



# UNIVERSITÀ DI PISA

CORSO DI MECCANICA DEI ROBOT

---

## Progetto *esapode sferico*

---



*Studenti*

Elia Zuccaro

Simone Mazzolini

Ilaria Martelli

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Modello del robot</b>	<b>4</b>
2.1	Schema del robot completo . . . . .	4
2.2	Modellazione CAD . . . . .	7
2.2.1	Disegno preliminare . . . . .	7
2.2.2	Ottimizzazione disegno . . . . .	7
2.2.3	Calcolo coppie ai motori . . . . .	11
2.3	Cinematica diretta . . . . .	14
2.3.1	Parte Inferiore . . . . .	14
2.3.2	Parte Superiore . . . . .	15
2.4	Cinematica Inversa . . . . .	17
2.4.1	End effector . . . . .	17
2.4.2	Soluzione manipolatore antropomorfo . . . . .	18
2.4.3	Centro della sfera inferiore . . . . .	20
2.4.4	Centro della sfera superiore . . . . .	23
2.5	Jacobiano . . . . .	25
2.5.1	Jacobiani analitici . . . . .	25
2.5.2	Jacobian geometrico . . . . .	26
2.6	Traiettorie . . . . .	28
2.7	Taratura controllori . . . . .	30
<b>3</b>	<b>Controllo degli attuatori</b>	<b>31</b>
3.1	Panoramica di comunicazione . . . . .	31
3.2	Dimensionamento batteria . . . . .	33
3.3	La scheda di controllo . . . . .	34
3.3.1	Hardware . . . . .	37
3.3.2	Rigenerazione della tensione ai servo . . . . .	39
3.4	Connessione con la Jetson . . . . .	40
3.5	Software per il controllo dei servo . . . . .	41
3.5.1	ax12.py . . . . .	41
3.5.2	servo.py . . . . .	42
3.5.3	pybullet_sim.py . . . . .	43
3.5.4	dynamixelUART.ino . . . . .	44
3.5.5	motor_control_ros.py . . . . .	44
<b>4</b>	<b>Software</b>	<b>45</b>
4.1	Mappa dei file . . . . .	47
4.1.1	Workspace del programma di simulazione . . . . .	47

4.2	Simulazione . . . . .	48
4.2.1	teleop_key . . . . .	49
4.2.2	trajectory_handler . . . . .	50
4.2.3	inverse_kinematic_server . . . . .	54
4.2.4	shellbot_kinematic_laws . . . . .	59
4.3	Modello del robot in simulazione . . . . .	59
4.4	Versione on-board . . . . .	60
4.4.1	motor_control_ros . . . . .	61
4.4.2	RT_inverse_kinematic_and_control . . . . .	63
4.4.3	ROS_connector . . . . .	65
<b>5</b>	<b>Problemi affrontati</b>	<b>67</b>
5.1	Daisy chain dei servo . . . . .	67
5.2	Il pin digitale . . . . .	68
5.3	Possibili soluzioni per sviluppi futuri . . . . .	70

## 1 Introduzione

In questa relazione viene presentato il lavoro svolto nel progetto "Esapode sferico", preso in carica per il corso di Meccanica dei Robot del corso di laurea in Ingegneria Robotica e dell'Automazione presso l'Università di Pisa.

L'obiettivo del progetto è stato quello di replicare un robot già esistente: in questo caso il robot da replicare è stato il MorpHex, un esapode in grado di assumere la forma di una sfera.

Qui un video del suo funzionamento:

[https://www.youtube.com/watch?v=yn3FWb-vQQ4t=48sab\\_channel=Zenta](https://www.youtube.com/watch?v=yn3FWb-vQQ4t=48sab_channel=Zenta)

Per la realizzazione del progetto è stato messo a disposizione un budget ed è stata concessa totale libertà su design e metodi realizzativi. L'assenza di linee guida prestabilite è stata un incentivo alla ricerca di soluzioni ai problemi affrontati di volta in volta, dalle fasi iniziali di progettazione a quelle finali di integrazione della parte simulativa con la parte hardware. In questa relazione vengono illustrati tutti gli aspetti dello svolgimento del progetto, dalla progettazione meccanica ed elettronica alla parte software, sia simulativa che on-board.

Lo scopo del progetto non è stato raggiunto nella sua interezza, dal momento che del robot è stato realizzato solo un prototipo, non completandone la realizzazione fisica. Sono stati infatti riscontrati problemi legati alla parte hardware, le cui possibili soluzioni avrebbero richiesto un profondo ripensamento di alcuni aspetti progettuali, e un conseguente impiego di tempi e risorse che si è ritenuto fossero al di fuori degli scopi di questo progetto.

Le possibili soluzioni sono esposte in questa relazione, in modo da fornire un punto di partenza a chi volesse partire dal punto di arrivo di questo progetto per portare a termine l'implementazione della parte hardware.

## 2 Modello del robot

### 2.1 Schema del robot completo

Il robot completo è costituito dalla *"somma"* di due esapodi, inferiore e superiore, entrambi strutture cinematiche parallele composte da sei catene seriali simmetriche. L'idea iniziale era quella di creare un robot simmetrico, in grado quindi di camminare con entrambe le metà. Tuttavia, il numero di motori necessari per fare ciò, e quindi il relativo peso, sarebbe stato eccessivo per il robot.

Si è preferito, perciò, fare un robot asimmetrico, costituito da una parte inferiore più complessa in grado di camminare e una parte superiore più semplice, che permetta al robot di aprirsi e chiudersi. Entrambe le metà sono formate da 6 gambe ciascuna e sono tali che, una volta chiuso, il robot risulti perfettamente simmetrico.

Ognuna delle gambe inferiori è descritta come un 4R spaziale, con il giunto iniziale - detto *"folding fan"* - comune a tutte le gambe e collegato ad essi tramite ruote dentate. Questo accorgimento permette di risparmiare 5 motori a discapito della mobilità, infatti il folding fan permette soltanto alle gambe inferiori di aprirsi e chiudersi contemporaneamente, impedendo qualsiasi movimento relativo fra di esse. Ognuna delle gambe superiori è invece descritta come un RR planare.

Di seguito è riportata la legenda dei sistemi di riferimento utilizzati con i relativi link mostrata in Fig. 1 (per quanto possibile si è cercato di rispettare la *convenzione di Denavit-Hartenberg*):

- Centro (grigio): origine del sistema di riferimento del robot, si trova al centro del ripiano centrale del robot; ha l'asse  $z$  rivolto verso l'alto e l'asse  $x$  in direzione della prima catena cinematica.
- Gambe inferiori:
  1. Link fisso (nero): link fittizio necessario per portare l'origine del sistema di riferimento a coincidere con l'inizio della prima gamba, ha ancora l'asse  $z$  verticale.
  2. Gamba 1 (rossa): corrisponde al folding fan e può ruotare attorno all'asse  $z_0$ .
  3. Gamba 2 (celeste): è costituita da un blocco formato da due motori e può ruotare attorno all'asse  $z_1$  (anch'esso verticale), tuttavia l'asse  $z_2$  è ruotato di  $90^\circ$  per allinearsi all'asse del motore successivo, secondo la convenzione di D-H.
  4. Gamba 3 (blu): è il primo giunto ad asse orizzontale ed è mosso dall'*AX-18* in quanto è il giunto più sollecitato durante il moto. In questo punto

si ha anche un’eccezione alla convenzione di D-H poiché è presente una traslazione lungo  $z$  nonostante gli assi degli ultimi giunti siano paralleli, tuttavia essendo fisicamente presente nel robot si è preferito inserirla in questo punto specifico.

5. End effector (verde): rappresenta la parte terminale del robot, dall’asse dell’ultimo giunto fino al tastatore.

- Gambe superiori:

1. Link fisso (nero): anche in questo caso si ha un link fittizio per arrivare all’inizio della catena cinematica, i sistemi di riferimento inferiori e superiori non sono allineati ma si ha una rotazione relativa lungo  $z$  necessaria per la chiusura.
2. Gamba superiore (arancione): link solidale al primo motore, il sistema di riferimento è posizionato sull’asse del secondo.
3. End effector superiore (giallo): è utile per la visualizzazione del modello, tuttavia fisicamente non esiste un end effector superiore in quanto il motore è direttamente collegato alla shell di copertura (vedi Par. [2.2.2](#)).

Oltre ai sistemi di riferimento indicati in legenda si devono aggiungere due terne, solidali agli end effector, che rappresentano posizione e orientazione del centro della shell inferiore (viola) e superiore (marrone). Queste terne saranno importanti in fase di apertura e chiusura del robot.

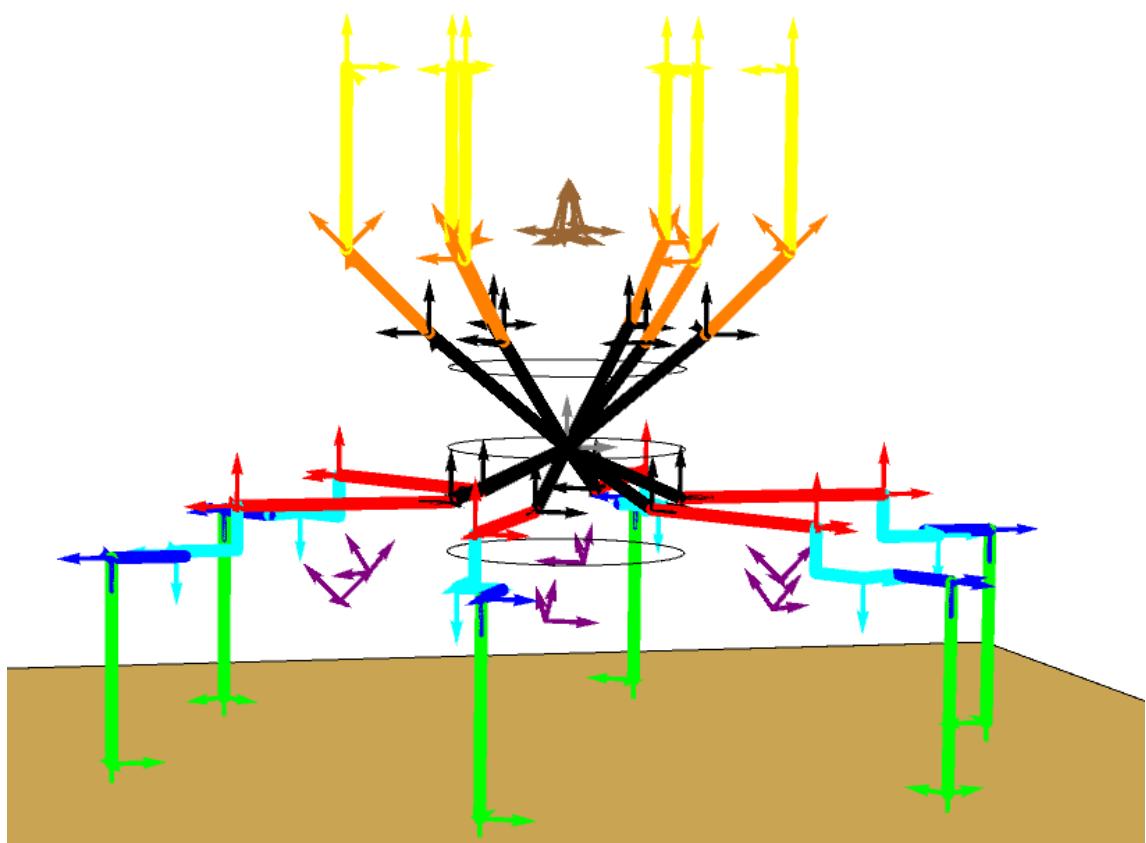


Figura 1: Schema completo del robot con link e sistemi di riferimento

## 2.2 Modellazione CAD

### 2.2.1 Disegno preliminare

La modellazione CAD è stata fatta utilizzando il programma *Autodesk Inventor Pro*. Per disegnare la struttura del robot si è partiti dagli elementi già noti e scelti nella fase preliminare del progetto, in particolare la scheda di controllo, una *Jetson Nano*, e i motori, *Dynamixel AX-12*. La prima ha determinato le dimensioni minime necessarie per l'alloggio centrale dei componenti (batteria e ingranaggi), i secondi sono necessari per conoscere forma e dimensioni dei link del robot.

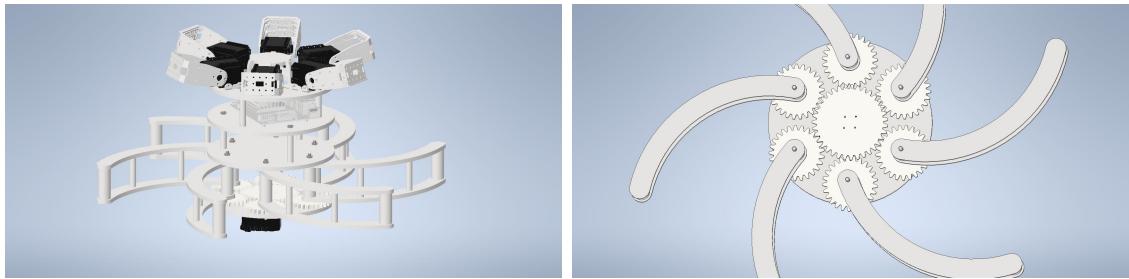


Figura 2: Prima ipotesi della struttura centrale del robot

Si ottiene così la parte centrale del robot mostrata in Fig. 2 formata da 4 ripiani (poi ridotti a 3) e gli ingranaggi del folding fan. Per rispettare la simmetria del robot il numero di denti negli ingranaggi deve essere divisibile per 6, inoltre la solare dovrà essere più grande degli altri ingranaggi. Anche il modulo delle ruote dentate presenta delle restrizioni, dovendo essere abbastanza grande da poter essere stampato in 3D ma abbastanza piccolo da non dare vita a ingranaggi troppo grandi.

Si decide, quindi, di utilizzare un modulo da 2 mm con ruote da 42 e 24 denti, ottenendo un rapporto di trasmissione di 1 : 1.75. Nonostante il rapporto di trasmissione sfavorevole e la necessità di muovere 6 gambe contemporaneamente, il motore centrale del robot non risulta molto sollecitato in quanto non ha carichi da vincere oltre all'inerzia stessa delle gambe e l'attrito fra ingranaggi e base.

Una volta ottenuta la parte centrale si prosegue la modellazione fino ad ottenere il robot completo mostrato in Fig. 3.

### 2.2.2 Ottimizzazione disegno

Il modello ottenuto è tutt'altro che definitivo. Infatti, con un peso ipotizzato di 3 kg, il robot pesa più di 6 kg a cui si devono aggiungere il peso della batteria, dei cavi e di tutti gli elementi unificati quali viti, bulloni e dadi. È ovvio che non sia possibile costruire un robot di queste dimensioni, ma si dovrà prima procedere a ridurne il peso senza comprometterne la rigidezza.

Progetto *esapode sferico*

---

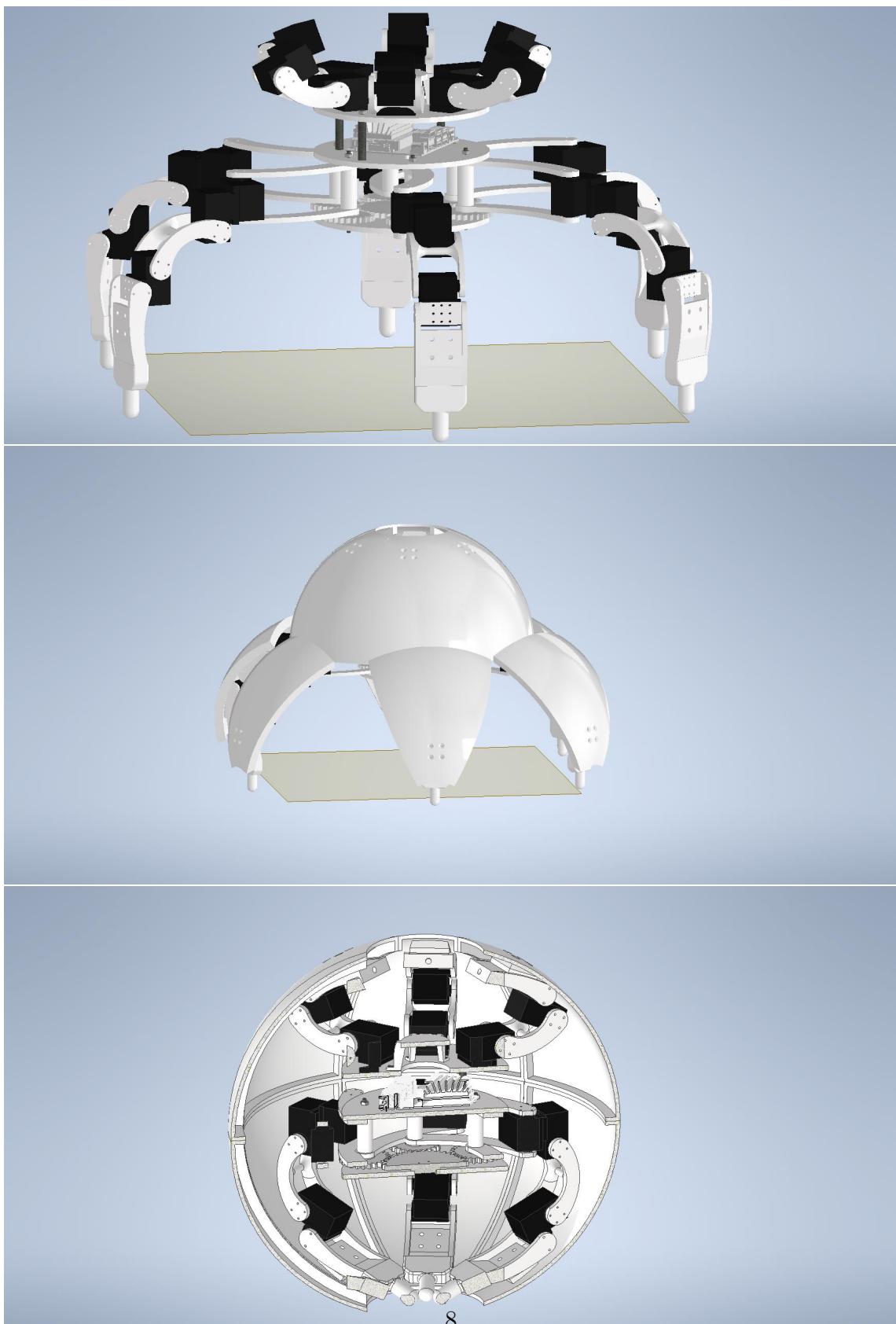


Figura 3: Struttura interna, robot completo in posizione di camminata e chiuso a palla (vista in sezione)

## Progetto esapode sferico

---

Per fare ciò si cerca per prima cosa di ridurre le dimensioni della sfera che ricopre il robot (che costituisce la maggior parte del peso) e di applicare la generazione di forme di Inventor. Questa funzione va a dividere i componenti in poligoni e, noti vincoli e carichi a cui saranno sottoposti, va a massimizzare la resistenza per componente riducendo il numero di poligoni di un valore fissato dall'utente.

Di seguito sono riportati i principali componenti che hanno subito modifiche. Oltre a questi si ha l'eliminazione dell'end effector superiore, collegando direttamente il secondo motore alla shell.

**Middle chassis** Il middle chassis è il componente su cui si trova montata la Jetson e presenta alcuni fori per il collegamento con la parte inferiore e superiore. Mentre la parte inferiore è simmetrica quella superiore presenta fori su una sola metà del cerchio, in quanto l'altra metà deve essere lasciata libera per le porte di collegamento della Jetson.

In questo caso la generazione di forme è molto semplice visto che il componente è simmetrico rispetto al proprio piano medio e può essere approssimato come una piastra circolare.

Il risultato mostra come alcuni fori in posizioni specifiche permettano di risparmiare sul peso dell'oggetto. Questi fori saranno utili inoltre per far passare i cavi di collegamento alla batteria e ai motori.

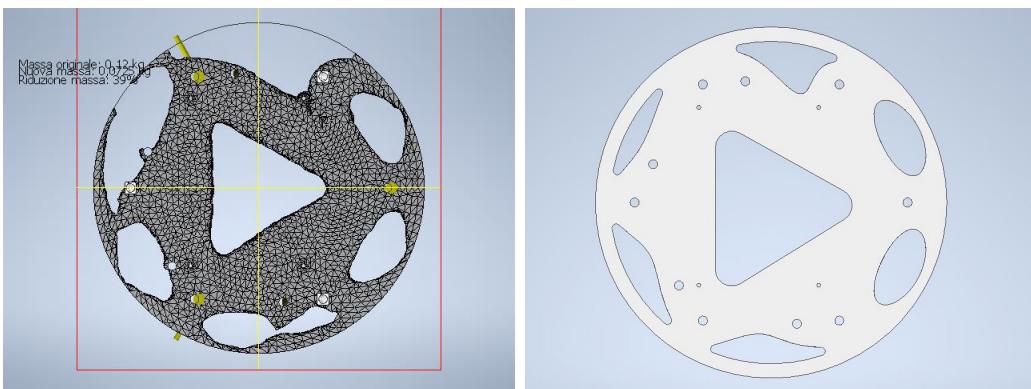


Figura 4: Generatore di forme e disegno finale del middle chassis

**Bottom chassis** Come il componente precedente anche il bottom chassis è simmetrico rispetto al piano medio. Su questo componente sono alloggiati gli ingranaggi, la batteria e da qui partono le gambe inferiori, sarà perciò più difficile rimuovere materiale da questo componente visto il gran numero di oggetti presenti. Il risultato mostra come la parte esterna della piastra sia scarica e per questo eliminabile.

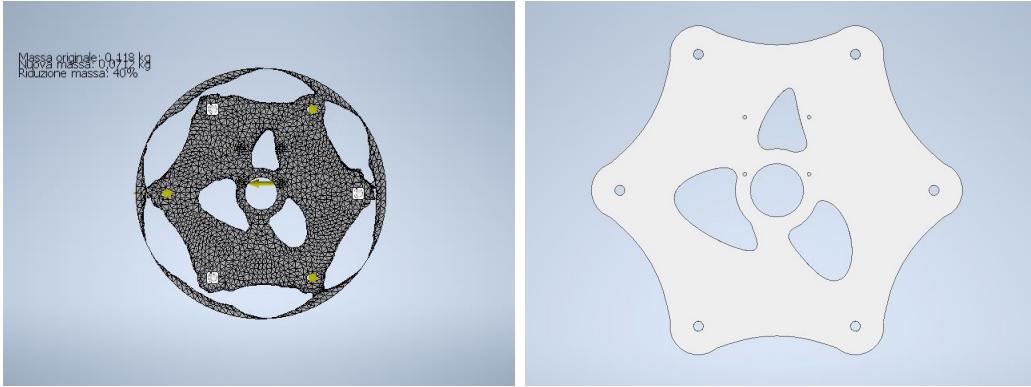


Figura 5: Forma finale del bottom chassis

**Gamba 1** Questo componente è sicuramente il più complesso e articolato di tutto il robot. Oltre ad essere uno dei componenti più grandi del robot è anche fra i più sollecitati (secondo solo alle barre filettate che però sono in acciaio), le sue dimensioni permettono al robot di aprirsi e avere lo spazio necessario per camminare.

