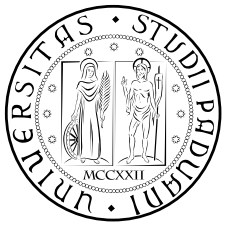

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI TECNICA E GESTIONE DEI SISTEMI INDUSTRIALI
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA MECCATRONICA

TESI DI LAUREA MAGISTRALE

SVILUPPO DI ALGORITMI DI CONTROLLO DI FORZA PER ROBOT COLLABORATIVI

Relatore: Prof. Oboe Roberto

Laureando: Falqueto Placido
1162797

ANNO ACCADEMICO: 2019-2020

SOMMARIO

In questa tesi vengono implementati algoritmi di controllo di forza sul robot collaborativo leggero Panda della Franka Emika. Risulta intuitivo implementare il controllo di impedenza, a causa della disponibilità dei segnali di coppia di ogni singolo giunto. La natura collaborativa del Panda, infatti, lo rende il perfetto candidato per applicazioni di questo genere.

In particolare ci si focalizza sul controllo di impedenza e le sue varie componenti (forze elastiche e di attrito viscoso). Oltre al controllo di forza si analizza la stima del peso del carico del manipolatore e della visualizzazione in tempo reale dei segnali del robot tramite Simulink.

La visualizzazione in realtime funziona egregiamente grazie anche alla frequenza di ciclo del controllo del Panda di 1Khz, che risulta abbastanza elevata da garantire un ottima qualità della gestione sia del controllo che delle funzioni secondarie, come la visualizzazione tramite comunicazione UDP. La stima del payload inoltre risulta molto precisa e garantisce il funzionamento degli algoritmi di controllo, permettendo una corretta compensazione delle forze peso e quindi il mantenimento stabile e stazionario del braccio.

"La scienza di oggi è la tecnologia di domani."

Edward Teller

RINGRAZIAMENTI

Le prime persone a cui devo dire grazie per questo traguardo sono i miei genitori, fonte di sostegno e di coraggio. Vorrei ringraziare il mio relatore prof. Oboe, per le opportunità che mi ha offerto e per la sua disponibilità. Ringrazio il dottorando del professore, Andrei, per il supporto fornitomi nell'esecuzione degli esperimenti. Ringrazio inoltre i miei colleghi di corso, in particolare Marco, Leandro, Nicola D. e Nicola T., per aver reso questo percorso memorabile. Un grazie speciale alla mia ragazza Elisabetta e ai miei amici che mi hanno supportato direttamente e indirettamente nella stesura di questa tesi.

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

1

INTRODUZIONE

In questo elaborato viene descritto l'utilizzo di algoritmi di controllo di forza nel robot collaborativo Panda della Franka Emika, e vengono approfonditi argomenti di supporto quali la comunicazione tra manipolatore e dispositivi di visualizzazione in tempo reale o di controllo remoto e la stima dei parametri dinamici del payload.

Il controllo di forza è reso necessario dalla volontà di lavorare a stretto contatto con il manipolatore, situazione estremamente comune nel campo della robotica riabilitativa, medicale ma, sempre più, anche nel settore industriale. Basti pensare al caso di un'operazione in cui l'obiettivo del robot è quello di supportare il peso di un utensile, mentre l'operatore esegue una lavorazione con esso. Grazie al manipolatore, l'operatore può lavorare con maggiore ergonomia.

Il controllo cinematico classicamente utilizzato per la grande maggioranza dei task di un manipolatore industriale ha dei considerevoli problemi per quanto riguarda la sicurezza degli operatori che lavorano nello stesso ambiente del robot, e in più non è sufficiente per tutte le applicazioni che vadano oltre alla mera movimentazione.

Per far fronte a queste problematiche ed ampliare gli ambiti di applicazione dei robot si sfruttano algoritmi di controllo di forza in combinazione con controlli cinematici, portando alla luce quelli che vengono chiamati controlli di impedenza. Questa categoria di controlli ha come obiettivo quello di rendere

più naturale l'interazione uomo-macchina, riducendo i rischi per l'operatore e al contempo rendendo possibile applicazioni nuove.

Questa tesi sarà suddivisa nei seguenti capitoli:

2. **Setup Sperimentale**

   Descrizione della componentistica hardware e del software utilizzati negli esperimenti di laboratorio.

3. **Visualizzazione in tempo reale dei dati dei sensori del robot tramite comunicazione udp**

   Descrizione del protocollo di comunicazione costruito e del suo funzionamento, prima dal lato del manipolatore e successivamente lato visualizzatore. Viene riportato anche un esempio di prova dell'effettivo funzionamento.

4. **Stima del peso del carico**

   Descrizione della dinamica del manipolatore e di come si giunga ad una stima del peso del carico a partire dai dati di coppia dei giunti disponibili e grazie alla pianificazione di traiettorie che eccitino sufficientemente i parametri da stimare. Eseguendo due volte la stessa traiettoria prima senza carico e poi con carico, ed eseguendo la differenza delle coppie misurate su entrambe le corse, si ottiene un segnale di coppia al netto degli effetti dovuti agli attriti.

5. **Controllo di Impedenza**

   Descrizione del significato di "Controllo di Impedenza" e analisi dei contenuti bibliografici individuati. Inizialmente si affronta la sintesi di un controllo che rimuova almeno in parte l'attrito percepito nelle movimentazioni del

manipolatore. Da questo si passa ad un controllo che aggiunga attrito viscoso e successivamente ad un controllo visco-elastico.

1.1 CONTROLLO DI FORZA

Il controllo dei robot può essere suddiviso principalmente in due categorie:

- Controllo di posizione
- Controllo di forza

Il controllo di posizione è quello più utilizzato per i robot industriali.

Come riportato in [8] ci sono due approcci interessanti per il controllo di forza:

- Controllo di impedenza
- Controllo ibrido

Il controllo di impedenza proposto da Hogan [2] ha come scopo il controllare la posizione e la forza regolando l'impedenza meccanica (definita nella sezione 5.2) dell'end-effector a forze esterne originate dal contatto con l'ambiente. Si può modificare il valore di impedenza a seconda del comportamento desiderato in una particolare direzione. Se si vuole una maggiore accuratezza si imporrà un valore di impedenza alto, al contrario se si punta ad avere un comportamento più asservente si porrà un valore basso di impedenza. Il controllo ibrido proposto da Raibert e Craig [10, 11] seleziona la direzione in cui la posizione dell'end-effector del manipolatore deve essere controllata e la direzione in cui la forza esercitata dall'end-effector

sull'ambiente deve essere controllata per una data operazione, ed esegue un controllo simultaneo delle traiettorie di posizione e forza desiderate. Il controllo ibrido però risulta in alcuni casi instabile, perché la dinamica del manipolatore non viene considerata rigorosamente. Tuttavia questo controllo fornisce un migliore inseguimento dei riferimenti di posizione e forza rispetto al controllo di impedenza.

Proprio per le migliori caratteristiche di stabilità del controllo di impedenza e di compatibilità con la presenza di operatori umani con cui possono avvenire contatti, verrà implementato quest'ultimo nel proseguo della tesi.

1.2 RISCHI PER LA SICUREZZA

È importante considerare la sicurezza nelle interazioni uomo-robot [18]. I robot possono produrre movimenti potenti e molto rapidi all'interno di un spazio operativo ampio. I rischi derivano da contatti non intenzionali tra i robot e gli operatori.

I classici robot industriali sono molto potenti e in genere svolgono il loro lavoro seguendo una movimentazione pianificata, a prescindere dalla presenza di persone nello spazio di lavoro. Questi manipolatori sono stati introdotti per sostituire i lavoratori umani che svolgevano compiti pericolosi, difficili, ripetitivi e non salutari. In passato, questi luoghi di lavoro pericolosi causavano lesioni e malattie ai lavoratori. Alcuni pericoli per la salute che colpiscono i lavoratori umani nelle aree di lavoro sono fumi tossici, calore, radiazioni, rumore, lesioni fisiche, etc...

Nei sistemi di produzione automatizzati, i robot sono impiegati per operazioni di assemblaggio, movimentazione, saldatura e rivestimento. Pertanto, i robot non solo migliorano la

sicurezza, ma anche la produttività nell'industria pesante.

Al contempo tali robot possono rappresentare un rischio per gli operatori.

Le cause di incidente con robot industriali sono principalmente tre:

- Errori di ingegneria
- Errori umani
- Scarse condizioni ambientali

Gli errori di ingegneria includono problemi alla meccanica del manipolatore come connessioni non sicure o elettronica danneggiata, oppure problemi del controllore come bug di programmazione o errori logici dell'algoritmo. Come conseguenze il robot potrebbe non fermarsi, raggiungere velocità incontrollate ed elevate. Questo genere di errori non può essere previsto.

Gli errori umani, sono più gestibili, avvengono per via di disattenzione, affaticamento, inosservanza delle procedure di sicurezza, corsi sulla sicurezza inadeguati oppure incorretta procedura di installazione del robot.

Fattori ambientali avversi includono temperature estreme e qualità bassa delle misure dei sensori.

In ambito internazionale la sicurezza delle interazioni uomo-robot vengono normate dal International Organization for Standardization (ISO). Le più recenti norme riguardanti la sicurezza per robot industriali sono riportati nella ISO 10218.

Gli incidenti vengono classificati in due tipi di lesioni:

- Lesioni dovute a urto
- Lesioni dovute a pizzicamento

L'urto corrisponde all'impatto tra braccio meccanico e operatore, mentre il pizzicamento avviene quando il robot blocca ed eventualmente schiaccia l'operatore o un suo arto.

La soluzione classica è quella di costruire una barriera fisica di sicurezza attorno al manipolatore. Queste possono essere complementate da barriere di sicurezza non fisiche come barriere luminose, tappetini di sicurezza, etc...

Determinare un comportamento reattivo adeguato quando il robot entra in contatto con l'uomo è una delle molte sfide che la robotica deve risolvere per garantire la sicurezza in ambienti dinamici abitati dall'uomo.

I sensori tattili hanno la capacità di rilevare le pressioni e la loro distribuzione. Il rilevamento dei contatti fornisce un feedback importante e diretto per il controllo, sia in caso di interazioni volontarie che involontarie con l'ambiente. Può essere applicato a un robot come una pelle artificiale e utilizzato per fermare in sicurezza i movimenti di un robot in una prospettiva centrata sull'uomo.

Vale la pena notare che posizionare i sensori tattili lungo il segmento principale del robot non porta più informazioni di quelle che verrebbero fornite dai sensori di coppia posizionati sui giunti. Avere un rilevamento tattile su parti dei segmenti del robot che si piegano o si ritraggono è più utile, in quanto consentono di rilevare se un arto umano viene agganciato tra due parti del robot.

Nel caso del Panda si sfruttano meramente le informazioni ottenute dai sensori di coppia dei giunti per individuare i contatti con l'ambiente o l'operatore.

Sono presenti inoltre dei dispositivi (figure 8, 7) aventi funzione di blocco istantaneo delle movimentazioni del manipolatore.

1.3 ROBOT COLLABORATIVI

Figura 1: Robot collaborativo.

Con robot collaborativo si intende una macchina concepita per interagire fisicamente con l'uomo in uno spazio di lavoro. La nascita dei robot collaborativi, detti anche *cobot*, e degli algoritmi di controllo di forza e interazione uomo-macchina è stata dettata dalla crescente applicazione di robot che lavorano a stretto contatto con operatori umani. Il primo cobot risale al 1996, creato da Edward Colgate e Michael Peshkin, professori alla Northwestern University, con lo scopo di rendere possibile l'interazione fisica diretta tra uomo e manipolatore controllato da un computer. Nel 2004 poi arrivò il primo robot industriale collaborativo *LBR 3* prodotto da KUKA. Questo robot leggero è stato il risultato di una lunga collaborazione con l'Istituto Aerospaziale tedesco. Oggi i cobot possono essere rivestiti di materiali adatti all'assorbimento degli urti, dotati di sensori tattili per un identificazione più capillare dei contatti e possono imparare i task dagli operatori in loco. La loro versatilità e compattezza li rende adatti a lavori a cui i classici robot industriali

non lo sono.

I cobot essendo leggeri e dotati di sensori di coppia/forza gestiscono eventuali collisioni avvenute durante le movimentazioni senza arrecare danni a cose e persone.

Tra gli importanti vantaggi è da ricordare la possibilità di installare un robot collaborativo senza gabbie di protezione. Ciò porta alla possibilità di posizionare il manipolatore in postazioni molto più contenute, o addirittura a contatto con il paziente nel caso di applicazioni riabilitative.

Possono essere utilizzati nel campo della riabilitazione ottenendo prestazioni riabilitative intense, specifiche ed ottimali per lo specifico paziente, garantendo interventi riabilitativi continuativi.

L'utilizzo di dispositivi robotici e la realtà virtuale possono essere usati per offrire un supporto riabilitativo innovativo, garantendo interventi riabilitativi che avvalorano l'intensità, la ripetitività, la significatività di un esercizio e la stimolazione multisensoriale, caratteristiche riabilitative che vengono attualmente privilegiate in accordo con le ampliate conoscenze dei meccanismi neurobiologici della plasticità cerebrale. Con l'utilizzo di dispositivi robotici la realizzazione di questi presupposti avviene mediante:

- l'esecuzione reiterata, supportata da strumenti robotici
- l'esaltazione dell'informazione sensoriale di ritorno, prodotta da sistemi in realtà virtuale

Esistono robot specifici per la riabilitazione di precisi distretti corporei (spalla, braccio, avambraccio, mano, arto inferiore) o funzioni (funzione del cammino, funzione dell'equilibrio) che possono essere utilizzati in molti ambiti riabilitativi e patolo-

gie, da quelle neurologiche sia del sistema nervoso centrale che di quello periferico, a quelle ortopediche, a casi di pazienti sottoposti ad amputazioni.

Si hanno prevalentemente due tipologie di strumenti robotici riabilitativi.

Sistemi end-effector che interagiscono con il paziente agganciandosi alla parte da riabilitare tramite l'end-effector appunto. Presenta un lento e complicato adattamento del paziente al dispositivo.

Sistemi esoscheletrici, sono dispositivi indossabili, presentano un elevato numero di gradi di libertà per permettere movimenti naturali della parte interessata. [16]

2

SETUP SPERIMENTALE

Nel seguito vengono descritte le componenti hardware e software utilizzate per lo sviluppo dell'attività di tesi.

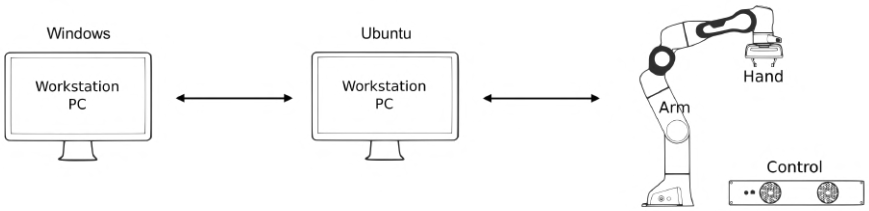

Figura 2: Panoramica setup sperimentale.

Dalla figura 2 si può avere una comprensione generale del setup utilizzato nelle prove sperimentali. Il robot collaborativo presente in laboratorio viene programmato da un computer con sistema operativo Linux Ubuntu a sua volta collegato ad una macchina con S.O. Windows per la visualizzazione in tempo reale di dati utili alla diagnostica.

Il robot e i computer sono collegati direttamente tramite cavi ethernet.

2.1 HARDWARE

Il computer con sistema operativo Windows è dotato di kernel real-time, in cui i dati vengono elaborati e formattati tramite MATLAB e Simulink.

Il robot utilizzato è un Panda della Franka Emika (fig. 3), e verrà descritto nel dettaglio nella sezione 2.1.1.

Figura 3: Robot collaborativo leggero Panda.

2.1.1 *Franka Emika's Panda*

Il Panda è un robot collaborativo leggero, con un peso di 18kg e uno spazio di lavoro di raggio 855mm come riportato in figura 4.

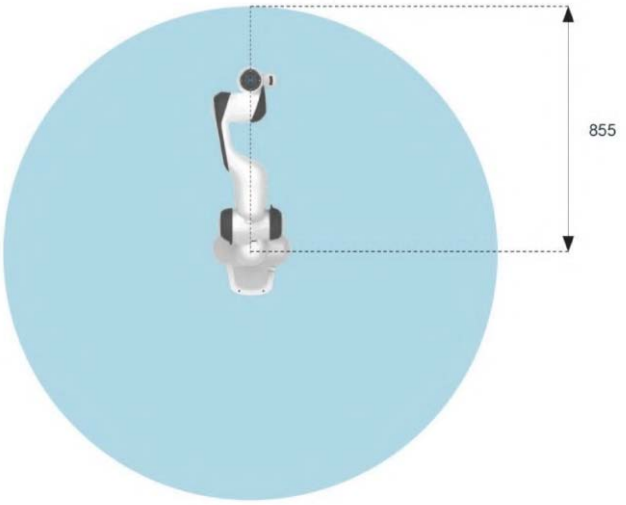

Figura 4: Spazio di lavoro visto la sopra.

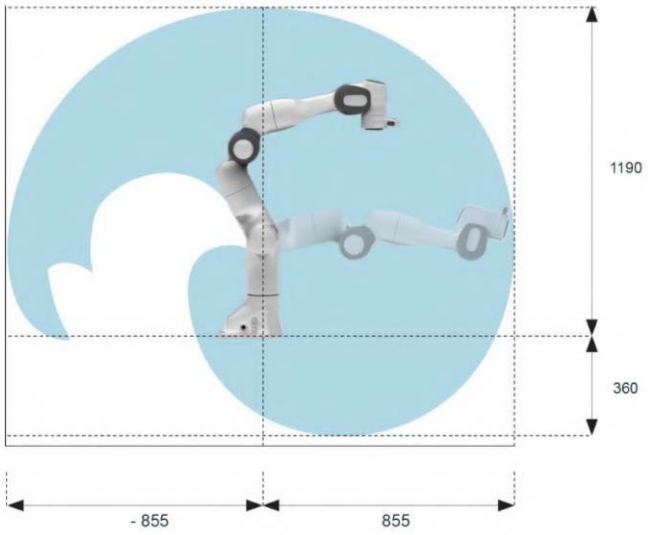

Figura 5: Spazio di lavoro visto dal lato.

È un manipolatore a 7 gradi di libertà che può lavorare con carichi fino ai 3kg. In figura 6 sono riportati i parametri di

Denavit-Hartenberg della catena cinematica del robot ed elencati in seguito nella tabella 1.

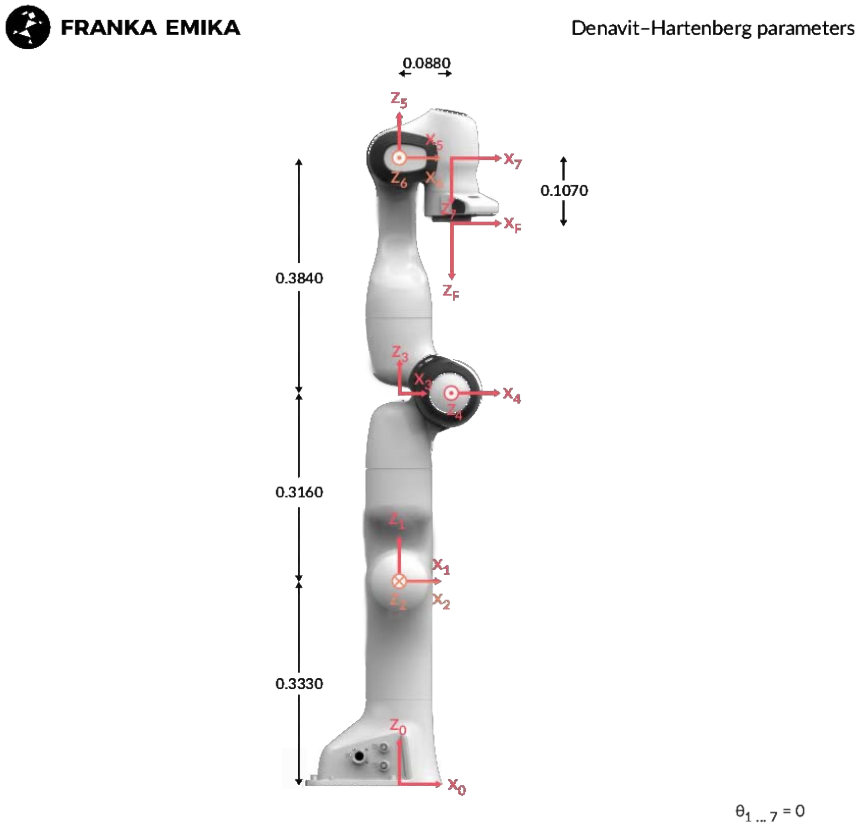

Figura 6: Parametri di Denavit-Hartenberg del cobot Panda.

Ogni giunto del Panda è dotato di freno, e questi vengono attivati allo spegnimento del robot e disattivati all'avvio tramite l'interfaccia Desk che verrà descritta nella sezione 2.2.1. Le pinze possono applicare una forza continuativa di 70N e una forza massima di 140N, con un apertura massima di 80mm e velocità di apertura/chiusura di 50mm/s.

Al robot sono collegati dei dispositivi esterni di attivazione per l'interruzione di sicurezza del moto del manipolatore.

Come riportato in figura 7 è presente un pulsante d'emergenza a fungo rosso che va premuto in caso di necessaria interruzione dei movimenti.

Giunto	a (m)	d (m)	α (rad)	θ (rad)
Giunto 1	0	0.333	0	θ_1
Giunto 2	0	0	$-\frac{\pi}{2}$	θ_2
Giunto 3	0	0.316	$\frac{\pi}{2}$	θ_3
Giunto 4	0.0825	0	$\frac{\pi}{2}$	θ_4
Giunto 5	-0.0825	0.384	$-\frac{\pi}{2}$	θ_5
Giunto 6	0	0	$\frac{\pi}{2}$	θ_6
Giunto 7	0.088	0	$\frac{\pi}{2}$	θ_7
Flangia	0	0.107	0	0

Tabella 1: Tabella di Denavit-Hartenberg

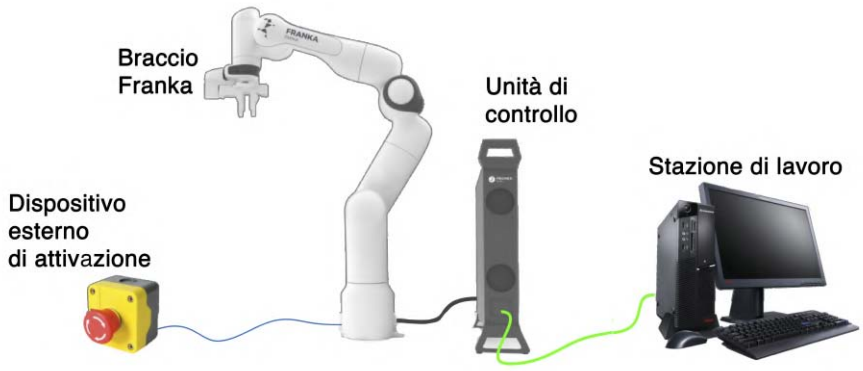

Figura 7: Collegamenti robot.

