire operazioni sugli array , matrici e operazioni su di essi.

Il secondo fornisce invece un modo semplice per effettuare la compilazione di blocchi di codice python in C cosi a effettuare lo speed up nelle porzioni di codice piu computazionalemnte onerose.

Grazie a questo approccio di programmazione misto e parallelo si è raggiunto il risultato di velocità di esecuzione desiderato, incrementando la velocità di un fattore 100.

Codice standard codice ibrido ottimizzato



E' possibile incrementare ulteriormente la velcoità del codice mendiante utilizzo della *GPU*, ciò non è stato eseguito poichè la *GPU* voleva essere impiegata per altri task, inoltre è stato raggiunto con notevole margine la velocità desiderata su processore e di conseguenza non si è sentita la necessità di incrementare ulteriormente la velcità.

Generazione di traiettorie

per far effettivmente muovere il robot è stato necessario costruire delle traiettore per fare in maniera tale che il robot avesse dei punti da seguire sul quale effettuare la cinematica inversa.

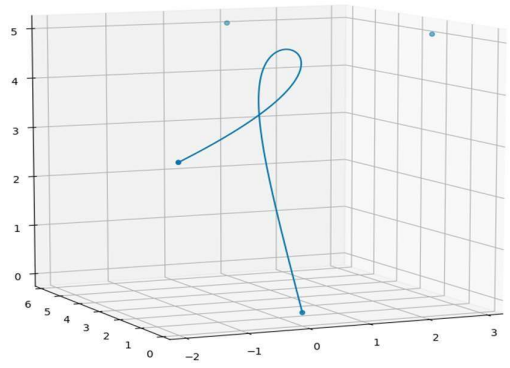
Assegnare i punti o angoli di giunto è stato catalogato come controproducente in quanto crea notevoli problemi:

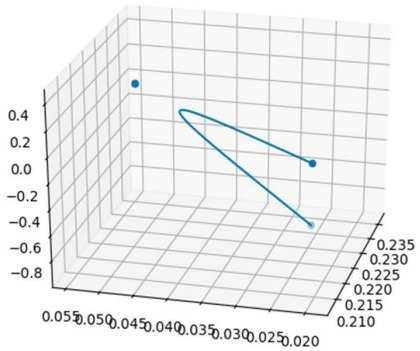
- è necessario assegnare tanti punti per creare una traiettoria
- nel caso di robot paralleli assegnando solamente i punti ed eseguendo la traiettoria in maniera lineare (nello spazio dei giunti) fa in modo che gli end effector(nello spazio) non rimangano fermi . e quindi il robot si comporterebbe da parallelo solo nel punto inizuale e finale
- Impossibilità di costruire traiettorie complesse.

Per via di questi problemi è stato deciso un diverso approccio, ossia è stato creato un software che genera delle curve $NURBS$, ossia curve polinomiali i cui parametri sono definiti a partire dai punti che vengono dati in input .

In particoalre verrà generata una curva 3d passante per una serie di punti sequenziali dati , in questa maniera è possibilie costruire una traiettoria dati n punti chiamati nodi. Dagli n nodi vine estratta la traiettoria e generati tutti i punti intermendi.

Una volta generati tutti i punti intermedi essi possono essere utilizzati per effettuare la cinemtica inversa ed avere quindi movimenti fluidi del robot. ciò crea un campionamento fine dello spazio 3d lungo la curva e quindi movimenti che seguono la curva in manira apparentemente continua.





E' da far presente che nella nostra implementazione le NURBS create vengono utilizzare per generare riferimenti di posizione del robot, non vengono invece usate nel caso di assegnamento di angoli.

Per questi ultimi si è scelto un approccio lineare che presenta però delle criticità, sappiamo infatti non essere ottimale in questi casi per via del fatto che non ci si sta muovendo lungo una superficie in *SO3*. Per la generazione di riferimenti migliori sarebbe necessario una implementazione di spline nello spazio dei quaternioni che quindi rispettano la rischiesta di appartenenza a una superficie di *SO3*.

Anche nel caso di approccio lineare però, il sistema sembra funzionare senza troppi problemi e di conseguenza per non introdurre complicazioni eccessive si è deciso di mantenere questa implementazione.

Per la visualizzazione e l'assegnazione delle posizioni è stato costruito un programma mediante *Jupiter notebook* che permette di assegnare posizioni e visualizzarle, cosi da avere una anteprima del comportamento dell'algoritmo prima di implementare effettivamente la configurazione decisa in simulazione.

La verifica in simulazione è comunque necessariaa per verificare eventuali malfunzionamenti dovuti a eventuali collisioni.

Assegnamento posizioni

Per effettuare l'assegnamento posizioni è stato suddiviso il problema in 3 parti:

- azioni
- sottoazioni
- azioni atomiche

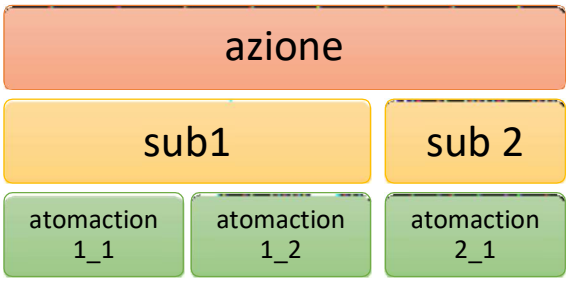
Le azioni sono azioni "complesse" che il robot è in grado di eseguire come ad esempio la camminata e la rotazione su se stesso.

le azioni sono composte da sottoazioni che vengono eseguite in maniera sequenzale e la successiva comincia una volta che la precedente è terminata.