Delle gambe completamente vuote come mostrate in Fig. 3 non sono sufficientemente rigide, al contrario se fossero completamente piene sarebbero troppo pesanti. Si è deciso perciò di riempirle e poi applicare la mesh generativa per calcolare le parti migliori da aggiungere in fase di disegno. Si è deciso inoltre di includere all'interno della gamba anche il pignone, in modo tale da ottenere un unico pezzo e non dover montare più componenti fra loro, così facendo anche la resistenza stessa del pezzo ne risente positivamente.

**Gamba 2 e gamba superiore** Nonostante non sia stato applicato il metodo della mesh generativa a queste due gambe è utile analizzarne brevemente il comportamento quando vengono sottoposte a carichi. Le due gambe presentano la stessa forma ma dimensioni differenti, per questo vengono analizzate insieme. I carichi principali analizzati sono 2:

- Il momento flettente dovuto alla coppia dei motori;
- Un carico di trazione o compressione dovuto ad errori di montaggio o imprecisioni nel pezzo stesso.

Nonostante il design molto semplice il componente risulta rigido se sottoposto a flessione lungo l'asse dei motori, ma molto elastico in direzione ortogonale. In questo modo un eventuale imprecisione nel montaggio non darà luogo ad elevate tensioni che potrebbero comprometterne la struttura. In Fig. 7 è mostrato il confronto fra i due carichi. A parità di coefficiente di sicurezza (circa 3.5) il componente ha una

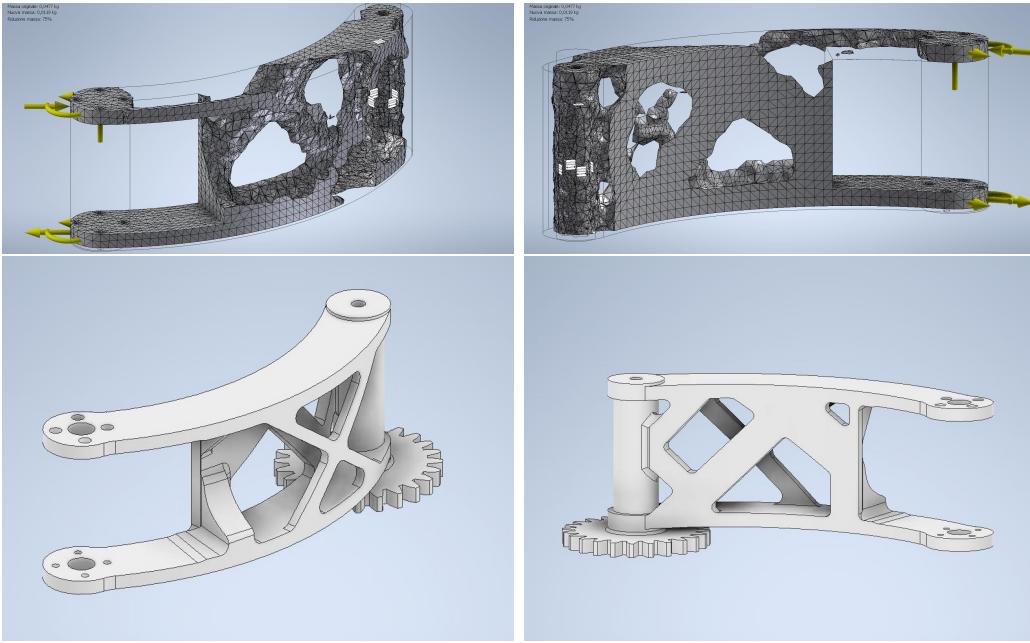


Figura 6: Gamba 1: risultato della mesh generativa (in alto) e disegno realizzativo (in basso).

deformazione 3 volte maggiore se sottoposto ad una trazione (o una compressione) in corrispondenza del montaggio dei due motori.

### 2.2.3 Calcolo coppie ai motori

A questo punto il robot ottenuto ha un peso totale di 3.8 kg, comprensivo anche della batteria, delle viti e di tutti gli altri elementi unificati. Per ottenere questo peso, oltre all'ottimizzazione degli elementi e la riduzione generale delle dimensioni del robot, si è considerata anche la riduzione di peso dovuta alla stampa 3D. Stampando in 3D, infatti, i componenti avranno una percentuale di vuoto al loro interno (detto infill) che li renderà più leggeri di un componente simile ma pieno. Questo valore è stato introdotto riducendo la densità del materiale fino a far corrispondere il peso dichiarato da Inventor con quello dato da Ultimaker Cura.

Oltre alla densità è stata ridotta anche la resistenza del materiale (in misura minore) sempre per lo stesso motivo.

Si vanno, quindi, a calcolare le coppie ai giunti necessarie per mantenere in equilibrio e muovere il robot.

Data la posizione di riposo in Fig. 3 è chiaro che il motore maggiormente sollecitato (e quindi critico per la stabilità del robot) sia il motore 3, cioè il primo motore ad

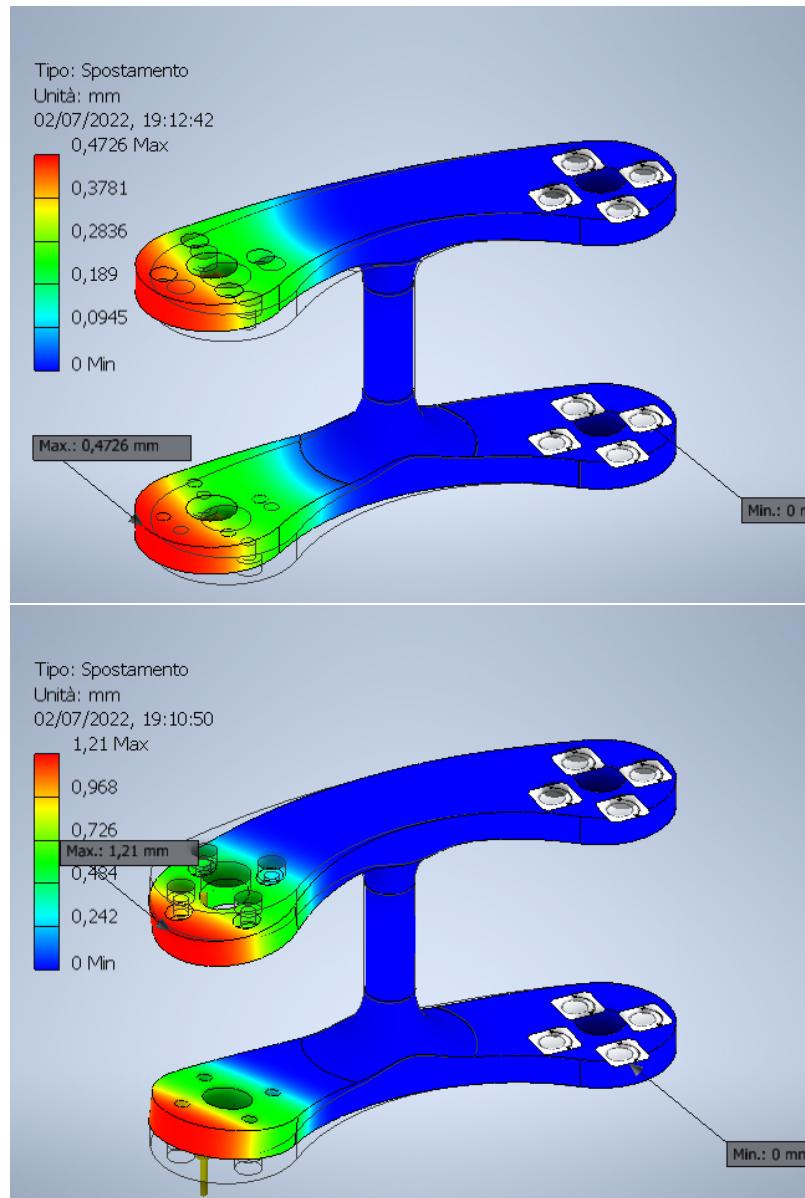


Figura 7: Confronto fra momento flettente (in alto) ed errore di montaggio (in basso) nella gamba 2.

asse orizzontale. Per calcolare il momento presente in quel punto si va a considerare il momento di maggiore carico: quando il robot ha 3 gambe alzate e sta cercando di effettuare un passo. In quell'istante tutto il peso del robot è distribuito su 3 punti di appoggio (in modo equo), mentre le altre 3 sono sollevate e si stanno muovendo. Dato che in linea di principio non è possibile calcolare le accelerazioni (e quindi le forze di inerzia) a cui sarà sottoposto il robot, si va a valutare il carico statico e si includono le forze di inerzia all'interno del coefficiente di sicurezza. C'è comunque da dire che in linea teorica il baricentro del robot non si dovrà alzare o abbassare durante la camminata, anche se non è possibile escludere che ciò accada.

Si suppone che nel punto di contatto fra l'end effector e il terreno si scarichi  $1/3$  del peso del robot. Poiché non è possibile calcolare la distribuzione del peso  $p(x)$  della gamba, si fanno due ipotesi, una cautelativa e una anticautelativa.

Nel primo caso si suppone che il peso della gamba sia trascurabile rispetto alla forza di contatto fra il robot e il terreno, in questo modo il momento necessario per tenere in piedi il robot sarà  $\frac{Mg}{3}l$ .

Nel secondo caso si considera l'intero peso  $P$  della gamba concentrato nel punto di contatto fra end effector e terreno, che si andrà quindi a sottrarre alla forza di contatto ottenendo:  $(\frac{Mg}{3} - P)l$ .

Sapendo che  $P = 0.27 \text{ kg}$  e  $l = 50 \text{ mm}$  otteniamo:

$$5.98 \text{ kg} \times \text{cm} \leq M_m \leq 6.33 \text{ kg} \times \text{cm}$$

Il valore ottenuto è di poco superiore ad  $\frac{1}{3}$  della coppia massima del motore ( $15 \text{ kg} \times \text{cm}$ ) valore considerato sicuro per un "utilizzo continuativo". Considerato che in posizione di riposo tutte e 6 le zampe saranno appoggiate a terra (e quindi la coppia si dimezzera), si può considerare questo valore come sicuro per i motori e per il robot.

La coppia massima può essere comunque raggiunta per pochi istanti senza danni per il motore, ma in via progettuale si è considerato di non superare mai la metà di quel valore.

## 2.3 Cinematica diretta

Per creare la struttura matematica del robot si è utilizzato *Wolfram Mathematica* in quanto molto adatto a lavorare con il calcolo simbolico, in seguito il codice è stato spostato su Matlab e in simulazione, per valutare la bontà delle equazioni ottenute, e infine è stato implementato sul robot.

### 2.3.1 Parte Inferiore

Per prima cosa si crea la tabella dei valori di DH, da utilizzare per creare la matrice omogenea di rototraslazione fra l'origine del robot e l'end effector.

Link	$\alpha$	$a$	$\theta$	$d$
0	0	$\frac{d_0}{2}$	$(k - 1)\frac{\pi}{3}$	$h_0$
1	0	$l_1$	$\theta_1$	0
2	$-\frac{\pi}{2}$	$l_2$	$\theta_2$	$h_2$
3	0	$l_3$	$\theta_3$	$h_3$
4	0	$l_4$	$\theta_4$	0

In tabella si possono riconoscere i 4 angoli di giunto  $\theta_i$ , l'angolo fittizio  $k\frac{\pi}{3}$  che rappresenta la rotazione fra una gamba e l'altra ( $k$  è il numero della gamba e va da 1 a 6) e le varie distanze  $h_i$  ed  $l_i$ .

In particolare  $h_3$  non sarebbe necessario per la convenzione di DH in quanto i due assi  $z_2$  e  $z_3$  sono paralleli, tuttavia è stato introdotto per portare l'end effector calcolato a coincidere con quello reale. Come vedremo più avanti questo valore creerà difficoltà nel calcolo della cinematica inversa.

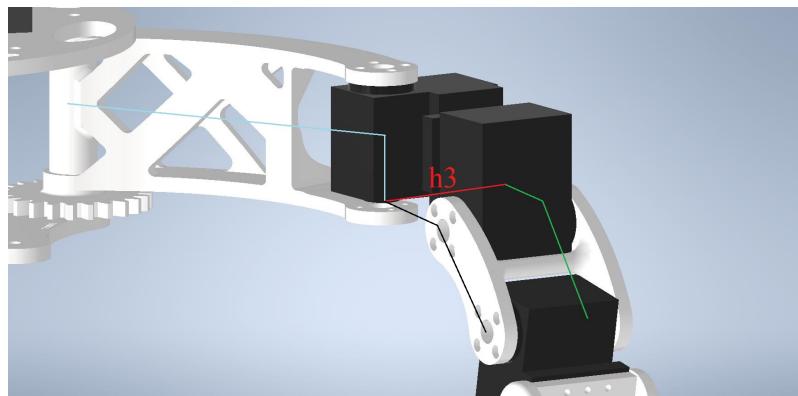


Figura 8: Il parametro  $h_3$  sposta la catena cinematica da quella teorica di DH (in nero) a quella reale del robot (in verde)

Una volta ottenuta la matrice omogenea di rototraslazione fra origine ed end effector, si applica il comando *FullSimplify* che permette di raggruppare i termini simili e semplificare le espressioni. È un comando molto potente ma impiega del tempo per completare i calcoli, per alleggerire il sistema si è deciso di salvare i risultati in un file *.m* e caricarli già semplificati negli altri notebook.

Per ottenere le coordinate del centro del robot si applica prima una traslazione pura lungo  $x$  ed  $y$  per arrivare al punto di contatto fra end effector e shell, ma sempre con il sistema di riferimento allineato all'end effector, e poi due rototraslazioni che portano il sistema di riferimento al centro della sfera descritta dalla shell, con asse  $z$  perpendicolare al piano di divisione fra le due metà e asse  $x$  diretto verso il centro dell'arco di circonferenza che delimita la shell.

In Fig. 9 sono rappresentati i sistemi di riferimento utilizzati e i valori numerici degli angoli alpha e beta.

La tabella di DH che si ottiene per le due rototraslazioni è:

<i>Link</i>	$\alpha$	$a$	$\theta$	$d$
5	0	$r1$	<i>alpha</i>	0
6	$\frac{\pi}{2}$	0	$\frac{\pi}{2} + \text{beta}$	0

### 2.3.2 Parte Superiore

Per la metà superiore le matrici si applica la stessa procedura, ottenendo delle matrici molto più semplici in quanto si hanno soltanto 2 giunti attuati anziché 4.

La tabella di DH in questo caso è:

<i>Link</i>	$\alpha$	$a$	$\theta$	$d$
0	$\frac{\pi}{2}$	$\frac{d_1}{2}$	$(k - 1)\frac{\pi}{3} + \text{gamma}$	$h_1$
1	0	$l_{s1}$	$\theta_{s1}$	0
2	0	$l_{s2}$	$\theta_{s2}$	0

Nuovamente si applica un traslazione pura e successivamente:

<i>Link</i>	$\alpha$	$a$	$\theta$	$d$
3	0	$r2$	<i>delta</i>	0
4	$-\frac{\pi}{2}$	0	$\frac{\pi}{2} + \text{epsilon}$	0

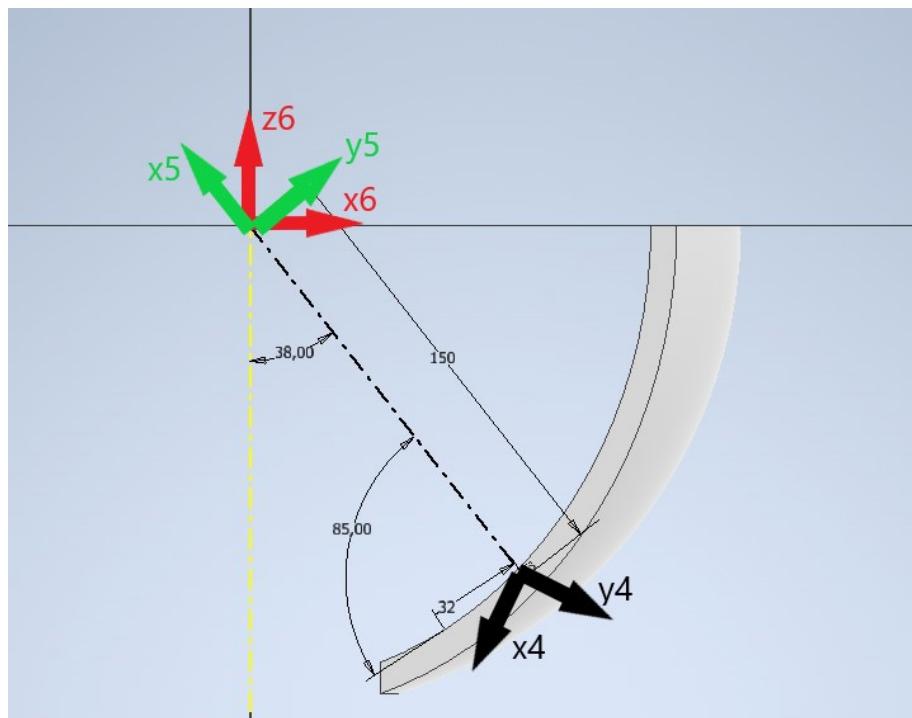


Figura 9: Angoli e sistemi di riferimento utilizzati per arrivare ad  $S_6$

L'angolo *gamma* permette il disallineamento fra la parte inferiore e quella superiore, è un angolo necessario per il montaggio della Jetson e inoltre risulta utile per la chiusura, in quanto le gambe non si chiuderanno necessariamente con un angolo di 60 ma saranno leggermente spostate.

## 2.4 Cinematica Inversa

Il calcolo della cinematica inversa risulta particolarmente importante per poter controllare in posizione i motori, tuttavia avendo più end effector a seconda della situazione (punta delle zampe e centro della sfera) si dovranno analizzare singolarmente e implementare lo switch da un sistema all'altro nel robot reale e simulato.

### 2.4.1 End effector

Per calcolare la cinematica inversa del robot si possono utilizzare la terza riga e la terza colonna della matrice omogenea  $T_4$  (per la visualizzazione delle matrici complete si rimanda ai notebook *D\_H algebrico.nb* e *Cinematica inversa.nb*):

$$T_4(3,:) = \begin{bmatrix} -\sin(\theta_3 + \theta_4) & -\cos(\theta_3 + \theta_4) & 0 & h_0 + h_2 - l_3 \sin(\theta_3) - l_4 \sin(\theta_3 + \theta_4) \end{bmatrix}$$

$$T_4(:,3) = \begin{bmatrix} \cos(\theta_1 + \theta_2 + \frac{1}{6}(2\pi k + \pi)) \\ \sin(\theta_1 + \theta_2 + \frac{1}{6}(2\pi k + \pi)) \\ 0 \\ 0 \end{bmatrix}$$

Dagli elementi (3,1) e (3,2) si può calcolare  $\theta_3 + \theta_4$ , nota la somma si calcola  $\theta_3$  dall'elemento (3,4) e infine  $\theta_4$  come differenza fra i due. L'algoritmo ottenuto è il seguente:

$$\begin{cases} \text{sum}_{34} = \arctan(-T_4(3,1), -T_4(3,2)) \\ \theta_3 = \arcsin\left(\frac{h_0 + h_2 - l_4 \sin(\text{sum}_{34}) - T_4(3,4)}{l_3}\right) \\ \theta_4 = \text{sum}_{34} - \theta_3 \end{cases}$$

Lo stesso algoritmo è applicabile per gli angoli  $\theta_1$  e  $\theta_2$  considerando l'elemento

$$\begin{aligned} T_4(1,4) &= \frac{1}{2}d_0 \sin\left(\frac{1}{6}(2\pi k + \pi)\right) \\ &+ h_3 \cos\left(\theta_1 + \theta_2 + \frac{1}{6}\pi(2k + 1)\right) + l_1 \sin\left(\theta_1 + \frac{1}{6}(2\pi k + \pi)\right) \\ &+ \sin\left(\theta_1 + \theta_2 + \frac{1}{6}\pi(2k + 1)\right) (l_2 + l_3 \cos(\theta_3) + l_4 \cos(\theta_3 + \theta_4)) \end{aligned}$$

e risolvendo prima per  $\theta_1$  e successivamente per  $\theta_2$ .

La soluzione trovata, per quanto elegante, non può essere utilizzata nella pratica. Infatti, le traiettorie che verranno date al robot avranno come input le posizioni  $x$ ,  $y$  e  $z$  dell'end effector, e non gli angoli di orientazione degli assi locali rispetto alla terna fissa. Per questo motivo bisogna cercare una soluzione differente che metta in relazione gli angoli di giunto con la quarta colonna della matrice omogenea.

Si cerca di semplificare il problema imponendo  $\theta_1 = 0$ . Sappiamo, infatti, che durante la camminata il robot dovrà essere il più aperto possibile per evitare collisioni fra le gambe, questa condizione è soddisfatta tenendo proprio  $\theta_1 = 0$ .

#### 2.4.2 Soluzione manipolatore antropomorfo

Si cerca di riportare il problema al manipolatore antropomorfo descritto nel libro *Robotica: modellistica, pianificazione e controllo* di Bruno Siciliano, Lorenzo Sciacchetta, Luigi Villani e Giuseppe Oriolo. Per farlo bisognerà prima semplificare la matrice in modo da togliere quei termini che ne complicano la soluzione.

Consideriamo il caso  $\theta_1 = 0$  e  $k = 1$  (si risolve per la prima gamba e per simmetria si ottengono le altre 5):

$$T_4(:, 1 : 3) = \begin{bmatrix} \cos(\theta_2) \cos(\theta_3 + \theta_4) & -\cos(\theta_2) \sin(\theta_3 + \theta_4) & -\sin(\theta_2) \\ \sin(\theta_2) \cos(\theta_3 + \theta_4) & -\sin(\theta_2) \sin(\theta_3 + \theta_4) & \cos(\theta_2) \\ -\sin(\theta_3 + \theta_4) & -\cos(\theta_3 + \theta_4) & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$T_4(:, 4) = \begin{bmatrix} \frac{d_0}{2} - h_3 \sin(\theta_2) + l_1 + \cos(\theta_2)(l_2 + l_3 \cos(\theta_3) + l_4 \cos(\theta_3 + \theta_4)) \\ h_3 \cos(\theta_2) + \sin(\theta_2)(l_2 + l_3 \cos(\theta_3) + l_4 \cos(\theta_3 + \theta_4)) \\ h_0 + h_2 - l_3 \sin(\theta_3) - l_4 \sin(\theta_3 + \theta_4) \\ 1 \end{bmatrix}$$

Possiamo notare che se non fosse presente il termine  $h_3$  la cinematica inversa avrebbe una forma più semplice e si potrebbe calcolare  $\theta_2$  dal rapporto degli elementi (1,4) e (2,4) della matrice.

