

UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA



Corso di Laurea Magistrale in
Ingegneria Informatica e dell'Automazione

**Physical AI per il tracciamento e la manipolazione
intelligente di oggetti**

Physical AI for intelligent object tracking and manipulation

Relatore:
PROF. ADRIANO MANCINI

Tesi di Laurea di:
AGNESE BRUGLIA

Correlatore:
DANIELE DI MUZIO

ANNO ACCADEMICO 2025-2026

Sommario

Il presente lavoro di tesi illustra lo sviluppo e l'integrazione di un sistema basato sui paradigmi della *Physical AI* per un manipolatore robotico collaborativo con il fine di implementare soluzioni per le operazioni di *pick-and-place* e di *visual servoing* di oggetti in ambito industriale. Lo scopo del progetto è di superare i limiti che caratterizzano le soluzioni basate sulle tradizionali traiettorie pre-programmate equipaggiando il robot di un sensore visivo che gli conferisce la consapevolezza del proprio ambiente di lavoro e che gli permette di rispondere in tempo reale a variazioni esterne come la presenza di certi oggetti in movimento.

L'architettura hardware utilizzata è composta dal cobot UR10e della Universal Robots, da una pinza pneumatica Schunk MPC 150 e da un sensore di profondità RGB-D Intel RealSense D435 montato sul polso del manipolatore in configurazione *eye-in-hand*.

L'ecosistema software, invece, è basato sul framework *ROS 2 Humble* installato su un sistema Linux dotato di kernel real-time, fondamentale per garantire il corretto controllo e funzionamento del braccio robotico. Il sistema di visione sfrutta i modelli della famiglia *YOLO*, sottoposti a *fine tuning* su un dataset personalizzato per la detection e il tracking di 6 classi di oggetti noti.

Al fine di convertire i dati visivi acquisiti in comandi di movimento fluidi per il robot e compensare il rumore tipico dei sensori RGB-D, il sistema integra una logica di controllo *Position-Based Visual Servoing (PBVS)* supportata dal framework *MoveIt 2*, che permette di valutare l'errore di posizionamento del robot rispetto al target direttamente nello spazio tridimensionale, garantendo così un'elevata precisione sia nel task di *pick-and-place* sia nel task di *visual servoing*.

Entrambe le soluzioni sviluppate hanno ottenuto risultati soddisfacenti. In particolare, la soluzione di *pick-and-place* presenta un elevato potenziale di impiego in un ambiente operativo reale, col presupposto che il modello di detection e tracking venga addestrato su un dataset specifico dell'applicazione; mentre la soluzione proposta per il task di *visual servoing* costituisce una buona base di partenza, offrendo interessanti margini di miglioramento per sviluppi futuri.

Abstract

This thesis describes the development and integration of a system based on the paradigms of *Physical AI* for a collaborative robotic manipulator, with the aim of implementing solutions for *pick-and-place* and *visual servoing* operations involving objects in an industrial setting. The goal of the project is to overcome the limitations of solutions based on traditional pre-programmed trajectories by equipping the robot with a visual sensor that provides it with awareness of its working environment and allows it to respond in real time to external changes, such as the presence of certain moving objects.

The hardware architecture consists of the UR10e cobot from Universal Robots, a Schunk MPC 150 pneumatic gripper, and an Intel RealSense D435 RGB-D depth sensor mounted on the manipulator's wrist in an *eye-in-hand* configuration.

The software ecosystem, on the other hand, is based on the *ROS 2 Humble* framework installed on a Linux system equipped with a real-time kernel, which is essential for ensuring the proper control and operation of the robotic arm. The vision system utilizes models from the *YOLO* family, which have been *fine-tuned* on a custom dataset for the detection and tracking of 6 classes of known objects.

In order to convert the acquired visual data into smooth motion commands for the robot and compensate for the noise typical of RGB-D sensors, the system incorporates a *Position-Based Visual Servoing (PBVS)* control logic supported by the *MoveIt 2* framework, which allows the robot's positioning error relative to the target to be evaluated directly in three-dimensional space, thus ensuring high precision in both *pick-and-place* and *visual servoing* tasks.

Both solutions developed presented great results. In particular, the *pick-and-place* solution shows great potential for use in a real-world operating environment, provided that the detection and tracking model is trained on an application-specific dataset; while the solution proposed for the *visual servoing* task serves as a good starting point, offering interesting room for improvement in future developments.

Indice

Sommario	2
Abstract	3
1 Introduzione	6
1.1 L'azienda IDEA Soc. Coop.	7
1.2 Struttura della tesi	8
2 Stato dell'arte	9
2.1 L'industria 5.0 e i robot collaborativi	9
2.2 Tecnologie RGB-D e ricostruzione 3D	10
2.2.1 Il modello Pin-Hole	11
2.2.2 Configurazioni di visione	13
2.2.3 Limiti dei sensori RGB-D	14
2.3 Deep learning e computer vision per il grasping	15
2.3.1 Object detection e modelli YOLO	16
2.3.2 Stima della posa 3D	18
2.4 Metodologie di object tracking	19
2.4.1 Tracking-by-detection	20
2.4.2 Object tracking nei modelli YOLO	21
2.4.3 Superare i limiti dell'object tracking	22
2.5 Visual servoing	22
2.5.1 Image-Based VS Position-Based Visual Servoing	23
2.6 Tecniche di denoising per sensori RGB-D	24
2.7 Sintesi dello stato dell'arte	26
2.8 Posizionamento della tesi	27
3 Architettura hardware	29
3.1 Robot collaborativi	29
3.1.1 UR10e	30
3.2 Pinza di presa	32
3.2.1 Circuito ad aria compressa	33
3.3 Intel RealSense D435	35
3.3.1 Intel RealSense SDK 2.0	37
4 Architettura software	39
4.1 Kernel real-time	40
4.1.1 Procedura di compilazione	41

4.2	ROS 2 Humble Hawksbill	42
4.3	Installazione di driver e librerie per ROS 2	43
4.3.1	Driver del manipolatore	43
4.3.2	MoveIt 2	45
4.3.3	Driver e librerie RealSense	46
4.3.4	OpenCV e cv_bridge	47
4.3.5	Ambiente virtuale ROS 2-Python	48
4.3.6	Driver per la scheda grafica	48
4.4	Calcolo del payload	49
4.5	Configurazione del TCP	51
4.5.1	MoveIt Setup Assistant	51
4.6	Compilazione del workspace	59
4.6.1	Errori in fase di build	60
4.7	Calibrazione della camera	62
4.8	File di avvio dell'ecosistema ROS 2	65
4.8.1	Problematiche riscontrate	67
5	Software di visione e controllo	70
5.1	Fine Tuning	70
5.1.1	Acquisizione del dataset e labeling	72
5.1.2	Addestramento dei modelli e studio comparativo	74
5.2	Pick-and-place statico	79
5.2.1	Sistema di visione	79
5.2.2	Sistema di controllo	84
5.2.3	Funzionamento generale	86
5.3	Visual servoing	88
5.3.1	Sistema di visione	88
5.3.2	Sistema di controllo	89
5.3.3	Funzionamento generale	91
5.4	Avvio degli script	92
6	Conclusioni e sviluppi futuri	93
6.1	Considerazioni	93
6.1.1	Pick-and-place	93
6.1.2	Visual servoing	95
6.1.3	Conclusioni	95
6.2	Sviluppi futuri	97

Capitolo 1

Introduzione

L'obiettivo principale di questo lavoro di tesi è l'implementazione di una soluzione di *Physical AI* applicata alla robotica collaborativa per il tracciamento e la manipolazione intelligente di oggetti. Nello specifico, il sistema sfrutta un approccio *Vision-to-Action* per risolvere i task di *pick-and-place* e di *visual servoing* di oggetti.

Con il termine “*Physical AI*” (intelligenza artificiale fisica) si definisce l'applicazione dei modelli di intelligenza artificiale all'interno di sistemi hardware al fine di renderli capaci di interagire fisicamente e in modo autonomo con il loro ambiente operativo. All'interno di questo paradigma, l'approccio “*Vision-to-Action*” rappresenta la pipeline che lega la percezione dell'ambiente, affidata a un sensore di visione, all'esecuzione meccanica: il sistema traduce i flussi di dati acquisiti dal sensore in comandi per il manipolatore invece di affidarsi a coordinate fisse e pre-programmate.

L'architettura proposta in questa tesi applica tali concetti a un *robot collaborativo*, garantendo l'accuratezza necessaria per le applicazioni di *pick-and-place* e di *visual servoing* grazie all'integrazione di una camera RGB-D in configurazione *eye-in-hand*, che permette di ricavare informazioni visive dall'ambiente di lavoro e dati di profondità in tempo reale.

Con lo scopo di ottenere un'accuratezza elevata nella componente di “*vision*” si è proposta la specializzazione di un modello appartenente alla famiglia *YOLO* tramite un processo di *fine tuning* su un dataset personalizzato, in maniera tale da garantire il rilevamento e il tracciamento delle pose di oggetti appartenenti a 6 classi note.

La componente di “*action*”, invece, è stata implementata sviluppando un'architettura di controllo basata su *Position-Based Visual Servoing (PBVS)* all'interno dell'ambiente ROS 2 Humble e supportata dal framework MoveIt 2, strategia che permette di calcolare l'errore di posizionamento direttamente nello spazio tridimensionale e di generare traiettorie per il posizionamento del robot in maniera precisa e fluida sia nel caso di target statico sia nel caso di target in movimento.

1.1 L'azienda IDEA Soc. Coop.

L'intero progetto descritto in questa tesi è stato realizzato in ambito di tirocinio curricolare presso IDEA Soc. Coop., società di ingegneria con sede legale ad Ancona. Nella figura 1.1 è mostrato il logo della società, mentre in [1] è riportato il sito aziendale.



Figura 1.1: Logo di IDEA Soc. Coop. (immagine tratta da [1])

IDEA è stata costituita nel 2007 come spin-off accademico dell'Università Politecnica delle Marche afferente al Dipartimento DIIGA dell'università stessa. La sua costituzione è stata possibile grazie alla vincita da parte dei tre soci fondatori del concorso regionale eCapital nel 2006 volto alla promozione di nuove idee imprenditoriali. IDEA si pone come partner tecnologico dei propri clienti fornendo soluzioni personalizzate, altamente innovative e chiavi in mano, partendo dallo studio di fattibilità fino alla consegna del prodotto/servizio finale.

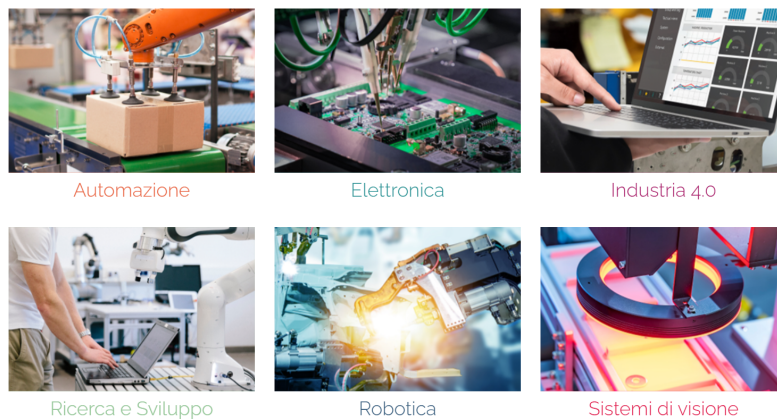


Figura 1.2: Settori in cui opera IDEA Soc. Coop. (immagine tratta da [1])

L'azienda è strutturata per operare in differenti settori, illustrati nella Figura 1.2:

- *automazione* e *robotica*: sviluppo di sistemi per test e collaudo di sistemi di supervisione, tracciabilità e controllo qualità;
- *elettronica*: progettazione e sviluppo di PCB;

- *industria 4.0*: interconnessione di impianti;
- *ricerca e sviluppo e sistemi di visione*: sviluppo di soluzioni IoT e AI.

1.2 Struttura della tesi

La presente tesi è strutturata come segue:

- capitolo 2: descrive lo stato dell'arte riguardo ai robot collaborativi, ai sensori RGB-D e alle relative sfide sul rumore, ai modelli di deep learning per l'object detection e l'object tracking e alle basi del visual servoing;
- capitolo 3: analizza i componenti hardware del sistema, partendo dal braccio robotico e dalla pinza per la presa e arrivando alle specifiche della camera;
- capitolo 4: illustra l'architettura software e il procedimento per il setup del sistema, descrivendo dei concetti fondamentali come l'adozione del kernel real-time, il framework ROS 2 Humble con MoveIt 2 per la pianificazione dei movimenti del robot e la procedura di configurazione del Tool Center Point (TCP), analizzando al contempo le problematiche affrontate;
- capitolo 5: spiega il processo di addestramento della rete neurale utilizzata, l'approccio al task di pick-and-place statico e l'approccio al task di visual servoing;
- capitolo 6: riassume i risultati ottenuti, discutendo le conclusioni del lavoro svolto e delineando i possibili sviluppi futuri.

Capitolo 2

Stato dell'arte

Il presente capitolo ha l'obiettivo di contestualizzare le scelte implementate nel progetto sviluppato e costituisce un riassunto dello stato dell'arte e di alcuni dei metodi esistenti e utilizzati nel contesto della computer vision e dell'automazione industriale. In particolare, parlando di sensori RGB-D e del controllo robotico basato su visione, per l'interazione tra un manipolatore industriale e l'ambiente reale è necessaria la sincronizzazione di tre componenti: l'acquisizione del dato spaziale, il riconoscimento dei target e la traduzione di tali informazioni in traiettorie di controllo per il robot.

Il capitolo inizia descrivendo che cosa si intende con “*industria 5.0*” e che cosa sono i robot collaborativi; successivamente vengono presi in esame i sensori di visione RGB-D, elencando i diversi possibili principi fisici di funzionamento e analizzando il rumore ottico, spesso presente nelle misure e responsabile della degradazione dell'accuratezza. Continua approfondendo le architetture di deep learning per l'object detection e la stima della posa, analizzando modelli popolari come YOLO di Ultralytics; infine, esamina gli algoritmi di tracciamento dei target e i metodi di inseguimento visivo di tali target, fondamentali per garantire che il tool di presa (d'ora in poi chiamato “*end effector*”) mantenga una distanza costante dall'oggetto.

L'obiettivo finale è mettere in luce i principali limiti che caratterizzano i sistemi attuali, come la latenza computazionale, la sensibilità ai riflessi e alle ombre su e intorno agli oggetti e l'instabilità della misura causata dal jitter della camera utilizzata. Tale analisi permetterà di definire il posizionamento della tesi nel contesto delle soluzioni esistenti, sottolineando la capacità dell'architettura proposta di mitigare le problematiche elencate unendo l'efficienza di calcolo richiesta dalle applicazioni industriali all'accuratezza necessaria per un grasping efficace.

2.1 L'industria 5.0 e i robot collaborativi

Nel corso del decennio passato si è sviluppato il paradigma dell'*industria 4.0*, focalizzato sulla digitalizzazione dei processi produttivi e sull'adozione dei *sistemi ciberfisici* (*Cyber-Physical Systems*, CPS) e dell'*internet delle cose* (*Internet of Things*, IoT), che aveva come obiettivo l'ottimizzazione dell'efficienza e la

riduzione dei costi di produzione attraverso il paradigma della manutenzione predittiva. In questo contesto, i robot manipolatori erano impiegati per eseguire compiti altamente ripetitivi a velocità elevate, lavorando in un ambiente di lavoro statico, predefinito e isolato dagli operatori umani tramite barriere di sicurezza.

Il paradigma dell'*industria 5.0* è nato come un'evoluzione dell'*industria 4.0*, focalizzandosi non più solamente sull'efficienza tecnologica e quantitativa, ma anche sullo sviluppo di processi a basso impatto ambientale e sul posizionamento dell'operatore umano al centro del processo produttivo e sull'utilizzo della tecnologia non più come sostituzione, ma come strumento ausiliario del lavoratore.

L'innovazione principale introdotta nel contesto dell'*industria 5.0* è rappresentata dai *robot collaborativi*, o *cobot*, che, a differenza dei manipolatori industriali utilizzati in passato, hanno la caratteristica fondamentale di essere progettati per operare in assenza di barriere di sicurezza condividendo lo spazio di lavoro con l'operatore umano. Perché questa collaborazione sia possibile, i *cobot* hanno dei sistemi di sicurezza integrati, come la limitazione di potenza e forza, per cui viene monitorato costantemente lo stato dei giunti tramite dei sensori in maniera tale da arrestare il robot in caso di superamento dei limiti o in caso di collisione con oggetti o persone, e interfacce di programmazione intuitive, che permettono a qualsiasi operatore di utilizzare intuitivamente il robot.

2.2 Tecnologie RGB-D e ricostruzione 3D

Il primo dei componenti necessari per l'interazione tra un manipolatore industriale e l'ambiente reale è rappresentato dal sensore di visione, il cui compito è acquisire il dato spaziale della scena. Normalmente, la tipologia di sensori impiegati per tale scopo è quella dei sensori di visione RGB-D (*Red-Green-Blue-Depth*), che, come evidenziato nell'articolo [2], integrano in un'unica pipeline hardware e software, illustrata nella Figura 2.1, la capacità di catturare l'informazione bidimensionale e la mappa di profondità associata, consentendo di ricostruire la tridimensionalità dell'ambiente operativo.

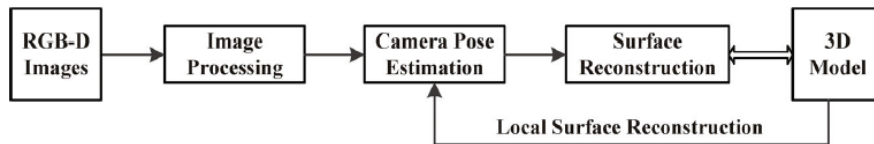


Figura 2.1: Pipeline di ricostruzione di una scena 3D a partire da un sensore RGB-D (immagine tratta da [2])

Come discusso nello studio proposto dallo stesso articolo [2] sulla ricostruzione 3D in ambienti indoor, i sensori di profondità possono essere classificati in tre macro-categorie principali, ciascuna caratterizzata da un diverso principio di funzionamento:

1. *Time-of-Flight (ToF)*: sono sensori basati sul tempo di volo e sono caratterizzati dal determinare la distanza dagli oggetti nella scena calcolando il ritardo tra l'emissione di un segnale luminoso e la sua ricezione da parte di un componente fotosensibile. Il principale vantaggio di questo tipo di sensori è rappresentato da avere una latenza computazionale minima, dovuta al fatto che il calcolo della profondità avviene a livello hardware e per singolo pixel, rendendo possibili frequenze di aggiornamento elevate (> 60 Hz), che sono ideali per applicazioni robotiche che richiedono una velocità elevata. Dall'altro lato, sono particolarmente suscettibili a riflessi luminosi nell'immagine e alla saturazione causata dalla diversa luce ambientale.
2. *luce strutturata*: in questa tipologia di sensori viene sfruttata la proiezione di un pattern luminoso, come una griglia di punti, sulla superficie degli oggetti; successivamente una o più telecamere acquisiscono l'immagine di tale pattern, che risulta deformato dalla topologia della scena, e attraverso algoritmi di decodifica e triangolazione ottica viene ricostruita la superficie 3D. Tali sensori garantiscono dei livelli di accuratezza e risoluzione metrica a corto raggio, ovvero inferiori al millimetro, molto elevati, il che li rende lo standard industriale per l'ispezione di qualità. Di contro, la necessità di elaborare pattern complessi introduce una latenza non trascurabile, limitandone l'efficienza e l'uso in situazioni di controllo real-time.
3. *stereovisione attiva*: questi sensori rappresentano un'evoluzione della stereoscopia passiva, il cui scopo è ricreare il modo in cui l'essere umano vede, ovvero con una visione binoculare che sfrutta la differenza tra due viste traslate. Tale metodologia, però, non risulta efficace nel caso di superfici omogenee o prive di texture; al contrario, la stereovisione attiva introduce un proiettore di texture a infrarossi non strutturato, approccio che è stato analizzato dettagliatamente nell'articolo [3] per lo studio di caratterizzazione della telecamera Intel RealSense D435. Questo proiettore permette di conferire in maniera artificiale un contrasto alla superficie dell'oggetto, consentendo così agli algoritmi di corrispondenza, di *stereo matching*, di calcolare la differenza tra le due viste traslate anche su superfici uniformi. Secondo gli autori dello studio [3], questa architettura costituisce un ottimo compromesso per il grasping robotico e il visual servoing, in quanto associa un frame-rate elevato, e quindi una bassa latenza, a dimensione e peso ridotti, che sono ideali per l'integrazione di un sensore su sistemi robotici mobili.

Nel contesto di questa tesi è stata utilizzata una Intel RealSense D435 (di cui si parla nel dettaglio nella Sezione 3.3), ovvero un sensore a stereovisione attiva. Per questo motivo, l'attenzione verrà rivolta maggiormente a questa tipologia di sensori.

2.2.1 Il modello Pin-Hole

Il *modello Pin-Hole*, mostrato nella Figura 2.2, è il concetto utilizzato in tutte le tipologie di sensori di visione grazie al quale è possibile effettuare la tran-

sizione dallo spazio bidimensionale dell'immagine espresso in pixel al dominio tridimensionale espresso in unità metriche.

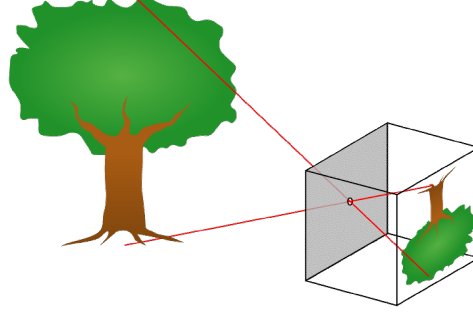


Figura 2.2: Modello Pin-Hole (immagine tratta da [4])

In particolare, siano $[X, Y, Z]^T$ le coordinate di un punto fisico P nello spazio tridimensionale riferite al sistema di riferimento della telecamera, e siano (u, v) le coordinate della sua proiezione sul piano dell'immagine espressa in pixel; allora, la trasformazione definita da questo modello è governata dalla matrice dei parametri intrinseci K , definita come:

$$K = \begin{bmatrix} f_x & 0 & c_u \\ 0 & f_y & c_v \\ 0 & 0 & 1 \end{bmatrix} \quad (2.1)$$

dove f_x e f_y rappresentano le lunghezze focali della telecamera scalate lungo gli assi ortogonali del sensore, mentre c_u e c_v indicano le coordinate del punto.

La relazione di proiezione è espressa dalla formula:

$$Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_u \\ 0 & f_y & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (2.2)$$

Invertendo tale formula, si ottiene l'operazione di de-proiezione geometrica, la quale permette di ricostruire la posizione cartesiana di un pixel di cui si conosce la profondità Z estratta dal sensore RGB-D utilizzato. Questa posizione viene espressa come segue:

$$X = \frac{(u - c_u) \cdot Z}{f_x} \quad (2.3)$$

$$Y = \frac{(v - c_v) \cdot Z}{f_y} \quad (2.4)$$

Ovviamente, l'accuratezza di questo processo di transizione è condizionata dalla capacità di stimare correttamente la matrice K oltre che dalla modellazione dei coefficienti di distorsione ottica (distorsione radiale e tangenziale indotta dalle lenti), che solitamente viene formalizzata utilizzando i modelli proposti della letteratura di Zhang [5]. Tuttavia, come evidenziato nello studio condotto nell'articolo [6], i parametri di calibrazione preimpostati dai costruttori dei

sensori spesso mostrano limiti di accuratezza rispetto ai requisiti richiesti nei contesti industriali. Così, per superare questi limiti, gli autori di tale articolo [6] introducono una metodologia che permette di unire i dati della camera a un sistema esterno di cattura del movimento, così da consentire di ricalibrare e allineare sia la distorsione dell'immagine sia l'allineamento dei dati di profondità, garantendo così una precisione maggiore.

2.2.2 Configurazioni di visione

Una volta determinati i parametri del sensore RGB-D, per integrare il sistema di visione all'interno del sistema di manipolazione è necessario calibrare il sensore. Come documentato nell'articolo [7], questa procedura è fondamentale per determinare la trasformazione di rototraslazione che lega il sistema di riferimento della telecamera al sistema di riferimento del robot, consentendo di tradurre le coordinate dei target in traiettorie eseguibili dal manipolatore.

Lo stesso articolo [7] distingue due tipologie differenti di configurazione dei sensori di visione:

1. configurazione *Eye-in-hand* o camera-on-wrist: il sensore è montato sul manipolatore, solitamente sulla flangia o vicino all'end-effector del robot. In questo caso, illustrato nella parte sinistra della Figura 2.3, la camera si muove in maniera solidale al braccio robotico e, come evidenziato dallo studio dell'articolo [8], il principale vantaggio di questo approccio è l'aumento della precisione del sensore in maniera proporzionale all'avvicinarsi dell'end-effector all'oggetto da afferrare. Dall'altra parte, lo svantaggio è rappresentato dalla limitazione del campo visivo, anche questo variabile in base alla vicinanza all'oggetto target, che introduce il rischio di ostruzione parziale della visuale dell'oggetto a causa della struttura stessa del tool di presa.
2. configurazione *Eye-to-hand* o fixed-camera: il sensore è posizionato in un punto fisso dell'ambiente di lavoro, come illustrato nella parte destra della Figura 2.3, ovviamente orientato in direzione dello spazio operativo del robot. Nell'articolo [8] viene evidenziato come questa configurazione, al contrario della precedente, garantisca una visione globale e stabile dell'ambiente, facilitando la pianificazione delle traiettorie, evitando in parte il problema dell'ostruzione del target ed evitando l'aggiunta di payload e cablaggi complessi sul polso del robot. Di contro, è chiaro che l'accuratezza è minore rispetto al caso eye-in-hand a causa della maggiore distanza dall'oggetto; inoltre, il braccio robotico stesso potrebbe causare lo stesso problema di ostruzione, parziale o totale, del target. Perciò, per lavorare con questo tipo di configurazione di camera, è necessario assicurarsi di prevenire ed evitare tale problematica.

Visti i limiti a cui sono soggette entrambe le tipologie di configurazione, la ricerca si è orientata verso lo sviluppo di sistemi multi-camera; in particolare,

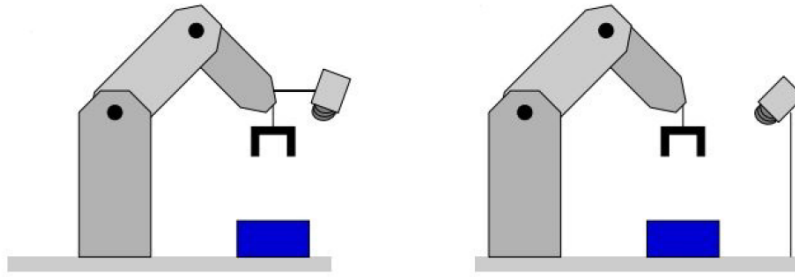


Figura 2.3: Tipologie di configurazione del sensore di visione (immagine tratta da [9])

gli autori dell'articolo [10] hanno introdotto un framework di *visual servoing*¹ basato sull'utilizzo sia di camere con configurazione eye-to-hand sia di camere eye-in-hand. Tale metodo sfrutta un filtro di Kalman esteso che unisce i dati di tutte le camere utilizzate, riuscendo così a ottimizzare la stima della posa dell'oggetto e riducendo anche l'effetto dell'ostruzione del target.

Allo stesso modo, gli autori dell'articolo [11] hanno affrontato i problemi legati ai due tipi di calibrazione proponendo un nuovo metodo di calibrazione per sistemi ibridi capace di calcolare in un unico passo ottimizzato le matrici di trasformazione della base del robot, dell'end-effector e di tutte le camere incluse nel contesto operativo del robot. Tale approccio ha permesso di ridurre di molto i tempi necessari per il setup del sistema e di aumentare considerevolmente la robustezza complessiva dell'architettura di grasping nonostante il rumore stocastico di misura.

Ovviamente, ogni configurazione di calibrazione proposta e analizzata ha i propri limiti operativi e non ce n'è una che sia universalmente migliore. Perciò, è compito del progettista del sistema valutare quale configurazione sia la più adatta in base al contesto dell'applicazione.

2.2.3 Limiti dei sensori RGB-D

Per quanto riguarda i sensori a stereovisione attiva che vengono impiegati in contesti industriali, nonostante ormai i moderni sistemi di visione siano ben consolidati e affidabili, sono comunque soggetti a dei loro limiti intrinseci. Ad esempio, sono sensibili alle luci ambientali e ai riflessi sugli oggetti; inoltre, le superfici degli oggetti da riconoscere e manipolare potrebbero essere tali da introdurre rumore ottico e geometrico e, quindi, peggiorare la qualità della misura. I limiti principali di un sensore RGB-D possono essere divisi in errori sistematici e non sistematici (come il rumore casuale a componente stocastica), e sono:

- dipendenza dalla distanza e dall'angolo di incidenza: gli autori dell'articolo [3] hanno modellato il comportamento del rumore *non sistematico* separando la componente assiale da quella laterale, dimostrando che l'errore assiale, quello lungo l'asse Z , non è uniforme nello spazio, ma cresce

¹Con “visual servoing” si intende la tecnica di controllo robotico in cui i dati di una o più camere vengono utilizzati in tempo reale per controllare il movimento del robot rispetto al suo ambiente o alla posizione di un target di riferimento.

in modo quadratico all'aumentare della distanza del target dal sensore ($e_{axial} \propto Z^2$). Inoltre, i dati che hanno raccolto dimostrano che la componente di rumore laterale cresce rapidamente quando l'angolo di incidenza del raggio ottico rispetto alla normale della superficie dell'oggetto si fa obliquo, ovvero nel caso di superfici curve o inclinate, creando così delle distorsioni evidenti.

- assorbimento luminoso e riflessione: i materiali caratterizzati da superfici scure o opache assorbono la radiazione infrarossa emessa dal proiettore attivo di texture; fenomeno che, come evidenziato dagli articoli [2] e [3], impedisce alle camere di estrarre feature valide e determinando aree prive di dati: *invalid depth pixels* o buchi di profondità. Invece, i materiali lucidi, come i materiali metallici, generano dei riflessi che saturano il sensore o alterano il calcolo della disparità stereo, danneggiando l'integrità dei dati raccolti e risultando in un'immagine con rumore stocastico non trascurabile.
- rumore sui bordi (*edge bleeding* e *flying pixels*): nel caso di contorni dell'oggetto da afferrare non chiari rispetto allo sfondo o al piano di lavoro, il sensore si confonde e non rispecchia la reale forma dell'oggetto. Come spiegato dettagliatamente nell'articolo [2], i pixel che si trovano in prossimità del bordo dell'oggetto possiedono delle informazioni che sono il risultato di un segnale misto, riflesso parzialmente dall'oggetto e parzialmente dalla superficie sottostante. L'algoritmo di corrispondenza stereo del sensore elabora questo segnale ambiguo generando punti fittizi fluttuanti nel vuoto, chiamati *flying pixels*. Gli autori dell'articolo [2] evidenziano come questi punti creino una transizione smussata dove, invece, dovrebbe esistere un gradiente netto, falsando così la reale geometria dell'oggetto e portando a instabilità nel calcolo della posa del target e nella successiva pianificazione delle traiettorie del robot.

2.3 Deep learning e computer vision per il grasping

Negli ultimi anni, il campo del grasping di oggetti tramite manipolatore industriale si è evoluto considerevolmente, arrivando ad affidarsi del tutto all'utilizzo di architetture di *deep learning* e, in particolare, ad architetture basate su *reti neurali convoluzionali*, *CNN*, di cui un esempio è mostrato nella Figura 2.4.

In generale, una CNN, come si vede dalla figura precedente, è composta da uno strato di input, uno strato di output e degli strati intermedi nascosti, di convoluzione, pooling e completamente connessi, che possono variare in numero a seconda della struttura della rete e delle esigenze dell'applicazione in cui questa viene utilizzata.

Questa evoluzione ha lo scopo di superare i limiti che caratterizzano gli approcci euristici tradizionali, che, nonostante siano efficaci in ambienti controllati, sono invece poco affidabili nei contesti delle applicazioni industriali reali. Al contrario, le CNN sono capaci di apprendere autonomamente gerarchie di rappresentazione visiva, permettendo di aumentare notevolmente le prestazioni

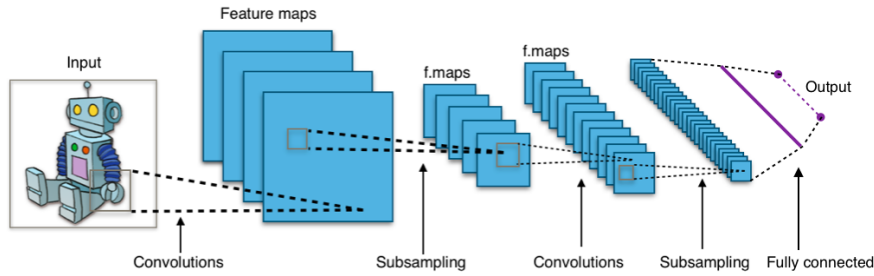


Figura 2.4: Esempio di CNN (immagine tratta da [12])

di un sistema robotico nel caso di disturbi esterni durante il funzionamento, come cambi di illuminazione, riflessi o ostruzioni degli oggetti da manipolare.

Tuttavia, nonostante le reti neurali migliorino la comprensione dello spazio operativo, impongono anche dei vincoli importanti per quanto riguarda le risorse computazionali necessarie per il loro funzionamento e latenza di inferenza: infatti, per un sistema di grasping efficace, esiste un tempo necessario per far sì che il sensore percepisca l'ambiente e decida di conseguenza; tale tempo di latenza, ovviamente, deve essere compatibile con i requisiti richiesti dal contesto in cui vengono applicate le CNN.

Per questo motivo, attualmente la ricerca in questo ambito cerca di bilanciare le potenzialità di modelli di CNN complessi con la necessità di utilizzare tali modelli su sistemi embedded, che hanno risorse computazionali limitate e, quindi, richiedono un'esecuzione rapida e leggera.

