POLITECNICO DI TORINO

**Corso di Laurea Magistrale in Ingegneria Meccanica**

Tesi di Laurea Magistrale

# Sviluppo di interfaccia uomo-macchina per telecontrollo di cobot attraverso sensori inerziali


**Relatori**
Prof. Stefano Paolo Pastorelli
Prof. Laura Gastaldi
Ing. Elisa Digo
Ing. Valerio Cornagliotto

**Candidato**
Giulio Manni

Ottobre 2022

# Abstract

La diffusione dell'Industria 4.0 ha dato enfasi al concetto di automazione, favorendo l'inserimento dei robot nell'ambiente di lavoro. Tra le molteplici applicazioni dei sistemi robotici, quella predominante è quella della manipolazione. Il fatto che un robot impiegato in questo contesto possa manipolare oggetti di forma diversa rende la sua programmazione più complessa. Una soluzione al problema può essere quella di creare dei team in cui umani e robot collaborino. In questo caso, i robot vengono definiti collaborativi o cobot. Attraverso la teleoperazione, le capacità cognitive dell'uomo possono essere sfruttate per prendere decisioni che sono particolarmente difficili da automatizzare. Tuttavia, è necessario sviluppare dei telecontrolli intuitivi che permettano anche a utenti non esperti di portare a termine le operazioni in maniera rapida. Una possibile opzione per realizzare un telecontrollo intuitivo è quella di tradurre la posa di parti del corpo dell'operatore in comandi per il robot. A questo proposito, i sensori inerziali rappresentano una soluzione efficace per l'analisi del movimento umano perché sono indossabili, economici e poco invasivi.

Questa tesi presenta lo sviluppo di un'interfaccia uomo-macchina che permette il telecontrollo di cobot attraverso l'utilizzo di sensori inerziali. Più nello specifico, l'obiettivo per il quale è stata sviluppata tale interfaccia è il controllo del robot in operazioni di manipolazione di oggetti con forme particolari. In questi casi, infatti, risulta più agevole sfruttare le capacità cognitive dell'uomo rispetto a una pre-programmazione. Inoltre, l'impiego di sensori inerziali è adeguato a sviluppare un telecontrollo in real-time, perché essi rappresentano un sistema di tracking sufficientemente preciso, portatile e poco invasivo. L'efficacia dell'interfaccia sviluppata è stata verificata tramite prove sperimentali effettuate nei laboratori del Politecnico di Torino durante le quali il sistema è stato utilizzato per effettuare delle operazioni di pick and place.

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

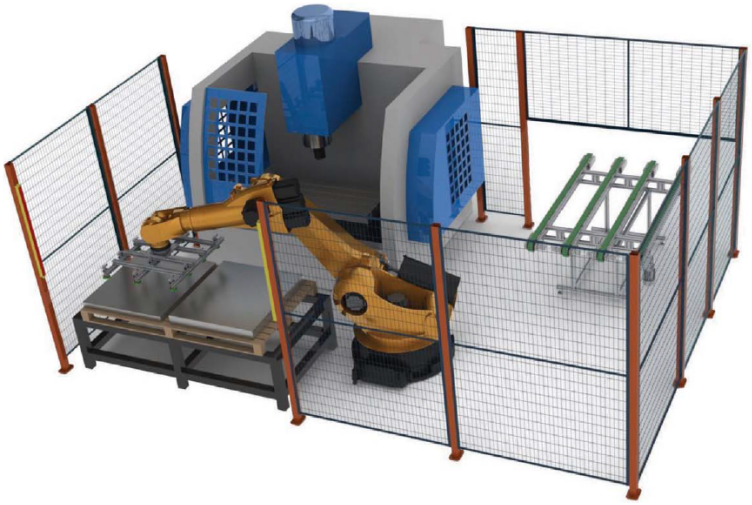
# Introduzione

## 1.1 Robotica collaborativa e teleoperazione

L'importanza dell'automazione e della robotica sta crescendo sempre di più nell'ambito della produzione industriale. Automatizzare i processi con l'utilizzo di macchine porta ad un aumento della produttività e dell'affidabilità oltre ad un maggior comfort per i lavoratori. Il concetto base che sta portando l'industria verso una maggiore automatizzazione è l'idea di permettere agli umani di focalizzare la loro attività in mansioni che richiedano delle capacità cognitive, lasciando che siano le macchine a svolgere le operazioni più ripetitive. Questo non significa però che umani e macchine debbano operare separatamente, al contrario il trend è quello di ideare sistemi nei quali ci sia collaborazione tra uomo e robot [1]. Si sviluppa così il concetto di robotica collaborativa, che prevede che macchina e uomo eseguano task in un ambiente condiviso e con diverse opzioni per il timing (in maniera continua, sincronizzata, alternata, etc..)

Per capire meglio il significato di robotica collaborativa, è importante analizzare più nel dettaglio il concetto di robotica industriale. Un robot industriale può essere definito come un device programmabile costituito da parti meccaniche, elettriche ed elettroniche in grado di eseguire complesse serie di azioni. I robot industriali tradizionali hanno tipicamente grandi dimensioni ed un peso elevato e vengono installati per eseguire operazioni che sarebbero difficoltose o pericolose per gli operatori umani come ad esempio la movimentazione di carichi pesanti. Solitamente i robot tradizionali sono ideati per applicazioni specifiche e isolati rispetto all'ambiente di lavoro umano. Inoltre, questa tipologia di robot ha un'importante limitazione dovuta a questioni di sicurezza. Infatti, nelle attuali applicazioni, l'operatore umano è tenuto lontano dall'area di lavoro del robot attraverso barriere fisiche con lo scopo di garantire la sicurezza dell'operatore stesso, come mostrato in Figura 1.1a Al contrario, i robot collaborativi sono robot progettati per lavorare fianco a fianco con la loro controparte umana e per condividere lo stesso spazio di lavoro e per questo sono chiamati "cobot". Le barriere sono eliminate e al loro posto sono adottati differenti meccanismi per impedire gli incidenti, come ad esempio la riduzione delle velocità di movimento, la riduzione delle forze applicate dal robot e l'utilizzo di sensori che permettono di localizzare l'operatore (Figura 1.1b). I cobot sono poi più leggeri rispetto ai robot industriali offrendo quindi una maggiore mobilità e la possibilità di essere facilmente spostati per essere utilizzati in ambienti differenti. Un altro vantaggio che i cobot presentano rispetto ai robot industriali tradizionali è la loro flessibilità: lo stesso robot può essere usato per eseguire un'ampia varietà di compiti e per questo può essere impiegato in un ampio range di settori industriali e non. In-

fine, essi sono facilmente programmabili e offrono una notevole capacità di calcolo, il che permette loro di lavorare in maniera efficiente e in sicurezza al fianco degli operatori umani [2] [3]. Tabella 1.1 riporta una sintesi delle differenze tra robot industriali tradizionali e collaborativi [4].



(a) Robot tradizionali: è necessario l'utilizzo di barriere per separare l'ambiente di lavoro umano da quello del robot.


(b) Robot collaborativi: permettono gli operatori umano di stare nello loro vicinanze e di lavorare allo stesso task in sicurezza.

Figura 1.1: Esempio di robot industriali tradizionali e collaborativi. [5]

Quindi, creare dei sistemi in cui robot e umani interagiscono e collaborano permette di conservare importanti caratteristiche dei robot industriali come la capacità di eseguire movimenti ripetuti e la possibilità di muovere e posizionare oggetti con grande precisione aggiungendo la capacità dell'uomo di adattarsi rapidamente a situazioni nuove e di supervisionare l'ambiente di lavoro condiviso. Tutto ciò permette di superare uno dei limiti dei sistemi di robotica industriale, quello di essere ideati e quindi pre-programmati per eseguire task specifici. I robot possono essere guidati dall'uomo quando ad esempio è necessario che essi eseguano delle operazioni che non erano state previste a priori.

Tabella 1.1: Confronto tra robot industriali tradizionali e robot collaborativi [5].

| **Robot industriali tradizionali** | **Robot industriali collaborativi** |
|---|---|
| Installazione fissa | Facilmente rilocabili |
| Rari cambi di task | Frequenti cambi di task |
| Interazione con gli operatori solo in fase di programmzione | Frequenti interazioni con gli operatori |
| Programmazione lead-through o off-line | Programmazione on-line o off-line e utilizzo in sistemi integrati |
| Operatori e robot sono separati da barriere | Spazio di lavoro condiviso |
| Impossibilità di interagire in sicurezza con le persone | Interazione sicura con le persone |
| Profittevole sono in produzioni su larga scala | Profittevole anche in piccole produzioni |
| Piccoli o grandi e molto veloci | Piccoli, lenti e facili da usare e da muovere |

## 1.2 Obiettivo

Attualmente, un robot impegnato nella manipolazione di oggetti può lavorare in scenari molto differenti tra loro e manipolare oggetti di forma diversa. Tutto ciò può rendere la sua programmazione molto complessa. Una soluzione consiste nel creare dei team in cui umani e robot collaborino.

Con teleoperazione si intende l'esecuzione di un'operazione a distanza e con la parola "distanza" ci si può riferire ad una distanza fisica che separa l'operatore dal robot ma anche ad una differenza di scala, quando ad esempio un chirurgo usa tecnologie di robotica per eseguire un'operazione a livello microscopico. Attraverso la teleoperazione le capacità cognitive dell'uomo possono essere sfruttate per prendere delle decisioni che sono particolarmente difficili da automatizzare. La teleoperazione, indipendentemente dalla scopo per cui viene utilizzata, presuppone l'uso di una tecnologia che permetta all'operatore umano di controllare il robot da remoto [6]. Risulta però irragionevole aspettarsi che gli operatori che guidano i robot siano degli esperti di robotica. Di conseguenza, è necessario sviluppare dei telecontrolli intuitivi che permettano ad utenti non esperti di portare a termine le operazioni in maniera rapida [7]. Una possibile soluzione consiste nell'utilizzare la posa di parti del corpo dell'operatore come comandi per il robot [8]. Ciò permette all'operatore di trasmettere al robot il movimento che vuole eseguire in maniera intuitiva e in real-time.

Al fine di generare dei comandi in input, la posizione o l'orientazione del corpo dell'operatore deve essere rilevata con l'utilizzo di appositi sensori. Questo può essere fatto usando diversi sistemi, tra cui quelli di visione, meccanici o inerziali.

Negli ultimi anni, una nuova generazione di sensori inerziali basata su micro sistemi elettro - meccanici ha dato un forte impulso agli studi di motion tracking. In particolare, è possibile trovare sensori inerziali a 6 gradi di libertà dotati di un accelerometro e un giroscopio o a 9 gradi di libertà quando viene aggiunto un magnetometro (in questo caso si parla di MIMU). Il grande interesse per questo tipo

di dispositivi è principalmente motivato dal fatto che essi permettano di superare alcuni problemi che invece si presentano utilizzando sistemi ottici o tracker di tipo meccanico, garantendo comunque dei costi contenuti. Gli IMU, infatti, rispetto ai sistemi ottici non presentano il problema dell'occlusione e permettono, in via teorica, una spazio di lavoro infinito. Rispetto si sistemi meccanici, invece, sono più economici e meno intrusivi, pagando però in accuratezza [9]. Per queste ragioni sono ampiamente utilizzati per creare tool indossabili per il tracciamento del movimento. Esistono tuttavia anche delle limitazioni legate a questa tipologia di sensori. Il loro principale svantaggio è dovuto al fatto che per ottenere una misura della posizione da un IMU è necessario integrare i dati di accelerazione, velocità angolare e campo magnetico direttamente misurati dal sensore, ma quest'operazione chiamta sensor fusion è numericamente instabile e produce un consistente drift. Inoltre, i MIMU, se utilizzati in contesto industriale, hanno un'ulteriore limitazione: le letture del magnetometro, che consentono una stima accurata dell'orientazione, sono negativamente influenzate da disturbi ferro-magnetici provenienti dagli altri dispositivi installati all'interno dell'ambiente di lavoro. Per questi motivi, se non utilizzati in concomitanza con altri tipi di sensori, essi non sempre permettono di raggiungere il livello di accuratezza necessario nei sistemi di robotica collaborativa. Nonostante questa limitazione, rappresentano comunque una soluzione interessante per il motion tracking.

L'obiettivo dell'attività di tesi è stato quello di sviluppare un sistema di telecontrollo basato sull'utilizzo di IMU. Nello specifico, questo sistema riesce a controllare un robot collaborativo durante operazioni di pick and place di oggetti poliedrici o comunque con forme differenti, operazioni per le quali risulta più agevole sfruttare le capacità cognitive umane rispetto ad una pre-programmazione. Il sistema è composto da un sensore inerziale posizionato sul dorso della mano dell'operatore capace di fornire in real-time informazioni di orientazione ad un computer centrale. All'interno del computer, un ambiente ROS interpreta le informazioni provenienti dal sensore e le trasforma in comandi per un cobot. Durante lo sviluppo di questo sistema, particolare attenzione è stata posta verso problematiche legate a ritardi e vibrazioni, che possono nascere durante i telecontrolli di robot in real-time e che sono state messe in evidenza da diversi autori [10] [11].

## 1.3 Struttura della tesi

I restanti capitoli di questa tesi sono organizzati come segue. Nel Capitolo 2 viene presentata la strumentazione utilizzata. Vengono introdotti i sensori inerziali descrivendone le specifiche tecniche ed analizzando in maniera approfondita le modalità di interfaccia con i sensori e le tipologie di dati che si possono ricevere. Successivamente è introdotto il robot collaborativo utilizzato, un UR3 prodotto da Universal Robot, descrivendone l'hardware e le differenti modalità di programmazione. Nel dettaglio, sono analizzate le interfacce che permettono la comunicazione in real-time tra il robot ed un computer esterno e la modalità di controllo del robot usata per realizzare l'applicazione finale. Il capitolo prosegue con una presentazione del 2F-85 Gripper, la pinza per robot utilizzata durante l'attività sperimentale, e si conclude con una breve descrizione di ROS, il meta-sistema operativo usato per sviluppare l'ambiente software che gestisce il telecontrollo.

Nel Capitolo 3 vengono descritti l'applicazione finale e gli algoritmi sviluppati. Vengono analizzati nel dettaglio, riportando i risultati di prove sperimentali, i

comandi move, speed e servo, comandi URScript (linguaggio di programmazione sviluppato da Universal Robot) che permettono di muovere il robot. Viene quindi presentato il telecontrollo realizzato, descrivendone la sua modalità di utilizzo e l'ambiente ROS che lo gestisce.

Nel Capitolo 4 è riportato un test effettuato per verificare il corretto funzionamento del telecontrollo. Il capitolo contiene una descrizione dell'ambiente nel quale si trova il robot durante il test, delle varie componenti hardware presenti, delle impostazioni software e dello scopo della prova. Viene anche mostrata l'esecuzione dell'operazione di pick and place attraverso dei frame video catturati durante il test.

Nel Capitolo 5 vengono presentate le conclusioni relative al lavoro svolto, delineando anche alcune potenziali applicazioni del sistema realizzato ed alcuni possibili sviluppi futuri.

# Capitolo 2

# Strumentazione e software

## 2.1 Sensori Inerziali (IMU)

Un'unità di misura inerziale o Inertial Measurement Units (IMU) è un dispositivo elettronico equipaggiato con un accelerometro e un giroscopio triassiali. Quando è presente anche un magnetometro, si parla di MIMU.

I sensori inerziali analizzati durante lo sviluppo di questa tesi sono due: gli Xsens - MTx e gli APDM - Opal.

### 2.1.1 Xsens - MTx

I Motion Trackers (MTx) sono IMU prodotti da Xsens, in grado di fornire in tempo reale accelerazioni lineari e velocità angolari lungo tre assi. Inoltre, questi sensori sono equipaggiati di un magnetometro 3D e di un processore integrato in grado di calcolare l'orientazione del sensore stesso. Si tratta di sensori indossabili molto usati per la biomeccanica e la realtà virtuale, che permettono un'accurata misurazione dell'orientazione dei segmenti del corpo umano.

Durante le attività svolte nell'ambito di questa tesi è stato utilizzato un kit di Xsens chiamato Xbus Kit e contenente: lo Xbus Master, 7 MTx e i cavi necessari all'alimentazione e alla trasmissione dei dati. L'Xbus Master è una centralina che garantisce la comunicazione tra IMU e PC. Infatti, fornisce agli IMU potenza elettrica e riceve i dati mentre vengono misurati inviandoli al PC via USB o Bluetooth. Come mostrato in Figura 2.1, è possibile creare una catena di IMU collegati tramite cavi connessa all'Xbus Master. La figura mostra la comunicazione via USB tra Xbus Master e PC.

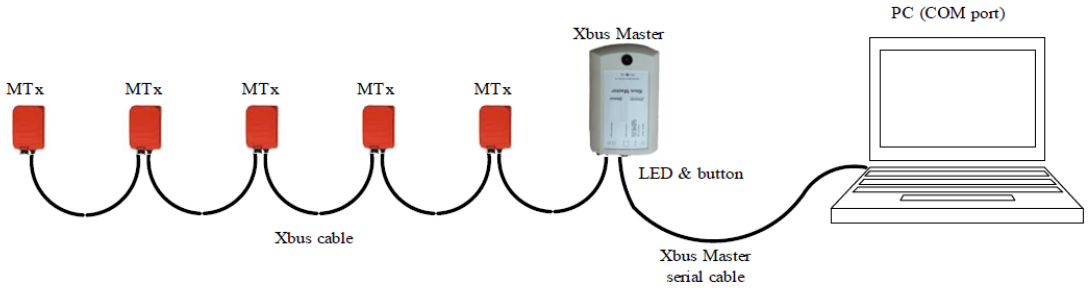



Figura 2.1: Xbus Kit e collegamenti tra i vari componenti

#### 2.1.1.1 Output degli MTx

Le due principali tipologie di dati che possono essere letti dagli MTx sono: *Calibrated data* e *Orientation data*. In questo paragrafo i due output vengono analizzati separatamente, ma è possibile configurare i sensori in modo da ricevere dei pacchetti di dati costituiti da una combinazione dei due.

**Calibrated data**

Le letture dei calibrated data, accelerazioni, velocità angolari e campo magnetico terrestre, vengono fornite rispetto al sistema di riferimento solidale al sensore e allineato con il case esterno degli MTx e riportato in Figura 2.2.

I sensori fisici all'interno degli MTx (accelerometri, giroscopi e magnetometri) sono calibrati dalla casa produttrice secondo un modello fisico di risposta dei sensori alla misura delle varie grandezze, in modo da fornire appunto dei calibrated data. Maggiori informazioni inerenti al modello fisico di risposta possono essere trovate nella documentazione tecnica dei sensori [12]. É comunque possibile settare i sensori in modo da ricevere dati grezzi non processati (un-calibrated data). Le specifiche relative a questa tipologia di output sono riportate nelle Tabelle 2.1 e 2.2. La massima frequenza di campionamento, per una comunicazione USB, è uguale a 100 Hz quando un solo MTx è collegato, si riduce a 50 Hz se gli MTx collegati sono da 2 a 5, mentre è pari a 25 Hz con 10 MTx collegati. Informazioni più dettagliate possono essere trovate in [12].

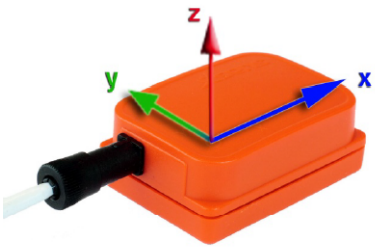



Figura 2.2: Sistema di riferimento locale dell'MTx.

Tabella 2.1: Specifiche Calibrated data. Queste specifiche sono valide per MTx con configurazione standard.

| | ACCELEROMETRO | GIROSCOPIO | MAGNETOMETRO |
|---|---|---|---|
| **Assi** | 3 assi | 3 assi | 3 assi |
| **Range** | ±5 g | ±1200 deg/s | ±750 mGauss |
| **Noise** | 0.002 $m/s^2/\sqrt{Hz}$ | 0.1°$/s/\sqrt{Hz}$ | 0.5 $mGauss/\sqrt{Hz}$ |
| **Frequenza** | fino a 100 Hz | fino a 100 Hz | fino a 100 Hz |
| **Bandwidth** | 30 Hz | 40 Hz | 10 Hz |
| **Risoluzione A/D** | 16 bits | 16 bits | 16 bits |

Tabella 2.2: Specifiche custom disponibili per gli MTx (configurazioni standard in grassetto).

| ACCELEROMETRO | variazione delle specifiche |
|---|---|
| **±5 g** | nessuna, vedi Tabella 2.1 |
| ±1.7 g | nessuna, vedi Tabella 2.1 |
| ±18 g | Noise: 0.004 $m/s^2/\sqrt{Hz}$ |
| **GIROSCOPIO** | **variazione delle specifiche** |
| **±1200 deg/s** | nessuna, vedi Tabella 2.1 |
| ±150 deg/s | Noise: 0.04°$/s/\sqrt{Hz}$ |

**Orientation data**
Il secondo output degli MTx consiste nell'orientazione del sistema di riferimento a loro solidale, indicato in Figura 2.3 con **S**, rispetto al sistema di riferimento terrestre, indicato con **G**. Di default, il sistema di riferimento terrestre G è definito come segue:

- X positiva nella direzione del Nord magnetico.
- Y perpendicolare a X e diretta verso Ovest.
- Z perpendicolare al piano XY e positiva secondo la regola della mano destra.

L'orientazione del sistema di riferimento **S** rispetto **G** può essere espressa secondo differenti modalità:

- Quaternioni
- Angoli di Eulero: roll, pitch, yaw (secondo la sequenza di Cardano)
- Matrice di rotazione (matrice dei coseni direttori)

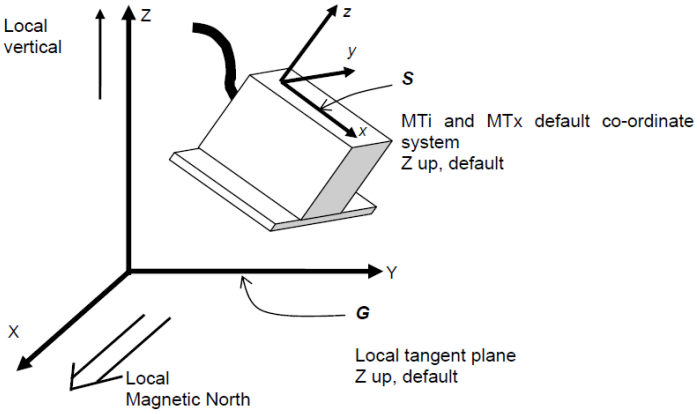



Figura 2.3: Orientamento dell'MTx nel sistema di riferimento terrestre.

In Tabella 2.3 sono riportate le specifiche relative agli orientation data. É possibile campionare a 120 Hz con un massimo di 5 sensori collegati, mentre la frequenza si abbassa a 64 Hz quando gli MTx sono 10.

Tabella 2.3: Specifiche orientation data.

| | |
|---|---|
| **Risoluzione angolare** | 0.05° |
| **Ripetibilità** | 0.2° |
| **Accuratezza statica (roll/pitch)** | 0.5° |
| **Accuratezza statica (hending)** | 1.0° |
| **Accuratezza dinamica** | 2° |
| **Frequenza** | max 120 Hz |

#### 2.1.1.2 Comunicazione con Xsens

Esistono tre tipi di interfacce con i sensori:

- MT Manager, GUI
- MT Software Development Kit, API
- MT Low-level Communication Protocol, comunicazione diretta

MT Manager è un software per piattaforme Windows. Questa GUI permette esclusivamente di visualizzare i dati misurati dagli IMU (presentati nel paragrafo 2.1.1.1) in real-time e di esportare dei record per un successivo post-processing. Non è possibile attraverso MT Manager processare i dati in tempo reale.

MT Software Development Kit fornisce una COM-object API e una DLL API. Si tratta di API che premettono di creare delle interfacce real-time con i sensori. La COM-object è stata creata per dare la possibilità agli sviluppatori di creare delle applicazioni in software come MATLAB, LabVIEW, Excel (Visual Basic), etc., mentre è possibile usare la DLL quando si programma in C, C++ o altri linguaggi di programmazione in quanto permette l'accesso diretto al codice sorgente. Le analisi effettuate hanno però evidenziato che questo metodo di comunicazione è ormai obsoleto. Sia MT Manager che le API fornite nel SDK sono delle interfacce di alto livello basate sul Low-level Communication Protocol.

MT Low-level Communication Protocol è un protocollo di comunicazione dati di basso livello attraverso una porta seriale sviluppato da Xsens. Si tratta della scelta migliore nel caso in cui si voglia pieno controllo, massima flessibilità e si abbia una forte esigenza di performance in real-time. Per questi motivi il Low-level Communication Protocol è stato analizzato in dettaglio al fine di creare dei codici matlab che permettessero di interfacciarsi con gli IMU.

##### 2.1.1.2.1 MT Low-Level Communication Protocol

Il Low-Level Communication Protocol sviluppato da Xsens per i sensori inerziali e basato sull'invio e la ricezione di messaggi standardizzati utilizza una comunicazione binaria attraverso una porta USB.

Figura 2.4 riporta i differenti livelli di comunicazione con l'hardware. Il Low-Level Communication Protocol rappresenta il livello più basso di comunicazione con i sensori, infatti le API e la GUI fornite da XSens sono state sviluppate sulla base di questo protocollo. Questa metodologia di comunicazione permette l'accesso a tutte le funzionalità degli MTx, ed inoltre presenta il grande vantaggio di poter essere utilizzata con qualsiasi linguaggio di programmazione.

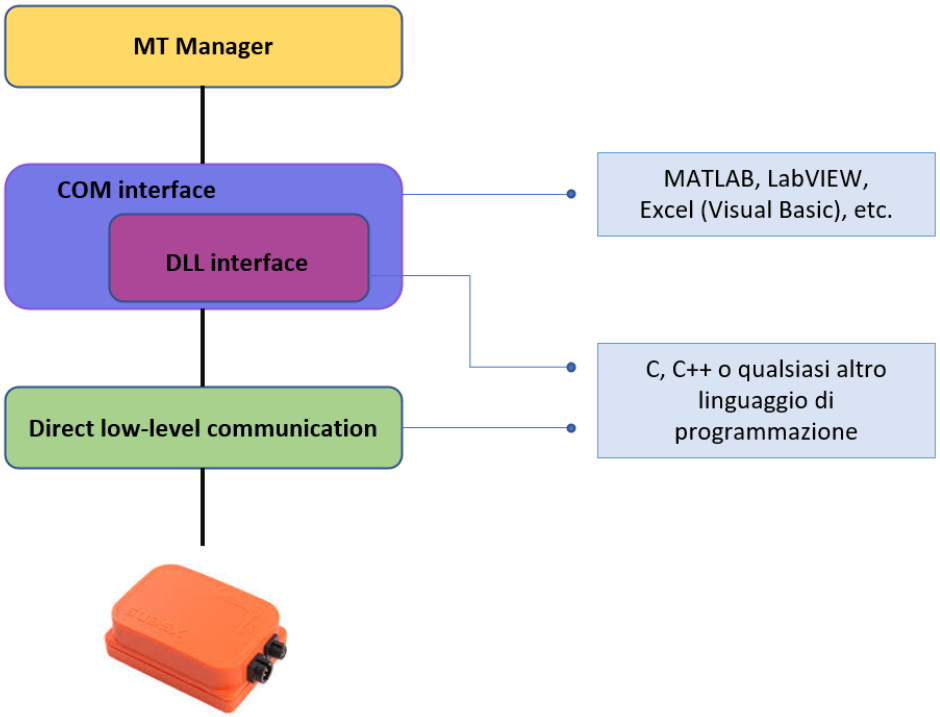



Figura 2.4: Livelli di comunicazione con gli MTx.