Oltre a tale dispositivo sono presenti altri due dispositivi riportati in figura 8.

(a) Dispositivo uomo-morto (b) Pulsante a fungo

Figura 8: Dispositivi di controllo movimentazioni

Un dispositivo vigilante, noto anche come dispositivo dell'uomo morto (fig. 8a), costituito da un unico pulsante che va tenuto premuto per poter eseguire movimentazioni. In caso d'emergenza se il tasto viene rilasciato o premuto con forza il moto del robot si interrompe istantaneamente.

È presente inoltre un pulsante simile al fungo rosso (fig. 8b), che va azionato con un gesto d'avvitamento per permettere l'avvio del robot e può essere premuto per interrompere ogni movimento.

I due dispositivi di figura 8 vanno utilizzati alternativamente. Il robot rimane bloccato se si prova ad utilizzarli simultaneamente.

Sul robot stesso è presente un interfaccia utente per il pilotaggio diretto (fig. 9), con la quale si possono controllare le pinze,

Figura 9: Tasti di pilotaggi locali.

muovere liberamente il braccio o eseguire altri task personalizzati.

L'unità di controllo viene collegata al robot e al computer come illustrato in figura 7, tramite un cavo proprietario per il braccio e un cavo Ethernet (TCP/IP) per il pc. Il controllore è alimentato tramite un power connector IEC 60320-C14 (V-Lock) ed ha un consumo medio di 300*W* (max. 600*W*). Il controller funziona alla frequenza di 1kHz, caratteristica che lo rende adatto a svariate applicazioni, e nel nostro caso renderà semplice l'implementazione della visualizzazione real-time dei segnali del robot.

Il robot è stato installato sopra ad un supporto metallico appositamente costruito tenendo conto delle problematiche vibrazionali, cosi che durante le movimentazioni si abbia il minimo disturbo esterno possibile. In figura 10 è riportata una fotografia del robot fissato sul supporto.

Nella tabella 2 sono riportati i limiti cinematici e dinamici nello spazio dei giunti del manipolatore.

Figura 10: Supporto del robot.

	Giunto 1	Giunto 2	Giunto 3	Giunto 4	Giunto 5	Giunto 6	Giunto 7	Unità
q_{max}	2.8973	1.7628	2.8973	-0.0698	2.8973	3.7525	2.8973	rad
q_{min}	-2.8973	-1.7628	-2.8973	-3.0718	-2.8973	-0.0175	-2.8973	rad
$\dot{q}_{max}$	2.1750	2.1750	2.1750	2.1750	2.6100	2.6100	2.6100	$\frac{rad}{s}$
$\ddot{q}_{max}$	15	7.5	10	12.5	15	20	20	$\frac{rad}{s^2}$
$\dddot{q}_{max}$	7500	3750	5000	6250	7500	10000	10000	$\frac{rad}{s^3}$
$\tau_{j_{max}}$	87	87	87	87	12	12	12	Nm
$\dot{\tau}_{j_{max}}$	1000	1000	1000	1000	1000	1000	1000	$\frac{Nm}{s}$

Tabella 2: Limiti cinematici e dinamici dei giunti

2.2 SOFTWARE

2.2.1 *Desk*

È disponibile un'interfaccia web di amministrazione e programmazione denominata *Desk*, accessibile all'indirizzo IP del controller, e con cui ci si connette al robot prima di iniziare a programmare, con la possibilità si sbloccare i freni di ogni giunto. In figura 11 è riportata la schermata principale di Desk.

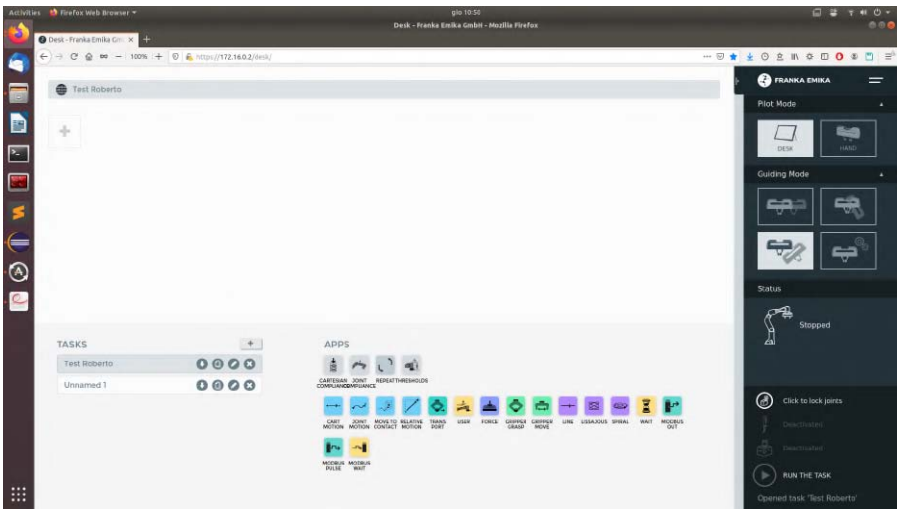

Figura 11: Interfaccia web Desk.

Da tale interfaccia si possono inoltre programmare task tramite un linguaggio di programmazione a blocchi intuitivo.

Nel caso delle attività di questa tesi, Desk è stato utilizzato solo per l'avvio e spegnimento del robot, e per la gestione delle impostazioni del manipolatore.

Prima di accendere il robot controllare sempre che:

- Il robot sia installato su di una base stabile, e che non rischi ribaltamenti durante i movimenti bruschi
- Il cavo di collegamento tra braccio e controller sia saldamente inserito da ambi i lati

- Il dispositivo di attivazione (fig. 8) sia collegato e vicino

Avere una buona performance della rete è cruciale durante il controllo del robot. Per questo motivo è raccomandato il collegamento diretto tra PC e controller, che dovranno apparire sotto la stessa rete locale. Il modo più semplice per giungere a tale scopo è utilizzando indirizzi IP statici. Qualsiasi due indirizzi che ricadano sotto la stessa rete vanno bene, ma quelli riportati in tabella 3 sono quelli utilizzati negli esperimenti di questa tesi.

	Workstation PC	Control
Indirizzo	192.168.0.1	172.16.0.2
Maschera di sottorete	24	24

Tabella 3: Indirizzi IP del robot e del controller

All'avvio del robot e all'apertura dell'interfaccia Desk viene presentata la schermata di figura 12.

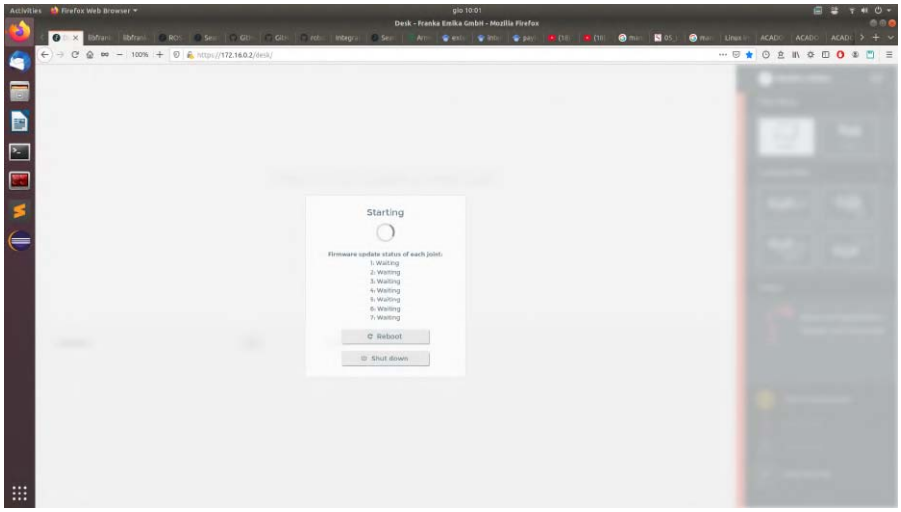

Figura 12: Collegamento al robot tramite Desk.

Tale schermata segnala l'eventuale presenza di aggiornamenti del firmware e attende l'instaurazione della connessione con i giunti.

In figura 11 si può notare la presenza di un pulsante per il blocco e sblocco dei freni dei giunti, situato nella colonna destra, sotto alla rappresentazione del robot. Quando ci si collega al robot dopo la sua accensione, la colonna di destra avrà il contorno giallo, che segnala il fatto che il robot sia frenato, come anche riportato affianco alla rappresentazione del robot, questa volta gialla. Questa situazione è riportata in figura 13.

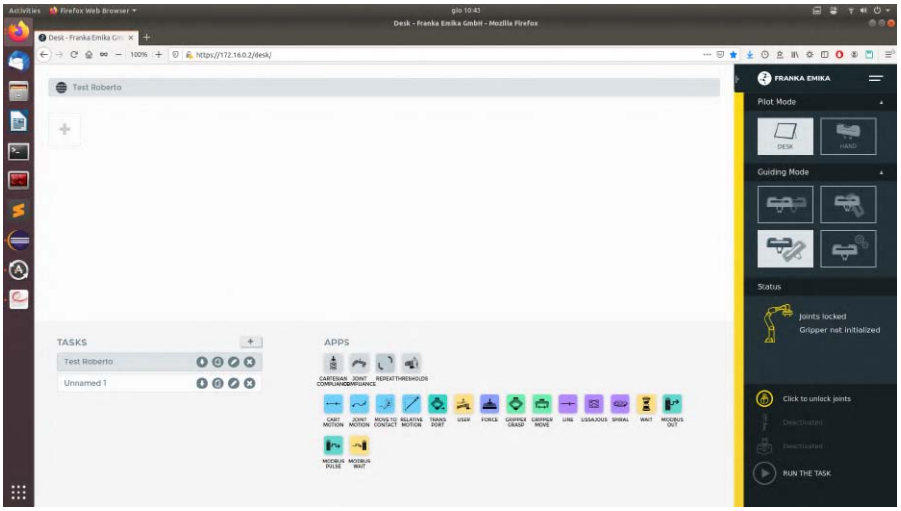

Figura 13: Interfaccia web Desk all'accensione del robot.

Nella schermata delle impostazioni di Desk, accessibile cliccando sul tasto nell'angolo in alto a destra, sono individuabili gli indirizzi IP del controllore e del robot.

Questi indirizzi sono modificabili alla schermata *NETWORK* come visibile in figura 14.

Rimanendo nella schermata *DASHBOARD*, riportata in figura 15, si ha una panoramica dello stato di collegamento al robot.

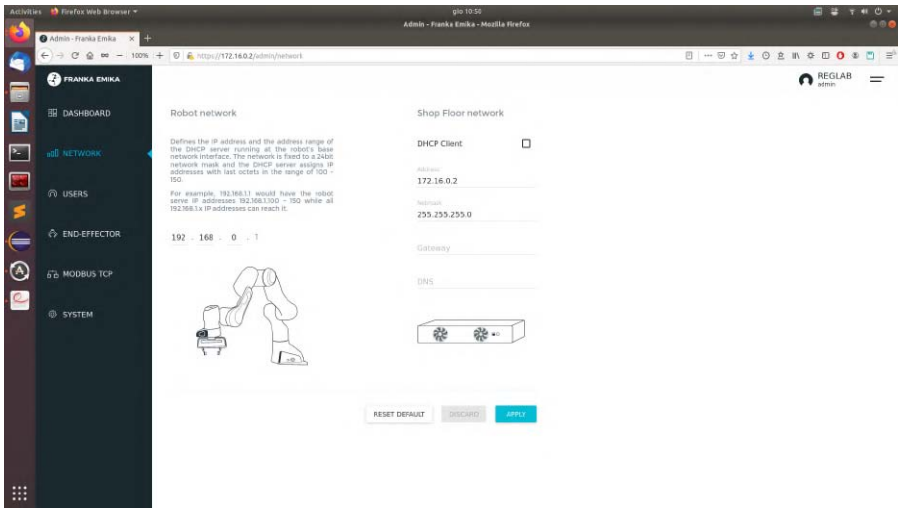

Figura 14: Impostazione degli indirizzi IP del Panda e controllore di Desk.

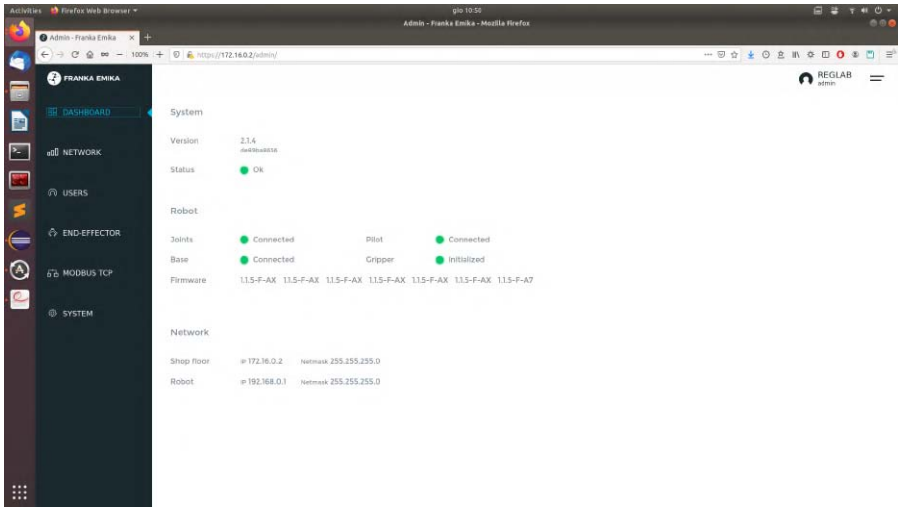

Figura 15: Impostazioni dell'interfaccia web Desk.

Nella schermata *END-EFFECTOR*, riportata in figura 16, si possono modificare i dati meccanici della pinza utilizzata (massa, posizione del centro di massa, matrice inerziale...) ed eseguirne l'inizializzazione (*homing*).

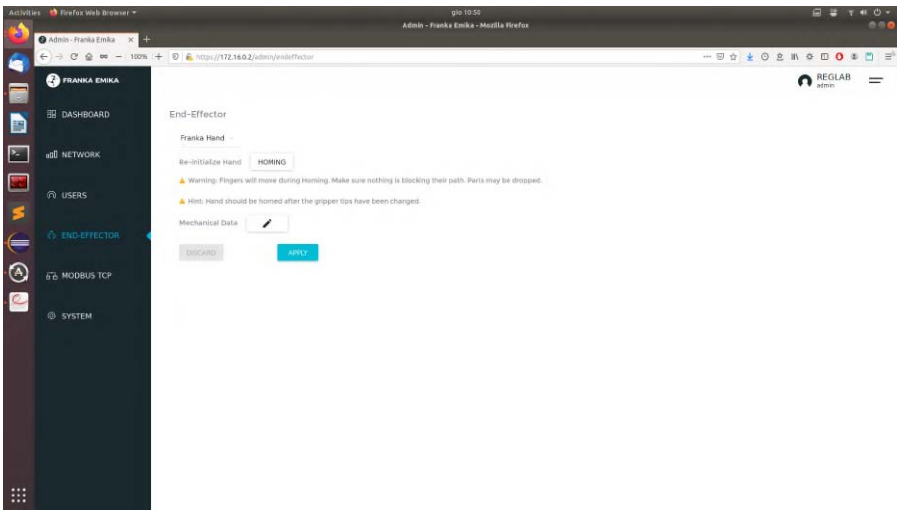

Figura 16: Impostazioni dell'interfaccia web Desk per l'end-effector.

2.2.2 *libfranka*

L'Interfaccia di Controllo Franka (FCI) permette una diretta e veloce connessione bidirezionale a basso livello con il braccio e pinza del robot. Rende disponibile lo stato attuale del manipolatore e il controllo diretto con una postazione PC esterna connessa tramite Ethernet.

Sfruttando la libreria C++ open source *libfranka* si può effettuare un controllo real-time a 1kHz tramite cinque diverse interfacce:

- Comandi di coppia con compensazione di gravità e attrito
- Comandi di posizione dei giunti
- Comandi di velocità dei giunti

- Comandi di posizione cartesiana
- Comandi di velocità cartesiana

Si ha, inoltre, accesso alle misure sempre ad 1kHz di:

- Dati dei giunti, come posizione, velocità e coppia lato link
- Stima di coppie applicate esternamente
- Informazioni su contatti e collisioni

e tramite i modelli della libreria si possono ottenere facilmente

- Cinematica diretta di tutti i giunti
- Matrice Jacobiana di tutti i giunti
- Matrice di inerzia, vettori di Coriolis, di gravità e centrifugo.

I comandi inviati al robot devono rispettare alcune condizioni raccomandate e necessarie. Le condizioni raccomandate dovrebbero essere rispettate per assicurare un funzionamento ottimale del Panda. Se le condizioni necessarie non vengono rispettate allora il moto viene interrotto.

La traiettoria finale del robot è il risultato dell'elaborazione della traiettoria definita dall'utente assicurando che le condizioni raccomandate vengano rispettate. Fintanto che le condizioni necessarie vengono rispettate, il robot proverà a seguire la traiettoria richiesta, riuscendoci solo se anche le condizioni raccomandate sono rispettate.

Ci sono quattro condizioni necessarie:

- $q_{min} < q_c < q_{max}$

- $-\dot{q}_{max} < \dot{q}_c < \dot{q}_{max}$
- $-\ddot{q}_{max} < \ddot{q}_c < \ddot{q}_{max}$
- $-\dddot{q}_{max} < \dddot{q}_c < \dddot{q}_{max}$

Le condizioni raccomandate sono:

- $-\tau_{j_{max}} < \tau_{j_d} < \tau_{j_{max}}$
- $-\dot{\tau}_{j_{max}} < \dot{\tau}_{j_d} < \dot{\tau}_{j_{max}}$

All'inizio della traiettoria devono essere rispettate anche le seguenti condizioni:

- $q = q_c$
- $\dot{q}_c = 0$
- $\ddot{q}_c = 0$

Le ultime due condizioni devono essere rispettate anche alla fine della traiettoria.

Se, ad esempio, il primo punto della traiettoria definita dall'utente è troppo diversa dalla posizione di partenza del robot ($q(t = 0) \neq q_c(t = 0)$) l'errore *start_pose_invalid* interromperà il movimento.

I valori limite sono riportati in tabella 2 della sezione 2.1.1.

2.2.3 *Eclipse*

Per la programmazione del robot si utilizza una macchina con sistema operativo Linux Ubuntu, e come ambiente di sviluppo si utilizza *Eclipse*, riportato in figura 17.

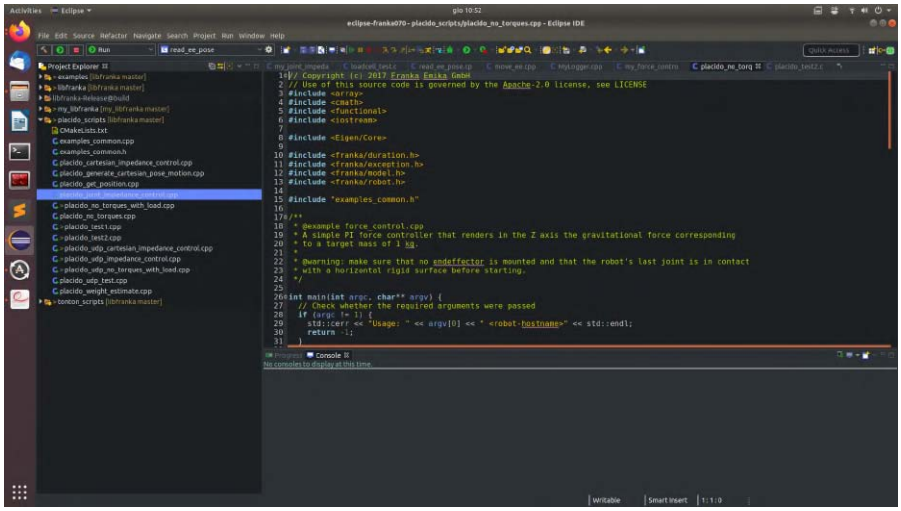

Figura 17: Schermata di eclipse.

Nella figura 18 sono state evidenziate le sezioni di interesse all'interno dell'ambiente di sviluppo.

In particolare si individuano sei componenti importanti che verranno descritte qui di seguito.

