


UNIVERSITÀ DEGLI STUDI DI BERGAMO

SCUOLA DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica

Classe n. L-8

Setup, studio e controllo di un manipolatore: il robot industriale ABB IRB120.

Relatore: Chiar.mo Prof. Previdi Fabio

Prova finale di Michele Piffari

Matricola n. 1040658

**ANNO
ACADEMICO**

2017/2018



*"The more we do, the more we can do, the more busy we are, the more
leisure we have"*
William Hazlitt

Indice

1	Introduzione	1
I	Il sistema	3
2	Il manipolatore IRB 120	5
2.1	Morfologia	6
2.2	Caratteristiche tecniche principali	7
2.2.1	Intervalli di movimento	7
2.2.2	Dimensioni fisiche	8
2.2.3	Range di lavoro e raggio di curvatura	8
2.2.4	Prestazioni	9
2.3	Cinematica	9
2.3.1	Sistemi di coordinate	9
2.3.2	Descrizione di posizione e orientamento di un sistema di riferimento	11
2.3.3	Cinematica diretta IRB 120	17
2.3.4	Cinematica inversa IRB 120	22
2.3.5	Posizione dell'end-effector: definizione variabili di giunto $\vartheta_1 - \vartheta_2 - \vartheta_3$	23
2.3.6	Orientamento dell'end-effector: definizione variabili di giunto $\vartheta_4 - \vartheta_5 - \vartheta_6$	26
3	Il controllore IRC5 Compact	31
3.1	Pannello di controllo	32
3.1.1	Pulsanti e interruttori	32
3.1.2	Interfacce di collegamento	33
3.1.3	Morsettiera di supporto	36
II	Il controllo del sistema	37
4	Movimentazione manuale	39

5	Movimentazione automatica	41
6	RobotStudio: programmazione statica	43
7	Programmazione dinamica: case study	47
8	ROS: <i>Robot Operating System</i>	49
8.1	ROS industrial e IRB 120	53
8.2	Comunicazione socket: concetti base	55
8.3	Messaggio scambiato tramite socket	57
8.4	Client part	60
8.5	Architettura ABB ROS Server	63
8.6	Server part	65
8.6.1	ROS_MotionServer	65
8.6.2	ROS_StateServer	66
8.6.3	ROS_Motion: effettiva movimentazione del manipolatore	67
8.6.4	ROS_Socket: apertura della connessione socket lato server	69
8.6.5	ROS_messages	70
9	Estensioni future	73

Elenco delle figure

2.1	Manipolatore ABB IRB120 (<i>a</i>) e relativo modello matematico (<i>b</i>)	6
2.2	Morfologia del manipolatore industriale in similitudine con la morfologia del braccio umano	7
2.3	Dimensioni fisiche espresse in mm dei bracci del manipolatore IRB 120	8
2.4	Range di lavoro (<i>a</i>) e raggio di curvatura (<i>b</i>)	8
2.5	Cinematica diretta e inversa[13]	9
2.6	Sistemi di riferimento per la gestione della traiettoria del manipolatore	10
2.7	Sistema di coordinate polso	11
2.8	Tool Center Point	12
2.9	Catena cinematica IRB 120	12
2.10	Rotazione di una terna rispetto ad un'altra	13
2.11	Semplice esempio di rotazione di sistemi di riferimento	14
2.12	Rotazioni elementari	15
2.13	Composizione di rotazioni in terna fissa e mobile [17]	16
2.14	Esempio di catena cinematica aperta formata da n bracci [17]	17
2.15	Posizionamento sistemi di riferimento solidale per due bracci qualisiasi	18
2.16	Rappresentazione dei parametri di Denavit-Hartenberg	20
2.17	Soluzione cinematica diretta manipolatore ABB IRB 120	21
2.18	Input problema cinematico inverso	22
2.19	Coordinate del centro del polso del manipolatore	23
2.20	Triangoli di supporto per il calcolo del angolo ϑ_2	24
2.21	Triangoli di supporto per il calcolo del angolo ϑ_3	26
2.22	Step iniziale per il calcolo algebrico dell'orientamento del polso	27
2.23	Sistemi di riferimento dei giunti utilizzati per trovare l'orientamento del polso	27
2.24	Caratteristiche della funzione atan_2	30
3.1	IRC5 Compact controller	31

3.2	Pulsanti e interruttori presenti sul pannello anteriore del controller IRC5	33
3.3	Connettori presenti sul pannello anteriore del controller IRC5	34
3.4	Porte di comunicazione IRC5	35
3.5	Collegamenti I/O dell'IRC 5 Compact	36
3.6	Collegamento alimentazione esterna dalla morsettiera dell'IRC5 Compact	36
4.1	Componenti principali della FlexPendant	39
4.2	Schermata di controllo della movimentazione manuale tramite joystick	40
5.1	Logo RobotStudio	41
5.2	Schema del funzionamento online del sistema di controllo automatico	42
6.1	Virtualizzazione del sistema fisico su piattaforma <i>RobotStudio</i>	43
6.2	Esemplificazione del livello di complessità dei sistemi simulabili in <i>RobotStudio</i>	44
6.3	Schema base della struttura del programma <i>RAPID</i>	45
7.1	Configurazione di una comunicazione OPC tra client e server	48
8.1	<i>Robot Operating System</i>	49
8.2	Struttura a componenti di ROS	49
8.3	Concetti fondamentali su cui si basa una "network" ROS	50
8.4	Elementi principali dell'architettura ROS	51
8.5	Comunicazione basata sul protocollo <i>publisher-subscribers</i>	52
8.6	Struttura completa del sistema utilizzato	52
8.7	ROS Industrial	53
8.8	Architettura ad alto livello di ROS Industrial	54
8.9	Schematizzazione della comunicazione socket tra client e server con relativo flusso di dati	55
8.10	Configurazione schematica della comunicazione socket	56
8.11	Struttura del messaggio scambiato tra client e server	60
8.12	Funzionalità del nodo <i>joint_trajectory_action</i>	61
8.13	Grafo delle dipendenze tra topics e nodi	63
8.14	Struttura multitasking del server ROS implementato in codice RAPID	64
8.15	Configurazioni dei task del ROSServer	65

Elenco del codice

2.1	Calcolo del parametro di giunto ϑ_2	25
2.2	Calcolo del parametro di giunto ϑ_3	25
2.3	Matrice di trasformazione omogenea dei primi tre giunti	26
2.4	Calcolo di ϑ_4	28
2.5	Calcolo di ϑ_5	29
2.6	Calcolo di ϑ_6	29
6.1	Esempio di programmazione <i>RAPID</i>	46
8.1	Semplice esempio di comunicazione publish-subscribe	53
8.2	Istruzione RAPID di creazione del socket	56
8.3	Definizione dei parametri del campo MSG_TYPE	58
8.4	Definizione dei parametri del campo COMM_TYPE	58
8.5	Definizione dei parametri del campo REPLY_CODE	59
8.6	Definizione dei codici con uso speciale	59
8.7	Definizione dei parametri del campo REPLY_CODE	59
8.8	Pubblicazione messaggio <i>trajectory</i> di esempio nel topic corrispondente	62
8.9	Ricezione delle traiettorie con relativo controllo d'integrità	65
8.10	Struttura che permette di riconoscere la natura del punto da aggiungere alla traiettoria in esame	66
8.11	Feedback continuo della posizione del manipolatore	67
8.12	Movimentazione effettiva del manipolatore	67
8.13	Analisi funzionamento routine ROS-init-socket	69
8.14	Analisi funzionamento routine ROS-wait-for-client	70
8.15	Campi caratterizzanti il tipo di variabili ROS-msg	71
8.16	Campi caratterizzanti il tipo di variabili ROS-msg-joint-traj-pt	71
8.17	Unpacking del messaggio ricevuto dal client	71
8.18	Packing del messaggio per spedizione verso il client come mezzo di feedback	72

ELENCO DEL CODICE

Capitolo 1

Introduzione

Questo progetto si inserisce all'interno del percorso di ricerca avviato dal laboratorio del CAL (*Control systems and Automation Laboratory*) dell'Università degli studi di Bergamo, il cui obiettivo è quello di creare un sistema autonomo di pick and place da inserire in ambiente agricolo.

Il fulcro dello studio condotto è stato quello di avviare il sistema: in un primo momento si è andato a studiare la componentistica hardware, composta dal manipolatore *ABB IRB 120* e dal controllore *IRC5 Compact*, mettendo in atto un lavoro di *unboxing* del sistema, installando i collegamenti fisici tra il manipolatore e il controllore stesso e cablando la connessione alla rete elettrica.

Dopo una prima fase appunto più pratica, che ha portato sostanzialmente alla cablatura del sistema unitamente all'analisi dello stesso, il cui report è contenuto nella parte I a pagina 5, il focus del progetto si è spostato sulla parte di controllo e gestione della traiettoria del manipolatore: sono stati esaminati soprattutto la parte di comunicazione PC/Controller e la cinematica del braccio robotico, implementata poi via software. Questa è proprio una peculiarità del mondo della robotica in cui si ha la necessità di focalizzarsi contemporaneamente sia sul design del robot (e quindi sulla geometria e sulle matrici caratterizzanti il movimento) sia sulla tipologì di utilizzo dello stesso robot da parte dell'utente finale.

Saranno quindi qui riportate le analisi condotte sul sistema a livello hardware e lo studio del controllo dal punto di vista software, in forma manualistica, per poter aiutare eventuali progetti futuri riguardanti lo sviluppo di sistemi basati su questa famiglia di manipolatori *ABB* per mezzo dello studio della componentistica nella parte (parte I a pagina 5) unitamente alle possibilità di controllo analizzate invece nella parte II a pagina 39.

L'obiettivo del progetto è quindi quello di andare a capire il funzionamento del manipolatore, studiandone la componente software *ABB*, prodotta in RAPID: una volta consolidata la conoscenza del dominio applicativo "software" si andrà a studiare il modo migliore per poter governare il movimento

CAPITOLO 1. INTRODUZIONE

del manipolatore stesso tramite PC, analizzando tutte le possibilità di comunicazione disponibili per poter fare dialogare il controllore *IRC5 Compact* con il braccio *ABB IRB 120*.

Parte I

Il sistema

Capitolo 2

Il manipolatore IRB 120

Il manipolatore ABB IRB 120 (*Industrial Robot Base*) rappresenta una soluzione *economica* e *affidabile* per produzioni in grande quantità su linee di produzione automatizzate, con un investimento contenuto: si tratta perciò di una nuova frontiera di manipolatori industriali che, secondo la *definizione ISO*[16], possono essere definiti come:

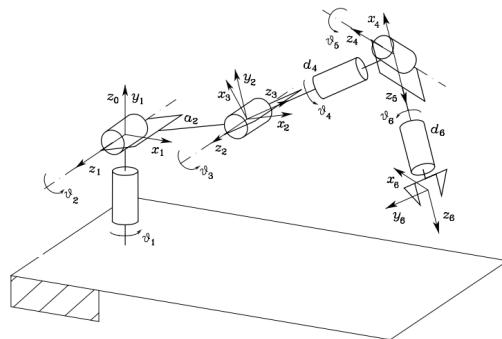
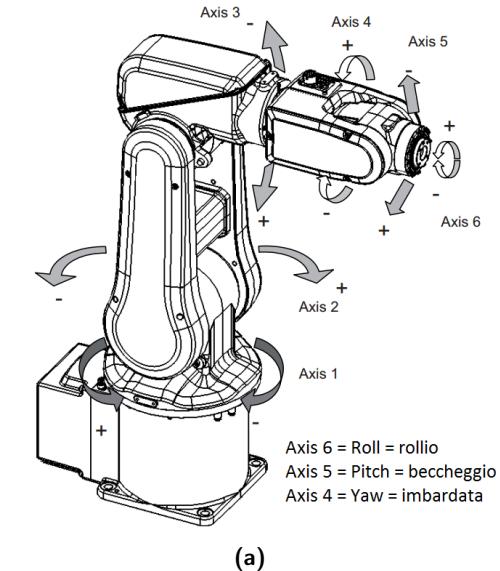
"Un manipolatore con più gradi di libertà, governato automaticamente, riprogrammabile, multiscopo, che può essere fisso sul posto o mobile per utilizzo in applicazioni di automazioni industriali"

Il manipolatore industriale oggetto dello studio in questo progetto, come esplicitato precedentemente, è l'ABB IRB 120, il quale è il più piccolo robot industriale universale commercializzato da ABB (*Asea Brown Boveri*) e che pesa solo 25kg, potendo movimentare carichi fino a 3/4 kg con uno sbraccio di 580 mm [3].

Queste caratteristiche rendono IRB 120 il manipolatore più facilmente integrabile presente sul mercato: progettato con una leggera struttura in alluminio, nella quale i potenti motori compatti assicurano un rapido avanzamento del robot (velocità di avanzamento lineare massima di 6000 mm/s) con elevata accelerazione pur mantenendo precisione, di circa *risoluzione* 0.01°, e agilità in qualsiasi applicazione e movimentazione.

2.1 Morfologia

L'IRB 120 fa parte quindi della classe dei manipolatori industriali: esso presenta 6 gradi di libertà (*numero di coordinate libere*), ovvero la movimentazione del braccio stesso avviene per mezzo di 5 *bracci* (*links*) collegati da 6 *giunti* (*joints*) di tipo rotoidale come si può vedere in figura 2.1.



(b)

Figura 2.1: Manipolatore ABB IRB120 (a) e relativo modello matematico (b)

Si intuisce facilmente come l'IRB 120 faccia parte della categoria dei manipolatori antropomorfi: le presenza di 6 giunti rotazionali va a creare una forte somiglianza con quello che è il braccio umano.

Ovviamente il braccio umano, il quale presenta 7 DOF (*Degrees Of Freedom*), ha una mobilità maggiore dovuta alla sua ridondanza: infatti il

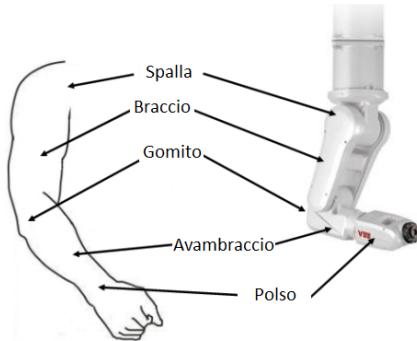


Figura 2.2: Morfologia del manipolatore industriale in similitudine con la morfologia del braccio umano

corpo umano è dotato di 2 braccia coordinate con 7 DOF ciascuno con mani con 18 DOF e flessibilità, il tutto coordinato da visione.[20]

Nonostante ciò 6 DOF risultano essere sufficienti: infatti un corpo rigido nello spazio è univocamente determinato da 6 parametri, tre dei quali individuano la posizione (x,y,z) e tre l'orientamento (ψ,θ,φ).

Tramite l'IRB 120 la posizione può essere quindi gestita tramite i primi tre giunti, mentre l'orientamento viene gestito tramite gli ultimi tre giunti della catena cinematica (capitolo 2.3 a pagina 9).

2.2 Caratteristiche tecniche principali

Nell'analisi della movimentazione del manipolatore IRB 120, si è rivelato necessario andare ad analizzare alcune delle caratteristiche fisiche principali quali:

2.2.1 Intervalli di movimento

Spostamento assi	Area di lavoro	Velocità massima
Asse 1 (<i>Rotazione</i>)	165° to -165°	250°/s
Asse 2 (<i>Braccio</i>)	110° to -110°	250°/s
Asse 3 (<i>Braccio</i>)	70° to -90°	250°/s
Asse 4 (<i>Polso</i>)	160° to -160°	320°/s
Asse 5 (<i>Brandeggio polso</i>)	120° to -120°	320°/s
Asse 6 (<i>Rotazione polso</i>)	400° to -400°	420°/s

2.2.2 Dimensioni fisiche

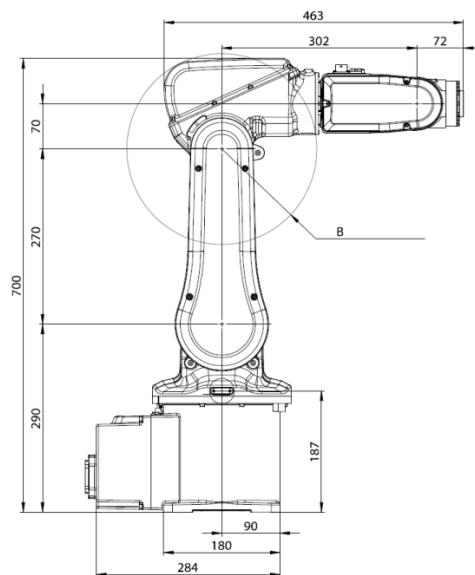


Figura 2.3: Dimensioni fisiche espresse in mm dei bracci del manipolatore IRB 120

2.2.3 Range di lavoro e raggio di curvatura

In base a quelle che sono le dimensioni fisiche del manipolatore (figura 2.3) e le massime angolazioni raggiungibili singolarmente da ognuno dei 6 joint, il TCP (*Tool Center Point* - 2.3.1 a pagina 11) può raggiungere le seguenti zone di lavoro:

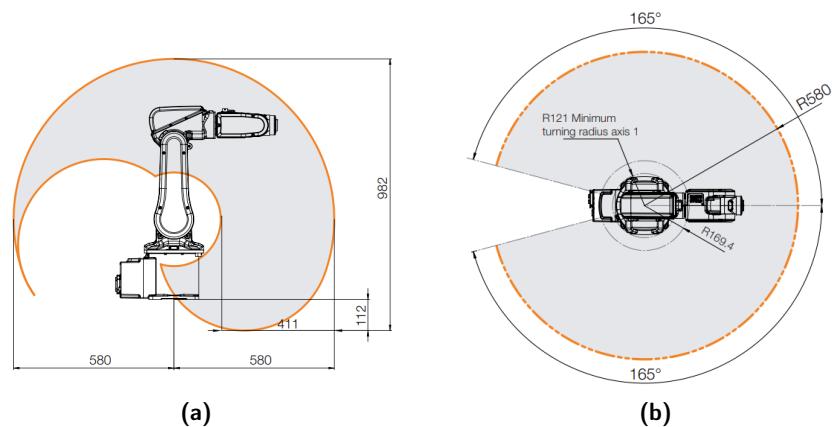


Figura 2.4: Range di lavoro (a) e raggio di curvatura (b)

2.3. CINEMATICA

2.2.4 Prestazioni

	IRB 120
Ciclo con 1kg di carico	
25 × 300 × 25mm	0.58s
25 × 300 × 25mm con riorientamento 180° asse 6	0.92s
Tempo di accelerazione 0 – 1m/s	0.07s

2.3 Cinematica

Per poter gestire e programmare la movimentazione del manipolatore ABB IRB 120 è necessario effettuare uno studio della sua cinematica (figura 2.5), ovvero di quello che è il legame tra lo stato dei 6 giunti e la posizione e l'orientamento dell'end-effector: questo deve essere gestito in base a quelli che sono i sistemi di riferimento principali e le dimensioni geometriche riguardanti il manipolatore. Analizzeremo innanzitutto i sistemi di coordinate utilizzati per definire la movimentazione del manipolatore ABB IRB 120, andando poi a sviluppare uno studio della cinematica diretta e inversa del manipolatore stesso, fornendo così gli strumenti necessari, a livello matematico, per definire con precisione la posizione del robot.

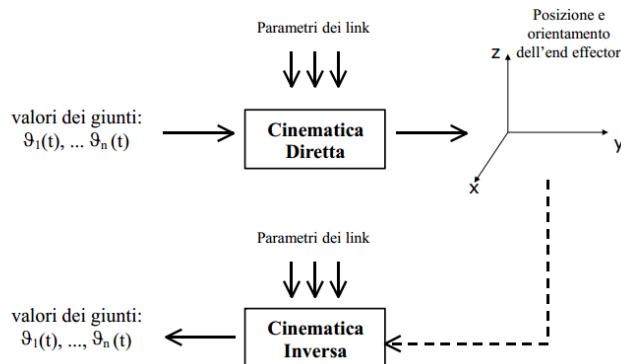


Figura 2.5: Cinematica diretta e inversa[13]

2.3.1 Sistemi di coordinate

Come già ribadito in precedenza, quello che si vuole andare a gestire è la traiettoria del manipolatore, ovvero la posizione dell'end effector: questo però è possibile solo dal momento in cui si definisce un punto di riferimento rispetto al quale si esplicita il moto, ovvero definendo un origine dalla quale si sviluppa poi un sistema di coordinate bi o tridimensionale.