Qualsiasi configurazione dei sensori può essere modificata utilizzando il communication protocol, agendo su frequenza, baud rate e output mode. I differenti output mode permetto all'utente di ricevere la tipologia di dati in output che preferisce.

Durante il loro funzionamento, gli MTx possono trovarsi in due diversi stati (Figura 2.5). Le configurazioni possono essere modificate in quello che è chiamato "Config State". In questo stato gli MT accettano messaggi che settano gli output mode o altri setting. Una volta impostata la configurazione che si preferisce l'utente può portare gli MT nel "Measurement State". In questo stato i sensori iniziano ad inviare messaggi contenenti i dati misurati in base alla configurazione corrente.

É possibile passare da Config a Measurement e viceversa utilizzando i messaggi `GoToConfig` e `GoToMeasurement` come riportato in Figura 2.5.

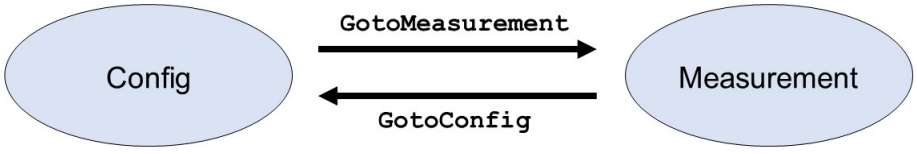



Figura 2.5: MT state

Come riportato precedentemente, la comunicazione con gli MT avviene attraverso lo scambio di messaggi, che sono realizzati secondo una struttura standard riportata in Figura 2.6.


Figura 2.6: Struttura standard dei messaggi del Low-Level Comunication Protocol

Di seguito è riportata una breve spiegazione sulla formulazione, in esadecimale, dei messaggi di questo protocollo.

La lunghezza massima dei messaggi è di 254 byte e, come si può osservare in Figura 2.6, ogni messaggio è costituito dai seguenti campi:

- Preamble (1 byte), ogni messaggio inizia con lo stesso Preamble il cui valore è 0xFA.
- BID o Address (1 byte), i messaggi inviati/ricevuti dal master hanno BID pari a 0xFF, mentre l'indirizzo degli MT parte da 0x01 e cresce per ogni senore collegato al master fino a 0x06.
- MID (1 byte), il MID identifica il tipo di messaggio, ad esempio `GoToConfig` ha MID pari a 0x30. La lista completa dei messaggi e i relativi MID è riportata in MT Low-Level Communication Protocol Documentation [13].
- LEN (1 byte), specifica il numero di byte di cui è composto il campo DATA del messaggio.
- DATA (0-254 byte), contiene i dati del messaggio. L'interpretazione dei dati è specifica del messaggio che stiamo leggendo e dipende dal MID.
- Checksum (1 byte), questo campo è utilizzato per identificare errori nella scrittura del messaggio. La tipologia di checksum utilizzata dal communication protocol e la 2's complement 8 bit checksum.

Generalmente, inviando un messaggio con un certo MID si riceve dal sensore un messaggio con MID incrementato di uno. Questo messaggio nella documentazione Xsens viene chiamato "acknowledge message". In base alla tipologia di messaggio l'acknowledge message può avere un campo DATA o può non averlo.

ESEMPIO - Setting dell'output mode in matrice di orientazione
Messaggio inviato:
`GoToConfig` = FA FF 30 00 D1 (esadecimale)
Messaggio ricevuto (Acknowledge):
`GoToConfigAck` = FA FF 31 00 D0 (esadecimale)
Messaggio inviato:
`SetOutputMode` = FA 01 D2 04 00 00 00 08 21 (esadecimale)
Messaggio ricevuto (Acknowledge):
`SetOutputModeAck` = FA 01 D3 00 2C (esadecimale)

Nell'esempio è riportata la sequenza di messaggi che deve essere inviata al fine di impostare l'output mode. É utile osservare che, prima del messaggio corrispondente a `SetOutputMode`, bisogna inviare il comando `GoToConfig`, perché i sensori normalmente si trovano in Measurement State mentre qualsiasi configurazione può essere eseguita soltanto in Config State.

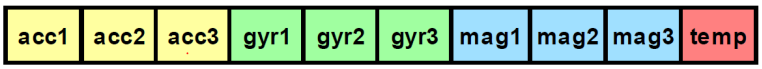
Un altro aspetto rilevante dell'esempio riportato in precedenza è la differenza di Addres tra i vari messaggi. `GoToConfig` ha come Addres il valore FF che corrisponde all'indirizzo del master. Infatti, messaggi come `GoToConfig` o `GoToMeasurement` vanno inviati al master. Altre tipologie di messaggi possono essere inviate sia al master che ai singoli device. Nell'esempio `SetOutputMode` viene mandato all'MT con indirizzo 01.

Considerando la struttura del messaggio che si riceve dal Master durante lo streaming, se i sensori si trovano in Config state, per avviare la lettura dei dati è necessario inviare il comando `GoToMeasurement`. Quando gli MT sono in Measurement state, inviano alla frequenza impostata dall'utente un messaggio contenente le misurazioni effettuate. Tale messaggio viene chiamato "`MTData`". La struttura del `MTData` è la seguente:

| | Preamble | BID | MID | LEN | DATA | Checksum |
|---|---|---|---|---|---|---|
| `MTData` = | FA | FF | 50 | XX | [...] | XX |

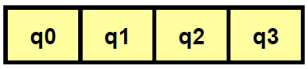
Il campo DATA e la sua lunghezza LEN dipendono dall'output mode. Le differenti formattazioni del campo DATA sono riportate di seguito.

**Un-calibrated raw data output mode (20 bytes)**
Contiene i dati grezzi di accelerazione, velocità angolare e campo magnetico negli assi X, Y, Z. Questi valori corrispondono alle letture del convertitore analogico-digitale dei sensori. I valori sono forniti come 16 bit unsigned integer.




**Calibrated data output mode (36 bytes)**
Contiene i dati calibrati di accelerazione, velocità angolare e campo magnetico negli assi X, Y, Z. I valori sono forniti come float.

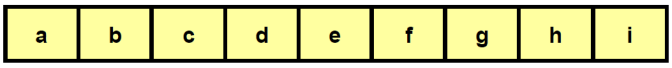

**Orientation data output mode – quaternion (16 bytes)**
Contiene i quaternioni q0, q1, q2, q3 che rappresentano l'orientazione dell'MTx. I valori sono forniti come float.




**Orientation data output mode – Euler angles (12 bytes)**
Contiene i tre angoli di Eulero che rappresentano l'orientazione dell'MTx. I valori sono forniti come float.


**Orientation data output mode – Matrix (36 bytes)**
Contiene gli elementi della matrice di rotazione che rappresentana l'orientazione dell'MTx. I valori sono forniti come float.

Nel caso in cui un unico MT è collegato al master, il campo DATA è formato come riportato sopra. Se più di un sensore è collegato allora la struttura dei dati si




$$R_{OS} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

| | | | | LEN | DATA | | Checksum |
|---|---|---|---|---|---|---|---|
| `MTData` = | FA | FF | 50 | 18 | \| roll pitch yaw \| MT 01 | roll pitch yaw \| MT 02 | XX |

ripete tante volte quanti sono gli MT collegati. Ad esempio, supponendo di leggere dati di orientazione da due IMU, il messaggio `MTData` è strutturato come segue.

Maggiori informazioni riguardo le modalità di comunicazione attraverso il Low-Level Communication Protocol sono contenute nella documentazione Xsens [13], mentre un codice di esempio che realizza uno streaming dati dai sensori utilizzando il Low Level Comunication Protocol è riportato in Appendice A.

### 2.1.2 APDM - Opal

Gli Opal sono IMU prodotti da APDM. Si tratta di sensori indossabili, wireless, con dimensioni e peso ridotti ed un'elevata autonomia, come è possibile notare in Tabella 2.4 in cui sono riportate le caratteristiche hardware di questi sensori. Sono equipaggiati di due accelerometri, un giroscopio, un magnetometro e un sensore di temperatura. Permettono quindi di misurare accelerazioni lineari, velocità angolari e campo magnetico di un sistema di riferimento posizionato al centro del monitor e orientato come in Figura 2.7 con una frequenza massima di 128 Hz.

Il kit utilizzato durante lo svolgimento delle attività di tesi è composto da un access point, una docking station, 5 opal e i cavi micro usb necessari all'alimentazione e alla trasmissione dati. L'access point permette la comunicazione wireless tra il PC a cui è collegato e gli Opal, mentre la docking station è utilizzata per configurare i sensori, per la ricarica e per effettuare il download dei dati. In Figura 2.8 è riportato il sistema nella sua configurazione di utilizzo.

Tabella 2.4: APDM - Opal caratteristiche hardware

| | |
|---|---|
| **Dimensioni** | 43.7 x 39.7 x 13.7 mm |
| **Peso** | ¡ 25 grammi |
| **Memoria** | 8 Gb ( 450h) |
| **Durata Batteria** | Logging: 16h, Streaming: 8h |

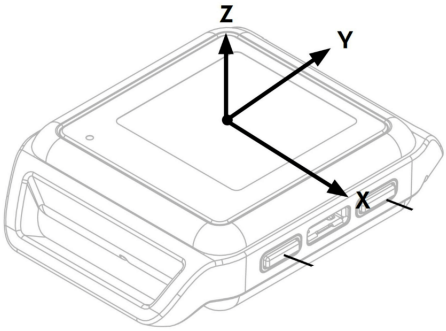



Figura 2.7: Sistema di riferimento del sensore Opal

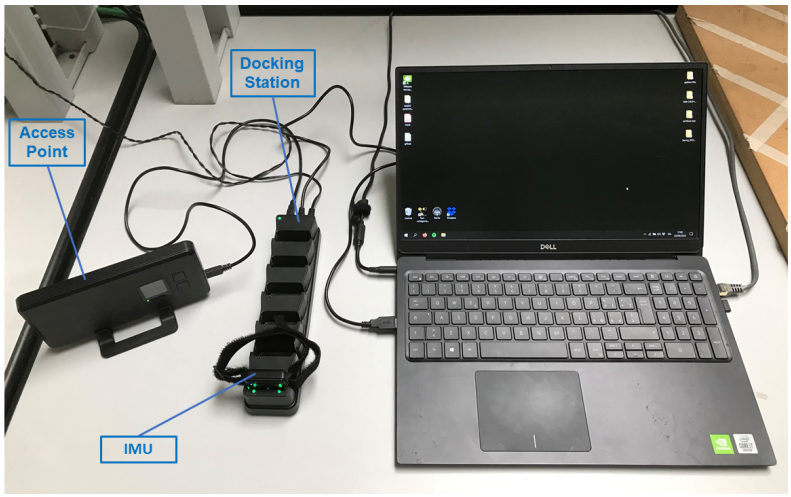



Figura 2.8: Sistema Opal: Access Point, Docking Station e sensore inerziale

#### 2.1.2.1 Output degli Opal

Come per gli MTx, le tipologie di dati che si possono leggere dagli Opal sono due: i dati di orientazione e i valori di accelerazione, velocità angolare e campo magnetico misurati rispetto il sistema di riferimento indicato in Figura 2.7.

I dati di orientazione provengono da un'elaborazione, eseguita dall'elettronica dell'Opal, delle misure fatte da i due accelerometri, il giroscopio e il magnetometro che equipaggiano il sensore. L'orientazione del sensore è fornita sotto forma di quaternioni. Per quanto riguarda i dati di accelerazione, velocità angolare e campo magnetico, è possibile ricevere i dati grezzi oppure è possibile richiedere al sensore dati elaborati dalla sua elettronica interna.

In Tabella 2.5 e 2.6 sono riportate le specifiche degli APDM - Opal.

Tabella 2.5: Specifiche Opal per letture di accelerazioni lineari, velocità angolari e campo magnetico.

Tabella 2.6: Specifiche Opal per letture di orientazione

| | |
|---|---|
| **Accuratezza statica (Roll/Pitch)** | 1.15 deg |
| **Accuratezza statica (Heading)** | 1.50 deg |
| **Accuratezza dinamica** | 2.80 deg |

| | ACCELEROMETRI (2) | GIROSCOPIO |
|---|---|---|
| **Assi** | 3 assi | 3 assi |
| **Range** | ±16g, ±200g | ±2000 deg/s |
| **Noise** | 120 $\mu g/\sqrt{Hz}$, 5 $\mu g/\sqrt{Hz}$ | 0.025 $deg/s/\sqrt{Hz}$ |
| **Frequenza** | da 20 a 128 Hz | da 20 a 128 Hz |
| **Bandwidth** | 50 Hz | 50 Hz |
| **Risoluzione A/D** | 14 bits, 17.5 bits | 16 bits |

| MAGNETOMETRO | |
|---|---|
| **Assi** | 3 assi |
| **Range** | ±Gauss |
| **Noise** | 2 $mGauss/\sqrt{Hz}$ |
| **Frequenza** | da 20 a 128 Hz |
| **Bandwidth** | 50 Hz |
| **Risoluzione A/D** | 12 bits |

#### 2.1.2.2 Comunicazione con gli Opal

La comunicazione con gli Opal può avvenire in due differenti modalità, le GUI sviluppate dalla casa produttrice e l'APDM Software Development Kit.

Le interfacce grafiche fornite da APDM (es. Motion Studio), permettono di configurare facilmente i sensori, visualizzare in real-time i dati raccolti e creare dei record delle sessioni di misura. Non è possibile attraverso questi software accedere ai dati in real-time.

L'APDM Software Development Kit (SDK) mette a disposizione la stessa interfaccia di basso livello con l'hardware su cui è basato Motion Studio. Fornisce quindi dei tool per realizzare applicazioni in Java, C o Python. Tali strumenti permettono agli sviluppatori di scrivere programmi per la configurazione e lo streaming dati dai sensori. Nel caso di questa tesi, l'esigenza di elaborare i dati in real-time ha reso necessario l'utilizzo dell'SDK. Alcuni file elaborati sulla base di questo strumento sono riportati nell'Appendice B.

Il Software Development Kit è strutturato come riportato in Tabella 2.7.

Tabella 2.7: Struttura del Solftware Developement Kit.

| | |
|---|---|
| /doc | Documentazione per gli utenti: Developers guide e descrizione dell'API |
| /include | Header da includere in applicazioni C o C++ |
| /libs | Librerie dinamiche per sistemi operaivi Windows, MacOS e Linux da importare nelle applicazioni in modo da abilitare l'accesso all'hardware |
| /samples | Codici di esempio in C, Java e Python che utilizzano le librerie contenute in /libs |

In base alla piattaforma utilizzata, una DLL, DYLIB, o una SO deve essere collegata all'applicazione sviluppata. Queste librerie contenute in /libs e in /samples forniscono l'accesso a tutte le funzioni necessarie a configurare e a comunicare con gli Opal, la docking station e l'access point.

L'uso delle librerie è basato sul concetto di *system context*. Un context rappresenta una correlazione logica tra access point, docking station e Opal. In sostanza allocando e inizializzando un context si crea la connessione tra l'appicazione che si sta sviluppando e tutto il sistema APDM-Opal. Un context permette quindi di connettersi all'hardware, di configurarlo e di ricevere in maniera sincronizzata tutte le letture dei sensori.

Tipicamente, un programma realizzato per acquisire le misurazioni dai sensori prevede prima una configurazione del sistema e poi uno streaming dei dati. Le modalità di acquisizione dati messe a disposizione da APDM sono le seguenti:

- Robust Synchronized Streaming, permette lo streaming da più sensori sincronizzati tra di loro. I dati vengono memorizzati all'interno degli Opal in modo che nessun dato venga perso in caso di interruzione del segnale wireless.
- Rapid Synchronized Streaming, simile alla modalità Robust Synchronized Streaming ma i dati non vengono salvati in memoria in modo da diminuire la latenza. La latenza in sistemi operativi Linux o Mac OS è tipicamente nel range da 8 ms a 25 ms, mentre in Windows varia tra i 10 ms e i 75 ms.
- Synchronized Logging, in questa modalità gli IMU sono sincronizzati e registrano dati all'interno della memoria di bordo.
- Low Power Logging, permette il massimo risparmi di batteria, gli opal non sono sincronizzati tra loro e ciascun sensore raccoglie dati in maniera indipendente.

I file messi a disposizione da APDM permettono di scrivere programmi che utilizzano i sistemi operativi e i linguaggi di programmazione riportati in Tabella 2.8.

Tabella 2.8: Librerie APDM, linguaggi di programmazione e sistemi operativi supportati.

| Linguaggio | Sistemi operativi supportati |
|---|---|
| C/C++ | Windows 10, 64bit<br>Windows 8, 64bit<br>Windows 7, 64bit<br>Mac OSX 10.5 o successivo<br>Linux, 64bit |
| Java | Windows 10, 64bit<br>Windows 7, 64bit<br>Mac OSX 10.5 o successivo<br>Linux, 64bit |
| Python 2.6 e 2.7 | Windows 10, 64bit<br>Linux, 32bit<br>Linux, 64bit |

Il Software Development Kit può essere scaricato nella sua versione più recente utilizzando il seguente link share.apdm.com, inserendo *SDK* come nome utente e password.

## 2.2 Universal Robots – UR3

Il robot utilizzato nell'attività di tesi è un UR3, prodotto dalla Universal Robots. Si tratta di un robot collaborativo leggero e compatto che grazie al suo ingombro minimo può essere installato facilmente nei pressi dei macchinari o in spazi di lavoro ristretti.

Questo cobot, a fronte di un peso molto ridotto pari a 11 kg, offre un payload di 3 kg. Ogni giunto del cobot è in grado di effettuare rotazioni di ±360 gradi, mentre il polso presenta una rotazione infinita. Ulteriori informazioni su caratteristiche e funzionalità dell' UR3 possono essere trovate in [14].


Figura 2.9: Universal Robots - UR3 overview.

Oltre al braccio robotico a 6 gradi di libertà, l'hardware dell'UR3 è constituito da una control box ed un Teach Pendant (TP) per programmare il robot.

La control box, che utilizza un sistema operativo dedicato chiamato Polyscope, controlla il braccio robotico e fornisce 16 DI, 16 DO, 2 AI e 2 AO, che possono essere impiegati per collegare strumentazione esterna al robot quali ad esempio sensori, PLC, pulsanti di emergenza etc. Sulla base del case della control box, è presente anche una porta ethernet utilizzabile per stabilire una connessione robot - computer esterno secondo il protocollo TCP/IP.

Il Teach Pendant è un touchscreen 12" collegato via cavo alla control box che viene usato per programmare il robot in maniera semplice ed intuitiva grazie ad una GUI dedicata [15].

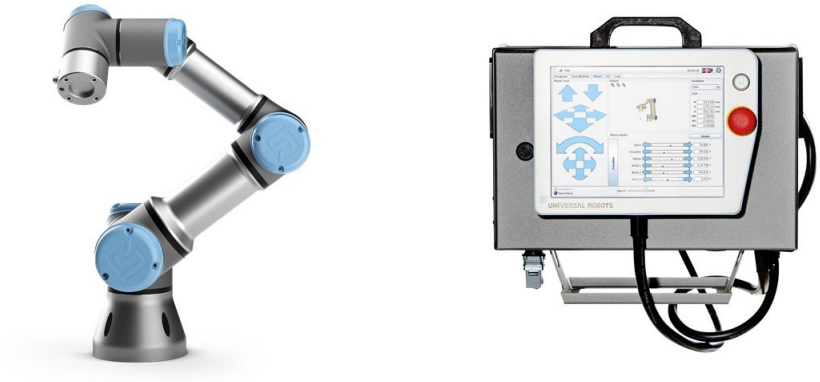


Figura 2.10: Hardware UR3 (UR3 a sinistra, controlbox con Teach Pendant a destra.

### 2.2.1 Programmazione

Gli UR possono essere programmati attraverso due differenti modalità:

- PolyScope (interfaccia grafica)
- URScript (linguaggio di programmazione creato dall'Universal Robot)

#### 2.2.1.1 PolyScope

PolyScope garantisce una notevole semplicità di programmazione assicurando comunque flessibilità e possibilità di reimpiego del robot. Figura 2.11 riporta una schermata Polyscope su TP. La Universal Robots mette a disposizione una serie di strumenti formativi, tra cui ebook e corsi online, che permettono di imparare questo tipo di programmazione. Tutto questo consente l'utilizzo dei cobot anche alle piccole aziende con una ridotta o nulla competenza robotica. Questo tipo di controllo del robot non richiede la scrittura di codice ed è basato su una programmazione ad albero resa graficamente e semplice da usare.

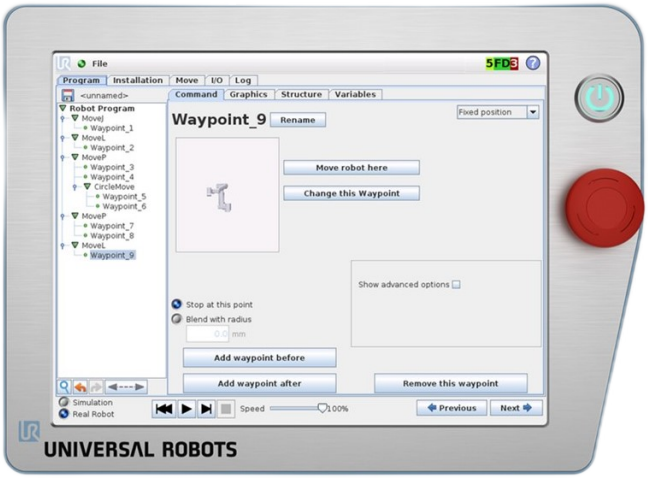



Figura 2.11: Esempio di programma realizzato in PolyScope su Teach Pendant

Utilizzando PolyScope è possibile generare dei programmi sia all'interno dell'ambiente di lavoro del robot, tramite il Teach Pendant che offline utilizzando URSim.

URSim è un softwere per sistemi operativi Linux che può essere utilizzato per la programmazione offline e per la simulazione di programmi di controllo del robot. Per utilizzare URSim l'UR consiglia di impiegare una virtual machine come VMWare Player e di VirtualBox.

A causa della mancanza di connessione con il braccio robotico, reale il simulatore ha alcune limitazioni riportate nella seguente lista:

- l'arresto di emergenza non può essere usato
- lo stato degli IO non può essere settato
- risulta possibile la collisione del robot con se stesso o con oggetti presenti nell'area di lavoro
- Force mode (controllo del movimento del robot basato su forze e momenti applicati dell'esterno) è disabilitato

L'interfaccia grafica dell'URSim, riportata in Figura 2.12, è simile alla GUI di Polyscope che si trova sul TP, anche le funzionalità di questo sofwere sono molto simili a PolyScope.

I programmi realizzati con l'ausilio di URSim possono essere caricati sul robot reale per essere eseguiti. [16]

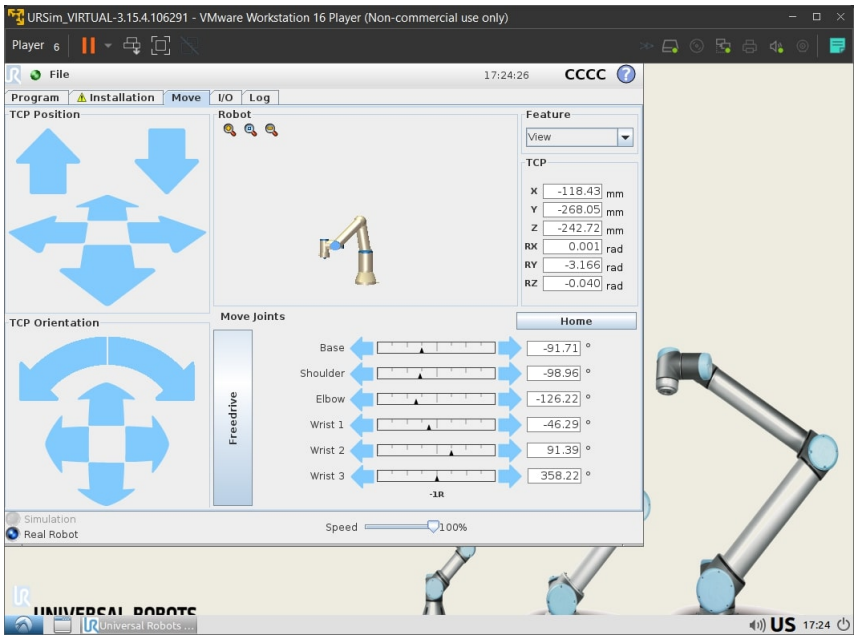



Figura 2.12: URSim CB-series lanciato in VMWare Player

#### 2.2.1.2 URScript

La programmazione attraverso la scrittura di codice è molto versatile. Mentre PolyScope risulta più adatto per lo sviluppo delle applicazioni della robotica industriale più tradizionale quali ad esempio la pallettizzazione, l'asservimento macchina e il pick and place, l'URScript peremtte di utilizzare il robot al pieno delle sue potenzialità. Questo linguaggio di programmazione include variabili, type e flow control statement, oltre a funzioni implementate per monitorare e comandare I/O e movimenti del robot. Le funzioni disponibili sono suddivise nelle seguenti categorie:

- Motion, funzioni utili a comandare i movimenti del robot (es. movej, speedj, servoj etc.)
- Internals, utili per ricevere degli output dalla control box riguardo lo stato del robot (es. get_actual_joint_positions, get_actual_tcp_pose, etc.) o per eseguire calcoli di cinematica (get_forward_kin, get_inverse_kin)
- Urmath, funzioni matematiche (es. atan2, d2r, interpolate_pose etc.)
- Interfaces, funzioni utili per la lettura di IO digitali e analogici (es. get_digital_-out, get_analog_in etc.)
- IOconfiguration, funzioni per configurare IO (es. set_tool_analog_input_domain, set_standard_analog_input_domain etc...)

Maggiori informazioni sulle funzioni di questo linguaggio di programmazione possono essere trovate nel manuale rilasciato della Universal Robot [17].

La programmazione del robot attraverso codice viene fatta realizzando uno script su un computer esterno, utilizzando un qualsiasi linguaggio di programmazione, e connettendosi al URControl, il sistema di controllo low-level della control box, usando una connessione di tipo TCP/IP (la comunicazione via TCP/IP viene descritta nel dettaglio nel paragrafo 2.2.1.3). Stabilita la connessione i comandi URScript devono essere inviati al socket sottoforma di testo in caratteri ASCII e ogni linea di codice deve terminare con "\n".

#### 2.2.1.3 Interfacce UR via TCP/IP

L'UR3 fornisce quattro client interface associate a quattro differenti porte, che permettono di stabilire una connessione via TCP/IP tra la control box ed un pc esterno. Tutte le porte rimangono contemporaneamente aperte e attraverso queste è possibile inviare comandi al robot o ricevere dati come posizione, temperatura, stato del robot etc. Le tipologie di dati trasmessi si differenziano in base alla porta scelta. Le principali caratteristiche delle quattro interfacce sono riportate in Tabella 2.9

Tabella 2.9: Panoramica delle quattro client interface via TCP/IP fornite da Universal Robots [18].

| | Porta | Riceve | Trasmette | Frequenza |
|---|---|---|---|---|
| **Primary client interface (PCI)** | 30001 | URScript | dati | 10 Hz |
| **Secondary client interface (SCI)** | 30002 | URScript | dati | 10 Hz |
| **Real-Time client interface (RTCI)** | 30003 | URScript | dati | 125 Hz |
| **Real-Time Data Exchange (RTDE)** | 30004 | RTDE protocol | RTDE protocol | 125 Hz |

La Primary client interface e la Secondary client interface sono principalmente usate per una comunicazione tra la control box e la GUI sul TP (Figura 2.13). Entrambe le interfacce lavorano ad una frequenza di 10 Hz; il controllore del robot fornisce un server che riceve comandi URscript e invia dati relativi allo stato del robot. L'obiettivo di questa tesi è stato quello di permettere un'interazione uomo-robot basata su sensori inerziali che lavorasse in real-time. Una frequenza di 10 Hz è

stata ritenuta troppo bassa per garantire i requisiti di real-time, per questo motivo queste due interfacce non sono state analizzate nel dettaglio.