Si creano, quindi, due matrici  $L$  ed  $R$  che rappresentano due traslazioni rigide, la

prima dall'origine del robot al giunto del primo motore e la seconda che vada ad annullare il contributo di  $h_3$ :

$$L = \begin{bmatrix} 1 & 0 & 0 & \frac{d_0}{2} + l_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_0 + h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -h_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Andiamo quindi a calcolare  $S = L^{-1}T_4R$  che rappresenta la matrice semplificata della catena cinematica.

$$S(:, 1 : 3) = \begin{bmatrix} \cos(\theta 2) \cos(\theta 3 + \theta 4) & -\cos(\theta 2) \sin(\theta 3 + \theta 4) & -\sin(\theta 2) \\ \sin(\theta 2) \cos(\theta 3 + \theta 4) & -\sin(\theta 2) \sin(\theta 3 + \theta 4) & \cos(\theta 2) \\ -\sin(\theta 3 + \theta 4) & -\cos(\theta 3 + \theta 4) & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$S(:, 4) = \begin{bmatrix} \cos(\theta 2)(l_2 + l_3 \cos(\theta 3) + l_4 \cos(\theta 3 + \theta 4)) \\ \sin(\theta 2)(l_2 + l_3 \cos(\theta 3) + l_4 \cos(\theta 3 + \theta 4)) \\ -l_3 \sin(\theta 3) - l_4 \sin(\theta 3 + \theta 4) \\ 1 \end{bmatrix}$$

Si può notare che adesso nella quarta colonna di S non compare più il termine  $h_3$ , possiamo perciò calcolare:

$$\theta_2 = \arctan \left( \frac{S(2, 4)}{S(1, 4)} \right)$$

che diventerà quindi

$$\theta_2 = \arctan \left( \frac{y}{x} \right)$$

dove  $x$  e  $y$  sono le coordinate desiderate dell'end effector. Nella pratica si usa la funzione *ArcTan* di Mathematica che tiene conto anche del quadrante in cui ci troviamo.

Per calcolare  $\theta_4$  sfruttiamo:

$$(S(1, 4) - l_2 \cos(\theta_2))^2 + (S(2, 4) - l_2 \sin(\theta_2))^2 + (S(3, 4))^2 = l_3^2 + l_4^2 + 2l_3l_4 \cos(\theta_4)$$

noto  $\theta_2$  otteniamo quindi:

$$\theta_4 = \arccos \left( \frac{(x - l_2 \cos(\theta_2))^2 + (y - l_2 \sin(\theta_2))^2 + (z)^2 - l_3^2 - l_4^2}{l_3l_4} \right)$$

Infine dall'espressione:

$$\begin{aligned} & \left[ (S(1, 4) - l_2 \cos(\theta_2))^2 + (S(2, 4) - l_2 \sin(\theta_2))^2 \right] (l_3 + l_4 \cos(\theta_4)) - (S(3, 4)) (l_4 \sin(\theta_4)) \\ & = (l_3^2 + l_4^2 + 2l_3 l_4 \cos(\theta_4)) \cos(\theta_3) \end{aligned}$$

si ottiene

$$\theta_3 = \arccos \left( \frac{\left[ (x - l_2 \cos(\theta_2))^2 + (y - l_2 \sin(\theta_2))^2 \right] (l_3 + l_4 \cos(\theta_4)) - (z) (l_4 \sin(\theta_4))}{l_3^2 + l_4^2 + 2l_3 l_4 \cos(\theta_4)} \right)$$

La soluzione è quella cercata a meno di una traslazione rigida e mette in relazione gli angoli di giunto con la posizione dell'end effector in modo univoco.

C'è da dire che il calcolo necessario per ottenere questo risultato non è semplice e abbiamo preferito non utilizzare questo metodo nel robot reale perché andrebbe a rallentare eccessivamente la CPU della Jetson, non permettendo di svolgere tutti i task necessari al controllo del robot. Rimane comunque un metodo valido per il calcolo delle traiettorie "offline", quando i tempi di calcolo non sono un problema. Per l'implementazione sul robot si utilizzerà la CLIK, molto più leggera dal punto di vista computazionale, anche se saranno necessari alcuni accorgimenti per discretizzare il sistema.

#### 2.4.3 Centro della sfera inferiore

Cercare una soluzione in forma chiusa per il centro della sfera del robot è ancora più complesso dato che i parametri *alpha* e *beta* ne complicano la soluzione. Esiste però una condizione in cui questa esiste e si può calcolare: quando il robot è chiuso a sfera.

Per chiudere il robot a sfera sono necessarie 4 condizioni:

$$\begin{cases} x = 0 \\ y = 0 \\ \Phi = 2\pi \\ \Theta = \text{gamma} - \frac{\pi}{3} \end{cases}$$

Dove  $\Theta$  e  $\Phi$  sono rispettivamente gli angoli di *yaw* e *pitch* e valgono  $\Theta = \theta_1 + \theta_2$  e  $\Phi = \text{alpha} + \text{beta} + \theta_3 + \theta_4$ .

Inizio risolvendo le prime due espressioni (vedi *Cinematica inversa.nb*). Dato che il termine

$$l_2 + l_3 \cos(\theta_3) + \text{xinf} \cos(\theta_3 + \theta_4) - \text{yinf} \sin(\theta_3 + \theta_4) + r_1 \cos(\text{alpha} + \theta_3 + \theta_4)$$

compare in entrambe le espressioni chiamo *mov* quel termine e *somma* la somma degli angoli  $\theta_1$  e  $\theta_2$ . Il sistema diventa quindi:

$$\begin{cases} h_3 \cos(\text{somma}) + l_1 \sin(\theta_1) + \text{mov} \sin(\text{somma}) = 0 \\ \frac{d_0}{2} - h_3 \sin(\text{somma}) + l_1 \cos(\theta_1) + \text{mov} \cos(\text{somma}) = 0 \end{cases}$$

Si ottengono due soluzioni:

$$\begin{aligned} \text{somma} &= 2 \arctan \left( \frac{-h_3 \sqrt{\frac{h_3^2 - l_1^2 \sin^2(\theta_1) + \text{mov}^2}{(h_3 - l_1 \sin(\theta_1))^2}} + l_1 \sin(\theta_1) \sqrt{\frac{h_3^2 - l_1^2 \sin^2(\theta_1) + \text{mov}^2}{(h_3 - l_1 \sin(\theta_1))^2}} + \text{mov}}{h_3 - l_1 \sin(\theta_1)} \right) \\ \text{somma} &= 2 \arctan \left( \frac{h_3 \sqrt{\frac{h_3^2 - l_1^2 \sin^2(\theta_1) + \text{mov}^2}{(h_3 - l_1 \sin(\theta_1))^2}} - l_1 \sin(\theta_1) \sqrt{\frac{h_3^2 - l_1^2 \sin^2(\theta_1) + \text{mov}^2}{(h_3 - l_1 \sin(\theta_1))^2}} + \text{mov}}{h_3 - l_1 \sin(\theta_1)} \right) \end{aligned}$$

con

$$\text{mov} = \frac{1}{2} \sqrt{d_0^2 + 4d_0 l_1 \cos(\theta_1) - 4h_3^2 + 4l_1^2 \sin^2(\theta_1) + 4l_1^2 \cos^2(\theta_1)}$$

Riprendendo la definizione di *mov* scritta in precedenza si può scrivere il sistema di equazioni:

$$\begin{cases} \text{mov} = l_2 + l_3 \cos(\theta_3) + x_{\text{inf}} \cos \theta_3 + \theta_4 - y_{\text{inf}} \sin(\theta_3 + \theta_4) + r_1 \cos(\alpha + \theta_3 + \theta_4) \\ \text{mov} = \frac{1}{2} \sqrt{d_0^2 + 4d_0 l_1 \cos(\theta_1) - 4h_3^2 + 4l_1^2 \sin^2(\theta_1) + 4l_1^2 \cos^2(\theta_1)} \\ \theta_3 + \theta_4 + \alpha + \beta = 2\pi \end{cases}$$

che ha come soluzione:

$$\begin{cases} \theta_3 = \pm \arccos \left( \frac{-2l_2 + 2x_{\text{inf}} \sin(\alpha + \beta) - 2y_{\text{inf}} \cos(\alpha + \beta) + 2r_1 \sin(\beta) - \sqrt{d_0^2 + 4d_0 l_1 \cos(\theta_1) - 4h_3^2 + 4l_1^2}}{2l_3} \right) \\ \theta_4 = 2\pi - \alpha - \beta - \theta_3 \end{cases}$$

Per valutare quale delle 4 soluzioni ottenute è quella corretta si vanno a disegnare i grafici delle espressioni precedenti in funzione di  $\theta_1$ .

Per quanto riguarda il grafico di  $\theta_1 + \theta_2$  si può notare come le due soluzioni trovate siano in realtà quattro metà di due soluzioni continue che si intersecano nel punto  $\theta_1 = -\arcsin \sqrt{\frac{h_3^2 + \text{mov}^2}{l_1^2}}$ . In quel punto le soluzioni coincidono.

Dai grafici di  $\theta_3$  e  $\theta_4$ , invece, otteniamo le due soluzioni "gomito alto" e "gomito basso". Possiamo notare che la soluzione gomito basso è quella in cui  $\theta_3$  è negativo, in questa configurazione, infatti, il braccio del robot tenderà ad andare prima verso l'alto e poi verso il basso con l'end effector. Il braccio sarà, quindi, più schiacciato verso il centro del robot, riducendo lo spazio interno alla sfera.

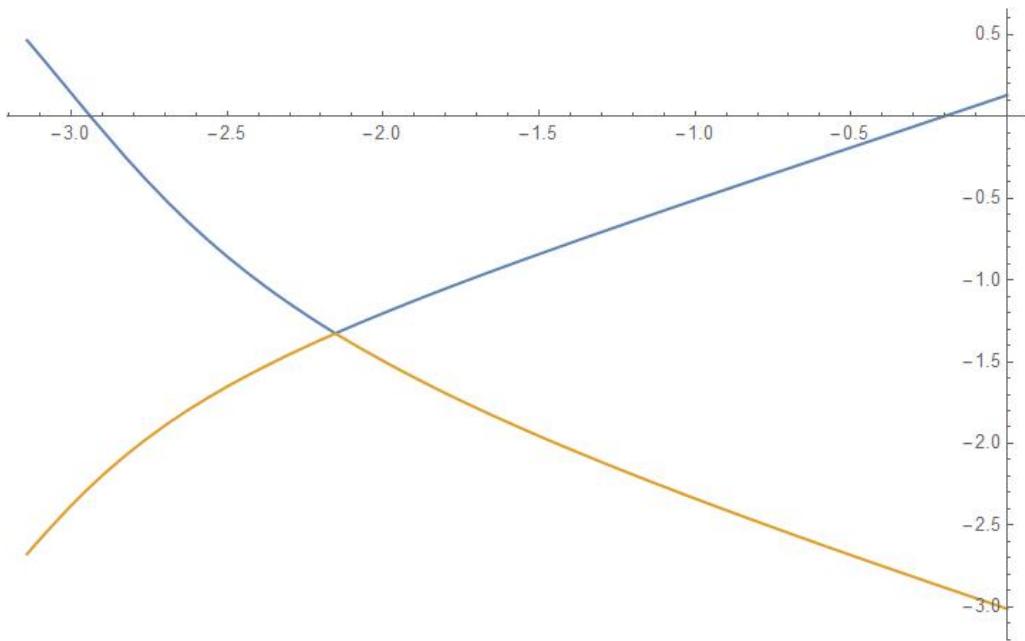


Figura 10: Andamento delle due soluzioni per  $\theta_1 + \theta_2$  al variare di  $\theta_1$ .

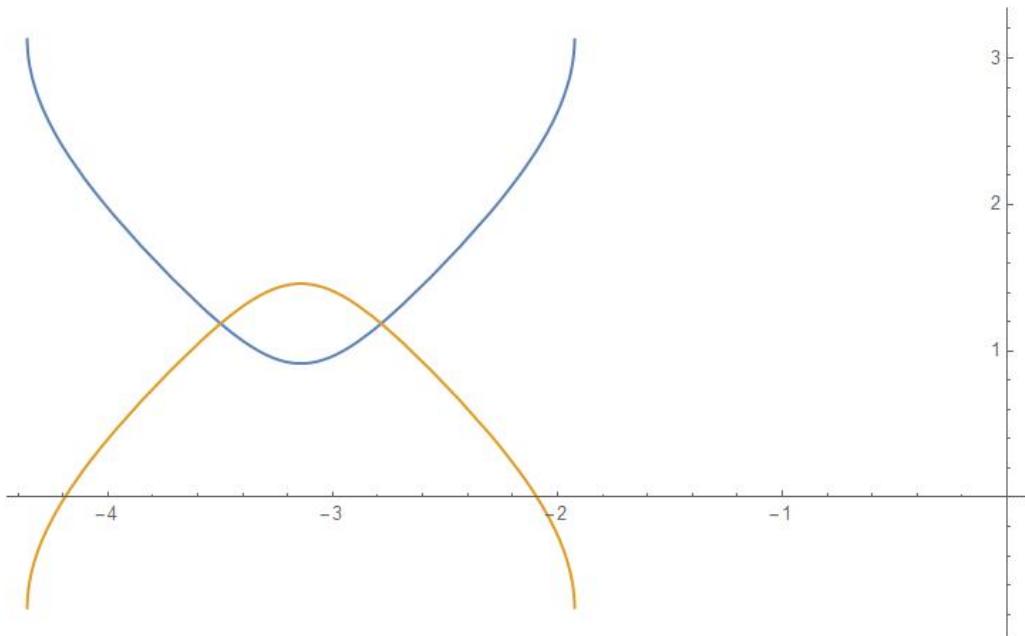


Figura 11: Soluzione "gomito basso" per la chiusura del robot ( $\theta_3$  in blu e  $\theta_4$  in arancione)

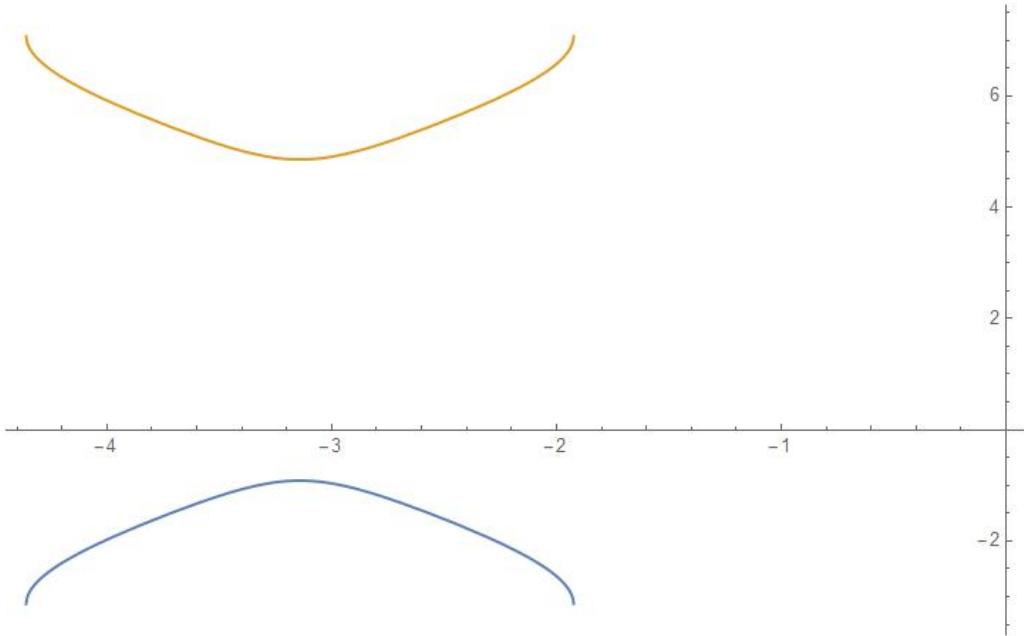


Figura 12: Soluzione "gomito alto" per la chiusura del robot ( $\theta_3$  in blu e  $\theta_4$  in arancione), in questo caso la gamba del robot è molto più distesa

Questa soluzione non è accettabile perché la shell inferiore andrebbe a collidere con quella superiore.

La soluzione corretta per la chiusura è quella a gomito alto mostrata in Fig. 12.

#### 2.4.4 Centro della sfera superiore

La shell superiore è formata da sei RR planari. Ponendoci nel piano su cui si muove il centro della sfera si ottiene un'espressione della matrice omogenea particolarmente semplice:

$$T_{s2}(:, 1 : 3) = \begin{bmatrix} -\sin(\delta + \epsilon + \theta_1 + \theta_2) & 0 & -\cos(\delta + \epsilon + \theta_1 + \theta_2) \\ 0 & 1 & 0 \\ \cos(\delta + \epsilon + \theta_1 + \theta_2) & 0 & -\sin(\delta + \epsilon + \theta_1 + \theta_2) \\ 0 & 0 & 0 \end{bmatrix}$$

$$T_{s2}(:, 4) = \begin{bmatrix} \frac{d_1}{2} + r_2 \cos(\delta + \theta_1 + \theta_2) + l_1 \cos(\theta_1) + x_{sup} \cos(\theta_1 + \theta_2) - y_{sup} \sin(\theta_1 + \theta_2) \\ 0 \\ r_2 \sin(\delta + \theta_1 + \theta_2) + h_1 + l_1 \sin(\theta_1) + x_{sup} \sin(\theta_1 + \theta_2) + y_{sup} \cos(\theta_1 + \theta_2) \\ 1 \end{bmatrix}$$

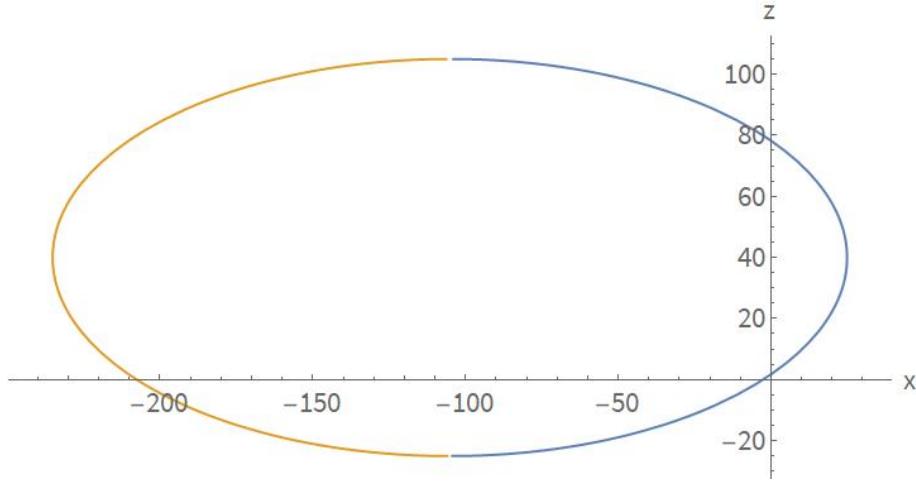


Figura 13: Luogo dei punti descritti dal centro della sfera quando la shell è verticale

Si scelgono come parametri di input l'inclinazione e la posizione lungo l'asse  $z$  del centro della sfera. Chiamiamo  $\psi$  l'angolo di inclinazione dell'asse  $z$  locale rispetto alla terna origine e  $h$  la quota. Si ottiene:

$$\begin{cases} \text{somma} = -\psi - \text{delta} - \text{epsilon} \\ \theta_{s1} = \arcsin\left(\frac{h-h_1-\text{ysup} \sin(\text{somma})-r_2 \sin(\text{somma}+\text{delta})}{l_{s1}}\right) \\ \theta_{s2} = \text{somma} - \theta_{s1} \end{cases}$$

Imponendo  $\psi = \frac{\pi}{2}$ , cioè che la shell stia verticale, si ottiene il grafico parametrico della posizione  $x$  del centro della sfera in funzione della quota  $h$  mostrato in figura 13. Le due soluzioni in figura sono dovute al fatto che  $\cos(\arcsin x) = \pm\sqrt{1-x^2}$ .

Di tutto il moto descritto in figura, soltanto la parte con  $x > 0$  è possibile nel robot reale, se fosse  $x < 0$  avremmo compenetrazione fra le shell. I due punti in cui si ha  $x = 0$  corrispondono alle due condizioni di chiusura della shell superiore, una a gomito alto e l'altra a gomito basso. Corrispondono ai valori  $h = 1.748$  per la soluzione a gomito basso, e  $h = 78.21$  per quella a gomito alto.

Nell'implementazione del robot entrambe sono importanti: la prima infatti viene utilizzata per chiudere il robot a sfera, la seconda per tenere le shell chiuse e lontane dalle gambe inferiori durante la camminata.

A differenza della parte inferiore, in cui è stata implementata la CLIK, nella parte superiore si è deciso di utilizzare proprio la soluzione in forma chiusa qui descritta. In questo modo possiamo muovere tutte le gambe superiori utilizzando due soli comandi, alleggerendo il sistema e sfruttando meno funzioni possibile.

## 2.5 Jacobiano

### 2.5.1 Jacobiani analitici

Per implementare la CLIK è necessario calcolare i Jacobiani analitici delle gambe inferiori. Avendo due possibili end effector (zampe e centro delle shell) si dovranno calcolare due Jacobiani distinti.

Innanzitutto si calcolano i vettori  $Q$  che rappresentano le coordinate generalizzate di posizione e orientazione dell'end effector:

$$Q(q) = \begin{bmatrix} x \\ y \\ z \\ \theta \\ \phi \\ \psi \end{bmatrix}$$

Dopodiché si calcolano i Jacobiani tramite la derivata dei vettori stessi <sup>1</sup>:

$$J_A(q) = \frac{\partial Q(q)}{\partial q}$$

Possiamo notare che il termine  $\psi$  è sempre nullo, sia nel vettore relativo all'end effector che in quello relativo al centro della sfera. È conveniente, perciò, eliminare quel termine, dato che non porta contributo al calcolo delle velocità ai giunti.

Allo stesso modo si possono eliminare i termini relativi agli angoli di inclinazione della gamba visto che non ne conosciamo il valore a priori.

Sappiamo inoltre che durante la camminata avremo  $\theta_1 = 0$  per migliorare il movimento del robot, si può quindi eliminare la prima colonna.

Si ottiene così un Jacobiano di ordine ridotto  $3 \times 3$  che verrà utilizzato nel calcolo della CLIK quando il robot cammina. Una matrice quadrata, che quindi non necessita di una pseudoinversa, permette sia di ridurre i tempi di calcolo (visto che la clik dovrà avvenire 6 volte, una per ogni gamba) sia di evitare che l'algoritmo converga ad un equilibrio che non è quello desiderato dall'utente.