1. **Project Explorer**

   In questa sezione si trovano tutti i file dei programmi scritti per il robot. Avendo utilizzato una struttura per il progetto gestita da *MakeFile*, bisogna avere l'accortezza di aggiornare, ogniqualvolta si crei un nuovo programma, il file "CMakeLists.txt" avente la seguente struttura:

```
1 cmake_minimum_required(VERSION 3.4)
2 project(libfranka_miei CXX)
3 list(INSERT CMAKE_MODULE_PATH 0 ${
      CMAKE_CURRENT_LIST_DIR}/../cmake)
```

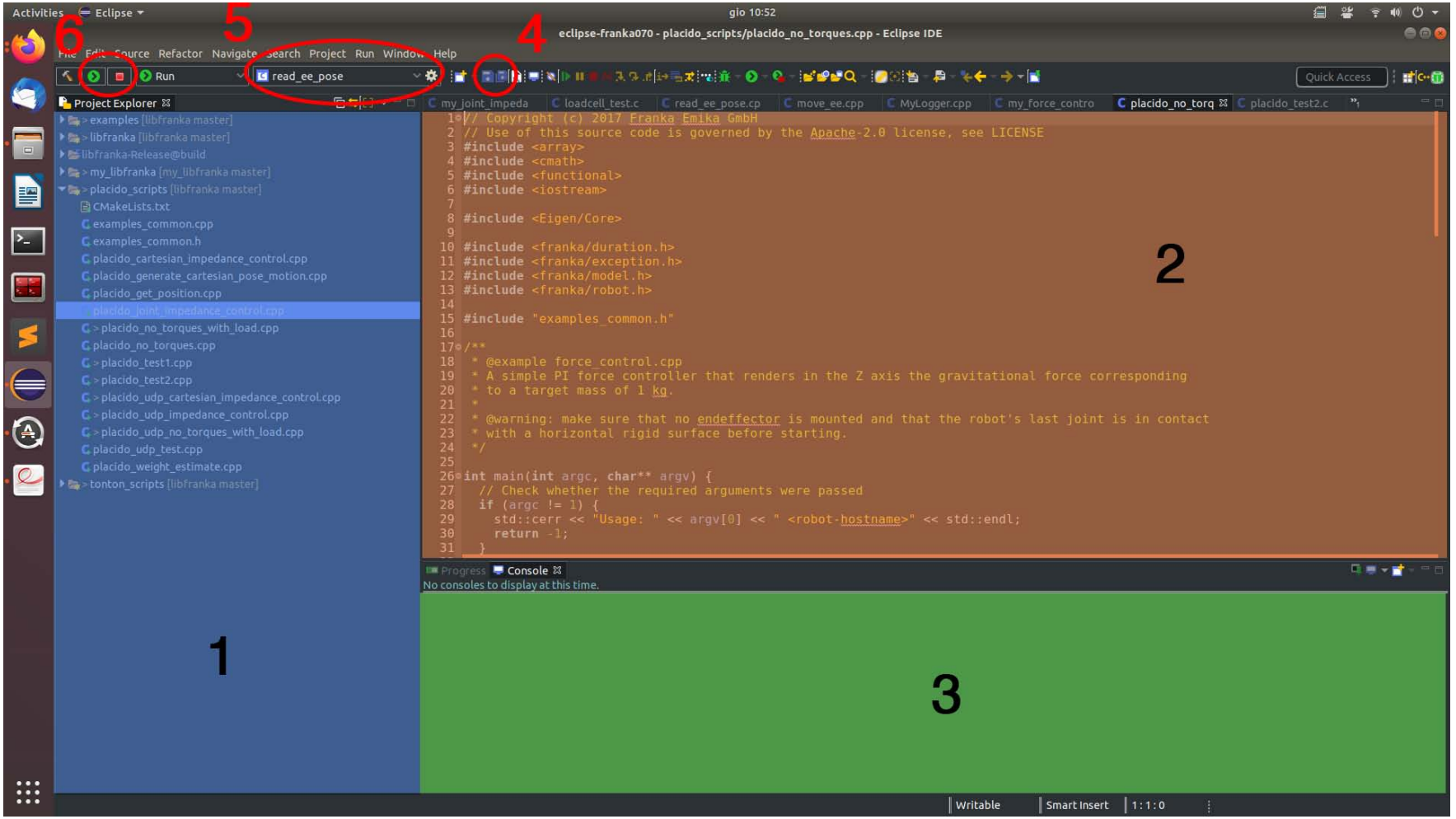

Figura 18: Struttura dell'interfaccia di eclipse

```
4 set(CMAKE_CXX_STANDARD 14)
5 set(CMAKE_CXX_STANDARD_REQUIRED ON)
6 find_package(Franka REQUIRED)
7 find_package(Eigen3 REQUIRED)
8 find_package(Poco REQUIRED COMPONENTS Foundation)
9 set(THREADS_PREFER_PTHREAD_FLAG ON)
10 find_package(Threads REQUIRED)
11
12 set(PLACIDO_SCRIPTS
13   placido_cartesian_impedance_control
14   placido_generate_cartesian_pose_motion
15   placido_joint_impedance_control
16   placido_weight_estimate
17   placido_get_position
18   placido_no_torques
19   placido_no_torques_with_load
20   placido_test1
21   placido_test2
22   placido_udp_test
23   placido_udp_no_torques_with_load
24   placido_udp_impedance_control
25   placido_udp_cartesian_impedance_control
26   placido_udp_weight
27   placido_udp_weight-gravity
28   placido_udp_final
29 )
30
31 foreach(placido_script ${PLACIDO_SCRIPTS})
32   add_executable(${placido_script} ${placido_script}.
         cpp)
33   target_link_libraries(${placido_script} Franka::
         Franka examples_common Eigen3::Eigen3)
```

```
34   target_link_libraries(${placido_script} Threads::
         Threads)
35 endforeach()
36
37 include(GNUInstallDirs)
38 install(TARGETS ${PLACIDO_SCRIPTS}
39   LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
40   RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
41 )
42
43 )
```

da riga 13 a riga 28 sono elencati i programmi scritti di cui si vuole procedere con la compilazione. Se si scrive un nuovo programma, questo dovrà essere aggiunto in questa lista, all'interno delle parentesi tonde.

Inoltre bisogna specificare se il proprio programma necessità di librerie particolari. Infatti a riga 34 viene segnalata la necessità di includere la libreria *Threads* in tutti i programmi. Questo perché si sfruttano le potenzialità dei thread per non rallentare il ciclo di controllo con operazioni di visualizzazione e comunicazione.

2. **Program Editor**

   In questo settore dell'IDE si scrive il programma vero e proprio. Si ha la possibilità quindi di usufruire di tutti i suggerimenti offerti dall'ambiente di sviluppo per l'autocompletamento della scrittura e l'evidenziazione di warnings ed errori.

3. **Terminale**

   Nel terminale si visualizza lo stato di compilazione ma anche l'eventuale scambio dati con il robot sia in input che output.

   Quindi si possono visualizzare valori di diagnosi o debug, come anche inserire valori per variare il controllo in tempo reale.

4. **Pulsanti di salvataggio**

   Il tasto più a sinistra salva il file aperto attualmente nel Program Editor, mentre quello di destra salva tutti i file del progetto.

5. **Selettore programma**

   Dal selettore bisogna individuare il programma che si ha intenzione di eseguire sul robot. Se il proprio programma non appare tra quelli nella tendina di selezione, bisogna cliccare con il tasto destro del mouse sul file eseguibile nel Project Explorer, e selezionare "Run As > Local C/C++ Application" per l'esecuzione sul robot.

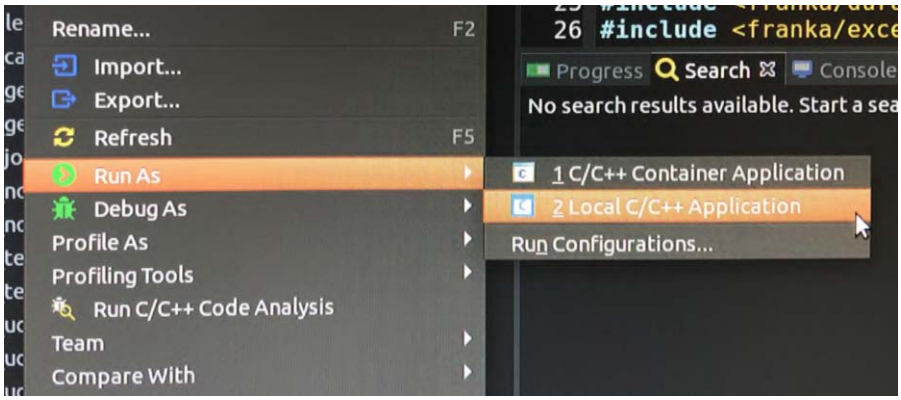

Figura 19: Esecuzione del programma sul robot.

6. **Pulsanti di gestione dell'esecuzione**

   Questi due tasti permettono di avviare o interrompere l'esecuzione del programma sul robot.

Per la compilazione del progetto bisogna spostarsi nel Project Explorer alla posizione "*nomeProgetto*-Release@Build", cliccare su "Build Targets"e successivamente su "all". Questa posizione è riportata nel Project Explorer di figura 20.

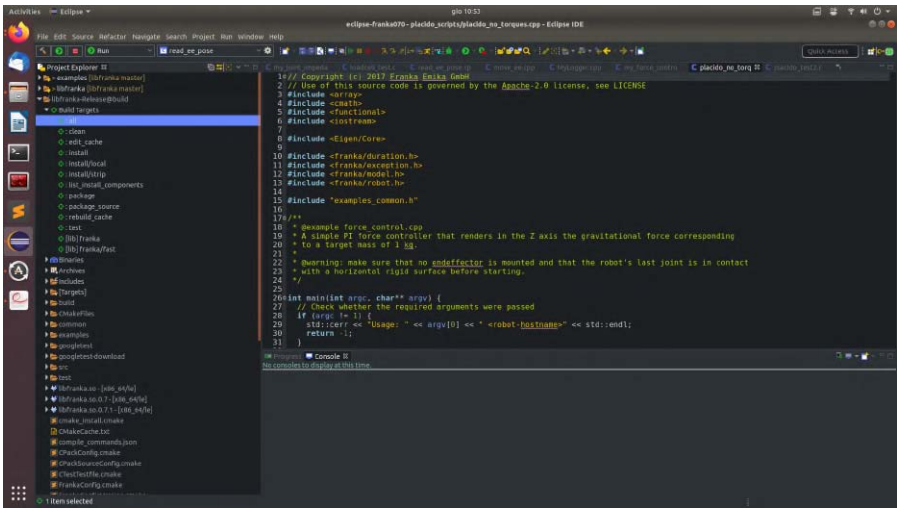

Figura 20: Compilazione del progetto su Eclipse.

Dopo una breve attesa, e in caso di compilazioni eseguite con successo e senza errori, si troveranno gli eseguibili nella directory Binaries.

Se venissero, invece, riscontrati errori nella compilazione, questi verrebbero elencati nel terminale.

2.2.4 *MATLAB e Simulink*

Sul computer con sistema operativo Windows è installato *MATLAB* e *Simulink*. In particolare si utilizza il modulo real-time di *Simulink* per la ricezione di pacchetti UDP a 1kHz. Ciò è per-

messo dal fatto che la macchina è dotata di kernel real-time, avendo quindi capacità di gestione di segnali avanzate.

La comunicazione con protocollo UDP avviene quindi tra l'IDE *Eclipse*, su PC con Ubuntu, e *Simulink*, su PC con Windows.

In figura 21 è riportato lo schema a blocchi utilizzato per la comunicazione UDP e visualizzazione real-time dei segnali del robot.

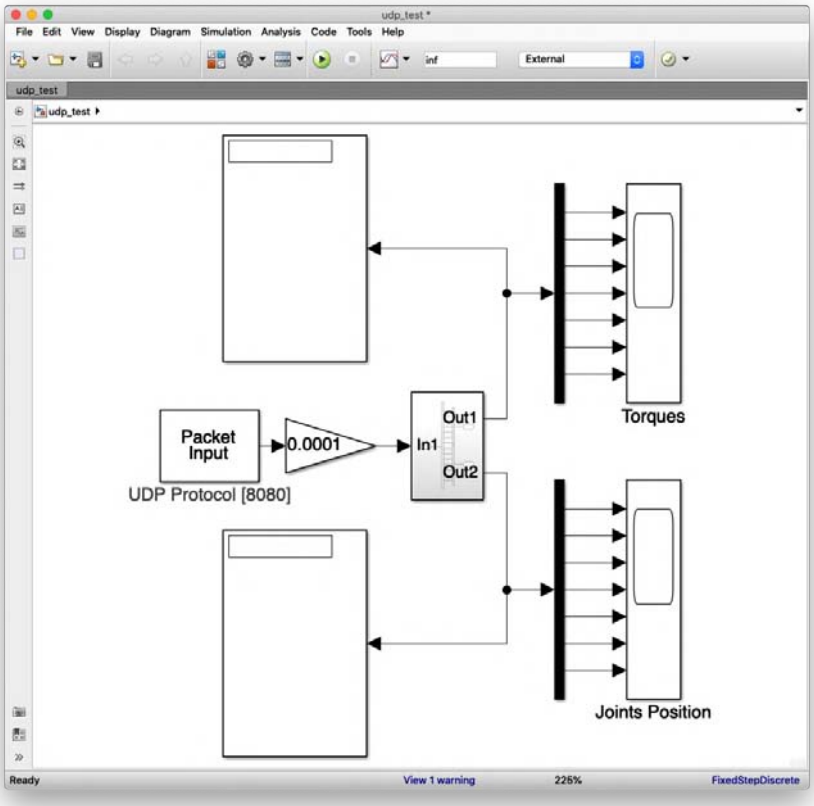

Figura 21: Schermata di Simulink con schema per la comunicazione UDP.

Simulink Desktop Real Time permette di eseguire modelli Simulink di sistemi di controllo e algoritmi di signal processing in real-time sul computer per una veloce prototipazione oppure per progetti con hardware-in-the-loop. Dallo schema Simulink si ha poi la possibilità di esportare il codice C, compilarlo e

eseguirlo in real-time su un computer Windows mentre si interfaccia con l'hardware tramite schede I/O. Il pacchetto Simulink Desktop Real-Time include drivers per il supporto di svariate schede di acquisizione I/O, abilitando l'interfaccia con sensori, attuatori e altri dispositivi per la sperimentazione, sviluppo e test di sistemi real-time.

Innanzitutto, da figura 21, si può notare come l'esecuzione dello schema a blocchi sia impostato su "External", in quanto si vuole usufruire delle capacità real-time del kernel. Andando nel menu a tendina "Simulation>Configuration" si devono impostare alcuni valori importanti per gli esperimenti. In figura 22 è visibile la finestra di impostazioni.

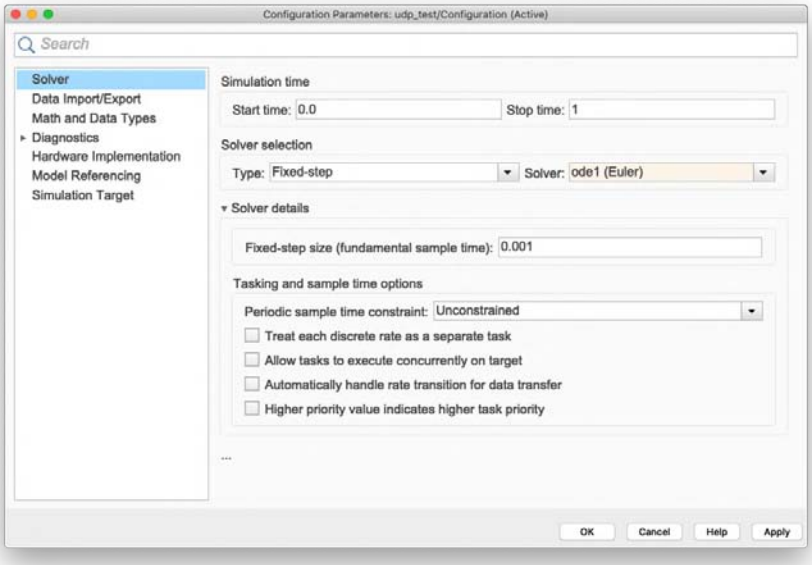

Figura 22: Finestra di impostazioni della simulazione Simulink.

Cliccando sul pannello "Solver", nella lista di selezione di sinistra, si accede alle impostazioni del risolutore utilizzato nelle simulazioni. Nel tipo di risolutore bisogna selezionare "Fixed-step", ovvero a passo di campionamento fisso, e come risolutore selezionare "Euler". Come passo di campionamento infine digi-

tare "0.001", in quanto la frequenza del ciclo di controllo del robot Panda è di 1kHz.

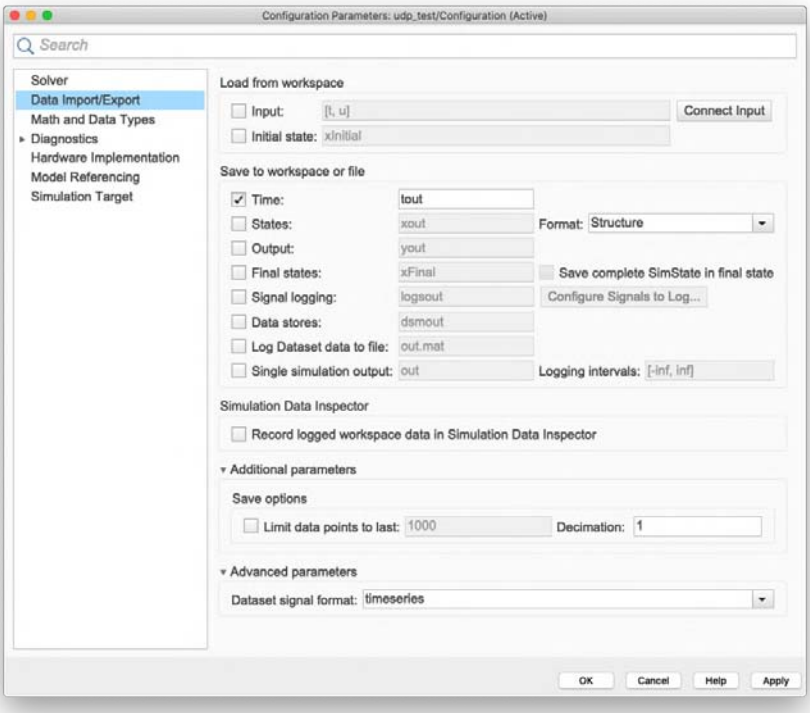

Figura 23: Finestra di impostazioni per la gestione dei dati della simulazione Simulink.

In figura 23 si può vedere la pagina accessibile cliccando su "Data Import/Export". Sotto le opzioni "Save Options" togliere la spunta su "Limit data points to last" così da esportare tutti i dati della simulazione, e non solo un numero limitato di essi. Inoltre, per comodità, si è selezionato tra i dati da salvare nel workspace di MATLAB il vettore del tempo, così da poter tracciare subito grafici a scopi diagnostici.

Su "Target Generation" (fig. 24) cliccare su "Browse..." e selezionare "sldrt.tlc" relativo a Simulink Desktop Real Time.

All'interno della finestra del modello Simulink andare nel nel menu "Tools>External Mode Control Panel".

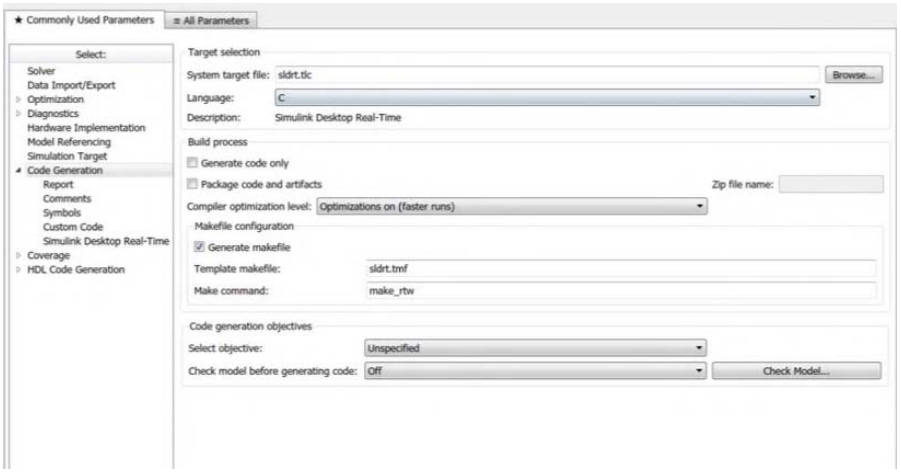

Figura 24: Finestra di impostazioni del target della simulazione Simulink.

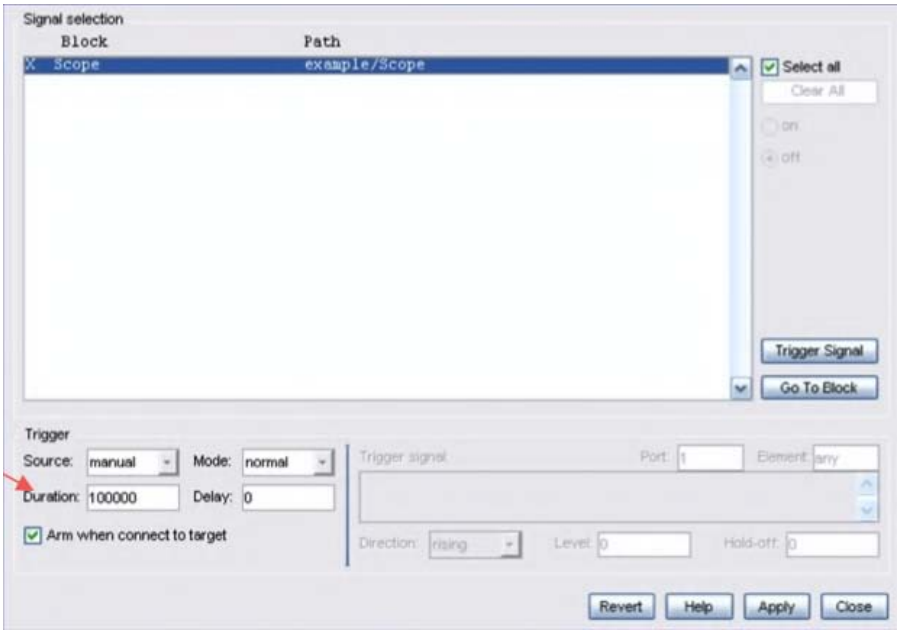

Figura 25: Finestra di impostazioni della gestione dei segnali Simulink.

Dalla finestra che appare cliccare su "Signal & Triggering". Si giungerà alla finestra di figura 25 in cui bisogna modificare la durata del Trigger, portandolo ad un valore alto, ad esempio 100000, che permette 100 secondi di simulazione visualizzabili sugli Scope prima di un reset.

Ora Simulink è pronto per le simulazioni. Basterà avviare l'esecuzione dello schema a blocchi e successivamente avviare il programma del robot per ricevere i segnali dei sensori.

2.2.4.1 *Robotics Toolbox*

In Simulink si è installata la Toolbox per la robotica creata da Peter Corke, professore alla Queensland University of Technology, direttore del QUT Centre for Robotics e direttore del ARC Centre of Excellence for Robotic Vision.

Si è deciso di utilizzare questo strumento così da potersi concentrare nella progettazione degli algoritmi complessi, e non dover calcolare manualmente matrici Jacobiane e vettori delle coppie dovute alla gravità.

Questa Toolbox MATLAB ha una ricca collezione di funzioni utili per lo studio e la simulazione di robot: manipolatori e robot mobili. Per i manipolatori, le funzioni includono cinematica, generazione della traiettoria, dinamica e controllo. Per i robot mobili, le funzioni includono la pianificazione del percorso, la pianificazione cinematica e dinamica, la localizzazione, la creazione di mappe e la localizzazione e mappatura simultanee (SLAM).

Include modelli Simulink per descrivere l'evoluzione dello stato di bracci robotici o di robot mobili nel tempo per una serie di strategie di controllo classiche. La casella degli strumenti fornisce anche funzioni per la manipolazione e la conversione tra

tipi di dati come vettori, matrici di rotazione, unità-quaternioni, quaternioni, trasformazioni omogenee e torsioni necessarie per rappresentare la posizione e l'orientamento in 2 e 3 dimensioni.