La Real - Time client interface è un'interfaccia simile alle primary e secondary interface. Il controller riceve comandi URscript e trasmette dati relativi allo stato del robot come ad esempio i gradi di libertà nei giunti, le velocità nello spazio giunti/operativo, la posizione del tool center point nello spazio operativo, etc.. Un file excel contenente la lista completa dei dati forniti da queste tre interfacce è scaricabile dal sito dell'Universal Robots [18]. La principale differenza con le due precedenti interfacce sta nel fatto che la RTCI stabilisce una comunicazione con il robot a 125 Hz.

La Real - Time Data Exchange, disponibile alla porta 30004, è una interfaccia scritta in Python che permette di sincronizzare applicazioni esterne con il controller dell'UR attraverso appunto una connessione secondo lo standard TCP/IP (Figura 2.13). La sincronizzazione è configurabile e può ad esempio includere i seguenti tipi di dati:

- Output: stato del robot, informazioni di posizione, velocità, accelerazione sia nello spazio giunti che nello spazio operativo, I/O analogici e digitali e output register(registri in cui possono essere usati per immagazzinare variabili per utilizzi vari).
- Input: parametri impiegati per configurare gli output e input register

L'utilizzo della RTDE è suddiviso in due step: una procedura di setup e un successivo synchronization loop. Nella fase di connessione a questa interfaccia, il client deve settare le variabili che verranno poi sincronizzate É possibile realizzare qualsiasi combinazione di input e output. Completato il setup, la sincronizzazione può essere avviata e messa in pausa. Una volta avviata, la RTDE genera dei messaggi di output ogni 8 ms. Il principale vantaggio di utilizzare una RTDE è quello di avere generalmente una latenza nulla [19].

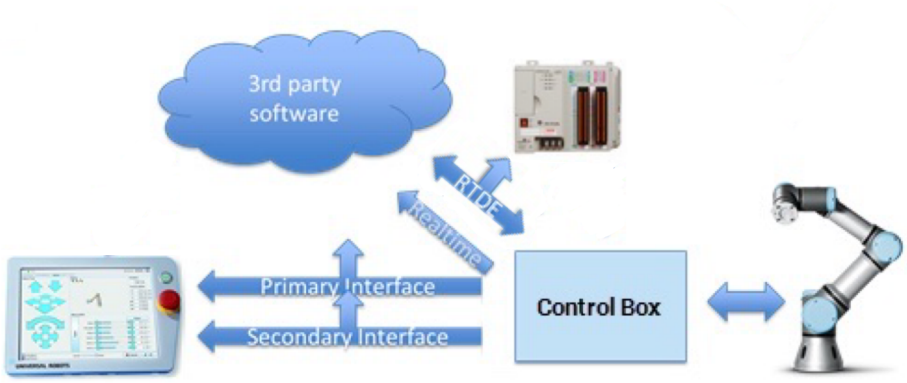



Figura 2.13: Overview delle client interface [20].

#### 2.2.1.4 Controllo robot

In questo paragrafo viene descritta la modalità di controllo dell'UR implementata durante le attività di tesi ( https://github.com/danielstankw/Servoj_RTDE_UR5.git).

Il controllo del robot è basato sugli strumenti di programmazione e comunicazione con l'UR descritti nei paragrafi 2.2.1.1 e 2.2.1.3. Gli elementi che permettono

questa modalità di comando sono riportati in Figura 2.14, si tratta di un programma realizzato in Polyscope utilizzando i comandi dell'URScript, un PC esterno nel quale è implementata un'interfaccia RTDE e la control box che permette la comunicazione tra l'applicazione eseguita sul PC e Polyscope. Il PC è collegato alla control box attraverso un cavo ethernet. Tra i due elementi è generata una connessione secondo il protocollo TCP/IP in cui il sistema di controllo dell'UR3 funge da server.

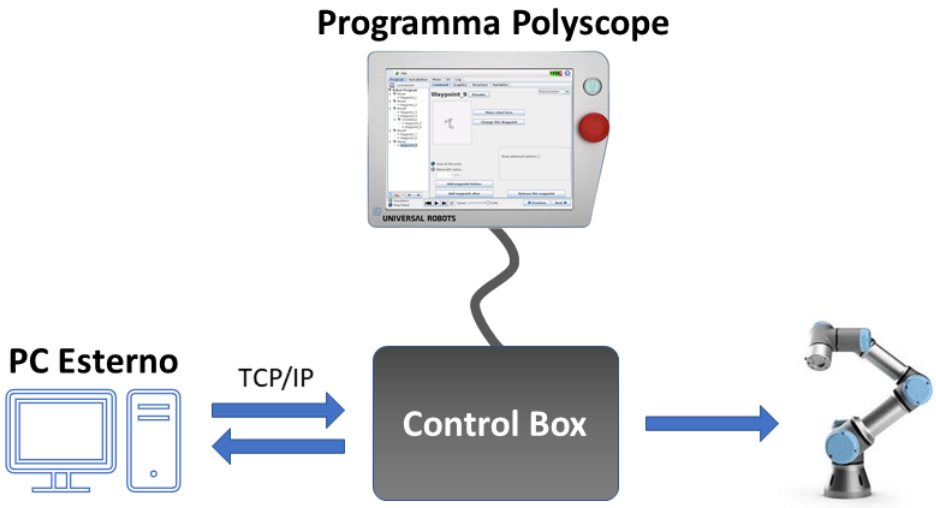



Figura 2.14: Sistema di controllo UR3

La modalità di controllo utilizzata è basata sull'utilizzo di due thread che vengono eseguiti contemporaneamente. In particolare si ha un'applicazione esterna e un programma in Polyscope.

I processi realizzati dai thread sono resi graficamente in Figura 2.15. Polyscope realizza il controllo vero e proprio del robot. Il programma è suddiviso in tre blocchi. La sezione BeforeStart è la prima ad essere eseguita e ha lo scopo di inizializzare tutte le variabili e di dare avvio al Robot Program, all'interno della quale sono implementati i comandi di movimento del robot. Un Thread viene eseguito in contemporanea con il RobotProgram, con lo scopo di leggere i dati inviati dall'applicazione esterna e di aggiornare le variabili di input ai comandi di movimento con gli ultimi dati ricevuti.

Lo scopo dell'applicazione esterna è quindi quello di fornire i dati per i comandi del robot secondo la procedura riportata nello schema in Figura 2.15. Innanzitutto, viene aperta la connessione con la RTDE e vengono quindi eseguiti il setup dell'interfaccia (in questa fase si settano gli input e gli output della RTDE) e il suo avvio. Avviando la Real-Time Client Interface, l'applicazione esterna viene sincronizzata con il programma in Polyscope. A questo punto si entra nel Control loop, che invia al programma in Polyscope i dati che vengono generati da un algoritmo implementato all'interno del Control loop stesso. Questi dati possono essere ad esempio informazioni di posa qualora si voglia muovere il robot da una configurazione ad un'altra oppure può trattarsi di velocità nello spazio giunti/operativo nel caso in cui si intenda eseguire un controllo in velocità. Più in generale si tratta di un target che si vuole che l'UR raggiunga. Oltre all'invio di dati, all'interno del Control loop avviene la ricezione di feedback. Per feedback si intendono sia le informazioni relative lo stato del robot (posa, velocità, accelerazioni etc.) che messaggi provenienti dal programma in polyscope.

Una volta che il task è stato eseguito, la connessione tra il pc esterno e la control box viene chiusa ed entrambi i programmi vengono terminati.

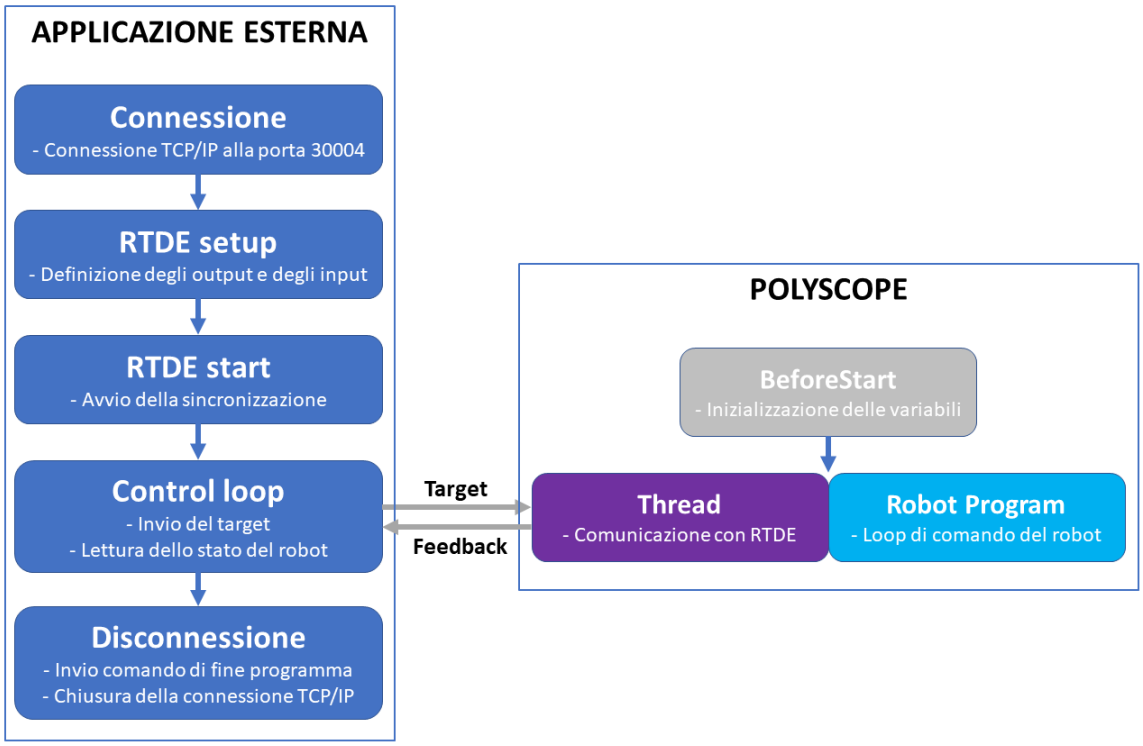



Figura 2.15: Schema dell'ambiente di comando del robot

## 2.3 ROBOTIQ 2F-85 Gripper

Come tool sulla flangia dell'UR3 è installato una pinza 2F-85 Gripper prodotta da ROBOTIQ e raffigurata in Figura 2.16. Si tratta di una pinza adattiva per robot collaborativi che consente una presa parallela sia interna che esterna. La programmazione di questo tool è molto rapida e non richiede alcuna specifica competenza di robotica. ROBOTIQ fornisce insieme all'hardware anche un software da installare in Polycope che permette il controllo della pinza direttamente dal Teach Pendant [21].

Le specifiche del gripper sono riportate in Tabella 2.10


Figura 2.16: 2F-85 Gripper

Tabella 2.10: Specifiche 2F-85 Gripper

| | |
|---|---|
| Corsa | 85 mm |
| Forza di presa | da 20 a 235 N |
| Carico massimo di presa | 5 kg |
| Peso della pinza | 0,9 kg |
| Velocità di chiusura | da 20 a 150 mm/s |
| Valore nominale del grado di protezione | IP40 |

## 2.4 ROS (Robot operating system)

ROS è un software open source per la programmazione di robot e per lo sviluppo di complesse applicazioni di robotica. ROS è considerato come un meta-operating-system in quanto compie differenti funzioni tipiche di un sistema operativo ma necessita di un sistema operativo host, come Linux, sul quale essere installato.

In particolare, come un sistema operativo, esso fornisce gli hardware abstraction layer che permettono di creare applicazioni senza preoccuparsi della compatibilità con le differenti tipologie di hardware.

La principale funzionalità di ROS è quella di fornire una piattaforma per lo scambio di messaggi, nella quale differenti processi e thread possono comunicare e inviarsi dati attraverso appunto lo scambio di messaggi [22] [23].

In questo paragrafo l'architettura di ROS viene descritta in due sezioni:

- ROS filesystem, in questa sezione viene spiegato come ROS è strutturato al suo interno, la struttura delle cartelle e il minimo numero di file necessari a farlo funzionare
- ROS computation graph, in questa sezione viene descritto il meccanismo con cui sono realizzati gli ambienti ROS e i processi che vengono eseguiti al loro interno

### 2.4.1 ROS filesystem

In questa sezione viene introdotta una parte della terminologia di ROS. Nello specifico saranno analizzati i concetti di ROS workspace e package.

**ROS workspace**
Il workspace, tipicamente chiamato "catkin" o "catkin_ws" è la cartella che contiene i package (progetti sviluppati), i file sorgente e le informazioni di configurazione. Un tipico workspace è riportato in Figura 2.17, esso è costituito da tre cartelle:

- `src`: contiene i package sviluppati oltre al file `CMakeLists.txt`, un file di configurazione che viene invocato nel momento in cui si configurano i package nel workspace.
- `buildt`: al suo interno troviamo informazioni di configurazione del workspace.
- `devel`: spazio di sviluppo, utilizzato per contenere i package già compilati che devono essere testati.

**ROS package**
Quando si parla di package si intende una tipica struttura di file e cartelle all'interno del sistema ROS. I package contengono i programmi eseguibili tramite i comandi ROS e la loro struttura è riportata in Figura 2.18.

I file e le cartelle su cui si lavora principalmente sono le seguenti:

- `package.xml`, `CMakeLists.txt`: file di configurazione.
- `scripts`: contiene gli eseguibili scritti in C++, Python o qualsiasi altro linguaggio di programmazione.

- `launch`: questa cartella contiene i launch file utilizzati per lanciare uno o più nodi ROS.
- `msg`: questa cartella è utilizzata per la creazione di message type customizzati.

```
└── catkin_ws
    ├── build
    │   ├── catkin
    │   ├── catkin_generated
    │   ├── Makefile
    │   └── ...
    ├── devel
    │   ├── setup.zsh
    │   └── ...
    └── src
        ├── CMakeLists.txt -> /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake
        └── ...
```

Figura 2.17: ROS workspace, struttura tipica

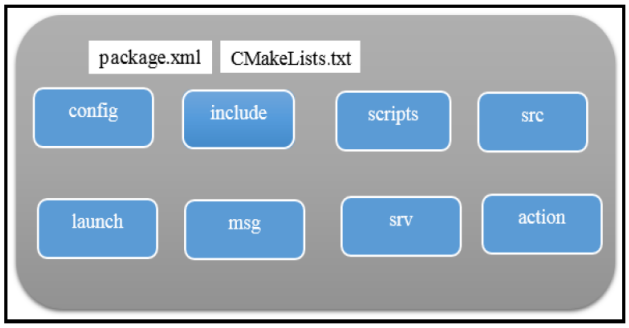



Figura 2.18: ROS package, struttura tipica

### 2.4.2 ROS computation graph

ROS crea un network in cui tutti i processi sono interconnessi. Ogni processo del sistema può accedere a questo network, interagire con altri processi e trasmettere dati al network. Questo network è chiamato computation graph. Gli elementi base del computation graph, illustrati in Figura 2.19, sono i nodi, il master, i messaggi, i topic, i sevices e i bags.

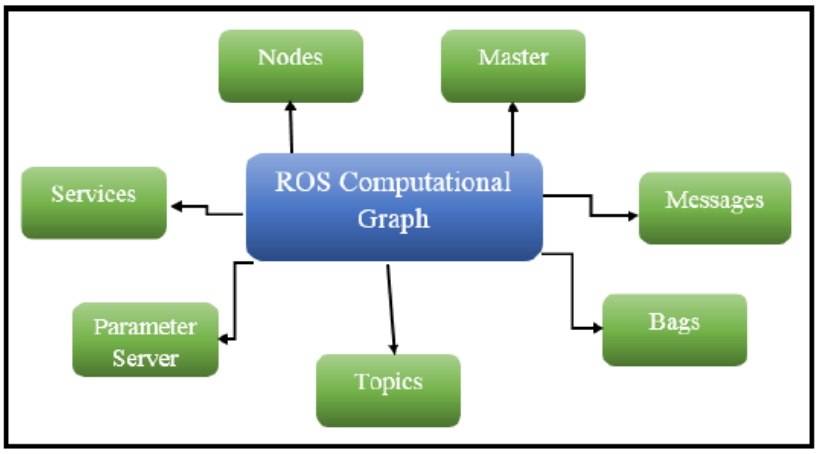



Figura 2.19: Gli elementi del computation graph

Gli elementi del computation graph principalmente utilizzati nell'ambito di questa tesi sono i seguenti:

- ROS nodes: i nodi corrispondono a processi che eseguono task specifici. Essi sono scritti utilizzando le client libraries di ROS come `rospy` e `roscpp` che supportano diversi metodi di comunicazione, principalmente i messaggi o i services. Utilizzando questi metodi di comunicazione, i nodi possono scambiare dati tra loro e creare un network che premette l'esecuzione di un particolare task.
- ROS master: il master è responsabile della creazione e della connessione dei nodi.
- ROS messages: i nodi comunicano tra loro attraverso i messaggi di differenti tipologie basati su gli standard primitive types (integer, floating, Boolean, etc.).
- ROS topics: ogni messaggio è trasportato usando i topic, un nodo può pubblicare un messaggio ad un topic o può ricevere un messaggio sempre tramite un topic. In Figura 2.20 è diagrammata la logica di comunicazione tra nodi.

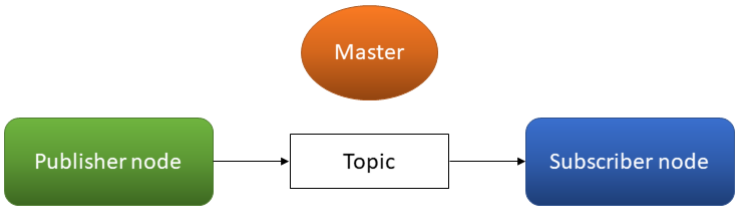



Figura 2.20: Comunicazione tra nodi attraverso topic, il Master gestisce la logica della comunicazione.

Un'applicazione robotica può contenere numerosi processi, che rilevano dati da sensori, che elaborano dati e che sono responsabili del controllo del robot. In questo contesto, l'utilità di ROS risiede proprio nel computation graph e negli elementi che lo compongono. Utilizzando i nodi è possibile separare codici e funzionalità, rendendo così il sistema più semplice e robusto. Inoltre, ROS riduce la complessità del codice facilitandone il debug.

# Capitolo 3

# Applicazioni e algoritmi sviluppati

## 3.1 Controllo movimenti robot

In questo paragrafo vengono analizzate le funzioni utilizzate per movimentare il robot e i risultati sperimentali che permettono di considerare valida la modalità di controllo scelta.

Come descritto nel paragrafo 2.2.1.4, il comando dell'UR è basato sull'utilizzo di thread. Si hanno tre thread: un'applicazione su un PC esterno e due thread implementati nel teach pendant. Dei due thread sul teach pendant, uno si occupa del controllo del robot, chiamando una funzione che muove l'UR all'interno di un loop infinito, mentre l'altro permette la comunicazione tra il programma che controlla il robot e il PC esterno, comunicazione necessaria in quanto le informazioni per movimentare il robot provengono dall'applicazione sul PC esterno. Tutti e tre i thread vengono eseguiti a 125 Hz.

L'URScript fornisce tre differenti funzioni per muovere il robot: move, speed e servo.

**Move**
Il comando move è adatto a semplici movimenti da un punto ad un altro. Il movimento può essere lineare nello spazio cartesiano o nello spazio giunti, o circolare nello spazio cartesiano.

Nonostante molteplici comandi move possano essere eseguiti in successione, il robot si ferma alla fine di ogni comando rendendo il movimento irregolare, a meno che non venga dato come parametro della funzione un raggio di raccordo. Il raggio di raccordo fa sì che il robot raccordi le traiettorie durante l'esecuzione dei comandi, in questo caso però il punto nel mezzo di due traiettorie consecutive non viene raggiunto. Tutto ciò rende il comando move inadatto a realizzare un preciso controllo della traiettoria in real-time e per questo la funzione non è stata analizzata ulteriormente e non è stata utilizzata nell'applicazione finale.

**Speed**
La funzione speed realizza un comando in velocità, muove il robot con una velocità costante sia nello spazio giunti che nello spazio cartesiano. Raggiunta la velocità, il robot continua a muoversi fin quando un nuovo comando non viene fornito o finché non vengono raggiunti i limiti dei giunti.

Nonostante il comando speed sia una valida soluzione per movimentare il robot, non è stato ritenuto adatto per un comando del robot eseguito allo scopo di orientare il gripper durante un'operazione di pick and place e per questo non è stato analizzato nello specifico.

**Servo**

Il comando servoj esegue un controllo della posizione dei giunti. Come riportato nell'URScript manual, la funzione servoj può essere utilizzata per un controllo in real-time del robot. Per questo motivo servoj è stata utilizzata per implementare il controllo della robot nell'applicazione finale.

Servoj agisce in modo simile ad un controllore PID, per ogni giunto calcola l'errore tra la posizione desiderata e la posizione attuale e muove il robot in accordo con l'errore calcolato. Raggiunta la posizione desiderata il comando ferma il movimento in maniera brusca, quindi molteplici comandi servo devono essere eseguiti in successione al fine mi muovere il braccio robotico senza interruzioni del movimento o vibrazioni.

La funzione prende in input i seguenti parametri:

- q: angoli in radianti che rappresentano le rotazioni di base, spalla, gomito, polso1, polso2, polso3. q è la posa target nello spazio giunti
- a: parametro previsto ma non implementato nella versione corrente
- v: parametro previsto ma non implementato nella versione corrente
- t: intervallo di tempo in secondi durante il quale il comando controlla il robot. Per un controllo a 125 Hz questo parametro viene settato a 0.008 s
- lookahead time: parametro simile al termine derivativo di un controllore PID. Valore compreso tra 0.03 e 0.2 secondi, va utilizzato per ammorbidire il movimento del robot.
- gain: Termine proporzionale. Varia nel range 100 - 2000.

Come riportato in precedenza il metodo move non è in grado di soddisfare i requisiti dell'applicazione sviluppata mentre il metodo speed non è stato ritenuto adatto per effettuare i movimenti in telecontrollo. Le attività di test si sono quindi concentrate sulla funzione servo. Al fine di verificare la validità di servoj sono stati condotti dei test per valutare l'errore di posizionamento commesso dal robot durante il movimento, la fluidità del movimento e come questi due fattori fossero influenzati da due parametri della funzione: lookahead time e gain. Durante i test, al robot viene fatta eseguire una traiettoria che lo porta da una posizione iniziale ad una posizione finale. Sfruttando la soluzione di comando basata sui thread descritta nel paragrafo 2.2.1.4, la traiettoria è calcolata nel PC esterno e con una frequenza di 125 Hz le informazioni di posa vengono passate ai thread in polyscope che eseguono il comando del robot. Il parametro di tempo passato a servoj è pari a 0.008 secondi (125Hz) e nell'esecuzione del test numerosi comandi servo vengono inviati al robot in successione al fine di eseguire tutta la traiettoria. Le modalità con cui sono stati eseguiti i test su servoj sono riportate più nel dettaglio nell'Appendice C insieme ai codici utilizzati.

I parametri di default della funzione servoj sono: lookahead time pari a 0.1 s e gain pari a 300. I risultati relativi a una prova eseguita con questi parametri sono riportati in in Figura 3.1 e in Figura 3.2. In Figura 3.1 osserviamo l'andamento della posizione del TCP nelle tre coordinate cartesiane, in blu è riportato il comando inviato al robot mentre in arancione si ha la posizione effettiva del TCP durante l'esecuzione della traiettoria. In Figura 3.2 sono riportati l'andamento della velocità lineare ottenuto dal calcolo analitico della traiettoria e la velocità del TCP misurata

durante la prova. Come si può osservare la traiettoria viene eseguita correttamente: la curva è continua, quindi il robot esegue il movimento senza mai fermarsi e le oscillazioni registrate nella misura della velocità non sono tali da produrre vibrazioni.

