UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Meccanica

Studio di fattibilità e progettazione preliminare di una cella robotica collaborativa per la decorazione alimentare basata su sistema di visione Eye-in-Hand

Feasibility Study and Preliminary Design of a Collaborative Robotic Cell for Food Decoration Based on an Eye-in-Hand Vision System

Relatore:

Prof. **Palmieri Giacomo**

Tesi di Laurea di:

Ceresoli Beatrice

A.A. 2025/2026

INDICE

INTRODUZIONE

Il presente lavoro di tesi nasce nell'ambito di un'esperienza di tirocinio svolta presso i-Labs Industry e si inserisce in un più ampio progetto di automazione dei processi di lavorazione alimentare. L'obiettivo generale del progetto è lo sviluppo di una cella robotizzata collaborativa per la decorazione automatica di vaschette di insalata russa, mediante la deposizione controllata di ciuffi di maionese lungo il bordo del prodotto, in sostituzione dell'operazione tradizionalmente eseguita a mano dall'operatore con la sac à poche.

Il lavoro si configura come uno studio di fattibilità, il cui scopo non è la realizzazione di un sistema industriale ottimizzato e pronto alla produzione, bensì la verifica della possibilità concreta di automatizzare il processo decorativo, individuandone le criticità tecniche e dimostrando, mediante un prototipo di laboratorio, che il gesto manuale dell'operatore può essere scomposto in funzioni controllabili e affidate a un sistema meccatronico integrato. In quest'ottica, il prototipo sviluppato costituisce una prima implementazione sperimentale, sulla base della quale valutare le prestazioni del sistema e le direzioni di miglioramento per una successiva industrializzazione.

A partire da queste premesse, il progetto è stato articolato attorno a tre obiettivi principali. Il primo è la replica fedele della forma dei ciuffi realizzati manualmente, ossia depositi di maionese con base regolare e punta ben definita rivolta verso l'alto, comparabili dal punto di vista qualitativo con quelli ottenuti dall'operatore. Il secondo obiettivo è la corretta esecuzione della traiettoria di decorazione da parte del robot collaborativo, guidata dalle informazioni acquisite dal sistema di visione, in modo che la deposizione avvenga lungo il bordo della vaschetta indipendentemente dal suo orientamento e dalle sue dimensioni. Il terzo obiettivo riguarda il rispetto dei tempi di produzione: pur trattandosi di uno studio di fattibilità, è stato assunto come riferimento il tempo ciclo dell'operazione manuale, pari a circa 11 s per vaschetta, con l'intento di verificare se la soluzione automatizzata potesse, in prospettiva, avvicinarsi e tendere a tale valore, individuando i fattori che ne determinano il limite inferiore.

Per conseguire tali obiettivi, la cella è stata progettata integrando un manipolatore collaborativo Universal Robots UR5e, una telecamera RGB-D Intel RealSense D455 in configurazione eye-in-hand e un end-effector meccatronico custom, costituito da una siringa di grande capacità azionata da un asse lineare motorizzato e da un sistema elettronico di comando basato su scheda Arduino. Il coordinamento dei sottosistemi è affidato a un programma supervisore eseguito su PC, che gestisce l'acquisizione delle immagini, la localizzazione della vaschetta, la generazione della traiettoria e la sincronizzazione tra il movimento del robot e l'erogazione del materiale.

La trattazione è organizzata in sette capitoli. Il **Capitolo 1** presenta lo stato dell'arte, analizzando il ruolo della robotica collaborativa nel settore alimentare, i sistemi di visione artificiale impiegati in ambito robotico e le tecnologie di dispensazione per prodotti food, con particolare attenzione alle proprietà reologiche della maionese che ne condizionano l'erogazione. Il **Capitolo 2** descrive il processo manuale, ne ricava i requisiti funzionali e introduce l'architettura generale della cella robotizzata, articolata nei sottosistemi di visione, robotico e di erogazione. Il **Capitolo 3** è dedicato ai componenti hardware: il robot UR5e, il sistema di visione e l'end-effector custom, comprensivo della struttura portante, del sistema di dispensazione e del sistema di attuazione e comando dell'asse lineare. Il **Capitolo 4** illustra il setup sperimentale e la procedura di calibrazione hand-eye del sistema robot-camera, necessaria a trasformare i punti osservati dalla telecamera in coordinate raggiungibili dal robot. Il **Capitolo 5** descrive il sistema di visione e la generazione automatica della traiettoria, dalla pipeline di elaborazione dell'immagine alla costruzione dei waypoint robotici. Il **Capitolo 6** approfondisce l'architettura di controllo integrato, presentando il firmware di controllo dell'asse lineare e i protocolli di comunicazione che collegano il PC, il robot e l'asse. Il **Capitolo 7**, infine, riporta la campagna sperimentale condotta sulla cella completa, le criticità riscontrate, le soluzioni adottate, la taratura dei parametri di processo e i risultati conseguiti.

1. STATO DELL'ARTE

1.1. ROBOTICA E ROBOT COLLABORATIVI NEL FOOD

Il termine "robot" deriva dalla parola ceca "robota", traducibile come "lavoro forzato". Il primo a introdurlo nel lessico moderno fu lo scrittore ceco Karel Čapek, che lo utilizzò per definire gli automi che lavoravano al posto degli operai nel dramma fantascientifico *R.U.R.* del 1920. Tra la fine degli anni '50 e gli inizi degli anni '60 il termine perse l'accezione letteraria per assumere un significato tecnico-industriale, infatti nel 1961 due ingegneri statunitensi, Joe Engelberger e George Devol, realizzarono il primo vero braccio robotico industriale, l'Unimate (*Figura 1.1*). Il progetto affondava le radici nel brevetto "Programmed Article Transfer", depositato da Devol nel 1954 e concesso nel 1961, tuttora considerato il documento fondante della robotica industriale moderna [1]. In generale, un robot può essere definito come un sistema automatizzato, programmabile e dotato di controllo autonomo, in grado di eseguire operazioni fisiche o logiche in sostituzione, totale o parziale, dell'intervento umano. Secondo la definizione formale dell'Organizzazione Internazionale per la Standardizzazione (ISO), si tratta di "una macchina manipolatrice automaticamente controllata, riprogrammabile, polivalente, con numerosi gradi di libertà, che può essere fissa o mobile per l'utilizzo in applicazioni di automazione industriale" [2].

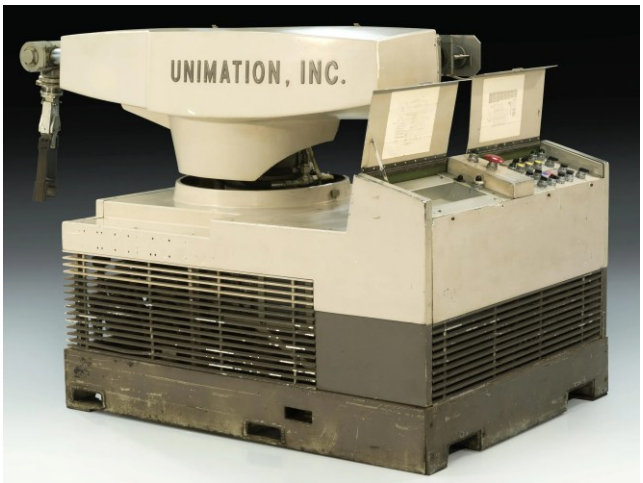


Figura 1.1: Unimate robot

Nel corso degli ultimi decenni l'adozione di sistemi robotici nei processi produttivi industriali ha subìto un'accelerazione significativa, guidata dalla crescente pressione competitiva sui mercati globali e dalla necessità di garantire standard elevati di qualità, ripetibilità e sicurezza. Settori come quello automobilistico ed elettronico, caratterizzati da elevati volumi produttivi e componenti standardizzati, sono stati i principali beneficiari di questa tecnologia, raggiungendo livelli di automazione difficilmente comparabili con altri comparti [2]. L'automazione e la robotica si sono affermate in questi contesti grazie alla loro capacità di eseguire operazioni di assemblaggio, saldatura, movimentazione e ispezione con precisione, continuità e velocità superiori rispetto all'operatore umano, abbattendo i costi operativi e migliorando la qualità del prodotto finale [4]. Nonostante l'industria alimentare sia un settore trainante dell'economia europea, il livello di penetrazione dell'automazione è ancora basso. Le ragioni sono molteplici e connesse tra loro. Sul piano economico, il basso margine di guadagno dei prodotti alimentari non incoraggia gli investimenti tecnologici a cui si aggiunge anche una struttura del mercato composta principalmente da piccole-medie imprese che limita ulteriormente la disponibilità di risorse da destinare all'innovazione [3]. Sul piano tecnico, la variabilità intrinseca delle materie prime rappresenta la sfida più significativa, in quanto queste si differenziano per forma, dimensioni, peso, consistenza e proprietà reologiche che rendono difficoltoso l'uso di robot pensati per applicazioni standardizzate [2]. A questo si aggiungono anche i requisiti di igiene imposti dalla normativa sulla sicurezza alimentare secondo i quali i componenti del robot a contatto con gli alimenti devono essere resistenti alla corrosione, lavabili e compatibili con i comuni detergenti. [2].

Nonostante queste criticità, l'avvento della robotica nel settore alimentare si è verificato a partire dalle applicazioni più generiche e comuni come la pallettizzazione e il confezionamento di prodotti a geometria finita, ovvero tutto quello che era più compatibile con le funzioni già consolidate dei robot. Questo ha prodotto un incremento di produttività rispetto alle linee manuali del 25% che ha incentivato sempre più l'uso dei robot anche per lavorazioni primarie come classificazione ottica di frutta e verdura e disossatura di carcasse animali, dove la variabilità del prodotto è massima e il carico sull'operatore è elevato [2]. In questo modo si è verificato il passaggio da una robotica confinata al fine linea a una robotica integrata nelle fasi produttive.

Tuttavia, il cambiamento più significativo si è verificato con l'avvento di robot collaborativi. A differenza dei robot industriali convenzionali, che operano in celle segregate e richiedono barriere fisiche di sicurezza, i cobot sono progettati per condividere lo spazio di lavoro con gli

operatori umani (*Figura 1.2*). Questo è reso possibile da sistemi integrati di limitazione di coppia e forza, rilevamento delle collisioni e monitoraggio continuo della velocità, che consentono l'arresto immediato del sistema in caso di contatto non previsto [4]. Le differenze rispetto ai robot tradizionali non sono solo tecniche: i cobot si caratterizzano per dimensioni compatte, payload generalmente inferiori a 10 kg, e interfacce di programmazione intuitive, incluso lo spostamento manuale del braccio, che rendono possibile il loro dispiegamento anche in contesti privi di figure specializzate in robotica [4].

a.  ***b.*** 

Figura 1.2: ***a.*** *Robot industriale;* ***b.*** *Robot collaborativo*

Questa accessibilità si inserisce in un contesto di cambiamento più ampio, definito dalla Commissione Europea nel gennaio 2021 nel documento "Industry 5.0: Towards a Sustainable, Human-Centered and Resilient European Industry" [5]. Mentre l'Industria 4.0 si basava sull'integrazione di tecnologie avanzate nei processi produttivi per aumentare al massimo l'efficienza, la produttività e l'automazione, l'Industria 5.0 sposta la domanda da "come è possibile produrre meglio?" a "come è possibile produrre meglio per l'uomo e il pianeta?", fondandosi su tre pilastri fondamentali che sono il benessere umano, la sostenibilità e la resilienza [5]. Secondo questa ottica quindi i robot non sono più visti come semplici strumenti di automazione ma parti integranti di una collaborazione uomo-macchina in cui ciascuna componente contribuisce con i propri punti di forza: la flessibilità cognitiva dell'operatore da un lato, e la precisione, la ripetibilità e la resistenza all'affaticamento del sistema robotico dall'altro. È proprio in questo quadro che i cobot trovano la loro collocazione più naturale, in particolare nelle PMI del settore alimentare, che rappresentano la quota maggioritaria del comparto e per le quali il costo e la complessità dei sistemi tradizionali hanno costituito a lungo una barriera insormontabile [3].

Dal punto di vista delle applicazioni concrete nell'industria alimentare, la letteratura documenta esperienze in diversi contesti produttivi. Accorsi et al. [3] hanno avanzato uno studio di fattibilità per l'utilizzo di cobot per confezionare pasti finiti in un impianto di produzione per la ristorazione, in particolare il focus è su una metodologia utilizzata per valutare la fattibilità tecnica ed economica della sostituzione delle attività manuali con soluzioni automatizzate e collaborative, valutando la cosiddetta idoneità all'automazione. I risultati indicano un recupero di circa il 70% dell'investimento iniziale entro due anni e di circa l'85% entro i due anni e mezzo, con una riduzione del carico ergonomico in tutti gli scenari simulati [3].

Nikolova-Alexieva et al. [5] esaminano l'automazione di una linea di confezionamento dello yogurt, dove il robot collaborativo esegue compiti quali l'etichettatura, l'ispezione visiva e lo smistamento in confezioni. La soluzione è supportata da una connessione IoT e da un gemello digitale. I risultati della simulazione indicano una riduzione del tempo di ciclo per confezione da circa 8 a circa 5 secondi, un tasso di errori di etichettatura inferiore allo 0,5% e un miglioramento dell'efficienza energetica di circa il 15%, con un ritorno sull'investimento stimato in circa diciotto mesi [4]. Gli autori presentano anche un secondo scenario applicativo: lo smistamento delle mele basato sull'ispezione visiva mediante intelligenza artificiale. In questo caso, il cobot lavora con una piattaforma AI edge e una telecamera di visione artificiale per il rilevamento dei difetti in tempo reale, dimostrando come il limite dei programmi standard preimpostati viene superato con sistemi di visione artificiale che affermano con chiarezza l'importanza della percezione visiva come la componente che consente al robot di adattare la propria azione alla variabilità del prodotto. I due casi studio confermano la flessibilità, l'adattabilità e l'applicabilità ingegneristica del modello in diversi ambienti produttivi con diversi gradi di complessità e sensibilità del prodotto. Allo stesso modo anche Bharathiraja e Keerthana [4] sottolineano come i sistemi di visione artificiale integrati con la cinematica robotica consentano l'ispezione al 100% dei prodotti in linea, con risultati oggettivi e ripetibili superiori per affidabilità al controllo manuale.

Questi esempi anticipano la centralità dei sistemi di visione nella progettazione di celle robotiche per l'industria alimentare, tema che sarà approfondito nel capitolo successivo.

1.2. SISTEMI DI VISIONE ARTIFICIALE IN ROBOTICA

I robot industriali di prima generazione operavano esclusivamente in ambienti strutturati, all'interno dei quali la posizione degli oggetti era nota a priori e il campo d'azione del robot era definito in anticipo. In tali contesti, il robot eseguiva traiettorie prestabilite in modo ripetitivo e deterministico, senza alcuna capacità di verificare in tempo reale se la traiettoria seguita fosse effettivamente appropriata rispetto all'oggetto presente nella scena. Questo tipo di approccio può essere efficace per la produzione in massa di componenti omogeni, ma d'altra parte è inadeguato quando si parla di oggetti la cui variabilità, in termini di forma, dimensione, posizione o orientamento, non è trascurabile. La visione artificiale supera questo limite fondamentale, consentendo al robot di adattarsi alle condizioni reali della scena. In termini generali, un sistema di visione artificiale include due parti: l'acquisizione delle informazioni visive, realizzata attraverso una varietà di dispositivi hardware quali telecamere, sensori di profondità e sistemi di illuminazione, e l'elaborazione delle informazioni acquisite, che avviene mediante algoritmi e procedure di analisi dell'immagine volti a estrarre le caratteristiche rilevanti della scena. Quest'ultima componente consente di ricavare informazioni a partire dai dati grezzi, rendendole disponibili al personale tecnico per le fasi di valutazione successive [6].

L'integrazione della visione nei sistemi robotici ha subito un'accelerazione significativa negli ultimi decenni, trainata dal miglioramento dell'hardware di acquisizione, dallo sviluppo di algoritmi di elaborazione delle immagini e dalla diffusione del deep learning come paradigma dominante per il riconoscimento visivo.

Il risultato è che la visione artificiale è oggi uno strumento consolidato nella robotica industriale e collaborativa, impiegata in applicazioni chiave, tra cui il rilevamento dei difetti, il controllo robotico o lo smistamento industriale, contribuendo a guidare lo sviluppo tecnologico e la crescita del mercato [7].

Un aspetto importante, quando si considera un sistema di visione, è come vengono elaborate le immagini acquisite. Come spiega Xiao et al. [6] le informazioni estratte dall'immagine grezza si articolano su tre livelli gerarchici.

Il livello basso riguarda l'acquisizione e il pre-processing ovvero vengono corretti i difetti introdotti dall'acquisizione come, ad esempio, riduzione del rumore mediante filtri, correzione delle non-uniformità dell'illuminazione, calibrazione del colore e compensazione della distorsione geometrica dell'ottica. Questi passaggi hanno un impatto determinante sulla

qualità dei risultati ottenuti nelle fasi successive, poiché molti algoritmi di analisi sono sensibili alle perturbazioni introdotte da condizioni di acquisizione non ideali [6].
A livello intermedio, l'elaborazione si basa sulla segmentazione dell'immagine, ovvero quel processo che va a dividerla in più parti o regioni che condividono caratteristiche comuni. Tecniche classiche come la sogliatura (thresholding), la segmentazione watershed e i filtri morfologici sono state ampiamente utilizzate in applicazioni di ispezione industriale per la loro semplicità computazionale e la facilità di implementazione. Tuttavia, questi metodi si basano su parametri sintonizzati manualmente che tendono a perdere efficacia quando le condizioni dell'immagine variano, ad esempio per cambiamenti di illuminazione o per la presenza di oggetti con caratteristiche visive simili allo sfondo [6].

Infine, a livello alto si trovano gli algoritmi di riconoscimento e classificazione delle informazioni estratte dalla segmentazione. Tra questi negli ultimi anni, il deep learning ha dimostrato eccellenti prestazioni computazionali in diversi campi, tra cui la visione artificiale, che rappresenta una delle aree di applicazione più importanti. Una limitazione importante del deep learning è la sua dipendenza da grandi quantità di dati etichettati per l'addestramento. In contesti industriali, raccogliere e annotare migliaia o decine di migliaia di immagini in condizioni reali è spesso costoso e logisticamente impegnativo, per questo spesso si ricorre al transfer learning, una tecnica dell'intelligenza artificiale e del deep learning che consiste nel prendere un modello già addestrato su un determinato compito e riutilizzarlo come punto di partenza per un compito diverso, ma correlato [6].
Un secondo aspetto rilevante riguarda il posizionamento della camera rispetto al robot, un fattore molto importante da considerare perché determina le caratteristiche operative del sistema di visione e il tipo di relazione che deve essere stabilita tra il sistema di riferimento della camera ("eye") e quello del robot ("hand"). Le configurazioni possibili, come è schematizzato in *Figura 1.3*, sono due: eye-to-hand e eye-in-hand [8].

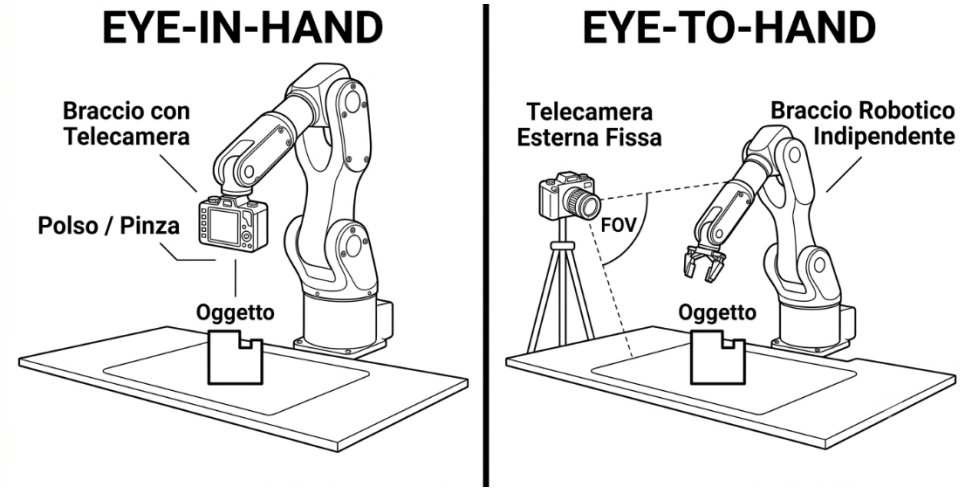


Figura 1.3: Configurazione eye-in-hand e eye-to-hand

Nella configurazione eye-to hand la telecamera è fissa nell'ambiente e osserva la scena dall'esterno, indipendentemente dal movimento del robot. Il vantaggio principale di questa soluzione è la stabilità del punto di vista: poiché la telecamera non si muove, la relazione geometrica tra il riferimento della telecamera e il riferimento del mondo rimane costante, semplificando il calcolo delle trasformazioni necessarie per tradurre le coordinate dell'immagine in coordinate del robot. Questa configurazione è particolarmente adatta per sistemi di ispezione su nastro trasportatore, dove gli oggetti transitano in posizione prevedibile davanti alla telecamera fissa. Il limite principale è che il robot può occludere parzialmente o totalmente il campo visivo della telecamera durante l'esecuzione del compito, rendendo impossibile l'acquisizione di informazioni visive aggiornate in quella fase [8].

Nella configurazione eye-in-hand (o camera-in-hand) la telecamera è montata direttamente sull'end-effector del robot e si muove solidalmente con esso. Questo approccio garantisce che la telecamera si trovi sempre nelle immediate vicinanze dell'oggetto di interesse, eliminando il problema dell'occlusione da parte del corpo del robot e fornendo immagini sempre aggiornate della zona di lavoro. Il costo di questa flessibilità è una maggiore complessità nel calcolo delle trasformazioni spaziali: poiché il punto di vista cambia ad ogni movimento del robot, è necessario disporre di una calibrazione precisa, ovvero una matrice di trasformazione che permetta il passaggio da coordinate espresse nel sistema di riferimento della camera a quelle espresse nel sistema di riferimento del robot [9]. Questa calibrazione è un prerequisito indispensabile per qualsiasi sistema di visione montato sul robot e il suo processo è trattato in dettaglio nel Capitolo…della presente tesi. La configurazione eye-in-hand è la scelta naturale

per le applicazioni collaborative in cui il robot deve interagire con oggetti in posizioni variabili, come nel caso della cella robotica oggetto di questo lavoro.

1.2.1. APPLICAZIONI NEL SETTORE ALIMENTARE

La maggior parte dei prodotti alimentari nelle varie fasi di trasformazione presenta una moltitudine di proprietà fisiche e condizioni di manipolazione complesse, il che rende difficile automatizzare in modo affidabile i vari processi, per questo è richiesta una robustezza e affidabilità che superano quelle dei tipici contesti industriali manufatturieri [7].

Tra le applicazioni più consolidate c'è la classificazione e l'ispezione qualitativa di prodotti freschi. Xiao et al. [6] passano in rassegna un ampio spettro di sistemi sviluppati per frutta e verdura: dal rilevamento dei difetti superficiali su mele, carote e agrumi, alla stima del grado di maturità tramite analisi del colore, fino alla misurazione volumetrica tramite telecamere a profondità. Per i prodotti di origine animale la visione artificiale consente di automatizzare compiti che tradizionalmente richiedevano operatori specializzati. Nel settore ittico, sistemi di visione 3D sono stati sviluppati per determinare automaticamente l'orientamento e i punti di taglio delle trote, stimare le dimensioni dei pesci e rilevare difetti come le macchie di gaping nei filetti di salmone [7]. Per la lavorazione della carne bovina, il progetto RoBUTCHER [8] ha sviluppato un sistema che utilizza dati RGB-D e sensori di forza, combinati con algoritmi di intelligenza artificiale, per guidare il coltello lungo la traiettoria di disossatura ottimale, adattandosi in tempo reale alla variabilità morfologica delle carcasse.

Un caso di particolare rilevanza per il presente lavoro è quello della deposizione controllata di ingredienti su superfici alimentari, come creme, salse e condimenti, dove la visione artificiale partecipa alla localizzazione e caratterizzazione geometrica della superficie su cui il materiale deve essere depositato. Questo scenario, che include anche la guarnizione automatizzata di vassoi di insalata russa con maionese oggetto della presente tesi, si colloca in un contesto emergente dell'automazione visiva in ambito alimentare, in cui la consistenza del prodotto introduce sfide aggiuntive rispetto alle applicazioni di ispezione e classificazione [8].

Nonostante i progressi documentati, l'adozione su scala industriale dei sistemi di visione artificiale nel settore alimentare è ancora ostacolata da alcune criticità strutturali. Le

condizioni ambientali tipiche degli impianti di lavorazione: umidità elevata, variazioni di illuminazione, superfici riflettenti e particelle in sospensione, possono degradare la qualità delle immagini acquisite e ridurre l'affidabilità dei sensori nel tempo. A queste si aggiunge la difficoltà di scalare i sistemi a nuovi prodotti o processi senza dover raccogliere nuovi dataset e ri-addestrare i modelli, un ostacolo particolarmente rilevante per le piccole e medie imprese del settore alimentare che gestiscono un'ampia gamma di prodotti con volumi di produzione relativamente ridotti [7].

Queste sfide confermano che, nonostante la visione artificiale rappresenti uno strumento consolidato per l'ispezione e la classificazione di prodotti alimentari, la sua applicazione a scenari che coinvolgono ingredienti semi-fluidi e superfici geometricamente variabili, come la guarnizione automatizzata oggetto della presente tesi, rimane un campo aperto. È in questo spazio che si inserisce il contributo di questo lavoro.

1.3. SISTEMI DI DISPENSAZIONE PER APPLICAZIONI FOOD E PROPRIETÀ REOLOGICHE DELLA MAIONESE

I sistemi di dispensazione per applicazioni alimentari rappresentano una classe di dispositivi progettati per trasferire, dosare ed eventualmente depositare un prodotto alimentare in modo controllato. Tali sistemi trovano impiego in numerose operazioni industriali, dal riempimento di contenitori al dosaggio di ingredienti, fino alla deposizione localizzata di salse, creme, emulsioni, puree, impasti e materiali semisolidi. Nel settore food, la dispensazione non riguarda quindi soltanto il controllo della quantità erogata, ma anche la qualità del deposito, la stabilità geometrica del materiale, la ripetibilità del processo e la compatibilità igienico-sanitaria dei componenti a contatto con l'alimento.

Nel caso di prodotti liquidi a bassa viscosità, il problema di dosaggio può essere affrontato mediante tecnologie relativamente semplici, come sistemi gravitazionali, valvole temporizzate o pompe centrifughe. Tuttavia, quando il prodotto presenta comportamento viscoso, semisolido o non newtoniano, come nel caso di salse e creme alimentari, la scelta del sistema di dispensazione diventa più complessa. Fellows evidenzia infatti che le proprietà fisiche degli alimenti, e in particolare viscosità, reologia e comportamento al flusso, influenzano direttamente la scelta delle apparecchiature di processo, delle pompe, delle valvole e dei sistemi di formatura o estrusione [10].

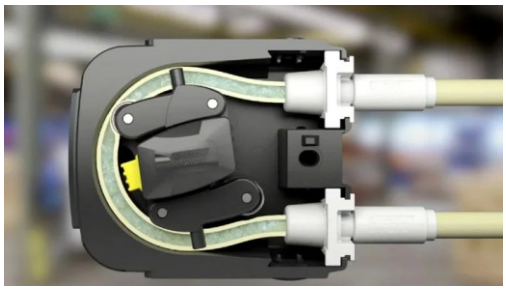
Tra le soluzioni maggiormente utilizzate nell'industria alimentare si possono individuare sistemi pneumatici, pompe peristaltiche e sistemi volumetrici basati su siringa, illustrate in *Figura 1.4*.

I sistemi pneumatici sfruttano l'azione di aria compressa per spingere il materiale attraverso un ugello. Essi consentono elevate velocità operative e una struttura relativamente semplice, ma il controllo accurato del volume erogato può risultare complesso a causa della comprimibilità del fluido di lavoro e delle variazioni di pressione durante il processo.

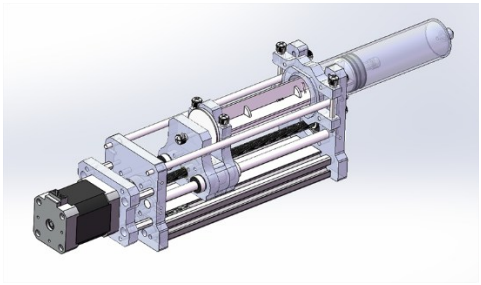
Le pompe peristaltiche costituiscono una valida alternativa per il trasferimento di liquidi e semiliquidi. Il principio di funzionamento si basa sulla compressione ciclica di un tubo flessibile mediante rulli rotanti, generando un flusso controllato senza che il prodotto entri in contatto con i componenti meccanici della pompa. Questa caratteristica rende tali sistemi particolarmente adatti ad applicazioni che richiedono elevati standard igienici. Tuttavia, la natura pulsante del flusso e la necessità di frequenti procedure di calibrazione possono limitarne l'impiego nei processi che richiedono elevata precisione di deposizione.

Una delle tecnologie più diffuse per il dosaggio preciso di piccoli volumi è rappresentata dalle syringe pump, costituite da una siringa azionata da un attuatore lineare che controlla il movimento del pistone. In tali sistemi la rotazione di un motore viene trasformata in uno spostamento lineare, consentendo il controllo diretto del volume erogato. Le syringe pump offrono elevata accuratezza, buona ripetibilità e la possibilità di generare flussi relativamente costanti, risultando particolarmente adatte ad applicazioni che richiedono il dosaggio controllato di quantità limitate di prodotto.

a.



b.



c.

Figura 1.4: ***a.*** *Sistema pneumatico per erogazione;* ***b.*** *Pompa peristaltica;* ***c.*** *Syringe pump*

Yang et al. evidenziano inoltre come le syringe pump presentino vantaggi significativi rispetto ai tradizionali sistemi pneumatici, grazie alla semplicità di controllo e alla capacità di regolare con precisione la pressione e il volume erogato mediante il movimento controllato del pistone. Gli autori descrivono una configurazione composta da una siringa commerciale accoppiata a un attuatore lineare comandato da motore passo-passo, soluzione che consente un'elevata accuratezza nelle operazioni di dispensazione [11].

Gervasi et al. (2021), nel confronto tra pompe peristaltiche e syringe pump per applicazioni di dosaggio di liquidi, sottolineano che le pompe peristaltiche sono adatte al trasferimento di volumi da millilitri a litri, ma presentano un flusso pulsante e richiedono calibrazione, mentre le syringe pump sono più indicate quando è richiesta una portata costante e un controllo accurato di piccoli volumi [12].

Nel caso della decorazione alimentare, il sistema a siringa può essere interpretato come una forma semplificata di estrusione controllata. Il prodotto viene spinto attraverso un ugello e depositato lungo una traiettoria definita, in modo analogo a quanto avviene nei processi di Direct Ink Writing applicati a materiali non newtoniani. Guo et al. (2023) mostrano che nei processi DIW la qualità del filamento depositato dipende dal comportamento shear-thinning del materiale, dalla geometria siringa-ugello, dalla distribuzione della velocità, dallo shear

stress e dalla stabilità del flusso in uscita. In particolare, gli autori evidenziano che materiali con comportamento shear-thinning sono adatti all'estrusione perché la viscosità diminuisce durante il passaggio nell'ugello e aumenta nuovamente dopo il deposito, contribuendo al mantenimento della forma [13].

Questa analogia è particolarmente utile per interpretare il caso della maionese. Anche se il processo sviluppato in questa tesi non è un processo di stampa 3D alimentare in senso stretto, esso presenta elementi comuni con l'estrusione controllata di materiali semisolidi: il prodotto viene contenuto in una siringa, spinto da uno stantuffo, forzato attraverso un ugello e depositato seguendo una traiettoria programmata. Di conseguenza, le proprietà reologiche del prodotto e la geometria dell'ugello diventano parametri fondamentali per garantire continuità del flusso e qualità del cordone.

La scelta di un sistema di dispensazione per prodotti alimentari semisolidi non può quindi prescindere dalla reologia. La reologia studia la deformazione e il flusso dei materiali sottoposti a sollecitazioni meccaniche. Rao sottolinea che molti alimenti fluidi e semisolidi non possono essere descritti come fluidi newtoniani, poiché la loro risposta dipende dalla microstruttura interna, dalla presenza di particelle disperse, goccioline emulsionate, macromolecole, reti colloidali e interazioni tra le fasi [14].

Per un fluido newtoniano, la relazione tra sforzo di taglio e shear rate è lineare:

$$\tau = \mu \cdot \dot{\gamma}$$

dove:

τ = sforzo di taglio [Pa]

μ = viscosità dinamica [Pa·s]

$\dot{\gamma}$ = shear rate [s^{-1}]

In questo caso la viscosità è costante e non dipende dalla velocità di deformazione. Al contrario, nei fluidi non newtoniani la viscosità apparente varia al variare dello shear rate. Molti prodotti alimentari, tra cui salse, condimenti, yogurt, puree e maionese, mostrano comportamento pseudoplastico o shear-thinning: la viscosità apparente diminuisce all'aumentare dello shear rate [14].

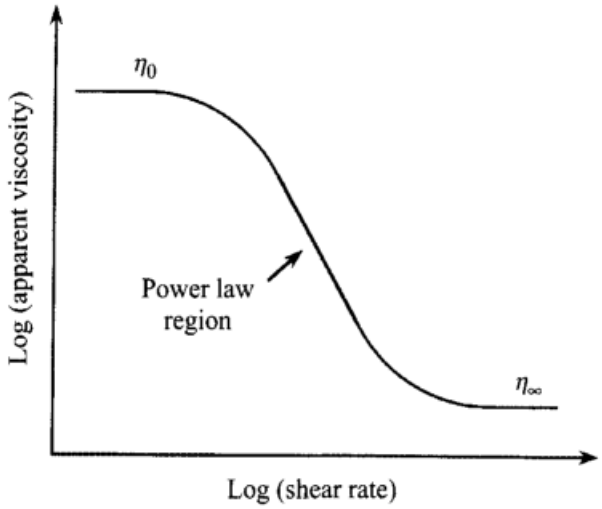


Figura 1.5: Viscosità apparente in funzione dello shear rate per un fluido pseudoplastico

La viscosità apparente può essere definita come:

$$\eta_a = \tau / \dot{\gamma}$$

dove:

η_a = viscosità apparente [Pa·s]

Per descrivere matematicamente il comportamento shear-thinning viene frequentemente utilizzato il modello di Ostwald-de Waele, noto anche come Power Law:

$$\tau = K \cdot \dot{\gamma}^n$$

dove:

K = indice di consistenza [Pa·s^n]

n = indice di comportamento al flusso

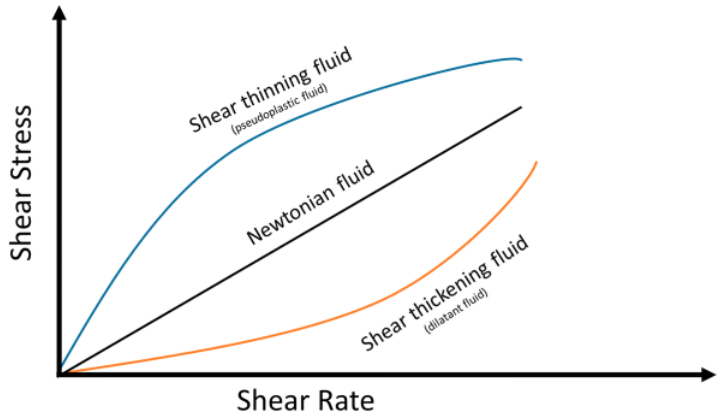


Figura 1.6: Confronto tra il comportamento reologico di un fluido shear-thinning (pseudoplastico), un fluido Newtoniano e un fluido shear-thickening (dilatante)

Per un fluido newtoniano si ha n = 1; per un fluido pseudoplastico si ha n < 1; per un fluido dilatante o shear-thickening si ha n > 1. Sostituendo la Power Law nella definizione di viscosità apparente si ottiene:

$$\eta_a = K \cdot \dot{\gamma}^{n-1}$$

Questa relazione mostra che, quando n < 1, l'esponente n − 1 è negativo e quindi la viscosità apparente diminuisce all'aumentare dello shear rate. Dal punto di vista ingegneristico, ciò significa che un materiale come la maionese può essere molto viscoso a riposo, ma diventare più facilmente estrudibile quando attraversa un ugello, dove i gradienti di velocità sono elevati. Rao evidenzia che il modello Power Law è ampiamente utilizzato per descrivere il comportamento di alimenti fluidi e semisolidi, pur essendo valido entro l'intervallo di shear rate in cui i parametri vengono determinati sperimentalmente [14].

Molti alimenti semisolidi presentano inoltre una tensione di snervamento, o yield stress, cioè un valore minimo di sforzo che deve essere superato affinché il materiale inizi a fluire. In presenza di yield stress, il comportamento può essere descritto mediante il modello di Herschel-Bulkley:

$$\tau = \tau_0 + K \cdot \dot{\gamma}^n$$

dove:

τ_0 = tensione di snervamento [Pa]

K = indice di consistenza [$Pa \cdot s^n$]

n = indice di comportamento al flusso

La presenza di τ_0 è particolarmente importante nella progettazione di un sistema di dispensazione, perché definisce la soglia minima di sforzo che deve essere superata per avviare l'erogazione. In un sistema a siringa, ciò si traduce nella necessità di generare una forza sufficiente sullo stantuffo per vincere la resistenza iniziale del materiale e le perdite di carico nell'ugello. Rao considera il concetto di yield stress una grandezza di forte interesse ingegneristico per la descrizione dei processi alimentari, anche quando la sua definizione teorica può risultare complessa [14].

Tra i prodotti alimentari caratterizzati da comportamento non newtoniano, la maionese rappresenta un caso particolarmente significativo. Essa è un'emulsione olio-in-acqua altamente concentrata, nella quale goccioline di olio sono disperse all'interno di una fase acquosa continua. McClements definisce le emulsioni alimentari come sistemi costituiti da due liquidi immiscibili, generalmente olio e acqua, nei quali una fase è dispersa sotto forma di goccioline nell'altra.

L'elevata frazione volumetrica di fase dispersa è uno dei motivi principali del comportamento semisolido della maionese. McClements osserva che, al crescere della frazione volumetrica delle goccioline, la viscosità dell'emulsione aumenta in modo non lineare. A basse concentrazioni di fase dispersa, come nel latte, il prodotto si comporta come un fluido a bassa viscosità; a concentrazioni intermedie, come nella panna, la viscosità aumenta; a concentrazioni molto elevate, come nella maionese, le goccioline risultano strettamente impacchettate e il sistema assume comportamento viscoelastico o gel-like [15].

La maionese presenta dunque una microstruttura nella quale le goccioline d'olio sono fortemente ravvicinate e interagiscono tra loro. Friberg et al. descrivono le emulsioni concentrate come sistemi nei quali la deformabilità delle goccioline, lo strato interfacciale e le interazioni colloidali influenzano la risposta meccanica complessiva. Nei sistemi molto concentrati, come la maionese, le goccioline possono risultare impacchettate e deformate, assumendo forme non perfettamente sferiche; ciò contribuisce alla formazione di una struttura resistente al flusso e alla comparsa di proprietà viscoelastiche [16].

La letteratura sperimentale conferma che la maionese è un materiale complesso dal punto di vista reologico. Štern et al. (2008) descrivono la maionese come un'emulsione di olio vegetale e acqua, nella quale il tuorlo d'uovo agisce da emulsionante. Gli autori evidenziano che la

maionese possiede una struttura semisolida con marcate proprietà viscoelastiche e che diventa più fluida a shear moderati. Nello stesso studio, condotto su maionese modificata con yogurt, vengono misurati yield value, viscosità apparente, tissotropia e moduli viscoelastici, mostrando che yield value e viscosità apparente diminuiscono sensibilmente all'aumentare della temperatura tra 15, 20 e 25 °C [17].

Questo risultato è rilevante per la progettazione di un sistema di dispensazione perché indica che la risposta della maionese non è indipendente dalle condizioni ambientali. A parità di sistema meccanico, una variazione di temperatura può modificare la viscosità apparente e quindi la forza necessaria per l'estrusione. Nel caso di una cella robotizzata, tale aspetto può influenzare l'avvio del flusso, la continuità del cordone e la quantità depositata.

Štern et al. (2008) mostrano inoltre che la maionese può essere considerata, dal punto di vista reologico, un corpo viscoplastico tissotropico. La tissotropia viene valutata come area compresa tra curva di salita e curva di discesa in una prova di flusso. Questo significa che il materiale non dipende soltanto dal valore istantaneo dello shear rate, ma anche dalla storia di taglio subita. In pratica, la maionese può ridurre la propria viscosità durante l'applicazione dello sforzo e recuperare parzialmente la struttura quando lo sforzo viene rimosso [17].

Anche Rukke e Schüller (2019) confermano il comportamento pseudoplastico e tissotropico della maionese. Gli autori hanno analizzato diverse maionesi commerciali, misurando modulo elastico, viscosità, area di isteresi, pH e distribuzione dimensionale delle particelle e i risultati mostrano che, pur variando per composizione e contenuto di grasso, presentano caratteristiche viscoelastiche e tissotropiche; inoltre, la dimensione delle particelle risulta un parametro importante perché influenza reologia, stabilità, texture e gusto dell'emulsione [18].