Si è sfruttata questa Toolbox per la possibilità di progettare la stima della massa del payload offline, sfruttando i segnali raccolti precedentemente in laboratorio.

Come si può vedere da figura 26 si ha la possibilità di visualizzare la movimentazione eseguita con il robot. Questa può essere una rappresentazione statica di un dato istante ma può anche essere un'animazione di tutta la movimentazione effettuata.

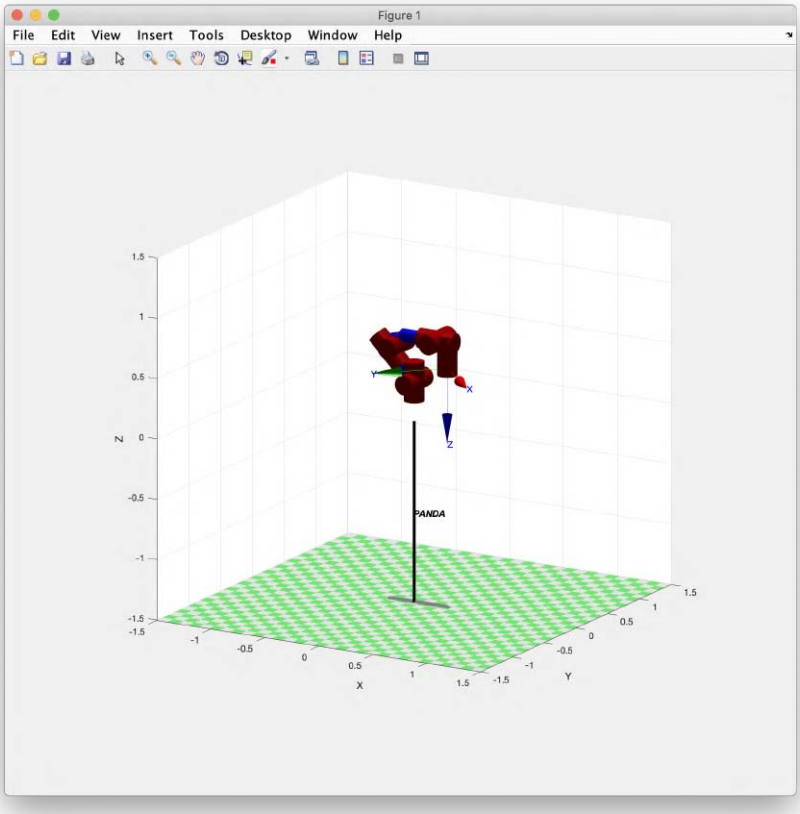

Figura 26: Visualizzazione della configurazione del braccio robotico del Panda, tramite Robotics Toolbox su Simulink.

Come prima cosa bisogna definire la struttura del robot con

cui si sta lavorando. Questo può avvenire fornendo i parametri di Denavit-Hartenberg oppure scegliendo tra i robot più comuni presenti in una libreria.

Il Panda è presente nella libreria di Robotics Toolbox e per caricarne il modello basta utilizzare il comando:

```
mdl_panda;
```

Di seguito vengono riportate a titolo di esempio il calcolo delle forze risultanti sull'end-effector partendo dai dati di coppia dei giunti e la successiva visualizzazione dell'animazione della movimentazione eseguita.

```
%% Creazione vettori di posizione e coppie
q=zeros(length(tout),7);
for i=1:7
   q(:,i)=JointsPosition.signals(i).values;
end

c=zeros(length(tout),7);
for i=1:7
   c(:,i)=Torques.signals(i).values;
end

%% Calcolo delle forze risultanti sull'end-effector
for i=1:length(tout0)
   J=panda.jacob0(q(i,:));
   f(i,:)=(J'*inv(J*J'))'*(c(i,:)');
   if(mod(i,100)==0)
      i
   end
end

%% Animazione movimento robot
```

```
22 panda.plot(q(1,:));
23 for i=1000:100:length(tout)
24    panda.animate(q(i,:));
25 end
```

3

VISUALIZZAZIONE IN TEMPO REALE DEI DATI DEI SENSORI ROBOT IN COMMUNICAZIONE UDP

3.1 INTRODUZIONE

A scopo propedeutico si è sviluppata la visualizzazione in tempo reale su *Simulink* dei dati ricevuti tramite pacchetti UDP dal robot.

L'obiettivo è quello di adattare in un momento successivo questa strategia per la visualizzazione di informazioni utili in realtà aumentata, grazie a dispositivi quali *Hololens* di *Microsoft*, così da avere l'immediata conoscenza dei dati importanti durante l'utilizzo del manipolatore. L'operatore vedrebbe così informazioni di coppia, velocità o qualsiasi altro dato voluto, in prossimità dell'end effector sovrapposti alla realtà, così da poter procedere nelle lavorazioni senza distogliere l'attenzione dal robot.

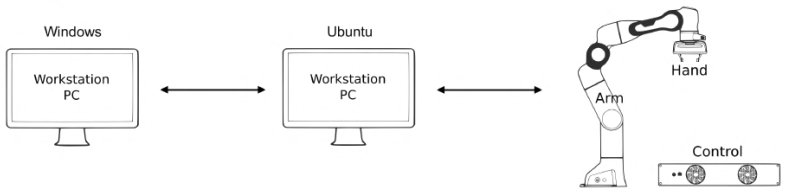

Figura 2: Panoramica setup sperimentale (immagine di pagina 11)

I due computer interessati in questo processo, riportati in

figura 2, sono direttamente connessi tra di loro tramite cavo ethernet, così da permettere una buona velocità delle comunicazioni. Si è deciso di utilizzare un protocollo di interfaccia semplice e preesistente, proprio per la natura di prototipazione di questa fase, optando quindi per lo scambio di pacchetti UDP attraverso socket.

Per instaurare lo scambio di pacchetti tra robot e visualizzatore è necessario conoscere gli indirizzi IP di entrambe le macchine. Successivamente al collegamento degli elaboratori, per l'identificazione dell'indirizzo del computer Linux è sufficiente aprire il terminale e utilizzare il comando *ifconfig*.

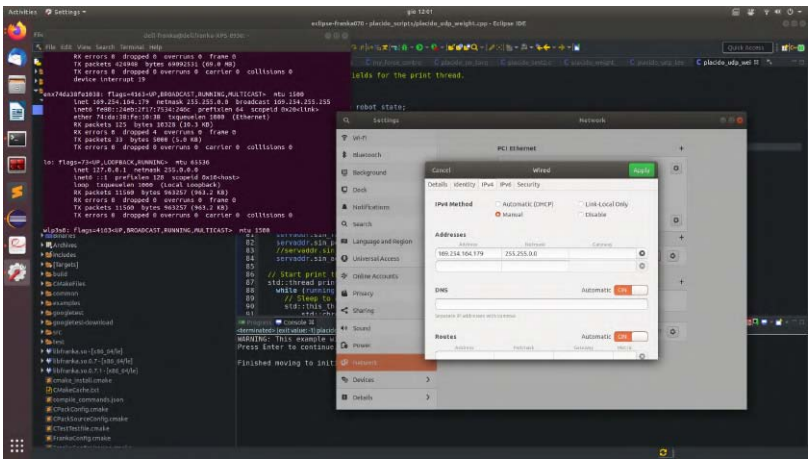

Figura 28: Identificazione dell'indirizzo IP nel collegamento Ubuntu-Windows della macchina Ubuntu.

In figura 28 è riportata la schermata del terminale in seguito all'utilizzo del comando *ifconfig*, e la finestra di configurazione dell'interfaccia di rete considerata, ovvero quella del collegamento Ubuntu-Windows. Da queste se ne ricava che l'indirizzo della macchina è 169.254.164.179 con una subnet mask 255.255.0.0. Ritroveremo questi dati nella sezione 3.3. Su Windows, recandosi sempre su un terminale, il comando da digitare è *ipconfig*.

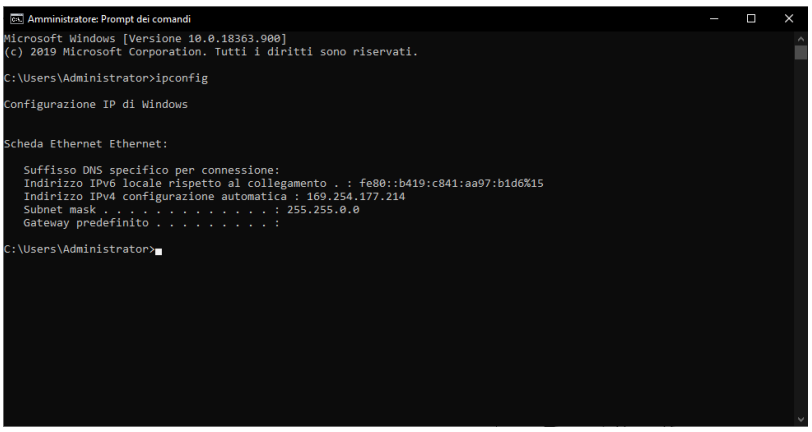

Figura 29: Identificazione dell'indirizzo IP nel collegamento Ubuntu-Windows della macchina Windows.

In figura 29 è riportata la finestra del terminale in seguito all'utilizzo del comando *ipconfig*. Si può notare che l'indirizzo IP è 169.254.177.214. Questo indirizzo ci servirà nella sezione 3.2

3.2 COMUNICAZIONE - LATO ROBOT

Utilizzando la libreria di C++ *"sys/socket.h"*, si instaura la connessione con il seguente codice:

```
1   // Creating socket file descriptor
2   if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
3      perror("socket creation failed");
4      exit(EXIT_FAILURE);
5   }
6
7   memset(&servaddr, 0, sizeof(servaddr));
8
9   // Filling server information
10  servaddr.sin_family = AF_INET;
11  servaddr.sin_port = htons(8080);
```

```
12    servaddr.sin_addr.s_addr = inet_addr("169.254.177.214");
```

in cui l'indirizzo 169.254.177.214, utilizzato alla riga 12, è quello della macchina Windows con installato *Simulink*, identificato in figura 29. Mentre a riga 11 viene definita la porta utilizzata nello scambio dei pacchetti UDP, in questo caso si è utilizzata quella standard, ovvero 8080.

L'invio di un pacchetto UDP è svolto dalla seguente linea di codice:

```
1    sendto(sockfd, &data, sizeof(data),MSG_CONFIRM, (const
        struct sockaddr *) &servaddr,sizeof(servaddr));
```

Come secondo parametro della funzione *sendto()* viene passata la variabile *data*, che si vuole inviare, per indirizzo.

Per implementare la comunicazione dati senza causare rallentamenti o interruzioni nel ciclo di controllo del robot si è creato un *thread* incaricato all'invio e ricezione di informazioni tramite socket. Inoltre per lo scambio di dati tra i thread si utilizza un *mutex*. Per questo sono state importate anche le librerie *thread* e *mutex*.

```
1 #include <mutex>
2 #include <thread>
3 ...
4 #define SIZE 32*(7*2)
5 // 32bits for each variable * 7 variables of joints * 2
      vectors (torques and positions)
6 ...
7 // Start print thread.
8 std::thread print_thread([print_rate, &print_data, &running,
       &sockfd, &servaddr]() {
9    while (running) {
```

```
10        // Sleep to achieve the desired print rate.
11        std::this_thread::sleep_for(
12        std::chrono::milliseconds(static_cast<int>((1.0 /
              print_rate * 1000.0))));
13
14        // Try to lock data to avoid read write collisions.
15        if (print_data.mutex.try_lock()) {
16           if (print_data.has_data) {
17              // Format data for the comunication
18              // 32bits mask to cut variables
19              std::bitset<SIZE> cut (std::string("
                    11111111111111111111111111111111"));
20              std::bitset<SIZE> data;
21              int n = 0;
22              for(int i=0;i<7;i++){
23                 std::bitset<SIZE> a = print_data.robot_state.
                       tau_J[i] * 10000;
24                       a = a & cut;
25                       a <<= n;
26                       n += 32;
27                       data = data | a;
28              }
29              for(int i=0;i<7;i++){
30                 std::bitset<SIZE> a = print_data.robot_state.
                       q[i] * 10000;
31              a = a & cut;
32              a <<= n;
33              n += 32;
34              data = data | a;
35              }
36
37              // Sending data with socket
```

```
38              sendto(sockfd, &data, sizeof(data),MSG_CONFIRM,
                    (const struct sockaddr *) &servaddr,sizeof(
                    servaddr));
39
40              // Clearing flag of mutex
41              print_data.has_data = false;
42          }
43          print_data.mutex.unlock();
44      }
45    }
46 });
```

Per la formattazione dei dati si è deciso di utilizzare una strategia a virgola fissa, conservando 4 cifre decimali, utilizzando quindi un fattore di scala 10000.

Nel codice di esempio riportato qui sopra, i dati di coppia di ogni singolo giunto vengono affiancati in una successione di bit e inviati come unico pacchetto attraverso il socket. Ad ogni singolo giunto sono dedicati 32 bit. Si osservi che vengono tenuti solo 32bit per ogni variabile, e i bit più significativi oltre al 32-esimo vengono semplicemente scartati. Ciò assicura una corretta riconversione in complemento a 2 in numeri decimali nel caso di numeri negativi. Se si volesse modificare la struttura del pacchetto, cambiando quindi il numero di variabili inviate, basterebbe modificare prima di tutto riga 4 adeguatamente, ovvero tenendo conto del numero di bit totali del pacchetto che si andrà a creare. Si può notare che riga 4 è stata strutturata in modo da facilitare ogni modifica. Si trova una spiegazione nel commento, su come sia strutturata, alla riga successiva. Dopodiché va aggiunto oppure tolto un ciclo for uguale a quelli di

righe 22 e 29, che ha lo scopo di inserire le variabili nella posizione corretta all'interno del pacchetto e con una formattazione adeguata.

3.3 COMUNICAZIONE - LATO VISUALIZZATORE

La visualizzazione dei dati ricevuti dal robot si è realizzata in ambiente *Simulink*, tramite il blocco *Packet Input* della libreria *Simulink Desktop Real-Time*, la cui finestra di configurazione è riportata in figura 31. In figura 30 è riportato uno schema di esempio per la gestione della ricezione e visualizzazione dei pacchetti contenenti le sole coppie dei sette giunti.

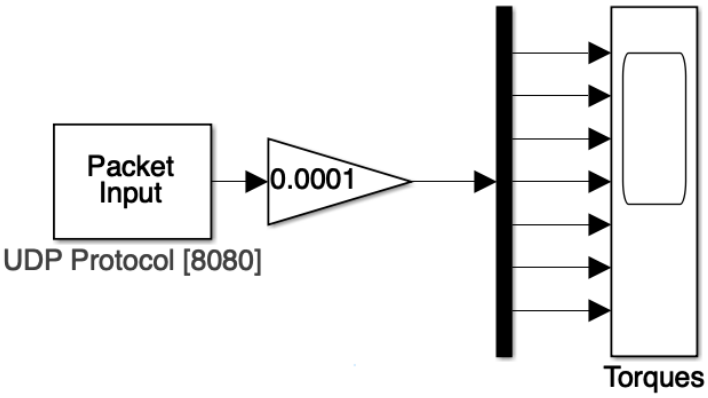

Figura 30: Ricezione e scalatura dei dati.

I campi importanti sono *Block output data types* e *Input packet size*, in cui si specifica rispettivamente il tipo e quantità di variabili ricevute e numero di byte in ingresso.

Inoltre, cliccando su "Board setup", si giunge alla finestra di figura 32 in cui è riportato l'indirizzo 169.254.177.179, che è quello della macchina Ubuntu che avevamo individuato in figura 28.

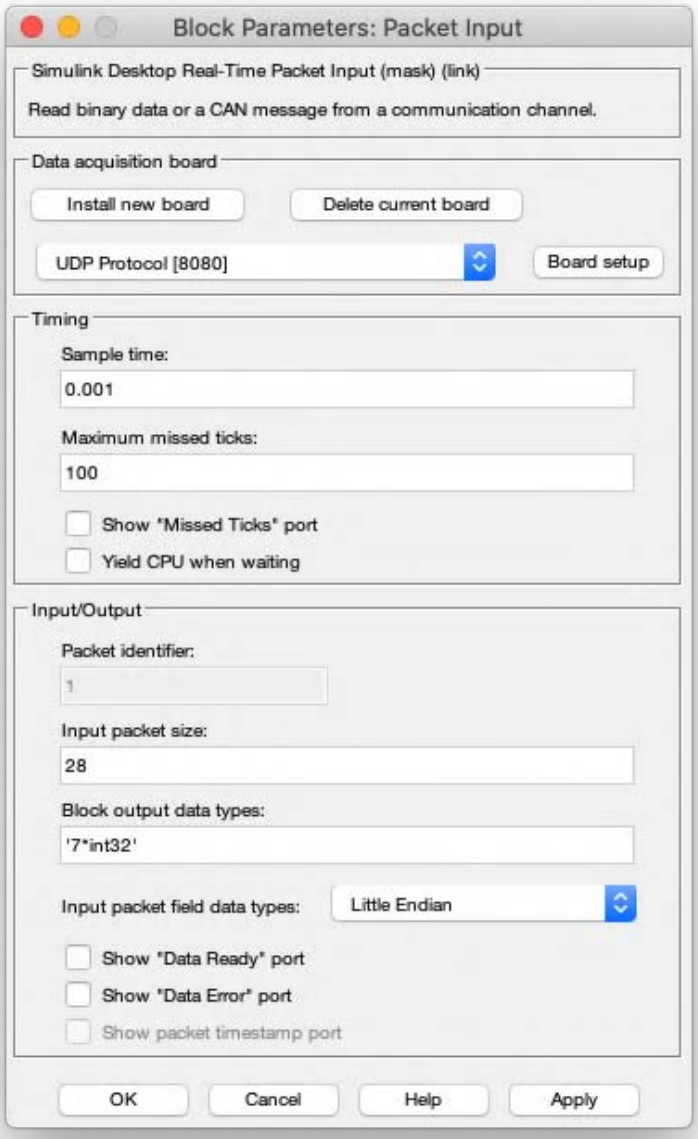

Figura 31: Finestra di configurazione del blocco *Packet Input*.

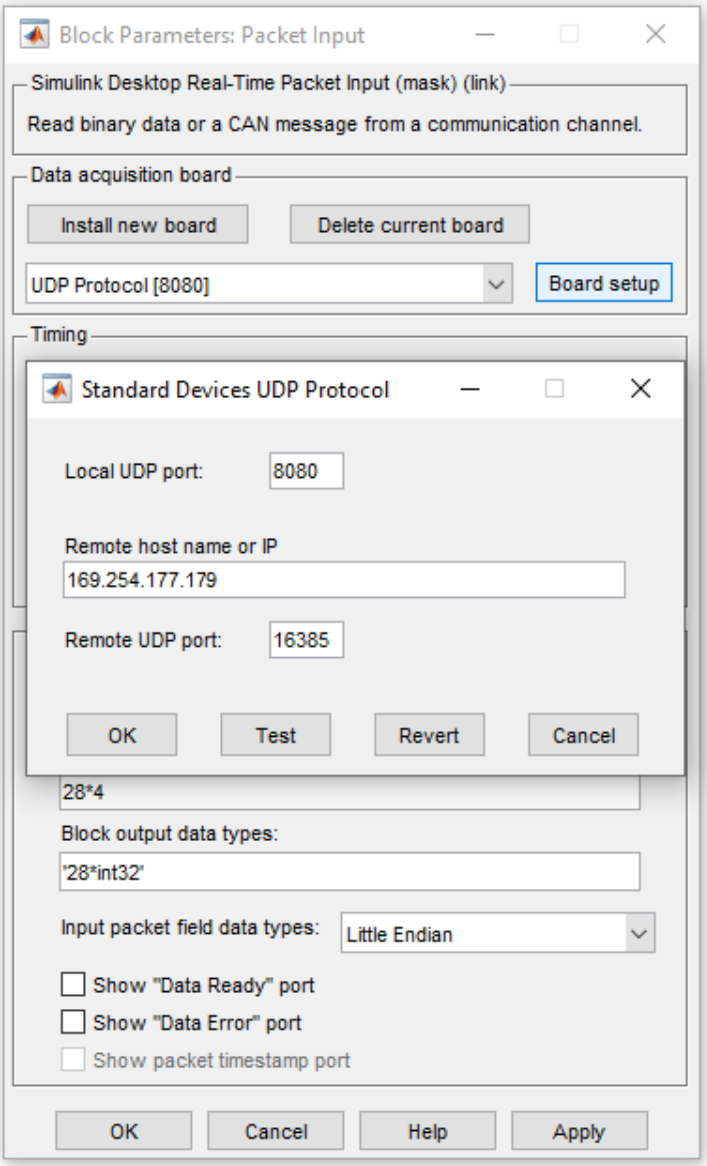

Figura 32: Impostazione dell'indirizzo per la comunicazione. *Packet Input*.

A differenza di quanto fatto nella sezione 3.2, ora la dimensione del pacchetto non va più definita in numero di bit bensì in numero di Byte. Guardando la figura 31 si può notare che, essendo il pacchetto formato solo da 7 coppie da 32 bit, si hanno 28 Byte.

$$7 \cdot 32\text{bit} = 224\text{bit}$$

$$7 \cdot \frac{32\text{bit}}{8\frac{\text{bit}}{\text{Byte}}} = 7 \cdot 4\text{Byte} = 28\text{Byte}$$

Viene definita anche la frequenza di campionamento di 1kHz, in corrispondenza con la frequenza di controllo del robot Panda, e un numero massimo di cicli di timer di ritardo del kernel real-time nella ricezione dei pacchetti, che se superato porta all'interruzione dell'esperimento.

In figura 30 è visibile l'operazione di scalatura, eseguita dal blocco a forma di triangolo che ha la funzione di moltiplicatore, dei numeri in virgola fissa ricevuti dal robot. Successivamente il pacchetto viene scomposto nelle 7 componenti relative, per quanto riguarda l'esempio riportato, alle coppie in ogni giunto, e visualizzato in forma di grafico tramite il blocco *scope*.

Nella prossima sezione (3.4), viene riportato un esempio di come risulti la visualizzazione dei dati su Simulink. Tali dati vengono esportati da Simulink a MATLAB, dove si possono successivamente salvare ed elaborare offline per fini diagnostici oppure per la progettazione di nuovi algoritmi.