Lo stesso procedimento si applica al Jacobiano relativo al centro della shell, soltanto che stavolta si hanno 4 parametri liberi:  $x$ ,  $y$ ,  $\theta$  e  $\phi$ .

---

<sup>1</sup>Per la visualizzazione delle matrici complete si rimanda al notebook *Jacobian.nb*

### 2.5.2 Jacobiano geometrico

Il calcolo del Jacobiano geometrico, sebbene non sia necessario per la CLIK, risulta utile per calcolare in modo più accurato la coppia ai giunti necessaria per muovere il robot. Si ha infatti:

$$\tau = J_P^T(q)w_P$$

dove  $w_P$  è *wrench* all'end effector, ossia il vettore di forze e coppie applicate alla fine della catena cinematica.

$$w_P = \begin{bmatrix} X \\ Y \\ Z \\ N \\ M \\ L \end{bmatrix}$$

È possibile calcolare il Jacobiano geometrico a partire da quello analitico molto semplicemente come:

$$J_P(q) = T(q)J_A(q)$$

dove

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\sin(\phi) & \sin(\theta) \cos(\phi) \\ 0 & 0 & 0 & 0 & \cos(\phi) & \sin(\theta) \sin(\phi) \\ 0 & 0 & 0 & 1 & 0 & \cos(\theta) \end{bmatrix}$$

Ponendoci nella condizione di riposo con solo 3 gambe a terra, cioè  $w_P = \begin{bmatrix} 0 \\ 0 \\ 12N \\ 0 \\ 0 \\ 0 \end{bmatrix}$  si ottiene

$$\tau = \begin{bmatrix} 0 \\ 0 \\ -6.075 \\ -1.194 \end{bmatrix} \text{ kg} \times \text{cm}$$

Il risultato è coerente con le ipotesi fatte a priori e garantisce la stabilità del robot sotto il proprio peso.

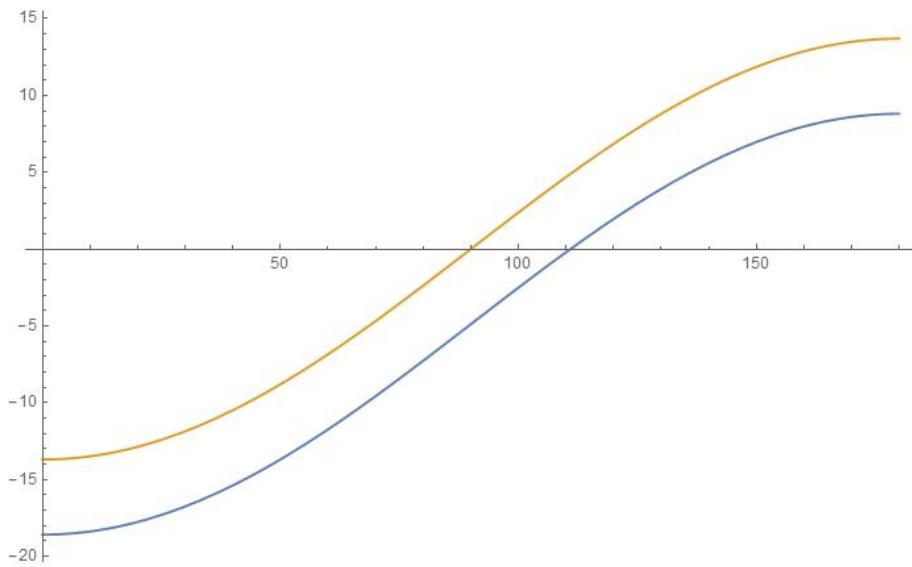


Figura 14:  $\tau_3$  (in blu) e  $\tau_4$  (in arancione) necessarie per tenere in piedi il robot in funzione dell'angolo  $\theta_4$

Possiamo rappresentare graficamente le coppie ai giunti 3 e 4 in funzione dell'angolo  $\theta_4$  quando  $\theta_1 = \theta_2 = \theta_3 = 0$ , cioè nella condizione più sfavorevole per il robot.

Possiamo notare che sotto ai  $50^\circ$  circa la coppia necessaria per tenersi in piedi è superiore alla coppia massima del motore, perciò il robot oltre quel valore non sarà più in grado di stare in piedi con solo 3 zampe appoggiate. Bisogna comunque notare che questa condizione è molto lontana dalla regione di funzionamento normale del robot ed è improbabile che si raggiunga senza l'intervento di agenti esterni.

## 2.6 Traiettorie

Per calcolare le traiettorie da dare agli end effector si è fatto affidamento a Mathematica, calcolando offline le posizioni dei giunti e poi implementandole in simulazione. Si è preferito questo approccio ad uno più generale, in cui le posizioni dipendono dai parametri del robot stesso, per rendere il codice più facilmente leggibile e interpretabile.

La posizione di riposo è stata data cercando di minimizzare la coppia al giunto 4 senza però ottenere una posizione "scomoda" per la camminata.

Poiché le gambe a terra formano una catena cinematica chiusa con il pavimento, si è deciso di mettere un sistema di riferimento solidale con il pavimento posto esattamente al di sotto del centro del robot. In condizioni normali i due sistemi di riferimento sono allineati, tuttavia quando il robot si inclina si avrà una rotazione attorno agli assi  $x$  o  $y$  (o entrambi) del sistema locale. La quota a cui dovranno stare gli end effector non sarà più fissa ma dipenderà dalle posizioni  $x$  e  $y$  degli stessi e dagli angoli di inclinazione  $\phi$  e  $\psi$ .

$$z = z_0 + x \tan(-\phi) + y \tan(-\psi)$$

I segni negativi sono dovuti al fatto che il robot non penserà di inclinarsi lui stesso, ma crederà di inclinare il pavimento di un angolo uguale in direzione opposta.

La stessa procedura viene utilizzata per la camminata, in cui le 3 zampe che restano a terra "spingono" il pavimento in direzione opposta a quella del moto.

Per la camminata in direzione  $x, y$  si utilizza un'interpolazione lineare fra la posizione attuale dei giunti e la posizione target, in modo che il robot si muova in direzione del target con una certa velocità gestita dall'utente.

La posizione target è uguale e opposta per le gambe a terra e quelle sollevate, in questo modo ogni passo sarà uguale al doppio del valore impostato dal target.

Per sollevare le gambe si utilizza come quota la funzione

$$z = z_0 + \sin(\pi\tau)$$

dove  $\tau$  è il tempo normalizzato per il passo.

Aumentando e diminuendo  $\tau$  si modifica la velocità di ogni passo, mentre modificando la distanza target si cambia la velocità di avanzamento del robot ad ogni passo. In via progettuale abbiamo deciso di mantenere  $\tau_{max}$  pari ad 1 s mentre gli altri parametri saranno gli input che l'utente dà per far camminare il robot.

È possibile far muovere il robot in una direzione qualsiasi semplicemente combinando la percentuale di avanzamento lungo  $x$  e  $y$ . Ciò non darà problemi alle gambe a terra

in quanto tutte riceveranno le stesse percentuali, portando il pavimento a muoversi di moto traslatorio puro.

Per quanto riguarda la rotazione su se stesso si è deciso di far muovere le gambe lungo una circonferenza passante per la posizione di riposo. In questo modo il sistema ruota attorno ad un asse verticale passante per il centro del robot e il sistema di riferimento del pavimento ruota attorno al proprio asse  $z$ .

Combinando moto rotatorio e moto traslatorio si ottiene comunque un modo rotatorio nel piano. Tuttavia il centro di istantanea rotazione non si trova allineato con il centro del robot ma spostato di una quantità proporzionale alla velocità di traslazione.

La garanzia che tutte e 3 le gambe a terra stiano ruotando allo stesso centro ce lo dà il fatto che sia la velocità angolare che quella lineare sono le stesse per tutte le gambe (perché definite dall'utente).

C'è comunque da considerare che non sono da escludere moti in direzioni non consentite. Infatti la CLIK contiene il termine  $ke$  che permette al manipolatore di recuperare eventuali errori dovuti a disturbi esterni. Questo contributo non è eliminabile proprio perché non è possibile garantire l'assenza di disturbi.

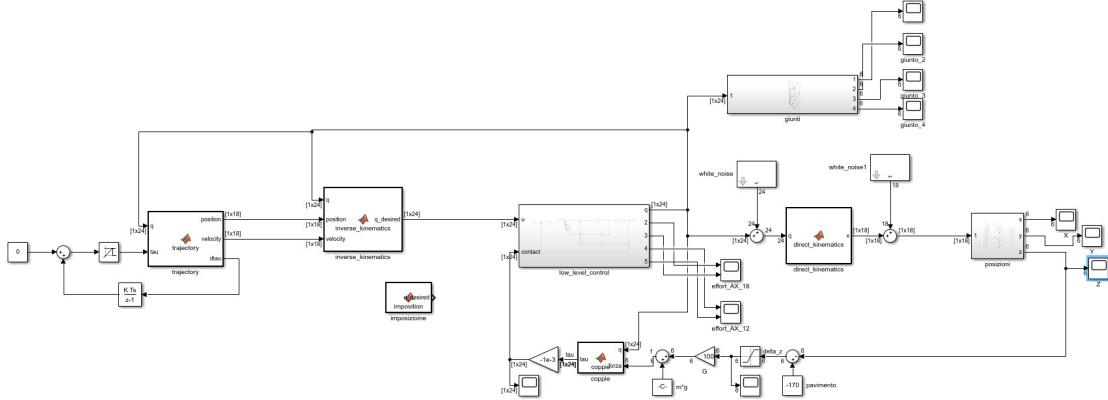


Figura 15: Schema Simulink del sistema robot + ambiente Gazebo

## 2.7 Taratura controllori

Per la taratura delle costanti di guadagno sia dei PID simulati che della CLIK si è utilizzato *Matlab Simulink* perché permette di fare molte simulazioni in poco tempo e controllare i risultati in tempo reale.

In figura 15 è mostrato lo schema Simulink utilizzato per tarare i controllori. Dato che l’ambiente di Gazebo simula con difficoltà i contatti fra il robot e il terreno (tanto che a robot fermo si può vedere un movimento simile ad uno scivolamento) si è cercato di riprodurre lo stesso comportamento su Simulink per andare poi a tarare i controllori.

Si è impostato un sistema massa-molla (non smorzato) asimmetrico, cioè la forza è unidirezionale. Quando l’end effector scende al di sotto del pavimento riceverà una forza verso l’alto proporzionale alla propria posizione, mentre quando si trova al di sopra sarà soggetto soltanto alla gravità.

Si aggiungono poi due rumori di processo (ai giunti e all’end effector) in modo da creare instabilità nel sistema.

La simulazione è stata effettuata in *tempo discreto* in quanto anche il robot ha un controllo TD, dove la CLIK viene discretizzata con il metodo di *Eulero in Avanti*:

$$q(k+1) = q(k) + \dot{q}(k)\Delta t$$

Facendo più simulazioni si ottiene che il sistema è stabile quando le costanti dei PID sono molto alte, mentre quella della CLIK è minore di 1. Si decide di scegliere  $k_{CLIK} = 0.1$  in modo da garantire stabilità per il sistema.

## 3 Controllo degli attuatori

### 3.1 Panoramica di comunicazione

Il microcontrollore comunica con le unità Dynamixel mandando e ricevendo pacchetti di dati. Ci sono due tipi di pacchetti: il “pacchetto di istruzioni” (instruction packet), mandato dal microcontrollore agli attuatori Dynamixel, e il “pacchetto di status” o “pacchetto di ritorno” (status/return packet), mandato dagli attuatori al microcontrollore.

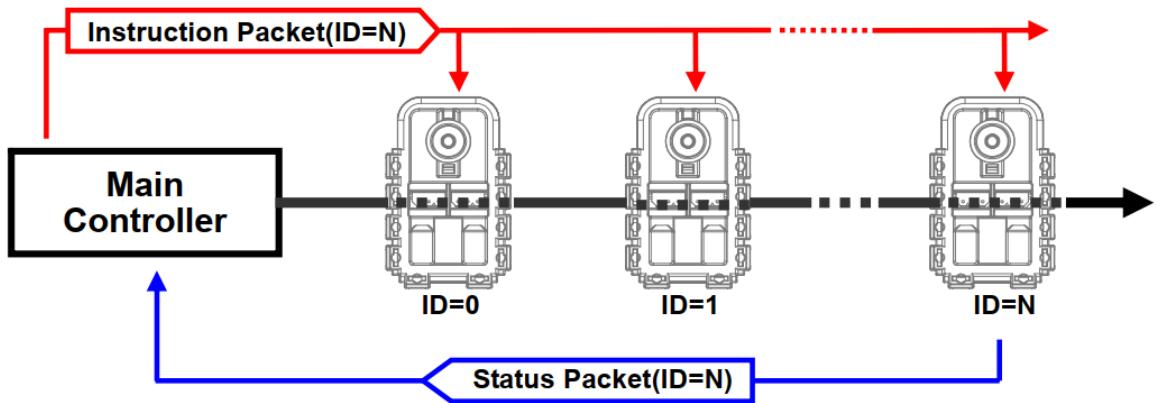
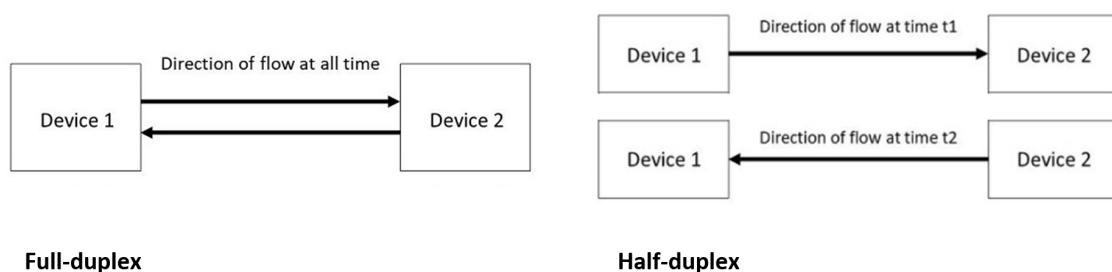


Figura 16: Bus dati

Il bus dati collega tutti i servo in serie, e il microcontrollore manderà instruction packet in cui è specificato un codice identificativo (ID), e solo il servo con l'ID richiesto risponderà con il corrispettivo return packet ed effettuerà l'istruzione richiesta. Questo sistema ovviamente richiede l'univocità di ID tra i servo nella stessa rete, perché se due o più servo presentassero lo stesso ID i pacchetti di ritorno colliderebbero risultando in problemi di comunicazione.

Gli attuatori Dynamixel comunicano tramite comunicazione seriale asincrona con 8 bit, 1 bit di stop e nessun bit di parità (sostituito dal byte checksum nello status packet), e richiedono quindi l'uso di una porta UART (Universal Asynchronous Receiver-Transmitter), e che la trasmissione dei dati sia in modalità half-duplex, per consentire l'invio di instruction packet e return packet su una singola linea dati.

La modalità half-duplex consiste nell'utilizzo di un singolo bus per la comunicazione, su cui entrambe le parti collegate possono sia ricevere che trasmettere, ma non allo stesso tempo, e si devono quindi alternare correttamente per evitare congestione di rete. Differisce quindi dalla comunicazione full-duplex in cui le linee dati sono distinte e il flusso di comunicazione su entrambe è univoco, cosa che permette la comunicazione bidirezionale ed evita che una parte debba aspettare di ricevere un messaggio per poter mandare il proprio (utilizzata ad esempio per le reti telefoniche); un vantaggio della comunicazione half-duplex tuttavia è che durante la trasmissione dei dati il mittente ha a sua disposizione l'intera larghezza di banda, che viene invece divisa tra mittente e ricevitore in full-duplex.



### 3.2 Dimensionamento batteria

Il robot è stato pensato come un ente a sé stante, non vincolato da alimentatori o cavi: una scelta importante è quindi stata quella di una batteria adatta. La tensione della batteria è stata decisa in modo che fosse pari alla tensione nominale dei servo, per potersi collegare direttamente ad essi.

Sono state inoltre fatte delle considerazioni sulla corrente erogata, considerando che il datasheet dei servo indica 1.5A in coppia di stallo, ma siccome non è ragionevole pensare che tutti i motori si trovino in coppia di stallo contemporaneamente, si è tenuto conto di questo caso solo come *worst-case scenario*. Si è quindi pensato di puntare più sulle performance che sul run time, ovvero privilegiare una batteria che possa garantire una scarica in continua piuttosto elevata anche se a discapito del tempo totale di utilizzo.

Alla fine è stata scelta una batteria LiPo Tattu da 11.1V a tre celle (3S), capacità di 2300 mAh e C-rating di 45C in scarica continua. In particolare il C-rating determina a quale corrente viene caricata o scaricata la batteria, secondo il calcolo [mAh \* C / 1000]: nel nostro caso si ha quindi  $2300\text{mAh} * 45\text{C} / 1000 = 103.5\text{A}$ , e considerando anche uno scarto dato dalla tendenza dei produttori a “gonfiare” il C-rating delle batterie che vendono, si può comunque considerare adeguato. Per il run time effettivo, si capisce che già a 20A una batteria da 2300mAh verrebbe scaricata in circa 6 minuti: tuttavia sperimentalmente è stato visto (tramite un alimentatore usato in fase di testing) che la corrente assorbita dai servo durante il movimento va da 2A fino a un massimo di 4A, permettendo tempi di utilizzo maggiori; è comunque da precisare che una batteria LiPo non dovrebbe mai essere scaricata sotto un certo limite, in questo caso fino a circa 8 o 9V.

La LiPo è dotata di un tipico connettore XT60, con cui può essere ricaricata, mediante un LiPro Balance Charger, e con cui viene collegata alla scheda di controllo.



Figura 17: Batteria LiPo Tattu by Gens ACE

### 3.3 La scheda di controllo

La scheda di controllo comprende due integrati principali, il modulatore di tensione step-down DC-DC LM2596S e il buffer o line driver SN74LS241. In più, dallo schema si può notare la resistenza di pull-up tra l'alimentazione del buffer e la linea dati, la capacità di protezione che funge da filtro sulla tensione per proteggere la rete da eventuali cali di tensione, e un safe switch.

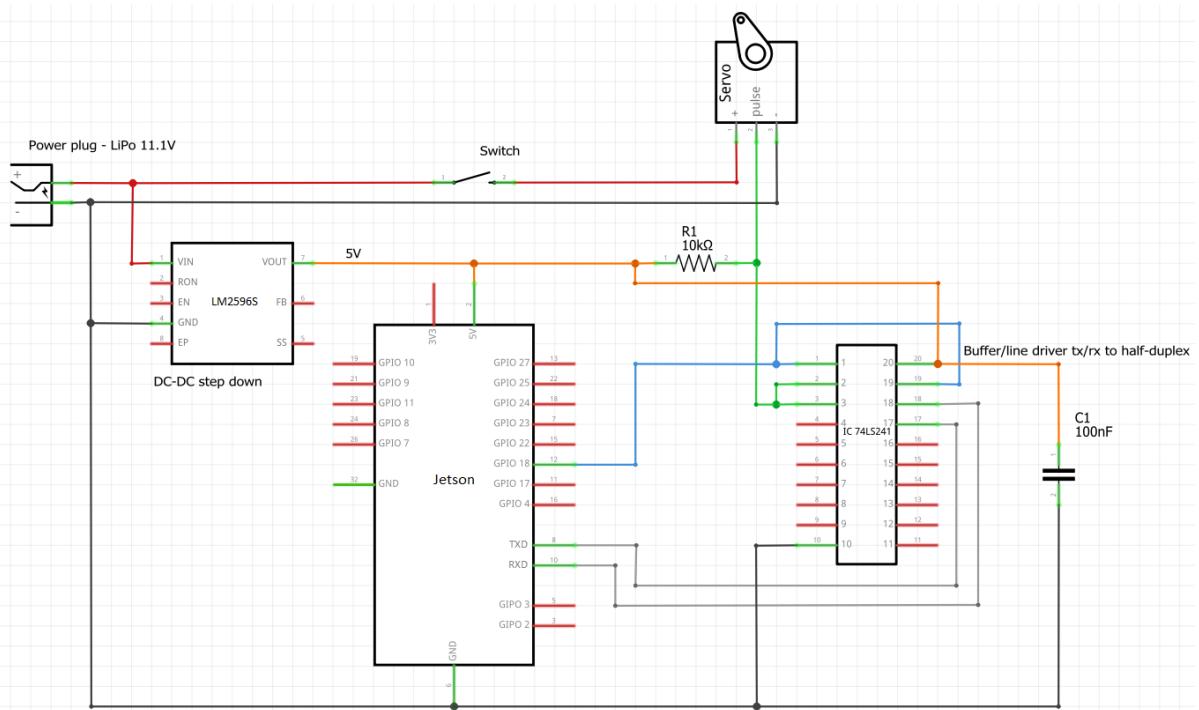


Figura 18: Schematico

Lo step-down è un componente che da una singola fonte di alimentazione (nel nostro caso una batteria LiPo a 11.1V che fornisce direttamente la tensione nominale consigliata per i servo Dynamixel) permette di ricavare un valore di tensione inferiore in uscita, nel nostro caso 5V per la Jetson. La Jetson necessita di valori di alimentazione piuttosto precisi, andando in brown-out se la tensione scende sotto 4.75V: lo step-down switching garantisce un valore di tensione molto stabile, ulteriormente garantito dalla capacità di protezione.

Utilizzando la funzione da terminale “screen” e cortocircuitando TX e RX della porta UART è possibile verificare il corretto funzionamento della porta usando uno

schermo virtuale come eco a comandi da tastiera che vengono inviati e ricevuti nella stessa porta: in questa configurazione si può facilmente controllare il valore di tensione della porta stessa usando un multimetro; in questo modo abbiamo potuto controllare che la porta UART del modulo USB-to-UART utilizzava funzioni con logica a 5V (impostabile tramite un jumper posto sul modulo stesso). Per tale motivo la parte del circuito relativa al buffer e alla linea dati dei servo viene alimentata dal pin 5V presente sulla Jetson stessa; questo perché in versioni precedenti si è utilizzato porte UART con logica a 3.3V, quindi era utile poter alimentare il buffer tramite il pin 3V3 della Jetson per avere due livelli di tensione diversi. Inoltre, i servo internamente hanno un pull-up a 5V per la logica di risposta, quindi è sembrato consone uniformare le tensioni con un modulo USB-to-UART che può essere settato a 5V.