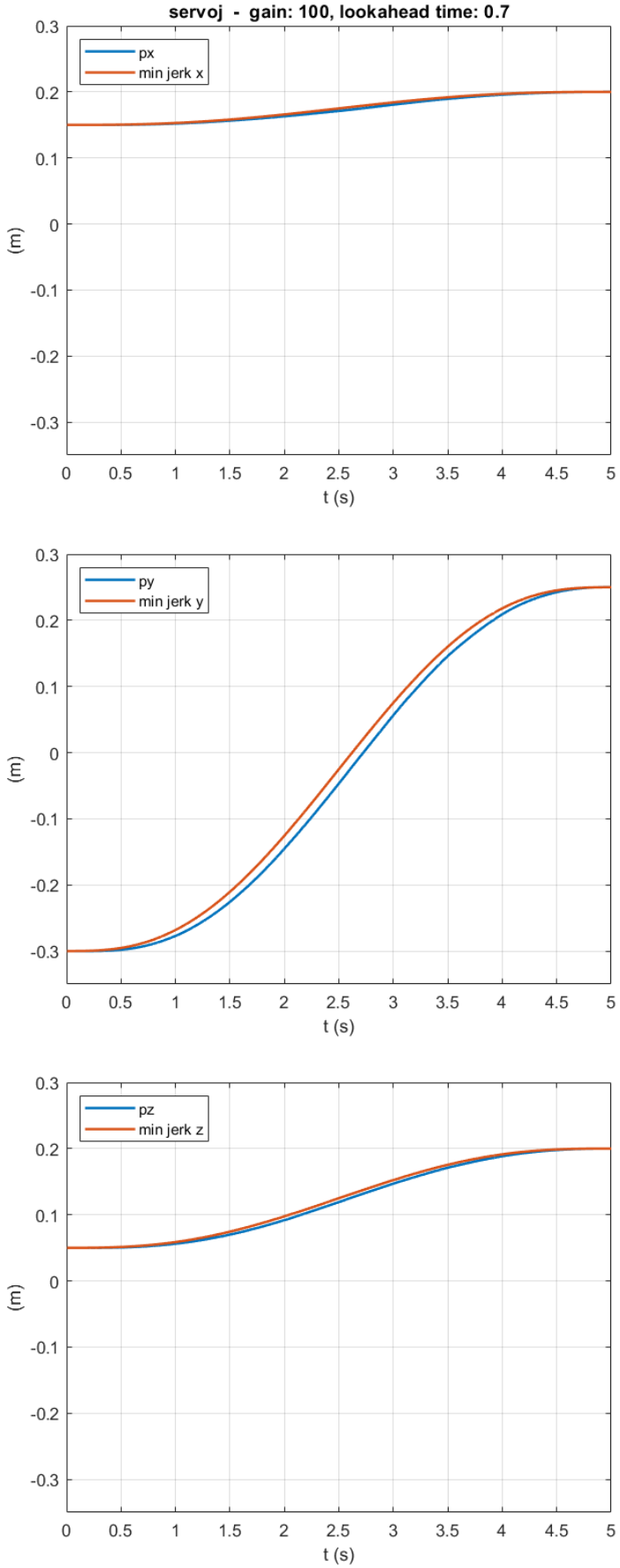



Figura 3.1: Esecuzione traiettoria tramite comando servoj con parametri di default e modalità di controllo robot basata sui thread. Posizione nelle tre coordinate cartesiani del TCP

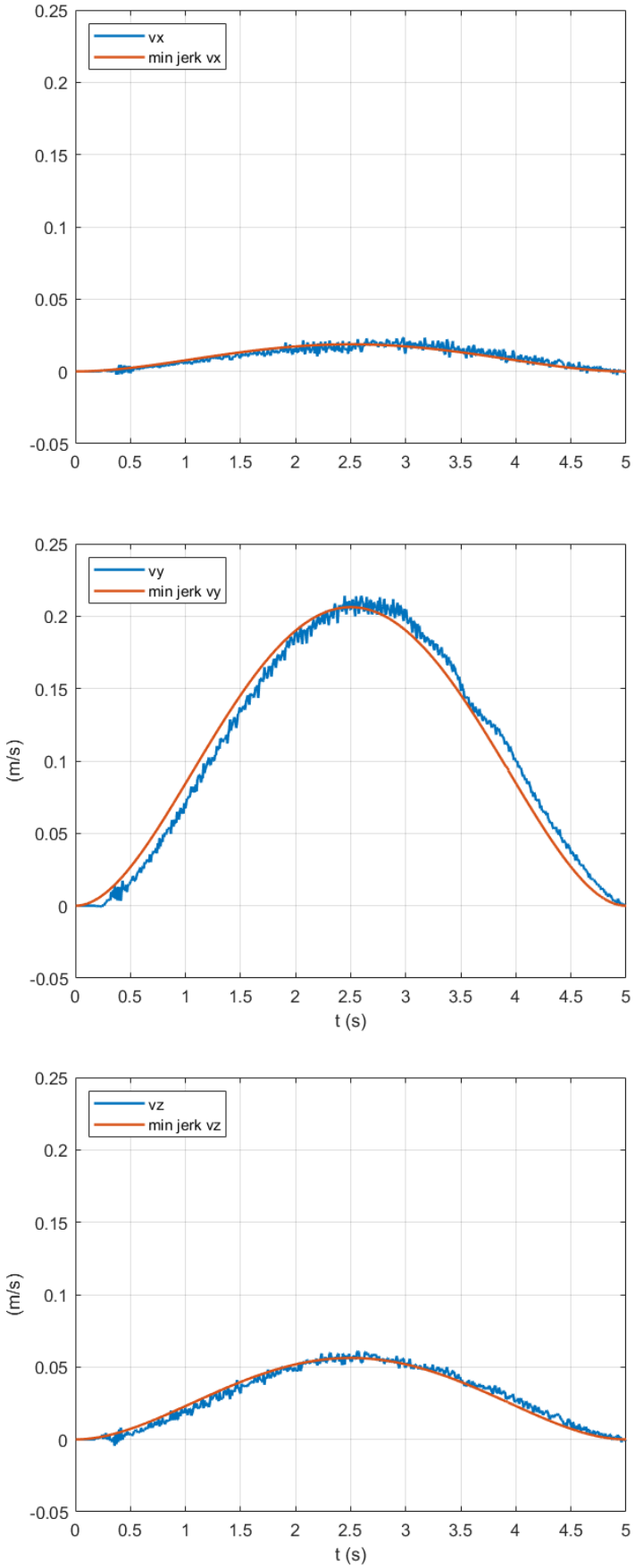



Figura 3.2: Esecuzione traiettoria tramite comando servoj con parametri di default e modalità di controllo robot basata sui thread. Velocità lineare nelle tre coordinate cartesiane.

Gli errori di posizionamento rilevati durante questa prova sono riportati in Tabella 3.1. Il massimo errore tridimensionale è pari a 2.28 cm mentre l'errore medio è di 1.19 cm, questi valori sono stati ritenuti in linea con i requisiti necessari per l'applicazione finale.

Tabella 3.1: Errori di posizionamento rilevati durante l'esecuzione della traiettoria attraverso il comando servoj con gain pari a 300 e lookahead time pari a 0.1 secondi

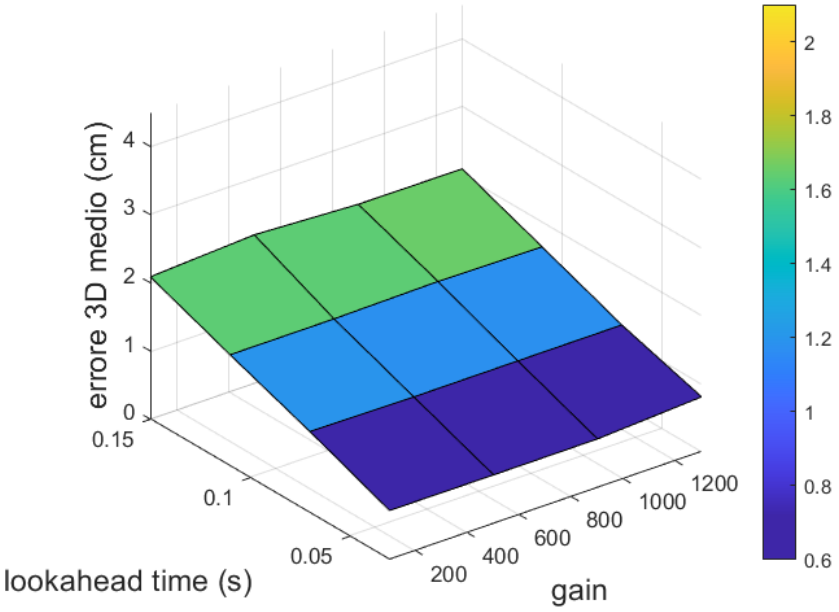
| | X | Y | Z | 3D |
|---|---|---|---|---|
| errore massimo [cm] | 0.40 | 2.17 | 0.59 | 2.28 |
| errore medio [cm] | 0.17 | 1.13 | 0.33 | 1.19 |

Va però tenuto conto che la capacità del robot di seguire la traiettoria dipende dai parametri gain e lookahead time della funzione servoj. A questa considerazione si può giungere già soltanto ricordando che servoj esegue un controllo della posizione del robot nello spazio giunti in maniera simile ad un controllore PID e per questo i parametri passati alla funzione influenzano la risposta del robot al comando.

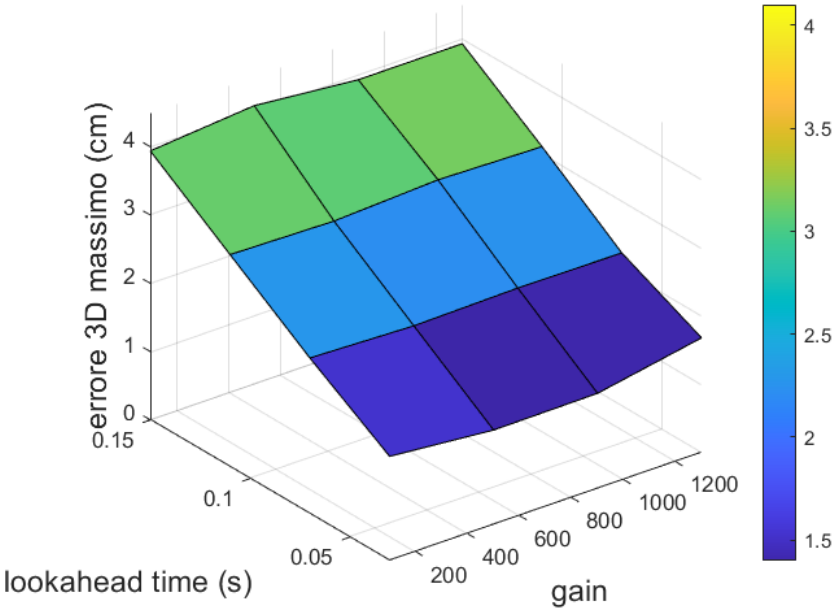
Per analizzare meglio l'influenza di questi parametri, oltre al test eseguito con valori di gain e lookahead time pari rispettivamente a 300 e 0.1 secondi, i cui risultati sono riportati in Figura 3.1 e 3.2, sono state eseguite altre prove con la stessa modalità ma variando gain e lookahead time. Gli output di tali prove sono riportati in Figura 3.3. In particolare, in Figura 3.3a è riportato l'andamento dell'errore di posizione tridimensionale medio mentre in Figura 3.3b è riportato l'andamento dell'errore di posizione tridimensionale massimo al variare dei due parametri. Quello che si osserva per entrambi gli errori è che il gain sembrerebbe non avere alcun effetto sull'esecuzione del movimento, infatti una variazione di gain non provoca una variazione sensibile dell'errore di posizionamento misurato durante l'esecuzione della traiettoria. Al contrario, il lookahead time ha un importante effetto. Nelle prove effettuate, un aumento di questo parametro ha comportato anche un aumento dell'errore di posizionamento.

Nonostante l'effetto del gain non sia osservabile confrontando gli errori di posizionamento, esso è però visibile se si guarda l'andamento delle velocità lineari del TCP misurate durante le prove. In Figura 3.4 sono riportate le velocità rilevate durante due test differenti. I grafici in Figura 3.4a riportano gli andamenti delle velocità misurati durante una prova con gain di 500, in Figura 3.4b ci sono i risultati con gain pari a 1700. Quello che si osserva è che un aumento di questo parametro produce un aumento dell'ampiezza delle oscillazioni della velocità che a sua volta comporta un aumento delle vibrazioni prodotte dal robot durante l'esecuzione della traiettoria.

Quello che si può concludere dai risultati riportati in Figura 3.3 e 3.4 è che nel caso in analisi bassi valori di gain e lookahead time producono i migliori risultati: il robot si muove senza produrre vibrazioni e la traiettoria eseguita si discosta di poco dalla traiettoria desiderata. Queste conclusioni non hanno però una valenza universale, infatti durante le attività svolte si è notato come la risposta del robot dipenda da diversi fattori, quali ad esempio il parametro t (tempo di comando) della funzione servoj e la traiettoria eseguita oltre ovviamente a gain e lookahead time. In generale, utilizzando valori alti di gain, si è sempre riscontrato un aumento delle vibrazioni e un robot più nervoso, mentre una limitazione dell'errore di posizionamento è stata sempre ottenuta con bassi valori di lookahead time.



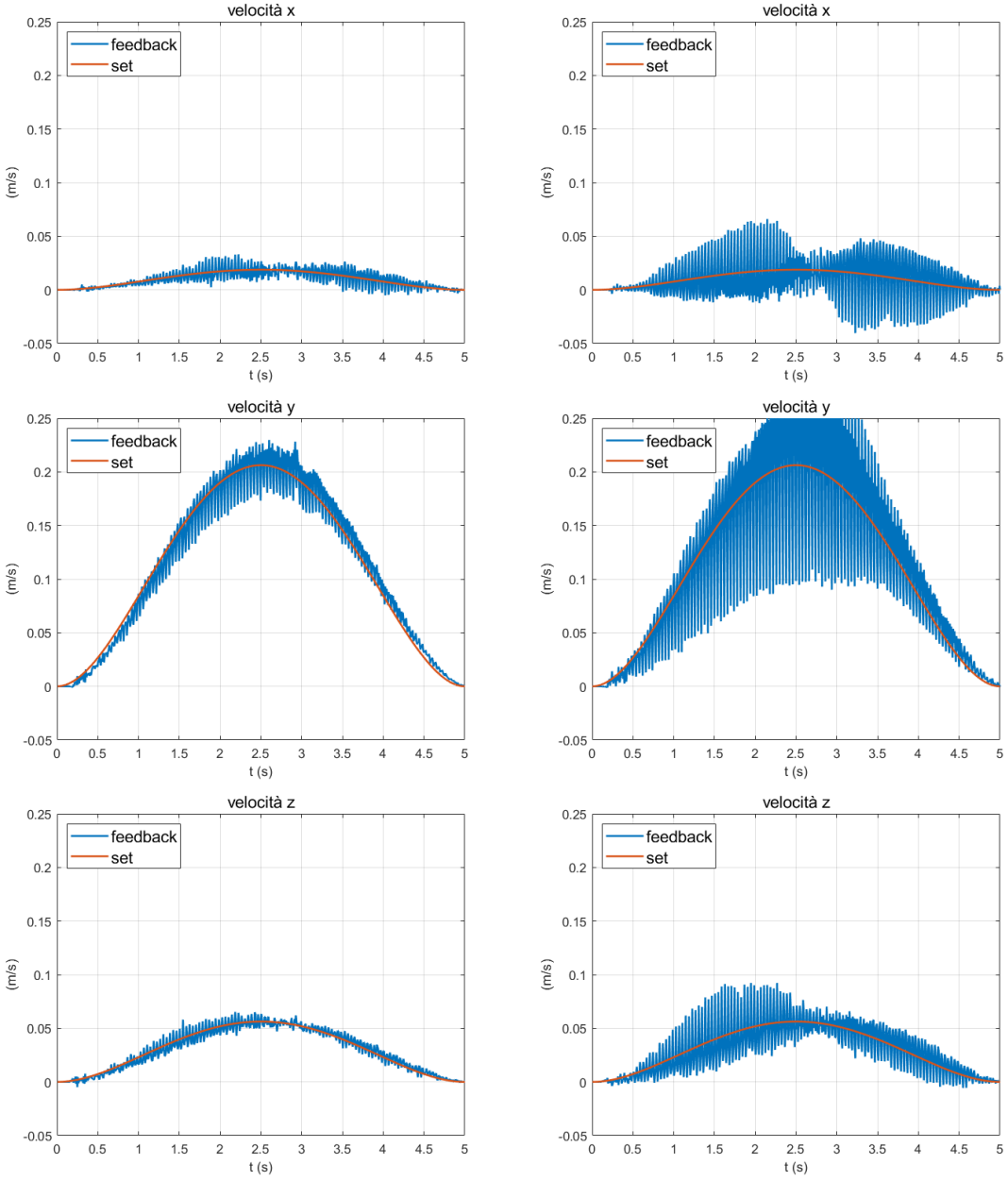

(a) Andamento dell'errore tridimensionale medio




(b) Andamento dell'errore tridimensionale massimo

Figura 3.3: Risultati delle analisi sui parametri gain e lookahead time

Gain = 500 Gain = 1700




(a) (b)

Figura 3.4: Velocità lineari misurate durante le prove di comando attraverso la funzione servo con gain variabile. (a) lookahead time = 0.03 secondi, gain = 500 (b) lookahead time = 0.03 secondi, gain = 1700

## 3.2 Telecontrollo per cobot in operazioni di pick and place

In questo paragrafo viene descritta l'applicazione sviluppata durante l'attività di tesi. Si tratta di un un sistema di telecontrollo basato sull'utilizzo di IMU per comandare un robot collaborativo UR3 durante operazioni di pick and place di oggetti poliedrici o con forme particolari.

L'interfaccia sviluppata permette di generare comandi per il robot attraverso l'orientazione della mano dell'operatore, la capacità di tradurre i movimenti della mano in comandi per il robot rende il telecontrollo realizzato particolarmente intuitivo. Oltre a una facilità di utilizzo, sono garantite precisione e fluidità nei movimenti del robot e un controllo in real-time, infatti i comandi vengono ricevuti, processati e inviati all'UR con una frequenza pari a 125 Hz.

Per realizzare il sistema sono state utilizzate la strumentazione e le relative modalità di utilizzo o programmazione presentate nel Capitolo 2. In particolare, l'orientazione della mano dell'operatore viene acquisita attraverso un sensore inerziale posizionato sul dorso della mano, i dati rilevati vengono processati all'interno di un ambiente ROS creato appositamente per consentire il telecontrollo dell'UR, infine sono prodotti i comandi per il robot che vengono inviati attraverso un'interfaccia di tipo RTDE.

Come riportato in precedenza il telecontrollo sviluppato è finalizzato a operazioni di pick and place di oggetti con forme non comuni o comunque differenti. Per afferrare oggetti di questo tipo risulta più conveniente sfruttare le capacità cognitive dell'operatore rispetto ad una pre-programmazione del robot. Per questo, il sistema ha lo scopo di dare all'operatore la possibilità di posizionare e orientare il gripper collegato al robot nella maniera che ritiene più appropriata per afferrare l'oggetto. Per posizionare e orientare il gripper l'applicazione sviluppata mette a disposizione quattro modalità di controllo: posizionamento, orientazione gripper, rotazione gripper, traslazione gripper lungo l'asse z.

**Posizionamento**
Questa modalità di comando permette di muovere il gripper nelle direzioni degli assi x e y indicati in Figura 3.5 mantenendone costante l'orientazione.

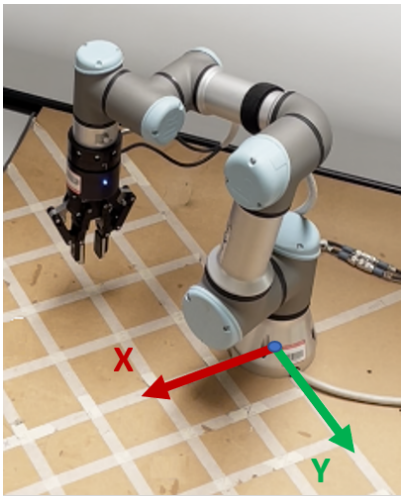



Figura 3.5: Assi x e y del piano di lavoro

Attraverso questo comando è possibile muovere il robot in tutto lo spazio operativo, questa tipologia di controllo risulta quindi utile a spostarsi sul punto in cui è posizionato l'oggetto che si intende prendere.

I comandi in fase di posizionamento devono essere dati come mostrato in Figura 3.6 e 3.7. Un movimento del robot lungo l'asse x è ottenuto flettendo o estendendo il polso. In particolare, estendendo il polso si muove il robot lungo l'asse x con verso negativo mentre flettendo il polso si ottiene un movimento con verso positivo. Per cambiare la posizione del gripper lungo l'asse y si può invece ruotare il polso, una rotazione a destra produce un movimento lungo l'asse y con verso positivo mentre ruotando il polso a sinistra si ottiene un movimento nel verso opposto.

La velocità con cui si muove il robot è proporzionale all'inclinazione della mano, maggiore è l'inclinazione maggiore è la velocità di movimento. In questo modo è possibile far eseguire al robot dei rapidi spostamenti per raggiungere velocemente l'oggetto che si vuole afferrare, quando poi la pinza si trova in prossimità dell'oggetto è possibile correggere la sua posizione con movimenti più lenti e precisi.

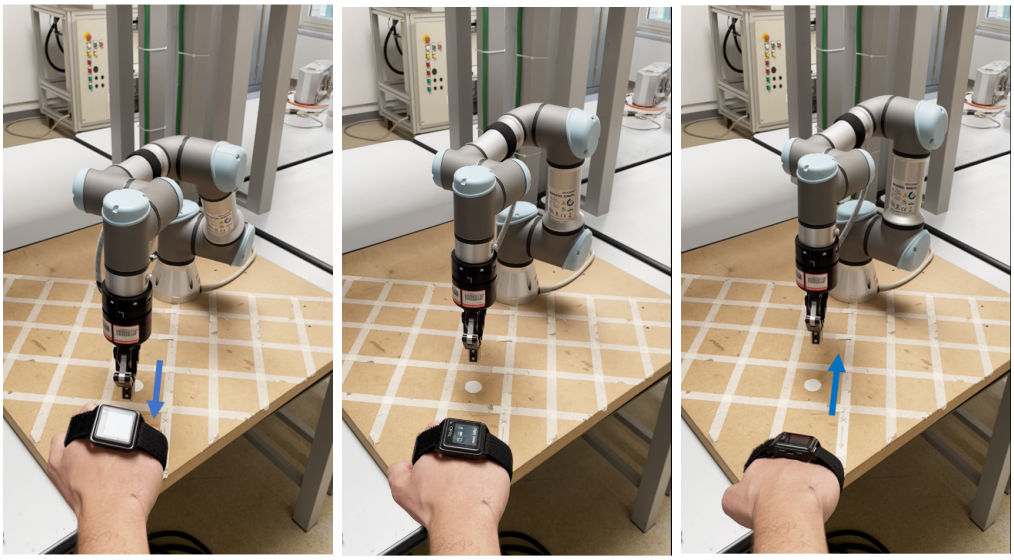


Figura 3.6: Comandi in modalità "Posizionamento", movimento lungo asse x.

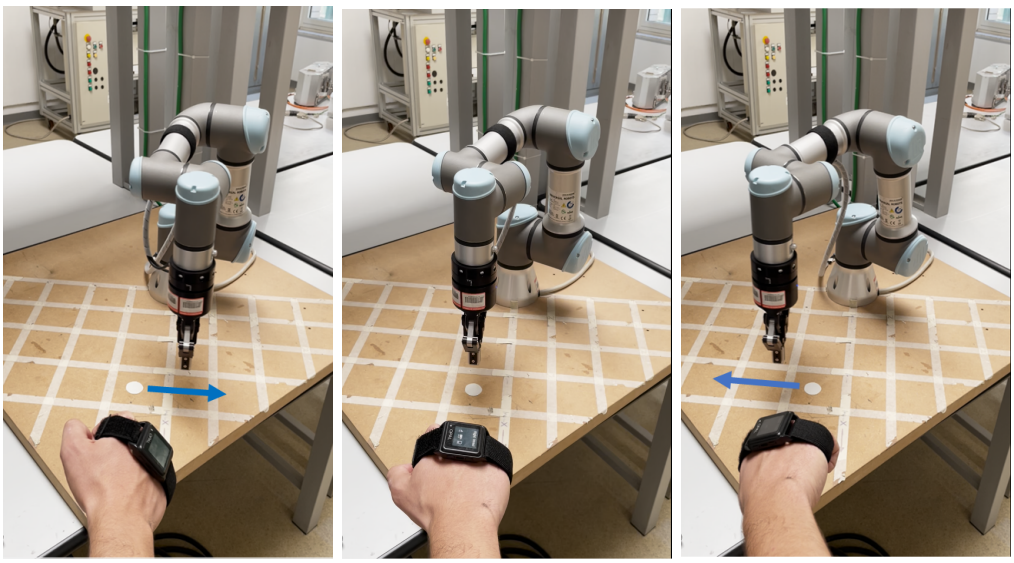


Figura 3.7: Comandi in modalità "Posizionamento", movimento lungo asse y.

**Orientazione gripper**
Attraverso questa modalità di controllo è possibile riorientare il gripper attorno all'oggetto che si vuole afferrare, mantenendolo ad una distanza costante. Il controllo in questo caso è particolarmente intuitivo, infatti il gripper si orienta nello stesso modo in cui è orientata la mano dell'operatore, come mostrato in Figura 3.8.

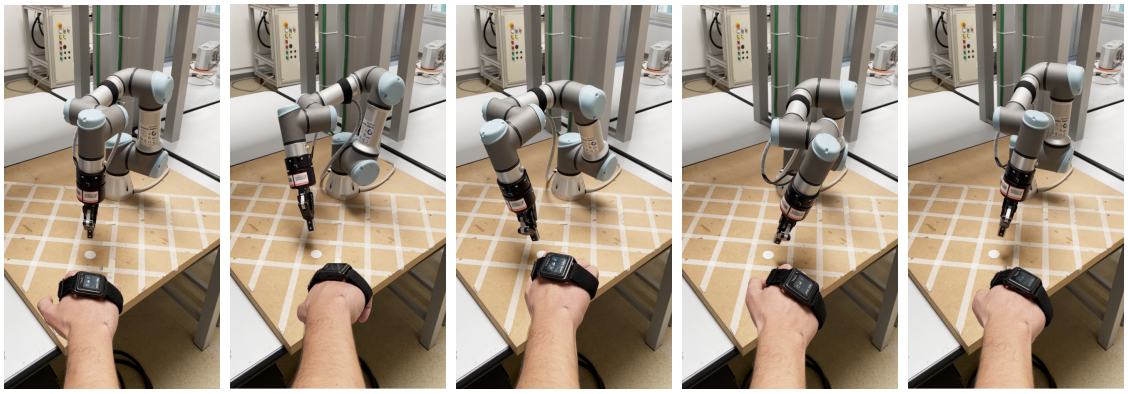


Figura 3.8: Comandi in modalità "Orientazione gripper"

Il controllo dell'orientazione rappresenta sicuramente il punto di forza dell'applicazione sviluppata. Infatti, nel caso di operazioni di pick and place di oggetti con forme non usuali e che possono essere presi solo da posizoni particolari, utilizzare questa modalità di controllo permette di orientare la pinza secondo la direzione in cui può essere effettuato l'afferraggio in maniera rapida e intuitiva in quanto i movimenti della mano dell'operatore vengono riprodotti in tempo reale dal robot.

**Rotazione gripper**
Attraverso questa modalità è possibile ruotare il gripper attorno all'asse z del TCP indicato in Figura 3.9.

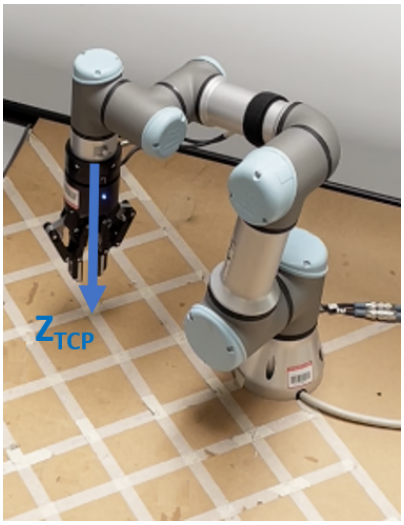



Figura 3.9: Asse z del tool centre point (TCP)

I comandi vanno inviati come mostrato in Figura 3.10. Ruotando il polso a sinistra si comanda una rotazione oraria, al contrario una rotazione del polso verso destra fa ruotare il gripper in senso antiorario. Come nel caso del comando Orientazione gripper, anche questa modalità di controllo ha lo scopo di dare la possibilità di orientare la pinza nel modo più opportuno per effettuare l'operazione di presa.

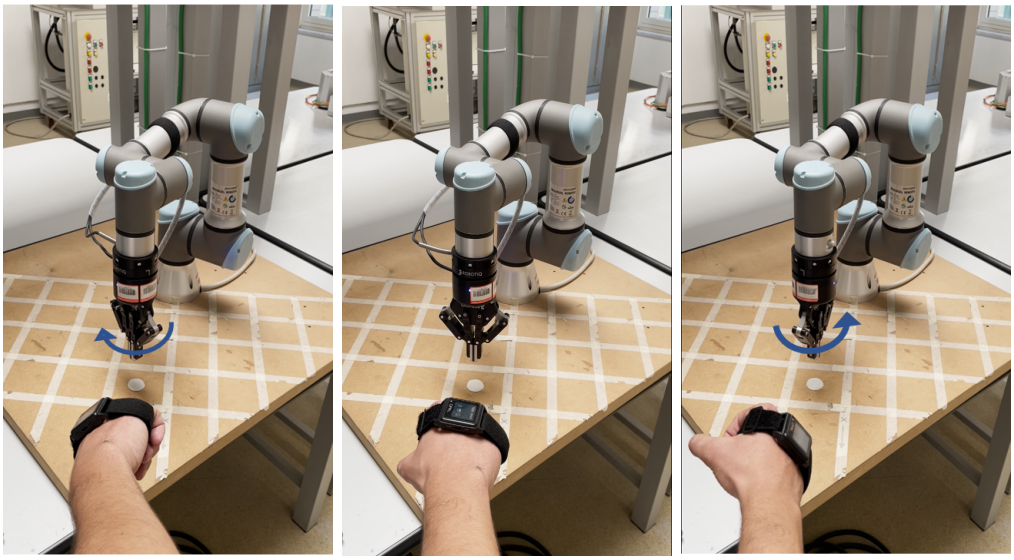


Figura 3.10: Comandi in modalità "Rotazione gripper", rotazione attorno l'asse z del TCP.