3.4 VERIFICA DEL FUNZIONAMENTO

Per verificare il funzionamento della comunicazione tra robot e visualizzatore si esegue una semplice movimentazione a velocità costante del solo giunto 1, mantenendo tutti gli altri giunti fermi. Si inviano sia i dati di coppia che di posizione dei singoli giunti tramite pacchetti UDP a Simulink. Lo schema utilizzato per la ricezione è quello di figura 21. In figura 34 sono riportati gli andamenti di tali grandezze.

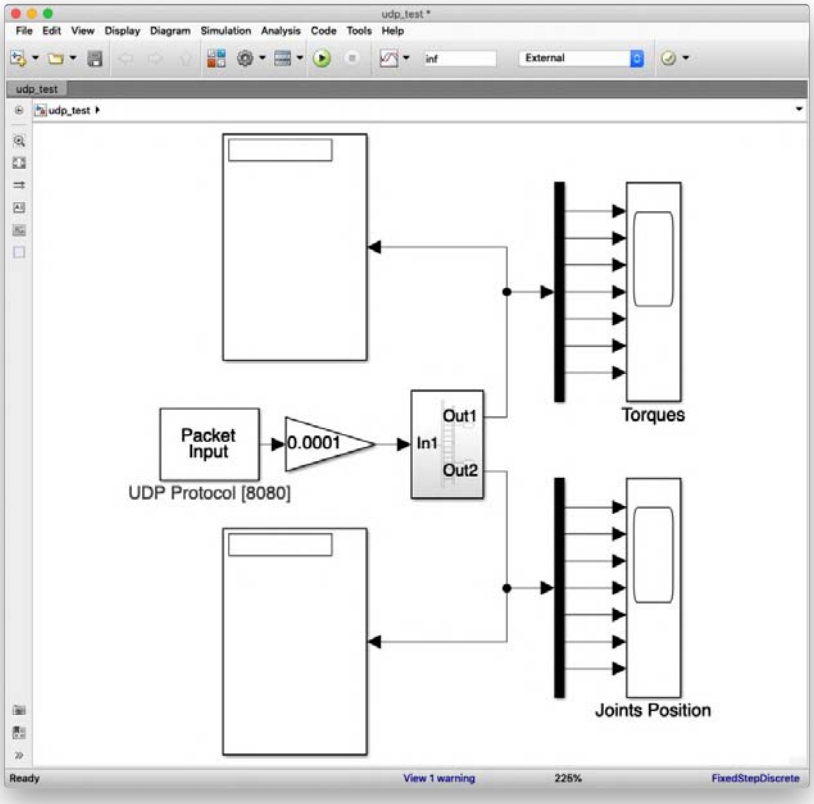

Figura 21: Schermata di Simulink con schema per la comunicazione UDP (immagine di pagina 33)

I pacchetti inviati sono quindi di dimensioni doppie rispetto a quanto visto finora, perché vengono inviati due set di informazioni: coppie e posizioni. Da lato robot si continua con l'af-

fiancamento delle variabili ottenendo un unico blocco di 384 bit. Su Simulink si ricevono in ingresso 56 byte, invece che 28, ottenendo come aspettato 14 variabili *int*. (Fig 31)

Queste variabili int poi vanno riportate a numeri decimali con virgola dividendo per il fattore di scala 10000.

Si può notare dai grafici di figura 34 che solo il primo giunto si muove.

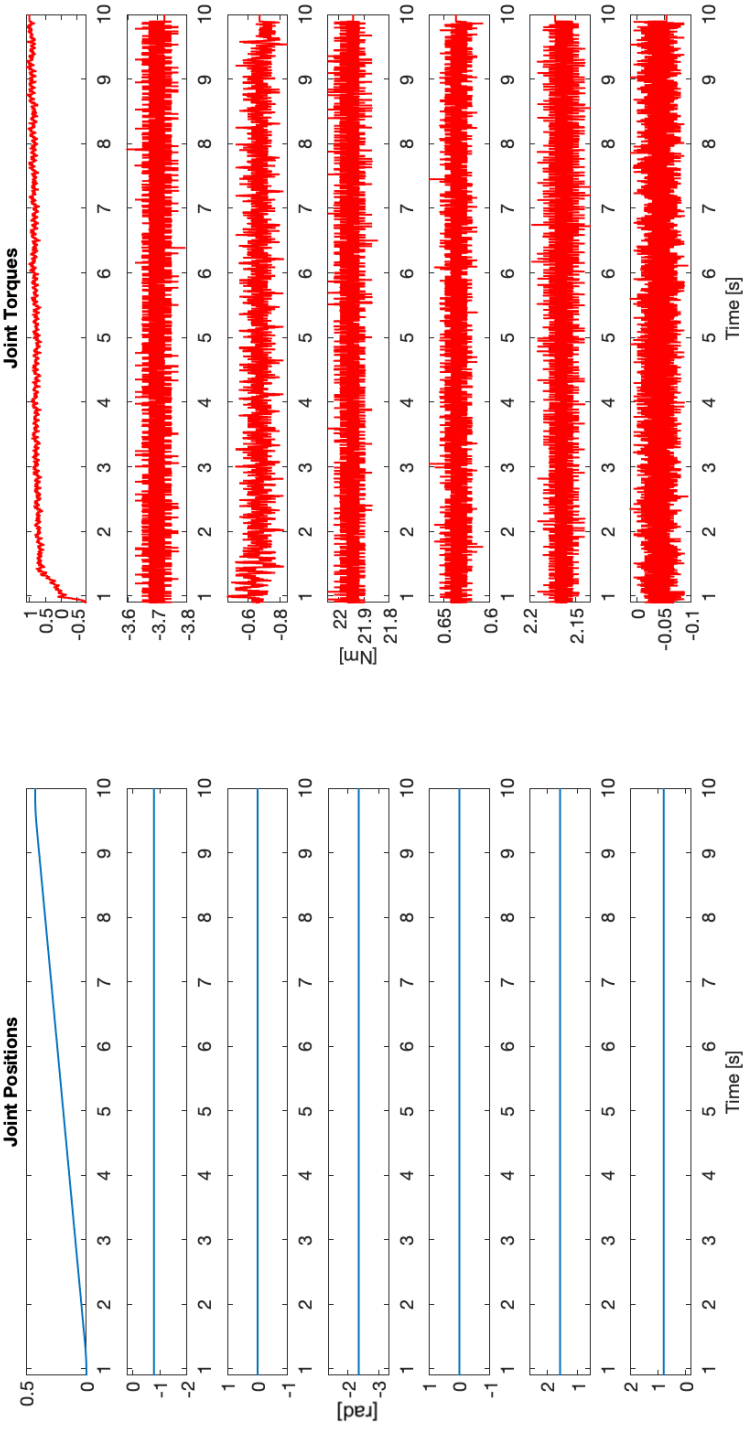

Figura 34: Verifica della comunicazione e visualizzazione dei dati robot

4

STIMA DEL PESO DEL CARICO

4.1 INTRODUZIONE

La stima dei parametri inerziali del carico è fondamentale per i controlli di forza, in quanto si ha la necessità di compensare esattamente le forze di gravità agenti sul braccio robotico e sul carico, così da mantenerlo in equilibrio statico. Inoltre tale stima deve essere sufficientemente accurata da garantire il corretto funzionamento del controllo.

Si presenterà, quindi, una fase preliminare in cui viene stimata la massa dell'oggetto trattenuto dal manipolatore, dopo la quale il robot avvierà il controllo di impedenza descritto in sezione 5.

I parametri inerziali del carico del manipolatore, descrivibile come corpo rigido, sono stati stimati tramite l'esecuzione di particolari traiettorie sufficientemente eccitanti per tali parametri.

In letteratura si trovano spiegazioni esaustive sulla stima sia dei parametri inerziali del carico che dei membri del manipolatore [12–15]. In particolare si è deciso di utilizzare un metodo più efficiente per la stima della massa che non necessitasse della conoscenza dei parametri inerziali dei membri. Questo metodo è descritto in [15].

Man mano che i manipolatori industriali evolvono verso strutture sempre più leggere, il contributo del carico alle coppie ri-

chieste agli attuatori aumenta in modo significativo. Pertanto, è necessaria una conoscenza accurata dei parametri inerziali del carico per l'ottimizzazione delle operazioni e il controllo basato su modello del robot.

Nel particolare caso dei robot collaborativi, che sono progettati per interagire fisicamente con gli umani in sicurezza, il controllo della coppia e il rilevamento delle collisioni sono funzioni di base, che necessitano di un modello dinamico molto accurato contenente una descrizione del carico. Questo metodo può essere utilizzato per compensare accuratamente la gravità dei carichi nel controllo della coppia e nel rilevamento delle collisioni, beneficiando della sua alta precisione.

Ci sono due metodi di approccio alla stima dei parametri inerziali del carico.

Uno presentata da Atkeson [14] in cui si misurano direttamente i parametri inerziali tramite un sensore di forza/coppia posizionato sul polso del robot. Questo tipo di metodo ha un'alta precisione senza necessitare di traiettorie appositamente progettate; tuttavia, è difficilmente applicabile, poiché la maggior parte dei robot industriali non è dotata di tali sensori di forza/coppia.

Il secondo metodo considera l'identificazione del carico del robot come un normale problema di identificazione dinamica del robot, poiché il carico è rigidamente fissato all'ultimo membro del manipolatore, e può quindi essere considerato come parte integrante dell'ultimo membro. Sono stati sviluppati diversi approcci di identificazione dinamica del payload di robot, tra cui quello descritto in [13].

La necessità di stimare anche i parametri dei membri porta a dover svolgere un intero esperimento di identificazione dinami-

ca del robot, il che rende difficile la sua applicazione industriale, poiché è complessa e gli errori dei parametri inerziali del robot vengono introdotti nei parametri inerziali del carico utile. Di conseguenza, l'accuratezza dell'identificazione dei metodi esistenti basati sulla corrente dell'attuatore è di circa uno o due ordini di grandezza inferiore a quella dei metodi che utilizzano il sensore di forza/coppia montato sul polso.

Inoltre è importante scegliere delle traiettorie che risultino in una buona eccitazione e accurata stima dei parametri.

Il metodo, utilizzato in questa tesi e descritto in [15], stima i parametri dei modelli di robot dinamici in base sia al movimento che ai dati di coppia degli attuatori misurati durante esperimenti di identificazione "ben progettati", ovvero muovendo il robot lungo traiettorie ottimizzate. I dati sulle coppie degli attuatori sono ottenute tramite le misure dei sensori di coppia dei giunti, senza quindi necessitare di sensori aggiuntivi. Il lavoro principale consiste nell'utilizzare la differenza della coppia degli attuatori tra movimentazioni lungo una stessa traiettoria con e senza carico utile.

4.2 INTRODUZIONE TEORICA

Il modello dinamico del manipolatore robotico ad n gradi di libertà (GdL) può essere derivato secondo il metodo Newton-Eulero o Lagrangiano. Entrambi producono l'equazione della dinamica con la stessa forma di espressione:

$$\tau = M(q, \theta^*)\ddot{q} + C(q, \dot{q}, \theta^*) + g(q, \theta^*) + \tau_f \tag{1}$$

che esprime la relazione dinamica tra il vettore delle coppie dell'attuatore τ e il moto dei membri. Nell'equazione 1 sono

presenti le posizioni dei giunti q, velocità $\dot{q}$ e accelerazioni $\ddot{q}$, i parametri inerziali di ciascun attuatore e membro θ^*. M è la matrice delle inerzie, C contiene le coppie dovute a forze di coriollis e centrifughe, g le coppie gravitazionali e τ_f le coppie di attrito complessive. M, C, g sono funzioni non lineari dei parametri di modello θ^*, vale a dire, la massa, la posizione del centro di gravità e i momenti e i prodotti di inerzia di ciascun membro.

Utilizzando dei parametri baricentrici oppure i parametri modificati θ di Newton-Eulero, al posto dei parametri inerziali originali θ^* si giunge a:

$$\tau = \Phi(q, \dot{q}, \ddot{q})\theta + \tau_f \tag{2}$$

che è lineare rispetto ai parametri inerziali, escludendo il termine dovuto agli attriti.

La matrice $\Phi_n(q, \dot{q}, \ddot{q})$ viene chiamata matrice di osservazione o di identificazione, e dipende solo dai dati sulla movimentazione.

Il carico può essere considerato parte dell'ultimo membro del robot. Così facendo il modello dinamico rimane lo stesso di equazione 7, tranne per i parametri baricentrici del link finale. I parametri inerziali $\hat{\theta}_n$ del settimo link deve combinare membro e carico assieme:

$$\hat{\theta}_n = [\hat{m}, \hat{mx}, \hat{my}, \hat{mz}, \hat{XX}, \hat{YY}, \hat{ZZ}, \hat{XY}, \hat{XZ}, \hat{YZ}]^\top \tag{3}$$

dove m è la massa, $MS = [mx, my, mz]$ sono i momenti del primo ordine e i rimanenti sono elementi del tensore d'inerzia I. [15]

Si può verificare che massa, momenti del primo ordine e tensore d'inerzia soddisfa il principio di sovrapposizione degli effetti, rendendo visibile l'indipendenza tra carico e settimo link.

$$\hat{m} = \hat{m}_n + \hat{m}_L \tag{4}$$

$$\hat{MS} = \hat{MS}_n + \hat{MS}_L \tag{5}$$

$$\hat{I} = \hat{I}_n + \hat{I}_L \tag{6}$$

Il modello dinamico si può quindi riscrivere come:

$$\tau = \Phi(q, \dot{q}, \ddot{q})\theta + \tau_f = \Phi(q, \dot{q}, \ddot{q})\theta_n + \Phi_n(q, \dot{q}, \ddot{q})\theta_L + \tau_f \tag{7}$$

La matrice $\Phi_n(q, \dot{q}, \ddot{q})$ contiene le ultime 10 righe della matrice di osservazione. Dunque, se si assume che l'attrito sia relativo solo al moto del robot, allora il contributo dato dal carico alle coppie degli attuatori può essere separata dal contributo dato da attuatori e link eseguendo la medesima traiettoria eccitante.

In generale, una buona scelta delle traiettorie risulta in una buona eccitazione e stima accurata dei parametri del robot.

$$\tau_{carico} = \Phi_n(q, \dot{q}, \ddot{q})\theta_L = \tau_{robot+carico} - \tau_{robot} \tag{8}$$

Questo ci evidenzia come usare la differenza delle coppie dell'attuatore lungo la stessa la stessa traiettoria nella condizione senza e con carico porti a stimare direttamente i parametri del

payload. I parametri di inerzia θ dei membri del robot non sono necessari, e la loro identificazione sarebbe stata complicata e avrebbe richiesto tempo. Inoltre, il modello di identificazione è notevolmente semplificato poiché servono solo le ultime dieci righe della matrice di osservazione del robot. Soprattutto, l'accuratezza dell'identificazione dei parametri di inerzia del carico utile può essere notevolmente migliorata, poiché si evitano gli errori di stima dei parametri degli attuatori e dei membri del robot.

Un fattore da tenere in considerazione, per quanto riguarda l'attrito, è la temperatura. Nell'equazione 8 l'attrito non compare perché perfettamente cancellato nella sottrazione, essendo considerato dipendente solo dal moto del manipolatore. Tale attrito, però, dipende anche dalla temperatura. La temperatura dell'ambiente può variare, come può variare anche la quantità di calore prodotto dagli attuatori durante il funzionamento. È evidente che se la temperatura rimane invariata allora possiamo considerare l'attrito dipendete prettamente dalla legge di moto. Tuttavia, bisogna notare che la generazione di calore da parte degli attuatori può essere considerevole in alcuni casi, in cui l'intervallo di tempo tra l'esecuzione delle due movimentazioni potrebbe essere prolungato e la temperatura dell'ambiente sia variata considerevolmente.

Le variazioni di temperatura tra il moto senza e con carico, quindi, vanno considerate, e anche l'attrito andrebbe identificato per una stima più accurata.

Inoltre, si precisa che i segnali di coppia di un attuatore fermo non vanno utilizzate per il processo di identificazione, a causa dell'incertezza data dall'attrito statico. Solo le coppie di attuatori in moto vanno usate per le stime, perché, in tal

caso, l'attrito è considerato come attrito dinamico (viscoso e coulombiano).

Per la stima eseguita in questa tesi non si eseguirà l'identificazione dell'attrito, e si presenterà solo la stima del primo parametro inerziale del carico: la massa. Per gli altri parametri si rimanda all'articolo [15].

La miglior traiettoria per l'identificazione della massa del carico è quella dove solo la massa contribuisce alla richiesta di coppia agli attuatori. Così l'accuratezza della stima non viene influenzata da altri parametri non considerati.

Quindi, una movimentazione a velocità uniforme è essenziale per eliminare l'effetto dato dal tensore d'inerzia. Quando sono in moto con velocità costanti gli ultimi quattro giunti e tutti gli altri giunti sono fermi alla posizione zero si può giungere ad una formulazione esplicita dell'equazione 8. Tenendo inoltre fermo alla posizione zero anche il quinto giunto si hanno ulteriori semplificazioni, giungendo alle seguenti equazioni:

$$\begin{aligned}\Delta\tau_4 = g(m_L d_5 sinq_4 + mz_L sin(q_4 + q_6) + \\ + (mx_L cosq_7 my_L sinq_7) cos(q_4 + q_6)) + \Delta\tau_{\dot{q}_4}\end{aligned} \tag{9}$$

$$\Delta\tau_5 = -q(mx_L sinq_7 + my_L cosq_7) sinq_4 + \Delta\tau_{\dot{q}_5} \tag{10}$$

$$\begin{aligned}\Delta\tau_6 = g(mz_L sin(q_4 + q_6) + \\ + (mx_L cosq_7 - my_L sinq_7) \times cos(q_4 + q_6)) + \Delta\tau_{\dot{q}_6}\end{aligned} \tag{11}$$

$$\Delta\tau_7 = -q(mx_L sinq_7 + my_L cosq_7) sin(q_4 + q_6)) + \Delta\tau_{\dot{q}_7} \tag{12}$$

Dove d_5 è la lunghezza del quinto link.

In queste equazioni sono presenti solo momenti dovuti alla gravità e momenti di coriollis e centripiti $\Delta\tau_{\dot{q}}$ causati dalla velocità. Non sono presenti coppie dovute da accelerazioni, il che vuol dire che la matrice delle inerzie non contribuisce nel modello dinamico quando ho moti uniformi.

Si noti che:

$$\Delta\tau_4 - \Delta\tau_6 = m_L g d_5 \sin q_4 \tag{13}$$

Ignorando i momenti dovuti a coriollis e centripeti. Abbiamo isolato con successo la massa m_L, ma dobbiamo eseguire la differenza tra $\Delta\tau_4$ e $\Delta\tau_6$, il che porta al sovrapporsi dei rumori di entrambi i segnali, aumentando la varianza della stima.

La traiettoria eccitante consiste nel mantenere:

- $q_4 + q_6 \equiv const$
- $q_5 \equiv 0$
- $q_7 \equiv const$

Con questi vincoli, $\Delta\tau_6$ rimane costante durante tutto il moto. Si può quindi utilizzare la media del segnale $\Delta\tau_6$ così da ridurre l'effetto del rumore a garantire una maggiore accuratezza. In [15], inoltre, viene dimostrato come le componenti di coriollis e centripete siano piccole e quindi trascurabili, quando la velocità $\dot{q}_4$ non è elevata.

4.3 STIMA DELLA MASSA

Per le prove di stima della massa del carico si è utilizzato un oggetto metallico dal peso di 0.307kg, e dei pesi per tara da 0.1kg. In figura 35 è riportata la stima ai minimi quadrati eseguita con MATLAB su una singola prova. Il segnale blu è la stima in ogni istante di tempo della massa mentre il valore costante rappresentato dalla retta rossa è la stima ai minimi quadrati. Lo script MATLAB per la stima ai minimi quadrati della massa è riportato in APPENDICE B.

Qui di seguito si riportano in tabella i risultati delle stime con la massa da 0.307kg:

	Massa effettiva [kg]	Massa stimata [kg]	Errore assoluto [kg]
Prova 1	0.307	0.297	0.01
Prova 2	0.307	0.301	0.006
Prova 3	0.307	0.303	0.004
Prova 4	0.307	0.299	0.008
Prova 5	0.307	0.305	0.002
Prova 6	0.307	0.294	0.013
Prova 7	0.307	0.302	0.005
Prova 8	0.307	0.294	0.013
Prova 9	0.307	0.296	0.011
Prova 10	0.307	0.304	0.003
Prova 11	0.307	0.306	0.001

Tabella 4: Stime della massa del carico

Si nota che le stime sono molto accurate e che si ha un errore inferiore ai 20g. Nell'articolo [15] si otteneva un errore massimo

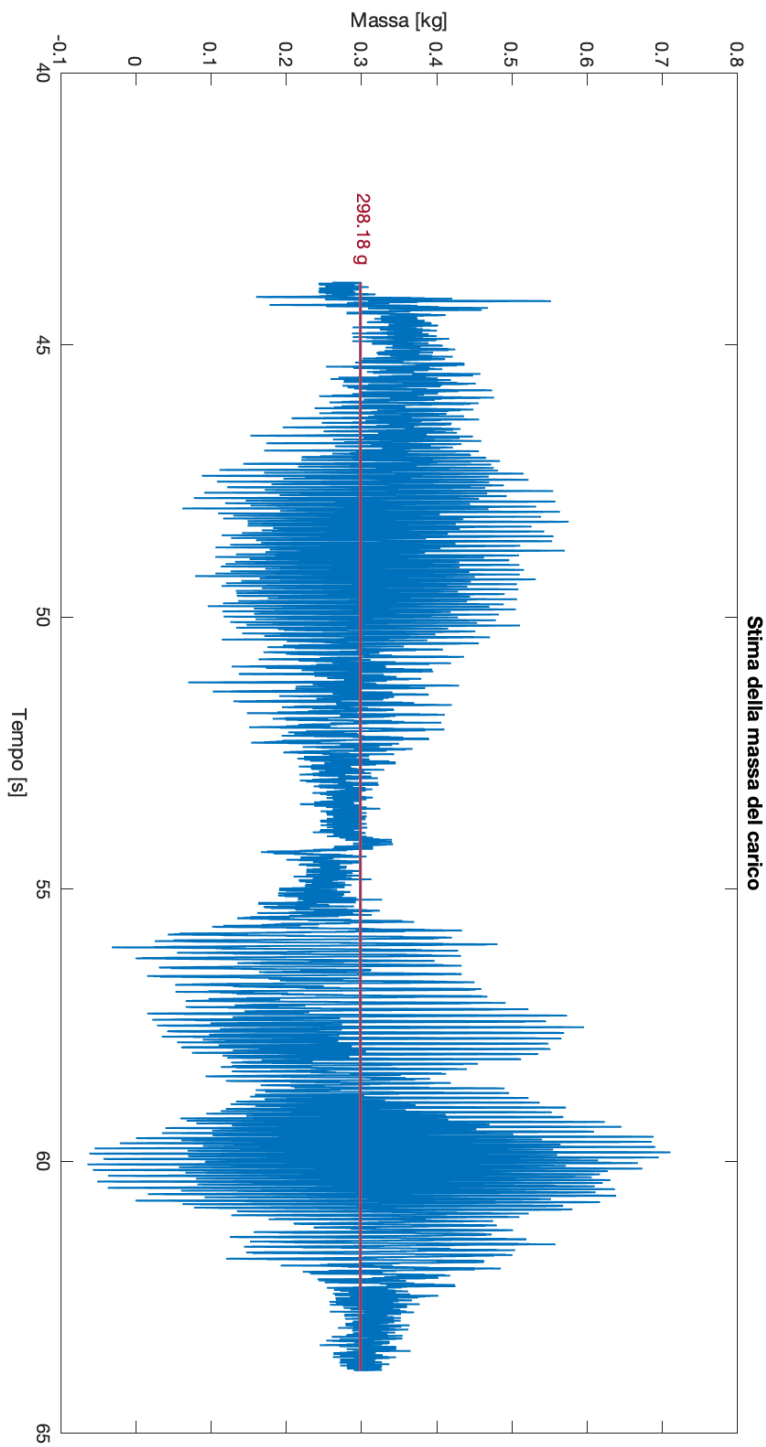

Figura 35: Stima ai minimi quadrati della massa (307 g effettivi).

di 30g, maggiore di quello riscontrato in questi esperimenti. La differenza può essere dovuta ad una migliore accuratezza dei sensori di coppia del robot Panda rispetto al robot industriale DRCA usato nell'articolo.

