Università Politecnica delle Marche

Ingegneria Informatica e dell'Automazione

Co-trasporto collaborativo Umano-Robot: implementazione in ROS2 con UR5 e inseguimento visivo con telecamera di profondità

Corso di

Dynamics and control of intelligent robots and vehicles

Anno accademico 2024-2025

Studenti:
Edoardo Palmoni
Matteo Stronati
Jacopo Tarulli

Professore:
Andrea Bonci

Dottorandi:
Alessandro Di Biase
Andrea Serafini

Dipartimento di Ingegneria dell'Informazione

Indice

Elenco delle figure

Elenco dei Listati

1 Introduzione

La presente relazione descrive in dettaglio lo sviluppo e l'implementazione di un sistema di controllo robotico in ambiente ROS2, concepito per favorire il co-trasporto di un telo tra un operatore umano e un braccio robotico UR5 (Universal Robots UR5). L'operatore umano deve riuscire a spostare un materiale flessibile e il braccio robotico deve seguire i movimenti dell'uomo mantenendo una distanza fissa da esso in modo da non deformare il materiale trasportato. Il task assegnato ha previsto la progettazione e la realizzazione di un'architettura software modulare che permettesse al robot di mantenere una distanza predefinita rispetto a un punto di riferimento specifico sul telo, rappresentato da un "punto rosso" tracciato dall'operatore. L'obiettivo ultimo è consentire un'interazione collaborativa fluida e sicura nel compito di manipolazione congiunta.

Figure 1.1: *Co-trasporto tra operatore umano e braccio robotico*

Il progetto ha avuto inizio con l'installazione dei driver ROS2 e la configurazione del braccio UR5. In seguito, lo sviluppo si è articolato su diverse componenti software interconnesse. Un modulo di percezione è responsabile del riconoscimento del punto rosso in tempo reale attraverso dati di visione. Successivamente, il lavoro si è focalizzato sullo sviluppo del modulo di controllo, che riceve la posizione 3D del punto rosso e, attraverso opportune trasformazioni cinematiche, calcola la sua posizione rispetto al frame di riferimento della base del robot. Su questa base, il controllore è stato progettato per mantenere l'end-effector del robot a una distanza e con un orientamento fissati rispetto al punto rosso, consentendo all'operatore di guidare il robot muovendo il telo. Un aspetto cruciale dell'implementazione è stata l'enfasi

sulla sicurezza operativa. Sono stati imposti vincoli rigidi di sicurezza sulle variazioni di posizione angolare dei giunti e sulla velocità lineare dell'end-effector, prevenendo movimenti bruschi o pericolosi che potrebbero compromettere la stabilità del co-trasporto o la sicurezza dell'operatore. L'intero sistema è stato testato e validato in ambiente reale. In sintesi, questa relazione illustra l'implementazione di un sistema robotico collaborativo in ROS2 che, attraverso la percezione visiva e un controllo cinematico attento ai vincoli di sicurezza, permette al robot UR5 di cooperare con un umano nel co-trasporto di un oggetto flessibile.
La restante parte della relazione è così strutturata:

- Nel **Capitolo 2** vengono presentati i componenti hardware principali, il braccio robotico UR5 e la videocamera RealSense. In seguito viene descritto il sistema nel suo complesso, inclusa la predisposizione dell'ambiente ROS2, i principi del controllore, cenni sulla cinematica diretta e inversa e il concetto di interazione uomo-robot.

- Nel **Capitolo 3** viene approfondita la parte software sviluppata, descrivendo i quattro script principali del progetto, inclusi quelli per il riconoscimento del punto rosso con calcolo della posizione attraverso le trasformazioni, e lo sviluppo del controllore robotico.

- Nel **Capitolo 4** sono illustrati i test implementativi, inclusi i comandi da terminale per l'interfacciamento con il robot e la videocamera, e la presentazione dei risultati di test reali tramite grafici degli errori.

- Nel **Capitolo 5** vengono riassunte le conclusioni del lavoro svolto e proposti possibili sviluppi futuri del sistema.

2 Sistema e Hardware

In questo capitolo viene descritta la struttura del sistema sviluppato per la realizzazione del co-trasporto collaborativo con il braccio robotico UR5. Viene presentata l'architettura generale, evidenziando i principali componenti hardware e software coinvolti, e viene fornita una panoramica della logica implementativa alla base del comportamento del robot.
Successivamente, viene introdotta la predisposizione dell'ambiente di lavoro, con particolare attenzione all'installazione e configurazione dei pacchetti necessari su Ubuntu, compreso il setup del driver ROS2 per l'UR5. L'interazione del sistema con il controllore del robot in modalità *External Control* è illustrata nella sezione dedicata, seguita dalla procedura di configurazione del teach pendant.
Si passa poi alla modellazione cinematica del robot (diretta e inversa) e, infine, all'interazione uomo-robot nel contesto collaborativo del co-trasporto.

2.1 Architettura generale

Il sistema ha come obiettivo la realizzazione di una modalità di co-trasporto collaborativo tra un operatore umano e il braccio UR5. Il robot è in grado di rilevare la posizione di un **marcatore visivo** (punto rosso, d'ora in poi *target*) tramite una telecamera montata sull'end-effector e di muoversi mantenendosi a una distanza prefissata dal target, seguendo i movimenti dell'operatore in tempo reale.

2.1.1 Componenti del sistema

I moduli principali sono mostrati in Figura 2.1:

- **UR5** (braccio collaborativo a 6 DoF);
- **PC con ROS2** (Ubuntu 22.04 LTS, ROS2 Humble);
- **Intel RealSense D435i** (telecamera RGB-D con IMU);
- **Rete Ethernet** per la comunicazione robot–PC.

(a) *(b)*

Figure 2.1: *Componenti principali: (a) UR5, (b) RealSense D435i.*

2.2 Hardware

2.2.1 UR5 e *control box*

Specifiche tecniche rilevanti

Il **robot collaborativo UR5** di *Universal Robots* è un braccio a 6 DoF progettato per compiti di manipolazione flessibili e collaborativi. Le specifiche principali sono:

- **Sbraccio operativo**: 850 mm;
- **Carico utile**: 5 kg;
- **Ripetibilità**: ±0.1 mm (e-Series ±0.03 mm);
- **Peso**: 18.4 kg;
- **Velocità**: fino a 180°/s per giunto, 1 m/s lineare;
- **Consumo**: 150–250 W (picco 570 W);
- **Protezione**: IP54.

Modalità di comunicazione

Il controllo avviene via Ethernet in modalità `external control`, usando socket TCP/IP (porte 30003/50001) per l'invio di comandi in tempo reale da ROS2.

Modalità manuale vs. automatica

- **Manuale**: movimento tramite teach pendant (*Free Drive*, jog di singoli giunti).
- **Automatica**: comandi da ROS2 attraverso la rete.

2.2.2 Intel RealSense D435i

Specifiche tecniche

- **FOV**: 85.2°×58° (H×V);
- **Range**: 0.2–10m;
- **Frame rate**: fino a 90fps;
- **Risoluzione**: fino a 1280×720 (RGB e depth).

Configurazione della camera

La D435i è montata su un supporto stampato in 3D fissato sopra l'end-effector. Il sensore è traslato di 3cm sugli assi x e y rispetto a `tool0` per migliorare la visibilità.
La connessione col PC ROS avviene via USB3.0; i dati sono pubblicati tramite il pacchetto `realsense2_camera` su topic standard (`/camera/color/image_raw`, ecc.).

2.2.3 Schema dei collegamenti

Diagramma fisico

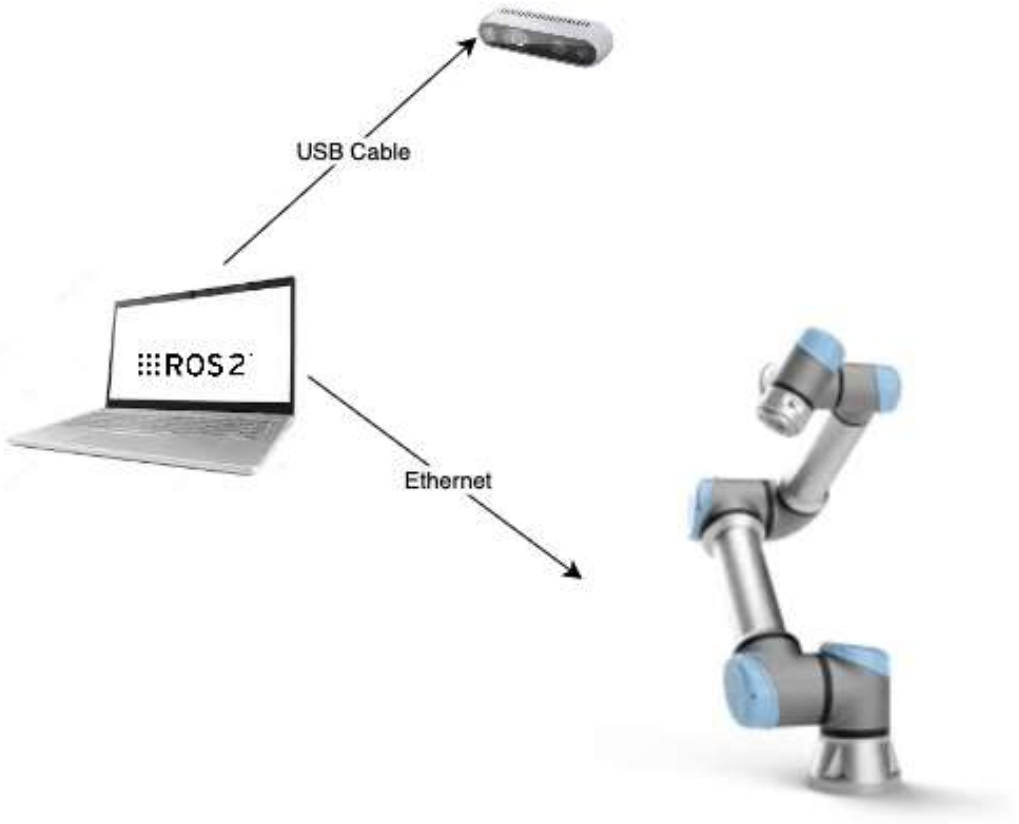

Figure 2.2: *Schema dei collegamenti tra UR5, RealSense e PC ROS.*

Rete e indirizzi IP

- `UR5`: 192.168.1.102
- `PC ROS`: 192.168.1.101

La RealSense comunica via USB e non richiede IP.

Connessioni principali

- **UR5 ↔ PC ROS**: Ethernet diretta.
- **PC ROS ↔ RealSense**: USB3.0.

Alimentazione e cablaggio

UR5 alimentato dal *control box*, RealSense via USB, PC da rete elettrica. I cavi sono fissati con fascette per evitare interferenze meccaniche.

2.3 Logica implementativa

La logica di funzionamento segue la sequenza ciclica:

1. **Acquisizione target**: la RealSense rileva il target rosso e ne stima la posizione 3D nel proprio frame.
2. **Trasformazione**: la posizione è trasformata nel frame `base_link` (riferimento inerziale del braccio).
3. **Generazione posa obiettivo**: si calcola la posa dell'end-effector a distanza prefissata dal target, con orientamento coerente.
4. **Cinematica Inversa (IK) e pianificazione**: MoveIt!, libreria ROS che offre risolutori per la cinematica, risolve la cinematica inversa e genera una *JointTrajectory*.
5. **Controllo**: la traiettoria è inviata al `scaled_joint_trajectory_controller`, che esegue il movimento.

2.4 Predisposizione dell'ambiente di lavoro

L'intero sistema di controllo e interazione con l'UR5 è stato sviluppato all'interno di un ambiente ROS2 su sistema operativo Ubuntu. Per garantire compatibilità con i pacchetti ufficiali forniti da *Universal Robots* e il corretto funzionamento dell'interfaccia di controllo, è stata utilizzata la distribuzione **Ubuntu 22.04 LTS** con **ROS2 Humble**. In questo ambiente si è provveduto a:

- configurare la rete Ethernet in modo da garantire la comunicazione in tempo reale tra il PC e l'UR5 (ad es. assegnando al PC l'IP statico 192.168.1.101 e al robot 192.168.1.102 sulla stessa subnet),
- installare i pacchetti ROS2 necessari, inclusi quelli per la visualizzazione (RViz) e per la pianificazione del moto (*MoveIt!*), che include i servizi di cinematica diretta e inversa (*/compute_fk*, */compute_ik*) e il *move_group* controller,
- utilizzare l'interfaccia *External Control* fornita dal driver ROS2 ufficiale di *Universal Robots* per il controllo a bassa latenza del robot.

La configurazione dell'ambiente è una fase fondamentale del progetto, in quanto richiede attenzione alle versioni dei pacchetti, alla corretta installazione delle dipendenze, e alla configurazione di rete tra robot e PC.

2.4.1 Installazione e configurazione del sistema

L'installazione dell'ambiente di lavoro è stata condotta con i seguenti passaggi principali:

1. **Installazione di ROS2 Humble:** è stata eseguita l'installazione tramite pacchetti APT ufficiali, seguendo la guida di installazione della distribuzione Humble [1]. Si è quindi installato il metapacchetto completo con:

```
sudo apt update && sudo apt upgrade
sudo apt install ros-humble-desktop-full
```

A installazione conclusa, sono stati aggiunti al file `.bashrc` i comandi di sourcing per attivare ROS2 in ogni nuovo terminale:

```
source /opt/ros/humble/setup.bash
```

2. **Creazione del workspace e installazione dei pacchetti UR5:** è stato creato un workspace ROS2 personalizzato e sono stati clonati i pacchetti ufficiali forniti da Universal Robots per ROS2. In particolare, sono stati scaricati:
   - il *Universal_Robots_ROS2_Driver* [2], contenente il driver ROS2 per l'UR5 (in modalità *External Control*),
   - il pacchetto *ur_description*, che include il modello URDF del robot,
   - il pacchetto *ur_moveit_config*, con le configurazioni pronte per MoveIt! sull'UR5,
   - i pacchetti *ros2_control* e *ros2_controllers*, necessari per il controllo dei giunti attraverso il controller ROS2.

   Dopo aver clonato i repository, il workspace è stato compilato usando `colcon build`. In questo modo sono stati generati tutti i file necessari (inclusi i file di configurazione per i controller e per MoveIt!) utili per l'integrazione completa del robot nel sistema ROS2.

3. **Configurazione della rete Ethernet:** per permettere la comunicazione con l'UR5, il PC è stato collegato al braccio robotico mediante cavo Ethernet diretto. Sono stati assegnati indirizzi IP statici compatibili, ad esempio 192.168.1.101 per il PC e 192.168.1.102 per l'UR5, con subnet mask 255.255.255.0. In questo modo PC e robot si trovano sulla stessa rete locale, requisito necessario per l'utilizzo del protocollo di `External Control`.

4. **Avvio dell'External Control sul robot**: L'avvio dell'External Control richiede che sul robot sia selezionato il programma *external_control.urp*, fornito con il pacchetto del driver ROS2. Questo programma imposta l'UR5 in ascolto sulla connessione TCP/IP per ricevere comandi da ROS.

5. **Avvio dei nodi e test di comunicazione:** infine, per verificare la corretta configurazione sono stati avviati i nodi ROS2 del driver UR5 utilizzando i file di launch forniti, con il comando:

   ```
   ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur5
   robot_ip:=192.168.1.102 launch_rviz:=true
   ```

   si avvia il driver in modalità *External Control* (verrà richiesto di avviare il programma `external-control.urp` sul robot). In parallelo su un altro terminale, si è lanciato MoveIt! con:

```
ros2 launch ur_moveit_config ur_moveit.launch.py ur_type:=ur5
use_fake_hardware:=false
```

Questo comando avvia il nodo di MoveIt! insieme a RViz, consentendo la visualizzazione del robot e l'attivazione dei servizi di cinematica diretta (*/compute_fk*) e inversa (*/compute_ik*) per l'UR5. Viene inoltre avviato il *move_group* controller che coordina la pianificazione delle traiettorie.

Questa configurazione iniziale permette di calcolare traiettorie articolari con MoveIt! e di inviarle al braccio UR5 tramite il *scaled_joint_trajectory_controller*.

2.4.2 *scaled_joint_trajectory_controller*

Il *scaled_joint_trajectory_controller* è il controller principale utilizzato nel progetto per comandare il robot UR5 da ROS2 in modalità External Control. Questo controller è una variante del classico *joint_trajectory_controller*, dotata di un meccanismo di scalatura temporale che permette al robot di:

- eseguire traiettorie in modo più fluido e sicuro,
- rispettare automaticamente i limiti di velocità e accelerazione impostati,
- evitare l'inserimento di errori di watchdog o il superamento di soglie, anche se la traiettoria viene modificata o interrotta dinamicamente.

Nel progetto, il controller è configurato con limiti di velocità relativamente bassi per garantire movimenti non bruschi, come richiesto dalle linee guida di sicurezza.
Il controller accetta messaggi sul topic:

```
/scaled_joint_trajectory_controller/follow_joint_trajectory
```

con messaggio di tipo *trajectory_msgs/JointTrajectory*. In ciascun messaggio si specifica una sequenza temporizzata di pose articolari, definita da:

- l'elenco dei nomi dei giunti (*joint_names*),
- le posizioni desiderate per ciascun giunto (obbligatorie), e opzionalmente le velocità e accelerazioni,
- i tempi relativi (campo *time_from_start*) per ciascun punto della traiettoria.

Il controller è inoltre integrato con MoveIt!: infatti, durante la pianificazione delle traiettorie, MoveIt! può generare automaticamente un messaggio *JointTrajectory* da inviare al controller tramite un'azione *FollowJointTrajectory*. Nel nostro progetto abbiamo utilizzato sia le traiettorie generate da MoveIt!, sia comandi generati dinamicamente dai nodi personalizzati in Python, per avere maggiore flessibilità nel controllo logico.
L'uso del *scaled_joint_trajectory_controller* ha garantito un'integrazione robusta tra il driver UR, MoveIt! e i nodi applicativi, offrendo un meccanismo sicuro e reattivo per il co-trasporto e il controllo in tempo reale.

2.5 Configurazione del teach pendant per l'External Control

Per abilitare la modalità *External Control* sul robot UR5, è necessario configurare opportunamente il teach pendant. I passaggi principali sono i seguenti:

1. **Accensione e inizializzazione del robot:**
   - Collegare il *control box* del robot e accendere l'unità. Quindi accendere il teach pendant.
   - Dal menù principale del teach pendant selezionare *Setup Robot* → *Initialization* → *Start*.
   - Attendere che il braccio si inizializzi; a questo punto il robot è pronto per essere mosso.