**Taslazione gripper lungo l'asse z**
Utilizzando il comando di traslazione è possibile avvicinare la pinza all'oggetto per effettuare il pick. Il moto avviene lungo l'asse z del TCP mostrata in Figura 3.9 con verso positivo. Il comando va effettuato estendendo il polso come indicato in Figura 3.11. Fin quando la mano rimane inclinata il robot continua a muoversi con velocità costante.

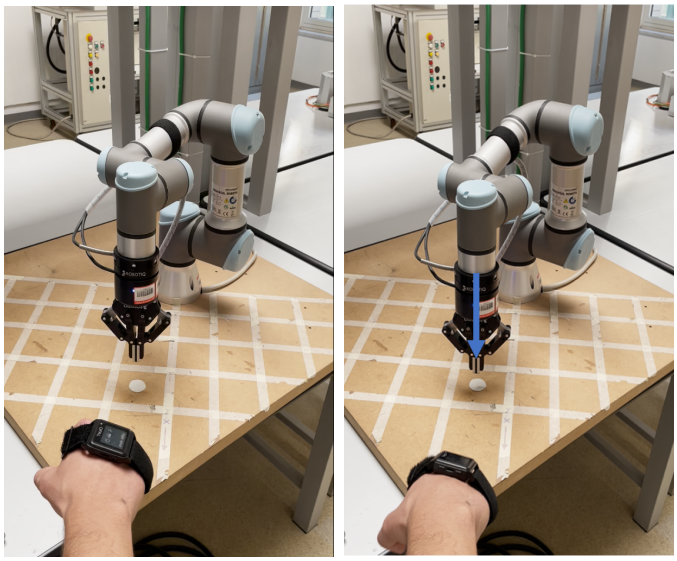


Figura 3.11: Comandi in modalità "Taslazione gripper lungo l'asse z", movimento lungo l'asse z del TCP.

Per gestire il telecontrollo sviluppato, per poter in sostanza passare da una modalità di controllo ad un'altra è stata sviluppata l'interfaccia riportata in Figura 3.12. Si tratta di un terminale Linux in grado di leggere input da tastiera e di convertirli in comandi per l'ambiente ROS che gestisce il telecontrollo. Come si può vedere nell'immagine, ad ogni modalità di controllo è associato un numero da 2 a 6, la modalità 1 consiste nel posizionamento iniziale del robot che avviene un'unica

volta all'avvio del programma. L'operatore può usare quest'interfaccia per cambiare tipologia di comando digitando sulla tastiera il numero corrispondente.

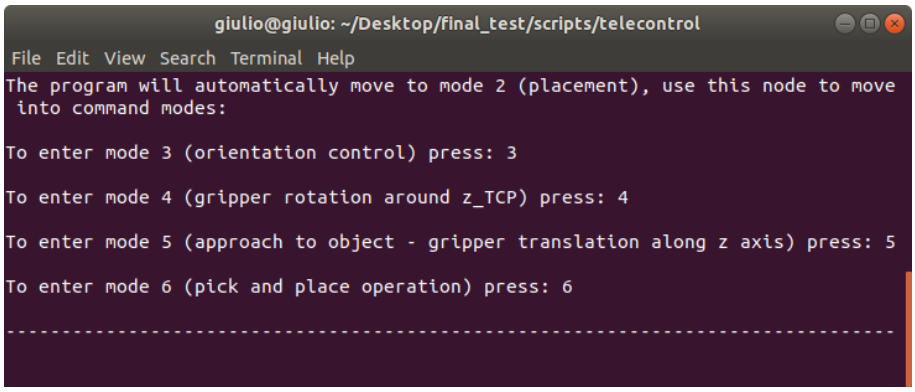



Figura 3.12: Interfaccia per la gestione del telecontrollo.

Tipicamente, per eseguire l'operazione di pick and place, si comanda il robot a partire dalla modalità 2 fino alla modalità 6, in corrispondenza della quale il robot esegue il pick e il place. Tuttavia, è possibile passare da una modalità di comando ad un altra in qualsiasi ordine.

## 3.3 Ambiente ROS sviluppato

La struttura dell'ambiente ROS implementato per realizzare il telecontrollo è schematizzata in Figura 3.13. Gli ovali rappresentano i nodi ROS, i rettangoli i topic, le frecce continue indicano lo scambio di informazioni tra topic e nodi e le frecce tratteggiate indicano le connessioni fisiche con l'hardware. Ad ogni nodo è associato un processo eseguito, la logica dietro il funzionamento di ogni processo e i codici realizzati sono descritti nei paragrafi successivi. L'ambiente ROS è stato interamente sviluppato in Python.

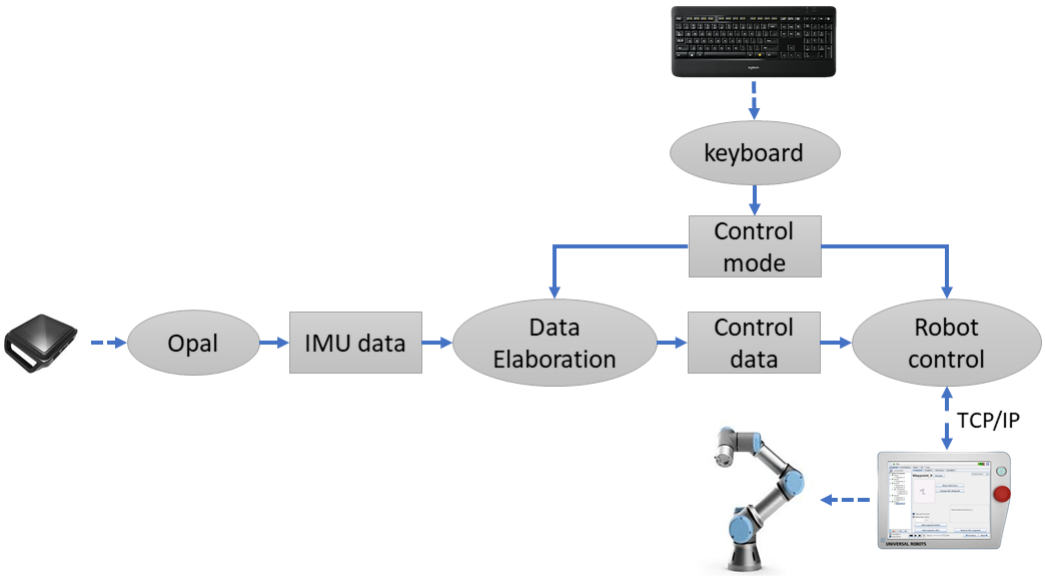



Figura 3.13: Schema dell'ambiente ROS sviluppato per realizzare l'applicazione finale.

### 3.3.1 Nodo Opal

Il nodo Opal realizza la connessione con i sensori e avvia lo streaming dai dati. Secondo la terminologia ROS esso è una Publisher e pubblica messaggi al topic *IMU data* con una frequenza di 200 Hz. Ogni messaggio inviato contiene i dati misurati dal sensore. La tipologia di messaggio utilizzato per questa comunicazione è quella dell'Imu message facente parte del pacchetto sensor_msg di ROS. Un Imu message ha la struttura riportata in Tabella 3.2, questa tipologia di messaggio viene utilizzata per trasmettere le informazioni di orientazione sotto forma di quaternioni, velocità angolari e accelerazioni lineari.

Come riportato precedentemente questo nodo viene eseguito con una frequenza di 200 Hz, ciononostante la massima frequenza di campionamento del sistema Opal è di 128 Hz. La scelta di utilizzare una frequenza di esecuzione del processo più alta della frequenza di campionamento è dovuta al fatto che, con l'hardware utilizzato durante le attività sperimentali, si riesce ad avere una maggiore fluidità e una riduzione di ritardi nella trasmissione dati dal sensore al PC.

Tabella 3.2: Struttura dell'Imu message di ROS.

| Tipo di dato | Tipo di messaggio ROS | Formato |
|---|---|---|
| Quaternioni | geometry_msgs/Quaternion | float64 x<br>float64 y<br>float64 z<br>float64 w |
| Velocità angolari | geometry_msgs/Vector3 | float64 x<br>float64 y<br>float64 z |
| Accelerazioni linerai | geometry_msgs/Vector3 | float64 x<br>float64 y<br>float64 z |

### 3.3.2 Nodo Keyboard

Come descritto nel paragrafo 3.2, l'applicazione sviluppata permette di riorientare il gripper in modo da permettere al braccio robotico di afferrare un oggetto poliedrico ed eseguire un'operazione di pick and place.

Al fine di eseguire questa operazione quattro differenti modalità di controllo del robot sono state sviluppate: posizionamento, orientazione gripper, rotazione gripper, traslazione gripper lungo l'asse z. Le quattro tipologie di comando appena elencate vengono eseguite attraverso comandi differenti. Risulta quindi necessario fornire, a chi utilizza il sistema, la possibilità di passare da una modalità di comando ad un'altra.

Il nodo keybord implementa l'interfaccia su terminale che è riportata in Figura 3.14, utilizzando tale schermata è possibile muoversi ad una modalità di controllo da un altra attraverso degli input da tastiera. Il programma dopo il suo avvio entra in automatico nella modalità 2, premendo "3" si passa a controllo orientazione, il tasto "4" fa entrare nel controllo della rotazione del gripper, con "5" si muove in programma in modalità traslazione lungo l'asse z del TCP e infine al tasto "6" è associata l'operazione di pick and place. Il passaggio dalle modalità dalla 2 alla modalità 5 può essere eseguito senza un ordine preciso, mentre la modalità 6 porta alla conclusione del task e quindi alla chiusura del programma in ROS.

In sintesi, la funzione del nodo keyboard è quella di leggere gli input provenienti dalla tastiera, a cui sono associate le modalità di comando, per poi comunicarli al resto dell'ambiente software.

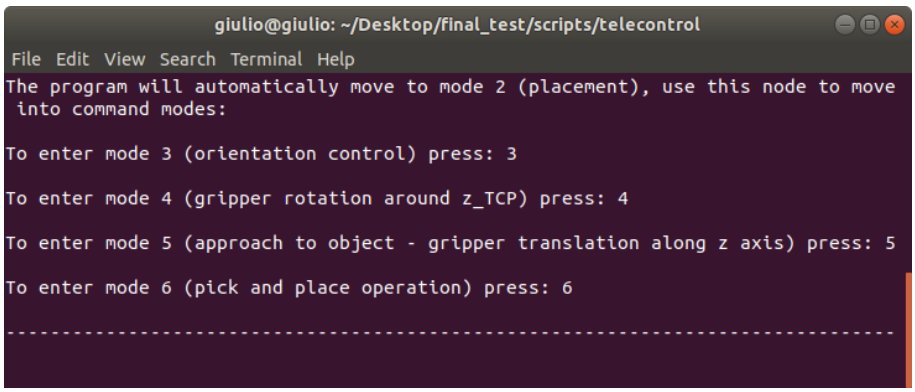



Figura 3.14: Schermata di interfaccia generata dal nodo keybord

Keyboard è un publisher e pubblica messaggi al topic *mode* contenenti un numero passato tramite tastiera. La tipologia di messaggi inviati da keyborad è Int8 contenuta nel pacchetto std_msg di ROS.

### 3.3.3 Nodo Elaborazione dati

Il nodo elaborazione dati ha lo scopo di generare le informazioni necessarie per il controllo del robot a partire dai comandi che l'operatore invia per mezzo dell'IMU posizionato sul dorso della sua mano.

Elaborazione dati è un subscriber dei topic *Imu data* e *control mode* dai quali riceve rispettivamente le letture dei sensori e la modalità di comando del robot, mentre è un publisher del topic *Control data*, topic utilizzato per inviare al nodo *robot control* i dati necessari a controllare l'UR3.

Come riportato precedentemente il robot viene controllato in real-time, per soddisfare questo requisito la tipologia di comando del robot scelta è quella descritta nel paragrafo 2.2.1.4. Tale soluzione è basata su un controllo della posizione del robot eseguito utilizzando la funzione servoj dell'URScript. L'input di servoj è un target di posa nello spazio giunti da raggiungere in 0,008 secondi, quindi lo scopo del nodo elaborazione dati è quello di convertire i dati provenienti del sensore in un'informazione di posa nello spazio operativo. La posa nello spazio operativo verrà poi inviata attraverso il nodo *robot control* al programma Polyscope che controlla il robot. All'interno di Polyscope con un calcolo di cinematica inversa la posa nello spazio operativo sarà tradotta in una posa nello spazio giunti in modo da poter essere fornita in input alla funzione servoj.

I processi svolti dal nodo elaborazione dati, che da un'orientazione del sensore portano ad ottenere una posa del TCP nello spazio operativo, sono sintetizzati nel diagramma in Figura 3.16. All'avvio del nodo, viene definita la matrice che rappresenta la posa del sistema di riferimento TCP rispetto il sistema di riferimento Base nel momento in cui, dopo un posizionamento iniziale, viene lasciato il controllo del robot all'operatore. I due sistemi di riferimento sono indicati in Figura 3.15.

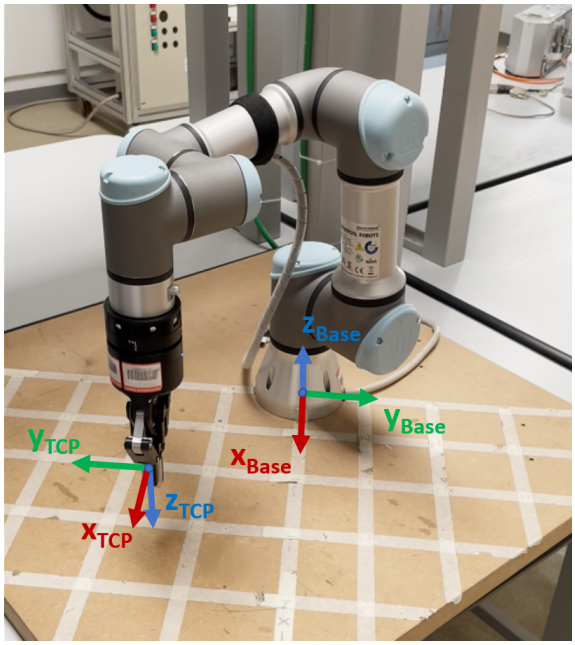



Figura 3.15: Sistemi di riferimento Base e TCP utilizzati all'interno del nodo elaborazione dati.

La matrice di posa del TCP rispetto il sistema di riferimento Base è definita come segue:

$$ {}^{B}A^{0}_{TCP} = \begin{bmatrix} {}^{B}R_{TCP} & {}^{B}\underline{p} \\ \underline{0}^{T} & 1 \end{bmatrix} $$

${}^{B}R_{TCP}$ è una matrice 3x3 che rappresenta l'orientazione del TCP rispetto la Base, mentre ${}^{B}\underline{p}$ è un vettore 1x3 e ne rappresenta la posizione. Questi due elementi sono inizializzati nel seguente modo:

$$ {}^{B}R_{TCP} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} $$

$$ {}^{B}\underline{p} = [0.35, -0.05, 0.065] $$

Intervenendo sul codice che realizza il nodo è possibile modificare l'inizializzazione della matrice ${}^{B}A^{0}_{TCP}$ in modo da avere un differente posizionamento iniziale. Non vi è però particolare necessità di compiere questa operazione in quanto una volta assunto il controllo del robot utilizzando il sensore IMU è possibile orientare e posizionare il TCP in qualunque modo.

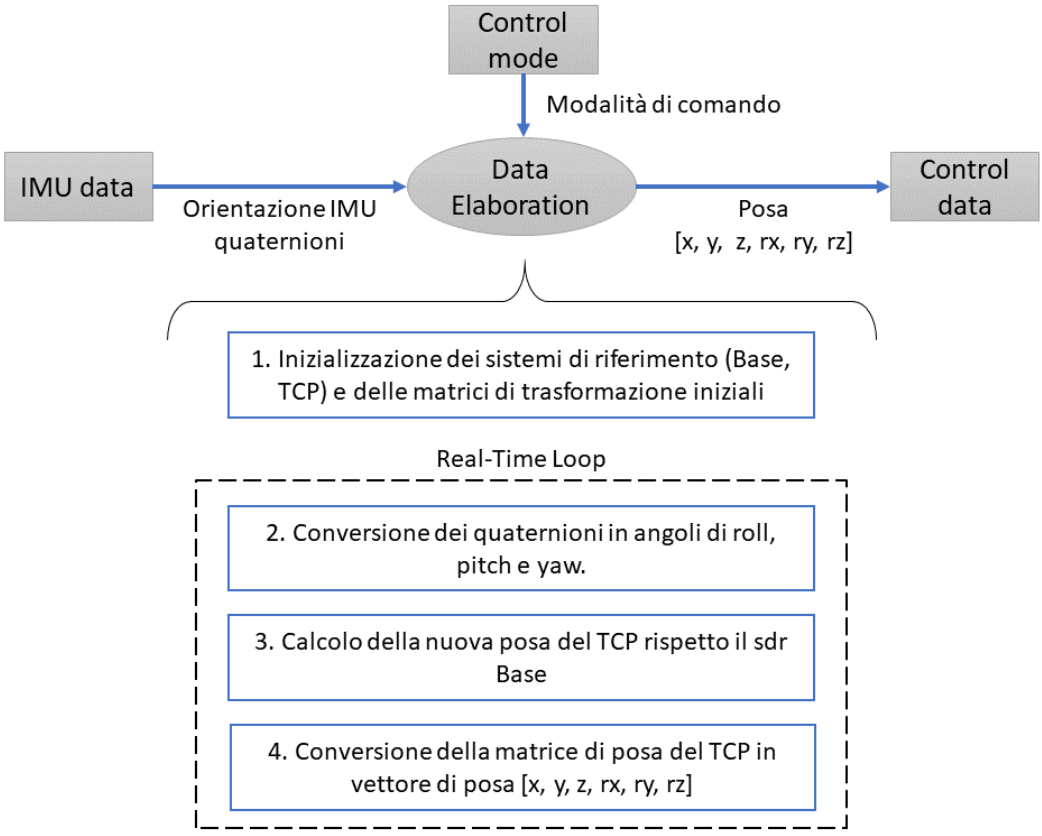



Figura 3.16: Processo di elaborazione dati IMU.

Dopo la fase di inizializzazione, effettuata un'unica volta, il programma entra all'interno di un loop eseguito in real-time con una frequenza di 200 Hz e continua ad eseguirlo fin quando il programma non viene interrotto. All'interno del loop tre

processi vengono realizzati in successione: conversione di quaternioni in angoli di Eulero, calcolo della nuova matrice di posa del TCP, conversione della matrice di posa in vettore di posa.

La conversione dei quaternioni in angoli di Eulero, può essere ottenuta tramite la seguente relazione:

$$\begin{bmatrix} roll \\ pitch \\ yaw \end{bmatrix} = \begin{bmatrix} \arctan \frac{2(q_0 q_1 + q_2 q_3)}{1-2(q_1^2+q_1^2)} \\ \arcsin 2(q_0 q_2 - q_3 q_1) \\ \arctan \frac{2(q_0 q_3 + q_1 q_2)}{1-2(q_2^2+q_3^2)} \end{bmatrix}$$

Tuttavia esistono diversi package Python che forniscono classi e funzioni per rappresentare, diagrammare, convertire e manipolare elementi matematici come matrici di rotazione, matrici di trasformazione, quaternioni etc., utilizzare questi package permette di snellire il codice facilitandone il debug. Nel caso di questa tesi per eseguire la conversione del formato dell'orientazione del sensore da quaternioni ad angoli di Eulero è stata utilizzata la funzione `euler_from_quaternion` del package `transformations`.

Successivamente, all'interno del loop, istante per istante, viene processato il calcolo della nuova matrice di posa del TCP rispetto la base in quanto la posizione del TCP varia in seguito ai comandi ricevuti. Tale calcolo avviene in maniera diversa a seconda della modalità con la quale si sta comandando il robot.

**Calcolo della matrice di posa del TCP in modalità Posizionamento**
L'obiettivo di questa modalità di controllo è quello di eseguire una traslazione del gripper nelle direzioni degli assi x e y illustrati in Figura 3.5. L'operatore invia il comando inclinando la mano o ruotando il polso. L'ambiente ROS interpreta i comandi inviati attraverso una valutazione dell'orientazione del sensore, in particolare degli angoli di roll e pitch. Esistono quattro possibili casi:

- **pitch** $> 10° \rightarrow$ Movimento gripper, proporzionale al modulo dell'angolo misurato, nella direzione delle x positive
- **pitch** $< -10° \rightarrow$ Movimento gripper, proporzionale al modulo dell'angolo misurato, nella direzione delle x negative
- **roll** $> 10° \rightarrow$ Movimento gripper, proporzionale al modulo dell'angolo misurato, nella direzione delle y negative
- **roll** $< -10° \rightarrow$ Movimento gripper, proporzionale al modulo dell'angolo misurato, nella direzione delle y positive

In seguito all'interpretazione del comando, la nuova matrice di posa del TCP ${}^{B}A'_{TCP}$ è calcolata a partire dalla posa attuale, espressa dalla matrice ${}^{B}A_{TCP}$, attraverso la seguente relazione:

$${}^{B}A'_{TCP} = Tras(\underline{s})\ {}^{B}A_{TCP}$$

eseguendo quindi una traslazione del sistema di riferimento TCP del vettore $\underline{s}$ nel sistema di riferimento Base. Il vettore $\underline{s}$ è un vettore 3x1 e i suoi tre elementi variano in base al comando fornito:

- Movimento gripper nella direzione delle x positive → $\underline{s} = [3\text{x}10^{-5} \cdot (pitch - 10°), 0, 0]$
- Movimento gripper nella direzione delle x negative → $\underline{s} = [3\text{x}10^{-5} \cdot (pitch + 10°), 0, 0]$
- Movimento gripper nella direzione delle y positive → $\underline{s} = [0, -3\text{x}10^{-5} \cdot (roll - 10°), 0]$
- Movimento gripper nella direzione delle y negative → $\underline{s} = [0, -3\text{x}10^{-5} \cdot (roll + 10°), 0]$

Successivamente la matrice ${}^{B}A'_{TCP}$ viene convertita nel vettore di posa [x, y, z, rx, ry, rz] che a sua volta sarà inviato al topic Control data.

Inoltre, alla fine di ogni ciclo eseguito con questa modalità di comando viene aggiornata la matrice ${}^{B}A_{Og}$. Questa matrice rappresenta la posa di un sistema di riferimento $Og$ rispetto la Base. Il sistema di riferimento $Og$ ha la stessa orientazione del sistema di riferimento Base ed è posizionato sul piano di lavoro, di conseguenza la coordinata z della sua posizione è pari a zero, mentre le coordinate x e y sono le stesse del sistema di riferimento $TCP$. La matrice ${}^{B}A_{Og}$ è quindi definita come segue:

$$ {}^{B}A_{Og} = \begin{bmatrix} 1 & 0 & 0 & p_{TCP,x} \\ 0 & 1 & 0 & p_{TCP,y} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Quindi il sistema di riferimento $Og$, riportato in Figura 3.17, si muove sul piano di lavoro seguendo i movimenti del sistema di riferimento $TCP$. Durante la modalità di comando orientazione gripper, $Og$ costituirà il sistema di riferimento dell'oggetto che si intende afferrare.

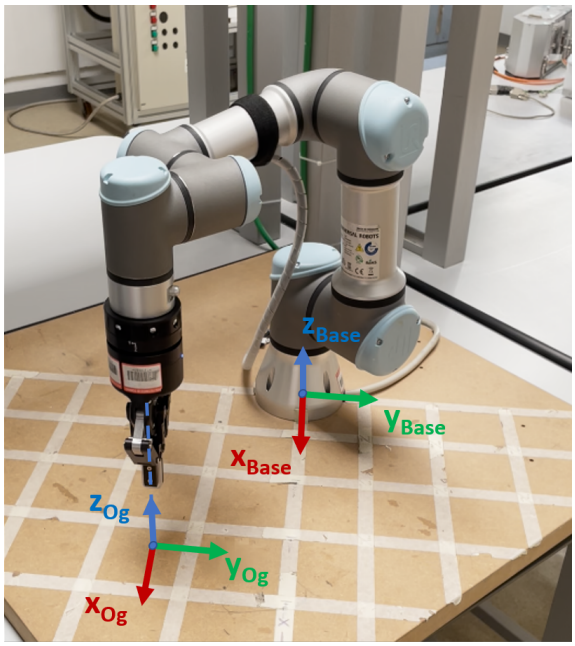



Figura 3.17: Sistema di riferimento $Og$

**Calcolo della matrice di posa del TCP in modalità Orientazione gripper**
L'obiettivo di questa modalità di controllo è quello di riorientare il gripper attorno all'oggetto da afferrare mantenendolo sempre alla stessa distanza. Come descritto nel paragrafo 3.2, il comando viene inviato dall'operatore in maniera molto intuitiva in quanto il robot si orienta allo stesso modo in cui è orientato il sensore sulla mano dell'operatore stesso. All'interno dell'ambiente ROS l'orientazione del sensore viene interpretata dalle letture degli angoli di roll e pitch. Successivamente, per ogni ciclo di calcolo, una nuova posa del $TCP$ viene calcolata eseguendo una doppia rotazione del sistema di riferimento $TCP$ rispetto il sistema di riferimento $Og$. La relazione matematica è la seguente:

$$^{B}A'_{TCP} = ^{B} A_{Og} Rot(y, pitch) Rot(x, roll) ^{Og}A^{0}_{TCP}$$

$^{B}A_{Og}$ indica la posa dell'oggetto rispetto la Base, questa matrice viene aggiornata durante ogni ciclo eseguito in modalità Posizionamento. Invece, la matrice $^{Og}A^{0}_{TCP}$ è un elemento utile ad eseguire questo calcolo, essa è definita come:

$$^{Og}A^{0}_{TCP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & p_{TCP,z} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

dove $p_{TCP,z}$ è l'altezza del TCP rispetto il piano di lavoro nel momento in cui si entra in questa modalità di comando.

Alla fine del ciclo di calcolo, la matrice $^{B}A_{TCP}$ viene convertita nel vettore di posa [x, y, z, rx, ry, rz].

**Calcolo della matrice di posa del TCP in modalità Rotazione gripper**
Questa modalità di comando permette di ruotare il gripper attorno al proprio asse. La rotazione avviene con velocità costante e l'operatore può comandare una rotazione in verso positivo o negativo ruotando la mano rispettivamente a sinistra o a destra. L'interpretazione del comando avviene sulla base delle letture dell'angolo di roll del sensore. In particolare, sono definiti tre casi:

- $-45° <$ **roll** $< 25° \rightarrow$ Comando di stop
- **roll** $< -45° \rightarrow$ Rotazione negativa
- **roll** $> 25° \rightarrow$ Rotazione positiva

Per ogni ciclo di calcolo la posa del TCP viene aggiornata in base al comando ricevuto. La matrice $^{B}A'_{TCP}$ che esprime la nuova posa viene calcolata postmoltiplicando la matrice che esprime la posa del TCP in quel momento $^{B}A_{TCP}$ per una matrice di rotazione rispetto l'asse z.

$$^{B}A'_{TCP} = ^{B}A_{TCP}\ Rot(z, \alpha)$$

L'angolo di rotazione $\alpha$ varia in base al comando fornito:

- Comando di stop $\rightarrow \quad \alpha = 0°$
- Rotazione positiva $\rightarrow \quad \alpha = +0.2°$

- Rotazione negativa $\rightarrow \quad \alpha = -0.2°$

Attraverso questo processo la posa del TCP viene modificata mantenendone fissa la posizione rispetto il sistema di riferimento *Base* e producendo una rotazione attorno al suo asse z.