Nell'ambiente di sviluppo RobotStudio (capitolo 5 a pagina 41) si lavora con più sistemi di riferimento, permettendo la creazione di aree di lavoro

complesse; questi specifici sistemi di coordinate, che permettono una gestione del moto più articolata, semplificando nello stesso tempo la programmazione fuori linea, sono:

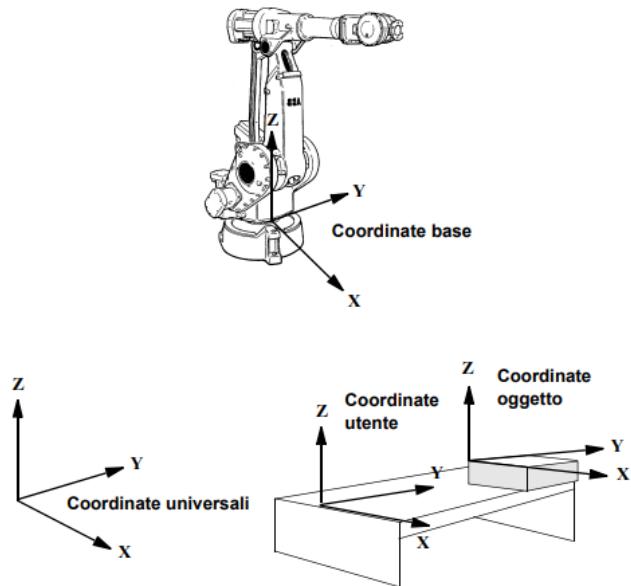


Figura 2.6: Sistemi di riferimento per la gestione della traiettoria del manipolatore

Sistema di coordinate universale: definito anche WCS (*World Coordinate System*) il quale definisce un riferimento rispetto al pavimento o alla cella di lavoro, che è il punto di partenza per gli altri sistemi di coordinate. Utilizzando questo sistema di coordinate, è possibile mettere in relazione la posizione del robot con un punto fisso in fabbrica.

Risulta quindi essere molto utile quando due robot lavorano insieme dovendo collaborare tra di loro.

Sistema di coordinate della base: esso ha come origine il punto centrale alla base del robot, ovver la base d'appoggio del robot.

Sistema di coordinate dell'utente: esso specifica la posizione di un'attrezzatura, fornendo così la possibilità di inserire sistemi di coordinate per diversi attrezzi e/o superfici di lavoro.

Sistema di coordinate dell'oggetto di lavoro: in sistemi di lavoro, anche di media complessità, un attrezzo potrebbe includere vari oggetti di lavoro che devono essere elaborati o gestiti dal robot. Questo consente di definire un sistema di coordinate per ogni oggetto (si possono quindi

avere più sistemi di coordinate oggetto) per poter regolare più facilmente il programma nel caso in cui un oggetto venga spostato oppure un oggetto nuovo, simile a quello precedente, debba essere programmato in una diversa posizione, permettendo una ridefinizione automatica della traiettoria.

Un sistema di coordinate di questo tipo viene definito sistema di coordinate oggetto, e viene ad essere posizionato rispetto al sistema di coordinate utente. Questo sistema di riferimento si adatta perfettamente alla programmazione off-line, in quanto le posizioni specificate possono essere ricavate direttamente da un disegno del work object, ovvero dell'area di lavoro (detta alternativamente anche workstation).

Sistema di coordinate del polso: esso rappresenta il sistema di riferimento del giunto numero 6; in sostanza, come per gli altri 5 assi, esso permette di definire la posizione del centro del giunto stesso. Merita una menzione particolare poiché questo sistema di riferimento definisce l'orientamento del manipolatore (figura 2.7).



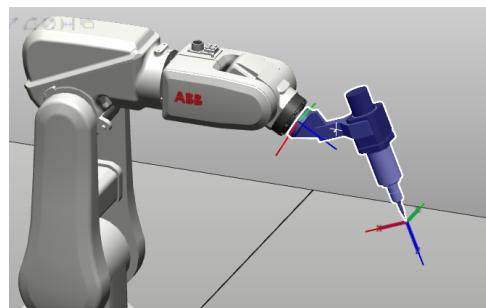
Figura 2.7: Sistema di coordinate polso

Sistema di coordinate dell'utensile di lavoro: esso permette di definire la posizione e l'orientamento dell'utensile agganciato al polso. Spesso chiamato TCP (*Tool Center Point*) la sua importanza è quindi estrema, poiché permette di gestire il punto con il quale il manipolatore andrà ad agire sull'oggetto, definendone la gestione della lavorazione richiesta (figura 2.8 nella pagina seguente)

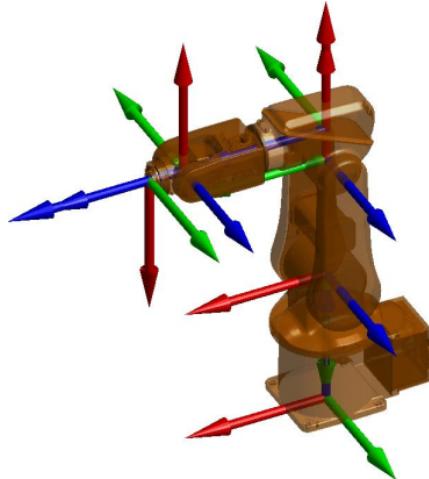
Da queste brevi descrizioni si capisce come la gestione della traiettoria del manipolatore debba occuparsi della posizione e dell'orientamento del *TCP* rispetto al sistema di riferimento oggetto: entrambi risultano essere definiti, a loro volta, rispetto al sistema di coordinate universale.

2.3.2 Descrizione di posizione e orientamento di un sistema di riferimento

Come si può vedere in figura 2.9 nella pagina successiva, l'ABB IRB 120 può essere modellizzato come un insieme di sistemi di riferimento: trattandosi infatti di un manipolatore robotico, esso è costituito da una sequenza di organi

**Figura 2.8:** Tool Center Point

meccanici (*links*), collegati tra loro da *joints*, individuando così una catena cinematica aperta, per l'assenza di anelli chiusi. Ad ogni link è associato un

**Figura 2.9:** Catena cinematica IRB 120

sistema di coordinate che ne caratterizza la posizione: quindi il fulcro del problema si riduce alla definizione della posizione e dell'orientamento di un generico sistema di riferimento, sfruttando poi le caratteristiche della catena cinematica (figura 2.9) per andare a posizionare il robot.

La posizione di un sistema di riferimento è banalmente individuata dalle coordinate x, y, z della sua origine, sempre riferite ad un sistema fissato. Ciò che comporta uno studio più approfondito è l'orientamento di una terna rispetto ad un'altra: questo è possibile descrivendo i versori della prima rispetto a quelli della seconda (figura 2.10 nella pagina successiva).

Rotazioni elementari

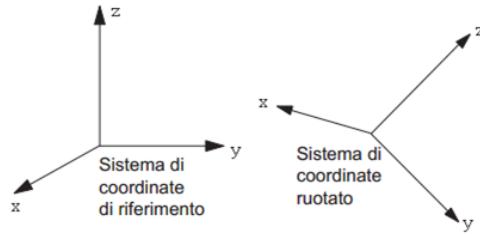


Figura 2.10: Rotazione di una terna rispetto ad un'altra

Riferendosi alla figura 2.10, andando a far coincidere le origini delle due terne, ovvero operando una traslazione della seconda terna rispetto alle prima, possiamo vedere i versori dei tre assi come tre vettori separati, i quali possono quindi essere espressi rispettivamente come:

- \hat{x} = versore dell'asse $x = (x_1, x_2, x_3) \Rightarrow$ Il versore direzionale dell'asse x presenta una componente x_1 lungo l'asse x del sistema di riferimento considerabile *fisso*, x_2 lungo y e x_3 lungo z ;
- \hat{y} = versore dell'asse $y = (y_1, y_2, y_3) \Rightarrow$ Il versore direzionale dell'asse y presenta una componente y_1 lungo l'asse x del sistema di riferimento considerabile *fisso*, y_2 lungo y e y_3 lungo z ;
- \hat{z} = versore dell'asse $z = (z_1, z_2, z_3) \Rightarrow$ Il versore direzionale dell'asse z presenta una componente z_1 lungo l'asse x del sistema di riferimento considerabile *fisso*, z_2 lungo y e z_3 lungo z ;

La matrice di rotazione prende quindi la seguente forma: il pedice posto alla sinistra della matrice indica il sistema di riferimento descritto, mentre l'apice indica il riferimento usato per la descrizione, ovvero *la matrice R (matrice 2.1) indica la matrice di rotazione che descrive la terna $\{B\}$ rispetto alla terna $\{A\}$* .

$${}^A_B R = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} = \begin{bmatrix} \hat{x}_B \cdot \hat{x}_A & \hat{y}_B \cdot \hat{x}_A & \hat{z}_B \cdot \hat{x}_A \\ \hat{x}_B \cdot \hat{y}_A & \hat{y}_B \cdot \hat{y}_A & \hat{z}_B \cdot \hat{y}_A \\ \hat{x}_B \cdot \hat{z}_A & \hat{y}_B \cdot \hat{z}_A & \hat{z}_B \cdot \hat{z}_A \end{bmatrix} \quad (2.1)$$

La matrice 2.1 è detta anche matrice dei coseni direttori dato che, per definizione di prodotto vettoriale ($\hat{x}_B \cdot \hat{x}_A = \| \hat{x}_B \| \cdot \| \hat{x}_A \| \cdot \cos \alpha = \cos \alpha$), i componenti della matrice di rotazione saranno solamente dei coseni degli angoli compresi tra i versori.

Per poter trovare la semplice matrice di rotazione che descrive la rotazione in figura 2.11 nella pagina successiva, le due possibilità che si hanno sono quelle di

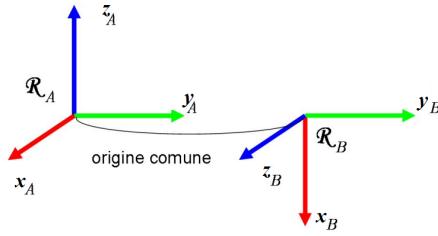


Figura 2.11: Semplice esempio di rotazione di sistemi di riferimento

- individuare i versori del sistema di riferimento B rispetto al sistema A;
- sviluppare i prodotti vettori (ricordando che esso si annulla per vettori ortogonali tra di loro)

Detto ciò possiamo trovare la matrice come:

$$\mathbf{x}_B = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, \mathbf{y}_B = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \mathbf{z}_B = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad (2.2)$$

La matrice di rotazione della figura 2.11 sarà quindi:

$${}^A_B \mathbf{R} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \quad (2.3)$$

Quelle considerate finora, tramite semplici esempi, rappresentano appunto rotazioni elementari (figura 2.12 nella pagina successiva), ovvero che avvengono lungo un determinato asse. Andando ad applicare quanto di semplice fatto nell'esempio in figura 2.11, siamo in grado di andare a trovare le matrici di rotazione per i casi di rotazioni note:

- **Rotazione intorno asse x di angolo γ :**

$$\mathbf{R}_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \gamma & -\sin \gamma \\ 0 & \sin \gamma & \cos \gamma \end{bmatrix} \quad (2.4)$$

- **Rotazione intorno asse y di angolo β :**

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix} \quad (2.5)$$

- **Rotazione intorno asse z di angolo α :**

$$\mathbf{R}_z(\alpha) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.6)$$

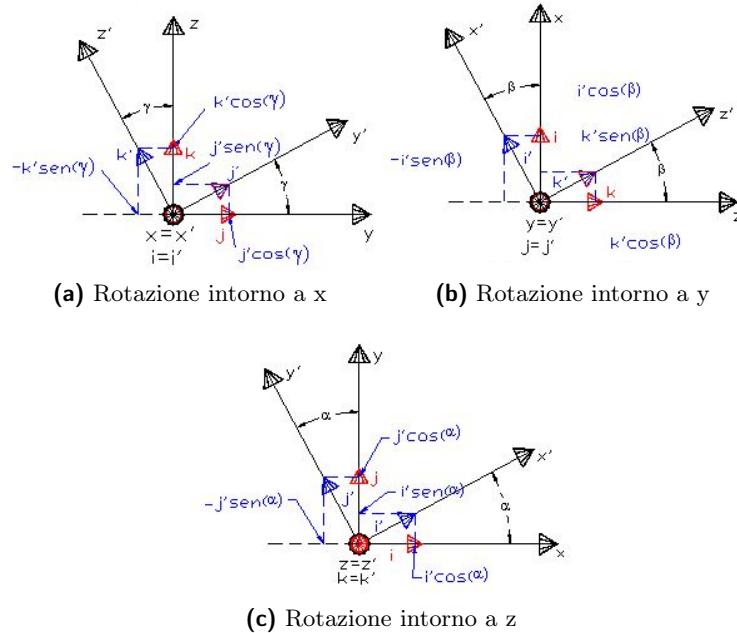


Figura 2.12: Rotazioni elementari

Composizione delle matrici di rotazione: rotazioni complesse

Concetto importante, che poi riapparirà, seppur se in forma più complessa, nella convenzione di Denavit-Hartenberg (sottosezione 2.3.3 a pagina 17), è quello di *composizione delle matrici di rotazione*.

Finora sono state analizzate solamente rotazioni di tipo elementare, ovvero rotazioni che avvengono solamente intorno ad un unico asse: ogni altra rotazione, e la corrispondente matrice, può essere ottenuta combinando opportunamente le rotazioni elementari.

Si tratta quindi di scomporre la rotazione complessiva in tante rotazioni elementari: la matrice di rotazione complessiva è sempre ottenibile dal prodotto delle matrici di rotazioni associate alle singole rotazioni: queste rotazioni "sequenziali" possono essere effettuate in

- Terna fissa: la rotazione avviene sempre intorno agli assi del sistema di riferimento "iniziale", che viene mantenuto fisso e costante durante tutte le rotazioni sequenziali
- Terna mobile: in questo caso la rotazione viene eseguita rispetto al sistema di coordinate ottenuto dalla precedente rotazione

Il concetto importante è però che, indipendentemente dal tipo di rotazione composta che si va a considerare (se rispetto ad una terna fissa oppure mobile), per poter descrivere rotazioni complesse è sufficiente effettuare

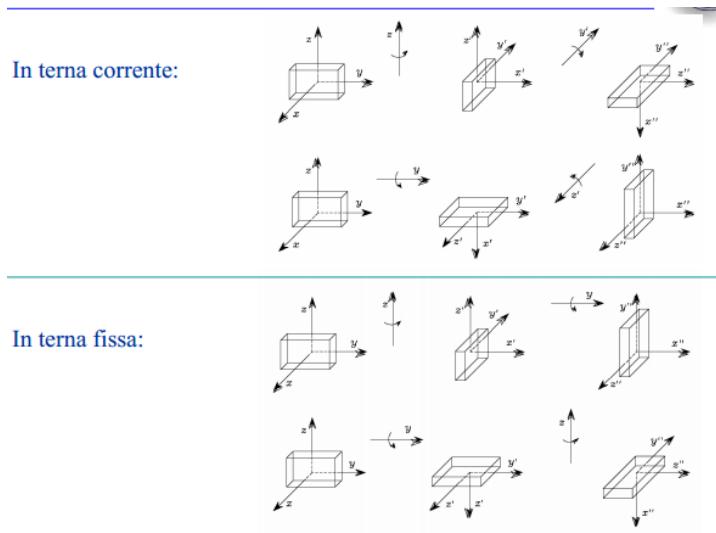


Figura 2.13: Composizione di rotazioni in terna fissa e mobile [17]

una moltiplicazione matriciale tra le matrici di rotazione di ogni singolo orientamento elementare.

Per esempio, supponendo di avere tre terne A, B e C diversamente orientate, si può individuare la matrice di rotazione che esprime l'orientamento della terna C rispetto ad A (${}^A_C R$) con una moltiplicazione matriciale di questo tipo

$${}^A_C R = {}^A_B R \cdot {}^B_C R \quad (2.7)$$

Va da sé che unendo le informazioni in merito a posizione e rotazione di un sistema di riferimento, si è in grado di individuare in maniera univoca una terna mobile rispetto ad una fissa.

Tipicamente tale descrizione reciproca tra terne è espressa mediante una singola matrice che comprende entrambe le informazioni di posizione e orientamento e prende il nome di *matrice di trasformazione omogenea* ed è così definita

$$\left[\begin{array}{c|c} {}^A_B R & pos_{B,A} \\ \hline \mathbf{0} & \mathbf{1} \end{array} \right]$$

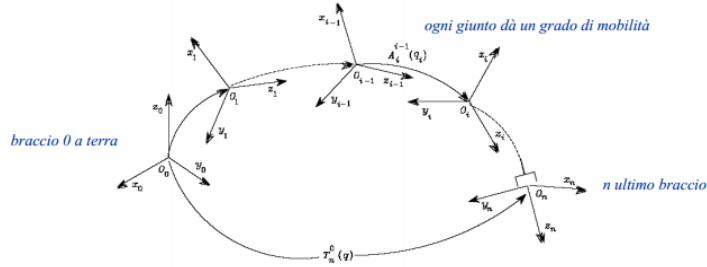


Figura 2.14: Esempio di catena cinematica aperta formata da n bracci [17]

2.3.3 Cinematica diretta IRB 120

Il problema cinematico diretto si impone il problema di trovare il posizionamento dell'end-effector dato il valore dei giunti del manipolatore.

Nello specifico l'IRB 120 fa parte della categoria dei bracci seriali, ovvero di quei manipolatori costituiti da una catena cinematica aperta, ovvero una catena in cui vi è una sola sequenza di *link* a connettere i due estremi. La risoluzione della cinematica diretta dell'IRB 120 è concettualmente molto semplice: si tratta di applicare una composizione di rotazioni in terna fissa a sistemi di riferimento solidali ad ognuno dei 6 bracci, partendo dalla terna base, ed ottenendo così orientamento e posizione dall'ultima terna della catena (end-effector) espressa rispetto alla base del manipolatore.

La composizione di rotazioni analizzata nella sottosezione 2.3.2 a pagina 15 era riferita alle sole matrici di rotazioni, le quali individuano esclusivamente informazioni in merito all'orientamento. Questo concetto di composizione, basata su moltiplicazioni matriciali, è però applicabile anche alle matrici di trasformazione omogenea, la cui forma è leggermente più complessa (data la presenza delle informazioni in merito alla posizioni come si vede nel paragrafo 2.3.2 nella pagina precedente).

Il procedimento è quindi concettualmente molto semplice: ma esiste una qualche convenzione per fissare le terne solidali ad ogni giunto (sezione 2.1 a pagina 6) in modo sistematico?

Metodo di Denavit-Hartenberg

Un possibile modo per poter trovare le 6 terne solidali ad ognuno degli altrettanti 6 giunti è basato sul metodo di Denavit-Hartenberg: in questa sede non entreremo nei dettagli di quello che è l'algoritmo iterativo per individuare e definire la terna solidale ad ogni giunto rispetto al precedente, ma ci limiteremo a rappresentarne gli aspetti più importanti e le conseguenze principali.

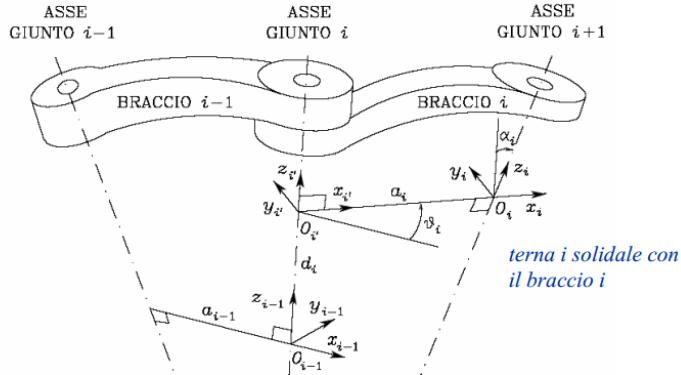


Figura 2.15: Posizionamento sistemi di riferimento solidale per due bracci qualsiasi

La scelta dei sistemi di riferimento solidali per ogni giunto è basata quindi su due convenzioni (dette di *Denavit-Hartenberg*):

- *Numerazione dei bracci e dei giunti*: ogni braccio (*link*) viene numerato da 0 a n a partire dalla base e arrivando all'organo terminale; ognuno di essi sarà individuato dai simboli L_0, L_1, \dots, L_n . Un manipolatore seriale (ovvero rappresentabile tramite una *catena cinematica aperta*) con n bracci, avrà $n - 1$ giunti, designati J_1, J_2, \dots, J_n . Il giunto J_i collega i bracci L_{i-1} e L_i .
- *Assegnazione degli assi z dei sistemi di riferimento*: al membro i -esimo ($0 \leq i \leq n$) si associa un sistema di riferimento solidale $\{O^i, x^i, y^i, z^i\}$ il cui asse z^i coincide con l'asse del giunto J_{i+1} , cioè del giunto a valle del membro nella catena cinematica. Come si osserva nella figura 2.15, gli assi z dei bracci $i - 1$ e i hanno la stessa direzione degli assi z rispettivamente dei giunti i e $i + 1$.