(a) *(b)* *(c)*

Figure 2.3: *Sequenza delle principali schermate del teach pendant: (a) Setup iniziale, (b) scelta dell'inizializzazione, (c) avvio del robot.*

2. **Configurazione di rete:**
   - Sempre in *Setup Robot*, selezionare *Network*.
   - Impostare un indirizzo IP statico per il robot (ad esempio 192.168.1.102) compatibile con quello del PC (192.168.1.101), verificando che la *subnet mask* sia impostata su 255.255.255.0. Questo garantisce che PC e robot siano sulla stessa rete locale.
3. **Abilitazione del modulo URCap *External Control*:**
   - Dal menù principale, andare in *Setup Robot* → *URCaps*.
   - Selezionare il modulo *External Control* e verificarne l'attivazione.
4. **Caricamento del programma di External Control:**
   - Dal menù principale, selezionare *Program* → *Load Program*.
   - Connettere una chiavetta USB contenente il file `external-control.urp`. Selezionare il file `external-control.urp` e caricarlo nel robot.
5. **Avvio del programma:**

- Premere il tasto *Play* sul teach pendant per avviare il programma. Se il file `external-control.urp` è stato caricato correttamente, sul display comparirà il messaggio *Waiting for connection....* Quando il driver ROS2 si connetterà, il display mostrerà lo stato *Status: Running.*

Una volta completata questa procedura, il robot è correttamente configurato per ricevere comandi dal nodo ROS2 `ur_control_node` in modalità External Control. La corretta esecuzione di questi passaggi è fondamentale per garantire una comunicazione stabile tra il sistema di controllo e il robot durante tutte le operazioni di co-trasporto.

2.6 Cinematica diretta e inversa dell'UR5

Il corretto controllo del robot UR5 si basa su una comprensione approfondita della **cinematica diretta** (forward kinematics) e della **cinematica inversa** (inverse kinematics). La cinematica diretta consente di calcolare la posizione e l'orientamento dell'end-effector dato un certo set di configurazioni articolari (angoli di giunto), mentre la cinematica inversa risolve il problema inverso: dato un obiettivo nello spazio cartesiano, determina le configurazioni dei giunti che lo raggiungono.

2.6.1 Cinematica diretta

Per il calcolo della cinematica diretta, si possono utilizzare i parametri di Denavit-Hartenberg (DH), che rappresentano ogni giunto con una sequenza standardizzata di trasformazioni. Tuttavia, nel contesto ROS2 e UR5 si utilizza tipicamente MoveIt! insieme a librerie di cinematica (ad esempio KDL o Trac_IK) che permettono un'integrazione diretta con il modello URDF del robot.
Nel progetto, la cinematica diretta è stata utilizzata per determinare:

- la trasformazione complessiva tra i frame del robot, utile ad esempio per stimare la posizione della telecamera montata sull'end-effector rispetto alla base,
- l'orientamento iniziale dell'end-effector, che viene mantenuto durante le operazioni di co-trasporto.

Il calcolo è stato effettuato tramite il servizio ROS2 */compute_fk* fornito da MoveIt!, al quale si fornisce un set di giunti e il frame di destinazione (ad es. *tool0*). Il *frame base_link* è fissato all'origine della base del robot ed è utilizzato come riferimento inerziale per il braccio.

2.6.2 Cinematica inversa

La cinematica inversa rappresenta una sfida più complessa, in quanto può ammettere più soluzioni (o nessuna soluzione valida) e richiede l'applicazione di vincoli per selezionare quella più adatta al nostro scopo. Ad esempio, l'UR5 ha 6 gradi di libertà e per molte pose esistono soluzioni multiple. Tra i vincoli adottati nel nostro progetto vi sono:

- il mantenimento di un asse dell'end-effector (tipicamente l'asse X) orientato verso il target,

- il blocco del *sesto giunto* (*wrist_3_joint*), per evitare rotazioni indesiderate dell'utensile che potrebbero far collidere o interferire l'attrezzo (ad esempio una staffa montata sul polso del robot) con il resto del braccio o con la telecamera.

Nel progetto, la cinematica inversa è stata risolta utilizzando il servizio ROS2 */compute_ik* fornito da MoveIt!. A partire da una *geometry_msgs/PoseStamped* rappresentante la posa desiderata dell'end-effector, il servizio restituisce una configurazione articolare compatibile. In caso di fallimento (*ik_failed*), il sistema può attivare un meccanismo di fallback o loggare l'evento, per evitare di inviare comandi non validi al robot.

2.7 Interazione uomo-robot

Il cuore del progetto è rappresentato dall'interazione collaborativa tra l'essere umano e il robot UR5, con l'obiettivo di realizzare un sistema di **co-trasporto** sicuro ed efficace. In questo contesto, il robot non opera in completa autonomia, ma coopera attivamente con l'operatore umano, seguendo i suoi movimenti e contribuendo allo spostamento di un oggetto condiviso.

2.7.1 Co-trasporto e comportamento collaborativo

Nel co-trasporto, il robot deve adattarsi alla posizione del target rosso e mantenere una certa distanza di sicurezza durante il movimento. Per questo motivo, il robot adotta un comportamento reattivo: ad ogni ciclo operativo, calcola una nuova posa dell'end-effector che si trovi a una distanza prefissata dal target (ad esempio 1 metro), mantenendo l'orientamento dell'utensile rivolto verso il target. In tal modo, l'operatore umano può guidare indirettamente il robot spostando il target visivo; di conseguenza il sistema ricalcola continuamente la posa desiderata dell'end effector e invia comandi aggiornati al controllore del robot.

2.7.2 Sicurezza

Per garantire un'interazione affidabile tra uomo e robot, il sistema implementa diverse strategie di sicurezza a livello di controllo e di software. Tali strategie sono fondamentali per prevenire situazioni potenzialmente pericolose, soprattutto in un contesto di lavoro condiviso. Le principali misure adottate includono:

- **Limitazioni cinematiche:** nel controller sono stati impostati limiti di velocità e accelerazione dei giunti piuttosto bassi, come consigliato dalle linee guida di sicurezza. Questo riduce il rischio di movimenti bruschi o imprevedibili.
- **Verifica di consistenza:** prima di inviare un nuovo comando di posa al robot, il sistema controlla che la differenza tra la configurazione attuale dei giunti e quella target non sia eccessivamente grande. Se il salto richiesto è troppo elevato (ad es. supera una certa soglia definita), il movimento viene annullato o eseguito a velocità ridotta per assicurare la continuità e la regolarità del moto.
- **Monitoraggio degli errori di tracking:** il sistema calcola e registra ad ogni ciclo l'errore tra la posa desiderata e quella effettivamente raggiunta. Un aumento improvviso di

questo errore può indicare un problema nella stima del target, nel calcolo della cinematica inversa o nell'esecuzione della traiettoria. In tal caso, il sistema può interrompere temporaneamente il movimento e loggare l'anomalia.

- **Gestione dei fallimenti:** la cinematica inversa potrebbe fallire in alcune situazioni (assenza di soluzioni). In caso di *ik_failed*, il sistema evita di inviare comandi instabili al controllore del robot e registra l'evento per analisi successive.
- **Vincoli articolari:** il blocco del giunto *wrist_3_joint* e altri vincoli specifici (ad es. limiti di giunto) impediscono movimenti indesiderati che potrebbero compromettere la presa o provocare collisioni.

Tutti questi accorgimenti rendono l'interazione più sicura e prevedibile, anche in presenza di piccoli errori di stima o disturbi esterni.

2.7.3 Adattamento e flessibilità

Il sistema è stato progettato per offrire grande flessibilità d'uso e adattarsi facilmente a contesti differenti. Alcuni aspetti che ne aumentano la versatilità includono:

- **Modularità del software:** i nodi ROS2 sono separati per rilevamento visivo, trasformazione delle coordinate, risoluzione IK e controllo robot. Questo consente di sostituire o modificare singoli moduli (ad esempio il metodo di visione) senza interferire sull'intero sistema.
- **Parametri configurabili a runtime:** distanza desiderata dal target, frequenza di aggiornamento, vincoli cinematici e altri parametri possono essere variati tramite parametri ROS o file `.yaml`, permettendo di adattare rapidamente il comportamento del robot a scenari diversi.
- **Fonti alternative del target:** sebbene nel nostro caso il target sia fornito da un marcatore rosso catturato da una videocamera, il sistema può facilmente essere esteso a sorgenti di target diverse, come altri sistemi di visione, sensori wearable, o un robot mobile (es. MiR200) che trasporta il target. La pipeline modulare consente questa integrazione senza modifiche sostanziali ai nodi principali.
- **Personalizzazione del comportamento:** a seconda dell'applicazione, è possibile variare la logica di controllo; ad esempio si può scegliere di seguire l'operatore a distanza fissa, mantenere l'orientamento dell'end effector verso il target o introdurre strategie di evitamento ostacoli. Tali comportamenti possono essere implementati modificando opportunamente il modulo di generazione della posa target o i vincoli di cinematica inversa.

Queste caratteristiche rendono il sistema adatto a un'ampia gamma di scenari collaborativi, oltre il semplice co-trasporto.

3 Software

Il presente capitolo descrive in dettaglio l'architettura software sviluppata per il sistema di co-trasporto robotico, un elemento cruciale per la gestione coordinata della percezione, della pianificazione del movimento e dell'esecuzione robotica. Il sistema è stato concepito attorno al Robot Operating System 2 (ROS2), un framework open-source flessibile e robusto, ampiamente adottato nella robotica per la gestione di sistemi complessi e distribuiti.L'adozione di ROS2 ha permesso di sfruttare un'infrastruttura modulare basata sul concetto di "nodi" indipendenti che comunicano attraverso meccanismi standardizzati quali Topic, Servizi e Actions. Questa modularità facilita lo sviluppo, il debug e la scalabilità del sistema, consentendo una chiara separazione delle responsabilità tra le diverse componenti software.
L'intero software applicativo, inclusi i tre script principali che formano la logica di controllo, è stato sviluppato interamente in Python. Questo linguaggio è stato scelto per la sua rapidità di sviluppo, l'ampia disponibilità di librerie per la robotica e la visione artificiale, e la sua eccellente integrazione con l'ambiente ROS2.
Il cuore dell'implementazione risiede in tre script Python principali, progettati per interagire armoniosamente con i driver hardware specifici:

- **Driver della Intel RealSense**: Responsabili dell'acquisizione dei dati visivi (immagini RGB e di profondità) dall'ambiente.
- **Driver del robot UR5 (integrato con MoveIt!)**: Gestiscono il controllo a basso livello del manipolatore, fornendo funzionalità di cinematica inversa (IK), cinematica diretta (FK) e l'esecuzione di traiettorie.

La logica implementativa del sistema si articola attraverso una pipeline sequenziale di elaborazione, garantendo un flusso continuo e coordinato di informazioni. Sono stati implementati, infatti, tre script Python per la gestione della visione della telecamera, la pianificazione delle traiettorie e il movimento del robot:

1. **Percezione** (Script red_dot_tracker.py): Questo script è il primo anello della catena percettiva. Il nodo RedDotToBaseNode si occupa di acquisire i dati grezzi dalla telecamera Intel RealSense, elaborare le immagini per identificare e localizzare un punto rosso specifico nello spazio 3D. Utilizzando le informazioni sulla posa corrente dell'end-effector del robot (ottenuta tramite cinematica diretta) e un offset predefinito della telecamera, converte le coordinate del punto rosso dal frame della telecamera a un frame di riferimento comune: la base del robot (base_link). La posizione 3D così calcolata viene poi pubblicata come topic in input per il modulo di controllo.
2. **Pianificazione e controllo** (Script trajectory_sender.py): Ricevendo la posizione del punto rosso dal modulo di percezione, questo script rappresenta il cervello decisionale del sistema. La sua logica implementa un controllo a tempo fisso che calcola iterativamente la "prossima posizione" desiderata per l'end-effector del robot, tenendo conto di vincoli di velocità massima e di sicurezza (es. limiti di movimento dei giunti). Per tradurre la posa desiderata dell'end-effector in una configurazione angolare dei giunti

del robot, il nodo RobotController si avvale di una classe ausiliaria, IKClient, che interagisce con i servizi di cinematica inversa (IK) forniti da MoveIt!. La configurazione dei giunti risultante, validata per la sicurezza, viene poi pubblicata come topic che andrà in input al modulo di esecuzione.

3. **Esecuzione del movimento**(Script mover.py): L'ultimo script della pipeline è dedicato all'interfacciamento diretto con il controller del robot per l'esecuzione fisica del movimento. Esso sottoscrive i comandi dei giunti calcolati dal controller e li impacchetta in un "Goal" per */scaled_joint_trajectory_controller/follow_joint_trajectory*, un ROS2 Action Server che è il meccanismo standard di ROS2 per gestire operazioni di controllo robotico a lungo termine con feedback. Una volta completato il movimento, il nodo SimpleRobotMover esegue una verifica della posizione finale effettivamente raggiunta dall'end-effector (nuovamente tramite cinematica diretta) e ne logga i dati, fornendo un feedback cruciale per l'analisi delle prestazioni del sistema.

Questa architettura a nodi disaccoppiati e la chiara pipeline logica garantiscono flessibilità, robustezza e la possibilità di estendere il sistema con nuove funzionalità o sensori in futuro. A questi script si aggiunge un quarto script (plot_error.py) che esula dalla logica implementativa in ROS2 e che si occupa di leggere i log da un file .csv e graficare tre errori significativi.

Nella Figura 3.1 è possibile osservare un diagramma esaustivo dell'architettura software del sistema di co-trasporto robotico, illustrando le interconnessioni tra i vari nodi ROS2, i topic, i servizi e le actions. Questa rappresentazione grafica è fondamentale per comprendere il flusso di dati e la logica di comunicazione tra le diverse componenti del sistema. Il diagramma è organizzato per mostrare chiaramente la pipeline di elaborazione, partendo dall'acquisizione dei dati sensoriali fino all'esecuzione del movimento robotico:

- **Acquisizione Dati Sensoriali**: Il processo inizia con il sensore Intel RealSense, che acquisisce immagini a colori e di profondità. Questi dati vengono pubblicati autonomamente, grazie ai driver Realsense, su topic ROS2 specifici come */camera/camera/color/image_raw*, */camera/camera/aligned_depth_to_color/image_raw* e */camera/camera/color/camera_info*. Contestualmente, lo stato attuale dei giunti del robot UR5 viene pubblicato sul topic */joint_states*.

- **Nodo di Percezione (RedDotBaseNode)**: Questo nodo sottoscrive i topic delle immagini e le informazioni della camera dalla Intel RealSense, oltre al topic */joint_states*. La sua funzione principale è identificare il punto rosso nell'immagine, calcolare la sua posizione 3D e trasformarla nel frame di riferimento della base del robot. Per effettuare questa trasformazione, RedDotBaseNode invia una "Service Call" al servizio */compute_fk* (Forward Kinematics), fornito dall'ambiente MoveIt!. Una volta determinata la posizione del punto rosso nel frame corretto, essa viene pubblicata sul topic */target_position*.

- **Nodo di Controllo (RobotController)**: Il RobotController è il nodo centrale per la pianificazione del movimento. Sottoscrive il topic */target_position* per ricevere la posizione desiderata del punto rosso. Al suo interno, il RobotController interagisce strettamente con l'IKClient. Il nodo IKClient ha il compito di gestire le richieste di

cinematica inversa e diretta: sottoscrive */joint_states* per ottenere lo stato attuale del robot e invia "Service Call" ai servizi */compute_ik* (Inverse Kinematics) e */compute_fk* per ottenere le configurazioni dei giunti o le pose dell'end-effector. Il RobotController utilizza queste informazioni per calcolare la prossima posizione che il robot deve raggiungere, applicando logiche di velocità e sicurezza. Infine, il RobotController pubblica i comandi dei giunti calcolati sul topic */robot/joint_commands*.

- **Nodo di Esecuzione del Movimento (SimpleRobotMover)**: Questo nodo è responsabile della traduzione dei comandi di alto livello in azioni fisiche del robot. Sottoscrive il topic */robot/joint_commands* pubblicato dal RobotController. Al ricevimento di un comando, il SimpleRobotMover prepara un "Goal" per l'Action Server ROS2 denominato */scaled_joint_trajectory_controller/follow_joint_trajectory*. Questo Action Server, parte integrante dei driver del robot UR5, è incaricato di eseguire la traiettoria dei giunti sul manipolatore. Dopo l'esecuzione del movimento, il SimpleRobotMover effettua un'ulteriore "Service call" al servizio */compute_fk* per determinare la posizione effettiva raggiunta dall'end-effector. Questi dati di posizione finale sono essenziali per l'analisi delle prestazioni del sistema e vengono loggati per una successiva valutazione.

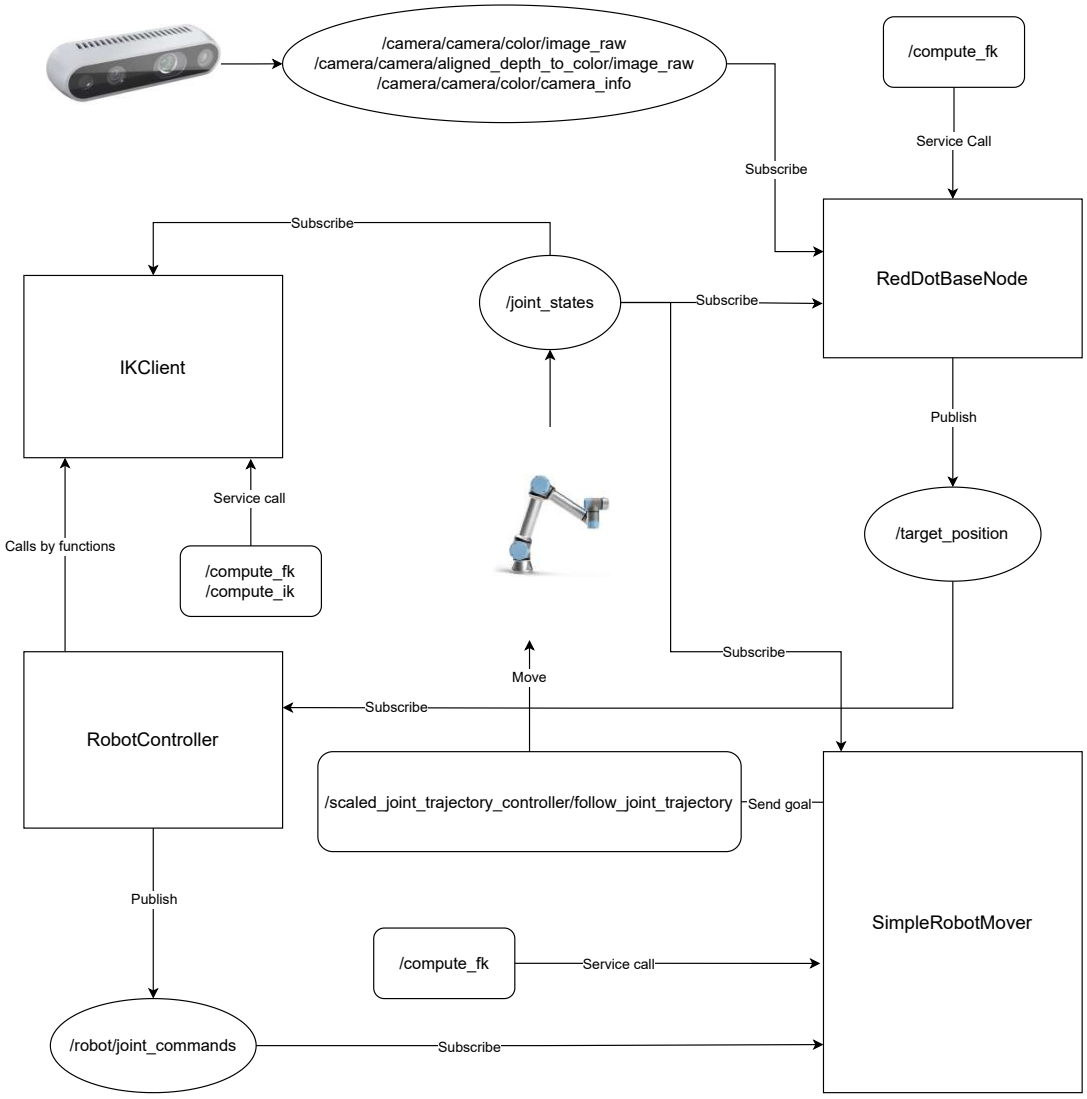

Figure 3.1: *Diagramma dell'architettura dei nodi ROS2 del sistema*

Nelle sezioni successive, verranno analizzati tutti gli script Python commentandoli nel dettaglio, spiegando la logica implementativa e il funzionamento specifico.