Infine, da ${}^{B}A'_{TCP}$ viene estratto il vettore di posa $p = [x, y, z, rx, ry, rz]$ da inviare al nodo Controllo robot.

**Calcolo della matrice di posa del TCP in modalità Traslazione gripper lungo l'asse z**

La modalità Traslazione gripper lungo l'asse z permette di muovere il gripper nella direzione del suo asse z con verso positivo. L'operatore comanda questo movimento attraverso un'estensione del polso. Questo comando viene interpretato dall'ambiente ROS attraverso l'angolo di pitch del sensore. In particolare, si hanno i seguenti due casi:

- **pitch** $> -35° \rightarrow$ Comando di stop
- **pitch** $< -35° \rightarrow$ Traslazione

Nel caso in cui si riceva una comando di traslazione la nuova posa del TCP, definita dalla matrice ${}^{B}A'_{TCP}$, è data dalla seguente relazione:

$${}^{B}A'_{TCP} = {}^{B}A_{TCP} Tras(\underline{s})$$

dove ${}^{B}A_{TCP}$ rappresenta la posa attuale rispetto la Base, mentre $\underline{s}$ è il vettore $[0, 0, 1\text{x}10^{-4}]$. Si esegue cioè una traslazione del sistema di riferimento $TCP$ lungo il suo asse z.

Dopo la procedura di calcolo della nuova posa del sistema di riferimento $TCP$, che rappresenta la fase 3 del processo svolto dal nodo elaborazione dati e diagrammato in Figura 3.16, all'interno del loop eseguito in real-time avviene la conversione da matrice a vettore della posa del TCP. Il vettore di posa ha la seguente forma:

$$p = [x, y, z, rx, ry, rz]$$

gli elementi $x, y$ e $z$ rappresentano la posizione del TCP mentre $rx, ry, rz$ rappresentano l'orientazione del TCP nella notazione asse-angolo. La posizione è ricavata estraendo la componente di traslazione da ${}^{B}A'_{TCP}$:

$${}^{B}A'_{TCP} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_1 \\ R_{21} & R_{11} & R_{23} & T_2 \\ R_{31} & R_{32} & R_{33} & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[x, y, z] = [T_1, T_2, T_3]$$

mentre le componenti di orientazione sono calcolate esprimendo la componente di rotazione di ${}^{B}A'_{TCP}$ come una rotazione di angolo $\theta$ rispetto un asse con versore $\hat{h}$, la procedura analitica è la seguente:

$$^{B}A'_{TCP} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_1 \\ R_{21} & R_{11} & R_{23} & T_2 \\ R_{31} & R_{32} & R_{33} & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

angolo di rotazione: $\theta = cos^{-1}(\frac{R_{11}+R_{22}+R_{33}}{2})$

versore asse di rotazione $\hat{h} = [h_x, h_y, h_z]$

$$sin\theta \neq 0 \qquad \Rightarrow \qquad \hat{h} = \frac{1}{2sin\theta} \begin{bmatrix} R_{32} - R_{23} \\ R_{13} - R_{31} \\ R_{21} - R_{12} \end{bmatrix}$$

$$sin\theta = 0 \qquad \Rightarrow \qquad \begin{cases} h_x = \pm\sqrt{\frac{R_{11}+1}{2}} \\ h_y = \pm\sqrt{\frac{R_{22}+1}{2}} \\ h_z = \pm\sqrt{\frac{R_{33}+1}{2}} \\ sgn(h_x h_y) = sgn(R21) \\ sgn(h_x h_z) = sgn(R31) \\ sgn(h_y h_z) = sgn(R32) \end{cases}$$

$[rx, ry, rz] = \theta \cdot \hat{h}$

Per mantenere il codice snello, questi calcoli non sono implementati in forma estesa, piuttosto si è deciso di utilizzare le funzioni e classi del package `spatialmath`.

Infine, l'ultimo passaggio eseguito in ogni loop del nodo elaborazione dati, è l'invio di un messaggio al topic *Control data* contenente il vettore di posa calcolato. La tipologia di messaggio utilizzata per questa comunicazione è Float32MultiArray facente parte del pacchetto std_msg di ROS.

### 3.3.4 Nodo Controllo robot

Il nodo Controllo robot, oltre a contenere il codice necessario a generare le comunicazioni del nodo con il resto dell'ambiente ROS contiene al suo interno uno dei thread che permettono di controllare il robot secondo la modalità di comando descritta nel paragrafo 2.2.1.4. Controllo robot infatti implementa un'interfaccia RTDE attraverso la quale i dati provenienti dal nodo elaborazione dati vengono inviati al programma Polyscope che controlla l'UR3.

La logica di funzionamento di questo nodo è riportata nello schema in Figura 3.18. Il nodo come primo step crea l'interfaccia RTDE con il robot stabilendo una comunicazione TCP/IP attraverso la quale è possibile lo scambio di informazioni con una frequenza pari a 125 Hz. Successivamente è eseguito un setup dell'interfaccia (si stabiliscono quali saranno gli input e gli output della comunicazione), infine si entra in un loop eseguito in real-time che termina solo nel caso in cui si intervenga da tastiera per bloccare il programma o quando il task che si vuole eseguire viene portato a compimento. All'interno del loop i dati provenienti dai topic *Control mode* e *Control data* vengono inviati al programma Polyscope in modo da comunicare la modalità di comando e di fornire gli input alla funzione servoj che attua i movimenti del robot

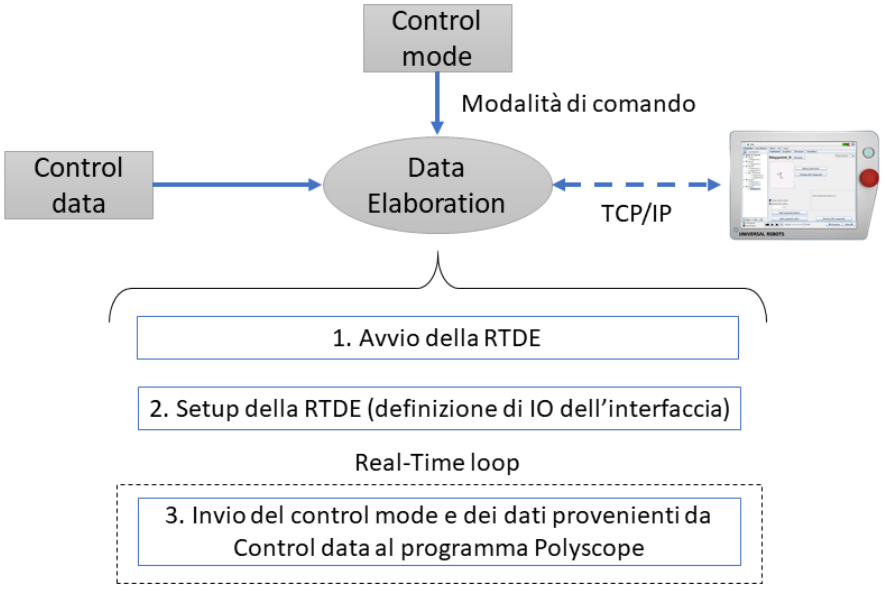



Figura 3.18: Logica di funzionamento del nodo controllo robot

# Capitolo 4

# Prova sperimentale

Una prova sperimentale è stata condotta con il fine di verificare l'efficacia del telecontrollo sviluppato. All'interno di questo capitolo sono descritti il setup della prova ed i risultati ottenuti.

## 4.1 Setup della prova

In Figura 4.1 è riportato l'hardware utilizzato durante la prova, composto da: un PC, il sistema APDM-Opal (Access Point, Docking Station e un sensore inerziale), un robot UR3 prodotto da Universal Robot e il Teach Pendant. Il PC ha un processore i7-10510U, 16 GB di RAM e Windows 10 come sistema operativo. Gli algoritmi sono eseguiti in Ubuntu 18.04 installato su VMware Workstation Player 16, una workstation a cui sono dedicati 6 GB di RAM e 8 core del processore. La versione di ROS utilizzata è ROS Melodic. Il sensore inerziale è configurato per effettuare uno streaming a 128 Hz di accelerazioni lineari, velocità angolari, campo magnetico e per inviare informazioni sulla sua orientazione sotto forma di quaternioni. Il sensore è posizionato sul dorso della mano dell'operatore con il monitor rivolto verso l'alto e con l'ingresso micro-usb diretto verso il polso (Figura 4.2). Il robot è fissato su un piano in legno ed è equipaggiato con un ROBOTIQ 2F-85 Gripper. Attraverso il Teach Pendant, alle estremità del piano sono state posizionate delle barriere virtuali che il robot non può oltrepassare, in questo modo è garantita la sicurezza dell'operatore durante la prova. Il PC e la control box dell'UR3 sono connessi via Ethernet e comunicano utilizzando un protocollo TCI/IP.


Figura 4.1: Hardware della prova sperimentale.

Figura 4.2: Posizione del sensore inerziale durante il telecontrollo.

## 4.2 La prova

Il test viene eseguito con lo scopo di teleguidare il robot durante un'operazione di pick and place. L'oggetto da afferrare e il target su cui posizionarlo sono indicati in Figura 4.3. Per comandare il robot con le modalità descritte nel paragrafo 3.2 è necessario posizionarsi come mostrato in Figura 4.3 e mantenere il braccio con il sensore in direzione dell'asse x del sistema di riferimento *Base*. Inoltre, il PC si trova vicino l'operatore ed è in una posizione comoda per utilizzare la tastiera.

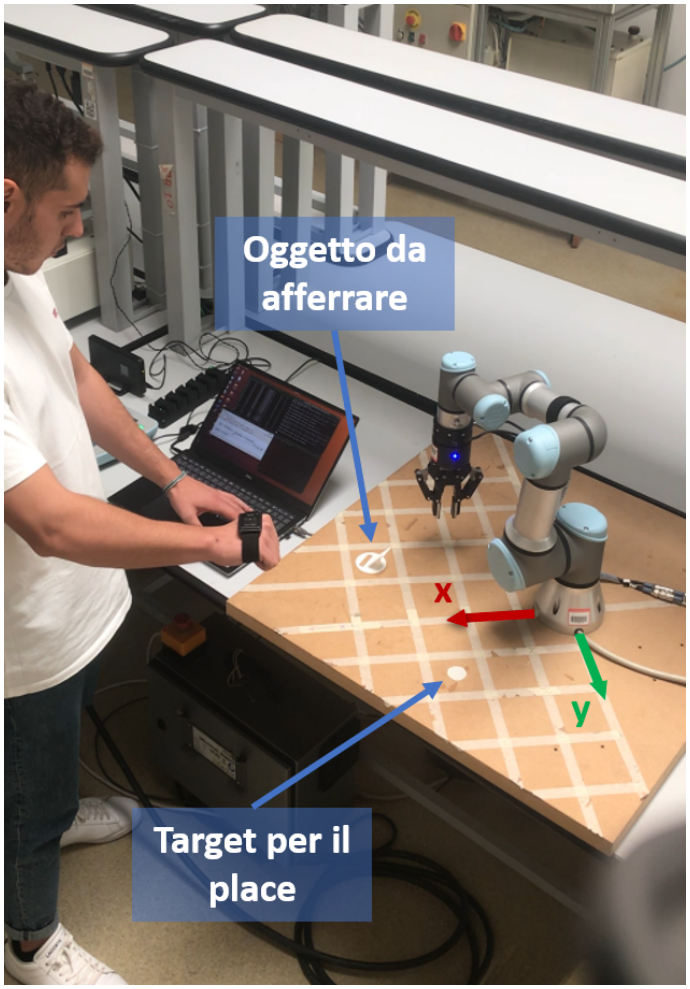



Figura 4.3: Obiettivo della prova.

L'esecuzione della prova inizia con l'avvio dell'ambiente ROS descritto nel paragrafo 3.3. Nel momento in cui tutti i nodi sono stati lanciati il computation graph

di ROS assume la struttura riportata in Figura 4.4 ed il controllo del robot viene lasciato all'operatore.

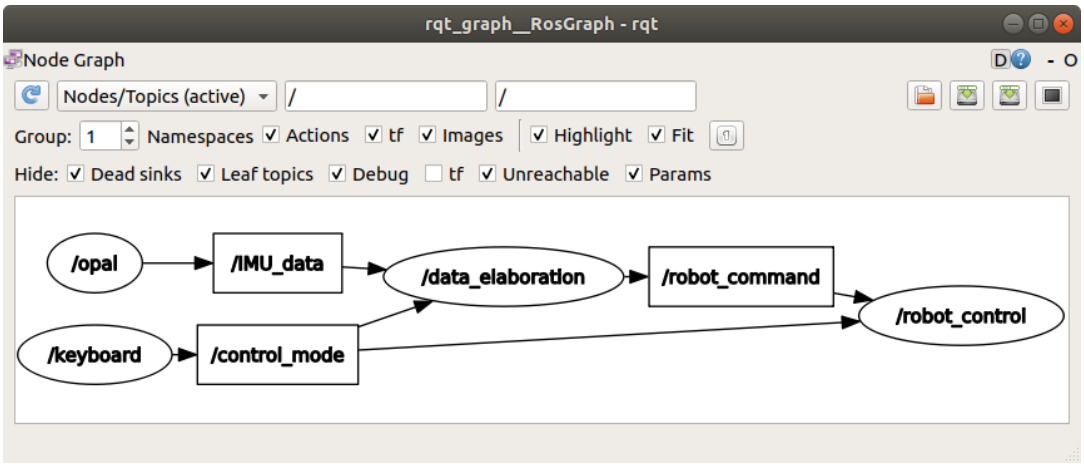



Figura 4.4: Struttura del computation graph di ROS all'avvio del sistema di telecontrollo.

A questo punto viene eseguito il pick and place dell'oggetto sfruttando le differenti modalità di controllo. In Figura 4.5 sono riportati dei frame catturati durante l'esecuzione del test. I sei frame raffigurano le sei fasi attraverso le quali il task è stato portato a termine, in particolare:

- frame 1: avvio del telecontrollo e posizionamento iniziale del robot
- frame 2: posizionamento del gripper sull'oggetto
- frame 3 e 4: riorientazione del gripper eseguita con lo scopo di avere il gripper nella direzione di presa
- frame 5: avvicinamento all'oggetto
- frame 6: pick and place

Per quanto riguarda il place dell'oggetto, questa fase è pre-programmata, per cui è stato il programma caricato sul TP a governare il place senza che vi fosse alcun intervento dell'operatore. Le sei fasi dell'operazione sono state eseguite muovendosi attraverso le differenti modalità di controllo. Il passaggio da una modalità all'altra è avvenuto seguendo le istruzioni fornite dall'interfaccia realizzata dal nodo keyboard e passando gli input tramite la tastiera del PC. L'esecuzione di tutto il processo ha richiesto circa 30 secondi e non si sono riscontrate problematiche. Oltre alla prova appena descritta, sono stati eseguiti ulteriori test in maniera simile, ma con l'oggetto posizionato diversamente. Tutte le prove hanno avuto un tempo di esecuzione che varia tra i 25 e i 30 secondi, e hanno dimostrato come il telecontrollo sviluppato dia sempre la possibilità di orientare e posizionare il gripper nella maniera più opportuna all'esecuzione dell'azione di presa.

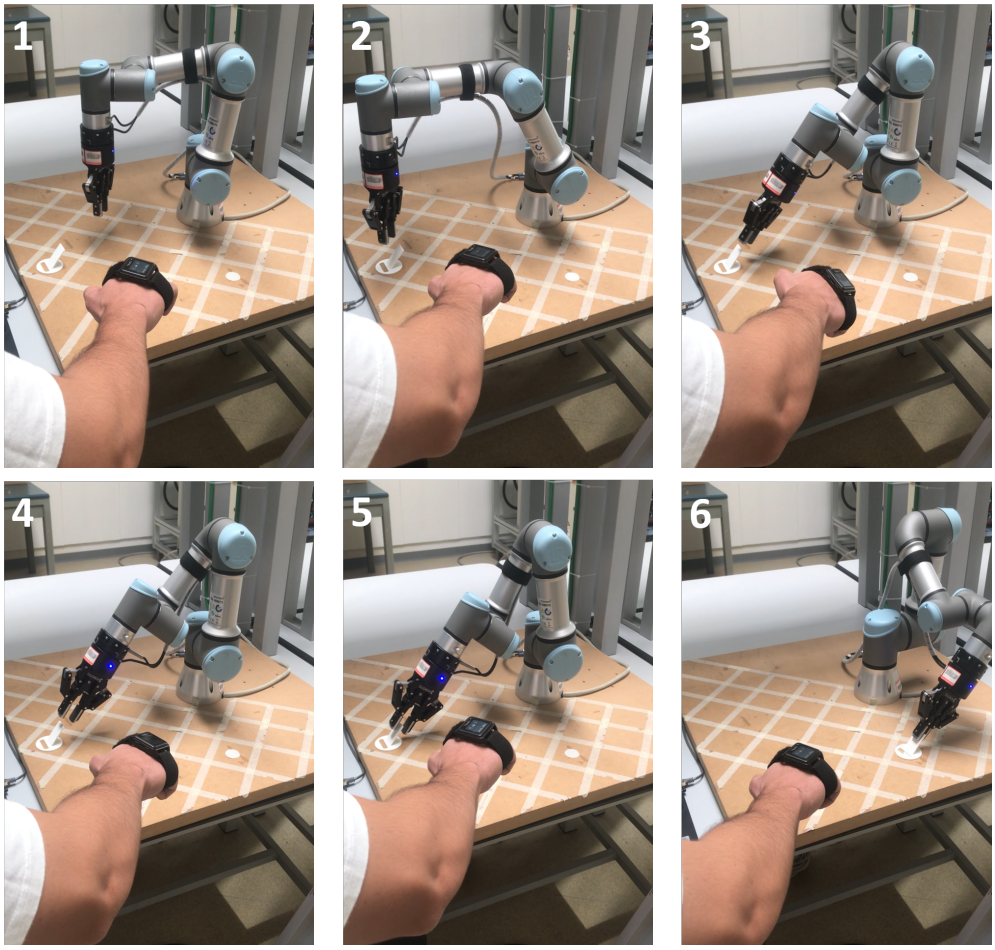



Figura 4.5: Frame della prova sperimentale

# Capitolo 5

# Conclusione

Negli ultimi anni, la cooperazione tra uomo e robot sta assumendo un ruolo sempre più centrale nell'ambito della produzione industriale. La flessibilità e la facilità di programmazione rappresentano i punti di forza dei cobot. Inoltre, creare sistemi in cui uomini e robot collaborino permette di sfruttare a pieno le capacità cognitive dell'uomo, lasciando ai robot le operazioni più ripetitive. Per questi motivi, i sistemi di robotica collaborativa sono sempre più integrati all'interno di catene produttive che includono sempre più varianti del prodotto (e persino prodotti specifici per il singolo cliente) oltre dimensioni dei lotti più ristrette e cicli di vita dei prodotti più brevi. Di conseguenza, oggigiorno, i cobot impegnati in ambito industriale lavorano su oggetti con forme differenti tra loro e in ambienti in continua evoluzione. Nasce quindi l'esigenza di sviluppare delle interfacce che permettano agli operatori di guidare i cobot, nell'esecuzione dei vari task, in maniera rapida e intuitiva.

Quindi, dal momento che l'interpretazione dei movimenti dell'operatore rappresenta uno dai modi più efficaci per comunicare con un robot, il lavoro di tesi si è focalizzato sullo sviluppo di un telecontrollo per cobot basato sull'interpretazione dell'orientazione della mano dell'operatore. All'interno dell'elaborato, viene presentato l'ambiente software sviluppato per gestire il sistema integrato, che è costituito da una tastiera, un IMU e un cobot. L'obiettivo finale dell'applicazione proposta consiste nel realizzare il telecontrollo del robot in operazioni di pick and place. Oltre alla descrizione dell'hardware utilizzato, la tesi contiene anche la descrizione delle logiche con cui sono state sviluppate le comunicazioni con il sensore e i comandi per robot. In particolare, uno dei focus ha riguardato la modalità di comando del robot, dimostrando con dati sperimentali come tale modalità fosse in grado di garantire un controllo in real-time e movimenti fluidi del robot. I risultati sperimentali ottenuti hanno evidenziato l'efficacia del telecontrollo e come esso soddisfi i requisiti che più spesso sono richiesti a un'interfaccia uomo-robot, come l'esecuzione dei comandi in tempo reale e la facilità di utilizzo. Inoltre, si può evidenziare come le quattro modalità di controllo testate durante le attività sperimentali siano sempre state sufficienti a posizionare e orientare il robot nella maniera più opportuna per effettuare l'azione di presa.

Possibili sviluppi futuri riguardano la modalità di gestione del telecontrollo da parte dell'operatore. Allo stato attuale, per cambiare tipologia di comando è necessario inviare un input tramite la tastiera. Quest'azione obbliga l'operatore a distogliere lo sguardo dal robot e quindi lo distrae dal task che sta compiendo. Sarebbe per cui opportuno dotare il sistema di telecontrollo di strumenti che permettano a chi lo usa di rimanere focalizzato sull'ambiente di lavoro del robot. A tal proposito,

si può investigare la possibilità di dotare l'operatore di un secondo IMU da posizionare ad esempio sull'altra mano. In questo modo, si potrebbero interpretare i gesti eseguiti con una mano come i comandi per il robot e i movimenti dell'altra mano come input per selezionare la modalità di comando. Un'ulteriore opzione potrebbe essere quella di utilizzare un sistema di riconoscimento vocale, che offrirebbe il vantaggio di non costringere l'operatore a effettuare movimenti che non siano quelli necessari al comando del robot, permettendogli così di rimanere concentrato. Attualmente, i sistemi di riconoscimento vocale in ambito industriale sono molto pochi e non vengono utilizzati principalmente per motivi legati all'inquinamento acustico e alle conseguenze non trascurabili che un errore nella comunicazione potrebbe avere. Nonostante ciò, sono numerosi gli studi portati avanti con lo scopo di aumentare le performance dei sistemi di riconoscimento vocale in commercio. In [24] viene analizzato il problema dell'inquinamento acustico durante il controllo di robot industriali e viene presentata una metodologia di miglioramento del segnale in grado di incrementare le performance di sistemi di riconoscimento della voce disponibili in commercio. Inoltre, sarebbe di particolare interesse l'integrazione dell'applicazione sviluppata in sistemi di realtà aumentata, sempre più diffusi in ambito industriale. In questo modo si potrebbe ottenere un controllo del robot eseguito da remoto. Infine, si potrebbe pensare di sviluppare ulteriormente il sistema proposto in modo che l'operatore possa guidare il robot anche nella fase di place e non soltanto in quella di pick.

# Bibliografia


[1] Gašper Škulj, Rok Vrabič e Primož Podržaj. "A Wearable IMU System for Flexible Teleoperation of a Collaborative Industrial Robot". In: *Sensors* 21.17 (2021), p. 5871.

[2] Federico Vicentini. "Collaborative Robotics: A Survey". In: *Journal of mechanical design* (2021). DOI: https://doi.org/10.1115/1.4046238.

[3] B.S.K.K. Ibrahim F. Sherwani Muhammad Mujtaba Asad. "Collaborative Robots and Industrial Revolution 4.0". In: *International Conference on Emerging Trends in Smart Technologies (ICETST)* (2020). DOI: 10.1109/ICETST49965.2020.9080724.

[4] Ana M Djuric, RJ Urbanic e JL Rickli. "A framework for collaborative robot (CoBot) integration in advanced manufacturing systems". In: *SAE International Journal of Materials and Manufacturing* 9.2 (2016), pp. 457–464.

[5] Valeria Villani et al. "Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications". In: *Mechatronics* 55 (2018), pp. 248–266.

[6] S Lichiardopol. "A survey on teleoperation". In: *Technische Universitat Eindhoven, DCT report* 20 (2007), pp. 40–60.

[7] Cassie Meeker e Matei Ciocarlie. "EMG-controlled non-anthropomorphic hand teleoperation using a continuous teleoperation subspace". In: *2019 International Conference on Robotics and Automation (ICRA)*. IEEE. 2019, pp. 1576–1582.

[8] Matteo Macchini et al. "Does spontaneous motion lead to intuitive Body-Machine Interfaces? A fitness study of different body segments for wearable telerobotics". In: *arXiv preprint arXiv:2011.07591* (2020).

[9] Alessandro Filippeschi et al. "Survey of Motion Tracking Methods Based on Inertial Sensors: A Focus on Upper Limb Human Motion". In: *Sensors* 17.6 (2017). ISSN: 1424-8220. DOI: 10.3390/s17061257. URL: https://www.mdpi.com/1424-8220/17/6/1257.

[10] Chenguang Yang et al. "Personalized variable gain control with tremor attenuation for robot teleoperation". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48.10 (2017), pp. 1759–1770.

[11] Darong Huang et al. "Disturbance observer enhanced variable gain controller for robot teleoperation with motion capture using wearable armbands". In: *Autonomous Robots* 44.7 (2020), pp. 1217–1231.

[12] Xsens Technologies B.V. *MTi and MTx User Manual and Technical Documentation*. 2009. URL: https://www.xsens.com/.

[13] Xsens Technologies B.V. *MT Low-Level Communication Protocol Documentation*. 2010. URL: https://www.xsens.com/hubfs/Downloads/Manuals/MT_Low-Level_Documentation.pdf.

[14] *CB3 Family - Collaborative robots by Universal Robots*. URL: https://www.universal-robots.com/cb3/.

[15] *Universal Robots - UR3 Technical Specifications*. URL: https://www.universal-robots.com/media/240787/ur3_us.pdf.

[16] *Universal Robots - Offline Simulator - CB Series*. URL: https://www.universal-robots.com/download/software-cb-series/simulator-non-linux/offline-simulator-cb-series-non-linux-ursim-3143/.

[17] *Universal Robots - Script Manual - CB Series*. URL: https://www.universal-robots.com/download/manuals-cb-series/script/script-manual-cb-series-sw3154/.

[18] *Universal Robots - Remote control via TCP/IP*. URL: https://www.universal-robots.com/articles/ur/interface-communication/remote-control-via-tcpip/.

[19] *Universal Robots - Real-Time Data Exchange (RTDE) Guide*. URL: https://www.universal-robots.com/articles/ur/interface-communication/real-time-data-exchange-rtde-guide/.

[20] *Universal Robots - Overview of client interfaces*. URL: https://www.universal-robots.com/articles/ur/interface-communication/overview-of-client-interfaces/.

[21] *Pinze modello 2F-85 e 2F-140 - ROBOTIQ*. URL: https://robotiq.com/it/prodotti/pinze-modello-2f-85-e-2f-140?ref=nav_product_new_button.