Una volta assegnati gli assi z_i , possiamo andare a individuare:

- O_i : l'origine del sistema di riferimento i -esimo è all'intersezione dell'asse z_i con la normale comune (definibile come il segmento ortogonale agli assi del giunto i -esimo e di quello $i + 1$ -esimo, come si può vedere nella figura 2.15); si indica con O'_i l'intersezione della normale comune con z_{i-1}
- x_i : è diretto lungo la normale comune agli assi z_i e z_{i-1} , con verso positivo dal giunto i al giunto $i + 1$
- y_i : completa una terna destrorsa.

2.3. CINEMATICA

Da questi passi iterabili per individuare i sistemi solidali ad ogni braccio, possiamo andare ad individuare quelli che sono i *parametri di Denavit-Hartenberg*, che ci permettono di definire una terna rispetto alla precedente:

- a_i : distanza di O_i da O'_i , ovvero tra l'asse di giunto i -esimo e l'asse $(i+1)$ -esimo; essendo una distanza, essa sarà perpendicolare ad entrambi gli assi di giunto
- d_i : coordinata su z_{i-1} di O_i
- α_i : angolo intorno all'asse x_i tra l'asse z_{i-1} e l'asse z_i , valutato positivo in senso antiorario; esso individua l'angolo esistente tra l'asse di giunto i -esimo e l'asse $(i+1)$ -esimo
- ϑ_i : angolo intorno all'asse z_{i-1} tra l'asse x_{i-1} e l'asse x_i valutato positivo in senso antiorario (detta anche *variabile di giunto* nel caso di giunti rotoidali come nei manipolatori antropomorfi)

Quelli sopraelencati rappresentano i cosiddetti parametri cinematici, fondamentali per la risoluzione della cinematica (sia diretta che inversa) di un generico manipolatore.

Per il manipolatore IRB 120 questi parametri sono riportati all'interno della tabella (si noti il legame dei parametri cinematici di braccio con la figura 2.3 a pagina 8):

Tabella 2.1: Parametri di Denavit-Hartenberg del manipolatore IRB 120

Braccio i -esimo	Parametri cinematici di braccio		Parametri cinematici di giunto	
	Lunghezza di braccio (a_i)	Torsione di braccio (α_i)	Scostamento di giunto (d_i)	Angolo di giunto (ϑ_i)
1	0	-90°	290	ϑ_1
2	270	0°	0	$\vartheta_2 - 90^\circ$
3	70	-90°	0	ϑ_3
4	0	90°	302	ϑ_4
5	0	-90°	0	ϑ_5
6	0	0°	72	$\vartheta_6 + 180^\circ$

Introduciamo il loro utilizzo per andare a rappresentare la matrice di trasformazione generica di un braccio: la matrice 2.8 nella pagina seguente permette di descrivere la trasformazione che il braccio (*link*) introduce sui due giunti che si trovano alla sua estremità.

Per andare quindi a ricavare la matrice di trasformazione omogenea del braccio riportato in figura su cui sono fissate le terne $\{i-1\}$ ed $\{i\}$. Per ricavare invece la matrice di trasformazione omogenea ${}^{i-1}_iT(Q_i)$, si vanno ad aggiungere tre terne ausiliarie $\{P\}, \{Q\}$ ed $\{R\}$, andando a scrivere le matrici di rotazioni che descrivono ciascuna di essa rispetto a quella precedente (passaggio che andremo ad omettere). Andando poi a sfruttare la teoria

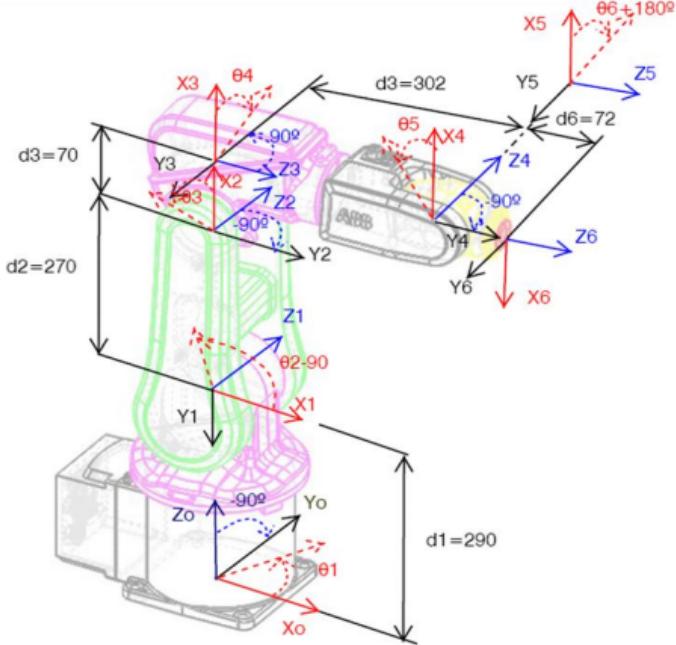


Figura 2.16: Rappresentazione dei parametri di Denavit-Hartenberg

della composizione di matrici di rotazione (valida anche per le matrici di trasformazione omogenee viste nella sezione 2.3.2 a pagina 16 a pagina 16) possiamo scrivere la matrice che descrive la terna $\{i\}$ rispetto alla terna $\{i-1\}$ attraverso la composizione delle quattro matrici di trasformazione relative agli altrettanti sistemi di riferimento: si tratta di andare a moltiplicare in sequenza, una matrice omogenea che descrive una rotazione, due matrici omogenee che rappresentano altrettante traslazioni ed infine un'altra matrice rappresentata rotazione ottenendo il seguente risultato

$${}_{i-1}^i T = {}_R^Q T {}_i^P T {}_Q^P T {}_{i-1}^{i-1} T = \left[\begin{array}{ccc|c} \cos \vartheta_i & -\sin \vartheta_i \cos \alpha_i & \sin \vartheta_i \sin \alpha_i & a_i \cos \vartheta_i \\ \sin \vartheta_i & \cos \vartheta_i \cos \alpha_i & -\sin \alpha_i & a_i \sin \vartheta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (2.8)$$

Quanto detto finora rappresenta la via risolutiva per risolvere la cinematica diretta del manipolatore in esame; andando ad analizzare ogni singola matrice di trasformazione $i-esima$ possiamo evidenziare ed isolare i seguenti passaggi (con la scritta s_i si indica il valore $\cos \vartheta_i$ e con s_i si indica invece il $\sin \vartheta_i$ visto che, per come è costruita la tabella 2.1 nella pagina precedente, le funzioni seno e coseno degli angoli di torsione avranno sempre o valore nullo oppure ± 1).

$${}_1^0 T = A_1 = \begin{bmatrix} c_1 & 0 & -s_1 & 0 \\ s_1 & 0 & c_1 & 0 \\ 0 & -1 & 0 & 290 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^1_2T = A_2 \begin{bmatrix} c_2 & -s_2 & 0 & 270c_2 \\ s_2 & c_2 & 1 & 270s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^2_3T = A_3 \begin{bmatrix} c_3 & 0 & -s_3 & 70c_3 \\ s_3 & 0 & c_3 & 70s_3 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^3_4T = A_4 \begin{bmatrix} c_4 & 0 & s_4 & 0 \\ s_4 & 0 & -c_4 & 0 \\ 0 & 1 & 0 & 302 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^4_5T = A_5 \begin{bmatrix} c_5 & 0 & -s_5 & 0 \\ s_5 & 0 & c_5 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$${}^5_6T = A_6 \begin{bmatrix} c_6 & -s_6 & 0 & 0 \\ s_6 & c_6 & 0 & 0 \\ 0 & 0 & 1 & 72 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

L'ultimo step per poter giungere così alla risoluzione del problema cinematico diretto per il manipolatore IRB 120 è quello di andare a comporre le informazioni in merito alle roto-traslazioni di ogni sistema di riferimento, partendo dalla base e giungendo fino all'end-effector, ottenendo così posizione e orientamento dello stesso (per non appesantire ulteriormente il discorso il risultato viene espresso in forma compatta):

Figura 2.17: Soluzione cinematica diretta manipolatore ABB IRB 120

$${}^0_6T = {}^0_1T {}^1_2T {}^2_3T {}^3_4T {}^4_5T {}^5_6T = A_1 A_2 A_3 A_4 A_5 A_6 \begin{bmatrix} R_{11} & R_{12} & R_{13} & P_x \\ R_{21} & R_{22} & R_{23} & P_y \\ R_{31} & R_{32} & R_{33} & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3.4 Cinematica inversa IRB 120

Come visto nella matrice in figura 2.17 nella pagina precedente, il problema cinematico diretto è sempre risolvibile attraverso la valutazione della matrice di trasformazione omogenea 0_6T .

Il problema inverso invece, risulta essere più complicato: infatti, in base alla posizione spaziale fornita tramite coordinate (x, y, z) , ci possono essere casi in cui vi è un'unica soluzione, altri in cui ve ne possono essere multiple e altri ancora in cui non vi è alcuna possibile soluzione (quando per esempio la coordinata che si chiede di raggiungere si trova fuori dal range di azione del manipolatore stesso).

Non è quindi garantita né l'esistenza né l'unicità della soluzione della cinematica inversa e, nel caso ci siano più soluzioni possibili, si cerca di scegliere la migliore, andando a minimizzare i movimenti dei giunti.

Uno dei più intuitivi modi per poter risolvere il problema cinematico inverso, è quello di andare ad egualiare la matrice di trasformazione dell'end-effector, che ne indica posizione ed orientamento rispetto alla base, con la sua espressione simbolica, ovvero la matrice che assegna un legame matematico a tutta la catena cinematica del manipolatore (figura 2.17 nella pagina precedente).

$${}^0_6T = \begin{bmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 2.18: Input problema cinematico inverso

Si ottengono in questo modo 12 equazioni in 6 incognite: le 9 equazioni della componente di rotazione non sono indipendenti, e si ottengono quindi 6 equazioni in 6 incognite (le variabili di giunto). Sicuramente questo rappresenta un metodo stabile per giungere alla soluzione del problema cinematico inverso: dovendo andare a lavorare con un manipolatore avente 6 gdl, l'inversione cinematica non è del tutto agevole, soprattutto per la complessità della parte matematica e di calcolo matriciale. Conviene quindi andare a studiare il sistema in due parti separate: sfruttando il *disaccoppiamento cinematico*, che si basa su un approccio geometrico-matematico, possiamo infatti studiare separatamente i primi tre giunti, che permettono di posizionare il polso in un punto desiderato, dagli ultimi tre, i quali servono per far assumere allo strumento di lavoro un qualsiasi orientamento.

2.3.5 Posizione dell'end-effector: definizione variabili di giunto $\vartheta_1 - \vartheta_2 - \vartheta_3$

Come detto, il primo approccio applicabile è quello di andare a lavorare sulle matrici di trasformazione (A_i), uguagliando

$${}^0T = {}^0T {}^1T {}^2T {}^3T {}^4T {}^5T {}^6T = A_1 A_2 A_3 A_4 A_5 A_6 = \begin{bmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Per trovare quindi le 6 variabili di giunto andremo quindi a pre-moltiplicare la matrice di trasformazione omogenea 0T per A_n^{-1} , iniziando con A_1^{-1} :

$$A_1^{-1} \times \begin{bmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = A_2 A_3 A_4 A_5 A_6$$

Ci si rende conto da subito come l'onere computazionale a livello matematico sia molto importante; il risultato sopra è ottenibile anche tramite le coordinate spaziali dell'end-effector, che permettono di andare ad imporre una risoluzione tramite metodo geometrico. Trovando le coordinate del centro del polso nel sistema di riferimento fisso della base e sfruttando la matrice di trasformazione omogenea che viene fornita come input al problema cinematico inverso, la quale fornisce tutti dati necessari per eseguire questo primo passo (sistema 2.19).

Figura 2.19: Coordinate del centro del polso del manipolatore

$$\begin{cases} x_P = P_x - 72a_x \\ y_P = P_y - 72a_y \\ z_P = P_z - 72a_z \end{cases}$$

Andando così a seguire un semplice ragionamento sul triangolo rettangolo individuato dalle coordinate x_p e y_p del polso, nel quale l'angolo compreso tra l'ipotenusa e il cateto individuato dalla direzione x è proprio l'angolo ϑ_1 (vedi figura 2.16 a pagina 20) che cerchiamo, ovvero

$$\vartheta_{1,1} = \tan^{-1}\left(\frac{y_P}{x_P}\right)$$

$$\vartheta_{1,2} = \vartheta_1 + 180^\circ$$

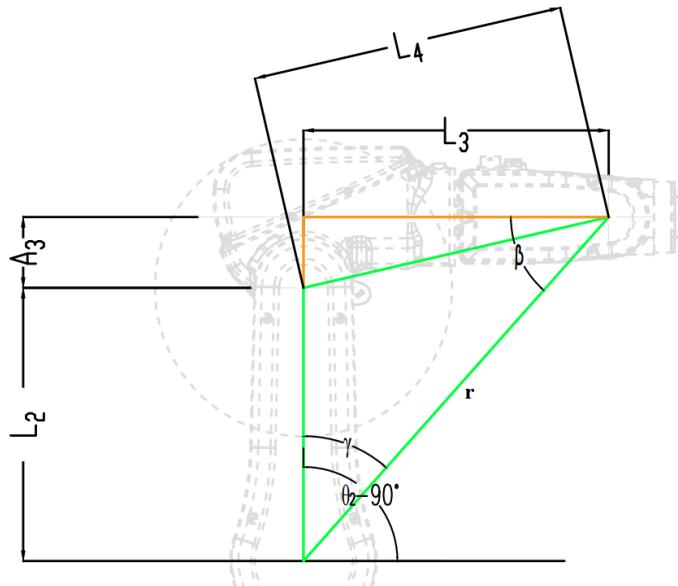


Figura 2.20: Triangoli di supporto per il calcolo del angolo ϑ_2

La stesso risultato riportato nelle equazioni soprastanti è ottenibile servendosi della funzione *atan2* (sottosezione 2.3.6 a pagina 30), ottenendo come risultato

$$\vartheta_{1,1} = \text{atan}_2(y_P, x_P)$$

$$\vartheta_{1,2} = \text{atan}_2(-y_P, -x_P)$$

Secondo un ragionamento molto simile, possiamo andare a calcolare ϑ_2 : si vede come, effettuando un ragionamento sui triangoli evidenziati in figura 2.20, possiamo evidenziare come l'angolo β sia pari all'angolo $(\vartheta_2 - 90^\circ) - \gamma$.

La lunghezza di r dipende però ovviamente anche da altri angoli (quali ϑ_3 per esempio): per procedere con il suo calcolo andiamo a ricavare le coordinate del polso (figura 2.19 nella pagina precedente) espresse però nel sistema di riferimento 1 la cui origine si trova nel vertice dove è presente l'angolo $\vartheta_2 - 90^\circ$.

Questo lo possiamo ottenere andando a trovare la matrice di Denavit-Hartenberg relativa al sistema di riferimento 1 e poi, con una semplice moltiplicazione matriciale, siamo in grado di ottenere le coordinate del polso rispetto al punto prima specificato.

2.3. CINEMATICA

Conoscendo queste coordinate risulta facile calcolare sia il valore dell'angolo β sia dell'angolo γ (sfruttando il teorema di Carnot che afferma che $a^2 = b^2 + c^2 - 2bc \cos \alpha$, con α angolo compreso tra b e c).

Lo script *MATLAB* che ne segue è il seguente:

```
%Parametri di Denavit-Hartenberg
L2=270; %a_2
L3=302;%d_4
A3=70;%a_3

%Calcolo della lunghezza L_4
L4 = sqrt(A3^2 + L3^2);

%Matrice di trasformazione omogenea del primo giunto: indica
%orientamento posizione del primo sistema di riferimento
%rispetto alla base
T01=Calcolo_DH_Matrix(robot, q, 1);

%Esprimo le coordinate del centro del polso rispetto alla
%base, utilizzando il concetto espresso in figura 3.18
p1 = inv(T01)[Pm; 1];

%Calcolo la distanza r che dipende quindi dalla posizione del
%centro del polso e non risulta essere costante
r = sqrt(p1(1)^2 + p1(2)^2);

%Calcolo gli angoli espressi in figura utilizzando i concetti
%matematici precedentemente esposti
beta = atan2(-p1(2), p1(1));
gamma = real(acos((L2^2+r^2-L4^2)/(2*r*L2)));

%Ho due possibili soluzioni: gomito su e gomito giu
q2(1) = pi/2 - beta - gamma;
q2(2) = pi/2 - beta + gamma;
```

Codice 2.1: Calcolo del parametro di giunto ϑ_2

Per il calcolo di ϑ_3 i parametri geometrici, ovvero le lunghezze dei lati dei triangoli presi in considerazione, sono sempre quelli presentati nel codice 2.1, cambia solamente il calcolo dell'angolo, come è ovvio che sia, il quale può essere così sviluppato:

```
%Calcolo degli angoli caratterizzanti
phi=acos((A3^2+L4^2-L3^2)/(2*A3*L4));
eta = real(acos((L2^2 + L4^2 - r^2)/(2*L2*L4)));

%Due possibili soluzioni: gomito su e gomito giu
q3(1) = pi - phi - eta;
q3(2) = pi - phi + eta;
```

Codice 2.2: Calcolo del parametro di giunto ϑ_3

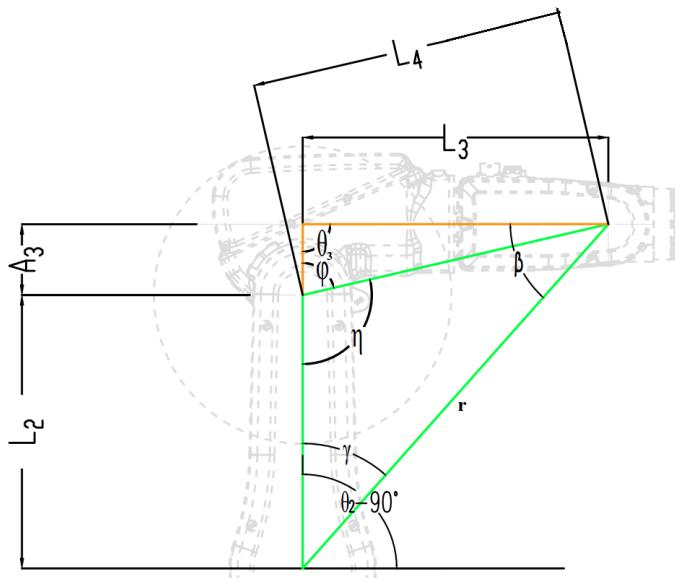


Figura 2.21: Triangoli di supporto per il calcolo del angolo ϑ_3

2.3.6 Orientamento dell'end-effector: definizione variabili di giunto $\vartheta_4 - \vartheta_5 - \vartheta_6$

Una volta calcolati i valori di $\vartheta_1 - \vartheta_2 - \vartheta_3$, la posizione del centro del polso del manipolatore risulta essere fissata: possiamo quindi calcolare la matrice di trasformazione omogenea che descrive la catena cinematica dei primi tre giunti.

```
%Calcolo della posizione e dell'orientamento del terzo
sistema di riferimento utilizzando gli angoli q (theta_1,
theta_2,theta_3) prima calcolati
T01=DH_Compute(robot, q, 1);
T12=DH_Compute(robot, q, 2);
T23=DH_Compute(robot, q, 3);
T03=T01*T12*T23;

%Vado ad estrarre poi le informazioni in merito alla
rotazione del sistema di riferimento, come rappresentato
in figura 3.10.
x3=T03(1:3,1);
y3=T03(1:3,2);
z3=T03(1:3,3);
```

Codice 2.3: Matrice di trasformazione omogenea dei primi tre giunti

Questa fornisce la posizione del polso: quello che andiamo ora a fissare, definendo i tre angoli $\vartheta_4 - \vartheta_5 - \vartheta_6$ è l'orientamento del polso stesso, che può essere definito in termini di angoli di Eulero, ovvero di rollio-beccheggio-

Figura 2.22: Step iniziale per il calcolo algebrico dell'orientamento del polso

$$A_4 A_5 A_6 = (A_1 A_2 A_3)^{-1} \cdot \begin{bmatrix} n_x & o_x & a_x & P_x \\ n_y & o_y & a_y & P_y \\ n_z & o_z & a_z & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

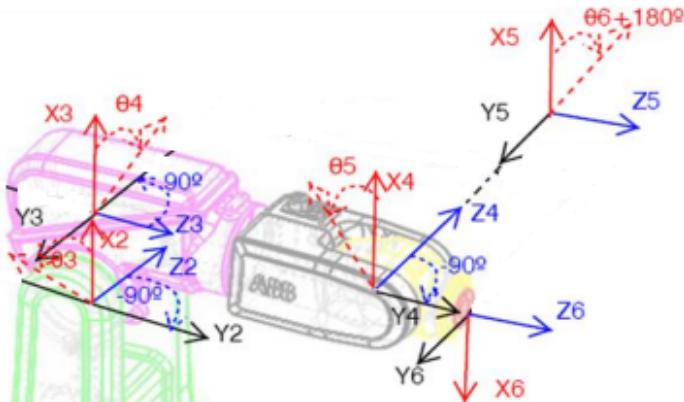


Figura 2.23: Sistemi di riferimento dei giunti utilizzati per trovare l'orientamento del polso

imbardata (*Raw-Pitch-Yaw*). Come fatto precedentemente per individuare la posizione, anche qui abbiamo due vie di risoluzione: potremmo andare a risolvere il problema inverso tramite una via algebrica, la quale si basa sugli angoli individuati precedentemente ($\vartheta_1 - \vartheta_2 - \vartheta_3$) partendo dalla formula in figura 2.22.