Le sottoazioni sono composte da azioni atomiche eseguite in maniera parallela , ossia tutte nello stesso movimento.

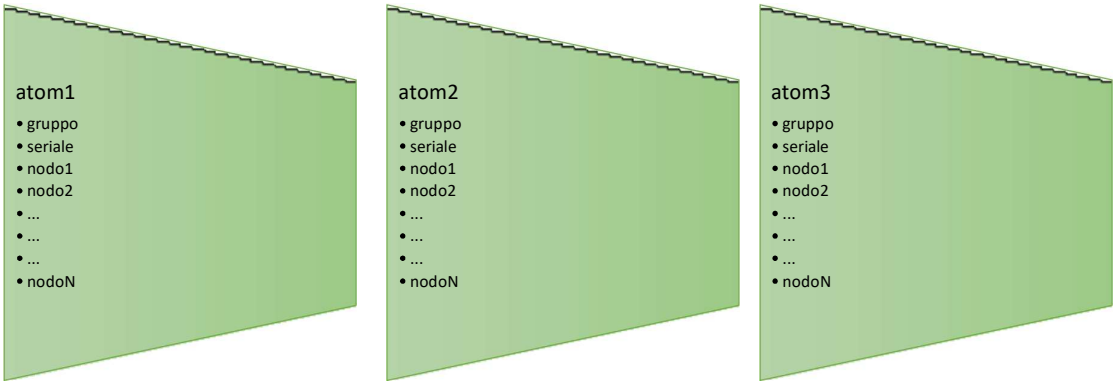
Questa struttura permette la creazione di azioni complesse a partire da un set di sottoazioni e azioni atomiche riutilizzabili per tutta una categoria di movimenti.

La singola azione si compone quindi in una struttura simile alla seguente:



Il set ha inoltre una prorpietà gerarchica e di priorità ed è del tutto generale e parametrica, il che significa che è possibile ,aggiungendo informaioni sull'ambiente, andare a definire un aggiornamento di parametri delle taiettorie che tiene conto delle variazioni nell'ambiente circostante.

Le azioni atomiche sono definite specificando il gruppo del parallelo da muovere , il seriale e dando i punti di controllo delle NURBS.



Ogni punto di controllo è definto a partire da:

- x
- y
- z
- yaw
- $Pitch$
- $roll$

In questo modo ogni azione atomica va a generare una traiettora che viene eseguita dal seriale specificato appartenente al gruppo specificato.

Le traiettorie vengono poi passate alla parte computazionale cinematiche che ne calcola la cinematica inversa e compone un set di angoli di giunto da inviare come riferimento alla simulazione cosi che il robot possa muoversi.

Aspetti simulativi e di controllo

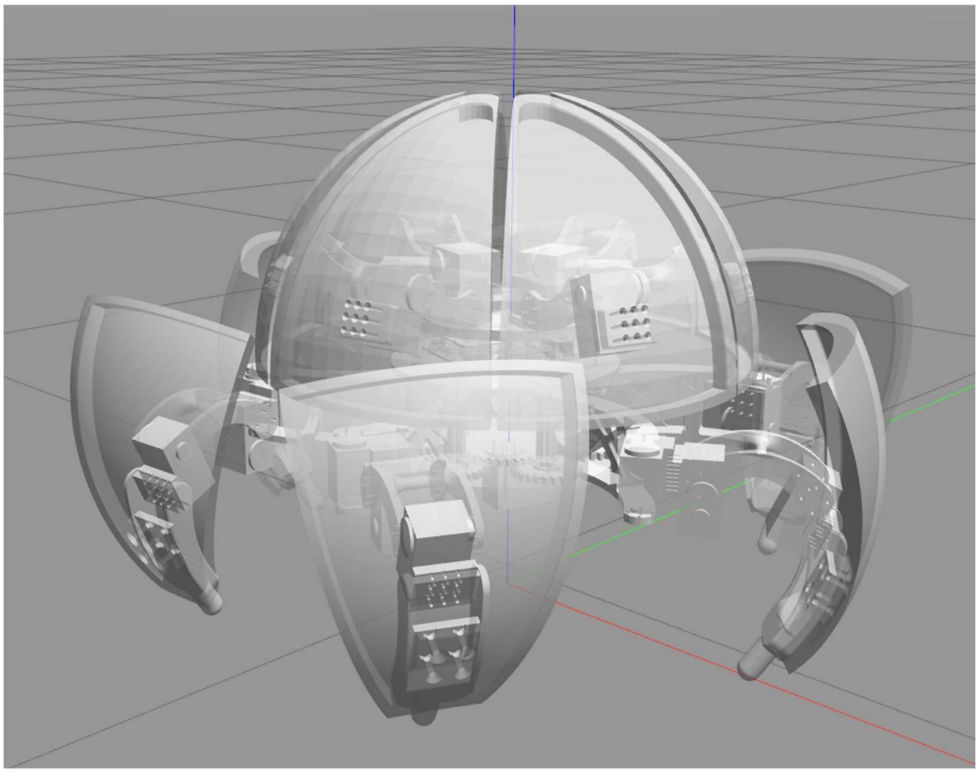
SIMULAZIONE

la simulazione è costruita su *Gazebo* e *ROS*, non si basa però sui software cinematici presenti in *ROS* ma sulla libreria cinematica custom definita precedentemente.