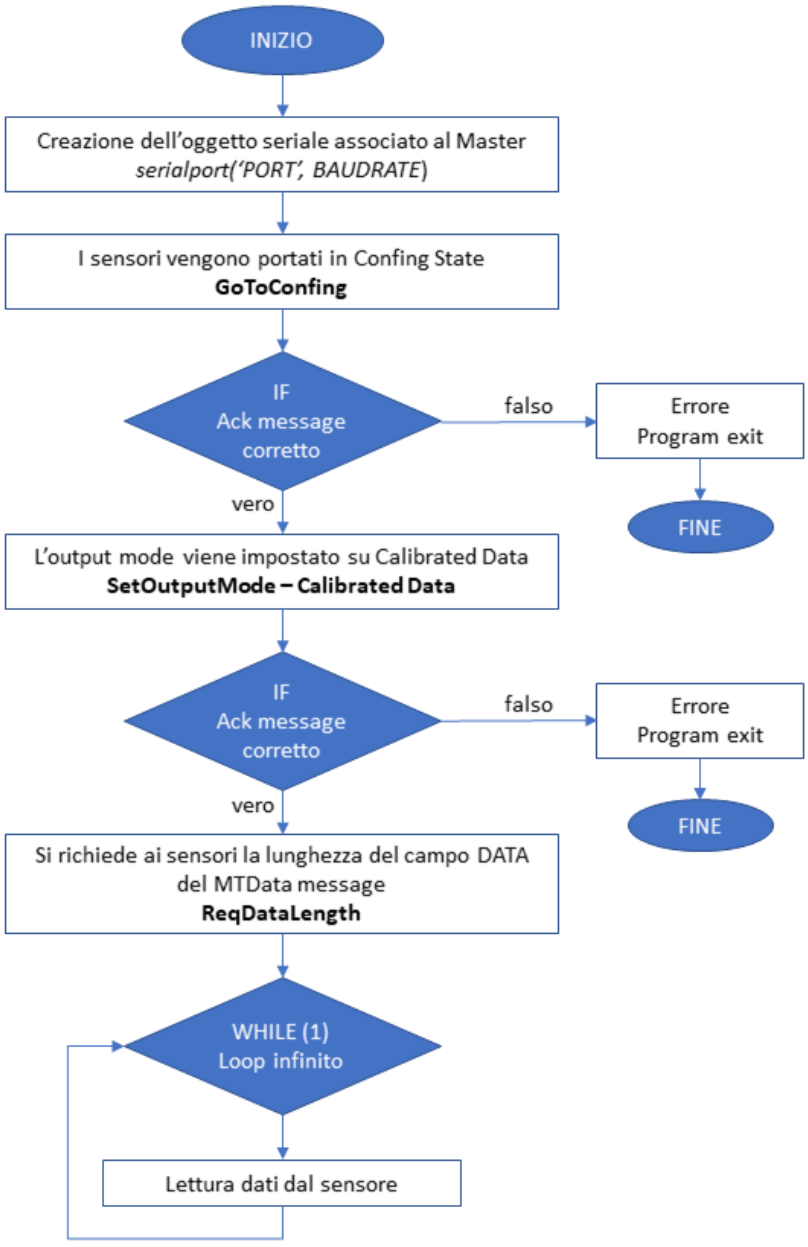
[22] *ROS/Introduction - ROS Wiki*. 2018. URL: http://wiki.ros.org/ROS/Introduction.

[23] Bipin; Kumar. *Robot Operating System Cookbook: Over 70 Recipes to Help You Master Advanced ROS Concepts*. Packt Publishing, Limited, 2018. ISBN: 1783987448.

# Appendice A

# Xsens-MTx Matlab R2020a scripts

**Streaming Xsens-MTx calibrated data** - Il codice che segue è un esempio di utilizzo del Low Level Comunication Protocol per realizzare uno streaming dei dati misurati da due MTx in calibrated data output mode.



```
close all; clear; clc

%% collegamento sensore
MT = serialport('COM3',115200);
fprintf('MT collegato“n“n')
pause(2)

%% MT set output mode in calibrated data
MT.flush
%GoToConfing
MT.flush; %Clear the buffers of the serialport device
write(MT,hex2dec(['FA';'FF';'30';'00';'D1']),'uint8'); %GotoConfig
    message
ack_dec = read(MT,5,'uint8'); % read Acknowledgment in dec
ack_hex = dec2hex(ack_dec); %Acknowledgment in hex

ack_hex_MT = ['FA';'FF';'31';'00';'D0']; %GotoConfig ack from
    comunication protocol
if strcmp(reshape(ack_hex',[1,10]),reshape(ack_hex_MT',[1,10]))
    fprintf('Config State“n“n')
https://www.overleaf.com/project/62a2119eff57d6840d5bb20felse
    error('Not in configuration mode. Try to clear the buffers of
    the serialport device using the flush method')
end

pause(2)

%SetOutputMode
%sensor 01
write(MT,hex2dec(['FA';'01';'D0';'02';'00';'02';'2B']),'uint8'); %
    SetOutputMode - Calibrated Data message
ack_dec = read(MT,5,'uint8'); %read Acknowledgment from MT in dec
ack_hex = dec2hex(ack_dec); %Acknowledgment in hex
ack_hex_MT = ['FA';'01';'D1';'00';'2E']; %SetOutputMode ack from
    comunication protocol
if strcmp(reshape(ack_hex',[1,10]),reshape(ack_hex_MT',[1,10]))
    fprintf('Sensor 01 Calibrated data output mode set“n')
else
    error('Not in calibrated data mode. Try to clear the buffers of
    the serialport device using the flush method')
end
%sensor 02
write(MT,hex2dec(['FA';'02';'D0';'02';'00';'02';'2A']),'uint8'); %
    SetOutputMode - Calibrated Data message
ack_dec = read(MT,5,'uint8'); %read Acknowledgment from MT in dec
ack_hex = dec2hex(ack_dec); %Acknowledgment in hex
ack_hex_MT = ['FA';'02';'D1';'00';'2D']; %SetOutputMode ack from
    comunication protocol
if strcmp(reshape(ack_hex',[1,10]),reshape(ack_hex_MT',[1,10]))
    fprintf('Sensor 02 Calibrated data output mode set“n')
else
    error('Not in calibrated data mode. Try to clear the buffers of
    the serialport device using the flush method')
end

%ReqDataLength
%sensor 01
txdata_dec = hex2dec(['FA';'01';'0A';'00';'F5']);
write(MT,txdata_dec,'uint8');
```

```
rxdata_dec = read(MT,7,'uint8');
dataLength_01 = rxdata_dec(6);
fprintf('Sensor 01 DATA field length: %i bytes"n', dataLength_01)
%sensor 02
txdata_dec = hex2dec(['FA';'02';'0A';'00';'F4']);
write(MT,txdata_dec,'uint8');
rxdata_dec = read(MT,7,'uint8');
dataLength_02 = rxdata_dec(6);
fprintf('Sensor 02 DATA field length: %i bytes"n"n', dataLength_02)
pause(3)
%% Measurement State 50Hz
disp('GoToMeasurement')
pause(2)
messageLength = 7 + dataLength_01 + dataLength_02; %Complete
    message is composed by Preamble, BID, MID, LEN, TimeSample and
    Checksum (7 byte) plus DATA field
data_dec = zeros(1,messageLength); %data array initialization

write(MT,hex2dec(['FA';'FF';'10';'00';'F1']),'uint8'); %
    GoToMeasurement message
data_dec(:) = read(MT,messageLength,'uint8')'; %read data from the
    MT sensor

while(1)
    MT.flush
    data_dec(:) = read(MT,messageLength,'uint8')'; %read data from
    the MT sensor
    %data interpretation sensor 01
    accX_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (7:10))',[1,8]))),'single');
    accY_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (11:14))',[1,8]))),'single');
    accZ_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (15:18))',[1,8]))),'single');
    gryX_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (19:22))',[1,8]))),'single');
    gryY_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (23:26))',[1,8]))),'single');
    gryZ_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (27:30))',[1,8]))),'single');
    magX_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (31:34))',[1,8]))),'single');
    magY_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (35:38))',[1,8]))),'single');
    magZ_01 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (39:42))',[1,8]))),'single');
    %data interpretation sensor 02
    accX_02 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (43:46))',[1,8]))),'single');
    accY_02 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (47:50))',[1,8]))),'single');
    accZ_02 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (51:54))',[1,8]))),'single');
    gryX_02 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (55:58))',[1,8]))),'single');
    gryY_02 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (59:62))',[1,8]))),'single');
    gryZ_02 = typecast(uint32(hex2dec(reshape(dec2hex(data_dec
    (63:66))',[1,8]))),'single');
```

```
90      magX'02 = typecast(uint32(hex2dec(reshape(dec2hex(data'dec
        (67:70))',[1,8]))),'single');
91      magY'02 = typecast(uint32(hex2dec(reshape(dec2hex(data'dec
        (71:74))',[1,8]))),'single');
92      magZ'02 = typecast(uint32(hex2dec(reshape(dec2hex(data'dec
        (75:78))',[1,8]))),'single');
93
94      fprintf('accX'01 = %.2f     accY'01 = %.2f      accZ'01 = %.2f
            accX'02 = %.2f      accY'02 = %.2f      accZ'02 = %.2f"n'
        ,...
95              accX'01,accY'01,accZ'01,accX'02,accY'02,accZ'02)
96  end
```

# Appendice B

# APDM-Opal Python 2.7 scripts

**Autoconfigure APDM-Opal** - Il codice che segue è un esempio di utilizzo del SDK rilasciato da APDM per configurare i sensori. Configurando i sensori con questo script viene abilitata la lettura di accelerazioni, velocità angolari e campo magnetico con una frequenza di 128 Hz. Assicurarsi che gli Opal siano collegati alla docking station prima di eseguire la configurazione.

```
import apdm

context = apdm.apdm_ctx_allocate_new_context()

try:
    apdm.apdm_ctx_open_all_access_points(context)

    """ streaming configuration """
    streaming_config = apdm.apdm_streaming_config_t()
    apdm.apdm_init_streaming_config(streaming_config)
    streaming_config.enable_accel = True
    streaming_config.enable_gyro = True
    streaming_config.enable_mag = True
    streaming_config.wireless_channel_number = 80
    streaming_config.output_rate_hz = 128
    r = apdm.
   apdm_ctx_autoconfigure_devices_and_accesspoint_streaming(context
   , streaming_config)
    if r != apdm.APDM_OK:
        raise Exception("Unable to autoconfigure system, r = " +
   apdm.apdm_strerror(r));

finally:
    apdm.apdm_ctx_disconnect(context)
    apdm.apdm_ctx_free_context(context)
```

**Stream data APDM-Opal** - Il codice che segue è un esempio di utilizzo del SDK rilasciato da APDM per per eseguire lo streaming dei dati dai sensori. La modalità di streaming deve essere compatibile con la configurazione dei sensori. Prima di lanciare questo codice bisogna assicurarsi che gli Opal siano scollegati dalla docking station e che lampeggino, con luce verde, in maniera sincronizzata con l'access point.

---

```
import apdm

try:
    num_aps = apdm.uintArray(1)
    return_code = apdm.apdm_ap_get_num_access_points_on_host1(
   num_aps)
    print("return code is " + str(return_code))
    print("Number of APs on host is " + str(num_aps))

    context = apdm.apdm_ctx_allocate_new_context()
    print(context)
    if context == None:
        raise Exception("unable to allocate new context")

    r = apdm.apdm_ctx_open_all_access_points(context)
    if r != apdm.APDM_OK:
        raise Exception("unable to open all access points")

    r = apdm.apdm_ctx_sync_record_list_head(context)
    if r != apdm.APDM_OK:
        raise Exception("unable to sync record head list")

    #DEVICE ID LIST
    numSensors = apdm.uintArray(1)
    retCode = apdm.apdm_ctx_get_expected_number_of_sensors2(context
   , numSensors)
    device_ids = []
    for x in range(numSensors[0]):
        device_ids.append(apdm.apdm_ctx_get_device_id_by_index(
   context, x))

    #Attempt to read data 10000 times. This may not equal the
   number of samples, however
    #as there may not be data available to read each time if
   reading faster than
    #the sampling rate
    for i in range(10000):
        r = apdm.apdm_ctx_get_next_access_point_record_list(context
   );
        if r != apdm.APDM_OK:
            if r != apdm.APDM_NO_MORE_DATA:
                raise Exception("error while getting next access
   point record list")

            apdm.apdm_usleep(50000)
            continue

        raw_record = apdm.apdm_record_t()
        for device_id in device_ids:
            ret = apdm.apdm_ctx_extract_data_by_device_id(context,
   device_id, raw_record);
            print("Received data status code: " + str(ret))
            print("Rec. sync_val64: " + str(raw_record.sync_val64))
```

```
            print("Rec.accl.x.axis.si: " + str(raw.record.
    accl.x.axis.si))
            print("Rec.accl.y.axis.si: " + str(raw.record.
    accl.y.axis.si))
            print("Rec.accl.z.axis.si: " + str(raw.record.
    accl.z.axis.si))
            print("Rec.gyro.x.axis.si: " + str(raw.record.
    gyro.x.axis.si))
            print("Rec.gyro.y.axis.si: " + str(raw.record.
    gyro.y.axis.si))
            print("Rec.gyro.z.axis.si: " + str(raw.record.
    gyro.z.axis.si))

finally:
    # Disconnect handles from AP's
    apdm.apdm.ctx.disconnect(context)

    # Free memory used by context
    apdm.apdm.ctx.free.context(context)
    context = None
```

# Appendice C

# Controllo Robot Python 3 e Polyscope 3.15 scripts

Come descritto nel paragrafo 2.2.1.4, il controllo del robot è basato su l'interazione tra un interfaccia RTDE, implementata su un pc esterno e un programma realizzato sul teach pendant. Il programma polyscope contiene i comandi URScript che realizzano il controllo effettivo del robot, esso è suddiviso in diverse sezioni, i loop, ad ogni loop è associato un tipo di comando del robot o un'operazione/movimento da eseguire. A gestire il programma polyscope è l'applicazione esterna, essa decide quale loop eseguire oltre ad passare i dati necessari all'esecuzione del loop stesso.

Gli script che seguono realizzano un controllo del robot che fa muovere l'UR da un punto A ad un punto C, passando da B, senza variarne l'orientazione del sistema di riferimento TCP. In Figura C.1 sono descritte le modalità con cui avvine il movimento. Il punto B viene raggiunto utilizzano il comando movej(), quindi con una pianificazione della traiettoria esegiota dal controllore del robot, mentre il passaggio da B a C avviene secondo una traiettoria calcolata nell'applicazione esterna e eseguita tramite servoj().

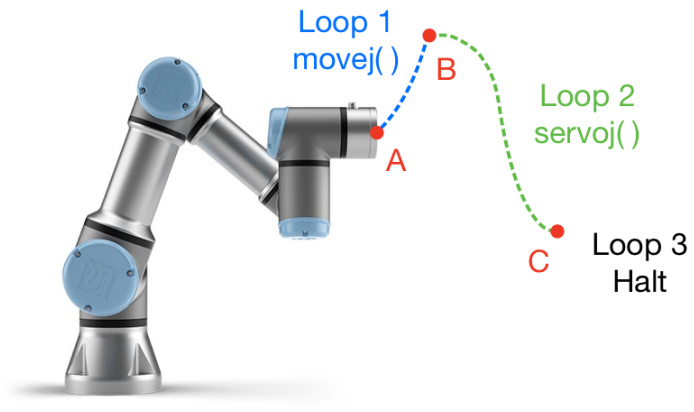
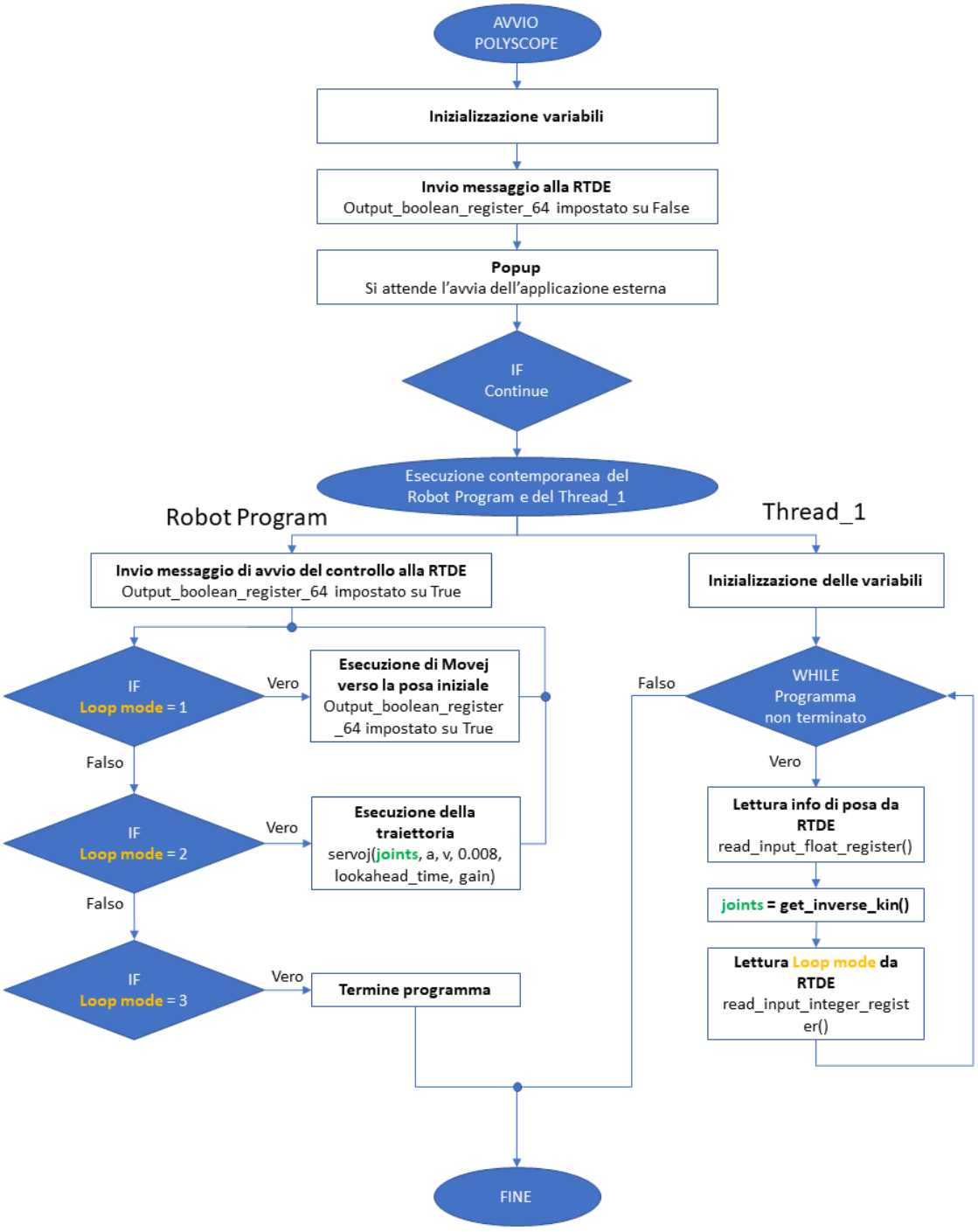



Figura C.1: Descrizione del task eseguito dal robot

La modalità di controllo implementata durante il lavoro della presente tesi è basata sul progetto caricato nella repository GitHub al seguente link https://github.com/danielstankw/Servoj_RTDE_UR5.git.

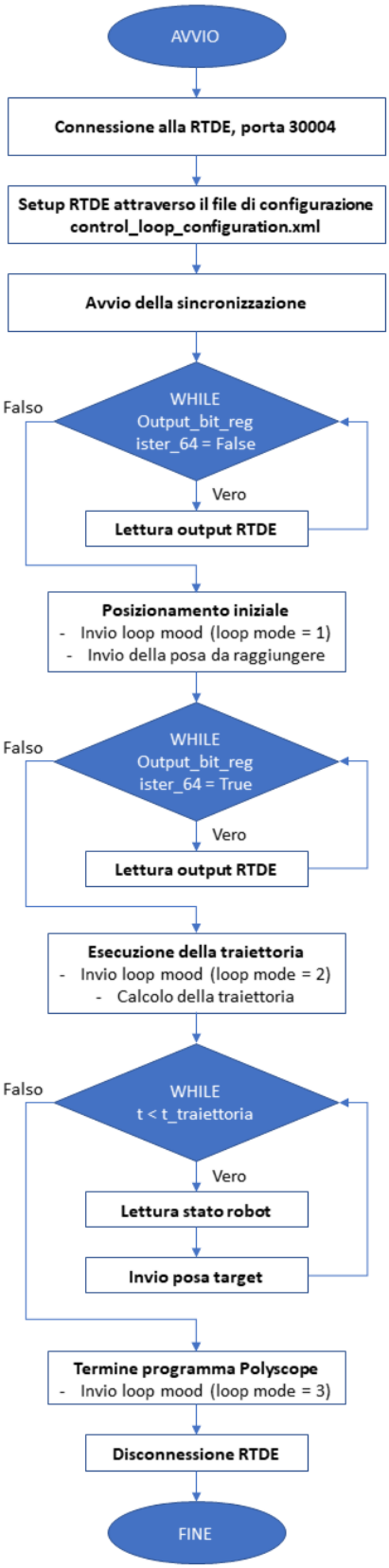
**Translation sample servoj** - File urp da caricare in Polyscope. Questo programma e suddiviso in 3 sezioni:

- BeforeStart, inizializza le variabili
- Robot Program, esegui i principali comandi di controllo come movej() e servoj(), modifica i valori nei registri della RTDE e implementa la logica di controllo
- Thread_1, programma eseguito in parallelo al Robot Program e responsabile della lettura degli input dai registri della RTDE



```
Program
  BeforeStart
    joints:= get_actual_joint_positions()
    write_output_boolean_register(64,  False  )
    Wait: 0.01
    popup("Registers reset, run PC code and then click Continue",
  title="Hi",blocking= True )
    mode:0
  Robot Program
    write_output_boolean_register(64,   True  )
    sync()
    Loop mode=1
      MoveJ
        joints
      write_output_boolean_register(64, False)
      sync()
    Loop mode=2
      servoj(joints, 0, 0, 0.008, lookahead_time, gain)
    Loop mode=3
      stopj(2)
      Halt
    sync()
  Thread_1
    tmp:=[0,0,0,0,0,0]
    gain:=read_input_integer_register(24)
    lookahead_time:=read_input_float_register(24)
    Loop
      tmp[0]=read_input_float_register(0)
      tmp[1]=read_input_float_register(1)
      tmp[2]=read_input_float_register(2)
      tmp[3]=read_input_float_register(3)
      tmp[4]=read_input_float_register(4)
      tmp[5]=read_input_float_register(5)
      joints:=get_inverse_kin(p[tmp[0],tmp[1],tmp[2],tmp[3],tmp
  [4],tmp[5]])
      sync()
      mode:=read_input_integer_register(0)
      sync()
```

**Min_jerk_servoj.py** - File Python che implementa la RTDE, gestisce la logica di controllo e invia input al programma Polyscope.



```
"""
Code based on servoj example from: https://github.com/davizinho5/
    RTDE_control_example
"""
import sys
sys.path.append('')
import logging
import rtde.rtde as rtde
import rtde.rtde_config as rtde_config

import time
import csv
from matplotlib import pyplot as plt
from min_jerk_planner_translation import PathPlanTranslation

plotter = True
save = False

# -------- functions ------------
def setp_to_list(setp):
    temp = []
    for i in range(0, 6):
        temp.append(setp.__dict__["input_double_register_%i" % i])
    return temp


def list_to_setp(setp, list):
    for i in range(0, 6):
        setp.__dict__["input_double_register_%i" % i] = list[i]
    return setp


# ------------- robot communication stuff -----------------
ROBOT_HOST = '192.168.146.129' # polyscope
#ROBOT_HOST = '192.168.56.103' #Lab
ROBOT_PORT = 30004
config_filename = '/home/giulio/Desktop/Servoj_RTDE_UR5-main/
    Servoj_RTDE_UR5-main/control_loop_configuration.xml'  # specify
    xml file for data synchronization

logging.getLogger().setLevel(logging.INFO)

conf = rtde_config.ConfigFile(config_filename)
state_names, state_types = conf.get_recipe('state')  # Define
    recipe for access to robot output ex. joints,tcp etc.
setp_names, setp_types = conf.get_recipe('setp')  # Define recipe
    for access to robot input
watchdog_names, watchdog_types= conf.get_recipe('watchdog')

# -------------------- Establish connection--------------------
con = rtde.RTDE(ROBOT_HOST, ROBOT_PORT)
connection_state = con.connect()

# check if connection has been established
while connection_state != 0:
    time.sleep(0.5)
    connection_state = con.connect()
print("---------------Successfully connected to the robot
    ------------"n")
```

```
# get controller version
con.get_controller_version()

# ------------------- setup recipes -----------------------------
FREQUENCY = 125  # send data in 500 Hz instead of default 125Hz
con.send_output_setup(state_names, state_types, FREQUENCY)
setp = con.send_input_setup(setp_names, setp_types)  # Configure an
    input package that the external application will send to the
   robot controller
watchdog = con.send_input_setup(watchdog_names, watchdog_types)

setp.input_double_register_0 = 0
setp.input_double_register_1 = 0
setp.input_double_register_2 = 0
setp.input_double_register_3 = 0
setp.input_double_register_4 = 0
setp.input_double_register_5 = 0
setp.input_double_register_24 = 0
setp.input_int_register_24 = 0

watchdog.input_int_register_0 = 0

# start data synchronization
if not con.send_start():
    sys.exit()

start_pose = [-0.18, -0.2, 0.15, -0.07, -3.00, 0.10]
desired_pose = [+0.3, -0.25, 0.35, -0.07, -3.00, 0.10]

orientation_const = start_pose[3:]

gain = 2000
lookahead_time = 0.03
setp.input_int_register_24 = gain
setp.input_double_register_24 = lookahead_time
state = con.receive()
tcp1 = state.actual_TCP_pose

print('actual_TCP_pose: ',tcp1, '“n')

#  --------------  mode = 1 (Connection) -------------
print('Boolean 1 is False, please click CONTINUE on the Polyscope“n
    ')
while True:
    state = con.receive()
    con.send(watchdog)
    # print(f”runtime state is -state.runtime_state˜”)
    #if state.output_bit_registers0_to_31 == True:
    if state.output_bit_register_64 == True:
        print('Boolean 1 is True, Robot Program can proceed to mode
    1“n')
        time.sleep(1)
        break

print(”-------Executing moveJ -----------“n”)
time.sleep(1)


watchdog.input_int_register_0 = 1
```

```
con.send(watchdog)  # sending mode == 1
list_to_setp(setp, start_pose)  # changing initial pose to setp
con.send(setp) # sending initial pose

print('Waiting for movej() to finish“n')
while True:
    state = con.receive()
    con.send(watchdog)
    #if state.output_bit_registers0_to_31 == False:
    if state.output_bit_register_64 == False:
        print('Proceeding to mode 2“n')
        time.sleep(1)
        break


print(”-------Executing servoJ  -----------“n”)
print ('servoJ parameters:' +
       '“n' + 'gain: ' + str(gain) +
       '“n' + 'lookahead_time: ' + str(lookahead_time) + '“n')
time.sleep(1)
watchdog.input_int_register_0 = 2
con.send(watchdog)  # sending mode == 2

trajectory_time = 5  # time of min_jerk trajectory
dt = 1/125  # 125 Hz    # frequency

# ------------------ Control loop initialization
    -------------------------

planner = PathPlanTranslation(start_pose, desired_pose,
    trajectory_time)
# ----------- minimum jerk preparation ----------------------


if plotter or save:
    time_plot = []

    min_jerk_x = []
    min_jerk_y = []
    min_jerk_z = []

    min_jerk_vx = []
    min_jerk_vy = []
    min_jerk_vz = []

    px = []
    py = []
    pz = []

    vx = []
    vy = []
    vz = []

#   -------------------------Control loop --------------------
state = con.receive()
tcp = state.actual_TCP_pose
t_current = 0
t_start = time.time()

while time.time() - t_start ¡ trajectory_time:
```

```
    t_init = time.time()
    state = con.receive()
    t_prev = t_current
    t_current = time.time() - t_start

    # print(f"dt:-t_current-t_prev~")
    # read state from the robot
    if state.runtime_state ¿ 1:
        #   ---------- minimum_jerk trajectory -------------
        if t_current ¡= trajectory_time:
            [position_ref, lin_vel_ref, acceleration_ref] = planner
    .trajectory_planning(t_current)

        # ------------------ impedance ----------------------
        current_pose = state.actual_TCP_pose
        current_speed = state.actual_TCP_speed

        pose = position_ref.tolist() + orientation_const

        list_to_setp(setp, pose)
        con.send(setp)

        if plotter or save:
            time_plot.append(time.time() - t_start)

            min_jerk_x.append(position_ref[0])
            min_jerk_y.append(position_ref[1])
            min_jerk_z.append(position_ref[2])

            min_jerk_vx.append(lin_vel_ref[0])
            min_jerk_vy.append(lin_vel_ref[1])
            min_jerk_vz.append(lin_vel_ref[2])

            px.append(current_pose[0])
            py.append(current_pose[1])
            pz.append(current_pose[2])

            vx.append(current_speed[0])
            vy.append(current_speed[1])
            vz.append(current_speed[2])

print(f"It took -time.time()-t_start~s to execute the servoJ")
print(f"time needed for min_jerk -trajectory_time~"n")

state = con.receive()
print('--------------------"n')
print(state.actual_TCP_pose)

# ========================mode 3=========================
watchdog.input_int_register_0 = 3
con.send(watchdog)


con.send_pause()
con.disconnect()


if plotter:
    # ----------- position -------------
    plt.figure()
```

```
        plt.plot(time_plot, min_jerk_x, label="x_min_jerk")
        plt.plot(time_plot, px, label="x_robot")
        plt.legend()
        plt.grid()
        plt.ylabel('Position in x[m]')
        plt.xlabel('Time [sec]')

        plt.figure()
        plt.plot(time_plot, min_jerk_y, label="y_min_jerk")
        plt.plot(time_plot, py, label="y_robot")
        plt.legend()
        plt.grid()
        plt.ylabel('Position in y[m]')
        plt.xlabel('Time [sec]')

        plt.figure()
        plt.plot(time_plot, min_jerk_z, label="z_min_jerk")
        plt.plot(time_plot, pz, label="z_robot")
        plt.legend()
        plt.grid()
        plt.ylabel('Position in z[m]')
        plt.xlabel('Time [sec]')

        # ----------- velocity -------------
        plt.figure()
        plt.plot(time_plot, min_jerk_vx, label="vx_min_jerk")
        plt.plot(time_plot, vx, label="vx_robot")
        plt.legend()
        plt.grid()
        plt.ylabel('Velocity [m/s]')
        plt.xlabel('Time [sec]')

        plt.figure()
        plt.plot(time_plot, min_jerk_vy, label="vy_min_jerk")
        plt.plot(time_plot, vy, label="vy_robot")
        plt.legend()
        plt.grid()
        plt.ylabel('Velocity [m/s]')
        plt.xlabel('Time [sec]')

        plt.figure()
        plt.plot(time_plot, min_jerk_vz, label="vz_min_jerk")
        plt.plot(time_plot, vz, label="vz_robot")
        plt.legend()
        plt.grid()
        plt.ylabel('Velocity [m/s]')
        plt.xlabel('Time [sec]')

        plt.show()

    if save:
        with open('g-0~`lt-1~.csv'.format(str(gain), str(int(
       lookahead_time*100))), 'w') as csvfile:
            writer = csv.writer(csvfile, lineterminator='"n')
            column_names = ['min_jerk_x', 'px', 'min_jerk_y',
                            'py', 'min_jerk_z', 'pz', 'min_jerk_vx', '
       vx',
                            'min_jerk_vy', 'vy', 'min_jerk_vz', 'vz']
            writer.writerow(column_names)
            for i in range(len(px)):
```

```
single_row_data = [
    str(min_jerk_x[i]),
    str(px[i]),
    str(min_jerk_y[i]),
    str(py[i]),
    str(min_jerk_z[i]),
    str(pz[i]),
    str(min_jerk_vx[i]),
    str(vx[i]),
    str(min_jerk_vy[i]),
    str(vy[i]),
    str(min_jerk_vz[i]),
    str(vz[i])]
writer.writerow(single_row_data)
```