La stima è, quindi, soddisfacente per il corretto funzionamento del controllo di impedenza.

Il codice per l'esecuzione della traiettoria eccitante e per la stima ai minimi quadrati è il seguente:

```
1  double time_max = 10.0;
2  double omega_max = 0.3;
3  double time = 0.0;
4
5  // Definizione della legge di moto per la stima della massa
       del payload
6  std::function<franka::JointVelocities(const franka::
       RobotState&, franka::Duration)> motion = [&](const
       franka::RobotState& robot_state, franka::Duration period
       ) -> franka::JointVelocities {
7     // Tempo trascorso da inizio moto
8     time += period.toSec();
9
10    // Processo dati per legge di moto
11    double cycle = std::floor(std::pow(-1.0, (time - std::
          fmod(time, time_max)) / time_max));
12    double omega = cycle * omega_max / 2.0 * (1.0 - std::cos
          (2.0 * M_PI / time_max * time));
13
14    // Fine moto
15    if (time >= 2 * time_max) {
16       std::cout << std::endl << "Finished motion" << std::
             endl;
17       return franka::MotionFinished(velocities);
```

```
18      }
19
20      // Dati dal robot
21      std::array<double, 7> gravity_array = model.gravity(
            robot_state);
22      Eigen::Matrix<double, 7, 1> tau(robot_state.tau_J.data())
            ;
23      Eigen::Matrix<double, 7, 1> q(robot_state.q.data());
24
25      velocities = {{0, 0, 0, omega, 0, -omega, 0}};
26
27      // Raccolgo dati di coppia nella movimentazione senza
            carico
28      if(z==0){
29         for(int j=0;j<7;j++){
30         tau0[i][j] = tau(j);
31      }
32      // Calcolo coppia dovuta al carico
33      } else{
34         if(i<i_old){
35            for(int j=0;j<7;j++){
36               tau_diff1[7*i+j]= tau(j)-tau0[i][j];
37               if(i==0)tau_diff[7*i+j] = 0;
38            }
39            hh[i] = 9.81*0.384*sin(q[3]);
40            cont[0]++;
41         }
42
43         // Aggiorno i dati per le comunicazioni UDP
44         if (print_data.mutex.try_lock()) {
45            print_data.has_data = true;
46            print_data.robot_state = robot_state;
```

```
47          print_data.gravity = gravity_array;
48          std::copy_n(&tau_diff[7*i], 7, print_data.tau_d.
                begin());
49          print_data.mutex.unlock();
50       }
51    }
52    i++;
53
54    return velocities;
55 };
56
57 for(z=0;z<2;z++){
58    // Muovo il robot nella configurazione iniziale
59    std::array<double, 7> q_goal = {{0, 0, 0, -3*M_PI_4, 0,
          3*M_PI_4, M_PI_4}};
60    MotionGenerator motion_generator(0.1, q_goal);
61    std::cout << "Press Enter to continue..." << std::endl;
62    std::cin.ignore();
63    robot.control(motion_generator);
64    std::cout << "Finished moving to joint configuration." <<
           std::endl;
65    if(z==0)std::cin.ignore();
66
67    time = 0.0;
68    i = 0;
69    // Avvio movimentazione eccitante per la stima
70    robot.control(motion);
71
72    i_old=i;
73
74    if(z==1){
75       // Preparazione delle matrici per la stima tramite
```

```
            minimi quadrati della massa del payload
76                  Eigen::Map<Eigen::Matrix<double, LEN, 7,
                        Eigen::RowMajor> > tau_d_aux(tau_diff);
77                  Eigen::MatrixXd tau_d(LEN,7);
78                  tau_d = tau_d_aux.block(0,0,cont,7);
79                  Eigen::Map<Eigen::Matrix<double, LEN, 1> >
                        H_aux(hh);
80                  Eigen::MatrixXd H(LEN,1);
81                  H = H_aux.block(0,0,cont,1);
82                  Eigen::Map<Eigen::Matrix<double, LEN, 1> >
                        Y_aux(tau_d.col(5).data());
83                  Eigen::MatrixXd Y(LEN,1);
84                  Y = Y_aux.block(0,0,cont,1);
85                  Y = tau_d.col(5)-tau_d.col(3);
86
87                  Eigen::MatrixXd stima(1,1);
88                  // Stima ai minimi quadrati
89                  stima=(H.transpose()*H).inverse()*H.
                        transpose()*Y;
90                  stima_massa = stima(0);
91                  std::cout << "MASSA:" << stima_massa << std
                        ::endl << "----" << std::endl;
92      }
93  }
```

In cui a riga 85 si effettua la stima ai minimi quadrati.

4.3.1 *Legge di moto*

Nella figura 36 sono riportati i segnali di posizione e coppia durante le movimentazioni per la stima della massa del carico.

Come riportato alla fine della sezione 4.2 solo i giunti 4 e 6 si

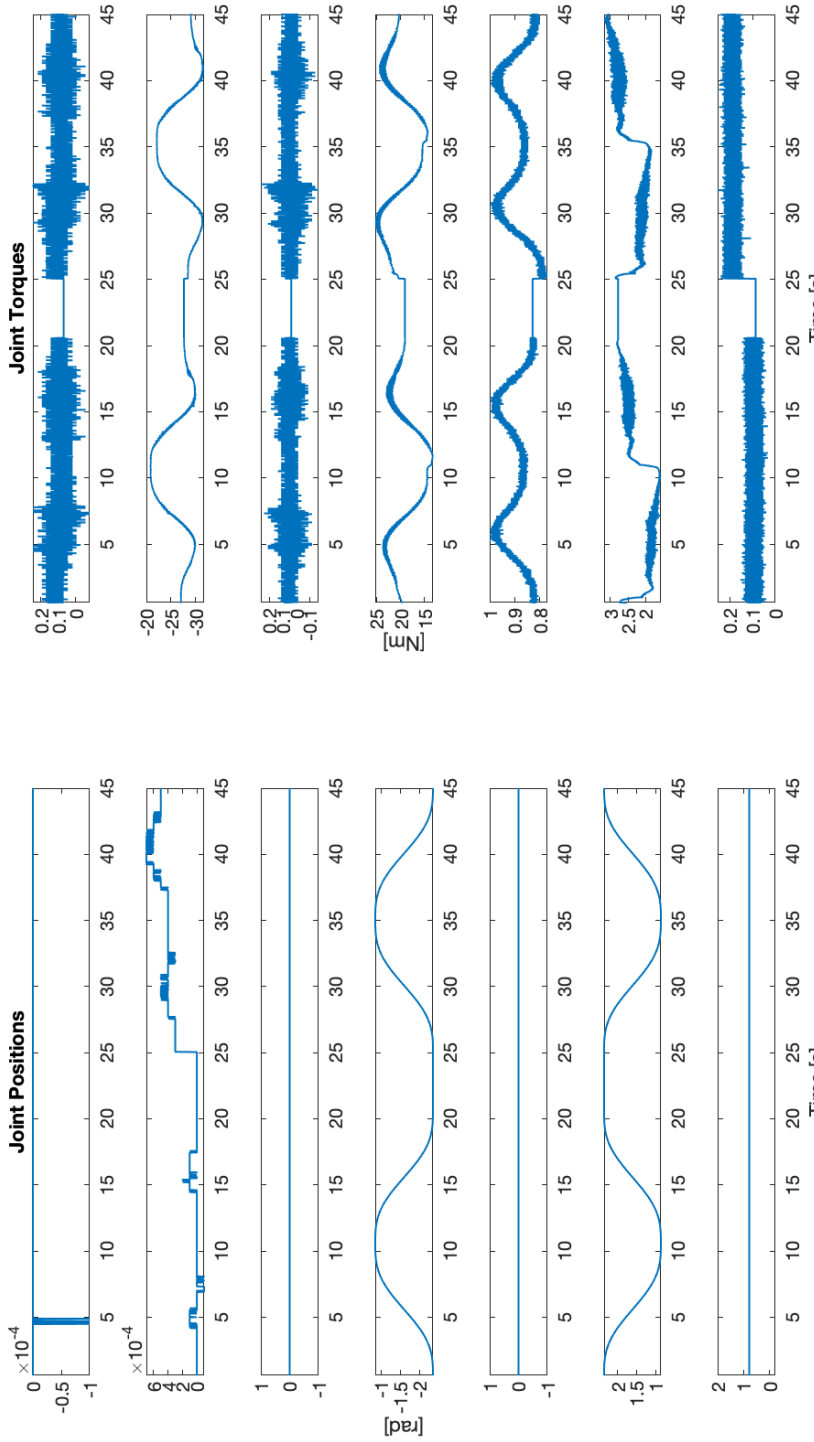

Figura 36: Posizioni e coppie durante movimentazione per la stima della massa del carico.

muovono, in sensi opposti.

$$q_4 + q_6 \equiv const$$

Si ricorda che la porzione di codice che gestisce la legge di moto è la seguente:

```
// Processo dati per legge di moto
double cycle = std::floor(std::pow(-1.0, (time - std::fmod(
    time, time_max)) / time_max));
double omega = cycle * omega_max / 2.0 * (1.0 - std::cos(2.0
     * M_PI / time_max * time));
```

Che svolgono i seguenti calcoli:

$$cycle = \left\lfloor (-1)^{\frac{t - mod(t, t_{max})}{t_{max}}} \right\rfloor$$

che restituisce 1 oppure -1. L'esponente sarà 0 se il tempo è minore di t_{max}, 1 se $t_{max} < t < 2t_{max}$, 2 se $2t_{max} < t < 3t_{max}$ e così via. Di conseguenza quando questo sarà dispari si avrà $cycle = -1$, quando sarà para $cycle = 1$. Portando ad eseguire la stessa traiettoria in andata e poi in ritorno.

$$\omega = \frac{cycle \cdot \omega_{max}}{2}(1 - \cos(2\pi \cdot \frac{t}{t_{max}}))$$

ω è la velocità della legge di moto cicloidale, che porta ad avere gli andamenti cinematici di figura 37.

Si ricorda che abbiamo analizzato solo il moto del giunto 4, ma il giunto 6 segue la stessa legge di moto ma in verso opposto.

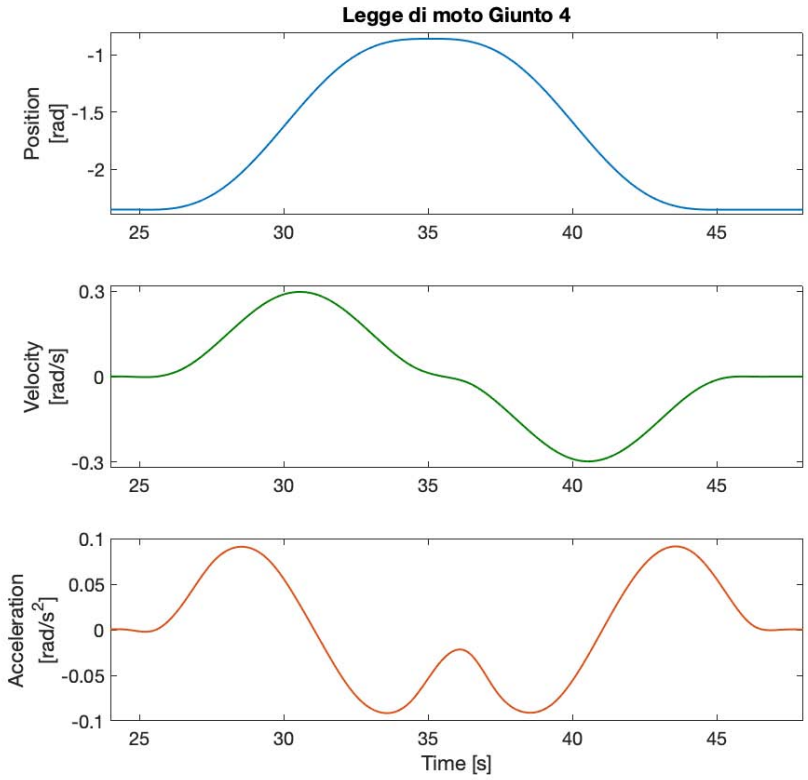

Figura 37: Posizione, velocità ed accelerazione della traiettoria per la stima della massa del carico.

5

CONTROLLO DI IMPEDENZA

5.1 INTRODUZIONE

Il controllo di forza in un robot collaborativo è di fondamentale importanza, in quanto è necessario gestire le interazioni dinamiche tra manipolatore e ambiente, e rendere ogni movimentazione innocua per gli operatori. Per tali robot il sistema di controllo deve essere implementato così che il manipolatore lavori adattandosi alle caratteristiche umane.

Si può progettare un controllo convenzionale cinematico per gestire le traiettorie di lavoro, e in aggiunta modellare la risposta da noi desiderata ai disturbi. Il controllo così strutturato prende il nome di *controllo di impedenza* [1–10].

5.2 INTRODUZIONE TEORICA

Come descritto in [2], i sistemi fisici posso essere di due tipi: ammettenze che accettano in ingresso forze e forniscono in uscita movimenti, oppure impedenze che hanno come input movimenti e danno in output forze.

Nelle interazioni tra due soggetti meccanici, come il manipolatore e l'ambiente, ci deve essere complementarietà tra i sistemi, con ciò si intende che se uno dei due è un'impedenza, l'altro dovrà essere un'ammettenza e viceversa. Tipicamente l'ambiente è costituito da inerzie e vincoli cinematici, che sono modella-

bili sempre come ammettenze, e solo in alcuni casi come impedenze. A tal proposito è opportuno che il manipolatore assuma le caratteristiche di un impedenza, così da garantire in tutti i casi la complementarietà con l'ambiente.

A differenza del controllo di forza tradizionale, in cui viene fornito a priori un profilo di forza di contatto e delle coppie da seguire, nel controllo di impedenza viene scelto il comportamento dinamico da assumere nelle deviazioni dalla traiettoria pianificata.

L'interazione dinamica tra manipolatore e ambiente viene regolata variando l'impedenza, da qui ne deriva il nome *controllo di impedenza*.

Il robot è in grado di imprimere coppie $\bar{T}$ e di misurare le posizioni degli attuatori $\bar{\theta}$.

Ogni impedenza ha come termine di ordine minore una rigidezza, ovvero una relazione statica tra forza in uscita e scostamento di ingresso.

Definendo $\bar{x}_0$ la posizione di equilibrio desiderata dell'end effector in assenza di forze esterne, e $\bar{x} = L(\bar{\theta})$ la posizione dell'end point, ricavata per cinematica diretta dalle informazioni sui giunti, si giunge alla seguente formulazione:

$$\bar{F} = k \cdot (\bar{x}_0 - \bar{x}) \tag{14}$$

Nell'equazione (14) la k è la costante elastica e $\bar{F}$ è la forza elastica derivante dall'errore di posizione $\bar{x}_0 - \bar{x}$. Ci si può riportare ad una formulazione che abbia come output le coppie dei vari giunti utilizzando lo Jacobiano e il principio dei lavori virtuali.

$$d\bar{x} = J(\bar{\theta}) \cdot d\bar{\theta} \tag{15}$$

$$\bar{T} = J^T(\bar{\theta}) \cdot \bar{F} \tag{16}$$

$$\bar{T} = J^T(\bar{\theta}) \cdot k \cdot (\bar{x}_0 - L(\bar{\theta})) \tag{17}$$

In [2] viene fatto notare che se si implementa un controllo molto rigido, quindi con k elevato, allora la relazione (17) realizza il controllo di posizione cartesiano dell'end point, eliminando il problema cinematico inverso.

Un altro termine importante regola la relazione tra forza e velocità:

$$\bar{F} = B \cdot \bar{v} \tag{18}$$

$$\bar{v} = J(\bar{\theta}) \cdot \bar{w} \tag{19}$$

$$\bar{T} = J^T(\bar{\theta}) \cdot B \cdot (J(\bar{\theta}) \cdot \bar{w}) \tag{20}$$

Il risultato è uno smorzamento della risposta dinamica del controllo. Secondo Hogan [2] il comportamento dinamico da imporre al manipolatore deve essere il più semplice possibile, ma mai più semplice di così. Si deve, quindi, sempre inserire almeno la rigidità e viscosità nel controllo di impedenza.

Si può pensare al controllo di impedenza come ad uno schema complesso che risponde come un sistema dinamico del secondo ordine.

Dalla figura 38 si ricava la formulazione (21)

$$M\ddot{x}(t) + B\dot{x}(t) + Kx(t) = F(t) \tag{21}$$

che nel dominio di Laplace è riscrivibile come:

$$X(s) \cdot (Ms^2 + Bs + K) = F(s) \tag{22}$$

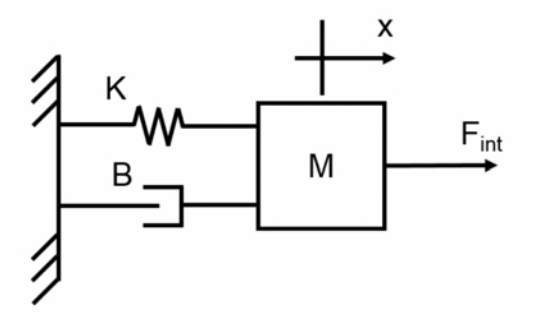

Figura 38: Modello di un sistema meccanico del secondo ordine con forza di interazione.

Si ricava infine la funzione di trasferimento, ingresso forza - uscita spostamento, denominata *impedenza*:

$$Z(s) = \frac{X(s)}{F(s)} = \frac{1}{Ms^2 + Bs + K} \tag{23}$$

Se il manipolatore durante i task potesse scegliere arbitrariamente $X(s)$ e $F(s)$ allora si avrebbe un controllo accurato e performante. Tuttavia, nel controllo di impedenza, tali grandezze sono correlate e non possono essere controllate indipendentemente.

Come impedenza solitamente si utilizza la struttura appena vista in (23), dove M, B e K rappresentano l'inerzia, lo smorzamento e la rigidità desiderati.

Nel dominio del tempo, l'impedenza si può esprimere dall'equazione differenziale (24).

$$M(\ddot{x} - \ddot{x}_d) + B(\dot{x} - \dot{x}_d) + K(x - x_d) = F(t) \tag{24}$$

dove $x(t)$ sono le posizioni effettive e $x_d(t)$ sono le posizioni della traiettoria pianificata.

Dai chiarimenti appena proposti, si può definire il controllo di impedenza uno schema di controllo capace di realizzare

contemporaneamente la regolazione del movimento e della forza di contatto semplicemente imponendo la relazione dinamica desiderata tra di essi. [6]

5.3 MOTO PRIVO DI ATTRITO - CONTROLLORE PI

Inizialmente si è implementato un controllo che cancellasse in parte gli attriti sui giunti del manipolatore, permettendo di manovrare liberamente nello spazio il braccio robotico senza percepire resistenza.

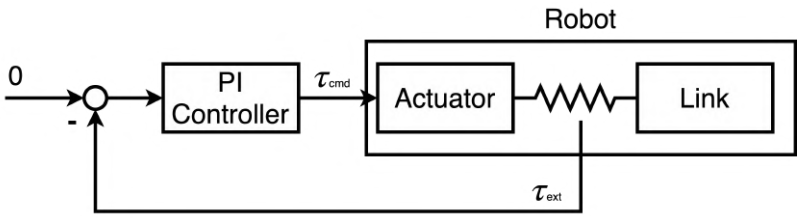

Figura 39: Schema semplificato del controllo di cancellazione dell'attrito.

In figura 39 è riportato uno schema semplificato del controllo che si vuole implementare, descrittivo di un singolo giunto ma facilmente estendibile a tutto il robot. Dal robot si utilizzano per la retroazione i valori di coppia esterna misurata lato link, al netto della componente gravitazionale. Questi valori vengono forniti direttamente dalla libreria *libfranka* senza la necessità di sottrarre manualmente l'effetto delle forze di gravità su ogni membro.

In figura 39 si nota che la misura della coppia esterna applicata al link proviene da un sensore posto su un giunto elastico. Retroazionare tale grandezza porta il controllore PI a generare un riferimento per il robot che induce l'attuatore ad inseguire il link, eliminando così ogni resistenza nel moto.