Per quanto riguarda la simulazione il robot è stato costruito mediante *URDF* nel quale vengono specificate le caratteristiche del robot, in particolare :

- cinematica
- giunti
- motori
- mesh
- parametri di simulazione

I file di simulazione si trovano all'interno del package *shell_robot_descrition* e contengono l'urdf. Esso è strutturato in vari file in cui sono definite le macro per la crazione del robot. Il file più interessante e' il file variables che permette di modificare tutti i parametri di simulazione.



Questa metodologia è stata scelta per evitare di dover modificare l'urdf ogni volta che il robot veniva aggiornato a livello meccanico.

Per costruire una simulazione funzionante è di fondamentale importanza effettuare i settaggi della simulazione in maniera più simile possibile alla realtà tendendo in consideraizone tutte le caratteristiche del robot.

Infatti la simulazione effettua una risoluzione delle equazioni differenziali legate alla fisica del sistema in maniera discreta e cio ci permette di verificare il comportamento del robot e i suoi movimenti con i parametri che avrebbe nella realtà, se la simulazione venisse settata male il comportamento del robot potrebbe differire notevolmente nella realtà ed e quindi necessario tenere presente questo aspetto nella costruzione della simulazione.

Gazebo supporta 4 motori fisici , quello utilizzato nella simulazione e` ODE(Open Dynamics Engine).

Una tipica simulazione procederà in questo modo:

1. Crea un mondo dinamico.
2. Crea corpi nel mondo dinamico.
3. Imposta lo stato (posizione, ecc.) Di tutti i corpi.
4. Crea giunti nel mondo delle dinamiche.
5. Attacca le articolazioni ai corpi.
6. Imposta i parametri di tutti i giunti.
7. Creare un mondo di collisioni e oggetti geometrici di collisione, se necessario.
8. Creare un gruppo di giunti per contenere i giunti di contatto.
9. Loop:
    1. Applicare forze ai corpi se necessario.
    2. Regolare i parametri del giunto secondo necessità.
    3. Rilevamento delle collisioni mediante callback.
    4. Creare un giunto di contatto per ogni punto di collisione e inserirlo nel gruppo giunto di contatto.
    5. Eseguire lo step di simulazione.
    6. Rimuovere tutti i giunti nel gruppo di giunti di contatto.
10. Distruggere le dinamiche e i mondi di collisione.

Oltre a tenere presente gli aspetti legati ai parametri collegati al robot e all'ambiente circostante è importante anche il condizionamento della simulazone.

Essa infatti deve essere ben condizionata, questo aspetto è legato alla convergenza delle soluzioni delle equazioni differenziali discrete che vengono risolte. Questo aspetto è principalmente legato a :

- numero di iterazioni a ogni step
- passo temporale della simulazione
- gestione dei contatti
- coefficienti di smorzamento della simulazione

E' facilmente verificabile il tuning di questi parametri avviando la simulazione e vedere il comportamento della simulazione.

In generale passi di simulazione piu piccoli condizionano meglio la simulazione ma generano tempi di simulazione molto elevati , analogamente il numero di iterazioni della simulazione.

Per quanto rigurda i coefficienti di smorzamento il tuning è effettuato mediante prova-errore verificando il comporamento.

I contatti devono essere gestiti sia tra il robot e se stesso sia tra il robot e l'ambiente circostante ed è quindi fondamentale esettare i parametri di contatto per evitare risonanze computazionali.

E' da notare che il *Real Time Factor* della simulazione da noi creata è di circa 0.6 , ossiail tempo nella simulazione scorre al 60% rispetto allo scorrere nel tempo della realtà, ciò accade poichè il calcolatore non riesce a effettuare i calcoli in real time.

Questo problema è principalmente legato al simulatore Gazebo che effettua tutti i calcolo sulla CPU , che notoriamente non è adatta alla gestione di calcoli simulativi/fisici e di grafica, sarebbe invece piu adatta la GPU per il calcolo parallelo.

Purtroppo gli ideatori del software gazebo non hanno implementato una versione GPU del software e di conseguenza si hanno limiti di real time factor nei computer tradizionali.

Per risolvere il problema del real time factor sono stati utilizzati parametri addizionali in modo da rendere la simulazione piu` stabile cosi' da poter abbassare il passo di simulazione. In particolare i parametri utilizzati nella fase di stabilizzazione e incremento della velocita' di simulazione sono i seguenti:

- *cfm*
- *epr*
- *step size*

Il parametro CFM (Constrain Force Mixing) introduce vincoli di tipo soft. La maggior parte dei vincoli sono per natura "rigidi". Ciò significa che i vincoli rappresentano condizioni che non vengono mai violate. I vincoli "soft" sono invece progettati per essere violati, cio' permette si simulare materiali non completamente rigidi.