---

**rtde.py** - File importato in Min_jerk_servoj.py, si tratta della libreria di funzioni della RTDE. Questo file è molto simile a quello scaricabile sul sito dell'Universal Robot ma contiene alcune modifiche, rtde.py può essere scaricato dalla repository GitHub https://github.com/danielstankw/Servoj_RTDE_UR5.git.

**min_jerk_planner_translation.py** - File importato in Min_jerk_servoj.py, contiene l'algoritmo di pianificazione della traiettoria. Si tratta di una pianificazione nello spazio operativo basata sul minimizzare la somma del jerk (derivata nel tempo dell'accelerazione) lungo la traiettoria. Questo file è stato scaricato dalla repository GitHub https://github.com/danielstankw/Servoj_RTDE_UR5.git ed è stato utilizzato senza modifiche.

**control_loop_configuration.xml** - Una configurazione RTDE può essere caricata da un file XML contenete la lista dei recipe (lista degli IO). Ogni recipe deve avere una key ed una lista di campi con il nome della variabile e la tipologia di dato a lei associato. Tutte le possibili tipologie di dati IO di questa interfaccia sono riportati nella giuda rilasciata da Universal Robot [19].

Il file seguente, control loop configuration.xml, è il file XML utilizzato per configurare la RTDE al fine di effettuare il controllo del robot.

```
¡?xml version="1.0"?¿
¡rtde_config¿
  ¡recipe key="state"¿
    ¡field name="runtime_state" type="UINT32"/¿
    ¡field name="actual_q" type="VECTOR6D"/¿
    ¡field name="actual_TCP_force" type="VECTOR6D"/¿
    ¡field name="actual_TCP_pose" type="VECTOR6D"/¿
    ¡field name="actual_TCP_speed" type="VECTOR6D"/¿
    ¡field name="output_int_register_24" type="INT32"/¿
    ¡field name="output_double_register_24" type="DOUBLE"/¿
    ¡field name="output_bit_register_64" type="BOOL"/¿
  ¡/recipe¿

  ¡recipe key="setp"¿
    ¡field name="input_double_register_0" type="DOUBLE"/¿
    ¡field name="input_double_register_1" type="DOUBLE"/¿
    ¡field name="input_double_register_2" type="DOUBLE"/¿
    ¡field name="input_double_register_3" type="DOUBLE"/¿
    ¡field name="input_double_register_4" type="DOUBLE"/¿
    ¡field name="input_double_register_5" type="DOUBLE"/¿
    ¡field name="input_double_register_24" type="DOUBLE"/¿
    ¡field name="input_int_register_24" type="INT32"/¿
  ¡/recipe¿

  ¡recipe key="watchdog"¿
    ¡field name="input_int_register_0" type="INT32"/¿
  ¡/recipe¿

¡/rtde_config¿
```

# Appendice D

# Ambiente ROS Python 2.7 e Python 3 scripts

L'ambiente ROS sviluppato durante le attività di tesi è diagrammato in Figura D.1

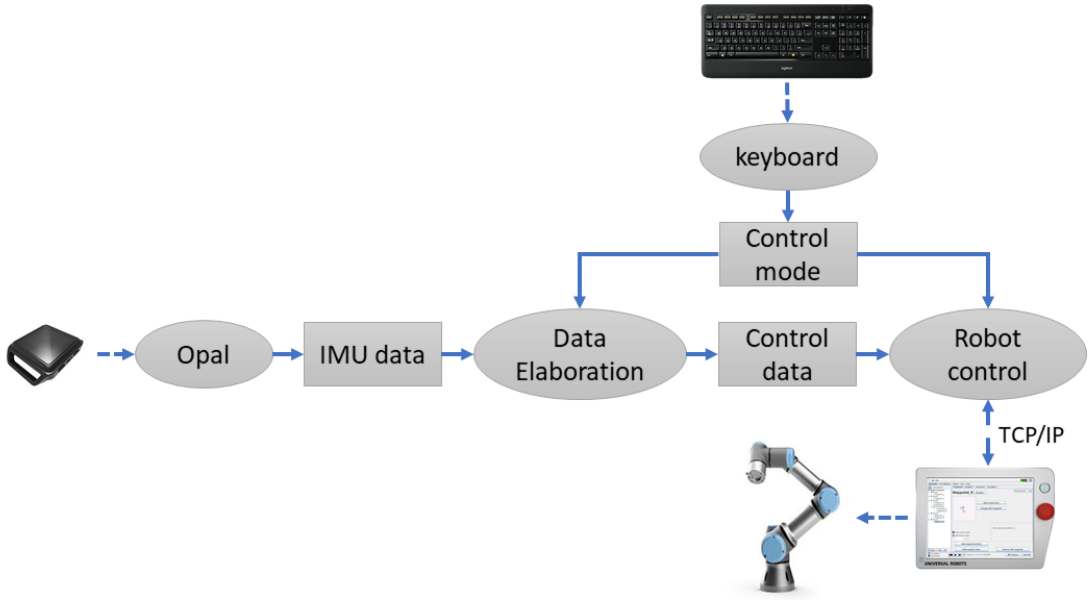



Figura D.1: Ambiente ROS sviluppato

### Nodo Opal


**opal_dataStream.py** - script per avviare il nodo Opal.

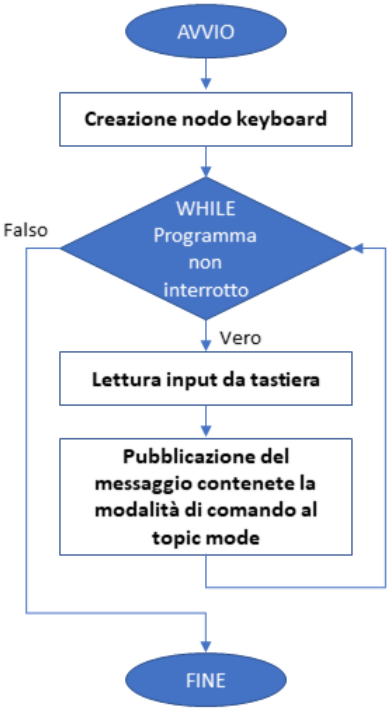
```
#!/usr/bin/env python2
import apdm

context = apdm.apdm.ctx.allocate.new.context()

try:
    apdm.apdm.ctx.open.all.access.points(context)

    """ streaming configuration """
    streaming.config = apdm.apdm.streaming.config.t()
    apdm.apdm.init.streaming.config(streaming.config)
    streaming.config.enable.accel = True
    streaming.config.enable.gyro = True
    streaming.config.enable.mag = True
    streaming.config.wireless.channel.number = 80
    streaming.config.output.rate.hz = 128
    r = apdm.
   apdm.ctx.autoconfigure.devices.and.accesspoint.streaming(context
   , streaming.config)
    if r != apdm.APDM.OK:
        raise Exception("Unable to autoconfigure system, r = " +
   apdm.apdm.strerror(r))
```

```
finally:
    apdm.apdm_ctx_disconnect(context)
    apdm.apdm_ctx_free_context(context)
    print("\n----------------------------\nopal autoconfigure
    succeeded\n\n" +
          "enable_accel = " + str(streaming_config.enable_accel) +
    "\n"
          "enable_gyro = " + str(streaming_config.enable_gyro) + "\
    n"
          "enable_mag = " + str(streaming_config.enable_mag) + "\n"
          "wireless_channel_number = " + str(streaming_config.
    wireless_channel_number) + "\n"
          "output_rate_hz = " + str(streaming_config.output_rate_hz
    ) + " Hz\n"
          "----------------------------\n" +
          "Undock Opals \n\n")
```

**opal_autoconfigure.py** - script per la configurazione del sensore.

```
#!/usr/bin/env python2
import apdm

context = apdm.apdm_ctx_allocate_new_context()

try:
    apdm.apdm_ctx_open_all_access_points(context)

    """ streaming configuration """
    streaming_config = apdm.apdm_streaming_config_t()
    apdm.apdm_init_streaming_config(streaming_config)
    streaming_config.enable_accel = True
    streaming_config.enable_gyro = True
    streaming_config.enable_mag = True
    streaming_config.wireless_channel_number = 80
    streaming_config.output_rate_hz = 128
    r = apdm.
    apdm_ctx_autoconfigure_devices_and_accesspoint_streaming(context
    , streaming_config)
    if r != apdm.APDM_OK:
        raise Exception("Unable to autoconfigure system, r = " +
    apdm.apdm_strerror(r))

finally:
    apdm.apdm_ctx_disconnect(context)
    apdm.apdm_ctx_free_context(context)
    print("\n----------------------------\nopal autoconfigure
    succeeded\n\n" +
          "enable_accel = " + str(streaming_config.enable_accel) +
    "\n"
          "enable_gyro = " + str(streaming_config.enable_gyro) + "\
    n"
          "enable_mag = " + str(streaming_config.enable_mag) + "\n"
          "wireless_channel_number = " + str(streaming_config.
    wireless_channel_number) + "\n"
          "output_rate_hz = " + str(streaming_config.output_rate_hz
    ) + " Hz\n"
          "----------------------------\n" +
          "Undock Opals \n\n")
```

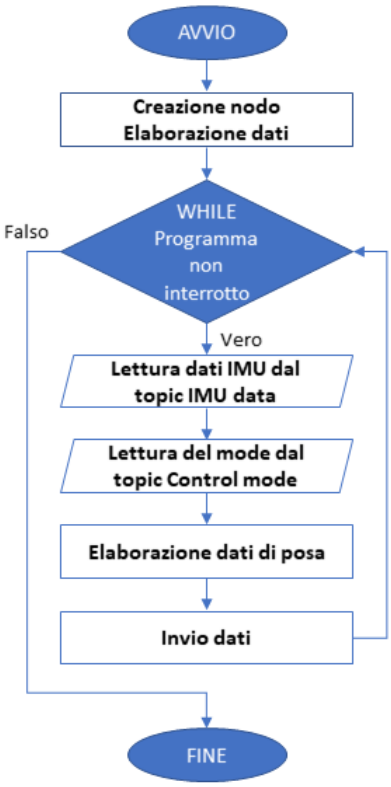
### Nodo keyboard




```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import Int8
import sys
import tty
import termios
import os
import time

def print_commands_list():
    print("The program will automatically move to mode 2 (placement
    ), use this node to move into command modes:\n")
    print("To enter mode 3 (orientation control) press: 3\n")
    print("To enter mode 4 (gripper rotation around z-TCP) press:
    4\n")
    print("To enter mode 5 (approach to object - gripper
    translation along z axis) press: 5\n")
    print("To enter mode 6 (pick and place operation) press: 6\n")
    print("
    ----------------------------------------------------------------------------------------
    n")

def getch():
    """
    Gets a single character from STDIO.
    """
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    try:
        tty.setraw(fd)
        return sys.stdin.read(1)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
```

```
def keys():
    pub = rospy.Publisher('mode',Int8,queue_size=10)
    rospy.init_node('keyboard')
    rospy.loginfo('keyboard node has been started“n“n')
    print_commands_list()
    rate = rospy.Rate(150)
    while not rospy.is_shutdown():
        k = getch()
        if k == '2':
            print('mode 2 (placement)“n')
            k = 2
            pub.publish(k)
        elif k == '3':
            print('mode 3 (orientation control)“n')
            k = 3
            pub.publish(k)
        elif k == '4':
            print('mode 4 (gripper rotation around z axis“n')
            k = 4
            pub.publish(k)
        elif k == '5':
            print('mode 5 (approach to object - gripper translation
    along z axis)“n')
            k = 5
            pub.publish(k)
        elif k == '6':
            print('mode 6 (pick and place operation)“n')
            k = 6
            pub.publish(k)
            time.sleep(5)
            os.system('clear')
            print_commands_list()

        rate.sleep()

if __name__=='__main__':
    try:
        keys()
    except rospy.ROSInterruptException:
        pass
```

### Nodo Elaborazione dati



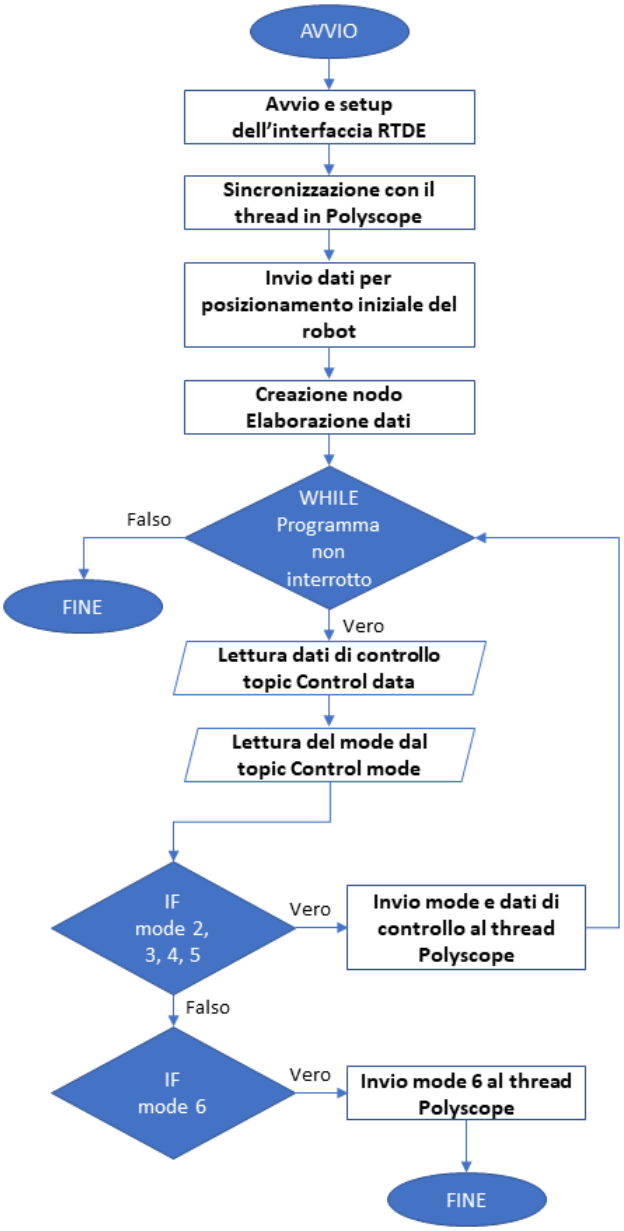

---

```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import Float32MultiArray, Int8
from sensor_msgs.msg import Imu
from spatialmath import SE3
from scipy.spatial.transform import Rotation
from transformations import euler_from_quaternion
import numpy as np
import math

def initial_poses():
    global BASE
    global AI_BASE
    global ATCP_O
    global ATCP_BASE
    global p_0
    global msg
    global loop_mode

    BASE = SE3() # wrf base
    AI_BASE = SE3(0.35,-0.05,0) #wrf object
    ATCP_O = SE3(np.array([ [1, 0, 0, 0    ],
                            [0,-1, 0, 0    ],
                            [0, 0,-1, 0.065],
                            [0, 0, 0, 1    ]]))
    ATCP_BASE = AI_BASE*ATCP_O  #starting wrf TCP with respect to
   base
    pos = np.ndarray.tolist(ATCP_BASE.t)
    r = Rotation.from_matrix(ATCP_BASE.R)
    rot = r.as_rotvec()
    p_0 = np.append(pos,rot)
```

```
    msg = Float32MultiArray()
    loop_mode = 2

def keyboard_command(k):
    global loop_mode
    loop_mode = k.data
    print('“nloop_mode = ' + str(loop_mode) + '“n')

def pose_callback(imu):
    global ATCP_BASE
    global AO_BASE
    roll, pitch, yaw = euler_from_quaternion([imu.orientation.x,
   imu.orientation.y, imu.orientation.z, imu.orientation.w])
    roll = math.degrees(roll)
    pitch = math.degrees(pitch)
    yaw = math.degrees(yaw)
    band = 10
    # rospy.loginfo(”roll: ” + str(roll) + ”  pitch: ” + str(pitch)
    + ”  yaw: ” + str(yaw)) #for testing purpose

    if loop_mode == 2:
        if pitch ¿ band:
            ATCP_BASE = SE3.Trans(0.00003*(pitch-band),0,0)*
   ATCP_BASE
        elif pitch ¡ -band:
            ATCP_BASE = SE3.Trans(-0.00003*(-pitch+band),0,0)*
   ATCP_BASE
        elif roll ¡ -band:
            ATCP_BASE = SE3.Trans(0,0.00003*(-roll-band),0)*
   ATCP_BASE
        elif roll ¿ band:
            ATCP_BASE = SE3.Trans(0,-0.00003*(roll-band),0)*
   ATCP_BASE
        AO_BASE = SE3(ATCP_BASE.t[0], ATCP_BASE.t[1], 0)
    elif loop_mode == 3:
        ATCP_BASE = AO_BASE*SE3.Ry(pitch,'deg')*SE3.Rx(roll,'deg')*
   ATCP_O
    elif loop_mode == 4:
        if roll ¿ 25:
            ATCP_BASE *= SE3.Rz(0.2,'deg')
        elif roll ¡ -45:
            ATCP_BASE *= SE3.Rz(-0.2,'deg')
    elif loop_mode == 5:
        if pitch ¡ -35:
            ATCP_BASE *= SE3.Trans(0,0,0.0001)
        elif roll ¿ 25:
            ATCP_BASE *= SE3.Trans(0,0,-0.0001)

    pos = np.ndarray.tolist(ATCP_BASE.t)
    r = Rotation.from_matrix(ATCP_BASE.R)
    rot = r.as_rotvec()
    msg.data = np.append(pos,rot)

    print(msg.data) #for testing purpose
    pub.publish(msg)

if ``name``==``main``':
    initial_poses()
    rospy.init_node(”data_elaboration”)
    rospy.loginfo(”Node has been started“n”)
```

```
    pub = rospy.Publisher("robot_command", Float32MultiArray)
    rospy.Subscriber("/mode", Int8, callback=keyboard_command)
    rospy.Subscriber("/IMU_data", Imu, callback=pose_callback)

    rospy.spin()
```

### Nodo Controllo robot




---

```
#!/usr/bin/env python3

import rospy
from std_msgs.msg import Float32MultiArray, Int8
from ur_msgs.msg import RobotStateRTMsg
import sys
sys.path.append('..')
import time

import rtde.rtde as rtde
import rtde.rtde_config as rtde_config

def setp_to_list(setp):
    temp = []
    for i in range(0, 6):
        temp.append(setp.__dict__["input_double_register_%i" % i])
    return temp
```

```
def list_to_setp(setp, list):
    for i in range(0, 6):
        setp.__dict__["input_double_register_%i" % i] = list[i]
    return setp

def first_start_to_False():
    global first_start
    first_start = False

def RTDE_start():
    #ROBOT_HOST = '192.168.146.129' #URSim
    ROBOT_HOST = '192.168.56.103' #UR3 Lab
    ROBOT_PORT = 30004
    config_filename = '/home/giulio/Desktop/final_test/scripts/
   telecontrol/RTDE_configuration.xml'

    global con
    global setp
    global watchdog

    conf = rtde_config.ConfigFile(config_filename)
    state_names, state_types = conf.get_recipe('state')
    setp_names, setp_types = conf.get_recipe('setp')
    watchdog_names, watchdog_types = conf.get_recipe('watchdog')

    con = rtde.RTDE(ROBOT_HOST, ROBOT_PORT)
    con.connect()
    connection_state = con.connect()

    while connection_state != 0 and not rospy.is_shutdown():
        rospy.loginfo("connection_state: ", connection_state)
        print("connection_state: ", connection_state)
        time.sleep(0.5)
        connection_state = con.connect()

    if connection_state != None:
        rospy.loginfo("Connected to the robot “n")

        # get controller version
        con.get_controller_version()

        # setup recipes
        con.send_output_setup(state_names, state_types)
        setp = con.send_input_setup(setp_names, setp_types)
        watchdog = con.send_input_setup(watchdog_names,
   watchdog_types)

        setp.input_double_register_0 = 0
        setp.input_double_register_1 = 0
        setp.input_double_register_2 = 0
        setp.input_double_register_3 = 0
        setp.input_double_register_4 = 0
        setp.input_double_register_5 = 0
        setp.input_double_register_24 = 0
        setp.input_int_register_24 = 0

        watchdog.input_int_register_0 = 0

        #start data synchronization
```

```
        if not con.send_start():
            rospy.loginfo("Unable to start sync \n")
            sys.exit()

def set_robot():
    global receive_keyboard_commands
    receive_keyboard_commands = False
    p_start = [0.35, -0.05, 0.065, 3.14159265, 0, 0]

    state = con.receive()
    tcp = state.actual_TCP_pose
    rospy.loginfo('actual_TCP_pose: ', str(tcp), '\n')
    print('actual_TCP_pose: ', str(tcp), '\n')

    #---------- mode = 1 (Connection) -----------
    rospy.loginfo('Boolean 1 is False, please click CONTINUE on the
    Polyscope\n')
    print('Boolean 1 is False, please click CONTINUE on the
   Polyscope\n')
    while True:
        state = con.receive()
        con.send(watchdog)
        if state.output_bit_register_64 == True:
            print('Boolean 1 is True, Robot Program can proceed to
   mode 1\n')
            time.sleep(1)
            break

    print("-------Executing moveJ -----------\n")
    time.sleep(1)

    list_to_setp(setp, p_start)  # changing initial pose to setp
    con.send(setp) # sending initial pose
    watchdog.input_int_register_0 = 1
    con.send(watchdog)  # sending mode == 1

    print('Waiting for movej() to finish\n')
    while True:
        state = con.receive()
        con.send(watchdog)
        if state.output_bit_register_64 == False:
            print('Proceeding to mode 2\n')
            time.sleep(1)
            break

    rospy.loginfo("---------- Control loop  --------\n")
    time.sleep(2)
    watchdog.input_int_register_0 = 2
    con.send(watchdog)  # sending mode == 2
    receive_keyboard_commands = True

def keyboard_command(k):
    global loop_mode
    if receive_keyboard_commands:
        loop_mode = k.data
        watchdog.input_int_register_0 = loop_mode
        con.send(watchdog)  # sending mode
        print('Proceeding to mode ' + str(loop_mode) + '\n')

def robot_control(msg):
```

```
    if first_start:
        global last_start_time
        last_start_time = time.time()
        first_start_to_False()

    list_to_setp(setp, msg.data)
    con.send(setp)

    if loop_mode == 6:
        print('Waiting for pick and place operation to finish“n')
        while True:
            state = con.receive()
            con.send(watchdog)
            if state.output_bit_register_64 == True:
                print('Operation finished“n')
                con.send_pause()
                con.disconnect()
                print("RTDE disconnected")
                time.sleep(1)
                break

    # restart RTDE every 15 seconds to avoid connection issues
    if time.time()-last_start_time ¿ 20:
        RTDE_start()
        last_start_time = time.time()

    if rospy.is_shutdown():
        return

def robot_node():
    rospy.init_node('robot_control')
    rospy.loginfo("Node has been started")
    sub1 = rospy.Subscriber("/mode", Int8, callback=
   keyboard_command)
    sub2 = rospy.Subscriber("/robot_command", Float32MultiArray,
   callback=robot_control)

    rospy.spin()

if __name__ == '__main__':
    try:
        global first_start
        first_start = True
        RTDE_start() #open RTDE connection and initialize variables
        set_robot() #sync with polyscop program then loop 1 and 2
        robot_node() #node initialization, send RTDE registers
    finally:
        con.send_pause()
        con.disconnect()
        rospy.loginfo("RTDE disconnected")
```