Un altro possibile approccio, decisamente meno oneroso a livello computazionale, è rappresentato da quello geometrico: andando a lavorare sull'orientamento dei vari sistemi di riferimento rappresentati in figura 2.23 siamo in grado di recuperare i valori di $\vartheta_4 - \vartheta_5 - \vartheta_6$. L'approccio che possiamo seguire per poter trovare l'angolo ϑ_4 è quello di andare a studiare l'orientamento dell'asse z_3 e z_6 , il quale è fornito in ingresso al problema cinematico inverso: valutando la posizione reciproca dei due assi sopracitati, possiamo andare a definire due casistiche di valori per ϑ_4 . In un caso esso è nullo, nell'altro può essere trovato andando sostanzialmente a costruire un triangolo rettangolo intorno a ϑ_4 , i cui lati possono essere considerati come gli assi x_3 , y_3 e z_3 .

Come si può notare nella figura 2.23, ϑ_4 è proprio l'angolo che si crea con x_3 e la trasposizione di z_4 sul giunto numero 3: operando un prodotto scalare, siamo in grado di trovare la proiezione di z_4 dapprima lungo x_3 e successivamente lungo y_3 , grazie al significato geometrico proprio del prodotto

scalare tra vettori.

Questi passaggi permettono di ottenere nient'altro che il valore del \cos e del \sin dell'angolo ϑ_4 , legati tra loro dalla funzione atan , come si può vedere nel codice 2.4.

```
%Vado a calcolare la matrice di trasformazione omogenea del
terzo giunto sfruttando le proprietà delle matrici di
trasformazione esposte nella sottosezione~\vref{
subsubsection:RotComplex}
T01=DH_Compute(robot, q, 1);
T12=DH_Compute(robot, q, 2);
T23=DH_Compute(robot, q, 3);
T03=T01*T12*T23;

%Orientamento dei tre assi del sistema di riferimento del
terzo giunto
x3=T03(1:3,1);
y3=T03(1:3,2);
z3=T03(1:3,3);

%Asse $z_6$ che corrisponde al vettore \emph{a} nella matrice
di trasformazione in input al problema cinematico
inverso
a=T(1:3,3);

%corrisponde al prodotto vettoriale
% find z4 normal to the plane formed by z3 and a

%Prodotto vettoriale: il risultato è un vettore ortogonale
al piano individuato da $z_3$ e $z_6$.
z4=cross(z3, a);

%Il modulo rappresenta l'area compresa tra i due assi: se è
circa nulla, vuol dire che i due assi sono paralleli (per
le proprietà del prodotto vettoriale)
if norm(z4) <= 0.000001
    q(4)=0;
else
    %Vado sostanzialmente a comportmi un triangolo rettangolo,
    in base a quello che è l'orientamento dell'ultimo
    sistema di riferimento: esso può essere più o meno "inclinato" a destra o sinistra oppure in alto o in
    basso
    cq4=wrist*dot(z4, -y3);
    sq4=wrist*dot(z4, x3);
    q(4)=atan2( sq4, cq4);
end
```

Codice 2.4: Calcolo di ϑ_4

Anche per il calcolo di ϑ_5 possiamo andare ad operare lo stesso ragionamento fatto per trovare ϑ_4 : in questo caso andremo ad effettuare la proiezione

2.3. CINEMATICA

dell'asse z_5 , il quale fa riferimento al giunto 5 in esame, sugli assi del giunto precedente, ovvero su x_4 e y_4 .

```
%Man mano vado a ricostruirmi orientamento e posizione della
  catena cinematica del manipolatore
T34=DH_Compute(robot, q, 4);
T04=T03*T34;
x4=T04(1:3, 1);
y4=T04(1:3, 2);

%Come si vede dalla figura~\vref{fig:CalcoloOrientamentoWrist}
% l'asse $z_5$ ha la stessa direzione di $z_6$
z5=T(1:3, 3);

%Vado, come per $\theta_4$ a lavorare sul triangolo
%rettangolo formato dagli assi
cq5=dot(z5, y4);
sq5=dot(z5, -x4);
q(5)=atan2(sq5, cq5);
```

Codice 2.5: Calcolo di ϑ_5

Ripetiamo infine lo stesso concetto geometrico presentato per il calcolo di ϑ_4 e ϑ_5 , anche per andare a trovare il valore di ϑ_6 .

```
x6=T(1:3, 1);

T45=DH_Compute(robot, q, 5);
T05=T04*T45;
x5=T05(1:3, 1);
y5=T05(1:3, 2);

cq6=dot(x6, -x5);
sq6=dot(x6, -y5);
q(6)=atan2(sq6, cq6);
```

Codice 2.6: Calcolo di ϑ_6

Quaternioni

Nello studio della cinematica inversa si è analizzato come le variabili note a priori, per poter giungere alla soluzione del problema stesso, sono la posizione e l'orientamento dell'end-effector rispetto alla base del manipolatore stesso.

Come visto nel paragrafo 2.3.2 a pagina 13 si è in grado di esprimere la rotazione di una terna mobile rispetto ad una fissa tramite una matrice detta *matrice di rotazione*: un modo però più conciso per poter esprimere l'informazione in merito alla rotazione è rappresentato dall'utilizzo dei quaternioni. Essi rappresentano delle entità complesse che, data una matrice di rotazione

in questa forma

$$\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} \quad (2.9)$$

possono essere scritti come:

$q_1 = \frac{\sqrt{x_1+y_2+z_3+1}}{2}$	
$q_2 = \frac{\sqrt{x_1-y_2-z_3+1}}{2}$	$signq_2 = sign(y_3 - z_2)$
$q_3 = \frac{\sqrt{y_2-x_1-z_3+1}}{2}$	$signq_3 = sign(z_1 - x_3)$
$q_4 = \frac{\sqrt{z_3-x_1-y_2+1}}{2}$	$signq_4 = sign(x_2 - y_1)$

La funzione atan2

Sempre nella risoluzione della cinematica inversa, per andare a calcolare i 6 parametri di giunto, si vanno ad utilizzare concetti geometrici-matematici, andando a richiamare soprattutto la geometria riguardante i triangoli rettangoli. Nello specifico, è importante saper gestire quello che è il corretto segno da assegnare agli angoli calcolati, e questo viene ad essere fatto tramite la funzione atan₂ la quale segue il comportamento presentato in figura 2.24.

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{se } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{se } x < 0 \wedge y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{se } x < 0 \wedge y < 0 \\ +\frac{\pi}{2} & \text{se } x = 0 \wedge y > 0 \\ -\frac{\pi}{2} & \text{se } x = 0 \wedge y < 0 \\ \text{non definita} & \text{se } x = 0 \wedge y = 0 \end{cases}$$

Figura 2.24: Caratteristiche della funzione atan₂

Capitolo 3

Il controllore IRC5 Compact

Il manipolatore IRB 120 per poter essere gestito e per poter funzionare necessita di ricevere dall'esterno sia dei segnali di controllo che di alimentazione: di questo compito se ne occupa il controller IRC 5 Compact (*Industrial Robot Controller*) (figura 3.1), un'unità di controllo compatta che ha in sé tutti i benefici del IRC5, tra cui la precisione dell'analisi del movimento e l'utilizzo del linguaggio di programmazione RAPID, oltre a dimensioni nettamente ridotte.

La parte di controllo della movimentazione, è individuata dalla presenza di un anello in retroazione sul sistema, in grado di valutare l'uscita e di andare a confrontarla con il valore di riferimento posto in ingresso. Questa parte di controllo della traiettoria e della movimentazione non deve però essere esplicitamente sviluppata come un blocco a se stante, ma risulta già essere sviluppato all'interno del controllore stesso. IRC5 Compact consente infatti ai robot di eseguire le proprie attività ad altissima efficienza, grazie ad una modellazione dinamica avanzata, che permette di ottimizzare automaticamente le prestazioni del robot riducendo i tempi di ciclo (*QuickMove^(TM)*) e garantendo che il percorso seguito dal robot sia lo stesso di quello programmato,



Figura 3.1: IRC5 Compact controller

indipendentemente dalla velocità di movimento (*TrueMove^(TM)*).

La tecnologia di IRC5 di ABB consente di prevedere il movimento del robot, e garantisce prestazioni elevate, senza necessità di messa a punto da parte del programmatore, ovvero senza la necessità di dover andare a gestire il controllo sulla posizione.

Tu programmi, e lui esegue alla perfezione.

Nella nostra applicazione, come in tutte le applicazioni che vanno a fare uso dei manipolatori industriali ABB, la presenza dell'IRC5 è inoltre fondamentale perché permette di offrire una via per interfacciarsi e per gestire la movimentazione del braccio: questo è possibile grazie al fatto che esso supporta tutti i bus di campo attualmente in commercio in maniera tale che il robot possa integrarsi in ogni tipo di rete.

Tra le numerose funzioni di networking ricordiamo il socket messaging (scambio di messaggi TCP/IP) e la possibilità di interfacciarsi con diversi sensori e accessi remoti: questo argomento è approfondito nella sottosezione 3.1.2 a fronte.

A livello "pratico", il controllore presenta delle interfacce destinate a gestire la parte di alimentazione e/o la parte di controllo del movimento del manipolatore: andiamo ora ad analizzare velocemente quelli che sono i principali collegamenti che il controllore offre.

3.1 Pannello di controllo

Il controller IRC5 presenta a bordo, oltre ad una serie di collegamenti, che gli permette di interfacciarsi al mondo esterno, anche una serie di pulsanti/interruttori situati sul pannello anteriore dell'IRC5 Compact.

3.1.1 Pulsanti e interruttori

La figura 3.2 nella pagina successiva mostra i pulsanti e gli interruttori situati sul pannello anteriore del controller IRC5 Compact. Le loro funzioni sono:

A: Interruttore principale di alimentazione

B: Pulsante di rilascio dei freni che permette di modificare manualmente la posizione degli assi del manipolatore.

C: Interruttore di scelta modalità di funzionamento del sistema robotico, la quale può essere:

- *Automatica*: definibile come “modalità di produzione” in cui il manipolatore si muove secondo quelle che sono le istruzioni RAPID (capitolo 6 a pagina 43) che sono caricate a bordo del controllore. In questa modalità di funzionamento la velocità non risulta essere

3.1. PANNELLO DI CONTROLLO

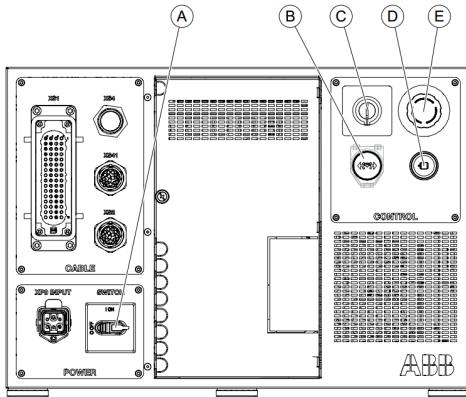


Figura 3.2: Pulsanti e interruttori presenti sul pannello anteriore del controller IRC5

limitata, ma bensì dipende da quelle che sono le informazioni di velocità inserite all'interno della codifica delle istruzioni di movimento.

- *Manuale:* in questa modalità il controllo avviene manualmente tramite il jog sulla flex pendant (figura 4 a pagina 39); la velocità di movimento risulta invece essere limitata a 250 mm/s.

D: Motori inseriti

E: Arresto di emergenza. La pressione di questo tasto si rende necessaria nel momento in cui si attiva la modalità di funzionamento automatica. In particolar modo essa permette di attivare i motori che inizialmente risultano essere in blocco per motivi di sicurezza

3.1.2 Interfacce di collegamento

Di seguito viene descritta l'interfaccia di collegamento sul IRC5 Compact. In prima battuta si analizzano le interfacce che permettono al controllore IRC5 di collegarsi al manipolatore IRB 120, fornendogli tutti i segnali necessari. In seconda battuta andremo ad analizzare rapidamente i collegamenti che il controllore stesso presenta verso il "mondo esterno". Partiamo analizzando i collegamenti interni tra manipolatore e controller in figura 3.3 nella pagina seguente

A: Connessione alla FlexPendant

B: Collegamento dell'alimentazione al robot

C: Collegamento degli assi aggiuntivi

D: Collegamento del robot

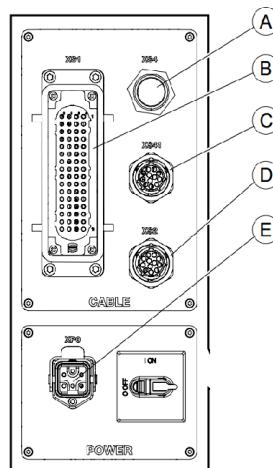


Figura 3.3: Connitori presenti sul pannello anteriore del controller IRC5

E: Collegamento dell'alimentazione di rete

Il pannello di controllo dell'IRC5, oltre ai collegamenti esposti in precedenza, presenta sicuramente altre porte di comunicazione, che permettono al controller di entrare in comunicazione con un PC/Sistema esterno.

In quello che è definito come il gruppo "computer" troviamo diversi collegamenti come si vede in figura 3.4 nella pagina successiva. I due collegamenti principali sono:

Porta di servizio X2: questa porta è destinata ai tecnici di servizio e ai programmati che si collegano direttamente al controller con un PC. La porta di servizio è configurata con un indirizzo IP fisso, che è identico per tutti i controller e non può essere modificato, ed è dotata di un server DHCP che assegna automaticamente un indirizzo IP al PC collegato: quindi, nel momento in cui si va a gestire la movimentazione del manipolatore tramite RobotStudio (capitolo 6 a pagina 43), il PC deve essere connesso al controller tramite la porta di servizio.

Durante il collegamento alla porta di servizio il PC deve essere impostato su “Ottieni automaticamente un indirizzo IP” come descritto in Service PC information nel Boot Application nella Flex Pendant (capitolo 4 a pagina 40).

Porta di rete della fabbrica X6: La porta di rete di fabbrica (WAN) è una porta pubblica destinata al collegamento del controller a una rete, tramite la porta X6 sul pannello di controllo. Per completezza si riporta in figura una panoramica dei collegamenti possibili via Ethernet/USB. Le impostazioni di rete possono essere configurate con qualsiasi indirizzo IP fornito, in genere, dall'amministratore della rete: per il PC dipendono

3.1. PANNELLO DI CONTROLLO

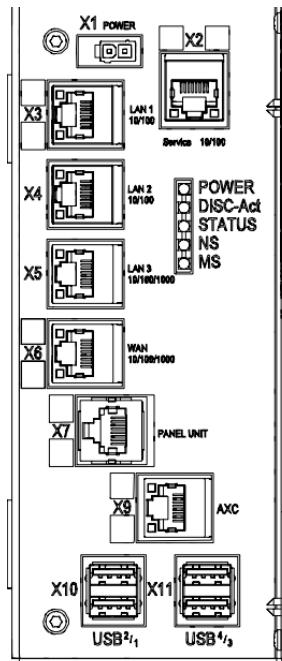


Figura 3.4: Porte di comunicazione IRC5

dalla configurazione della rete medesima da parte dell'amministratore di rete. Questo ci permette di affermare che un secondo modo per comunicare con il controllore può essere tramite una rete (cablata o meno) alla quale risulta essere connesso il controllore stesso.

Per la porta WAN non si possono utilizzare i seguenti indirizzi IP, in quanto già assegnati ad altre funzioni sul controller IRC5:

- 192.168.125.0 - 255
- 192.168.126.0 - 255
- 192.168.127.0 - 255
- 192.168.128.0 - 255
- 192.168.129.0 - 255
- 192.168.130.0 - 255

La porta WAN non può trovarsi su una subnet con uno degli indirizzi IP riservati sopra riportati. Se occorre utilizzare una subnet mask nell'intervallo B della classe di indirizzi, specificare un indirizzo privato di classe B per evitare conflitti.

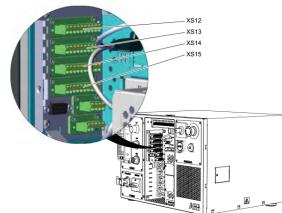


Figura 3.5: Collegamenti I/O dell'IRC 5 Compact

3.1.3 Morsettiera di supporto

Sul pannello di controllo dell'IRC5 Compact è inoltre presente una morsettiera che permette di effettuare collegamenti esterni (a livello "elettrico"). Vediamo quelli che sono i principali collegamenti e le loro funzionalità.

XS7-XS8-XS9 Questi connettori sono collegati internamente alla scheda di sicurezza e contengono i seguenti segnali:

- Arresto automatico
- Arresto generale
- Arresto di emergenza esterno
- Alimentazione esterna

XS12...XS15 Questi connettori (figura 3.5) sono collegati internamente all'unità I/O: essi contengono 16 segnali di ingresso digitali, 16 segnali di uscita digitali, 24 V e 0 V per le uscite e 0 V per gli ingressi. 24 V e 0 V devono essere di alimentazione esterna.

XS16 Questo connettore (figura 3.6) è collegato internamente all'unità I/O e all'unità di distribuzione dell'alimentazione a 24 V con un assorbimento di corrente non superiore a 6A.

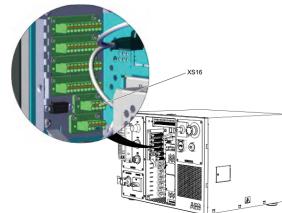


Figura 3.6: Collegamento alimentazione esterna dalla morsettiera dell'IRC5 Compact

Parte II

Il controllo del sistema

Capitolo 4

Movimentazione manuale

Una prima via per controllare il sistema, è rappresentata dalla modalità manuale: essa permette di gestire direttamente, in *tempo reale*, la posizioni di ognuno dei 6 giunti del manipolatore IRB 120, ad una velocità però limitata per questioni di sicurezza.