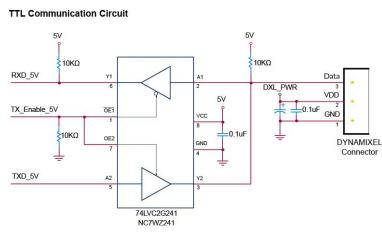


Figura 19: Circuito TTL e schema Servo AX



Figura 20: Modulo USB-to-UART

Il buffer viene invece utilizzato per combinare i pin TX e RX della porta UART su microcontrollore in un'unica linea dati utilizzata dai servo in half-duplex. Esso funziona attraverso un pin digitale che da Jetson comanda l'accensione dei transistor che regolano l'apertura o la chiusura delle linee collegate ai canali TX e RX dalla UART, in modo che solo una delle due alla volta sia collegata con la linea dati dei servo.

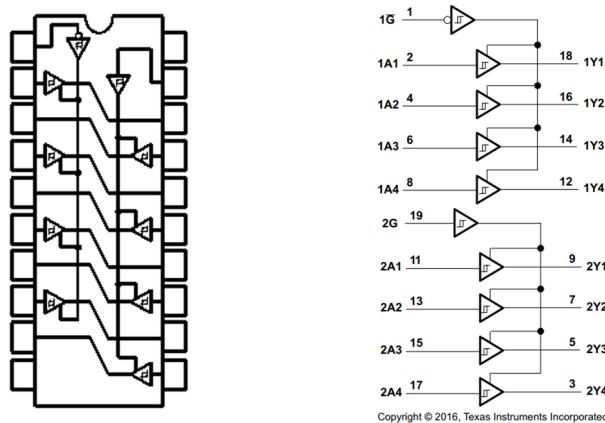


Figura 21: Buffer line driver

Infine è stato inserito uno switch tra la batteria e il collegamento ai servo, in modo da poter disconnettere rapidamente i motori in caso di overload / overheating (o anche solo per risparmiare carica della batteria), ma mantenere l'alimentazione sulla Jetson.

In fase di testing si è anche provato ad aggiungere un nuovo componente, un logic level converter, per cercare di migliorare il segnale del pin digitale che manda lo switch tra lettura e scrittura dei dati: questo perché tale segnale lavora a logica 3.3V invece che 5V come il resto del circuito, e si è pensato che rigenerandolo con un level converter a 5V si sarebbe potuto migliorare i segnali.

Purtroppo si è subito notato che il nuovo componente portava sì il valore alto del pin digitale in uscita a 5V, ma alzava anche il minimo a 1V invece che tenerlo a 0V, rendendo quindi il segnale più deteriorato; questo si è pensato dipendesse dalla resistenza di pull-down interna del pin della Jetson, che rimanendo collegata insieme ai due pull-up (a 5V e a 3.3V) del level converter ha come risultato questo comportamento indesiderato in logica 0.

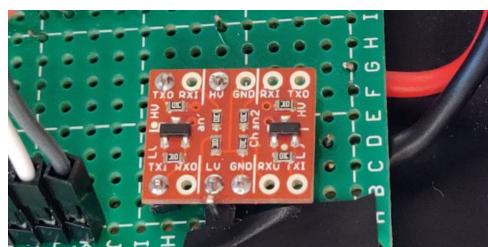


Figura 22: Logic Level Converter (non connesso nella versione finale)

### 3.3.1 Hardware

In fase preliminare la scheda di controllo aveva come base una semplice breadboard, per poi essere sostituita in seguito da una versione più stabile montata e saldata su millefori.

Infatti, pur essendo più facilmente modificabile e adattabile e quindi ideale durante la progettazione, la versione su breadboard utilizza semplici cavetti e header inseriti a pressione, e per questo presenta un elevato rischio che venga perso il collegamento tra i vari componenti mentre il robot si muove.

La versione su millefori invece presenta componenti saldati su di essa, e tutti i collegamenti sono stati fatti, come è convenzione, sul retro, in modo che le varie parti siano ben visibili e che i cavi abbiano un'area a loro riservata e che non siano di intralcio. Quest'ultimi inoltre sono stati scelti considerando anche il colore dell'isolante che li riveste, in modo da avere un *color code* che renda immediatamente riconoscibili le parti del circuito collegate insieme e le loro funzioni (visibile in Figura 25):

- Blu - Collegamento tra alimentazione esterna a 11.1V, l'ingresso dello step-down e lo switch a monte dei servo;
- Rosso - Alimentazione a 11.1V a valle dello switch, viene mandata ai tre header che rigenerano la tensione sulla daisy chain ai servo (spiegazione in [3.3.2]);
- Arancione - Alimentazione a 5V in uscita dallo step-down e al buffer;
- Bianco - Neutro, collega alimentazione esterna, step-down, buffer, servo e Jetson;
- Verde - Collegamenti di dati tra buffer e Jetson (TX, RX e il pin digitale);
- Giallo - Dati inviati in daisy chain dal buffer a tutti i servo.

Progetto *esapode sferico*

---

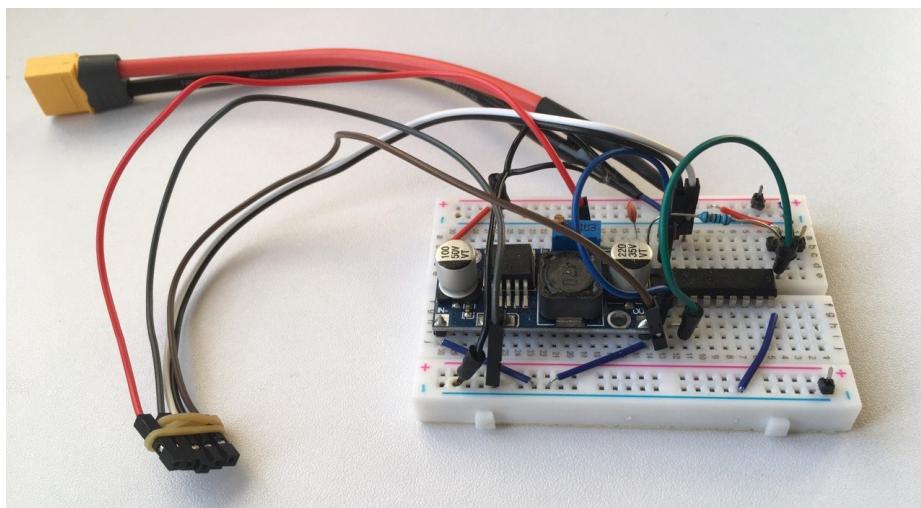


Figura 23: Scheda di controllo, versione breadboard (prototipo)

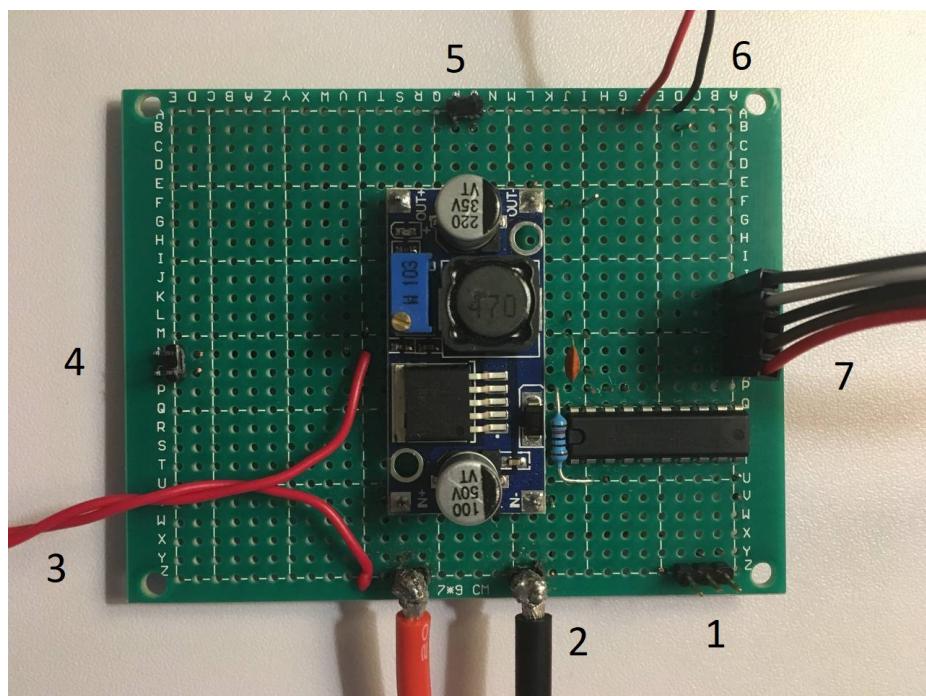


Figura 24: Scheda di controllo, versione millefori (definitiva): [1] header inizio data line + alimentazione gambe 1-2; [2] connettore XT60 per alimentazione da LiPo; [3] safe switch; [4] header alimentazione gambe 3-4; [5] header alimentazione gambe 5-6; [6] connettore barrel jack per alimentazione Jetson; [7] ai PIN della Jetson (in rosso alimentazione UART 5V)

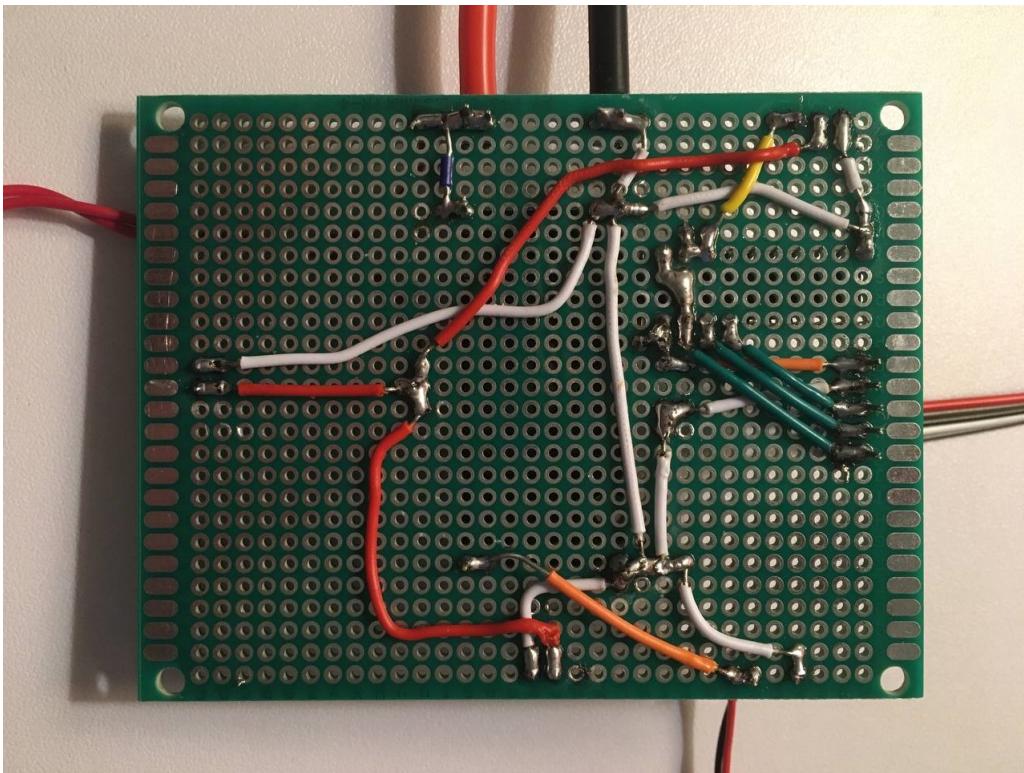


Figura 25: Scheda di controllo, versione millefori - retro

### 3.3.2 Rigenerazione della tensione ai servo

Durante la progettazione si è dovuto affrontare un problema riguardante il collegamento di tutti i 19 motori in daisy chain: poiché la linea di fase è a 11.1V, la caduta di tensione data dalle resistenze parassite sui cavi diventava tale da impedire il corretto funzionamento di gran parte dei servo in fondo alla serie. Si è pensato quindi di risolvere tale problema costruendo la millefori in modo che la tensione diretta dalla batteria non arrivasse solo al primo attuatore della serie, ma al primo servo di una gamba ogni due, ovvero tre serie di sei motori ciascuna (più un motore nel folding fan), creando quindi tre ramificazioni diverse a innestarsi nella rete di cavi 3-pin; il bus dati è invece stato lasciato in serie continua condivisa da tutti i servo, poiché la tensione su di esso è di 5V, ed essendo piuttosto inferiore rispetto all'alimentazione a 11.1V, le resistenze parassite hanno un effetto molto più blando che non inficia sulla buona trasmissione dei dati; questo è anche più vantaggioso se si pensa che avere diverse linee dati comporterebbe l'aggiunta al circuito di un altro buffer e l'uso di una diversa porta UART, cosa che aumenterebbe la complessità del controllo e potrebbe portare a problemi di sincronizzazione.

### 3.4 Connessione con la Jetson

Per semplificare le interazioni con la Jetson, si è pensato di dotarla di un adattatore USB wireless per consentirgli di connettersi alla rete wi-fi e da lì essere riconosciuta da pc collegati alla stessa rete; in particolare l'adattatore USB scelto è un TP-Link Archer T2U Nano. Si è quindi utilizzata un'applicazione apposita per visualizzare lo schermo della Jetson da remoto, prima con VNC Viewer e poi con NoMachine (alternativa più performante che risente molto meno del lag).

Installando Visual Studio Code su Jetson e l'estensione “Remote SSH” da pc, è possibile inoltre usare VScode da remoto aprendo, modificando ed eseguendo file (quest'ultima solo nel caso in cui non si debba aprire un'interfaccia grafica) direttamente da pc.

Questo permette di bypassare il collegamento con microUSB e PuTTY, che tra l'altro consente solo l'accesso al terminale e non all'intera interfaccia del sistema operativo su Jetson, e rende più comodo lavorare sul robot senza dover avere sempre un cavo che colleghi la scheda al pc.

Una cosa da tenere in considerazione per la connessione wireless è che la Jetson ha un IP dinamico, quindi è necessario controllare ad ogni avvio quale sia quello corrente e che non sia cambiato dall'ultima accensione: sono stati utilizzati quindi programmi come Advanced IP Scanner oppure si è usato l'admin login del router utilizzato (192.168.1.1) per identificare tutti gli apparecchi collegati; passando a NoMachine si è inoltre notato che essa riesca a trovare automaticamente l'IP nella maggior parte dei casi; un'ultima opzione consiste nel collegare la Jetson a microUSB, aprire il terminale con PuTTY, e usare il comando `ip a`: l'IP viene riportato sotto wlan0.

### 3.5 Software per il controllo dei servo

- motor\_control\_ros.py
- pybullet\_sim.py
- check\_servo\_utilities.py
  - ➔ servo.py
  - ➔ ax12.py
- + dynamixelUART.ino

#### 3.5.1 ax12.py

Il file `ax12.py` include la classe `Ax12`, che gestisce la comunicazione con i servo tramite protocollo Dynamixel 1.0 compatibile con i servo Dynamixel AX-12A e AX-18A.

All'interno della classe la porta UART `ttyUSB0` viene inizializzata con baud rate pari a 1000000 (1M) Bd. Sono stati testati vari baud rate differenti, ma la velocità richiesta per completare almeno la lettura di posizione corrente e la richiesta di raggiungimento di una nuova posizione deve essere tale da andare di pari passo con la frequenza della CLIK; per questo motivo è stato quindi scelto il massimo baud rate consentito dai servo.

Ogni operazione è gestita da una funzione interna alla classe che manda il pacchetto di istruzioni desiderato e tutte le funzioni terminano con una `readData` che riceve il pacchetto status dai motori.

La sottofunzione `readData`, poiché utilizzata sempre a prescindere dall'istruzione richiesta, è stata elaborata per gestire efficacemente eventuali errori di trasmissione, identificandone la gravità e agendo di conseguenza. In particolare è stata ridimensionata l'influenza dei timeout error che avrebbero altrimenti bloccato l'esecuzione del programma, e si è preferito optare per un sistema di richiesta ripetuta, per cui se la ricezione non va a buon fine il programma manda un messaggio di errore che consente di provare nuovamente un certo numero di volte (determinato dalla variabile `MAX_LOOP` in `servo.py`, dove le funzioni vengono chiamate) prima di considerare il dato perso.

Inoltre è stato implementato un controllo ulteriore tramite checksum, l'ultimo byte dello status packet, che funge da versione avanzata del bit di parità e permette di identificare con precisione gli errori di invio confrontando il checksum ricevuto con il checksum calcolato dai parametri del pacchetto e verificandone la corrispondenza.

$$\text{Check Sum} = \sim (\text{ID} + \text{Length} + \text{Instruction} + \text{Parameter1} + \dots \text{Parameter N})$$

Figura 26: Calcolo checksum

### 3.5.2 servo.py

Il file `servo.py` include la classe Servo, che comprende la classe Ax12 e permette di mascherare le funzioni della classe Ax12 con operazioni di livello più alto.

All'interno della classe vengono specificati i limiti di rotazione e i valori di offset per ogni motore, che grazie alla simmetria del sistema sono condivisi tra giunti dello stesso livello di gambe diverse.

La prima operazione effettuata è `checkServo`, che manda l'istruzione ping a una serie di ID e raccoglie tutti quelli che hanno risposto in una lista: se tale lista non è pari al numero di servo che ci si aspettava, il programma si interrompe con una chiamata a `servoError`, una sottoclass appositamente creata per gestire tale errore, e vengono stampati a schermo alcuni parametri rilevanti al debug, ovvero il numero di servo richiesti, il numero di servo trovati, e la lista di ID dei servo che hanno correttamente risposto alla funzione ping.

Se i servo sono stati trovati tutti correttamente, si procede con l'inizializzazione di alcuni parametri di interesse, tra cui la coppia massima e i valori limite di tensione minima e massima.

Le funzioni principali sono `updateAngles` e `setAngleAll`: la prima legge i registri in cui viene allocata la posizione corrente del servo (in un range da 0 a 300 gradi) e li salva all'interno della classe Servo per poterli utilizzare rapidamente in seguito; la seconda è una funzione di movimento e comanda i servo in modo che si spostino a un angolo prestabilito a velocità prestabilita, prendendo come argomento una lista in cui gli angoli desiderati sono ordinati per ID crescente. Queste due funzioni sono quelle che a programma avviato vengono usate a ogni iterazione, e consentono la ricezione di dati e la comunicazione di comandi tra il programma che genera le

traiettorie e il programma di controllo motori.

Durante l'esecuzione vengono inoltre stampati a schermo dei valori di interesse (ogni unità di tempo prestabilita, per evitare il cluttering dello schermo), tra cui i valori correnti di posizione, utile per assicurarsi che l'esecuzione avvenga correttamente e per il debugging, e la tensione interna dei servo, per monitorare in tempo reale il livello di carica della batteria.

Altre funzioni utilizzate in fase di progettazione sono `changeID` e `changeBaudrate`, per assegnare l'ID e il baud rate ai motori, nel programma di supporto `check_servo_utilities.py`.

### 3.5.3 `pybullet_sim.py`

Per facilitare i test è stato creato un codice PyBullet (che si appoggia sull'ambiente OpenGL) per la simulazione rapida in visualizzazione 3D dei giunti di un singolo arto meccanico, chiamato `pybullet_sim.py`.

Tale programma carica un file URDF, scritto in `urdf.xacro` e poi convertito, che replica i 4 link e 3 giunti di una gamba dell'esapode, imposta i limiti di movimento come rotazioni minime e massime e tiene conto degli offset in modo che la configurazione degli angoli (0,0,0) sia consistente con una gamba i cui link sono perfettamente allineati in linea retta.

Una volta avviato, i motori possono essere controllati semplicemente tramite gli slider presenti nella scheda Params. Questo metodo è stato usato in fase di progettazione per trovare gli stessi valori empirici di displacement e limiti, e in seguito per assicurarsi, in qualsiasi caso si rendesse necessario, il corretto funzionamento delle gambe.

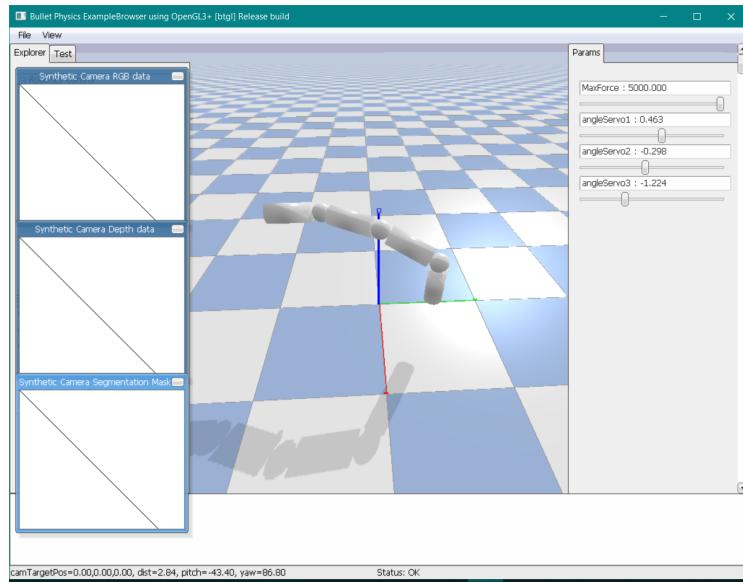


Figura 27: Simulazione PyBullet

### 3.5.4 dynamixelUART.ino

Inoltre sempre in fase di testing si è utilizzato anche un codice scritto in C per Arduino, chiamato `dynamixelUART.ino`, per poter provare il collegamento dei motori anche su una scheda differente, nel nostro caso un Arduino Uno.

Il codice sfrutta la funzione `SoftwareSerial` per usare due pin digitali della scheda come porta UART full-duplex; in tal modo la scheda Arduino di prova può essere collegata direttamente al resto del circuito, e lascia libera la porta seriale per il collegamento con il PC, in modo da dividere la parte che carica il programma sulla scheda dalla parte che comunica con i servo.

Le funzioni usate sono unicamente per effettuare il ping e settare il baudrate, tradotte da Python a C.

### 3.5.5 motor\_control\_ros.py

Il file che infine permette di controllare i servo e ricevere gli angoli voluti dal programma che genera le traiettorie è `motor_control_ros.py`, che viene lanciato tramite il comando `rosrun shellbot_pkg motor_control_ros.py`, e avvia l'interconnessione in ROS dei dati presi in input dal `teleop_key` (una speciale interfaccia per il movimento del robot, vedi Paragrafo 4.2.1).

Per una descrizione più specifica, si rimanda al Paragrafo 4.4.1.

## 4 Software

Il software utilizzato nel corso di questo progetto è stato sviluppato in due versioni: inizialmente è stata sviluppata una versione simulata, e successivamente questa è stata modificata per produrne una utilizzabile a bordo del robot. In entrambe le versioni del programma lo scopo è quello di permettere all'utente di interagire col robot tramite input da tastiera, per comandare i suoi movimenti e ricevere da esso le informazioni più rilevanti.

Il programma di simulazione e la sua versione on-board sono stati sviluppati quasi interamente in linguaggio Python, utilizzando la libreria *rospy* disponibile nel noto framework per sviluppo di applicazioni software in ambito robotico *ROS (Robot Operating System)*. Questo framework consente l'esecuzione parallela di diversi programmi e la loro comunicazione, allo scopo di permettere a ciascuno di essi di occuparsi di uno specifico task e rendere il codice modulare e riutilizzabile nelle sue singole parti. Grazie a queste caratteristiche è stato possibile passare dalla simulazione al programma on-board in modo semplice, sostituendo l'ambiente di simulazione con un programma di interfacciamento fra scheda e motori e lasciando inalterato tutto il resto.

Il programma di simulazione è stato testato in *Gazebo*, l'ambiente simulativo più frequentemente utilizzato insieme al framework ROS. Il modello del robot inserito nell'ambiente di Gazebo è stato generato tramite lo standard per rappresentazione di robot *URDF*, all'interno del quale sono stati inseriti i pezzi generati in CAD.

Di seguito il robot in simulazione, sia nella forma di esapode che nella forma di sfera.

## Progetto *esapode sferico*

---

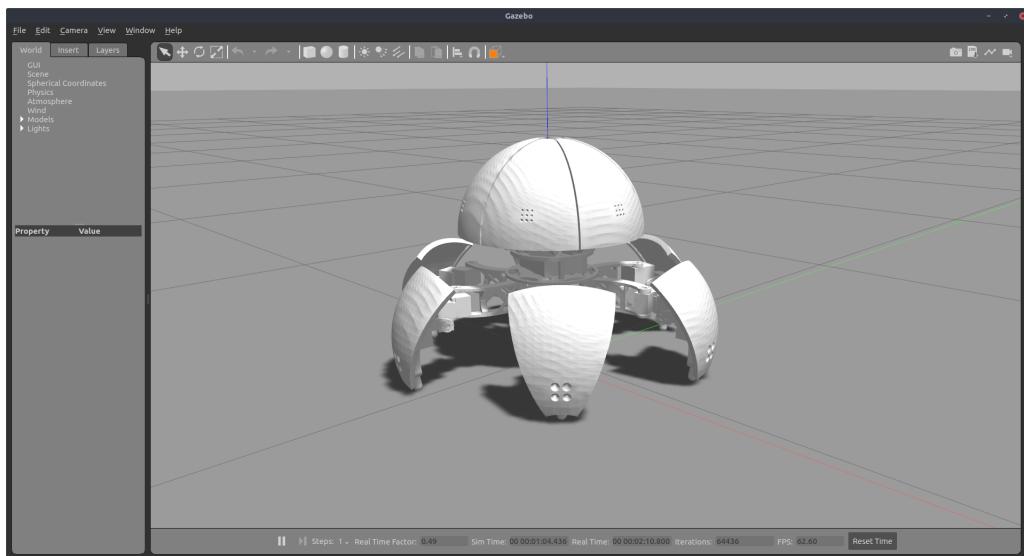


Figura 28: Robot nella sua forma di esapode

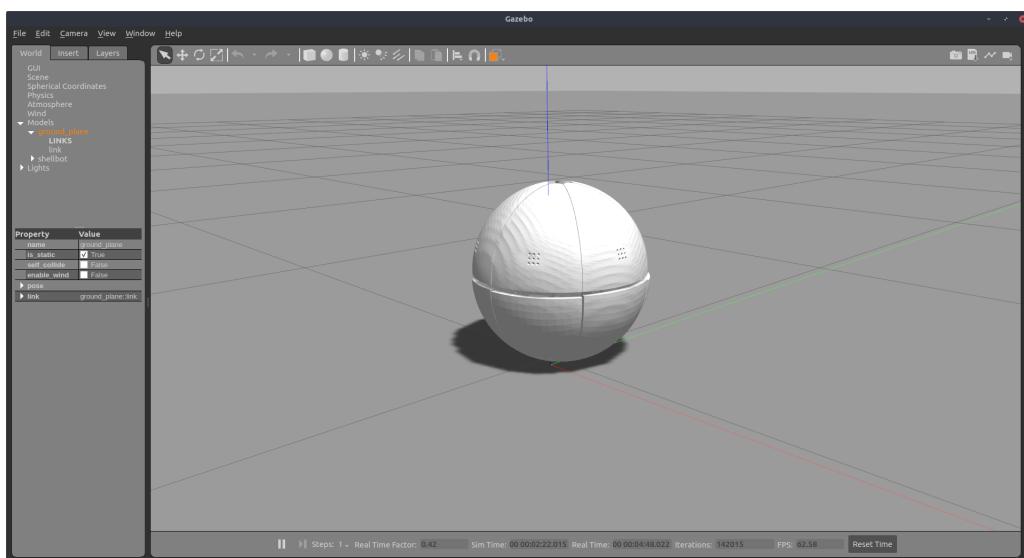


Figura 29: Robot nella sua forma di sfera

In questo capitolo vengono presentati nell'ordine una mappa dei file, il programma di simulazione, la rappresentazione del robot in simulazione e la versione del programma on-board, evidenziandone analogie e differenze con la versione simulativa.

## 4.1 Mappa dei file

Tutti i file sviluppati per la realizzazione di questo progetto sono contenuti in due *ROS workspace*, eccezione fatta per i programmi di test dei motori e per i file di disegno meccanico e modellazione del robot. Questi due workspace implementano la simulazione e il programma on-board.

### 4.1.1 Workspace del programma di simulazione

Un ROS workspace è un insieme di file con una struttura automaticamente generata al momento della sua prima compilazione. La cartella contenente il ROS workspace della simulazione è *shellbot\_2.2\_ws*, e per accedere ai file di interesse il percorso da seguire è: *shellbot\_2.2\_ws* → *src* → *shellbot\_pkg*.

In questo modo si accede all'unico ROS package presente nel workspace. Al suo interno sono contenuti diversi file e cartelle, descritti di seguito:

- **scripts**: cartella contenente i file Python relativi a nodi ROS e librerie utilizzate. Il suo contenuto è illustrato in dettaglio nel paragrafo successivo.
- **launch**: cartella contenente i file *.launch* per l'avvio della simulazione in ROS. Al suo interno sono contenuti i file:
  - **simulation.launch**: file di lancio del programma di simulazione.
  - **RT\_simulation.launch**: file di lancio del programma di simulazione con sezione di esecuzione in tempo reale.
  - **init\_gazebo.launch**: file di lancio del simulatore Gazebo, incluso nel file *simulation.launch*.
  - **teleop\_key.launch**: file di lancio della finestra di input da tastiera.
- **models**: cartella contenente i file URDF e XACRO per la generazione del modello del robot nel simulatore Gazebo. Al suo interno sono contenuti i file:
  - **shell\_robot.urdf.xacro**: file principale, il quale richiama tutte le altre macro per la composizione del robot.
  - **chassis.urdf.xacro**: file di descrizione del corpo centrale del robot.
  - **shell\_robot\_bottom\_leg.urdf.xacro**: file di descrizione di una singola gamba inferiore del robot.
  - **shell\_robot\_upper\_leg.urdf.xacro**: file di descrizione di una singola gamba superiore del robot.

- **variables.urdf.xacro**: file contenente i parametri fisici del robot, raccolti qui e richiamati nei singoli file.
- **meshes**: cartella contenente i file *.stl* relativi ai pezzi meccanici del robot, richiamati all’interno dell’URDF.
- **config**: cartella contenente i file di configurazione *.yaml*.
- **worlds**: cartella contenente il l’ambiente in cui viene fatto apparire il robot nel simulatore Gazebo.

## 4.2 Simulazione

Ogni script presente in un progetto ROS prende il nome di *”nodo”*, e si occupa di elaborare dati in ingresso e produrre un output disponibile al sistema. Gli output prodotti dagli script prendono il nome di *”messaggi”*, e sono accessibili a qualunque nodo ne abbia bisogno per lo svolgimento del proprio task.

ROS mette a disposizione uno strumento di visualizzazione schematica dei nodi e dei messaggi correntemente attivi all’interno di un programma in esecuzione, il quale prende il nome di *”rqt\_graph”*. Nella figura seguente viene illustrato l’*rqt\_graph* del programma di simulazione: gli elementi con contorno ellittico rappresentano i nodi, mentre quelli con contorno rettangolare rappresentano i messaggi.

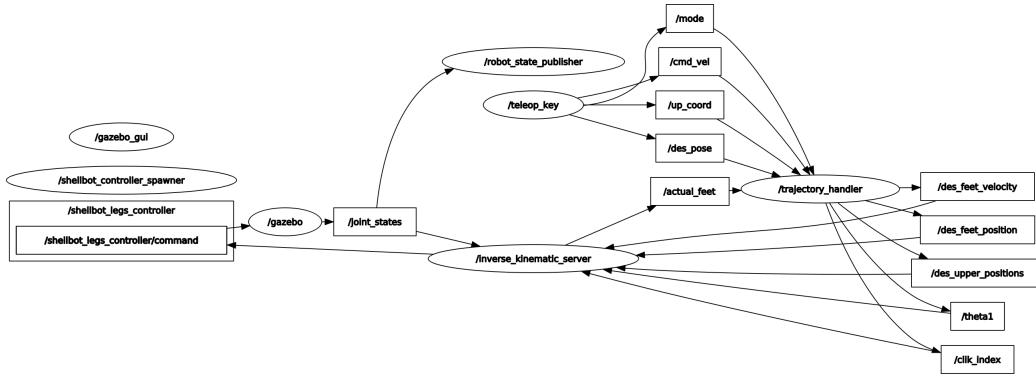


Figura 30: *rqt\_graph* della simulazione

I tre nodi principali del programma simulativo sono *teleop\_key*, *trajectory\_handler* e *inverse\_kinematic\_server*, e sono anche gli unici in comune fra programma simulativo e programma on-board. Gli altri tre presenti in figura riguardano l’esecuzione dell’ambiente di simulazione Gazebo, e non sono utili ai fini dell’utilizzo sul robot reale. Di seguito una descrizione di ciascuno dei tre sopra citati e della libreria *shellbot\_kinematic\_laws*, contenente i calcoli relativi alla cinematica del robot.

```

SHELLBOT GUI - HEXAPOD MODE

----- WALKING -----
7 8 9      rot_L  forw  rot_R
4 5 6    ->   left  stop  right
           2          back

----- POSE -----
q  w  e  ->  -yaw  +pitch  +yaw
a  s  d  ->  -roll  -pitch  +roll

0 --> trigger shell mode

HEXAPOD MODE VELOCITIES (%)
forward: 0.00
lateral: 0.00
rotation: 0.00

CTRL-C or ESC to quit gui

```

```

SHELLBOT GUI - SHELL MODE

----- MOVING -----
7 8 9      rot_L  forw  rot_R
4 5 6    ->   left  stop  right
           2          back

1 -> trigger hexapod mode

SHELL MODE VELOCITIES (%)
forward: none
lateral: none

CTRL-C or ESC to quit gui

```

Figura 32: GUI per la modalità shell

Figura 31: GUI per la modalità esapode

#### 4.2.1 teleop\_key

Il nodo `teleop_key` ha il compito di produrre le interfacce grafiche utili all’interazione tra utente e programma, di leggere gli input da tastiera e comunicarli al nodo `trajectory_handler`. Il robot ha due modalità di funzionamento, a cui corrispondono due differenti interfacce grafiche, una relativa alla modalità esapode (*hexapod mode*) e una relativa alla modalità sfera (*shell mode*). Di seguito le due rispettive interfacce grafiche:

Ognuna delle due interfacce contiene la mappatura tra tasti e azioni relative alla modalità corrente. I comandi di movimento vengono espressi come percentuale di una lunghezza di passo massima, da eseguire in un intervallo di tempo prefissato. Ogni percentuale relativa a un movimento viene visualizzata nell’interfaccia grafica.

Questo nodo produce i seguenti topic:

- **/cmd\_vel**: comando di velocità di movimento del robot, diviso in componente lineare e angolare. Ognuna delle due è a sua volta espressa lungo gli assi x, y e z, per un totale di sei componenti di movimento.
- **/des\_pose**: comando di posa del robot, diviso nelle tre componenti di rollio, beccheggio e imbardata.
- **/up\_coord**: comando di gestione della parte superiore del robot. È composto da due valori: *upper\_position* gestisce l’altezza desiderata della parte superiore, mentre *upper\_angle* gestisce l’angolo di apertura delle shell superiori.

- **/mode:** comando di modalità del robot, espresso come stringa. I due valori testuali sono *"hexapod"* e *"shell"*.

#### 4.2.2 trajectory\_handler

Il nodo trajectory\_handler si occupa di ascoltare i valori contenuti nei quattro topic prodotti dal nodo teleop\_key e di tradurli in delle traiettorie desiderate, le quali vengono successivamente comunicate al nodo inverse\_kinematic\_server. A seconda del movimento richiesto dal nodo teleop\_key le traiettorie possono essere riferite a diverse parti del robot.

Le traiettorie desiderate vengono comunicate al nodo inverse\_kinematic\_server tramite cinque differenti topic:

- **/des\_feet\_position:** vettore di 24 elementi contenente le coordinate desiderate delle gambe inferiori. La dimensione di 24 elementi è dovuta al fatto che, a seconda delle necessità, i movimenti desiderati possono essere riferiti sia agli end-effector delle gambe (le punte degli ultimi link), che ai centri delle sfere virtuali di cui ogni shell inferiore fa parte. Nel primo caso, per ognuna delle gambe occorrono tre coordinate per esprimere la posizione desiderata (posizione in coordinate cartesiane), mentre nel secondo caso ne occorrono quattro (coordinate cartesiane x e y, angoli di imbardata e beccheggio)
- **/des\_feet\_velocity:** comando di posa del robot, diviso nelle tre componenti di rollio, beccheggio e imbardata.
- **/des\_upper\_positions:** comando di gestione della parte superiore del robot. È composto da due valori: *upper\_position* gestisce l'altezza desiderata della parte superiore, mentre *upper\_angle* gestisce l'angolo di apertura delle shell superiori.
- **/theta1:** comando di angolo desiderato al giunto dell'ingranaggio centrale.
- **/mode:** comando di modalità del robot, espresso come stringa. I due valori testuali sono *"hexapod"* e *"shell"*.
- **/clik\_index:** valore numerico intero che va a selezionare la specifica clik da far eseguire al nodo inverse\_kinematic\_server.

A seconda dei messaggi inviati nel topic /mode, di cui questo nodo è in ascolto, le traiettorie da eseguire sono differenziate in quattro possibilità modalità di movimento presenti in questo nodo:

- **hexapod:** modalità di funzionamento da esapode.
- **shell:** modalità di funzionamento da sfera.

- **to\_exapod**: modalità di passaggio da sfera a esapode.
- **to\_shell**: modalità di passaggio da esapode a sfera.

La selezione della modalità avviene tramite una *callback* presente all'interno di questo nodo, chiamata **mode\_update**. Una callback è una funzione che il nodo esegue nel momento in cui riceve un messaggio da un topic al quale è sottoscritto. Questo permette al nodo di "ascoltare" informazioni dall'ambiente di ROS e quindi di interagire con gli altri nodi.

```
# callback for /mode topic listener
def mode_update(mode_msg):

    global mode
    mode_received = mode_msg.data

    if mode == "hexapod" and mode_received == "shell":
        print("3...2...1...")
        time.sleep(3)    # time given in order the robot to stop his movements, if any
        mode = "to_shell"
    elif mode == "shell" and mode_received == "hexapod":
        print("3...2...1...")
        time.sleep(3)    # time given in order the robot to stop his movements, if any
        mode = "to_hexapod"
```

Figura 33: Funzione di cambio modalità

Come visibile in figura, qualora al nodo trajectory\_handler arrivi un messaggio di modalità diverso da quello corrente, il nodo interromperà il suo funzionamento per tre secondi, un tempo sufficiente al robot per fermare gli eventuali movimenti in corso. Fatto questo, il nodo si porterà in una delle due modalità di passaggio.

Nel seguito di questo paragrafo viene illustrato il funzionamento di ognuna delle modalità sopra citate.

1. **hexapod**: è la modalità di partenza del robot all'interno della simulazione, con le gambe inferiori aperte e poggiate per terra e quelle superiori chiuse a emisfera. La condizione iniziale a cui è posto il robot è descritta da un insieme di traiettorie desiderate delle punte dei piedi (quindi in coordinate cartesiane), indipendenti dal tempo, che impongono al robot di stare fermo sul posto. Le velocità desiderate per le punte dei piedi vengono ovviamente poste a zero.

## Progetto esapode sferico

---

```
# rest condition
if not is_moving and tau == 0.:

    for leg_j in range(1, 7):

        # still robot trajectories
        x_still = dist * cos(2*(leg_j-1)*pi/6) - l_23_z * sin(2*(leg_j-1)*pi/6)
        y_still = l_23_z * cos(2*(leg_j-1)*pi/6) + dist * sin(2*(leg_j-1)*pi/6)
        z_still = height + x_still * tan(theta) + y_still * tan(psi)

        des_feet_position[(leg_j-1),0:3] = np.array([x_still, y_still, z_still])      # fills des_feet_position with rest conditions
        des_feet_velocity[(leg_j-1),0:3] = np.array([0, 0, 0])                         # fills des_feet_velocity with rest conditions

    gait_state = 0
```

Figura 34: "Still trajectories"

Nel momento in cui il nodo teleop\_key pubblica dei comandi di movimento, il nodo trajectory\_handler attiva la relativa callback, nella quale vengono ricevute le percentuali di movimento desiderato, ognuna di esse viene normalizzata rispetto alla somma di tutte e tre. I movimenti del robot vengono espressi come combinazione lineare di tre movimenti elementari: due movimenti di traslazione lungo due assi ortogonali fra loro e che descrivono un piano parallelo a quello del terreno e un movimento di rotazione lungo l'asse verticale (normale al pavimento). In base alle percentuali di questi movimenti elementari e del loro segno, il robot produrrà dei movimenti dati dalla somma di quelli elementari.

Modulo e segno dei movimenti elementari sono memorizzati nelle variabili globali *forward\_speed\_perc*, *lateral\_speed\_perc* e *rotation\_speed\_perc*. Se una di queste tre risulta diversa da zero, la variabile booleana "*is\_moving*" viene posta al valore *True* e viene abilitato il movimento del robot, tramite le funzioni *next\_positions* e *next\_velocities*.

```
# callback for /cmd_vel topic listener
def on_move_update(msg):

    global forward_speed_perc, lateral_speed_perc, rotation_speed_perc, is_moving

    if enable_commands == True:
        forward_speed_perc = msg.linear.x
        lateral_speed_perc = msg.linear.y
        rotation_speed_perc = msg.angular.z
    if enable_commands == False:
        forward_speed_perc = 0.
        lateral_speed_perc = 0.
        rotation_speed_perc = 0.

    if (forward_speed_perc != 0) or (lateral_speed_perc != 0) or (rotation_speed_perc != 0):
        forward_speed_perc = forward_speed_perc * abs(forward_speed_perc) / (abs(forward_speed_perc) + abs(lateral_speed_perc) + abs(rotation_speed_perc))
        lateral_speed_perc = lateral_speed_perc * abs(lateral_speed_perc) / (abs(forward_speed_perc) + abs(lateral_speed_perc) + abs(rotation_speed_perc))
        rotation_speed_perc = rotation_speed_perc * abs(rotation_speed_perc) / (abs(forward_speed_perc) + abs(lateral_speed_perc) + abs(rotation_speed_perc))

    # flag for distinguishing moving or still robot
    is_moving = (forward_speed_perc != 0) or (lateral_speed_perc != 0) or (rotation_speed_perc != 0) or (tau != 0)
```

Figura 35: Funzione di "ascolto" dei comandi da tastiera

2. **shell:** nel funzionamento in modalità sfera non sono stati implementati movimenti. Quando il robot si trova in questo stato mantiene la sua posizione,

imponendo delle posizioni desiderate tali da garantire il mantenimento della chiusura, e ponendo le velocità desiderate a zero.

3. **to\_shell**: quando il robot è in modalità esapode, premendo il tasto zero sul tastierino numerico si attiva il passaggio alla modalità sferica. Questo passaggio avviene tramite una procedura composta da diversi step, chiamati nel programma "*closure\_step*", rappresentati da degli indici numerici da zero a sei. Di seguito una breve descrizione di ciascuno step.

- **closure\_step 0**: nel primo step ci si assicura che le gambe superiori siano chiuse e portate nella loro posizione più alta possibile. Per farlo si impone il valore zero alla variabile *up\_angle* e il valore uno alla variabile *up\_quote*.
- **closure\_step 1 e 2**: lo step 1 viene eseguito in modo coordinato allo step 2, e contiene al suo inizio un check per consentire il passaggio allo step 3. Negli step 1 e 2 si chiede al robot rispettivamente di chiudere il folding fan e di fare dei piccoli passi in direzione radiale e ovviamente rivolti verso il centro del robot stesso. Questa coppia di operazioni viene ripetuta per quattro volte, prima di passare al terzo step.
- **closure\_step 3**: in questo step si portano le gambe superiori nella loro posizione più bassa, quindi quella con la variabile *up\_quote* posta a uno.
- **closure\_step 4**: passaggio di chiusura vero e proprio. Le gambe inferiori vengono portate a tre a tre nelle posizioni che vanno a comporre la sfera della modalità "shell". Un aspetto importante da sottolineare è che questi movimenti, essendo riferiti ai centri delle sfere di cui ogni shell inferiore fa parte, necessitano di una diversa inversione cinematica rispetto ai movimenti fin qui eseguiti..

```

# full robot closing
elif closure_step == 4:

    tau += real_dt / (GAIT_STEP_TIME * 2)

    for leg_j in range(1, 7):
        des_feet_position[(leg_j-1),:] = bottom_closure_centers(leg_j-1, is_first_step, tau)
        des_feet_velocity[(leg_j-1),:] = bottom_closure_centers_velocities(leg_j-1, is_first_step, tau)

    if tau >= 1 and is_first_step:
        is_first_step = False
        tau = 0

    elif tau >= 1 and not is_first_step:
        tau = 0
        closure_step = 5

    clik_index = 1

    des_feet_msg.data = kinematic_laws.flatten(des_feet_position)
    des_feet_vel_msg.data = kinematic_laws.flatten(des_feet_velocity)
    theta1_msg.data = q1

    des_feet_pub.publish(des_feet_msg)
    des_feet_vel_pub.publish(des_feet_vel_msg)
    clik_index_msg.data = clik_index
    clik_index_pub.publish(clik_index_msg)
    theta1_pub.publish(theta1_msg)

```

Figura 36: Closing step 4 e cambio di clik\_index

- **closure\_step 5:** step finale, nel quale si effettua soltanto il cambio di modalità andando a porre la variabile ”mode” a ”shell”.
4. **to\_exapod:** anche il passaggio da sfera a esapode avviene per step, questa volta chiamati ”*opening\_step*”. A differenza della procedura di chiusura, gli step di apertura non verranno descritti nel dettaglio, poiché, salvo alcuni dettagli, questa è il duale della precedente. E’ sufficiente percorrere gli step descritti nella procedura di chiusura al contrario per ottenere quella di apertura.

#### 4.2.3 inverse\_kinematic\_server

Il nodo `inverse_kinematic_server` si occupa di ”ascoltare” le traiettorie di posizioni e velocità desiderate dal nodo `trajectory_handler` e di produrre in uscita gli angoli di giunto desiderati da inviare ai motori simulati (con degli accorgimenti per adattare il codice al caso di robot reale).

Laddove possibile, il nodo utilizza la soluzione in forma chiusa del problema di inversione cinematica, ma per la maggior parte dei movimenti la soluzione implementata è il metodo *CLIK* (*Closed-Loop Inverse Kinematics*). In particolare, la soluzione in forma chiusa è utilizzata per i movimenti delle gambe superiori, mentre per le gambe inferiori si utilizza la CLIK.

Al momento dell’inizializzazione il nodo attiva le sue sottoscrizioni e pubblicazioni, e successivamente avvia un ciclo di esecuzione della funzione principale del programma, la funzione `eval_inversion`. Al suo interno viene eseguita l’inversione cinematica di tutto il robot, prendendo le variabili necessarie dalle callback associate ai messaggi provenienti dal nodo `trajectory_handler`.

## Progetto esapode sférico

---

```
# listen to des_pose and publish desired joint angles at rate 1/DT
if __name__ == "__main__":
    rospy.init_node('inverse_kinematic_server', anonymous = False)
    rospy.Subscriber('des_feet_position', Float64MultiArray, update_des_feet_position)
    rospy.Subscriber('des_feet_velocity', Float64MultiArray, update_des_feet_velocity)
    rospy.Subscriber('theta1', Float64, theta1_update)
    rospy.Subscriber('clik_index', Float64, clik_index_update)
    rospy.Subscriber('des_upper_positions', Float64MultiArray, update_des_upper_positions)
    rospy.Subscriber('joint_states', JointState, get_simulated_joints)

    motor_controller_pub = rospy.Publisher('/shellbot_legs_controller/command', Float64MultiArray, queue_size=1)
    actual_feet_pub = rospy.Publisher('actual_feet', Float64MultiArray, queue_size=1)
    actual_upper_pub = rospy.Publisher('actual_upper', Float64MultiArray, queue_size=1)
    inv_kin_pos_pub = rospy.Publisher('inv_kin_pos', Float64MultiArray, queue_size=1)
    qdot_pub = rospy.Publisher('qdot', Float64MultiArray, queue_size=1)

    print("inverse_kinematic_server is online.")
    rate = rospy.Rate(1 / DT) # Hz

    try:
        while not rospy.is_shutdown(): # check for Ctrl-C
            eval_inversion(motor_controller_pub) # main function
            rate.sleep() # "rate" (instead of "time") can handle simulated time
    except rospy.ROSInterruptException:
        pass
```

Figura 37: main function del nodo inverse\_kinematic\_server

La funzione *eval\_inversion* si occupa dell'inversione cinematica del robot, della costruzione degli angoli motore da inviare al simulatore e della loro pubblicazione. Per questo motivo l'unico argomento che le viene passato è proprio il publisher degli angoli di giunto.

A seconda che le traiettorie desiderate per le gambe inferiori siano riferite alle punte dei piedi o ai centri delle sfere, le inversioni cinematiche saranno differenti, e la loro selezione è gestita dalla variabile *clik\_index*, riempita dall'apposita callback. Le funzioni cinematiche sono richiamate dalla libreria *shellbot\_kinematic\_laws*, descritta nel seguito. Nelle figure seguenti vengono riportate le due inversioni cinematiche e la callback che le gestisce.

## Progetto esapode sferico

---

```
# ..... CLIK - bottom legs tips .....
if clik_index == 0:
    for j in range(1, 7):
        J_a_inv_j = kinematic_laws.qbj_jacob(j, joint_states[(4*j-4) : (4*j)])
        qk_old = np.array([theta1, joint_states[4*j-3], joint_states[4*j-2], joint_states[4*j-1]]) # inverse of jacobian matrix (3x3)
        x_j = x[3*j-3 : 3*j] # current joints positions
        # position vector j_th bottom foot

        ke = np.multiply((x_d[j-1, 0:3] - x_j), k) # 1x3
        alpha = np.multiply(x_d_dot[j-1, 0:3], 1)

        q_dot.append(theta1)
        q_dot.append(J_a_inv_j.dot(alpha + ke)[0])
        q_dot.append(J_a_inv_j.dot(alpha + ke)[1])
        q_dot.append(J_a_inv_j.dot(alpha + ke)[2])

        qk[4*j-4 : 4*j] = qk_old + np.multiply(q_dot[4*j-4 : 4*j], DT) - qk_offset # joint angles j-th bottom leg
```

Figura 38: CLIK riferita alle punte delle gambe inferiori

```
# ..... CLIK - bottom legs shell centers .....
elif clik_index == 1:
    for j in range(1, 7):

        J_a_inv_j = kinematic_laws.center_jacob(j, joint_states[(4*j-4) : (4*j)])
        qk_old = np.array([joint_states[4*j-4], joint_states[4*j-3], joint_states[4*j-2], joint_states[4*j-1]]) # inverse of jacobian matrix (4x4)
        x_j = x_centers[4*j-4 : 4*j] # current joints positions
        # position vector j_th bottom center

        ke = np.multiply((x_d[j-1, :] - x_j), k_vec) # 1x4
        alpha = np.multiply(x_d_dot[j-1, :], 0.3)

        q_dot.append(J_a_inv_j.dot(alpha + ke)[0])
        q_dot.append(J_a_inv_j.dot(alpha + ke)[1])
        q_dot.append(J_a_inv_j.dot(alpha + ke)[2])
        q_dot.append(J_a_inv_j.dot(alpha + ke)[3])

        qk[4*j-4 : 4*j] = qk_old + np.multiply(q_dot[4*j-4 : 4*j], DT) - qk_offset # joint angles j-th bottom leg
```

Figura 39: CLIK riferita ai centri delle sfere descritte dalle gambe inferiori

```
# callback for /clik_index topic listener
def clik_index_update(clik_index_msg):
    global clik_index

    if clik_index != clik_index_msg.data:
        clik_index = clik_index_msg.data
        print("Clik index changed: ", clik_index)
```

Figura 40: callback function per la variabile clik\_index

Le due CLIK sopra illustrate sono gestite da un *if* logico che va a selezionare una delle due. Per quanto riguarda invece la parte superiore del robot, la sua inversione cinematica in forma chiusa è valutata ad ogni esecuzione della funzione eval\_inversion. Successivamente viene costruito il vettore da destinare ai motori, ossia il vettore *list\_of\_desidered\_angles*.

```
# ----- upper legs inverse kinematics -----
upper_joint_angles_i = kinematic_laws.up_inversion(up_position, up_angle) - q_u_offset
for i in range(1, 7):
    upper_joint_angles.append(upper_joint_angles_i[0])
    upper_joint_angles.append(upper_joint_angles_i[1])
# full joint positions vector
list_of_desired_angles = qk + upper_joint_angles
```

Figura 41: Inversione cinematica delle gambe superiori e costruzione del vettore di angoli di giunto desiderati

Il vettore list\_of\_desired\_angles contiene trentasei elementi, perché nel simulatore ci sono trentasei motori simulati: ventiquattro per le gambe inferiori e dodici per le gambe superiori.

La differenza con il robot reale sta nella gestione dei motori delle gambe inferiori, che nella realtà sono diciannove. I sei motori in più presenti nel simulatore sono dovuti al fatto che in simulazione si è preferito modellare il motore del folding fan come sei motori agenti sugli assi degli ingranaggi appartenenti alle singole gambe piuttosto che come un singolo motore centrale.

La ragione di questa scelta risiede nel fatto che questa configurazione di motori si presta meglio ad essere implementata nel simulatore, il quale avrebbe potuto fornire risultati poco soddisfacenti in fatto di interazioni fra ingranaggio centrale e ingranaggi delle singole gambe.

Per portare il vettore di angoli di giunto desiderati al caso di robot reale è sufficiente rimuovere gli angoli relativi agli ingranaggi delle singole gambe e inserire il valore dell'angolo di giunto del folding fan. Questo avrà un valore pari a uno qualsiasi degli angoli virtuali (che ricordiamo essere tutti uguali, in quanto risultato dell'interazione dell'ingranaggio centrale con gli ingranaggi delle gambe), scalato per il rapporto di trasmissione fra ingranaggio centrale e quello di una singola gamba.

## Progetto esapode sferico

---

```
# ----- real robot: joint number reduction from 36 to 31 and transmission ratio division -----
if (motors_number == 31):

    leg_gear_joint_angle = list_of_desired_angles[0]                                # joint angle of first leg gear
    transmission_rate = 1.0                                                       # inserted in the on-board version
    central_motor_joint_angle = leg_gear_joint_angle * transmission_rate

    # starting legs gears angles removing. Indexes: 0, 4, 8, 12, 16, 20
    del list_of_desired_angles[20]
    del list_of_desired_angles[16]
    del list_of_desired_angles[12]
    del list_of_desired_angles[8]
    del list_of_desired_angles[4]
    del list_of_desired_angles[0]

    # central joint angle inserting
    list_of_desired_angles.insert(0, central_motor_joint_angle)
```

Figura 42: Angoli di giunto nel caso di robot reale

L'algoritmo CLIK richiede la conoscenza degli angoli di giunto correnti, per farne la differenza con quelli desiderati e calcolare l'errore tra i due valori. Nella simulazione questi valori vengono pubblicati nell'ambiente ROS dal simulatore Gazebo, il quale costituisce un vero e proprio nodo di ROS. Gli angoli di giunto vengono inseriti all'interno di un vettore a disposizione del nodo inverse\_kinematic\_server, tramite l'apposita callback *get\_simulated\_joints*.

Questa callback è di particolare importanza, perché in ogni sua esecuzione, oltre alla lettura degli angoli simulati, viene effettuato il calcolo delle posizioni degli end-effector di interesse, anche qui differenziati tramite la variabile *clik\_index*. Le posizioni vengono calcolate per cinematica diretta, tramite delle funzioni presenti nella libreria shellbot\_kinematic\_laws, e vengono rese disponibili sia all'interno del nodo che all'interno dell'ambiente ROS, per permettere al nodo trajectory\_handler di utilizzarle.

```
# current feet positions calculation (f is the direct kinematics function)
for j in range(1, 7):
    # x_j --> pose (x, y, z, phi, theta, psi)
    x_j = kinematic_laws.f(j, joint_states[4*j-4], joint_states[4*j-3], joint_states[4*j-2], joint_states[4*j-1])[0:3,0]    # take position but not orientation
    x[3*j-3 : 3*j] = x_j    # fill feet position vector - only positions

# current upper legs coordinates
x_up = kinematic_laws.f_upper(joint_states[24], joint_states[25])

# current bottom centers coordinates
if clik_index == 1:

    for j in range(1, 7):
        x_center_j = kinematic_laws.f_center(j, joint_states[4*j-4], joint_states[4*j-3], joint_states[4*j-2], joint_states[4*j-1])
        x_centers[4*j-4 : 4*j-2] = x_center_j[0:2,0]          # fill center pose vector - 4/6 values
        x_centers[4*j-2 : 4*j] = x_center_j[3:5,0]
```

Figura 43: Calcolo della cinematica diretta

#### 4.2.4 shellbot\_kinematic\_laws

Di tutti gli script descritti fin ora, la libreria shellbot\_kinematic\_laws è l'unica a non costituire un nodo ROS. Al suo interno è presente l'insieme di funzioni cinematiche, dirette e inverse, utili al funzionamento del nodo inverse\_kinematic\_server, oltre alla definizione dei parametri geometrici del robot.

Di seguito una descrizione di ciascuna di esse:

- **f\_bottom**: funzione cinematica diretta relativa alle punte delle gambe inferiori.
- **f\_center**: funzione cinematica diretta relativa ai centri delle sfere descritte dalle shell delle gambe inferiori.
- **f\_upper**: funzione cinematica diretta relativa ai centri delle sfere descritte dalle shell delle gambe superiori.
- **qbj\_jacob**: calcolo del jacobiano analitico relativo alle punte delle gambe inferiori.
- **centerj\_jacob**: calcolo del jacobiano analitico relativo ai centri delle sfere descritte dalle shell delle gambe inferiori.
- **up\_inversion**: funzione cinematica inversa relativa ai centri delle sfere descritte dalle shell delle gambe superiori.

### 4.3 Modello del robot in simulazione

Nell'ambiente simulativo Gazebo viene testato il comportamento del robot, il quale deve essere quindi riprodotto in maniera quanto più dettagliata e fedele possibile alla realtà. Lo standard di rappresentazione informatica di strutture robotiche è l'*URDF (Unified Robot Description Format)*, utilizzato anche in questo progetto. Dal momento che URDF non prevede l'inserimento di variabili, per sfruttarne a pieno le potenzialità questo viene utilizzato insieme al linguaggio *Xacro (XML macro)*, il quale permette sia la composizione modulare di varie porzioni di codice URDF che l'introduzione di variabili.

I file utilizzati per la rappresentazione del robot sono elencati nella sezione di mappatura dei file, esposta in precedenza.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name ="shell_robot">

    <!-- legs index variables definitions -->
    <xacro:arg name="leg_num_B" default="" />
    <xacro:arg name="leg_num_U" default="" />

    <!-- xacro files including -->
    <xacro:include filename="$(find shellbot_pkg)/models/chassis.urdf.xacro" />
    <xacro:include filename="$(find shellbot_pkg)/models/shell_robot_bottom_leg.urdf.xacro"/>
    <xacro:include filename="$(find shellbot_pkg)/models/shell_robot_upper_leg.urdf.xacro"/>
    <xacro:include filename="$(find shellbot_pkg)/models/variables.urdf.xacro"/>

    <!-- bottom legs definitions - index: leg_B -->
    <xacro:bottom_leg leg_num_B="1"/>
    <xacro:bottom_leg leg_num_B="2"/>
    <xacro:bottom_leg leg_num_B="3"/>
    <xacro:bottom_leg leg_num_B="4"/>
    <xacro:bottom_leg leg_num_B="5"/>
    <xacro:bottom_leg leg_num_B="6"/>

    <!-- upper legs definitions - index: leg_U -->
    <xacro:upper_leg leg_num_U="1"/>
    <xacro:upper_leg leg_num_U="2"/>
    <xacro:upper_leg leg_num_U="3"/>
    <xacro:upper_leg leg_num_U="4"/>
    <xacro:upper_leg leg_num_U="5"/>
    <xacro:upper_leg leg_num_U="6"/>

    <!-- physics engine definition, included in variables.urdf.xacro -->
    <xacro:ode_solver/>

    <!-- Gazebo plugin for ROS Control -->
    <gazebo>
        <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
            <robotNamespace>/</robotNamespace>
            <legacyModeNS>true</legacyModeNS>
        </plugin>
    </gazebo>
</robot>
```

Figura 44: File principale del modello in URDF

Nel file è possibile osservare l'inclusione degli altri file e le loro istanze, distinte da un indice numerico che va da 1 a 6, presenti sia per le gambe inferiori che superiori. Gli indici numerici permettono di posizionare le gambe nella loro posizione corretta, andando a moltiplicare un sesto dell'angolo giro nei file di descrizione di gambe inferiori e superiori.

#### 4.4 Versione on-board

In questo paragrafo vengono illustrate le differenze tra il programma di simulazione e la sua versione on-board, presente sulla scheda NVIDIA Jetson Nano a bordo del robot.

Di seguito viene riportato l'rqt\_graph della versione on-board, tramite il quale è possibile apprezzare le differenze con la versione simulata. Quella rappresentata in figura è la versione finale testata sul robot, nella quale alcune delle funzionalità implementate nel passaggio da simulazione e robot reale sono state compattate nello

script RT\_inverse\_kinematic\_and\_control. Nel seguito verrano descritte entrambe le versioni on-board, spiegandone analogie e differenze.

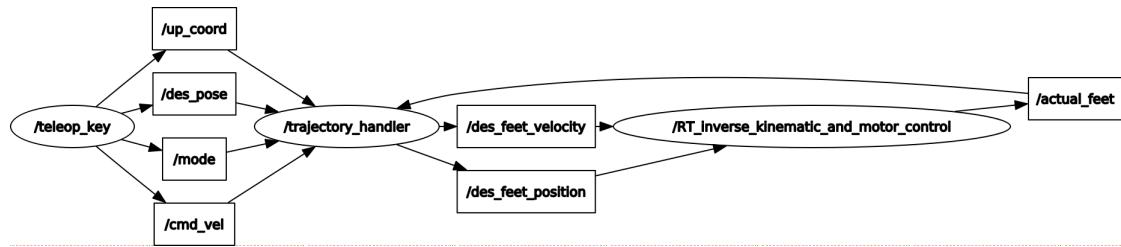


Figura 45: rqt\_graph del software a bordo del robot

Il nodo teleop\_key funziona esattamente come nella simulazione, pubblicando gli stessi topic già visti in precedenza. Anche il nodo trajectory\_handler funziona allo stesso modo, "ascoltando" i messaggi pubblicati dal nodo teleop.key e producendo le traiettorie desiderate sia in termini di posizioni che di velocità.

Tuttavia dall'rqt\_graph della versione on-board si può notare come manchino i topic /des\_upper\_positions, /theta1 e /clik\_index: questa scelta è dovuta al fatto che nella versione del robot reale non sono implementati i movimenti delle gambe superiori e i movimenti di chiusura e apertura, quindi la pubblicazione di quei topic risulta superflua.

E' di particolare interesse la descrizione dello script *RT\_inverse\_kinematic\_and\_control*, il cui nome è esplicativo delle sue funzionalità. Questo script si occupa di invertire la cinematica del robot per produrre gli angoli di giunto desiderati da mandare ai motori, comunicare con essi per trasmettere tali angoli, leggere gli angoli di giunto correnti e metterli a disposizione all'interno dell'ambiente di ROS.

La sigla RT (Real-Time) rappresenta il fatto che lo script cerca di mantenere dei vincoli temporali di esecuzione di queste funzioni. Questa necessità nasce dal tentativo di risolvere alcuni problemi riscontrati nella comunicazione tra scheda e motori con una gestione in tempo reale di alcune parti del codice di simulazione.

Nel seguito una descrizione dei nodi appena citati.

#### 4.4.1 motor\_control\_ros

Il nodo motor\_control\_ros si occupa di interfacciare il nodo inverse\_kinematic\_server con i motori. Svolge quindi le veci del nodo automaticamente creato dal simulatore Gazebo al momento del lancio della simulazione, andando a comunicare gli angoli di

giunto desiderati ai motori e leggendo gli angoli di giunto correnti per comunicarli al nodo `inverse_kinematic_server`.

Oltre alle funzioni di lettura e scrittura, nel caso di robot reale occorre anche avere delle funzioni ausiliare per interagire coi motori. Tali funzioni sono presenti nella libreria `servo.py`, la quale a sua volta richiama altre funzioni della libreria `ax12.py`, contenente funzioni per l'interazione di basso livello coi motori Dynamixel AX12, ma valide anche per i Dynamixel AX18.

Nella sua inizializzazione, il nodo `motor_control_ros` crea un'istanza della classe `Servo`, contenuta nell'omonima libreria python, e imposta gli angoli di giunto desiderati ai valori di riposo del robot nella sua forma di esapode.

```
# Update frequency for joints desired angles (pay attention to
# gait_handler.DT value, something faster than that is just useless)
DT = 0.5 # [s]

joints_msg = Float64MultiArray()
joints_msg.layout.dim = [MultiArrayDimension()]
joints_msg.layout.dim[0].stride = 19

s = Servo(used_servo=19,minID=0,maxID=18,baudrate=1000000,setup=True,verbose=True)

command_rad = []

legs = 6
command_deg = [0] * 19
command_deg[0] = s.conv_to_servo(0, 0.) # command servo 0 folding fan

for i in range(1,legs+1):

    command_deg[3*i-2] = s.conv_to_servo(3*i-2, 0.)
    command_deg[3*i-1] = s.conv_to_servo(3*i-1, -4.*pi/180)
    command_deg[3*i] = s.conv_to_servo(3*i, 89.*pi/180)

joint_angles = [0] * 19
joint_angles[0] = 0.

for i in range(1,legs+1):

    joint_angles[3*i-2] = 0.
    joint_angles[3*i-1] = -4.*pi/180
    joint_angles[3*i] = 89.*pi/180
```

Figura 46: Inizializzazione del nodo `motor_control_ros`

La parte del nodo eseguita in maniera ciclica contiene due funzioni. La prima, `checkChangedAngles`, si occupa di controllare se gli angoli di giunto desiderati siano differenti da quelli correnti, e nel caso lo siano di inviare ai motori i nuovi angoli di giunto tramite la funzione `setAngleAll`. La seconda, `updateAngleAll`, effettua una lettura di tutti gli angoli di giunto correnti e li salva nel vettore `curr_angles`. Questi valori sono pubblicati ad ogni ciclo di esecuzione, in modo che l'ambiente ROS abbia sempre a disposizione le letture degli angoli più aggiornate. Tutte le funzioni citate sono presenti nella libreria `servo.py`.

```

rate = rospy.Rate(1 / DT) # Hz

try:
    prev_t = time()

    while not rospy.is_shutdown():
        t2 = time()

        if s.checkChangedAngles(command_deg):
            s.setAngleAll(command_deg)
            print("command_deg = ",end="")
            for i in range(0, len(command_deg)):
                print(round(command_deg[i],2),end="\t")
            print()

            s.updateAngleAll()
            elapsed2 = time() - t2
            print("elapsed2 = ", elapsed2)

            if time() - prev_t > 5:
                s.printAngles()
                s.printVoltage()
                prev_t = time()

            # get joints angles
            joint_angles[0] = s.conv_from_servo(0, s.curr_angles[0])
            for i in range(1, s.num):
                joint_angles[i] = s.conv_from_servo(i, s.curr_angles[i])

            joints_msg.data = joint_angles
            joint_states_pub.publish(joints_msg)

```

Figura 47: Corpo centrale del nodo motor\_control\_ros

#### 4.4.2 RT\_inverse\_kinematic\_and\_control

Come accennato in precedenza, lo script RT\_inverse\_kinematic\_and\_control è stato creato successivamente alla versione del software on-board contenente i nodi inverse\_kinematic\_server e motor\_control\_ros, quindi alla versione in tutto e per tutto analoga alla simulazione.

Questa prima versione del software on-board ha presentato dei problemi in fatto di comunicazione fra scheda e motori. Si è riscontrato infatti che spesso i motori non ricevessero comandi di movimento corretti, e che quindi il robot avesse movimenti indesiderati.

Uno dei tentativi effettuati per risolvere questo problema è stato quello di imporre un vincolo temporale all'esecuzione dei processi di lettura di angoli di giunto, inversione cinematica e pubblicazione degli angoli desiderati, tramite lo script descritto in questa sezione. E' importante sottolineare che si tratti di uno script e non di un nodo ROS, perché ROS non prevede meccanismi di esecuzione del codice in tempo reale.

## Progetto *esapode sferico*

---

In questo script viene utilizzata la libreria Python "threading", tramite cui è possibile definire un processo (thread), e un suo tempo di esecuzione desiderato, da poter monitorare per vedere se il vincolo temporale di esecuzione sia soddisfatto o meno. Di seguito un'immagine del codice eseguito ciclicamente dal processo in questione, nel quale sono compattate le funzioni svolte dai nodi `inverse_kinematic_server` e `motor_control_ros` nella precedente versione del software on-board.

```
### control loop
while True:

    if self.ROS_connector_RT.DataAvailable():

        print("DataAvailable: ", self.ROS_connector_RT.DataAvailable())

        # for timing debug
        # before next iteration begins, measure how much time passed since the last iteration started
        if PRINT_MEAS_PERIOD:

            self.meas_period = (time.time()-self.t_init)*1000

        # new iteration starting point
        # save entry time
        self.t_init=time.time()

        # THREAD MAIN CODE BEGIN
        timestamp = time.time()

        x_d, x_d_dot = self.ROS_connection_get_data()

        eval_inversion_and_motor_control(x_d, x_d_dot) # read, calculate, write (for motors)

        global prev_t

        if time.time() - prev_t > 5:
            s.printAngles()
            s.printVoltage()
            prev_t = time.time()

        print('real-time execution period:', time.time() - timestamp)

        self.ROS_connector_RT.Set_data(x)
        # THREAD MAIN CODE END

        self.wait_for_next_period()
```

Figura 48: Ciclo di esecuzione dello script `RT_inverse_kinematic_and_control`

La funzione condizionale verifica ad inizio ciclo l'arrivo o meno di nuove posizioni desiderate provenienti dal nodo `trajectory_handler`. In caso affermativo, il programma richiede al nodo `ROS_connector` di salvare posizioni e velocità desiderate rispettivamente nei vettori `x_d` e `x_d_dot`, i quali rappresentano gli argomenti da passare alla funzione principale del programma, la funzione `eval_inversion_and_motor_control`. Questa non è altro che una versione compatta della funzione `eval_inversion` presente nel nodo `inverse_kinematic_server` e delle funzioni presenti nel corpo centrale del nodo `motor_control_ros`, entrambe descritte in precedenza. Dopo l'esecuzione di `eval_inversion_and_motor_control`, lo script comanda al nodo `ROS_connector` di inviare gli angoli di giunto calcolati ai motori.

## Progetto *esapode sferico*

---

Inoltre, ad ogni iterazione del ciclo il programma stampa il tempo di esecuzione impiegato, indicando il rispetto o meno del tempo di esecuzione previsto. Di seguito un’immagine delle funzioni di misurazione dei tempi di calcolo impiegato da ogni singolo ciclo.

```
def execution_time_analysis(self):
    # exe debug info
    if PRINT_MEAS_EXE_TIME and PRINT_MEAS_PERIOD:
        self.exe_time = (time.time() - self.t_init)*1000
        if (self.exe_time >= PERIOD_PERC_ALERT/100*THREAD_PERIOD*1000) and (self.exe_time < THREAD_PERIOD*1000):
            print('||'+color.BOLD + color.YELLOW + ' exe time[ms]: ' + color.END , round(self.exe_time,4), ' \t ||')
        elif self.exe_time >= THREAD_PERIOD*1000:
            print('||'+color.BOLD + color.RED + ' exe time[ms]: ' + color.END , round(self.exe_time,4), ' \t ||' + color.BOLD + ' period[ms]: ' + color.END, round(self.meas_period,4), ' \t ||')
        else:
            print('||'+color.BOLD + ' exe time[ms]: ' + color.END , round(self.exe_time,4), ' \t ||' + color.BOLD + ' period[ms]: ' + color.END, round(self.meas_period,4), ' \t ||')

    elif PRINT_MEAS_PERIOD:
        print('||'+color.BOLD + ' period[ms]: ' + color.END , round(self.meas_period,4), ' \t ||')

    elif PRINT_MEAS_EXE_TIME:
        self.exe_time = (time.time() - self.t_init)*1000
        if self.exe_time >= PERIOD_PERC_ALERT/100*THREAD_PERIOD*1000:
            print('||'+color.BOLD + color.RED + ' exe time[ms]: ' + color.END , round(self.exe_time,4), ' \t ||')
        else:
            print('||'+color.BOLD + ' exe time[ms]: ' + color.END , round(self.exe_time,4), ' \t ||')

def wait_for_next_period():
    self.execution_time_analysis()
    time.sleep = max(min(self.T_thread - (time.time() - self.t_init), self.T_thread),0.001)
    time.sleep(time.sleep)
```

Figura 49: Funzioni di analisi dei tempi di esecuzione

### 4.4.3 ROS\_connector

Per il suo funzionamento all’interno dell’ambiente ROS, lo script *RT\_inverse\_kinematic\_and\_control* ha bisogno del nodo-libreria *ROS\_connector*, che fa le sue veci per le attività di pubblicazione e sottoscrizione ai topic necessari. L’espressione ”nodo-libreria” è dovuta al fatto che questo nodo contiene solo funzioni di pubblicazione e sottoscrizione ROS e funzioni di comunicazione con lo script *RT\_inverse\_kinematic\_and\_control*.

Nella figura seguente è riportata la definizione della classe omonima del nodo. Al suo interno sono definiti gli attributi della classe, corrispondenti ai messaggi ROS di competenza del nodo, l’inizializzazione del nodo e le sue sottoscrizioni e pubblicazioni.

## Progetto *esapode sferico*

---

```
class ROS_connector():
    def __init__(self, position_topic, velocity_topic, actual_feet_topic):
        self.data_position = Float64MultiArray()
        self.data_velocity = Float64MultiArray()
        self.position_topic = position_topic
        self.velocity_topic = velocity_topic
        self.actual_feet_topic = actual_feet_topic
        self.actual_feet_data = Float64MultiArray()
        self.actual_feet_data.layout.dim = [MultiArrayDimension()]
        self.actual_feet_data.layout.dim[0].stride = 18
        self.ready_position = False

        rospy.init_node('ROS_connection_node', anonymous=True, disable_signals=True)
        rospy.Subscriber(self.position_topic, Float64MultiArray, self.position_callback)
        rospy.Subscriber(self.velocity_topic, Float64MultiArray, self.velocity_callback)
        self.actual_feet_pub = rospy.Publisher('actual_feet', Float64MultiArray, queue_size=1)

    def position_callback(self,data):
        self.data_position = data
        self.ready_position = True

    def velocity_callback(self,data):
        self.data_velocity = data
```

Figura 50: Inizializzazione del nodo ROS\_connector

Infine il nodo contiene le funzioni *SetData*, *GetData* e *DataAvailable*, anch'esse messe a disposizione dello script *RT\_inverse\_kinematic\_and\_control*. La funzione *SetData* permette allo script di inviare i messaggi di angoli di giunto desiderati ai motori mentre la funzione *GetData* permette allo script di ricevere posizioni e velocità desiderate dal nodo *trajectory\_handler*. La funzione *DataAvailable* restituisce un valore logico con valore di verità *True* nel momento in cui ci siano nuovi dati di posizione (quindi anche di velocità) ancora non richiesti dallo script *RT\_inverse\_kinematic\_and\_control*, *False* in tutti gli altri casi.

```
def ROS_node(self):
    print("\tStarting ROS node.")
    rospy.spin()

def Get_data(self):
    return self.data_position.data, self.data_velocity.data

def Set_data(self,data):
    self.actual_feet_data.data = data
    self.actual_feet_pub.publish(self.actual_feet_data)
    self.ready_position = False

def DataAvailable(self):
    return self.ready_position

def start_thread(self):
    print("Starting thread")
    threading.Thread(target=self.ROS_node).start()
```

Figura 51: Funzioni contenute nel nodo ROS\_connector

## 5 Problemi affrontati

Il progetto inizialmente si poneva l’ambizioso obiettivo di progettare e montare un esapode che oltre a comminare avesse la possibilità di chiudersi a sfera e poter rotolare. Il progetto iniziale infatti prevedeva 31 motori in totale (18 per le gambe inferiori, 12 per le gambe superiori e 1 per il folding fan).

Si è però incorsi in vari problemi di implementazione, che hanno costretto alla riduzione della complessità totale a “solo” un esapode a base circolare per un totale di 19 motori.

### 5.1 Daisy chain dei servo

I motori scelti sono pensati per lavorare in daisy chain con una stessa porta UART a controllarli, come visto nel Paragrafo 3.1. Tuttavia, all’atto pratico si sono rivelati meno affidabili di quanto sperato con fault inaspettati, problemi di comunicazione e cadute di tensione non trascurabili (quest’ultimo difetto è stato però risolto da millefori rigenerando la tensione ogni due gambe, elaborazione già trattata nel Paragrafo 3.3.2).

Già in fase di stesura del programma si è vista necessaria l’aggiunta di un sistema che gestisse timeout error in modo che non interrompessero l’esecuzione ma venissero gestiti con una subroutine apposita (dettagli nel Paragrafo 3.5); inoltre il protocollo che usano i servo stessi è in grado di mandare dei segnali specifici per una serie di errori che fanno parte dell’error management, da un livello di tensione in ingresso troppo basso, all’overheating, a un errore di checksum: è stato però visto come spesso i servo riportassero alcuni di questi errori in maniera immotivata, ad esempio un errore di temperatura troppo elevata mentre il motore è fresco al tatto.

Un altro tipo di fault riscontrato è la possibilità che un servo smetta di rispondere ai comandi in modo imprevedibile, cosa che dopo alcuni test per verificare la mancata risposta alla ping, costringeva meramente a sostituire il motore. Per un singolo motore si è anche avuto a che fare con un fault di tipo meccanico, per cui il servo (uno degli AX-18A, che non si trovava neanche sotto sforzo fino a quel momento) ha iniziato a sperimentare forti vibrazioni che lo hanno reso inutilizzabile.

Il problema più importante affrontato è stato però quello sulla comunicazione, alla base di ogni altro problema logistico e la cui individuazione di anche solo un possibile approccio risolutivo ha portato all’insorgenza di ulteriori problematiche.

## 5.2 Il pin digitale

Come spiegato nel Paragrafo 3.3, per la comunicazione ai servo è necessario un pin digitale che effettui lo switch del buffer tra un'azione di trasmissione ai servo a una di ricezione dai servo in modalità half-duplex. Quando ci si è accorti che i motori fallivano la ping se collegati tutti e 19 assieme, oppure riuscivano nella ping ma fallivano i comandi di lettura dati e di movimento, sono state prese in considerazione varie strade da percorrere per cercare una soluzione.

Prima si è provato a considerare il checksum e far passare solo i valori su cui c'era la sicurezza che fossero corretti, sperando che su un determinato numero di prove si potesse controllare i servo anche se a una frequenza inferiore, data statisticamente dal fallimento in media di un certo numero di tentativi, ma è stato sperimentato che la percentuale “dati corretti / dati sbagliati” fosse troppo bassa, e ciò portava solo a far fallire i comandi ripetutamente.

Poi si è provato vari baud rate tra quelli disponibili, sperando che un baud rate più basso facilitasse la comunicazione, ma al contrario una volta portati a baud rate 57600, uno di quelli più comuni, essi sparivano dalla recezione della porta UART, ed è stato necessario un programma a parte su Arduino per riportarli a baud rate più alti (vedi Paragrafo 3.5.4). Anche da Arduino si potevano notare comunque alcuni motori che non rispondevano a prescindere dal baud rate richiesto, in un pattern casuale rispetto all'ordine dei servo stessi.

Un altro tentativo è stato fatto utilizzando un modulo USB-to-UART più potente, in modo da avere un baud rate massimo più alto, per cercare di avere più possibilità che i motori funzionassero a 1M. Inizialmente non è stato riscontrato alcun cambiamento nel comportamento dei servo, quindi si è ricorsi all'utilizzo di un oscilloscopio per analizzare bit per bit i messaggi scambiati: un errore che è stato subito evidente è stata la mancanza di una temporizzazione corretta tra gli switch del pin digitale, che portava all'inizio della ricezione prima che la trasmissione avesse completato il suo messaggio, o viceversa.

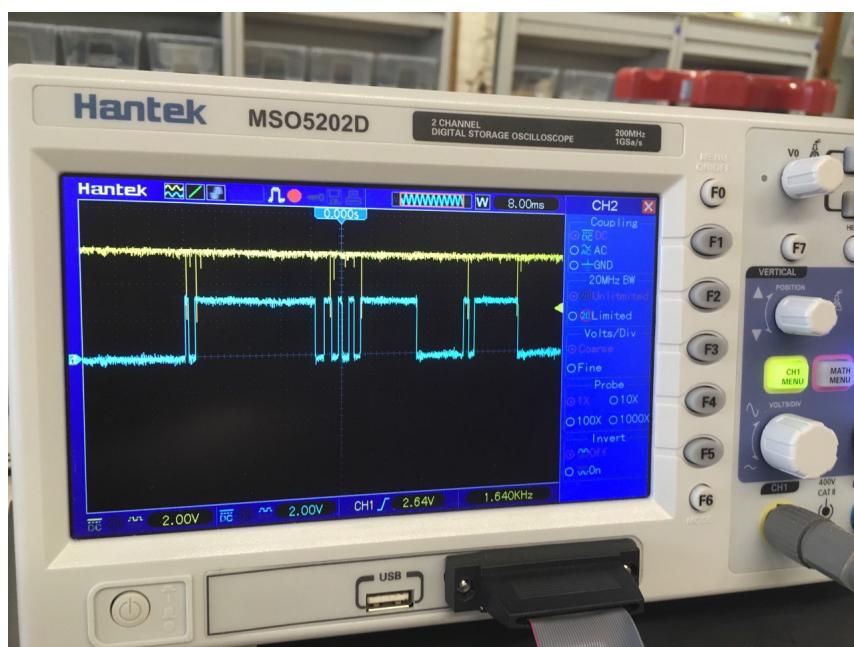
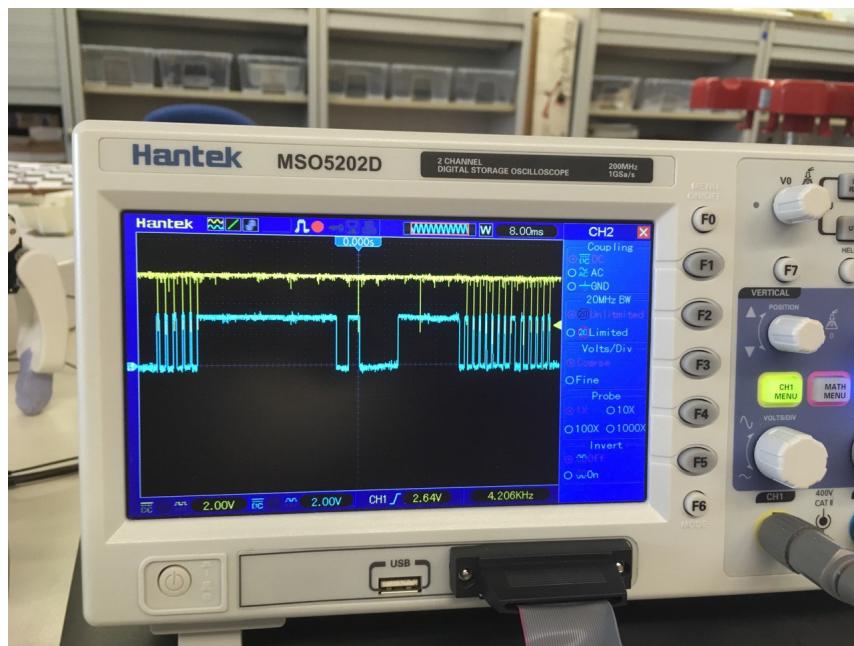


Figura 52: Esempi di comunicazione errata dati dallo switch scorretto del pin digitale. Il canale in giallo indica la comunicazione, quello blu la tensione del pin digitale. Si può notare come il pin digitale rimanga alzato (TX) per un tempo anche 10 volte superiore al tempo corretto, impedendo ai motori di comunicare con la Jetson.

Si è quindi rivisto il codice per la gestione dei delay interni alla classe Ax12 (Paragrafo 3.5.1) che prima erano settati a valori costanti e tarati per la comunicazione a 1M baud rate, e sono stati impostati come valori modulari che variano a seconda del baud rate impostato e il numero di byte richiesto nella comunicazione, in modo che fosse più flessibile.

Immediatamente si è riscontrato un netto miglioramento nelle comunicazioni, ma purtroppo non abbastanza da far muovere il robot: infatti, per quanto l'esapode effettui correttamente la ping per tutti i motori e riesca a mettersi in piedi raggiungendo la propria posizione di partenza senza problemi, una volta fatto partire un comando di movimento come una rotazione o la camminata in avanti, si riscontrano nuovamente le failure dei servo, e dall'osservazione tramite oscilloscopio si notano comunque fronti in cui il pin digitale aspetta prima di switchare un tempo anche 10 o 20 volte più lungo di quello che gli è stato richiesto nei delay.

Un tentativo periferico è stato fatto a questo punto anche sulla possibilità di alzare il livello di tensione del pin digitale, che da Jetson arriva a 3.3V, per uniformarlo ai 5V a cui lavora il resto del sistema (compresi i servo) utilizzando un logic level converter, ma senza risultato. Le dinamiche in dettaglio sono state affrontate nel Paragrafo 3.3.

Infine, si è pensato di rendere più precisi i delay (che comunque di default su Linux dovrebbero essere piuttosto precisi, sicuramente più che da Windows) implementando una struttura a thread che contenesse il nodo per la gestione dei servo (`motor_control_ros`) e quello per l'inversione cinematica (`inverse_kinematic_server`) e avere così un nuovo script unificato (`RT_inverse_kinematic_and_control`, vedi Paragrafo 4.4.2).

In questo modo si sarebbero rese resistenti agli interrupt le operazioni che gestiscono le funzioni `sleep`, in modo che venissero svolte atomicamente senza che il codice potesse essere interrotto da eventuali chiamate di sistema esterne, e rendere il programma un sistema in tempo reale. Purtroppo anche questa prova non ha sortito gli effetti sperati.

### 5.3 Possibili soluzioni per sviluppi futuri

I problemi riscontrati nella realizzazione hardware sono relativi alla comunicazione tra la scheda Jetson Nano e i motori: per garantire un corretto movimento del robot è necessario che le operazioni di lettura degli angoli di giunto, inversione cinematica e invio di angoli di giunto desiderati avvengano in tempi prestabiliti e a una frequenza adeguata.

Si è verificato che la scheda presente a bordo nel robot non è in grado di garantire le tempistiche desiderate per una catena di diciannove motori. Il fatto che sulla scheda sia presente un sistema operativo, con processi continuamente in corso in background e il fatto che ROS non garantisca un comportamento in tempo reale portano all'introduzione di ritardi imprevedibili, portando il robot a bloccarsi o a non seguire le traiettorie desiderate.

Una possibile strada per risolvere questo problema è quella di far eseguire le operazioni di lettura degli angoli, inversione cinematica e scrittura degli angoli desiderati da una scheda di basso livello programmata in C, ad esempio una scheda STM32 Nucleo. Il vantaggio di aggiungere questa scheda è data dal fatto che questa, essendo priva di un sistema operativo e occupandosi solo di eseguire quelle specifiche operazioni, sarebbe probabilmente in grado di eseguirle ad alta frequenza, senza i ritardi sopra menzionati.

Le operazioni di lettura input da utente e calcolo delle traiettorie possono continuare ad essere eseguite dalla Jetson Nano nel framework ROS, per poi comunicare le posizioni desiderate alla scheda di basso livello.

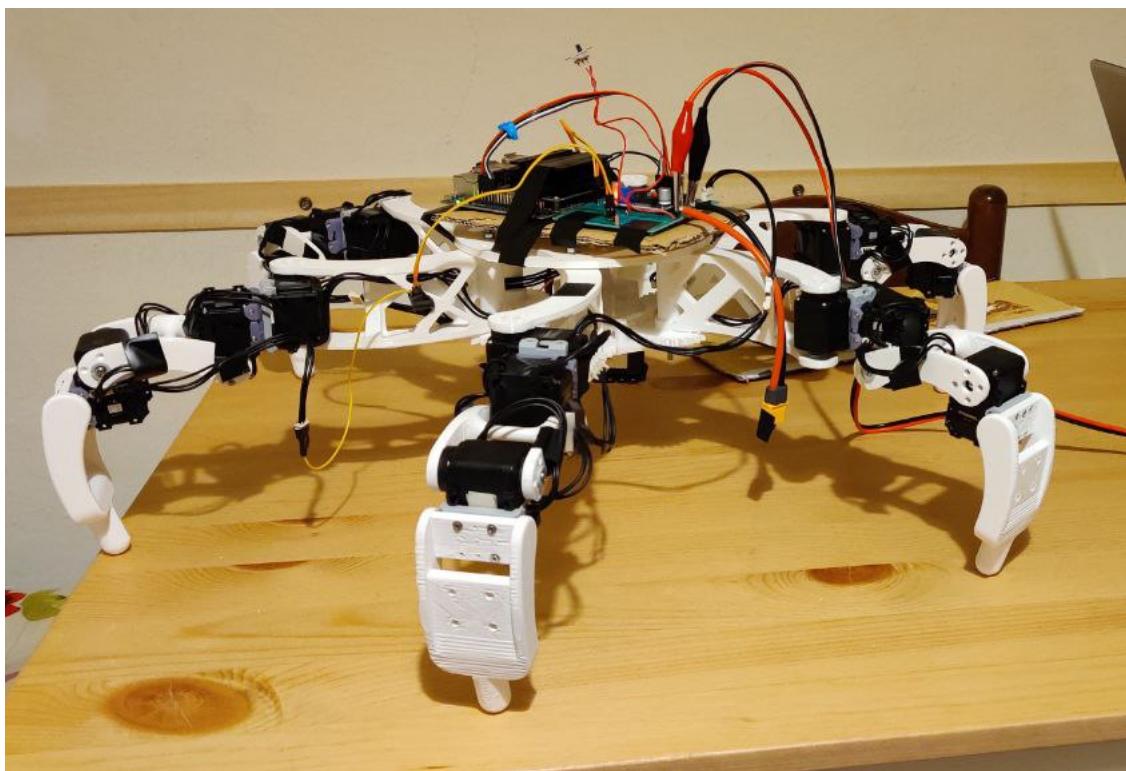


Figura 53: Immagine del robot realizzato