Di seguito è riportata la porzione di codice del programma del robot adibita al controllo appena descritto.

```
1
2     constexpr double k_p{0.2};
3     constexpr double k_i{3.0};
4     // define callback for the torque control loop
5     double time = 0.0;
6     auto force_control_callback = [&](const franka::
          RobotState& robot_state, franka::Duration period) ->
           franka::Torques {
7       time += period.toSec();
8
9       // get state variables
10      std::array<double, 42> jacobian_array = model.
            zeroJacobian(franka::Frame::kEndEffector,
            robot_state);
11
12      Eigen::Map<const Eigen::Matrix<double, 6, 7> >
            jacobian(jacobian_array.data());
13      Eigen::Map<const Eigen::Matrix<double, 7, 1> >
            tau_measured(robot_state.tau_J.data());
14      Eigen::Map<const Eigen::Matrix<double, 7, 1> > gravity
            (model.gravity(robot_state).data());
15      Eigen::Map<const Eigen::Matrix<double, 7, 1> >
            coriolis(model.coriolis(robot_state).data());
16
17      Eigen::VectorXd tau_d(7), desired_force_torque(6),
            tau_cmd(7), tau_ext(7);
18      desired_force_torque.setZero();
19      tau_ext << tau_measured - gravity;
20      tau_d << jacobian.transpose() * desired_force_torque;
21      tau_error_integral += period.toSec() * (tau_d -
```

```
            tau_ext);
22        // FF + PI control
23        tau_cmd << tau_d + k_p * (tau_d - tau_ext) + k_i *
              tau_error_integral + coriolis;
24
25        std::array<double, 7> tau_d_array{};
26        Eigen::VectorXd::Map(&tau_d_array[0], 7) = tau_cmd;
27        return tau_d_array;
28      };
```

La taratura del PI è stata fatta sperimentalmente. La linea di codice in cui viene elaborato il valore del comando da inviare al robot è la seguente:

```
1 tau_cmd << tau_d + k_p * (tau_d - tau_ext) + k_i *
    tau_error_integral + coriolis;
```

Si noti che può essere definito anche un valore di feedforward, che nel codice riportato era impostato a zero, nel caso si voglia fare seguire un particolare profilo di coppia ai giunti del robot.

Si può definire la dinamica risultante da questo controllo come ad impedenza minima, in quanto il manipolatore ostacola al minimo il moto impresso dall'ambiente.

5.4 MOTO PSEUDO-VISCOSO

Ottenuto il moto senza attriti del manipolatore, si procede con la gestione dell'attrito viscoso virtualmente aggiunto dal controllo. Mantenendo il controllo PI si può variare arbitrariamente la quantità percepita di attrito nel moto.

Si possono seguire due strategie diverse:

- controllo dei giunti

- controllo cartesiano

di cui il primo ha una struttura più semplice e diretta e il secondo porta ad un esperienza ottimale per l'operatore.

5.4.1 *Controllo viscoso dei giunti*

Si procederà ora con la descrizione del controllo viscoso dei giunti.

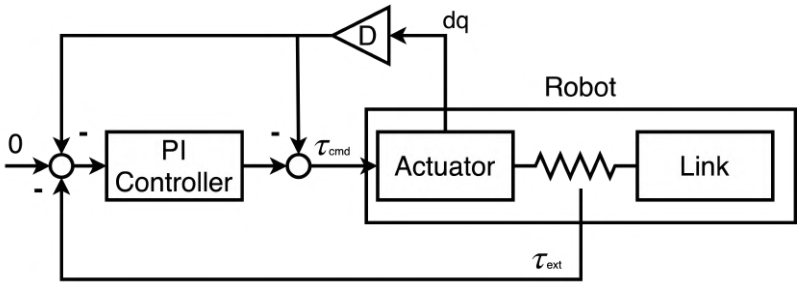

Figura 40: Schema semplificato del controllo viscoso dei giunti.

In figura 40 si nota una retroazione aggiuntiva rispetto a fig.39. Come per il precedente schema anche questo è descrittivo di un singolo giunto. Il valore usato per il feedback è quello di velocità angolare dei giunti `dq`. Da essa si calcola una forza proporzionale a `dq` ma che si oppone al moto, quindi di segno opposto. Tale grandezza viene sommata al risultato del controllo PI e inviato come comando agli attuatori del manipolatore. Per evitare che il controllo PI azzeri a regime questa nuova componente, si deve avere l'accortezza di rimuoverla preventivamente dall'input del controllore PI.

Per evitare, inoltre, comportamenti instabili del controllo, si gestiscono le costanti proporzionale e integrale del controllore PI inversamente proporzionali alla costante D di attrito viscoso virtuale. Se si lasciassero costanti, infatti, al crescere di D si ot-

terrebbero delle vibrazioni degli attuatori, dovute alla non esatta cancellazione preventiva della componente di attrito viscoso virtuale.

Di seguito è riportata la porzione di codice che implementa il controllo di forza:

```
1    // define callback for the torque control loop
2         double time = 0.0;
3         auto force_control_callback = [&](const franka::
              RobotState& robot_state, franka::Duration period
              ) -> franka::Torques {
4           time += period.toSec();
5
6           if (attrito_read.mutex.try_lock()) {
7               // Smoothly update the damping to reach the
                    desired target value
8               attrito_read.attrito = filter_gain *
                    attrito_read.target_attrito + (1 -
                    filter_gain) * attrito_read.attrito;
9         attrito_read.mutex.unlock();
10      }
11
12          // get state variables
13          std::array<double, 42> jacobian_array = model.
                zeroJacobian(franka::Frame::kEndEffector,
                robot_state);
14
15          Eigen::Map<const Eigen::Matrix<double, 6, 7> >
                jacobian(jacobian_array.data());
16          Eigen::Map<const Eigen::Matrix<double, 7, 1> >
                tau_measured(robot_state.tau_J.data());
17          Eigen::Map<const Eigen::Matrix<double, 7, 1> > dq(
                robot_state.dq.data());
```

```
18          Eigen::Map<const Eigen::Matrix<double, 7, 1> >
                gravity(model.gravity(robot_state).data());
19          Eigen::Map<const Eigen::Matrix<double, 7, 1> >
                coriolis(model.coriolis(robot_state).data());
20
21          Eigen::VectorXd tau_d(7), tau_cmd(7), tau_ext(7);
22
23          // FF + PI control
24
25          tau_ext << tau_measured - gravity + attrito_read.
                attrito * d_gains * dq;
26        tau_error_integral += period.toSec() * tau_ext;
27        tau_cmd << - (1 - attrito_read.attrito) * k_p *
              tau_ext - (1 - attrito_read.attrito) * k_i *
              tau_error_integral - attrito_read.attrito *
              d_gains * dq + coriolis;
28
29
30          std::array<double, 7> tau_d_array{};
31          Eigen::VectorXd::Map(&tau_d_array[0], 7) = tau_cmd
                ;
32
33          // Update data for UDP comunication.
34         if (print_data.mutex.try_lock()) {
35           print_data.has_data = true;
36           print_data.robot_state = robot_state;
37           print_data.tau = tau_ext;
38           print_data.mutex.unlock();
39         }
40
41          return tau_d_array;
42        };
```

5.4.2 *Controllo viscoso cartesiano*

Per implementare il controllo cartesiano con attrito viscoso virtuale è necessario riportare lo schema complessivo del controllo, e non del singolo giunto, in quanto il vettore $\dot{\bar{q}}$ contenente le sette velocità angolari dei giunti deve essere utilizzato nella sua interezza per giungere alle grandezze cinematiche dell'end-effector, e poi ritornare ad una formulazione delle coppie di comando per gli attuatori. In figura 41 è riportato uno schema semplificato del controllo qui descritto.

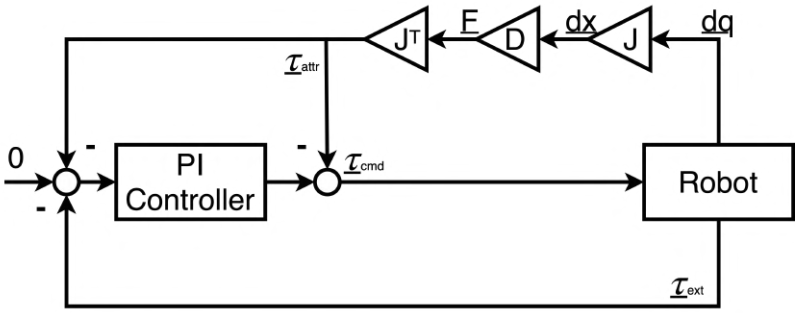

Figura 41: Schema semplificato del controllo viscoso cartesiano.

Per le trasformazioni da grandezze dei giunti e grandezze dell'end-effector si utilizza la matrice Jacobiana, facilmente calcolabile con la libreria *libfranka*.

Le velocità angolari dei link $\dot{\bar{q}}$ vengono riportate a velocità lineari e angolari dell'organo terminale del manipolatore. A questo punto si calcolano le forze resistenti proporzionali alle velocità, e successivamente si ritorna a grandezze di giunto, utilizzando la matrice Jacobiana trasposta.

Il controllo cartesiano è potenzialmente rischioso in termini di singolarità, mentre il controllo dei giunti porta sempre a configurazioni realizzabili e non singolari. [7]

5.5 CONTROLLO VISCO-ELASTICO

Anche nel caso del controllo visco-elastico si può procedere sviluppandolo come controllo cartesiano o di giunto.

5.5.1 *Controllo visco-elastico dei giunti*

Grazie alle funzionalità della libreria *libfranka*, definendo contemporaneamente il controllo di posizione e il controllo di forza, si sfrutta il calcolo della cinematica inversa.

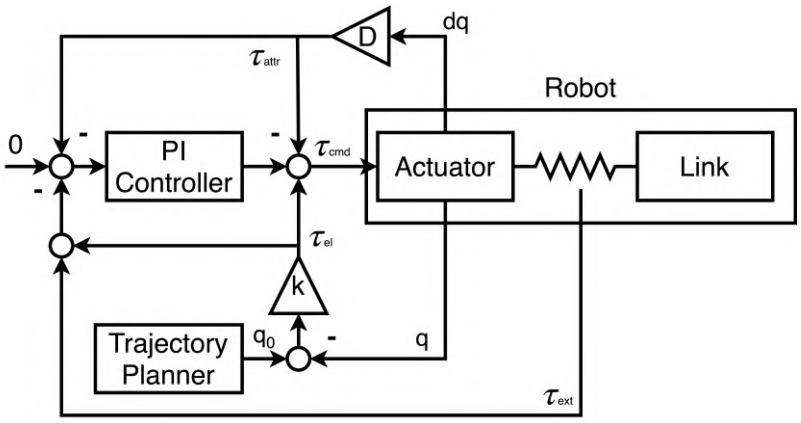

Figura 42: Schema semplificato del controllo visco-elastico dei giunti.

Come riportato in figura 42, si esegue una semplice differenza tra la posizione angolare desiderata e quella attuale del giunto. Tali informazioni sono note grazie alla risoluzione del problema cinematico inverso svolto automaticamente nel controllo di posizione dalla libreria *libfranka* .

Di seguito è riportata la porzione di codice contenente il controllo di posizione e il controllo di forza.

```
1     // Define callback function to send Cartesian pose goals
          to get inverse kinematics solved.
```

```
2     auto cartesian_pose_callback = [=, &time, &vel_current,
          &running, &angle, &initial_pose](const franka::
          RobotState& robot_state, franka::Duration period) ->
           franka::CartesianPose {
3       // Update time.
4       time += period.toSec();
5       if (time == 0.0) {
6         // Read the initial pose to start the motion from in
               the first time step.
7         initial_pose = robot_state.O_T_EE_c;
8       }
9       // Compute Cartesian velocity.
10      if (vel_current < vel_max && time < run_time) {
11        vel_current += period.toSec() * std::fabs(vel_max /
              acceleration_time);
12      }
13      if (vel_current > 0.0 && time > run_time) {
14        vel_current -= period.toSec() * std::fabs(vel_max /
              acceleration_time);
15      }
16      vel_current = std::fmax(vel_current, 0.0);
17      vel_current = std::fmin(vel_current, vel_max);
18      // Compute new angle for our circular trajectory.
19      angle += period.toSec() * vel_current / std::fabs(
            radius);
20      if (angle > 2 * M_PI) {
21        angle -= 2 * M_PI;
22      }
23      // Compute relative y and z positions of desired pose.
24      double delta_y = radius * (1 - std::cos(angle));
25      double delta_z = radius * std::sin(angle);
26      franka::CartesianPose pose_desired = initial_pose;
```

```
27        pose_desired.O_T_EE[13] += delta_y;
28        pose_desired.O_T_EE[14] += delta_z;
29        // Send desired pose.
30        if (time >= run_time + acceleration_time) {
31          running = false;
32          return franka::MotionFinished(pose_desired);
33        }
34        return pose_desired;
35      };
36      // Set gains for the joint impedance control.
37      // Stiffness
38      const std::array<double, 7> k_gains = {{600.0, 600.0,
            600.0, 600.0, 250.0, 150.0, 50.0}};
39      // Damping
40      const std::array<double, 7> d_gains = {{50.0, 50.0,
            50.0, 50.0, 30.0, 25.0, 15.0}};
41
42  // Define callback for the joint torque control loop.
43      std::function<franka::Torques(const franka::RobotState&,
            franka::Duration)>
44          impedance_control_callback = [&print_data, &model,
              k_gains, d_gains](const franka::RobotState&
              state, franka::Duration /*period*/) -> franka::
              Torques {
45        // Read current coriolis terms from model.
46        std::array<double, 7> coriolis = model.coriolis(state)
              ;
47        // Compute torque command from joint impedance control
              law.
48        // Note: The answer to our Cartesian pose inverse
              kinematics is always in state.q_d with one
49        // time step delay.
```

```
50      std::array<double, 7> tau_d_calculated;
51      for (size_t i = 0; i < 7; i++) {
52        tau_d_calculated[i] =
53            k_gains[i] * (state.q_d[i] - state.q[i]) -
                  d_gains[i] * state.dq[i] + coriolis[i];
54      }
55      // The following line is only necessary for printing
            the rate limited torque. As we activated
56      // rate limiting for the control loop (activated by
            default), the torque would anyway be
57      // adjusted!
58      std::array<double, 7> tau_d_rate_limited =
59          franka::limitRate(franka::kMaxTorqueRate,
                tau_d_calculated, state.tau_J_d);
60
61      // Send torque command.
62      return tau_d_rate_limited;
63    };
```

5.5.2 *Controllo visco-elastico cartesiano*

Nel controllo visco-elastico cartesiano devo calcolare le forze elastica e di attrito viscoso e successivamente riportarle a valori di coppia per gli attuatori, tramite la matrice Jacobiana.

Dalla figura 43, si può riscontrare l'implementazione dell'equazione (17), descrittiva della forza elastica agente sull'end-effector.

Di seguito è riportato il codice che implementa questo controllo. Si osservi che per il calcolo dello scostamento dalla posizione pianificata si utilizzano i quaternioni.

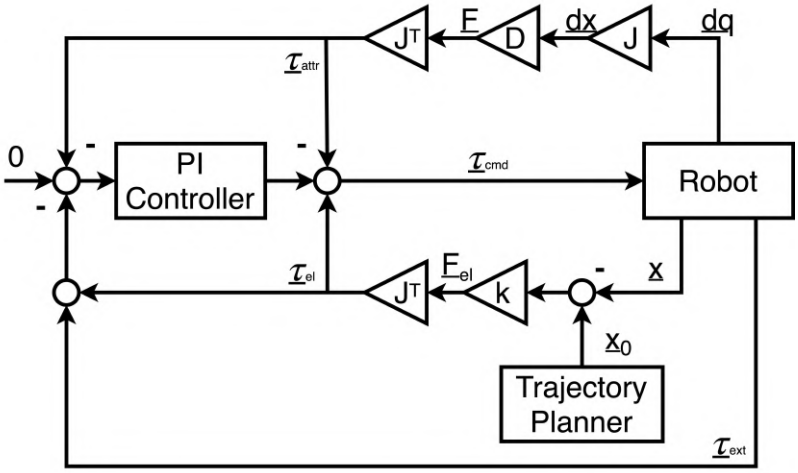

Figura 43: Schema semplificato del controllo visco-elastico cartesiano.

```
// define callback for the torque control loop
    std::function<franka::Torques(const franka::RobotState&,
        franka::Duration)>
        impedance_control_callback = [&](const franka::
            RobotState& robot_state, franka::Duration /*
            duration*/) -> franka::Torques {
      // get state variables
      std::array<double, 7> coriolis_array = model.coriolis(
          robot_state);
      std::array<double, 42> jacobian_array =
          model.zeroJacobian(franka::Frame::kEndEffector,
              robot_state);
      // convert to Eigen
      Eigen::Map<const Eigen::Matrix<double, 7, 1>> coriolis
          (coriolis_array.data());
      Eigen::Map<const Eigen::Matrix<double, 6, 7>> jacobian
          (jacobian_array.data());
      Eigen::Map<const Eigen::Matrix<double, 7, 1>> q(
          robot_state.q.data());
      Eigen::Map<const Eigen::Matrix<double, 7, 1>> dq(
          robot_state.dq.data());
```

```
13      Eigen::Affine3d transform(Eigen::Matrix4d::Map(
            robot_state.O_T_EE.data()));
14      Eigen::Vector3d position(transform.translation());
15      Eigen::Quaterniond orientation(transform.linear());
16      // compute error to desired equilibrium pose
17      // position error
18      Eigen::Matrix<double, 6, 1> error;
19      error.head(3) << position - position_d;
20      // orientation error
21      // "difference" quaternion
22      if (orientation_d.coeffs().dot(orientation.coeffs()) <
             0.0) {
23        orientation.coeffs() << -orientation.coeffs();
24      }
25      // "difference" quaternion
26      Eigen::Quaterniond error_quaternion(orientation.
            inverse() * orientation_d);
27      error.tail(3) << error_quaternion.x(),
            error_quaternion.y(), error_quaternion.z();
28      // Transform to base frame
29      error.tail(3) << -transform.linear() * error.tail(3);
30      // compute control
31      Eigen::VectorXd tau_task(7), tau_d(7);
32      // Spring damper system with damping ratio=1
33      tau_task << jacobian.transpose() * (-stiffness * error
             - damping * (jacobian * dq));
34      tau_d << tau_task + coriolis;
35      std::array<double, 7> tau_d_array{};
36      Eigen::VectorXd::Map(&tau_d_array[0], 7) = tau_d;
37      return tau_d_array;
38    };
```

In APPENDICE A è riportato un codice riassuntivo contenen-

te:

- comunicazione UDP
- stima della massa del carico
- controllo di impedenza (visco-elastico cartesiano)

CONCLUSIONI

La tesi si è focalizzata nell'implementazione di algoritmi di controllo di forza sul robot collaborativo leggero Panda della Franka Emika. I controlli di impedenza risultano di facile implementazione su questo genere di robot a causa della semplice gestione dei segnali di coppia dei giunti disponibili. Nella tesi si è mantenuto il controllo di impedenza il più semplice possibile, ma in futuro si può facilmente aggiungere la componente inerziale nel controllo di impedenza che non è stata considerata in questo elaborato. Si ricordi comunque che secondo Hogan [2] il controllo di impedenza deve contenere almeno la componente elastica e viscosa.

Per poter eseguire questo genere di controllo anche in presenza di carichi non noti a priori, nasce la necessità di stimare i parametri inerziali del payload. In questo elaborato si è spiegato come identificare la massa del carico e si sono tralasciate la posizione del centro di massa e la matrice inerziale a causa della mancanza di tempo per finire le prove sperimentali. Delle posizioni del centro di massa si è verificato tramite MATLAB che il procedimento porta a valori sensati, e una prima implementazione sul robot a portato a risultati non corretti, motivo per cui si è deciso di limitarsi alla stima della massa.

Durante l'implementazione della comunicazione tramite pacchetti UDP per la visualizzazione dei segnali in tempo reale non si sono riscontrati problemi di alcun tipo, grazie anche all'elevata frequenza di funzionamento del ciclo di controllo del robot Panda (1kHz). In questo ambito si hanno svariati

possibili sviluppi. La visualizzazione può essere portata alla realtà aumentata/mista, su visori come *Hololens*, per la visione sovrapposta alla realtà dei dati di forza/coppia del robot.

Si può anche sfruttare la comunicazione UDP nella direzione opposta, per controllare in remoto il manipolatore, tramite dispositivi quali il *Phantom*, un controllore aptico.