2.3.1 Object detection e modelli YOLO

Esistono due principali paradigmi di computer vision basata su deep learning per object detection, ovvero *two-stage* e *one-stage*:

- i modelli *two-stage* suddividono il problema della detection di un oggetto, come dice il nome, in due step sequenziali separati, di cui il primo è un *RPN*, *Region Proposal Network*, che ha lo scopo di identificare le *RoI* - *Regions of Interest* - che hanno un'alta probabilità di contenere un oggetto. Il secondo step ritaglia e ridimensiona le *RoI* identificate e le passa a una rete neurale, che ne classifica il contenuto e ne definisce il *bounding box*. Questo approccio ha il vantaggio di garantire un'accuratezza molto elevata, soprattutto per quanto riguarda la segmentazione dei dettagli, però la sua natura sequenziale introduce un collo di bottiglia e una latenza elevata, rendendolo poco efficace in contesti dinamici real-time.
- i modelli *one-stage*, a differenza degli altri, eliminano la fase di generazione delle *RoI*, unendo la localizzazione e la classificazione in un unico problema: l'immagine di input viene processata una sola volta da una sola CNN, che predice sia le coordinate dei bounding box sia la probabilità della classe di appartenenza, presentando quindi il vantaggio principale di ridurre di molto i tempi di calcolo.

Per quanto riguarda le applicazioni di grasping robotico, attualmente la tipologia di modelli che viene maggiormente utilizzata è quella di tipo *one-stage*, poiché, in quanto leggeri e veloci, tali modelli sono ottimali per consentire il controllo del robot in tempo reale.

Di questa categoria fanno parte i modelli della famiglia *YOLO* - *You Only Look Once* - di Ultralytics, che sono nati proprio con l'obiettivo di unificare la pipeline di detection, che fino a quel momento era spezzata in più step, puntando alla semplicità operativa: meno componenti significa meno possibilità di failure e meno richiesta di risorse computazionali.

Per questo motivo, i modelli one-stage come YOLO sono preferiti a quelli two-stage nei contesti in cui la latenza è critica e il sistema deve girare su dispositivi embedded privi di GPU dedicate. Come dimostrato dagli autori dell'articolo [13], infatti, l'utilizzo dei modelli YOLO, nel loro caso della versione YOLOv8, permette di ottenere una detection dell'oggetto target robusta e accurata; inoltre, le informazioni estratte riguardo all'orientamento dell'oggetto consentono di avere frequenze tali da riuscire a controllare un robot manipolatore minimizzando l'uso di risorse hardware.

Sul sito ufficiale di Ultralytics [14] sono disponibili diverse versioni di modelli YOLO, di cui le principali sono raffigurate nella Figura 2.5, che ne mette a confronto le prestazioni.

Ovviamente, ogni versione successiva rilasciata da Ultralytics apporta modifiche e miglioramenti rispetto alla precedente; in particolare, la prima versione introduce il paradigma one-shot e le successive (YOLO9000 e YOLOv2) presentano innovazioni con lo scopo di stabilizzare il training e la generalizzazione, come l'uso di *anchor boxes*², e un trade-off ottimale tra velocità e accuratezza. A partire da YOLOv3 e YOLOv4, invece, vengono introdotti dei pattern architettureali come Darknet, che diventano lo standard per l'estrazione di feature multiple.

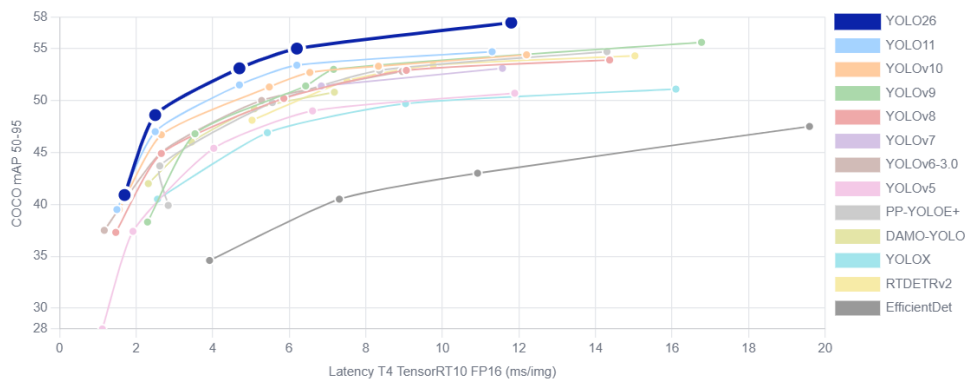


Figura 2.5: Confronto tra le prestazioni dei modelli YOLO (immagine tratta da [14])

²Con “anchor boxes” si intende un insieme di rettangoli di riferimento con dimensioni e proporzioni predefinite che vengono calcolati statisticamente sul dataset di addestramento. Così, la rete neurale non deve creare i bounding box da zero, ma calcola l'offset tra i box e la forma reale degli oggetti.

Versioni più recenti, invece, come YOLOv8, YOLOX o YOLOv10, si sono concentrate sull'efficienza introducendo varianti anchor-free. Come evidenziato nell'articolo [15], tale transizione è estremamente rilevante nel contesto della robotica, poiché, in scenari multi-oggetto complessi, l'utilizzo di questi meccanismi anchor-free e basati su heatmap consente di stimare il punto centrale dell'oggetto per il grasping, riuscendo a semplificare la struttura della rete ed i colli di bottiglia computazionali.

Concretamente, i modelli YOLO vengono utilizzati sfruttando il paradigma del *transfer learning* combinato al *supervised learning*. Addestrare una rete neurale che risulti accurata partendo da zero richiede un'enorme mole di dati oltre che a tempi di addestramento molto lunghi e risorse computazionali elevatissime. Perciò, molto spesso vengono utilizzati i modelli di YOLO con i loro pesi pre-addestrati sul dataset *COCO - Common Objects in Context* - (il cui sito ufficiale è [16]), che contiene milioni di istanze di oggetti reali appartenenti a 80 classi diverse.

Sfruttando quindi questi modelli pre-addestrati, viene poi effettuato il processo di *fine-tuning*, che consiste nel riaddestrare tutti o solo alcuni dei livelli della rete utilizzando un dataset personalizzato e contenente solo i target dell'applicazione.

Il risultato di questo processo è un modello specializzato che richiede solo poche centinaia o migliaia di immagini dello scenario specifico di lavoro e che permette di controllare il manipolatore in sicurezza.

2.3.2 Stima della posa 3D

Dopo aver scelto il modello per l'object detection e una volta localizzato l'oggetto nell'immagine, è necessario tradurre la posa di tale oggetto in coordinate per il movimento del robot.

Esistono diversi approcci in letteratura per risolvere questo problema, tra cui due principali: la *6D pose estimation* e *approcci generativi ed end-to-end*.

I modelli per la *6D pose estimation*, come PoseCNN, DOPE o DenseFusion, si chiamano così in quanto calcolano i 6 gradi di libertà del target rispetto alla camera, ovvero $X, Y, Z, Roll, Pitch$ e Yaw . Tali modelli, nonostante siano caratterizzati da una precisione molto elevata, hanno lo svantaggio di richiedere sia un costo computazionale considerevole sia, nella maggior parte dei casi, la disponibilità di modelli CAD dell'oggetto per l'addestramento.

La seconda tipologia di modelli è costituita da quelli *generativi ed end-to-end*, come Dex-Net, GG-CNN o architetture basate su fusione RGB-D profonda, come GR-ConvNet, analizzata dagli autori dell'articolo [17]. Tali approcci sono caratterizzati dal mappare l'immagine visiva direttamente in una mappa di qualità della presa, angolo e apertura del gripper senza calcolare la posa 6D dell'oggetto in maniera esplicita. Nonostante tali modelli mostrino una flessibilità notevole per quanto riguarda il generalizzare su oggetti mai visti, nell'articolo [17] vengono messe in luce due problematiche di questo approccio: la prima è che in scenari industriali dinamici queste reti sono soggette al fenomeno del *catastrophic forgetting*, che porta il modello a perdere la capacità di manipolare gli oggetti appresi in passato ogni volta che viene addestrato su nuovi target; la

seconda è che per unire i dati RGB con la mappa di profondità (D) in maniera efficace e senza perdere informazioni, questi sistemi richiedono architetture molto complesse, aumentando di molto il carico computazionale della rete.

2.4 Metodologie di object tracking

Con “*object tracking*” ci si riferisce a un processo dinamico nella computer vision che consiste nell’identificare entità specifiche in un video e nel monitorarne il movimento attraverso una sequenza di fotogrammi. A differenza dell’analisi statica delle immagini, che tratta separatamente ogni frame, il tracking introduce la dimensione del tempo. Questo consente ai sistemi di intelligenza artificiale di assegnare un numero identificativo, un ID univoco, come è mostrato nella Figura 2.6, a ogni elemento rilevato, come un’auto, una persona o un animale, e di mantenere tale identità mentre l’oggetto si muove, cambia orientamento o viene temporaneamente oscurato.

Quindi, per fare in modo che il robot riesca a seguire i movimenti di un oggetto target dopo che tale oggetto sia stato identificato e localizzato all’interno dello spazio operativo, il sistema deve riuscire a riconoscere che l’oggetto che si muove rimane lo stesso nel tempo. Nelle applicazioni reali di manipolazione, questa situazione rappresenta una sfida per diversi motivi: gli oggetti non sono sempre statici, ma possono spostarsi su nastri trasportatori o subire perturbazioni introdotte dall’interazione fisica; oltre a ciò, la visione di una camera è sempre affetta da rumore, occlusioni temporanee del target e jitter.



Figura 2.6: Esempio di object tracking (immagine tratta da [18])

L’object tracking viene utilizzato in moltissimi contesti, soprattutto in quelli in cui non basta sapere cosa sia un oggetto, ma è fondamentale capire la direzione in cui si muove nel tempo. I principali ambiti applicativi includono la *robotica industriale* e la *manipolazione*, in cui viene utilizzato per afferrare correttamente oggetti in movimento su nastri trasportatori, *guida autonoma e automotive*, in cui è fondamentale per tracciare la traiettoria di pedoni, ciclisti e altri veicoli circostanti, *interazione uomo-macchina* e *realtà aumentata*, in cui serve per tracciare i movimenti delle mani o del corpo in contesti di gaming e chirurgia robotica assistita, e molti altri ancora.

2.4.1 Tracking-by-detection

I sistemi di tracking di oggetti applicati nella robotica collaborativa e industriale sono basati prevalentemente sul paradigma del *tracking-by-detection*, approccio in cui un algoritmo di object detection come *YOLO* viene eseguito per localizzare i target e, successivamente, un algoritmo di tracking si occupa di assegnare l'ID univoco a ciascun oggetto identificato. Questo processo si articola in tre fasi principali, all'interno delle quali l'architettura software di controllo è rappresentata dall'algoritmo SORT (*Simple Online and Realtime Tracking*):

1. *detection*: un modello di object detection scansiona ogni frame dell'immagine per localizzare gli oggetti di interesse producendo i bounding box che li racchiudono.
2. *predizione del movimento*: in questa fase, l'algoritmo SORT istanzia un *filtro di Kalman* per ciascun oggetto identificato, con l'obiettivo di stimarne la posizione futura di tali oggetti basandosi sulla velocità e sulla traiettoria calcolate nei frame precedenti.
3. *associazione dei dati*: in questa fase, SORT si occupa di risolvere il problema dell'accoppiamento ottimale tra le vecchie tracce predette e i nuovi box geometrici: vengono combinate la metrica della *Intersection over Union (IoU)* - per misurare quantitativamente la sovrapposizione geometrica dei box - con l'algoritmo Ungherese (*Hungarian Algorithm*), un metodo di ottimizzazione combinatoria che assegna l'identificativo corretto a ciascun oggetto.

Tuttavia, l'algoritmo SORT fallisce nei casi di occlusioni prolungate dell'oggetto target; per questo motivo, in letteratura sono stati introdotti algoritmi come DeepSORT e StrongSORT.

Ad esempio, gli autori dell'articolo [19] dimostrano che tali modelli estendono la metrica di associazione dei dati includendo vettori di feature visive estratti grazie a delle reti neurali dedicate alla Re-Identificazione (*Re-ID*), per cui l'associazione dei bounding box non avviene più solamente per vicinanza geometrica, ma anche per somiglianza estetica. Di contro, come sostengono nello stesso articolo [19], è chiaro che gestire scenari industriali complessi richiede delle ottimizzazioni attente e rigorose sia per quanto riguarda la precisione della detection sia per l'addestramento dei modelli di Re-ID, per essere sicuri che la somiglianza visiva tra oggetti della stessa classe o identici non generi ambiguità nella fase di assegnazione degli ID.

Parallelamente alle soluzioni proposte, che sono esclusivamente basate su deep learning, sono anche stati sviluppati degli algoritmi ibridi a efficienza computazionale molto elevata, che si sono dimostrati particolarmente adatti a essere eseguiti in tempo reale su sistemi embedded. Tra questi, si citano i *Kernelized Correlation Filters (KCF)*, algoritmi che sfruttano le proprietà matematiche della *matrice circolante*³ nel dominio delle frequenze, ottenuta tramite la *Fast*

³Una matrice circolante è una matrice quadrata in cui ogni riga è lo scorrimento ciclico della riga precedente. In ambito di tracking, permette di modellare matematicamente tutte le possibili traslazioni spaziali del target.

Fourier Transform, per tracciare i target a frequenze elevatissime. Come dimostrato dagli autori dell'articolo [20], l'algoritmo KCF può essere esteso anche allo spazio tridimensionale - passaggio fondamentale per la manipolazione robotica - utilizzando i dati raccolti da una camera RGB-D: sfruttando la mappa di profondità acquisita dal sensore, il sistema riesce a calcolare la distanza reale dell'oggetto dalla camera, consentendo così di mappare la traiettoria in tempo reale.

Nel caso in cui più oggetti nello spazio operativo presentino caratteristiche fortemente simili o addirittura identiche, i metodi di tracking standard mostrebbero dei limiti, rischiando di non distinguere due oggetti l'uno dall'altro. Per risolvere questa problematica, gli autori dell'articolo [21] propongono l'utilizzo di modelli grafici probabilistici applicati direttamente a dati RGB-D; approccio che consente di effettuare il tracking simultaneo di più oggetti con le stesse caratteristiche, imponendo dei vincoli spaziali espliciti grazie all'ipotesi che due oggetti diversi non possono trovarsi nella stessa posizione e quindi avere le stesse coordinate 3D. Così facendo, è possibile garantire un input robusto per la pianificazione delle traiettorie e per il controllo del robot.

2.4.2 Object tracking nei modelli YOLO

Nei modelli della famiglia YOLO, a partire dalla versione YOLOv8 e in tutte le architetture successive, la pipeline di tracking è integrata all'interno del framework, consentendo di ridurre notevolmente la latenza complessiva ed eseguendo sia detection che tracking in un unico flusso logico.

I modelli YOLO supportano due algoritmi di object tracking, ovvero *ByteTrack* e *BoT-SORT*:

- *ByteTrack*: questo algoritmo ha il merito di risolvere uno dei problemi principali del tracking-by-detection, ovvero la perdita della traccia causata dall'occlusione degli oggetti o dalla sfocatura del movimento. A differenza dei tracker tradizionali, invece di scartare le detection con una percentuale di confidenza bassa, prende tutte le detection e le divide, distinguendo quelle ad alta confidenza, che vengono associate normalmente tramite IoU, e quelle a bassa confidenza, che, invece, vengono confrontate con le predizioni del filtro di Kalman dei tracker esistenti. Questa logica consente di mantenere l'ID dell'oggetto anche nel caso in cui dovesse essere quasi o completamente coperto o sfocato.
- *BoT-SORT*: questo è l'algoritmo più avanzato integrato in YOLO; è in grado di unire i vantaggi dei metodi SORT, DeepSORT e ByteTrack, ottimizzandoli per camere in movimento o manipolatori con configurazione *eye-in-hand*. Nel dettaglio, il vantaggio principale introdotto da BoT-SORT è rappresentato dalla compensazione del movimento della camera, a cui ci si riferisce con *CMC* (*Camera Motion Compensation*), basata su algoritmi tali che, se la telecamera si muove perché montata sulla flangia del robot, sono in grado di correggere la posizione dei bounding box, evitando che il movimento del braccio robotico venga interpretato come un movimento dell'oggetto.

2.4.3 Superare i limiti dell'object tracking

Per quanto si possa cercare di ottimizzare il sistema di object tracking, non è possibile evitare il rumore ad alta frequenza, dovuto in gran parte alle fluttuazioni nei pixel dei sensori e all'approssimazione metrica delle camere di profondità. Nel caso in cui tale segnale rumoroso fosse direttamente utilizzato per controllare il robot, rischierebbe di provocare vibrazioni, scatti improvvisi e usura precoce dei giunti del braccio; per questo motivo, è fondamentale inserire un modulo per la stima dello stato e per il filtraggio dei dati.

Normalmente, gli strumenti più utilizzati sono il classico *filtro di Kalman* e la sua estensione non lineare, cioè il *filtro di Kalman esteso*, che consentono di modellare la cinematica dell'oggetto, ovvero la posizione, la velocità e l'accelerazione, insieme alla covarianza del rumore del sensore. Durante la fase di predizione, l'algoritmo stima dove si troverà l'oggetto al millisecondo successivo; mentre, nella fase successiva, di correzione, confronta la predizione con la nuova misura arrivata dalla camera, calcolando il guadagno per pesare l'incertezza del modello rispetto a quella del sensore. Questo processo permette di smussare i salti di profondità e garantire traiettorie fluide.

Tuttavia, il rumore ambientale a cui sono soggetti i segnali negli ambienti industriali dinamici può variare in modo imprevedibile. Per questo motivo, gli autori dell'articolo [22] propongono l'integrazione di sistemi di controllo visivo assistiti da logica fuzzy, approccio che permette di modificare dinamicamente i parametri di regolazione del loop visivo in base all'errore di stima. Questo sistema compensa le imprecisioni tipiche dell'object tracking e riduce la complessità del tuning dei parametri, garantendo una traiettoria fluida e stabile anche nel caso in cui il target si muovesse in modo repentino o non deterministico.

Infine, come analizzato dagli autori dell'articolo [23], le problematiche che riguardano la stabilità del dato visivo acquisito dalla camera, compromessa da variazioni della luce, ombre o riflessi, possono essere affrontate attraverso l'implementazione di strutture di controllo avanzate, come i controllori PID in cascata: in questa configurazione, il loop esterno, ovvero quello di visione, gestisce l'errore di posizione fornito dall'algoritmo di tracking filtrato, mentre il loop interno, di controllo dei motori, stabilizza la velocità del movimento dei giunti del robot. Unendo il filtraggio a monte e il controllo in cascata a valle gli autori di [23] sono in grado di minimizzare l'errore di inseguimento dinamico, eliminando del tutto il jitter e garantendo che il robot esegua sia l'approccio che il grasping dell'oggetto in maniera fluida, continua e sicura.

2.5 Visual servoing

L'ultima componente di questa tesi è rappresentata dal *visual servoing*, una delle metodologie più consolidate ed efficienti per l'integrazione dei sistemi di computer vision nei contesti di controllo di un manipolatore robotico.

Il visual servoing è basato sul concetto di *visual feedback*⁴, il quale permette

⁴Con "visual feedback" si intende un'architettura a ciclo chiuso in cui la variabile di stato del sistema viene stimata esternamente attraverso l'elaborazione di immagini e in cui l'errore calcolato tra la posa attuale del target e quella desiderata viene utilizzato dal controllore per

di controllare la posa dell'*end-effector* basandosi sull'errore calcolato a partire dalle immagini acquisite dalla camera. Concretamente, sfruttando il concetto dell'object tracking approfondito nella Sezione 2.4, permette al manipolatore di seguire un oggetto target emulandone i movimenti.

Questa sezione analizza i principali approcci proposti nella letteratura per la soluzione del task di visual servoing, ovvero l'*Image-Based Visual Servoing (IBVS)* e il *Position-Based Visual Servoing (PBVS)*, per poi approfondirne le problematiche di implementazione in tempo reale all'interno dell'ecosistema moderno di ROS2 (di cui si parla nella Sezione 4.2).

2.5.1 Image-Based VS Position-Based Visual Servoing

A partire dalla letteratura classica della robotica, in particolare dalle pubblicazioni come [24], i sistemi di visual servoing sono stati divisi in due famiglie:

1. *Image-Based Visual Servoing*: l'errore viene calcolato direttamente nello spazio bidimensionale del piano dell'immagine, espresso in pixel; in seguito, il controllore cerca di far coincidere le feature visive del frame attuale, come punti, linee e baricentri, con un set di feature del frame precedente. Infine, la relazione tra la variazione delle feature nell'immagine e la velocità della camera viene mappata tramite la matrice Jacobiana dell'immagine stessa.

Il vantaggio della famiglia dell'IBVS è rappresentato dalla sua robustezza in caso di errori di calibrazione della camera e del robot, dovuta al fatto che il controllo avviene solamente nel dominio dei pixel.

Dall'altra parte, però, come analizzato anche dagli autori dell'articolo [25], presenta forti limitazioni geometriche. Ad esempio, nel caso di task di assemblaggio industriale assistito da visione, l'IBVS controlla il robot nello spazio bidimensionale dell'immagine, senza però avere informazioni esplicite per quanto riguarda la traiettoria tridimensionale; ciò rende impossibile pianificare un percorso evitando eventuali collisioni del robot con oggetti o persone presenti nell'ambiente operativo, rischiando così impatti pericolosi per il manipolatore. Per affrontare e risolvere tale problematica, è necessario implementare algoritmi di pianificazione proiettiva per garantire la sicurezza.

Un'altra criticità presentata da questa famiglia di visual servoing è il mantenimento delle feature all'interno del campo visivo del sensore: se il target dovesse uscire dall'inquadratura a causa di movimenti inaspettati, il loop di controllo si interromperebbe. Tale problematica diventa particolarmente limitante, ad esempio, in contesti come quelli analizzati nell'articolo [26], ovvero in sistemi soggetti a manovre aggressive come i quadricotteri. In contesti simili, secondo gli autori dell'articolo, per garantire la visibilità del target, l'IBVS deve essere affiancato da framework predittivi non lineari (*Nonlinear Model Predictive Control - NMPC*), che consentono di imporre vincoli espliciti sulla cinematica dell'immagine.

correggere l'azione dei motori.

2. *Position-Based Visual Servoing*: in questo caso, l'errore viene calcolato direttamente nello spazio cartesiano tridimensionale ed espresso rispetto a una terna di riferimento che può essere solidale alla base del robot, alla flangia o al sensore stesso. In questo approccio, le feature estratte dall'immagine vengono elaborate tramite un modello geometrico del target o accoppiate a dati di profondità per stimare le coordinate 3D esplicite dell'oggetto rispetto alla camera; in questo modo, il controllore riesce a ridurre l'errore tra la posa dell'end-effector e quella del target, consentendo una pianificazione della traiettoria nello spazio operativo che sia lineare, intuitiva e prevedibile.

Spesso l'IBVS viene preferito al PBVS per la sua indipendenza dai modelli di calibrazione rigorosi; tuttavia, studi comparativi recenti hanno dimostrato che sempre più spesso è da preferire il PBVS per quanto riguarda le applicazioni industriali ad alta precisione. Ad esempio, gli autori dell'articolo [27] propongono uno studio sperimentale focalizzato sull'assemblaggio robotico di grandi strutture composite, confrontando le due diverse metodologie ed evidenziando come l'approccio PBVS, rispetto all'IBVS, si dimostri notevolmente più rapido e leggermente più accurato. Secondo l'articolo [27], questo succede perché l'estrazione del target sul piano bidimensionale dipende comunque dall'accuratezza del modello della camera, annullando di fatto il vantaggio teorico di robustezza del metodo IBVS e premiando invece la stabilità del PBVS.

Nell'articolo [28] viene presentata un'applicazione avanzata ed efficiente del metodo PBVS, nel quale viene implementato un sistema di controllo visivo per la manipolazione aerea utilizzando un sistema a doppio braccio robotico montato su un esacottero. Gli autori dell'articolo sfruttano una camera di profondità Intel RealSense D435i insieme a un modello YOLO per la detection per estrarre contemporaneamente le informazioni sui pixel e i dati di profondità degli oggetti da manipolare. Attraverso delle trasformazioni di coordinate, il sistema è in grado di mappare in real-time le coordinate dell'oggetto nello spazio 3D per controllare i bracci robotici e guidarli verso l'oggetto, implementando anche strategie di *self-balancing*⁵ per mitigare i disturbi dinamici sul sistema.

2.6 Tecniche di denoising per sensori RGB-D

In un sistema di controllo basato sulla posizione (PBVS) le componenti di efficacia e accuratezza dipendono strettamente dalla qualità dei dati tridimensionali estratti dai sensori di visione. Tuttavia, le camere a tecnologia RGB-D (Sezione 2.2) sono soggette a numerose fonti di rumore di diversa natura, da cui non possono prescindere; rumore che causa fluttuazioni nei valori di profondità, pixel mancanti e incertezze geometriche in corrispondenza delle discontinuità di pendenza o dei bordi degli oggetti. In letteratura, il processo tramite cui si cerca di ridurre tale rumore al minimo, il *denoising*, viene diviso in tipologie di approcci:

⁵Nei sistemi di manipolazione aerea, il movimento dei bracci robotici modifica di continuo il centro di massa e genera forze e coppie di reazione che disturbano il velivolo. La strategia di *self-balancing* indica un controllo dinamico in grado compensare queste perturbazioni, regolando la spinta dei motori per riuscire a mantenere il sistema in volo stabile.

filtri analitici guidati, reti neurali convoluzionali e accelerazione hardware per il rispetto dei vincoli real-time.

Per fare in modo di preservare la nitidezza dei confini e dei bordi degli oggetti ed evitare la sfocatura tipica dei filtri gaussiani classici, la ricerca si è orientata verso i filtri guidati dal colore, i *Color-Guided Filters*, in cui l'immagine RGB viene utilizzata come guida per ripulire la mappa di profondità (D). Come evidenziato dagli autori dell'articolo [29], i metodi tradizionali di analisi delle immagini RGB-D spesso devono affrontare il problema del *texture coping*, ovvero il fenomeno per cui variazioni fotometriche o di colore presenti nell'immagine RGB, come texture superficiali, scritte o ombre, vengono erroneamente interpretate dall'algoritmo come discontinuità geometriche e quindi trasferite nella mappa di profondità, generando così dei falsi rilievi tridimensionali su superfici che, nella realtà, sono piane. Nell'articolo [29] si propone di risolvere questo problema tramite un framework ottimizzato (illustrato nella Figura 2.7) basato su un modello di *depth-driven color flattening*, combinato con un modello *Markov Random Field - MRF*; architettura che permette di limitare fortemente il rumore nelle immagini, preservando le discontinuità strutturali dell'oggetto e aumentando l'accuratezza di un sistema di tipo PBVS.

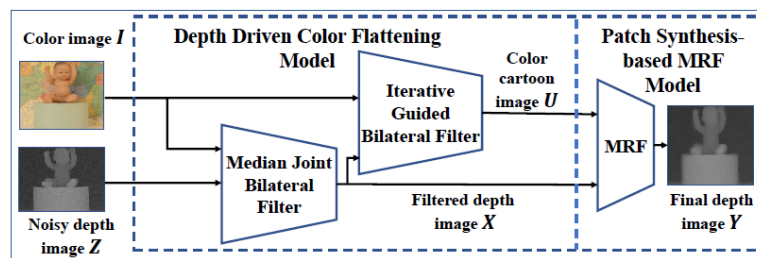


Figura 2.7: Framework proposto nell'articolo [29]

Parallelamente agli approcci analitici, il deep learning ha introdotto soluzioni in grado di mappare in modo non lineare le relazioni tra texture e profondità. In questo contesto, gli autori dell'articolo [30] propongono una rete convoluzionale leggera, *Light CNN*, mostrata nella Figura 2.8, composta solamente da tre strati: proiezione ad alta dimensione, completamento dei dati mancanti e ricostruzione dell'immagine. Tale modello elabora contemporaneamente l'immagine in scala di grigi e la mappa di profondità rumorosa, e il suo addestramento viene ottimizzato utilizzando strategie di data augmentation e una loss function pesata. Il lavoro dimostra come sia possibile completare i dati mancanti e rimuovere il rumore, preservando al contempo la reattività del sistema e garantendo una velocità di elaborazione che risulta compatibile con il paradigma di controllo real-time.

Tuttavia, implementare algoritmi di denoising complessi all'interno del loop di controllo di un manipolatore introduce inevitabilmente un problema di overhead, che causa a sua volta una latenza non trascurabile nel sistema. Per questo motivo, attualmente la ricerca si concentra sull'ottimizzazione architetturale a basso livello, producendo risultati come quelli raggiunti dagli autori dell'articolo [31]. Questo studio introduce un acceleratore hardware ad alta efficienza

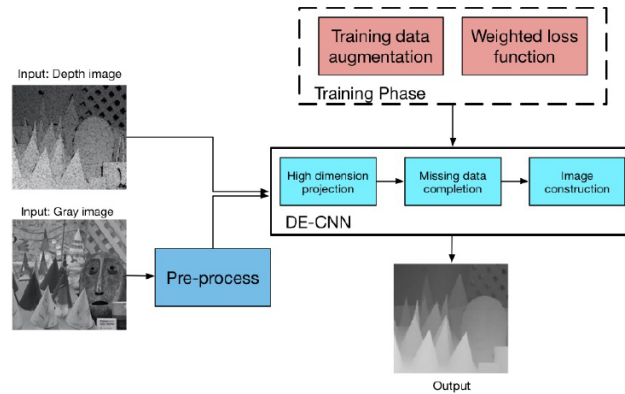


Figura 2.8: Light CNN proposta nell'articolo [30]

energetica basato su una versione ottimizzata della rete DnCNN, *Light-DnCNN*: sfruttando la *Depth-wise Separable Convolution* e una *Fused-Layer Architecture*, dimostra la possibilità di eseguire il denoising di immagini corrotte da rumore gaussiano bianco a frequenze elevate, mantenendo al contempo un ridotto consumo di potenza.

Combinare algoritmi di filtraggio robusti o reti neurali snelle con soluzioni efficienti è fondamentale nell'ambito della robotica collaborativa guidata da visione, e assicura sia che il segnale di feedback fornito al controllore non contenga fluttuazioni sia che i tempi di calcolo complessivi del ciclo di controllo non vengano alterati, riuscendo a garantire il rispetto dei requisiti real-time.

2.7 Sintesi dello stato dell'arte

L'analisi della letteratura scientifica degli ultimi anni portata avanti in questo capitolo ha mostrato che riuscire a unire in maniera efficace computer vision e controllo robotico dipende fortemente da un *trade-off* tra accuratezza, stabilità di controllo e sostenibilità computazionale in tempo reale.

Se da un lato l'avvento del deep learning e di modelli moderni come YOLO ha rivoluzionato l'object detection offrendo una robustezza mai riscontrata prima per quanto riguarda l'identificazione degli oggetti target, dall'altro lato, utilizzare solamente l'estrazione del bounding box di tale oggetto non basta per la manipolazione in ambienti industriali dinamici. Quindi, vengono sfruttati algoritmi ibridi come SORT, basati sul filtro di Kalman e l'algoritmo Ungherese, o soluzioni a filtri di correlazione come i KCF [20], i quali risolvono la problematica della generazione di traiettorie ad altissima frequenza, nonostante non risolvano del tutto il problema di stabilità nel caso di rumore introdotto da luci, ombre o riflessi [23].

In secondo luogo, nel dominio del visual servoing, è fondamentale scegliere la soluzione migliore per il contesto in cui si opera tra IBVS e PBVS: come analizzato negli articoli [25] e [26], nonostante l'IBVS sia tollerante a eventuali errori nella calibrazione, soffre di limitazioni geometriche nello spazio 3D, impe-

dendo una pianificazione *collision-free*. Al contrario, gli studi dell'articolo [27] dimostrano che, in contesti caratterizzati da alta precisione, l'approccio PBVS non solo supera l'IBVS in termini sia di accuratezza sia di rapidità, ma ha anche ottime prestazioni se implementato su sistemi embedded e affiancato da sensori RGB-D, come dimostrato dall'architettura proposta nell'articolo [28].

Tuttavia, affidarsi a tali sensori RGB-D introduce la problematica del rumore presente nei dati di profondità, soprattutto in corrispondenza dei bordi dei target. Per affrontare e superare questo limite, ci sono studi che propongono l'utilizzo di filtri analitici guidati dal colore per preservare le discontinuità geometriche, limitando anche il problema del texture coping, come mostrato nell'articolo [29]; altri studi, invece, propongono l'utilizzo di reti neurali convoluzionali leggere, come lo studio dell'articolo [30]. Nonostante ciò, l'utilizzo di algoritmi di denoising complessi introduce un overhead non trascurabile; quindi, la ricerca si sta focalizzando sulla progettazione di acceleratori hardware dedicati ad alta efficienza energetica, come nell'articolo [31], in grado di garantire risposte a frequenze elevate.

2.8 Posizionamento della tesi

Nonostante i molteplici vantaggi che comporta l'utilizzo di robot collaborativi (Sezione 2.1), nella maggior parte delle applicazioni industriali un cobot esegue traiettorie pre-programmate, attuando le misure di sicurezza solamente una volta che l'impatto, anche se di lieve intensità, è già avvenuto. Per questo motivo, per il progetto sviluppato in questa tesi è stato integrato nel sistema del robot un sensore di visione, in maniera tale che il manipolatore sia consapevole dell'ambiente di lavoro in cui si muove garantendo non solo la sicurezza fisica dell'operatore, ma anche la risposta al variare dell'ambiente stesso, come nel caso di oggetti in movimento.

Nel contesto del panorama scientifico illustrato in questo capitolo, l'architettura sviluppata in questo lavoro di tesi si posiziona all'interno della metodologia del *Position-Based Visual Servoing (PBVS)*, proponendo una soluzione integrata e ottimizzata per cercare di superare sia il problema della latenza sia il problema del rumore dei dati RGB-D.