3.1 Analisi script del nodo di percezione: red_dot_tracker.py

Lo script red_dot_tracker.py implementa l'importante nodo RedDotBaseNode che costituisce il modulo di percezione visiva del sistema, responsabile dell'identificazione di un punto rosso nel telo e della determinazione della sua posizione tridimensionale rispetto alla base del robot. Questo script Python integra funzionalità di visione artificiale, acquisizione dati da sensori e trasformazioni di coordinate, operando come il primo anello della catena di controllo che

trasforma i dati grezzi in informazioni significative per la pianificazione del movimento. Nella Figura 3.2 si può osservare il diagramma di flusso esplicativo di questo script e del nodo RedDotBaseNode.

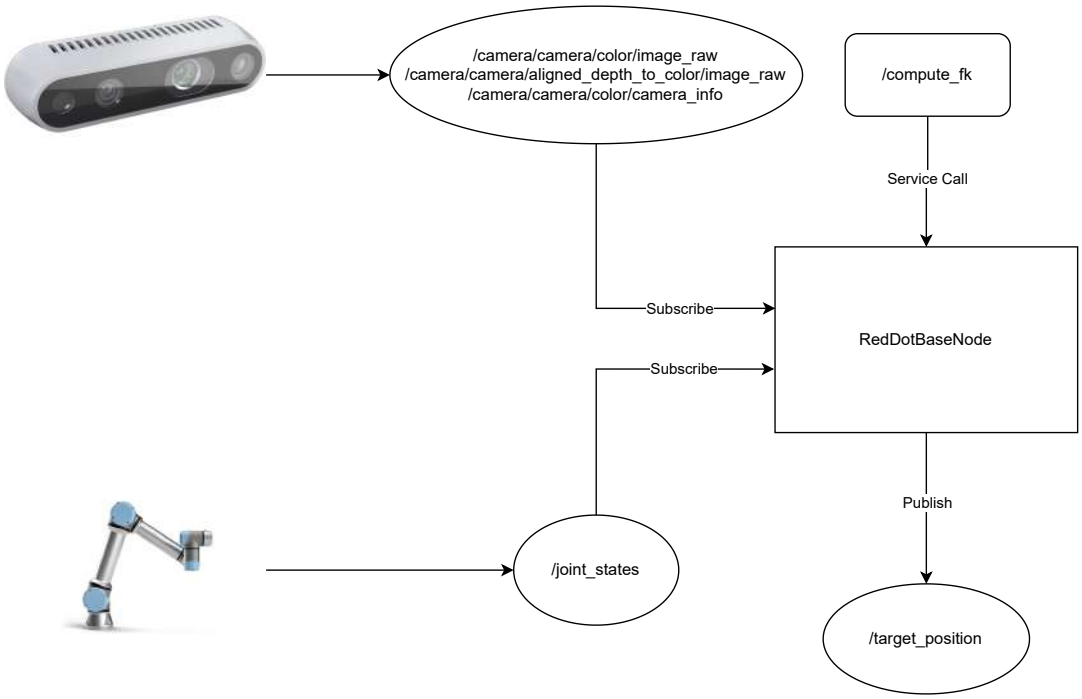

Figure 3.2: *Diagramma relativo allo script red_dot_tracker.py*

Di seguito, viene riportata un'analisi dettagliata delle principali parti di codice che compongono lo scipt totale.

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

import rclpy
from scipy.spatial.transform import Rotation as R
import cv2
import numpy as np
from rclpy.node import Node
from sensor_msgs.msg import Image, CameraInfo, JointState
from geometry_msgs.msg import Point
from moveit_msgs.srv import GetPositionFK
from std_msgs.msg import Header
from cv_bridge import CvBridge
```

Listato 3.1: Librerie utilizzate in red_dot_tracker.py

Il codice nel Listato 3.1 include gli import necessari per le funzionalità del nodo:

- rclpy: La libreria client Python per ROS2, fondamentale per creare nodi, publisher, subscriber e client/server di servizi/action.

- scipy.spatial.transform.Rotation as R: Utilizzata per manipolare rotazioni tridimensionali, in particolare per convertire quaternioni in matrici di rotazione, indispensabile per le trasformazioni di coordinate.
- cv2 (OpenCV): La libreria di visione artificiale, impiegata per l'elaborazione delle immagini (conversione di spazio colore, mascheramento, rilevamento contorni, calcolo del baricentro e visualizzazione).
- numpy as np: Fondamentale per operazioni numeriche su array, specialmente per calcoli vettoriali e matriciali nelle trasformazioni 3D.
- rclpy.node.Node: La classe base da cui ogni nodo ROS2 deve ereditare.
- sensor_msgs.msg.Image, CameraInfo, JointState: Tipi di messaggi ROS2 utilizzati per immagini (colore e profondità), parametri intrinseci della telecamera e lo stato dei giunti del robot, rispettivamente.
- geometry_msgs.msg.Point: Tipo di messaggio ROS2 per rappresentare un punto 3D, utilizzato per pubblicare la posizione del punto rosso.
- moveit_msgs.srv.GetPositionFK: Tipo di servizio ROS2 per richiedere la cinematica diretta (Forward Kinematics) a MoveIt!, permettendo di ottenere la posa di un link del robot data una configurazione dei giunti.
- std_msgs.msg.Header: Un tipo di messaggio standard ROS2 che include informazioni comuni come il timestamp e l'ID del frame, utilizzate nelle richieste di servizio.
- cv_bridge.CvBridge: Un bridge che consente la conversione tra i messaggi immagine di ROS2 e i formati immagine di OpenCV, facilitando l'elaborazione delle immagini.

```
class RedDotToBaseNode(Node):
    def __init__(self):
        super().__init__('red_dot_to_base_node')

        self.bridge = CvBridge()

        self.depth_image = None
        self.intrinsics = None
        self.joint_names = None
        self.joint_positions = None
        self.latest_color_image = None

        self.cli = self.create_client(GetPositionFK, '/compute_fk')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Aspettando il servizio /compute_fk...')

        self.req = GetPositionFK.Request()
        self.req.header = Header(frame_id='base_link')
        self.req.fk_link_names = ['tool0']

        self.create_subscription(Image, '/camera/camera/color/image_raw',
            self.image_callback, 10)
```

```
22          self.create_subscription(Image, '/camera/camera/
                aligned_depth_to_color/image_raw', self.depth_callback, 10)
23          self.create_subscription(CameraInfo, '/camera/camera/color/
                camera_info', self.camera_info_callback, 10)
24          self.create_subscription(JointState, '/joint_states', self.
                joint_state_callback, 10)
25
26          self.target_pub = self.create_publisher(Point, '/target_position',
                10)
```

Listato 3.2: Inizializzazione del nodo e delle sottoscrizioni

Nel Listato 3.2, il nodo viene inizializzato. Viene istanziato un oggetto CvBridge per facilitare la conversione tra i messaggi immagine di ROS2 (sensor_msgs/Image) e i formati immagine di OpenCV, essenziali per l'elaborazione della visione artificiale. Vengono dichiarate alcune variabili d'istanza per memorizzare i dati in arrivo da sensori e robot, come l'immagine di profondità (self.depth_image), i parametri intrinseci della camera (self.intrinsics), e lo stato dei giunti del robot (self.joint_names, self.joint_positions).
Un passaggio fondamentale è la creazione di un client per il servizio /compute_fk (Forward Kinematics). Questo servizio, fornito da pacchetti di cinematica di MoveIt!, è utilizzato per calcolare la posa (posizione e orientamento) di un link del robot (in questo caso, tool0) rispetto a un frame di riferimento specificato (base_link), data una configurazione dei giunti. Infine, vengono stabilite le sottoscrizioni ai topic ROS2:

- /camera/camera/color/image_raw: per l'immagine a colori dalla Intel RealSense.
- /camera/camera/aligned_depth_to_color/image_raw: per l'immagine di profondità allineata.
- /camera/camera/color/camera_info: per i parametri intrinseci della telecamera.
- /joint_states: per ricevere lo stato attuale dei giunti del robot UR5.

Viene inoltre creato un publisher sul topic /target_position che sarà utilizzato per pubblicare la posizione 3D del punto rosso rilevato, sotto forma di messaggio geometry_msgs/Point.

```
1  def joint_state_callback(self, msg):
2          self.joint_names = msg.name
3          self.joint_positions = msg.position
4
5  def camera_info_callback(self, msg):
6      if not self.intrinsics:
7          self.intrinsics = {
8              'fx': msg.k[0],
9              'fy': msg.k[4],
10             'ppx': msg.k[2],
11             'ppy': msg.k[5]
12         }
13
14 def depth_callback(self, msg):
15     try:
16         self.depth_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='
               passthrough')
17     except Exception as e:
```

```
18          self.get_logger().error(f'Errore nella conversione depth: {e}')
```

Listato 3.3: Callback per l'acquisizione dati

Nel Listato 3.3 sono descritte le funzioni di callback che gestiscono i dati in arrivo dai sensori della camera e dal robot. Queste funzioni vengono invocate automaticamente ogni volta che un nuovo messaggio è disponibile sul topic sottoscritto:

- joint_state_callback(self, msg): Questa funzione viene attivata ogni volta che il nodo riceve un messaggio JointState sul topic /joint_states. Al suo interno, vengono semplicemente aggiornate le variabili d'istanza self.joint_names e self.joint_positions con i nomi e le posizioni attuali dei giunti del robot. Questi dati sono fondamentali per la successiva chiamata al servizio di cinematica diretta.

- camera_info_callback(self, msg): Questa callback è progettata per acquisire i parametri intrinseci della telecamera (matrice K). Viene eseguito solo una volta, alla connessione della telecamera (controllando if not self.intrinsics), per salvare i valori di fx, fy, ppx e ppy, essenziali per la trasformazione da coordinate pixel a coordinate metriche 3D.

- depth_callback(self, msg): Gestisce i messaggi Image provenienti dal topic dell'immagine di profondità. Utilizza CvBridge per convertire il messaggio ROS2 in un formato OpenCV (numpy array), con desired_encoding='passthrough' per mantenere il tipo di dati originale dell'immagine di profondità. Viene inclusa una gestione delle eccezioni per catturare eventuali errori durante la conversione.

```
1  def image_callback(self, msg):
2          try:
3              color_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='
                   bgr8')
4              self.latest_color_image = color_image.copy()
5          except Exception as e:
6              self.get_logger().error(f'Errore RGB: {e}')
7              return
8
9          if self.intrinsics is None or self.depth_image is None or self.
               joint_positions is None:
10             return
11
12         hsv = cv2.cvtColor(color_image, cv2.COLOR_BGR2HSV)
13         mask1 = cv2.inRange(hsv, np.array([0, 120, 70]), np.array([10, 255,
               255]))
14         mask2 = cv2.inRange(hsv, np.array([170, 120, 70]), np.array([180,
               255, 255]))
15         mask = mask1 | mask2
16         contours, _ = cv2.findContours(mask, cv2.RETR_TREE, cv2.
               CHAIN_APPROX_SIMPLE)
17
18         if not contours:
19             cv2.imshow("RGB", color_image)
20             cv2.waitKey(1)
21             return
22
23         largest = max(contours, key=cv2.contourArea)
```

```
24          M = cv2.moments(largest)
25          if M["m00"] == 0:
26              return
27          cx = int(M["m10"] / M["m00"])
28          cy = int(M["m01"] / M["m00"])
29
30          if 0 <= cy < self.depth_image.shape[0] and 0 <= cx < self.depth_image
                .shape[1]:
31              depth_value = self.depth_image[cy, cx] / 1000.0
32              if not (0.01 < depth_value < 2.0):
33                  return
34
35              fx = self.intrinsics['fx']
36              fy = self.intrinsics['fy']
37              ppx = self.intrinsics['ppx']
38              ppy = self.intrinsics['ppy']
39
40              point_cam = np.array([
41                  (cx - ppx) * depth_value / fx,
42                  (cy - ppy) * depth_value / fy,
43                  depth_value
44              ])
45
46              ref_point=(643,103)
47              cv2.circle(color_image, ref_point, radius=5, color=(255,0,0),
                    thickness=-1)
48              cv2.circle(color_image, (cx, cy), 5, (0, 255, 0), -1)
49              cv2.putText(color_image, f"{depth_value:.2f} m", (cx+10, cy-10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,255,0), 1)
50              cv2.imshow("RGB", color_image)
51              cv2.waitKey(1)
52
53              self.req.robot_state.joint_state.name = self.joint_names
54              self.req.robot_state.joint_state.position = self.joint_positions
55
56              future = self.cli.call_async(self.req)
57              future.add_done_callback(lambda fut: self.handle_fk_response(fut,
                     point_cam))
```

Listato 3.4: Elaborazione dell'immagine e rilevamento del punto rosso

La funzione image_callback presente nel Listato 3.4 è il cuore del modulo di percezione. Per prima cosa, converte l'immagine a colori (sensor_msgs/Image) in un formato OpenCV (bgr8) utilizzando CvBridge.
Il processo di riconoscimento e calcolo della profondità del punto rosso include i seguenti passaggi sequenziali:

1. **Conversione in HSV e Mascheramento**: L'immagine a colori viene convertita nello spazio colore HSV (Hue, Saturation, Value). Vengono create due maschere binarie (mask1, mask2) utilizzando cv2.inRange per isolare i pixel che rientrano nell'intervallo di colore rosso. La prima maschera (mask1) copre l'intervallo HSV da [0,120,70] a [10,255,255], mentre la seconda maschera (mask2) copre l'intervallo da [170,120,70] a [180,255,255]. Queste due maschere sono necessarie perché il colore rosso ha valori di tonalità che "avvolgono" lo spettro cromatico, situandosi alle sue estremità.

2. **Rilevamento Contorni e Baricentro**: Sull'immagine filtrata con la doppia maschera, cv2.findContours viene utilizzato per identificare i contorni degli oggetti rossi. Viene selezionato il contorno più grande, assumendo che corrisponda al punto rosso di interesse. Il baricentro (cx, cy) di questo contorno viene calcolato usando i momenti di immagine (cv2.moments).

3. **Estrazione Profondità e Calcolo 3D**: Una volta identificato il centro del punto rosso nell'immagine 2D, si estrae il valore di profondità corrispondente dalla self.depth_image (convertito in metri). Con le coordinate pixel (cx, cy), il valore di profondità e i parametri intrinseci della telecamera (fx, fy, ppx, ppy), viene calcolata la posizione tridimensionale del punto (point_cam) nel frame della telecamera, utilizzando le equazioni di proiezione prospettica inversa.

4. **Visualizzazione (Debug)**: Il codice include funzionalità di debug per visualizzare il punto rosso rilevato e la sua profondità sull'immagine a colori, mostrando anche un punto di riferimento blu, calcolato come il punto che il robot deve idealmente "puntare" in ogni istante.

5. **Richiesta di Cinematica Diretta (FK)**: Prima di trasformare la posizione del punto rosso nel frame del robot, è necessario conoscere la posa attuale del tool0 (il punto di riferimento sull'end-effector) rispetto alla base del robot. Per fare ciò, il nodo prepara una richiesta (self.req) per il servizio /compute_fk, popolandola con i nomi e le posizioni attuali dei giunti (self.joint_names, self.joint_positions). La richiesta viene inviata in modo asincrono tramite self.cli.call_async(self.req), e una callback (handle_fk_response) viene registrata per gestire la risposta non appena disponibile, passando ad essa la posizione 3D del punto nel frame della camera.

```
def handle_fk_response(self, future, point_cam):

        try:
            pose = future.result().pose_stamped[0].pose
            pos = np.array([pose.position.x, pose.position.y, pose.position.z
                ])
            quat = [pose.orientation.x, pose.orientation.y, pose.orientation.
                z, pose.orientation.w]
            R_tool = R.from_quat(quat).as_matrix()

            offset = np.array([0.0, -0.06, -0.035])
            cam_origin = pos + R_tool @ offset
            R_cam = R_tool
            point_base = cam_origin + R_cam @ point_cam

            self.get_logger().info(f"Punto rosso nel frame base_link: X={
                point_base[0]:.2f}, Y={point_base[1]:.2f}, Z={point_base[2]:.2
                f}")

            msg = Point()
            msg.x = round(float(point_base[0]), 2)
            msg.y = round(float(point_base[1]), 2)
            msg.z = round(float(point_base[2]), 2)
            self.target_pub.publish(msg)
```

```
21
22
23          except Exception as e:
24              self.get_logger().error(f'Errore FK: {e}')
```

Listato 3.5: Trasformazione di coordinate e pubblicazione

Nel Listato 3.5 è presente la funzione handle_fk_response, le cui operazioni sono:

1. **Recupero Posa tool0**: Viene estratta la posa (posizione e orientamento) del tool0 dal risultato del servizio FK. La posizione è un vettore 3D e l'orientamento è un quaternione, che viene convertito in matrice di rotazione (R_tool) usando scipy.spatial.transform.Rotation.

2. **Calcolo Posa Camera**: Viene definito un offset rigido (misurato a priori in base alla posizione della telecamera nel robot) che descrive la traslazione della telecamera rispetto al tool0 del robot. La posizione dell'origine della telecamera nel frame base_link (cam_origin) viene calcolata sommando la posizione del tool0 con la rotazione dell'end-effector applicata all'offset. Si assume che l'orientamento della telecamera (R_cam) sia lo stesso del tool0.

3. **Trasformazione del Punto**: La posizione 3D del punto rosso nel frame della telecamera (point_cam) viene trasformata nel frame base_link (point_base) applicando la rotazione e la traslazione dell'origine della telecamera.

4. **Pubblicazione**: La posizione finale del punto rosso nel frame base_link viene arrotondata e incapsulata in un messaggio geometry_msgs/Point. Questo messaggio viene infine pubblicato sul topic /target_position, rendendolo disponibile per i nodi successivi nel pipeline di controllo del robot.

5. **Gestione Errori**: Un blocco try-except cattura e logga eventuali errori come problemi col servizio di MoveIt!, stato dei giunti non valido, che possono verificarsi durante l'elaborazione della risposta FK, garantendo la robustezza del nodo.