Oltre alla composizione, anche il processo di omogeneizzazione influenza fortemente le proprietà reologiche della maionese. Bredikhin et al. (2022), studiando una maionese con il 76% di fase oleosa e aggiunta di olio di lino, mostrano che il tempo di omogeneizzazione e la velocità del rotore modificano viscosità apparente, indice di consistenza e indice di flusso. Gli autori confermano che i campioni analizzati appartengono ai sistemi non newtoniani di tipo pseudoplastico e riportano valori dell'indice di flusso $n < 1$, coerenti con il comportamento shear-thinning. Inoltre, osservano che un eccessivo tempo di omogeneizzazione o una velocità del rotore troppo elevata possono distruggere la struttura dell'emulsione, riducendo consistenza e viscosità apparente [19].

Questo aspetto è importante perché dimostra che la maionese non deve essere considerata soltanto come un materiale viscoso, ma come una struttura emulsionata sensibile alla storia

meccanica. Durante un processo di dispensazione, il prodotto subisce deformazioni e tagli nel serbatoio, nella siringa, nell'ugello e durante il deposito. Sollecitazioni eccessive o non controllate potrebbero alterare la microstruttura dell'emulsione, mentre una sollecitazione adeguata può favorire il flusso senza compromettere la stabilità del prodotto.

Dal punto di vista applicativo, il comportamento shear-thinning della maionese è vantaggioso. Durante il passaggio attraverso l'ugello, il prodotto è sottoposto a shear rate elevati: in queste condizioni la viscosità apparente diminuisce e l'estrusione risulta facilitata. Una volta depositato, lo shear rate diminuisce bruscamente; la viscosità apparente aumenta nuovamente e la struttura viscoelastica contribuisce al mantenimento della forma. Questa combinazione di elevata viscosità a riposo, riduzione della viscosità durante l'estrusione e recupero strutturale dopo il deposito è particolarmente favorevole per applicazioni decorative.

Il comportamento può essere rappresentato qualitativamente con un grafico viscosità apparente–shear rate. In tale grafico, l'asse orizzontale rappresenta lo shear rate $\dot{\gamma}$ [s^{-1}], mentre l'asse verticale rappresenta la viscosità apparente η_a [Pa·s]. Per un fluido shear-thinning, la curva è decrescente: a bassi shear rate la viscosità è elevata, mentre ad alti shear rate diminuisce progressivamente. Questo andamento è coerente con il modello Power Law nel caso $n < 1$ e con le evidenze sperimentali riportate negli studi sulla maionese [14][18][19].

La letteratura mostra, quindi, che la dispensazione di prodotti alimentari semisolidi deve essere affrontata come un problema integrato di meccanica, reologia e controllo. Le caratteristiche reologiche della maionese evidenziano la necessità di adottare un sistema di dispensazione capace di generare una spinta controllata e ripetibile, mantenendo al contempo un'elevata precisione volumetrica. La presenza di comportamento pseudoplastico e tissotropico richiede infatti un controllo accurato della velocità di estrusione e della pressione applicata, al fine di ottenere depositi uniformi e geometricamente stabili.

Alla luce delle tecnologie analizzate nello stato dell'arte, la soluzione basata su una syringe pump azionata da un attuatore lineare rappresenta un compromesso efficace tra semplicità costruttiva, accuratezza di dosaggio e capacità di gestire materiali viscosi. Per questo motivo tale architettura è stata scelta come base per lo sviluppo dell'end-effector oggetto del presente lavoro, descritto nei capitoli successivi.

2. ANALISI DEL PROCESSO E ARCHITETTURA DEL SISTEMA

2.1. DESCRIZIONE DEL PROCESSO MANUALE E REQUISITI FUNZIONALI

Il processo oggetto di studio riguarda la decorazione o deposizione controllata di materiale alimentare mediante un principio analogo a quello della sac à poche. Nel processo manuale, l'operatore impugna il sistema di erogazione, esercita una pressione sul materiale contenuto nel serbatoio flessibile e contemporaneamente guida l'ugello lungo la traiettoria desiderata. La qualità del risultato finale dipende quindi dalla capacità dell'operatore di coordinare tre aspetti fondamentali: la pressione esercitata sul materiale, la velocità di avanzamento dell'ugello e la distanza rispetto alla superficie di deposizione.

Nel caso di materiali alimentari viscosi o semi-solidi, come creme, salse o emulsioni, il controllo manuale può risultare complesso a causa della risposta non immediata del materiale alla pressione applicata. Una variazione eccessiva della forza esercitata può produrre accumuli localizzati, discontinuità o variazioni di sezione del cordone depositato. Allo stesso modo, una velocità di avanzamento non uniforme può generare irregolarità geometriche, soprattutto nelle zone di cambio direzione o in corrispondenza dell'avvio e dell'arresto dell'erogazione.

L'automazione del processo ha quindi l'obiettivo di rendere più ripetibile e controllabile un'operazione tradizionalmente affidata all'esperienza dell'operatore. In particolare, l'impiego di una cella robotizzata consente di separare il controllo della traiettoria dal controllo dell'erogazione. Il manipolatore robotico si occupa della movimentazione spaziale dell'ugello, mentre un sistema di attuazione dedicato regola l'avanzamento dello stantuffo e, di conseguenza, la fuoriuscita del materiale.

Dal punto di vista funzionale, la cella deve soddisfare alcuni requisiti principali. In primo luogo, il sistema deve essere in grado di localizzare la vaschetta sul piano di lavoro, indipendentemente dal suo orientamento. In secondo luogo, il robot deve poter generare una traiettoria coerente con la geometria del prodotto da decorare. In terzo luogo, il tool di erogazione deve garantire un controllo sufficientemente regolare della portata, in modo da ottenere un deposito continuo e uniforme. Infine, l'intero sistema deve essere riconfigurabile, così da poter modificare traiettoria, ugello o parametri di erogazione in funzione dell'applicazione.

I requisiti funzionali possono quindi essere ricondotti a quattro macro-funzioni: acquisizione della scena, elaborazione delle informazioni geometriche, movimentazione dell'utensile e controllo della deposizione. La prima funzione è svolta dal sistema di visione RGB-D, che permette di acquisire informazioni sia cromatiche sia geometriche. La seconda funzione è affidata alla pipeline di visione artificiale, che consente di individuare il pezzo e ricavare i punti utili per la traiettoria. La terza funzione è svolta dal robot collaborativo, che movimenta l'end-effector nello spazio operativo. La quarta funzione è affidata al tool di erogazione, nel quale un asse lineare motorizzato controlla l'avanzamento dello stantuffo.

Nel complesso, il passaggio da un processo manuale a un processo robotizzato non consiste semplicemente nella sostituzione dell'operatore con un robot, ma nella scomposizione del gesto manuale in funzioni controllabili. Il movimento della mano viene sostituito dalla traiettoria robotica, la pressione esercitata dall'operatore viene sostituita dall'avanzamento controllato dello stantuffo, mentre l'osservazione visiva del prodotto viene affidata alla camera RGB-D e agli algoritmi di elaborazione dell'immagine.

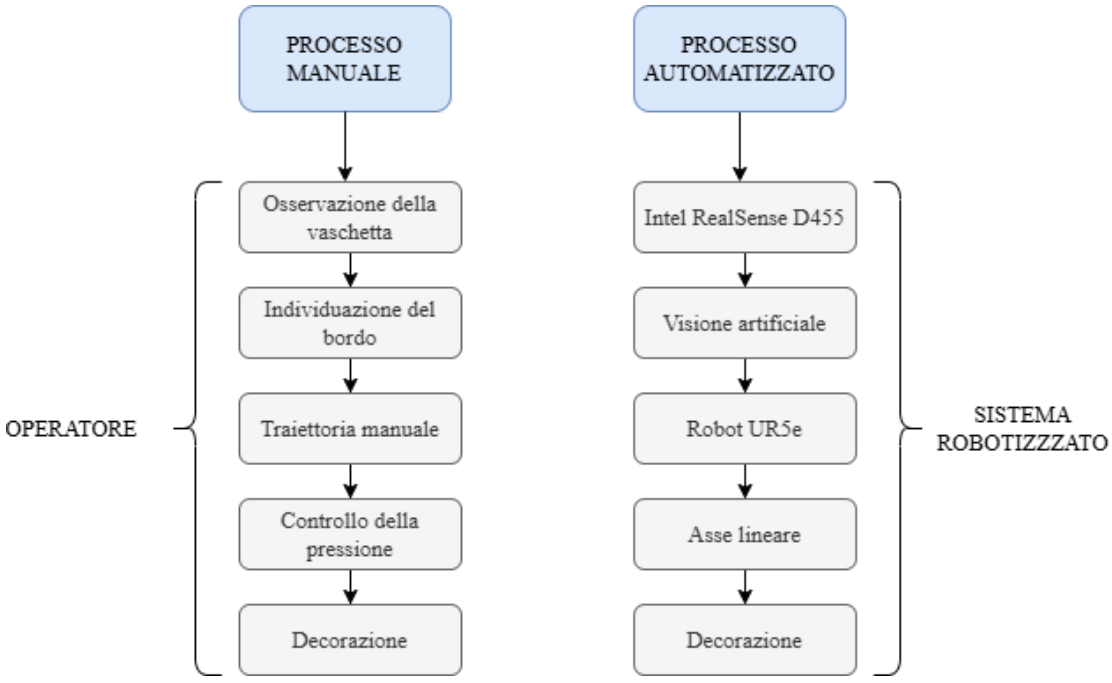


Figura 2.1: Confronto schematico tra processo manuale e processo robotizzato

2.2. ARCHITETTURA GENERALE DELLA CELLA ROBOTIZZATA

La soluzione sviluppata per l'automazione del processo di decorazione è costituita da una cella robotizzata composta da un manipolatore collaborativo Universal Robots UR5e, una telecamera Intel RealSense D455 e un sistema di erogazione appositamente progettato per la deposizione di materiali alimentari viscosi. L'intero sistema, visibile in *Figura 2.2*, è installato su un banco di lavoro che definisce l'area operativa all'interno della quale viene posizionata la vaschetta da decorare.

Figura 2.2: Layout generale della cella robotizzata

Il robot UR5e rappresenta l'elemento principale della cella ed è responsabile della movimentazione del tool di erogazione lungo la traiettoria di deposizione. All'estremità del manipolatore è installata la telecamera RGB-D Intel RealSense D455, configurata secondo un approccio eye-in-hand, che consente di acquisire informazioni geometriche direttamente dal punto di vista dell'end-effector. Sul medesimo organo terminale è montato il sistema di

erogazione, costituito da una siringa, da un asse lineare motorizzato per l'azionamento dello stantuffo e da un ugello intercambiabile per la deposizione del materiale.
Dal punto di vista funzionale, la cella può essere suddivisa in tre sottosistemi principali:

- il *sottosistema di visione*, incaricato dell'acquisizione della scena e dell'estrazione delle informazioni geometriche necessarie al processo;
- il *sottosistema robotico*, responsabile della movimentazione dell'utensile all'interno dello spazio di lavoro;
- il *sottosistema di erogazione*, deputato al controllo della fuoriuscita del materiale attraverso l'ugello.

La comunicazione tra i diversi sottosistemi, schematizzata nella *Figura 2.3*, è gestita da un elaboratore esterno sul quale vengono eseguiti gli algoritmi di visione artificiale e le procedure di pianificazione della traiettoria. Le informazioni acquisite dalla telecamera vengono elaborate per ricavare la geometria utile alla decorazione; tali informazioni vengono successivamente utilizzate per generare il percorso che il robot dovrà seguire durante la fase di deposizione.

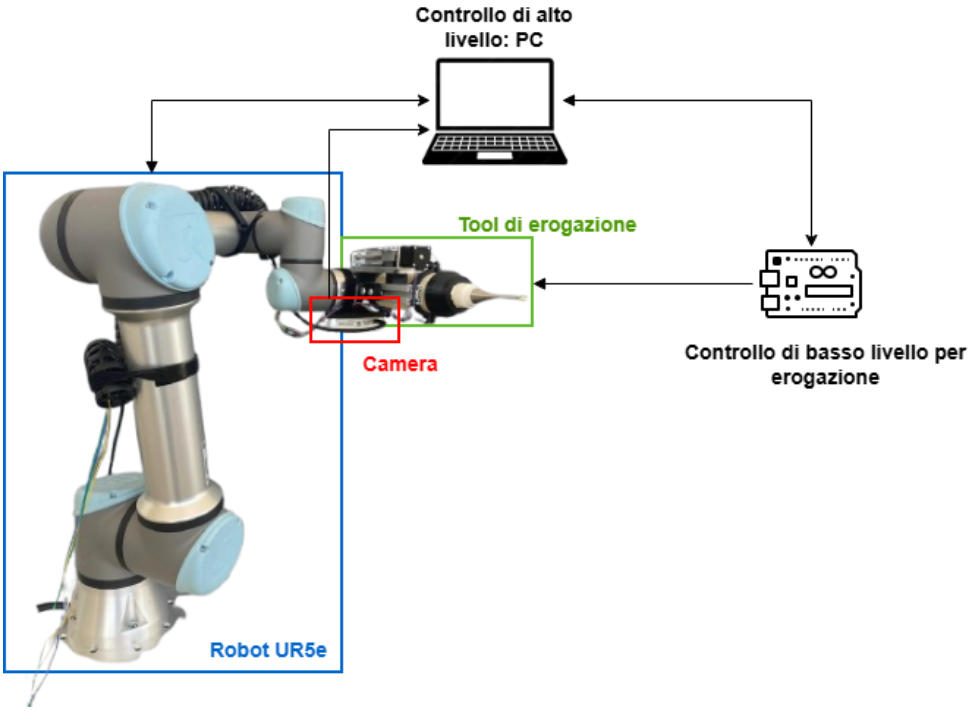


Figura 2.3: Architettura meccatronica della cella robotizzata

Dal punto di vista fisico, il layout della cella è stato progettato in modo da garantire la completa sovrapposizione tra l'area osservabile dalla telecamera e l'area raggiungibile dal manipolatore. Tale condizione risulta fondamentale affinché le informazioni geometriche estratte dal sistema di visione possano essere utilizzate direttamente dal robot durante la fase di esecuzione della traiettoria.

Il ciclo operativo della cella prevede l'acquisizione della scena mediante la telecamera RGB-D, l'elaborazione delle informazioni geometriche necessarie alla generazione della traiettoria, la trasformazione delle coordinate nel sistema di riferimento del robot e l'esecuzione della deposizione mediante il tool di erogazione. I dettagli relativi alla calibrazione robot-camera, alla pipeline di visione artificiale e al controllo sincronizzato del sistema di erogazione saranno approfonditi nei capitoli successivi.

L'architettura proposta consente di realizzare una soluzione modulare e riconfigurabile, nella quale i diversi sottosistemi mantengono funzioni ben distinte ma operano in maniera coordinata. Tale approccio permette di adattare il sistema a vaschette di diverse dimensioni e a differenti strategie di deposizione, senza modificare l'intera struttura della cella robotizzata.

3. COMPONENTI HARDWARE

3.1. ROBOT COLLABORATIVO UR5e

Il sistema robotico impiegato nella cella è il robot collaborativo UR5e prodotto da Universal Robots. Il manipolatore è costituito da una struttura seriale antropomorfa a sei gradi di libertà, composta da una base fissa e da sei giunti rotoidali azionati elettricamente. Tale configurazione consente il posizionamento e l'orientamento arbitrario dell'end-effector all'interno dello spazio di lavoro.

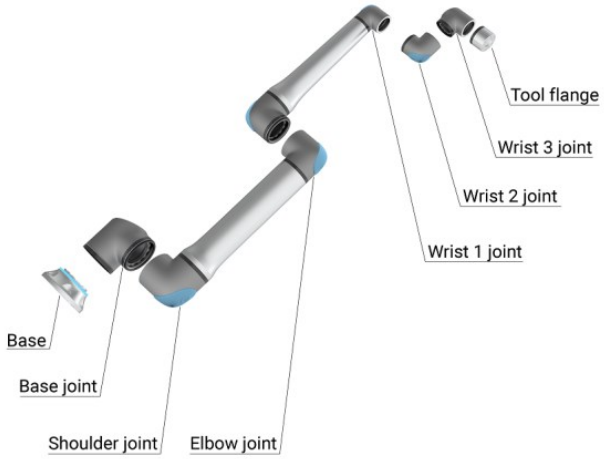


Figura 3.1: Componenti principali del robot UR5e

Nella *Figura 3.1* sono evidenziati i principali elementi costitutivi del braccio robotico [20]: I primi tre giunti (Base, Shoulder ed Elbow) determinano principalmente il posizionamento del TCP (Tool Center Point) nello spazio, mentre i tre giunti del polso (Wrist 1, Wrist 2 e Wrist 3) consentono l'orientamento dell'end-effector rispetto al pezzo da lavorare.

La presenza di sei assi controllati permette di gestire indipendentemente posizione e orientamento dell'utensile, caratteristica fondamentale per applicazioni nelle quali è necessario mantenere l'ugello di deposizione correttamente orientato rispetto alla superficie da decorare.

Le principali caratteristiche tecniche del robot sono riportate nella *Tabella 3.1*.

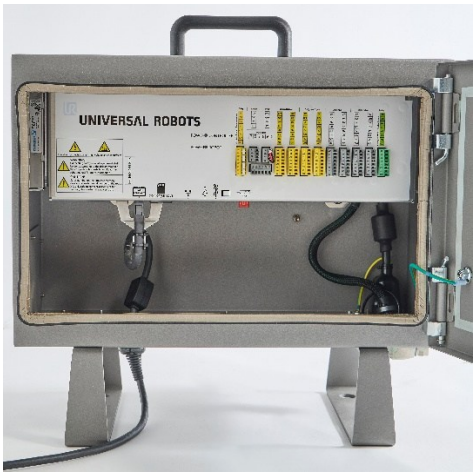
Parametro	**Valore**

Gradi di libertà	6
Payload massimo	5 kg
Sbraccio massimo	850 mm
Ripetibilità	±0,03 mm
Peso del manipolatore	20,6 kg
Velocità TCP massima	1 m/s
Classe di protezione	IP54

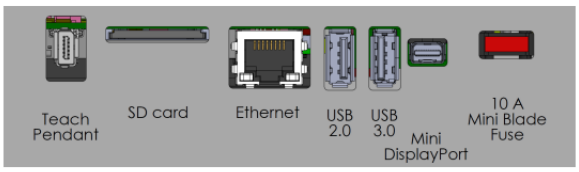
Tabella 3.1: Specifiche tecniche del robot UR5e

Oltre al braccio robotico, il sistema UR5e comprende un'unità di controllo (*Figura 3.2**a***) e un terminale operatore denominato Teach Pendant. L'unità di controllo è responsabile dell'alimentazione del robot, della gestione degli azionamenti dei sei assi e delle funzioni di sicurezza integrate. Essa rappresenta inoltre il punto di interfaccia tra il manipolatore e i dispositivi esterni, mettendo a disposizione le porte di connessione (*Figura 3.2**b***) e gli ingressi/uscite (I/O) necessari all'integrazione del sistema (*Figura3.2**c***). Le porte consentono il collegamento con apparecchiature esterne, mentre gli I/O costituiscono le interfacce elettriche utilizzate per lo scambio di segnali, la comunicazione e la configurazione delle funzionalità del robot.

a.



b.



c.

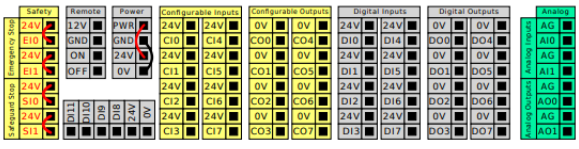


*Figura 3.2: **a.** Unità di controllo; **b.** Porte di connessione esterne; **c.** Gruppi di ingresso e uscita (I/O)*

Il Teach Pendant rappresenta l'interfaccia uomo-macchina del sistema e consente la programmazione, la configurazione e il monitoraggio del robot attraverso l'ambiente software PolyScope. Tramite tale dispositivo, raffigurato nella *Figura 3.3*, è possibile definire i parametri operativi, configurare gli utensili e gestire le principali funzionalità del manipolatore.

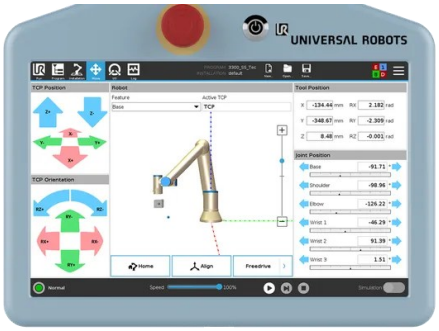


Figura 3.3: Teach Pendant

Le prestazioni del robot risultano adeguate all'applicazione sviluppata, poiché il carico complessivo della telecamera e del sistema di erogazione applicato alla flangia è significativamente inferiore al payload nominale.

Lo spazio di lavoro del robot è definito dall'insieme delle posizioni raggiungibili dal Tool Center Point (TCP) in funzione della configurazione cinematica e dei limiti articolari dei sei giunti. Il reach massimo del manipolatore è pari a 850 mm e rappresenta la massima distanza raggiungibile tra il TCP e l'asse della base robotica.

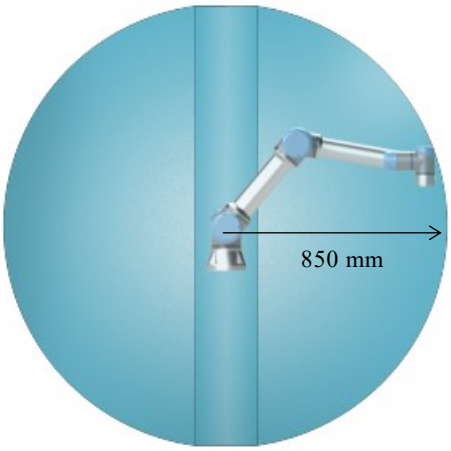


Figura 3.4: Spazio di lavoro del robot UR5e

La *Figura 3.4* mostra il volume operativo dichiarato dal costruttore [20]. Nel layout della cella il robot è stato posizionato in modo da garantire la completa copertura dell'area di deposizione, evitando configurazioni cinematiche prossime ai limiti articolari e riducendo il rischio di singolarità durante l'esecuzione delle traiettorie.

La scelta dell'UR5e è stata motivata da diversi fattori. Innanzitutto, il robot offre un buon compromesso tra dimensioni, payload e volume operativo. Inoltre, la disponibilità di interfacce software dedicate facilita l'integrazione con applicazioni sviluppate esternamente in ambiente Python.

Dal punto di vista applicativo, la ripetibilità del manipolatore e la fluidità dei movimenti consentono di eseguire traiettorie di deposizione con elevata regolarità, contribuendo alla qualità del deposito finale. La natura collaborativa del sistema rappresenta infine un vantaggio in prospettiva industriale, poiché permette l'integrazione della cella in contesti nei quali operatori e robot possono condividere il medesimo ambiente di lavoro.

3.2. SISTEMA DI VISIONE ARTIFICIALE

Il sistema di visione artificiale costituisce uno dei sottosistemi fondamentali della cella robotizzata, in quanto permette di acquisire informazioni geometriche sull'ambiente di lavoro e sul prodotto da decorare. Nel presente progetto tale funzione è affidata a una camera Intel RealSense D455, *Figura 3.5*, appartenente alla famiglia RealSense D400 Series, integrata direttamente sull'end-effector del robot collaborativo UR5e.

Figura 3.5: Intel RealSense D455

La scelta di utilizzare una camera RGB-D deriva dalla necessità di disporre non solo di un'immagine bidimensionale della scena, ma anche di informazioni metriche relative alla

profondità. Una tradizionale camera RGB fornisce infatti dati legati esclusivamente al colore e all'intensità luminosa dei pixel, mentre una camera RGB-D associa a ciascun punto dell'immagine anche una distanza rispetto al sensore. Questa caratteristica risulta particolarmente utile in applicazioni robotiche, nelle quali il riconoscimento dell'oggetto deve essere collegato alla successiva movimentazione del robot nello spazio.

L'obbiettivo della camera è riconoscere la posizione e l'orientamento della vaschetta sul piano di lavoro e fornire informazioni tridimensionali utilizzabili per la generazione della traiettoria. La camera è installata secondo una configurazione eye-in-hand, cioè solidale al polso del robot, in questo modo il sensore si muove insieme all'end-effector. Questo tipo di approccio permette alla camera di poter acquisire la scena da un punto di vista più vantaggioso in quanto può essere spostato con i movimenti del robot e inoltre dà alla camera la possibilità di esplorare l'area di lavoro. D'altra parte, però, è richiesta una corretta calibrazione camera-robot (hand-eye calibration) affinché le coordinate rilevate possano essere trasformate accuratamente nel sistema di riferimento del manipolatore.

La Intel RealSense D455 è una camera di profondità basata su tecnologia stereo, progettata per applicazioni di visione tridimensionale, robotica, prototipazione hardware e sviluppo software. La famiglia RealSense D400 Series integra un modulo stereo di profondità e un processore di visione dedicato, con connessione verso il processore host tramite USB 2.0/USB 3.1 Gen 1 o interfaccia MIPI, a seconda della configurazione del prodotto. In questo progetto, lo scambio di dati tra la camera e il computer, che coordina tutti i sottoinsiemi, avviene tramite interfaccia USB 3.1 Gen 1, necessaria per sostenere il flusso dati generato dai sensori RGB e depth ad elevata frequenza di acquisizione. Nella *Figura 3.6* l'intero sistema camera è racchiuso in un diagramma a blocchi che evidenzia alcune componenti come:

- una coppia di sensori stereo per la misura della profondità;
- un proiettore infrarosso;
- un sensore RGB;
- il processore RealSense Vision Processor D4;
- un'unità inerziale IMU a 6 gradi di libertà, non rappresentata nello schema.

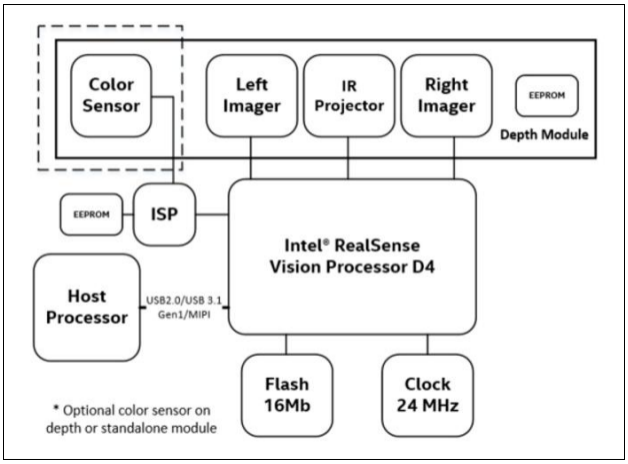


Figura 3.6: diagramma a blocchi del sistema camera [21]

Le principali specifiche tecniche sono racchiuse nella *Tabella 3.2*:

Tecnologia di misura	Stereo vision attiva
Range ideale	0,4 m - 6 m
Risoluzione massima depth	1280 × 720 px
Min-Z alla massima risoluzione	0,52 m
Accuratezza depth	<2% fino a 4m
Risoluzione massima RGB	1280 × 800 px
Frame Rate RGB/depth	30 fps – 90 fps

Tabella 3.2 : Specifiche tecniche Intel RealSense D455

Il principio di misura della profondità della RealSense D455 si basa sulla visione stereoscopica. Tale principio è analogo al funzionamento della visione umana: due sensori osservano la stessa scena da due posizioni leggermente differenti e, confrontando le due immagini, è possibile stimare la distanza degli oggetti osservati. La visione stereo è una delle principali tecnologie impiegate nella visione artificiale tridimensionale, insieme alla profilazione 3D, alla Time of Flight (ToF) e alla luce strutturata. Come per altri tipi di tecnologia di imaging 3D, l'obiettivo della visione stereoscopica è quello di risolvere il problema della percezione della profondità, essenziale in tutte le applicazioni 3D.

Nel sistema stereo sono presenti due sensori, indicati come imager sinistro e imager destro, separati da una distanza nota detta baseline. Quando un punto della scena viene osservato dai

due sensori, esso viene proiettato in posizioni differenti sui rispettivi piani immagini $[x_L,y_L]^T$ e $[x_R,y_R]^T$. Dal momento che le due camere sono allineate lungo l'asse y, le coordinate verticali coincidono $y_L = y_R$, mentre la differenza tra le coordinate orizzontali prende il nome di disparità d

$$d = x_L - x_R$$

come mostrato in *Figura 3.7a*.

Conoscendo la baseline B e la lunghezza focale f, la depth può essere calcolata sfruttando la similitudine dei due triangoli (*Figura 3.7b*):

$$Z = \frac{f \cdot B}{d}$$

dove:

- Z è la profondità del punto;
- f è lunghezza focale della camera;
- B è la baseline stereo;
- d è la disparità.

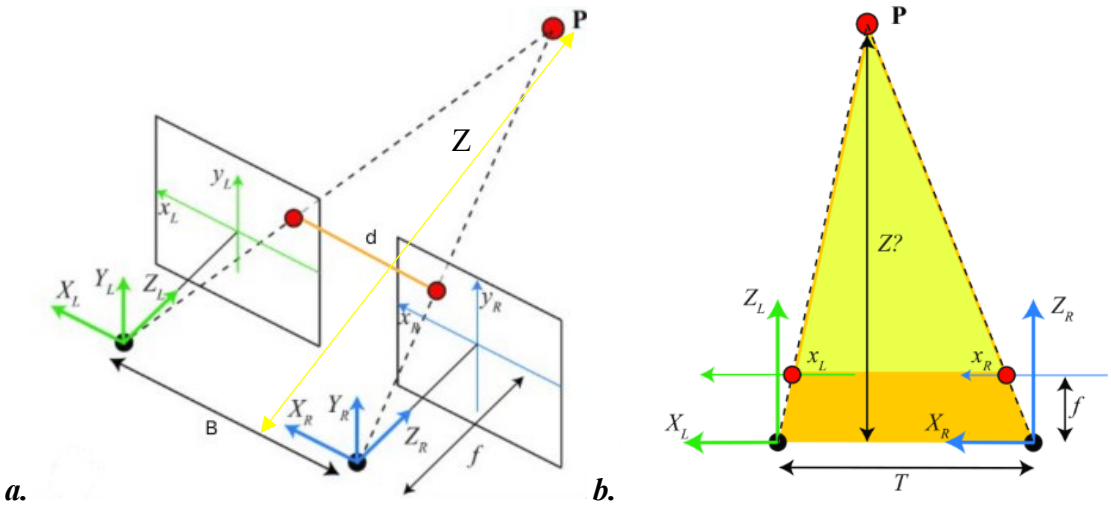


Figura 3.7 : Triangolazione stereo

È possibile notare come Z e d siano inversamente proporzionali: questo sta a indicare che gli oggetti più vicini alla camera generano una disparità maggiore, mentre quelli più lontani

producono una disparità più bassa. che la stima della profondità, all'aumentare della lontananza, diventa sempre più sensibile agli errori di misura della disparità.
Il modulo di profondità D450 installato nella telecamera D455 ha una baseline pari a 95 mm, maggiore rispetto ad altri modelli. Questo risulta vantaggioso perché permette di incrementare la precisione della triangolazione alle medie e lunghe distanze.
La D455 non utilizza una stereo visione puramente passiva, ma una stereo visione attiva. Nei sistemi stereo passivi, il calcolo della profondità dipende dalla presenza di dettagli visibili sulla superficie osservata. Se l'oggetto presenta superfici uniformi, lisce o poco texturizzate, può essere difficile individuare corrispondenze affidabili tra l'immagine sinistra e l'immagine destra. La RealSense D455 integra un proiettore infrarosso che genera una trama luminosa non visibile all'occhio umano, in questo modo vengono introdotti pattern infrarossi artificiali che facilitano il riconoscimento dei punti corrispondenti tra le immagini stereo.
Il processo che viene eseguito dalla camera può essere scomposto in tre fasi: il proiettore IR illumina la scena, i sensori destro e sinistro acquisiscono le immagini e infine il processore D4 calcola la disparità e stima la profondità. Il risultato è quindi una mappa di profondità, cioè un'immagine in cui ogni pixel è associato ad un valore di distanza. È importante sottolineare, come viene evidenziato nel manuale [21] e nella *Figura 3.8*, che la depth calcolata non è la distanza euclidea tra punto di osservazione e punto osservato (range), ma è la distanza rispetto ad un piano parallelo agli imagers e rappresenta la coordinata lungo l'asse ottico della camera (asse Z del sistema di riferimento della camera).

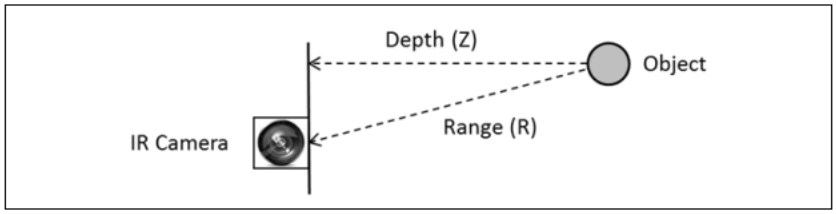


Figura 3.8: differenza tra range e depth nel calcolo della profondità con camera Intel RealSense D455

Un altro parametro da tenere in considerazione per il dimensionamento della cella robotizzata è il FOV (field of view), ovvero il campo visivo che determina l'area del piano di lavoro osservabile dalla camera ad una determinata distanza. Quello relativo al modulo depth è pari a circa 87° in orizzontale, 58° in verticale e 95° in diagonale, mentre quello del sensore RGB

è pari a circa 90° in orizzontale, 65° in verticale e 98°in diagonale. La scelta della posa di osservazione è stata guidata anche da considerazioni legate alle limitazioni del sensore di profondità. In particolare, è stato necessario verificare che la vaschetta non ricadesse nella zona di *Invalid Depth Band*, ovvero quella regione laterale dell'immagine in cui la triangolazione stereo non produce misure valide per mancanza di sovrapposizione tra i campi visivi dei due imager, come illustrato in *Figura 3.9* [21].

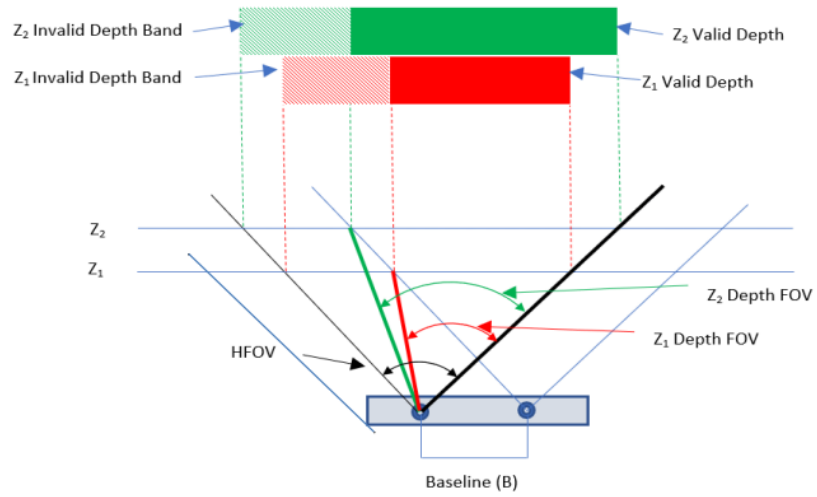


Figura 3.9: Rappresentazione schematica dell'Invalid Depth Band e del campo visivo (FOV)

Altrettanto rilevante è la distanza minima alla quale il sensore è in grado di fornire dati di profondità validi. Per i modelli D450/D455, il manuale riporta valori di Min-Z variabili in funzione della risoluzione impostata. Ad esempio, alla risoluzione 1280 × 720 la distanza minima è pari a 520 mm, mentre a risoluzioni inferiori può scendere fino a 180 mm nel caso del formato 424 × 240. Per l'applicazione sviluppata, questo aspetto deve essere considerato nella scelta della posa di acquisizione: la camera deve essere collocata a una distanza sufficiente dal piano di lavoro per garantire una misura depth valida, ma non eccessiva, per non ridurre la risoluzione utile sull'oggetto.

Dal punto di vista dell'integrazione nella cella, la camera è montata sull'end-effector del robot mediante un supporto dedicato stampato 3D, che verrà trattato approfonditamente nel capitolo 3.3. Le dimensioni nominali sono pari a circa 124 × 29 × 26 mm e massa pari a circa 116 g e queste rendono il dispositivo sufficientemente compatto per essere installato sul polso del robot. Il fissaggio tramite viti M4 deve essere sufficientemente rigido da garantire l'assenza di

giochi o cedimenti strutturali durante il movimento del robot, condizione necessaria per mantenere costante la posizione relativa tra camera ed end-effector.
La scelta della Intel RealSense d455 è stata motivata da diversi fattori: la presenza del sensore RGB-D ha permesso l'acquisizione sia di informazioni visive che metriche, la tecnologia stero attiva ha consentito una maggiore robustezza rispetto alle superfici inquadrate e, infine la compattezza ne ha facilitato l'integrazione del polso del robot insieme al tool di erogazione.

3.3. END - EFFECTOR

Il tool di erogazione costituisce l'organo terminale della cella robotizzata ed è stato progettato per depositare in maniera controllata la maionese sul contenuto delle vaschette. Il sistema non ha un'unica funzione, ma integra in un unico assieme il contenimento del materiale, il sistema di attuazione per l'estrusione, la struttura di supporto e il sistema di visione. La soluzione sviluppata può essere quindi considerata come un end-effector meccatronico customizzato per l'applicazione in esame. Esso è installato sulla flangia terminale del robot collaborativo e permette di eseguire sia la fase di acquisizione delle informazioni geometriche tramite la camera RGB-D, sia la fase di deposizione del prodotto. Il tool è costituito da una struttura modulare in quanto è realizzata con profilati in alluminio, sui quali sono fissati i diversi componenti realizzati tramite stampa 3D. Tali componenti svolgono la funzione di collegamento, centraggio e supporto tra siringa, asse lineare, camera e flangia del robot.

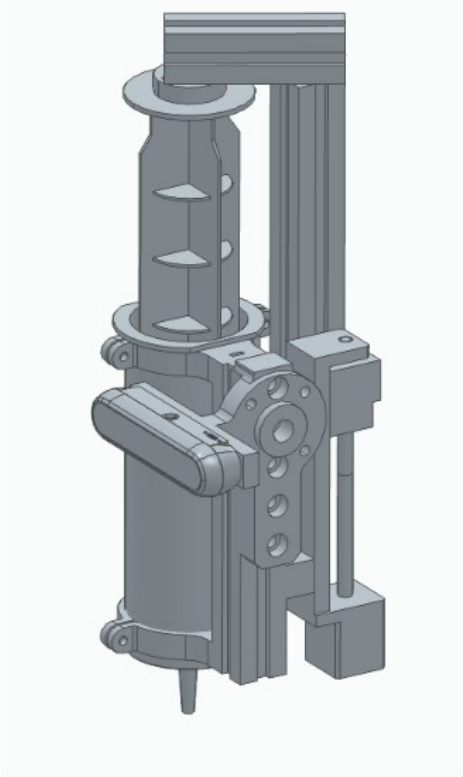


Figura 3.10 : Modello CAD dell'end-effector

L'architettura del tool può essere suddivisa nelle seguenti parti:

- struttura portante;
- sistema di dispensazione;
- asse lineare di attuazione;
- sistema elettronico di comando;
- sistema di visione integrato.

Questa suddivisione consente di descrivere il tool non come un semplice assemblaggio di componenti, ma come un sistema progettato per garantire il corretto accoppiamento tra movimentazione robotica, visione artificiale ed erogazione del prodotto.

La progettazione è stata guidata da una serie di requisiti funzionali come la possibilità di contenere un volume sufficiente di prodotto per eseguire più cicli consecutivi senza troppe operazioni di ricarica frequenti. Per questo scopo è stata scelta una siringa con capacità 550 cm^3 che è risultata adeguata per i test e compatibile con gli ingombri. Un altro requisito è legato al controllo della quantità del materiale erogato che deve essere il più possibile regolabile e ripetibile. La scelta è ricaduta su un sistema di attuazione lineare elettrico, comandato da un motore passo-passo, in grado di trasformare un comando elettrico in uno

spostamento controllato dello stantuffo. Dal punto di vista meccanico, gli ingombri devono essere compatibili con il campo visivo della camera che deve essere adeguato durante l'acquisizione e non avere interferenze con la siringa, con l'ugello o con il movimento dell'asse lineare. Inoltre, la massa deve essere limitata a quello che è il carico utile del robot UR5e, perché influenza sia le prestazioni dinamiche che la precisione del moto durante le traiettorie. Per questo è stata adottata una soluzione modulare basata su profilati leggeri e componenti stampati 3D.

Infine, il sistema deve consentire una certa facilità di montaggio, smontaggio e manutenzione, considerando che il contatto con prodotti alimentari viscosi richiede operazioni periodiche di pulizia.

3.3.1. STRUTTURA PORTANTE E COMPONENTI STAMPATI 3D

La struttura portante del tool è realizzata mediante profilati modulari in alluminio. Questa scelta consente di ottenere un buon compromesso tra rigidezza, leggerezza e semplicità di assemblaggio. L'elemento strutturale principale è costituito da profilati Bosch Rexroth a sezione quadrata 30 × 30 mm, dotati di scanalature longitudinali per il fissaggio di staffe e supporti come è possibile vedere dalla sezione in *Figura 3.11*.

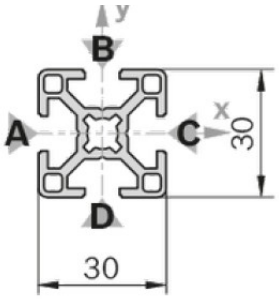


Figura 3.11: Sezione del profilato Bosch 30x30 mm

Il profilato utilizzato appartiene alla famiglia dei profilati modulari con scanalatura 8, caratteristica che consente il montaggio di accessori e componenti mediante dadi scorrevoli inseriti direttamente nelle cave laterali. Questa soluzione permette di variare la posizione dei diversi elementi lungo il telaio senza dover realizzare nuove forature o lavorazioni meccaniche sui profilati. In particolare, il profilato verticale principale, tagliato per una lunghezza di 185 mm, svolge la funzione di elemento portante, poiché sostiene l'asse lineare, i supporti della siringa e l'interfaccia di collegamento con la

flangia robotica. Il tool comprende anche un telaio secondario che garantisce l'accoppiamento tra l'attuatore lineare e il sistema di erogazione attraverso due profilati con le stesse specifiche descritte in precedenza, posizionati ortogonalmente tra loro: il profilo più lungo, di 190 mm, è parallelo all'asse della siringa e al profilato portante, mentre il profilo trasversale, di 52 mm, ne completa il collegamento. Questa struttura svolge principalmente una funzione di supporto e vincolo geometrico, mantenendo allineati l'asse di avanzamento dell'attuatore e l'asse dello stantuffo della siringa. Il collegamento meccanico tra la slitta dell'asse lineare e il pistone è realizzato mediante una piastra metallica avvitata con tre viti M4 sulla slitta e un accoppiamento vite M6-dado sul telaio a L. Tale piastra permette di trasferire il moto lineare generato dall'attuatore alla parte mobile del sistema di erogazione e, allo stesso tempo, consente di compensare la distanza laterale tra l'asse della vite dell'attuatore e l'asse dello stantuffo della siringa. In questo modo, l'avanzamento della slitta viene trasformato in una spinta assiale sul pistone, evitando che sullo stantuffo si generino componenti di carico trasversali significative. La struttura a L contribuisce quindi a irrigidire il collegamento e a mantenere stabile la geometria del sistema durante l'erogazione del materiale.

Dal punto di vista costruttivo, la sezione 30 × 30 mm rappresenta un compromesso adeguato tra rigidezza e massa. Una sezione più piccola avrebbe ridotto ulteriormente il peso dell'end-effector, ma avrebbe offerto minore rigidezza flessionale; al contrario, profilati di dimensioni maggiori avrebbero aumentato massa e ingombri, penalizzando il comportamento dinamico del robot. L'impiego di una struttura in alluminio permette inoltre di mantenere contenuto il peso complessivo del tool, aspetto rilevante poiché l'intero assieme è installato direttamente sulla flangia terminale del robot collaborativo UR5e.

Ai profilati sono collegati diversi componenti realizzati mediante stampa 3D in tecnologia FDM (Fused Deposition Modeling) utilizzando PETG (Polyethylene Terephthalate Glycol) come materiale. L'impiego della stampa 3D si è rivelato particolarmente vantaggioso in quanto consente una prototipazione più rapida e più economica rispetto ai tempi e ai costi tipici delle lavorazioni meccaniche tradizionali. Tali elementi svolgono una funzione fondamentale nell'integrazione dei sottosistemi, poiché permettono di collegare componenti commerciali aventi geometrie e interfacce differenti.

Il primo elemento progettato è il componente che completa il collegamento tra il telaio a L e lo stantuffo della siringa *Figura 3.12*, dotato di un inserto filettato in ottone, non modellato sul file CAD, alloggiato al centro e di quattro fori passanti con svasatura cilindrica disposti perimetralmente. I quattro fori consentono il fissaggio del disco direttamente allo stantuffo della siringa, mediante l'accoppiamento vite M5-dado. Il collegamento al telaio a L avviene invece tramite l'inserto filettato centrale: una vite, inserita in un foro passante ricavato sul telaio a L, si avvita nell'inserto e va in battuta sulla scanalatura del profilato, vincolando così il disco, e di conseguenza lo stantuffo, alla struttura a L. In questo modo, il moto trasmesso dalla slitta dell'asse lineare al telaio a L viene infine trasferito allo stantuffo, completando la catena di trasmissione del moto fino al sistema di erogazione.

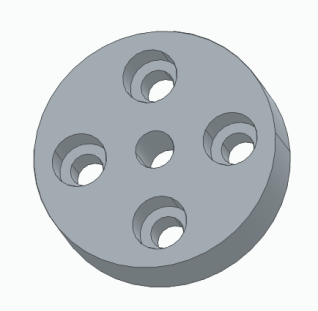


Figura 3.12 : Disco di collegamento telaio a L e stantuffo

Successivamente è stata progettata la flangia di interfaccia che collega l'intero tool al polso del robot UR5e (*Figura 3.13*). Nella porzione inferiore, il componente presenta una serie di fori passanti con svasatura cilindrica allineati che ne consentono il fissaggio diretto al profilato strutturale verticale, mediante l'inserimento di dadi scorrevoli nelle scanalature laterali secondo la stessa logica di montaggio adottata per gli altri elementi del telaio. Questa interfaccia garantisce un collegamento rigido tra la struttura portante e il resto dell'assieme, mantenendo la possibilità di regolare la posizione della flangia lungo il profilato in fase di allineamento del sistema. Risalendo verso il robot, la porzione superiore della flangia si interfaccia direttamente con il polso dell'UR5e attraverso un foro centrale e un pattern di fori passanti coassiali, conforme allo standard ISO 9409-1-50-4-M6 tipico dei cobot collaborativi. Su una porzione a sbalzo adiacente trova inoltre alloggiamento la camera Intel RealSense D455, montata in configurazione eye-in-hand: questa scelta consente di mantenere la camera in prossimità dell'asse di lavoro del robot, aspetto rilevante ai fini della successiva calibrazione hand-eye.

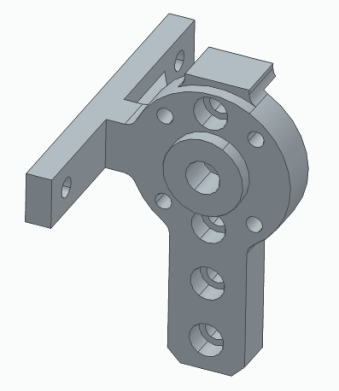


Figura 3.13 : Flangia di interfaccia tool-robot UR5e

Tra i componenti stampati 3D ci sono anche i supporti della siringa (*Figura 3.14*) che presentano una geometria sagomata in funzione del diametro del corpo siringa e hanno il compito di vincolarla alla struttura portante impedendone rotazioni o spostamenti laterali durante la fase di erogazione. Ciascun supporto presenta una geometria a C, dotata di due alette laterali forate che, mediante l'accoppiamento vite-dado, ne consentono la chiusura attorno al cilindro della siringa. Sul lato opposto rispetto all'apertura, è presente un'ulteriore aletta forata che ne consente il fissaggio al profilato strutturale principale, mediante una vite accoppiata a un dado scorrevole inserito nella scanalatura del profilato stesso. Questa soluzione garantisce un bloccaggio stabile e regolabile del componente commerciale, evitando soluzioni di fissaggio permanenti e permettendo, all'occorrenza, la rimozione o la sostituzione della siringa stessa. La presenza di più punti di vincolo lungo l'altezza della siringa consente di mantenere l'asse del serbatoio allineato con la direzione di avanzamento dello stantuffo.

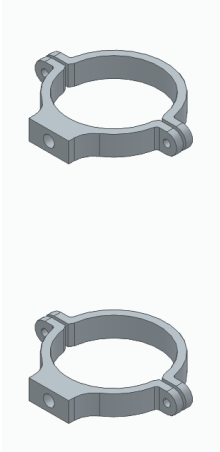


Figura 3.14 : Supporti della siringa

Infine, il collegamento tra l'asse lineare e il profilato strutturale è realizzato mediante due blocchetti di fissaggio *Figura 3.15* , posizionati in modo da avere interasse dei fori a 100 mm lungo lo sviluppo dell'attuatore. Ciascun blocchetto presenta un foro centrale, con sede cilindrica per l'alloggiamento incassato della testa della vite, che ne consente il fissaggio al profilato strutturale mediante l'accoppiamento vite-dado nella relativa scanalatura; lateralmente, due fori permettono invece il collegamento diretto alla base dell'asse lineare, in corrispondenza dei fori di passaggio presenti su quest'ultima.

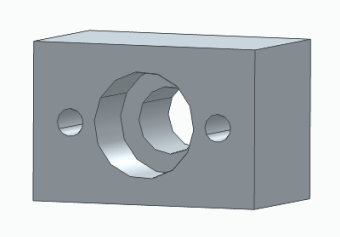


Figura 3.15: Blocchetti di fissaggio tra asse lineare e profilato

Nel complesso questa struttura, che comprende profilati in alluminio e parti stampate 3D, funge da supporto meccanico al sistema di dispensazione, il cui elemento centrale, la siringa custom, viene descritto nel paragrafo seguente.

3.3.2. SISTEMA DI DISPENSAZIONE

Il sistema di dispensazione è basato su una siringa cilindrica in materiale plastico con capacità nominale pari a 550 ml, utilizzata come serbatoio del prodotto alimentare. La scelta di una siringa di grande capacità consente di disporre di un volume sufficiente per eseguire più cicli di decorazione consecutivi, riducendo le interruzioni dovute alle operazioni di riempimento. Inoltre, il corpo trasparente permette di verificare visivamente il livello residuo del prodotto e l'eventuale presenza di bolle d'aria, che potrebbero generare discontinuità durante il deposito (*Figura 3.16*).

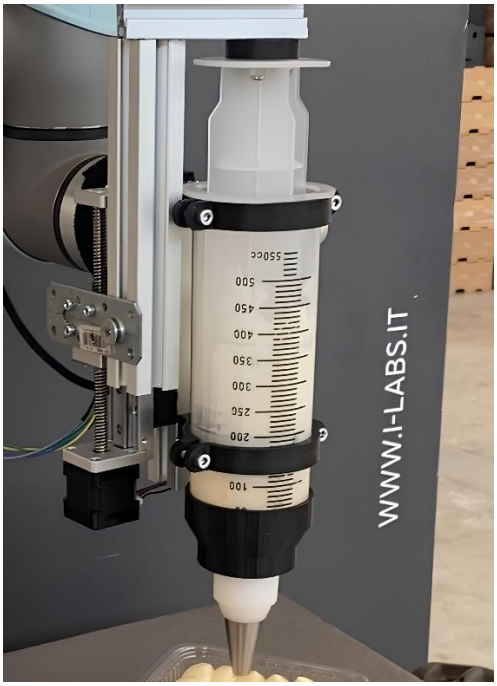


Figura 3.16: Sistema di erogazione completo

La siringa commerciale non è stata tuttavia utilizzata nella sua configurazione originaria. In particolare, la parte terminale è stata modificata mediante taglio, poiché l'estrusore fornito con il componente non risultava adatto all'applicazione prevista dal progetto. L'uscita originale della siringa era infatti pensata per un impiego generico e non permetteva un deposito di materiale adatto a replicare i ciuffi di maionese realizzati manualmente con sac à poche. Poiché uno degli obiettivi del progetto era automatizzare il processo mantenendo una forma uguale a quella ottenuta dall'operatore, si è reso necessario adottare punte commerciali per sac à poche, già progettate per generare forme decorative attraverso specifiche geometrie di uscita. Per consentire l'accoppiamento tra la siringa modificata e tali punte commerciali è stata quindi progettata un'interfaccia di adattamento dedicata. Il componente è stato modellato in ambiente CAD (*Figura 3.17a*) e successivamente realizzato mediante stampa 3D (*Figura 3.17 b-c*). L'interfaccia svolge una duplice funzione: da un lato permette il collegamento meccanico tra il corpo della siringa tagliata e la punta da sac à poche, dall'altro garantisce la continuità del condotto di uscita del prodotto, evitando disallineamenti o variazioni brusche di sezione che potrebbero alterare il flusso della maionese. La scelta di utilizzare punte da sac à poche consente inoltre di rendere il sistema più riconfigurabile, in quanto le punte sono montate mediante un attacco Wilson (*Figura 3.17d*), la cui parte interna risulta incastrata

all'interfaccia tramite un anello dedicato (*Figura 3.17c*), mentre la parte esterna dell'attacco, avvitabile e svitabile, ne consente la rapida sostituzione. Variando la tipologia di punta (*Figura 3.17e*) è infatti possibile modificare la geometria del deposito, adattando il tool a diverse decorazioni senza dover riprogettare l'intero sistema di dispensazione. In questo modo, l'interfaccia custom rappresenta un elemento chiave del progetto, poiché permette di integrare un componente commerciale generico, la siringa da 550 ml, con un accessorio specifico per la decorazione alimentare, rendendo il sistema più coerente con l'obiettivo di replicare automaticamente il gesto manuale dell'operatore.

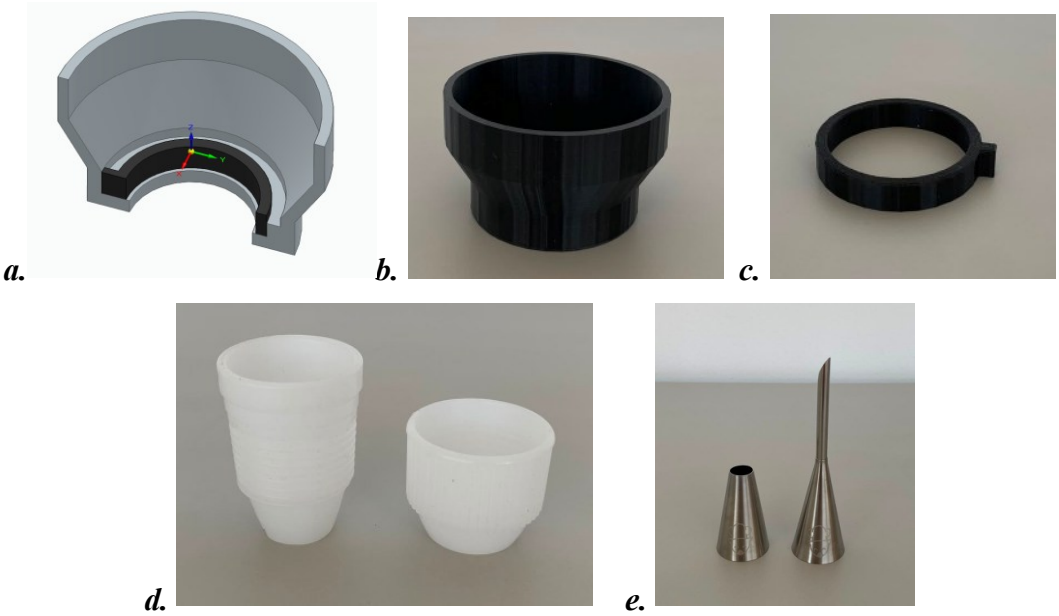


Figura 3.17: Componenti del sistema di accoppiamento: ***a.*** *sezione CAD dell'interfaccia;* ***b.*** *interfaccia reale stampata 3D;* ***c.*** *anello di incastro stampato 3D;* ***d.*** *attacco Wilson (parte interna e esterna);* ***e.*** *punte intercambiali in acciaio per sac à poche.*

Un'altra modifica significativa ha riguardato anche lo stantuffo interno della siringa. Nella configurazione originale, lo stantuffo è dotato di una guarnizione in gomma che realizza una tenuta quasi completa contro la parete interna del cilindro. Questa soluzione, tipica delle siringhe, è adatta quando si desidera aspirare o spingere un fluido

mantenendo il volume interno isolato dall'esterno. Tuttavia, nel caso in esame, tale configurazione non risultava ottimale, poiché la tenuta generata dalla guarnizione tendeva a intrappolare aria all'interno della siringa e a creare un effetto di depressione o sovrapressione indesiderata durante il movimento dello stantuffo.

Per questo motivo la guarnizione originale è stata rimossa e sostituita con un disco stampato in 3D, mostrato in *Figura 3.18*. Il componente è stato progettato per mantenere il contatto funzionale con il prodotto senza realizzare una tenuta ermetica completa: il disco presenta infatti un leggero gioco rispetto alla parete interna della siringa, in modo da consentire la fuoriuscita dell'aria residua durante l'avanzamento dello stantuffo. Questo aspetto risulta particolarmente rilevante con prodotti viscosi come la maionese, dove l'aria eventualmente intrappolata tenderebbe a comprimersi e successivamente a espandersi in modo non controllato durante la corsa del pistone, generando ritardi nell'inizio dell'erogazione o variazioni localizzate di portata che comprometterebbero la qualità visiva dei ciuffi depositati. L'adozione di uno stantuffo modificato con gioco controllato consente quindi di favorire lo sfogo dell'aria residua e di migliorare la continuità del flusso in uscita.

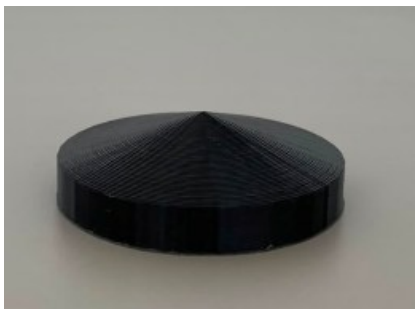


Figura 3.18 : inserto sostitutivo della guarnizione originale dello stantuffo

Il principio di funzionamento del sistema rimane comunque riconducibile a un meccanismo volumetrico. L'avanzamento dello stantuffo riduce il volume disponibile all'interno della siringa e genera la fuoriuscita del prodotto attraverso l'ugello. In prima approssimazione, trascurando la comprimibilità del materiale, le deformazioni elastiche del sistema, i giochi e le perdite, il volume teoricamente erogato può essere espresso come:

$$V = A_s \cdot \Delta s$$

dove Vè il volume erogato, A_sè l'area interna della siringa e Δs è lo spostamento dello stantuffo.

Analogamente, la portata volumetrica può essere espressa come:

$$Q = A_s \cdot v_s$$

dove Qè la portata volumetrica e v_sè la velocità di avanzamento dello stantuffo.

Tali relazioni evidenziano il legame diretto tra il movimento imposto dall'asse lineare e la quantità di prodotto depositata e in particolare, il controllo della velocità di avanzamento dello stantuffo permette di regolare la portata in uscita.

L'area interna della siringa può essere calcolata a partire dal diametro interno D_s mediante la relazione:

$$A_s = \frac{\pi D_s^2}{4}$$

e, nota la corsa utile dell'asse lineare pari a 100 mm, il volume teoricamente erogabile nell'intera corsa può essere stimato come:

$$V_{100} = A_s \cdot 100$$

dove V_{100} rappresenta il volume teorico associato a un avanzamento dello stantuffo pari a 100 mm.

Per applicare le relazioni al caso specifico, è stato misurato il diametro interno della siringa, pari a: $D_s = 66$ mm, in questo modo l'area della sezione interna è circa 3421,19 mm^2. Considerando la corsa utile dell'asse lineare pari a 100 mm, il volume teoricamente erogabile durante un avanzamento completo dello stantuffo è: V_{100} = 342119 mm^3. Dal momento che 1 ml = 1000 mm^3, il V_{100} è pari a circa 342 ml rispetto ad un volume nominale di 550 ml.

È importante osservare che tali relazioni descrivono un comportamento ideale. Nel sistema reale, la quantità effettivamente erogata può differire dal valore teorico a causa di diversi fattori, tra cui la viscosità del prodotto, la presenza di aria residua, il gioco introdotto dallo stantuffo modificato, le perdite laterali, la geometria dell'ugello e il comportamento reologico del materiale. Queste formule risultano comunque un

riferimento utile in fase di progetto, per dimensionare corsa e velocità dell'asse lineare in funzione del volume di prodotto desiderato.

Nel complesso il sistema di dispensazione risulta semplice, economico e facilmente riconfigurabile, caratteristiche particolarmente adatte alla fase prototipale del progetto.

3.3.3. SISTEMA DI ATTUAZIONE E COMANDO DELL'ASSE LINEARE

Il sistema di attuazione dell'asse lineare è incaricato di generare il moto di avanzamento necessario al funzionamento del tool di deposizione. Oltre alla parte meccanica, è presente anche una parte elettronica composta da un motore passo-passo, da un driver di potenza, da una scheda Arduino, da un alimentatore esterno e da una scheda di interfaccia dei segnali provenienti dal robot. A differenza dei componenti strutturali descritti nei paragrafi precedenti, che hanno principalmente funzione di supporto e collegamento, l'asse lineare costituisce l'elemento attivo del sistema, poiché trasforma un comando elettrico in uno spostamento controllato lungo una direzione prestabilita. Rispetto a una soluzione pneumatica, l'attuazione elettromeccanica consente di controllare direttamente posizione e velocità dell'asse in modo preciso, evitando circuiti ausiliari in pressione e semplificando la gestione del prototipo.

Prima di descrivere i singoli componenti è utile inquadrare l'architettura complessiva, interpretabile come una catena di conversione del segnale e della potenza: il robot invia segnali digitali a 24 V alla scheda optoisolata, che li adatta ai livelli logici dell'Arduino (3,3 V); l'Arduino genera i segnali di comando STEP, DIR ed ENA per il driver DM320T; il driver, alimentato a 24 V, pilota il motore passo-passo; il motore mette in rotazione la vite dell'asse lineare, che converte la rotazione in traslazione della slitta.

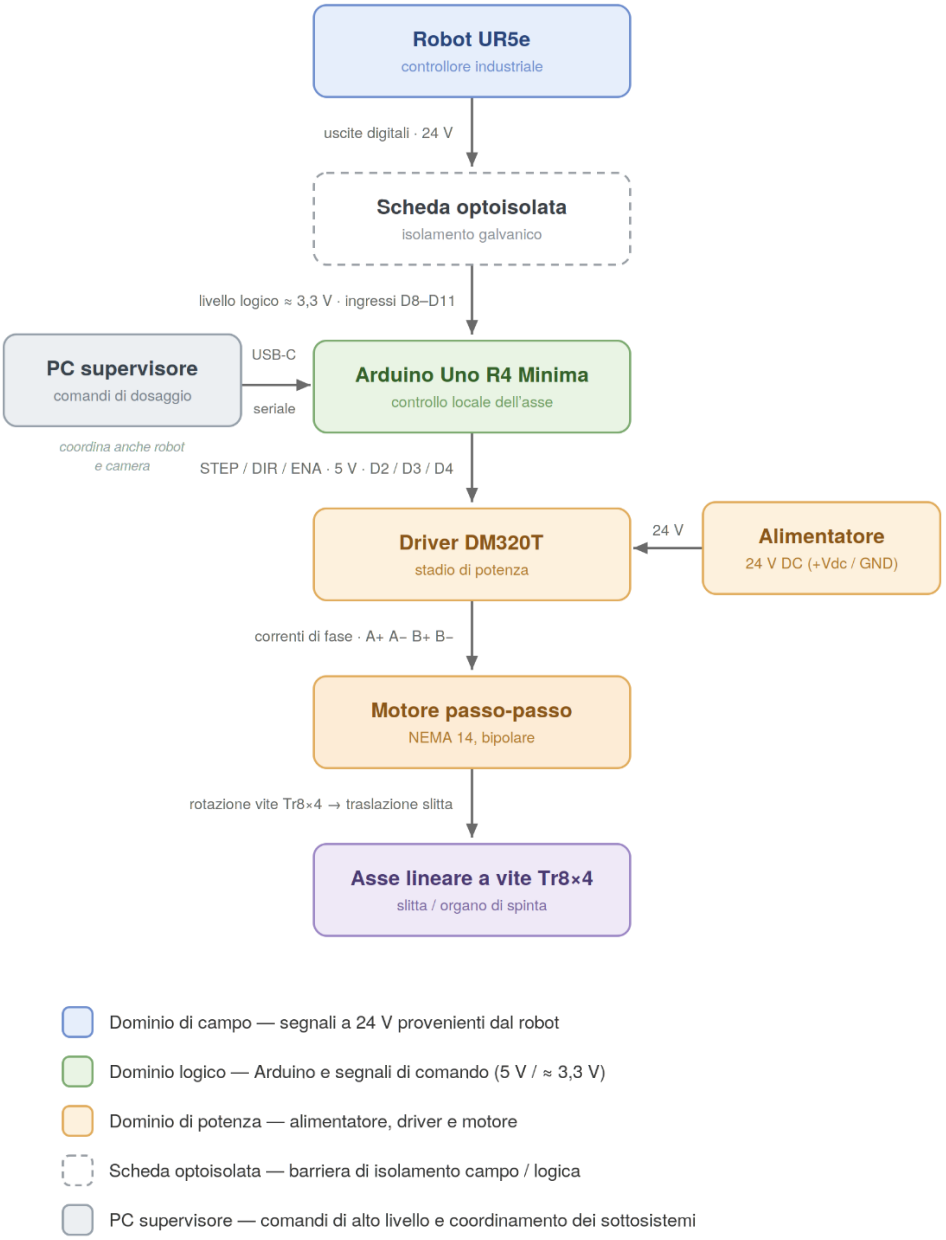


Figura 3.19 : Schema a blocchi dell'architettura hardware del sistema di attuazione

L'architettura distingue chiaramente i tre domini elettrici del sistema: il dominio di campo, costituito dai segnali a 24 V provenienti dal robot; il dominio logico, rappresentato dalla scheda Arduino e dai segnali di comando verso il driver; il dominio di potenza, comprendente alimentatore, driver e motore. Tale separazione rende il sistema più leggibile e più sicuro, riduce il rischio di sovraccaricare la scheda di controllo e semplifica il cablaggio in fase prototipale. Come illustrato nella *Figura 3.19*, l'Arduino

riceve i comandi da due sorgenti distinte: il robot, i cui segnali digitali raggiungono gli ingressi D8, D9 e D11 attraverso la scheda optoisolata, e il PC, connesso via USB-C, che invia i comandi seriali di dosaggio e provvede contestualmente all'alimentazione della logica. Il PC svolge in realtà il ruolo di supervisore dell'intera cella, coordinando anche i movimenti del robot e l'acquisizione delle immagini; rispetto all'asse lineare esso rappresenta quindi la seconda sorgente di comando, la cui logica di gestione e di priorità rispetto ai segnali del robot è approfondita nel seguito della trattazione.

L'asse lineare che è stato scelto è un attuatore a vite con corsa utile pari a 100 mm. Il moto rotatorio del motore passo-passo è convertito in moto traslatorio mediante una vite Tr8×4 (*Figura 3.20*). La sigla indica una vite trapezoidale con diametro nominale di 8 mm e passo di avanzamento pari a 4 mm: a ogni rotazione completa della vite la slitta avanza teoricamente di 4 mm lungo la direzione dell'asse. La scelta di una trasmissione a vite trapezoidale è coerente con le esigenze del sistema, in quanto rappresenta una soluzione semplice e compatta, adatta a un sistema sperimentale, oltre a un effetto autobloccante che mantiene la posizione a motore non alimentato. Le grandezze di interesse sono la corsa utile, la risoluzione teorica, la rigidezza della guida, la capacità di carico assiale e trasversale e la ripetibilità del posizionamento.

Figura 3.20 : Asse lineare elettromeccanico impiegato per l'attuazione del tool

Nel tool realizzato l'asse lineare è montato verticalmente a lato del corpo della siringa. L'attuatore non è coassiale con lo stantuffo: i due elementi seguono direzioni verticali distinte e parallele, separate lateralmente, e il moto della slitta è trasferito allo stantuffo

attraverso un collegamento meccanico laterale. Tale configurazione non costituisce un errore di allineamento, ma una scelta progettuale dettata dai vincoli di ingombro dell'end-effector: disporre l'attuatore a lato consente di contenere le dimensioni del tool, lasciando spazio al corpo della siringa, ai supporti, all'ugello e all'interfaccia con la flangia robotica.

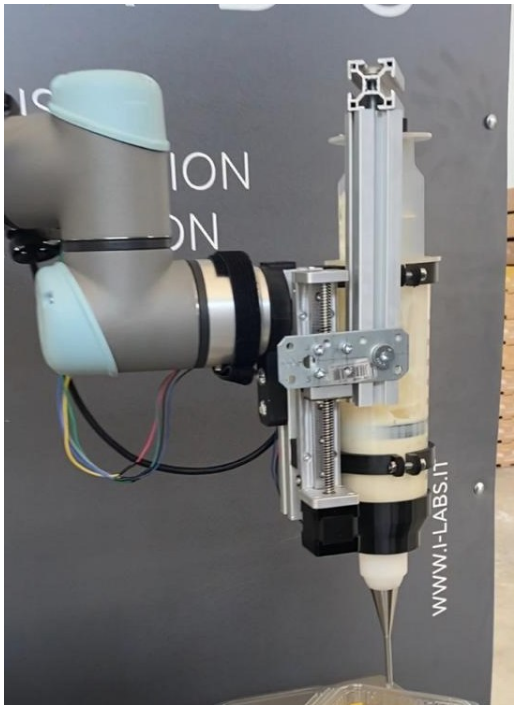


Figura 3.21 : Configurazione reale dell'asse lineare integrato nel tool

Dal punto di vista delle sollecitazioni, l'asse non lavora quindi in condizione di carico puramente assiale. La forza utile richiesta dal processo agisce in direzione verticale, ma viene trasmessa a una certa distanza laterale dalla guida dell'attuatore; tale distanza genera un carico eccentrico, con conseguente momento flettente sulla slitta e sugli elementi di collegamento. Il momento può essere espresso in forma semplificata come:

$$M = F_a \cdot e$$

dove M è il momento flettente generato dalla configurazione non coassiale, F_a è la forza necessaria all'avanzamento dell'organo di spinta ed e è la distanza laterale tra l'asse di traslazione dell'attuatore e la linea di azione della forza trasmessa.

Questo momento non rappresenta necessariamente una criticità, ma un aspetto da considerare nella caratterizzazione del sistema. Nell'asse lineare la vite ha la funzione di convertire il moto rotatorio in moto traslatorio, mentre guida e slitta vincolano il moto e contrastano le reazioni laterali; nel prototipo le sollecitazioni dovute all'eccentricità sono pertanto assorbite dalla guida, dai supporti di collegamento e dalla struttura portante del tool. Si possono distinguere tre contributi: il carico assiale F_a, associato alla forza utile di avanzamento; il carico radiale F_r, dovuto alle reazioni laterali della trasmissione non coassiale; il momento flettente M, prodotto dal braccio laterale tra la linea di azione della forza e la guida.

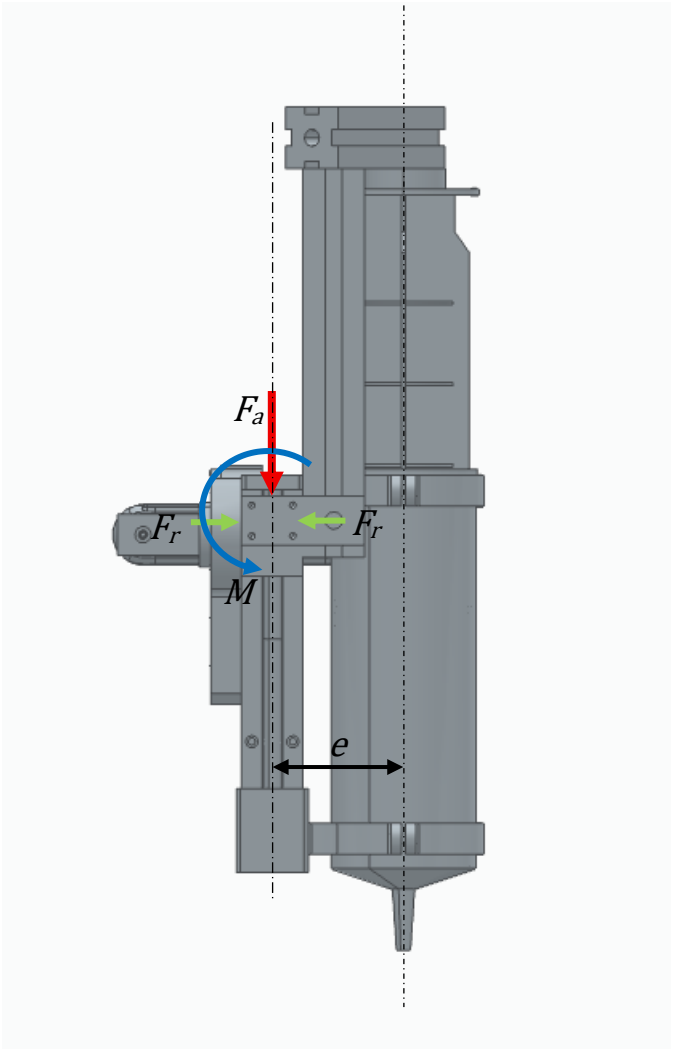


Figura 3.22 : Schema semplificato delle sollecitazioni

La soluzione adottata costituisce dunque un compromesso tra compattezza, semplicità costruttiva e rigidezza: una disposizione coassiale avrebbe ridotto i momenti sulla slitta, ma a costo di un maggiore ingombro del tool, mentre la configurazione affiancata privilegia l'integrazione sull'end-effector accettando carichi eccentrici compatibili con una prima implementazione sperimentale. Come precedentemente descritto, il legame tra rotazione del motore e avanzamento lineare è definito dal passo della vite e dal numero di impulsi necessari a una rotazione completa. Indicando con p il passo della vite e con Nrev il numero di impulsi per giro, lo spostamento associato a un singolo impulso è:

$$\Delta s = p / Nrev$$

Il motore passo-passo presenta 200 passi interi per giro, corrispondenti a un angolo di passo di 1,8°; il driver è configurato con un microstepping pari a 1/2, da cui Nrev = 400 impulsi/giro. Con p = 4 mm/giro si ottiene:

$$\Delta s = 4 / 400 = 0{,}01 \text{ mm/impulso}$$

a cui corrisponde una risoluzione teorica di 100 impulsi/mm. Tale valore rappresenta il rapporto di conversione tra comando elettrico e spostamento fisico della slitta. La risoluzione teorica non coincide necessariamente con l'accuratezza reale, che dipende anche da giochi, attriti, deformazioni dei supporti, eventuali perdite di passi e tolleranze di montaggio; per il prototipo sviluppato essa è comunque adeguata a un controllo fine dell'avanzamento.

L'azionamento dell'asse è affidato a un motore passo-passo bipolare in formato NEMA 14, scelto per il compromesso favorevole tra ingombro, coppia disponibile e facilità di pilotaggio in catena aperta. Questo tipo di motore è adatto ad applicazioni in cui si richiede uno spostamento incrementale controllato senza ricorrere, almeno in prima configurazione, a una retroazione di posizione: ogni impulso ricevuto dal driver produce un avanzamento elementare del rotore e, tramite la vite, uno spostamento lineare della slitta. Costruttivamente il motore è composto da uno statore con avvolgimenti e da un rotore che si orienta in funzione della sequenza di eccitazione delle fasi. Il collegamento al driver avviene mediante quattro conduttori, corrispondenti alle due fasi e identificati

come A+, A−, B+, B−. Il motore non è alimentato dalla scheda Arduino, bensì dal driver di potenza che genera le correnti di fase. Funzionalmente il motore deve erogare la coppia necessaria a vincere gli attriti della vite, il peso del gruppo mobile in risalita, la resistenza del sistema di spinta e gli effetti della configurazione non coassiale; le velocità moderate e la corsa limitata riducono le sollecitazioni dinamiche e il rischio di perdita di passi.

Lo stadio di potenza è costituito dal driver digitale STEPPERONLINE DM320T, che riceve i segnali logici a bassa tensione dalla scheda Arduino e li traduce nelle correnti necessarie ad alimentare il motore, separando la parte di controllo dalla parte di potenza. Il driver è alimentato da un alimentatore esterno a 24 V in corrente continua. Sulla morsettiera di potenza sono presenti i morsetti di alimentazione +Vdc e GND e le quattro uscite verso le fasi del motore A+, A−, B+, B−, attraverso cui il driver controlla la corrente negli avvolgimenti e ne determina la rotazione.

Sul lato dei comandi il DM320T riceve tre ingressi logici: il segnale PUL (o STEP), costituito da un treno di impulsi in cui ogni fronte determina l'avanzamento del motore di un micro-passo; il segnale DIR, il cui livello logico stabilisce il verso di rotazione e quindi di traslazione dell'asse; il segnale ENA, che abilita o disabilita lo stadio di potenza. La corrente di fase e il livello di microstepping si impostano tramite i DIP-switch presenti sul corpo del dispositivo. Gli ingressi PUL, DIR ed ENA sono internamente optoisolati, condizione che ne consente il pilotaggio diretto a livello logico.

Figura 3.23 : Box con a sinistra scheda Arduino Uno R4 Minima e a destra driver DM320T

Nella configurazione realizzata il driver è alloggiato nello stesso box della scheda Arduino, il cui collegamento riguarda i segnali STEP, DIR ed ENA, mentre i collegamenti di potenza riguardano l'alimentazione a 24 V e le quattro uscite verso il motore.

La scheda Arduino Uno R4 Minima, basata sul microcontrollore Renesas RA4M1, svolge il ruolo di unità di controllo locale dell'asse e di interfaccia tra i segnali esterni e il driver. Tre uscite digitali sono dedicate al pilotaggio del DM320T — mappate sui pin D2 (STEP→PUL), D3 (DIR) e D4 (ENA), con livelli logici a 5 V — mentre alcuni ingressi digitali (pin D8, D9, D11) ricevono i segnali provenienti dal robot attraverso la scheda optoisolata. Arduino non eroga potenza all'attuatore: la sua funzione è generare i segnali logici di comando per il driver, mantenendo locale la gestione del moto mentre il robot si limita a richiedere i movimenti tramite segnali digitali. La scheda Arduino è alimentata e connessa al PC tramite cavo USB-C, collegamento che assicura l'alimentazione della logica e la comunicazione con il computer durante l'esecuzione della prova, mentre la potenza del motore resta separata sull'alimentatore a 24 V.

Come già anticipato, l'alimentatore esterno fornisce l'energia necessaria al driver e al motore. È impiegata un'alimentazione in corrente continua a 24 V con erogazione fino a 20 A, valore tipico dei sistemi di automazione e compatibile con il driver. L'alimentatore è connesso ai morsetti di ingresso del driver +Vdc e GND, da cui il DM320T preleva l'energia per generare le correnti di fase.

L'interfacciamento tra robot e Arduino è realizzato mediante una scheda su millefori contenente optoisolatori e resistenze. La sua funzione non è comandare il motore né sostituire il driver, ma adattare i segnali digitali del robot ai livelli compatibili con Arduino. Le uscite digitali del robot UR lavorano a 24 V, tensione che non può essere collegata direttamente agli ingressi della scheda Arduino, che operano a 5 V, pena il danneggiamento della scheda. La scheda optoisolata separa elettricamente il dominio del robot da quello logico dell'Arduino: il segnale del robot attiva il lato di ingresso dell'optoisolatore, mentre sul lato di uscita viene generato un segnale a livello logico (circa 3,3 V) compatibile con l'elettronica dell'Arduino.

Il principio di funzionamento si basa su un accoppiamento ottico interno. Il lato di ingresso contiene un LED, quello di uscita un elemento fotosensibile, tipicamente un fototransistor: quando il LED è alimentato dal segnale del robot, il fototransistor commuta e genera il corrispondente segnale logico verso Arduino, senza continuità

elettrica diretta tra i due circuiti. Le resistenze montate sulla scheda limitano la corrente nei LED interni, proteggendo i componenti.

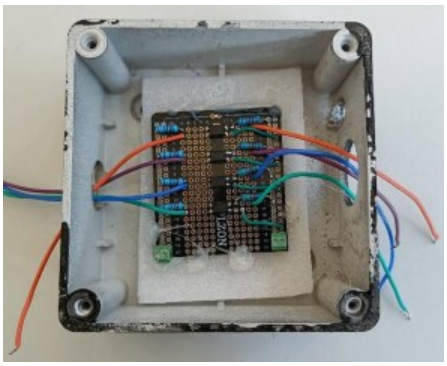


Figura 3.24: Scheda millefori per l'adattamento dei segnali proveniente dal robot UR

Concludendo, si può affermare che il modulo di attuazione così diviso, ovvero con un dominio di potenza separato da quello di logica, risulta compatto, integrabile e adatto ad una prima implementazione sperimentale del processo robotizzato.

4. SETUP SPERIMENTALE E CALIBRAZIONE HAND-EYE DEL SISTEMA ROBOT-CAMERA

La corretta esecuzione delle traiettorie generate dal sistema di visione richiede che i punti individuati dalla camera siano espressi in un sistema di riferimento compatibile con quello del robot. La camera Intel RealSense D455, infatti, acquisisce le informazioni geometriche della scena nel proprio sistema di riferimento, mentre il robot UR5e esegue i movimenti rispetto al sistema di riferimento della base. È quindi necessario determinare una catena di trasformazioni geometriche che consenta di convertire un punto osservato dalla camera in una posizione raggiungibile dal TCP operativo del robot.

Come già descritto in precedenza, la calibrazione è stata svolta considerando la camera montata direttamente sull'end-effector del robot, in configurazione eye-in-hand. In questa configurazione il sensore di visione si muove solidalmente al polso robotico e mantiene una posizione relativa costante rispetto al tool di erogazione. Il TCP attivo del robot è stato definito in corrispondenza della punta della siringa, cioè nel punto effettivo di deposizione del materiale alimentare. Di conseguenza, la trasformazione stimata dalla procedura di calibrazione non è riferita genericamente alla flangia meccanica del robot, ma al sistema di riferimento operativo del tool, coincidente con la punta di erogazione.

La stima della matrice di trasformazione non dipende esclusivamente dall'algoritmo numerico utilizzato, ma anche dalla configurazione fisica del sistema ovvero dalla posizione della camera rispetto al tool, dalla definizione del TCP, dal posizionamento della scacchiera e dalla scelta delle pose robotiche. Per questo motivo, prima di descrivere la calibrazione vera e propria, viene presentato il setup sperimentale utilizzato durante l'acquisizione dei dati.

4.1. CONFIGURAZIONE SPERIMENTALE E ACQUISIZIONE DEI DATI

Il setup sperimentale utilizzato per la calibrazione è costituito dal robot collaborativo UR5e, dal tool di erogazione montato sulla flangia terminale, dalla camera Intel RealSense D455 e da una scacchiera di calibrazione posizionata sul piano di lavoro. La camera è fissata

rigidamente alla struttura del tool, in prossimità della siringa, in modo da osservare l'area operativa in cui vengono collocati gli oggetti da decorare. Il robot UR5e è installato su una base rigida e opera sopra un piano di lavoro orizzontale. La camera D455 è solidale al medesimo end-effector e si muove insieme al tool durante tutte le fasi di acquisizione. Tale configurazione consente di mantenere costante la trasformazione geometrica tra camera e TCP, condizione fondamentale per l'applicazione della calibrazione hand-eye. Durante la calibrazione, la scacchiera è stata collocata sul piano di lavoro in posizione fissa. Il robot è stato movimentato in diverse pose, variando sia la posizione sia l'orientamento del tool rispetto alla scacchiera. Per ciascuna posa sono stati acquisiti due insiemi di dati: da un lato l'immagine RGB della scacchiera fornita dalla camera, dall'altro la posa effettiva del TCP restituita dal controllore del robot. In questo modo è stato possibile associare, per ogni configurazione, la posa del target rispetto alla camera e la posa del TCP rispetto alla base robot.

Figura 4.1: Setup sperimentale con scacchiera di calibrazione posizionata sul piano di lavoro

Nel sistema sviluppato, il TCP operativo è stato definito in corrispondenza della punta della siringa, ossia nel punto in cui il materiale viene effettivamente depositato sulla superficie del prodotto. Questa scelta è fondamentale perché l'obiettivo finale del sistema non è semplicemente localizzare la camera rispetto alla flangia del robot, ma trasformare i punti riconosciuti dal sistema di visione in posizioni raggiungibili dalla punta di erogazione. Dal punto di vista cinematico, è importante distinguere tra flangia robotica, end-effector fisico e TCP. La flangia rappresenta l'interfaccia meccanica terminale del robot; l'end-effector comprende tutti i componenti montati sulla flangia, cioè supporti, camera, siringa e asse lineare; il TCP, invece, è il punto operativo utilizzato dal controllore per rappresentare la posa dell'utensile nello spazio. Nel caso in esame, il TCP non coincide con il centro della flangia, ma è traslato fino alla punta della siringa. La posa del TCP rispetto alla base robot viene indicata come:

$$^{B}T_{TCP}$$

dove B rappresenta il sistema di riferimento della base del robot e TCP il sistema di riferimento associato alla punta della siringa. Tale trasformazione viene fornita direttamente dal controllore del robot, una volta impostato il TCP attivo nel Teach Pendant. Nella schermata del Teach Pendant è possibile verificare i parametri del TCP attivo associato al tool di erogazione. In particolare, il TCP risulta definito mediante una traslazione lungo Y di -263 mm e lungo Z di 94 mm rispetto al sistema di riferimento della flangia robotica e una rotazione di $\frac{\pi}{2}$ attorno l'asse X sempre rispetto al SR base della flangia. Questa definizione consente al robot di comandare i movimenti considerando come punto di riferimento non la flangia, ma la punta effettiva dell'utensile.

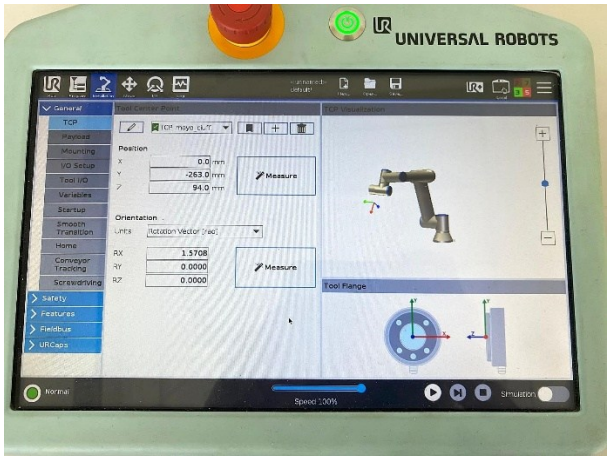


Figura 4.2: Definizione del TCP operativo sul Teach Pendant

Poiché durante l'intera procedura di calibrazione è stato mantenuto attivo il TCP corrispondente alla punta della siringa, la trasformazione stimata dalla calibrazione hand-eye può essere interpretata come la trasformazione tra camera e TCP operativo. Nel seguito, tale trasformazione viene indicata come:

$$^{TCP}T_C$$

dove C rappresenta il sistema di riferimento della camera.

La procedura sperimentale prevede l'acquisizione di una serie di immagini della scacchiera da diverse pose del robot. Per ogni configurazione del manipolatore, il sistema esegue le seguenti operazioni:

1. movimentazione del robot verso una posa predefinita;
2. acquisizione dell'immagine RGB tramite la camera D455;
3. rilevamento automatico degli angoli interni della scacchiera;
4. stima della posa della scacchiera rispetto alla camera;
5. acquisizione della posa del TCP rispetto alla base robot.

La scacchiera utilizzata per la calibrazione è stata modellata nel codice come una griglia di punti tridimensionali appartenenti a un piano. Il sistema di riferimento della scacchiera viene indicato con S. I punti della scacchiera hanno coordinata $Z = 0$, mentre le coordinate X e Y sono definite in funzione della dimensione dei quadrati. Nel caso considerato, il target è stato impostato con una configurazione di 6 x 9 punti interni e lato del quadrato pari a:

$$l = 0{,}035\ m$$

L'acquisizione delle immagini è stata effettuata utilizzando lo stream RGB della camera Intel RealSense D455, con risoluzione pari a:

$$640\ x\ 480\ px$$

e frequenza di acquisizione pari a 30 fps. Per ciascuna posa è stata salvata l'immagine grezza (*Figura 4.3a*) e, successivamente, l'immagine con la scacchiera rilevata (*Figura 4.3b*). Questo

passaggio è utile sia per documentare la procedura sperimentale sia per verificare visivamente la correttezza del rilevamento.

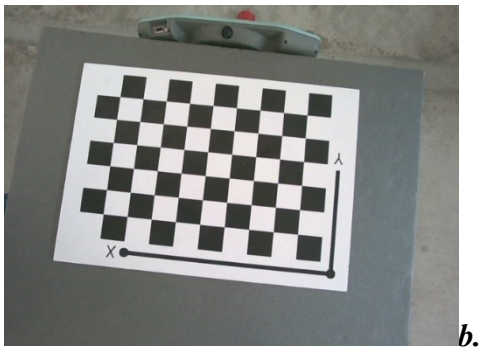
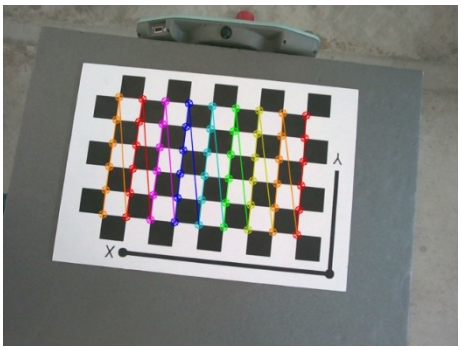
a.  ***b.*** 

Figura 4.3: Immagini salvate della prima posa:
a. *immagine grezza della scacchiera;* ***b.*** *immagine con scacchiera rilevata*

Il rilevamento della scacchiera avviene tramite l'identificazione degli angoli interni del pattern. Dopo una prima individuazione dei corner, la loro posizione viene affinata a livello sub-pixel, in modo da migliorare l'accuratezza della successiva stima di posa. Le immagini nelle quali la scacchiera non viene correttamente rilevata vengono escluse dalla calibrazione, poiché non forniscono una coppia affidabile tra posa camera-target e posa robot. Definito il setup sperimentale e acquisiti i dati necessari, la fase successiva consiste nella stima delle trasformazioni geometriche tra i sistemi di riferimento coinvolti. In particolare, a partire dalle immagini della scacchiera e dalle pose del TCP fornite dal robot, viene dapprima stimata la posa del target rispetto alla camera e successivamente la trasformazione rigida tra camera e TCP operativo.

4.2. CALIBRAZIONE HAND-EYE

Definito il setup sperimentale e acquisiti i dati necessari, la fase successiva consiste nella stima delle trasformazioni geometriche tra i sistemi di riferimento coinvolti. L'obiettivo della procedura è collegare il sistema di riferimento della camera Intel RealSense D455 al sistema di riferimento operativo del robot, in modo che i punti individuati dal sistema di visione possano essere espressi rispetto alla base del robot UR5e e utilizzati per la generazione delle traiettorie. In particolare, a partire dalle immagini della scacchiera e dalle pose del TCP fornite

dal robot, viene dapprima stimata la posa del target rispetto alla camera e successivamente la trasformazione rigida tra camera e TCP operativo. È importante precisare che tale procedura riguarda la stima dei parametri estrinseci del sistema, cioè delle trasformazioni geometriche tra sistemi di riferimento differenti. I parametri intrinseci della camera, invece, non sono stati stimati sperimentalmente in questa fase, ma sono stati ricavati dalla calibrazione di fabbrica della RealSense D455 tramite le funzioni della libreria pyrealsense2.

Prima di descrivere nel dettaglio le singole fasi computazionali, è opportuno chiarire l'origine delle relazioni matematiche impiegate e il ruolo svolto dalle librerie software utilizzate. Le formule introdotte di seguito non derivano direttamente dal codice Python, riportato integralmente in APPENDICE A, ma dalla modellazione geometrica della camera, dalla teoria delle trasformazioni rigide e dalla formulazione classica del problema di calibrazione hand-eye. Il codice implementa numericamente tali modelli mediante funzioni del modulo *calib3d* di OpenCV.

OpenCV, acronimo di Open Source Computer Vision Library, è una libreria open-source ampiamente utilizzata nell'ambito della visione artificiale e dell'elaborazione delle immagini. Essa mette a disposizione algoritmi per il trattamento delle immagini, il rilevamento di feature, la calibrazione delle camere, la stima della posa di oggetti nello spazio e la ricostruzione geometrica. La stessa documentazione ufficiale descrive OpenCV come una libreria open-source per computer vision e machine learning, contenente un insieme esteso di algoritmi ottimizzati per applicazioni di visione artificiale [22].

Nel presente lavoro, OpenCV non viene utilizzata come ambiente di controllo del robot, ma come strumento per risolvere numericamente alcuni passaggi della calibrazione geometrica robot-camera. In particolare, la libreria consente di passare dalle informazioni estratte dall'immagine, cioè le coordinate dei corner della scacchiera, alla stima delle trasformazioni spaziali necessarie per collegare il sistema di riferimento della camera al TCP operativo.

Prima di procedere con la stima delle trasformazioni geometriche, è necessario distinguere tra parametri intrinseci ed estrinseci della camera. I parametri intrinseci descrivono il modello interno di proiezione del sensore e dipendono dalle caratteristiche ottiche della camera e dalla risoluzione dello stream utilizzato. Essi comprendono le lunghezze focali f_x e f_y, le coordinate del punto principale c_x e c_y, e i coefficienti di distorsione dell'immagine.

Nel presente lavoro, tali parametri non sono stati determinati mediante una procedura di calibrazione intrinseca sperimentale, ma sono stati estratti dal profilo dello stream RGB della

RealSense D455. Nel codice Python, dopo l'avvio dello stream, viene acquisito il profilo video attivo della camera e da esso vengono ricavati gli intrinseci mediante la funzione:

get_intrinsics()

In particolare, i valori di f_x, f_y, c_xe $c_y$vengono utilizzati per costruire la matrice intrinseca della camera:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Dove f_x e f_y rappresentano le lunghezze focali espresse in pixel, mentre c_x e c_y individuano il punto principale dell'immagine. I coefficienti di distorsione restituiti dalla camera vengono invece utilizzati come ingresso della funzione solvePnP.

Questa distinzione è fondamentale: la matrice K e i coefficienti di distorsione sono considerati dati noti, derivanti dalla calibrazione di fabbrica della camera; la procedura descritta nel seguito ha invece lo scopo di stimare le trasformazioni estrinseche, cioè la posa della scacchiera rispetto alla camera e la posa della camera rispetto al TCP operativo.

Una volta rilevati gli angoli interni della scacchiera nell'immagine, la prima trasformazione da determinare è la posa della scacchiera rispetto alla camera, indicata come:

$$^{C}T_{S}$$

dove C rappresenta il sistema di riferimento della camera e S il sistema di riferimento solidale alla scacchiera.

Il problema viene formulato come un problema Perspective-n-Point, cioè come la stima della posa di un oggetto tridimensionale a partire da un insieme di corrispondenze tra punti 3D noti e punti 2D rilevati nell'immagine. Nel caso in esame, i punti 3D sono i vertici interni della scacchiera, espressi nel sistema S, mentre i punti 2D sono le relative coordinate pixel individuate nell'immagine RGB.

Il modello geometrico alla base della stima può essere scritto come:

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K[\quad {}^{C}R_S \qquad {}^{C}p_S]\begin{bmatrix} X_S \\ Y_S \\ Z_S \\ 1 \end{bmatrix}$$

dove (u, v) sono le coordinate del punto nell'immagine, (X_S, Y_S, Z_S) sono le coordinate del punto nel sistema della scacchiera, K è la matrice intrinseca della camera, ${}^{C}R_S$ è la matrice di rotazione della scacchiera rispetto alla camera e ${}^{C}p_S$ è il vettore di traslazione dell'origine della scacchiera espresso nel sistema camera.
La matrice intrinseca della camera è definita come:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

dove f_x e f_y rappresentano le lunghezze focali espresse in pixel, mentre c_x e c_y individuano il punto principale dell'immagine. La relazione di proiezione può essere interpretata geometricamente mediante il modello pinhole, nel quale il centro ottico della camera rappresenta il centro di proiezione, mentre il piano immagine raccoglie la proiezione prospettica dei punti tridimensionali osservati. Lo schema riportato in *Figura 4.4* mostra il significato dei principali parametri coinvolti: il punto tridimensionale P, il punto immagine (u,v), l'asse ottico, il punto principale (c_x, c_y) e il sistema di riferimento della camera [23].

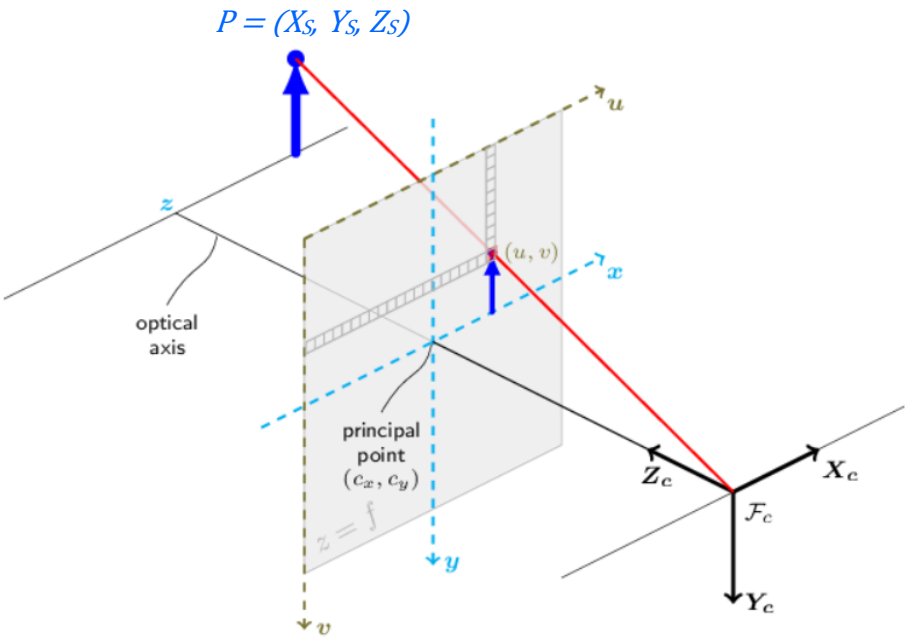


Figura 4.4: Schema del modello di camera pinhole e proiezione prospettica di un punto tridimensionale sul piano immagine.

Nel codice Python, la stima viene effettuata mediante la funzione:

cv2.solvePnP(objectPoints, imagePoints, cameraMatrix, distCoeffs)

La funzione solvePnP riceve in ingresso i punti tridimensionali dell'oggetto, le corrispondenti coordinate immagine, la matrice intrinseca della camera e i coefficienti di distorsione. Secondo la documentazione OpenCV, essa restituisce un vettore di rotazione rvec e un vettore di traslazione tvec che trasformano i punti dal sistema di riferimento dell'oggetto al sistema di riferimento della camera. Nel presente caso, l'oggetto è la scacchiera; pertanto, l'uscita di solvePnP rappresenta proprio la trasformazione della scacchiera rispetto alla camera, cioè ${}^{C}T_{S}$ [23].

Il vettore di rotazione rvec restituito da solvePnP non è direttamente una matrice di rotazione, ma una rappresentazione compatta asse-angolo della rotazione. Per costruire la trasformazione omogenea è quindi necessario convertirlo in una matrice 3×3. Questa conversione viene effettuata nel codice mediante la funzione:

cv2.Rodrigues(rvec)

che consente di passare da un vettore di rotazione a una matrice di rotazione, o viceversa; questa rappresentazione è comunemente usata nelle procedure di ottimizzazione geometrica, tra cui solvePnP.

Una volta ottenuti ${}^{C}R_S$ e ${}^{C}P_S$ la trasformazione omogenea associata alla posa della scacchiera rispetto alla camera viene costruita nella forma:

$$ {}^{C}T_S = \begin{bmatrix} {}^{C}R_S & {}^{C}p_S \\ 000 & 1 \end{bmatrix} $$

Questa matrice permette di trasformare un punto espresso nel sistema della scacchiera nel corrispondente punto espresso nel sistema della camera:

$$ {}^{C}P = {}^{C}T_S {}^{S}P $$

In questa fase, quindi, non viene ancora calcolata la relazione tra camera e robot. Viene soltanto stimata, per ciascuna posa acquisita, la posizione relativa della scacchiera rispetto alla camera. Tale informazione costituirà uno dei due insiemi di dati necessari alla calibrazione hand-eye.

La seconda trasformazione necessaria è quella tra la camera e il TCP operativo del robot. Poiché la camera è montata rigidamente sul tool, la sua posizione relativa rispetto al TCP rimane costante durante tutte le acquisizioni. La trasformazione incognita è quindi:

$$ {}^{TCP}T_C $$

Dove TCP rappresenta il sistema di riferimento associato alla punta della siringa, mentre C rappresenta il sistema di riferimento della camera.

È importante sottolineare che, nella formulazione OpenCV, il sistema solidale al robot è indicato come gripper. Nel presente lavoro, tale sistema non coincide genericamente con la flangia meccanica del robot, ma con il TCP attivo, poiché durante l'acquisizione il TCP era impostato sulla punta della siringa. Pertanto, nella notazione della tesi, il "gripper" della funzione OpenCV viene interpretato come TCP.

Per ciascuna posa valida sono disponibili due trasformazioni:

$$^{B}T_{TCP,i}$$

fornita dal robot, e:

$$^{C}T_{S,i}$$

ottenuta mediante solvePnP.

La prima trasformazione descrive la posa del TCP rispetto alla base robot. Nel codice essa viene ricavata dalla posa effettiva del TCP e convertita in matrice omogenea mediante una funzione ausiliaria che trasforma il vettore posa $[x, y, z, r_x, r_y, r_z]$ in una matrice 4×4. La seconda trasformazione descrive invece la posa della scacchiera rispetto alla camera. Poiché la scacchiera rimane fissa sul piano di lavoro, la sua posa rispetto alla base robot deve essere teoricamente costante per tutte le acquisizioni. Per ogni posa i, vale quindi:

$$^{B}T_{S} = {}^{B}T_{TCP,i}\ {}^{TCP}T_{C}\ {}^{C}T_{S,i}$$

dove $^{B}T_{S}$ è la posa della scacchiera rispetto alla base robot, $^{B}T_{TCP,i}$ è la posa del TCP rispetto alla base per la posa i-esima, $^{TCP}T_{C}$ è la trasformazione incognita tra TCP e camera, e $^{C}T_{S,i}$ è la posa della scacchiera rispetto alla camera.

Questa relazione esprime la catena cinematica completa dalla scacchiera alla base robot, passando attraverso camera e TCP. La trasformazione incognita è $^{TCP}T_{C}$, cioè la posa della camera rispetto al TCP operativo.

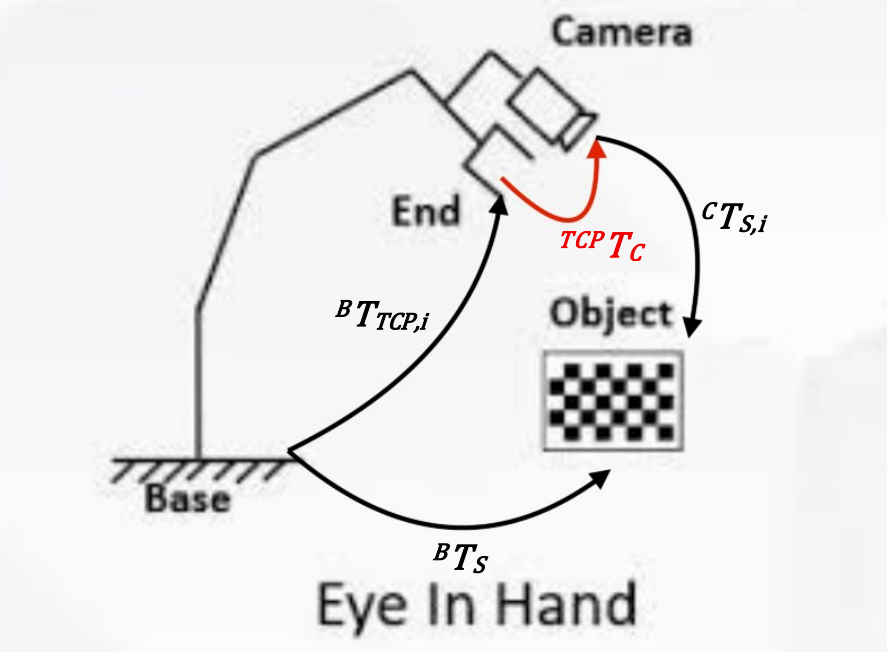


Figura 4.5: Rappresentazione della catena cinematica del sistema

In forma classica, il problema hand-eye può essere ricondotto all'equazione:

$$A_i X = X B_i$$

dove X rappresenta la trasformazione incognita tra la camera e il sistema solidale al robot, mentre A_ie $B_i$sono trasformazioni relative ottenute confrontando coppie di pose. Nel caso in esame, la trasformazione incognita X coincide con: $X = {}^{TCP}T_C$. L'equazione $A_i X = X B_i$ deriva dalla condizione di invarianza della posa della scacchiera rispetto alla base robot. Per due acquisizioni differenti, indicate con i e j, si può scrivere:

$$^{B}T_{TCP,i}\ ^{TCP}T_C\ ^{C}T_{S,i} = {}^{B}T_{TCP,j}\ ^{TCP}T_C\ ^{C}T_{S,j}$$

Poiché la scacchiera rimane fissa sul piano di lavoro, entrambe le catene cinematiche devono restituire la stessa posa del target rispetto alla base robot. Riorganizzando l'equazione precedente si ottengono le trasformazioni relative:

$$A_{ij} = (^{B}T_{TCP,j})^{-1}\ ^{B}T_{TCP,i}$$

$$B_{ij} = {}^{C}T_{S,j}\ (CT_{S,i})^{-1}$$

e quindi la relazione precedentemente descritta che esprime il vincolo fondamentale della calibrazione hand-eye, ovvero il moto relativo misurato dal robot deve essere coerente, attraverso la trasformazione incognita X, con il moto relativo osservato dalla camera rispetto alla scacchiera.
La funzione OpenCV utilizzata per stimare questa trasformazione è:

cv2.calibrateHandEye

La documentazione OpenCV definisce gli ingressi della funzione come le rotazioni e traslazioni da gripper a base e da target a camera, e restituisce la rotazione e la traslazione da camera a gripper, cioè ${}^{g}T_{c}$. Nel presente lavoro, sostituendo g con TCP, l'uscita della funzione corrisponde alla trasformazione ${}^{TCP}T_{C}$. Nel codice Python sviluppato, la funzione viene alimentata con quattro insiemi di dati: [$R_{gripper2base}$, $t_{gripper2base}$, $R_{target2cam}$, $t_{target2cam}$] che nel codice Python equivalgono a [${}^{B}R_{TCP,i}$, ${}^{B}p_{TCP,i}$, ${}^{C}R_{S,i}$, ${}^{C}p_{S,i}$].
Il termine *target* utilizzato da OpenCV indica la scacchiera di calibrazione, mentre il termine *camera* indica il sistema di riferimento della D455. Analogamente, il termine *gripper* viene reinterpretato come TCP attivo, coerentemente con la configurazione adottata nella prova.
La funzione calibrateHandEye consente di selezionare diversi metodi di risoluzione del problema hand-eye; in questo caso la stima finale è stata ottenuta utilizzando il metodo di Tsai-Lenz, richiamato mediante il parametro: cv2.CALIB_HAND_EYE_TSAI. Il metodo di Tsai-Lenz rappresenta uno degli approcci classici per la calibrazione hand-eye ed è definito separabile, poiché affronta la stima della trasformazione incognita in due fasi successive: prima viene determinata la componente rotazionale e successivamente quella traslazionale. Il principio alla base del metodo è che il moto relativo misurato dal robot tra due pose deve essere compatibile con il moto relativo osservato dalla camera rispetto alla scacchiera tra le stesse acquisizioni. Tali moti non coincidono direttamente, poiché sono espressi in sistemi di riferimento differenti, ma sono legati dalla trasformazione incognita: $X = {}^{TCP}T_{C}$
A partire dall'equazione hand-eye:

$$A_{ij}X = XB_{ij}$$

il metodo stima dapprima la rotazione R_X, cioè l'orientamento della camera rispetto al TCP. Questa stima viene effettuata confrontando le rotazioni relative associate ai movimenti del robot con quelle ricavate dalle osservazioni della camera. Una volta determinata la rotazione, la traslazione p_X viene calcolata risolvendo la parte traslazionale del problema. Indicando le trasformazioni relative nella forma:

$$A_{ij} = \begin{bmatrix} R_{A_{ij}} & p_{A_{ij}} \\ 0 & 1 \end{bmatrix}$$

$$B_{ij} = \begin{bmatrix} R_{B_{ij}} & p_{B_{ij}} \\ 0 & 1 \end{bmatrix}$$

e la trasformazione incognita come:

$$X = \begin{bmatrix} R_X & p_X \\ 0 & 1 \end{bmatrix}$$

la relazione traslazionale può essere scritta, una volta nota R_X, come:

$$(R_{A_{ij}} - I)p_X = R_X p_{B_{ij}} - p_{A_{ij}}$$

dove I è la matrice identità. In questa fase l'unica incognita è il vettore p_X, che rappresenta la posizione dell'origine della camera rispetto al TCP operativo. Poiché la procedura utilizza più pose, ogni coppia di acquisizioni fornisce un vincolo sulla stessa trasformazione incognita. In condizioni ideali tutti i vincoli sarebbero perfettamente compatibili; nella pratica, invece, sono presenti errori dovuti al rilevamento dei corner, alla stima PnP, alla risoluzione dell'immagine, alle tolleranze di montaggio e alla definizione del TCP. Per questo motivo la trasformazione ${}^{TCP}T_C$ viene stimata come soluzione che rende complessivamente più coerenti i vincoli disponibili, secondo un criterio di tipo ai minimi quadrati. La qualità della calibrazione dipende quindi anche dalla varietà delle pose acquisite. Configurazioni troppo simili tra loro, soprattutto dal punto di vista dell'orientamento, forniscono vincoli poco informativi e possono rendere la stima meno robusta. Per questo motivo, durante la procedura sperimentale, il robot è stato movimentato in 18 pose differenti, variando sia la posizione sia l'orientamento del tool rispetto alla scacchiera.

Nel codice sono stati valutati anche altri metodi disponibili in OpenCV, tra cui Park, Horaud, Andreff e Daniilidis. Il confronto tra le diverse soluzioni consente di valutare la stabilità numerica della calibrazione: risultati simili tra metodi diversi indicano una maggiore coerenza

della stima, mentre differenze marcate possono evidenziare una distribuzione non ottimale delle pose o errori nella procedura di acquisizione. La matrice finale adottata viene quindi salvata sia in formato testo sia in formato NumPy, in modo da poter essere richiamata nelle successive fasi di trasformazione dei punti e generazione delle traiettorie.

La trasformazione ottenuta assume la forma:

$$^{TCP}T_C = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

dove la sottomatrice 3×3descrive l'orientamento della camera rispetto al TCP, mentre il vettore:

$$^{TCP}p_C = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

descrive la posizione dell'origine del sistema camera rispetto alla punta della siringa.

$$^{TCP}T_C = \begin{bmatrix} -1.096e-03 & 1.121e-02 & 9.999e-01 & 7.848e-02 \\ 9.999e-01 & 1.334e-02 & 9.466e-04 & -6.811e-02 \\ -1.333e-02 & 9.99e-01 & -1.123e-02 & -2.633e-01 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Una volta determinata la trasformazione $^{TCP}T_C$, è possibile calcolare la posa della camera rispetto alla base del robot per qualsiasi configurazione del manipolatore. La catena cinematica utilizzata è:

$$^{B}T_C = {}^{B}T_{TCP}\,{}^{TCP}T_C$$

Questa relazione deriva dalla composizione di trasformazioni omogenee. Un punto espresso nel sistema camera viene dapprima trasformato nel sistema TCP e successivamente nel sistema della base robot. La documentazione OpenCV richiama l'uso delle trasformazioni

omogenee per esprimere il cambiamento di coordinate tra sistemi di riferimento diversi, combinando una matrice di rotazione e un vettore di traslazione.
Dato un punto acquisito dalla camera:

$$^{C}P = \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix}$$

la sua posizione nel sistema di riferimento della base robot è data da:

$$^{B}P = {}^{B}T_C \, {}^{C}P$$

Sostituendo la catena cinematica completa si ottiene:

$$^{B}P = {}^{B}T_{TCP} \, {}^{TCP}T_C \, {}^{C}P$$

Questa equazione rappresenta il collegamento fondamentale tra il sistema di visione e il sistema robotico. I punti individuati dalla camera D455, inizialmente espressi nel sistema camera, vengono trasformati nel sistema di riferimento della base del robot UR5e. In questo modo possono essere utilizzati per generare traiettorie eseguibili dal TCP operativo, cioè dalla punta della siringa. Nel contesto applicativo della decorazione alimentare, questa trasformazione è indispensabile per associare il profilo della vaschetta riconosciuto dalla camera alla traiettoria che dovrà essere seguita dalla punta di erogazione. La precisione della calibrazione camera-TCP influenza quindi direttamente la corrispondenza tra la traiettoria calcolata dal sistema di visione e la traiettoria realmente eseguita dal robot.
La validazione della calibrazione è stata effettuata mediante una verifica sperimentale diretta, confrontando le coordinate dei punti trasformati nel sistema di riferimento della base robot con misure fisiche rilevate sul piano di lavoro. L'obiettivo della validazione non era quindi effettuare una caratterizzazione metrologica ad alta precisione del sistema, ma verificare che la calibrazione fosse sufficientemente accurata per consentire il corretto trasferimento delle informazioni dalla camera al robot. La procedura ha confermato che i punti espressi nel sistema della camera, una volta trasformati nel sistema della base robot, risultavano coerenti con la posizione reale degli elementi osservati. In generale, errori traslazionali dell'ordine di

pochi millimetri e rotazioni inferiori al grado possono essere considerati accettabili per una prima implementazione sperimentale, soprattutto considerando la presenza di componenti prototipali, supporti stampati in 3D, tolleranze di montaggio e possibili deformazioni locali del sistema di fissaggio. Nel caso specifico dell'applicazione sviluppata, la precisione richiesta non è assimilabile a quella di un'operazione robotica di assemblaggio fine, ma deve essere sufficiente a garantire il corretto allineamento tra la traiettoria generata dal sistema di visione e l'area di deposizione sulla vaschetta.

La calibrazione ottenuta rimane valida finché la configurazione meccanica del sistema non viene modificata. Qualsiasi variazione della posizione della camera, smontaggio del tool, sostituzione dei supporti, modifica della siringa o ridefinizione del TCP richiede una nuova procedura di calibrazione. Allo stesso modo, urti, giochi meccanici o deformazioni del supporto possono alterare la trasformazione camera-TCP e compromettere la precisione del sistema.

In conclusione, la calibrazione hand-eye costituisce un passaggio fondamentale per l'integrazione tra visione artificiale e controllo robotico. Essa consente di trasformare le informazioni acquisite dalla camera D455 in coordinate utilizzabili dal robot UR5e, rendendo possibile l'esecuzione automatizzata delle traiettorie di decorazione mediante il tool di erogazione sviluppato.

5. SISTEMA DI VISIONE E GENERAZIONE DELLA TRAIETTORIA

A valle della procedura di calibrazione robot-camera descritta nel capitolo precedente, il sistema è in grado di trasformare i punti acquisiti dalla camera Intel RealSense D455 nel sistema di riferimento della base del robot. Questa informazione costituisce il presupposto fondamentale per la generazione automatica della traiettoria di decorazione, poiché consente di associare le coordinate immagine della vaschetta a punti tridimensionali raggiungibili dal TCP operativo, coincidente con la punta di erogazione del materiale.

Il software sviluppato, riportato in APPENDICE B, ha quindi il compito di integrare tre funzioni principali: l'acquisizione delle immagini RGB-D dalla camera, l'elaborazione dell'immagine per individuare la geometria della vaschetta e la trasformazione dei punti rilevati in waypoint robotici. In questo modo, il robot può portarsi inizialmente in una posa di osservazione, acquisire la scena, riconoscere la vaschetta e generare automaticamente una sequenza di punti lungo i quali eseguire il movimento di deposizione. La traiettoria prodotta non è costituita soltanto dai vertici della vaschetta, ma da un contorno denso di punti, calcolato lungo i lati della sagoma rilevata e successivamente modificato per ottenere una traiettoria più adatta alla realizzazione dei ciuffi di maionese. Il programma integrale

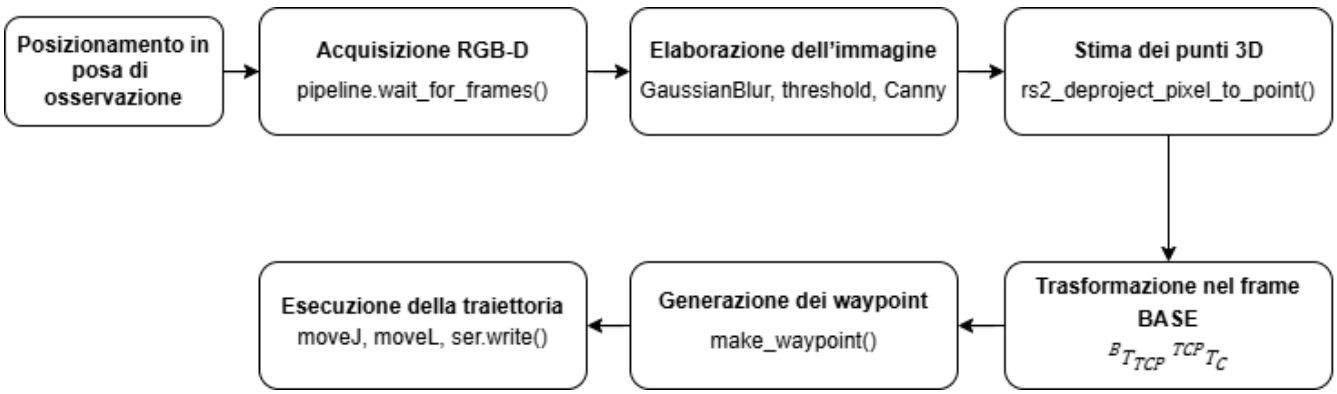


Figura 5.1: Schema generale della pipeline software

Nella prima fase del programma il software stabilisce le connessioni necessarie con il robot collaborativo UR5e, con la camera Intel RealSense D455 e con il sistema di attuazione dell'asse lineare. Questa fase consente di predisporre il sistema all'acquisizione delle immagini, alla

ricezione delle informazioni di stato del robot e alla successiva esecuzione della traiettoria di deposizione. Gli aspetti relativi alle modalità di comunicazione, ai protocolli utilizzati e alla sincronizzazione tra robot e asse lineare verranno approfonditi nel capitolo dedicato al sistema di comunicazione e controllo.
Prima di avviare la fase di visione, il robot viene portato in una configurazione prestabilita, indicata nel codice come posa di osservazione. Tale posa è definita mediante il vettore di giunti:

$$\text{OBSERVE\_DEG} = [-7.14°, -97.76°, -77.54°, 175.65°, 32.03°, -90.45°]$$

Poiché i comandi di movimentazione del robot richiedono che le posizioni articolari siano espresse in radianti, il vettore viene successivamente convertito mediante la funzione np.deg2rad(), ottenendo il vettore OBSERVE_RAD. Il raggiungimento della posa di osservazione viene quindi effettuato tramite il comando:

```
rtde_c.moveJ(OBSERVE_RAD, MOVE_SPEED, MOVE_ACCEL)
```

Il comando moveJ realizza un movimento di tipo articolare, cioè definito nello spazio dei giunti del robot. A differenza di un movimento cartesiano lineare, come quello ottenuto con moveL, il comando moveJ non impone al TCP di seguire una traiettoria retti linea nello spazio operativo, ma calcola una transizione tra la configurazione articolare corrente e quella target, movimentando i sei giunti fino al raggiungimento dei valori specificati in OBSERVE_RAD. Questo tipo di movimento è particolarmente adatto per portare il robot in una configurazione nota di partenza, come la posa di osservazione, poiché consente di raggiungere rapidamente e in modo controllato una determinata configurazione dei giunti.
Nel comando utilizzato, OBSERVE_RAD rappresenta quindi il vettore delle posizioni angolari target dei sei giunti, mentre le due costanti successive, MOVE_SPEED e MOVE_ACCEL, definiscono rispettivamente la velocità massima e l'accelerazione massima ammesse durante il movimento articolare. Tali parametri non descrivono direttamente la velocità lineare della punta dell'utensile, ma regolano la rapidità con cui il robot esegue la transizione nello spazio dei giunti.
Il profilo di velocità adottato dal controllore può essere schematizzato mediante una curva di tipo trapezoidale, come riportato in *Figura 5.2* [24]. In una prima fase il robot accelera progressivamente fino al valore di velocità impostato; successivamente, se lo spostamento richiesto lo consente, mantiene una fase a velocità pressoché costante; infine decelera fino ad

arrestarsi in corrispondenza della configurazione target. La pendenza dei tratti di accelerazione e decelerazione dipende dal parametro MOVE_ACCEL, mentre il valore massimo raggiungibile nella fase centrale dipende dal parametro MOVE_SPEED. Nel caso in cui lo spostamento tra la configurazione iniziale e quella finale sia ridotto, il robot potrebbe non raggiungere la velocità massima impostata e il profilo assumere una forma più simile a quella triangolare.

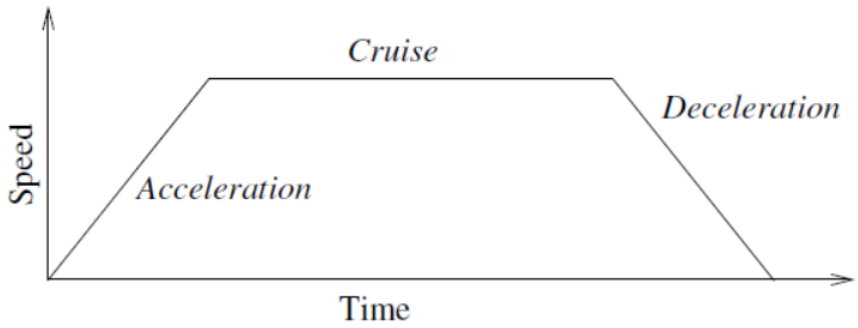


Figura 5.2: Profilo qualitativo della velocità durante un movimento robotico

Nel presente lavoro i valori di velocità e accelerazione sono stati definiti come MOVE_SPEED = 0,5 in rad/s e MOVE_ACCEL = 0,5 in rad/s^2, al fine di ottenere un movimento controllato verso la posa di osservazione. Questa scelta è coerente con la natura sperimentale del sistema, poiché riduce movimenti bruschi dell'end-effector, limita eventuali vibrazioni della camera e permette di acquisire l'immagine da una configurazione stabile e ripetibile. In questo modo la trasformazione tra camera e base robot può essere calcolata utilizzando la posa effettiva del TCP in osservazione, ottenuta tramite:

rtde_r.getActualTCPPose()

La posa TCP restituita dal robot, espressa nella forma $[x, y, z, r_x, r_y, r_z]$, viene poi convertita in una matrice omogenea mediante la funzione ausiliaria pose_to_matrix(). Tale funzione utilizza cv2.Rodrigues() per convertire il vettore di rotazione in una matrice di rotazione 3x3, costruendo poi la matrice omogenea:

$$ {}^{B}T_{TCP,obs} = \begin{bmatrix} {}^{B}R_{TCP,obs} & {}^{B}p_{TCP,obs} \\ 0 & 1 \end{bmatrix} $$

dove B indica il sistema di riferimento della base robot e TCP il sistema di riferimento associato alla punta di erogazione. Successivamente viene caricata la matrice di calibrazione hand-eye precedentemente stimata e salvata nel file handeye_result_chess_ciuffi.npy. Tale matrice rappresenta la trasformazione tra camera e TCP operativo:

$$^{TCP}T_C$$

Pertanto, per ogni punto acquisito dalla camera, la trasformazione completa utilizzata dal software è:

$$^{B}P = \quad ^{B}T_{TCP,obs} \quad ^{TCP}T_C \quad ^{C}P$$

dove ^{C}P è il punto espresso nel sistema della camera e ^{B}P è lo stesso punto espresso rispetto alla base del robot. Questa relazione costituisce il collegamento operativo tra il sistema di visione e la generazione della traiettoria.

Dopo l'inizializzazione del robot, viene avviata la pipeline della camera RealSense D455. Nel codice vengono abilitati due stream: lo stream RGB a colori e lo stream di profondità. Entrambi sono configurati con risoluzione 640x480 pixel e frequenza di acquisizione pari a 30 fps:

```
config.enable_stream(rs.stream.color, 640, 480, rs.format.bgr8, 30)
config.enable_stream(rs.stream.depth, 640, 480, rs.format.z16, 30)
```

La camera non viene quindi utilizzata come dispositivo di acquisizione a singolo scatto, ma opera in streaming continuo. Ad ogni iterazione del ciclo principale, il software acquisisce una coppia di frame RGB e depth mediante:

```
pipeline.wait_for_frames()
```

Poiché il frame RGB e il frame di profondità provengono da sensori fisicamente distinti all'interno della camera, è necessario allinearli geometricamente. Nel software questa operazione viene effettuata mediante:

```
align = rs.align(rs.stream.color)
aligned = align.process(frames)
```

L'allineamento consente di riferire le informazioni di profondità allo stesso sistema pixel dell'immagine RGB. In questo modo, il pixel *(u,v)* individuato sull'immagine a colori può essere associato al corrispondente valore di profondità. Questo passaggio è fondamentale perché il riconoscimento della vaschetta avviene sull'immagine RGB, mentre la ricostruzione tridimensionale richiede il valore di profondità associato agli stessi pixel. La libreria RealSense fornisce strumenti specifici per lo streaming RGB-D, l'allineamento tra stream e l'accesso ai parametri intrinseci ed estrinseci del sensore.
Sul frame di profondità vengono inoltre applicati alcuni filtri di post-processing:

```
decimation_filter()
spatial_filter()
temporal_filter()
hole_filling_filter()
```

Il filtro di decimazione consente di ridurre il rumore e semplificare la mappa di profondità; il filtro spaziale regolarizza localmente il dato depth; il filtro temporale riduce le oscillazioni tra frame consecutivi; infine, il filtro di riempimento dei buchi permette di attenuare la presenza di pixel privi di misura valida. L'obiettivo complessivo di questa catena di filtraggio è rendere più stabile la successiva conversione dei punti immagine in coordinate tridimensionali.

a. 

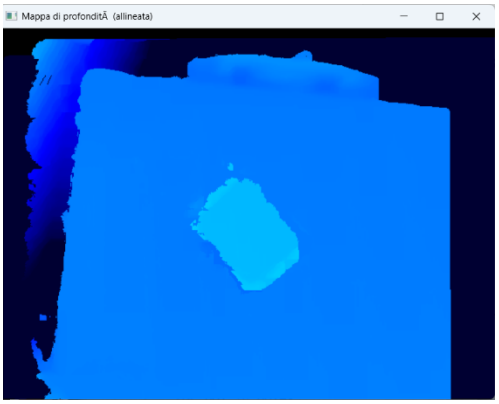
b. 

Figura 5.3: ***a.*** *Frame RGB acquisito dalla RealSense D455;* ***b.*** *relativa mappa di profondità*

Una volta acquisito il frame RGB, il software procede con la fase di elaborazione dell'immagine. Tale fase è implementata nella funzione detect_boxes(color_image), che ha il compito di individuare la sagoma della vaschetta all'interno della scena.
Il primo passaggio consiste nella conversione dell'immagine a colori in scala di grigi:

```
gray = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)
```

La conversione in scala di grigi riduce l'informazione dell'immagine alla sola intensità luminosa, semplificando le successive operazioni di segmentazione. Successivamente viene applicato un filtro gaussiano:

```
blurred = cv2.GaussianBlur(gray, (7, 7), 0)
```

Il filtro gaussiano attenua le variazioni locali di intensità e riduce il rumore ad alta frequenza, rendendo più regolare il bordo della vaschetta. Dopo il filtraggio, l'immagine viene binarizzata tramite una soglia fissa:

```
_, binary = cv2.threshold(blurred, 160, 255, cv2.THRESH_BINARY)
```

La binarizzazione separa le regioni chiare da quelle scure e consente di ottenere un'immagine più adatta all'estrazione dei contorni. Sul risultato binario viene quindi applicato l'algoritmo di Canny:

```
edges = cv2.Canny(binary, CANNY_LOW, CANNY_HIGH)
```

con soglie pari a CANNY_LOW = 50 e CANNY_HIGH = 150. L'operatore di Canny consente di evidenziare i bordi dell'oggetto, cioè le zone dell'immagine in cui si verifica una variazione significativa di intensità.
Per chiudere eventuali discontinuità presenti nei bordi rilevati, il software applica una trasformazione morfologica di chiusura mediante un kernel rettangolare 5x5:

```
closed = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel)
```

Questa operazione migliora la continuità del profilo della vaschetta, favorendo l'individuazione di un contorno chiuso.

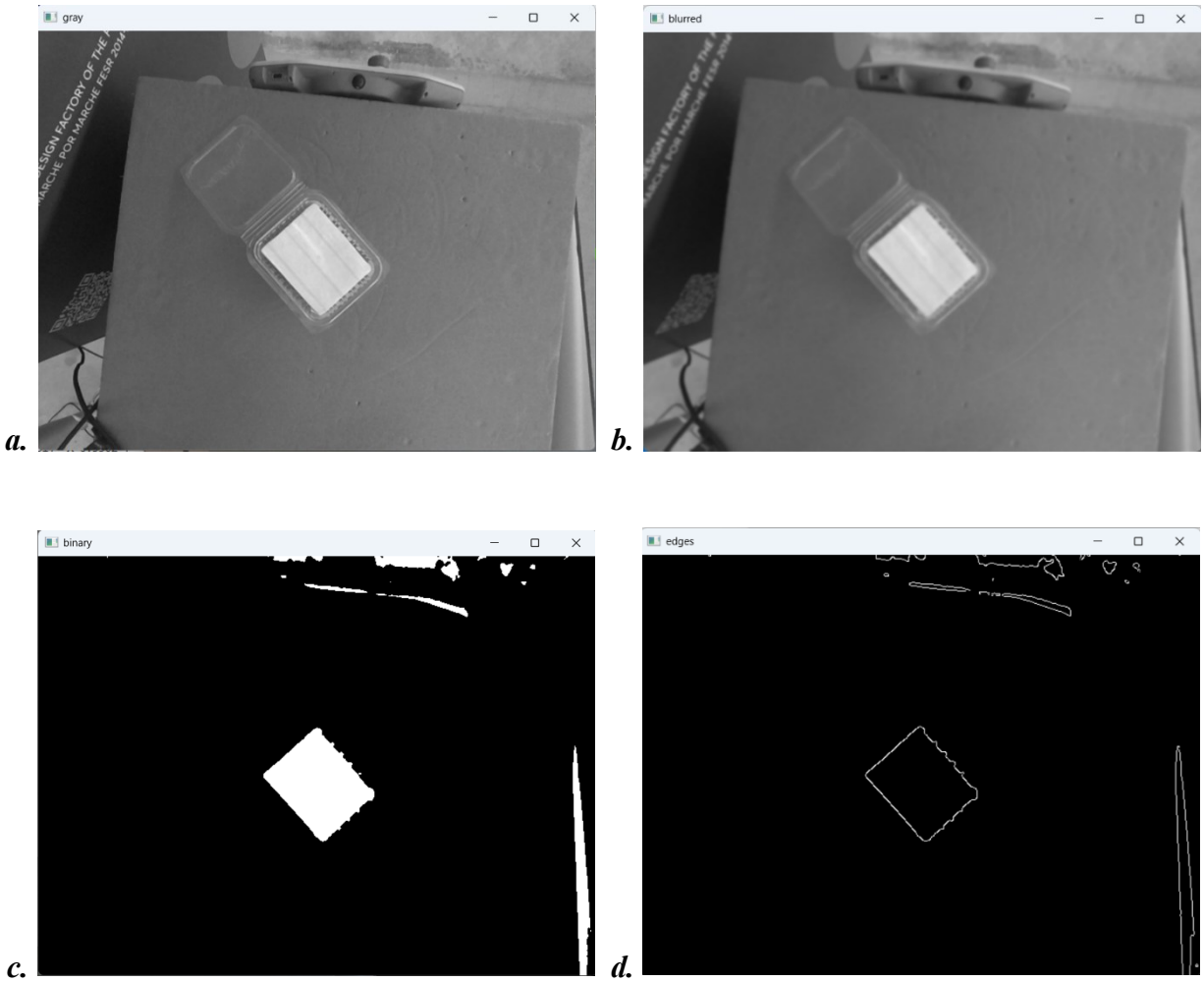


Figura 5.4: Sequenza di elaborazione immagine: ***a.*** *scala di grigi;* ***b.*** *immagine filtrata;* ***c.*** *binarizzazione;* ***d.*** *bordi Canny*

I contorni vengono quindi estratti con:

```
cv2.findContours(closed, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
```

La modalità RETR_EXTERNAL consente di considerare soltanto i contorni esterni, mentre CHAIN_APPROX_NONE mantiene tutti i punti del contorno rilevato. La documentazione OpenCV descrive findContours() come una funzione per individuare i contorni in immagini

binarie, mentre approxPolyDP() viene utilizzata per approssimare una curva con una poligonale avente un numero inferiore di vertici [25].
Ogni contorno individuato viene sottoposto a una serie di filtri geometrici. Per prima cosa, viene calcolata l'area mediante:

```
cv2.contourArea(cnt)
```

e vengono scartati i contorni con area inferiore a MIN_CONTOUR_AREA = 1000 pixel o superiore a MAX_CONTOUR_AREA = 500000 pixel. Successivamente viene calcolato il perimetro:

```
peri = cv2.arcLength(cnt, True)
```

e il contorno viene approssimato con una poligonale mediante:

```
approx_raw = cv2.approxPolyDP(cnt, POLY_EPS * peri, True)
```

Nel software il parametro POLY_EPS è pari a 0.04; ciò significa che la tolleranza dell'approssimazione è proporzionale al perimetro del contorno. Vengono mantenuti soltanto i contorni la cui approssimazione presenta quattro vertici, poiché la vaschetta viene ricondotta a una forma rettangolare o quadrangolare. Infine, mediante cv2.boundingRect(), viene calcolato il rettangolo contenente il contorno e viene verificato che il rapporto tra larghezza e altezza ricada nell'intervallo:

$$0.3 < \frac{w}{h} < 3.5$$

Questa condizione permette di eliminare oggetti o disturbi con forma non compatibile con quella della vaschetta.

Per rendere più stabile il rilevamento della vaschetta, il software non utilizza direttamente i vertici ottenuti da approxPolyDP() nel singolo frame perchè anche se la vaschetta è ferma, il contorno rilevato può variare leggermente da un'immagine alla successiva a causa del rumore dell'immagine, della sogliatura, delle condizioni di illuminazione e delle approssimazioni

introdotte dagli algoritmi di elaborazione. Tali piccole variazioni potrebbero produrre una traiettoria non perfettamente stabile, con conseguenti oscillazioni dei waypoint generati.
Per ridurre questo effetto, il programma applica una stabilizzazione temporale attraverso la funzione

stabilize_vertices()

A ogni frame viene calcolato, a partire dal contorno rilevato, il rettangolo ruotato di area minima che racchiude la vaschetta mediante la funzione cv2.minAreaRect(). Tale rettangolo è descritto da cinque grandezze: le coordinate del centro, la larghezza, l'altezza e l'angolo di orientamento $[x_c, y_c, w, h, \theta]$. Questi valori non vengono usati singolarmente, ma vengono inseriti in una memoria temporanea contenente gli ultimi dieci rilevamenti, definita nel codice tramite il parametro VERTEX_HISTORY = 10.
In questo modo, per ogni nuovo frame, il software dispone non solo della misura corrente, ma anche delle misure ricavate nei frame immediatamente precedenti. Su questi valori viene calcolata una media mobile, ottenendo un rettangolo medio più regolare e meno sensibile alle oscillazioni istantanee del rilevamento. Se il numero di frame memorizzati supera il valore impostato, il dato più vecchio viene automaticamente eliminato e sostituito dal nuovo rilevamento. Il sistema considera quindi sempre una finestra temporale aggiornata degli ultimi frame acquisiti.
A partire dal rettangolo medio così ottenuto, vengono poi ricavati i quattro vertici stabilizzati tramite cv2.boxPoints(). Infine, i vertici vengono ordinati secondo un criterio geometrico basato sull'angolo rispetto al centroide, in modo da mantenere una sequenza coerente dei punti tra un frame e il successivo. Questa procedura consente di ottenere una sagoma della vaschetta più stabile e quindi una generazione dei waypoint più regolare e ripetibile.
La stabilizzazione temporale ha quindi una doppia funzione: riduce le oscillazioni locali dei vertici e garantisce una numerazione coerente dei punti. Questo aspetto è particolarmente importante perché la traiettoria robotica viene generata collegando tra loro i punti del contorno; un cambiamento improvviso nell'ordine dei vertici produrrebbe una traiettoria discontinua o non coerente con la geometria reale della vaschetta.
Dopo aver individuato e stabilizzato i quattro vertici della vaschetta, il software non utilizza direttamente il contorno grezzo estratto da OpenCV. Tale contorno, infatti, può risultare irregolare e sensibile al rumore. Per ottenere una traiettoria più regolare e ripetibile, viene costruito un

contorno artificiale denso, generato sui segmenti retti che collegano i vertici stabilizzati. Questa operazione è implementata nella funzione:

build_contour_with_vertices()

La funzione riceve in ingresso il contorno, i vertici stabilizzati, il centro della vaschetta e alcuni parametri di regolazione. Per ogni lato della vaschetta viene considerato il segmento compreso tra un vertice e il successivo. Su ciascun segmento vengono generati punti intermedi equidistanti, separati da un passo pari a VERTEX_STEP = 20 px.
In questo modo, la traiettoria non è composta soltanto dai quattro vertici, ma da una sequenza più fitta di punti distribuiti lungo il perimetro della vaschetta.
Un ulteriore accorgimento riguarda l'applicazione di un rientro verso il centroide della vaschetta. Per evitare che la traiettoria venga generata esattamente sul bordo esterno della vaschetta, ogni punto del contorno viene traslato verso il centro attraverso la funzione:

inset_point(px, py, cx, cy, inset_px)

che calcola la direzione dal punto considerato verso il centroide e sposta il punto di una distanza prefissata in pixel. Nel codice vengono utilizzati due valori distinti: VERTEX_INSET_PX = 15 px per i vertici e CONTOUR_INSET_PX = 10 px per i punti intermedi.
La scelta di applicare un rientro verso il centro differenziato permette di trattare in modo diverso i vertici e i punti sui lati. I vertici, infatti, sono zone più critiche perché corrispondono ai cambi di direzione della traiettoria; un rientro leggermente maggiore consente di evitare il passaggio troppo vicino agli spigoli della vaschetta. I punti intermedi, invece, vengono mantenuti più prossimi al bordo, in modo da seguire comunque l'andamento generale della geometria rilevata.
Il risultato di questa fase è una lista ordinata di coordinate pixel:

$$P_i^{px} = (u_i, v_i)$$

che rappresentano i punti del contorno utilizzabili per la successiva generazione della traiettoria robotica.

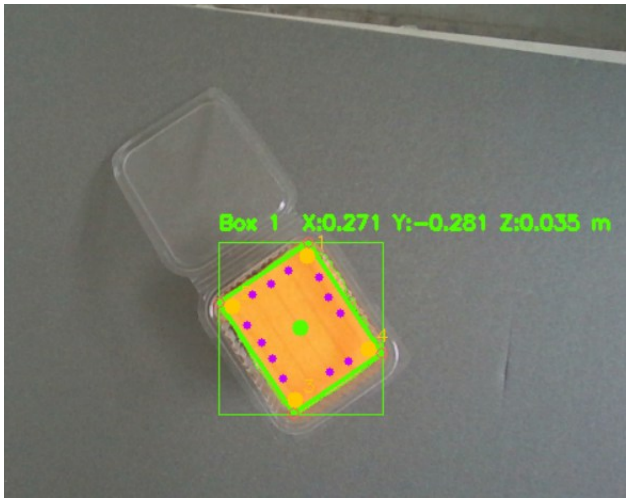


Figura 5.5: Contorno denso generato sui lati della vaschetta

I punti ottenuti nella fase precedente sono inizialmente espressi in coordinate pixel. Per poter essere utilizzati dal robot, essi devono essere trasformati in coordinate tridimensionali metriche. Questa conversione avviene in due passaggi: prima il pixel viene deproiettato nel sistema di riferimento della camera, poi il punto viene trasformato nel sistema di riferimento della base robot.

Il software utilizza la funzione robust_depth_at() per ricavare il valore di profondità associato a ciascun pixel. Invece di considerare il singolo valore depth nel pixel (u, v), viene analizzata una piccola regione quadrata centrata sul punto, definita dal parametro roi_half = 8. Da questa regione vengono selezionati soltanto i valori di profondità compresi tra DEPTH_MIN_MM = 10 mm e DEPTH_MAX_MM = 700 mm.

Se il numero di valori validi è sufficiente, il software utilizza la mediana della regione come stima robusta della profondità:

$$Z_C = median(ROI_{depth})$$

L'uso della mediana consente di ridurre l'influenza di eventuali valori anomali o pixel privi di misura affidabile. Se invece la regione contiene un numero insufficiente di valori validi, il punto viene scartato e non viene utilizzato per la generazione della traiettoria.

Una volta ottenuta la profondità, il pixel viene trasformato in un punto 3D nel sistema camera tramite la funzione RealSense:

```
rs.rs2_deproject_pixel_to_point(intrinsics, [px, py], depth_m)
```

Questa funzione utilizza i parametri intrinseci della camera e la misura di profondità per passare dalle coordinate immagine (u, v)alle coordinate spaziali (X_C, Y_C, Z_C). La documentazione RealSense include la funzione rs2_deproject_pixel_to_point() tra gli strumenti per convertire un pixel e una profondità in un punto 3D usando gli intrinseci della camera.
Il punto in coordinate camera viene quindi espresso in forma omogenea:

$$^{C}P = \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix}$$

e trasformato nel sistema base robot mediante la catena cinematica: $^{B}P = {}^{B}T_{TCP,obs}\ {}^{TCP}T_C\ {}^{C}P$.
Nel codice questa operazione è implementata nella funzione pixel_to_base(), dove la trasformazione viene calcolata con:

p_base = T_base_ee_obs @ T_ee_cam @ p_cam

Il risultato è una terna di coordinate:

$$^{B}P_i = (X_i, Y_i, Z_i)$$

espressa in metri rispetto alla base del robot UR5e.
La stessa procedura viene applicata sia al centro della vaschetta sia ai punti del contorno denso. Il centro viene calcolato mediante la funzione box_center_to_base(), mentre i punti del contorno vengono convertiti durante la fase di esecuzione del comando di movimento. Eventuali punti per i quali non sia disponibile una misura di profondità valida vengono esclusi dalla lista dei waypoint.

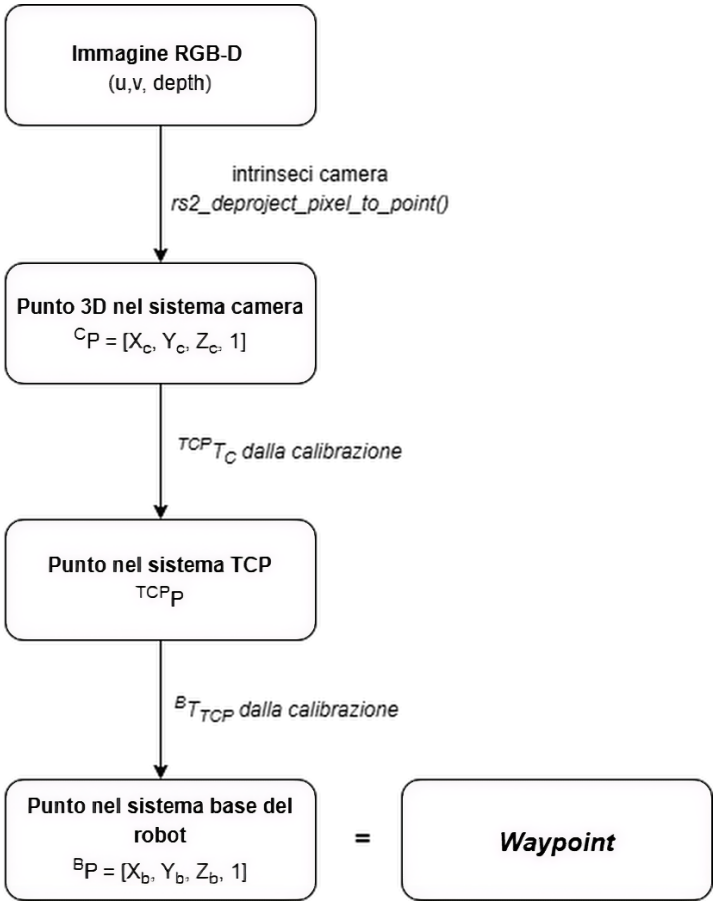


Figura 5.6: Schema della trasformazione

Una volta ottenuti i punti tridimensionali del contorno, il software procede alla generazione dei waypoint robotici. Un waypoint è una posa del TCP, costituita da una posizione (X, Y, Z) e da un orientamento (r_x, r_y, r_z). Nel codice la costruzione della posa avviene tramite la funzione:

make_waypoint(x, y, z, rvec_target, z_offset)

Questa funzione parte dalla posa TCP di osservazione e ne modifica la posizione, sostituendo le coordinate *x, y, z* con quelle del punto da raggiungere. L'orientamento viene invece imposto mediante il vettore rvec_target, calcolato in modo da orientare correttamente la punta di erogazione rispetto al piano di lavoro.

Per evitare che la punta della siringa lavori esattamente alla quota del punto rilevato dalla camera, vengono introdotti alcuni offset verticali. In particolare, il codice definisce: Z_OFFSET_M = 0.02 m per il punto centrale e Z_CONTOUR_OFFSET_M = 0.017 m per i punti del contorno. Questi

offset rappresentano un sollevamento della traiettoria rispetto alla quota misurata, utile per mantenere una distanza controllata tra l'ugello e la superficie di deposizione.
La generazione dei waypoint è attivata dall'operatore tramite il tasto spazio. Fino a quel momento, il sistema continua ad acquisire ed elaborare immagini in tempo reale. Quando viene premuto il tasto spazio, il software seleziona la prima vaschetta valida rilevata, salva un frame annotato con timestamp e converte in 3D tutti i punti del contorno denso. Il frame salvato rappresenta quindi una documentazione dell'immagine utilizzata per la generazione della traiettoria.
È presente anche una modalità alternativa, attivabile con il tasto V, nella quale la traiettoria viene generata utilizzando soltanto i quattro vertici stabilizzati. Questa modalità è stata implementata per prove semplificate e per verificare il corretto funzionamento della trasformazione geometrica senza utilizzare il contorno denso.
Un altro aspetto importante è l'orientamento del tool rispetto al piano di lavoro della vaschetta, infatti, prima di eseguire la traiettoria, il software modifica l'orientamento del TCP per rendere la punta di erogazione coerente con la direzione di deposizione. Questa operazione è gestita dalla funzione:

compute_ready_pose()

La funzione esegue inizialmente una rotazione del sesto giunto del robot di $+90°$:

rotated_joints[5] += math.pi / 2

che consente al tool di portarsi in una configurazione più favorevole all'esecuzione della traiettoria. Successivamente, a partire dalla posa TCP raggiunta letta tramite la funzione rtde_r.getActualTCPPose(), il software costruisce un nuovo orientamento target imponendo che l'asse Z del tool sia diretto verso il basso, ovvero nella stessa direzione dell'asse Z della base robot ma con verso opposto.

$$z_{tool,target} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$$

In questo modo la punta di erogazione viene orientata in modo coerente con la deposizione del materiale sul piano di lavoro.

L'asse X del tool viene invece ottenuto proiettando sul piano XY l'asse X attuale, in modo da mantenere una direzione coerente con la configurazione raggiunta dopo la rotazione del sesto giunto. L'asse Y viene calcolato come prodotto vettoriale tra Z e X, così da ottenere una terna ortonormale.
La matrice di rotazione target viene quindi costruita nella forma:

$$R_{target} = [x_{tool} \quad y_{tool} \quad z_{tool}]$$

e convertita in vettore di rotazione sempre tramite la funzione cv2.Rodrigues(R_target).
Infine, il software calcola la configurazione articolare corrispondente alla posa target mediante cinematica inversa:

rtde_c.getInverseKinematics(ready_tcp, rotated_joints)

e invia al robot un comando moveJ per raggiungere la configurazione pronta all'esecuzione della traiettoria.
Questa fase è importante perché permette di separare il problema della posizione dei punti, ricavata dalla camera, dal problema dell'orientamento dell'utensile, che deve essere coerente con la deposizione del materiale.

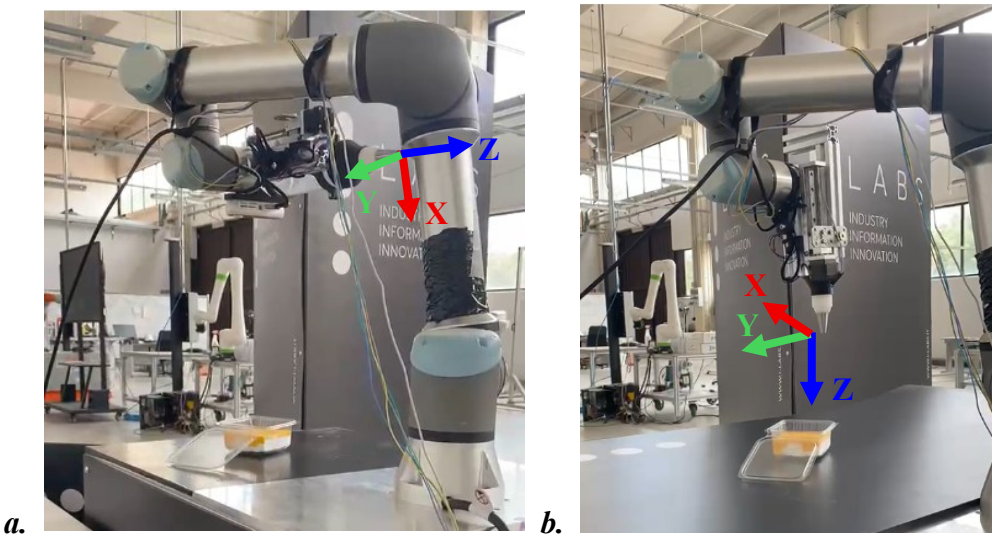


*Figura 5.7 : Rappresentazione dell'orientamento del TCP prima (**a.**) e dopo (**b.**) la rotazione*

La traiettoria eseguita dal robot non consiste in un semplice collegamento lineare tra i punti del contorno. Per ottenere un movimento più adatto alla realizzazione dei ciuffi di maionese, il software introduce dei via-point intermedi tra un punto del contorno e il successivo. Questa logica è implementata nella funzione:

compute_arc_via()

Per ogni coppia di punti consecutivi P_i e P_{i+1}, viene calcolato un punto intermedio partendo dal punto medio del segmento:

$$P_{mid} = \frac{P_i + P_{i+1}}{2}$$

Tale punto viene poi spostato verso il centro della vaschetta secondo il parametro ARC_CENTER_PULL = 0.65 e sollevato verticalmente di ARC_LIFT_M = 0.02 m. Il via-point risultante può essere interpretato come:

$$P_{via} = P_{mid} + \lambda(P_c - P_{mid}) + \Delta z$$

dove P_c è il centro della vaschetta, $\lambda = 0.65$ è il coefficiente di richiamo verso il centro e $\boldsymbol{\Delta}$z è il sollevamento verticale.

In questo modo, tra due punti consecutivi del contorno, il robot non procede direttamente lungo una linea retta sul bordo, ma esegue una traiettoria composta da due tratti: un primo movimento dal punto corrente al via-point sollevato e spostato verso il centro e un secondo movimento dal via-point al punto successivo del contorno. Nel codice, questi due tratti vengono eseguiti tramite due comandi lineari:

rtde_c.moveL(make_waypoint(*via_xyz, ...), MOVE_SPEED, MOVE_ACCEL)

rtde_c.moveL(make_waypoint(*p_next_z, ...), MOVE_SPEED, MOVE_ACCEL)

Questa strategia genera una traiettoria alternata tra bordo e zona interna della vaschetta, consentendo al tool di realizzare la decorazione, ovvero di creare ciuffi con punte rivolte verso il centro. La sequenza completa di movimento è gestita dalla funzione:

```
move_to_box_and_trace(coord_center, contour_3d_points)
```

Essa prevede le seguenti fasi:

- calcolo della posa pronta tramite compute_ready_pose();
- movimento lineare verso il primo punto del contorno;
- attivazione del sistema di erogazione tramite comando seriale;
- percorrenza del contorno denso con via-point intermedi;
- esecuzione di un via-point finale verso il centro;
- ritorno alla posa di osservazione tramite moveJ.

Durante la traiettoria, il software invia anche comandi seriali al sistema di attuazione dell'asse lineare. In particolare, dopo il raggiungimento del primo punto viene inviato:

```
ser.write(b"s 6 670")
```

mentre durante la percorrenza dei segmenti viene inviato:

```
ser.write(b"s 5 670")
```

Questi comandi permettono di sincronizzare il movimento del robot con l'avanzamento dell'asse lineare che spinge lo stantuffo della siringa. La descrizione dettagliata del protocollo seriale e della logica di comando dell'asse lineare può essere approfondita nel capitolo dedicato alla comunicazione e al controllo, mentre in questa sede è sufficiente evidenziare che l'erogazione viene coordinata con la traiettoria generata dal sistema di visione.

Il software comprende anche una sezione dedicata alla visualizzazione dei risultati, implementata nella funzione:

```
draw_results()
```

Questa funzione sovrappone all'immagine RGB le informazioni estratte dalla pipeline di visione: contorno rilevato, vertici stabilizzati, punti del contorno denso, centro della vaschetta e coordinate 3D calcolate nel sistema base robot. Viene inoltre generato un pannello laterale contenente informazioni numeriche sui punti individuati, sul numero di waypoint generati e sui parametri principali utilizzati per la traiettoria.

L'interfaccia grafica, *Figura 5.8* , consente quindi di verificare in tempo reale il comportamento del sistema prima dell'esecuzione del movimento. In particolare, l'operatore può controllare se la vaschetta è stata correttamente individuata, se il centro è coerente con la posizione reale dell'oggetto, se i punti del contorno sono distribuiti correttamente e se sono disponibili misure di profondità valide.

Sono inoltre previsti alcuni comandi da tastiera:

- SPAZIO: genera ed esegue la traiettoria sul contorno denso;
- V: esegue la modalità semplificata basata sui quattro vertici;
- S: salva uno snapshot dell'interfaccia;
- D: stampa informazioni di debug sulla profondità e sulle trasformazioni geometriche;
- Q: termina il programma.

Il comando di debug D è particolarmente utile durante la fase sperimentale, poiché permette di confrontare il punto espresso nel sistema camera, nel sistema TCP e nel sistema base robot. Il software stampa, inoltre, la direzione dell'asse Zdella camera nel sistema base e confronta la posizione calcolata tramite pipeline con una verifica geometrica basata sull'origine della camera e sulla direzione dell'asse ottico. Questo consente di individuare eventuali errori nella calibrazione, nell'allineamento depth-color o nella definizione della posa di osservazione.

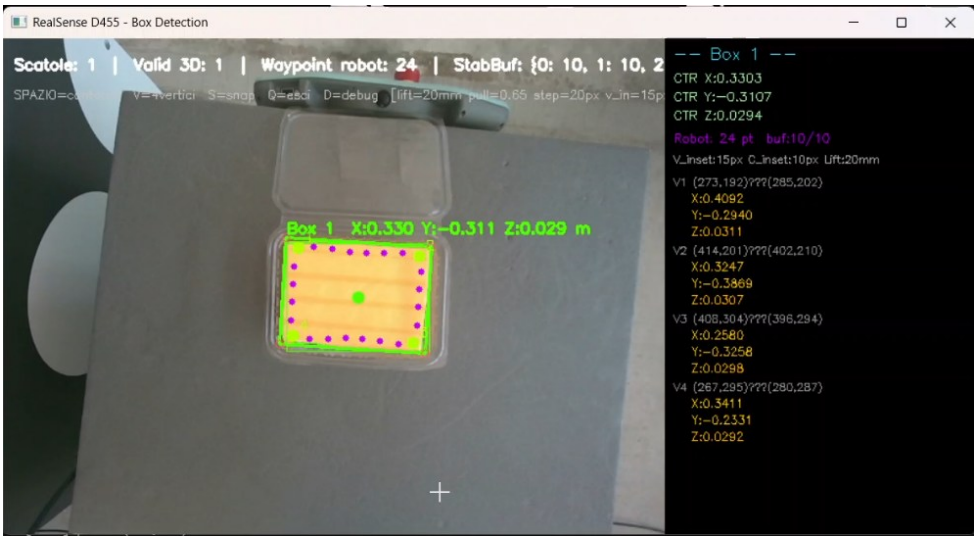


Figura 5.8 : Screenshot dell'interfaccia software con contorno rilevato, punti della traiettoria e pannello delle coordinate

Nel complesso, questo capitolo rappresenta il collegamento operativo tra la calibrazione robot-camera e l'esecuzione fisica della decorazione. La calibrazione permette di trasformare i punti camera nel sistema robot; la pipeline di visione individua tali punti sulla vaschetta; la generazione della traiettoria li converte in waypoint eseguibili dal robot UR5e e sincronizzabili con il sistema di erogazione dell'asse lineare.

6. ARCHITETTURA DI CONTROLLO INTEGRATO

I capitoli precedenti hanno descritto i sottosistemi della cella dal punto di vista costruttivo e funzionale e, nel Capitolo 5, il programma supervisore che ne coordina il funzionamento realizzando l'acquisizione delle immagini, la localizzazione del vassoio e la generazione della traiettoria. Il presente capitolo approfondisce i due elementi software e di comunicazione su cui tale integrazione si appoggia: da un lato il firmware di controllo dell'asse lineare, ossia la parte software dell'elettronica di dosaggio; dall'altro i protocolli di comunicazione che collegano il PC ai diversi sottosistemi e il robot all'asse.

Il controllo della cella è organizzato su due livelli: quello superiore che è il programma supervisore sul PC e quello inferiore che è il firmware embedded della scheda Arduino che riceve i comandi relativi all'asse e li traduce nel movimento del motore passo-passo. I due livelli dialogano, insieme agli altri sottosistemi, attraverso i canali di comunicazione successivamente descritti.

6.1. FIRMWARE DI CONTROLLO DELL'ASSE LINEARE

Il firmware che governa l'asse lineare costituisce la parte software dell'elettronica di dosaggio. È scritto in linguaggio C++ nell'ambiente di sviluppo PlatformIO, configurato per la scheda Arduino Uno R4 Minima. Esso converte i comandi ricevuti, dal PC via seriale o dal robot tramite segnali digitali, nel movimento controllato del motore. Il codice sorgente completo, riportato in APPENDICE C, è organizzato in moduli, ciascuno con una responsabilità definita come riassunto nella *Tabella 6.1*.

Modulo	**File**	**Responsabilità**
Configurazione	*config.h*	Costanti hardware (pin), parametri meccanici, limiti di velocità e indirizzi EEPROM.
Movimento	*axis.cpp / .h*	Generazione degli impulsi, rampe di accelerazione, movimenti assoluti e relativi, spinta.

Modulo	File	Responsabilità
Segnali	*signals.cpp / .h*	Lettura degli ingressi digitali dal robot, con gestione della logica attiva alta o bassa.
Comandi seriali	*serial_cmd.cpp / .h*	Parser dei comandi da PC e menu interattivo di setup e manutenzione.
Persistenza	*settings.cpp / .h*	Salvataggio e ripristino di home, velocità e posizione in EEPROM.
Programma principale	*main.cpp*	Funzioni setup() e loop(): priorità dei comandi e gestione dei segnali.

Tabella 6.1 : Moduli del firmware e relative responsabilità

Il legame tra il dominio digitale (impulsi) e quello fisico (millimetri) è definito da poche costanti raccolte in *config.h*. Il passo della vite è SCREW_LEAD_MM = 4,0 mm/giro e il driver è impostato a DRIVER_PPR = 400 impulsi/giro (microstepping 1/2), da cui deriva la risoluzione STEPS_PER_MM = 400/4 = 100 impulsi/mm, pari a 0,01 mm per impulso. La corsa utile è limitata via software a AXIS_MAX_MM = 100 mm. Tale risoluzione, già ricavata nel Capitolo 3 dal punto di vista meccanico, è qui codificata come parametro del firmware. Sempre in *config.h* sono inoltre definiti i parametri di temporizzazione e di rampa. La larghezza dell'impulso di STEP è STEP_PULSE_US = 5 µs. Il ritardo tra impulsi consecutivi è vincolato tra un valore minimo MIN_DELAY_US = 200 µs e un valore massimo MAX_DELAY_US = 10000 µs: poiché un ritardo minore corrisponde a una frequenza di impulsi maggiore, il primo limite fissa la velocità massima consigliata (circa 50 mm/s) e il secondo la velocità minima (circa 1 mm/s). Sono inoltre presenti i parametri della rampa di velocità e gli indirizzi EEPROM in cui vengono memorizzati home, velocità e posizione, ciascuno come numero in virgola mobile a 4 byte. In assenza di valori validi in memoria, il firmware adotta i valori di default: home a 0 mm, velocità 30 mm/s. A titolo di esempio, per spostarsi da 0 a 50 mm il firmware genera 50 × 100 = 5000 impulsi; alla velocità massima, con un impulso ogni 200 µs, il movimento dura circa un secondo.

Il modulo di movimento espone alcune funzioni elementari: il movimento assoluto verso una quota (*axisMoveTo*), il movimento relativo di una distanza (*axisMoveRel*), la spinta in avanti

finché un segnale resta attivo (*axisPushBySignal*) e la spinta in avanti per una durata prefissata (*axisPushByTime*). Tutti i movimenti sono vincolati all'intervallo di corsa [0, 100] mm.
Il firmware non genera impulsi a velocità costante, ma applica una rampa trapezoidale di velocità: all'avvio gli impulsi sono più distanziati (fase di accelerazione), raggiungono quindi la velocità di crociera e infine rallentano prima dell'arresto (fase di decelerazione). Questo profilo riduce gli strappi meccanici e il rischio di perdita di passi, aspetto particolarmente importante quando si spinge un fluido viscoso come la maionese. La conversione tra velocità e temporizzazione è affidata alla funzione *axisSpeedToDelay*, che traduce la velocità in mm/s nel ritardo tra impulsi consecutivi (in µs), saturandolo tra MIN_DELAY_US e MAX_DELAY_US. Il numero di passi dedicati all'accelerazione e alla decelerazione è pari a circa un quinto dei passi totali, comunque limitato tra 10 e 200, mentre il ritardo iniziale è un multiplo di quello di regime.

Ogni singolo passo è generato attraverso una funzione interna (*doOneStep*) che applica un limite software sulla corsa: il movimento viene bloccato se la posizione scenderebbe al di sotto di 0 mm o supererebbe i 100 mm, estremi della corsa utile. In questo modo l'asse resta sempre confinato nell'intervallo ammesso [0, 100] mm. Dopo ogni movimento la posizione aggiornata viene salvata in memoria.
La persistenza è gestita dal modulo dedicato, che memorizza in EEPROM tre valori: la posizione di home, la velocità di default e la posizione attuale. Al riavvio del sistema la posizione viene ripristinata, così che la configurazione sia conservata anche dopo lo spegnimento. I valori letti vengono validati, ovvero viene effettuato un controllo dell'intervallo ammesso ed un'eventuale esclusione dei valori non numerici, prima di essere accettati, evitando comportamenti anomali nel caso in cui la EEPROM non sia ancora stata inizializzata.

Il programma principale stabilisce, a ogni iterazione del ciclo, una gerarchia di priorità tra le sorgenti di comando. I comandi seriali provenienti dal PC vengono valutati per primi. Se è aperto il menu di setup, la modalità di manutenzione, i segnali del robot vengono completamente ignorati, a garanzia della sicurezza durante la configurazione manuale. In assenza di tali condizioni vengono valutati i segnali digitali del robot, nell'ordine indicato in *Tabella 6.2*. Questa gerarchia assicura che l'intervento dell'operatore da PC abbia sempre la precedenza sul comando automatico del robot.

Pin	Nome firmware	Funzione	Comportamento
D8	SIG_PUSH_FWD	Spinta avanti	Avanza con rampa finché il segnale resta alto.
D9	SIG_PULL_BWD	Ritorno indietro	Arretra passo-passo finché il segnale resta alto, fermandosi a 0 mm (limite software).
D11	SIG_GOTO_ZERO	Ritorno a 0 mm	Ritorno automatico a 0 mm con rampa.

Tabella 6.2 : Segnali digitali del robot e relativa funzione nel firmware

Il firmware accetta due tipologie di comando provenienti dal PC tramite la porta seriale: i comandi rapidi, destinati al funzionamento automatico, e i comandi del menu di setup, riservati alla configurazione e alla manutenzione. I comandi sono testuali, non distinguono tra maiuscole e minuscole, sono terminati dal carattere di newline e, quando previsto, accettano uno o più parametri numerici separati da spazi. Ad ogni comando l'Arduino risponde sulla seriale con un messaggio di stato.

I comandi rapidi sono quelli impiegati durante il ciclo automatico, anche dal programma supervisore. Comprendono la spinta a tempo, eventualmente seguita da un ritorno, e i posizionamenti notevoli. Mentre quelli del menu di setup costituisce la modalità di manutenzione: finché è aperto i segnali del robot sono ignorati. Da esso è possibile configurare i parametri persistenti (salvati in EEPROM), muovere manualmente l'asse e forzare la posizione logica. Il menu distingue la velocità di default, persistente, dalla velocità di sessione, valida solo fino all'uscita; in assenza di comandi per cinque minuti il menu si chiude automaticamente. La *Tabella 6.3* ne riporta sintassi, ingressi e descrizione:

Comando	Ingressi	Descrizione
S <vel> <ms>	vel: velocità [mm/s]; ms: durata [ms]	Spinta in avanti per la durata indicata alla velocità data; al termine salva la posizione. Parametri validati (vel > 0, ms ≠ 0).

Comando	Ingressi	Descrizione	
SR <vel> <ms> <mm>	vel [mm/s]; ms [ms]; mm: rientro [mm]	Esegue la spinta a tempo come S e quindi un ritorno relativo di mm verso lo zero (limitato a 0 mm).	COMANDI RAPIDI
Z (o zero)	—	Movimento assoluto a 0 mm con rampa.	
F (o fc)	—	Movimento assoluto al finecorsa (100 mm).	
HOME	—	Movimento assoluto alla posizione di home configurata.	
pos	—	Stampa sulla seriale la posizione logica attuale.	
setup	—	Apre il menu di setup e manutenzione (sospende i segnali del robot).	
help (o ?)	—	Elenca i comandi disponibili.	
sethome <mm>	mm ∈ [0, 100]	Imposta la posizione di home e la salva in EEPROM.	COMANDI SETUP
setspeed <v>	v ∈ (0, 200] [mm/s]	Imposta la velocità di default e la salva in EEPROM.	
info	—	Mostra le impostazioni correnti (home, velocità, posizione) e la velocità di sessione.	
goto <mm>	mm ∈ [0, 100]	Movimento assoluto alla quota indicata.	
move <mm>	mm (anche negativo)	Movimento relativo, limitato all'intervallo [0, 100] mm.	
home	—	Va alla posizione di home.	
zero	—	Va a 0 mm.	

Comando	Ingressi	Descrizione
fc	—	Va al finecorsa (100 mm).
pos	—	Mostra la posizione attuale.
setpos <mm>	mm ∈ [0, 100]	Forza la posizione logica senza muovere l'asse (riallineamento) e la salva in EEPROM.
speed <v>	v ∈ (0, 200] [mm/s]	Imposta la velocità di sessione (non salvata).
help (o ?)	—	Mostra l'elenco dei comandi del menu.
fine	—	Esce dal setup e riattiva i segnali del robot.

Tabella 6.3: Comandi rapidi e comandi del menu di setup e manutenzione

A seconda della fase di lavoro, il firmware viene impiegato in due modi distinti. Durante il normale ciclo di dosaggio funziona in modalità automatica: riceve i comandi rapidi dal programma supervisore, oppure i segnali digitali direttamente dal robot, ed esegue le spinte e i posizionamenti senza che sia richiesto alcun intervento da parte dell'operatore. Accanto a questa modalità ne è prevista una seconda, pensata per la configurazione e la manutenzione del sistema, che si basa su un menu di setup interattivo.

Il menu di setup viene utilizzato attraverso la porta seriale e non richiede alcun software dedicato. Poiché la comunicazione avviene tramite semplice scambio di testo, è infatti sufficiente un comune emulatore di terminale, come PuTTY, collegato alla scheda Arduino alla velocità di 115200 baud. Una volta aperto il menu, l'operatore può digitare i comandi descritti in precedenza e dialogare direttamente con il firmware: in questo modo è possibile modificare e salvare in EEPROM i parametri persistenti, come la posizione di home e la velocità di default, muovere manualmente l'asse per eseguire prove o riallineamenti, forzare la posizione logica e interrogare lo stato corrente del sistema.

Il vantaggio di questa soluzione è che tutti i settaggi possono essere modificati in modo semplice e immediato, agendo da terminale senza dover ricompilare il firmware né riprogrammare la scheda. Questa caratteristica si è rivelata particolarmente comoda nelle fasi

di messa a punto e di diagnostica, durante le quali è spesso necessario variare rapidamente i parametri di funzionamento e verificarne subito l'effetto sull'asse.

6.2. PROTOCOLLI DI COMUNICAZIONE

L'integrazione dei sottosistemi si appoggia su quattro canali di comunicazione. Il programma supervisore sul PC dialoga con il robot tramite il protocollo RTDE su rete Ethernet, con la camera tramite collegamento USB e l'SDK Intel RealSense, e con l'asse lineare tramite porta seriale; a questi si aggiunge il collegamento diretto tra robot e asse, realizzato mediante segnali digitali. I canali gestiti dal PC sono inizializzati all'avvio del programma supervisore e mantenuti attivi per l'intera sessione di lavoro.

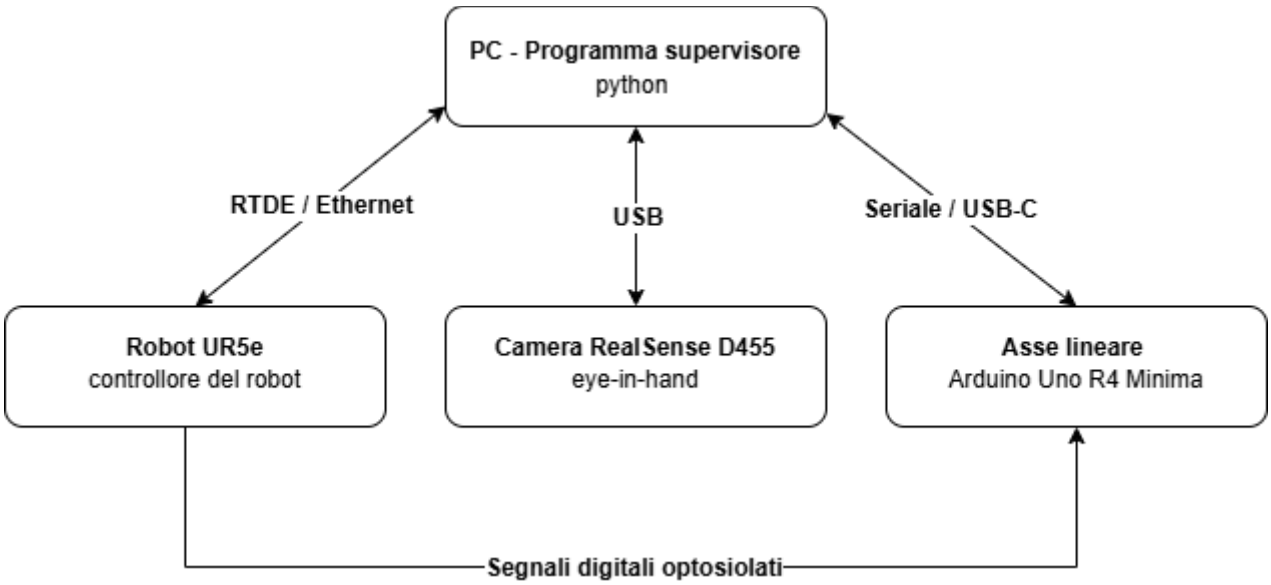


Figura 6.1 : Protocolli di comunicazione della cella robotizzata

La comunicazione tra il PC e il robot avviene tramite il protocollo RTDE (Real-Time Data Exchange) di Universal Robots. RTDE è un protocollo basato su TCP/IP che opera su una porta dedicata del controllore (30004) e consente uno scambio dati bidirezionale con la logica di controllo del robot senza alterarne le proprietà di tempo reale. Lo scambio si articola in una procedura di setup, durante la quale il client definisce le variabili da sincronizzare, e in un ciclo di sincronizzazione, in cui i dati vengono trasferiti periodicamente. Sui robot e-Series, la frequenza di scambio può raggiungere i 500 Hz, pari alla frequenza del ciclo di controllo in tempo reale del robot.

Il protocollo distingue i dati in uscita, output, inviati dal robot all'applicazione, quali posizioni di giunto, posa dell'utensile e stato degli I/O, dai dati in ingresso, input, inviati dall'applicazione al robot. L'accesso è realizzato mediante la libreria *ur_rtde*, che mette a disposizione interfacce distinte per il comando dei movimenti e per la lettura dello stato, istanziate specificando l'indirizzo IP del controllore. L'impiego applicativo di tali interfacce nella movimentazione e nella generazione della traiettoria è descritto nel Capitolo 5.

La camera Intel RealSense D455 è collegata al PC tramite cavo USB. Il dialogo con il dispositivo è mediato dall'SDK Intel RealSense (*librealsense*), reso disponibile in ambiente Python attraverso la libreria *pyrealsense2*. L'SDK gestisce l'acquisizione in streaming dei flussi a colori e di profondità, l'accesso ai parametri intrinseci del sensore e le operazioni di allineamento e deproiezione necessarie a passare dalle coordinate immagine a quelle tridimensionali. A differenza degli altri canali, che trasportano comandi e stati, questo collegamento veicola un flusso continuo di dati di immagine.

Il comando dell'asse lineare dal PC avviene attraverso un'interfaccia seriale. La scheda Arduino Uno R4 Minima, collegata tramite cavo USB-C, è riconosciuta dal sistema operativo come porta seriale virtuale; sullo stesso cavo viaggiano sia l'alimentazione della logica sia la comunicazione, alla velocità di 115200 baud. Lato PC la comunicazione è gestita in Python mediante la libreria pyserial. Su questo canale il supervisore invia al firmware i comandi testuali descritti in precedenza, sia i comandi rapidi per il funzionamento automatico, sia quelli del menu di setup, e riceve i messaggi di stato emessi dall'Arduino. Il canale seriale trasporta comandi a livello logico: la generazione delle correnti di fase del motore è demandata al driver, mentre la temporizzazione degli impulsi è gestita internamente dal firmware.

Oltre al comando dal PC, l'asse può essere comandato direttamente dal robot. A tale scopo si prendono in considerazione tre uscite digitali libere del robot a 24 V collegate agli ingressi D8–D11 dell'Arduino attraverso la scheda di optoisolatori descritta nel Capitolo 3, che ne realizza l'adattamento di livello e l'isolamento galvanico. La funzione associata a ciascun segnale è quella già riportata in Tabella 6.3. A differenza del canale seriale, che veicola comandi testuali, questo collegamento trasferisce comandi mediante il semplice stato logico (alto/basso) delle linee. La gerarchia con cui il firmware gestisce le due sorgenti di comando,

PC e robot, rispetta quella annunciata in precedenza, ovvero i segnali da PC hanno la precedenza su quelli del robot.

Nel loro insieme, il firmware di controllo dell'asse e i protocolli di comunicazione descritti costituiscono l'infrastruttura su cui il programma supervisore si appoggia per coordinare il funzionamento della cella. Le prestazioni del sistema integrato sono valutate nel capitolo successivo.

7. TEST SPERIMENTALI E RISULTATI

I capitoli precedenti hanno descritto la cella robotizzata nei suoi sottosistemi costruttivi e funzionali: nel Capitolo 5, la generazione della traiettoria a partire dai dati di visione e, nel Capitolo 6, l'architettura di controllo che coordina il programma supervisore, il robot e l'asse lineare di dosaggio. Il presente capitolo riporta la campagna sperimentale condotta sulla cella completa, con l'obiettivo di verificare la capacità del sistema di realizzare la decorazione automatica della vaschetta di insalata russa mediante la deposizione di ciuffi di maionese lungo il bordo del prodotto. Vengono descritte le criticità riscontrate nella fase iniziale, le soluzioni adottate sul sistema di erogazione, la taratura empirica dei parametri di processo, la valutazione del tempo ciclo e i risultati conseguiti nella configurazione finale.

L'obiettivo della sperimentazione è la valutazione del comportamento complessivo della cella nell'esecuzione del compito decorativo, con particolare attenzione alla qualità e all'uniformità dei ciuffi depositati lungo il perimetro della vaschetta. Per ciuffo si intende il deposito di maionese caratterizzato da base regolare e punta rivolta verso l'alto, ottenuto secondo la logica di traiettoria con via-point sollevato e tirato verso il centro descritta nel Capitolo 5. Si considera soddisfacente un risultato in cui i ciuffi presentino dimensioni omogenee, spaziatura regolare e assenza di filamenti di coda o di fusioni tra depositi adiacenti.

L'esecuzione delle prove direttamente sull'insalata russa non è risultata praticabile: il prodotto reale è deperibile, non riproducibile in modo identico tra una prova e la successiva e poco adatto a una campagna iterativa che richiede numerose ripetizioni a parità di condizioni geometriche. Si è quindi scelto di semplificare il bersaglio di prova realizzando un provino rappresentativo. Il provino è costituito dalla stessa vaschetta trasparente impiegata per il confezionamento del prodotto, riempita con un blocco di polistirolo sagomato in modo da occuparne il volume interno e ricoperto superiormente da uno strato di colore arancione, così da riprodurre l'aspetto visivo della superficie dell'insalata russa. Questa soluzione offre una superficie rigida, planare e stabile su cui depositare i ciuffi a quota costante, garantisce la riutilizzabilità del bersaglio tra prove successive e mantiene sia la geometria reale della vaschetta sia il contrasto cromatico tra la superficie interna e il bordo. Sono state inoltre impiegate tre vaschette di dimensioni diverse (*Figura 7.1*), allo scopo di verificare la capacità del sistema di adattarsi automaticamente a formati differenti, coerentemente con la generazione automatica del contorno.

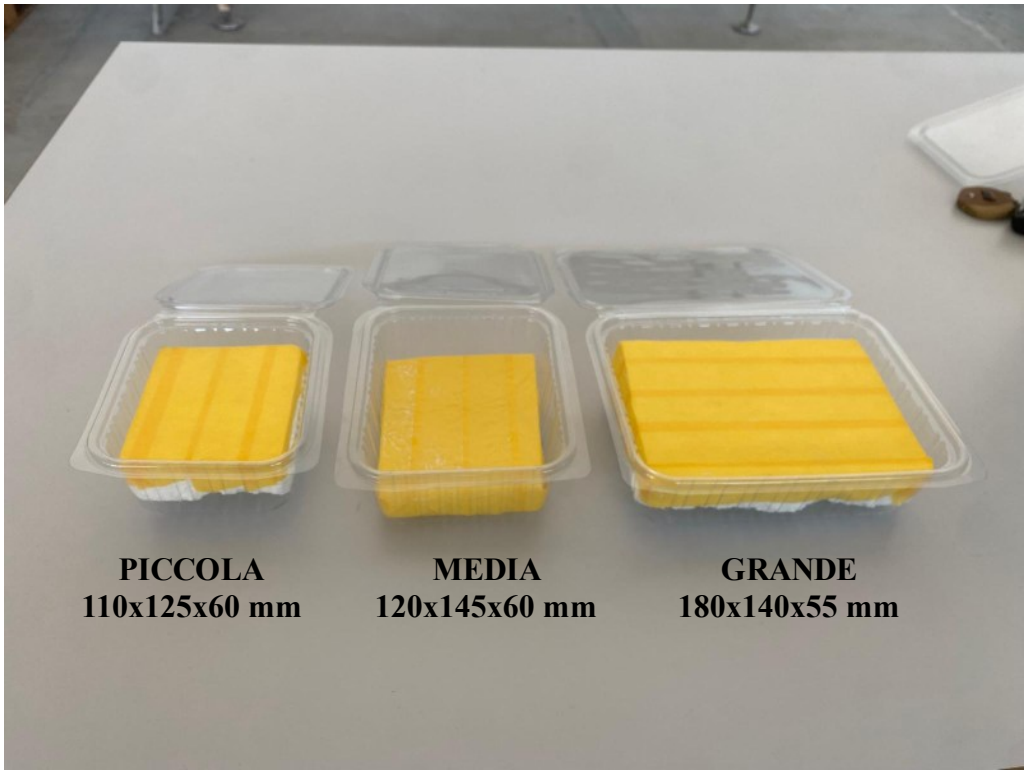


Figura 7.1: Vaschette di diverse dimensioni utilizzate nei test sperimentali

La valutazione adottata in questa campagna è di natura qualitativa e visiva, in coerenza con la finalità del lavoro: uno studio di fattibilità volto a verificare se il processo sia realizzabile e produca una decorazione accettabile in termini di uniformità, spaziatura e forma dei ciuffi. Non si è invece fatto ricorso a una misura quantitativa sistematica della massa per singolo ciuffo, sia perché non praticabile nel setup adottato, sia perché le principali grandezze che la influenzano, ovvero la temperatura e la consistenza della maionese, non erano controllabili in modo rigoroso con l'attrezzatura di laboratorio disponibile. Per queste ragioni le prove sono state condotte in forma iterativa, modificando un parametro alla volta e osservandone l'effetto sul deposito, sia su supporti di prova sia sul provino.

Nella fase iniziale della sperimentazione il sistema di dosaggio non consentiva un'erogazione controllata e ripetibile della maionese. L'analisi dei depositi ottenuti ha permesso di individuare due cause distinte, entrambe riconducibili al sistema di erogazione e non alla logica di movimentazione del robot.

La prima criticità è legata all'estrusore inizialmente adottato, la cui geometria dell'ugello non garantiva un distacco netto del materiale al termine della spinta. Al termine di ogni erogazione la maionese tendeva a permanere in adesione alla punta, generando filamenti di coda e depositi dalla

forma irregolare. Per tentare di ottenere un distacco più pulito e cercare di interrompere il cordone di maionese in modo netto, si è reso necessario ricorrere al comando di risucchio "*sr 20 1000 10"* con velocità dell'asse di 20 mm/s, durata della spinta di 1000 ms e risucchio di 10 mm. Tale accorgimento, pur mitigando in parte il fenomeno, non ha consentito di ottenere depositi regolari.
La seconda criticità è di natura meccanica e riguarda lo stantuffo della siringa. Nelle prime configurazioni lo stantuffo conservava la guarnizione originale della siringa: durante l'avanzamento essa generava un attrito elevato contro la parete del cilindro e, a monte, una depressione che ostacolava lo scorrimento. L'effetto combinato di attrito e vuoto impediva un avanzamento regolare dello stantuffo e, di conseguenza, una fuoriuscita costante della maionese. Per vincere questa resistenza è stato necessario operare con velocità di spinta molto elevate e con tempi di spinta molto lunghi, ben superiori a quelli successivamente impiegati nella configurazione finale.
Il risultato di queste condizioni operative è documentato dalle prove di deposizione su carta riportate nella *Figura 7.2*, in cui i depositi presentano dimensioni e forme fortemente disomogenee, con filamenti, code laterali e fusioni tra ciuffi contigui.

Figura 7.2: Prove di deposizione su carta nella configurazione iniziale: depositi di dimensione e forma irregolari

L'individuazione delle due cause descritte ha indirizzato la messa a punto del sistema di erogazione. Da un lato si è proceduto alla sostituzione dell'estrusore con un ugello tipico della sac à poche, in grado di assicurare un distacco netto del materiale al termine della spinta; dall'altro è stata eliminata la guarnizione responsabile dell'attrito e della depressione, ripristinando uno scorrimento regolare dello stantuffo.

La rimozione delle due criticità ha avuto un effetto immediato sul comportamento del dosaggio. Venuta meno la resistenza all'avanzamento, è stato possibile abbandonare le velocità e i tempi di spinta elevati imposti dalla configurazione iniziale e operare con valori molto più contenuti e controllabili, ottenendo finalmente un'erogazione ripetibile. Inoltre, grazie al distacco netto garantito dal nuovo ugello, il comando di risucchio "*sr <vel> <ms> <mm>"* non è più risultato necessario: nella configurazione finale la deposizione di ciascun ciuffo è ottenuta con la sola spinta a tempo *"s <vel> <ms>"*, senza rientro dell'asse. Il deposito così ottenuto presenta base regolare e punta ben definita, coerente con la forma di riferimento riportata in *Figura 7.3*.

Figura 7.3: Ciuffo con base regolare e punta ben definita

Stabilizzato il sistema di erogazione, la fase successiva ha riguardato la taratura dei parametri che concorrono alla formazione del ciuffo e la sincronizzazione tra il moto del robot e l'estrusione della maionese. I parametri coinvolti appartengono a due gruppi distinti, di cui il primo gruppo riguarda la geometria della traiettoria e determina la forma del ciuffo. Esso comprende il sollevamento del via-point rispetto al piano del contorno, definito da ARC_LIFT_M, che governa l'altezza alla quale la punta di erogazione si solleva per tirare verso l'alto la punta del ciuffo; lo spostamento del via-point verso il centro della vaschetta, definito da ARC_CENTER_PULL, che orienta la formazione della punta verso l'interno del prodotto; e l'offset verticale di erogazione Z_CONTOUR_OFFSET_M che fissa la quota della punta rispetto ai punti del contorno. Il secondo gruppo riguarda il dosaggio ed è costituito dalla velocità e dalla durata della spinta dell'asse lineare, trasmesse al firmware tramite il comando seriale "*s <vel> <ms>"*.
La taratura ha portato a distinguere il primo ciuffo dai successivi. Il primo deposito viene realizzato con una spinta leggermente maggiore, in modo da compensare il riempimento iniziale

dell'ugello, mentre i ciuffi successivi adottano una spinta lievemente ridotta. I valori adottati nella configurazione finale sono riassunti in *Tabella 7.1*.

Parametro	**Valore adottato**	**Funzione**
Z_CONTOUR_OFFSET_M	17 mm	Quota di erogazione sopra il punto di contorno
ARC_LIFT_M	20 mm	Sollevamento del via-point per la formazione della punta
ARC_CENTER_PULL	0,65	Spostamento del via-point verso il centro (65 %)
Comando primo ciuffo	*"s 6 670"*	Spinta a 6 mm/s per 670 ms
Comando ciuffi successivi	*"s 5 670"*	Spinta a 5 mm/s per 670 ms
Tempo di attesa (primo ciuffo)	2 s	Sincronizzazione ed estrusione del materiale
Tempo di attesa (ciuffi successivi)	1 s	Sincronizzazione ed estrusione del materiale
SERIAL_INTERVAL	100 ms	Intervallo minimo tra impulsi seriali consecutivi

Tabella 7.1: Parametri di processo adottati nella configurazione finale

Un aspetto determinante per la qualità del risultato è la sincronizzazione tra il movimento del robot e l'erogazione della maionese. Nel programma supervisore, i comandi di movimento *moveL* e i comandi di spinta inviati sulla porta seriale sono eseguiti in successione all'interno dello stesso flusso. Il comando *moveL*, tuttavia, è di tipo bloccante: il programma resta in attesa del suo completamento e solo al termine del movimento procede all'istruzione successiva. Di conseguenza, la *ser.write* posta dopo il movimento non verrebbe eseguita in modo simultaneo all'arrivo del robot nel punto, ma con un ritardo che disallinea la spinta dell'asse rispetto al moto, compromettendo la corretta formazione del ciuffo.

Per ovviare a questo problema, dopo l'invio del comando seriale viene introdotto un intervallo di attesa, *time.sleep()*, durante il quale il programma resta sospeso e il robot mantiene la posizione raggiunta. Questa attesa assolve a due funzioni complementari: concede alla spinta seriale il

tempo di essere effettivamente attivata ed eseguita prima che il flusso prosegua con il movimento successivo e garantisce anche il tempo necessario per una corretta estrusione del materiale, evitando che il robot riprenda a muoversi prima del completamento dell'erogazione. L'attesa assicura quindi che la spinta avvenga mentre il robot si trova nella posizione prevista e non durante lo spostamento.
Coerentemente con la distinzione tra primo ciuffo e ciuffi successivi, la durata dell'attesa è maggiore per il primo deposito e ridotta per quelli seguenti. La sequenza che ne risulta, ripetuta lungo l'intero contorno, alterna quindi raggiungimento della posizione, invio della spinta a tempo e attesa di esecuzione ed estrusione, assicurando la corretta coordinazione tra movimentazione ed erogazione lungo tutto il percorso.

Individuata la combinazione corretta tra velocità dell'asse lineare e durata della spinta, la sperimentazione si è concentrata sulla valutazione del tempo ciclo dell'operazione, inteso come il tempo necessario a completare la decorazione di una vaschetta. Durante la fase di taratura del dosaggio la velocità e l'accelerazione di movimentazione del robot erano state mantenute su valori conservativi, pari a 0,2 m/s, allo scopo di privilegiare la stabilità e la ripetibilità del deposito; una volta consolidata la qualità dei ciuffi, tali valori sono stati progressivamente incrementati per ridurre la durata complessiva del ciclo fino ad un valore di 0,5 m/s.
L'incremento della velocità del robot riduce il tempo impiegato nei trasferimenti tra un punto di deposizione e il successivo, ma non agisce sulle pause associate a ciascun ciuffo. Il tempo ciclo, infatti, è dominato dalle fasi di spinta e di attesa descritte nel paragrafo precedente, che costituiscono un limite inferiore indipendente dalla velocità di movimentazione: oltre una certa soglia, l'ulteriore aumento della velocità del robot non produce una riduzione apprezzabile del tempo ciclo, poiché il collo di bottiglia diventa la deposizione del materiale.
Poiché il numero di ciuffi, e quindi di pause di deposizione, cresce con il perimetro della vaschetta, il tempo ciclo aumenta con le dimensioni del formato. La valutazione è stata pertanto ripetuta sulle tre vaschette di dimensioni diverse: con la velocità conservativa iniziale il tempo ciclo era dell'ordine di alcune decine di secondi per il formato maggiore, ridottosi sensibilmente con la velocità di movimentazione incrementata fino al valore limite imposto dalla somma delle pause di erogazione. I tempi ciclo misurati nella configurazione finale sono riassunti in *Tabella 7.2.*

Formato vaschetta	Numero di ciuffi	Velocità robot iniziale (0,2 m/s)	Velocità robot incrementata (0,5 m/s)
Piccolo	16	≈ 51 s	≈ 30 s
Medio	18	≈ 58 s	≈ 35 s
Grande	24	≈ 70 s	≈ 45 s

Tabella 7.2: Tempo ciclo indicativo in funzione del formato della vaschetta e della velocità di movimentazione del robot

I valori riportati hanno carattere indicativo. Lo scopo della campagna non era infatti l'ottimizzazione del tempo ciclo, bensì la verifica di fattibilità del processo: numerosi parametri, dalla velocità e accelerazione di movimentazione fino alla durata delle attese di erogazione e alla logica di sincronizzazione, restano suscettibili di ulteriore miglioramento. Nel contesto del presente lavoro e del setup sperimentale adottato era sufficiente accertare che, già con una configurazione di laboratorio non ottimizzata, il tempo ciclo si stabilizzasse entro un intervallo accettabile e ripetibile per ciascun formato. Questo risultato è significativo in chiave di fattibilità: esso dimostra che il limite inferiore del tempo ciclo è dominato dalle fasi di deposizione e non dalla movimentazione, e che l'integrazione del sistema in un contesto automatizzato, con una gestione più efficiente delle transizioni e delle attese, consentirebbe con ragionevole certezza un'ulteriore riduzione dei tempi rispetto a quelli qui riportati.

Nella configurazione finale la cella è in grado di realizzare in modo autonomo la decorazione della vaschetta, depositando lungo l'intero perimetro una sequenza di ciuffi di maionese di forma e dimensione omogenee e con spaziatura regolare. I risultati ottenuti sono riportati nella *Figura 7.4(a,b,c)*, relativa ai tre formati di vaschetta impiegati.

a.  **b.**  **c.** 

Figura 7.4: Risultato finale: ***a.*** *formato piccolo,* ***b.*** *formato medio,* ***c.*** *formato grande*

Dal punto di vista qualitativo il risultato è confrontabile con la decorazione eseguita manualmente, con il vantaggio di una maggiore costanza nella spaziatura e nell'allineamento dei ciuffi lungo il bordo, garantita dalla generazione automatica del contorno. Il corretto adattamento del sistema ai tre formati di prova conferma inoltre la capacità della cella di operare su vaschette di dimensioni diverse senza necessità di riprogrammazione, grazie al rilevamento automatico della geometria. Permangono alcuni limiti di ripetibilità, riconducibili principalmente alla sensibilità del processo alle condizioni della maionese, in particolare alla sua temperatura e consistenza, e al livello residuo di prodotto all'interno della siringa, che influenza la pressione di erogazione. Tali fattori, pur non compromettendo il risultato complessivo, indicano le direzioni lungo le quali il sistema potrebbe essere ulteriormente irrobustito.

CONCLUSIONE

Il presente lavoro di tesi ha avuto come obiettivo lo sviluppo e la verifica di fattibilità di una cella robotizzata collaborativa per la decorazione automatica di vaschette di insalata russa mediante deposizione di ciuffi di maionese. Il progetto, svolto nell'ambito di un tirocinio presso i-Labs Industry, ha portato alla realizzazione di un prototipo funzionante, integrando un robot collaborativo UR5e, un sistema di visione RGB-D Intel RealSense D455 in configurazione eye-in-hand e un end-effector meccatronico custom per l'erogazione del materiale, il tutto coordinato da un programma supervisore eseguito su PC.

Rispetto agli obiettivi iniziali, i risultati sperimentali descritti nel Capitolo 7 hanno confermato la realizzabilità del processo. Il sistema è in grado di depositare lungo l'intero perimetro della vaschetta una sequenza di ciuffi di maionese con base regolare e punta ben definita, qualitativamente confrontabili con quelli ottenuti manualmente dall'operatore, con il vantaggio di una maggiore costanza nella spaziatura e nell'allineamento, garantita dalla generazione automatica del contorno. Il primo obiettivo, relativo alla replica fedele della forma dei ciuffi, può quindi ritenersi raggiunto, una volta superate le criticità iniziali del sistema di erogazione mediante la sostituzione dell'estrusore con un ugello da sac à poche e la rimozione della guarnizione responsabile dell'attrito e della depressione sullo stantuffo.

Anche il secondo obiettivo, relativo alla corretta esecuzione della traiettoria guidata dalla visione, è stato conseguito. La procedura di calibrazione hand-eye e la pipeline di elaborazione dell'immagine hanno consentito al sistema di riconoscere autonomamente la geometria della vaschetta e di generare una traiettoria coerente con il suo profilo. Il corretto adattamento del sistema ai tre formati di prova ha dimostrato la capacità della cella di operare su vaschette di dimensioni diverse, e indipendentemente dal loro orientamento sul piano di lavoro, senza necessità di riprogrammazione manuale.

Per quanto riguarda il terzo obiettivo, relativo al tempo ciclo, i risultati vanno interpretati alla luce della natura di studio di fattibilità del lavoro. I tempi ciclo misurati nella configurazione finale, compresi tra circa 30 e 45 s a seconda del formato, restano superiori al tempo di riferimento dell'operazione manuale, pari a circa 11 s. La campagna sperimentale ha tuttavia messo in evidenza un risultato significativo in chiave di fattibilità: il limite inferiore del tempo ciclo non è dominato

dalla velocità di movimentazione del robot, bensì dalle fasi di spinta e di attesa associate alla deposizione di ciascun ciuffo. L'aumento della velocità di movimentazione, oltre una certa soglia, non produce infatti una riduzione apprezzabile del tempo complessivo, poiché il collo di bottiglia diventa l'erogazione del materiale. L'individuazione di questo limite costituisce un'indicazione preziosa per le successive fasi di sviluppo, suggerendo che il margine di miglioramento risiede principalmente in una gestione più efficiente della sincronizzazione tra movimento ed estrusione e nell'ottimizzazione dei tempi di attesa, piuttosto che nell'incremento della velocità del robot.

Permangono alcuni limiti di ripetibilità, riconducibili principalmente alla sensibilità del processo alle condizioni della maionese, in particolare alla sua temperatura e consistenza, e al livello residuo di prodotto all'interno della siringa, che influenza la pressione di erogazione. Tali fattori, non controllabili in modo rigoroso con l'attrezzatura di laboratorio disponibile, non hanno compromesso il risultato complessivo, ma indicano le direzioni lungo le quali il sistema potrebbe essere ulteriormente irrobustito.

Nel complesso, il lavoro ha dimostrato che l'automazione del processo di decorazione è tecnicamente realizzabile e che il prototipo sviluppato costituisce una base solida per una successiva industrializzazione. Tra i possibili sviluppi futuri si collocano l'ottimizzazione della logica di sincronizzazione e dei tempi ciclo al fine di avvicinarsi al benchmark dell'operazione manuale, l'introduzione di un controllo della temperatura e della consistenza del prodotto, l'eventuale adozione di una retroazione di posizione sull'asse di dosaggio e l'estensione del sistema a geometrie di decorazione più complesse, sfruttando la riconfigurabilità garantita dall'architettura modulare adottata.

BIBLIOGRAFIA

[1] G. C. Devol, Jr., "Programmed Article Transfer," U.S. Patent 2 988 237, June 13, 1961.

[2] S. B. Kokane, S. S. Kalamnurikar, and S. B. Khose, "Robotics in food processing industries: A review," *The Pharma Innovation Journal*, vol. 11, no. 7S, pp. 948–953, Jul. 2022.

[3] R. Accorsi, A. Tufano, A. Gallo, F. G. Galizia, G. Cocchi, M. Ronzoni, A. Abbate, and R. Manzini, "An application of collaborative robots in a food production facility," *Procedia Manufacturing*, vol. 38, pp. 341–348, 2019.

[4] R. Bharathiraja and K. Keerthana, "Automation and Robotics in Manufacturing and Food Industries," in *Smart and Sustainable Technologies*, Metro Tech Publishing House, May 2025, ch. 7, pp. 47–54

[5] V. Nikolova-Alexieva, K. Valeva, M. Terziyska, and N. Shakev, "Collaborative Robots as an Engineering Tool for the Transition of the Food Industry to Industry 5.0," *Engineering Proceedings*, vol. 100, no. 1, p. 57, 2025.

[6] Z. Xiao, J. Wang, L. Han, S. Guo, and Q. Cui, "Application of Machine Vision System in Food Detection," *Frontiers in Nutrition*, vol. 9, art. 888245, 2022, doi: 10.3389/fnut.2022.888245.

[7] S. Vasudevan, M. L. Mekhalfi, C. Blanes, M. Lecca, F. Poiesi, P. I. Chippendale, P. M. Fresnillo, W. M. Mohammed, and J. L. Martinez Lastra, "Machine Vision and Robotics for Primary Food Manipulation and Packaging: A Survey," *IEEE Access*, vol. 12, 2024, doi: 10.1109/ACCESS.2024.3479781.

[8] A. P. Singh, D. Romanov, E. Misimi, and A. Mason, "Vision-based approaches for cutting food products with robots: A review," *Robotics and Autonomous Systems*, vol. 194, 2025.

[9] R. Y. Tsai and R. K. Lenz, "A New Technique for Fully Autonomous and Efficient 3D Robotics Hand/Eye Calibration," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 3, pp. 345–358, Jun. 1989.

[10] P. J. Fellows, *Food Processing Technology: Principles and Practice*, 2nd ed. Cambridge, U.K.: Woodhead Publishing, 2000.

[11] W.-T. Yang, M. Hirao, and M. Tomizuka, "Design, Modeling, and Parametric Analysis of a Syringe Pump for Soft Pneumatic Actuators," in *2023 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, 2023, pp. 317–322.

[12] A. Gervasi, P. Cardol, and P. E. Meyer, "Open-hardware wireless controller and 3D-printed pumps for efficient liquid manipulation," *HardwareX*, vol. 9, art. e00199, May 2021.

[13] Z. Guo, F. Fei, X. Song, and C. Zhou, "Analytical Study and Experimental Verification of Shear-Thinning Ink Flow in Direct Ink Writing Process," *Journal of Manufacturing Science and Engineering*, vol. 145, no. 7, art. 071001, Jul. 2023.

[14] M. A. Rao, *Rheology of Fluid and Semisolid Foods: Principles and Applications*, 2nd ed., Food Engineering Series. New York, NY, USA: Springer, 2007.

[15] D. J. McClements, *Food Emulsions: Principles, Practices, and Techniques*, 2nd ed., CRC Series in Contemporary Food Science. Boca Raton, FL, USA: CRC Press, 2005.

[16] S. E. Friberg, K. Larsson, and J. Sjöblom, Eds., *Food Emulsions*, 4th ed., Food Science and Technology Series. New York, NY, USA: Marcel Dekker, 2004.

[17] P. Štern, J. Pokorný, A. Šedivá, and Z. Panovská, "Rheological and sensory characteristics of yoghurt-modified mayonnaise," *Czech Journal of Food Sciences*, vol. 26, no. 3, pp. 190–198, 2008.

[18] E.-O. Rukke and R. B. Schüller, "Rheological properties of different types of mayonnaise," *Annual Transactions of the Nordic Rheology Society*, vol. 27, pp. 165–171, 2019.

[19] S. A. Bredikhin, A. N. Martekha, V. N. Andreev, E. A. Soldusova, and N. A. Karpova, "Investigation of the structural and mechanical characteristics of mayonnaise with the addition of linseed oil," *IOP Conference Series: Earth and Environmental Science*, vol. 979, no. 1, art. 012089, 2022.

[20] Universal Robots A/S, *UR5e Manuale Utente, PolyScope 5*, documento n. 710-970-00, Odense, Denmark: Universal Robots A/S, 2025.

[21] RealSense Technology, *Intel® RealSense™ Camera 400 Series Product Family Datasheet*, Document no. 337029-017, Rev. 023, Mar. 2026.

[22] OpenCV.org, "About," *opencv.org*, 2026. [Online]. Available: https://opencv.org/about/.

[23] OpenCV Team, "Camera Calibration and 3D Reconstruction," *OpenCV 4.13.0 Documentation*, 2026. [Online]. Available: https://docs.opencv.org/4.13.0/d9/d0c/group__calib3d.html

[24] Universal Robots A/S, "Move," *PolyScope 5 Software Manual, SW5.25*, Universal Robots A/S, 2026. [Online]. Available: https://www.universal-robots.com/manuals/EN/HTML/SW5_25/Content/prod-usr-man/software/PolyScope/content/BasicProgNodes/commandtab_move_en.htm

[25] OpenCV Team, "Contour Features," *OpenCV 4.13.0 Documentation, OpenCV-Python Tutorials*, 2026. [Online]. Available: https://docs.opencv.org/4.13.0/dd/d49/tutorial_py_contour_features.html

APPENDICE A

```
import numpy as np
import cv2
import pyrealsense2 as rs
from scipy.spatial.transform import Rotation as R
from rtde_receive import RTDEReceiveInterface
from rtde_control import RTDEControlInterface
import time
import os

# =========================
# CONFIG
# =========================
ROBOT_IP = "192.168.1.172"

# Pose robot
poses_J_rad = np.array(
    [[-0.19256431261171514, -1.5568978006816288, -1.61773681640625, 3.1814328867145996,
0.4814609885215759, -1.5020173231707972],
    [-0.4063828627215784, -2.1987768612303675, -1.0315446853637695, 3.5250021654316406,
0.20557045936584473, -2.3131096998797815],
    [-0.7504928747760218, -2.252383371392721, -1.0321599245071411, 3.409557028407715,
0.5860270857810974, -2.21971303621401],
    [0.22195741534233093, -2.032619615594381, -1.0322474241256714, 3.170455141658447, -
0.019376103078023732, -2.232065025960104],
    [0.43592211604118347, -2.228217741052145, -1.2208538055419922, 2.652054472560547, -
0.7951300779925745, -1.9351571241961878],
    [-0.18044931093324834, -2.3725620708861292, -1.0333237648010254, 4.734252893343129,
0.028758414089679718, -3.63447076479067],
    [-1.0325496832477015, -1.946329256097311, -1.4312549829483032, 5.1305212555225275,
0.742164671421051, -3.675234858189718],
    [-0.7112811247455042, -1.8565436802306117, -1.4492357969284058, 4.44751946508374,
0.5162081122398376, -2.9438725153552454],
    [-0.5814340750323694, -2.1857792339720667, -1.312865972518921, 4.7317268186952255,
0.33532533049583435, -3.4028647581683558],
    [-0.8978441397296351, -2.166797777215475, -1.3129005432128906, 4.73170304016725,
0.7498118877410889, -3.4031084219561976],
    [-0.22261268297304326, -1.8867751560606898, -1.2679729461669922,
4.4430810648151855, -0.15644532838930303, -3.319209400807516],
    [-0.8464439550982874, -1.7687584362425746, -1.2704339027404785, 4.338836832637451,
0.6819520592689514, -2.9723196665393274],
    [-1.166736904774801, -1.7848030529417933, -1.2950299978256226, 3.6262246805378417,
0.9547579884529114, -2.51552921930422],
    [-0.8044536749469202, -1.782849451104635, -1.2952221632003784, 3.733546419734619,
0.6912281513214111, -2.4776304403888147],
```

```
    [-0.463907543812887, -1.7486711941161097, -1.295563817024231, 3.7309200006672363,
0.3985993266105652, -2.4722583929644983],
    [-0.33871156374086553, -1.9039517841734828, -1.294213056564331, 3.88263384878125,
0.2969833314418793, -2.5374582449542444],
    [-0.17311507860292608, -1.7421413860716761, -1.7260864973068237, 3.007383032436035,
0.11999348551034927, -1.1658347288714808],
    [-0.6143792311297815, -1.8466273746886195, -1.3192148208618164, 5.499810862332144,
2.6519687175750732, 1.0354912281036377]
    ])

# ========================
# CHESSBOARD CONFIG
# ========================
CHESSBOARD_SIZE = (6, 9)
SQUARE_SIZE = 0.035

objp = np.zeros((CHESSBOARD_SIZE[0]*CHESSBOARD_SIZE[1], 3), np.float32)
objp[:, :2] = np.mgrid[0:CHESSBOARD_SIZE[0], 0:CHESSBOARD_SIZE[1]].T.reshape(-1, 2)
objp *= SQUARE_SIZE

valid_poses = 0
total_poses = len(poses_J_rad)

# ========================
# CARTELLA SALVATAGGIO FOTO  <-- AGGIUNTO
# ========================
SAVE_DIR = "handeye_images_ciuffi"
os.makedirs(SAVE_DIR, exist_ok=True)
print(f"Immagini salvate in: {os.path.abspath(SAVE_DIR)}\n")

# ========================
# REALSENSE
# ========================
pipeline = rs.pipeline()
config = rs.config()
config.enable_stream(rs.stream.color, 640, 480, rs.format.bgr8, 30)
pipeline.start(config)
align = rs.align(rs.stream.color)

profile = pipeline.get_active_profile()
intr = profile.get_stream(rs.stream.color)\
    .as_video_stream_profile().get_intrinsics()

camera_matrix = np.array([
    [intr.fx, 0, intr.ppx],
    [0, intr.fy, intr.ppy],
    [0, 0, 1]
])
```

```
dist_coeffs = np.array(intr.coeffs)

# ========================
# ROBOT
# ========================
rtde_r = RTDEReceiveInterface(ROBOT_IP)
rtde_c = RTDEControlInterface(ROBOT_IP)

# ========================
# UTILS
# ========================
def pose_to_matrix(pose):
    t = np.array(pose[:3])
    r = R.from_rotvec(pose[3:])
    T = np.eye(4)
    T[:3, :3] = r.as_matrix()
    T[:3, 3] = t
    return T

def invert(T):
    R_ = T[:3, :3]
    t_ = T[:3, 3]
    T_inv = np.eye(4)
    T_inv[:3, :3] = R_.T
    T_inv[:3, 3] = -R_.T @ t_
    return T_inv

# ========================
# STORAGE
# ========================
R_gripper2base = []
t_gripper2base = []
R_target2cam = []
t_target2cam = []

# ========================
# LOOP POSE
# ========================
print("\nAvvio acquisizione automatica...\n")

for i, pose in enumerate(poses_J_rad):
    print(f"Movimento alla posa {i+1}")

    rtde_c.moveJ(pose, speed=0.3, acceleration=0.3)
    time.sleep(3)

    start = time.time()
    while time.time() - start < 2:
```

```
        frames = pipeline.wait_for_frames()
        aligned = align.process(frames)
        color = aligned.get_color_frame()
        if not color:
            continue

        image = np.asanyarray(color.get_data())
        cv2.imshow("Live camera", image)

        if cv2.waitKey(1) & 0xFF == 27:
            break

    # ========================
    # SALVATAGGIO FOTO RAW  <-- AGGIUNTO
    # ========================
    raw_path = os.path.join(SAVE_DIR, f"pose_{i+1:02d}_raw.png")
    cv2.imwrite(raw_path, image)
    print(f"  Foto raw salvata: {raw_path}")

    # ========================
    # RILEVAMENTO SCACCHIERA
    # ========================
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    ret, corners = cv2.findChessboardCorners(gray, CHESSBOARD_SIZE,
cv2.CALIB_CB_ADAPTIVE_THRESH + cv2.CALIB_CB_NORMALIZE_IMAGE)

    if not ret:
        print(f"Scacchiera NON rilevata, salto questa posa ({i+1}/{total_poses})")
        continue

    corners = cv2.cornerSubPix(
        gray,
        corners,
        (11,11),
        (-1,-1),
        criteria=(cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
    )

    cv2.drawChessboardCorners(image, CHESSBOARD_SIZE, corners, ret)
    cv2.imshow("Scacchiera", image)
    cv2.waitKey(200)

    # ========================
    # SALVATAGGIO FOTO CON SCACCHIERA  <-- AGGIUNTO
    # ========================
    chess_path = os.path.join(SAVE_DIR, f"pose_{i+1:02d}_chess.png")
    cv2.imwrite(chess_path, image)
    print(f"  Foto scacchiera salvata: {chess_path}")
```

```
# ========================
# STIMA POSA (PnP)
# ========================
imgp = corners.reshape(-1, 2).astype(np.float32)

success, rvec, tvec = cv2.solvePnP(
    objp,
    imgp,
    camera_matrix,
    dist_coeffs
)

if not success:
    print(f"solvePnP fallito ({i+1}/{total_poses})")
    continue

rvec = rvec.astype(np.float64)
tvec = tvec.astype(np.float64)

cv2.drawFrameAxes(image, camera_matrix, dist_coeffs, rvec, tvec, 0.05)
cv2.imshow("Frame assi", image)
cv2.waitKey(300)

# ========================
# SALVATAGGIO FOTO CON ASSI  <-- AGGIUNTO
# ========================
axes_path = os.path.join(SAVE_DIR, f"pose_{i+1:02d}_axes.png")
cv2.imwrite(axes_path, image)
print(f"  Foto assi salvata: {axes_path}")

# ========================
# CAMERA -> TARGET
# ========================
R_ct, _ = cv2.Rodrigues(rvec)
R_target2cam.append(R_ct)
t_target2cam.append(tvec.reshape(3,1))

print("R_target2cam acquisita")
print("t_target2cam acquisita")

# ========================
# ROBOT
# ========================
ee_pose = rtde_r.getActualTCPPose()
T = pose_to_matrix(ee_pose)

R_gripper2base.append(T[:3, :3])
```

```
    t_gripper2base.append(T[:3, 3].reshape(3,1))
    print("Campione acquisito\n")

# ========================
# CLEANUP
# ========================
pipeline.stop()
cv2.destroyAllWindows()

# ========================
# CALIBRAZIONE
# ========================
print("Calcolo hand-eye...\n")
# ── CONFRONTO METODI ── incolla queste righe prima di calibrateHandEye ──
methods = {
    "TSAI":       cv2.CALIB_HAND_EYE_TSAI,
    "PARK":       cv2.CALIB_HAND_EYE_PARK,
    "HORAUD":     cv2.CALIB_HAND_EYE_HORAUD,
    "ANDREFF":    cv2.CALIB_HAND_EYE_ANDREFF,
    "DANIILIDIS": cv2.CALIB_HAND_EYE_DANIILIDIS,
}
print("\n=== CONFRONTO METODI ===")
for name, method in methods.items():
    R_c2g, t_c2g = cv2.calibrateHandEye(
        R_gripper2base, t_gripper2base,
        R_target2cam,   t_target2cam,
        method=method
    )
    errs = []
    T_c2g = np.eye(4); T_c2g[:3,:3]=R_c2g; T_c2g[:3,3]=t_c2g.flatten()
    for i in range(len(R_gripper2base)):
        T_bg = np.eye(4); T_bg[:3,:3]=R_gripper2base[i];
T_bg[:3,3]=t_gripper2base[i].flatten()
        T_tc = np.eye(4);
T_tc[:3,:3]=R_target2cam[i];   T_tc[:3,3]=t_target2cam[i].flatten()
        T_bt = T_bg @ T_c2g @ T_tc
        if i==0: T_ref=T_bt.copy(); continue
        diff = T_ref @ np.linalg.inv(T_bt)
        errs.append(np.linalg.norm(diff[:3,3])*1000)
    print(f"{name:12s}: t=[{t_c2g[0,0]*1000:7.2f}, {t_c2g[1,0]*1000:7.2f},
{t_c2g[2,0]*1000:7.2f}] mm  err_medio={np.mean(errs):.2f}mm  max={np.max(errs):.2f}mm")
# ── FINE CONFRONTO ──

R_cam2gripper, t_cam2gripper = cv2.calibrateHandEye(
    R_gripper2base,
    t_gripper2base,
    R_target2cam,
    t_target2cam,
```

```
    method=cv2.CALIB_HAND_EYE_TSAI
)
print("R_c2g", R_cam2gripper)
print("t_c2g:", t_cam2gripper)

T_cam2gripper = np.eye(4)
T_cam2gripper[:3, :3] = R_cam2gripper
T_cam2gripper[:3, 3] = t_cam2gripper.flatten()

print("RISULTATO:\n")
print(T_cam2gripper)

np.savetxt("handeye_result_chess_ciuffi.txt", T_cam2gripper)
np.save("handeye_result_chess_ciuffi.npy", T_cam2gripper)
print("\nSalvato su file")

# ========================
# VERIFICA CALIBRAZIONE
# ========================
print("\n=== VERIFICA CALIBRAZIONE ===\n")

T_cam2gripper = np.eye(4)
T_cam2gripper[:3, :3] = R_cam2gripper
T_cam2gripper[:3, 3] = t_cam2gripper.flatten()

errori_rot = []
errori_tra = []

for i in range(len(R_gripper2base)):
    # Ricostruisci T_base_gripper
    T_base_gripper = np.eye(4)
    T_base_gripper[:3, :3] = R_gripper2base[i]
    T_base_gripper[:3, 3] = t_gripper2base[i].flatten()

    # Ricostruisci T_target2cam
    T_target2cam = np.eye(4)
    T_target2cam[:3, :3] = R_target2cam[i]
    T_target2cam[:3, 3] = t_target2cam[i].flatten()

    # Equazione hand-eye: T_base_gripper @ T_cam2gripper @ T_target2cam = costante
    # Cioè: T_base_target deve essere uguale per tutte le pose
    T_base_target = T_base_gripper @ T_cam2gripper @ T_target2cam

    if i == 0:
        T_base_target_ref = T_base_target.copy()
        print(f"Pose 1 (riferimento):")
        print(f"  T_base_target =\n{T_base_target_ref}")
        continue
```

```
    # Errore rispetto alla prima posa
    diff = T_base_target_ref @ np.linalg.inv(T_base_target)

    # Errore rotazione (angolo in gradi)
    R_diff = diff[:3, :3]
    angle = np.arccos(np.clip((np.trace(R_diff) - 1) / 2, -1, 1))
    err_rot = np.rad2deg(angle)

    # Errore traslazione (mm)
    err_tra = np.linalg.norm(diff[:3, 3]) * 1000

    errori_rot.append(err_rot)
    errori_tra.append(err_tra)

    print(f"Pose {i+1:>2}: err_rot={err_rot:6.3f}°  err_tra={err_tra:6.2f} mm")

print(f"\nErrore medio rotazione    : {np.mean(errori_rot):.3f}°")
print(f"Errore massimo rotazione  : {np.max(errori_rot):.3f}°")
print(f"Errore medio traslazione  : {np.mean(errori_tra):.2f} mm")
print(f"Errore massimo traslazione: {np.max(errori_tra):.2f} mm")

print("\n--- Valori accettabili ---")
print("  Rotazione  < 1.0°  → buono,  < 0.5° → ottimo")
print("  Traslazione < 5 mm → buono,  < 2 mm → ottimo")

# ==========================
# SALVATAGGIO ERRORI SU FILE
# ==========================
NOME_FILE_ERRORI = "errore_ciuffi.txt"
with open(NOME_FILE_ERRORI, "w", encoding="utf-8") as f:
    f.write("=== VERIFICA CALIBRAZIONE HAND-EYE ===\n\n")
    for i, (er, et) in enumerate(zip(errori_rot, errori_tra)):
        f.write(f"Pose {i+2:>2}: err_rot={er:6.3f}°  err_tra={et:6.2f} mm\n")
    f.write(f"\nErrore medio rotazione    : {np.mean(errori_rot):.3f}°\n")
    f.write(f"Errore massimo rotazione  : {np.max(errori_rot):.3f}°\n")
    f.write(f"Errore medio traslazione  : {np.mean(errori_tra):.2f} mm\n")
    f.write(f"Errore massimo traslazione: {np.max(errori_tra):.2f} mm\n")
    f.write("\n--- Valori accettabili ---\n")
    f.write("  Rotazione  < 1.0° → buono,  < 0.5° → ottimo\n")
    f.write("  Traslazione < 5 mm → buono,  < 2 mm → ottimo\n")
print(f"\nErrori salvati su: {NOME_FILE_ERRORI}")

# ==========================
# VERIFICA ALTERNATIVA - TUTTE LE COPPIE
# ==========================
print("\n=== VERIFICA SU TUTTE LE COPPIE ===\n")
```

```
errori = []
n = len(R_gripper2base)

for i in range(n):
    for j in range(i+1, n):
        T_bg_i = np.eye(4); T_bg_i[:3,:3]=R_gripper2base[i];
T_bg_i[:3,3]=t_gripper2base[i].flatten()
        T_bg_j = np.eye(4); T_bg_j[:3,:3]=R_gripper2base[j];
T_bg_j[:3,3]=t_gripper2base[j].flatten()
        T_tc_i = np.eye(4);
T_tc_i[:3,:3]=R_target2cam[i];   T_tc_i[:3,3]=t_target2cam[i].flatten()
        T_tc_j = np.eye(4);
T_tc_j[:3,:3]=R_target2cam[j];   T_tc_j[:3,3]=t_target2cam[j].flatten()

        T_bt_i = T_bg_i @ T_cam2gripper @ T_tc_i
        T_bt_j = T_bg_j @ T_cam2gripper @ T_tc_j

        diff = np.linalg.inv(T_bt_i) @ T_bt_j
        err = np.linalg.norm(diff[:3,3]) * 1000
        errori.append(err)

print(f"Errore medio su tutte le coppie: {np.mean(errori):.2f} mm")
print(f"Errore mediano                 : {np.median(errori):.2f} mm")
print(f"Errore max                     : {np.max(errori):.2f} mm")
```

APPENDICE B

```
import math
import serial
import time
import pyrealsense2 as rs
import numpy as np
import cv2
import sys
from collections import deque
from rtde_control import RTDEControlInterface
from rtde_receive import RTDEReceiveInterface
from scipy.spatial.transform import Rotation as R

# ═══════════════════════════════════════
# ROBOT E SERIALE
# ═══════════════════════════════════════
rtde_c = RTDEControlInterface("192.168.1.172")
rtde_r = RTDEReceiveInterface("192.168.1.172")
```

```python
ser = serial.Serial("COM3", 115200)
time.sleep(2)


def pose_to_matrix(pose):
    x, y, z, rx, ry, rz = pose
    R_mat, _ = cv2.Rodrigues(np.array([rx, ry, rz]))
    T = np.eye(4)
    T[:3, :3] = R_mat
    T[:3, 3] = [x, y, z]
    return T


# ── Posa di osservazione ──
OBSERVE_DEG = np.array([-7.14, -97.76, -77.54, 175.65, 32.03, -90.45])
OBSERVE_RAD = np.deg2rad(OBSERVE_DEG)

print("[INFO] Movimento verso posa di osservazione...")
rtde_c.moveJ(OBSERVE_RAD, 0.2, 0.2)
print("[INFO] Robot in posa di osservazione.")

OBSERVE_TCP   = rtde_r.getActualTCPPose()
T_ee_cam      = np.load("handeye_result_chess_ciuffi.npy")
print(f"Traslazione salvata nel file: {T_ee_cam[:3,3]*1000} mm")
print("T_ee_cam inversa traslazione (mm):", np.linalg.inv(T_ee_cam)[:3,3]*1000)
T_base_ee_obs = pose_to_matrix(OBSERVE_TCP)
print("T_base_ee_obs", T_base_ee_obs)
print(f"[INFO] TCP osservazione: {OBSERVE_TCP}")


# ═══════════════════════════════════════════
# PARAMETRI TUNABILI
# ═══════════════════════════════════════════
COLOR_W, COLOR_H, FPS = 640, 480, 30
DEPTH_W, DEPTH_H      = 640, 480

# ─ Rilevamento ─
CANNY_LOW           = 50
CANNY_HIGH          = 150
MIN_CONTOUR_AREA    = 1_000
MAX_CONTOUR_AREA    = 500_000
ASPECT_RATIO_MIN    = 0.3
ASPECT_RATIO_MAX    = 3.5
POLY_EPS            = 0.04
VALID_VERTEX_COUNTS = {4}

# ─ Stabilizzazione temporale vertici ─
VERTEX_HISTORY = 10   # frame di storia per la media mobile
```

```
# — Depth —
MIN_VALID_DEPTH_RATIO = 0.1
DEPTH_MIN_MM          = 10
DEPTH_MAX_MM          = 700

# — Movimento robot —
Z_OFFSET_M         = 0.02    # offset sopra il centro scatola [m]
Z_CONTOUR_OFFSET_M = 0.017   # offset sopra ogni punto del contorno [m]
MOVE_SPEED         = 0.2     # m/s  (tratti di discesa verso p_next)
MOVE_ACCEL         = 0.2     # m/s²
MOVE_SPEED2        = 0.2     # m/s  (tratti obliqui verso via_xyz)
MOVE_ACCEL2        = 0.2     # m/s²

# — Traiettoria arco tra punti del contorno —
ARC_LIFT_M       = 0.02   # altezza del via-point sopra il piano del contorno [m]
ARC_CENTER_PULL  = 0.65   # [0..1] quanto tirare il via-point verso il centro

# — Contorno su segmenti retti tra vertici —
VERTEX_INSET_PX    = 15   # pixel di rientro verso il centroide per i VERTICI
CONTOUR_INSET_PX   = 10   # pixel di rientro verso il centroide per i punti INTERMEDI
VERTEX_STEP        = 20   # distanza in pixel tra punti equidistanziati sul lato retto

# — Seriale pompa —
SERIAL_INTERVAL = 0.10   # secondi tra un impulso e il successivo


# ═══════════════════════════════════════════
# STABILIZZAZIONE TEMPORALE DEI VERTICI
# ═══════════════════════════════════════════
_vertex_buffers: dict[int, deque] = {}


def _canonical_order(pts: np.ndarray, cx: float, cy: float) -> np.ndarray:
    """
    Ordina i vertici per angolo (arctan2) rispetto al centroide.
    Restituisce un array (N, 2).
    """
    angles = np.arctan2(pts[:, 1] - cy, pts[:, 0] - cx)
    return pts[np.argsort(angles)]


def stabilize_vertices(box_idx: int, contour: np.ndarray, center: tuple) -> np.ndarray:
    """
    Usa minAreaRect sul contorno grezzo per ottenere i 4 vertici del rettangolo
    ruotato minimo, poi stabilizza con media mobile su (cx, cy, w, h, angle).

    Ritorni
```

```
    -------
    np.ndarray shape (4, 1, 2) dtype int32 — vertici stabilizzati in ordine canonico
    """
    global _vertex_buffers

    rect     = cv2.minAreaRect(contour)
    rcx, rcy = rect[0]
    rw, rh   = rect[1]
    angle    = rect[2]

    if box_idx not in _vertex_buffers:
        _vertex_buffers[box_idx] = deque(maxlen=VERTEX_HISTORY)

    buf = _vertex_buffers[box_idx]
    buf.append(np.array([rcx, rcy, rw, rh, angle], dtype=np.float32))

    stacked = np.stack(buf, axis=0)
    avg     = stacked.mean(axis=0)
    avg_cx, avg_cy, avg_w, avg_h, avg_angle = avg

    stable_rect     = ((avg_cx, avg_cy), (avg_w, avg_h), avg_angle)
    box_pts         = cv2.boxPoints(stable_rect)
    box_pts_ordered = _canonical_order(box_pts, avg_cx, avg_cy)

    return box_pts_ordered.reshape(-1, 1, 2).astype(np.int32)


# ═══════════════════════════════════════
# HELPERS DEPTH
# ═══════════════════════════════════════
def robust_depth_at(depth_frame, px, py, roi_half=8):
    x0 = max(0, px - roi_half)
    y0 = max(0, py - roi_half)
    x1 = min(DEPTH_W - 1, px + roi_half)
    y1 = min(DEPTH_H - 1, py + roi_half)
    depth_image = np.asanyarray(depth_frame.get_data())
    roi   = depth_image[y0:y1, x0:x1].astype(np.float32)
    valid = roi[(roi > DEPTH_MIN_MM) & (roi < DEPTH_MAX_MM)]
    if valid.size == 0 or valid.size / roi.size < MIN_VALID_DEPTH_RATIO:
        return None
    return float(np.median(valid)) / 1000.0


def pixel_to_base(intrinsics, depth_frame, px, py):
    depth_m = robust_depth_at(depth_frame, int(px), int(py))
    if depth_m is None:
        return None
    p_cam_xyz = rs.rs2_deproject_pixel_to_point(
```

```
        intrinsics, [float(px), float(py)], depth_m
    )
    p_cam  = np.array([p_cam_xyz[0], p_cam_xyz[1], p_cam_xyz[2], 1.0])
    p_base = T_base_ee_obs @ T_ee_cam @ p_cam
    return tuple(round(v, 4) for v in p_base[:3])


# ═══════════════════════════════════════════
# INSET: sposta i punti verso il centroide
# ═══════════════════════════════════════════
def inset_point(px, py, cx, cy, inset_px):
    dx   = cx - px
    dy   = cy - py
    dist = math.sqrt(dx * dx + dy * dy)
    if dist < 1e-6:
        return int(px), int(py)
    ratio = min(inset_px / dist, 1.0)
    return int(round(px + dx * ratio)), int(round(py + dy * ratio))


# ═══════════════════════════════════════════
# CONTORNO SU SEGMENTI RETTI TRA VERTICI
# ═══════════════════════════════════════════
def build_contour_with_vertices(contour, vertices, center,
                                vertex_inset_px, contour_inset_px, step):
    """
    Unisce i vertici stabilizzati con segmenti RETTI, piazza punti
    equidistanziati di `step` pixel su ciascun lato e applica inset
    DIFFERENZIATO: vertex_inset_px per i vertici, contour_inset_px
    per i punti intermedi.
    """
    cx, cy   = center
    vert_pts = vertices.reshape(-1, 2).astype(np.float32)
    n_verts  = len(vert_pts)
    final_px = []

    for seg in range(n_verts):
        p_start = vert_pts[seg]
        p_end   = vert_pts[(seg + 1) % n_verts]

        # Vertice di partenza → inset VERTEX_INSET_PX
        ivx_s, ivy_s = inset_point(
            float(p_start[0]), float(p_start[1]), cx, cy, vertex_inset_px
        )
        final_px.append((ivx_s, ivy_s))

        seg_vec = p_end - p_start
        seg_len = float(np.linalg.norm(seg_vec))
```

```
        if seg_len < 1e-6:
            continue

        # Punti intermedi → inset CONTOUR_INSET_PX
        n_intermediate = int(seg_len // step)
        for k in range(1, n_intermediate):
            t   = (k * step) / seg_len
            px  = float(p_start[0]) + t * float(seg_vec[0])
            py  = float(p_start[1]) + t * float(seg_vec[1])
            ipx, ipy = inset_point(px, py, cx, cy, contour_inset_px)
            final_px.append((ipx, ipy))

    total_vert = n_verts
    total_samp = len(final_px) - total_vert
    print(f"[CONTOUR] vertici={total_vert}  intermedi={total_samp}  "
          f"totale={len(final_px)} pt  step={step}px  "
          f"v_inset={vertex_inset_px}px  c_inset={contour_inset_px}px")
    return final_px


# ═══════════════════════════════════════
# RILEVAMENTO SCATOLE
# ═══════════════════════════════════════
def detect_boxes(color_image):
    gray    = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)
    cv2.imshow('gray', gray)
    blurred = cv2.GaussianBlur(gray, (7, 7), 0)
    cv2.imshow('blurred', blurred)
    _, binary = cv2.threshold(blurred, 160, 255, cv2.THRESH_BINARY)
    cv2.imshow('binary', binary)

    edges   = cv2.Canny(binary, CANNY_LOW, CANNY_HIGH)
    cv2.imshow('edges', edges)
    kernel  = cv2.getStructuringElement(cv2.MORPH_RECT, (5, 5))
    closed  = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel)
    contours, _ = cv2.findContours(
        closed, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE
    )

    boxes = []
    for cnt in contours:
        area = cv2.contourArea(cnt)
        if not (MIN_CONTOUR_AREA < area < MAX_CONTOUR_AREA):
            continue
        peri       = cv2.arcLength(cnt, True)
        approx_raw = cv2.approxPolyDP(cnt, POLY_EPS * peri, True)
        if len(approx_raw) not in VALID_VERTEX_COUNTS:
            continue
```

```
        x, y, w, h = cv2.boundingRect(approx_raw)
        if not (ASPECT_RATIO_MIN < w / float(h) < ASPECT_RATIO_MAX):
            continue
        cx = x + w // 2
        cy = y + h // 2
        boxes.append({
            'contour'     : cnt,
            'center'      : (cx, cy),
            'bbox'        : (x, y, w, h),
            'vertices_raw': approx_raw,
        })

    boxes.sort(key=lambda b: cv2.contourArea(b['contour']), reverse=True)

    for box_idx, box in enumerate(boxes):
        stable = stabilize_vertices(
            box_idx,
            box['contour'],
            box['center']
        )
        box['vertices'] = stable

    return boxes


# ============================================
# COORDINATE 3D CENTRO
# ============================================
def box_center_to_base(depth_frame, intrinsics, box):
    cx, cy = box['center']
    return pixel_to_base(intrinsics, depth_frame, cx, cy)


# ============================================
# COSTRUZIONE WAYPOINT
# ============================================
def make_waypoint(x, y, z, rvec_target, z_offset=0.0):
    wp    = list(OBSERVE_TCP)
    wp[0] = x
    wp[1] = y
    wp[2] = z + z_offset
    wp[3] = float(rvec_target[0])
    wp[4] = float(rvec_target[1])
    wp[5] = float(rvec_target[2])
    return wp


# ============================================
```

```python
# CALCOLO VIA-POINT ARCO
# ============================================
def compute_arc_via(p_curr, p_next, center_3d, z_offset, lift, center_pull):
    cx_3d, cy_3d, _ = center_3d
    mx_seg = (p_curr[0] + p_next[0]) / 2.0
    my_seg = (p_curr[1] + p_next[1]) / 2.0
    mx = mx_seg + center_pull * (cx_3d - mx_seg)
    my = my_seg + center_pull * (cy_3d - my_seg)
    mz = (p_curr[2] + p_next[2]) / 2.0 + lift
    return (mx, my, mz)


# ============================================
# ROTAZIONE J6 + ALLINEAMENTO Z_TOOL
# ============================================
def compute_ready_pose():
    current_joints  = list(rtde_r.getActualQ())
    rotated_joints  = current_joints.copy()
    rotated_joints[5] += math.pi / 2
    print("[READY] Rotazione J6 +90°...")
    rtde_c.moveJ(rotated_joints, 0.3, 0.3)

    tcp_after = rtde_r.getActualTCPPose()
    print(f"[READY] TCP dopo J6: {tcp_after}")

    R_after, _ = cv2.Rodrigues(np.array(tcp_after[3:6]))
    x_tool_projected = np.array([R_after[0, 0], R_after[1, 0], 0.0])
    norm = np.linalg.norm(x_tool_projected)
    if norm < 1e-6:
        x_tool_projected = np.array([1.0, 0.0, 0.0])
    else:
        x_tool_projected /= norm

    z_tool_target = np.array([0.0, 0.0, -1.0])
    y_tool_target = np.cross(z_tool_target, x_tool_projected)
    y_tool_target /= np.linalg.norm(y_tool_target)

    R_target    = np.column_stack([x_tool_projected, y_tool_target, z_tool_target])
    rvec, _     = cv2.Rodrigues(R_target)
    rvec_target = rvec.flatten()
    print(f"[READY] RVEC_TARGET: {rvec_target}")

    ready_tcp    = list(tcp_after)
    ready_tcp[3] = float(rvec_target[0])
    ready_tcp[4] = float(rvec_target[1])
    ready_tcp[5] = float(rvec_target[2])

    ready_rad = rtde_c.getInverseKinematics(ready_tcp, rotated_joints)
```

```
    if not ready_rad:
        print("[WARN] IK fallita — fallback su config ruotata.")
        ready_rad = rotated_joints

    print("[READY] moveJ allineamento Z_tool perpendicolare...")
    rtde_c.moveJ(ready_rad, 0.3, 0.3)
    print("[READY] Allineamento completato.")

    return rvec_target, ready_rad


# ═══════════════════════════════════════
# MOVIMENTO ROBOT — CONTORNO CON ARCHI
# ═══════════════════════════════════════
def move_to_box_and_trace(coord_center, contour_3d_points):
    rvec_target, _ = compute_ready_pose()

    if not contour_3d_points:
        print("[WARN] Nessun punto 3D del contorno, salto il tracciamento.")
    else:
        n = len(contour_3d_points)
        print(f"[ROBOT] → Contorno ({n} waypoint)")

        # ── 1. Vai direttamente al primo punto del contorno ──
        vx0, vy0, vz0 = contour_3d_points[0]
        rtde_c.moveL(
            make_waypoint(vx0, vy0, vz0, rvec_target, z_offset=Z_CONTOUR_OFFSET_M),
            MOVE_SPEED, MOVE_ACCEL
        )
        ser.write(b"s 6 670")
        time.sleep(2)
        print(f"[ROBOT]   → Primo punto: ({vx0:.3f},{vy0:.3f},{vz0:.3f})")

        # ── 2. Percorri il contorno con archi ──
        for i in range(n - 1):
            p_curr = contour_3d_points[i]
            p_next = contour_3d_points[i + 1]

            p_curr_z = (p_curr[0], p_curr[1], p_curr[2] + Z_CONTOUR_OFFSET_M)
            p_next_z = (p_next[0], p_next[1], p_next[2] + Z_CONTOUR_OFFSET_M)

            via_xyz = compute_arc_via(
                p_curr_z, p_next_z,
                coord_center,
                z_offset=0.0,
                lift=ARC_LIFT_M,
                center_pull=ARC_CENTER_PULL
            )
```

```
            rtde_c.moveL(
                make_waypoint(*via_xyz, rvec_target, z_offset=0.0),
                MOVE_SPEED2, MOVE_ACCEL2
            )

            ser.write(b"s 5 670")

            rtde_c.moveL(
                make_waypoint(*p_next_z, rvec_target, z_offset=0.0),
                MOVE_SPEED, MOVE_ACCEL
            )
            time.sleep(1)

            if i % 10 == 0 or i == n - 2:
                print(f"[ROBOT]   {i+1}/{n-1}  "
                      f"via=({via_xyz[0]:.3f},{via_xyz[1]:.3f},{via_xyz[2]:.3f})  "
                      f"to=({p_next[0]:.3f},{p_next[1]:.3f})")

        # ── 3. Arco finale dall'ultimo punto verso il centro ──
        #       (un solo moveL verso il via-point, poi moveJ diretto)
        p_last     = contour_3d_points[-1]
        p_last_z   = (p_last[0], p_last[1], p_last[2] + Z_CONTOUR_OFFSET_M)
        cx_3d, cy_3d, cz_3d = coord_center
        p_center_z = (cx_3d, cy_3d, cz_3d + Z_CONTOUR_OFFSET_M)

        via_final = compute_arc_via(
            p_last_z, p_center_z,
            coord_center,
            z_offset=0.0,
            lift=ARC_LIFT_M,
            center_pull=ARC_CENTER_PULL
        )

        print(f"[ROBOT] → Via-point finale verso centro: {via_final}")
        rtde_c.moveL(
            make_waypoint(*via_final, rvec_target, z_offset=0.0),
            MOVE_SPEED2, MOVE_ACCEL2
        )
        print("[ROBOT]   Arco finale completato.")

    # ── 4. Ritorna alla posa di osservazione ──
    print("[ROBOT] → Ritorno alla posa di osservazione...")
    rtde_c.moveJ(OBSERVE_RAD, 0.2, 0.2)
    print("[ROBOT]   Pronto per nuova acquisizione.")


# ═══════════════════════════════════════════
```

```
# MOVIMENTO ROBOT — SOLO 4 VERTICI (tasto V)
# ============================================
def move_to_box_vertices(coord_center, vertices_base):
    X, Y, Z = coord_center
    rvec_target, _ = compute_ready_pose()

    wp_center = make_waypoint(X, Y, Z, rvec_target, z_offset=Z_OFFSET_M)
    print(f"[ROBOT] → Centro (4-vertici): {wp_center[:3]}")
    rtde_c.moveL(wp_center, MOVE_SPEED, MOVE_ACCEL)

    if vertices_base:
        pts    = np.array(vertices_base)
        cx_b   = pts[:, 0].mean()
        cy_b   = pts[:, 1].mean()
        angles = np.arctan2(pts[:, 1] - cy_b, pts[:, 0] - cx_b)
        order  = np.argsort(angles)
        sorted_verts = [vertices_base[i] for i in order]
        n = len(sorted_verts)

        vx0, vy0, vz0 = sorted_verts[0]
        rtde_c.moveL(
            make_waypoint(vx0, vy0, vz0, rvec_target, z_offset=Z_CONTOUR_OFFSET_M),
            MOVE_SPEED, MOVE_ACCEL
        )

        for i in range(n):
            p_curr = sorted_verts[i]
            p_next = sorted_verts[(i + 1) % n]

            p_curr_z = (p_curr[0], p_curr[1], p_curr[2] + Z_CONTOUR_OFFSET_M)
            p_next_z = (p_next[0], p_next[1], p_next[2] + Z_CONTOUR_OFFSET_M)

            via_xyz = compute_arc_via(
                p_curr_z, p_next_z,
                coord_center,
                z_offset=0.0,
                lift=ARC_LIFT_M,
                center_pull=ARC_CENTER_PULL
            )

            print(f"[ROBOT]   V{i+1}→V{(i+1)%n+1}  "
                  f"via=({via_xyz[0]:.3f},{via_xyz[1]:.3f},{via_xyz[2]:.3f})")

            rtde_c.moveL(
                make_waypoint(*via_xyz, rvec_target, z_offset=0.0),
                MOVE_SPEED, MOVE_ACCEL
            )
            rtde_c.moveL(
```

```
                make_waypoint(*p_next_z, rvec_target, z_offset=0.0),
                MOVE_SPEED, MOVE_ACCEL
            )

        print("[ROBOT]   Contorno 4-vertici chiuso.")

    rtde_c.moveJ(OBSERVE_RAD, 0.2, 0.2)
    print("[ROBOT]   Pronto per nuova acquisizione.")


# ═══════════════════════════════════════
# VISUALIZZAZIONE
# ═══════════════════════════════════════
def draw_results(color_image, boxes, coords_3d, intrinsics, depth_frame,
                 robot_px_per_box=None):
    PANEL_W       = 300
    FONT          = cv2.FONT_HERSHEY_SIMPLEX
    COLOR_HDR     = (255, 220, 60)
    COLOR_VERT    = (0, 200, 255)
    COLOR_MISS    = (80, 80, 255)
    COLOR_CONT    = (255, 0, 200)
    COLOR_VERT_PX = (0, 200, 255)
    COLOR_VERT_IN = (0, 255, 180)   # colore vertici insetted (diverso dai punti
intermedi)
    LINE_H        = 22

    vis = color_image.copy()

    for i, (box, coord) in enumerate(zip(boxes, coords_3d)):
        cx, cy     = box['center']
        x, y, w, h = box['bbox']
        color      = (0, 255, 80) if coord else (0, 80, 255)

        cv2.drawContours(vis, [box['contour']], -1, color, 1)
        cv2.drawContours(vis, [box['vertices']], -1, color, 2)
        cv2.rectangle(vis, (x, y), (x + w, y + h), color, 1)
        cv2.circle(vis, (cx, cy), 6, color, -1)

        # Punti del contorno denso: vertici (verde acqua) vs intermedi (viola)
        if robot_px_per_box is not None and i < len(robot_px_per_box):
            vert_pts_px = set()
            for j, (vx, vy) in enumerate(box['vertices'].reshape(-1, 2)):
                ivx, ivy = inset_point(float(vx), float(vy), cx, cy, VERTEX_INSET_PX)
                vert_pts_px.add((ivx, ivy))

            for (spx, spy) in robot_px_per_box[i]:
                if (spx, spy) in vert_pts_px:
                    cv2.circle(vis, (int(spx), int(spy)), 5, COLOR_VERT_IN, -1)
```

```
                else:
                    cv2.circle(vis, (int(spx), int(spy)), 3, COLOR_CONT, -1)

        # Vertici stabilizzati: originale (blu chiaro) e insetted (verde acqua)
        for j, (vx, vy) in enumerate(box['vertices'].reshape(-1, 2)):
            ivx, ivy = inset_point(float(vx), float(vy), cx, cy, VERTEX_INSET_PX)
            cv2.circle(vis, (int(vx), int(vy)), 4, (100, 100, 255), 1)
            cv2.circle(vis, (ivx, ivy), 6, COLOR_VERT_IN, -1)
            cv2.putText(vis, str(j + 1), (ivx + 6, ivy - 6), FONT, 0.45, COLOR_VERT_IN,
1)

        if coord:
            X, Y, Z = coord
            label = f"Box {i+1}  X:{X:.3f} Y:{Y:.3f} Z:{Z:.3f} m"
        else:
            label = f"Box {i+1}  [depth N/A]"
        cv2.putText(vis, label, (x, max(y - 10, 15)), FONT, 0.50, color, 2)

    n_valid   = sum(1 for c in coords_3d if c is not None)
    n_robot   = sum(len(px) for px in robot_px_per_box) if robot_px_per_box else 0
    buf_sizes = {k: len(v) for k, v in _vertex_buffers.items()}
    cv2.putText(vis,
                f"Scatole: {len(boxes)}  |  Valid 3D: {n_valid}  |  "
                f"Waypoint robot: {n_robot}  |  StabBuf: {buf_sizes}",
                (10, 30), FONT, 0.5, (255, 255, 255), 2)
    info_lbl = (f"lift={ARC_LIFT_M*1000:.0f}mm pull={ARC_CENTER_PULL:.2f} "
                f"step={VERTEX_STEP}px v_in={VERTEX_INSET_PX}px
c_in={CONTOUR_INSET_PX}px "
                f"hist={VERTEX_HISTORY}fr")
    cv2.putText(vis,
                f"SPAZIO=contorno  V=4vertici  S=snap  Q=esci  D=debug  [{info_lbl}]",
                (10, 58), FONT, 0.38, (200, 200, 200), 1)

    panel = np.zeros((COLOR_H, PANEL_W, 3), dtype=np.uint8)
    row   = 20

    for i, (box, coord) in enumerate(zip(boxes, coords_3d)):
        if row > COLOR_H - LINE_H:
            break

        cv2.putText(panel, f"-- Box {i+1} --", (8, row), FONT, 0.52, COLOR_HDR, 1)
        row += LINE_H

        if coord:
            X, Y, Z = coord
            for lbl, val in [("X", X), ("Y", Y), ("Z", Z)]:
                cv2.putText(panel, f"CTR {lbl}:{val:.4f}", (8, row),
                            FONT, 0.40, (180, 255, 180), 1)
```

```
                row += LINE_H - 4
            row += 4
        else:
            cv2.putText(panel, "CTR [N/A]", (8, row), FONT, 0.42, COLOR_MISS, 1)
            row += LINE_H

        n_robot_box = (len(robot_px_per_box[i])
                       if robot_px_per_box and i < len(robot_px_per_box) else 0)
        buf_len = len(_vertex_buffers.get(i, []))
        cv2.putText(panel,
                    f"Robot: {n_robot_box} pt  buf:{buf_len}/{VERTEX_HISTORY}",
                    (8, row), FONT, 0.38, COLOR_CONT, 1)
        row += LINE_H - 2
        cv2.putText(panel,
                    f"V_inset:{VERTEX_INSET_PX}px C_inset:{CONTOUR_INSET_PX}px "
                    f"Lift:{ARC_LIFT_M*1000:.0f}mm",
                    (8, row), FONT, 0.34, (180, 180, 180), 1)
        row += LINE_H

        for j, (vx, vy) in enumerate(box['vertices'].reshape(-1, 2)):
            if row > COLOR_H - LINE_H:
                break
            ivx, ivy = inset_point(float(vx), float(vy), *box['center'],
VERTEX_INSET_PX)
            v3d = pixel_to_base(intrinsics, depth_frame, ivx, ivy) \
                  if depth_frame is not None else None

            cv2.putText(panel,
                        f"V{j+1} ({vx},{vy})→({ivx},{ivy})",
                        (8, row), FONT, 0.34, (150, 150, 150), 1)
            row += LINE_H - 6

            if v3d:
                vX, vY, vZ = v3d
                for lbl, val in [("X", vX), ("Y", vY), ("Z", vZ)]:
                    coord_color = COLOR_VERT
                    if lbl == "Z" and coord is not None:
                        coord_color = (0, 180, 255) if abs(vZ - coord[2]) < 0.005 \
                                      else (0, 80, 255)
                    cv2.putText(panel, f"   {lbl}:{val:.4f}",
                                (8, row), FONT, 0.36, coord_color, 1)
                    row += LINE_H - 6
            else:
                cv2.putText(panel, "   [depth N/A]",
                            (8, row), FONT, 0.36, COLOR_MISS, 1)
                row += LINE_H - 6

            row += 2
```

```
        row += 6

    return np.hstack([vis, panel])


# ═══════════════════════════════════════
# UTILITIES INTERNE
# ═══════════════════════════════════════
def _select_target(last_boxes, last_coords_3d):
    for box, coord in zip(last_boxes, last_coords_3d):
        if coord is not None:
            return coord, box
    print("[WARN] Nessuna scatola valida — movimento annullato.")
    return None, None


def _show_moving_overlay(vis):
    cv2.putText(vis, ">>> ROBOT IN MOVIMENTO <<<", (80, 240),
                cv2.FONT_HERSHEY_SIMPLEX, 1.0, (0, 0, 255), 3)
    cv2.imshow("RealSense D455 - Box Detection", vis)
    cv2.waitKey(1)


# ═══════════════════════════════════════
# MAIN
# ═══════════════════════════════════════
def main():
    pipeline = rs.pipeline()
    config   = rs.config()
    config.enable_stream(rs.stream.color, COLOR_W, COLOR_H, rs.format.bgr8, FPS)
    config.enable_stream(rs.stream.depth, DEPTH_W, DEPTH_H, rs.format.z16,  FPS)

    print("[INFO] Avvio RealSense D455...")
    try:
        profile = pipeline.start(config)
    except RuntimeError as e:
        print(f"[ERRORE] Impossibile avviare la camera: {e}")
        sys.exit(1)

    align         = rs.align(rs.stream.color)
    color_profile = profile.get_stream(rs.stream.color)
    intrinsics    = color_profile.as_video_stream_profile().get_intrinsics()

    decimation = rs.decimation_filter()
    decimation.set_option(rs.option.filter_magnitude, 1)
    spatial   = rs.spatial_filter()
    temporal  = rs.temporal_filter()
    hole_fill = rs.hole_filling_filter()
```

```
    snapshot_count    = 0
    last_boxes        = []
    last_coords_3d    = []
    last_depth        = None
    robot_px_per_box = []

    print("[INFO] Streaming attivo.")
    print(f"       VERTEX_HISTORY={VERTEX_HISTORY} frame")
    print(f"       ARC_LIFT_M={ARC_LIFT_M*1000:.0f}mm  ARC_CENTER_PULL={ARC_CENTER_PULL
:.2f}")
    print(f"       VERTEX_STEP={VERTEX_STEP}px")
    print(f"       VERTEX_INSET_PX={VERTEX_INSET_PX}px  CONTOUR_INSET_PX={CONTOUR_INSET
_PX}px")
    print(f"       SERIAL_INTERVAL={SERIAL_INTERVAL*1000:.0f}ms")
    print("       SPAZIO → contorno su rette tra vertici + arco finale verso centro")
    print("       V      → centro + 4 vertici (stabilizzati)")
    print("       S      → snapshot   |   Q → esci   |   D → debug depth")

    try:
        while True:
            frames  = pipeline.wait_for_frames()
            aligned = align.process(frames)

            color_frame = aligned.get_color_frame()
            depth_frame = aligned.get_depth_frame()
            if not color_frame or not depth_frame:
                continue

            depth_frame = decimation.process(depth_frame)
            depth_frame = spatial.process(depth_frame)
            depth_frame = temporal.process(depth_frame)
            depth_frame = hole_fill.process(depth_frame)
            depth_frame = depth_frame.as_depth_frame()

            # — Colorizzatore per la mappa di profondità —
            colorizer = rs.colorizer()
            colorizer.set_option(rs.option.color_scheme, 0)                  # 0 = Jet
            colorizer.set_option(rs.option.histogram_equalization_enabled, 0)  # range
lineare, non equalizzato
            colorizer.set_option(rs.option.min_distance, DEPTH_MIN_MM / 1000.0)  # 0.01
m
            colorizer.set_option(rs.option.max_distance, DEPTH_MAX_MM / 1000.0)  # 0.70
m

            depth_colormap = np.asanyarray(colorizer.colorize(depth_frame).get_data())
            cv2.imshow("Mappa di profondità (allineata)", depth_colormap)
```

```
color_image = np.asanyarray(color_frame.get_data())
cv2.imshow("immagine RGB", color_image)

boxes     = detect_boxes(color_image)
coords_3d = []
for box in boxes:
    coord = box_center_to_base(depth_frame, intrinsics, box)
    coords_3d.append(coord)
    if coord:
        cx, cy  = box['center']
        X, Y, Z = coord
        print(f"[LIVE] pixel=({cx},{cy})  X={X:.4f} Y={Y:.4f} Z={Z:.4f} m")

last_boxes     = boxes
last_coords_3d = coords_3d
last_depth     = depth_frame

robot_px_per_box = []
for box in boxes:
    px_list = build_contour_with_vertices(
        box['contour'],
        box['vertices'],
        box['center'],
        VERTEX_INSET_PX,    # inset per i vertici
        CONTOUR_INSET_PX,   # inset per i punti intermedi
        VERTEX_STEP
    )
    robot_px_per_box.append(px_list)

vis = draw_results(color_image, boxes, coords_3d, intrinsics, depth_frame,
                   robot_px_per_box)
cv2.imshow("RealSense D455 - Box Detection", vis)

key = cv2.waitKey(1) & 0xFF

if key == ord('q'):
    break

elif key == ord('s'):
    rgb_clean = color_image.copy()
    pair = np.hstack([rgb_clean, depth_colormap])
    fname = f"figura_5_3_{snapshot_count:04d}.png"
    cv2.imwrite(fname, pair)
    print(f"[INFO] Coppia RGB+depth salvata: {fname}")
    snapshot_count += 1

elif key == ord(' '):
    target_coord, target_box = _select_target(last_boxes, last_coords_3d)
```

```
        if target_coord is None:
            continue

        # ── Salvataggio frame annotato con timestamp ──
        ts    = time.strftime("%Y%m%d_%H%M%S")
        fname = f"frame_spazio_{ts}.png"
        cv2.imwrite(fname, vis)
        print(f"[INFO] Frame annotato salvato: {fname}")

        target_idx = last_boxes.index(target_box)
        final_px   = robot_px_per_box[target_idx]

        contour_3d = []
        skipped    = 0
        for (spx, spy) in final_px:
            p3d = pixel_to_base(intrinsics, last_depth, spx, spy)
            if p3d is not None:
                contour_3d.append(p3d)
            else:
                skipped += 1

        print(f"[INFO] Waypoint 3D validi: {len(contour_3d)}  "
              f"|  Saltati (depth N/A): {skipped}")
        print(f"[INFO] Centro: {target_coord}")

        _show_moving_overlay(vis)
        move_to_box_and_trace(target_coord, contour_3d)
        print("[INFO] Sequenza completata. Pronto per nuova acquisizione.")

    elif key == ord('v'):
        target_coord, target_box = _select_target(last_boxes, last_coords_3d)
        if target_coord is None:
            continue

        vertices_px   = target_box['vertices'].reshape(-1, 2)
        vertices_base = []
        for (vx, vy) in vertices_px:
            ivx, ivy = inset_point(
                float(vx), float(vy),
                *target_box['center'],
                VERTEX_INSET_PX   # usa VERTEX_INSET_PX per i vertici
            )
            p3d = pixel_to_base(intrinsics, last_depth, ivx, ivy)
            if p3d is not None:
                vertices_base.append(p3d)
                print(f"[VERTEX] px_orig=({vx},{vy})  "
                      f"px_inset=({ivx},{ivy})  base={p3d}")
            else:
```

```
                        print(f"[VERTEX] px_orig=({vx},{vy})  "
                              f"px_inset=({ivx},{ivy})  depth N/A — saltato")

                print(f"[INFO] Centro: {target_coord}  |  "
                      f"Vertici validi: {len(vertices_base)}/4")

                _show_moving_overlay(vis)
                move_to_box_vertices(target_coord, vertices_base)
                print("[INFO] Sequenza completata. Pronto per nuova acquisizione.")

            elif key == ord('d'):
                if last_boxes and last_depth:
                    box    = last_boxes[0]
                    cx, cy = box['center']

                    depth_image = np.asanyarray(last_depth.get_data())
                    raw_depth   = depth_image[cy, cx] / 1000.0

                    p_cam_xyz = rs.rs2_deproject_pixel_to_point(
                        intrinsics, [float(cx), float(cy)], raw_depth
                    )
                    p_cam = np.array([*p_cam_xyz, 1.0])

                    print(f"[DEBUG] P_cam (RealSense frame) = {p_cam[:3]}")
                    p_ee   = T_ee_cam @ p_cam
                    print(f"[DEBUG] P_ee  (EE frame)        = {p_ee[:3]}")
                    p_base = T_base_ee_obs @ p_ee
                    print(f"[DEBUG] P_base (base frame)     = {p_base[:3]}")
                    print(f"[DEBUG] TCP obs                 = {OBSERVE_TCP}")

                    T = T_base_ee_obs
                    print(f"[DEBUG] T_base_ee_obs traslazione : {T[:3,3]}")
                    print(f"[DEBUG] OBSERVE_TCP xyz           : {OBSERVE_TCP[:3]}")

                    Rmat        = T[:3,:3]
                    ortho_check = Rmat @ Rmat.T
                    print(f"[DEBUG] R @ R.T (deve essere I):\n{ortho_check}")
                    print(f"[DEBUG] det(R) = {np.linalg.det(Rmat):.6f}")

                    R_base_cam    = T_base_ee_obs[:3,:3] @ T_ee_cam[:3,:3]
                    z_cam_in_base = R_base_cam @ np.array([0, 0, 1])
                    print(f"[DEBUG] Asse Z camera nel frame base: {z_cam_in_base}")
                    print(f"[DEBUG] Componente verticale: {z_cam_in_base[2]:.4f}")

                    cam_origin   = T_base_ee_obs @ T_ee_cam @ np.array([0,0,0,1.0])
                    z_dir        = T_base_ee_obs[:3,:3] @ T_ee_cam[:3,:3] @
np.array([0,0,1])
                    depth_m      = robust_depth_at(last_depth, cx, cy)
```

```
                    p_geometrico = cam_origin[:3] + depth_m * z_dir

                    print(f"[VERIFICA] Camera origin  : {cam_origin[:3]}")
                    print(f"[VERIFICA] Z direction    : {z_dir}")
                    print(f"[VERIFICA] Depth          : {depth_m:.4f} m")
                    print(f"[VERIFICA] P geometrico   : {p_geometrico}")
                    print(f"[VERIFICA] P pipeline     : {p_base[:3]}")

                    print(f"[BIAS] Depth raw pixel singolo: {depth_image[cy,cx]} mm")
                    print(f"[BIAS] Depth robust_median ROI: "
                          f"{robust_depth_at(last_depth, cx, cy)*1000:.1f} mm")
                    print(f"[BIAS] Differenza:              "
                          f"{depth_image[cy,cx] - robust_depth_at(last_depth, cx,
cy)*1000:.1f} mm")

    finally:
        pipeline.stop()
        ser.close()
        cv2.destroyAllWindows()
        print("[INFO] Pipeline fermata.")
        rtde_c.stopScript()


if __name__ == "__main__":
    main()
```

APPENDICE C

config.h

Costanti hardware, meccanica e limiti.

```
#pragma once

// ============================================================
//  config.h  –  Costanti hardware e limiti dell'asse lineare
//  Arduino UNO R4 Minima + DM320T + Tr8x4 + UR robot
//
//  NOTA: PIN_D1..PIN_D4 sono già definiti dal framework Renesas
//  in pins_arduino.h – usiamo prefissi MTR_ e SIG_ per evitare
//  conflitti.
// ============================================================

// --- Pin driver motore ---
constexpr int MTR_STEP = 2;
constexpr int MTR_DIR  = 3;
constexpr int MTR_EN   = 4;

// --- Pin segnali robot UR (via optoisolatori) ---
constexpr int SIG_PUSH_FWD  = 8;   // D1: spinta avanti (continua, finché HIGH)
constexpr int SIG_PULL_BWD  = 9;   // D2: ritorno indietro (continuo, finché HIGH)
constexpr int SIG_ENDSTOP   = 10;  // D3: finecorsa fisico 100mm (reed via robot I/O)
```

```
constexpr int SIG_GOTO_ZERO = 11;  // D4: torna a 0mm

// Impostare true se gli optoisolatori invertono la logica
constexpr bool SIGNALS_ACTIVE_LOW = false;

// --- Meccanica ---
constexpr float SCREW_LEAD_MM  = 4.0f;   // passo vite Tr8x4
constexpr int   DRIVER_PPR     = 400;     // pulse/rev impostati su DM320T
constexpr float STEPS_PER_MM   = static_cast<float>(DRIVER_PPR) / SCREW_LEAD_MM; // 100 step/mm
constexpr float AXIS_MAX_MM    = 100.0f;

// --- Direzioni motore ---
constexpr bool DIR_FORWARD  = true;
constexpr bool DIR_BACKWARD = false;

// --- Timing impulso STEP ---
// DM320T: pulse width minimo ~2.5µs; usiamo 5µs per margine
constexpr unsigned int STEP_PULSE_US = 5;

// --- Limiti velocità ---
// MAX_DELAY_US → velocità minima assoluta (~1 mm/s a 100 step/mm)
// MIN_DELAY_US → velocità massima:
//   100 step/mm: 200µs/step = 5000 step/s = 50 mm/s
//   Abbassare MIN_DELAY_US aumenta la velocità massima (verificare fisicamente).
constexpr unsigned int MAX_DELAY_US = 10000;  // ~1  mm/s
constexpr unsigned int MIN_DELAY_US = 200;    // ~50 mm/s (massimo consigliato)

// --- Rampa ---
constexpr long  RAMP_STEPS_MIN    =  10;
constexpr long  RAMP_STEPS_MAX    = 200;
constexpr float RAMP_START_FACTOR = 4.0f;

// --- EEPROM addresses (float = 4 byte ciascuno) ---
constexpr int EEPROM_ADDR_HOME     =  0;
constexpr int EEPROM_ADDR_SPEED    =  4;
constexpr int EEPROM_ADDR_POSITION =  8;

// --- Valori di default se EEPROM non inizializzata ---
constexpr float DEFAULT_HOME_MM    =  0.0f;
constexpr float DEFAULT_SPEED_MM_S = 30.0f;
```

settings.h

Interfaccia del modulo di persistenza.

```
#pragma once
#include <Arduino.h>

// ============================================================
//  settings.h  —  Parametri configurabili e persistenza EEPROM
// ============================================================

// Parametri globali (definiti in settings.cpp)
extern float homePositionMM;
extern float defaultSpeedMM_s;
extern float currentPositionMM;

void settingsLoad();
void settingsSave();
void settingsSavePosition();
void settingsPrint();
```

signals.h

Interfaccia per la lettura dei segnali.

```
#pragma once
#include <Arduino.h>
```

```
3
4  // ============================================================
5  //  signals.h  –  Lettura segnali robot UR via optoisolatori
6  // ============================================================
7
8  void signalsInit();
9  bool signalRead(int pin);
```

axis.h

Interfaccia del modulo di movimento.

```
1  #pragma once
2  #include <Arduino.h>
3
4  // ============================================================
5  //  axis.h  –  Controllo movimento asse lineare
6  // ============================================================
7
8  void axisInit();
9  void axisEnable();
10 void axisDisable();
11
12 // Movimento assoluto verso targetMM a velocità mm_s
13 void axisMoveTo(float targetMM, float mm_s);
14
15 // Movimento relativo di deltaMM a velocità mm_s
16 void axisMoveRel(float deltaMM, float mm_s);
17
18 // Spinta avanti continua finché il segnale sul pin rimane alto
19 void axisPushBySignal(int signalPin, float mm_s);
20
21 // Spinta avanti per una durata fissa in millisecondi
22 void axisPushByTime(unsigned long ms, float mm_s);
23
24 // Conversione velocità → delay tra step (µs)
25 unsigned int axisSpeedToDelay(float mm_s);
```

serial_cmd.h

Interfaccia del parser dei comandi.

```
1  #pragma once
2
3  // ============================================================
4  //  serial_cmd.h  –  Parser comandi seriali e menu interattivi
5  // ============================================================
6
7  // Chiama in loop() per gestire i comandi in arrivo
8  void serialCmdPoll();
9
10 // Indica se la modalità manutenzione è attiva
11 //  (usato in loop() per sopprimere i segnali robot)
12 bool maintenanceModeActive();
```

main.cpp

setup() e loop() principale.

```
1  #include <Arduino.h>
2  #include "config.h"
3  #include "settings.h"
4  #include "signals.h"
5  #include "axis.h"
6  #include "serial_cmd.h"
```

```
 7
 8  // ============================================================
 9  //  main.cpp  —  Setup e loop principale
10  //
11  //  Segnali robot:
12  //    D1 = spinta avanti continua (finché HIGH)
13  //    D2 = ritorno indietro continuo (finché HIGH)
14  //    D3 = finecorsa fisico 100mm (reed via robot I/O → Arduino)
15  //    D4 = torna a 0mm
16  // ============================================================
17
18  void setup() {
19      Serial.begin(115200);
20      delay(500);
21
22      axisInit();
23      signalsInit();
24
25      axisEnable();
26
27      settingsLoad();
28
29      Serial.println();
30      Serial.println(F("============================="));
31      Serial.println(F(" Asse lineare UR — pronto"));
32      Serial.println(F("============================="));
33      settingsPrint();
34      Serial.println(F("'help' per i comandi  |  'setup' per configurazione e movimento manuale"));
35      Serial.println();
36  }
37
38  void loop() {
39      // Comandi seriali (ha priorità; può bloccare il loop se si entra nei menu)
40      serialCmdPoll();
41
42      // Segnali robot ignorati durante la manutenzione
43      if (maintenanceModeActive()) return;
44
45      // D1: spinta avanti continua
46      if (signalRead(SIG_PUSH_FWD)) {
47          axisPushBySignal(SIG_PUSH_FWD, defaultSpeedMM_s);
48          // Piccolo debounce per non rientrare subito
49          delay(50);
50          return;
51      }
52
53      // D2: ritorno indietro continuo
54      if (signalRead(SIG_PULL_BWD)) {
55          // Muoviamo un passo alla volta finché D2 rimane alto
56          while (signalRead(SIG_PULL_BWD)) {
57              float target = currentPositionMM - (1.0f / STEPS_PER_MM);
58              if (target < 0.0f) break;
59              // Un singolo step alla volta con delay diretto
60              unsigned int d = axisSpeedToDelay(defaultSpeedMM_s);
61              digitalWrite(MTR_DIR, DIR_BACKWARD);
62              delayMicroseconds(50);
63              digitalWrite(MTR_STEP, HIGH);
64              delayMicroseconds(STEP_PULSE_US);
65              digitalWrite(MTR_STEP, LOW);
66              unsigned int rest = (d > STEP_PULSE_US) ? (d - STEP_PULSE_US) : 1;
67              delayMicroseconds(rest);
68              currentPositionMM = target;
69          }
70          settingsSavePosition();
71          Serial.print(F("Ritorno terminato. Pos: "));
72          Serial.print(currentPositionMM, 2);
73          Serial.println(F(" mm"));
74          delay(50);
75          return;
76      }
77
78      // D3: finecorsa fisico — aggiorna posizione logica
79      if (signalRead(SIG_ENDSTOP)) {
80          Serial.println(F("D3: Finecorsa 100mm, posizione aggiornata."));
81          currentPositionMM = AXIS_MAX_MM;
82          settingsSavePosition();
83          while (signalRead(SIG_ENDSTOP)) delay(10);
```

```
        return;
    }

    // D4: torna a 0mm
    if (signalRead(SIG_GOTO_ZERO)) {
        Serial.println(F("D4: -> 0 mm"));
        axisMoveTo(0.0f, defaultSpeedMM_s);
        while (signalRead(SIG_GOTO_ZERO)) delay(10);
        return;
    }
}
```

axis.cpp

Movimento, rampe e singolo passo.

```
#include "axis.h"
#include "config.h"
#include "settings.h"
#include "signals.h"
#include <math.h>

// ============================================================
//  axis.cpp
// ============================================================

// --- Init / enable / disable ---

void axisInit() {
    pinMode(MTR_STEP, OUTPUT);
    pinMode(MTR_DIR,  OUTPUT);
    pinMode(MTR_EN,   OUTPUT);
    axisDisable();
}

void axisEnable()  { digitalWrite(MTR_EN, LOW);  }
void axisDisable() { digitalWrite(MTR_EN, HIGH); }

// --- Conversione velocità ---

unsigned int axisSpeedToDelay(float mm_s) {
    if (mm_s <= 0.0f) mm_s = 0.1f;
    float stepsPerSec = mm_s * STEPS_PER_MM;
    unsigned int d = static_cast<unsigned int>(1'000'000.0f / stepsPerSec);
    if (d < MIN_DELAY_US) d = MIN_DELAY_US;
    if (d > MAX_DELAY_US) d = MAX_DELAY_US;
    return d;
}

// --- Singolo step ---
// Ritorna false se il movimento è stato bloccato (limite SW o finecorsa HW).

static bool doOneStep(bool direction, unsigned int delayUS) {
    float delta   = direction ? (1.0f / STEPS_PER_MM) : -(1.0f / STEPS_PER_MM);
    float nextPos = currentPositionMM + delta;

    // Limite software indietro
    if (!direction && nextPos < 0.0f) return false;

    // Finecorsa fisico in avanti (D3)
    if (direction && signalRead(SIG_ENDSTOP)) {
        currentPositionMM = AXIS_MAX_MM;
        settingsSavePosition();
        Serial.println(F("Finecorsa 100mm: posizione aggiornata."));
        return false;
    }

    digitalWrite(MTR_DIR, direction);
    digitalWrite(MTR_STEP, HIGH);
    delayMicroseconds(STEP_PULSE_US);
    digitalWrite(MTR_STEP, LOW);

    unsigned int rest = (delayUS > STEP_PULSE_US) ? (delayUS - STEP_PULSE_US) : 1;
```

```
    delayMicroseconds(rest);

    currentPositionMM = nextPos;
    return true;
}

// --- Calcolo delay rampa ---

static unsigned int rampDelay(unsigned int startDelay, unsigned int targetDelay,
                              long step, long totalSteps, long rampSteps) {
    if (step < rampSteps) {
        // Accelerazione
        return startDelay - static_cast<unsigned int>(
            static_cast<float>(startDelay - targetDelay) * step / rampSteps);
    }
    if (step > (totalSteps - rampSteps)) {
        // Decelerazione
        return targetDelay + static_cast<unsigned int>(
            static_cast<float>(startDelay - targetDelay) * (step - (totalSteps - rampSteps)) /
rampSteps);
    }
    return targetDelay;
}

static long calcRampSteps(long totalSteps) {
    long r = totalSteps / 5;
    if (r < RAMP_STEPS_MIN) r = RAMP_STEPS_MIN;
    if (r > RAMP_STEPS_MAX) r = RAMP_STEPS_MAX;
    return r;
}

// --- Movimento con rampa (relativo, mm) ---

static void moveMM(float mm, float mm_s) {
    if (fabsf(mm) < 0.001f) return;

    long totalSteps = lroundf(fabsf(mm) * STEPS_PER_MM);
    bool direction  = (mm >= 0.0f) ? DIR_FORWARD : DIR_BACKWARD;

    digitalWrite(MTR_DIR, direction);
    delayMicroseconds(50);

    unsigned int targetDelay = axisSpeedToDelay(mm_s);
    unsigned int startDelay  = static_cast<unsigned int>(targetDelay * RAMP_START_FACTOR);
    if (startDelay > MAX_DELAY_US) startDelay = MAX_DELAY_US;

    long rampSteps = calcRampSteps(totalSteps);

    for (long i = 0; i < totalSteps; i++) {
        unsigned int d = rampDelay(startDelay, targetDelay, i, totalSteps, rampSteps);
        if (!doOneStep(direction, d)) break;
    }

    settingsSavePosition();
}

// --- API pubblica ---

void axisMoveTo(float targetMM, float mm_s) {
    targetMM = constrain(targetMM, 0.0f, AXIS_MAX_MM);
    float delta = targetMM - currentPositionMM;
    if (fabsf(delta) < 0.01f) {
        Serial.println(F("Gia' in posizione."));
        return;
    }
    Serial.print(F("-> ")); Serial.print(targetMM, 2);
    Serial.print(F(" mm  (da ")); Serial.print(currentPositionMM, 2); Serial.println(F(" mm)"));

    moveMM(delta, mm_s);

    Serial.print(F("Pos: ")); Serial.print(currentPositionMM, 2); Serial.println(F(" mm"));
}

void axisMoveRel(float deltaMM, float mm_s) {
    float target = currentPositionMM + deltaMM;
    target = constrain(target, 0.0f, AXIS_MAX_MM);
    float actualDelta = target - currentPositionMM;
```

```
    if (fabsf(actualDelta) < 0.01f) {
        Serial.println(F("Gia' in posizione."));
        return;
    }
    Serial.print(F("Rel ")); Serial.print(deltaMM, 2); Serial.println(F(" mm"));
    moveMM(actualDelta, mm_s);
    Serial.print(F("Pos: ")); Serial.print(currentPositionMM, 2); Serial.println(F(" mm"));
}

// --- Spinta continua con rampa di accelerazione ---
//   Comune tra signal-driven e time-driven: il loop si ferma
//   quando la condizione esterna viene meno oppure si tocca il finecorsa.

static void pushLoop(float mm_s, unsigned long timeLimitMs, int signalPin) {
    // timeLimitMs > 0  → modalità temporizzata
    // signalPin   >= 0 → modalità segnale

    digitalWrite(MTR_DIR, DIR_FORWARD);
    delayMicroseconds(50);

    unsigned int targetDelay = axisSpeedToDelay(mm_s);
    unsigned int startDelay  = static_cast<unsigned int>(targetDelay * RAMP_START_FACTOR);
    if (startDelay > MAX_DELAY_US) startDelay = MAX_DELAY_US;

    unsigned long startTime = millis();
    long stepCount = 0;

    while (true) {
        // Condizione di uscita
        if (timeLimitMs > 0 && (millis() - startTime) >= timeLimitMs) break;
        if (signalPin >= 0 && !signalRead(signalPin)) break;

        unsigned int d;
        if (stepCount < RAMP_STEPS_MAX) {
            d = startDelay - static_cast<unsigned int>(
                static_cast<float>(startDelay - targetDelay) * stepCount / RAMP_STEPS_MAX);
        } else {
            d = targetDelay;
        }

        if (!doOneStep(DIR_FORWARD, d)) break;
        stepCount++;
    }

    settingsSavePosition();
    Serial.print(F("Spinta terminata. Pos: "));
    Serial.print(currentPositionMM, 2);
    Serial.println(F(" mm"));
}

void axisPushBySignal(int signalPin, float mm_s) {
    Serial.print(F("Spinta avanti (segnale)  vel: "));
    Serial.print(mm_s, 1); Serial.println(F(" mm/s"));
    pushLoop(mm_s, 0, signalPin);
}

void axisPushByTime(unsigned long ms, float mm_s) {
    Serial.print(F("Spinta avanti "));
    Serial.print(ms); Serial.print(F(" ms  vel: "));
    Serial.print(mm_s, 1); Serial.println(F(" mm/s"));
    pushLoop(mm_s, ms, -1);
}
```

signals.cpp

Lettura degli ingressi digitali.

```
#include "signals.h"
#include "config.h"

// ============================================================
//  signals.cpp
// ============================================================
```

```

void signalsInit() {
    pinMode(SIG_PUSH_FWD, INPUT);
    pinMode(SIG_PULL_BWD, INPUT);
    pinMode(SIG_ENDSTOP, INPUT);
    pinMode(SIG_GOTO_ZERO, INPUT);
}

bool signalRead(int pin) {
    bool raw = (digitalRead(pin) == HIGH);
    return SIGNALS_ACTIVE_LOW ? !raw : raw;
}
```

serial_cmd.cpp

Parser comandi e menu di setup.

```
#include "serial_cmd.h"
#include "config.h"
#include "settings.h"
#include "axis.h"
#include <Arduino.h>

// ============================================================
//  serial_cmd.cpp
// ============================================================

static bool s_maintenanceActive = false;

bool maintenanceModeActive() { return s_maintenanceActive; }

// ============================================================
//  Menu SETUP  (configurazione + movimento manuale unificati)
//  Mentre il menu è aperto i segnali robot sono IGNORATI.
// ============================================================

static void printSetupHelp() {
    Serial.println();
    Serial.println(F("============================"));
    Serial.println(F("          MENU SETUP        "));
    Serial.println(F("  segnali robot IGNORATI     "));
    Serial.println(F("============================"));
    Serial.println(F("--- Configurazione ---"));
    Serial.println(F("  sethome <mm>   -> imposta HOME            es: sethome 5"));
    Serial.println(F("  setspeed <v>   -> velocita' default mm/s   es: setspeed 30"));
    Serial.println(F("  info           -> mostra impostazioni"));
    Serial.println(F("--- Movimento manuale ---"));
    Serial.println(F("  goto <mm>      -> posizione assoluta       es: goto 50"));
    Serial.println(F("  move <mm>      -> movimento relativo       es: move -10"));
    Serial.println(F("  home           -> vai alla HOME"));
    Serial.println(F("  zero           -> torna a 0 mm"));
    Serial.println(F("  fc             -> vai al finecorsa (100mm)"));
    Serial.println(F("  pos            -> posizione attuale"));
    Serial.println(F("  setpos <mm>    -> forza posizione logica   es: setpos 0"));
    Serial.println(F("  speed <v>      -> velocita' sessione (non salvata)"));
    Serial.println(F("--- Altro ---"));
    Serial.println(F("  help / ?       -> questo messaggio"));
    Serial.println(F("  fine           -> esci dal setup"));
    Serial.println();
}

static void menuSetup() {
    s_maintenanceActive = true;
    float speedSES = defaultSpeedMM_s;  // velocita' locale di sessione, non persistente

    Serial.println(F("*** SETUP ATTIVO - segnali robot IGNORATI ***"));
    printSetupHelp();

    unsigned long deadline = millis() + 300000UL;

    while (true) {
        if (!Serial.available()) {
```

```
            if (millis() > deadline) {
                Serial.println(F("Timeout, uscita automatica dal setup."));
                break;
            }
            continue;
        }

        String cmd = Serial.readStringUntil('\n');
        cmd.trim();
        deadline = millis() + 300000UL;
        if (cmd.length() == 0) continue;

        if (cmd.equalsIgnoreCase("fine")) {
            Serial.println(F("Uscita dal setup. Segnali robot ATTIVI."));
            break;
        }
        else if (cmd.startsWith("sethome")) {
            float v = cmd.substring(7).toFloat();
            if (v >= 0.0f && v <= AXIS_MAX_MM) {
                homePositionMM = v;
                settingsSave();
                Serial.print(F("HOME -> ")); Serial.print(homePositionMM, 2); Serial.println(F("
mm (salvato)"));
            } else { Serial.println(F("Fuori range [0-100], ignorato.")); }
        }
        else if (cmd.startsWith("setspeed")) {
            float v = cmd.substring(8).toFloat();
            if (v > 0.0f && v <= 200.0f) {
                defaultSpeedMM_s = v;
                speedSES = v;
                settingsSave();
                Serial.print(F("Speed default -> ")); Serial.print(defaultSpeedMM_s, 2);
Serial.println(F(" mm/s (salvato)"));
            } else { Serial.println(F("Valore non valido [0.1-200], ignorato.")); }
        }
        else if (cmd.equalsIgnoreCase("info")) {
            settingsPrint();
            Serial.print(F("  Speed sessione : ")); Serial.print(speedSES, 2); Serial.println(F("
mm/s (non salvata)"));
        }
        else if (cmd.equalsIgnoreCase("pos")) {
            Serial.print(F("Pos: ")); Serial.print(currentPositionMM, 3); Serial.println(F("
mm"));
        }
        else if (cmd.equalsIgnoreCase("home")) {
            axisMoveTo(homePositionMM, speedSES);
        }
        else if (cmd.equalsIgnoreCase("zero")) {
            axisMoveTo(0.0f, speedSES);
        }
        else if (cmd.equalsIgnoreCase("fc")) {
            axisMoveTo(AXIS_MAX_MM, speedSES);
        }
        else if (cmd.startsWith("goto")) {
            float t = cmd.substring(4).toFloat();
            if (t >= 0.0f && t <= AXIS_MAX_MM) axisMoveTo(t, speedSES);
            else { Serial.print(F("Fuori range [0-")); Serial.print(AXIS_MAX_MM, 0);
Serial.println(F("]")); }
        }
        else if (cmd.startsWith("move")) {
            float d = cmd.substring(4).toFloat();
            axisMoveRel(d, speedSES);
        }
        else if (cmd.startsWith("setpos")) {
            float p = cmd.substring(6).toFloat();
            if (p >= 0.0f && p <= AXIS_MAX_MM) {
                currentPositionMM = p;
                settingsSavePosition();
                Serial.print(F("Posizione logica forzata a ")); Serial.print(p, 2);
Serial.println(F(" mm"));
            } else { Serial.println(F("Fuori range, ignorato.")); }
        }
        else if (cmd.startsWith("speed")) {
            float v = cmd.substring(5).toFloat();
            if (v > 0.0f && v <= 200.0f) {
                speedSES = v;
```

```
                Serial.print(F("Speed sessione: ")); Serial.print(v, 1); Serial.println(F(" mm/s
(usa setspeed per salvare)"));
            } else { Serial.println(F("Valore non valido [0.1-200].")); }
        }
        else if (cmd.equalsIgnoreCase("help") || cmd.equals("?")) {
            printSetupHelp();
        }
        else {
            Serial.print(F("Sconosciuto: '")); Serial.print(cmd); Serial.println(F("'  (help per i
comandi)"));
        }
    }

    s_maintenanceActive = false;
}

// ============================================================
//  Parser comandi principali (chiamato da loop)
// ============================================================

static void printHelp() {
    Serial.println(F("Comandi:"));
    Serial.println(F("  S <vel> <ms>          -> spinta per ms millisecondi a vel mm/s"));
    Serial.println(F("  SR <vel> <ms> <mm>    -> spinta + ritorno relativo di mm"));
    Serial.println(F("  Z                     -> torna a 0 mm"));
    Serial.println(F("  F                     -> vai al finecorsa (100mm)"));
    Serial.println(F("  HOME                  -> vai alla HOME"));
    Serial.println(F("  pos                   -> posizione attuale"));
    Serial.println(F("  setup                 -> menu setup e manutenzione"));
    Serial.println(F("  help / ?              -> questo messaggio"));
}

void serialCmdPoll() {
    if (!Serial.available()) return;

    String cmd = Serial.readStringUntil('\n');
    cmd.trim();
    if (cmd.length() == 0) return;

    if (cmd.equalsIgnoreCase("setup")) {
        menuSetup();
        return;
    }
    if (cmd.equalsIgnoreCase("pos")) {
        Serial.print(F("Pos: ")); Serial.print(currentPositionMM, 3); Serial.println(F(" mm"));
        return;
    }
    if (cmd.equalsIgnoreCase("Z") || cmd.equalsIgnoreCase("zero")) {
        Serial.println(F("-> 0 mm"));
        axisMoveTo(0.0f, defaultSpeedMM_s);
        return;
    }
    if (cmd.equalsIgnoreCase("F") || cmd.equalsIgnoreCase("fc")) {
        Serial.println(F("-> Finecorsa (100mm)"));
        axisMoveTo(AXIS_MAX_MM, defaultSpeedMM_s);
        return;
    }
    if (cmd.equalsIgnoreCase("HOME")) {
        Serial.println(F("-> HOME"));
        axisMoveTo(homePositionMM, defaultSpeedMM_s);
        return;
    }

    // SR <vel> <ms> <mm>
    if (cmd.startsWith("SR ") || cmd.startsWith("sr ")) {
        String params = cmd.substring(3);
        params.trim();
        int s1 = params.indexOf(' ');
        if (s1 < 0) { Serial.println(F("Uso: SR <vel> <ms> <mm>")); return; }
        float vel = params.substring(0, s1).toFloat();
        String rest = params.substring(s1 + 1);
        rest.trim();
        int s2 = rest.indexOf(' ');
        if (s2 < 0) { Serial.println(F("Uso: SR <vel> <ms> <mm>")); return; }
        unsigned long ms = rest.substring(0, s2).toInt();
        float retractMM  = rest.substring(s2 + 1).toFloat();
```

```
200         if (vel <= 0.0f || ms == 0 || retractMM < 0.0f) { Serial.println(F("Parametri non
validi.")); return; }
201         axisPushByTime(ms, vel);
202         if (retractMM > 0.01f) {
203             float target = currentPositionMM - retractMM;
204             if (target < 0.0f) target = 0.0f;
205             Serial.print(F("Ritorno -> ")); Serial.print(target, 2); Serial.println(F(" mm"));
206             axisMoveTo(target, defaultSpeedMM_s);
207         }
208         return;
209     }
210
211     // S <vel> <ms>
212     if (cmd.startsWith("S ") || cmd.startsWith("s ")) {
213         String params = cmd.substring(2);
214         params.trim();
215         int s1 = params.indexOf(' ');
216         if (s1 < 0) { Serial.println(F("Uso: S <vel> <ms>")); return; }
217         float vel        = params.substring(0, s1).toFloat();
218         unsigned long ms = params.substring(s1 + 1).toInt();
219         if (vel <= 0.0f || ms == 0) { Serial.println(F("Parametri non validi.")); return; }
220         axisPushByTime(ms, vel);
221         return;
222     }
223
224     if (cmd.equalsIgnoreCase("help") || cmd.equals("?")) {
225         printHelp();
226         return;
227     }
228
229     Serial.print(F("Sconosciuto: '")); Serial.print(cmd); Serial.println(F("'  (help per i
comandi)"));
230 }
```

settings.cpp

Lettura/scrittura EEPROM.

```
1  #include "settings.h"
2  #include "config.h"
3  #include <EEPROM.h>
4  #include <math.h>
5
6  // ============================================================
7  //  settings.cpp
8  // ============================================================
9
10 float homePositionMM  = DEFAULT_HOME_MM;
11 float defaultSpeedMM_s = DEFAULT_SPEED_MM_S;
12 float currentPositionMM = 0.0f;
13
14 // --- Helpers EEPROM ---
15 static void eepromWriteFloat(int addr, float val) {
16     auto* p = reinterpret_cast<byte*>(&val);
17     for (int i = 0; i < 4; i++) EEPROM.write(addr + i, p[i]);
18 }
19
20 static float eepromReadFloat(int addr) {
21     float val;
22     auto* p = reinterpret_cast<byte*>(&val);
23     for (int i = 0; i < 4; i++) p[i] = EEPROM.read(addr + i);
24     return val;
25 }
```

```
 26
 27  // --- API pubblica ---
 28
 29  void settingsLoad() {
 30      float h   = eepromReadFloat(EEPROM_ADDR_HOME);
 31      float spd = eepromReadFloat(EEPROM_ADDR_SPEED);
 32      float pos = eepromReadFloat(EEPROM_ADDR_POSITION);
 33
 34      if (!isnan(h)   && h   >= 0.0f && h   <= AXIS_MAX_MM)  homePositionMM   = h;
 35      if (!isnan(spd) && spd >  0.0f && spd <= 200.0f)        defaultSpeedMM_s = spd;
 36
 37      if (!isnan(pos) && pos >= 0.0f && pos <= AXIS_MAX_MM) {
 38          currentPositionMM = pos;
 39          Serial.print(F("Posizione ripristinata da EEPROM: "));
 40          Serial.print(currentPositionMM, 2);
 41          Serial.println(F(" mm"));
 42      } else {
 43          currentPositionMM = 0.0f;
 44      }
 45  }
 46
 47  void settingsSave() {
 48      eepromWriteFloat(EEPROM_ADDR_HOME,     homePositionMM);
 49      eepromWriteFloat(EEPROM_ADDR_SPEED,    defaultSpeedMM_s);
 50      eepromWriteFloat(EEPROM_ADDR_POSITION, currentPositionMM);
 51  }
 52
 53  void settingsSavePosition() {
 54      eepromWriteFloat(EEPROM_ADDR_POSITION, currentPositionMM);
 55  }
 56
 57  void settingsPrint() {
 58      Serial.println();
 59      Serial.print(F("  HOME          : ")); Serial.print(homePositionMM,    2); Serial.println(F("
mm"));
 60      Serial.print(F("  Velocita' def.: ")); Serial.print(defaultSpeedMM_s,  2); Serial.println(F("
mm/s"));
 61      Serial.print(F("  Pos. attuale  : ")); Serial.print(currentPositionMM, 2); Serial.println(F("
mm"));
 62      Serial.println();
 63  }
```