Sfruttando i canali RGB-D forniti dal sensore Intel RealSense installato utilizzando la configurazione *eye-in-hand*, il sistema calcola l'errore di posizionamento dell'end-effector direttamente nello spazio tridimensionale, cercando di mitigare l'incertezza tipica di tali sensori ed evitando il ricorso a reti di denoising o ad acceleratori hardware dedicati.

Invece, per quanto riguarda la detection e il tracking di oggetti, l'idea di questa tesi è:

- costruire un dataset personalizzato;
- selezionare un modello di YOLO pre-addestrato;
- effettuare il fine-tuning utilizzando il dataset personalizzato, in maniera tale da specializzare il modello a operare nel contesto del lavoro e aumentarne l'accuratezza;

- sfruttare il paradigma di object tracking integrato in YOLO.

Tutto ciò con lo scopo di disaccoppiare l'estrazione della feature spaziali del target dalla logica di avvicinamento del braccio.

Inoltre, la gestione di tutto l'ecosistema avviene attraverso *ROS2* (Sezione [4.2](#)), affrontando la sfida rappresentata dalle diverse frequenze a cui lavorano l'elemento di visione e il controllore del robot.

Capitolo 3

Architettura hardware

Nel presente capitolo verrà analizzata l'architettura hardware impiegata per il progetto sviluppato in questa tesi, focalizzandosi dapprima sul sistema del robot e successivamente sul sistema di visione.

Il sistema del manipolatore utilizzato per effettuare la presa degli oggetti e per seguirne i movimenti comprende un robot e una pinza. In questo capitolo verranno descritti nel dettaglio entrambi i componenti che fanno parte di tale sistema; in particolare, il robot utilizzato è un robot collaborativo della casa produttrice *Universal Robots*, mentre la pinza è una pinza ad aria compressa della casa produttrice *Schunk*.

Invece, al fine di soddisfare i requisiti del progetto, per il sistema di visione si è scelto di utilizzare una camera RGB-D, in grado di garantire un compromesso ottimale tra accuratezza, costi, ingombro e peso complessivo: era necessaria una camera che consentisse il tracciamento real-time degli oggetti e che al contempo non risultasse pesante.

3.1 Robot collaborativi

I robot collaborativi, o cobot, sono dei robot progettati per condividere lo spazio di lavoro con l'operatore umano per affiancarlo in lavori molto ripetitivi o pesanti.

L'introduzione dei cobot nei contesti produttivi dell'Industria 5.0 porta con sé la necessità di garantire la sicurezza del sistema, vista la mancanza di barriere protettive tra l'operatore umano e il robot. Per questo motivo, l'integrazione di tali sistemi è regolata da standard internazionali: normative che definiscono le modalità operative di cooperazione, requisiti minimi di progettazione e limiti cinematici e dinamici necessari a prevenire e ridurre gli infortuni degli operatori.

Attualmente, questo tipo di robot viene utilizzato in molteplici settori, a partire da quello manifatturiero e industriale, dove automatizzano compiti di assemblaggio e controllo qualità, a quello della logistica, fino ad arrivare al settore medico e sanitario, dove vengono utilizzati, ad esempio, nel supporto alla chirurgia e nelle attività di riabilitazione.

3.1.1 UR10e

Il robot utilizzato per questo progetto è un robot collaborativo della *Universal Robots*, nello specifico, il modello è l'*UR10e*, illustrato nella Figura 3.1. Tra le specifiche base del robot si segnalano uno sbraccio di $1300mm$, un payload massimo di $12.5kg$ e un peso totale di $33.5kg$; per maggiori informazioni e approfondimenti tecnici riguardo questo modello si rimanda al datasheet ufficiale [32], al manuale utente [33] e al sito della casa produttrice [34].

Il sistema del robot è formato da tre macro-componenti principali:

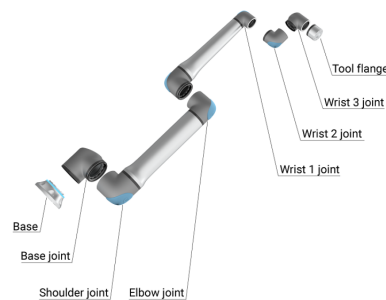
1. *braccio robotico*: illustrato nella Figura 3.1a, è la struttura meccanica vera e propria del sistema. Si tratta di un manipolatore antropomorfo a sei gradi di libertà movimentato da sei giunti rotanti, ognuno con un'ampiezza di lavoro pari a $\pm 360^\circ$. Come mostrato nella Figura 3.1b, tali giunti sono:

- giunto di base;
- giunto di spalla;
- giunto di gomito;
- giunto di polso 1;
- giunto di polso 2;
- giunto di polso 3.

All'estremità del braccio, in corrispondenza del giunto di polso 3, è presente la *flangia meccanica*, che permette l'interfacciamento con i tool di presa, gli *end-effector*.



(a) Braccio robotico (immagine tratta da [32])



(b) Giunti del robot (immagine tratta da [33])

Figura 3.1: Robot collaborativo UR10e

2. *unità di controllo* o *control box*: illustrata nella Figura 3.2, è il nucleo dell'intero sistema, costituisce il controllore del robot. Si occupa di eseguire gli algoritmi di cinematica diretta e inversa per pianificare le traiettorie dei giunti; inoltre, gestisce gli input e gli output, permettendo al robot di interagire con periferiche esterne come sensori e attuatori.

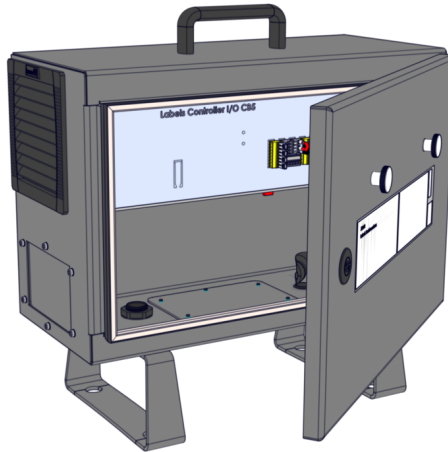


Figura 3.2: Unità di controllo del robot UR10e (immagine tratta da [9])

3. *teach pendant*: illustrato nella Figura 3.3, è l'interfaccia uomo-macchina utilizzata dall'operatore. Consiste in uno schermo touch screen di 12" grazie al quale è possibile programmare e controllare il braccio robotico; inoltre, è provvisto di un pulsante di accensione, un pulsante per l'arresto di emergenza, una porta USB e un pulsante di freedrive, che permette di spostare manualmente il robot nella posizione desiderata.



Figura 3.3: Teach pendant del robot UR10e (immagine tratta da [9])

L'architettura standard prevede che il *braccio robotico* e il *teach pendant* siano coordinati e governati direttamente dall'*unità di controllo* e che l'intero sistema possa essere collegato con unità di calcolo esterne tramite interfaccia Ethernet, sfruttando il protocollo *TCP/IP*.

Tuttavia, nel contesto di questa tesi, l'unità di controllo è stata superata per implementare un'architettura di controllo tramite nodi ROS, che consente di trasmettere le posizioni desiderate per i giunti del robot direttamente dal software, sfruttando il protocollo TCP/IP. Di conseguenza, anche l'ambiente di programmazione *PolyScope*, integrato nel *teach pendant* e normalmente re-

sponsabile della programmazione del robot e della gestione dei task, è stato sostituito da una logica di orchestrazione dei task sviluppata interamente in codice Python, coerentemente con il framework ROS adottato.

Infine, in quanto robot collaborativo, l'UR10e è dotato di funzionalità di sicurezza conformi alle normative internazionali che monitorano costantemente lo stato cinematico e dinamico del robot, consentendo la condivisione dello spazio di lavoro tra uomo e macchina senza bisogno barriere di sicurezza. Le principali funzioni di sicurezza sono:

- limitazioni di forza, potenza e momento angolare del robot;
- limitazioni cinematiche di posizione e velocità, configurabili in modo indipendente per ciascun giunto;
- stop di emergenza, azionabile sia dal pulsante presente sul teach pendant sia da un dispositivo esterno.

3.2 Pinza di presa

Al fine di afferrare e manipolare gli oggetti posizionati sul piano di lavoro, è fondamentale l'integrazione di un opportuno tool di presa. Tali dispositivi possono essere di diversa natura in base ai requisiti applicativi del progetto, spaziando dai più semplici, come le pinze ad aspirazione, ai più complessi, come le mani robotiche.

Nel contesto di questo lavoro, si è scelto di utilizzare una pinza pneumatica prodotta da *Schunk*, nello specifico il modello *Schunk MPC 150*, raffigurato in Figura 3.4.



Figura 3.4: Schunk MPC 150 (immagine tratta da [35])

Alcune delle caratteristiche principali della pinza sono:

- lunghezza: $12.3cm$;
- larghezza: $6.4cm$;
- altezza: $4.4cm$;
- peso: $0.94kg$;

- tempo di reazione per l'apertura: 0.11s;
- tempo di reazione per la chiusura: 0.11s.

Per ulteriori dettagli tecnici si rimanda al sito ufficiale della Schunk [35].

Per poter afferrare degli oggetti, la pinza deve essere dotata di una coppia di griffe, ovvero gli elementi che entrano direttamente in contatto con l'oggetto target. Per questo progetto sono state progettate e realizzate delle griffe personalizzate tramite *Sintetizzazione Laser Selettiva, SLS*, tecnologia basata sulla sintetizzazione di particelle di un polimero, in questo caso è stato utilizzato il nylon, e sulla realizzazione di una struttura solida a partire da un modello 3D dell'oggetto da realizzare. Nella Figura 3.5 si può vedere una rappresentazione della pinza utilizzata con le griffe personalizzate.

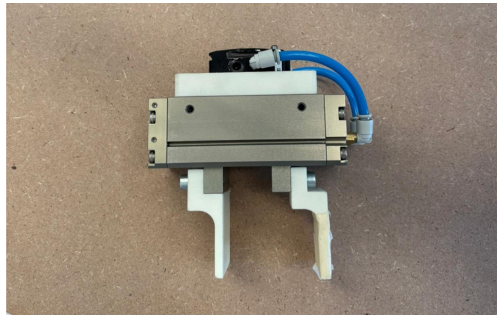


Figura 3.5: Pinza Schunk con griffe personalizzate

Infine, il collegamento tra la pinza e la flangia del robot avviene tramite una giunzione intermedia, il *cambio tool* della Schunk, illustrato nella Figura 3.6.



Figura 3.6: Cambio tool Schunk (immagine tratta da [36])

3.2.1 Circuito ad aria compressa

Essendo la pinza di natura pneumatica, è stato necessario progettare e integrare un circuito ad aria compressa dedicato, i cui componenti principali sono:

- un sistema per la *regolazione dell'aria compressa* costituito da un regolatore pneumatico AR20-F02-B e da un relè a pressione IS10M-20-6L-A, entrambi della casa produttrice SMC, Figura 3.7, per garantire il mantenimento della pressione ottimale e il monitoraggio di eventuali cali di pressione nella linea;



Figura 3.7: Sistema di regolazione dell'aria (immagine tratta da [9])

- un' *elettrovalvola monostabile* VP342-5Y01-02FA della casa produttrice SMC, Figura 3.8, responsabile delle logiche di abilitazione e disabilitazione della morsa di presa;

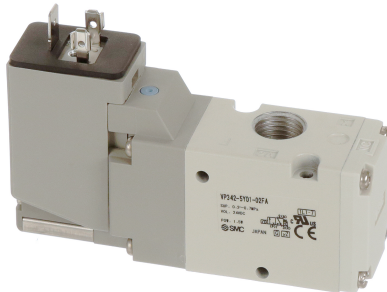


Figura 3.8: Elettrovalvola monostabile (immagine tratta da [37])

- due *elettrovalvole bistabili* SY5220-5Y0-01F-0 della casa produttrice SMC, Figura 3.9, di cui la prima è configurata per gestire le operazioni di bloccaggio e sbloccaggio (*lock/unlock*) del sistema di cambio tool sul polso del robot, mentre la seconda è dedicata al controllo per l'apertura e la chiusura delle griffe della pinza.

La Figura 3.10 rappresenta il circuito cablato finale, mostrando i collegamenti tra i vari componenti.



Figura 3.9: Elettrovalvola bistabile (immagine tratta da [38])

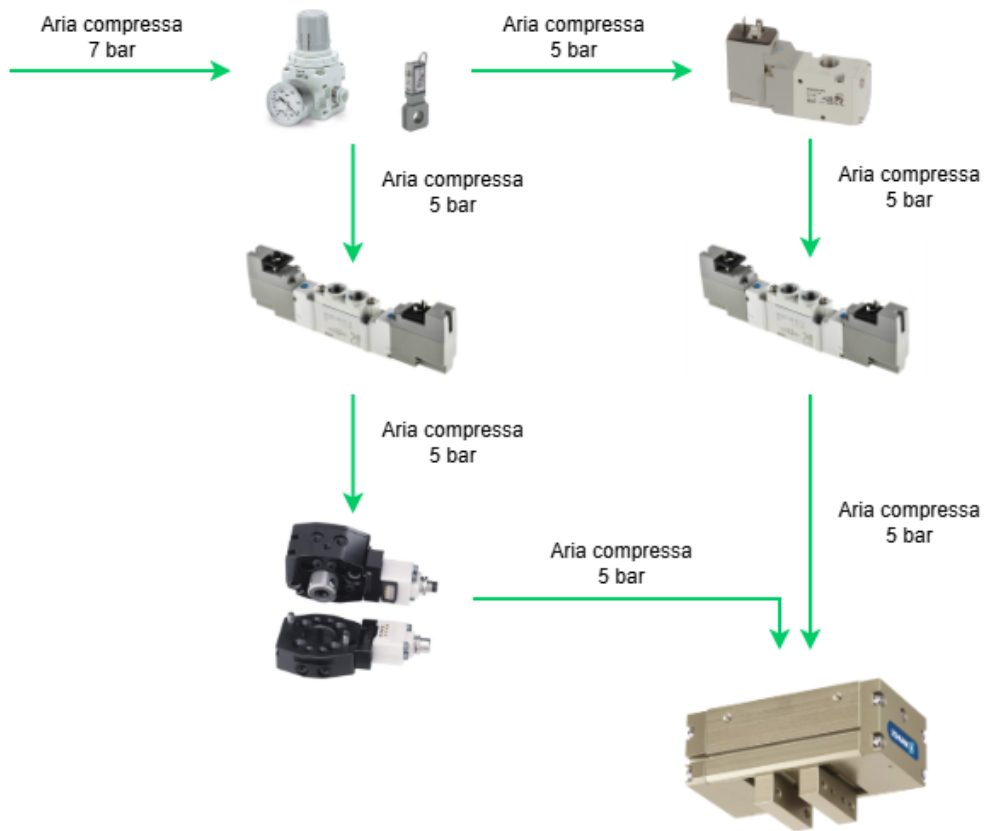


Figura 3.10: Circuito ad aria compressa

3.3 Intel RealSense D435

La camera utilizzata per il sistema di visione di questo progetto è la *Intel RealSense D435*, illustrata nella Figura 3.11, un sensore di profondità che sfrutta il principio della visione stereoscopica per la ricostruzione tridimensionale dello spazio, il cui funzionamento è spiegato nella Sezione 2.2.

In quanto camera stereoscopica, come si può vedere nell'immagine 3.12, la RealSense D435 è composta da quattro moduli fondamentali allineati sullo stesso asse: una camera RGB, “*RGB Module*”, un proiettore di pattern a infrarossi,



Figura 3.11: Intel RealSense D435 (immagine tratta da [39])

“*IR Projector*”, e due sensori a infrarossi posizionati alle estremità, denominati rispettivamente “*Left Imager*” e “*Right Imager*”, che si trovano a due angolazioni differenti e permettono la ricostituzione dell’immagine tridimensionale nello spazio.

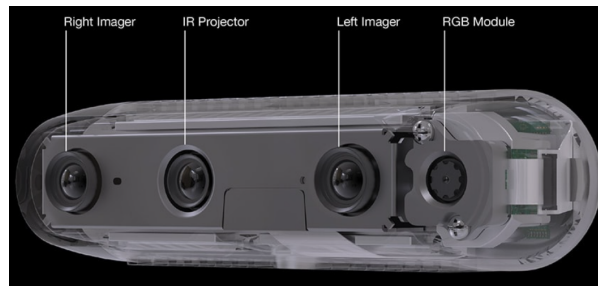


Figura 3.12: Moduli della Intel Realsense D435 (immagine tratta da [39])

Di questi, i moduli *Left Imager* e *Right Imager* sono dei sensori basati sulla tecnologia *global shutter*, che rappresenta la modalità con cui viene effettuato lo scatto per la cattura della scena. Tale metodologia è ideale per contesti di lavoro real-time, poichè, a differenza dei sensori a *rolling shutter*, che scansionano la scena riga per riga causando degli sfasamenti temporali, implica l’acquisizione simultanea di tutti i pixel, annullando quasi del tutto fenomeni come *motion blur* causati dal movimento rapido del manipolatore o dell’oggetto target.

Nella Figura 3.13 è possibile osservare la configurazione con cui è stata utilizzata la camera nel contesto di questa tesi, ovvero *eye-in-hand*, montata tramite un tool personalizzato sul giunto finale del robot per fare in modo che si muova solidalmente alla pinza di presa.

Alcune delle specifiche base di questo modello di camera, descritte nel dettaglio nel sito ufficiale [39] e nel datasheet della serie del modello [40], includono:

- dimensioni di $90 \times 25 \times 25$ mm;
- range operativo compreso tra $\sim 0.3m$ - $\sim 3m$;
- risoluzione massima dell’immagine RGB pari a 1920×1080 pixel;
- risoluzione massima dell’immagine di profondità pari a 1280×720 pixel;
- campo visivo (FOV) di $87^\circ \times 58^\circ$;

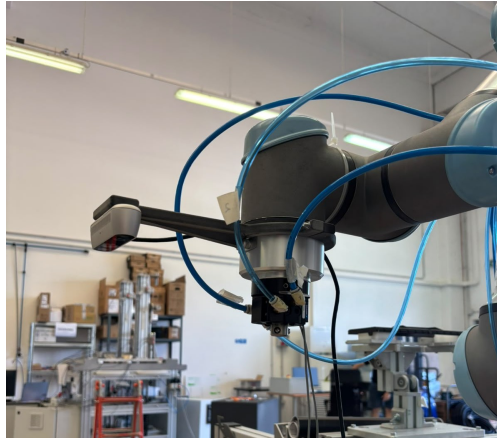


Figura 3.13: Configurazione della RealSense D435 sul polso del robot

- frame rate di profondità pari a 90 FPS;
- frame rate RGB pari a 30 FPS;

Grazie alle sue caratteristiche, la RealSense D435 permette di ottenere un ottimo compromesso tra accuratezza, compattezza hardware e sostenibilità computazionale, proprietà che la rende ideale per l'applicazione nel contesto di controllo robotico real-time.

3.3.1 Intel RealSense SDK 2.0

Intel RealSense SDK 2.0 è una libreria cross-platform utilizzata per la gestione e l'elaborazione dei flussi di dati della camera; funge da *middleware* tra il sensore e l'architettura di alto livello di ROS2 e consente il controllo effettivo della camera e di tutte le tipologie di stream e funzionalità supportate dal sensore. Per maggiori dettagli sull'installazione si rimanda alla Sezione [4.3.3](#).

Questa libreria mette a disposizione diversi tool, in particolare:

- *Intel RealSense Viewer*: applicazione che consente di interfacciarsi facilmente e rapidamente con la camera allo scopo di utilizzare tutte le funzioni che si hanno a disposizione. Nell'immagine [3.14](#) si mostra l'interfaccia principale dell'applicazione, all'interno della quale sono abilitate sia la camera di profondità, a sinistra, sia la camera RGB, a destra. Inoltre, dal menù a tendina presente sulla sinistra dell'immagine è possibile modificare le impostazioni per entrambe le tipologie di camere;
- *Depth Quality Tool*: applicazione che permette di testare la qualità dell'immagine di profondità, di valutare alcune delle metriche che descrivono la bontà dell'acquisizione e di registrare e salvare dati per analisi offline;
- *debug tools*: tool che permettono di analizzare informazioni e log di dati provenienti dalla camera;

- *esempi di codice*: esempi di codice forniti da poter eseguire per testare le capacità della camera e conoscere gli strumenti che possono essere utilizzati;
- *wrappers*: API fornite per differenti linguaggi di programmazione che permettono l'integrazione con tecnologie e software di terze parti, ad esempio ROS, LabVIEW, OpenCV, Matlab e altri.

In particolare, all'interno del lavoro sviluppato in questa tesi l'SDK svolge una funzione fondamentale, ovvero quella dell'*allineamento spaziale*. La camera RGB e i sensori per il calcolo della profondità occupano posizioni diverse sul sensore, quindi, i rispettivi sistemi di riferimento risultano traslati l'uno rispetto all'altro. L'SDK utilizza un algoritmo di allineamento spaziale, proiettando pixel-per-pixel la mappa di profondità sul piano bidimensionale dell'immagine RGB e assicurando che a ogni coordinata del pixel dell'oggetto individuato dalla rete neurale corrisponda il corretto valore di profondità.

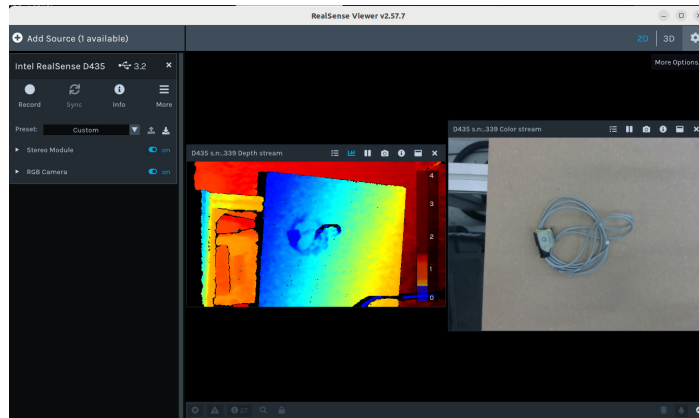


Figura 3.14: RealSense Viewer

Capitolo 4

Architettura software

In questo capitolo viene descritta dettagliatamente l'architettura software implementata per il controllo del braccio robotico e per l'integrazione del sistema di visione. Inoltre, vengono approfondite le fasi di inizializzazione dell'ambiente di sviluppo, l'installazione dei driver e delle librerie e, in particolar modo, la risoluzione delle criticità affrontate durante le varie fasi.

L'intera architettura software dell'applicazione è stata sviluppata sfruttando come base il sistema operativo *Linux Ubuntu*, in particolare la versione *22.04.5*, installata su un PC *Dell Inspiron 5570*, equipaggiato con un processore *Intel Core i7-8550U*, con architettura quad-core a 64 bit, 32 GB di RAM, una GPU integrata *Intel UHD Graphics 620* e una GPU dedicata *AMD Radeon*.

Nella Figura 4.1 vengono mostrati la composizione dell'architettura realizzata, l'interazione dei vari componenti e come viaggiano il flusso di dati della camera e il flusso di controllo del robot.

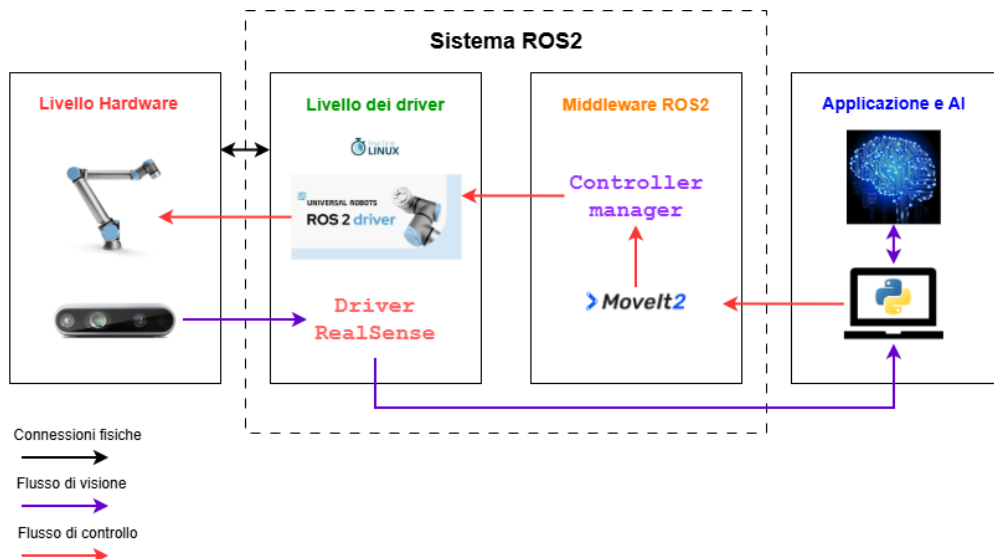


Figura 4.1: Architettura del sistema

In particolare, la Figura 4.1 mostra che il sistema è composto da quattro livelli principali:

1. *hardware*: comprende il robot manipolatore e la camera RealSense;
2. *driver*: garantisce la comunicazione tra i dispositivi fisici e l'ecosistema ROS 2;
3. *middleware ROS 2*: comprende il *Controller Manager* e *MoveIt 2*, responsabili rispettivamente della gestione dei controllori e della pianificazione dei movimenti del manipolatore;
4. *applicazione*: sviluppata in Python, integra algoritmi per l'elaborazione dei dati acquisiti dalla camera e per generare i comandi operativi per il robot.

Nella figura vengono anche distinti in base al colore, rispettivamente viola e rosso, il flusso dei dati acquisiti dalla camera e il flusso dei comandi di controllo del robot.

4.1 Kernel real-time

Nel contesto del controllo robotico industriale, la comunicazione tra l'unità di calcolo e il controllore del manipolatore è caratterizzata da vincoli temporali rigidi, per soddisfare i quali, è necessario sostituire il kernel Linux standard con Ubuntu con un *kernel real-time*.

Un kernel real-time è focalizzato sul *determinismo temporale*, che non implica una maggiore velocità di calcolo assoluta, ma fa in modo che l'esecuzione di un certo task o la risposta a un evento asincrono avvenga in un intervallo di tempo massimo, evitando così la variabilità e l'incertezza sui tempi di risposta che caratterizzano un sistema operativo standard.

Per questo motivo, è fondamentale utilizzare un kernel real-time nei contesti in cui si lavora con un braccio robotico come l'UR10e utilizzato in questo lavoro di tesi: il controller del robot scambia flussi di dati e accetta comandi di traiettoria operando a una frequenza fissa e costante di 500 Hz, ovvero con un ciclo di controllo chiuso di 2 ms.

Nel caso di utilizzo di un kernel standard, la latenza introdotta da altri processi concorrenti, come l'algoritmo di visione o aggiornamenti di background del sistema operativo, rischia di causare la perdita di uno o più pacchetti nella finestra temporale stabilita di 2 ms, situazione che viene interpretata dal controllore del robot come un'anomalia di sincronizzazione software e che, per motivi di sicurezza, attiva una procedura di blocco d'emergenza. Queste interruzioni improvvise, ovviamente, rischiano di causare delle sollecitazioni meccaniche sui giunti del robot che usurano i componenti stessi del braccio.

Nelle prime fasi di sviluppo del progetto, si è tentato di utilizzare una macchina virtuale, ma i test sperimentali hanno dimostrato che l'hypervisor non è compatibile con i requisiti del robot, non riuscendo a reggere la frequenza di trasmissione di 500 Hz.

4.1.1 Procedura di compilazione

Per installare un kernel real-time sul PC utilizzato è stata seguita la pipeline ufficiale di setup di Universal Robots [41]. Al fine di preservare la stabilità operativa complessiva, è stata selezionata una versione di kernel che fosse vicina alla versione 6.8.0 del kernel di Ubuntu 22.04.5, ovvero la 6.6.133, accoppiata alla patch real-time *patch-6.6.133-rt73.patch.xz*.

Durante questo procedimento sono state incontrate due criticità.

La prima durante la fase di compilazione: è stato necessario modificare il comando della build, che nel tutorial era:

```
$ make -j `getconf _NPROCESSORS_ONLN` deb-pkg
```

perché, a partire dalla versione 6.3 del kernel, il comando tradizionale per la generazione dei pacchetti Debian, ovvero “*make deb-pkg*”, richiede che i codici sorgente si trovino in un repository Git locale, poiché, per compilare l’indice dei sorgenti, l’utility è basata sul comando “*git archive*”. Per questo motivo, la compilazione del kernel è stata lanciata tramite il comando:

```
$ make -j `getconf _NPROCESSORS_ONLN` bindeb-pkg
```

Anche la seconda criticità è avvenuta nella stessa fase. In particolare, si è verificato l’errore:

```
SSL error:FFFFFFFF80000002:system library::No such  
file or directory
```

Questo errore si manifestava poiché la configurazione di default richiede che ogni modulo compilato (.ko) abbia la firma digitale, ma se il sistema non trova questa chiave perché il percorso nel file *.config* è sbagliato o punta a chiavi che esistono solo sui server di Debian, la compilazione si blocca con l’errore “*SSL error*”. Per risolvere questo problema, è stato necessario accedere all’interfaccia di configurazione con il comando:

```
$ make menuconfig
```

navigare nel percorso “*Enable loadable module support* → *Module signature verification*” e disattivare la firma automatica togliendo la spunta alla variabile “*CONFIG_MODULE_SIG_ALL*”, consentendo così il completamento della build.

Infine, è stato modificato il file “*/etc/security/limits.conf*” integrando le seguenti direttive per il gruppo *@realtime*:

```
@realtime soft rtprio 99  
@realtime hard rtprio 99  
@realtime soft memlock unlimited  
@realtime hard memlock unlimited
```

La rimozione dei limiti sulla voce “*memlock*”, ovvero impostarla a “*unlimited*”, vieta al kernel di spostare le pagine di memoria RAM allocate dai driver del robot all’interno dello spazio di *swap* su disco fisso, evitando così ritardi imprevisti di I/O.

4.2 ROS 2 Humble Hawksbill

Il nucleo dell'applicazione e la sua infrastruttura di comunicazione sono basati sul framework *ROS* (*Robot Operating System*), costituito da un set di librerie software e di tool ottimizzati e pensati per lo sviluppo di applicazioni con robot e sistemi hardware con i quali può interfacciarsi. Nello specifico, è stata utilizzata la versione *ROS 2 Humble Hawksbill*, che introduce un'architettura *middleware* che garantisce uno scambio dati a bassa latenza tra i diversi processi presenti nel sistema, chiamati *node*. Nella Figura 4.2 è mostrato il logo di ROS 2 Humble Hawksbill.



Figura 4.2: ROS 2 Humble Hawksbill (immagine tratta da [42])

Questa versione del framework è stata scelta sulla base del sistema operativo utilizzato nell'applicazione; questo perché a ogni versione di sistema operativo Linux corrisponde una diversa distribuzione ROS, sviluppata per far sì che non si incorra in bug o problematiche derivanti da eventuali incompatibilità.

Per l'installazione di ROS 2 Humble Hawksbill è stata seguita la pipeline mostrata nel sito ufficiale [43], secondo cui, per l'installazione vera e propria, bisogna eseguire il seguente comando sul terminale di Linux:

```
$ sudo apt install ros-humble-desktop
```

Inoltre, è buona norma creare in maniera permanente l'ambiente ROS 2, in modo tale da poter eseguire liberamente i pacchetti necessari senza dover ogni volta ricrearlo. Per fare ciò, bisogna aprire il file *bashrc* da terminale digitando il seguente comando:

```
$ sudo nano ~/.bashrc
```

e inserire in fondo al file il comando per la creazione permanente dell'ambiente ROS 2:

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

Per sviluppare, integrare e compilare permanentemente e correttamente i pacchetti software e i driver, è stato configurato uno spazio di lavoro dedicato, chiamato *workspace di ROS 2*, seguendo le istruzioni del tutorial dedicato [44].

Secondo tali linee guida, prima di passare alla creazione dell'effettiva cartella del workspace, è necessario installare dei tool essenziali, tra cui *rosdep*, l'utilità per la risoluzione automatica delle dipendenze di sistema, e le estensioni necessarie per l'automazione della build.

Una volta fatto questo, si può passare alla generazione della cartella del workspace e della sotto-cartella dei sorgenti, “*src*”, il cui compito è di contenere i repository del progetto. I comandi per la creazione di tali cartelle sono:

```
$ mkdir -p ~/ros2_ws/src
$ cd ~/ros2_ws/src
```

4.3 Installazione di driver e librerie per ROS 2

Una volta creato il workspace di ROS 2 e configurato l'ambiente operativo base, è stato possibile passare all'installazione nella cartella *src* dei pacchetti software, dei driver e delle librerie necessarie per il progetto.

4.3.1 Driver del manipolatore

Per consentire la comunicazione, il monitoraggio dello stato cinematico e l'invio dei comandi al robot dall'interno dell'ambiente ROS 2, è stato integrato il pacchetto ufficiale *Universal Robots ROS 2 Driver*, che fa da ponte tra ROS 2 e il controller del robot.

Per installare questo driver è stato seguito il tutorial ufficiale [45] per ROS 2 Humble.

Per prima cosa è necessario installare il driver effettivo con il comando:

```
$ sudo apt-get install ros-humble-ur
```

In seguito, è necessario installare l'UR Cap per il controllo remoto del robot, come specificato dal tutorial ufficiale [46]. Bisogna quindi scaricare il file dalla pagina [47] e inserirlo all'interno del teach pendant del robot tramite una chiavetta USB.