```
1  def main():
2      rclpy.init()
3      node = RedDotToBaseNode()
4      try:
5          rclpy.spin(node)
6      except KeyboardInterrupt:
7          pass
8      finally:
9          node.destroy_node()
10         rclpy.shutdown()
11         cv2.destroyAllWindows()
12
13 if __name__ == '__main__':
14     main()
```

Listato 3.6: Funzione main() dello script red_dot_tracker.py

Infine, nel Listato 3.6, è presente la funzione main() che è il punto di ingresso dello script. Inizializza l'ambiente ROS2 (rclpy.init()), crea un'istanza del nodo RedDotToBaseNode e avvia il ciclo di esecuzione ROS2 (rclpy.spin(node)). Il rclpy.spin blocca il processo finché

il nodo non viene interrotto (ad esempio, con Ctrl+C). Un blocco try-finally assicura che il nodo venga distrutto correttamente (node.destroy_node()), l'ambiente ROS2 venga spento (rclpy.shutdown()) e tutte le finestre OpenCV vengano chiuse (cv2.destroyAllWindows()) al termine dell'esecuzione, anche in caso di interruzione.

3.2 Analisi script del nodo di controllo: trajectory_sender.py

Lo script Python trajectory_sender.py racchiude la logica centrale per la pianificazione del movimento del robot. Questo script è cruciale in quanto gestisce sia le complesse operazioni di cinematica (diretta e inversa) sia il ciclo di controllo che traduce la posizione desiderata di un target in comandi specifici per i giunti del robot. Al suo interno, sono definite due classi principali che operano sinergicamente: la classe IKClient, dedicata all'interfacciamento con i servizi di cinematica di MoveIt!, e la classe RobotController, che implementa l'algoritmo di controllo basato sull'input del punto rosso per generare le traiettorie del robot. Nella Figura 3.3 si può osservare il diagramma di flusso esplicativo di questo script e dei nodi IKClient e RobotController.

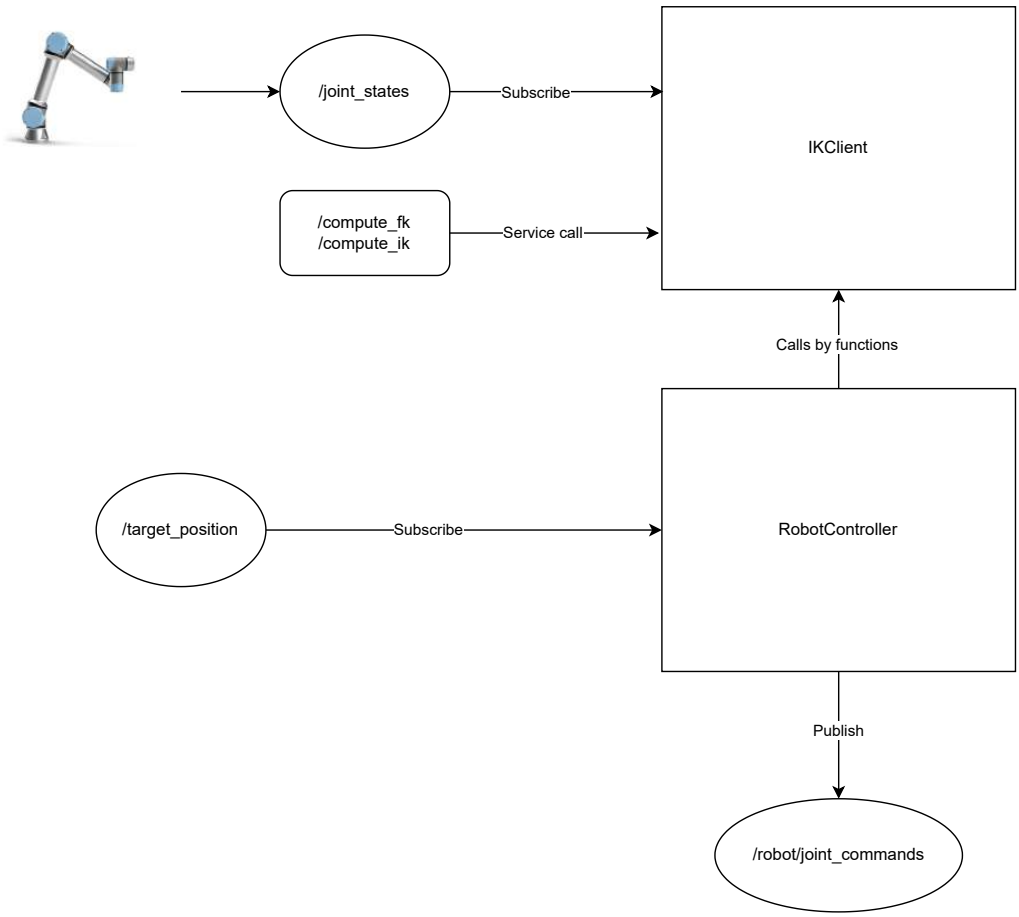

Figure 3.3: *Diagramma relativo allo script trajectory_sender.py*

Di seguito, viene riportata un'analisi dettagliata delle principali parti di codice che compongono lo script totale.

```
1  #!/usr/bin/env python3
2
3  import rclpy
4  import csv
5  from datetime import datetime
6  import numpy as np
7  from rclpy.node import Node
8  from moveit_msgs.msg import JointConstraint
9  from moveit_msgs.srv import GetPositionIK, GetPositionFK
10 from geometry_msgs.msg import PoseStamped, Pose, Point, Quaternion
11 from sensor_msgs.msg import JointState
12 from std_msgs.msg import Header
13
14 JOINT_NAMES = [
15     "shoulder_pan_joint", "shoulder_lift_joint", "elbow_joint",
16     "wrist_1_joint", "wrist_2_joint", "wrist_3_joint"
```

```
17 ]
18
19 TIMER = 0.4
20 MAX_SPEED=0.1
```

Listato 3.7: Librerie e costanti usate nello script trajectory_sender.py

Il Listato 3.7 contiene la sezione di codice che definisce le dipendenze e alcune costanti cruciali per il funzionamento dei nodi IKClient e RobotController.
Le librerie usate sono:

- rclpy: Libreria client Python per ROS2, essenziale per l'interazione con l'ambiente ROS2.
- csv: Utilizzata per la gestione e la scrittura su file CSV, impiegata per il logging delle posizioni del robot.
- datetime: Permette di generare timestamp per i dati loggati.
- numpy as np: Fondamentale per operazioni numeriche su array e vettori, ampiamente utilizzata nei calcoli di posizione e traiettoria.
- rclpy.node.Node: La classe base per tutti i nodi ROS2.
- moveit_msgs.msg.JointConstraint: Tipo di messaggio per definire vincoli sui giunti nelle richieste di cinematica inversa.
- moveit_msgs.srv.GetPositionIK, GetPositionFK: Tipi di servizio ROS2 per richiedere rispettivamente la cinematica inversa (IK) e la cinematica diretta (FK) a MoveIt!.
- geometry_msgs.msg.PoseStamped, Pose, Point, Quaternion: Tipi di messaggi ROS2 per rappresentare pose (posizione e orientamento), punti e quaternioni, utilizzati per definire e comunicare le posizioni nello spazio 3D.
- sensor_msgs.msg.JointState: Tipo di messaggio ROS2 per lo stato dei giunti del robot (nomi e posizioni).
- std_msgs.msg.Header: Messaggio standard ROS2 che include informazioni comuni come timestamp e frame ID.

Le variabili globali definite, invece, sono:

- JOINT_NAMES: Una lista costante che definisce i nomi dei sei giunti del robot UR5 in un ordine specifico. Questo è cruciale per garantire che i comandi e le letture dei giunti siano sempre allineati correttamente.
- TIMER: Imposta il tempo di campionamento del ciclo di controllo in secondi (0.4 s). Questo valore determina la frequenza con cui il RobotController calcola e invia nuovi comandi di movimento.
- MAX_SPEED: Definisce la velocità massima lineare consentita per l'end-effector del robot in metri al secondo (0.1 m/s). Questa costante è utilizzata per limitare la dimensione dei passi di movimento, garantendo un'operazione fluida e sicura.

```
class IKClient(Node):

    def __init__(self):
        super().__init__('ik_client')
        self.cli = self.create_client(GetPositionIK, '/compute_ik')
        self.fk_cli = self.create_client(GetPositionFK, '/compute_fk')

        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Attendo /compute_ik...')
        while not self.fk_cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Attendo /compute_fk...')

        self.ik_req = GetPositionIK.Request()
        self.fk_req = GetPositionFK.Request()

        self.joint_state = None

        self.subscription = self.create_subscription(
            JointState,
            '/joint_states',
            self.joint_states_callback,
            10
        )

    def joint_states_callback(self, msg):
        self.joint_state = msg
        self.subscription = None
```

Listato 3.8: Inizializzazione IKClient e funzioni di callback

Nel Listato 3.8 c'è l'inizializzazione della classe IKClient che è un nodo ROS2 ausiliario, progettato per gestire tutte le interazioni con i servizi di cinematica inversa (IK) e diretta (FK) forniti dall'ambiente MoveIt!. Nonostante sia un nodo a sé stante, nel contesto di questo script, la sua istanza viene creata e utilizzata direttamente dalla classe RobotController, agendo come un'interfaccia specializzata per le trasformazioni cinematiche.
All'interno del costruttore, il nodo IKClient viene inizializzato con il nome "ik_client". Contestualmente, vengono creati i client per i servizi ROS2 di cinematica inversa (/compute_ik) e cinematica diretta (/compute_fk), entrambi forniti da MoveIt!, e il nodo attende in modo proattivo che questi servizi siano disponibili prima di procedere. Vengono inoltre preparati gli oggetti per le richieste IK e FK, e una variabile self.joint_state viene allocata per memorizzare l'ultimo stato dei giunti del robot. Fondamentale per il suo funzionamento, il nodo sottoscrive il topic /joint_states, assicurandosi di ricevere in tempo reale la configurazione attuale dei giunti, dato che tali informazioni sono spesso indispensabili per risolvere i problemi di cinematica.
La funzione joint_states_callback(self, msg) è una semplice callback che viene invocata la prima volta che un nuovo messaggio JointState viene ricevuto sul topic /joint_states. Il suo unico scopo è aggiornare la variabile self.joint_state con i dati più recenti. Viene impostato self.subscription = None poichè IKClient necessita di sapere lo stato dei giunti solo alla prima iterazione, in modo da conoscere la configurazione iniziale, nelle successive iterazioni non riceve più aggiornamenti continui sullo stato dei giunti. Dunque, sostanzialmente, tale funzione di callback ha uno scopo solo nella prima iterazione del loop.

```
def get_end_effector_pose(self):

        if not hasattr(self, 'joint_state') or len(self.joint_state.position)
             != 6:
            self.get_logger().warn("Joint state non valido o non ancora
                ricevuto.")
            return None

        fk_req = GetPositionFK.Request()
        fk_req.header = Header(frame_id='base_link')
        fk_req.fk_link_names = ['tool0']
        fk_req.robot_state.joint_state.name = self.joint_state.name
        fk_req.robot_state.joint_state.position = self.joint_state.position
        future = self.fk_cli.call_async(fk_req)
        rclpy.spin_until_future_complete(self, future)

        if not future.result() or len(future.result().pose_stamped) == 0:
            self.get_logger().error("Errore nel calcolo della FK.")
            return None

        return future.result().pose_stamped[0].pose

def send_ik_request(self, pose):

    self.ik_req.ik_request.group_name = 'ur_manipulator'
    self.ik_req.ik_request.pose_stamped = PoseStamped()
    self.ik_req.ik_request.pose_stamped.header.frame_id = 'base_link'
    self.ik_req.ik_request.pose_stamped.pose = pose

    current_position = self.joint_state.position[
        self.joint_state.name.index("wrist_3_joint")
    ]

    constraint = JointConstraint()
    constraint.joint_name = "wrist_3_joint"
    constraint.position = current_position
    constraint.tolerance_above = 0.6
    constraint.tolerance_below = 0.6
    constraint.weight = 1.0
    self.ik_req.ik_request.constraints.joint_constraints = [constraint]

    future = self.cli.call_async(self.ik_req)
    rclpy.spin_until_future_complete(self, future)
    result = future.result()

    if result:
        if result.error_code.val != 1:
            self.get_logger().error(f"Errore nella richiesta IK: {result.
                error_code.val}")
        else:
            self.get_logger().info("Soluzione IK trovata.")
    else:
        self.get_logger().error("Errore nella risposta IK.")

    return result
```

Listato 3.9: Metodi di IKClient per la cinematica

Nel Listato 3.9 sono presenti i due metodi che IKClient offre per interagire con i servizi di cinematica di MoveIt!: get_end_effector_pose() per la cinematica diretta e send_ik_request() per la cinematica inversa.

- get_end_effector_pose(self): Questo metodo calcola la posa corrente dell'end-effector (tool0) rispetto al frame base_link utilizzando il servizio di cinematica diretta (/compute_fk). Dopo aver verificato la validità dello stato dei giunti, popola la richiesta FK e la invia in modo asincrono, bloccando l'esecuzione fino alla ricezione della risposta per poi restituire la posa calcolata o loggare un errore in caso di fallimento.
- send_ik_request(self, pose): Questo metodo gestisce l'invio di una richiesta di cinematica inversa (IK) al servizio /compute_ik al fine di determinare una configurazione dei giunti che porti l'end-effector alla pose desiderata. La richiesta specifica il gruppo manipolatore e la posa target, e include un vincolo sul giunto wrist_3_joint per stabilizzare l'orientamento. La richiesta IK viene inviata in modo asincrono, e il nodo attende il risultato, validando l'error_code della risposta prima di restituire la soluzione dei giunti o segnalare un errore.

```
class RobotController(Node):

    def __init__(self, ik_client):

        super().__init__('robot_controller')

        self.ik_client = ik_client #salva il client IK
        self.red_dot = None

        self.subscription = self.create_subscription(
            '/target_position',
            self.target_callback,
            10
        )

        self.joint_command_publisher = self.create_publisher(JointState, '/
            robot/joint_commands', 10)
        self.timer = self.create_timer(TIMER, self.execute)

        self.next_pos_file = open("next_pos_log.csv", "w", newline='')
        self.next_pos_writer = csv.writer(self.next_pos_file)
        self.next_pos_writer.writerow(["timestamp", "x", "y", "z"])

        self.target_pos_file = open("target_pos_log.csv", "w", newline='')
        self.target_pos_writer = csv.writer(self.target_pos_file)
        self.target_pos_writer.writerow(["timestamp", "x", "y", "z"])
```

Listato 3.10: Inizializzazione della classe RobotController

Il Listato 3.10 contiene l'inizializzazione del nodo RobotController configurato per gestire il controllo del robot. Riceve e memorizza un'istanza dell'IKClient per accedere ai servizi di cinematica, e inizializza una variabile per la posizione del punto rosso, che sarà aggiornata tramite una sottoscrizione al topic /target_position. Il nodo crea inoltre un publisher per inviare i comandi ai giunti del robot sul topic /robot/joint_commands. Un timer ROS2 viene impostato per attivare periodicamente la funzione execute(), che costituisce il cuore del ciclo

di controllo. Infine, il costruttore prepara e apre due file CSV per il logging, dove verranno registrate le posizioni calcolate e quelle target per successive analisi.

```
def target_callback(self, msg):
        self.red_dot = np.array([msg.x, msg.y, msg.z])

def is_trajectory_safe(self, current_joints, target_joints, thresholds):

    for i, (name, c, t, thr) in enumerate(zip(JOINT_NAMES, current_joints,
        target_joints, thresholds)):
        diff = abs(t - c)
        if diff > thr:
            self.get_logger().warn(
                f"[SAFETY] Giunto '{name}' supera soglia: |{t:.2f} - {c:.2f}|
                    = {diff:.2f} > {thr:.2f}"
            )
            return False
    return True

def destroy_node(self):

    self.next_pos_file.close()
    self.target_pos_file.close()
    super().destroy_node()
```

Listato 3.11: Funzioni di supporto di RobotController

Il Listato 3.11 contiene le seguenti funzioni di supporto, appartenenti alla classe RobotController:

- target_callback(self, msg): Questa funzione viene automaticamente richiamata ogni volta che un nuovo messaggio Point viene ricevuto sul topic /target_position. Il suo unico compito è aggiornare la variabile d'istanza self.red_dot con le coordinate X, Y, Z del punto rosso, convertendole in un array NumPy per facilitare i calcoli successivi.

- is_trajectory_safe(self, current_joints, target_joints, thresholds): Questa funzione di sicurezza è cruciale per prevenire movimenti bruschi o potenzialmente dannosi del robot. Prende in input le posizioni attuali dei giunti, le posizioni target dei giunti (calcolate dall'IK) e una lista di soglie massime di variazione per ciascun giunto. Itera attraverso ogni giunto, confrontando la differenza assoluta tra la posizione corrente e quella target con la soglia definita per quel giunto. Se la variazione per un qualsiasi giunto supera la sua soglia, la funzione logga un avviso di sicurezza e restituisce False, indicando che la traiettoria non è considerata sicura. Se tutte le variazioni rientrano nelle soglie, la funzione restituisce True. Questa verifica viene eseguita prima di pubblicare i comandi al robot.

- destroy_node(self): Questo metodo viene invocato quando il nodo sta per essere spento (ad esempio, tramite Ctrl+C). Sono presenti le seguenti operazioni di pulizia: self.next_pos_file.close() e self.target_pos_file.close() assicurano che i file CSV utilizzati per il logging vengano correttamente chiusi, salvando tutti i dati scritti; super().destroy_node() chiama il metodo destroy_node della classe base Node di ROS2 per una corretta disattivazione delle risorse ROS2 del nodo.