Ciò è possibile grazie alla *Flex pendant* in alcuni casi indicata anche come *TPU* o Teach Pendant Unit, la quale è utilizzata per eseguire molte delle operazioni correlate al funzionamento di un sistema robotico: esecuzione dei programmi, movimento manuale del manipolatore, modifica di programmi a bordo del controllore ecc... [18]

Per capire quali sono gli utilizzi principali in cui la FlexPendant ricopre un ruolo di primaria importanza, andiamo ad analizzare quelle che sono le principali parti *fisiche* componenti l'unità stessa (figura 4.1):

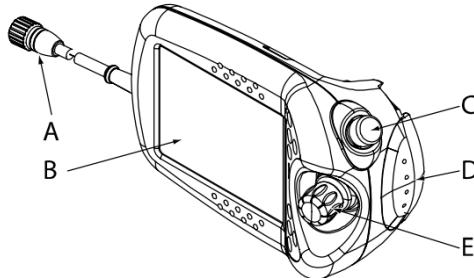


Figura 4.1: Componenti principali della FlexPendant
[18]

I componenti principali della FlexPendant sono:

- A: Connettore
- B: Touch screen
- C: Pulsante di arresto di emergenza

D: Dispositivo di attivazione

E: Joystick

Si vede come, la FlexPendant, sia decisamente adatta per la movimentazione manuale tramite il pratico joystick: esso permette di gestire contemporaneamente la movimentazione di tre assi per volta, unitamente ad un supporto visivo a schermo, che permette di configurare la movimentazione.

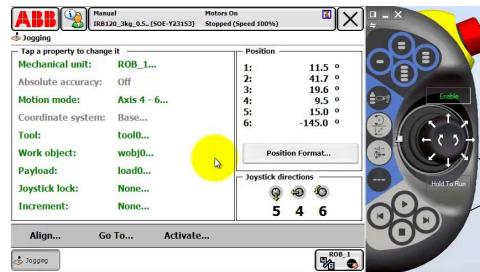


Figura 4.2: Schermata di controllo della movimentazione manuale tramite joystick

Come si può vedere in figura 4.2, la FlexPendant permette di visualizzare, in tempo reale, la posizione dei 6 giunti, tramite una visualizzazione che può essere impostanti in gradi oppure in radienti, permettendo anche di gestire la movimentazione giunto per giunto oppure in maniera lineare o circolare, a seconda delle esigenze.

Si può inoltre andare a selezionare quale dei 3 giunti (1-2-3 o 4-5-6) sono oggetto di movimentazione tramite jog manuale, quale manipolatore, nei sistemi multimove, si vuole gestire, visualizzando unitamente anche i sistemi di coordinate alle quali il moto si riferisce. Questa modalità di utilizzo rappresenta sicuramente un modo rapido e sicuro per gestire in maniera diretta la posizione del manipolatore: a livello di prestazioni e di funzionalità, come si può ben capire, è decisamente però una modalità molto limitante, al solo scopo dimostrativo e/o di setting-up.

Capitolo 5

Movimentazione automatica

Una soluzione decisamente più robusta, in grado di rendere la gestione della movimentazione più flessibile e precisa, è rappresentata dal controllo tramite l'ambiente di sviluppo ufficiale di ABB, il quale presenta nel software *RobotStudio* l'elemento chiave.



Figura 5.1: Logo RobotStudio

RobotStudio è un'applicazione distribuita ufficialmente da ABB il quale mette a disposizione tutti gli strumenti per aumentare la redditività del sistema robotizzato, consentendo di svolgere attività di formazione, programmazione e ottimizzazione senza interferire con la produzione: esso è basato su ABB VirtualController, una copia esatta del software che controlla il funzionamento dei robot in produzione. In questo modo si possono effettuare simulazioni 3D estremamente realistiche, utilizzando programmi e file di configurazione reali, identici a quelli usati sull'impianto.

Il punto centrale è proprio questo: RobotStudio mette a disposizione la possibilità di simulare, su PC, il funzionamento e il controllo del manipolatore, il quale, una volta terminato lo studio in un ambiente virtuale, potrà essere spostato sul controllore reale, che sarà poi in grado di determinare un controllo automatizzato del manipolatore stesso, permettendo

- Riduzione del rischio
- Avvio più veloce
- Meno tempo per le modifiche

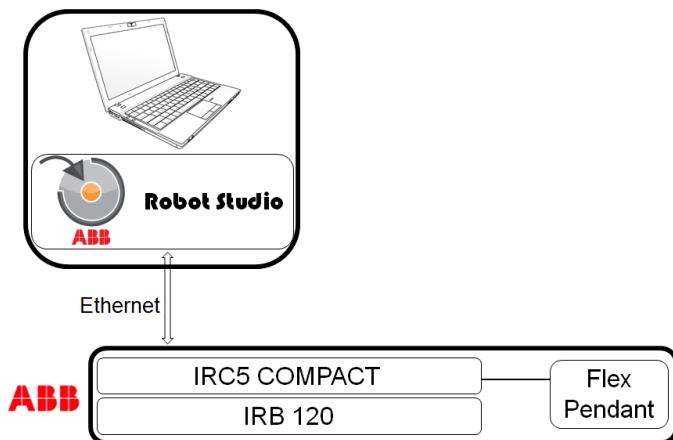


Figura 5.2: Schema del funzionamento online del sistema di controllo automatico

- Aumento della produttività

RobotStudio mette a disposizione delle funzioni di utilizzo in una modalità d'uso detta *offline*: questo significa che, quando *RobotStudio* funziona senza essere connesso ad un controller reale, esso permette di simulare, tramite un controller *IRC5* virtuale su PC, il funzionamento del manipolatore. L'altra modalità di funzionamento è detta *online*, e si presenta nel momento in cui *RobotStudio* va ad agire direttamente su un controller reale.

Nel momento in cui si va a lavorare in modalità online, è ovviamente necessario introdurre un collegamento tra il PC e il controllore: l'*IRC5 Compact* offre molte possibilità di collegamento; una prima possibilità si presenta andando ad inserire il controller in una rete, per esempio in una rete interna di fabbrica, rendendo possibile la comunicazione di tutti i nodi della rete stessa con il controllore.

Un modo decisamente più diretto per poter gestire la programmazione online, è rappresentato dal collegamento del controller al PC tramite la porta di servizio (3.1.2 a pagina 34): con un semplice cavo Ethernet è possibile mettere in comunicazione il PC con il controller *IRC5 Compact*, per caricare il codice sorgente a bordo del controller stesso, permettendo quindi un controllo del manipolatore in modalità online, la quale ci consentirà, come riportato nella sezione 7 a pagina 47, di aprire una comunicazione *socket* per effettuare lo scambio di particolari tipologie di messaggi (definiti come *package*).

Capitolo 6

RobotStudio: programmazione statica

Come prima spiegato, *RobotStudio* permette di effettuare una programmazione del movimento che può essere considerata statica: infatti, una volta parametrizzato e settato il codice da fornire al controller IRC5 Compact, non si ha la possibilità di interagire in "tempo reale" con il posizionamento del robot.

Ovviamente la caratteristica "saliente" di una gestione di questo tipo è il fatto che si può operare una pianificazione del movimento in ambito virtuale, sfruttando un controller simulato, il quale offre le funzionalità del controller fisico, ma solamente su una piattaforma virtuale. Nell'ambiente di sviluppo *RobotStudio*, rappresentato nella figura 6.1, si ha un forte orientamento ad una programmazione *statica*: il software mette infatti a disposizione la possibilità di andare a fissare dei sistemi di riferimento (*target*) che, se inseriti all'interno di un percorso (*path*), permettono di definire una movimentazione che il manipolatore poi potrà andare ad eseguire nella realtà, a patto che le



Figura 6.1: Virtualizzazione del sistema fisico su piattaforma *RobotStudio*

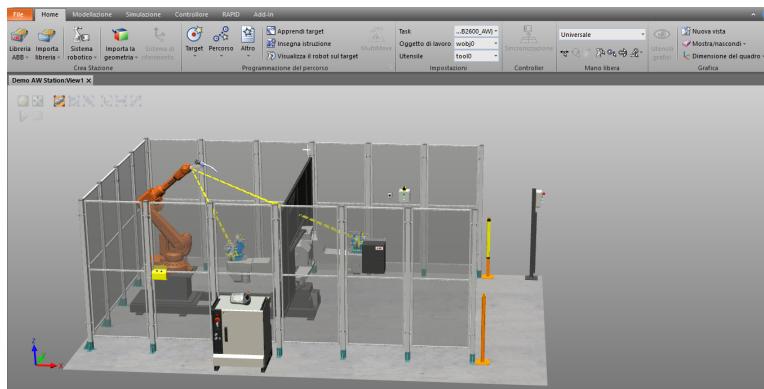


Figura 6.2: Esemplificazione del livello di complessità dei sistemi simulabili in *RobotStudio*

posizioni stesse siano raggiungibili dal manipolatore stesso. La creazione di path è solamente una delle tante possibilità di virtualizzazione che *RobotStudio* mette a disposizione: si possono andare a ricreare sistemi con geometrie molto più complesse come, per esempio, sistemi di saldatura (figura 6.2) oppure di pick and place, permettendone così un’analisi del funzionamento e della gestione in maniera completamente svincolata dalla realtà. Altre funzionalità implementate nell’ambiente di *RobotStudio*, che permettono un controllo più capillare del manipolatore, sono:

- Tabelle degli eventi: permettono di verificare la struttura del programma e la sua logica visualizzando gli stati degli I/O.
- Rilevamento delle collisioni: si tratta di uno strumento più orientato agli ambienti reali di operazione utilizzato per prevenire possibili collisioni del robot con l’ambiente esterno durante l’esecuzione del programma.
- Visual Basic for Applications (*VBA*) per la creazione di interfacce di utilizzo

Tutto questo è però possibile, oltre a particolari motori grafici che permettono di modellizzare il sistema reale, anche grazie al linguaggio di programmazione *RAPID*.

Il cuore della gestione sta quindi nel codice *RAPID*: si tratta di un linguaggio ad alto livello steso ad hoc da ABB nel 1994 per poter controllare e gestire la movimentazione dei propri manipolatori, tramite specifiche istruzioni che permettono di definire le modalità, le tempistiche, le velocità di movimento dei robot.

Generalizzando, potremmo dedurre che il programma è una sequenza di istruzioni che controlla il robot e che generalmente consiste di tre parti (figura 6.3 a fronte): routine principale, subroutine e dati del programma.

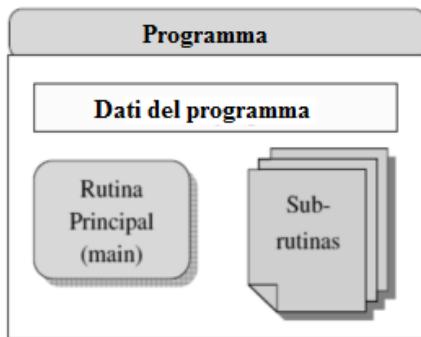


Figura 6.3: Schema base della struttura del programma *RAPID*

- Routine principale: routine dove si inizia l'esecuzione (main)
- Soubroutine: servono a dividere il programma in parti più piccole per ottenere un programma modulare
- Dati del programma: definire posizioni, valori numerici, sistemi di coordinate, ecc.

Quindi, oltre alle più comuni strutture di programmazione, quali per esempio *if*, *cicli for*, sono presenti particolari tipologie di istruzioni (istruzioni per spostare il robot oppure per impostare un determinato output), il cui comportamento è definito tramite argomenti che gli vengono passati in ingresso, e anche specifiche tipologie di dati, il cui ruolo è mirato ad una più facile gestione del movimento del robot, come si può facilmente notare nel listato 6.1 nella pagina successiva.

Tutte queste istruzioni che vanno a gestire la movimentazione e la traiettoria del manipolatore ABB possono essere raggruppate in uno o svariati moduli: ciascun modulo può contenere una o svariate procedure. Si possono distinguere:

Moduli di programma: in essi viene memorizzato il codice RAPID che è individuato da un file che con estensione .mod (Module1.mod per esempio). Per il controller del robot non fa alcuna differenza se il programma è scritto in più moduli: il motivo di utilizzare più moduli in un programma è soltanto quello di rendere il programma più facile da comprendere, e più agevole da riutilizzare da parte dei programmatore. Ovviamente, nonostante si possa avere più moduli installati sul controllore, è importante che solamente uno di essi contenga la procedura main, che verrà poi eseguita ripetutivamente dal controller.

Moduli di sistema: questa particolare tipologia di moduli, salvati con estensione .sys, permettono di mantenere dati e procedure nel sistema, anche se il programma viene cambiato. Ad esempio, se una variabile

persistente, di tipo *tooldata* viene dichiarata in un modulo di sistema, una calibrazione dell'utensile viene preservata, anche se venisse ad essere caricato un nuovo programma. In questa tipologia di moduli non si ha però la possibilità di inserire delle soubrutine di *main*: ciò significa che l'esecuzione del codice non potrà mai partire da questi moduli, ma dovrà iniziare da moduli di programma, i quali poi potranno andare a richiamare le funzioni strutturate all'interno del modulo di sistema.

```

MODULE Module1

CONST robtarget HomePosition
    :=[[364.302,0,593.9999],[0.498,0,0.802,0],[0,0,0,0],[9E+09,9E
    +09,9E+09,9E+09,9E+09,9E+09]];
% NB: La posizione e' definita rispetto al sistema di coordinate
% oggetto corrente
CONST robtarget PosizioneIngresso_10
    :=[[515,325,75],[0,0.604,-0.5,0],[0,0,0,0],[9E+09,9E+09,9E+09,9E
    +09,9E+09,9E+09]];
TASK PERS wobjdata Wobj_CellaPianta:=[FALSE,TRUE
    ,","",[[200,200,0],[1,0,0,0]],[[0,0,0],[1,0,0,0]]];
VAR robtarget TargetMobile;
VAR num NumeroCicli := 0;
VAR num m := 1;
CONST robtarget PuntoIntermedio
    :=[[650,-215.8,150],[0.046,0.007,0.882,-0.002],[0,0,0,0],[9E
    +09,9E+09,9E+09,9E+09,9E+09]];
PROC main()
IF (NumeroCicli = 0) THEN
    TPErase;
    %Per limitare le possibilita' di avere singolarita'
    SingArea\LockAxis4;
    MoveL HomePosition,v50,z5,tool0\WObj:=wobj0;
ENDIF
%m = numero di righe che si e' intenzionati a scansionare; es: m =
1 -> visito solamente la prima riga
IF (NumeroCicli MOD m = 0 AND (NOT(NumeroCicli = 0))) THEN
    Y_Offset := 0;
    X_Offset := X_Offset -21;
ENDIF
TPWrite("Offset Y: "+ValToStr(Y_Offset));
TPWrite("Offset X: "+ValToStr(X_Offset));
TargetMobileEstrazione := Offs(PartenzaCellaEstrazione,X_Offset,
    Y_Offset,0);
TargetMobile := Offs(PartenzaCella,X_Offset,Y_Offset,0);
Y_Offset := Y_Offset - 22;
NumeroCicli :=NumeroCicli+1;
ENDPROC
ENDMODULE

```

Codice 6.1: Esempio di programmazione *RAPID*

Presentare però in questa sede il linguaggio *RAPID*, con il giusto livello di approfondimento, sarebbe praticamente impossibile e, al contempo, fuoriluogo: per una maggiore trattazione si rimanda il lettore ai riferimenti [5] e [2].

Capitolo 7

Programmazione dinamica: case study

La pianificazione della traiettoria e la gestione della movimentazione in *RobotStudio* è sicuramente molto "user-friendly": si ha la possibilità di generare automaticamente codice *RAPID* a seconda di come viene configurato il sistema nella sezione di visualizzazione, creando traiettorie ad hoc, con il minimo sforzo.

Il problema di un approccio di questo tipo è però che non si ha la possibilità di gestire in "tempo-reale" la movimentazione del robot: una volta caricato il listato *RAPID* a bordo del controllore, esso lo eseguirà in maniera ciclica, fino a che non giungerà ad un punto di arresto inserito nel codice oppure fino a quando non verrà arrestato dall'esterno (tramite, per esempio, il fungo d'emergenza).

In progetti molto complessi vi è però spesso la necessità di creare traiettorie adattive, a seconda di quello che è lo stato dell'ambiente in cui il robot si trova a dover lavorare: è necessario quindi sapere gestire la traiettoria del manipolatore a seconda di quelle che sono le informazioni ricevute per esempio da fotocamere, sensori visuali o di posizione, variando in maniera dinamica la posizione e l'orientamento dell'end-effector, e quindi del robot stesso, in base al feedback ricevuto. Nello studio delle diverse possibilità di controllo del manipolatore IRB 120 di ABB sono state prese in considerazioni molte alternative: si è pensato inizialmente di andare a fornire a *RobotStudio* un file con estensione .txt contenente tutte le coordinate che il manipolatore doveva andare a raggiungere. In questa prima soluzione la parte dinamica sarebbe stata fornita da una scrittura continua del file .txt delle coordinate, in maniera tale che il codice statico strutturato in ambiente *RAPID* potesse poi andare gestire in maniera iterativa la posizione del manipolatore.

Una seconda soluzione presa in considerazione, è stata quella di utilizzare un metodo di controllo basato sulla comunicazione tramite *OPC Server*: *OPC* (OLE for Process Control) è uno standard di comunicazione nel campo

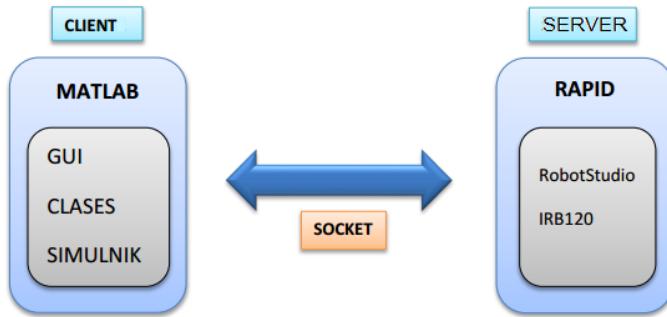


Figura 7.1: Configurazione di una comunicazione OPC tra client e server

di controllo e supervisione dei processi industriali, basato su una tecnologia Microsoft, che offre un’interfaccia comune per la comunicazione consentendo a singoli componenti software di interagire e condividere dati attraverso un’architettura client-server, il cui ipotetico schema sarebbe potuto essere quello in figura 7.1. Questo tipo di soluzione è decisamente performante poichè permette di andare a lavorare in ambiente *MATLAB*, tramite alcuni toolbox dedicati che consentono una gestione maggiormente ottimizzata della cinematica inversa, con la possibilità anche di andare a lavorare con sistemi costruiti tramite blocchi in ambiente Simulink.

La terza soluzione presa in considerazione è stata quella di andare ad utilizzare ROS per andare a gestire dinamicamente la traiettoria del manipolatore, a seconda dello stato dell’ambiente: quindi, per la disponibilità di informazioni reperibili sul web, unitamente ad un *know-how* di partenza in ambiente di sviluppo Ubuntu, è stato scelto di andare a seguire e sviluppare un controllo basato su questa opzione.

Capitolo 8

ROS: *Robot Operating System*



Figura 8.1: *Robot Operating System*

Andiamo ora ad introdurre, all'interno di questo progetto, una piattaforma software chiamata *Robot Operating System*, più comunemente detta ROS, la quale risulta essere frutto di un enorme sviluppo subito negli ultimi anni dalla comunità della robotica, il quale ha permesso la creazione di algoritmi e ambienti in grado di andare a gestire piattaforme robotiche con un sempre maggiore livello di autonomia e intelligenza. ROS, come si può vedere [21], è

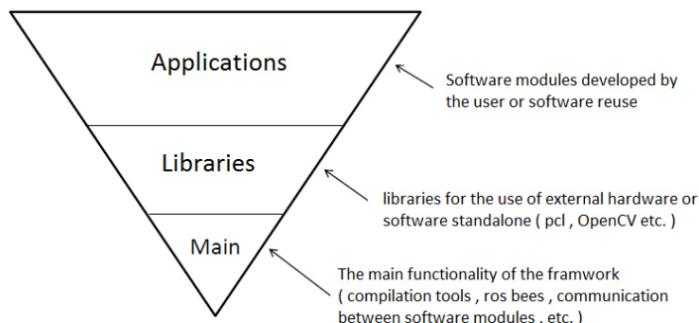


Figura 8.2: Struttura a componenti di ROS

definibile come:

ROS è un sistema operativo ope-source, con una struttura basata su componenti, che fornisce librerie e tools per aiutare gli sviluppatori a creare robot applications. Esso fornisce astrazione dell'hardware, driver dei

dispositivi, librerie, strumenti di visualizzazione, comunicazione a scambio di messaggi tra processi (message passing), gestione dei pacchetti e molto altro.