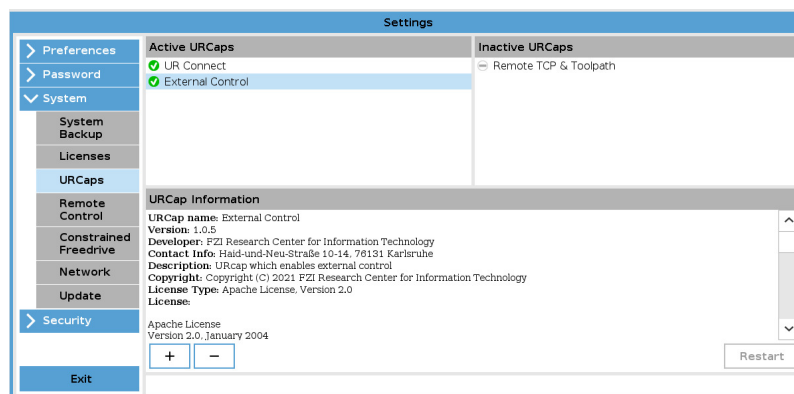


Figura 4.3: UR Cap

A questo punto, dal menù hamburger in alto a destra, selezionare “Settings” per entrare nelle impostazioni del robot e successivamente, nel menù sulla sinistra, selezionare “System-URCaps”; aggiungere quindi il file presente sulla chiavetta dal simbolo in basso a sinistra. Se l’UR Cap è stato selezionato correttamente, la vista del menù sarà quella mostrata nella Figura 4.3.

Per completare l’installazione è necessario riavviare il teach pendant, spostarsi nella sezione “Installation”, presente nel menù in alto a sinistra, e successivamente nella scheda “URCaps”: viene mostrato l’external control appena installato. Una volta selezionato, bisogna inserire l’indirizzo IP del PC remoto, come mostrato nella Figura 4.4.

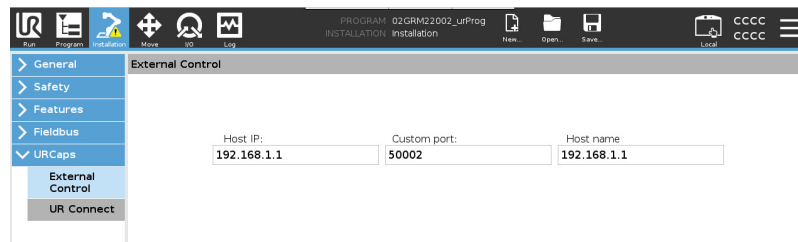


Figura 4.4: Impostazioni UR Cap

Infine, bisogna spostarsi nella sezione “Program” e creare un nuovo programma, inserendo l’external control tramite la scheda URCaps. Se tutto è stato installato correttamente, la schermata finale sarà come quella mostrata nella Figura 4.5.

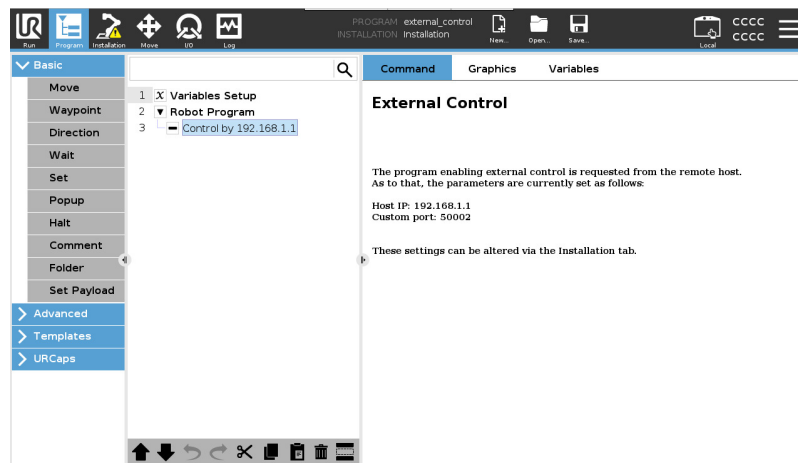


Figura 4.5: Programma con external control

Per utilizzare il programma creato e quindi controllare il robot tramite nodi ROS 2, è sufficiente avviare il programma tramite il tasto play presente nella parte inferiore dell’interfaccia del teach pendant dopo aver avviato tutti i nodi come descritto nella Sezione 4.8.

L’ultimo step del tutorial riguarda la configurazione di rete del PC [48]. Per garantire la comunicazione tra il computer e il robot, è necessario collegare

questi due componenti tramite connessione Ethernet diretta e assegnare al PC l'IP statico all'interno della stessa sottorete del robot. Nello specifico, al control box del robot è stato assegnato l'indirizzo 192.168.1.40, mentre al PC è stato assegnato l'indirizzo 192.168.1.1 e subnet mask 255.255.255.0.

Come ultima cosa, è bene controllare se la connessione tra PC e robot avviene correttamente con il comando:

```
$ ping 192.168.1.40
```

A questo punto, per verificare di aver eseguito tutto correttamente, è possibile eseguire il seguente comando, che inizializza il driver:

```
ros2 launch ur_robot_driver ur_control.launch.py
ur_type:=ur10e robot_ip:=192.168.1.40
```

Una volta stabilita la connessione con il robot e avviato il driver, viene avviato anche *RViz*, ovvero lo strumento di visualizzazione grafica dell'ecosistema ROS 2. Il risultato di tale comando è illustrato nella Figura 4.6.

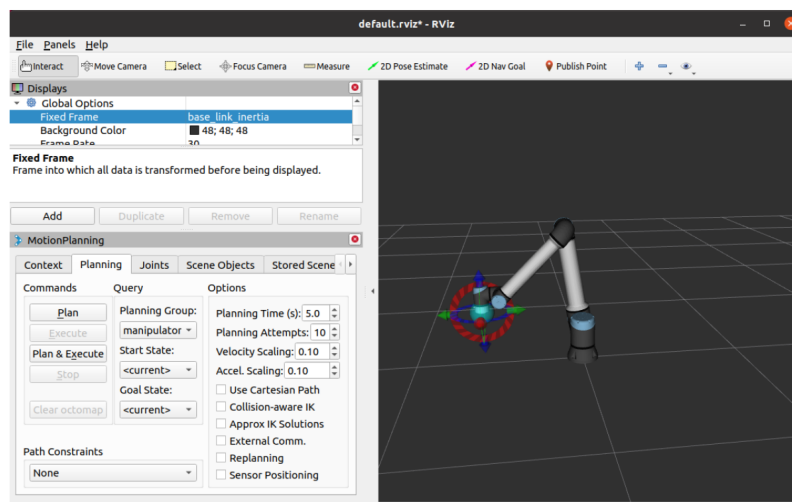


Figura 4.6: RViz

Parallelamente al driver, è stato integrato il pacchetto *ros2-humble-ros2controlcli* per l'interazione a riga di comando con il Controller Manager. Questa interfaccia si è rivelata fondamentale per quanto riguarda l'interazione con i diversi controllori ROS 2 del robot, come per passare dal controllo di traiettoria a quello di forza, senza dover riavviare tutto l'ecosistema. Il comando per installarla è:

```
$ sudo apt install ros-humble-ros2controlcli
```

4.3.2 MoveIt 2

MoveIt rappresenta lo standard de facto a livello globale per lo sviluppo di applicazioni di manipolazione robotica, integrando algoritmi di cinematica, pianificazione delle traiettorie e percezione tridimensionale. All'interno dell'applicazione

sviluppata, sono stati utilizzati *MoveIt 2* e il suo modulo *MoveIt Servo*, che hanno il compito di tradurre le coordinate tridimensionali stimate dal sistema di visione in movimenti fisici del manipolatore che siano fluidi, precisi e sicuri. Questi strumenti fungono da organo decisionale cinematico del sistema, interponendosi tra i nodi di percezione ad alto livello e i driver hardware a basso livello del robot.

Nel dettaglio, *MoveIt 2*, è incaricato della pianificazione point-to-point, ovvero, quando riceve una posa target nello spazio cartesiano espressa tramite una terna di coordinate e un quaternion di orientamento, calcola la cinematica inversa per determinare la configurazione dei giunti corrispondente, dopo di che genera una traiettoria che eviti collisioni e ne esegue la parametrizzazione temporale, definendo le velocità e accelerazioni (dentro i limiti dinamici del robot) per arrivare alla configurazione desiderata. Nel contesto del progetto di questa tesi, *MoveIt 2* è stato utilizzato per la gestione del task di *pick-and-place*, in cui i movimenti del manipolatore sono scanditi da pose statiche. L'integrazione e la configurazione del framework sono state eseguite seguendo le linee guida ufficiali [49].

Dall'altra parte, il modulo *MoveIt Servo*, a differenza della pianificazione classica, non richiede una posa finale statica, ma accetta in ingresso un flusso continuo di velocità, che viene trasmesso tramite messaggi ROS 2 di tipo *geometry_msgs/msg/TwistStamped*, per le velocità lineari e angolari nel dominio cartesiano, o *control_msgs/msg/JointJog*, per le velocità dirette nello spazio dei giunti. Questo modulo opera a frequenze alte, nel caso particolare del progetto analizzato a 50 Hz, e permette di evitare collisioni. All'interno dell'architettura proposta, *MoveIt Servo* è delegato al task di *visual servoing*, in cui le coordinate stimate dal sistema di visione vengono convertite in vettori di velocità, permettendo così al manipolatore di inseguire l'oggetto sul piano di lavoro in modo fluido e di correggere la traiettoria sulla base dei dati visivi più recenti.

Al fine di mettere in relazione il sistema di visione con il robot, è stato integrato il pacchetto *moveit2_calibration*, fondamentale per la procedura di calibrazione *hand-eye*, di cui si parla nella Sezione 4.7, il cui compito è determinare la matrice di trasformazione tra l'end-effector del robot e il frame della camera. La configurazione di questo pacchetto è stata effettuata seguendo il tutorial ufficiale [50], a differenza del quale, però, l'installazione è avvenuta nella cartella "src" del workspace di ROS 2 che è stato creato. Inoltre, il comando:

```
vcs import src < src/moveit_calibration/
moveit_calibration.repos --skip-existing
```

non funzionava, poichè nella repository GitHub non esisteva il file "moveit_calibration.repos", quindi, questo passaggio è stato saltato.

4.3.3 Driver e librerie RealSense

L'infrastruttura software per la gestione del sensore di visione, l'Intel RealSense D435, è suddivisa in due livelli principali: la libreria di basso livello *librealsense2*, che serve per la comunicazione diretta con l'hardware e per lo streaming dei flussi

video, e il driver ufficiale *realsense2_camera*, che serve per la gestione dei flussi video e per la loro conversione in topic di ROS 2.

Per l'installazione del driver *realsense2_camera* è stata seguita la procedura ufficiale [51], secondo la quale, bisogna clonare una repository GitHub all'interno della cartella “src” del workspace ROS 2 digitando i seguenti comandi:

```
$ cd ~/ros2_ws/src/  
$ git clone https://github.com/realsenseai/realsense-  
  ros.git -b ros2-master
```

Successivamente, installare le dipendenze utilizzando il pacchetto *rosdep*:

```
rosdep install -i --from-path src --rosdistro humble  
  --skip-keys=librealsense2 -y
```

Per la libreria *librealsense2*, invece, si è seguita la procedura ufficiale di installazione [52], che, nella sua versione standard, prevede l'utilizzo di moduli *dkms*. Tuttavia, operando in un ambiente con patch real-time, tali moduli falliscono in fase di compilazione; quindi, per ovviare a questo vincolo, è stato necessario installare esclusivamente i pacchetti di utility e sviluppo, tramite i comandi:

```
$ sudo apt-get install librealsense2-utils  
$ sudo apt-get install librealsense2-dev
```

4.3.4 OpenCV e cv_bridge

Per l'implementazione della pipeline di visione e per la manipolazione dei flussi video è stato necessario integrare la libreria *OpenCV* affiancata dal pacchetto *cv_bridge*, strumenti software dedicati all'elaborazione digitale delle immagini.

OpenCV si occupa della gestione dei dati estratti dalla telecamera; il suo utilizzo nel progetto è fondamentale per svolgere diversi compiti, come la decodifica e la manipolazione dei frame, la rappresentazione visiva dei risultati della detection (ad esempio il disegno dei *bounding box* attorno agli oggetti riconosciuti) e la generazione di finestre grafiche, utili all'operatore per verificare la correttezza dell'inquadratura e dei risultati. Questa libreria si può installare con il seguente comando:

```
$ pip install opencv-python
```

All'interno dell'ecosistema ROS 2, tuttavia, i flussi video vengono pubblicati dal nodo della camera come messaggi nativi di tipo *sensor_msgs/msg/Image*, e non come matrici *OpenCV*, rendendo i frame originari incompatibili con la libreria. Per questo motivo, si è utilizzato il modulo *cv_bridge*, che funge da interprete e da ponte bidirezionale, intercettando il messaggio ROS 2 e convertendolo in una matrice *OpenCV* per l'elaborazione, operando poi la conversione inversa al termine dell'elaborazione della rete neurale. Si può installare questo modulo con il comando:

```
$ git clone https://github.com/ros-perception/  
  vision_opencv.git -b humble
```

4.3.5 Ambiente virtuale ROS 2-Python

Al fine di garantire l'isolamento delle dipendenze software necessarie per i moduli di deep learning ed evitare eventuali conflitti con le librerie Python globali del sistema operativo, è stato creato un *ambiente virtuale* dedicato.

Inizialmente è stata creata una cartella specifica come spazio di lavoro per gli script Python all'interno del workspace ROS 2:

```
$ cd ~/ros2_ws/src
$ mkdir 00grs25001.py
$ cd 00grs25001.py
```

Successivamente, è stato creato l'ambiente virtuale in questa maniera:

```
$ python3 -m venv --system-site-packages .venv
```

Come si può notare dal comando precedente, è stato utilizzato il flag “*--system-site-packages*”: questo flag permette al *virtual environment* di ereditare i pacchetti Python già installati globalmente tramite il gestore APT di Ubuntu, mantenendo però una directory locale, “*.venv/lib/python3.10/site-packages*”, in maniera tale che l'installazione tramite *pip* di framework complessi come *PyTorch* possa essere fatta localmente, evitando interferenze con il sistema ospite.

In seguito, l'attivazione dell'ambiente virtuale è stata integrata all'interno del file *bashrc*, a cui si è acceduto dapprima con il comando:

```
$ sudo nano ~/.bashrc
```

e a cui si è aggiunto in coda il comando per l'attivazione dell'ambiente virtuale: la sequenza di caricamento delle variabili d'ambiente è stata strutturata inserendo lo script di attivazione del virtual environment subito dopo l'ambiente globale di ROS 2:

```
# 1. Ambiente globale ROS 2 Humble
source /opt/ros/humble/setup.bash

# 2. Attivazione dell'ambiente virtuale Python
source ~/ros2_ws/src/00grs25001.py/.venv/bin/activate
```

4.3.6 Driver per la scheda grafica

Parlando di deep learning applicato alla visione artificiale in tempo reale, è di cruciale importanza la scelta dell'unità di computazione utilizzata per la rete neurale. Nel dettaglio, i modelli di object detection come quelli della famiglia YOLO beneficiano dell'esecuzione su *GPU* rispetto a *CPU*, in quanto la scheda video è ingegnerizzata per ridurre di molto il tempo di inferenza della rete.

Tuttavia, nel cercare di permettere l'esecuzione su *GPU* del modello riaddestrato sul dataset personalizzato, è stato riscontrato un limite hardware intrinseco e non evitabile: la scheda grafica integrata nel PC utilizzato, il modello *AMD Radeon 530*, è considerata obsoleta. A causa di questa incompatibilità generazionale, non è stato possibile utilizzare la scheda video.

Per questo motivo, si è deciso di eseguire la pipeline della rete neurale su *CPU*.

A questo scopo, all'interno dell'ambiente virtuale precedentemente configurato (come descritto nella Sezione 4.3.5), è stata installata la versione del framework *PyTorch* ottimizzata per l'esecuzione su *CPU*, seguendo le istruzioni del sito ufficiale [53], con il seguente comando:

```
$ pip3 install torch torchvision --index-url https://
download.pytorch.org/whl/cpu
```

L'esecuzione del modello sulla *CPU* invece che sulla *GPU*, ovviamente, comporta un inevitabile incremento dei tempi di inferenza, che si riflette direttamente sul sistema riducendo la prontezza di risposta del manipolatore e introducendo ritardi nell'aggiornamento delle traiettorie.

4.4 Calcolo del payload

Visto il setup della pinza, montata sulla flangia del robot, e della camera, montata sul polso, è stato necessario configurare correttamente il *payload*, ovvero il carico sull'end-effector del robot, poiché una stima errata di questo valore potrebbe comportare problemi anche gravi, come l'usura dei riduttori dei giunti e l'attivazione improvvisa dei sistemi di sicurezza in seguito a un falso rilevamento di collisione.

Per la famiglia Universal Robots e-Series, per l'operazione di calcolo del payload è presente una procedura interamente automatizzata integrata nel *teach pendant*. Questa procedura si compone delle seguenti fasi:

- *setup iniziale*: vengono montati sul manipolatore tutti i tool, nel caso specifico di questa tesi la pinza e la camera; poi bisogna spostarsi nella sezione "Installation" e nella scheda "Payload", come si vede nella Figura 4.7, poi selezionare "Measure", facendo iniziare la procedura automatizzata del calcolo del payload;

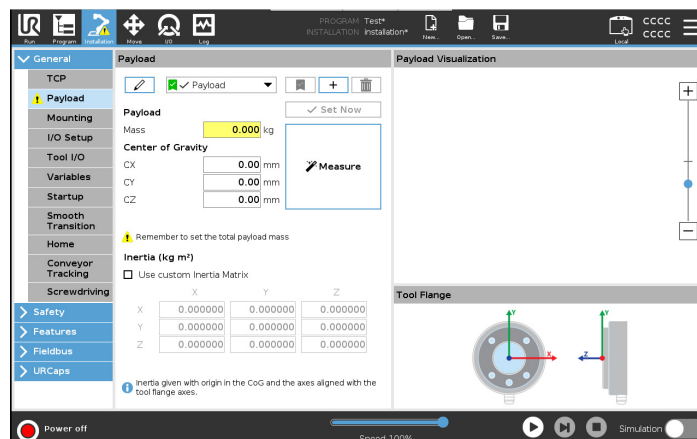


Figura 4.7: Calcolo del payload - Home

- *inizio della procedura*: si apre una schermata, come quella mostrata nella Figura 4.8, che spiega che per eseguire la calibrazione è necessario scegliere 4 pose del robot che siano diverse tra loro in maniera tale da permettere un calcolo corretto;

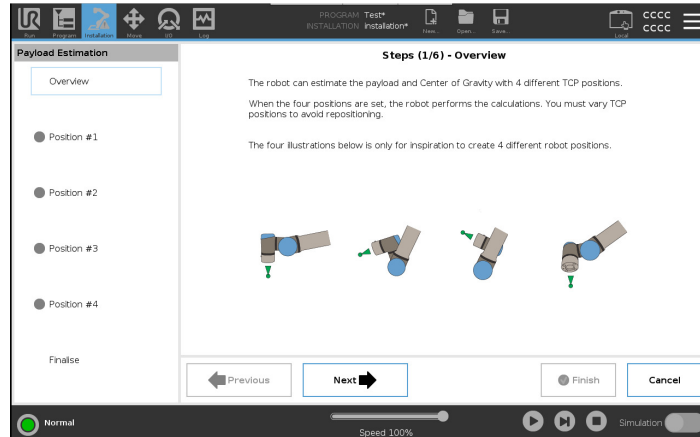


Figura 4.8: Calcolo del payload - Overview

- *esecuzione della routine di movimento*: bisogna spostare il robot nella posizione desiderata cliccando “Set Position”, come mostrato nella Figura 4.9, poi “Move Here” e poi “Next”, per ognuna delle 4 posizioni;

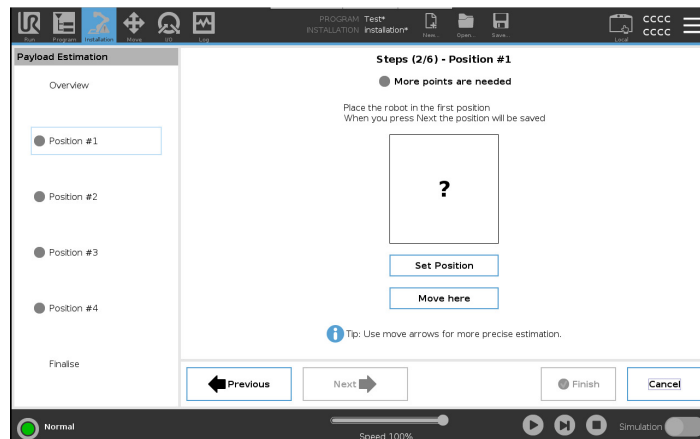


Figura 4.9: Calcolo del payload - Posizionamento

- *identificazione del payload*: alla fine, come illustrato nella Figura 4.10, vengono calcolati automaticamente sia il *payload*, espresso in *kg*, sia le coordinate spaziali del centro di gravità (x, y, z) rispetto al frame della flangia, espresse in millimetri.

Alla fine della procedura, è necessario salvare le modifiche apportate al file di installazione per rendere permanenti le misure appena calcolate.

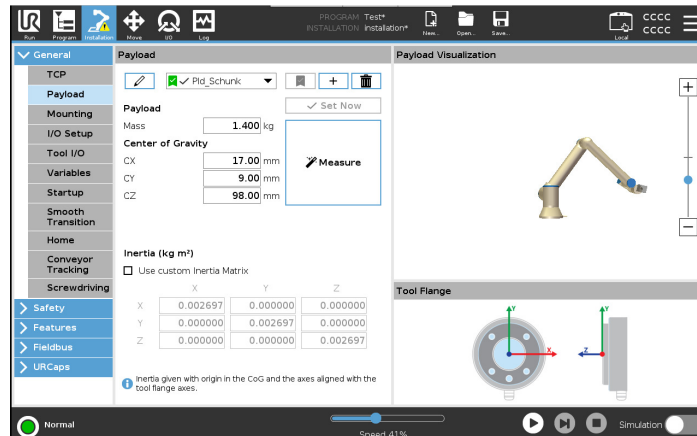


Figura 4.10: Calcolo del payload - Final

4.5 Configurazione del TCP

Per consentire al robot di svolgere i due compiti di manipolazione e di inseguimento degli oggetti con l'accuratezza migliore possibile, è stato necessario definire un nuovo *TCP*, *Tool Center Point*, che fosse solidale al tool di presa e, in particolare, al punto di presa. Di default, la catena cinematica fornita dai driver ufficiali del robot termina sul frame della flangia del robot, denominato *tool0*, e l'aggiunta di una pinza porta con sé un *offset* geometrico, sia di traslazione che di rotazione, che deve essere integrato nel file di configurazione del robot, il file *URDF/XACRO*. Questo passaggio è fondamentale: fa in modo che il tool di presa sia visto come parte integrante del manipolatore stesso, cosicché possa essere possibile per gli algoritmi di cinematica inversa e di pianificazione di *MoveIt 2* calcolare le traiettorie rispetto al punto di contatto reale tra la pinza e l'oggetto, oltre che evitare collisioni del tool con altri oggetti o persone.

Il processo ha richiesto come prima cosa l'inizializzazione di un pacchetto ROS 2 dedicato alla definizione del modello esteso del robot, denominato *my_robot_description*, all'interno della cartella dedicata del workspace:

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_cmake
  my_robot_description
$ mkdir -p my_robot_description/urdf
```

All'interno della cartella *urdf* della directory appena creata, è stato generato il file "*ur10e_gripper.urdf.xacro*", script che sia include le definizioni geometriche principali del robot sia specifica un nuovo link rigido, denominato "*gripper_tcp_link*", collegato al frame *tool0* tramite un giunto di tipo fisso, *gripper_tcp_joint*, introducendo i parametri fisici e l'offset misurato.

4.5.1 MoveIt Setup Assistant

Nonostante l'aggiunta del nuovo link nel file di configurazione del robot, per fare in modo che *gripper_tcp_link* fosse considerato da *MoveIt 2* come l'estre-

mità della catena cinematica del manipolatore e quindi incluso nel calcolo della cinematica inversa, è stato utilizzato il tool grafico *MoveIt Setup Assistant*, eseguendo preliminarmente la conversione e la validazione del file *xacro* in un formato *urdf* standard:

```
$ xacro ur10e_gripper.urdf.xacro > ur10e_gripper.urdf
$ check_urdf ur10e_gripper.urdf
$ ros2 run moveit_setup_assistant
  moveit_setup_assistant
```

L'ultimo comando apre l'interfaccia di configurazione guidata del tool, quella mostrata nella Figura 4.11. Qui è necessario cliccare su "Create New MoveIt Configuration Package", inserire nello spazio apposito il file *urdf* precedentemente creato e cliccare su "Load File".

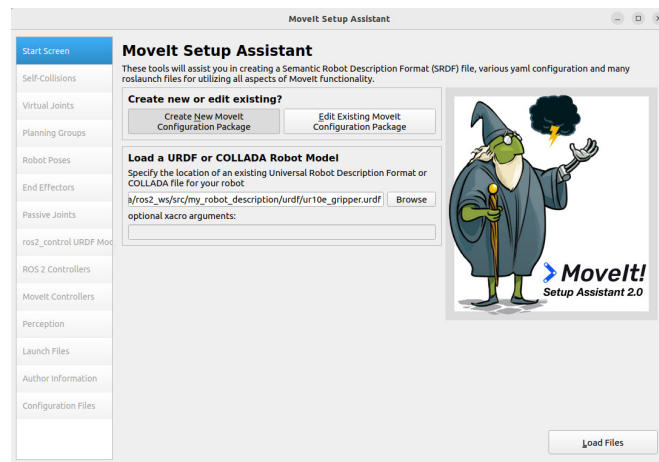


Figura 4.11: MoveIt Setup Assistant

Successivamente, basta seguire la pipeline definita sulla sinistra della schermata e definire tutti gli elementi necessari.

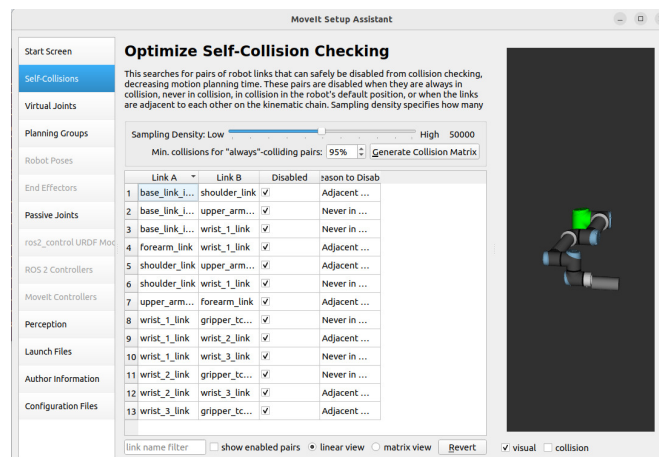


Figura 4.12: Ottimizzazione del controllo di auto collisione

In particolare, il secondo passo, mostrato nella Figura 4.12, consiste nell'ottimizzazione del controllo di auto collisione del manipolatore. È bastato settare il parametro di “Sampling Density” a 50'000 e cliccare “Generate Collision Matrix”: il processo viene fatto in automatico dal setup assistant.

In seguito, è stato definito un giunto virtuale, come si vede nella Figura 4.13, che serve per determinare la relazione spaziale e cinematica tra il manipolatore e l'ambiente circostante, permettendo di ancorare la radice dell'albero delle trasformate del robot a un sistema di riferimento globale fisso, denominato “*world*”.

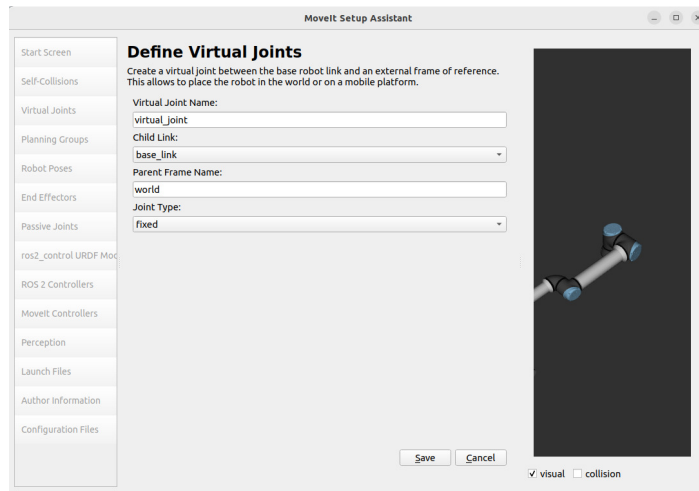


Figura 4.13: Definizione dei giunti virtuali

La Figura 4.14 mostra la schermata iniziale per la definizione dei *planning groups*, che rappresenta la fase centrale del *MoveIt Setup Assistant*.

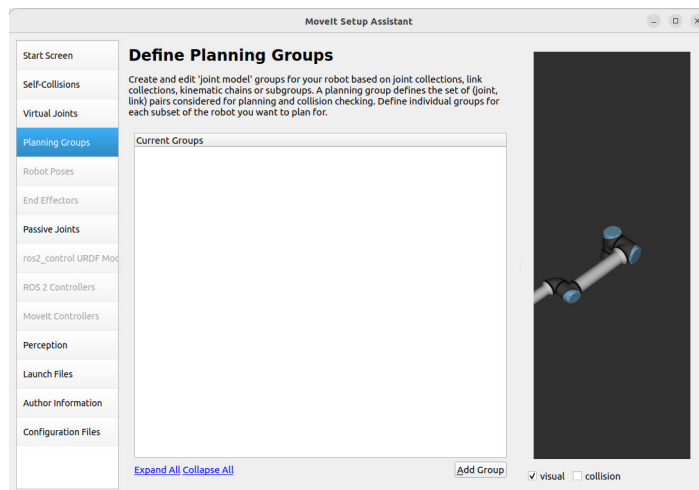


Figura 4.14: Definizione dei planning groups

Un gruppo di pianificazione è una collezione di giunti e collegamenti strutturati in un'unica entità, per la quale MoveIt calcolerà le traiettorie di movimento:

è il passaggio effettivo per far riconoscere la pinza come parte integrante del manipolatore. A partire dalla schermata mostrata nella figura, è necessario quindi cliccare su “Add Group”.

Il primo gruppo che è stato aggiunto, come si vede nella Figura 4.15, è *ur_manipulator*, la catena cinematica principale che include i sei giunti del braccio UR10e più il giunto fisso della pinza. Come si vede nell’immagine, è stato definito il nome del gruppo ed è stato scelto come risolutore cinematico *kdl_kinematics_plugin/KDLKinematicPlugin*; in seguito, si è cliccato su “Add Kin. Chain” per definire la catena cinematica.

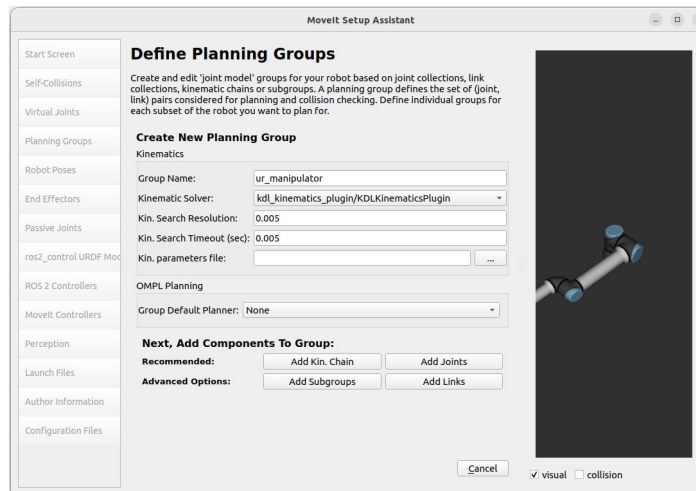


Figura 4.15: Definizione del gruppo *ur_manipulator*

Si apre una schermata come quella mostrata nella Figura 4.16, dove bisogna selezionare come “Base Link” il link della base del robot, ovvero l’omonimo “*base_link*”, e come “Tip Link” il link della pinza, ovvero “*gripper_tcp_link*”. In seguito premere su “Save”.

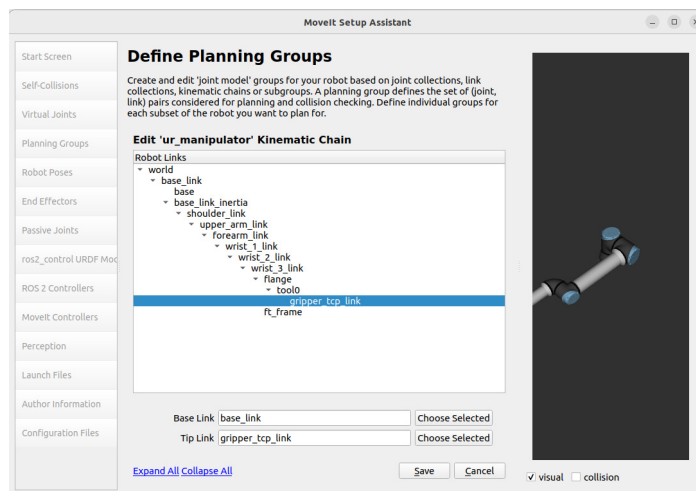


Figura 4.16: Definizione della catena cinematica del gruppo *ur_manipulator*

Una volta fatto ciò, si riapre una schermata come quella della Figura 4.14, da cui bisogna cliccare il tasto per aggiungere un altro gruppo. Il secondo planning group che bisogna definire è “*gripper*”, quello dedicato al tool di presa.

Come illustrato nella Figura 4.17, dopo aver definito il nome del gruppo, il valore del parametro “Kinematic Solver” è stato impostato a *None*. Questa scelta deriva dalla natura pneumatica della pinza, che, non avendo giunti attivi, non possiede gradi di libertà interni che abbiano bisogno di un algoritmo di cinematica inversa: la sua movimentazione nello spazio, infatti, è determinata solamente dal polso del robot al quale è ancorato.

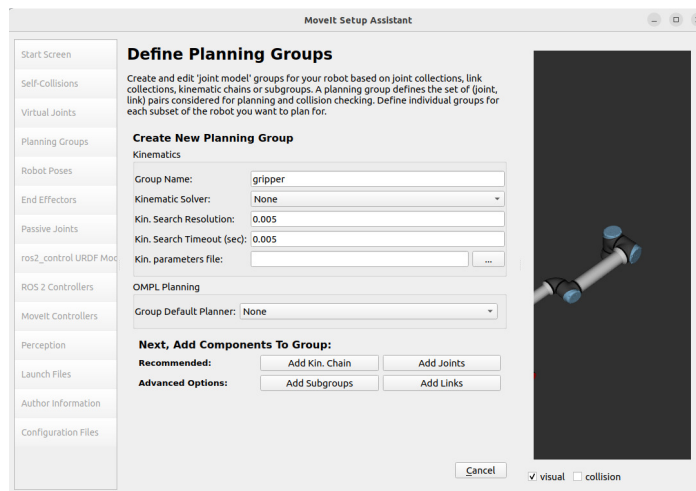


Figura 4.17: Definizione del gruppo *gripper*

Successivamente, cliccando su “Add Links” si apre una schermata come quella mostrata nella Figura 4.18, in cui è necessario aggiungere sia il link fisso della pinza, ovvero “*gripper_tcp_link*”, sia il link genitore attivo, ovvero “*tool0*”, e, infine, cliccare su “Save”.

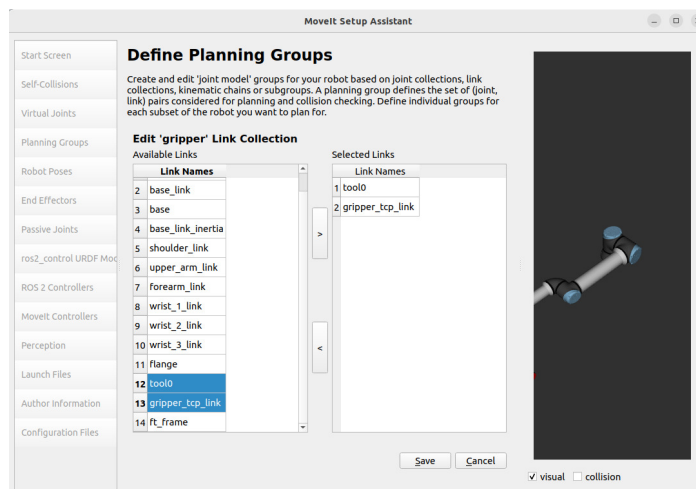


Figura 4.18: Definizione dei link appartenenti al gruppo *gripper*

Il risultato della procedura riguardante i planning groups è quello mostrato nella Figura 4.19, dove si vedono sia il gruppo “*ur_manipulator*” sia il gruppo “*gripper*”.

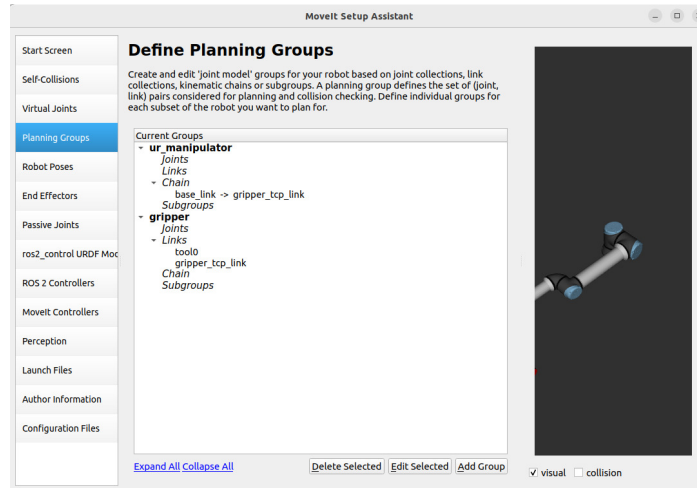


Figura 4.19: Planning groups definiti

Per lo step successivo è necessario definire l’end-effector del manipolatore, che, nel caso di questa tesi, è proprio la pinza di presa. Quindi, come si vede nella Figura 4.20, sono stati definiti il nome dell’end-effector, il gruppo di appartenenza, il link genitore e il gruppo genitore, settato a “*ur_manipulator*” in quanto genitore del gruppo “*gripper*”. Dopo di che, bisogna cliccare su “Save”.

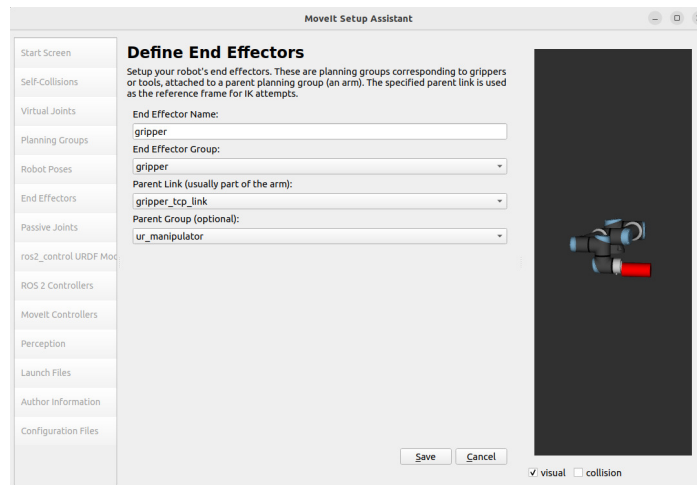


Figura 4.20: Definizione dell’end-effector

La fase successiva, quella di configurazione dei controllori, denominata “*Setup ROS 2 Controllers*”, funge da ponte tra l’ambiente di pianificazione ad alto livello, ovvero *MoveIt*, e l’infrastruttura di controllo hardware a basso livello, ovvero *ROS 2*. Quindi, questo step ha due scopi principali:

- definire l'architettura del controllore software, solitamente un *JointTrajectoryController*, associato al gruppo di pianificazione *ur_manipulator*, che riceve la traiettoria da MoveIt e genera i comandi da inviare al robot;
- configurare il canale di comunicazione tra MoveIt e il robot, esponendo un'interfaccia di tipo *FollowJointTrajectory*: MoveIt invia il comando della traiettoria a questa interfaccia, che monitora l'avanzamento dei motori e notifica il successo o l'eventuale fallimento del movimento.

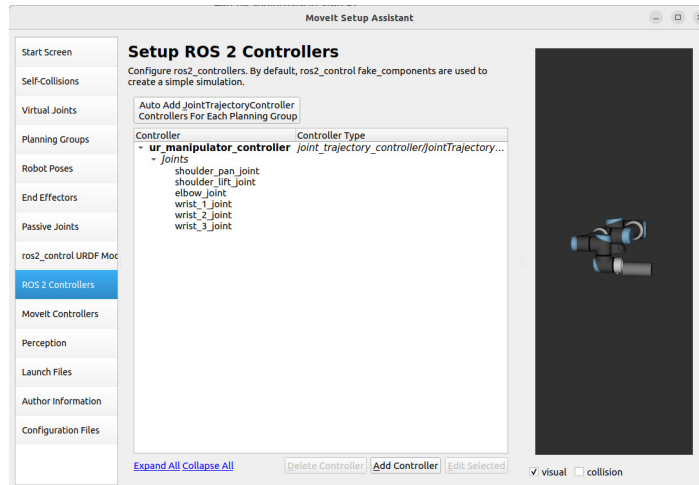


Figura 4.21: Setup ROS 2 Controllers

Come mostrato nella Figura 4.21, è sufficiente cliccare sul tasto “Auto Add JointTrajectoryController Controllers For Each Planning Group”, e il setup assistant genera automaticamente i controllori.

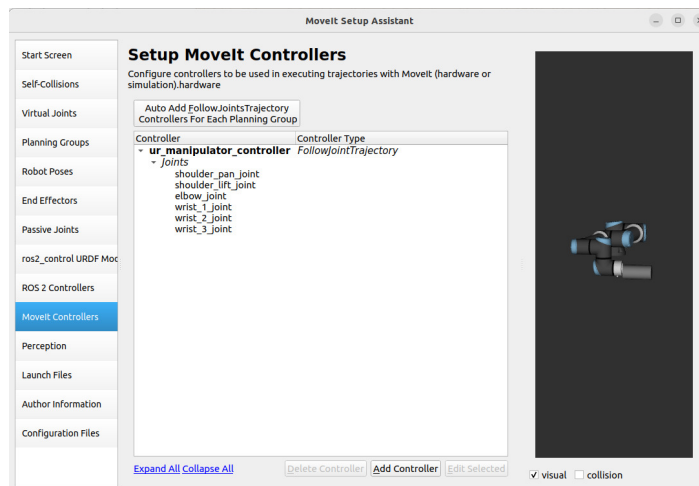


Figura 4.22: Setup MoveIt Controllers

Invece, lo step “*Setup MoveIt Controllers*”, Figura 4.22, al contrario della fase precedente, che delinea i controllori lato hardware, definisce l'interfaccia di uscita

del pianificatore, istruendo MoveIt sul canale in cui inoltrare le traiettorie calcolate. Concretamente, viene mappato il gruppo *ur_manipulator* con il rispettivo *topic* di Action esposto dal driver reale del robot, */joint_trajectory_controller/follow_joint_trajectory*. Anche qui, come si vede nella figura, c'è un tasto che permette lo svoglimento automatico del setup.

Come si vede nella Figura 4.23, è stato necessario aggiungere delle informazioni fittizie riguardo all'autore: come nome è stato inserito "Idea" e come email "idea@todo.todo".

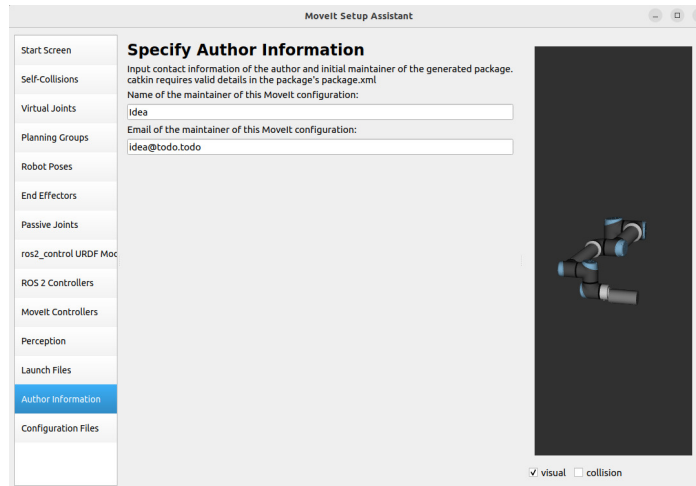


Figura 4.23: Definizione delle informazioni dell'autore

Nella fase conclusiva, *Generate Configuration Files* (Figura 4.24), l'interfaccia grafica traduce l'intera configurazione appena definita in codice sorgente, esportando autonomamente un nuovo pacchetto ROS 2 indipendente, in questo caso chiamato "*ur10e_gripper_moveit_config*".

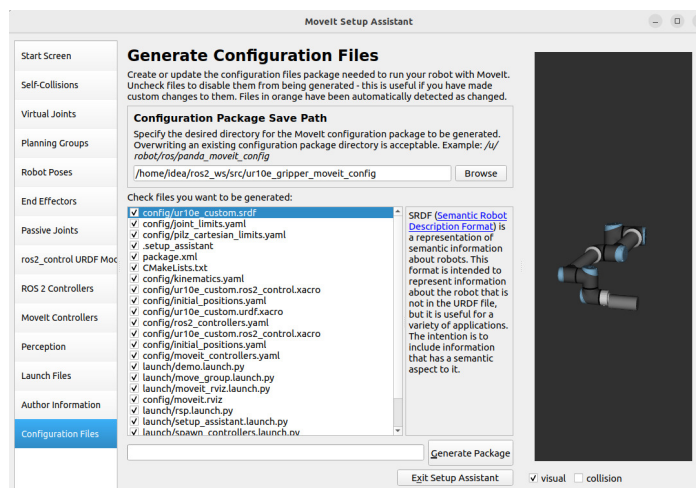


Figura 4.24: Generazione dei file finali

Questo passaggio è fondamentale, in quanto genera file core del framework, come il *.srdf*, che definisce i gruppi di pianificazione e le matrici di collisione interna, i file *.yaml* di configurazione per i controllori e i limiti dei giunti e gli script di avvio in Python.

4.6 Compilazione del workspace

La compilazione dei diversi moduli integrati nel workspace è affidata a *colcon*, il meta-build system di riferimento per l'ecosistema ROS 2.

L'architettura software di un sistema robotico basato su ROS 2 di solito è formata da diversi pacchetti, nodi e librerie interdipendenti: *colcon* permette di fare la build dell'intero workspace come un unico organismo. Questo approccio permette di evitare conflitti tra i vari path ed errori di linking.

Prima della compilazione, il workspace è composto solamente della cartella *src*; invece, dopo, vengono generate tre nuove directory:

- *build*: contiene i file generati durante il processo di compilazione, come i file oggetto *.o*, le cache di CMake e i file di configurazione temporanei;
- *install*: rappresentando l'output finale del processo, contiene i binari eseguibili, le librerie dinamiche, *.so*, gli script e le risorse di runtime;
- *log*: contiene i file di testo con i log e gli output di terminale relativi a ciascuna invocazione del comando di build, risultando indispensabile per il *debugging*.

Il comando di compilazione adottato per la fase di build dell'ecosistema è stato:

```
$ cd ~/ros2_w
$ colcon build --symlink-install --cmake-args -
  DCMAKE_BUILD_TYPE=Release --mixin release
```

Durante la fase di compilazione, sono emerse diverse problematiche, che verranno analizzate nella Sezione 4.6.1 insieme alle strategie adottate per mitigarle.

Una volta risolte queste criticità e una volta che la build sia andata a buon fine, è necessario inserire l'istruzione di inizializzazione del *workspace* in coda al file di configurazione *bashrc*, per evitare l'esecuzione manuale del comando *source* ogni volta che si apre un nuovo terminale. Per fare ciò, bisogna aprire il file *bashrc*:

```
$ sudo nano ~/.bashrc
```

e inserire il comando in coda:

```
# 1. Ambiente globale ROS 2 Humble
source /opt/ros/humble/setup.bash

# 2. Attivazione dell'ambiente virtuale Python
source ~/ros2_ws/src/00grs25001.py/.venv/bin/activate
```

```
# 3. Source del workspace ROS 2
source ~/ros2_ws/install/setup.bash
```

4.6.1 Errori in fase di build

OpenCV

Il primo problema che si è riscontrato in fase di compilazione ha riguardato il pacchetto *vision_opencv*. Il fallimento della build è stato ricondotto a una modifica introdotta nelle versioni più recenti del pacchetto: nelle versioni più moderne, i file di intestazione, gli *header file*, hanno abbandonato la nomenclatura classica *.h* in favore di un'estensione *.hpp*. Tuttavia, ROS 2 Humble fa ancora riferimento agli standard precedenti (*.h*), per questo motivo, il compilatore non era in grado di risolvere i percorsi di inclusione per alcune librerie.

Per risolvere questa incompatibilità, è stata necessaria una fase di *refactoring*, ovvero è stato ispezionato manualmente il codice sorgente dove si erano verificati gli errori e sono stati modificati tutti gli header *.hpp* che davano problemi in *.h*.

In particolare, gli header file modificati sono stati:

- nel file *handeye_context_widget.h*, a cui si è acceduto con il comando:

```
$ nano ~/ros2_ws/src/moveit_calibration/
  moveit_calibration_gui/
  handeye_calibration_rviz_plugin/include/
  moveit/handeye_calibration_rviz_plugin/
  handeye_context_widget.h
```

sono stati modificati:

```
#include <image_geometry/pinhole_camera_model.hpp
> -> .h
#include <moveit/
  common_planning_interface_objects/
  common_objects.hpp> -> .h
#include <moveit/utils/rclcpp_utils.hpp> -> .h
```

- nel file *handeye_control_widget.h*, a cui si è acceduto con il comando:

```
$ nano ~/ros2_ws/src/moveit_calibration/
  moveit_calibration_gui/
  handeye_calibration_rviz_plugin/include/
  moveit/handeye_calibration_rviz_plugin/
  handeye_control_widget.h
```

sono stati modificati:

```
#include <moveit/move_group_interface/
  move_group_interface.hpp> -> .h
#include <moveit/planning_scene_monitor/
  planning_scene_monitor.hpp> -> .h
```


- nel file *handeye_target_widget.h*, a cui si è acceduto con il comando:

```
$ nano ~/ros2_ws/src/moveit_calibration/
  moveit_calibration_gui/
  handeye_calibration_rviz_plugin/include/
  moveit/handeye_calibration_rviz_plugin/
  handeye_target_widget.h
```

è stato modificato:

```
#include <cv_bridge/cv_bridge.hpp> -> .h
```

- nel file *handeye_control_widget.cpp*, a cui si è acceduto con il comando:

```
$ nano ~/ros2_ws/src/moveit_calibration/
  moveit_calibration_gui/
  handeye_calibration_rviz_plugin/src/
  handeye_control_widget.cpp
```

è stato modificato:

```
#include <moveit/utils/moveit_error_code.hpp> ->
.h
```

Errori di MoveIt Setup Assistant

Un'altra problematica riscontrata è stata causata dai file autogenerati dal tool *MoveIt Setup Assistant*.

Durante la generazione del pacchetto *ur10e_gripper_moveit_config*, l'assistente non ha copiato i file di configurazione *.yaml* originali che contenevano i limiti fisici dei giunti, il che ha comportato dei crash del processore *xacro* per file non trovato. Quindi, è stato necessario creare una nuova directory in questa maniera:

```
$ mkdir -p ~/ros2_ws/src/my_robot_description/config/
ur10e
```

e copiare i file originali dalla cartella di sistema alla cartella locale:

```
$ cp /opt/ros/humble/share/ur_description/config/
ur10e/*.yaml ~/ros2_ws/src/my_robot_description/
config/ur10e/
```

Infine, si è dovuto modificare il file *CMakeLists.txt* nella cartella *ros2_ws/src/my_robot_description* per fare in modo che contenesse dei comandi che rendessero visibili la nuova cartella creata durante la compilazione:

```
install(
  DIRECTORY urdf config
  DESTINATION share/${PROJECT_NAME}
)
```

4.7 Calibrazione della camera

Per un corretto utilizzo della camera è necessario eseguire la procedura di calibrazione: questa fa in modo che la camera sia in grado di stabilire la sua posizione all'interno del sistema determinando l'offset presente tra il sistema di riferimento del robot e quello del sensore. Nel progetto analizzato, questo obiettivo è stato raggiunto tramite una calibrazione *hand-eye* nella configurazione *eye-in-hand*.

Per effettuare questo passaggio, è stato necessario avviare tutti i nodi ROS 2, come mostrato nella Sezione 4.8 (escluso quello dedicato alla calibrazione). Dalla schermata di RViz che si apre, bisogna aggiungere un nuovo pannello alla visuale cliccando il tasto “Add” che si trova sulla sinistra, come mostrato nella Figura 4.25.

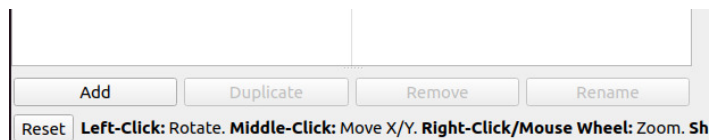


Figura 4.25: Aggiunta di un pannello su RViz

Poi, nella finestra che si apre, mostrata nella Figura 4.26, selezionare la voce “*HandEyeCalibration*”.

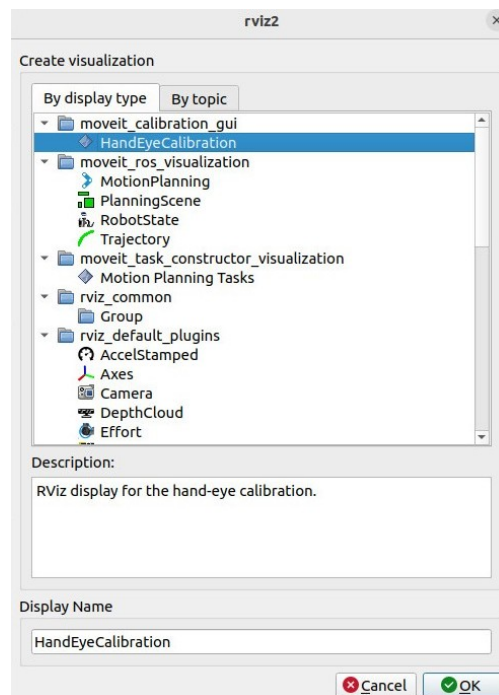


Figura 4.26: Scelta del pannello “*HandEyeCalibration*”

Verrà visualizzato un nuovo pannello, come quello nella Figura 4.27, in cui è possibile selezionare le prime specifiche per il target per la calibrazione. In

questo caso, sono stati impostati un numero di righe pari a 4 e un numero di colonne pari a 3.

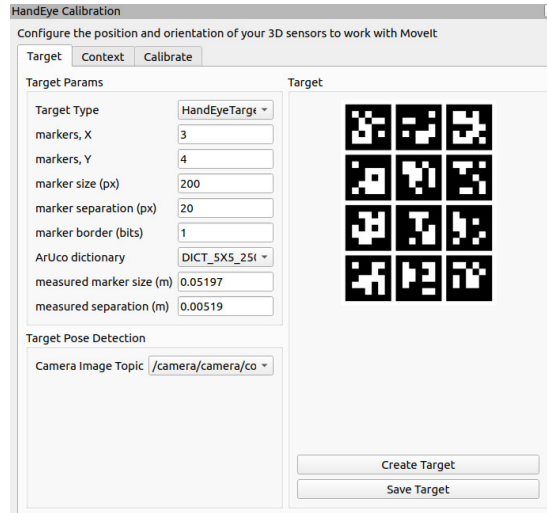


Figura 4.27: Pannello di calibrazione - Target

Il target così generato è quello illustrato nella Figura 4.28, che è stato stampato su un foglio cartaceo in formato A4 e vincolato saldamente al piano di lavoro, in maniera tale che non si spostasse durante la procedura.

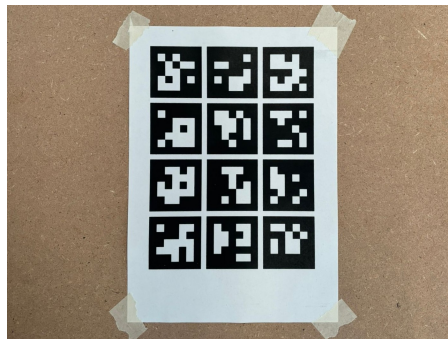


Figura 4.28: Target per la calibrazione

Successivamente, sono state misurate le caratteristiche geometriche del pattern e sono state impostate nel pannello, come si vede nella Figura 4.27: “Marker Size” è stata impostata a 0.05197 m e “Marker Separation” è stata impostata a 0.00519 m.

La Figura 4.29 mostra la scheda successiva, in cui sono stati impostati i frame di riferimento nel seguente modo:

- sensor configuration: *eye-in-hand*;
- sensor frame: *camera_color_optical_frame*;
- object frame: *handeye_target*;

- end-effector frame: *tool0*;
- robot base frame: *base_link*.

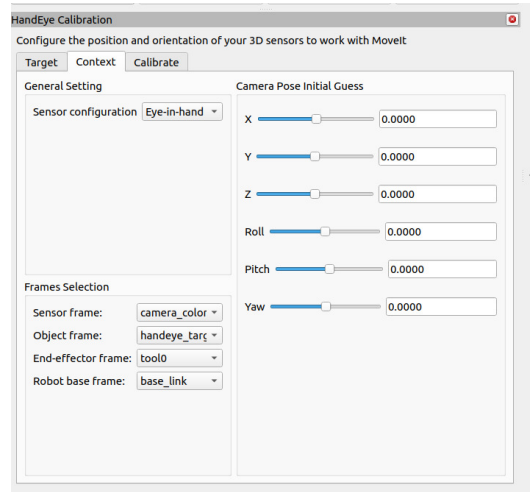


Figura 4.29: Pannello di calibrazione - Context

Per la raccolta dei dati è stato configurato un programma sul *teach pendant* del robot con un set di 20 pose diverse tra loro, variando l'orientamento e la distanza dell'end-effector dal target e raccogliendo un dato per ogni posa, come mostrato nella Figura 4.30, che mostra l'ultima scheda del pannello di calibrazione.

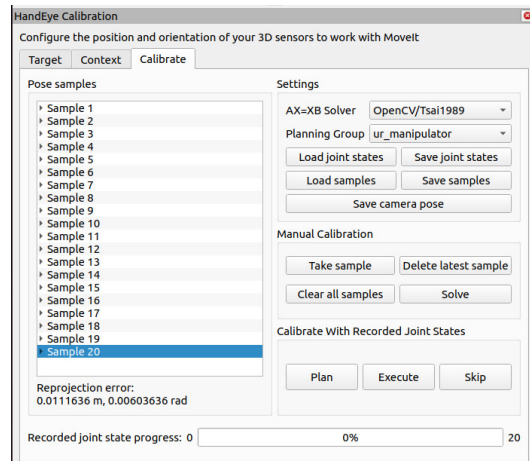


Figura 4.30: Pannello di calibrazione - Calibrate

Una volta raccolti tutti i dati, è stato salvato il file risultante cliccando sul tasto "Save camera pose": il file conteneva l'offset del frame di riferimento della camera rispetto al frame della flangia del manipolatore:

- *traslazione asse x*: -3.2 cm;

- *traslazione asse y*: -17.5 cm;
- *traslazione asse z*: -1.5 cm.

I parametri di rototraslazione ottenuti sono stati incorporati nel file di *launch* del sistema integrato di cui si parla nella Sezione 4.8, e, in particolare, nella Figura 4.34.

4.8 File di avvio dell'ecosistema ROS 2

Per fare in modo di avviare tutti i nodi del sistema contemporaneamente, è stato creato un file di avvio principale, articolato in quattro fasi sequenziali: la risoluzione dei percorsi delle directory, il caricamento e parsing dei file *urdf*, *srdf* e *yaml*, la dichiarazione delle istanze dei nodi e, infine, la temporizzazione effettiva dell'avvio, per fare in modo di attivare certi nodi prima di altri.

Configurazione dei sottosistemi

La configurazione dei sottosistemi è la fase principale del file di launch. In particolare, questi sottosistemi sono:

- *driver del robot*, Figura 4.31: include il launch nativo *ur_control.launch.py*, istanziando l'IP del robot a 192.168.1.40 e utilizzando i file di configurazione *my_robot_calibration.yaml* e di descrizione *ur10e_gripper.urdf.xacro* creati precedentemente (Sezione 4.5);

```
# Blocco A: Driver UR10e Reale
launch_ur = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(ur_driver_dir, 'launch', 'ur_control.launch.py')]),
    launch_arguments={
        'ur_type': 'ur10e',
        'robot_ip': '192.168.1.40',
        'use_fake_hardware': 'false',
        'initial_joint_controller': 'scaled_joint_trajectory_controller',
        'activate_joint_controller': 'true',
        'kinematics_params_file': '/home/idea/ros2_ws/my_robot_calibration.yaml',
        'description_package': 'my_robot_description',
        'description_file': 'ur10e_gripper.urdf.xacro',
        'launch_rviz': 'false'
    }.items()
)
```

Figura 4.31: Nodo del robot

- *MoveIt 2 e RViz*, Figura 4.32: il nodo *move_group* viene alimentato con i file *urdf*, *xacro* e *srdf* che descrivono il robot, abilitando il monitoraggio della scena e interfacciandosi con l'interfaccia RViz2;
- *camera RealSense*, Figura 4.33: il driver della camera RealSense viene avviato attivando l'allineamento dei frame RGB e di profondità (parametro *'align_depth.enable': 'true'*) e la generazione della nuvola di punti (parametro *'pointcloud.enable': 'true'*);
- *calibrazione della camera*, Figura 4.34: il nodo *static_transform_publisher* del pacchetto *tf2_ros* pubblica la trasformazione tra la flangia del robot

```
# Blocco B: Move Group Node
move_group_node = Node(
    package="moveit_ros_move_group",
    executable="move_group",
    output="screen",
    parameters=[
        robot_description,
        robot_description_semantic,
        robot_description_kinematics,
        controllers_yaml_content,
        {
            "publish_monitored_planning_scene": True,
            "moveit_controller_manager": "moveit_simple_controller_manager/MoveItSimpleControllerManager"
        }
    ],
)

# Blocco C: RViz 2
rviz_node = Node(
    package="rviz2",
    executable="rviz2",
    name="rviz2",
    output="screen",
    arguments=["-d", os.path.join(moveit_config_dir, "config", "moveit.viz")],
    parameters=[
        robot_description,
        robot_description_semantic,
        robot_description_kinematics,
    ],
)
```

Figura 4.32: Nodo di move_group e rviz

```
# Blocco D: Telecamera RealSense
launch_realsense = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(realsense_dir, 'launch', 'rs_launch.py')]),
    launch_arguments={
        'enable_sync': 'true',
        'pointcloud.enable': 'true',
        'align_depth.enable': 'true',
        'camera_namespace': '',
        'base_frame_id': 'camera_link'
    }.items()
)
```

Figura 4.33: Nodo della camera

e il frame di riferimento della camera, applicando sia una traslazione sia i quaternioni di rotazione;

```
# Blocco E: Static TF Eye-in-Hand
eye_in_hand_tf = Node(
    package="tf2_ros",
    executable="static_transform_publisher",
    name="eye_in_hand_calibration",
    arguments=[
        "--frame-id", "tool0",
        "--child-frame-id", "camera_camera_link",
        "--x", "-0.032", "--y", "-0.18", "--z", "-0.015",
        "--qx", "-0.00926484", "--qy", "-0.000520518", "--qz", "0.0406417", "--qw", "0.999131",
    ],
)
```

Figura 4.34: Nodo di calibrazione della camera

- *MoveIt servo*, Figura 4.35: viene istanziato il *servo_node_main* per il visual servoing utilizzando il file “*ur_servo_config.yaml*” precedentemente creato e contenente i limiti di velocità e accelerazione raggiungibili dal robot.

Sequenzializzazione dell'avvio

Il problema principale di avviare tutti i nodi con un unico file è l'inizializzazione concorrente: il driver hardware del robot ha bisogno di un intervallo di tempo non trascurabile per avviarsi e deve essere avviato per primo, quindi, se il nodo

```
# Blocco F: MoveIt Servo Node
servo_node = Node(
    package='moveit_servo',
    executable='servo_node_main',
    name='servo_node',
    parameters=[
        robot_description,
        robot_description_semantic,
        robot_description_kinematics,
        joint_limits_yaml_content,
        servo_params,
    ],
    output='screen'
)
```

Figura 4.35: Nodo di MoveIt Servo

move_group o il modulo *servo* cercassero di avviarsi prima che il primo fosse attivo, fallirebbero, portando al blocco del sistema.

Per risolvere questo problema, è stato introdotto un *TimerAction* da 10.0 secondi, in maniera tale che il driver del robot faccia in tempo ad attivarsi prima che venga eseguito il resto dei componenti, come si può vedere nella Figura 4.36.