```
def execute(self):

        if self.red_dot is None:
            self.get_logger().info("Aspettando coordinate dal topic /
                target_position...")
            return

        tool_pose = self.ik_client.get_end_effector_pose()
        if tool_pose is None:
            self.get_logger().error("Impossibile ottenere la pose del tool0."
                )
            return

        initial_orientation = Quaternion(
            x=0.6139006598391906,
            y=0.536655579596502,
            z=-0.44428853171535015,
            w=-0.3711259480596406
        )

        desired_orientation = initial_orientation
        tool_pos = np.array([
            tool_pose.position.x,
            tool_pose.position.y,
            tool_pose.position.z
        ])

        target_x = self.red_dot[0] + 1.0
        target_y = self.red_dot[1]
        target_z = np.clip(self.red_dot[2], 0.45, 0.60)
        target_pos = np.round([target_x, target_y, target_z], 2)

        displacement = target_pos - tool_pos
        dist = np.linalg.norm(displacement)
        self.get_logger().info(f"DISTANZA DA FARE: {dist:.4f} m")
        self.get_logger().info(f"[DEBUG] tool_pos: x={tool_pos[0]:.3f}, y={
            tool_pos[1]:.3f}, z={tool_pos[2]:.3f}")

        max_step = MAX_SPEED * TIMER
        if dist > max_step:
            direction = displacement / dist
            next_pos = tool_pos + direction * max_step
            reached_final_target = False
        else:
            next_pos = target_pos
            reached_final_target = True
        if reached_final_target:
            self.get_logger().info("[STEP] POSIZIONE DIRETTA.")
        else:
            self.get_logger().info("[STEP] POSIZIONE INTERMEDIA.")

        next_pos = np.round(next_pos, 2)
        desired_pose = Pose()
        desired_pose.position = Point(
            x=next_pos[0],
            y=next_pos[1],
```

```
            z=next_pos[2]
        )
        desired_pose.orientation = desired_orientation

        self.get_logger().info(
            f"[CHECK] TARGET POSE: {target_pos} m"
        )
        self.get_logger().info(
            f"[CHECK] NEXT POS: {next_pos} m"
        )
        self.get_logger().info(
            f"[CHECK] Distanza dal punto rosso: {np.linalg.norm(np.array([
                desired_pose.position.x, desired_pose.position.y, desired_pose
                .position.z]) - self.red_dot):.4f} m"
        )

        response = self.ik_client.send_ik_request(desired_pose)
        if response is None or response.error_code.val != 1:
            self.get_logger().error("IK fallita.")
            return
        joint_positions = list(response.solution.joint_state.position)
        joint_names = response.solution.joint_state.name
        ik_solution = {name: pos for name, pos in zip(joint_names,
            joint_positions)}
        current_joint_state = self.ik_client.joint_state
        if current_joint_state and 'wrist_3_joint' in current_joint_state.
            name:
            index = current_joint_state.name.index('wrist_3_joint')
            ik_solution['wrist_3_joint'] = current_joint_state.position[index
                ]
        ordered_positions = [ik_solution[name] for name in JOINT_NAMES]
        current_positions = [self.ik_client.joint_state.position[self.
            ik_client.joint_state.name.index(name)] for name in JOINT_NAMES]

        joint_thresholds = [1.5, 1.5, 1.3, 0.8, 0.8, 1.0]
        if not self.is_trajectory_safe(current_positions, ordered_positions,
            joint_thresholds):
            self.get_logger().error("Movimento annullato: cambiamento troppo
                brusco su uno o piu' giunti.")
            return

        joint_state_msg = JointState()
        joint_state_msg.header.stamp = self.get_clock().now().to_msg()
        joint_state_msg.name = JOINT_NAMES
        joint_state_msg.position = ordered_positions
        self.joint_command_publisher.publish(joint_state_msg)

        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")
        self.next_pos_writer.writerow([timestamp, next_pos[0], next_pos[1],
            next_pos[2]])
        self.next_pos_file.flush()
        self.target_pos_writer.writerow([timestamp, target_pos[0], target_pos
            [1], target_pos[2]])
        self.target_pos_file.flush()
```

Listato 3.12: Funzione di controllo principale: metodo execute()

Nel Listato 3.12 si può apprendere la logica implementativa del controllo effettuato. La funzione execute(), infatti, viene richiamata periodicamente dal timer e orchestra l'intero processo di controllo del movimento del robot. La si può suddividere in questa serie di operazioni sequenziali:

1. **Verifica Dati Disponibili**: Inizia controllando se la posizione del punto rosso è stata ricevuta dal nodo di percezione. Se non è ancora disponibile, il ciclo si interrompe e attende il prossimo tick del timer. Successivamente, ottiene la posa corrente dell'end-effector (tool0) utilizzando il metodo get_end_effector_pose() dell'ik_client. Se la posa non è disponibile, logga un errore e si interrompe.

2. **Definizione Orientamento Desiderato**: Viene definito un orientamento fisso per l'end-effector tramite un quaternione predefinito, misurato a priori e considerato ideale per il task (initial_orientation). Questo orientamento viene mantenuto costante durante l'intero inseguimento del target.

3. **Calcolo della Posizione Target Cartesiana**: La posizione attuale dell'end-effector (tool_pos) viene estratta dalla posa ottenuta, mentre la posizione desiderata finale (target_pos) viene calcolata in base alla posizione del punto rosso (self.red_dot). In particolare:

   - La coordinata X del target viene impostata 1 metro davanti la X del punto rosso.
   - La coordinata Y del target è mantenuta uguale alla Y del punto rosso.
   - La coordinata Z del target è mantenuta uguale alla Z del punto rosso, ma viene vincolata a rimanere tra 0.45 e 0.60 metri (np.clip), per mantenere l'end-effector ad un'altezza operativa sicura e utile.

   La target_pos viene arrotondata a due cifre decimali.

4. **Controllo di Velocità (Pianificazione del Passo)**: Viene calcolato il vettore displacement tra la posizione attuale dell'end-effector e la target_pos, successivamente viene determinata la distanza euclidea di questo spostamento. A questo punto, max_step viene calcolato come il prodotto tra la MAX_SPEED (velocità lineare massima consentita) e il TIMER (intervallo di tempo del ciclo). Questo definisce la distanza massima che il robot può percorrere in un singolo passo del timer, se la distanza totale da percorrere è maggiore di max_step, il robot non tenterà di raggiungere direttamente il target in un solo passo. Invece, viene calcolata una next_pos intermedia avanzando di max_step nella direzione del target. Questo implementa un semplice controllo di velocità lineare. Se, invece, la distanza è inferiore o uguale a max_step, significa che il robot può raggiungere la target_pos finale in questo singolo passo, e next_pos viene impostata direttamente su target_pos. La next_pos calcolata viene arrotondata a due cifre decimali per coerenza. Il calcolo della next_pos si può riassumere con la seguente formula:

$$\text{next\_pos} = \begin{cases} \text{tool\_pos} + \text{direction} \times \text{max\_step} & \text{se dist} > \text{max\_step} \\ \text{target\_pos} & \text{altrimenti (se dist} \leq \text{max\_step)} \end{cases}$$

5. **Costruzione della Posa Desiderata e Chiamata IK**: Viene costruita una desired_pose combinando la next_pos calcolata e la desired_orientation costante. Questa

desired_pose viene passata al metodo send_ik_request() dell'ik_client. Questo richiede a MoveIt! di trovare una configurazione dei giunti che permetta al robot di raggiungere quella posa cartesiana. Se la chiamata IK fallisce (la response è None o error_code.val non è 1), viene loggato un errore e la funzione si interrompe.

6. **Elaborazione della Soluzione IK e Vincolo sul Giunto wrist_3_joint**: Se l'IK ha successo, le posizioni dei giunti dalla response.solution vengono estratte in un dizionario ik_solution. Viene applicato un ulteriore vincolo manuale: la posizione del giunto wrist_3_joint nella soluzione IK viene forzatamente impostata alla sua posizione attuale (ottenuta da self.ik_client.joint_state). Questo serve a mantenere il tool con un orientamento costante.

7. **Controllo di Sicurezza is_trajectory_safe()**: Prima di comandare il robot, viene invocata la funzione self.is_trajectory_safe(), confrontando le posizioni attuali dei giunti con le ordered_positions (soluzione IK) usando joint_thresholds predefinite. Se la funzione is_trajectory_safe() restituisce False (indicando un movimento troppo brusco su uno o più giunti), viene loggato un errore e il movimento viene annullato (return), prevenendo potenziali danni.

8. **Pubblicazione dei Comandi dei Giunti**: Se tutti i controlli di sicurezza sono superati, viene creato un messaggio JointState a cui vengono assegnati il timestamp corrente, i JOINT_NAMES e le ordered_positions. Il messaggio JointState viene pubblicato sul topic /robot/joint_commands, inviando l'istruzione di movimento al nodo di esecuzione del movimento.

9. **Logging**: Un timestamp viene generato e sia la next_pos che la target_pos (coordinate cartesiane) vengono scritte nei rispettivi file CSV (next_pos_log.csv e target_pos_log.csv), fornendo un registro storico dei movimenti e degli obiettivi. flush() assicura che i dati vengano scritti immediatamente su disco.

```
def main(args=None):
    rclpy.init(args=args)
    ik_client = IKClient()
    rclpy.spin_once(ik_client)

    while ik_client.joint_state is None:
        rclpy.spin_once(ik_client, timeout_sec=0.1)
        ik_client.get_logger().info("Waiting for the first /joint_states
            message...")
    robot_controller = RobotController(ik_client)
    try:
        rclpy.spin(robot_controller)
    except KeyboardInterrupt:
        pass
    finally:

        robot_controller.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
```

```
20      main()
```

Listato 3.13: Funzione main() dello script trajectory_sender.py

Infine, nel Listato 3.13, è presente la funzione main() che avvia l'applicazione ROS2, inizializzando la libreria rclpy e creando un'istanza del nodo IKClient. Il programma attende poi che l'IKClient riceva il primo stato dei giunti prima di procedere. Successivamente, viene creata l'istanza del RobotController, a cui viene passato l'IKClient per la gestione della cinematica. Il nodo robot_controller viene quindi avviato nel suo ciclo di elaborazione principale e, in caso di interruzione (ad esempio, tramite Ctrl+C), la funzione main() garantisce la corretta chiusura delle risorse, inclusa la distruzione del nodo e lo shutdown di rclpy.

3.3 Analisi script del nodo di movimento: mover.py

Il file simple_robot_mover.py contiene il nodo ROS2 SimpleRobotMover, che rappresenta l'interfaccia di basso livello per il controllo e il movimento effettivo del robot. Questo nodo riceve i comandi di posizione dei giunti calcolati dal RobotController e li traduce in traiettorie che vengono inviate a un Action Server specifico (il FollowJointTrajectory) incaricato di eseguire il movimento fisico del robot. In aggiunta, SimpleRobotMover continua a monitorare lo stato dei giunti e della posa dell'end-effector per fini di logging. Nella Figura 3.4 si può osservare il diagramma di flusso esplicativo di questo script e del nodo SimpleRobotMover.

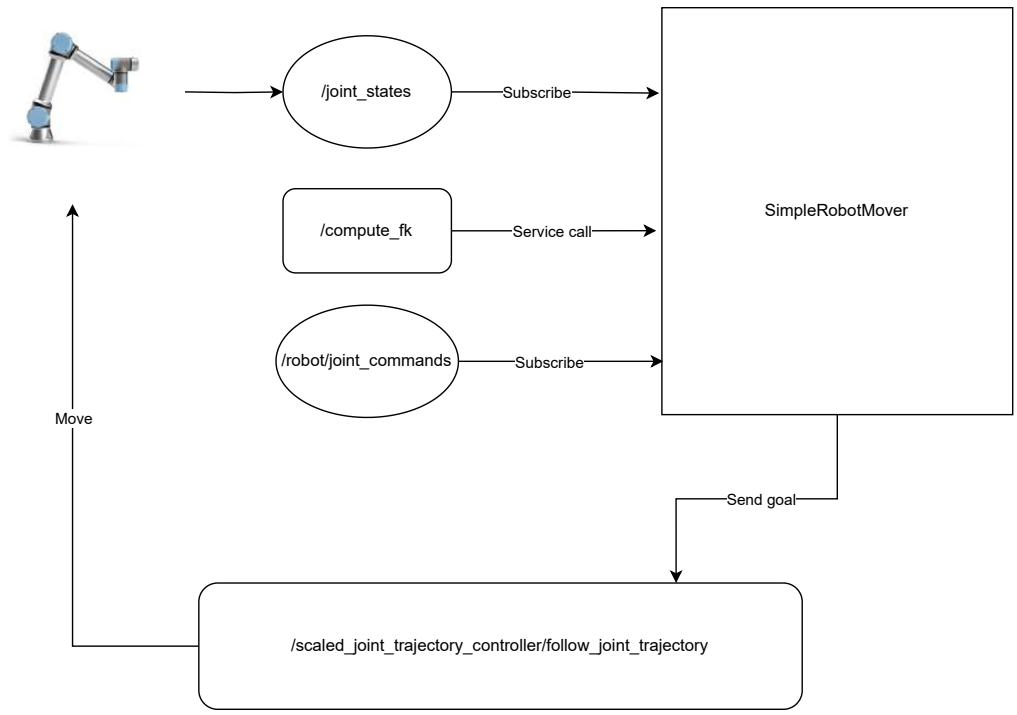

Figure 3.4: *Diagramma relativo allo script mover.py*

Di seguito, viene riportata un'analisi dettagliata delle principali parti di codice che compongono lo script totale.

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

import rclpy
import csv
from datetime import datetime
import numpy as np
from rclpy.node import Node
from rclpy.duration import Duration as rclpyDuration
from sensor_msgs.msg import JointState
from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
from control_msgs.action import FollowJointTrajectory
from moveit_msgs.srv import GetPositionFK
from rclpy.action import ActionClient
from std_msgs.msg import Header

JOINT_NAMES = [
    "shoulder_pan_joint", "shoulder_lift_joint", "elbow_joint",
    "wrist_1_joint", "wrist_2_joint", "wrist_3_joint"
]
ACTION_SERVER_NAME = '/scaled_joint_trajectory_controller/
    follow_joint_trajectory'
TIMER_DURATION_SECONDS = 0.4
```

Listato 3.14: Librerie e costanti usate nello script mover.py

Il codice presente nel Listato 3.14 definisce le dipendenze e le costanti necessarie per il funzionamento del nodo SimpleRobotMover.
Le librerie usate sono:

- rclpy: La libreria client Python per ROS2.
- csv, datetime, numpy as np: Usate per il logging dei dati e manipolazioni numeriche, come negli script precedenti.
- rclpy.node.Node: La classe base per i nodi ROS2.
- rclpy.duration.Duration as rclpyDuration: Utilizzata per specificare durate temporali nei messaggi ROS2.
- sensor_msgs.msg.JointState: Messaggio per lo stato dei giunti del robot.
- trajectory_msgs.msg.JointTrajectory, JointTrajectoryPoint: Messaggi specifici per definire traiettorie dei giunti, inclusi i punti di passaggio e i tempi.
- control_msgs.action.FollowJointTrajectory: Il tipo di azione ROS2 utilizzato per inviare comandi di traiettoria ai controllori del robot.
- moveit_msgs.srv.GetPositionFK: Utilizzato per richiedere la cinematica diretta.
- rclpy.action.ActionClient: La classe per creare un client per un'azione ROS2.
- std_msgs.msg.Header: Messaggio standard per includere informazioni comuni come il frame ID.

Le variabili globali definite, invece, sono:

- JOINT_NAMES: Una lista costante che definisce i nomi dei sei giunti del robot UR5 in un ordine specifico. Questo è cruciale per garantire che i comandi e le letture dei giunti siano sempre allineati correttamente.
- ACTION_SERVER_NAME: Stringa che definisce il nome dell'Action Server ROS2 (/scaled_joint_trajectory_controller/follow_joint_trajectory) a cui questo nodo invierà le traiettorie. Questo server è il punto finale che riceve i comandi e li esegue sul robot fisico o simulato.
- TIMER_DURATION_SECONDS: Imposta la durata (0.4 secondi) di ciascun punto della traiettoria dei giunti inviata al robot. Questo tempo coincide con il TIMER utilizzato dal RobotController, garantendo una sincronizzazione tra la frequenza di calcolo e l'esecuzione del movimento.

```
class SimpleRobotMover(Node):
    def __init__(self):
        super().__init__('simple_robot_mover_node')

        self.fk_cli = self.create_client(GetPositionFK, '/compute_fk')
        while not self.fk_cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Waiting for FK service...')
        self.get_logger().info("FK service available")
        self.joint_state = None

        self.joint_state_subscription = self.create_subscription(
            JointState,
            '/joint_states',
            self.joint_state_callback,
            10
        )

        self._action_client = ActionClient(self, FollowJointTrajectory,
            ACTION_SERVER_NAME)
        self.get_logger().info(f"Waiting for action server: {
            ACTION_SERVER_NAME}")
        self._action_client.wait_for_server()
        self.get_logger().info("Action server available!")

        self.joint_command_subscription = self.create_subscription(
            JointState,
            '/robot/joint_commands',
            self.joint_command_callback,
            10
        )
        self.get_logger().info(f"Subscribed to /robot/joint_commands")
        self.get_logger().info("SimpleRobotMover node initialized. Waiting
            for joint commands...")

        self.end_effector_log_file = open("end_effector_reached_pos_log.csv",
             "w", newline='')
        self.end_effector_log_writer = csv.writer(self.end_effector_log_file)
```

```
34          self.end_effector_log_writer.writerow(["timestamp", "x", "y", "z"])
```

Listato 3.15: Inizializzazione della classe SimpleRobotMover

Nel Listato 3.15 è presente l'inizializzazione della classe SimpleRobotMover con il costruttore che si occupa di configurare tutte le connessioni ROS2 necessarie, inclusi client di servizio, sottoscrizioni e, soprattutto, l'Action Client per il controllo del robot. In particolare ci sono le seguenti operazioni:

- **Client FK**: Crea un client per il servizio di cinematica diretta. Il nodo attende che questo servizio sia disponibile prima di procedere.
- **Sottoscrizione allo Stato dei Giunti**: Crea un subscriber per il topic /joint_states, che aggiornerà self.joint_state.
- **Action Client**: Crea un ActionClient per l'azione FollowJointTrajectory. Questo è il meccanismo principale per inviare comandi di movimento al robot. Il nodo attende attivamente che l'Action Server specificato in ACTION_SERVER_NAME sia disponibile (self._action_client.wait_for_server()). FollowJointTrajectory è lo standard ROS2 per comandare i movimenti delle giunzioni di un robot lungo una traiettoria definita, utilizzando il modello di comunicazione Action per una gestione robusta e flessibile del controllo robotico.
- **Sottoscrizione ai Comandi dei Giunti**: Ccrea un subscriber per il topic /robot/-joint_commands. Questo topic è quello su cui il RobotController pubblica le posizioni dei giunti target che il robot deve raggiungere. Quando un messaggio arriva su questo topic, viene attivata la joint_command_callback.
- **Logging dell'End-Effector**: Viene aperto un file CSV (end_effector_reached_pos_log.csv) per registrare le posizioni 3D dell'end-effector effettivamente raggiunte dal robot. Vengono preparati il csv.writer e le intestazioni delle colonne.

```
1  def joint_state_callback(self, msg: JointState):
2          self.joint_state = msg
3
4  def joint_command_callback(self, msg: JointState):
5
6      joint_positions = list(msg.position)
7      self.get_logger().info(f"Received joint command: {joint_positions}")
8      self.send_trajectory(joint_positions, None)  t
9
10 def goal_response_callback(self, future):
11
12     goal_handle = future.result()
13     if not goal_handle.accepted:
14         self.get_logger().info('Goal rejected')
15         return
16     self.get_logger().info('Goal accepted')
17
18     self._result_future = goal_handle.get_result_async()
19     self._result_future.add_done_callback(self.result_callback)
20
21 def result_callback(self, future):
```

```
22      result = future.result().result
23      self.get_logger().info(f'Risultato del movimento: {result.error_code}')
24
25      final_pose = self.get_end_effector_pose()
26      if final_pose:
27
28          timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")
29          self.end_effector_log_writer.writerow([
30              timestamp,
31              final_pose.position.x,
32              final_pose.position.y,
33              final_pose.position.z
34          ])
35          self.end_effector_log_file.flush()
36          self.get_logger().info(f"Registrata posizione finale end-effector (
                XYZ): "
37                                 f"X={final_pose.position.x:.3f}, Y={final_pose.
                                       position.y:.3f}, Z={final_pose.position.z:.3f}
                                       ")
38      else:
39          self.get_logger().error("Impossibile registrare la posizione finale
                dell'end-effector.")
```

Listato 3.16: Funzioni di callback della classe SimpleRobotMover

Nel Listato 3.16 sono presenti le quattro funzioni di callback della classe SimpleRobotMover, esse sono essenziali per gestire gli eventi asincroni, come la ricezione di nuovi stati dei giunti, i comandi di movimento e le risposte dall'Action Server. In particolare:

- joint_state_callback(self, msg: JointState): Questa callback è estremamente semplice ma fondamentale. Viene richiamata ogni volta che il nodo riceve un messaggio JointState sul topic /joint_states. Il suo unico scopo è aggiornare la variabile d'istanza self.joint_state con i dati più recenti relativi alla posizione e velocità dei giunti del robot.