Prima di entrare nello studio dell'architettura della nostra applicazione, risulta essere utilie andare ad introdurre quelli che sono i componenti principali: ROS, essendo infatti un sistema operativo, crea una rete (figura 8.3) dove tutti i processi sono connessi tra di loro.

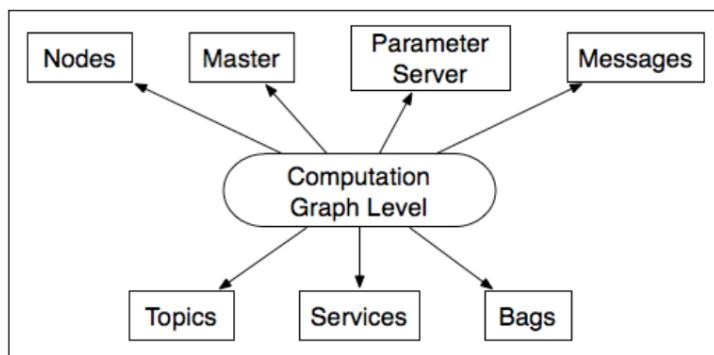


Figura 8.3: Concetti fondamentali su cui si basa una "network" ROS

Nello specifico, all'interno di questa rete, possiamo andare ad individuare diversi elementi caratterizzanti, quali:

Nodes: un *nodo* è un processo che esegue delle operazioni di calcolo, fornendo così un modo per poter andare a dividere le funzionalità del sistema in più blocchi, semplificandone di molto il funzionamento e l'utilizzo. Le operazioni a cui si faceva riferimento in precedenza sono sostanzialmente implementate tramite l'utilizzo di diverse librerie (ROS client library) come roscpp, che permette di produrre nodi scritti in C++, oppuree rospy, per l'ambiente Python. Ogni nodo può andare a comunicare con altri nodi attraverso *topics*, RPC services e tramite *Parameter server*.

Per comprendere meglio il concetto di nodo, è bene evidenziare come, in un sistema di controllo per sistemi robotici, sono compresi solitamente più nodi, uno per esempio va a controllare gli azionamenti delle ruote, uno computa la posizione, uno calcola il percorso (*path planning*) etc.

Ad ogni singolo nodo è associato un nome univoco all'interno del sistema, in maniera tale che la comunicazione tra di essi possa avvenire senza ambiguità. In particolar modo, l'architettura "a nodi" che caratterizza il sistema ROS si basa sul concetto di.

Roscore: il nodo roscore, considerabile come il nodo master, è l'elemento fondamentale che si occupa di andare a coordinare la connessione tra i nodi che sono legati a roscore stesso gestendone la comunicazione

Rosnode: esso rappresenta la tipologia di nodi sviluppati dall'utente, tramite la stesura di codice, quale per esempio C++ oppure Python.

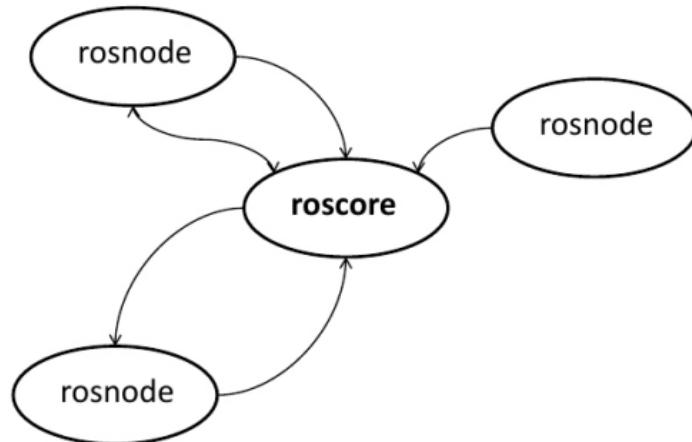


Figura 8.4: Elementi principali dell'architettura ROS

Topics: essi non sono altro che "bus" attraverso i quali i nodi vanno a scambiarsi messaggi, rendendo possibile una programmazione multi-thread, ignorando i problemi relativi alla sincronizzazione tra processi. Nello specifico, tramite l'utilizzo di topics, si va a creare una semantica "anonima" di comunicazione, definita *publish-subscribers*, che permette di disaccoppiare, come detto in precedenza, le parti di produzione delle informazioni da quelle che le informazioni invece vanno a consumarle.

In questo protocollo di comunicazione e scambio di messaggi, i nodi non sono a conoscenza del destinatario con cui stanno comunicando. I nodi stessi sono invece interessati al tipo di dati che si vanno a scambiare tramite un dato topic: infatti, ogni singolo topic, è fortemente tipizzato da ROS, il chè permette ad ogni topic di avere più subscribers.

Questo significa che tramite un dato topic, i nodi che vi sono connessi (*subscribers*), possono scambiarsi solamente una certa tipologia di messaggi: quindi è molto importante sottolineare il fatto che un nodo può diventare *publisher* su un certo topic, solamente se il tipo di messaggio che vogliono pubblicare coincide con quello "caratterizzante" il topic in questione e allo stesso tempo può sottoscrivere un topic solamente se la tipologia dei messaggi coincide.

Services: il modello publish/subscriber è sicuramente un paradigma di comunicazione molto flessibile, ma il suo trasporto unidirezionale *multi a multi* non è appropriato per le interazioni di richiesta e risposta tipiche della comunicazione tra nodi. Questo concetto di richiesta/risposta è quindi realizzato tramite *service*, come si vede nella figura 8.5, il quale

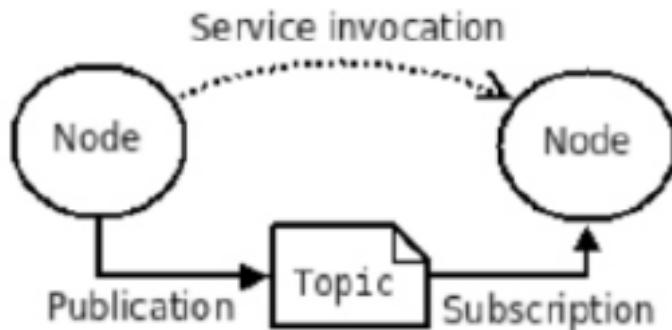


Figura 8.5: Comunicazione basata sul protocollo *publisher-subscribers*

è definito da una coppia di messaggi: uno per la richiesta e uno per la risposta. In sostanza un nodo ROS offre un servizio, sotto un certo nome, a tutti gli altri nodi del sistema, i quali possono richiedere il dato servizio inviando il messaggio di richiesta, attendendo poi la risposta.

Messages: i nodi comunicano tra di loro pubblicando messaggi all'interno di topics definiti. Nello specifico, questi messaggi sono rappresentati da una semplice struttura di dati, comprendente dei campi tipizzati. Come evidenziato nel punto precedente, i nodi possono anche scambiarsi un messaggio di richiesta e risposta come parte di una chiamata di servizio ROS i quali sono definiti nei file srv.

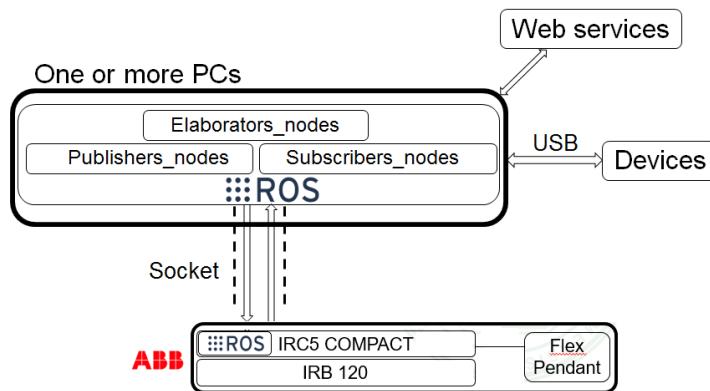


Figura 8.6: Struttura completa del sistema utilizzato

I due elementi sicuramente caratterizzanti la comunicazione e lo scambio di informazioni all'interno di ROS sono sicuramente i nodi, i quali rappresentano programmi che si scambiano dati attraverso topics: ogni singolo topic ha un proprio nome e può accettare solamente un tipo di messaggio.



Figura 8.7: ROS Industrial

In ROS sono inclusi molti tipi di messaggi, a seconda della tipologia di informazioni che si ha la necessità di trasmettere: un esempio di messaggio utilizzato molto spesso all'interno di questo progetto è "*geometry_msgs/Pose*", il quale ha una propria struttura interna composta da 7 variabili di tipo float, 3 delle quali definiscono la posizione e 4 invece definiscono l'orientamento (in quaternioni).

Per mandare messaggi è quindi necessario eseguire un'azione di *publish* su uno specifico topic per quella data tipologia di messaggi; per riceverli è invece necessario *subscribe*, ovvero sottoscriversi ad un determinato topic.

```
%Node 1:  
#include <geometry_msgs/Pose.h>  
ros::Publisher pub = n.advertise<geometry_msgs::Pose>("object_position",10);  
pub.publish(send_position);  
  
%Node 2:  
ros::Subscriber sub  
handle_function(geometry_msgs::Pose received_position)  
{  
...  
}  
ros::Subscriber sub = n.subscribe("object_position",10,handle_function);
```

Codice 8.1: Semplice esempio di comunicazione publish-subscribe

8.1 ROS industrial e IRB 120

La struttura di ROS presentata in precedenza rappresenta un'architettura generica, la quale però non fornisce gli strumenti adatti per poter andare a gestire la traiettoria e la movimentazione dei manipolatori utilizzati in ambito industriale. Le potenzialità dell'ambiente ROS possono però essere estese anche all'ambito manifatturiero e produttivo, permettendo anche di realizzare applicazioni robotiche di produzione che in precedenza erano tecnicamente irrealizzabili e proibitive. Questo è possibile grazie a *ROS-I* (ROS Industrial) il quale, tramite la presenza molte repository online open source, fornisce

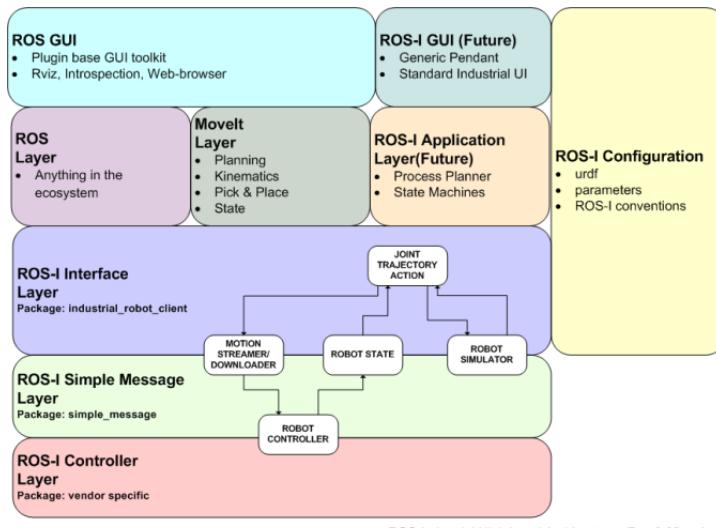


Figura 8.8: Architettura ad alto livello di ROS Industrial

interfacce per i più diffusi manipolatori industriali, end-effector, sensori e dispositivi connessi in rete, fornendo molti benefici come

- Aumentare le potenzialità di ROS
- Applicare la strada a nuove applicazioni
- Rendere la programmazione dei robot più ad alto livello
- Ridurre i costi
- Totalmente open-source con una corposa comunità di ricerca e sviluppo

Benefici che sono sostanzialmente dovuti alla forte struttura stratificata che *ROS-I* fornisce (figura 8.8).

Volendo quindi andare a gestire la movimentazione di un manipolatore ABB, dovremo essere in grado di dare a *ROS-I* le giuste informazioni in merito alla geometria del robot stesso unitamente alle caratteristiche intrinseche: questo è possibile grazie ai pacchetti software che *ROS-Industrial* mette a disposizione. Tali pacchetti possono essere suddivisi in

- Pacchetti software generici, in grado di fornire delle funzionalità generali
- Pacchetti software specifici, i quali forniscono supporto per il setup e la configurazione di molte piattaforme robotiche

8.2. COMUNICAZIONE SOCKET: CONCETTI BASE

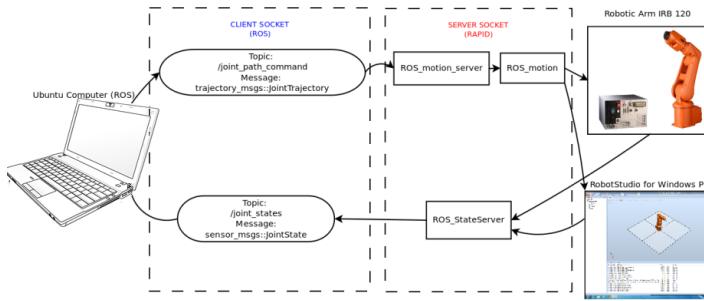


Figura 8.9: Schematizzazione della comunicazione socket tra client e server con relativo flusso di dati

Tra questi pacchetti specifici, troviamo anche quelli che permettono di andare ad eseguire un setup dell’ambiente ROS per quello che riguarda manipolatori della linea di ABB. Andiamo quindi ora ad analizzare nello specifico quella che è la struttura *publish-subscriber* che il pacchetto ABB implementato in *ROS-I* mette a disposizione.

8.2 Comunicazione socket: concetti base

Come già precedentemente presentato, il classico flusso di lavoro e di comunicazione tra il robot IRB 120 e il controller IRC5 Compact è rappresentato dall’andare a creare un programma in linguaggio RAPID, simulandolo in RobotStudio e caricandolo nel controller IRC5 per poi eseguirlo.

Il nostro obiettivo è quello però di andare ad effettuare un controllo esterno, permettendo l’esecuzione di applicazioni più complesse, come quelle che includono sensori esterni: questa esigenza può essere soddisfatta andando ad utilizzare un socket per permettere al client, ovvero i nodi in ambiente ROS, di comunicare con il server, che nel nostro caso sarà rappresentato dal controller IRC5 Compact.

Un socket è oggetto software che permette l’invio e la ricezione di dati, tra host remoti (tramite una rete) o tra processi locali (Inter-Process Communication). Più precisamente, il concetto di socket si basa sul modello Input/Output su file di Unix, quindi sulle operazioni di open, read, write e close; l’utilizzo, infatti, avviene secondo le stesse modalità, aggiungendo i parametri utili alla comunicazione, quali indirizzi, numeri di porta e protocolli.

Con tale termine (che letteralmente vuol dire “presa”), in generale, si definisce una rappresentazione a livello software utilizzata per interfacciare due terminali (endpoint) in gioco in una connessione. In altre parole, potremmo considerare i socket come delle prese (una per ogni macchina) che siano interconnesse tra loro attraverso un ipotetico cavo in cui passi il flusso di dati che i computer si scambiano.

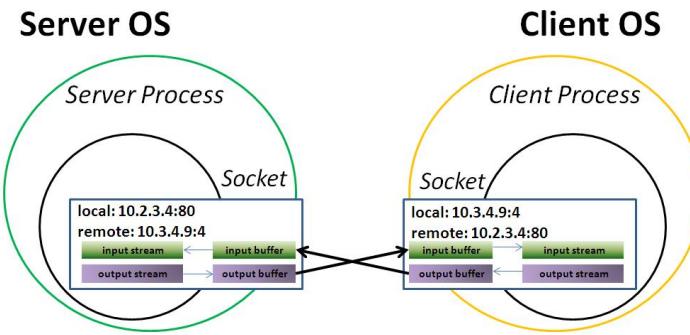


Figura 8.10: Configurazione schematica della comunicazione socket

Un esempio che rende bene l’idea è quello di pensare ai socket come alle prese telefoniche presenti ai due capi opposti durante una conversazione al telefono. Le due persone che colloquiano al telefono comunicano attraverso le rispettive prese. La conversazione, in tal caso, non finirà finché non verrà chiusa la cornetta e fino ad allora la linea resterà occupata. Questa conversazione deve però essere univocamente individuata dato che sui sistemi interlocutori possono esserci molti processi: bisogna quindi avere un modo per indirizzare precisamente il processo con cui si sta dialogando, utilizzando le porte, ovvero dei numeri che identificano i processi in esecuzione. Gli interlocutori, quindi, memorizzano indirizzo e porta della controparte, in un indirizzo socket, formato così:

- Indirizzo IP (32 bit)
- Numero di porta (16 bit)

In base a come avviene la comunicazione, ovvero lo scambio di pacchetti, si possono avere diverse tipologie di comunicazione socket: in questa applicazione si va a sfruttare la solidità del protocollo TCP (come si può vedere nel listato 8.2 dove non è specificato l’argomento UDP), il quale garantisce una comunicazione affidabile, full-duplex, e con un flusso di byte di lunghezza variabile: essa prende il nome di *Stream socket* e presenta sostanzialmente 4 fasi ben definite, che poi andremo anche a ritrovare nella parte di codice RAPID.

```
IF (SocketGetStatus(server_socket) = SOCKET_CLOSED) SocketCreate server_socket;
```

Codice 8.2: Istruzione RAPID di creazione del socket

Creazione del socket: client e server creano i loro rispettivi socket, e il server lo pone in ascolto su una porta. Dato che il server può creare più connessioni con client diversi (ma anche con lo stesso), ha bisogno di una coda per gestire le varie richieste.

Richiesta di connessione: il client effettua una richiesta di connessione verso il server. Da notare che possiamo avere due numeri di porta diversi, perchè una potrebbe essere dedicata solo al traffico in uscita, l'altra solo in entrata; questo dipende dalla configurazione dell'host. In sostanza, non è detto che la porta locale del client coincida con quella remota del server: il server riceve la richiesta e, nel caso in cui sia accettata, viene creata una nuova connessione.

Comunicazione: ora client e server comunicano attraverso un canale virtuale, tra il socket del primo, ed uno nuovo del server, creato appositamente per il flusso dei dati di questa connessione. È possibile, quindi, che ci siano molti client a comunicare con il server, ciascuno verso il data socket creato dal server per loro.

Chiusura della connessione: essendo il TCP un protocollo orientato alla connessione, quando non si ha più la necessità di comunicare, il client lo comunica al server, che ne deistanzia il data socket. La connessione viene così chiusa.

8.3 Messaggio scambiato tramite socket

La comunicazione tra client e server, la cui struttura interna sarà spiegata nelle sezioni 8.4 a pagina 60 e 8.6 a pagina 65, avviene tramite l'apertura di una connessione socket basata su un protocollo di trasporto di tipo TCP, il quale garantisce una comunicazione più sicura e controllata. Nello specifico client e server si scambiano messaggi che hanno una struttura predefinita, in maniera tale che entrambi siano in grado di interpretare i dati ricevuti/spediti. Ovviamente questa struttura deve essere definita all'interno dei file di supporto presenti all'interno del *catkin workspace* installato in ambiente Unix: il file che specifica la struttura del messaggio è il file *simple_message.h* che può essere trovato sotto il percorso `|catkin_ws|src|industrial_core|simple_message|include|simple_message.h`.

All'interno di questo file si può vedere come il messaggio sia strutturato in 3 macro-parti:

Prefix: non è considerato parte del messaggio e contiene la lunghezza, in bytes, dell'intero messaggio composto da Header+Body

Header: è formato sostanzialmente da 3 sotto campi i quali rappresentano dei parametri che consentono una corretta identificazione del tipo di messaggio (*message type code*) che si sta mandando e dello stato della comunicazione. Essi sono:

MSG_TYPE: permette di identificare il tipo di messaggio tramite dei valori standard e specifici a seconda del tipo di robot con cui si

sta lavorando, tramite la dichiarazione di un'enumerazione, come si può vedere nel listato 8.3, il quale è definito sempre all'interno del file *simple_message.h*.

```
namespace StandardMsgTypes
{
    enum StandardMsgType
    {
        INVALID = 0,
        PING = 1,

        JOINT_POSITION = 10,
        JOINT = 10,
        READ_INPUT = 20,
        WRITE_OUTPUT = 21,

        JOINT_TRAJ_PT = 11,    %Joint trajectory point message (typically for
                             %streaming)
        JOINT_TRAJ = 12,      %Joint trajectory message (typically for
                             %trajectory downloading)
        STATUS = 13,          %Robot status message (for reporting the robot
                             %state)
        JOINT_TRAJ_PT_FULL = 14, %Joint trajectory point message (all message
                             %fields)
        JOINT_FEEDBACK = 15,    %Feedback of joint pos/vel/accel

        % Begin vendor specific message types (only define the beginning enum
        % value,
        % specific enum values should be defined locally, within in the range
        % reserved
        % here. Each vendor can reserve up 100 types

        SWRI_MSG_BEGIN      = 1000,
        UR_MSG_BEGIN        = 1100,
        ADEPT_MSG_BEGIN     = 1200,
        ABB_MSG_BEGIN       = 1300,
        FANUC_MSG_BEGIN     = 1400,
        MOTOMAN_MSG_BEGIN   = 2000
    };
}
typedef StandardMsgTypes::StandardMsgType StandardMsgType;
```

Codice 8.3: Definizione dei parametri del campo MSG_TYPE

COMM_TYPE: codici che permettono di identificare il tipo di comunicazione, indipendentemente dal tipo di robot con cui si sta lavorando.

```
namespace CommTypes
{
    enum CommType
    {
        INVALID = 0,
        TOPIC = 1,
        SERVICE_REQUEST = 2,
        SERVICE_REPLY = 3
    };
}
typedef CommTypes::CommType CommType;
```

Codice 8.4: Definizione dei parametri del campo COMM_TYPE

REPLY_CODE: dato la presenza di una sistema di feedback che permette di verificare la corretta movimentazione del manipolatore, è necessario che il server non sia in grado solamente di ricevere traiettorie ma anche di rielaborarle, costruendo un messaggio nel formato standard, e poi inviarle. Si ha quindi la necessità di avere dei parametri che permettono di ritornare informazioni rilevanti in caso di successo oppure di errore.

8.3. MESSAGGIO SCAMBIATO TRAMITE SOCKET

```
namespace ReplyTypes
{
    enum ReplyType
    {
        INVALID = 0,
        SUCCESS = 1,
        FAILURE = 2
    };
}
typedef ReplyTypes::ReplyType ReplyType;
```

Codice 8.5: Definizione dei parametri del campo REPLY_CODE

Body: bytarray composto dai dati che effettivamente si vogliono trasferire tra client e server, tra cui anche la posizione che i 6 giunti del manipolatore devono assumere

Oltre a queste 3 macro-parti in cui è strutturato il messaggio, sono presenti anche dei "Special sequence code" i quali permettono di andare a discriminare i vari punti della traiettoria, riconoscendo quelli che sono i punti iniziali e finali della traiettoria stessa, come è descritto nel listato 8.6.

```
namespace SpecialSeqValues
{
    enum SpecialSeqValue
    {
        START_TRAJECTORY_DOWNLOAD = -1, % Downloading drivers only: signal start of trajectory
        START_TRAJECTORY_STREAMING = -2, % deprecated, please use START_TRAJECTORY_STREAMING
                                         instead
        START_TRAJECTORY_STREAMING = -2, % Streaming drivers only: signal start of trajectory
        END_TRAJECTORY = -3, % Downloading drivers only: signal end of trajectory
        STOP_TRAJECTORY = -4 % Server should stop the current motion (if any) as soon as
                           possible
    };
}
typedef SpecialSeqValues::SpecialSeqValue SpecialSeqValue;
```

Codice 8.6: Definizione dei codici con uso speciale

Quindi, tutti i parametri fin qui definiti all'interno dell'ambiente Unix, devono trovare un'ovvia corrispondenza nel server installato sul controller: questo effettivamente avviene, tramite la definizione di variabili locali, come evidenziato nel codice 8.7 caricato a bordo del controller.

```
CONST num ROS_MSG_TYPE_INVALID      := 0;
CONST num ROS_MSG_TYPE_JOINT       := 10;
CONST num ROS_MSG_TYPE_JOINT_TRAJ_PT := 11;
CONST num ROS_COM_TYPE_TOPIC       := 1;
CONST num ROS_COM_TYPE_SRV_REQ     := 2;
CONST num ROS_COM_TYPE_SRV_REPLY   := 3;
CONST num ROS_REPLY_TYPE_INVALID   := 0;
CONST num ROS_REPLY_TYPE_SUCCESS   := 1;
CONST num ROS_REPLY_TYPE_FAILURE   := 2;

CONST num ROS_TRAJECTORY_START_DOWNLOAD := -1;
CONST num ROS_TRAJECTORY_END := -3;
CONST num ROS_TRAJECTORY_STOP := -4;

CONST num ROS_MSG_MAX_JOINTS := 10;
```

Codice 8.7: Definizione dei parametri del campo REPLY_CODE

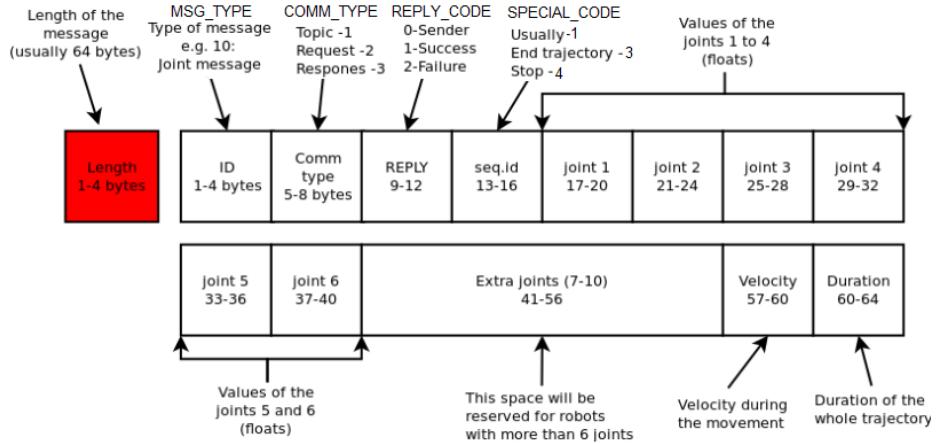


Figura 8.11: Struttura del messaggio scambiato tra client e server

8.4 Client part

Nella comunicazione socket che si va ad aprire, siamo in grado di individuare client e server progettati "ad hoc" per la gestione dei manipolatori ABB.

Le principali funzionalità che un client di questo tipo dovrebbe avere, sono quelle di andare a gestire lo stato corrente dei giunti del manipolatore, ricevendo le informazioni dal controller IRC5 Compact e rielaborandole secondo le necessità. Il primo step è sicuramente quello di andare ad avviare due packages che si trovano all'interno della nostra applicazione:

- abb_driver
- unibg_robot_controller

Infatti tutto il software ROS, come accennato in precedenza, presenta un'organizzazione a package: nello specifico, un package ROS rappresenta una collezione di file, genericamente sia file eseguibili che di supporto, utilizzati per scopi specifici ed organizzati all'interno di una struttura gerarchica ben precisa (si troveranno sempre due file specifici: il *manifesto* con estensione .xml ed un file *CMakeList*, un file .txt da fornire come input al CMake build per costruire la struttura software dei package). Nello specifico dei due package sopra riportati, andiamo a lanciare solamente due parti specifiche, che poi andranno ad avviare l'intero insieme dei nodi. Queste parti sono:

robot_interface.launch: questo file .launch fornisce gli strumenti necessari per andare ad aprire l'effettiva comunicazione socket tra il sistema ROS e il manipolatore ABB, utilizzando il protocollo standard di ROS Industrial spiegato nel paragrafo 8.3 a pagina 57. Possiamo infatti

passare a questo particolare tipo di file alcune informazioni chiave, quali l'indirizzo IP del robot da gestire unitamente al file URDF necessario per definire la geometria del manipolatore stesso.

Utilizzando il comando `roslaunch` si è in grado di lanciare files con estensione .launch, permettendo di andare ad avviare più nodi contemporaneamente. Andando a lanciare il file `robot_interface.launch` si vanno ad avviare nello specifico i seguenti nodi:

robot_state: esso si connette al modulo di programma *State_Server* caricato a bordo del controller, ed è responsabile della pubblicazione dello stato corrente del robot e del valore in gradi che stanno assumendo i 6 giunti del manipolatore.

motion_download_interface: gestisce la movimentazione del robot, spedendo al controllore IRC5 Compact punti formanti una traiettoria.

joint_trajectory_action: si tratta di un nodo che permette di fornire al controller i punti che il manipolatore deve raggiungere (figura 8.12) e toccare per seguire una specifica traiettoria, segnalando allo stesso tempo la corretta esecuzione della stessa e il raggiungimento dell'obiettivo.

Questo nodo presenta inoltre la possibilità di imporre dei vincoli alla movimentazione, dando la possibilità di poter interrompere l'esecuzione della traiettoria stessa nel momento in cui i vincoli risultano essere violati.

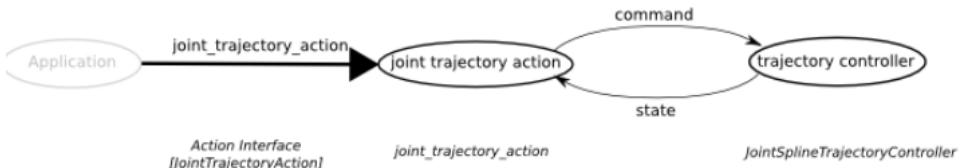


Figura 8.12: Funzionalità del nodo `joint_trajectory_action`

unibg_simple_move_node: nel punto precedente, l'avvio del file `robot_interface.launch` ci ha permesso di avviare contemporaneamente più nodi. In questo caso, utilizzando il comando `rosrun`, andiamo ad avviare un singolo nodo ROS, il quale permette di andare a pubblicare messaggi di tipo `control_msgs::FollowJointTrajectoryActionGoal` sul topic `joint_trajectory_action/goal`.

Nello specifico, il messaggio che viene pubblicato sul topic ha un a struttura ben precisa, la quale è formata da:

std_msgs/Header header: esso è utilizzato per trasmettere l'informazione in merito al timestamp della comunicazione unitamente

all'identificativo del tipo di sistema di riferimento utilizzato per definire le coordinate (*no_frame* or *global_frame*).

actionlib_msgs/GoalID goal_id: esso contiene un timestamp che rappresenta l'istante in cui è stata richiesta una certa coordinata; questo istante è utilizzato dal server, posto sul controller, quando tenta di andare ad annullare tutte le traiettorie richieste prima di un certo tempo. Oltre al timestamp è presente un campo identificativo univoco che permette di associare un certo feedback ad una specifica traiettoria.

control_msgs/FollowJointTrajectoryGoal goal in questo campo del messaggio pubblicato dal nostro nodo *unibg_simple_move_node* si vanno a definire tra parametri importanti:

goal_time_tolerance: indica la tolleranza temporale entro il quale si impone il raggiungimento del punto stabilito

path_tolerance e goal_tolerance: se la misura dei parametri di joint di velocità, posizione e accelerazione cade fuori dal range di tolleranza, la traiettoria è abortita.

trajectory_msgs/JointTrajectory trajectory: qui è contenuta l'informazione più importante, ovvero quella che riguarda i punti che vanno a comporre la traiettoria, unitamente al nome dei joint.

L'informazione contenuta nel messaggio denominato precedentemente come *goal* sarà quindi poi pubblicato dal nostro nodo, tramite i comandi mostrati nel listato 8.8.

```
ros::Publisher pub_joint_action = n.advertise<control_msgs::FollowJointTrajectoryActionGoal>("joint_trajectory_action/goal", 1);
control_msgs::FollowJointTrajectoryActionGoal goal;

goal.goal.trajectory.header.stamp = ros::Time::now() + ros::Duration(1.0);
int ind = 0;
goal.goal.trajectory.points[ind].positions.resize(6);
goal.goal.trajectory.points[ind].positions[0] = -1.3963;
goal.goal.trajectory.points[ind].positions[1] = 0.65;
goal.goal.trajectory.points[ind].positions[2] = 0.5461;
goal.goal.trajectory.points[ind].positions[3] = 0.0;
goal.goal.trajectory.points[ind].positions[4] = 0.3747;
goal.goal.trajectory.points[ind].positions[5] = 0.0;
goal.goal.trajectory.points[ind].time_from_start = ros::Duration(5.0);
```

Codice 8.8: Pubblicazione messaggio *trajectory* di esempio nel topic corrispondente

Gli elementi caratterizzanti il sistema sono però i topic: infatti, una volta avviati i nodi (per mezzo dei comandi *rosrun* e *roslaunch*), sono avviati alcuni topic specifici, i quali possono andare a scambiare solamente determinate tipologie di messaggi (per la struttura dei messaggi legati ai singoli topic fare riferimento a [22]). I topic che si creano, come si può vedere nella figura 8.13 a fronte sono:

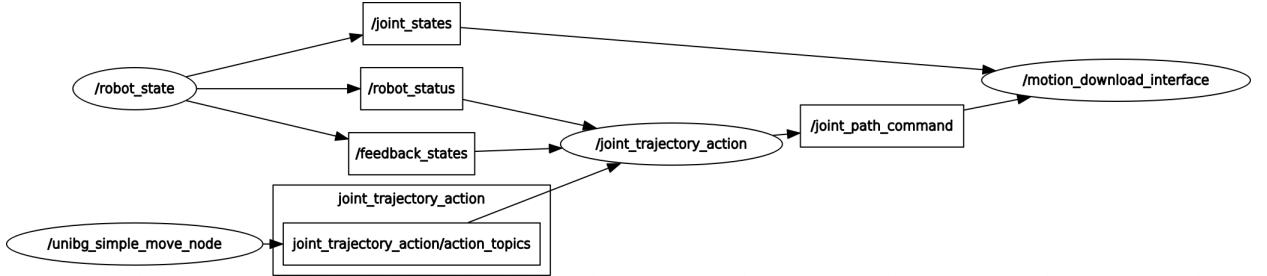


Figura 8.13: Grafo delle dipendenze tra topics e nodi

- /feedback_states
- /joint_path_command
- /joint_trajectory_action/cancel
- /joint_trajectory_action/feedback
- /joint_trajectory_action/goal
- /joint_trajectory_action/result
- /joint_trajectory_action/status
- /robot_status

8.5 Architettura ABB ROS Server

Come visto in precedenza (capitolo 8 a pagina 51), il fulcro del controllo tramite ROS sta proprio in un continuo scambio di messaggi sul modello *publisher-subscriber*, caratterizzante *ROS*: essa può essere quindi considerata come una struttura client-server, dove il client è rappresentato da un insieme di nodi *ROS* in grado di pubblicare particolari tipologie di messaggi (sezione 8.4 a pagina 60).

Il server è invece rappresentato, per forza di cose, dal controllore IRC5 Compact, il quale deve essere in grado di andare a ricevere le informazioni necessarie per gestire la traiettoria del robot, per mezzo dell’instaurazione di una comunicazione tramite socket.

Per poter andare a configurare il controllore è necessario fare riferimento ancora all’ambito di programmazione *RobotStudio*: in questo caso però, una volta configurato il codice RAPID e caricato a bordo del controller stesso, non avremo più bisogno di modificare la struttura, poiché la traiettoria verrà gestita direttamente dall’ambiente ROS. Nello specifico risultano essere necessari tre task:

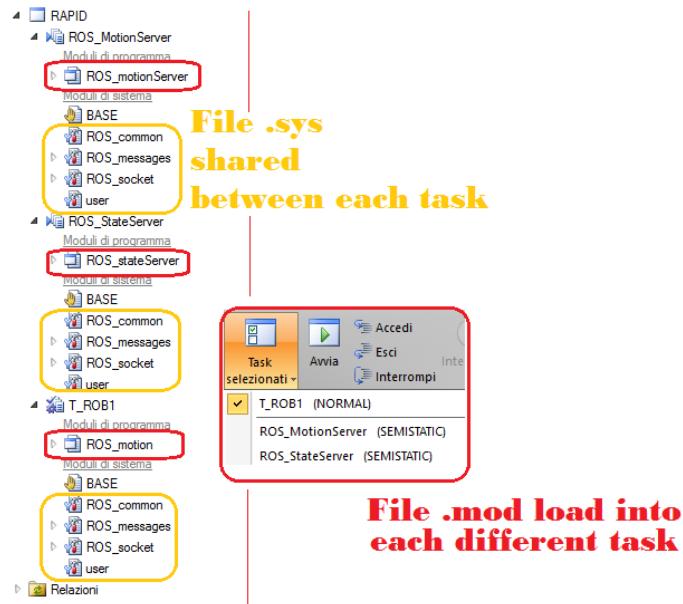


Figura 8.14: Struttura multitasking del server ROS implementato in codice RAPID

- T_ROB1
- ROS_StateServer
- ROS_MotionServer

La struttura principale della nostra applicazione *RAPID* è quindi formata da più programmi *RAPID*, i quali procedono la loro esecuzione in parallelo, a partire dal momento in cui l'esecuzione inizia, fino ad un tempo potenzialmente infinito(a meno di errori presenti all'interno del codice stesso): non tutti e tre i task hanno però lo stesso peso e le stesse funzionalità, come si può vedere nella figura 8.15 nella pagina successiva.

Il task T_ROB1 rappresenta l'unico task che può andare gestire direttamente la movimentazione del manipolatore tramite istruzioni di movimento RAPID, essendo configurato come *Motion Task*: per le proprietà della programmazione in RAPID, qualsiasi task impostato come task di movimento, deve essere di tipo NORMAL, ovvero deve poter essere eseguito in primo piano. Questo implica che, per andare a movimentare il manipolatore, è necessario, come primo step, andare ad attivare questo task.

ROS_StateServer e ROS_MotionServer sono invece configurati come task SEMISTATIC: si tratta quindi di task in *background*, il cui funzionamento non può essere controllato, ovvero vengono eseguiti automaticamente nel momento in cui il controllore viene avviato.

Questa rappresenta però la struttura "statica" dei task che vanno a comporre il server ROS (sezione 8.6 a fronte): in ognuno di esso è necessario

Name	Type	Trust Level	Entry	Motion Task
ROS_StateServer	SEMISTATIC	NoSafety	main	NO
ROS_MotionServer	SEMISTATIC	SysStop	main	NO
T_ROB1	NORMAL		main	YES

Figura 8.15: Configurazioni dei task del ROSserver

andare a caricare moduli di programma (capitolo 6 a pagina 45) contenenti le istruzioni RAPID da eseguire. Si tratta dei moduli *ROS StateServer.mod*, *ROS MotionServer.mod* e *TROB_1*, ai quali vanno aggiunti anche i moduli di sistema *ROS common.sys*, *ROS socket.sys* e *ROS messages.sys*, i quali risultano essere condivisi tra tutti i task.

La parte "dinamica" dell'intera applicazione risulta essere invece sviluppata e costruita in ambiente Unix, grazie alle funzionalità offerte dal sistema operativo ROS (sezione 8.4 a pagina 60).

8.6 Server part

8.6.1 ROS_MotionServer

Essendo che il modulo caricato all'interno del task omonimo si trova in continua esecuzione, una volta che viene stabilita la connessione socket tra client e server, esso inizia a ricevere messaggi contenenti i punti formanti le traiettorie inviate dal client ROS, come si può vedere nel listato:

```

LOCAL CONST num server_port := 11000;

LOCAL VAR socketdev server_socket;
LOCAL VAR socketdev client_socket;
LOCAL VAR ROS_joint_trajectory_pt trajectory{MAX_TRAJ_LENGTH};
LOCAL VAR num trajectory_size;

PROC main()
    VAR ROS_msg_joint_traj_pt message;

    TPWrite "MotionServer: Waiting for connection.";
    ROS_init_socket server_socket, server_port;
    ROS_wait_for_client server_socket, client_socket;

    WHILE ( true ) DO
        % Recieve Joint Trajectory Pt Message
        ROS_receive_msg_joint_traj_pt client_socket, message;
        trajectory_pt_callback message;
    ENDWHILE

    ERROR (ERR_SOCK_TIMEOUT, ERR_SOCK_CLOSED)
    IF (ERRNO=ERR_SOCK_TIMEOUT) OR (ERRNO=ERR_SOCK_CLOSED) THEN
        SkipWarn; % TBD: include this error data in the message logged
        below?
        ErrWrite \W, "ROS MotionServer disconnect", "Connection lost.
                    Resetting socket.";
        ExitCycle; % restart program
    ENDIF
ENDPROC

```

```

    ELSE
        TRYNEXT;
    ENDIF
UNDO
    IF (SocketGetStatus(client_socket) <> SOCKET_CLOSED) SocketClose
        client_socket;
    IF (SocketGetStatus(server_socket) <> SOCKET_CLOSED) SocketClose
        server_socket;
ENDPROC

```

Codice 8.9: Ricezione delle traiettorie con relativo controllo d'integrità

Nella ricezione dei punti che compongono una data traiettoria è però importante andare a distinguerne due: il punto iniziale e il punto finale della traiettoria stessa.