A livello tecnico il vincolo viene descritto da :

$$J \cdot v = c$$

dove v è un vettore di velocità per i corpi coinvolti, J è una matrice `` Jacobiana '' con una riga per ogni grado di libertà che l'articolazione rimuove dal sistema e c è un vettore del lato destro. Nella fase successiva, viene calcolato un vettore λ (della stessa dimensione di c) in modo tale che le forze applicate ai corpi per preservare il vincolo del giunto siano:

$$F = J^T \cdot \lambda$$

Il parametro CFM modifica l'equazione precedente imponendo :

$$J \cdot v = c + CFM \cdot \lambda$$

Dove CFM e` una matrice diagonale quadrata. Un valore di CFM diverso da zero consente di violare l'equazione originale di una quantita' proporzionale a $CFM \cdot \lambda$.

L'aumento del CFM, in particolare il CFM globale, può ridurre gli errori numerici nella simulazione. Se il sistema è quasi singolare, ciò può aumentare notevolmente la stabilità. Infatti, se il sistema si comporta male, una delle prime cose da provare è aumentare il CFM globale.

E' da notare che utilizzare questo approccio mediante CFM diminuisce l'accuratezza della simulazione ma introduce una notevole stabilita' simulativa.

Il parametro EPR gestisce una correzione dell'errore di contatto: quando un'articolazione collega due corpi, tali corpi devono avere determinate posizioni e orientamenti l'uno rispetto all'altro. Tuttavia, è possibile che i corpi si trovino in posizioni in cui i vincoli articolari non sono soddisfatti. Questo "errore congiunto" può verificarsi in due modi:

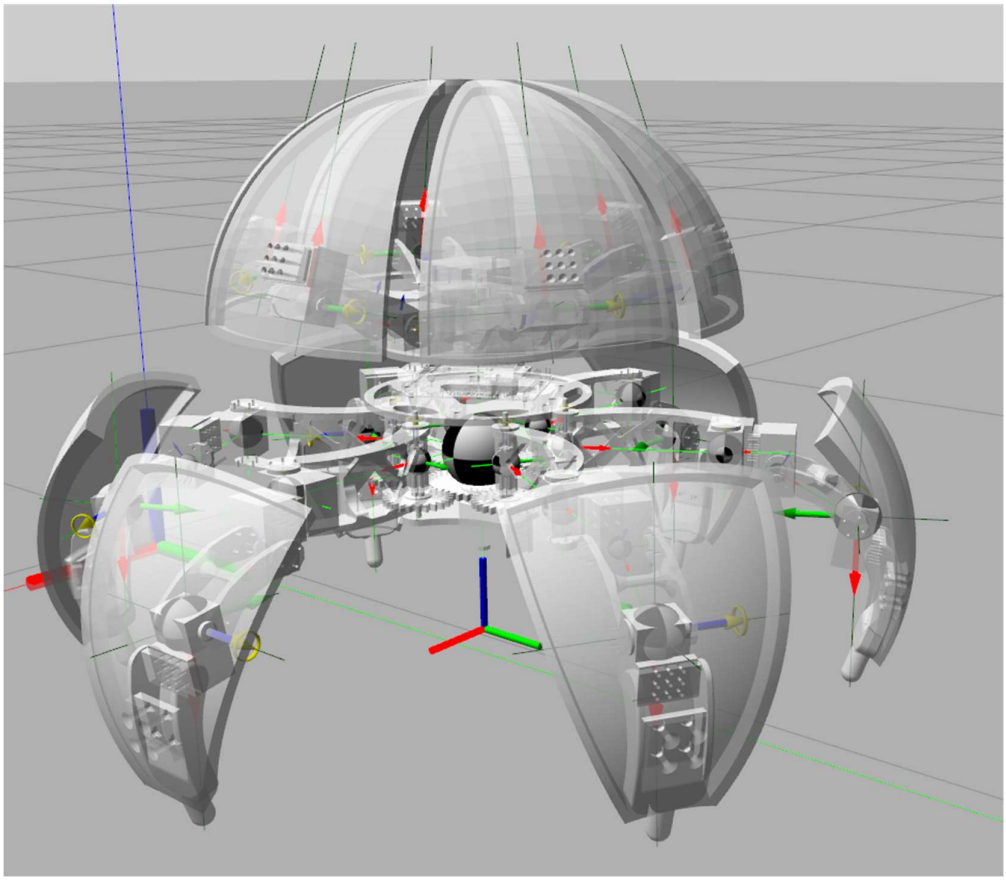
1. Se l'utente imposta la posizione / orientamento di un corpo senza impostare correttamente la posizione / orientamento dell'altro corpo.
2. Durante la simulazione, gli errori possono insinuarsi in quel risultato che i corpi si allontanano dalle loro posizioni richieste.

Esiste un meccanismo per ridurre l'errore articolare: durante ogni fase della simulazione ogni giunto applica una forza speciale per riportare i propri corpi nel corretto allineamento. Questa forza è controllata dal *parametro di riduzione* degli *errori* (ERP), che ha un valore compreso tra 0 e 1.