```
return LaunchDescription([
    launch_ur,
    TimerAction(
        period=10.0,
        actions=[
            move_group_node,
            servo_node,
            rviz_node,
            launch_realsense,
            eye_in_hand_tf
        ]
    )
])
```

Figura 4.36: Avvio finale dei nodi

Avvio del file

Per avviare il file completo, è sufficiente utilizzare il seguente comando:

```
$ ros2 launch ur10e_gripper_moveit_config
    total_startup.launch.py
```

poi, una volta che si avvia il primo nodo, quello del driver del robot, abilitare l'external control dal teach pendant del manipolatore e aspettare l'avvio di tutti gli altri nodi.

4.8.1 Problematiche riscontrate

Durante la fase di avvio dei nodi, sono state affrontate e risolte diverse criticità, che verranno analizzate di seguito.

Problemi di parsing

Inizialmente, il sistema era stato configurato con la lingua italiana; tuttavia, questa impostazione ha generato un'anomalia: all'avvio di *MoveIt 2*, l'applicativo eseguiva il parsing del file di configurazione dei limiti cinematici del robot al percorso `"/opt/ros/humble/share/ur_moveit_config/config/joint_limits.yaml"`, ma, a causa di un bug di interpretazione dei dati di ROS 2 Humble, il punto, che nel file di configurazione è utilizzato come separatore decimale, non veniva riconosciuto come tale, a causa del fatto che in Italia il separatore decimale standard è la virgola. Di conseguenza, i valori numerici espressi in virgola mobile, come 5.0, sono stati convertiti dal parser in stringhe di testo, "5.0". Per questo motivo, visto che *MoveIt 2* trovava un tipo di dato testuale al posto di una variabile numerica di tipo *double*, ha sollevato un'eccezione. Per risolvere questa problematica, è stato riconfigurato il sistema impostando la lingua e la localizzazione del sistema inglese.

Conflitti di dipendenze tra NumPy e OpenCV

Si è riscontrato un problema legato alla gestione delle dipendenze in ambiente Python. Nello specifico, nello sviluppo basato su ROS 2 Humble su Ubuntu 22.04, lo standard è rappresentato dalla libreria *NumPy* nella sua versione 1.x; tuttavia, durante l'installazione del pacchetto *ultralytics* tramite il gestore *pip*, necessario per l'esecuzione di YOLO, è stata installata automaticamente l'ultima release disponibile di NumPy, ovvero la 2.4.6, incompatibile con ROS 2.

Il problema risiedeva nel pacchetto *cv_bridge*, che serve a convertire i messaggi immagine di ROS 2 in matrici compatibili con OpenCV. In particolare, la libreria *opencv-python* installata con *pip* richiedeva la versione aggiornata a NumPy 2, mentre *cv_bridge* e ROS 2 Humble la versione 1.x, generando un conflitto e portando al fallimento dell'inizializzazione dei nodi di visione.

Per risolvere questa incompatibilità, è stato necessario effettuare un *downgrade*, forzando l'installazione di versioni precedenti di questi pacchetti.

Per prima cosa, è stata installata una versione di *NumPy* appartenente alla release 1.x:

```
$ pip install "numpy<2"
```

Successivamente, è stata installata una distribuzione di *OpenCV* compatibile con la versione di NumPy appena installata:

```
$ pip install "opencv-python<4.9"
```

MoveIt 2

Un'altra problematica è stata riscontrata durante l'avvio del componente *move_group*: non era possibile per il sistema comunicare con il controller del robot, in particolare, il messaggio di errore era:

```
No action namespace specified for controller
ur_manipulator_controller through parameter
```



```
moveit_simple_controller_manager.  
ur_manipulator_controller.action_ns
```

La causa era un errore nel file *moveit_controllers.yaml*, che si trova al path */home/idea/ros2_ws/src/ur10e_gripper_moveit_config/config*, nel quale non era presente il campo “*action_ns*”, senza il quale MoveIt non sapeva a che *topic* inoltrare i comandi d'azione. Quindi, è stato necessario inserire la riga:

```
action_ns: follow_joint_trajectory
```

all'interno della voce “*ur_manipulator_controller*” nel file specificato.

Capitolo 5

Software di visione e controllo

In questo capitolo verrà analizzato nel dettaglio il software sviluppato in questo progetto; in particolare, verrà descritto il processo di *fine tuning* del modello YOLO e le fasi di addestramento che hanno permesso di ottimizzare la rete per il riconoscimento, la localizzazione e il tracking delle classi di oggetti selezionate, per poi continuare con il modo in cui sono stati risolti i task di *pick-and-place* e di *visual servoing*, sia per quanto riguarda il sistema di visione sia per quanto riguarda il controllo del robot.

5.1 Fine Tuning

Come introdotto nella Sezione 2.8, nel lavoro sviluppato in questa tesi la detection e il tracking dell'oggetto target sono stati possibili grazie all'utilizzo di un modello appartenente alla famiglia YOLO; nello specifico, si è optato per la variante *YOLO12n*. La scelta di questa specifica versione, ovvero la *nano* della release più recente, è dovuta al *trade-off* vantaggioso tra accuratezza computazionale e latenza offerto da tale modello: infatti, grazie alla sua architettura leggera è possibile ridurre di molto i tempi di inferenza. Per maggiori dettagli è possibile consultare il sito ufficiale [54].

Nonostante i loro molteplici lati positivi, i modelli YOLO pre-addestrati e disponibili sul sito di Ultralytics sono stati pensati per scopi generali. Per questo motivo, per adattare le capacità della rete neurale al contesto di applicazione specifico, è stata necessaria una fase di *fine tuning*. Nelle sezioni seguenti verranno descritte nel dettaglio le procedure di generazione e labeling del dataset personalizzato e la fase di addestramento del modello.

Trattandosi di una versione base, il caso di studio del lavoro presentato è stato circoscritto a detection, tracking, presa e servoing di un set limitato di oggetti noti: sono stati realizzati sei oggetti target, illustrati nella Figura 5.1, tramite Sintetizzazione Laser Selettiva (SLS). Di questi oggetti, due sono cubi, due sono parallelepipedi e due sono sfere troncate alla base, al fine di garantirne la stabilità sul piano di lavoro. Inoltre, sono stati divisi tra le colorazioni grigia e rossa, tre elementi ognuna, per valutare la robustezza del sistema rispetto a variazioni di colore.

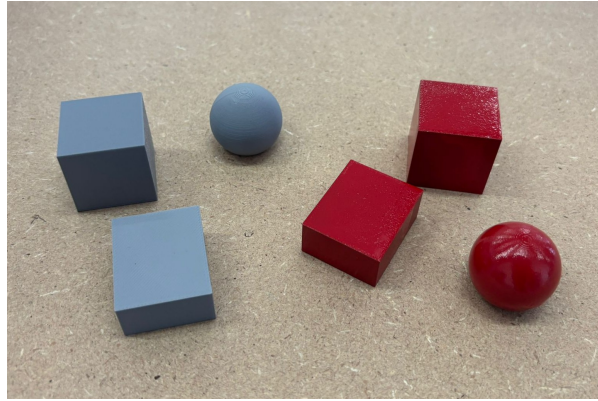


Figura 5.1: Oggetti realizzati per l'applicazione

Come mostrato nella Figura 5.2, nella configurazione di chiusura della pinza, la distanza tra le griffe è pari a 29 mm. Poiché il tool utilizzato per la presa è di tipo pneumatico, ovvero è caratterizzato dall'applicazione di una pressione costante, afferrare componenti più grandi di questo valore avrebbe comportato il rischio di rottura dei target stessi, rendendo necessario dimensionare gli oggetti personalizzati in funzione di questo vincolo.

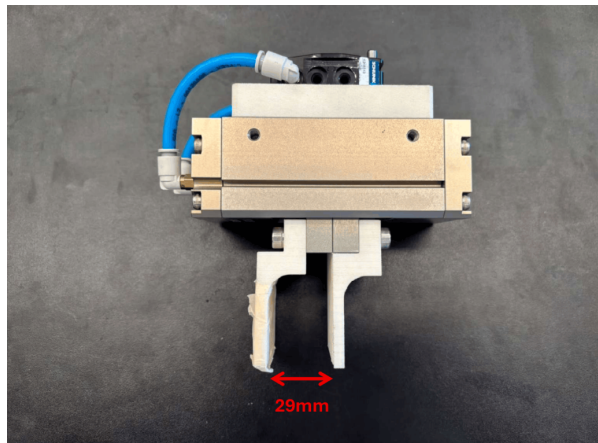


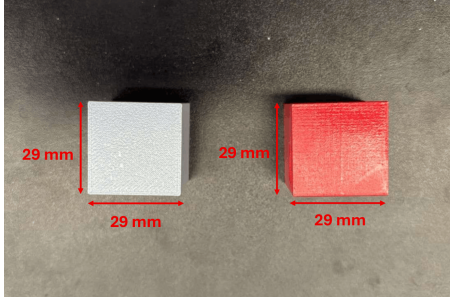
Figura 5.2: Misura della chiusura della pinza

In particolare, nella Figura 5.3 vengono illustrate le dimensioni dei vari oggetti:

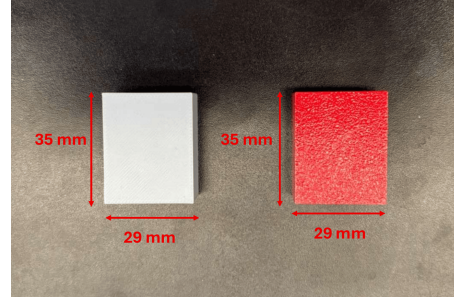
- i lati dei cubi, mostrati nella Figura 5.3a, misurano 29 mm;
- i lati dei parallelepipedi, illustrati nelle Figure 5.3b, 5.3c e 5.3d, misurano 35 mm, 29 mm e 15 mm;
- il diametro delle sfere misura 29 mm, Figura 5.3e, mentre la loro altezza è di 20 mm, Figura 5.3f.

Per quanto riguarda i parallelepipedi, si osserva che l'operazione di afferraggio è vincolata all'orientamento dell'oggetto, risultando quindi possibile solo su

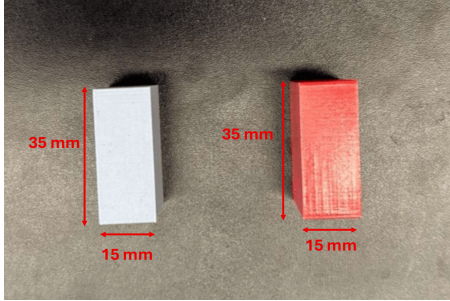
due delle tre facce. Nello specifico, la presa può essere eseguita sulla faccia da 35×29 mm (Figura 5.3b) e su quella da 29×15 mm (Figura 5.3d); invece, la presa sulla superficie da 35×15 mm (Figura 5.3c) non risulta attuabile.



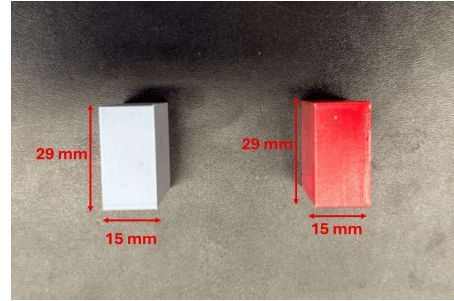
(a) Cubo



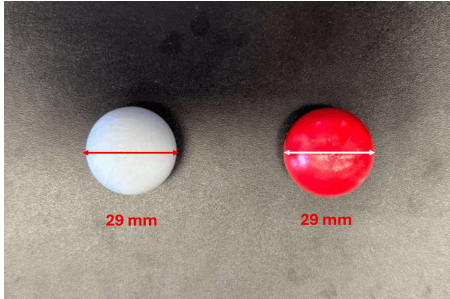
(b) Faccia maggiore del parallelepipedo



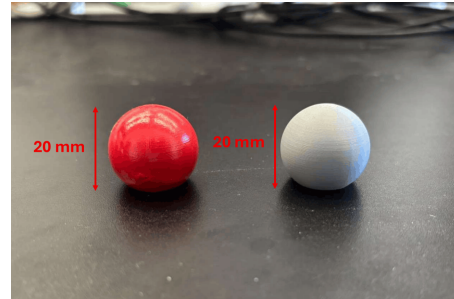
(c) Faccia intermedia del parallelepipedo



(d) Faccia minore del parallelepipedo



(e) Sfera in orizzontale



(f) Sfera in verticale

Figura 5.3: Dimensioni dei sei oggetti realizzati

5.1.1 Acquisizione del dataset e labeling

Una volta definiti i vincoli fisici degli oggetti, si è passato alla creazione di un dataset personalizzato per l'addestramento del modello.

Il processo di raccolta delle immagini è stato ingegnerizzato utilizzando un programma di movimento, implementato sul *teach pendant* del robot, che definiva 40 pose diverse per il manipolatore: la strategia di raccogliere immagini in così tante pose differenti tra loro ha impedito che la rete neurale sviluppasse un *bias* legato al background, facendo in modo che il modello non associas-

se la presenza dell'oggetto esclusivamente al piano di supporto o allo sfondo, garantendone così una maggiore capacità di generalizzazione.

Inoltre, è stato sviluppato uno script dedicato in linguaggio Python per ottimizzare i tempi e automatizzare il processo di cattura delle immagini. Questa procedura prevedeva come prima cosa l'avvio dello script; in seguito, quando il robot raggiungeva la posa, il software gestiva in modo automatico sia la creazione delle 6 directory organizzate, una per ogni classe di oggetto, sia il salvataggio del frame nella cartella corrispondente alla pressione del tasto "s".

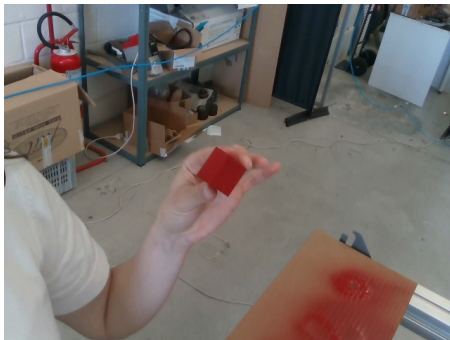
Per ogni tipologia di oggetto sono state acquisite 5 immagini dalla camera RealSense per ogni posa definita, per un totale di 1200 immagini, di cui 200 per target. Al fine di rendere il modello robusto rispetto a oclusioni parziali tipiche dell'ambito di applicazione, all'interno del dataset è stata inclusa intenzionalmente anche la mano dell'operatore durante le fasi di manipolazione e posizionamento. Nella Figura 5.4 vengono mostrati degli esempi di immagini che fanno parte del dataset creato.



(a) Cubo grigio



(b) Cubo rosso e sfera rossa



(c) Parallelepipedo rosso



(d) Parallelepipedo grigio

Figura 5.4: Esempio di immagini contenute nel dataset

Una volta terminata la fase di raccolta dei frame, si è passati alla fase di annotazione, o "*labeling*", che è stata eseguita sfruttando la piattaforma cloud *Roboflow* (il cui sito ufficiale è [55]), uno strumento per la gestione dei dataset di computer vision. In questa maniera, ogni immagine è stata analizzata singolarmente e sono stati definiti a mano i *bounding box* (semplici, non orientati) che racchiudevano gli oggetti target presenti nella scena. Nella Figura 5.5 si

può vedere un esempio del processo di labeling di un'immagine in cui sono presenti 4 oggetti: una sfera grigia, un parallelepipedo rosso, un cubo rosso e un parallelepipedo grigio.

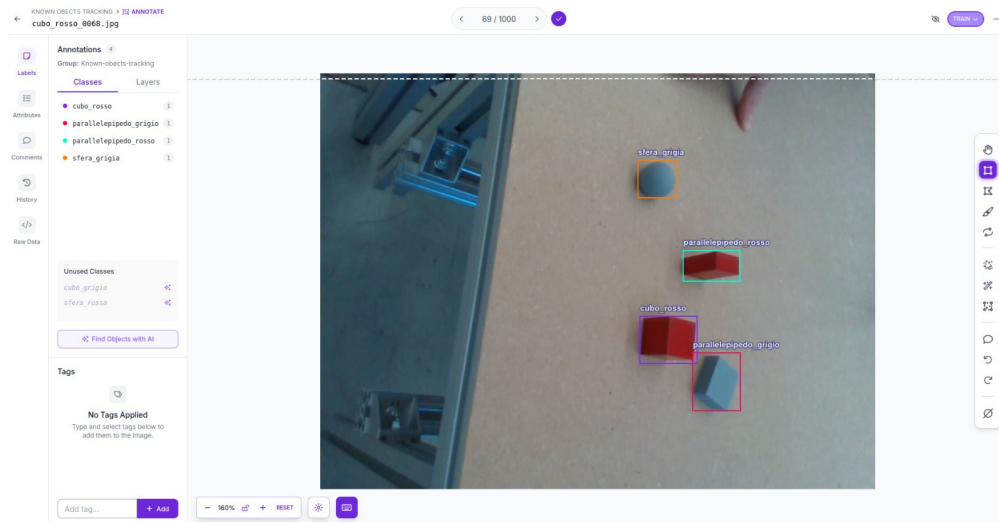


Figura 5.5: Procedura di labeling delle immagini

L'utilizzo di Roboflow ha consentito inoltre di esportare l'intero pacchetto di dati direttamente nel formato richiesto dall'architettura YOLO, facilitando ulteriormente l'intero processo.

5.1.2 Addestramento dei modelli e studio comparativo

Una volta terminata la fase di labeling del dataset, si è passato all'addestramento e allo studio comparativo di diversi modelli della famiglia YOLO, con l'obiettivo di capire quale garantisca il miglior compromesso tra accuratezza di detection ed efficienza in fase di inferenza e che, quindi, fosse il migliore per il progetto sviluppato.

Come prima cosa, è stato registrato un video in cui venivano mostrati i 6 oggetti disposti in diverse configurazioni, sia fermi sia in movimento, per simulare le condizioni operative reali dello spazio di lavoro del robot. Questo video, utilizzato nella fase di inferenza, è stato acquisito mediante la camera Intel RealSense D435 in configurazione eye-in-hand, in maniera tale da valutare le prestazioni dei modelli addestrati in uno scenario d'uso reale, con una risoluzione di 1280×720 e un frame rate di 30 FPS.

Inoltre, è stato salvato nel formato *AVI* (*Audio Video Interleave*), poichè tale formato garantisce la massima compatibilità e stabilità con le librerie di elaborazione immagini, in particolare con *OpenCV*, e comprime i dati permettendo comunque di mantenere un'elevata qualità del video, preservando così la nitidezza dei bordi degli oggetti ed evitando problematiche come il motion blur, che degraderebbero le prestazioni della rete.

La fase effettiva di addestramento e confronto dei modelli YOLO è stata effettuata su una workstation dedicata, diversa rispetto a quella su cui poi è stato

sviluppato il progetto, di cui si parla nell'introduzione del Capitolo 4. Questa scelta è dovuta ai vincoli hardware di questo secondo PC, di cui si parla nella Sezione 4.3.6, che ha portato a dover eseguire le reti neurali direttamente sulla CPU. Ovviamente, in sede di addestramento, questo comporta tempi molto più elevati; per questo motivo, si è scelto di sfruttare un computer con una GPU dedicata che garantisca prestazioni molto più elevate. Nella Tabella 5.1 sono riportate le specifiche hardware del computer dedicato alla fase di addestramento e inferenza.

Componente	Specifica Tecnica
CPU	AMD Ryzen 9 5950X, 16-Core, 32 thread
RAM	48 GiB
GPU dedicata	NVIDIA GeForce RTX 3060
Architettura di Accelerazione	NVIDIA CUDA Cores & Tensor Cores
SO	Linux, Kernel 6.14.0-37-generic, X11, Qt 6.11.1

Tabella 5.1: Specifiche dei componenti del PC dedicato all'addestramento dei modelli YOLO

Lo studio comparativo ha coinvolto un set composto da 8 modelli della famiglia YOLO, ovvero le varianti *nano* (*n*) e *small* (*s*) delle versioni *YOLOv10*, *YOLO11*, *YOLO12* e *YOLO26*.

Per rendere il processo automatizzato, è stato sviluppato uno script in linguaggio Python che esegue in modo sequenziale una pipeline per ognuno degli 8 modelli. La prima fase di questa pipeline è la fase di *training*, in cui ogni modello è stato riaddestrato sul dataset personalizzato con 100 *epoche*, utilizzando una risoluzione dell'immagine impostata a 640×640 pixel, un *batch* di 32 campioni e delegando l'esecuzione alla GPU dedicata.

La seconda fase della pipeline è quella dell'*estrazione dei pesi ottimali*, per cui, al termine di ciascuna esecuzione, lo script isola automaticamente il file dei pesi che ha registrato le migliori metriche di accuratezza sul set di validazione.

Completato l'addestramento, si passa alla fase di *inferenza* sul video precedentemente acquisito, denominato "*video_inference.avi*".

Nella Figura 5.6 si possono vedere le metriche di training di ciascun modello, che forniscono un quadro chiaro delle prestazioni delle diverse topologie di rete. Da questa immagine può notare che:

- i grafici delle funzioni di costo, sia del set di training sia del set di validation, mostrano una discesa ripida nelle prime epoche e subito dopo una stabilizzazione: in particolare, la *Box Loss* e la *Class Loss* convergono ai valori minimi. Questo comportamento, insieme alla somiglianza presentata dalle curve di addestramento e validazione, dimostrano che i modelli possiedono una buona capacità di generalizzazione;
- *precision* e *recall* mostrano una crescita rapida verso i valori massimi già entro la prima metà dell'addestramento, dimostrando che le reti apprendono a estrarre le feature dominanti in tempi brevi;

- le metriche di $mAP@50$ (*mean Average Precision*) e di $mAP@50-95$ registrano un incremento iniziale seguito dalla saturazione nell'ultimo quarto del training. La stabilità di questi valori nelle epoche finali conferma che le 100 epoche sono state un parametro ottimale per consentire l'ottimizzazione dei modelli.

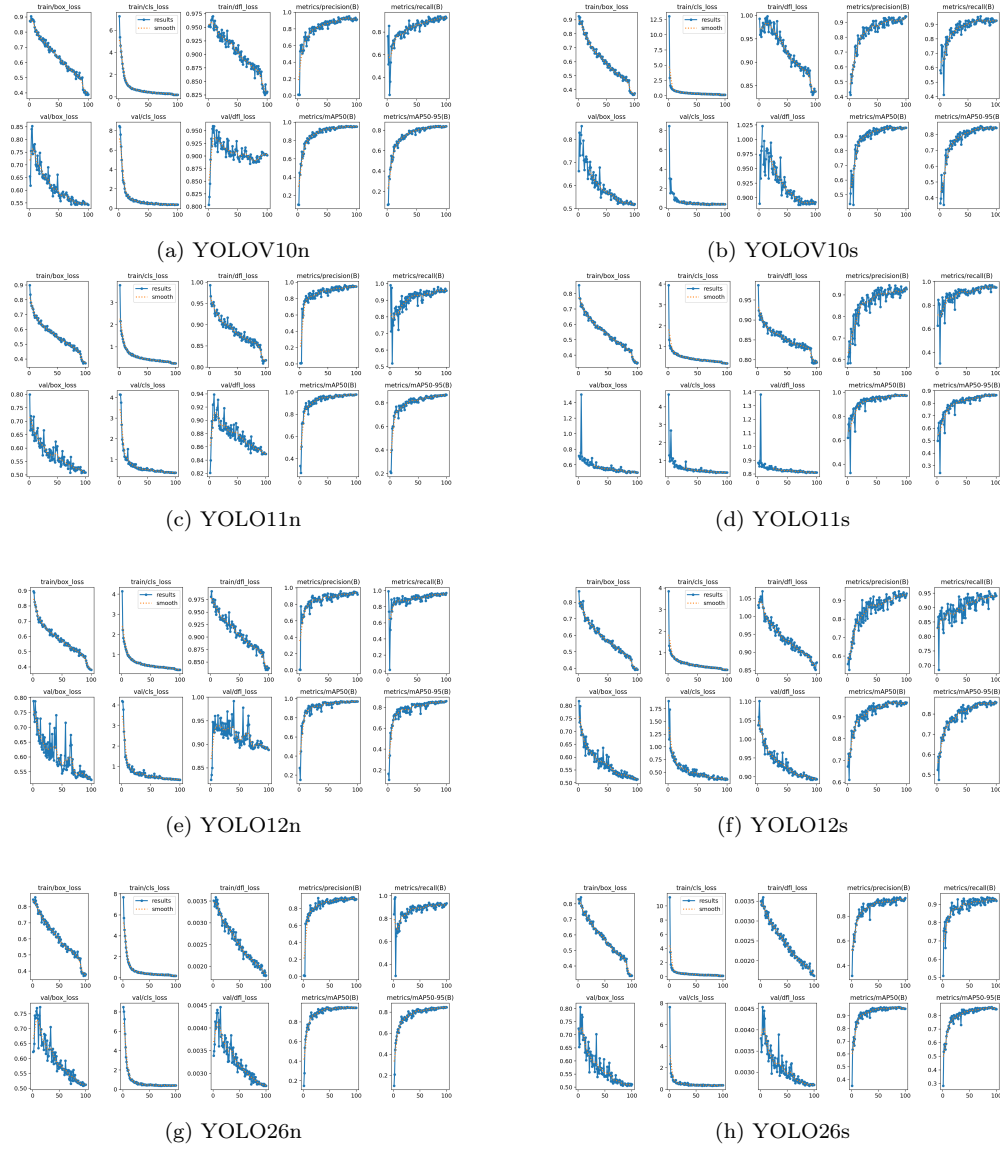


Figura 5.6: Metriche di training dei modelli a confronto

La Figura 5.7 mostra le matrici di confusione normalizzate dei modelli messi a confronto, mentre la Tabella ?? mostra solamente i risultati principali di queste stesse matrici, riportando i valori presenti nelle diagonal, ovvero le classificazioni avvenute correttamente, e i principali errori di classificazione. Dalla figura e dalla tabella emergono delle considerazioni importanti:

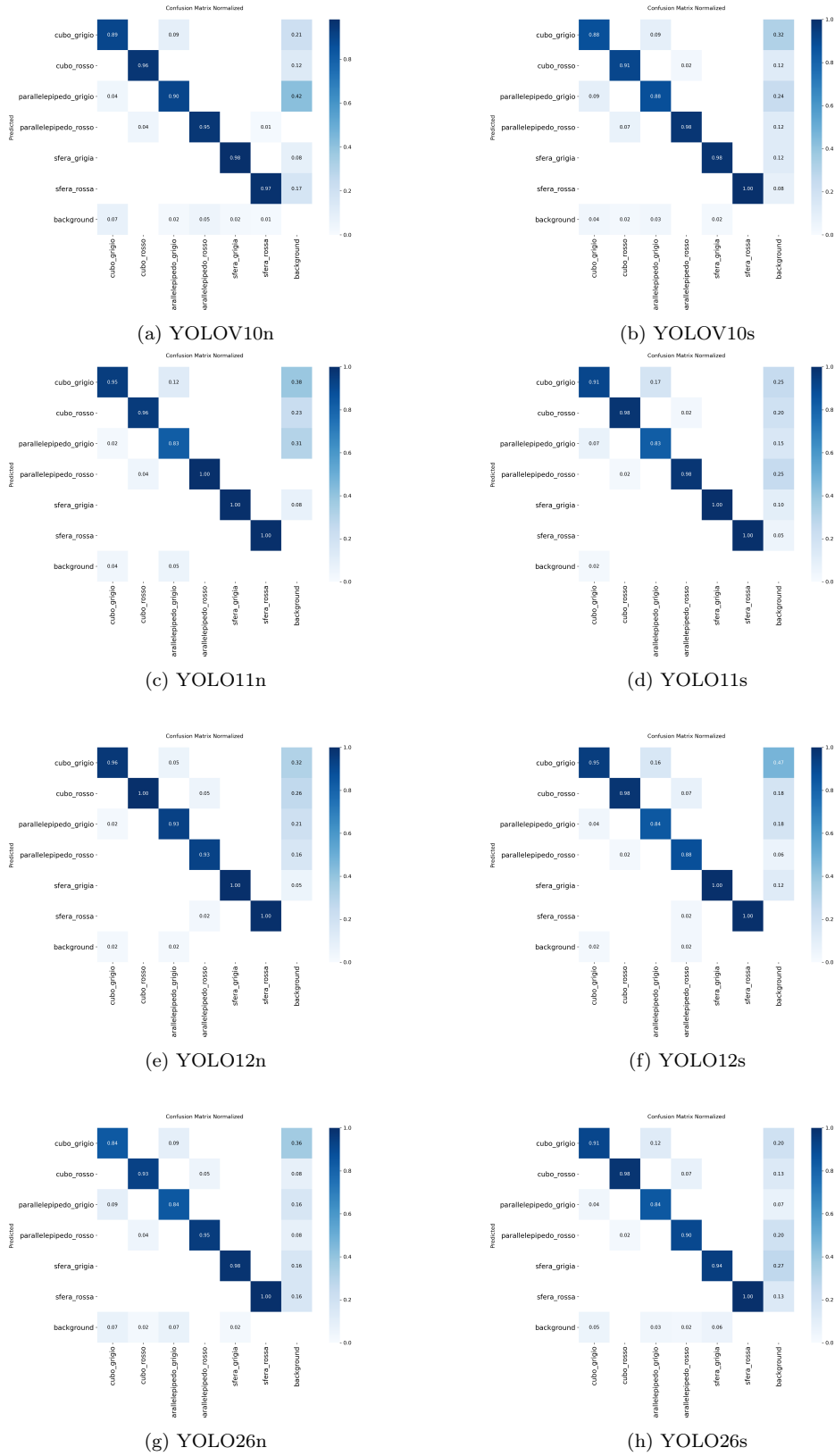


Figura 5.7: Matrici di confusione normalizzate dei modelli a confronto

	Modelli YOLO analizzati							
	v10n	v10s	11n	11s	12n	12s	26n	26s
<i>Classificazioni corrette</i>								
Cubo Grigio	0.89	0.88	0.95	0.91	0.96	0.95	0.84	0.91
Cubo Rosso	0.96	0.91	0.96	0.98	1.00	0.98	0.93	0.98
Parallelepipedo Grigio	0.90	0.88	0.83	0.83	0.93	0.84	0.84	0.84
Parallelepipedo Rosso	0.95	0.98	1.00	0.98	0.93	0.88	0.95	0.90
Sfera Grigia	0.98	0.98	1.00	1.00	1.00	1.00	0.98	0.94
Sfera Rossa	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00
<i>Principali errori di classificazione</i>								
Vero Parall. Grigio → Predetto Cubo Grigio	0.09	0.09	0.12	0.17	0.05	0.16	0.09	0.12
Vero Parall. Rosso → Predetto Cubo Rosso	-	0.02	-	0.02	0.05	0.07	0.05	0.07
Vero Background → Predetto Cubo Grigio	0.21	0.32	0.38	0.25	0.32	0.47	0.36	0.20
Vero Background → Predetto Parall. Grigio	0.42	0.24	0.31	0.15	0.21	0.18	0.16	0.07

Tabella 5.2: Riassunto delle matrici di confusione normalizzate dei modelli

- le matrici presentano valori di confidenza elevati lungo la diagonale, situazione che indica che le reti hanno delle ottime prestazioni e che hanno imparato a discriminare bene le diverse classi;
- le varianti *small*, in quanto modelli più complessi rispetto alle varianti *nano*, presentando valori mediamente più elevati lungo la diagonale, sono più accurate;
- in diverse matrici della griglia è presente un tasso d'errore non nullo dovuto alla confusione tra la classe *parallelepipedo* e *cubo* della stessa colorazione.

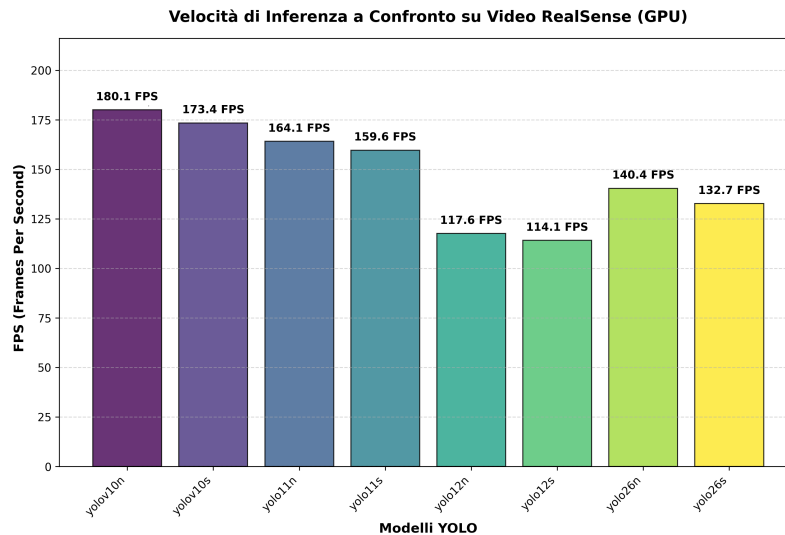


Figura 5.8: Confronto tra modelli - FPS di inferenza

Infine, la Figura 5.8 illustra i risultati dello studio comparativo condotto sulla velocità di inferenza, espressa in FPS, dei vari modelli, effettuato utilizzando il video precedentemente acquisito. Il dettaglio principale che si nota esaminando

questa immagine è che tutti i modelli appartenenti alla versione *small* della stessa release, essendo delle reti più complesse rispetto a quelle delle versioni *nano*, pur avendo delle prestazioni migliori in termini di accuratezza, hanno velocità minori.

Alla fine, la scelta del modello da integrare nel sistema di visione è ricaduta su *YOLO12n*. Questo perché, nonostante dallo studio sia risultato che architetture precedenti siano caratterizzate da velocità superiori, *YOLO12n* presenta una media di 117.6 FPS sulla GPU dedicata, un valore ampiamente abbondante rispetto al frame rate di acquisizione della camera RealSense utilizzata. Inoltre, *YOLO12n* risulta più accurata rispetto agli altri modelli, rappresentando il perfetto punto di trade-off tra queste due variabili.

5.2 Pick-and-place statico

Il sistema di *pick-and-place* ha lo scopo di individuare un oggetto target (appartenente a una delle 6 classi predefinite) disposto in maniera arbitraria sul piano di lavoro, stimarne la posa tridimensionale e controllare il movimento del robot per eseguire una pipeline predefinita di operazioni: il centraggio rispetto all'oggetto a 33 cm di distanza, la presa, il trasferimento e il rilascio all'interno di una scatola.

5.2.1 Sistema di visione

È stato creato uno script Python, chiamato "*vision_node.py*", al fine di gestire la percezione dello spazio operativo e la stima della posa degli oggetti: questo modulo ha lo scopo di elaborare i flussi di dati generati dalla camera, estraendo le coordinate del centro geometrico del target e l'orientamento dello stesso (*yaw*) sul piano di lavoro sfruttando il modello YOLO riaddestrato (Sezione 5.1).

Il nodo della camera manda l'ultimo frame RGB acquisito dalla scena sotto forma di messaggio ROS 2; all'arrivo di questo frame, la matrice di pixel corrispondente viene convertita in un array NumPy mediante la libreria *CvBridge* e sottoposta alla fase di inferenza da parte della rete neurale. Il framework restituisce la classe di appartenenza dell'oggetto e le coordinate del bounding box che lo circonda, associandovi anche un indice di confidenza: se questo indice supera la soglia minima, impostata a 0.6 (60%), viene estratto il centro geometrico del bounding box espresso in pixel (u, v) dell'immagine.

Sfruttando il centro geometrico, viene isolata una *regione di interesse* (a cui d'ora in poi ci si riferirà con "*ROI*") di 5×5 pixel attorno a questo. Partendo da questa ROI, viene calcolata la mediana della profondità, ovvero la distanza del centro dalla camera a cui si trova il target, utilizzando la mappa di profondità estratta dal sensore. La scelta di utilizzare la mediana è causata dalla volontà di evitare outliers e di ottenere un valore stabile.

Successivamente, applicando le equazioni del *modello Pin-Hole* e utilizzando i parametri intrinseci del sensore, quali le distanze focali (*self.camera_info.k[0]*, *self.camera_info.k[4]*) e punti principali (*self.camera_info.k[2]* e *self.camera_info.k[5]*) estratti dal topic *CameraInfo*, vengono eseguite la conversione della

distanza z_c da mm a m e la de-proiezione spaziale per calcolare le coordinate cartesiane reali x_c e y_c come mostrato nella Figura 5.9:

```
z_c = depth_mm / 1000.0
x_c = (u - self.camera_info.k[2]) * z_c / self.camera_info.k[0]
y_c = (v - self.camera_info.k[5]) * z_c / self.camera_info.k[4]
```

Figura 5.9: Conversione delle coordinate del centro geometrico dell'oggetto

Oltre a queste tre coordinate, il nodo di visione include nel messaggio che invia al sistema di controllo anche l'orientamento dell'oggetto rispetto al piano di lavoro (espresso in quaternione), corrispondente all'angolo di cui dovrà ruotare il polso del manipolatore per posizionare la pinza per garantire una presa efficace. In questa fase, la pipeline di elaborazione dell'immagine e di stima dell'orientamento varia in base alla classe di appartenenza dell'oggetto.

Sfera rossa e sfera grigia

Nel caso della sfera, a causa della simmetria rotazionale dell'oggetto, l'orientamento di approccio della pinza risulta indifferente, garantendo comunque una presa efficace. Di conseguenza, l'angolo di *yaw* viene impostato a 0.0 rad, consentendo al manipolatore di eseguire la presa mantenendo l'orientamento di base del polso.

Cubo e parallelepipedo rossi e grigi

Nel caso in cui l'oggetto non presenti una simmetria sferica, ma appartenga alle classi dei cubi o dei parallelepipedi, è necessario calcolare l'angolo di orientamento.

Inizialmente, l'algoritmo isola la ROI RGB corrispondente al bounding box dell'oggetto, a cui, per evitare troncamenti dovuti a imperfezioni, viene applicato un *padding* di 10 pixel lungo i bordi.

Per isolare esclusivamente la faccia superiore del pezzo discriminandola dal piano d'appoggio, viene sfruttata la mappa di profondità associata alla ROI appena estratta: viene individuato il valore della distanza più vicino alla camera calcolando il 5° percentile dei pixel validi, strategia che previene l'influenza degli *outliers* o del rumore impulsivo del sensore, e vengono selezionati solamente i pixel che si trovano nel range di distanza compreso tra il valore ottenuto $\pm$ un δ di 3 mm.

La maschera di profondità viene applicata alla ROI RGB: l'immagine risultante subisce una conversione in scala di grigi e un filtraggio mediante un filtro gaussiano con kernel 5×5 ; dopo di che, viene utilizzata la funzione "*cv2.minAreaRect*" per calcolare il rettangolo ruotato ad area minima, ottenendo così l'angolo di inclinazione grezzo. I quattro vertici geometrici restituiti dal *bounding box orientato* vengono ordinati distinguendo in *top-left*, *bottom-right*, *top-right*, *bottom-left*, passaggio che serve per definire i vettori direzionali ortogonali dei lati dell'oggetto ed estrarne le lunghezze in pixel.

Per convertire queste lunghezze in dimensioni espresse in millimetri, viene ancora una volta utilizzato il *modello Pin-Hole* sfruttando la distanza della superficie precedentemente calcolata. Questa conversione viene eseguita nel seguente modo:

$$L_{maggiore} = \max \left(\frac{d_{alto_px} \cdot z}{f_{media}}, \frac{d_{sinistro_px} \cdot z}{f_{media}} \right) \quad (5.1)$$

$$L_{minore} = \min \left(\frac{d_{alto_px} \cdot z}{f_{media}}, \frac{d_{sinistro_px} \cdot z}{f_{media}} \right) \quad (5.2)$$

dove le grandezze sono definite nella seguente maniera:

- $L_{maggiore}, L_{minore}$: rappresentano, rispettivamente, la stima dei lati maggiore e minore dell'oggetto espressi in millimetri;
- d_{alto_px} : indica l'estensione della ROI, espressa in pixel, calcolata lungo l'asse verticale della camera;
- $d_{sinistro_px}$: indica l'estensione della ROI, espressa in pixel, calcolata lungo l'asse orizzontale della camera;
- z : rappresenta la distanza del target dalla camera espressa in millimetri;
- f_{media} : rappresenta la lunghezza focale media della camera espressa in pixel.

Una volta ottenute le dimensioni reali espresse in millimetri, viene eseguito un controllo basato sulle specifiche geometriche degli oggetti per determinare l'orientamento desiderato. Definendo dy_h e dx_h i lati maggiore e minore dell'oggetto individuato, si distingue il calcolo dell'angolo nelle diverse casistiche:

- *cubo*: essendo formato da facce identiche tra loro, è sempre possibile la presa; quindi, l'angolo viene calcolato in questa maniera:

$$\text{target_angle} = \text{np.degrees}(\text{np.arctan2}(\text{dy_h}, \text{dx_h}))$$

- *parallelepipedo - faccia grande da $35 \times 29 \text{ mm}$* : in questa configurazione il sistema approva la presa e calcola l'angolo vincolandolo lungo il lato minore, quello da 29 mm, come mostrato nella Figura 5.10;
- *parallelepipedo - faccia piccola da $29 \times 15 \text{ mm}$* : anche in questo caso il sistema ritiene la presa valida e l'angolo viene estratto identificando quale dei due vettori corrisponde al lato da 29 mm, come si vede ancora nella Figura 5.10;
- *parallelepipedo - faccia media da $35 \times 15 \text{ mm}$* : in questa configurazione la presa non è possibile, quindi, la posa viene marcata come non idonea (come si vede alla fine della Figura 5.10).

```

# Caso parallelepipedo
if "parallelepipedo" in object_type:
    # Caso A: Oggetto sdraiato in piano sulla superficie grande (Faccia 35x29 mm)
    if (30.0 <= lato_maggiore_mm <= 41.0) and (23.0 <= lato_minore_mm <= 33.0):
        is_valid_grasp = True
        stato_faccia = "Faccia 35x29 [PRENDO SUL LATO DA 29]"
        # Determinazione del lato corretto su cui posizionare la pinza
        if size_h_mm < size_v_mm:
            angle_29mm = np.degrees(np.arctan2(dy_h, dx_h))
        else:
            angle_29mm = np.degrees(np.arctan2(dy_v, dx_v))
        target_angle = angle_29mm
    # Caso B: Oggetto di coltello sulla faccia piccola (Faccia 29x15 mm)
    elif (24.0 <= lato_maggiore_mm <= 33.0) and (10.0 <= lato_minore_mm <= 22.0):
        is_valid_grasp = True
        stato_faccia = "Faccia 29x15 [PRENDO SUL LATO DA 29]"
        if size_h_mm > size_v_mm:
            angle_29mm = np.degrees(np.arctan2(dy_h, dx_h))
        else:
            angle_29mm = np.degrees(np.arctan2(dy_v, dx_v))
        target_angle = angle_29mm
    # Caso C: Oggetto posizionato sulla faccia media (Faccia 35x15 mm). Presa non possibile
    elif (30.0 <= lato_maggiore_mm <= 41.0) and (10.0 <= lato_minore_mm <= 22.0):
        is_valid_grasp = False
        stato_faccia = "Faccia 35x15 [NON AFFERRABILE - RISCHIO COLLISIONE]"

```

Figura 5.10: Codice per l'estrazione dell'orientamento per il parallelepipedo

L'angolo così ottenuto è calcolato in riferimento all'asse x dell'immagine; quindi, visto che per la configurazione del robot è necessario un angolo espresso rispetto all'asse y della camera, questo viene ruotato di $+90.0^\circ$, spostando lo zero sull'asse desiderato. L'angolo finale viene poi confinato all'interno dell'intervallo di $[-90^\circ, +90^\circ]$ per garantire una simmetria nell'applicazione.

Per garantire una presa efficace e per prevenire fenomeni di instabilità, come il *flickering* del target o le oscillazioni della posa causate da variazioni della luminosità ambientale o dal rumore che caratterizza la camera, è stata implementata una strategia di stabilizzazione temporale basata su una finestra mobile. Nel dettaglio, è stato implementato un meccanismo per il quale le coordinate x_c, y_c, z_c e l'angolo di yaw calcolati sul frame corrente vengono inseriti in una coda; una volta acquisiti i dati di 10 frame viene calcolata la media aritmetica delle coordinate lineari e la media circolare trigonometrica per l'angolo. Solamente quando sono stati acquisiti tutti i frame ed è stata calcolata la media, il sistema di visione manda il messaggio con la posa dell'oggetto al sistema di controllo. Il contro di questa strategia è rappresentato dal ritardo introdotto dal tempo di acquisizione dei frame e dal tempo di calcolo, considerato però accettabile nel contesto di questo progetto.

Nelle Figure 5.11 e 5.12 vengono mostrati i risultati finali di questa pipeline di elaborazione, rispettivamente per il cubo grigio e per il parallelepipedo rosso. In particolare, le Figure 5.11a e 5.12a mostrano, da sinistra a destra la pipeline effettiva: l'oggetto rilevato tramite la rete neurale, la maschera RGB che isola l'oggetto allo scopo di calcolare la dimensione (in millimetri) dei lati maggiore e minore, la maschera di profondità che isola la superficie superiore del target e il bounding box orientato con la direzione ottenuta. Le Figure 5.11b e 5.12b, invece, mostrano il risultato finale della pipeline, evidenziando il centro geometrico dell'oggetto e l'orientamento che deve seguire la pinza per una presa efficace.

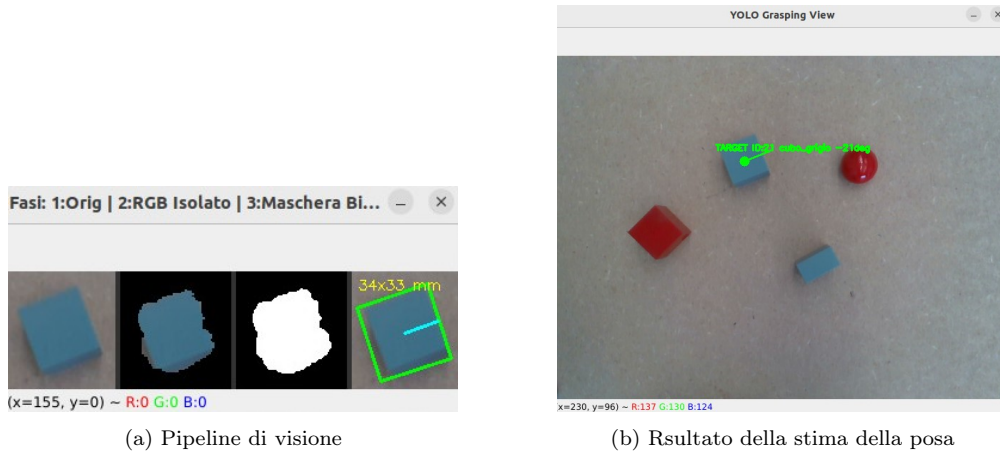


Figura 5.11: Stima della posa del cubo grigio

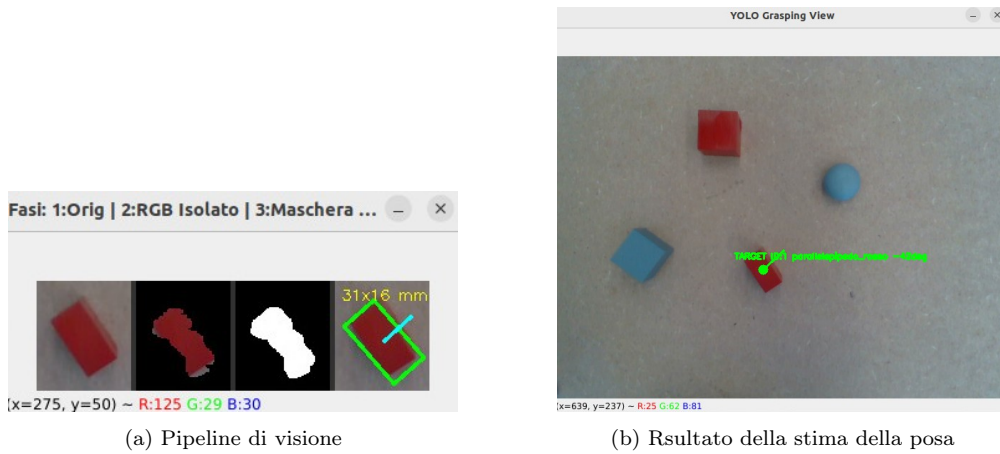


Figura 5.12: Stima della posa del parallelepipedo rosso

Conversione in quaternioni e pubblicazione del messaggio

Poiché l'angolo calcolato è espresso in gradi e l'architettura ROS 2 accetta solamente la rappresentazione in quaternioni, il nodo converte lo *yaw* in un quaternion standard $q = [x, y, z, w]$. Considerando che la rotazione avviene solo attorno all'asse z del frame della camera, la trasformazione diventa:

$$x = 0.0, \quad y = 0.0, \quad z = \sin\left(\frac{\theta}{2}\right), \quad w = \cos\left(\frac{\theta}{2}\right) \quad (5.3)$$

Così, i dati di posizione (x, y, z) e il rispettivo quaternion di rotazione attorno all'asse z vengono impacchettati in un messaggio *geometry_msgs/Pose Stamped*, in cui il campo *header.frame_id* è "camera_color_frame", assegnando al messaggio il *timestamp* corrente letto dal clock di sistema. Infine, viene pubblicato sul topic */target_pose*.

5.2.2 Sistema di controllo

Per il sistema di controllo del manipolatore e per il coordinamento delle sue traiettorie è stato creato un altro script Python, denominato “*robot_node.py*”.

Per evitare l’insorgenza di problematiche tipiche di applicazioni di robotica industriale guidata da visione artificiale, come la latenza di comunicazione o l’arresto del nodo di visione, che potrebbero causare il movimento del robot verso coordinate che non sono più sincronizzate con lo stato reale dell’ambiente di lavoro, all’interno della funzione di *callback* di ricezione del messaggio è stato implementato un meccanismo di *watchdog* temporale. Ogni volta che si registra un messaggio di posa, lo script controlla il *timestamp* associato alla coordinata ricevuta e calcola la differenza rispetto al tempo corrente del clock di sistema: se questa differenza supera una certa soglia, fissata a 0.5 secondi, il dato viene considerato obsoleto e non viene utilizzato, quindi il robot non si muove.

Per abilitare e utilizzare il tool di presa, il nodo di controllo comunica con il servizio I/O del controller UR. Nella Figura 5.13 è riportata la configurazione dei pin di output digitale, che sono stati rinominati per poter essere facilmente interpretabili e che permettono il *lock* (pin 2) e *unlock* (pin 3) della pinza, la sua *abilitazione* (pin 4), l’*apertura* (pin 0) e la *chiusura* (pin 1).

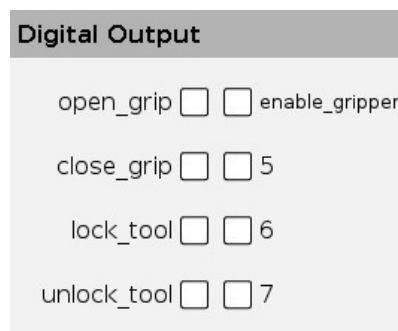


Figura 5.13: Pin di output digitale

Nella pipeline di esecuzione del task sono state definite due configurazioni articolari fisse espresse in gradi nello spazio dei giunti:

- *HOME_POS*: la posa iniziale in cui la camera è orientata verso il piano di lavoro; in questa posizione i giunti assumono i valori: 92.07° , -81.85° , -141.60° , -44.11° , 89.84° , -0.51° ;
- *DEST_POS*: rappresenta la posizione di rilascio dell’oggetto, quella raggiunta al termine del trasporto; in questa configurazione i giunti assumono i valori: 108.54° , -132.38° , -86.75° , -50.84° , 89.85° , 10.01° .

Il flusso effettivo di movimentazione del robot è suddiviso in tre fasi principali che vengono analizzate di seguito.

Inizializzazione e posizionamento in HOME_POS

La prima fase è quella di inizializzazione del sistema e di posizionamento nella posizione di home. Nel dettaglio, viene effettuata una verifica dello stato attuale

dei giunti: se il robot non si trova nella configurazione di HOME_POS, viene generata ed eseguita una traiettoria per raggiungerla con una velocità fissa di 0.3 rad/s. Parallelamente, viene verificato che la pinza di presa sia abilitata (DO 4 = True) e che sia aperta (DO 0 = True).

Approccio e centraggio

In questa fase il sistema attende di ricevere un dato di visione valido: grazie al meccanismo di *watchdog* temporale, il robot accetta solo misurazioni considerate recenti, ovvero con una latenza inferiore a 0.5 secondi, ignorando invece dati che risultano obsoleti.

Una volta acquisite le coordinate dell'oggetto rispetto al frame della camera, vengono calcolati il delta necessario per raggiungere la distanza di lavoro ottimale, impostata a 33 cm dal target, e il vettore di spostamento nel piano xy della camera, che permette di centrare l'obiettivo rispetto al centro geometrico dell'oggetto. Tramite la libreria *tf2*, il vettore di spostamento calcolato viene trasformato dal sistema di riferimento della camera, ovvero *camera_color_frame*, a quello del robot, ovvero *base_link*; dopo di che, questa nuova posa cartesiana viene tradotta in comandi di giunto, permettendo così al manipolatore di centrarsi sopra il target alla distanza desiderata. Questa fase è fondamentale per il corretto calcolo dell'orientamento dell'oggetto: a una distanza minore e a una visione perpendicolare corrispondono una maggiore nitidezza dell'immagine e calcoli più precisi.

Presca e rilascio

L'ultima fase, quella di presa e rilascio, è articolata a sua volta in una sequenza di operazioni, ovvero:

1. *ricezione della posa*: passati 6 secondi di attesa per la stabilizzazione del sistema, lo script estrae la posa definitiva dell'oggetto dall'ultimo messaggio ricevuto dal nodo di visione;
2. *allineamento sul piano xy*: viene posizionato il TCP sopra l'oggetto da afferrare effettuando un movimento solamente sul piano xy , ruotando contemporaneamente il giunto del polso per allineare la pinza lungo il lato su cui effettuare la presa;
3. *presa*: viene effettuato il movimento lungo l'asse z a velocità ridotta, ovvero 0.1 rad/s, per evitare che eventuali collisioni del manipolatore con il piano di lavoro avvengano a velocità troppo elevate. Una volta che il manipolatore è in posizione, viene modificato lo stato degli output digitali per chiudere la pinza (DO 1 = True);
4. *retreat*: dopo aver afferrato l'oggetto, il robot esegue un movimento in verticale di 30 cm, con il fine di evitare eventuali ostacoli circostanti;
5. *trasporto e rilascio*: il robot si muove verso la posizione di destinazione, DEST_POS, e modifica lo stato degli output digitali per aprire la pinza

(DO 1 = False e DO 0 = True), rilasciando così l'oggetto e completando il ciclo.

5.2.3 Funzionamento generale

Le Figure 5.14, 5.15 e 5.16 mostrano rispettivamente i diagrammi di sequenza della fase di posizionamento in home, della fase di acquisizione e processamento delle immagini e della pipeline di pick-and-place.

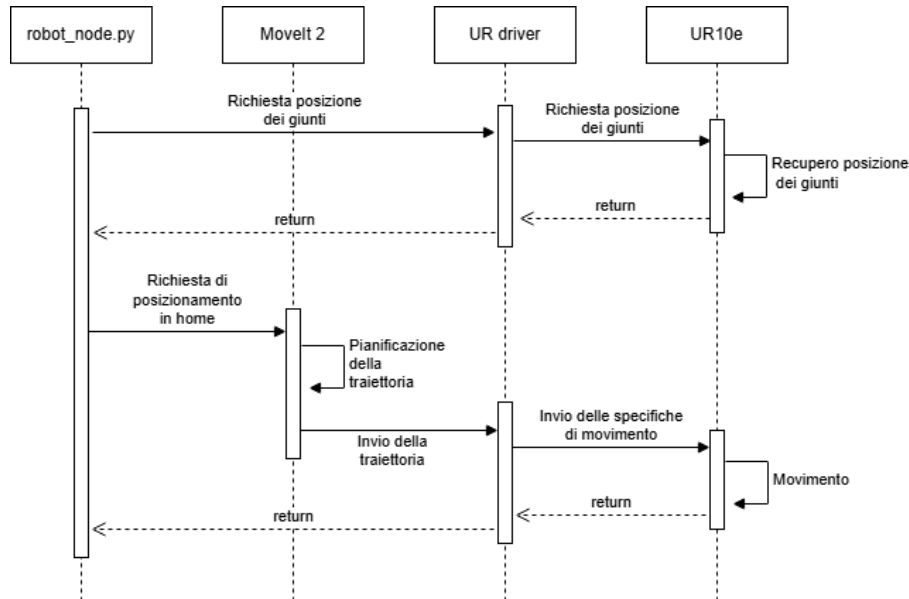


Figura 5.14: Pick-and-place - Diagramma di sequenza del posizionamento in home

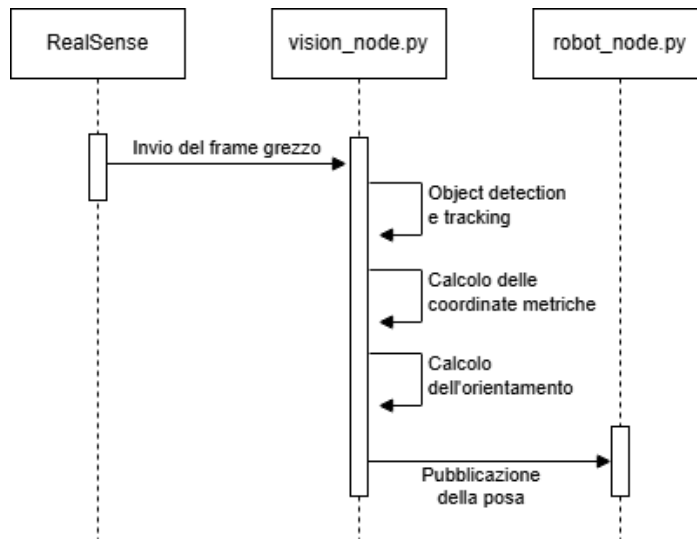


Figura 5.15: Pick-and-place - Diagramma di sequenza della pipeline di visione

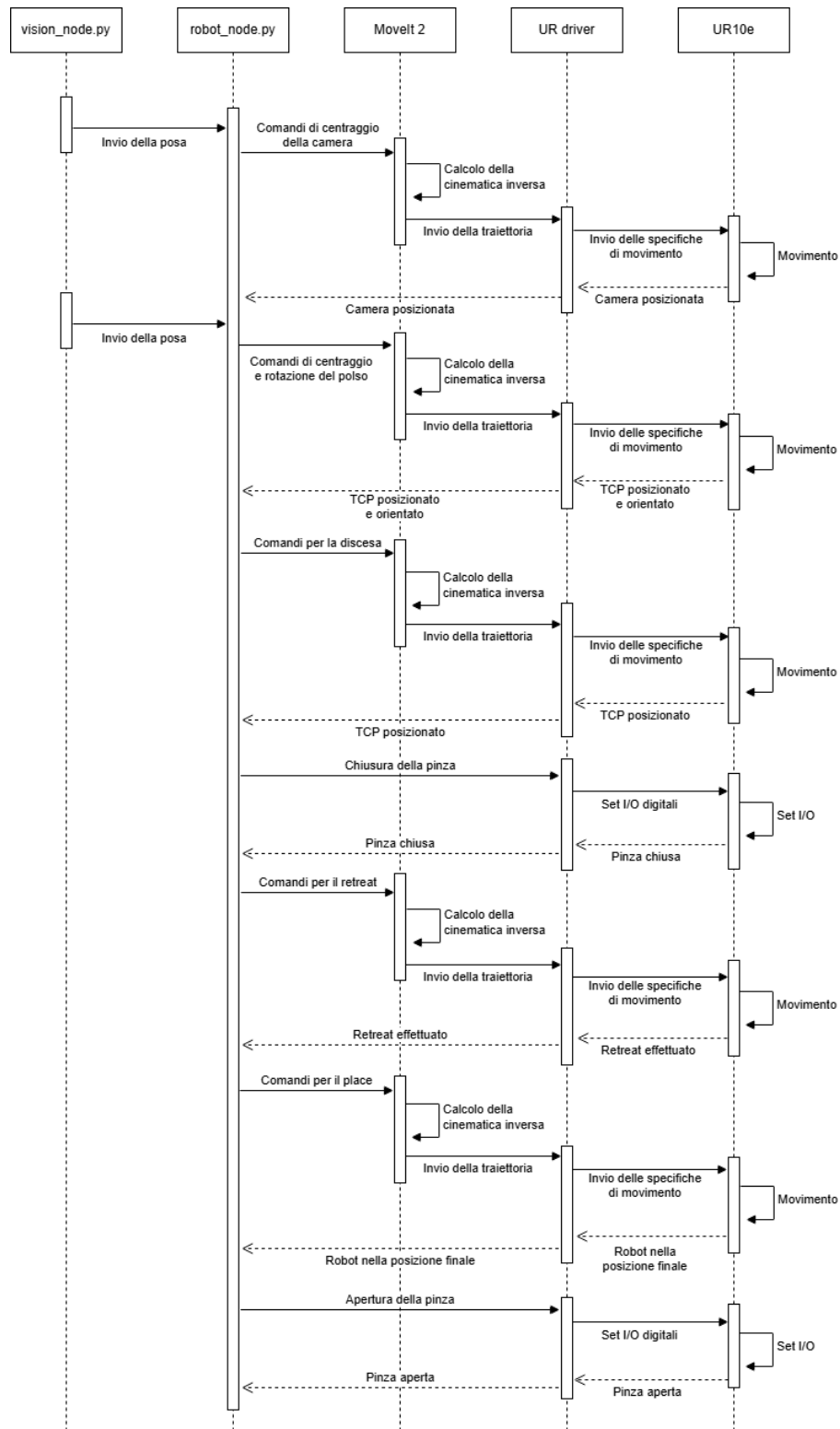


Figura 5.16: Diagramma di sequenza del pick-and-place

5.3 Visual servoing

Il task di *visual servoing* si inserisce nel contesto specifico dell'automazione industriale in cui gli oggetti si muovono sul piano di lavoro, ad esempio lungo nastri trasportatori, con lo scopo di seguire tale target ed emulare i suoi movimenti. Questa applicazione richiede un'architettura software in grado di operare a ciclo chiuso ad alta frequenza e in grado di minimizzare l'errore tra la posizione corrente del robot e la posizione del target in movimento. Nel caso particolare di questo progetto, si è configurato il controllo in maniera tale da mantenere una distanza costante di 33 cm tra la camera e l'oggetto.

5.3.1 Sistema di visione

Per l'implementazione del sistema di visione nel caso di visual servoing, è stato creato uno script Python dedicato, chiamato "*servo_vision_node.py*", che utilizza lo stesso modello sfruttato per il task di pick-and-place (di cui si parla nella Sezione 5.2.1) per la detection e il tracking dell'oggetto, introducendo però delle variazioni nella stima della distanza dal frame della camera e nella latenza di pubblicazione.

Per garantire una stima robusta della distanza della superficie superiore dell'oggetto dal frame della camera, viene sfruttata la mappa di profondità fornita dal sensore, isolando solamente i punti corrispondenti al bounding box del target rilevato. Sulla ROI così estratta, viene effettuata un'operazione di padding di 5 pixel lungo i bordi, in maniera tale da prevenire l'inclusione di punti appartenenti allo sfondo.

Sulla ROI ottenuta, vengono filtrati i pixel non validi, ovvero con valore nullo, ed estratto il 5° percentile della distribuzione delle profondità, approccio che permette di isolare efficacemente la superficie dell'oggetto più vicina alla camera scartando gli *outliers* e fornendo una coordinata z affidabile anche durante il movimento del robot o del target.

Il requisito fondamentale di questo task è la sincronizzazione tra il frame acquisito e il corrispondente comando di movimento del robot: qualsiasi ritardo in questa fase corrisponde a un ritardo nel sistema di controllo del manipolatore. Per questo motivo, a differenza della strategia utilizzata per il pick-and-place (Sezione 5.2.1), non viene calcolato l'angolo di orientamento dell'oggetto (lo *yaw*) e le coordinate (x, y, z) vengono pubblicate come messaggio di ROS 2 senza attendere l'acquisizione di 10 frames. Tuttavia, la strategia di deproiezione delle coordinate rimane la stessa, come mostrato nella Figura 5.9.

Nella Figura 5.17 viene mostrato il risultato della pipeline del sistema di visione applicato al task di visual servoing: nella Figura 5.17a si vede il caso di diversi oggetti sul piano di lavoro (ci si concentra sull'oggetto con l'ID minore) e camera a 50 cm, mentre nella Figura 5.17b si vede il caso di singolo oggetto nella scena e robot in movimento, con la camera che si trova a 30 cm dal target.

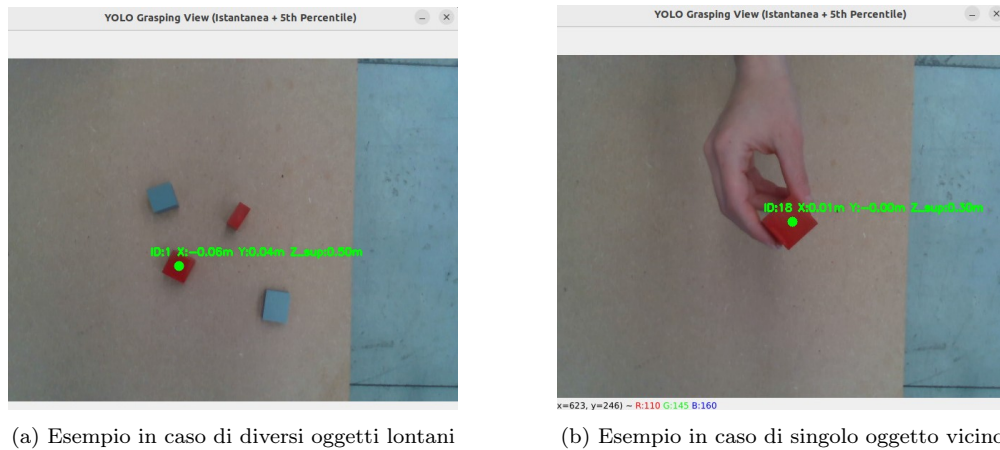


Figura 5.17: Stima delle coordinate per il visual servoing

5.3.2 Sistema di controllo

Per il sistema di controllo del task di *visual servoing* è stato creato uno script Python dedicato, chiamato “*servoing_test.py*”. In questa applicazione, a differenza del caso di pick-and-place, il robot non viene pilotato inviando vettori di posizione ai giunti, ma inviando flussi continui di velocità (messaggi di tipo *TwistStamped*) ad alta frequenza.

L’architettura sviluppata è basata su due pilastri fondamentali che verranno analizzati individualmente di seguito: una *macchina a stati finiti* e un *controllore PID*.

Macchina a stati finiti

Il funzionamento del manipolatore è stato scandito tramite l’utilizzo di una macchina a stati finiti. Tali stati, che corrispondono a diverse fasi della pipeline di controllo, sono:

1. *HOME*: all’avvio, il robot si configura in una posizione chiamata “*HOME_POS*”, la stessa utilizzata per il pick-and-place (Sezione 5.2.2), raggiunta la quale lo stato viene cambiato in *SEARCH*;
2. *SEARCH*: il manipolatore rimane in una condizione di *standby*, abilitando il nodo di *servo_node* tramite il servizio */servo_node/start_servo*, e rimandando in ascolto del topic di visione; quando il sensore rileva un oggetto e pubblica una posa, lo stato passa a *VISUAL_SERVOING*;
3. *VISUAL_SERVOING*: è il nucleo operativo del programma, in cui il robot esegue comandi ad alta frequenza, regolando le velocità in base al feedback del sensore. Nel caso in cui il target venga perso per più di un certo tempo limite, fissato a 3.0 s, l’errore di posizione si annulla, il sistema interrompe l’invio di velocità e lo stato viene riportato a *SEARCH*.

Controllore Proporzionale-Integrale

Mentre il sistema si trova nello stato di *VISUAL_SERVOING*, la logica di avvicinamento e inseguimento dell'oggetto è gestita da un controllore di tipo *Proporzionale-Integrale (PI)*, da cui è stata esclusa la componente derivativa per evitare l'amplificazione del rumore intrinseco ad alta frequenza dovuto alle stime del sistema di visione.

A ogni iterazione del ciclo di controllo, il software calcola il vettore $e(t) = [e_x, e_y, e_z]^T$, che rappresenta l'errore misurato come differenza tra il frame della camera e le coordinate del target, in cui l'errore lungo l'asse z viene calcolato considerando il vincolo di mantenere una distanza fissa di 33 cm dall'oggetto e di mantenere il target centrato rispetto al frame della camera.

La velocità lineare istantanea da inviare al manipolatore viene calcolata tramite la seguente equazione (applicata alle tre coordinate):

$$\mathbf{v}(t) = K_p \cdot \mathbf{e}(t) + K_i \int_0^t \mathbf{e}(\tau) d\tau \quad (5.4)$$

dove il guadagno proporzionale è settato $K_p = 5.0$ e modula la reattività con cui viene compensato l'errore; mentre il guadagno integrale, settato a $K_i = 0.1$, garantisce l'annullamento dell'errore a regime.

Contestualmente al controllore, sono stati utilizzati due filtri con lo scopo di rendere il task robusto e compatibile con i limiti di velocità e accelerazione del manipolatore industriale:

- *sistema anti-windup*: nel caso in cui il robot raggiungesse un limite meccanico o non potesse muoversi, il termine integrale continuerebbe a crescere, provocando instabilità (*windup*); per questo motivo, viene limitato l'accumulo massimo a $MAX_INTEGRAL = \pm 0.05$ per prevenire questo fenomeno;
- *saturazione della velocità*: se il vettore velocità uscente dal controllore supera la soglia massima fissata a $MAX_VEL = 0.15$ m/s, viene settato a questo valore, in maniera tale da mantenere il robot entro i suoi limiti operativi.

Pubblicazione del comando

Il controllore PI restituisce il vettore delle velocità espresso nel frame di riferimento della camera, ovvero *camera_color_frame*, ma, poiché *moveit_servo* richiede i comandi rispetto al frame della base del robot, ovvero *base_link*, viene effettuata una trasformazione tramite la libreria *tf2*.

Una volta calcolato, questo vettore viene incapsulato all'interno di un messaggio di tipo *TwistStamped*, in cui viene assegnato "*base_link*" come valore dell'attributo *frame_id* del messaggio. Infine, il messaggio viene pubblicato sul topic */servo_node/delta_twist_cmds*.

5.3.3 Funzionamento generale

La Figura 5.18 mostra il diagramma di sequenza del posizionamento del robot nella configurazione di home, analogo al caso del pick-and-place.

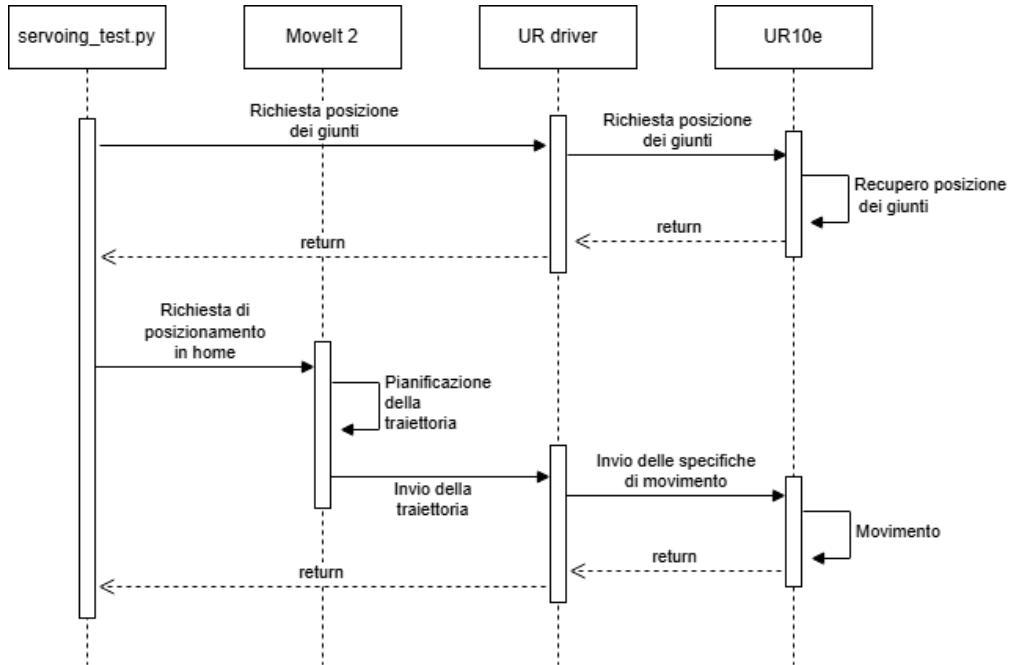


Figura 5.18: Visual servoing - Diagramma di sequenza del posizionamento in home

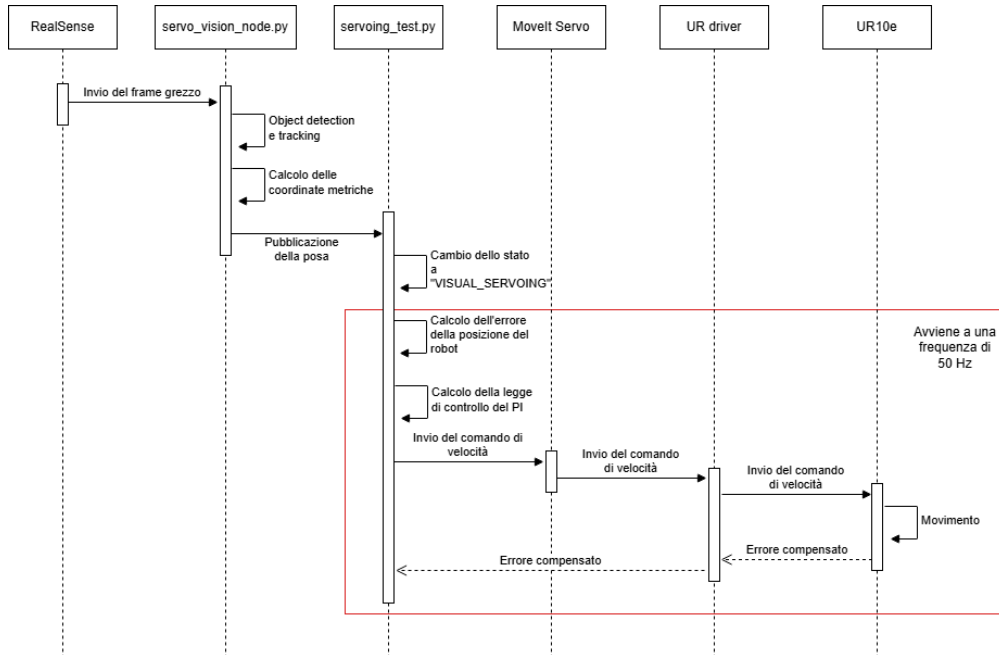


Figura 5.19: Visual servoing - Diagramma di sequenza dello stato di `"VISUAL_SERVOING"`

Nella Figura 5.19, invece, è mostrato il diagramma di sequenza della fase di inseguimento dell'oggetto: in particolare, il gruppo di azioni contenute nel riquadro rosso è quello che viene eseguito a una frequenza di 50 Hz e che permette al manipolatore di inseguire il target in maniera fluida.

5.4 Avvio degli script

Per l'esecuzione effettiva dei task di pick-and-place e visual servoing è necessario avviare i nodi ROS 2 come mostrato nella Sezione 4.8. Una volta attivi, bisogna aprire un terminale nella cartella del progetto che contiene gli script da avviare e, per il pick-and-place, digitare prima il comando:

```
$ python3 Camera/vision_node.py
```

aspettare che sia operativo e digitare il comando di avvio per il controllo del robot:

```
$ python3 Robot/robot_node.py
```

Invece, per il visual servoing, bisogna avviare il sistema di visione:

```
$ python3 Camera/servo_vision_node.py
```

e, una volta attivo, fare la stessa cosa per il sistema di controllo con il comando:

```
$ python3 Robot/servoing_test.py
```


Capitolo 6

Conclusioni e sviluppi futuri

Nel presente capitolo verranno analizzate le performance ottenute in sede di test dell'architettura sviluppata, traendo le conclusioni sul lavoro svolto e individuando i possibili sviluppi futuri del progetto.

Inoltre, si segnala che è stato fatto uso dell'LLM *Gemini* di Google [56] per la revisione stilistica, l'ottimizzazione del registro linguistico e l'attività di rephrasing di alcune sezioni della presente tesi.

6.1 Considerazioni

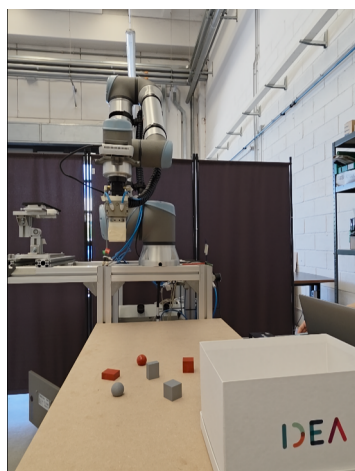
Il sistema sviluppato è stato testato all'interno di un ambiente di lavoro controllato utilizzando gli oggetti target creati apposta per il progetto.

6.1.1 Pick-and-place

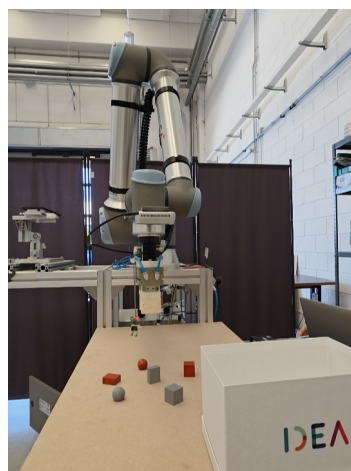
I test effettuati sul sistema di pick-and-place statico hanno avuto esiti molto positivi, dimostrando l'elevata affidabilità della pipeline di visione e di elaborazione dell'immagine: l'introduzione del filtro a media mobile su una finestra di 10 frame si è rivelata efficace per la rimozione del rumore del sensore, riuscendo a limitare le oscillazioni dell'angolo di *yaw* e garantendo una presa stabile e precisa.

La Figura 6.1 illustra la sequenza operativa eseguita durante la fase di test del sistema di pick-and-place statico. Nello specifico, questa sequenza è articolata nelle seguenti fasi:

- nella Figura 6.1a si vede il manipolatore nella posizione di *home*, fase preliminare in cui viene selezionato come target l'oggetto sul piano di lavoro che presenta l'ID minore, in questo caso, il cubo grigio;
- nella Figura 6.1b viene mostrato il posizionamento della camera in corrispondenza del centro geometrico della superficie superiore del target e alla distanza fissata di 33 cm;
- nella Figura 6.1c si vede il posizionamento del TCP al di sopra del cubo mantenendo la distanza di 33 cm e, contemporaneamente, ruotando il



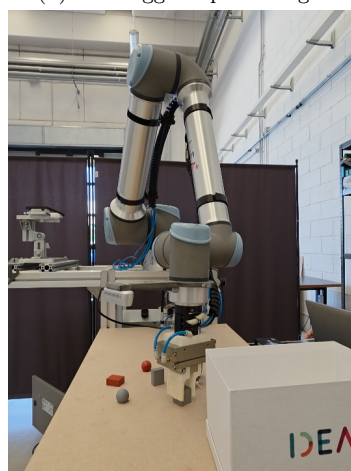
(a) Posizione di home



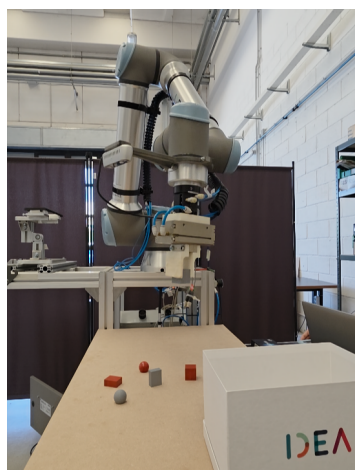
(b) Centraggio sopra il target



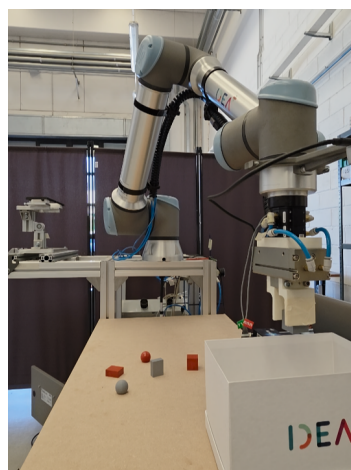
(c) Orientamento lungo il target



(d) Presa



(e) Retreat



(f) Rilascio

Figura 6.1: Sequenza di pick-and-place per il cubo grigio

giunto del polso per allineare la pinza rispetto all'angolo stimato dell'oggetto;

- la Figura 6.1d mostra la fase di presa;
- la Figura 6.1e illustra la fase di retreat verticale di 30 cm lungo l'asse z del robot;
- nella Figura 6.1f si vede la fase finale di rilascio del target nella posizione di destinazione.

Un risultato particolarmente importante legato a questa applicazione riguarda la robustezza dell'algoritmo di visione basato sul 5° percentile: rispetto a una semplice media, questo approccio ha permesso di rendere il calcolo della componente z dell'oggetto insensibile agli outliers e ai disturbi causati dall'illuminazione ambientale o dalla risoluzione della camera. Inoltre, l'integrazione del watchdog temporale ha diminuito di molto il rischio di collisioni, bloccando il movimento del robot nel caso di latenze superiori a 0.5 secondi.

6.1.2 Visual servoing

I test effettuati per l'applicazione di visual servoing hanno evidenziato l'efficacia del passaggio dal controllo di posizione al controllo di velocità, oltre alla capacità del controllore proporzionale-integrale di permettere al manipolatore di inseguire in maniera fluida il target in movimento sul piano orizzontale.

La Figura 6.2 mostra i risultati di tali test sul cubo rosso. La Figura 6.2a mostra la prima fase di posizionamento del robot nella configurazione di home e il passaggio allo stato di *SEARCH*. La Figura 6.2b mostra il passaggio dello stato a *VISUAL_SERVOING* e un primo inseguimento dell'oggetto, per cui il manipolatore si avvicina al target posizionato sul piano di lavoro. Subito dopo, nella Figura 6.2c si vede uno spostamento manuale del cubo e nella Figura 6.2d una successiva correzione della posizione del manipolatore.

Nel complesso, il sistema è mantenuto stabile grazie all'implementazione del meccanismo di *anti-windup*, che impedisce la saturazione dei registri della componente integrale durante i cambi di posizione del target.

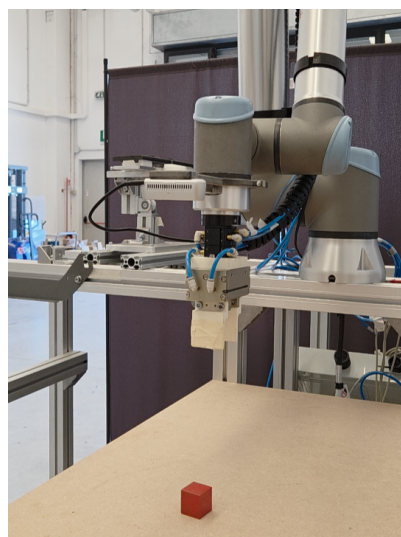
6.1.3 Conclusioni

L'obiettivo principale di questo lavoro di tesi era rappresentato dalla progettazione e dall'implementazione di un'architettura software capace di eseguire i compiti di *pick-and-place* e di *visual servoing* facendo fronte alle problematiche sia hardware sia software tipiche di un ambiente di lavoro reale.

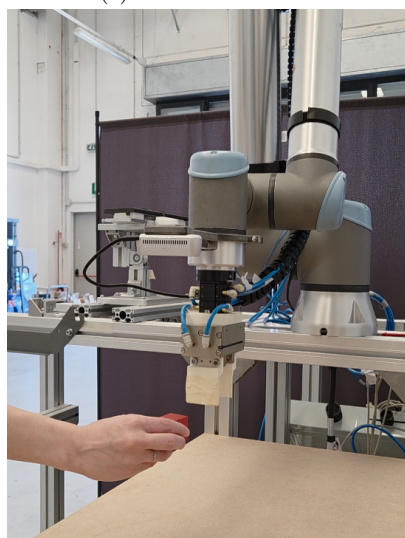
Per quanto riguarda l'applicazione di *pick-and-place* statico, il sistema realizzato ha dimostrato un'elevata efficacia complessiva, garantendo una presa precisa e ripetibile e una movimentazione fluida. Il principale limite riscontrato è rappresentato dalla latenza causata dalla pipeline di elaborazione delle immagini: il modello per l'object detection e per il tracking è stato vincolato all'esecuzione su CPU, limitando la velocità di inferenza della rete. Questo,



(a) Posizione di home



(b) Inizio dell'inseguimento



(c) Cambio di posizione del target



(d) Inseguimento

Figura 6.2: Sequenza di visual servoing per il cubo rosso

insieme ai tempi di acquisizione dei frame della camera e insieme al tempo necessario per l'esecuzione della pipeline di stima della posa dell'oggetto e della stabilizzazione dell'orientamento, ha introdotto un ritardo complessivo che ha causato una latenza nel movimento del manipolatore, rallentando l'esecuzione dell'operazione complessiva.

Analogamente, la stessa criticità è stata riscontrata nell'applicazione di *visual servoing*. Nonostante il controllore proporzionale-integrale si sia dimostrato accurato per l'inseguimento dei movimenti del target e per il mantenimento della camera alla distanza desiderata, il ritardo introdotto dal sistema di visione ha mostrato effetti ancora più evidenti rispetto al caso statico: durante il movimento del braccio, infatti, l'acquisizione dei frame ha introdotto un *lag* intrinseco e non eliminabile. Questo sfasamento tra lo stato reale dell'ambiente di lavoro e l'informazione elaborata dalla camera riduce la stabilità del sistema nel caso di movimenti rapidi dell'oggetto target. Per questo motivo, l'ottimizzazione di queste problematiche rappresenta il punto di partenza ideale per possibili sviluppi futuri del progetto presentato.

6.2 Sviluppi futuri

Il lavoro proposto in questa tesi apre la strada a molteplici possibilità di miglioramento e risoluzione di alcune delle problematiche affrontate, nonché a diversi sviluppi futuri. Tra i principali si segnalano:

- *esecuzione* del modello di rete neurale su una GPU dedicata, in maniera tale da diminuire la latenza complessiva sul sistema;
- *estensione* del sistema di pick-and-place alla presa di oggetti non noti;
- *integrazione* della camera nel file (*.xacro*) di configurazione del robot, in maniera tale da farla risultare parte della catena cinematica del braccio;
- *miglioramento* del sistema di visual servoing per includere movimenti di rotazione del polso del robot in base all'orientamento dell'oggetto e per includere movimenti non confinati al solo piano di lavoro orizzontale;
- *inclusione* di un *VLM* (*Vision Language Model*) per permettere all'operatore di interagire con il sistema e specificare la tipologia o le caratteristiche dell'oggetto che si vuole afferrare o inseguire.

Bibliografia

- [1] IDEA Soc. Coop. *IDEA Soc. Coop.* URL: <https://www.idea-online.it/>.
- [2] Jianwei Li et al. «High-quality indoor scene 3D reconstruction with RGB-D cameras: A brief review». In: *Computational Visual Media* 8.3 (2022), pp. 369–393. DOI: [10.1007/s41095-021-0250-8](https://doi.org/10.1007/s41095-021-0250-8).
- [3] Min Sung Ahn et al. «Analysis and Noise Modeling of the Intel RealSense D435 for Mobile Robots». In: *2019 16th International Conference on Ubiquitous Robots (UR)*. 2019, pp. 707–711. DOI: [10.1109/URAI.2019.8768489](https://doi.org/10.1109/URAI.2019.8768489).
- [4] Wikipedia. *Pinhole camera model*. URL: https://en.wikipedia.org/wiki/Pinhole_camera_model.
- [5] Z. Zhang. «A flexible new technique for camera calibration». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.11 (2000), pp. 1330–1334. DOI: [10.1109/34.888718](https://doi.org/10.1109/34.888718).
- [6] Jianchu Lin et al. «A Method for Depth Camera Calibration Based on Motion Capture System». In: *2023 International Conference on the Cognitive Computing and Complex Data (ICCD)*. 2023, pp. 242–246. DOI: [10.1109/ICCD59681.2023.10420758](https://doi.org/10.1109/ICCD59681.2023.10420758).
- [7] Ikenna Enebuse et al. «A Comparative Review of Hand-Eye Calibration Techniques for Vision Guided Robots». In: *IEEE Access* 9 (2021), pp. 113143–113155. DOI: [10.1109/ACCESS.2021.3104514](https://doi.org/10.1109/ACCESS.2021.3104514).
- [8] A. Muis e K. Ohnishi. «Eye-to-hand approach on eye-in-hand configuration within real-time visual servoing». In: *The 8th IEEE International Workshop on Advanced Motion Control, 2004. AMC '04*. 2004, pp. 647–652. DOI: [10.1109/AMC.2004.1297945](https://doi.org/10.1109/AMC.2004.1297945).
- [9] Daniele Di Muzio. «Controllo real-time di un sistema integrato robot collaborativo e sistema di visione basato su reti neurali per il riconoscimento e la presa di oggetti». Tesi di Laurea Magistrale. Università Politecnica delle Marche, 2022.
- [10] V. Lippiello, B. Siciliano e L. Villani. «Eye-in-Hand/Eye-to-Hand Multi-Camera Visual Servoing». In: *Proceedings of the 44th IEEE Conference on Decision and Control*. 2005, pp. 5354–5359. DOI: [10.1109/CDC.2005.1583013](https://doi.org/10.1109/CDC.2005.1583013).

- [11] Shunan Ren et al. «A Simultaneous Hand-Eye Calibration Method for Hybrid Eye-in-Hand/Eye-to-Hand System». In: *2017 IEEE 7th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*. 2017, pp. 568–573. DOI: [10.1109/CYBER.2017.8446185](https://doi.org/10.1109/CYBER.2017.8446185).
- [12] Wikipedia. *Rete neurale convoluzionale*. URL: https://it.wikipedia.org/wiki/Rete_neurale_convoluzionale.
- [13] Muhammad Arsalan et al. «Deep Learning Based Autonomous Robotic Grasping and Sorting System for Industry 4.0». In: *2024 International Conference on Robotics and Automation in Industry (ICRAI)*. 2024, pp. 1–6. DOI: [10.1109/ICRAI62391.2024.10894445](https://doi.org/10.1109/ICRAI62391.2024.10894445).
- [14] Ultralytics. *Ultralytics YOLO*. URL: <https://docs.ultralytics.com/>.
- [15] Mingming Meng et al. «Multi-object recognition and grasping detection based on the anchor-free network». In: *2023 16th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. 2023, pp. 1–6. DOI: [10.1109/CISP-BMEI60920.2023.10373320](https://doi.org/10.1109/CISP-BMEI60920.2023.10373320).
- [16] COCO. *COCO Dataset*. URL: <https://cocodataset.org/#home>.
- [17] Qiaokang Liang et al. «Robotic Grasping Detection Based on Continual Learning Using Perceptual Loss and Multibranch Deep Fusion». In: *IEEE Sensors Journal* 25.24 (2025), pp. 44962–44972. DOI: [10.1109/JSEN.2025.3628829](https://doi.org/10.1109/JSEN.2025.3628829).
- [18] Ultralytics. *blog/top-8-open-source-object-tracking-tools-and-algorithms*. URL: <https://www.ultralytics.com/blog/top-8-open-source-object-tracking-tools-and-algorithms#1-ultralytics-yolo-models-and-the-ultralytics-python-package>.
- [19] Hyeon-Seok Sim e Hyun-Chong Cho. «Enhanced DeepSORT and StrongSORT for Multicattle Tracking With Optimized Detection and Re-Identification». In: *IEEE Access* 13 (2025), pp. 19353–19364. DOI: [10.1109/ACCESS.2025.3535092](https://doi.org/10.1109/ACCESS.2025.3535092).
- [20] Yue Ma et al. «KCF based 3D object tracking via RGB-D camera of a quadrotor». In: *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2017, pp. 939–944. DOI: [10.1109/ICUAS.2017.7991411](https://doi.org/10.1109/ICUAS.2017.7991411).
- [21] Carl Yuheng Ren et al. «3D Tracking of Multiple Objects with Identical Appearance Using RGB-D Input». In: *2014 2nd International Conference on 3D Vision*. Vol. 1. 2014, pp. 47–54. DOI: [10.1109/3DV.2014.39](https://doi.org/10.1109/3DV.2014.39).
- [22] Xianke Zhang et al. «Fuzzy Adaptive Visual Servo Control for Dynamic Target Tracking of Robotic Manipulators». In: *2025 5th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*. 2025, pp. 403–409. DOI: [10.1109/ICAIRC68035.2025.11385041](https://doi.org/10.1109/ICAIRC68035.2025.11385041).

- [23] Rihem Farkh et al. «Intelligent Visual Tracking Robots Using Cascade PID Controllers». In: *2023 IEEE Third International Conference on Signal, Control and Communication (SCC)*. 2023, pp. 1–6. DOI: [10.1109/SCC59637.2023.10527483](https://doi.org/10.1109/SCC59637.2023.10527483).
- [24] Francois Chaumette e Seth Hutchinson. «Visual servo control. I. Basic approaches». In: *IEEE Robotics Automation Magazine* 13.4 (2006), pp. 82–90. DOI: [10.1109/MRA.2006.250573](https://doi.org/10.1109/MRA.2006.250573).
- [25] Zhiyuan Chen, Tiemin Li e Yao Jiang. «Image-Based Visual Servoing With Collision-Free Path Planning for Monocular Vision-Guided Assembly». In: *IEEE Transactions on Instrumentation and Measurement* 73 (2024), pp. 1–17. DOI: [10.1109/TIM.2024.3463014](https://doi.org/10.1109/TIM.2024.3463014).
- [26] Chao Qin et al. «Perception-Aware Image-Based Visual Servoing of Aggressive Quadrotor UAVs». In: *IEEE/ASME Transactions on Mechatronics* 28.4 (2023), pp. 2020–2028. DOI: [10.1109/TMECH.2023.3276211](https://doi.org/10.1109/TMECH.2023.3276211).
- [27] Yuan-Chih Peng et al. «Comparing Position- and Image-Based Visual Servoing for Robotic Assembly of Large Structures». In: *2020 IEEE 16th International Conference on Automation Science and Engineering (CASE)*. 2020, pp. 1608–1613. DOI: [10.1109/CASE48305.2020.9217028](https://doi.org/10.1109/CASE48305.2020.9217028).
- [28] Jianchuan Guo et al. «Position-Based Visual Servoing of a Dual-Arm System for Aerial Manipulation». In: *2024 8th International Symposium on Computer Science and Intelligent Control (ISCSIC)*. 2024, pp. 237–241. DOI: [10.1109/ISCSIC64297.2024.00056](https://doi.org/10.1109/ISCSIC64297.2024.00056).
- [29] Mostafa M. Ibrahim e Qiong Liu. «Optimized Color-guided Filter for Depth Image Denoising». In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 8568–8572. DOI: [10.1109/ICASSP.2019.8683416](https://doi.org/10.1109/ICASSP.2019.8683416).
- [30] Xin Zhang e Ruiyuan Wu. «Fast depth image denoising and enhancement using a deep convolutional network». In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2016, pp. 2499–2503. DOI: [10.1109/ICASSP.2016.7472127](https://doi.org/10.1109/ICASSP.2016.7472127).
- [31] Xuyang Duan, Ruiqi Xie e Jun Han. «An Energy-Efficient Image Denoising Accelerator with Depth-wise Separable Convolution and Fused-Layer Architecture». In: *2021 IEEE 14th International Conference on ASIC (ASICON)*. 2021, pp. 1–4. DOI: [10.1109/ASICON52560.2021.9620485](https://doi.org/10.1109/ASICON52560.2021.9620485).
- [32] UR. *ur10e e-series datasheets*. URL: https://www.universal-robots.com/media/1807466/ur10e_e-series_datasheets_web.pdf.
- [33] UR. *User manual UR10e*. URL: https://www.universal-robots.com/manuals/it/PDF/SW5_24/user-manual-UR10e-PDF-online/711-055-00_UR10e_User_Manual_it_Global.pdf.
- [34] UR. *Universal Robots*. URL: <https://www.universal-robots.com/it/prodotti/serie-ur/>.

- [35] Schunk. *Schunk MPC 150*. URL: <https://schunk.com/it/it/sistemi-di-presa/pinza-parallela/mpc/mpc-150/p/000000000001343470>.
- [36] Schunk. *Cambio tool*. URL: <https://schunk.com/it/it/tecnologia-di-automazione/cambio-utensile/sws/swk-007-000-000-in03/p/000000000001405836?bmeocat=retailer>.
- [37] RS. *SMC VP342-5Y01-02FA*. URL: <https://mx-en.rs-online.com/product/smc-valves/vp342-5yo1-02fa/71266885/>.
- [38] RS. *SMC SY5220-5Y0-01F-0*. URL: <https://it.rs-online.com/web/p/elettrovalvole-per-pneumatica/7013072>.
- [39] Intel. *RealSense D435*. URL: <https://www.realsenseai.com/products/stereo-depth-camera-d435/>.
- [40] Intel. *RealSense D400 series Datasheet*. 2026. URL: <https://www.realsenseai.com/wp-content/uploads/2026/03/RealSense-D400-Series-Datasheet-Mar-2026.pdf>.
- [41] UniversalRobots. *UniversalRobots - Setup for real-time scheduling*. URL: https://github.com/UniversalRobots/Universal_Robots_Client_Library/blob/master/doc/real_time.rst.
- [42] open robotics. *ROS 2 Humble Hawksbill Release*. URL: <https://www.openrobotics.org/blog/2022/5/24/ros-2-humble-hawksbill-release>.
- [43] ROS. *ROS2 Humble installation*. URL: <https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html>.
- [44] ROS. *ROS2 Humble tutorials - workspace creation*. URL: <https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html>.
- [45] UniversalRobots. *Universal Robots ROS2 Driver*. URL: https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver/tree/humble.
- [46] UniversalRobots. *Installing a URCap on a e-Series robot*. URL: https://docs.ros.org/en/ros2_packages/humble/api/ur_robot_driver/doc/installation/install_urcap_e_series.html.
- [47] UniversalRobots. *UniversalRobotsExternalControlURCap*. URL: https://github.com/UniversalRobots/Universal_Robots_ExternalControl_URCap/releases.
- [48] UniversalRobots. *Universal Robots Robot Setup*. URL: https://docs.ros.org/en/ros2_packages/humble/api/ur_robot_driver/doc/installation/robot_setup.html.
- [49] MoveIt. *MoveIt - Getting Started*. URL: https://moveit.picknik.ai/humble/doc/tutorials/getting_started/getting_started.html.

-
- [50] AndrejOrsula. *MoveIt 2 - Calibration*. URL: https://github.com/AndrejOrsula/moveit2_calibration/tree/ros2_port.
 - [51] realsenseai. *realsense - ros*. URL: <https://github.com/realsenseai/realsense-ros?tab=readme-ov-file>.
 - [52] realsenseai. *librealsense*. URL: https://github.com/realsenseai/librealsense/blob/master/doc/distribution_linux.md.
 - [53] pytorch. *pytorch - Get Started*. URL: <https://pytorch.org/get-started/locally/>.
 - [54] Ultralytics. *YOLO12*. URL: <https://docs.ultralytics.com/models/yolo12#performance-metrics>.
 - [55] Roboflow. *Roboflow*. URL: <https://roboflow.com/>.
 - [56] Google. *Gemini*. URL: <https://gemini.google.com>.

Elenco delle figure

1.1	Logo di IDEA Soc. Coop. (immagine tratta da [1])	7
1.2	Settori in cui opera IDEA Soc. Coop. (immagine tratta da [1])	7
2.1	Pipeline di ricostruzione di una scena 3D a partire da un sensore RGB-D (immagine tratta da [2])	10
2.2	Modello Pin-Hole (immagine tratta da [4])	12
2.3	Tipologie di configurazione del sensore di visione (immagine tratta da [9])	14
2.4	Esempio di CNN (immagine tratta da [12])	16
2.5	Confronto tra le prestazioni dei modelli YOLO (immagine tratta da [14])	17
2.6	Esempio di object tracking (immagine tratta da [18])	19
2.7	Framework proposto nell'articolo [29]	25
2.8	Light CNN proposta nell'articolo [30]	26
3.1	Robot collaborativo UR10e	30
3.2	Unità di controllo del robot UR10e (immagine tratta da [9])	31
3.3	Teach pendant del robot UR10e (immagine tratta da [9])	31
3.4	Schunk MPC 150 (immagine tratta da [35])	32
3.5	Pinza Schunk con griffe personalizzate	33
3.6	Cambio tool Schunk (immagine tratta da [36])	33
3.7	Sistema di regolazione dell'aria (immagine tratta da [9])	34
3.8	Elettrovalvola monostabile (immagine tratta da [37])	34
3.9	Elettrovalvola bistabile (immagine tratta da [38])	35
3.10	Circuito ad aria compressa	35
3.11	Intel RealSense D435 (immagine tratta da [39])	36
3.12	Moduli della Intel Realsense D435 (immagine tratta da [39])	36
3.13	Configurazione della RealSense D435 sul polso del robot	37
3.14	RealSense Viewer	38
4.1	Architettura del sistema	39
4.2	ROS 2 Humble Hawksbill (immagine tratta da [42])	42
4.3	UR Cap	43
4.4	Impostazioni UR Cap	44
4.5	Programma con external control	44
4.6	RViz	45
4.7	Calcolo del payload - Home	49
4.8	Calcolo del payload - Overview	50

4.9	Calcolo del payload - Posizionamento	50
4.10	Calcolo del payload - Final	51
4.11	MoveIt Setup Assistant	52
4.12	Ottimizzazione del controllo di auto collisione	52
4.13	Definizione dei giunti virtuali	53
4.14	Definizione dei planning groups	53
4.15	Definizione del gruppo <i>ur_manipulator</i>	54
4.16	Definizione della catena cinematica del gruppo <i>ur_manipulator</i>	54
4.17	Definizione del gruppo <i>gripper</i>	55
4.18	Definizione dei link appartenenti al gruppo <i>gripper</i>	55
4.19	Planning groups definiti	56
4.20	Definizione dell'end-effector	56
4.21	Setup ROS 2 Controllers	57
4.22	Setup MoveIt Controllers	57
4.23	Definizione delle informazioni dell'autore	58
4.24	Generazione dei file finali	58
4.25	Aggiunta di un pannello su RViz	62
4.26	Scelta del pannello " <i>HandEyeCalibration</i> "	62
4.27	Pannello di calibrazione - Target	63
4.28	Target per la calibrazione	63
4.29	Pannello di calibrazione - Context	64
4.30	Pannello di calibrazione - Calibrate	64
4.31	Nodo del robot	65
4.32	Nodo di move_group e rviz	66
4.33	Nodo della camera	66
4.34	Nodo di calibrazione della camera	66
4.35	Nodo di MoveIt Servo	67
4.36	Avvio finale dei nodi	67
5.1	Oggetti realizzati per l'applicazione	71
5.2	Misura della chiusura della pinza	71
5.3	Dimensioni dei sei oggetti realizzati	72
5.4	Esempio di immagini contenute nel dataset	73
5.5	Procedura di labeling delle immagini	74
5.6	Metriche di training dei modelli a confronto	76
5.7	Matrici di confusione normalizzate dei modelli a confronto	77
5.8	Confronto tra modelli - FPS di inferenza	78
5.9	Conversione delle coordinate del centro geometrico dell'oggetto	80
5.10	Codice per l'estrazione dell'orientamento per il parallelepipedo	82
5.11	Stima della posa del cubo grigio	83
5.12	Stima della posa del parallelepipedo rosso	83
5.13	Pin di output digitale	84
5.14	Pick-and-place - Diagramma di sequenza del posizionamento in home	86
5.15	Pick-and-place - Diagramma di sequenza della pipeline di visione	86
5.16	Diagramma di sequenza del pick-and-place	87
5.17	Stima delle coordinate per il visual servoing	89

5.18	Visual servoing - Diagramma di sequenza del posizionamento in home	91
5.19	Visual servoing - Diagramma di sequenza dello stato di “VISUAL_SERVOING”	91
6.1	Sequenza di pick-and-place per il cubo grigio	94
6.2	Sequenza di visual servoing per il cubo rosso	96

Elenco delle tabelle

5.1	Specifiche dei componenti del PC dedicato all'addestramento dei modelli YOLO	75
5.2	Riassunto delle matrici di confusione normalizzate dei modelli . .	78