- joint_command_callback(self, msg: JointState): Questa funzione è la porta d'ingresso per i comandi di movimento. Viene attivata quando il SimpleRobotMover riceve un messaggio JointState dal RobotController (sul topic /robot/joint_commands). Estrae le posizioni dei giunti dal messaggio e le passa direttamente al metodo send_trajectory(), che si occuperà di inviare la richiesta di movimento all'Action Server del robot.

- goal_response_callback(self, future): Questa callback viene invocata dopo che il SimpleRobotMover ha inviato un Goal all'Action Server e ha ricevuto una risposta preliminare sull'accettazione o il rifiuto dell'obiettivo. Essa verifica se il Goal è stato accepted dal server. Se è stato rifiutato, logga un messaggio appropriato. Se il Goal è stato accettato, imposta un'ulteriore callback (self.result_callback) che verrà chiamata quando l'Action Server avrà completato l'esecuzione della traiettoria e fornirà il risultato finale.

- result_callback(self, future): Questa è la callback finale del ciclo di un'azione. Viene richiamata quando l'Action Server ha terminato di eseguire la traiettoria e ha restituito il risultato. Logga il codice di errore del risultato per indicare il successo o il fallimento

del movimento. Si occupa anche del logging della posa finale dell'end-effector, indipendentemente dall'esito, tenta di ottenere la posa finale dell'end-effector utilizzando il metodo get_end_effector_pose(). Se la posa è disponibile, registra le sue coordinate X, Y, Z in un file CSV con un timestamp, fornendo un registro delle posizioni effettivamente raggiunte dal robot. Questo è utile per l'analisi post-esecuzione e la verifica della precisione. È importante sottolineare che l'Action Server FollowJointTrajectory gestisce i comandi concorrenti tramite **preemption**, ovvero se un nuovo obiettivo di traiettoria interrompe sempre quello in corso, garantendo che il robot si adegui sempre al comando più recente. Questa nota è importante, poichè a causa dei tempi di elaborazione, non ci si troverà mai in una condizione ottimale deterministica e quindi spesso il movimento viene interrotto da una nuova richiesta di movimento.

```
def get_end_effector_pose(self):
        if self.joint_state is None or len(self.joint_state.position) != 6:
            self.get_logger().warn("Joint state non valido o non ancora
                ricevuto.")
            return None

        fk_req = GetPositionFK.Request()
        fk_req.header = Header()
        fk_req.header.frame_id = 'base_link'
        fk_req.fk_link_names = ['tool0']
        fk_req.robot_state.joint_state.name = self.joint_state.name
        fk_req.robot_state.joint_state.position = self.joint_state.position

        future = self.fk_cli.call_async(fk_req)
        rclpy.spin_until_future_complete(self, future)

        if not future.result() or len(future.result().pose_stamped) == 0:
            self.get_logger().error("Errore nel calcolo della FK.")
            return None

        return future.result().pose_stamped[0].pose

def send_trajectory(self, joint_positions, desired_pose):

        goal = FollowJointTrajectory.Goal()
        goal.trajectory = JointTrajectory()
        goal.trajectory.joint_names = JOINT_NAMES

        point = JointTrajectoryPoint()
        point.positions = joint_positions
        point.velocities = [0.0] * len(joint_positions)
        point.time_from_start = rclpyDuration(seconds=TIMER_DURATION_SECONDS)
            .to_msg()

        goal.trajectory.points.append(point)

        self.get_logger().info(f"Sending trajectory goal for joint positions:
            {joint_positions}")
        future = self._action_client.send_goal_async(goal)
        future.add_done_callback(self.goal_response_callback)
```

Listato 3.17: Funzioni di interazione con servizi e azioni

Il Listato 3.17 contiene i metodi della classe SimpleRobotMover che implementano le funzioni principali, ovvero il recupero della posa dell'end-effector e l'invio effettivo delle traiettorie al robot. Le due funzioni seguono questa logica:

- get_end_effector_pose(self): Questo metodo è responsabile del calcolo della posa (posizione e orientamento) corrente dell'end-effector del robot, identificato dal link tool0, rispetto al frame di riferimento base_link. Utilizza il servizio di cinematica diretta (/compute_fk) offerto da MoveIt!. Il metodo verifica innanzitutto la disponibilità e validità dello stato attuale dei giunti, poi costruisce e invia la richiesta FK al servizio. L'esecuzione del nodo viene bloccata in attesa della risposta, e il metodo restituisce la posa calcolata o None in caso di errore.

- send_trajectory(self, joint_positions, desired_pose): Questo è il metodo cruciale per comandare il movimento del robot. Prende in input le posizioni target dei giunti (joint_positions) e costruisce un Goal di tipo FollowJointTrajectory. Successivamente crea un messaggio JointTrajectory e vengono impostati i nomi dei giunti (JOINT_NAMES). Definisce un singolo JointTrajectoryPoint contenente le joint_positions desiderate, le velocità impostate a zero (per raggiungere la posizione e fermarsi), e un time_from_start pari a TIMER_DURATION_SECONDS. Questo indica al controllore del robot quanto tempo impiegare per raggiungere quel punto. Dopo che il Point viene aggiunto alla traiettoria, il goal viene inviato in modo asincrono all'Action Server. Una callback (self.goal_response_callback) viene registrata per gestire la risposta dell'Action Server sull'accettazione o il rifiuto dell'obiettivo, permettendo al nodo di procedere senza bloccarsi.

```
1  def main(args=None):
2
3      rclpy.init(args=args)
4      node = SimpleRobotMover()
5      rclpy.spin(node)
6      node.destroy_node()
7      rclpy.shutdown()
8
9  if __name__ == '__main__':
10     main()
```

Listato 3.18: Funzione main() dello script mover.py

Nel Listato 3.18 è presente la funzione main() che gestisce l'inizializzazione del nodo ROS2 e il suo ciclo di vita. Difatti, viene inizializzato il nodo, creata un'istanza di SimpleRobotMover che viene mantenuto attivo fino a quando non verrà interrotto (ad esempio, tramite Ctrl+C). Quando il ciclo viene interrotto, viene chiamato il metodo node.destroy_node() che assicura che il nodo rilasci correttamente le sue risorse, inclusa la chiusura dei file CSV di logging, come implementato nel metodo destroy_node() della classe SimpleRobotMover.

3.4 Analisi script di visualizzazione degli errori: plot_error.py

Lo script plot_error.py è uno strumento post-elaborazione che analizza le performance del sistema di controllo robotico. Il suo obiettivo è leggere i dati di logging (posizioni target,

posizioni calcolate come "next_pos", e posizioni effettivamente raggiunte dall'end-effector) dai file CSV generati dagli script trajectory_sender.py e mover.py. Successivamente, calcola diversi tipi di errori di tracciamento e li visualizza tramite grafici, fornendo un'analisi quantitativa della precisione e della reattività del robot.

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

import matplotlib.pyplot as plt
import numpy as np
import csv
from datetime import datetime
import sys

def read_csv_data(filepath):
    raw_timestamps = []
    xs = []
    ys = []
    zs = []
    try:
        with open(filepath, 'r') as file:
            reader = csv.reader(file)
            header = next(reader)

            for row in reader:

                raw_timestamps.append(datetime.strptime(row[0], "%Y-%m-%d %H
                    :%M:%S.%f"))
                xs.append(float(row[1]))
                ys.append(float(row[2]))
                zs.append(float(row[3]))

    except FileNotFoundError:
        print(f"Errore: Il file '{filepath}' non e' stato trovato.")
        return np.array([]), np.array([]), np.array([]), np.array([])
    except Exception as e:
        print(f"Errore durante la lettura del file '{filepath}': {e}")
        return np.array([]), np.array([]), np.array([]), np.array([])

    if not raw_timestamps:
        print(f"Avviso: Il file '{filepath}' e' vuoto o non contiene dati
            validi.")
        return np.array([]), np.array([]), np.array([]), np.array([])

    start_time = raw_timestamps[0]
    times_sec = np.array([(t - start_time).total_seconds() for t in
        raw_timestamps])

    return times_sec, np.array(xs), np.array(ys), np.array(zs)

def plot_robot_errors_no_pandas(target_pos_file, next_pos_file,
    reached_pos_file):

    print(f"Caricamento dati da '{target_pos_file}'...")
    time_target, target_xs, target_ys, target_zs = read_csv_data(
        target_pos_file)
    print(f"Caricamento dati da '{next_pos_file}'...")
```

```
47      time_next, next_xs, next_ys, next_zs = read_csv_data(next_pos_file)
48      print(f"Caricamento dati da '{reached_pos_file}'...")
49      time_reached, reached_xs, reached_ys, reached_zs = read_csv_data(
            reached_pos_file)
50
51      if len(time_target) == 0 or len(time_next) == 0 or len(time_reached) ==
            0:
52          print("Uno o piu' file CSV sono vuoti o non leggibili. Impossibile
                generare i grafici.")
53          return
54
55      min_len = min(len(time_target), len(time_next), len(time_reached))
56      if min_len == 0:
57          print("Nessun dato comune tra i file CSV dopo il caricamento.
                Impossibile generare i grafici.")
58          return
59
60      time_target = time_target[:min_len]
61      target_xs, target_ys, target_zs = target_xs[:min_len], target_ys[:min_len
            ], target_zs[:min_len]
62
63      time_next = time_next[:min_len]
64      next_xs, next_ys, next_zs = next_xs[:min_len], next_ys[:min_len], next_zs
            [:min_len]
65
66      time_reached = time_reached[:min_len]
67      reached_xs, reached_ys, reached_zs = reached_xs[:min_len], reached_ys[:
            min_len], reached_zs[:min_len]
68
69      error_x_target_reached = next_xs - reached_xs
70      error_y_target_reached = next_ys - reached_ys
71      error_z_target_reached = next_zs - reached_zs
72
73      total_error_target_reached = np.sqrt(
74          error_x_target_reached**2 +
75          error_y_target_reached**2 +
76          error_z_target_reached**2
77      )
78
79      error_x_next_target = next_xs - target_xs
80      error_y_next_target = next_ys - target_ys
81      error_z_next_target = next_zs - target_zs
82
83      plt.figure(figsize=(12, 7))
84      plt.plot(time_target, error_x_target_reached, label='Errore X', color='
            red', linewidth=1.5)
85      plt.plot(time_target, error_y_target_reached, label='Errore Y', color='
            green', linewidth=1.5)
86      plt.plot(time_target, error_z_target_reached, label='Errore Z', color='
            blue', linewidth=1.5)
87      plt.xlabel('Tempo (s)', fontsize=12)
88      plt.ylabel('Errore (m)', fontsize=12)
89      plt.title('Errore Individuale: (Posizione calcolata - Posizione Raggiunta
            ) per Asse', fontsize=14)
90      plt.legend(fontsize=10)
91      plt.grid(True, linestyle='--', alpha=0.7)
```

```
92      plt.tight_layout()
93      plt.show()
94
95
96      plt.figure(figsize=(12, 7))
97      plt.plot(time_target, total_error_target_reached, label='Errore Totale',
            color='red', linewidth=2)
98      plt.xlabel('Tempo (s)', fontsize=12)
99      plt.ylabel('Distanza Euclidea (m)', fontsize=12)
100     plt.title('Errore Totale: Distanza Euclidea (Posizione calcolata -
            Posizione Raggiunta)', fontsize=14)
101     plt.legend(fontsize=10)
102     plt.grid(True, linestyle='--', alpha=0.7)
103     plt.tight_layout()
104     plt.show()
105
106
107     plt.figure(figsize=(12, 7))
108     plt.plot(time_target, error_x_next_target, label='Errore X', color='red',
             linewidth=1.5)
109     plt.plot(time_target, error_y_next_target, label='Errore Y', color='green
            ', linewidth=1.5)
110     plt.plot(time_target, error_z_next_target, label='Errore Z', color='blue
            ', linewidth=1.5)
111     plt.xlabel('Tempo (s)', fontsize=12)
112     plt.ylabel('Errore (m)', fontsize=12)
113     plt.title('Errore di Spostamento: (Next Pos - Target Pos) per Asse',
            fontsize=14)
114     plt.legend(fontsize=10)
115     plt.grid(True, linestyle='--', alpha=0.7)
116     plt.tight_layout()
117     plt.show()
118
119 def main(args=None):
120
121     target_csv = "target_pos_log.csv"
122     next_csv = "next_pos_log.csv"
123     reached_csv = "end_effector_reached_pos_log.csv"
124
125     plot_robot_errors_no_pandas(target_csv, next_csv, reached_csv)
126
127 if __name__ == '__main__':
128     main()
```

Listato 3.19: Script plot_error.py

Dopo il caricamento e la sincronizzazione dei dati, vengono calcolati tre tipi di errore:

1. **Errore di tracking globale (Next Pos - Reached Pos)**: Viene calcolata la differenza tra la next_pos (la posizione calcolata dal RobotController come prossimo passo) e la reached_pos (la posizione effettivamente raggiunta dal robot), indicando la precisione con cui il controllore di basso livello del robot riesce a raggiungere la posizione comandata, l'errore viene espresso come un errore di distanza euclidea.

2. **Errore di tracking per assi (Next Pos - Reached Pos)**: Come per il precedente, tuttavia in questo grafico vengono plottati gli errori individuali per ogni asse X, Y e Z.

3. **Errore di spostamento (Next Pos - Target Pos)**: Viene calcolata la differenza tra la next_pos (la posizione intermedia calcolata) e la target_pos (la posizione finale ideale del punto rosso). Questo errore misura quanto la logica di 'step' del RobotController stia seguendo da vicino il target finale, o quanto dista il prossimo passo calcolato dal punto finale desiderato.

4 Test e risultati

Questo capitolo descrive in dettaglio il setup iniziale per la le modalità con cui sono stati condotti i test, i parametri osservati, i criteri di valutazione adottati e i risultati ottenuti. Particolare attenzione è stata posta alla precisione del movimento, alla robustezza del sistema nel gestire condizioni non ideali (come il fallimento della cinematica inversa), alla sicurezza durante l'interazione uomo-robot e alla reattività del sistema.

4.1 Setup

4.1.1 Ambiente di test

L'esecuzione dell'intero progetto è avvenuta all'interno del laboratorio, in ambiente controllato e privo di ostacoli, pensato per garantire condizioni ripetibili e sicure durante tutte le fasi di test.
Il robot UR5 è montato su una piattaforma mobile MIR200, utilizzata unicamente come supporto statico: il sistema di locomozione del MiR non è stato impiegato durante l'esecuzione del progetto. Questa configurazione ha permesso di avere una base solida ma rialzata per il braccio robotico, facilitando l'interazione con la scena posta a livello banco.
L'end effector del robot, come precisato precedentemente, è equipaggiato con una videocamera Intel RealSense D435i, montata rigidamente in posizione retrostante rispetto al TCP, con configurazione *eye-in-hand*. La distanza tra il centro ottico della camera e l'end effector era di circa 6 cm in direzione longitudinale e 3 cm in altezza rispetto al TCP, come visibile in Figura 4.1. Questa vicinanza consente una visione diretta dell'ambiente circostante al punto di lavoro, con una prospettiva dinamica legata al movimento del braccio.

Figure 4.1: *Illustrazione del posizionamento della camera rispetto all'end-effector*

Nel campo visivo della camera veniva posizionato un marcatore rosso (target), montato all'estremità di un telo, manipolato da uno dei partecipanti del progetto. In particolare, i test dinamici venivano svolti coinvolgendo attivamente tre persone:

- Un operatore si occupava di gestire il movimento del target, muovendolo nello spazio per simulare scenari realistici e valutare la reattività del sistema robotico.
- Un secondo operatore teneva in mano il controller (pad) collegato al robot, pronto ad azionare in qualsiasi momento il pulsante di arresto di emergenza (fungo di stop) per garantire la sicurezza durante le fasi di test.
- Un terzo operatore gestiva l'interfaccia software da PC, eseguendo gli script, monitorando l'output dei nodi ROS2 e salvando i dati sperimentali.

Il target veniva spostato in varie posizioni e direzioni all'interno del campo visivo della RealSense, per mettere alla prova la pipeline di rilevamento, la trasformazione delle coordinate e il comportamento del robot in tempo reale. Questo setup ha permesso di validare il sistema in condizioni semi-realistiche, mantenendo allo stesso tempo un elevato livello di controllo e sicurezza.
L'intero banco di lavoro forniva uno spazio operativo di circa 2 metri per lato, sufficiente a consentire movimenti completi del braccio UR5 senza collisioni e a garantire un'adeguata copertura visiva per la camera.

4.1.2 Configurazione del sistema

Prima di avviare i test, è stata effettuata una configurazione completa del sistema, sia a livello hardware che software, per assicurare la corretta esecuzione di tutti i nodi ROS2 e la sincronizzazione tra i componenti.
Per il check hardware si visionano:

- **Braccio robotico:** UR5 della Universal Robots, montato su base mobile MiR200.
- **Camera:** Intel RealSense D435i, connessa via USB 3.0 al PC e montata rigidamente a 3 cm dall'end effector.
- **PC di controllo:** Laptop con CPU Intel i7, 16 GB di RAM, sistema operativo Ubuntu 22.04 LTS.
- **Controller UR5:** Collegamento via Ethernet, utilizzando il driver ufficiale UR ROS2.

Dal punto di vista software, è stata utilizzata la distribuzione ROS2 Humble, insieme a diversi pacchetti fondamentali:

- ***ur_robot_driver*** − per la connessione diretta al robot;
- ***ur_moveit_config*** −per l'utilizzo di MoveIt e l'accesso ai servizi di cinematica diretta e inversa;
- ***realsense2_camera*** − per la gestione dei flussi video RGB e depth, oltre alla pointcloud;

Prima del lancio dei nodi ROS2, la procedura di setup inizia con la verifica della connettività tra PC e controller del robot tramite un semplice ping all'indirizzo IP assegnato al braccio:

```
ping 192.168.1.102
```

dove *192.168.1.102* è l'indirizzo IP assegnato al controller del robot UR5.
Sono stati inoltre impiegati pacchetti ROS2 custom sviluppati per il rilevamento del target, la trasformazione delle coordinate tra frame e il controllo dei giunti del robot. Una volta accertata la connessione, viene lanciato il driver del robot con il seguente comando:

```
ros2 launch ur_robot_driver ur_control.launch.py ur_type:=ur5
robot_ip:=192.168.1.102 launch_rviz:=true
```

Questo comando avvia la comunicazione con l'UR5 e apre RViz per la visualizzazione del modello del robot. Successivamente, viene lanciato MoveIt per abilitare i servizi di cinematica diretta e inversa con hardware reale:

```
ros2 launch ur_moveit_config ur_moveit.launch.py ur_type:=ur5
use_fake_hardware:=false
```

Infine, per la gestione della videocamera RealSense, si utilizza un comando personalizzato che configura la risoluzione, la sincronizzazione tra flussi, l'allineamento tra RGB e depth e l'attivazione della pointcloud:

```
ros2 launch realsense2_camera rs_launch.py \
depth_module.profile:=640x480x30 \
rgb_camera.profile:=640x480x30 \
pointcloud.enable:=true \
unite_imu_method:=linear_interpolation \
enable_sync:=true \
align_depth.enable:=true \
device_type:=d435i
```

4.2 Obiettivi dei test

I test sono stati progettati per validare il comportamento del sistema sviluppato in condizioni operative realistiche, valutando sia le performance funzionali che gli aspetti legati alla sicurezza e alla robustezza. Gli obiettivi principali possono essere sintetizzati come segue:

- **Verifica della precisione nella localizzazione del target:** Il sistema deve essere in grado di stimare con accuratezza la posizione tridimensionale del punto target rilevato dalla videocamera, assicurando una corretta trasformazione delle coordinate nel frame del robot.
- **Validazione della generazione della pose target:** Il calcolo della posa dell'end effector, a una distanza prestabilita dal punto target e con orientamento vincolato, deve produrre risultati coerenti e compatibili con l'area di lavoro del robot.
- **Misura della reattività e dei tempi di risposta:** È necessario valutare il tempo che intercorre tra la rilevazione del target e l'esecuzione del movimento da parte del robot, per verificare che il sistema risponda in modo fluido e continuo ai cambiamenti nella scena.

- **Analisi della precisione del movimento del robot:** Viene misurato lo scostamento tra la posizione desiderata (target) e la posizione effettivamente raggiunta dall'end effector, in termini di errore euclideo medio e massimo.
- **Robustezza della cinematica inversa:** Il sistema deve essere in grado di gestire eventuali fallimenti nella risoluzione della cinematica inversa (ad esempio, in prossimità di singolarità o limiti articolari), evitando comportamenti instabili o movimenti non pianificati.
- **Controllo del rispetto dei vincoli di sicurezza:** Si verifica che vengano rispettati i vincoli imposti in fase di progettazione, come lo spostamento massimo consentito tra due pose successive, in modo da garantire un'interazione sicura tra robot e operatore.

4.3 Osservazioni sui test effettuati

4.3.1 Verifica della precisione nella localizzazione del target

Uno degli aspetti fondamentali del sistema è la corretta localizzazione spaziale del punto target rilevato dalla videocamera RealSense montata sul robot. La precisione di questo processo influisce direttamente sulla qualità del movimento del braccio robotico, sulla riuscita del task di co-trasporto e sulla sicurezza complessiva dell'interazione.
Il punto target è rappresentato da un **marker rosso di forma quadrata**, applicato su un telo nero, scelto per la sua elevata visibilità e facilità di segmentazione nell'immagine RGB. Il marker ha dimensioni di **circa 5×5 cm**, un compromesso ideale tra visibilità anche a distanze superiori a 1 metro e sufficiente risoluzione per consentire un'accurata individuazione del centroide attraverso tecniche di elaborazione delle immagini.
Il processo di localizzazione si articola in più fasi:

- **Rilevamento del marker rosso nell'immagine RGB**, tramite sogliatura nel dominio HSV per isolare le componenti cromatiche corrispondenti al rosso.
- **Calcolo della posizione 3D del punto**, ricavata proiettando il centroide dell'area rilevata nella mappa di profondità della RealSense, usando i parametri intrinseci della camera per stimare la coordinata tridimensionale rispetto al suo frame.
- **Trasformazione della posizione nel frame base del robot**, ottenuta concatenando la trasformazione fissa tra la camera e *tool0* con la trasformazione *tool0* → *base_link*, calcolata mediante cinematica diretta.

Un elemento cruciale per garantire l'**accuratezza della trasformazione finale** è la **precisa misura della posizione e dell'orientazione della camera rispetto al frame *tool0***. Anche piccoli errori nelle distanze in altezza o lunghezza, o nella rotazione della camera, possono portare a significativi scostamenti nella posizione del punto stimato nel frame base, rendendo inattendibile l'intera catena cinematica. Questo comprometterebbe direttamente l'efficacia della cinematica inversa, generando soluzioni errate o portando a situazioni di *IK fallita*. Il test ha quindi l'obiettivo di valutare:

- la ripetibilità della stima del punto 3D,

- l'accuratezza rispetto a riferimenti noti,
- l'influenza di errori di calibrazione tra camera e *tool0*,
- l'effetto di condizioni ambientali variabili (luce, distanza, angolo di osservazione) sulla stabilità della rilevazione.

4.3.2 Validazione della generazione della pose target

La generazione della pose target dell'end effector rappresenta una fase critica nel processo di controllo del robot, poiché determina il punto nello spazio verso cui il braccio dovrà muoversi mantenendo una configurazione coerente con le capacità cinematiche del sistema.
Nel contesto di questo progetto, la posa viene calcolata a partire dalla posizione del marker rosso rilevato nella scena, con l'obiettivo di mantenere una distanza fissa (1 metro) tra l'end effector e il target. L'orientamento dell'end effector viene vincolato affinché:

- Il piano X-Y risulti parallelo al piano del suolo,
- L'asse X venga mantenuto orientato nella direzione del punto rosso, simulando una "direzione di spinta",
- Venga rispettato un vincolo sull'asse Z, mantenendo l'altezza dell'end-effector entro limiti operativi.

Durante la fase di validazione, sono stati osservati diversi fattori critici:

- **Pose incompatibili con l'area di lavoro:** in alcuni casi, la posizione target ricadeva troppo lontano o in configurazioni dove il braccio entra in singolarità (tipicamente quando il polso o l'asse *shoulder_pan_joint* si allineano). In questi casi il servizio */compute_ik* restituiva *ik_failed*, segnalando l'impossibilità di raggiungere la configurazione desiderata.
- **Orientamenti non realizzabili:** quando la rotazione richiesta all'end effector comportava un eccessivo twist attorno all'asse del polso (violando il range di *wrist_3_joint*), l'IK restituiva soluzioni inadatte o falliva.
- **Sensibilità all'accuratezza della posizione:** anche piccoli errori nel calcolo della trasformazione del punto target influenzavano il calcolo della direzione dell'asse Z dell'end effector, portando a rotazioni involontarie.

4.3.3 Misura della reattività e dei tempi di risposta

La reattività rappresenta un parametro fondamentale nella valutazione di un sistema robotico interattivo, in particolare in scenari dinamici di co-trasporto, dove la posizione del target può variare rapidamente a causa del movimento umano. L'obiettivo è garantire che il robot reagisca in modo fluido ai cambiamenti, mantenendo un comportamento coerente e privo di ritardi percepibili.
Il tempo totale di risposta può essere scomposto in diverse fasi distinte:

- **Acquisizione dell'immagine:** La RealSense D435i fornisce immagini RGB e di profondità fino a 90 fps, ma per questo task sono sufficienti 30 fps per evitare sovraccarichi. Il tempo di acquisizione è dunque inferiore a 33 ms.
- **Stima della posizione 3D del target:** La ricostruzione 3D dalla mappa di profondità e la trasformazione nel frame *base_link* (mediante cinematica diretta) introduce un tempo medio di 10–15 ms.
- **Generazione della pose target:** Include il calcolo della direzione dell'asse Z, la normalizzazione dei vettori e la creazione della quaternion. Richiede in media meno di 5 ms.
- **Risoluzione della cinematica inversa:** La chiamata al servizio */compute_ik* ha un tempo variabile: nei casi di configurazioni regolari, la risposta arriva entro 10–20 ms; nei casi complessi o prossimi a singolarità può superare i 50 ms o fallire.
- **Comando al controller del robot:** Il messaggio viene pubblicato su */joint_trajectory_controller/joint_trajectory*, e il tempo tra la pubblicazione e l'inizio del movimento è determinato dalla frequenza del controller e dai limiti di accelerazione impostati.

Durante i test, il tempo complessivo che intercorre tra la rilevazione del target e l'inizio del movimento del robot è stato mediamente inferiore a 150–200 ms, rientrando quindi nei limiti accettabili per un'interazione in tempo quasi reale. Tuttavia, questo tempo può aumentare in presenza di:

- Picchi di carico CPU nella pipeline ROS2,
- Fallimenti o timeout del servizio di IK,
- Variazioni rapide del target che richiedono frequenti aggiornamenti della pose.

Il sistema è stato progettato per generare nuove pose ogni 0,4 secondi, intervallo scelto come compromesso tra reattività e stabilità, per evitare che micro-variazioni del target portino a movimenti caotici o instabili.

4.3.4 Analisi della precisione del movimento del robot

Per valutare l'effettiva precisione del sistema, è stato eseguito un confronto sistematico tra la posa target calcolata tramite cinematica inversa e la posa effettivamente raggiunta dall'end effector, ricavata tramite cinematica diretta. Questo confronto è fondamentale per verificare la bontà dell'intera pipeline, inclusi i passaggi di trasformazione, risoluzione IK e controllo del movimento.
Il processo di valutazione si articola nei seguenti step:

- **Calcolo della posa target:** A partire dalla posizione 3D del quadrato rosso, si genera una posa target mantenendo la distanza fissa (1 m) e un orientamento predefinito dell'end effector.
- **Risoluzione della cinematica inversa:** La posa viene passata al servizio */compute_ik*, che restituisce una configurazione articolare compatibile.

- **Comando di esecuzione:** I joint target vengono inviati al *scaled_joint_trajectory_controller*, che li esegue nel tempo minimo compatibile con i limiti imposti.
- **Misura dell'errore:** Una volta completato il movimento, viene recuperata la posa effettiva tramite */compute_fk*, e si calcola l'errore euclideo come distanza tra i centri delle due pose:

$$\epsilon = \| Ptarget - Peffettiva \|$$

Tra le principali cause di errore si identificano:

- **Risoluzione della mappa di profondità:** La precisione della RealSense D435i degrada oltre i 2 metri e in presenza di superfici riflettenti, rischiando di compromettere il riconoscimento univoco del punto rosso.
- **Errori nella trasformazione camera → *base_link*:** Eventuali imprecisioni nelle misure di montaggio o nella trasformazione statica possono introdurre offset sistematici.
- **Limitazioni dell'IK solver:** Il solver di MoveIt! può convergere a soluzioni localmente valide ma non ottimali, soprattutto vicino a zone di singolarità.
- **Inerzia del sistema di controllo:** Anche se il controller esegue la traiettoria, è possibile che l'end effector si arresti leggermente prima o dopo il punto previsto, specialmente in presenza di cambi rapidi di traiettoria, tale errore si può osservare nei grafici nelle Figure 4.4 e 4.5 .

4.3.5 Robustezza della cinematica inversa

L'obiettivo di questa parte della sperimentazione è valutare la robustezza del modulo di cinematica inversa (IK – Inverse Kinematics), ovvero la sua capacità di fornire soluzioni valide per pose target generate in diversi scenari, inclusi quelli vicini ai limiti del workspace, in configurazioni ridondanti o potenzialmente singolari. L'affidabilità della cinematica inversa è infatti cruciale per garantire il comportamento coerente e prevedibile del robot in fase operativa.
Per effettuare una valutazione esaustiva della robustezza del calcolo IK, sono stati progettati diversi casi di test, suddivisi in tre classi:

1. **Pose target interne al workspace**, ben centrate, prive di vincoli critici.
2. **Pose target prossime ai limiti del workspace** o ai limiti articolari.
3. **Pose target con orientamenti sfavorevoli o ridondanze**, dove esistono molte soluzioni possibili ma è richiesto il rispetto di vincoli aggiuntivi (es. vincolo sull'asse Z o *wrist_3_joint* fisso).

Per ogni test, è stata generata una pose target e inviata al modulo di IK (*IKClient*). Il sistema ha poi analizzato l'esito della chiamata:

- Soluzione trovata e valida
- Nessuna soluzione trovata (errore IK)

- Soluzione violava vincoli (errore IK)

Un modulo IK robusto deve:

- Restituire soluzioni affidabili per la maggior parte delle pose plausibili nel workspace operativo del robot.
- Riconoscere correttamente le configurazioni insostenibili (singolarità, target troppo lontani o con orientamento impossibile).
- Gestire correttamente vincoli articolari e orientamenti imposti (es. mantenere il polso fermo o l'orientamento dell'end-effector).
- Fornire un meccanismo di fallback, evitando l'invio di comandi invalidi al controller del robot.

Durante i test, sono emersi alcuni casi critici, in particolare:

- In prossimità dell'asse verticale del robot, la soluzione IK risulta più ambigua (infinite rotazioni sul polso).
- Pose molto alte o molto basse rispetto al piano base portano più facilmente a fallimenti per violazione di *shoulder_lift_joint*.
- Quando viene imposto un vincolo sull'orientamento (es. asse Z dell'end-effector parallelo al suolo), il solver fatica a trovare una soluzione valida nei pressi di configurazioni singolari.

Questi risultati suggeriscono che, sebbene il modulo IK sia generalmente affidabile, è opportuno anticipare i limiti del solver e prevedere meccanismi di gestione dell'errore.

4.3.6 Controllo del rispetto dei vincoli di sicurezza

In questa sezione viene analizzato in dettaglio il meccanismo implementato per il controllo dei vincoli di sicurezza nel sistema robotico, con particolare attenzione ai limiti articolari, al mantenimento di distanze minime dal target, alla gestione dei fallimenti della cinematica inversa e al controllo dinamico della velocità di movimento.
Tra gli obiettivi imposti si ha:

- Impedire il superamento dei limiti articolari del robot.
- Mantenere una distanza costante di sicurezza tra end-effector e target (pari a 1 metro).
- Bloccare l'esecuzione in caso di soluzione IK assente o pericolosa.
- Regolare la velocità del movimento per evitare spostamenti eccessivamente rapidi.

Tra le strategie adottate si ha:

- **Controllo dei limiti articolari:** Ogni soluzione di cinematica inversa ricevuta viene verificata rispetto ai limiti meccanici di ciascun giunto. In caso di superamento di uno o più limiti, la posa non viene inviata al controller, e l'evento viene loggato.

- **Gestione dei fallimenti della cinematica inversa:** In caso di errore o assenza di soluzione da parte del solver IK, il sistema non invia comandi al robot, evitando movimenti pericolosi o indefiniti. È previsto un fallback che mantiene l'ultima configurazione valida.

- **Controllo dinamico della velocità tramite waypoint intermedi:** Quando la distanza tra la posizione corrente dell'end-effector e la nuova posa target supera una soglia predefinita, il sistema evita un movimento diretto che implicherebbe un'elevata velocità. In tali casi viene generata una pose intermedia lungo la direzione verso il target, situata a una frazione del percorso (es. 50%). Il robot si muove prima verso questa posizione intermedia e solo successivamente verso quella finale.
  Questo meccanismo riduce i picchi di velocità e migliora la stabilità e la prevedibilità del movimento.

Si può quindi osservare che:

- Il sistema mostra una solida capacità di prevenzione dei rischi, soprattutto grazie ai controlli a più livelli prima dell'invio del comando al robot.

- Il meccanismo di pose intermedia contribuisce a smussare transizioni improvvise e a ridurre picchi di velocità, migliorando la fluidità e la sicurezza generale del comportamento.

- Il controllo integrato tra validazione IK, vincoli spaziali e velocità consente un comportamento reattivo ma stabile, anche in presenza di variazioni rapide nella scena o nella posizione del target.

Nelle Figure 4.2 e 4.3 si possono notare gli output in termini di log utili e coordinate inviate al robot, ma anche una visualizzazione in tempo reale di ciò che vede la telecamera Realsense montata nell'UR5. In quest'ultima immagine si osserva il punto effettivamente rilevato e sui cui si basano i calcoli (verde) e un punto di target che indica la traiettoria che il robot dovrebbe seguire (blu): la situazione ideale prevede che l'errore di distanza tra il punto verde e il punto blu sia minimo, in rispetto della logica di sicurezza adottata. Dunque se nei log si vede che si sta mandando una Posizione Intermedia al robot, il punto blu sarà verosimilmente lontano da quello verde; al contrario, se si invia al robot la Posizione Diretta, la distanza tra i due punti sarà minima.