L'ERP specifica quale proporzione dell'errore congiunto verrà corretta durante la fase di simulazione successiva. Se ERP = 0, non viene applicata alcuna forza di correzione e alla fine i corpi si allontaneranno mentre la simulazione procede. Se ERP = 1, la simulazione tenterà di correggere tutti gli errori del giunto durante il passaggio temporale successivo. Tuttavia, non è consigliabile impostare ERP = 1, poiché l'errore del giunto non verrà completamente risolto a causa di varie approssimazioni interne. Si consiglia un valore ERP = da 0,1 a 0,8 (0,2 è l'impostazione predefinita).

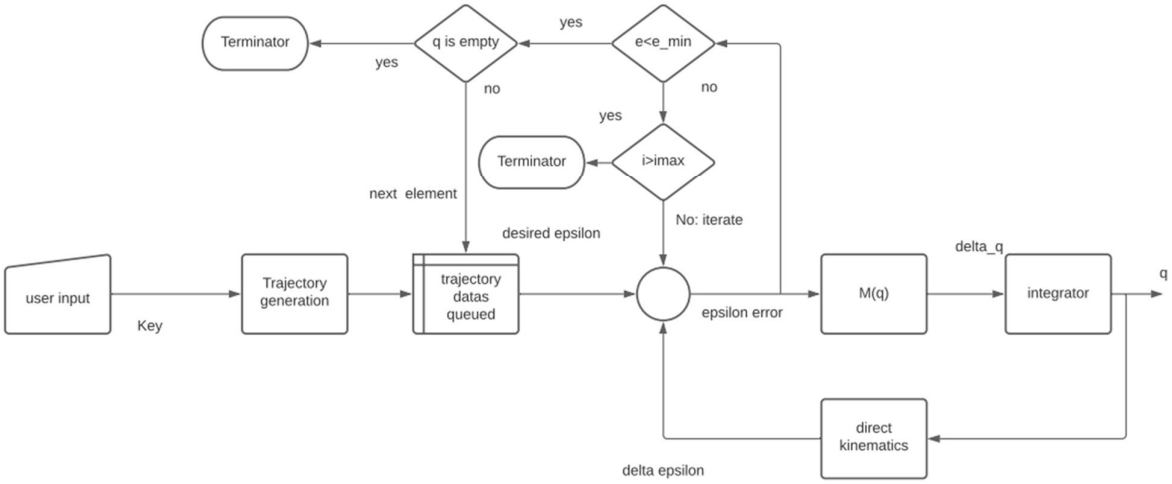
È possibile impostare un valore ERP globale che influisce sulla maggior parte dei giunti nella simulazione.

Una volta introdotti questi parametri e’ possibile aumentare max step size , ossia dimensione massima del passo temporale che può essere preso da un risolutore di passi temporali variabile (come simbody) durante la simulazione. Per i motori fisici con risolutori a passo fisso (come ODE), questa è semplicemente la dimensione del passo temporale.



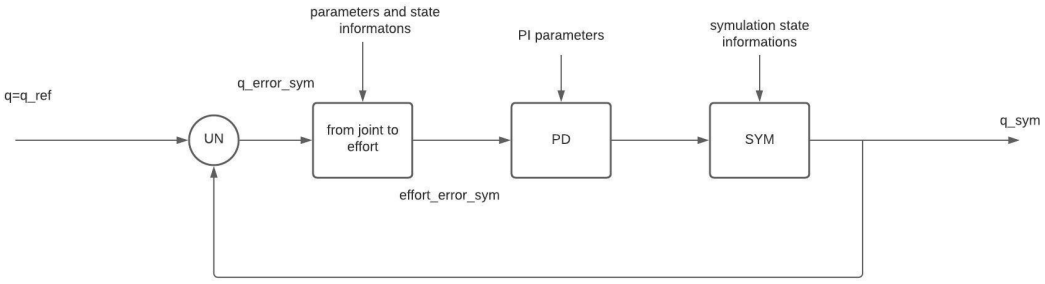
CONTROLLO

Il controlo avviene con una logica data dal seguente schema per quanto riguarda la parte di generazione dei riferimenti di angoli:



Quando l'utente seleziona una azione , il programma la interpreta e genera un set di riferimeti da eseguire in sequenza che vengono inseriti in una coda, successivamente il programma inizia le iterazioni e per ogni riferimento da inseguire (che risulta essere un riferimento in generale su tutte le coordinate di tutti i giunti del robot), calcola cinematica inversa, diretta e impone un nuovo q fino a che l'errore non raggiunge un valore accettabile o le iterazioni superano il numero massimo consentito.

In questo modo vengono generati i riferimenti da passare al "vero" controllore.



I riferimenti generati dal precedente controllore vengono inseriti nel circuito di controllo di ros , trasformati in riferimenti in coppia e mediante un PD viene effettuato l'inseguimento del riferimento su tutti i giunti. La simulazione che a ogni timestep riceve la variabile di coppia da assegnare , svolge il suo dovere muovendo i

motori e applicando una coppia proporzionale all'errore.

Si noti che l'intero sistema di controllo è MIMO, abbiamo infatti tutti i segnali vettoriali e in generale la simulazione è un sistema fortemente accoppiato, in quanto i suoi parametri variano al variare della configurazione inerziale del robot.

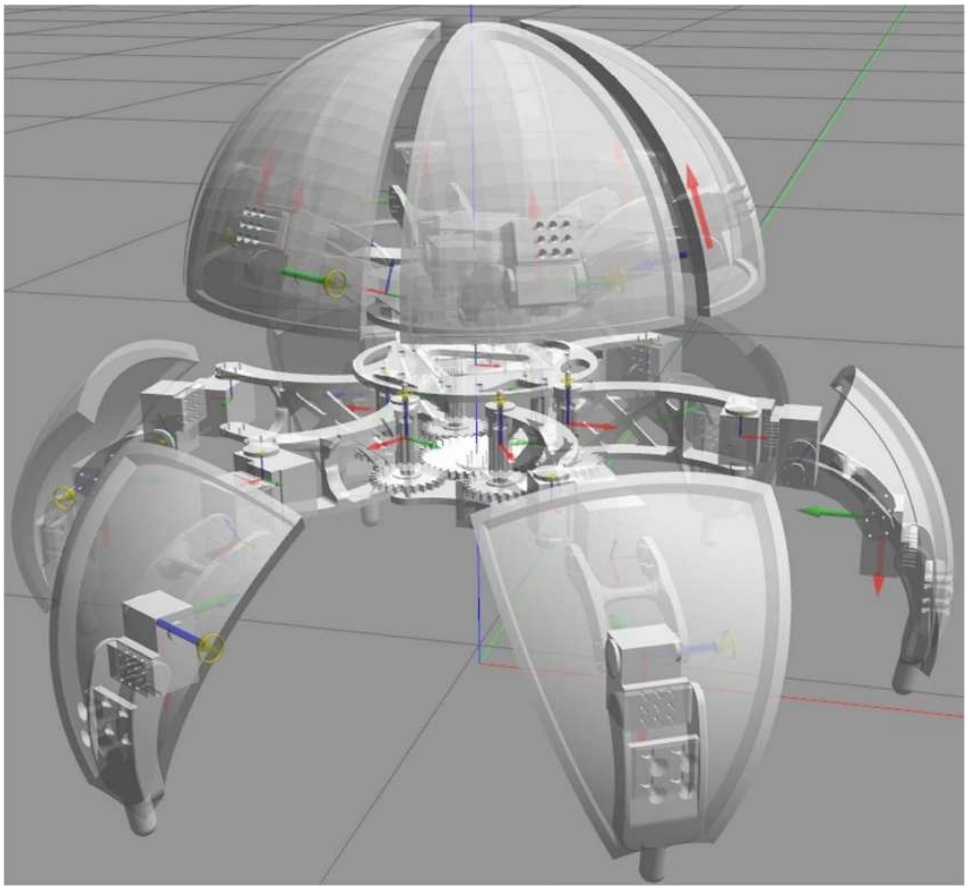
In configurazioni che non si allontanano eccessivamente dai punti prestabiliti il sistema si comporta comunque, con buona approssimazione , come N sistemi SISO , dove N è il numero di variabili di giunto da assegnare.

Il secondo sistema di controllo risulta una scatola nera , ossia non sappiamo come si comporti il sistema all'interno se non per la configurazione dei parametri PD.

Ciò accade poichè non siamo in grado di modellare sistemi complessi come il robot,

poichè esso è caratterizzato da inerzie variabili nel tempo a seconda della configurazione, configurazione variabile, contatti ecc ...

La simulazione viene infatti effettuata a prova di verifica che tutto ciò che non è possibile modellare con precisione funzioni come previsto (sotto dovute ipotesi).



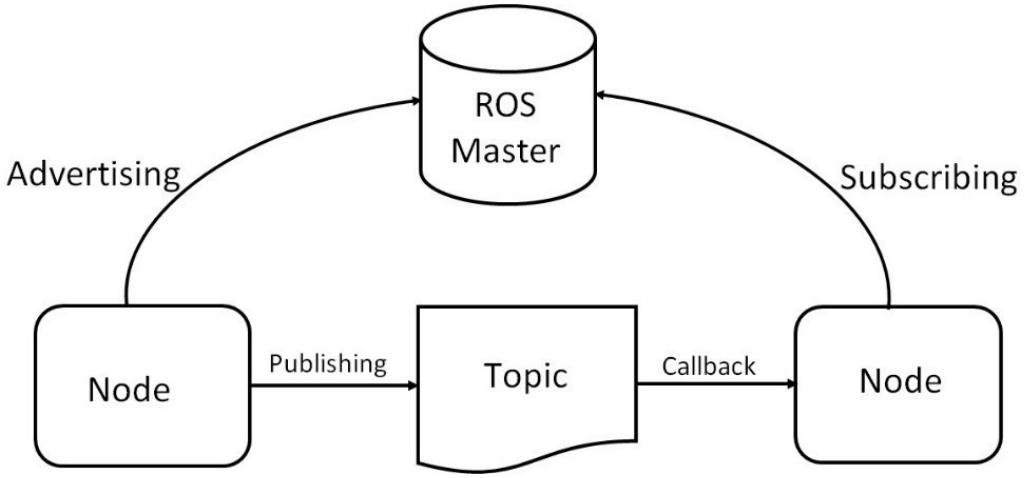
Si noti che mediante il Dynamic Reconfigure messo a disposizione da ROS è possibile variare in maniera dinamica i parametri del controllore PD e, in particolare la sua rigidezza data dal parametro P (si compora infatti come una molla caratterizzato da rigidezza K).

Ulteriori sviluppi di questo sistema potrebbero essere interessanti apportando alcune modifiche per ottenere ,mediante simulazione, una stima parametrica e di conseguenza rendere gli algoritmi di inseguimento di traiettorie robusti a variazioni parametriche o a settaggi imprecisi.

Ros

ros è un framework di svilupo robotico utile per interfacciarsi con vari programmi ed eseguire operazioni standard in ambito robotico. La versione da noi utilizzata e ros Noetics su ubuntu 20.04.

Nello sviluppo del robot sono stati utilizzati solo alcuni strumenti messi a disposizione da ros, in particolare nodi e topic, esclusivamente per far comunicare tra loro i vari componenti del programma, per il resto è stato tutto implementato " from scratch".



I nodi rappresentano programmi in esecuzione autonoma che possono interfacciarsi con alti programmi mediante topic, ossia argomenti sul quale vengono pubblicati dei messaggi, siano essi stringhe, numeri, array o immagini.

Tutto il programma, dall'interfaccia grafica al programma di calcolo si interfacciano mediante il sistema di topic e messaggi di ros. Ros infatti permette di inviare e ricevere messaggi mediante protocollo seriale tra vari programmi in esecuzione simultanea tramite un sistema ad eventi.

in questa maniera il programma è stato suddiviso in varie sezioni introperanti che fanno uso di diversi spazi di lavoro e che possono interfacciarsi con la simulazione.

La simulazione vine infatti controllata completamente dall'utente che invando comandi fa eseguire al sistema tutti i calcoli necessari per la cinematica inversa e invia successivamente gli angoli di giunto alla simulazione che, mediante la sua fisica seguira i riferimenti.

E' possibile vedere come i vari componenti dell'ecosistema si interfacciano mediante ros, in particolare è possibile vedere il grafo del sistema.

Tutti i componenti di un grafo ros si interfacciano con il nodo Master al quale vengono passate tutte le informazioni he scorrono all'interno dei canali di comunicazione tra i vari programma

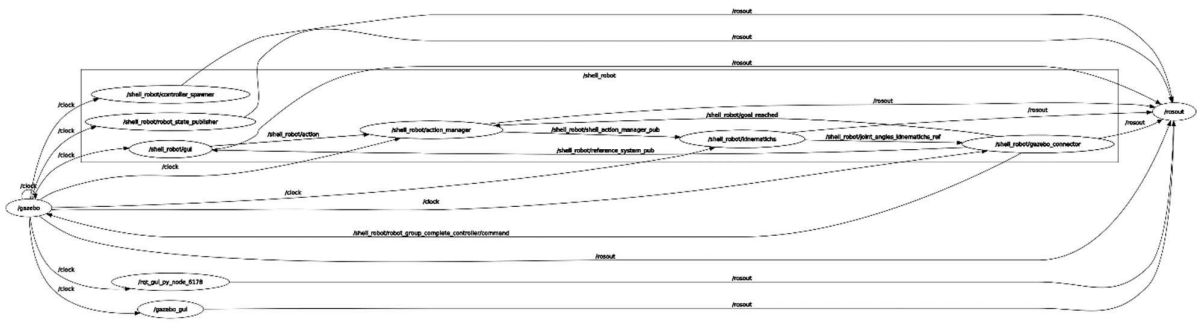
questo ci da una idea di come i vari componenti del sistema si interfacciano tra loro.

Una parte di ros fortemente utile è stata l'implementazione dei controllori del robot in simulazione. Il controllo avviene infatti imponendo dei riferimenti sugli angoli di giunto dati dalla cinematica inversa.

Questi riferimenti vengono poi passati ai controllori sui singoli giunti che effettuano un inseguimento del riferimento in coppia, ossia assegnando una posizione, il controllore calcola e impone un andamento di coppia adeguato per inseguire il riferimento .

I controllori scelti sono controllori PD , adatti in caso di applicazioni robotiche e che mantengono la stabilità dei singoli sistemi di attuazione.

Il grafo risultante dal sistema che è stato costruito è decisamente complesso e ha varie interconnessioni , sia con la simulazione, che tra i vari programmi. Si hanno inoltre una serie di connessioni interne di debug presenti nell'implementazione di ROS.



Interfaccia grafica e comandi

Per il controlo da parte dell’utente è stato deciso di utilizzare un controllo mediante computer, ciò porta diversi vantaggi:

- risparmio costo joystick
- controllo diretto su tutte le caratteristiche
- eventuale feedback visivo del robot
- Monitoraggio real-time

Di fatto questa implementazione diventa in stile “videogame” , l’idea infatti è quella di poter controllare il robot come si controllerebbe un personaggio in un videogioco.

l'interfaccia grafica del robot shell è composta da un terminale , un spazio di log , uno di selezione della modalità operativa , uno legato al controllo e uno di feedback di gazebo.