APPENDIX

A

CODICE DI RIEPILOGO

Per raggruppare i concetti esaminati in questa tesi, si è riportato di seguito un programma del robot Panda in cui si ritrovano le tre componenti studiate:

- Comunicazione tramite UDP
- Stima della massa del carico
- Controllo di impedenza

```
1  #include <array>
2  #include <atomic>
3  #include <cmath>
4  #include <functional>
5  #include <iostream>
6  #include <iterator>
7  #include <mutex>
8  #include <thread>
9  #include <math.h>
10 #include <bitset>
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <unistd.h>
15 #include <string.h>
16 #include <sys/types.h>
17 #include <sys/socket.h>
18 #include <arpa/inet.h>
```

```
19 #include <netinet/in.h>
20
21 #include <Eigen/Dense>
22
23 #include <franka/duration.h>
24 #include <franka/exception.h>
25 #include <franka/model.h>
26 #include <franka/rate_limiting.h>
27 #include <franka/robot.h>
28
29 #include "examples_common.h"
30
31 // Porta per la comunicazione UDP
32 #define PORT    8080
33 // Dimensione dei pacchetti UDP in bit
34 #define SIZE 32*(7*4)
35 // Lunghezza dei vettori per la stima della massa del
       payload
36 #define LEN 20000
37
38 namespace {
39 template <class T, size_t N>
40 std::ostream& operator<<(std::ostream& ostream, const std::
       array<T, N>& array) {
41     ostream << "[";
42     std::copy(array.cbegin(), array.cend() - 1, std::
           ostream_iterator<T>(ostream, ","));
43     std::copy(array.cend() - 1, array.cend(), std::
           ostream_iterator<T>(ostream));
44     ostream << "]";
45     return ostream;
46 }
```

```
47 }
48
49 int main(int argc, char** argv) {
50     // Controllo se ho il numero di argomenti corretto da
           linea di comando.
51     if (argc != 1) {
52         std::cerr << "Usage: " << argv[0] << " <robot-
               hostname>" << std::endl;
53         return -1;
54     }
55
56     // Frequenza di ciclo pre la comunicazione UDP
57     const double print_rate = 1000.0;
58
59     const char* fc_ip = "172.16.0.2";
60
61     // Inizializzazione dei dati nel mutex per lo scambio
           dati con il thread per la comunicazione UDP.
62     struct {
63         std::mutex mutex;
64         bool has_data;
65         franka::RobotState robot_state;
66         std::array<double, 7> gravity;
67         std::array<double, 7> tau_d;
68     } print_data{};
69     std::atomic_bool running{true};
70
71     // Inizializzazione dei dati nel mutex per l'
           aggiornamento dolce del coefficiente d'attrito.
72     struct {
73       std::mutex mutex;
74       double attrito;
```

```
75        double target_attrito;
76    } attrito_read{};
77    std::atomic_bool running_control{false};
78
79    // Porto a zero valori di aggiornamento attrito
80    if (attrito_read.mutex.try_lock()) {
81        attrito_read.attrito = 0.0;
82        attrito_read.target_attrito = 0.5;
83        attrito_read.mutex.unlock();
84    }
85    // Imposto filtro per variazione dolce attrito
86    constexpr double filter_gain{0.001};
87
88    int sockfd;
89    struct sockaddr_in servaddr;
90
91    // Creo variabile file descriptor per il socket
92    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
93        perror("socket creation failed");
94        exit(EXIT_FAILURE);
95    }
96
97    memset(&servaddr, 0, sizeof(servaddr));
98
99    // Informazioni del server (ROBOT)
100   servaddr.sin_family = AF_INET;
101   servaddr.sin_port = htons(PORT);
102   servaddr.sin_addr.s_addr = inet_addr("169.254.177.214");
103
104   // Avvio thread di comunicazione UDP
105   std::thread print_thread([print_rate, &print_data, &
          running, &sockfd, &servaddr]() {
```

```
106         while (running) {
107             // Gestisco la frequenza di comunicazione
108             std::this_thread::sleep_for(std::chrono::
                    milliseconds(static_cast<int>((1.0 /
                    print_rate * 1000.0))));
109
110             // Blocco i dati per evitare collisioni
111             if (print_data.mutex.try_lock()) {
112                 // Verifico la presenza di dati nuovi
113                 if (print_data.has_data) {
114                     // Formattazione dati per la creazione
                            dei pacchetti UDP
115                     std::bitset<SIZE> cut (std::string("
                            11111111111111111111111111111111"));
116                     std::bitset<SIZE> y;
117                     int n = 0;
118                     for(int i=0;i<7;i++){
119                         std::bitset<SIZE> a = print_data.
                                robot_state.tau_J[i] * 10000;
120                         a = a & cut;
121                         a <<= n;
122                         n += 32;
123                         y = y | a;
124                     }
125                     for(int i=0;i<7;i++){
126                         std::bitset<SIZE> a = print_data.
                                robot_state.q[i] * 10000;
127                         a = a & cut;
128                         a <<= n;
129                         n += 32;
130                         y = y | a;
131                     }
```

```
132                     for(int i=0;i<7;i++){
133                         std::bitset<SIZE> a = print_data.
                                gravity[i] * 10000;
134                         a = a & cut;
135                         a <<= n;
136                         n += 32;
137                         y = y | a;
138                     }
139                     for(int i=0;i<7;i++){
140                         std::bitset<SIZE> a = print_data.
                                tau_d[i] * 10000;
141                         a = a & cut;
142                         a <<= n;
143                         n += 32;
144                         y = y | a;
145                     }
146                     sendto(sockfd, &y, sizeof(y),
147                            MSG_CONFIRM, (const struct
                                sockaddr *) &servaddr,
148                            sizeof(servaddr));
149 
150                     print_data.has_data = false;
151                 }
152                 print_data.mutex.unlock();
153             }
154         }
155     });
156 
157     // Avvio thread di aggiornamento valore attrito
158     std::thread attrito_thread([&attrito_read, &running, &
            running_control]() {
159         while(running){
```

```
160         if(running_control) {
161            double aux;
162            // Richiedo all'utente la percentuale di attrito
                   desiderata
163            std::cout << "Inserire la percentuale di attrito
                   viscoso [0;1]" << std::endl;
164            std::cin >> aux;
165            // Mantengo l'input entro un intervallo
                   accettabile
166            if(aux < 0.1) aux = 0.1;
167            if(aux > 1) aux = 1;
168            // Blocco i dati per evitare collisioni
169            if (attrito_read.mutex.try_lock()) {
170              attrito_read.target_attrito = aux;
171              attrito_read.mutex.unlock();
172            }
173         }
174       }
175    });
176
177     // Parametri per la compliance nel controllo di
           impedenza
178     const double translational_stiffness{500.0};//200
179     const double rotational_stiffness{200.0};//20
180     Eigen::MatrixXd stiffness(6, 6), damping(6, 6);
181     stiffness.setZero();
182     stiffness.topLeftCorner(3, 3) << translational_stiffness
            * Eigen::MatrixXd::Identity(3, 3);
183     stiffness.bottomRightCorner(3, 3) <<
           rotational_stiffness * Eigen::MatrixXd::Identity(3,
           3);
184     damping.setZero();
```

```
185     damping.topLeftCorner(3, 3) << 2.0 * sqrt(
            translational_stiffness) *
186                                        Eigen::MatrixXd::
                                               Identity(3, 3);
187     damping.bottomRightCorner(3, 3) << 2.0 * sqrt(
            rotational_stiffness) *
188                                            Eigen::MatrixXd::
                                                   Identity(3,
                                                   3);
189
190     try {
191         // Connessione al robot
192         franka::Robot robot(fc_ip);
193         setDefaultBehavior(robot);
194         // Caricamento del modello cinematico e dinamico del
                Panda
195         franka::Model model = robot.loadModel();
196         // Imposto il carico a zero
197         robot.setLoad(0.0,{0,0,0},{0,0,0,0,0,0,0,0,0,0});
198
199         // Configuro i valori di collision detection
200         robot.setCollisionBehavior({{100.0, 100.0, 100.0,
                100.0, 100.0, 100.0, 100.0}},
201                                    {{100.0, 100.0, 100.0,
                                           100.0, 100.0, 100.0,
                                           100.0}},
202                                    {{100.0, 100.0, 100.0,
                                           100.0, 100.0,
                                           100.0}},
203                                    {{100.0, 100.0, 100.0,
                                           100.0, 100.0,
                                           100.0}});
```

```
204
205         // Variabili di supporto per la stima della massa
                del payload
206         int i_old = 0;
207         double tau0[LEN][7] = {};
208         double tau_diff[LEN*7] = {};
209         double hh[LEN] = {};
210         int cont = 0;
211         double time_max = 10.0;
212         double omega_max = 0.3;
213         double time = 0.0;
214         int i = 0;
215         int z = 0;
216         double stima_massa = 0;
217
218         std::array<double, 16> initial_pose;
219
220         // Definizione del controllo di forza
221         std::function<franka::Torques(const franka::
                RobotState&, franka::Duration)>
222         impedance_control_callback = [&](const franka::
                RobotState& robot_state,
223                                          franka::Duration
                                             period) ->
                                             franka::Torques
                                              {
224             // Tempo trascorso da inizio moto
225             time += period.toSec();
226             // Salvo posizione iniziale del robot
227             if (time == 0.0) {
228               initial_pose = robot_state.O_T_EE;
229             }
```

```
230
231             // Blocco i dati per evitare collisioni
232             if (attrito_read.mutex.try_lock()) {
233                 // Aggiorna dolcemente lo smorzamento fino
                        al raggiungimento del valore desiderato
234                 attrito_read.attrito = filter_gain *
                        attrito_read.target_attrito + (1 -
                        filter_gain) * attrito_read.attrito;
235                 attrito_read.mutex.unlock();
236             }
237
238             // Elaboro dati per la movimentazione
239             constexpr double kRadius = 0.1;
240           double angle = M_PI * (1 - std::cos(std::fmod(M_PI
                / 10.0 * time,M_PI)));
241           double delta_x = -kRadius * (std::cos(angle) - 1);
242           double delta_z = kRadius * std::sin(angle);
243             std::array<double, 16> new_pose = initial_pose;
244             new_pose[12] += delta_x;
245             new_pose[14] += delta_z;
246             // Creo variabili per moto del controllo di
                    Impedenza
247             Eigen::Affine3d initial_transform(Eigen::
                    Matrix4d::Map(new_pose.data()));
248             Eigen::Vector3d position_d(initial_transform.
                    translation());
249             Eigen::Quaterniond orientation_d(
                    initial_transform.linear());
250
251             // Ottengo variabili dello stato del robot
252             std::array<double, 7> coriolis_array = model.
                    coriolis(robot_state);
```

```
253             std::array<double, 7> gravity_array = model.
                    gravity(robot_state);
254             std::array<double, 42> jacobian_array =
255             model.zeroJacobian(franka::Frame::kEndEffector,
                    robot_state);
256             // Converto i vettori in matrici
257             Eigen::Map<const Eigen::Matrix<double, 7, 1> >
                    coriolis(coriolis_array.data());
258             Eigen::Map<const Eigen::Matrix<double, 6, 7> >
                    jacobian(jacobian_array.data());
259             Eigen::Map<const Eigen::Matrix<double, 7, 1> > q
                    (robot_state.q.data());
260             Eigen::Map<const Eigen::Matrix<double, 7, 1> >
                    dq(robot_state.dq.data());
261             Eigen::Affine3d transform(Eigen::Matrix4d::Map(
                    robot_state.O_T_EE.data()));
262             Eigen::Vector3d position(transform.translation()
                    );
263             Eigen::Quaterniond orientation(transform.linear
                    ());
264
265             Eigen::VectorXd tau_task(7), tau_d(7);
266
267             // Calcolo errore di posizione rispetto alla
                    posizione desiderata
268             Eigen::Matrix<double, 6, 1> error;
269             error.head(3) << position - position_d;
270             // Errore di orientazione
271             if (orientation_d.coeffs().dot(orientation.
                    coeffs()) < 0.0) {
272               orientation.coeffs() << -orientation.coeffs();
273             }
```

```
274             // Quaternione differenza
275             Eigen::Quaterniond error_quaternion(orientation.
                    inverse() * orientation_d);
276             error.tail(3) << error_quaternion.x(),
                    error_quaternion.y(), error_quaternion.z();
277             // Trasformo al sistema di riferimento base
278             error.tail(3) << -transform.linear() * error.
                    tail(3);
279
280             // Eseguo il controllo di impedenza
281             tau_task << jacobian.transpose() * (-stiffness *
                     error -attrito_read.attrito * damping * (
                    jacobian * dq));
282             tau_d << tau_task + coriolis;
283             std::array<double, 7> tau_d_array{};
284             Eigen::VectorXd::Map(&tau_d_array[0], 7) = tau_d
                    ;
285
286             // Aggiorno i dati per le comunicazioni UDP
287          if (print_data.mutex.try_lock()) {
288             print_data.has_data = true;
289             print_data.robot_state = robot_state;
290             print_data.gravity = gravity_array;
291             print_data.mutex.unlock();
292          }
293
294             return tau_d_array;
295         };
296
297         // Definizione della legge di moto per la stima
                della massa del payload
298         std::function<franka::JointVelocities(const franka::
```

```
RobotState&, franka::Duration)> motion = [&](
const franka::RobotState& robot_state, franka::
Duration period) -> franka::JointVelocities {
// Tempo trascorso da inizio moto
time += period.toSec();

// Processo dati per legge di moto
double cycle = std::floor(std::pow(-1.0, (time -
     std::fmod(time, time_max)) / time_max));
double omega = cycle * omega_max / 2.0 * (1.0 -
    std::cos(2.0 * M_PI / time_max * time));

franka::JointVelocities velocities = {{0, 0, 0,
    0, 0, 0, 0}};

// Fine moto
if (time >= 2 * time_max) {
    std::cout << std::endl << "Finished motion"
        << std::endl;
    return franka::MotionFinished(velocities);
}

// Dati dal robot
std::array<double, 7> gravity_array = model.
    gravity(robot_state);
Eigen::Matrix<double, 7, 1> tau(robot_state.
    tau_J.data());
Eigen::Matrix<double, 7, 1> q(robot_state.q.data
    ());

velocities = {{0, 0, 0, omega, 0, -omega, 0}};

```

```
321             // Raccolgo dati di coppia nella movimentazione
                    senza carico
322             if(z==0){
323                 for(int j=0;j<7;j++){
324                     tau0[i][j] = tau(j);
325                 }
326             // Calcolo coppia dovuta al carico
327             } else{
328                 if(i<i_old){
329                     for(int j=0;j<7;j++){
330                         tau_diff[7*i+j]= tau(j)-tau0[i][j];
331                         if(i==0)tau_diff[7*i+j] = 0;
332                     }
333                     hh[i] = 9.81*0.384*sin(q[3]);
334                     cont++;
335                 }
336
337                 // Aggiorno i dati per le comunicazioni UDP
338                 if (print_data.mutex.try_lock()) {
339                     print_data.has_data = true;
340                     print_data.robot_state = robot_state;
341                     print_data.gravity = gravity_array;
342                     std::copy_n(&tau_diff[7*i], 7,
                            print_data.tau_d.begin());
343                     print_data.mutex.unlock();
344                 }
345             }
346             i++;
347
348             return velocities;
349         };
350
```

```
351         for(z=0;z<2;z++){
352             // Muovo il robot nella configurazione iniziale
353             std::array<double, 7> q_goal = {{0, 0, 0, -3*
                    M_PI_4, 0, 3*M_PI_4, M_PI_4}};
354             MotionGenerator motion_generator(0.1, q_goal);
355             std::cout << "Press Enter to continue..." << std
                    ::endl;
356             std::cin.ignore();
357             robot.control(motion_generator);
358             std::cout << "Finished moving to joint
                    configuration." << std::endl;
359             if(z==0)std::cin.ignore();
360
361             time = 0.0;
362             i = 0;
363             // Avvio movimentazione eccitante per la stima
364             robot.control(motion);
365
366             i_old=i;
367
368             if(z==1){
369                 // Preparazione delle matrici per la stima
                        tramite minimi quadrati della massa del
                        payload
370                 Eigen::Map<Eigen::Matrix<double, LEN, 7,
                        Eigen::RowMajor> > tau_d_aux(tau_diff);
371                 Eigen::MatrixXd tau_d(LEN,7);
372                 tau_d = tau_d_aux.block(0,0,cont,7);
373                 Eigen::Map<Eigen::Matrix<double, LEN, 1> >
                        H_aux(hh);
374                 Eigen::MatrixXd H(LEN,1);
375                 H = H_aux.block(0,0,cont,1);
```

```
376                 Eigen::Map<Eigen::Matrix<double, LEN, 1> >
                        Y_aux(tau_d.col(5).data());
377                 Eigen::MatrixXd Y(LEN,1);
378                 Y = Y_aux.block(0,0,cont,1);
379                 Y = tau_d.col(5)-tau_d.col(3);
380
381                 Eigen::MatrixXd stima(1,1);
382                 // Stima ai minimi quadrati
383                 stima=(H.transpose()*H).inverse()*H.
                        transpose()*Y;
384                 stima_massa = stima(0);
385                 std::cout << "MASSA:" << stima_massa << std
                        ::endl << "----" << std::endl;
386             }
387
388         }
389
390         const std::array< double, 3 > l {0, 0, 0.03};
391         const std::array< double, 9 > in
                {0.03,0,0,0,0.03,0,0,0,0.03};
392
393         std::cout << "Press Enter to continue..." << std::
                endl;
394         std::cin.ignore();
395
396         if(stima_massa<0) stima_massa=0;
397
398         // Muovo il robot nella configurazione di partenza
                della legge di moto
399         std::array<double, 7> q_goal1 = {{0, -M_PI_4, 0, -3
                * M_PI_4, 0, M_PI_2, M_PI_4}};
400         MotionGenerator motion_generator1(0.1, q_goal1);
```

```
401         robot.control(motion_generator1);
402
403         // Impostazione della massa del carico
404         robot.setLoad(stima_massa,l,in);
405
406         time = 0.0,
407         running_control = true;
408         // Avvio controllo di impedenza
409         robot.control(impedance_control_callback);
410
411     } catch (const franka::Exception& ex) {
412         running = false;
413         std::cerr << ex.what() << std::endl;
414     }
415
416     if (print_thread.joinable()) {
417         print_thread.join();
418     }
419     if (attrito_thread.joinable()) {
420     attrito_thread.join();
421     }
422     close(sockfd);
423     std::cout << "Finished execution of program." << std::
            endl;
424     return 0;
425 }
```

B

SCRIPT MATLAB PER LA STIMA AI MINIMI QUADRATI DELLA MASSA DEL CARICO

Lo script MATLAB per la stima offline della massa del carico è riportata qui di seguito. Per il suo funzionamento si utilizzano i dati ricevuti tramite comunicazione UDP.

```
% massa effetiva del carico
payload=307;

% creazione vettore dei tempi
tout=[0:0.001:100]';
tout=tout(1:length(JointsPosition.signals(1).values));

% trova primo elemento diverso da zero e ultimo elemento utile
i = 1;
nontrovato = true;
while nontrovato
    if(diff(i,4)~=0)
        start=i;
        nontrovato = false;
    end
    i = i+1;
end

i = length(diff)-1;
nontrovato = true;
while nontrovato
```

```
22      if(diff(i,4)~=diff(i+1,4))
23          last=i
24          nontrovato = false;
25      end
26      i = i-1;
27  end
28
29  % Stima ai Minimi quadrati
30  H=9.81*0.384*sin(JointsPosition.signals(4).values(start:last
        ));
31  Y=mean(diff((start:last),6))-diff((start:last),4);
32  stima=double(inv(H'*H)*H'*Y)*1000
33  errore=stima-payload;
34
35  % plot stima peso
36  figure;
37  plot(tout(start:last),Y./H,'LineWidth',2);
38  hold on;
39  plot([tout(start) tout(last)],[stima stima]./1000,'LineWidth
        ',2);
```

BIBLIOGRAFIA

[1] G. Ferretti, G. Magnani, P. Rocco, F. Cecconello e G. Rossetti. "Impedance control for industrial robots". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 4. Apr. 2000, 4027–4032 vol.4. DOI: 10.1109/ROBOT.2000.845359.

[2] N. Hogan. "Impedance Control: An Approach to Manipulation". In: *1984 American Control Conference*. Giu. 1984, pp. 304–313. DOI: 10.23919/ACC.1984.4788393.

[3] G. J. G. Lahr, J. V. R. Soares, H. B. Garcia, A. A. G. Siqueira e G. A. P. Caurin. "Understanding the Implementation of Impedance Control in Industrial Robots". In: *2016 XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium (LARS/SBR)*. Ott. 2016, pp. 269–274. DOI: 10.1109/LARS-SBR.2016.52.

[4] R. Ikeura e H. Inooka. "Variable impedance control of a robot for cooperation with a human". In: *Proceedings of 1995 IEEE International Conference on Robotics and Automation*. Vol. 3. Mag. 1995, 3097–3102 vol.3. DOI: 10.1109/ROBOT.1995.525725.

[5] L. Bohan, Z. Yue, Z. Xinying e Z. Guangshang. "Research on Position-Based Impedance Control in Cartesian Space of Robot Manipulators". In: *2019 2nd World Conference on Mechanical Engineering and Intelligent Manufacturing (WC-*

MEIM). Nov. 2019, pp. 549–551. DOI: 10.1109/WCMEIM48965.2019.00117.

[6] P. Song, Y. Yu e X. Zhang. "Impedance Control of Robots: An Overview". In: *2017 2nd International Conference on Cybernetics, Robotics and Control (CRC)*. Lug. 2017, pp. 51–55. DOI: 10.1109/CRC.2017.20.

[7] P. M. Kebria, A. Khosravi, S. Nahavandi, A. Homaifar e M. Saif. "Experimental Comparison Study on Joint and Cartesian Space Control Schemes for a Teleoperation System Under Time-Varying Delay". In: *2019 IEEE International Conference on Industrial Technology (ICIT)*. Feb. 2019, pp. 108–113. DOI: 10.1109/ICIT.2019.8755087.

[8] T. Yoshikawa. "Force control of robot manipulators". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. Vol. 1. Apr. 2000, 220–226 vol.1. DOI: 10.1109/ROBOT.2000.844062.

[9] O. Khatib e J. Burdick. "Motion and force control of robot manipulators". In: *Proceedings. 1986 IEEE International Conference on Robotics and Automation*. Vol. 3. Apr. 1986, pp. 1381–1386. DOI: 10.1109/ROBOT.1986.1087493.

[10] T. Yoshikawa. "Dynamic hybrid position/force control of robot manipulators–Description of hand constraints and calculation of joint driving force". In: *IEEE Journal on Robotics and Automation* 3.5 (ott. 1987), pp. 386–392. ISSN: 2374-8710. DOI: 10.1109/JRA.1987.1087120.

[11] Marc H. Raibert e John J. Craig. "Hybrid Position / Force Control of Manipulators 1". In: 2008.

[12] S. Lin e K. H. Yae. "Identification of Unknown Payload and Environmental Parameters for Robot Compliant Motion". In: *1992 American Control Conference*. Giu. 1992, pp. 2952–2956. DOI: 10.23919/ACC.1992.4792687.

[13] W. Khalil, M. Gautier e P. Lemoine. "Identification of the payload inertial parameters of industrial manipulators". In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. Apr. 2007, pp. 4943–4948. DOI: 10.1109/ROBOT.2007.364241.

[14] Christopher G. Atkeson, Chae H. An e John M. Hollerbach. "Estimation of Inertial Parameters of Manipulator Loads and Links". In: *The International Journal of Robotics Research* 5.3 (1986), pp. 101–119. DOI: 10.1177/027836498600500306. eprint: https://doi.org/10.1177/027836498600500306. URL: https://doi.org/10.1177/027836498600500306.

[15] Yunfei Dong, Tianyu Ren, Ken Chen e Dan Wu. "An efficient robot payload identification method for industrial application". In: *Industrial Robot: An International Journal* 45 (lug. 2018). DOI: 10.1108/IR-03-2018-0037.

[16] Anna Boido. *FISIOTERAPIA/ROBOTICA – La robotica: un innovativo strumento al servizio del fisioterapista e dei suoi interventi riabilitativi*. URL: http://www.riabilitazioneinfo.it/fisioterapia-e-robotica-la-robotica-un-innovativo-strumento-al-servizio-del-fisioterapista-e-dei-suoi-interventi-riabilitativi-di-anna-boido-dottore-in-fisioterapia/. (accessed: 11.07.2020).

[17] M. Keppler, D. Lakatos, A. Werner, F. Loeffl, C. Ott e A. Albu-Schäffer. "Visco-Elastic Structure Preserving Impedance (VESπ) Control for Compliantly Actuated Robo-

ts". In: *2018 European Control Conference (ECC)*. Giu. 2018, pp. 255–260. DOI: 10.23919/ECC.2018.8550200.

[18] M. Vasic e A. Billard. "Safety issues in human-robot interactions". In: *2013 IEEE International Conference on Robotics and Automation*. Mag. 2013, pp. 197–204. DOI: 10.1109/ICRA.2013.6630576.