Quindi, unitamente alla ricezione del messaggio per mezzo della procedura ROS_receive_msg_joint_traj_pt condivisa tramite il modulo di sistema ROS_messages, si va a richiamare la procedura locale trajectory_pt_callback la quale, con una semplice struttura switch case effettuato sul campo *sequence_id* del messaggio, permette di andare a verificare che tipo di punto stiamo andando ad analizzare.

```

% use sequence_id to signal start/end of trajectory download
TEST message.sequence_id
CASE ROS_TRAJECTORY_START_DOWNLOAD:
    TPWrite "Traj START received";
    trajectory_size := 0;                                % Reset trajectory size
    add_traj_pt point;                                 % Add this point to the
                                                       trajectory
CASE ROS_TRAJECTORY_END:
    TPWrite "Traj END received";
    add_traj_pt point;                                % Add this point to the
                                                       trajectory
    activate_trajectory;
CASE ROS_TRAJECTORY_STOP:
    TPWrite "Traj STOP received";
    trajectory_size := 0;  % empty trajectory
    activate_trajectory;
    StopMove; ClearPath; StartMove; % redundant, but re-issue stop
                                    command just to be safe
DEFAULT:
    add_traj_pt point;                                % Add this point to the
                                                       trajectory
ENDTEST

```

Codice 8.10: Struttura che permette di riconoscere la natura del punto da aggiungere alla traiettoria in esame

8.6.2 ROS_StateServer

Il compito principale del modulo di programma contenuto in questo task, è quello di andare ad inviare messaggi broadcast contenenti informazioni e dati in merito alla posizione dei giunti. In particolar modo la funzionalità di questo modulo, il quale rimane in esecuzione continuamente in background, è

8.6. SERVER PART

molto importante: esso permette, una volta stabilita la connessione socket, di inviare al client, ad un determinato rate (*update_rate*), la posizione corrente dei 6 giunti.

Si chiude così l'anello di retroazione, tramite un feedback che risulta essere molto importante per quelle applicazioni in cui è necessaria un'elevata precisione nella movimentazione.

```
LOCAL CONST num server_port := 11002;
LOCAL CONST num update_rate := 0.10; % broadcast rate (sec)

LOCAL VAR socketdev server_socket;
LOCAL VAR socketdev client_socket;

PROC main()

    TPWrite "StateServer: Waiting for connection.";
    ROS_init_socket server_socket, server_port;
    ROS_wait_for_client server_socket, client_socket;

    % Spedizione con periodo pari a \emph{update_rate} del valore
    % attuale dei joint
    WHILE (TRUE) DO
        send_joints;
        WaitTime update_rate;
    ENDWHILE
    ...

```

Codice 8.11: Feedback continuo della posizione del manipolatore

8.6.3 ROS_Motion: effettiva movimentazione del manipolatore

In questo task, all'interno del modulo caricatovi a bordo, come anticipato nel paragrafo 8.5 a pagina 64, avremo le effettive istruzioni che vanno a mettere in movimento il robot, essendo che esso è caricato all'interno del task T_ROB1, il quale è definito come l'unico *Motion Task*.

Il concetto base è che dall'ambiente ROS, più nello specifico dal modulo ROS_motion_server.mod, si ricevono un insieme di punti (o *targets*), i quali vanno a comporre la traiettoria che il manipolatore dovrà andare a seguire.

In sostanza si scandisce con un ciclo tutta la lista dei target, la cui lettura è eseguita nella soubrutine *init trajectory()*, tramite l'utilizzo di un lock sulla lettura cosicchè nessuno vada sovrascrivere la traiettoria mentre se ne sta effettuando l'acquisizione.

```
% Il server e' settato per ricevere una serie di target e poi andare
% ad eseguire un insieme conosciuto di posizioni, non dando la
% possibilita' di aggiungere target "on-the-fly"

%Definizione di un nuovo tipo di variabile come
%RECORD ROS_joint_trajectory_pt
% robjoint joint_pos; --> posizione in gradi di ognuno dei 6 assi
% num duration; --> durata della movimentazione
```

```

%ENDRECORD
LOCAL VAR ROS_joint_trajectory_pt trajectory{MAX_TRAJ_LENGTH};
LOCAL VAR num trajectory_size := 0;
LOCAL VAR intnum intr_new_trajectory;

PROC main()
    VAR num current_index;
    VAR jointtarget target;
    VAR speeddata move_speed := v10; % velocita' di default
    VAR zonedata stop_mode;
    VAR bool skip_move;

    WHILE true DO
        % La variabile "ROS_new_trajectory" e' una variabile di tipo "
        % PERSISTENT bool", %la quale permette di mantenere l'ultimo
        % valore salvato, anche in caso di %interruzione dell'
        % esecuzione, ed e' necessaria per segnalare se e' disponibile
        % o meno una nuova %traiettoria
        IF (ROS_new_trajectory)
            % Acquisizione, con lock, della traiettoria
            init_trajectory;

            % Analizzo i punti componenti la traiettoria
            IF (trajectory_size > 0) THEN
                FOR current_index FROM 1 TO trajectory_size DO
                    %Acquisisco i valori in gradi dei 6 giunti
                    target.robax := trajectory{current_index}.joint_pos;

                    % Se il primo punto e' vicino (entro i limiti di tolleranza)
                    % al punto da raggiungere, allora devo saltare questo
                    % target
                    skip_move := (current_index = 1) AND is_near(target.robax,
                        0.1);
                    IF (current_index = trajectory_size)
                        stop_mode := fine; % stop at path end

                    =====
                    % Effettiva esecuzione del movimento
                    IF (NOT skip_move)
                        MoveAbsJ target, move_speed, \T:=trajectory{current_index
                            }.duration, stop_mode, tool0;
                    =====
                ENDFOR

                trajectory_size := 0; % trajectory done
            ENDIF

            WaitTime 0.05; % Throttle loop while waiting for new command
        ENDWHILE

        ERROR
            ErrWrite \W, "Motion Error", "Error executing motion. Aborting
                trajectory.";
            abort_trajectory;
    ENDPROC

```

Codice 8.12: Movimentazione effettiva del manipolatore

8.6.4 ROS_Socket: apertura della connessione socket lato server

Nell'architettura del server, ROS_socket rappresenta un modulo di sistema (vedi 6 a pagina 45): esso quindi risulta essere condiviso tra tutti i task, i quali hanno quindi la possibilità di richiamare le funzioni che vi sono al suo interno. Nello specifico, il modulo di sistema in esame, contiene 4 routine che permettono di gestire l'interazione con il socket, le quali sono:

ROS_init_socket: in questa funzione si vanno a definire i parametri della connessione, quali l'indirizzo IP e il numero della porta, e successivamente lo si va a creare, associandolo alla variabile di tipo *socketdev* fornita in ingresso alla funzione.

```
% SocketGetStatus e' un istruzione che restituisce lo stato  
    corrente di un socket,  
% il quale puo' essere:  
% - SOCKET_CREATED  
% - SOCKET_CONNECTED  
% - SOCKET_BOUND  
% - SOCKET_LISTENING  
% - SOCKET_CLOSED  
% Se non esiste alcuna connessione socket la vado a creare (   
    connessione basata su TCP poiche' non e' specificato  
    alcun  
% argomento in input all'istruzione SocketCreate).  
IF (SocketGetStatus(server_socket) = SOCKET_CLOSED)  
    SocketCreate server_socket;  
% Se il socket e' stato gia' creato allora associo al socket  
    stesso l'indirizzo ip specificato del server e il numero  
    della  
% sua porta. Da evidenziare il fatto che l'indirizzo del  
    server e' ottenuto automaticamente, come indirizzo del  
    controller  
% tramite l'istruzione GetSysInfo  
IF (SocketGetStatus(server_socket) = SOCKET_CREATED)  
    SocketBind server_socket, GetSysInfo(\LanIp), port;  
  
% Qui il controller, che lavora come un server, inizia  
    effettivamente ad ascoltare e ricevere le comunicazioni  
    in ingresso  
IF (SocketGetStatus(server_socket) = SOCKET_BOUND)  
    SocketListen server_socket;
```

Codice 8.13: Analisi funzionamento routine ROS-init-socket

ROS_wait_for_client: con questa funzione si va a stabilire l'effettiva connessione tra client e server della comunicazione socket.

Nello specifico in questa funzione, si ha una gestione della connessione che possiamo definire "temporizzata": infatti, tra i parametri di input è presente la variabile numerica facoltativa *wait_time* che permette di specificare la quantità massima di tempo riservata all'esecuzione del programma per l'attesa di connessioni in arrivo. Nel caso in cui questo tempo scada, senza che giunga alcuna connessione in ingresso, si andrà

a generare un errore (ERR_SOCK_TIMEOUT) che poi verrà ad essere gestito.

```

VAR string client_ip;
VAR num time_val := WAIT_MAX;
% La funzione present verifica l'utilizzo o meno dell'
% argomento facoltativo "wait_time"
% nella chiamata a questa routine
IF Present(wait_time) time_val := wait_time;

IF (SocketGetStatus(client_socket) <> SOCKET_CLOSED)
    SocketClose client_socket;
WaitUntil (SocketGetStatus(client_socket) = SOCKET_CLOSED);

% L'istruzione SocketAccept, utilizzabile solamente lato
% server, permette di andare ad accettare una
% connessione in arrivo: una volta accettata la connessione,
% lo stato del socket diventa di SOCKET_LISTENING
SocketAccept server_socket, client_socket, \ClientAddress:-
    client_ip, \Time:=time_val;
TPWrite "Client at "+client_ip+" connected.";

```

Codice 8.14: Analisi funzionamento routine ROS-wait-for-client

ROS_receive_msg: questo routine è prevalentemente utilizzata dal modulo di sistema *ROS_messages.sys*, il quale se ne serve per andare a salvare momentaneamente in una variabile "buffer" i dati ricevuti dalla comunicazione socket, tramite l'utilizzo dell'istruzione *UnpackRawBytes*, necessaria per raggruppare il contenuto di un contenitore di tipo rawbytes.

Successivamente tutti i dati ricevuti vengono inseriti all'interno della variabile message, che è quindi pronta per la conversione e per essere elaborata.

ROS_send_msg: questa routine riceve la posizione corrente dei joint del manipolatore, all'interno di una variabile di tipo *ROS_msg*. I dati, in formato raw bytes, sono analizzati, riuniti secondo la struttura del messaggio del protocollo (sezione 8.3 a pagina 57) ed infine vengono spediti attraverso il socket.

Si tratta quindi di una comunicazione di feedback del server verso il client: in sostanza permette al codice sviluppato in ROS di verificare la correttezza della movimentazione.

8.6.5 ROS_messages

In questo modulo di sistema, si vanno a creare 4 tipologie di variabili utilizzate per contenere i dati inerenti alle traiettorie e ai punti che le compongono. Sono presenti inoltre 2 procedure che permettono di operare una trasformazione e uno scambio di messaggi.

ROS_receive_msg_joint_traj_pt: in questa procedura si va a convertire il messaggio *raw_message*, mandato dal client, nella variabile *message*. Si tratta di variabili che hanno sostanzialmente una struttura diversa: *raw_message* è una variabile di tipo ROS_msg la cui struttura è riportata nel listato seguente, mentre *message* è la variabile che viene effettivamente utilizzata per effettuare la movimentazione desiderata, ed essa è di tipo ROS_msg_joint_traj_pt, la cui struttura è invece riportata nel listato 8.16 e nella figura 8.11 a pagina 60.

```
RECORD ROS_msg
    num msg_type;
    num comm_type;
    num reply_code;
    rawbytes data;
ENDRECORD
```

Codice 8.15: Campi caratterizzanti il tipo di variabili ROS-msg

```
RECORD ROS_msg_joint_traj_pt
    num msg_type;
    num comm_type;
    num reply_code;
    num sequence_id;
    robjoint joints;  % in DEGREES
    num velocity;
    num duration;
ENDRECORD
```

Codice 8.16: Campi caratterizzanti il tipo di variabili ROS-msg-joint-traj-pt

Il trasferimento effettivo da una forma di messaggio all'altra, che altro non è che il cuore di questa procedura, unitamente alla conversione in gradi, è riportato nel codice 8.17.

```
UnpackRawBytes raw_message.data, 1, message.sequence_id, \IntX:=DINT;
UnpackRawBytes raw_message.data, 5, message.joints.rax_1, \Float4;
UnpackRawBytes raw_message.data, 9, message.joints.rax_2, \Float4;
UnpackRawBytes raw_message.data, 13, message.joints.rax_3, \Float4;
UnpackRawBytes raw_message.data, 17, message.joints.rax_4, \Float4;
UnpackRawBytes raw_message.data, 21, message.joints.rax_5, \Float4;
UnpackRawBytes raw_message.data, 25, message.joints.rax_6, \Float4;
% Skip bytes 29-44. UNUSED Reserved for Joints 7-10. TBD: copy to extAx?
UnpackRawBytes raw_message.data, 29+(ROS_MSG_MAX_JOINTS-6)*4, message.velocity, \
    \Float4;
UnpackRawBytes raw_message.data, 33+(ROS_MSG_MAX_JOINTS-6)*4, message.duration, \
    \Float4;
% Convert data from ROS units to ABB units
message.joints := rad2deg_robjoint(message.joints);
```

Codice 8.17: Unpacking del messaggio ricevuto dal client

ROS_send_msg_joint_data: in questa routine invece si va ad effettuare un'operazione nella direzione opposta di quella compiuta all'interno della procedura ROS_receive_msg_joint_traj_pt. Si ricostruisce quindi la struttura del messaggio adatta per essere inviata al client, come feedback.

```
PackRawBytes message.sequence_id, raw_message.data, 1, \IntX:=DINT;
PackRawBytes ROS_joints.rax_1, raw_message.data, 5, \Float4;
PackRawBytes ROS_joints.rax_2, raw_message.data, 9, \Float4;
PackRawBytes ROS_joints.rax_3, raw_message.data, 13, \Float4;
PackRawBytes ROS_joints.rax_4, raw_message.data, 17, \Float4;
PackRawBytes ROS_joints.rax_5, raw_message.data, 21, \Float4;
PackRawBytes ROS_joints.rax_6, raw_message.data, 25, \Float4;
FOR i FROM 1 TO ROS_MSG_MAX_JOINTS-6 DO % Insert placeholders for joints 7-10 (
    message expects 10 joints)
    PackRawBytes 0, raw_message.data, 25+i*4, \Float4;
ENDFOR

ROS_send_msg client_socket, raw_message;
```

Codice 8.18: Packing del messaggio per spedizione verso il client come mezzo di feedback

Capitolo 9

Estensioni future

Lo studio condotto in questa sede ha permesso di andare a fornire il *know-how* necessario per poter avviare progetti di ricerca riguardanti questa tipologia di manipolatori: nello specifico siamo riusciti ad avere un quadro più completo di quelle che possono essere le possibili tecniche di gestione e controllo del movimento. Sicuramente, con il tempo avuto a disposizione, ci siamo trovati di fronte a prendere delle scelte: il compito dunque di questo percorso è stato proprio quello di andare ad aprire la strada per lavori/progetti futuri. Nello specifico, è stato sempre tenuto presente che, il primissimo step futuro da eseguire, sarà quello di andare a creare una movimentazione autonoma del robot, per mezzo dell'utilizzo di sensori visuali, quali telecamere, webcam oppure sensori visuali 3D. Contemporaneamente all'introduzione di una gestione autonoma del manipolatore, bisognerà andare a valutare le informazioni fornite come feedback dalla cinematica inversa: sarà infatti necessario andare a discriminare le soluzioni valide da quelle che invece non sono in grado di andare a rispettare vincoli di tipo fisico. Questo potrebbe essere possibile grazie all'utilizzo di nuovi algoritmi di controllo, unitamente ad una stesura di codice completamente orientato verso l'ambiente Matlab.

Bibliografia

- [1] ABB. *Application manual - IRC5 OPC Server help* 3HAC023113-001. Ver. 4.1. 2015.
- [2] ABB. *Istruzioni RAPID, Funzioni e Tipi di dati* - 3HAC050917-007. Ver. E. 2009.
- [3] ABB. *Manuale del prodotto - IRB 120* 3HAC035728-007. Ver. 16.2. 2016.
- [4] ABB. *Manuale del prodotto - IRC5 Compact* 3HAC047138-007. Ver. 17.1. 2017.
- [5] ABB. *Manuale tecnico di riferimento RAPID* - 3HAC16580-7. Ver. H. 2009. URL: <http://www-lar.deis.unibo.it/people/rfalconi/Panoramica%20RAPID.pdf>.
- [6] *Pagina di riferimento ABB sezione Robotics and Motion per manipolatore*. 2018. URL: <http://new.abb.com/products/robotics/industrial-robots/irb-120>.
- [7] *Pagina di riferimento ABB sezione Robotics and Motion per il controllore*. 2018. URL: <http://new.abb.com/products/robotics/it/controllori/irc-5>.
- [8] *Appunti del corso di robotica*. Slides. 2006.
- [9] *Arte 3.2.0: robot IRB 120 ABB*. 2014. URL: <http://arvc.umh.es/arte/doc/arte3.2.0/robots/ABB/IRB120/index.html>.
- [10] Josè Axel e López Cordoba. «*Messa in azione del braccio robotico e sviluppo di applicazioni*». tesi. UNIVERSIDAD DE JEAN Escuela Politecnica Superior de Linares, 2016.
- [11] Corrado Guarino Lo Bianco. *Analisi e controllo dei manipolatori industriali*. A cura di Pitagora Editrice Bologna. 2011.
- [12] ilian Bonec. *Cinématique du robot IRB 120*. Slides. ETS: École de technologie supérieure.
- [13] Davide Brugali. *Meccanica dei manipolatori - Corso di robotica*. Slides. Università degli Studi di Bergamo.

- [14] Oguz Gencer. «Robotic arm controlling & simulation in MATLAB Simulink». tesi. T.R. ISTANBUL AYDIN UNIVERSITY ENGINEERING FACULTY, 2014.
- [15] *IRc5 e FlexPendant: manuale dettagliato di utilizzo*. 2008. URL: http://serioussurvivor.com/wp-content/uploads/2017/03/IRC5-with-FlexPendant-3HAC16590-1_revK_en.pdf.
- [16] International Standard Organization ISO. *Definizione di robot industriale - ISO TR/8373-2.3*.
- [17] L.Sciavicco. *Robotica industriale – Modellistica e controllo di robot manipolatori*. A cura di Mc Graw-Hill. 2000.
- [18] *Manuale operativo e guida introduttiva, IRC5 e RobotStudio*. 2008. URL: http://www-lar.deis.unibo.it/people/rfalconi/GettingStarted_IT.pdf.
- [19] Javier Moriano Martín. «*Trajectory planning for the IRB120 robotic arm using ROS*». tesi. UNIVERSIDAD DE ALCALÁ Escuela Politécnica Superior, 2014.
- [20] *Meccanica - componenti,mobilità,strutture*. URL: <http://www.itiomar.it/studenti/omarobot/1-meccanica.pdf>.
- [21] *Reference page ROS*. 2018. URL: <http://wiki.ros.org/it>.
- [22] *Reference per la struttura dei messaggi*. 2018. URL: <http://docs.ros.org/api/>.
- [23] *Socket: cosa sono e come funzionano*. 2006. URL: <http://www.html.it/articoli/i-socket-1/>.
- [24] *Socket: cosa sono e come funzionano*. 2014. URL: <https://fortyzone.it/socket-cosa-sono/>.
- [25] *Structure of ROS*. 2018. URL: <http://wiki.ros.org/ROS/Concepts>.