L'interfaccia infatti serve da controllere per il robot in simulazione, sono presenti una serie di comandi standard che permettono di effettuare movimenti in simulazione(ad esempio "e" per far ruotare il robot e "l" per farlo abbassare), e inoltre viene dato un feedback da ros che infoma il sistema di innterfaccia sulla poszione e eventuali rotazioni del robot in sumulazione e questi vengon orestituiti all'utente.

Il terminale permette di visionare le varie opzioni mediante comandi help che mostrano una guida generale all'interfaccia e "*keys*" che danno una guida ai comandi.

Il controllo mediante interfaccia avviene mediante i seguenti bindings da tastiera:

W : movimento in avanti

Q : rotazione a sinistra

E : rotazione a destra

S : movimento indietro

A : movimento laterale a sinistra

D : movimento laerale a destra

U: innalzamento floating base

I : ritorna in condizione idle

L : abbassamento floating base

Freccia direzionale destra : inclina floating base a destra

Freccia direzionale sinistra : inclina floating base a sinistra

Freccia direzionale su : inclina floating base in avanti

Freccia direzionale giu :inclina floating base indietro

Ogni comando inviato invia un log che indica che il comando è stato ricevuto correttamente. Per evitare di entrare in conflitto con il terminale le azioni possono essere inviate selezionando lo spazio nero (che dovrebbe ospitare una eventuale telecamera una volta che il robot sarà costruito).

Il tasto *exit* termina l'interfaccia (ma non la simulazione) mentre i tasti *shell/hexapod* servono per cambiare modalità e trasformare il robot nelle 2 configurazioni.

Struttura del catkin

Il catkin è un ambiente di sviluppo indipendete strutturato a package nel quale vengono inseriti e codificati i vari nodi e moduli.

I package vengono inseriti nella cartella src , nel caso del nostro catkin, denominato shellbot_ws , sono presenti vari package da noi creati :

- *shell_robot_kineamtichs*
- *shell_robot_gui*
- *shell_robot_gazebo_connector*
- *shell_robot_description*
- *shell_robot_control*
- *shell_robot_action_manager*
- *shell_robot_sym*

Shell_robot_gui si occupa della creazione della gui e del rispettivo nodo che permette di comuniare i vari comandi al resto del sistema

Shell_robot_action_manager traduce I comandi in azioni di base che il robot deve seguire che vengono inviati a *shell_robot kinematichs*

Shell_robot_kinematichs effettua i calcoli di cinematica inversa e passa i vari riferimenti al *Gazebo_connector*

Gazebo connector è un modulo implementato per connettere la simulazione al programma. Esso è stato concepito per essere utlizzato esclusivamente con la simulazione , è pero semplicemente sostituituibile nel caso di robot reale .

Questa costruzione permette di fatto di sostituire solo un package per connettere il robot reale al programma rispetto alla simulazione.

Gazebo connector riceve quindi i riferimenti dal *package kinematichs* e gli attuali angoli di giunto della simulazione da gazebo, questo permette di verificare, una volta che il riferimento è stato raggiunto, di passare al successivo. I riferimenti vengono inviati ai controllori che agiscono sulla simulazione.

Il modulo *Shell_robot_control* serve infatti per generare i controllori e *Shell_robot_description* contiene *l'URDF* e tutte le informazioni relative alla simulazione.

Tutto il sistema può essere avviato mediante il package *Shell_robot_sym* che ha al suo interno un file *.launch* che avvia tutto il sistema di simulazione aprendo in sequenza :

- programma di calcolo
- gui
- simulazione

Ogni package può funzionare in maniera completamente indipendete e svolgendo una specifica funzione. Questo permette di mantenere la modularità del codice.

A ogni package corrisponde un programma che assolve a una specifica funzione e comunica con gli altri programmi mediante *nodi* e *topic*.

E' di particolare il funzionamento non tanto come programmi singoli ma come ecosistema, tutti questi package insieme fanno infatti funzionare la simulazione nel suo complesso.

Riferimenti

Libreria Robopy - GitHub - ATLED-3301/robopy

Ros – https://www.ros.org/

Ubuntu - https://www.ubuntu-it.org/

Numpy - https://numpy.org/

Numba - https://numba.pydata.org/

Dispense robotica - http://www.dimnp.unipi.it/gabiccini-m/

Python - https://www.python.org/

Gazebo - http://gazebosim.org/

URDF - https://wiki.ros.org/urdf

Spiegazione urdf - https://www.youtube.com/watch?v=W-Y5lF0Uv3I

Shell robot URDF - GitHub - ATLED-3301/shell-robot-urdf

Joint in gazebo - http://gazebosim.org/tutorials/?tut=ros_urdf

Jupiter notebook - https://jupyter.org

Matplotlib grafici - https://matplotlib.org/stable/index.html

ODE - http://ode.org/ode-latest-userguide.html#sec_3_7_0

Gazebo/ode - http://gazebosim.org/tutorials?tut=physics_params&cat=physics#Overview

Morphex - https://www.youtube.com/watch?v=yn3FWb-vQQ4