```
[INFO] [1749552051.022954476] [robot_controller]: DISTANZA DA FARE: 0.0200 m
[INFO] [1749552051.023343402] [robot_controller]: [DEBUG] tool_pos: x=-0.610, y=-0.070, z=0.450
[INFO] [1749552051.023675726] [robot_controller]: [STEP] POSIZIONE DIRETTA.
[INFO] [1749552051.024200067] [robot_controller]: [CHECK] TARGET POSE: [-0.61 -0.05  0.45] m
[INFO] [1749552051.024639125] [robot_controller]: [CHECK] NEXT POS: [-0.61 -0.05  0.45] m
[INFO] [1749552051.025000770] [robot_controller]: [CHECK] Distanza dal punto rosso: 1.0000 m
[INFO] [1749552051.026268874] [ik_client]: Soluzione IK trovata.
[INFO] [1749552051.422629746] [robot_controller]: DISTANZA DA FARE: 0.0597 m
[INFO] [1749552051.422823854] [robot_controller]: [DEBUG] tool_pos: x=-0.591, y=-0.051, z=0.450
[INFO] [1749552051.422982487] [robot_controller]: [STEP] POSIZIONE INTERMEDIA.
[INFO] [1749552051.423310226] [robot_controller]: [CHECK] TARGET POSE: [-0.65 -0.06  0.45] m
[INFO] [1749552051.423540448] [robot_controller]: [CHECK] NEXT POS: [-0.63 -0.06  0.45] m
[INFO] [1749552051.423706874] [robot_controller]: [CHECK] Distanza dal punto rosso: 1.0200 m
[INFO] [1749552051.424462017] [ik_client]: Soluzione IK trovata.
[INFO] [1749552051.823082499] [robot_controller]: DISTANZA DA FARE: 0.0328 m
[INFO] [1749552051.823285244] [robot_controller]: [DEBUG] tool_pos: x=-0.609, y=-0.050, z=0.450
[INFO] [1749552051.823436060] [robot_controller]: [STEP] POSIZIONE DIRETTA.
[INFO] [1749552051.823746014] [robot_controller]: [CHECK] TARGET POSE: [-0.64 -0.06  0.45] m
[INFO] [1749552051.823969350] [robot_controller]: [CHECK] NEXT POS: [-0.64 -0.06  0.45] m
[INFO] [1749552051.824142459] [robot_controller]: [CHECK] Distanza dal punto rosso: 1.0000 m
[INFO] [1749552051.825015353] [ik_client]: Soluzione IK trovata.
[INFO] [1749552052.223896147] [robot_controller]: DISTANZA DA FARE: 0.0108 m
[INFO] [1749552052.224172660] [robot_controller]: [DEBUG] tool_pos: x=-0.629, y=-0.060, z=0.450
[INFO] [1749552052.224336178] [robot_controller]: [STEP] POSIZIONE DIRETTA.
[INFO] [1749552052.224681937] [robot_controller]: [CHECK] TARGET POSE: [-0.64 -0.06  0.45] m
[INFO] [1749552052.224906712] [robot_controller]: [CHECK] NEXT POS: [-0.64 -0.06  0.45] m
[INFO] [1749552052.225134354] [robot_controller]: [CHECK] Distanza dal punto rosso: 1.0000 m
[INFO] [1749552052.225976614] [ik_client]: Soluzione IK trovata.
[INFO] [1749552052.621584585] [robot_controller]: DISTANZA DA FARE: 0.0416 m
[INFO] [1749552052.622237199] [robot_controller]: [DEBUG] tool_pos: x=-0.640, y=-0.060, z=0.450
[INFO] [1749552052.622605986] [robot_controller]: [STEP] POSIZIONE INTERMEDIA.
[INFO] [1749552052.623152438] [robot_controller]: [CHECK] TARGET POSE: [-0.68 -0.06  0.46] m
[INFO] [1749552052.623612842] [robot_controller]: [CHECK] NEXT POS: [-0.68 -0.06  0.46] m
[INFO] [1749552052.623964814] [robot_controller]: [CHECK] Distanza dal punto rosso: 1.0000 m
[INFO] [1749552052.625252989] [ik_client]: Soluzione IK trovata.
[INFO] [1749552053.024956276] [robot_controller]: DISTANZA DA FARE: 0.0754 m
[INFO] [1749552053.025444031] [robot_controller]: [DEBUG] tool_pos: x=-0.640, y=-0.060, z=0.450
[INFO] [1749552053.025899190] [robot_controller]: [STEP] POSIZIONE INTERMEDIA.
[INFO] [1749552053.026926967] [robot_controller]: [CHECK] TARGET POSE: [-0.71 -0.08  0.47] m
[INFO] [1749552053.027926748] [robot_controller]: [CHECK] NEXT POS: [-0.68 -0.07  0.46] m
[INFO] [1749552053.028681886] [robot_controller]: [CHECK] Distanza dal punto rosso: 1.0301 m
[INFO] [1749552053.030764963] [ik_client]: Soluzione IK trovata.
[INFO] [1749552053.424896364] [robot_controller]: DISTANZA DA FARE: 0.0273 m
[INFO] [1749552053.425282585] [robot_controller]: [DEBUG] tool_pos: x=-0.679, y=-0.060, z=0.460
[INFO] [1749552053.425662944] [robot_controller]: [STEP] POSIZIONE DIRETTA.
[INFO] [1749552053.426269257] [robot_controller]: [CHECK] TARGET POSE: [-0.66 -0.08  0.46] m
[INFO] [1749552053.426740900] [robot_controller]: [CHECK] NEXT POS: [-0.66 -0.08  0.46] m
[INFO] [1749552053.427157944] [robot_controller]: [CHECK] Distanza dal punto rosso: 1.0000 m
[INFO] [1749552053.429149153] [ik_client]: Soluzione IK trovata.
```

Figure 4.2: *Output posizione intermedia e diretta*

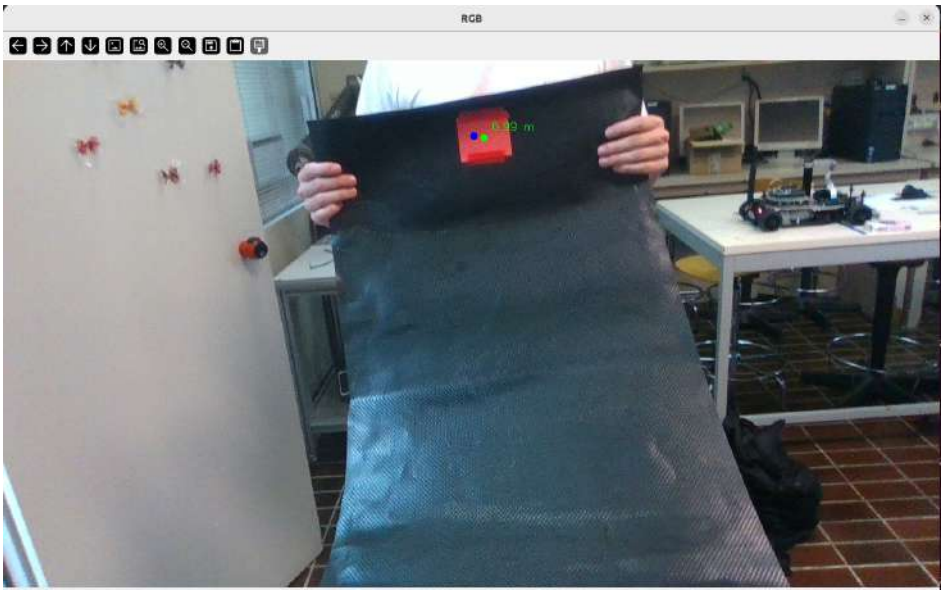

Figure 4.3: *Visualizzazione della camera in tempo reale*

4.4 Risultati

4.4.1 Accuratezza della posizione raggiunta

L'accuratezza della posizione raggiunta dal robot è stata valutata confrontando, per ogni nuovo target rilevato, la pose desiderata (calcolata a partire dalla posizione del punto rosso mantenendo una distanza di 1 metro) con la pose effettivamente raggiunta dall'end effector, stimata tramite cinematica diretta.
L'errore è stato calcolato come distanza euclidea tra le due posizioni, e monitorato durante l'intera durata del test. In particolare, il sistema è stato configurato per operare con un **tempo di campionamento di 0,4 secondi** e una **velocità massima** dell'end effector **di 0,1 m/s**, valori che garantiscono un comportamento fluido e controllato, evitando accelerazioni brusche.
Di seguito viene riportata l'evoluzione nel tempo dell'errore *Next Pos - Reached Pos* mostrato su ogni asse:

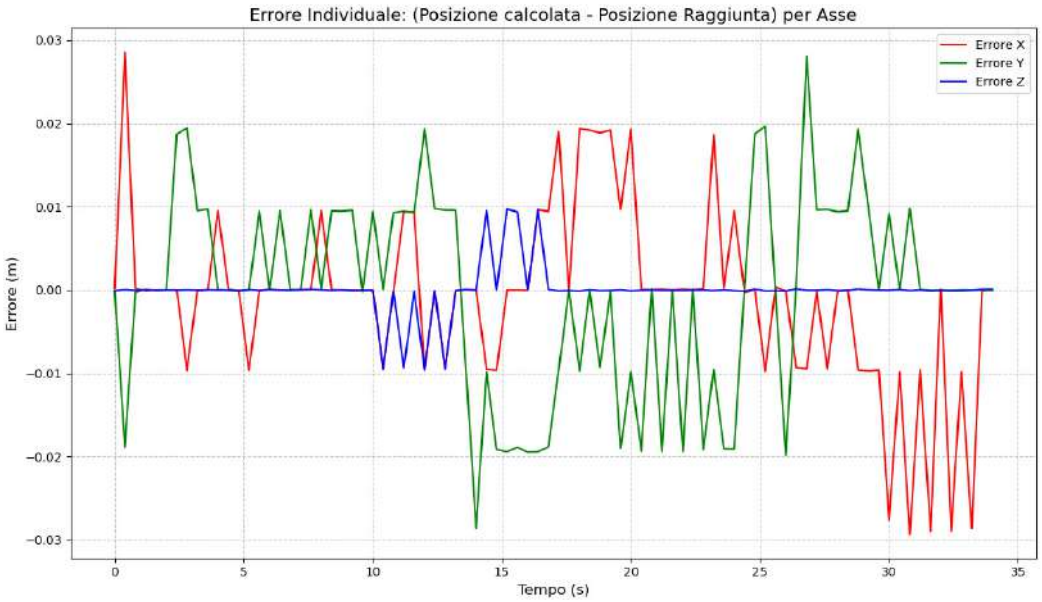

Figure 4.4: *Grafico errore individuale per asse*

Nella Figura 4.4 vengono evidenziati gli errori presenti per ogni singolo asse, evidenziando che l'errore minore si ha nell'asse z (nelle altezze), mentre si hanno errori di fedeltà del robot pù marcati, ma comunque minimi, negli altri due assi. Nella Figura 4.5 si può, invece, osservare l'errore totale di posizionamento del robot calcolato sui tre assi come distanza euclidea. Anche da questo grafico, ci si accorge che l'errore è minimo e accettabile, indicando che il robot riesce a seguire molto accuratamente le posizioni target inviate dal controllore.

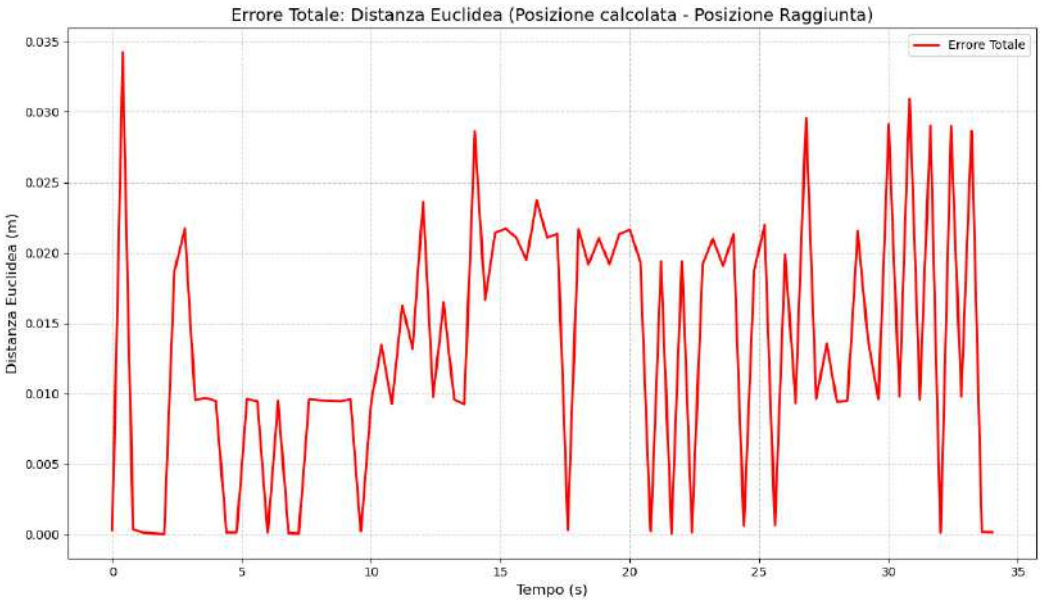

Figure 4.5: *Grafico errore totale distanza euclidea*

L'errore medio osservato durante l'esecuzione è risultato circa pari a 0.021 m, restando ampiamente entro la soglia di tolleranza prefissata di ± 0.05 m. Questi risultati dimostrano una buona coerenza tra la posizione target e quella effettivamente raggiunta dal robot, confermando la validità della pipeline di localizzazione, generazione della pose target e controllo del movimento.

4.4.2 Errore di spostamento

L'errore di spostamento è da intendersi come l'errore che il sistema di controllo introduce tra la posizione calcolata come desiderata e quella, invece, inviata al robot come posizione intermedia, in rispetto della sicurezza di movimento.
È importante notare che in presenza di movimenti ampi e improvvisi del punto rosso, il sistema applica, infatti, una logica di sicurezza: se la variazione tra configurazione attuale e desiderata è troppo elevata, l'end effector **raggiunge una posizione intermedia**, posta lungo la direzione target ma compatibile con i vincoli di sicurezza imposti. Questo meccanismo ha garantito una maggiore stabilità e affidabilità operativa.
In particolare, per valutare la reattività e l'efficacia del controllo del robot viene generato un grafico che mostra l'errore di spostamento del robot dato dalla differenza tra la posizione desiderata calcolata al passo precedente e la posizione effettivamente inviata al controllore nel passo corrente:

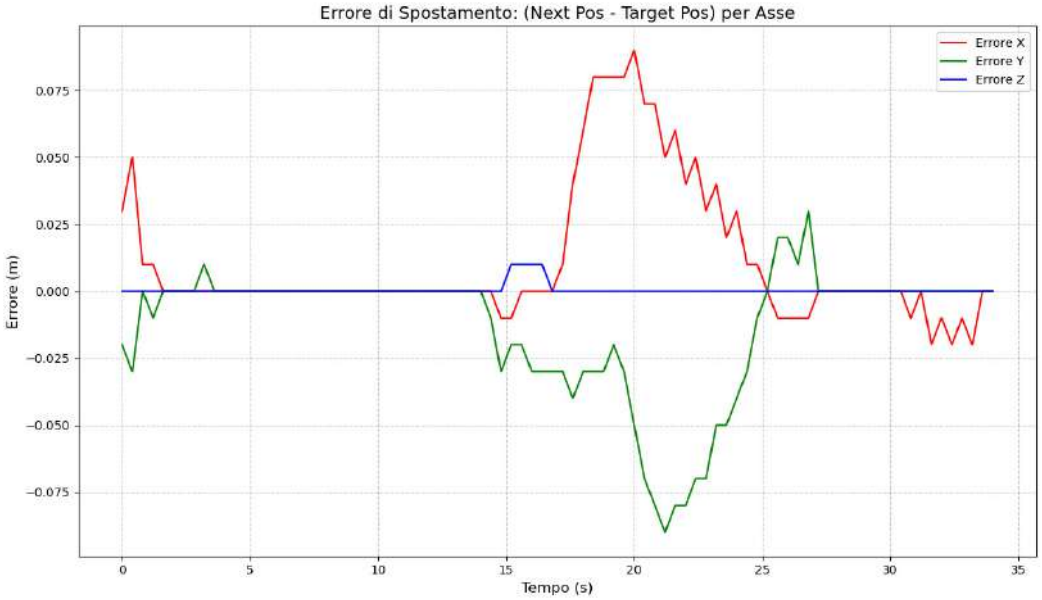

Figure 4.6: *Grafico errore di spostamento per asse*

Dal grafico in Figura 4.6 si può determinare che il sistema di controllo mostra un certo ritardo o incertezza nel caso di traiettorie ampie, adottando correttamente la logica di sicurezza. Si notano errori di spostamento tra la posizione desiderata e quella effettivamente inviata al controllore specialmente negli assi X e Y, dove l'errore massimo è di circa 9 cm.

5 Conclusioni e sviluppi futuri

5.1 Discussione dei risultati

L'analisi dei risultati ottenuti evidenzia diversi aspetti positivi del sistema sviluppato, così come alcune criticità emerse durante l'esecuzione dei test.
Dal punto di vista dell'**accuratezza**, il robot ha mostrato una buona capacità di raggiungere la posizione desiderata mantenendo una distanza costante di 1 metro dal target rilevato. L'errore medio registrato si è mantenuto attorno ai 2 centimetri, un valore soddisfacente per l'applicazione prevista, confermando l'efficacia del sistema di percezione, trasformazione e generazione della pose target.
In termini di *reattività*, grazie al tempo di campionamento di 0.4 secondi e alla velocità massima di 0.1 m/s, il sistema ha dimostrato un buon compromesso tra reattività e sicurezza. In presenza di target con variazioni di posizione significative, il robot ha adattato il proprio comportamento fermandosi in una posizione intermedia se la variazione era troppo elevata, garantendo così movimenti fluidi e sicuri.
L'*affidabilità* del comportamento è risultata elevata: le soluzioni di cinematica inversa sono state valide nella maggior parte dei casi e i controlli di sicurezza (come la verifica della variazione dei giunti o la gestione degli errori IK) hanno funzionato correttamente. L'interazione uomo-robot è stata fluida, grazie anche alla presenza di operatori dedicati al monitoraggio e alla gestione in tempo reale del sistema.
Tuttavia, alcune limitazioni sono emerse durante i test:

- La frequenza di campionamento imposta a 0,4 secondi, seppur sufficiente in molti scenari, si è rivelata talvolta limitante in caso di movimenti rapidi del target.
- Il sistema ha operato prevalentemente sul piano perpendicolare all'asse z del *tool0*, limitando la varietà dei movimenti robotici e richiedendo vincoli sull'orientamento dei giunti.
- L'ambiente di test statico e localizzato ha impedito di verificare il comportamento del robot in scenari più dinamici o su distanze maggiori.

5.2 Conclusioni

Il progetto ha dimostrato con successo la fattibilità di un sistema di co-trasporto uomo-robot basato su UR5 e ROS2, raggiungendo gli obiettivi principali di:

- **Integrazione hardware/software** tra braccio robotico, sensore di profondità e framework ROS2
- **Percezione visiva affidabile** attraverso algoritmi di computer vision robusti
- **Controllo cinematico preciso** con gestione avanzata delle singolarità

- **Interazione sicura** grazie a meccanismi di protezione multi-livello

L'approccio adottato si è rivelato particolarmente efficace nella gestione delle transizioni tra stati operativi, garantendo continuità nell'esecuzione anche in condizioni non ideali. La scelta di ROS2 ha permesso di sfruttare appieno i vantaggi di un'architettura distribuita, modulare e real-time, particolarmente adatta per applicazioni robotiche complesse.

5.3 Sviluppi futuri

Alla luce dei risultati ottenuti e delle limitazioni identificate, si delineano diverse direzioni per il miglioramento e l'estensione del sistema. Sul piano hardware, un primo sviluppo riguarda l'integrazione completa del MiR200 come base mobile attiva, abilitando algoritmi di motion planning coordinato tra il braccio robotico e la piattaforma. Un ulteriore potenziamento prevede l'adozione di sensori aggiuntivi, come force/torque sensors, per implementare strategie di controllo ibrido posizione/forza, nonché l'impiego di sistemi di visione stereo per aumentare l'affidabilità del tracking in ambienti dinamici, queste adozioni porterebbero, sicuramente, a un miglioramento nell'accuratezza globale del sistema ma avrebbero un costo elevato per cui il trade-off *costo-efficacia* è sfavorevole.

Dal punto di vista software, è previsto il passaggio a un'architettura multi-rate con thread dedicati a percezione e controllo, rispettivamente ad alta e bassa frequenza. Verranno inoltre esplorati algoritmi di predizione del movimento umano basati su modelli cinematici, insieme a tecniche di controllo adattivo dei parametri cinematici in funzione del carico trasportato. Infine, si considera una miglioria dell'applicazione considerando un orientamento non fisso che si possa adattare alle gesture dell'operatore umano, implementando, inoltre, un controllo più efficace assegnando le velocità, anzichè solamente la posa finale del robot.

Per quanto riguarda la validazione, è fondamentale estendere i test a scenari non strutturati, caratterizzati dalla presenza di ostacoli dinamici, e condurre valutazioni ergonomiche dell'interazione uomo-robot. Verranno inoltre introdotte metriche quantitative standardizzate per il benchmarking del sistema, accompagnate da analisi di usabilità con utenti non esperti.

Queste evoluzioni aprono la strada a un ampio spettro di applicazioni potenziali, dalla logistica intelligente per il co-trasporto di carichi pesanti in ambienti industriali, al supporto in ambito medico per attività ripetitive, fino all'impiego in cantieri edili per il sollevamento di materiali e all'assistenza domestica per persone con mobilità ridotta. Il sistema sviluppato rappresenta quindi un promettente prototipo per soluzioni robotiche collaborative, con ampie possibilità di evoluzione verso scenari operativi sempre più complessi e non strutturati.

Bibliografia

[1] "Guida ufficiale ROS, author= ROS, url = https://docs.ros.org."

[2] "Driver ROS2 Humble, author= Felix Exner, url = https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver."