UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

SVILUPPO DI UNA LIBRERIA SOFTWARE PER LA PROGRAMMAZIONE DEL ROBOT MANIPOLATORE COMAU SMART SIX

Relatore:
Chiar.mo Prof. Ing. STEFANO CASELLI

Correlatore:
Dott. Ing. JACOPO ALEOTTI

Tesi di laurea di:
DAVIDE VALERIANI

13 dicembre 2010

UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

DEVELOPMENT OF A SOFTWARE LIBRARY FOR PROGRAMMING THE COMAU SMART SIX ROBOT MANIPULATOR

Supervisor:
Chiar.mo Prof. Ing. STEFANO CASELLI

Tutor:
Dott. Ing. JACOPO ALEOTTI

Thesis of:
DAVIDE VALERIANI

13th December 2010

Alla mia famiglia

Un grande ringraziamento desidero rivolgerlo al mio correlatore, il Dott. Ing. Jacopo Aleotti, che mi ha seguito assiduamente fin dai primi tempi, aiutandomi a superare le difficoltà e i problemi.

Ringrazio inoltre il mio relatore, il Prof. Ing. Stefano Caselli, per i preziosi suggerimenti e consigli che mi ha fornito durante tutto il lavoro di tesi.

Anche tutti i ragazzi e le ragazze del laboratorio di robotica meritano un grande ringraziamento per le lunghe giornate di lavoro assieme, in particolare Christian Alzapiedi, che per primo mi ha introdotto nel mondo del robot Comau, Andrea Lattanzi, per l'aiuto che mi ha fornito con le threads e per aver testato la libreria, e Marco Tarasconi, per l'aiuto nel controllo del gripper.

Ringrazio tutti i miei compagni di studio, che per 3 lunghi e faticosi anni mi hanno fatto compagnia a lezione, durante gli esami e le intense giornate di studio: Domenico, Pietro, Manuel, Michele, Alessandro, Giuseppe, Emanuele e Flavio.

Infine, ringrazio la mia famiglia per il sostegno e la comprensione durante quelle giornate in cui, tornando a casa, risultavo scontroso a causa di un insuccesso nelle prove del mio lavoro di tesi. Un ringraziamento particolare va a mio nonno Candido per il sostegno a volte assillante con cui ha accompagnato i miei studi universitari. E un grazie anche a mio nonno Davide, mio omonimo, che sicuramente da lassù ha tifato per me.

Prima Legge: Un robot non può recare danno agli esseri Umani, né può permettere che, a causa del suo mancato intervento, gli esseri Umani ricevano danno.

Seconda Legge: Un robot deve obbedire agli ordini impartiti dagli esseri Umani, a meno che ciò non contrasti con la Prima Legge.

Terza Legge: Un robot deve salvaguardare la propria esistenza, a meno che ciò non contrasti con la Prima o la Seconda Legge.

Isaac Asimov

Indice

Capitolo 1

Introduzione

La maggior parte dei manipolatori robotici industriali viene utilizzata con programmi semplici, spesso scritti dalla stessa ditta produttrice, che permettono al manipolatore di svolgere operazioni ripetitive. Tuttavia, specie in ambito accademico, può essere utile utilizzare il manipolatore per scopi didattici e permettendo a qualsiasi utente con una buona base di conoscenze dei linguaggi di programmazione ad alto livello di scrivere programmi per il manipolatore.

Il robot *Comau Smart Six*, manipolatore a 6 giunti rotoidali, ad esempio, può essere programmato solamente utilizzando una libreria software proprietaria (PDL2) basata su un linguaggio di scripting procedurale.

Il principale scopo della tesi elaborata è quindi quello di realizzare una libreria software in C++ per la programmazione del robot, più flessibile e semplice da utilizzare della libreria proprietaria.

La libreria progettata è basata su un'architettura Client/Server, in cui Client e Server comunicano tra loro utilizzando una socket. Il lato client è costituito dall'applicazione utente scritta in C++, mentre il lato server è costituito da una applicazione di servizio scritta nel linguaggio proprietario PDL2.

Il lato client è costituito da una classe C++, denominata *RobotComau*, caratterizzata da numerosi metodi pubblici che consentono di controllare altrettante tipologie di moto del robot. Tali metodi inviano i comandi da svolgere al manipolatore sulla socket e ricevono informazioni di controllo da parte del Server.

Il lato server è costituito da un programma PDL2 che riceve i comandi da un

client, li traduce nelle corrispondenti istruzioni del linguaggio proprietario e li fa eseguire al robot. Infine, invia al Client, sempre mediante la socket, la conferma di avvenuta esecuzione dell'istruzione (*acknowledgment*).

L'ultima funzionalità offerta dalla libreria riguarda la possibilità di muovere il gripper robotico montato sul manipolatore, lo *Schunk Pg70* della serie PowerCube, una pinza servo-elettrica modulare e dotata di sensori e microcontrollori interni. I comandi a tale pinza vengono costruiti attraverso l'utilizzo della libreria SchunkSerialProtocol e inviati tramite la porta seriale dell'elaboratore alla pinza.

La fig. 1.1 mostra l'architettura del sistema, specificando la distinzione tra il lato client e il lato server.

Nelle pagine seguenti, dopo una panoramica dei dispositivi hardware e degli strumenti software utilizzati, verrà presentata la libreria software realizzata, analizzandone i dettagli implementativi dei vari componenti, per poi presentare alcune prove sperimentali svolte utilizzando la stessa libreria. Infine, verranno suggerite alcune possibili estensioni future della libreria.

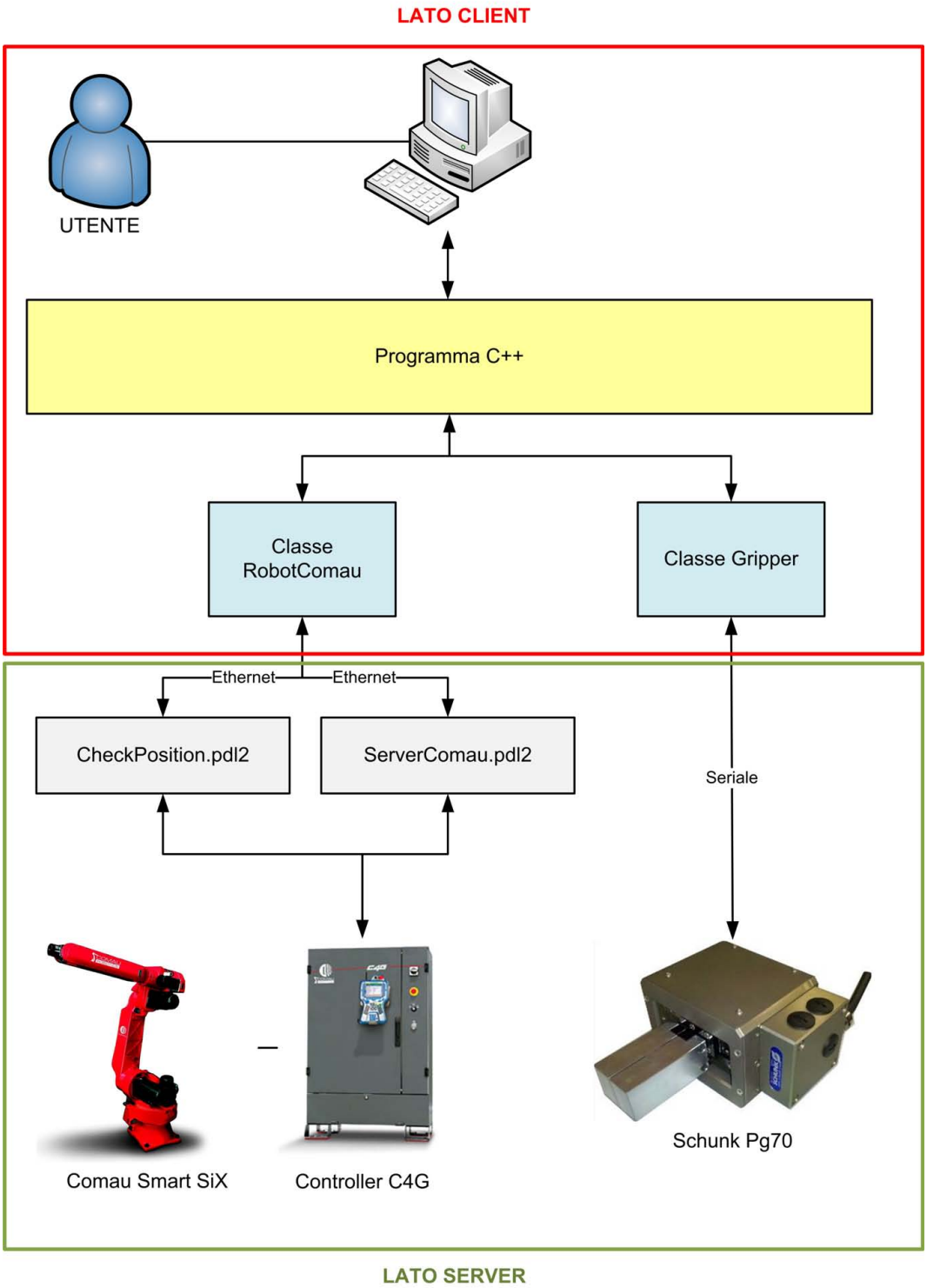

Figura 1.1: Comunicazione tra i moduli software e l'hardware

Capitolo 2

Descrizione dei dispositivi

I dispositivi utilizzati in questa tesi sono:

- Manipolatore Comau Smart SiX
- Gripper robotico Schunk Pg70

In questo capitolo verranno presentate le principali caratteristiche dei due dispositivi.

2.1 Il manipolatore Comau Smart SiX

Il robot *Comau Smart SiX* è un manipolatore a 6 giunti rotoidali destinato a scopi tipicamente industriali. I sei giunti, tutti rotoidali, possono essere divisi in due gruppi: i primi tre rappresentano la spalla, i secondi tre il polso. Ogni giunto può essere ruotato di 360°, ad eccezione dell'ultimo che può compiere fino a 7 giri in entrambi i sensi.

Alla flangia, ovvero la parte terminale del braccio, può essere attaccato un qualunque attrezzo per i vari tipi di lavorazione. Il manipolatore oggetto di questa tesi è stato dotato di una pinza Schunk Pg70, le cui caratteristiche verranno descritte in seguito.

Il robot ha una capacità di carico di 6kg, ovvero può trasportare oggetti di un peso massimo complessivo di 6kg. In questo carico va incluso il peso della pinza.

Figura 2.1: Il controllore Comau C4G

Il manipolatore viene gestito dal controllore C4G (fig. 2.1), che contiene l'hardware e il software per poter far muovere il robot. Il controllore è dotato di una manopola per accendere e spegnere il robot e di una chiave per selezionare la modalità di lavoro del robot stesso. Per gli scopi di questa tesi verrà utilizzata sempre la modalità AUTO.

Al controllore è collegato il terminale di programmazione WiTP (fig. 2.2), un grosso telecomando che permette di controllare il robot a distanza. Dal WiTP è possibile avviare programmi, controllare lo stato del robot, gestire le segnalazioni di allarmi, e molto altro.

Il controllore può essere collegato al computer per mezzo di un cavo di rete LAN RJ-45 e gestito per mezzo del programma WinC4G che verrà descritto in seguito.

Figura 2.2: Il terminale di programmazione WiTP

2.1.1 Sistema di riferimento

Le due terne principali del robot sono la terna di base (b) e la terna dell'utensile (t). Solitamente, tutti i moti vengono riferiti alla terna di base. La posizione delle terne è indicata nella fig. 2.3, tratta dal manuale Comau.

Per come è posizionato il robot in laboratorio (fig. 2.4), però, conviene vederlo da un'altra angolazione, riportata in fig. 2.5, più realistica e comprensibile.

Come vediamo, Xb è rivolta verso l'utente, Yb verso il tavolo e Zb, di conseguenza, verso il soffitto. Ciò significa che se volessimo spostare il braccio verso destra (nella fig. 2.5) dovremmo aumentare il valore di Yb, mentre per muoverlo verso l'alto, dovremmo aumentare quello di Zb.

I valori limite lungo i tre assi sono, all'incirca e tenendo conto della presenza del tavolo, i seguenti (espressi in millimetri):
$X \in [-700, 700]$
$Y \in [95, 1250]$
$Z \in [550, 1550]$

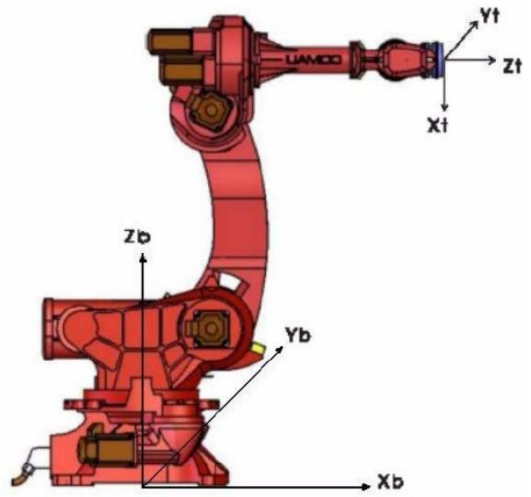

Figura 2.3: Orientazione della terna di base e della terna utensile

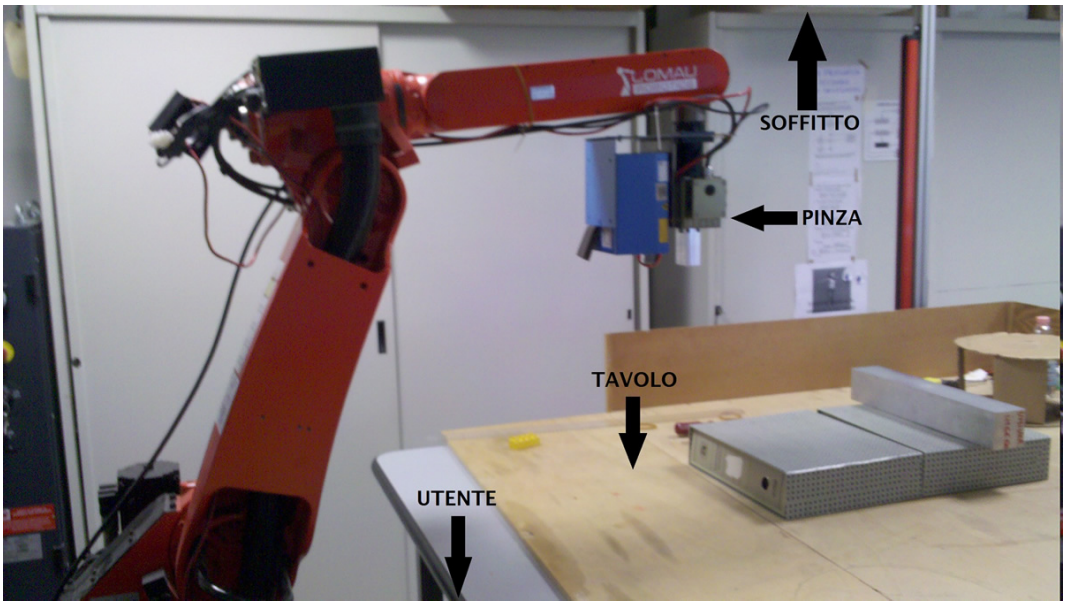

Figura 2.4: Posizionamento del robot nel laboratorio di robotica dell'Università di Parma

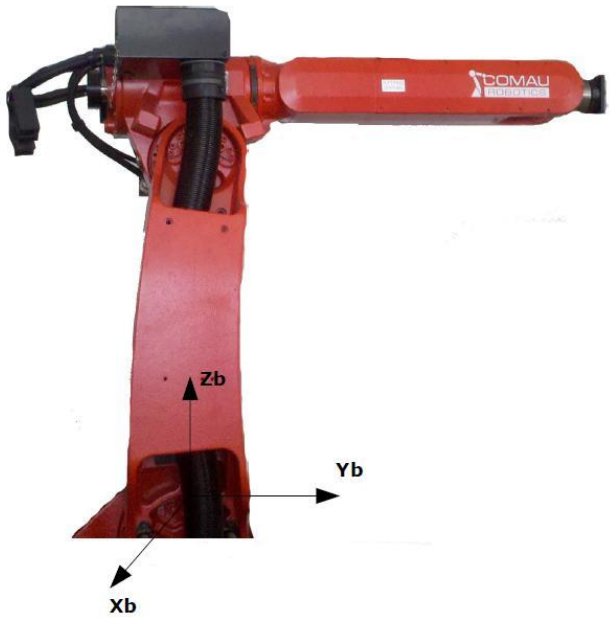

Figura 2.5: Orientazione della terna utensile

La posizione assunta in fig. 2.5 dal robot è la seguente:
$X = 0$
$Y = 800$
$Z = 1200$
$A = 90$
$B = 90$
$C = 90$

2.1.2 Esecuzione di programmi

In questo capitolo ci si riferirà al robot *Comau Smart SiX* installato nel laboratorio di robotica della Palazzina 1 della sede scientifica di Ingegneria dell'Università degli Studi di Parma.

2.1.2.1 Accensione del robot

Per accendere il robot, occorre eseguire i seguenti passi:

- accendere l'interruttore posto a fianco della porta di ingresso del laboratorio;

- accendere il controllore C4G girando la manopola posta sopra al pulsante rosso di emergenza in senso orario.

A questo punto, dopo alcuni secondi di attesa, il robot sarà pronto all'uso.

2.1.2.2 Collegamento del robot al PC

Per collegare il computer al robot, occorre eseguire i seguenti passi:

- collegare il cavo di rete grigio alla scheda di rete del computer;
- impostare l'indirizzo IP dinamico (DHCP) nella connessione alla rete locale LAN del proprio computer;
- installare il programma WinC4G nel proprio computer che consente la comunicazione con il robot;
- avviare il programma WinC4G e cliccare su OK e sulla finestra che appare;
- nella finestra Nuova connessione e proprietà, nella casella Indirizzo, scrivere l'indirizzo IP del robot: 160.78.28.98; quindi, fare clic su Applica; la schermata che si presenta è simile a quella di fig. 2.6;
- cliccare sul pulsante con le due frecce gialle (indicato, nella fig. 2.6, dalla freccia rossa 1), a fianco del pulsante Salva, per connettersi al robot;
- inserire nome utente e password negli appositi campi, cliccando poi su Connetti; dopo qualche istante, se non ci sono errori di comunicazione, dovrebbe apparire un messaggio che avvisa che la versione del software è differente: è possibile ignorare il messaggio cliccando su Ok;
- cliccare sul pulsante indicato, in fig. 2.6, dalla freccia blu (2) per aprire il terminale del robot, ovvero l'interfaccia che permette di inviare i comandi allo stesso, come quello per mettere in esecuzione un programma PDL2; il terminale che apparirà sarà simile a quello mostrato nella fig. 2.7.

Da questo terminale è possibile inviare comandi al robot. Le voci del menu sono selezionabili con le frecce direzionali e invio per confermare oppure mediante

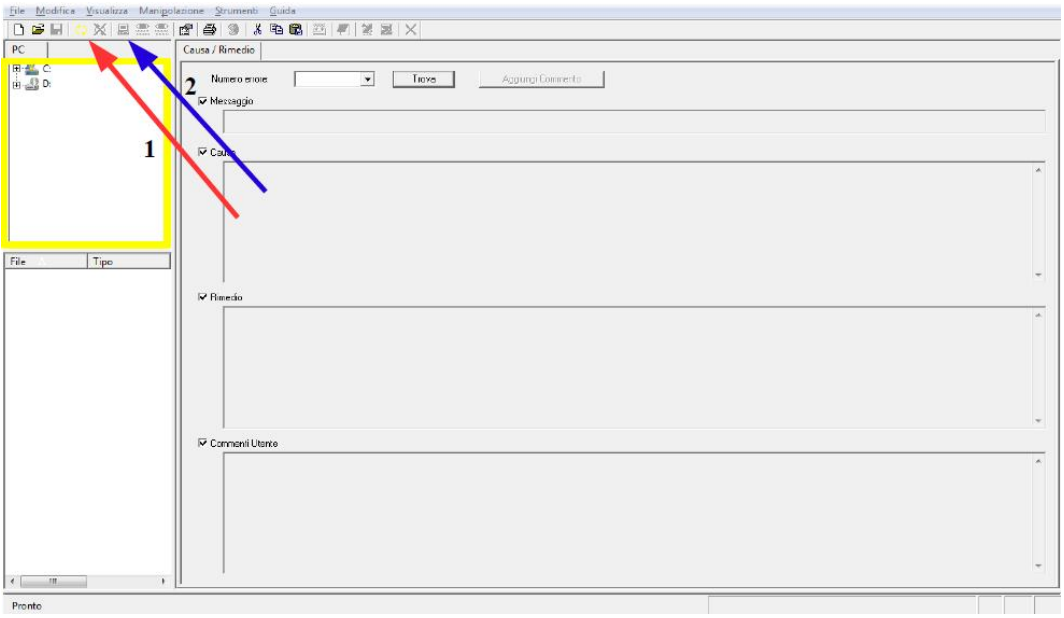

Figura 2.6: Interfaccia del programma WinC4G

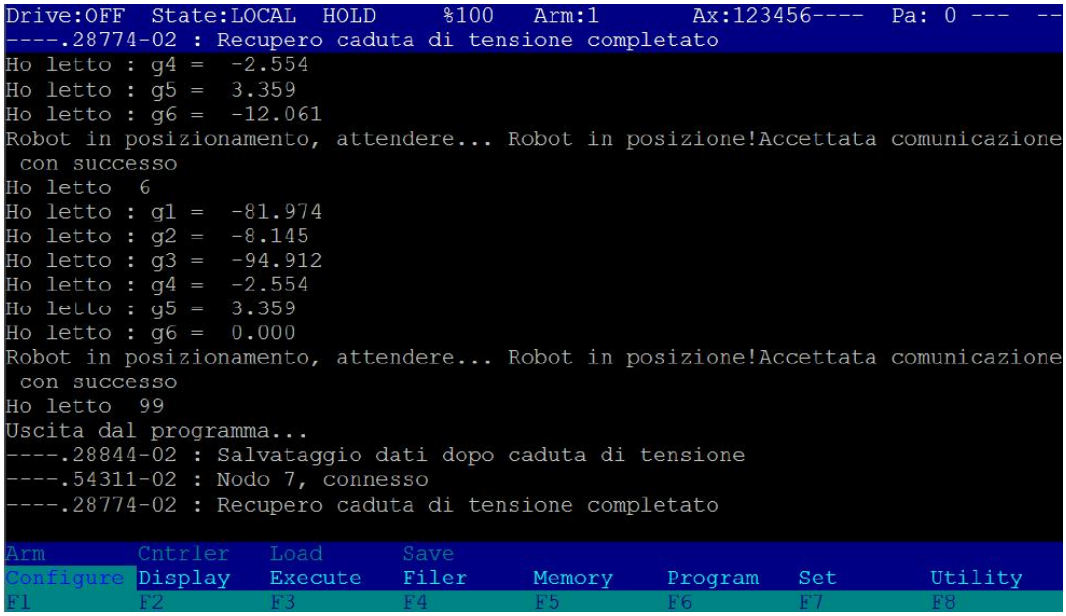

Figura 2.7: Teminale del robot

le scorciatoie da tastiera (tasti funzione) indicate sotto ogni voce oppure premendo la lettera maiuscola corrispondente alla funzione.

Di seguito troviamo spiegate le funzioni del terminale più utilizzate.

2.1.2.3 Esecuzione di un programma

Per mettere in esecuzione un programma occorre, prima di tutto, caricarlo nella memoria di massa del robot, seguendo i seguenti passi:

- nella finestra in alto a sinistra di WinC4G, selezionare nella scheda PC il programma da caricare, cliccare con il tasto destro su di esso e scegliere Copia;
- cliccare sulla scheda Robot, selezionare l'unità UD con il tasto destro del mouse e scegliere Incolla.

Per mettere in esecuzione un programma caricato in memoria, occorre seguire i seguenti passi:

- aprire il terminale del robot nel programma WinC4G;
- scegliere la voce Program (tasto F6 o P);
- scegliere la voce Go (tasto F4 o G);
- inserire il nome del programma da eseguire o scrivendone il nome oppure premendo il tasto F12 e selezionandolo, mediante le frecce direzionali, dalla lista dei programmi caricati in memoria che appare.

Ora che il programma è caricato in memoria, è possibile eseguirlo seguendo i seguenti passi:

- porre il robot nello stato di Drive ON, prendendo il terminale WiTP e premendo il pulsante blu R5, posto sulla destra dello schermo;
- sempre sul terminale WiTP, premere il pulsante verde START per avviare il programma;
- al termine dell'esecuzione del programma, premere nuovamente il pulsante R5 per rimettere il robot nello stato di Drive OFF.

2.1.2.4 Disattivazione di un programma

Per bloccare l'esecuzione di un programma ci sono tre metodi. Il primo metodo è quello più utilizzato:

- dall'interfaccia di fig. 2.7 scegliere la voce Program (tasto F6 o P) e poi Deactiv (tasto F2 o D);
- inserire il nome del programma da disattivare, oppure selezionarlo dall'elenco che appare premendo F12;
- premere il pulsante R5 sul terminale WiTP per porre il robot nello stato di Drive OFF.

Il secondo metodo consiste nel saltare i primi due passi del metodo precedente, premendo solamente il pulsante R5 sul terminale WiTP per porre il robot nello stato di Drive OFF. Questo potrebbe generare un errore del sistema in quanto il programma, ancora in esecuzione, si troverebbe impossibilitato a far svolgere operazioni al robot. Pertanto, questo metodo viene raramente utilizzato.

Il terzo e ultimo metodo è utile nei casi di emergenza, quando cioè si vuole interrompere immediatamente il programma per questioni di sicurezza e consiste nel premere il grosso pulsante rosso posto all'estremità superiore del terminale WiTP, il quale provoca un arresto immediato del robot e un errore. Per sbloccarlo, occorre poi ruotare il pulsante stesso in senso orario.

Con gli ultimi due metodi, se si volesse poi eseguire un nuovo programma, occorrerebbe prima disattivare il precedente mediante la voce Deactiv del menu Program del terminale.

2.1.2.5 Uscita da un errore

Nel caso in cui, per un qualche motivo (come la pressione del pulsante rosso di emergenza sul WiTP), dovesse presentarsi un errore, individuato da ALARM sullo schermo del WiTP e dal led rosso lampeggiante, il controllore non permetterebbe di eseguire nuove operazioni sul robot perchè, appunto, lo stesso si troverebbe in stato ALARM. Per tornare allo stato di HOLD (normale), è sufficiente premere il pulsante bianco RESET sul WiTP.

Per visualizzare gli errori occorre premere il pulsante L2 sul terminale WiTP e poi vedere i dettagli di un singolo errore selezionandolo con le frecce blu e premendo il testo Enter.

2.1.2.6 Riavvio del controllore

Per riavviare il controllore, dalla schermata principale del terminale di fig. 2.7 (se non appaiono le voci del menu indicate in figura, premere il tasto Esc finchè non vengono visualizzate) è sufficiente premere la scorciatoia da tastiera CCRC, che corrisponde alle opzioni *Configure - Cntrler - Restart - Cold*, e premere Invio per confermare. La memoria principale verrà azzerata (ovviamente non quella di massa) e verrà eseguita una ripartenza *a freddo*.

2.1.2.7 Spegnimento del controllore

Una volta terminato l'utilizzo del robot, occorre spegnerlo. Per farlo, occorre eseguire i seguenti passi:

- dalla schermata principale del terminale di fig. 2.7 (se non appaiono le voci del menu indicate in figura, premere il tasto Esc finchè non vengono visualizzate) digitiamo la scorciatoia da tastiera CCRS, che corrisponde alle opzioni *Configure - Cntrler - Restart - Shutdown*, e premiamo Invio per confermare;
- attendere circa 30 secondi per dare tempo al controllore di spegnersi;
- ruotare la manopola nera posta sul controllore in senso antiorario;
- abbassare l'interruttore nel quadro elettrico di fianco alla porta di ingresso del laboratorio.

2.1.2.8 Esecuzione di istruzioni PDL2

Un'opzione utile in fase di test è l'esecuzione di singole istruzioni PDL2. Per eseguire un'istruzione, seguire i seguenti passi:

- porre il robot nello stato di Drive ON premendo il pulsante blu R5 sul terminale WiTP;

- dalla schermata principale del terminale di fig. 2.7 scegliere Execute;
- scrivere l'istruzione da eseguire premendo Invio per confermare.

2.2 Il gripper robotico Schunk Pg70

Un gripper robotico (in gergo *pinza*) è lo strumento fondamentale di ogni robot in quanto fornisce al manipolatore la capacità di afferrare oggetti, proprio come se il *braccio* del robot fosse dotato di una *mano*.
Il gripper utilizzato e a cui si fa riferimento in questa tesi è lo Schunk Pg70, una pinza servoelettrica a due giffe parallele caratterizzata da un'apertura massima di 70 mm, da un peso di 1.4 Kg, da una portata massima di 1.0 Kg e da una forza sviluppata tra i 30 e i 200 N.
Lo Schunk Pg70 è in grado di effettuare movimenti (in generale, di apertura o chiusura) pilotati attraverso 5 diverse grandezze: posizione, velocità, accelerazione, jerk e tempo, ovvero è possibile programmare la pinza affinché esegua diversi tipi di moto, caratterizzati da particolari valori delle grandezze precedenti. È inoltre possibile impostare forza e velocità di presa attraverso il controllo della corrente fornita. I valori limite per le cinque grandezze fondamentali per il controllo del moto sono i seguenti:

- Posizione: compresa tra 0 e 68 $[mm]$.
- Velocità: compresa tra 0 e 82 $[mm/s]$.
- Accelerazione: compresa tra 0 e 328 $[mm/s^2]$.
- Jerk: compreso tra 0 e 10000 $[mm/s^3]$
- Corrente: compreso tra 0 e 6.5 $[A]$

2.2.1 Collegamento al PC

A differenza del *Comau Smart SiX*, la pinza non viene gestita dal controllore C4G ma attraverso un proprio controllore interno che comunica con il computer attraverso una normale porta seriale RS232. Pertanto, per collegarla al computer, si utiliz-

Figura 2.8: Il gripper Schunk Pg70

zerà questa interfaccia, assieme ad un adattatore RS232-USB per permettere anche a computer portatili, spesso senza porta seriale, di controllare il gripper. Oltre al collegamento dati, è poi necessario il collegamento della pinza alla rete elettrica al fine di fornirle l'alimentazione necessaria al suo funzionamento.
Una volta collegato l'adattatore RS232-USB al computer, il sistema operativo installerà una nuova porta seriale identificata da un nome del tipo *COM<numero progressivo>*. Il numero progressivo assegnato deve essere minore di 8, altrimenti la pinza non funzionerà. Per scoprire l'identificativo dato dal sistema alla porta seriale, occorre cliccare con il tasto destro del mouse su Risorse del computer, scegliere Gestione, cliccare su Gestione periferiche e cercare, nell'elenco, la dicitura "Porta di comunicazione (COM<numero>)".

2.2.2 Comandi

Il controllore della pinza riconosce comandi ben specifici, organizzati in un generico pacchetto della comunicazione seriale. La struttura di tale pacchetto è molto semplice (fig. 2.9):

- 8 bit per la tipologia di comunicazione
- 8 bit per la selezione dell'utensile utilizzato
- 8 bit per la lunghezza del pacchetto

Figura 2.9: La generica struttura di un pacchetto di comunicazione seriale

- 8 bit per il comando da inviare
- un numero variabile di bit per il corpo del messaggio
- 16 bit per il controllo di validità (CRC)

Nel caso della pinza Schunk Pg70, il corpo del messaggio è costituito dalle opzioni relative al comando inviato, come possono essere i valori di posizione, velocità, accelerazione, jerk e corrente del moto da realizzare.

Per la costruzione di tali pacchetti da inviare al controllore della pinza, si utilizza una versione modificata per Windows della libreria SchunkSerialProtocol, sviluppata da Marco Tarasconi [1] per l'ambiente Orchestra di Linux, che permette, tra le altre cose, di preparare i dati che si vogliono inviare alla pinza secondo la struttura del pacchetto di fig. 2.9. Per i dettagli implementativi di tale libreria, si consulti la tesi di laurea [1].

Capitolo 3

Strumenti software

3.1 PDL2

Il PDL2 (Program Definition Language - versione 2) è un linguaggio Pascal-like, creato da Comau, particolarmente indicato per la programmazione di robot quali, appunto, il controllore C4G. I programmi PDL2 si dividono in due grandi categorie:

- **programmi holdable**, caratterizzati dall'attributo HOLD, sono controllati da START e HOLD e possono contenere istruzioni di moto;
- **programmi non holdable**, caratterizzati dall'attributo NOHOLD, sono usati per il controllo di processo e non possono contenere istruzioni di moto.

La libreria software sviluppata presentata più avanti utilizza sia un programma holdable, per l'invio delle istruzioni di moto, sia un programma non holdable, per il controllo della posizione del robot durante il moto. Il PDL2 è un linguaggio case insensitive, pertanto non differenzia le lettere maiuscole da quelle minuscole (Move = move = MOVE). I commenti iniziano con due trattini $--$ e terminano con la fine della riga. Non sono possibili, pertanto, commenti su più righe, salvo iniziare ogni riga con $--$. In questa tesi, che funge anche da manuale sintetico dell'uso del linguaggio PDL2, nei frammenti di codice riportati le parole riservate vengono scritte in maiuscolo e le componenti facoltative vengono racchiuse tra parentesi angolari <>.

3.1.1 Struttura del programma

La struttura di un programma PDL2 è la seguente:

```
PROGRAM nome <attributi>
<codice da includere>
<dichiarazione di costanti, variabili, tipi>
<dichiarazioni di routine>
BEGIN <CYCLE>
<istruzioni da eseguire>
END nome
```

Il file deve essere salvato con lo stesso nome del programma, ovvero nome.pdl2. Il programma è diviso in due sezioni, separate dalla clausola BEGIN:

- **sezione dichiarativa**, in cui vengono dichiarati tipi, variabili e routine definite dall'utente e variabili, tipi e routine importate da altri programmi;
- **sezione esecutiva**, contenente le istruzioni che il controllore deve eseguire per svolgere un compito.

L'opzione CYCLE permette di eseguire ciclicamente il programma. Ogni istruzione viene separata dalla successiva da un "a capo", a differenza del Pascal in cui viene separata da un ";".

3.1.2 Sezione dichiarativa

I **tipi** vengono dichiarati dopo la parola riservata TYPE. La struttura di dichiarazione dei record è la seguente:

```
TYPE nome_tipo = RECORD <GLOBAL>
nome_campo : tipo_campo
...
ENDRECORD
```

L'opzione GLOBAL indica la dichiarazione di un tipo globale, esportabile in altri programmi. La struttura di dichiarazione dei nodi (punti), tipo di variabile utile per la definizione di traiettorie (insiemi di nodi), è la seguente:

```
TYPE nome_tipo = NODEDEF <GLOBAL>
nome_campo_predefinito <NOTEACH>
...
nome_campo : tipo_campo <NOTEACH>
...
ENDNODEDEF
```

Ogni nodo può essere caratterizzato, oltre che da campi definiti dall'utente, da alcuni campi predefiniti, identificati da un nome maiuscolo preceduto dal simbolo '$', ad esempio $MAIN_POS. Un esempio di dichiarazione di un nodo è il seguente:

```
TYPE my_node = NODEDEF
$MAIN_POS
nome : STRING[20]
cognome : STRING[20]
ENDNODEDEF
```

L'opzione NOTEACH indica che il campo non può essere modificato durante la modifica di un cammino (PATH).

Le **costanti** vengono dichiarate dopo la parola riservata CONST. La struttura di dichiarazione delle costanti è la seguente:

```
CONST nome_costante = valore
```

Le **variabili** utilizzate all'interno del programma vengono dichiarate dopo la parola riservata VAR. La struttura di dichiarazione delle variabili è la seguente:

```
VAR nome_variabile : tipo_variabile <opzioni>
```

Su ogni riga è possibile dichiarare una o più variabili dello stesso tipo. La parola riservata VAR si inserisce solo prima della prima variabile dichiarata.

I tipi di variabile disponibili sono:

- **INTEGER**: rappresenta un numero intero compreso tra -2147483647 e +2147483647;
- **REAL**: rappresenta un numero decimale o un numero espresso in notazione scientifica; viene sempre visualizzato usando 8 cifre significative;

- **BOOLEAN**: rappresenta una variabile che può assumere solo due valori: TRUE (vero, 1) o FALSE (falso, 0);

- **STRING**: rappresenta una serie di caratteri ASCII (da 0 a 2048) racchiusi tra apici; all'atto della dichiarazione, occorre dichiarare la lunghezza massima della stringa tra parentesi quadre, ad esempio x: STRING[15] rappresenta una stringa lunga al massimo 15 caratteri;

- **ARRAY**: rappresenta una collezione di dati tutti dello stesso tipo, che non può essere ARRAY, NODE o PATH; può avere una o due dimensioni e, ogni dimensione, può contenere al massimo 65535 elementi; il numero di elementi massimo deve essere dichiarato: ad esempio, x: ARRAY [100,100] OF REAL rappresenta un array bidimensionale (matrice) di massimo 100 numeri reali sia sulle righe che sulle colonne; per accedere a un elemento, è sufficiente indicare riga e colonna: x[15,11]. Il primo elemento dell'array è raggiungibile dalla posizione 1, ovvero scrivendo x[1];

- **RECORD**: rappresenta una collezione di uno o più dati, anche di tipi diversi, eccetto SEMAPHORE, RECORD, NODE o PATH; la dimensione massima di un record è di 65535 bytes; per accedere ad un campo del record, occorre indicare il nome della variabile record e il nome del campo separati da un punto .; solitamente, il record viene dichiarato come nuovo tipo dopo la parola riservata TYPE;

- **VECTOR**: rappresenta un vettore avente direzione e verso, solitamente utilizzato per rappresentare un punto nello spazio cartesiano; è caratterizzato da tre componenti reali x, y, z; è un esempio di tipo RECORD predefinito;

- **POSITION**: rappresenta una posizione di una terna cartesiana rispetto a una di riferimento (di solito la terna di base) sia come collocazione (x, y, z), misurata in millimetri, che come orientazione (α, β, γ), misurata in gradi, che come configurazione del robot; è pertanto composta da sei componenti reali (x, y, z, a, e, r) e una stringa: x, y, z rappresentano la posizione rispetto alla terna di base, a, e, r rappresentano gli angoli di Eulero in cui la prima rotazione viene effettuata rispetto all'asse Z, la seconda rotazione rispetto all'asse

Y (della terna risultante) e la terza rotazione rispetto all'asse Z (della terna risultante); è un esempio di tipo RECORD predefinito;

- **JOINTPOS**: rappresenta la posizione dei sei giunti in gradi; è un array di sei elementi;
- **XTNDPOS**: rappresenta la posizione con un numero più elevato di parametri (poco utilizzato);
- **NODE**: è simile al RECORD ma può contenere gruppi di campi nodo predefiniti indicati con *$nome_campo*;
- **PATH**: rappresenta una sequenza di nodi che vengono attraversati in un'istruzione di moto; è un RECORD contenente, tra gli altri, il campo NODE (array di nodi) in cui vengono specificati i nodi che compongono il cammino; si dichiara specificando il tipo nodo collegato, ad esempio x : PATH OF *mynode*;
- **SEMAPHORE**: rappresenta un semaforo, utile per la mutua esclusione di una risorsa in caso di programmazione concorrente, manipolato con le istruzioni WAIT e SIGNAL.

Le opzioni possibili sono:

- **EXPORTED FROM nome_programma** per importare la variabile da un altro programma, in cui è dichiarata con la clausola GLOBAL;
- il valore di inizializzazione della variabile racchiuso tra parentesi, ad esempio x : INTEGER (0);
- **NOSAVE** per evitare che la variabile venga salvata in un file .var a parte;
- **CONST** per rendere l'operazione di scrittura della variabile un'operazione privilegiata, eseguibile solamente da utenti privilegiati.

Per importare tutte le variabili, i tipi e le routine dichiarate GLOBAL da un altro programma, occorre inserire, subito dopo l'intestazione del programma, la clausola **IMPORT** 'nome_programma'.

3.1.3 Operazioni

Le operazioni aritmetiche sono:

- $+$: somma
- $-$: differenza
- $*$: prodotto
- $/$: divisione
- DIV: quoziente
- MOD: resto
- $**$: elevamento a potenza
- $+=$: incremento intero
- $-=$: decremento intero

Le operazioni relazionali sono:

- $<$: minore
- $>$: maggiore
- $=$: uguale
- $<=$: minore o uguale
- $>=$: maggiore o uguale
- $<>$: diverso

Le operazioni logiche sono:

- AND
- OR
- XOR

- NOT

Le operazioni tra vettori (VECTOR) sono:

- $\#$: prodotto vettoriale
- $@$: prodotto scalare

L'operatore ":"applicato a due posizioni (POSITION) permette di esprimere la seconda posizione rispetto alla prima, mentre se applicato a una posizione e a un vettore esprime il vettore in quella posizione. È l'equivalente di una matrice di rotazione che esprime una terna rispetta a un'altra.

L'operatore di assegnazione è ":=".

3.1.4 Controllo di flusso

Le istruzioni per il controllo di flusso disponibili sono quelle standard della maggior parte dei linguaggi. In particolare:

- **IF**: permette di scegliere tra due possibili percorsi di azioni; la struttura è la seguente:

```
IF condizione THEN
<istruzioni>
<ELSE
<istruzioni>...>
ENDIF
```

- **SELECT**: permette di scegliere tra tanti possibili percorsi di azioni; la struttura è la seguente:

```
SELECT variabile OF
CASE (valore) : <istruzioni>
<CASE (valore): <istruzioni>
...>
<ELSE: <istruzioni>...>
ENDSELECT
```

- **FOR**: utile per ripetere una serie di istruzioni un numero preciso di volte; la struttura è la seguente:

```
FOR contatore:=valore_iniziale TO valore_finale DO
<istruzioni>...
ENDFOR
```

  Esiste anche la versione inversa, che ha la seguente struttura:

```
FOR contatore:=valore_finale DOWNTO valore_iniziale DO
<istruzioni>...
ENDFOR
```

- **WHILE**: esegue una serie di istruzioni finché è verificata una certa condizione; la struttura è la seguente:

```
WHILE condizione DO
<istruzioni>
ENDWHILE
```

- **REPEAT**: ripete una serie di istruzioni fino a quando una condizione non diventa vera (inverso del WHILE); la struttura è la seguente:

```
REPEAT
<istruzioni>
UNTIL condizione
```

- **GOTO**: trasferisce il controllo del programma a un punto preciso dello stesso individuato da un'etichetta, dichiarata come *nome_etichetta::*istruzioni...; la struttura è la seguente:

```
GOTO nome_etichetta
```

- **DELAY**: sospende il programma per un certo numero di millisecondi; la struttura è la seguente:

```
DELAY millisecondi
```

3.1.5 Istruzioni di moto

La struttura di un'istruzione di moto è la seguente:

```
MOVE <braccio> traiettoria destinazione <opzioni>
    <sincronizzazione>
```

La clausola *braccio* serve per indicare quale sarà il braccio interessato al moto. Infatti, un programma PDL2 può controllare più bracci. La clausola viene usata nel modo seguente:

```
MOVE ARM[1] TO destinazione
```

Se non viene indicata, il moto interesserà il braccio di default, memorizzato nella variabile di sistema $DFT_ARM, che può essere settato dal programmatore con un attributo al programma in questo modo:

```
PROGRAM arm PROG_ARM=1
```

La clausola *traiettoria* serve per indicare quale percorso il robot dovrà seguire per raggiungere un punto. Se non specificata, viene usata quella di default memorizzata nella variabile di sistema $MOVE_TYPE. Può assumere tre valori:

- **LINEAR**: utile per il moto nello spazio operativo, muove il polso del robot lungo una traiettoria lineare;
- **CIRCULAR**: utile per il moto nello spazio operativo, muove il polso del robot lungo una traiettoria circolare (prevede la definizione di un punto intermedio utilizzando la clausola VIA);
- **JOINT**: utile per il moto nello spazio dei giunti; è la modalità predefinita.

La clausola *destinazione* permette di indicare il tipo di moto ovvero la destinazione da raggiungere. I valori ammessi sono i seguenti:

- **TO** *dest*: muove il braccio verso la destinazione indicata da *dest*, che può essere espressa come POSITION (MOVE TO POS($x, y, z, a, e, r, config$) in cui x, y, z, a, e, r rappresentano la posizione e l'orientazione mentre $config$ è una stringa di configurazione, di solito impostata come stringa vuota ", utilizzata per precisare come il robot dovrà raggiungere la posizione finale), JOINTPOS (MOVE TO , β, , δ, , , che muove solo i giunti 2 e 4) o XTNDPOS;

- **NEAR** *dest* **BY** *distanza*: muove il braccio verso la destinazione indicata da *dest* fermandosi a *distanza* millimetri; *dest* può essere espressa come POSITION, JOINTPOS o XTNDPOS;
- **AWAY** *distanza*: si allontana di *distanza* millimetri dalla posizione corrente lungo la direzione di avvicinamento;
- **RELATIVE** VEC(x, y, z) <IN *terna*>: sposta il robot dalla posizione corrente di un vettore misurato in millimetri che indica lo spostamento lungo i tre assi; *terna* specifica rispetto a quale terna muoversi (TOOL, BASE o UFRAME);
- **ABOUT** VEC(x, y, z) BY *angolo* <IN *terna*>: sposta il robot dalla posizione corrente di un vettore (come la precedente) e ruota l'utensile di *angolo* gradi rispetto all'asse z;
- **BY** $\{\alpha, \beta, \gamma, \delta, \omega, \iota\}$: sposta il robot verso una destinazione indicata come elenco di espressioni reali che corrispondono al movimento incrementale dei giunti del braccio: α indica il movimento da far fare al giunto 1, β il movimento del giunto 2, e così via;
- **FOR** *distanza* TO *dest*: permette di effettuare un movimento parziale verso *dest* di *distanza* millimetri.

Le clausole opzionali sono tre:

- **ADVANCE**: permette all'interprete del programma di eseguire le istruzioni successive anche se l'istruzione corrente non è terminata; nel caso di istruzioni di moto successive identificate dalla clausola MOVEFLY (spiegata più avanti), permette di eseguire un moto unico dovuto all'interpolazione dei due moti; è obbligatoria se si utilizza l'istruzione MOVEFLY;
- **TIL**: permette di specificare una lista di condizioni che portano alla cancellazione del moto; ad esempio MOVE TO *destinazione* TIL TIME 100 AFTER START blocca il moto dopo 100 millisecondi;
- **WITH**: permette di settare variabili di moto predefinite o abilitare gestori di condizioni per la durata del moto, separate da virgola.

L'istruzione **MOVEFLY** permette di effettuare moti continui con movimenti dello stesso tipo, interpolando i movimenti; in questo modo, se dopo un'istruzione di moto continuo (MOVEFLY LINEAR TO) ci fosse un'istruzione classica di moto (ad esempio, MOVE LINEAR TO), il moto risultante sarebbe la somma dei due contributi, in quanto la seconda istruzione di moto verrebbe eseguita quando la prima è ancora in esecuzione.

Oltre alle istruzioni di moto "semplice", il PDL2 offre anche la possibilità di muovere il robot lungo un cammino prestabilito, mediante l'istruzione **MOVE ALONG** che ha la seguente struttura:

```
MOVE <braccio> ALONG cammino <[range di nodi]> <opzioni>
```

dove *cammino* è una variabile di tipo PATH. L'istruzione non fa altro che muovere il robot verso tutti (o quelli appartenenti al range specificato) i nodi del cammino, con le opzioni eventualmente indicate. Il range di nodi da attraversare si può specificare:

- indicando nodo iniziale e nodo finale: MOVE ALONG *path*[2..5], comporta il moto dal nodo 2 al nodo 5; è possibile anche un movimento all'indietro, se il nodo finale è minore di quello iniziale;
- indicando solo il nodo iniziale: MOVE ALONG *path*[2..], comporta il moto dal nodo 2 fino all'ultimo nodo del cammino.

Le opzioni disponibili sono ADVANCE e WITH che hanno lo stesso significato di quelle descritte in precedenza. Nel caso si volesse un moto continuo, si può utilizzare la **MOVEFLY ALONG**, che, come la MOVEFLY TO, permette di non fermare il braccio dopo l'ultimo nodo processato ma di continuare con le istruzioni successive.

3.1.6 Le routine: procedure e funzioni

Una **procedura** è una sequenza di istruzioni che viene invocata ed eseguita come fosse una singola istruzione. La struttura è la seguente:

```
ROUTINE nome <(parametro : tipo_parametro, ...)>
<costanti, variabili, tipi>
BEGIN
```

```
<istruzioni>
<RETURN <valore_di_ritorno>>
END nome
```

La clausola RETURN è utile per restituire il controllo al programma chiamante prima della fine "naturale"della procedura. Per invocarla, è sufficiente scrivere il nome della procedura e, tra parentesi, gli eventuali parametri da passare.

Una **funzione** è una sequenza di operazioni che danno un risultato che viene restituito dalla stessa. Pertanto, la funzione, oltre a un nome, deve avere un tipo, che indica il tipo di dato che verrà restituito dalla funzione stessa. Viene invocata all'interno di un'espressione. La struttura è la seguente:

```
ROUTINE nome <(parametro : tipo_parametro, ...)>:
    tipo_funzione
<costanti, variabili>
BEGIN
<istruzioni>
RETURN (valore_di_ritorno di tipo tipo_funzione)
END nome
```

Per invocarla, è sufficiente scrivere all'interno di un'espressione il nome della funzione e, tra parentesi, gli eventuali parametri da passare. Nel **passaggio dei parametri** occorre fare attenzione al tipo di parametro passato:

- se è STRING, non è possibile passare una dimensione massima;
- se è JOINTPOS o XTNDPOS, non è possibile passare il numero del braccio da muovere (verrà usato quello di default);
- se è ARRAY, non è possibile specificare una dimensione; per farlo, occorre creare un tipo apposito e indicare il parametro di quel tipo; ad esempio, creare il tipo VET = ARRAY [100] OF REAL e poi dichiarare la routine come ROUTINE nome (x: VET) invece che ROUTINE nome (x: ARRAY[100] OF REAL) che non funzionerebbe.

Inoltre, i parametri possono essere passati in due modalità :

- **per riferimento**, ovvero fornendo al parametro accesso diretto al contenuto dell'argomento: nome_routine (parametro)
- **per valore**, ovvero passando alla routine una copia del valore dell'argomento, che verrà poi distrutta al termine della routine: nome_routine ((parametro))

Da notare che il tipo di passaggio dei parametri viene specificato all'atto della chiamata alla routine e non all'atto della dichiarazione, come invece avviene nella maggior parte dei linguaggi di programmazione.

Funzioni e procedure possono anche essere importate da un altro programma, mediante la clausola **EXPORTED FROM**. In questo caso, ci si limita a dichiarare l'intestazione della routine in questo modo:

```
ROUTINE nome EXPORTED FROM nome_programma
```

3.1.7 Scrittura sul terminale

Può essere utile, per fini di debug o di visualizzazione di risultati, utilizzare il terminale del robot per visualizzare messaggi o valori di variabili. Per scrivere sul terminale, occorre utilizzare l'istruzione WRITE del PDL2, molto simile alle istruzioni omonime degli altri linguaggi di programmazione. Dopo l'istruzione WRITE occorre indicare dove si vuole scrivere: per scrivere sul terminale, si utilizza la clausola predefinita LUN_CRT. Le stringhe di testo da visualizzare vanno comprese tra apici. Nel caso si voglia concatenare testo da scrivere o variabili da visualizzare, occorre separare i dati da virgola. Per stampare un "a capo", si utilizza la parola riservata NL (New Line). Ad esempio, per visualizzare sul terminale il contenuto della variabile x preceduto da un testo introduttivo e seguito da un ritorno a capo, occorre scrivere:

```
WRITE LUN_CRT ('La variabile x vale ', x, NL)
```

3.1.8 Esempio esplicativo

Di seguito viene riportato un esempio di un programma PDL2 che permette di far muovere il robot lungo una traiettoria semicircolare, per simulare un braccio

che saluta l'utente. Le istruzioni sono precedute da un commento esplicativo per semplificare la comprensione del programma.

```
PROGRAM helloworld PROG_ARM = 1
BEGIN
-- Muove il braccio del robot in una posizione iniziale
MOVE LINEAR TO POS(0, 500, 1000, 90, 180, 90)
-- Permette di eseguire ciclicamente il moto
CYCLE
-- Stampa un saluto sul terminale del robot
  WRITE LUN_CRT ('Ciao!')
-- Muove il braccio del robot con traiettoria circolare
  MOVE CIRCULAR TO POS(0, 800, 1000, 90, 180, 90)
       VIA POS(0, 650, 1200, 90, 180, 90)
-- Muove il braccio del robot alla posizione iniziale
  MOVE CIRCULAR TO POS(0, 500, 1000, 90, 180, 90)
       VIA POS(0, 650, 1200, 90, 180, 90)
END helloworld
```

Capitolo 4

Progettazione e realizzazione del sistema

La comunicazione tra il computer e i diversi dispositivi utilizzati, ovvero il robot *Comau Smart SiX* e la pinza *Schunk Pg70*, avviene in due modi piuttosto diversi per i due strumenti. Per quanto riguarda il robot, viene utilizzata un'architettura di tipo *client/server*, ovvero la libreria software C++ invia richieste ad un programma server scritto in linguaggio PDL2 che poi fa eseguire le operazioni richieste al robot. Il meccanismo di comunicazione utilizzato tra PC e pinza, invece, consiste semplicemente nell'invio di comandi alla pinza attraverso la porta seriale, il cui controllore si preoccupa poi di trasformare i comandi in operazioni da svolgere.

Di seguito viene illustrata l'implementazione dei seguenti moduli software:

- Server PDL2 per la comunicazione con il robot
- Programma PDL2 per l'invio delle coordinate del robot
- Programma PDL2 per i controlli di sicurezza
- Classe C++ per la comunicazione con il robot
- Classe C++ per la comunicazione con il gripper

4.1 Il programma PDL2 ServerComau

Il programma ServerComau ha come obiettivo principale quello di ricevere le richieste da parte del client, tradurle in linguaggio PDL2, far eseguire i comandi al robot e, infine, inviare i dati richiesti al client. Come si è detto, il server e il client comunicano attraverso una socket. Il programma segue l'algoritmo mostrato in fig. 4.1. Di seguito ne vengono presentate in dettaglio le caratteristiche principali.

4.1.1 Apertura della socket di comunicazione

Come prima cosa, il server apre una socket che permette di comunicare e scambiare dati (in ingresso e in uscita) con il client che invia le richieste e si mette in attesa di una connessione da parte di un client. Per fare questo, viene utilizzato il seguente codice:

```
vi_netlun := 22
OPEN FILE vi_netlun ('NETT:', 'rw')
vi_local_port := default_port
DV4_CNTRL(ki_dv_tcp_connessione, (vi_netlun),
      (vi_local_port))
```

La prima istruzione assegna un valore alla variabile intera *vi_netlun* che rappresenta il descrittore della socket.
L'istruzione OPEN FILE assegna al descrittore una nuova socket di tipo TCP/IP, a causa del parametro *NETT:*, bidirezionale, grazie al parametro *rw*read/write).
Il terzo parametro assegna alla variabile intera *vi_local_port* il valore di una costante, *default_port*, definita all'inizio del programma, che definisce la porta sulla quale il server rimarrà in attesa di connessioni. Questa modalità permette di modificare rapidamente il numero della porta su cui rimanere in ascolto, semplicemente modificando il valore di una costante.
L'ultima istruzione è quella che veramente crea la comunicazione, sfruttando la primitiva PDL2 *DV4_CNTRL* che ammette tre parametri: la funzione richiesta alla primitiva, identificata dalla costante *ki_dv_tcp_connessione* e pari ad un codice intero che identifica la funzione di apertura della socket TCP/IP, il descrittore della

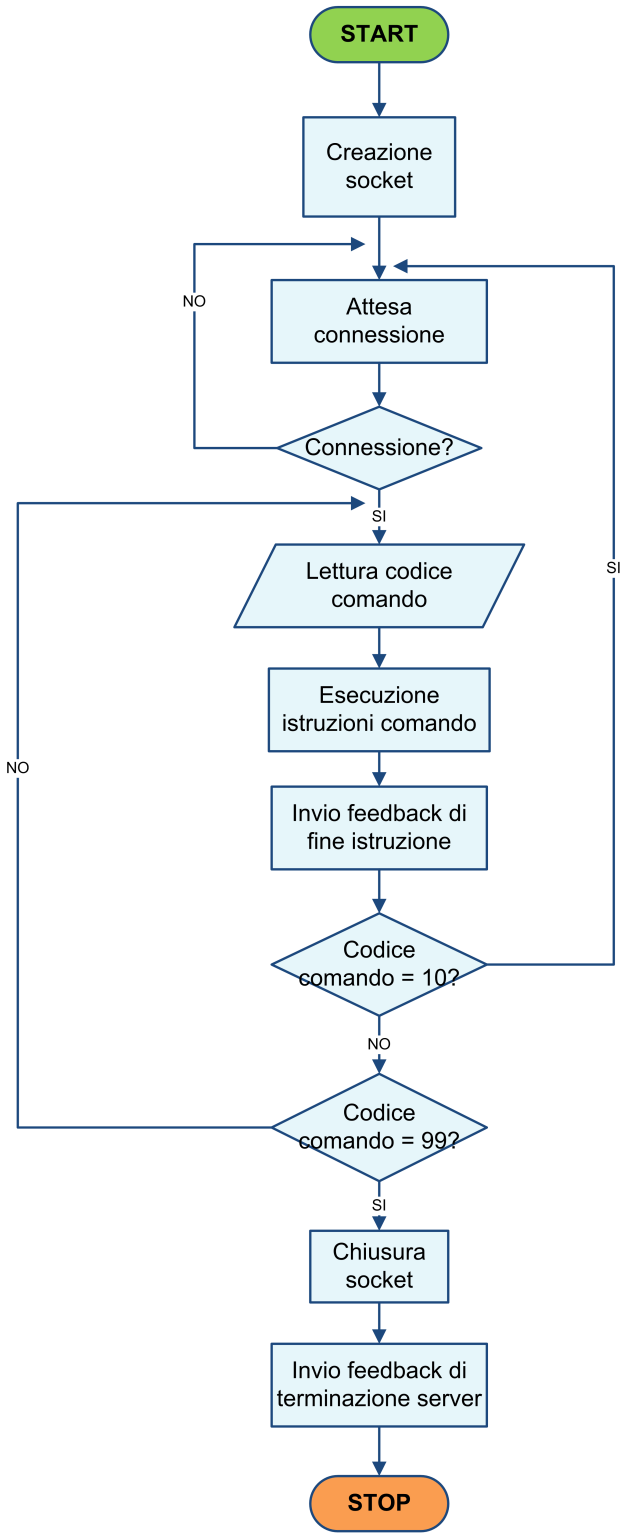

Figura 4.1: Algoritmo di funzionamento di ServerComau.pdl2

socket passato per indirizzo e il numero della porta su cui accettare connessioni, anch'esso passato per indirizzo. A questo punto, il server si mette in attesa dell'arrivo di una nuova connessione da parte di un client.

4.1.2 Accettazione della connessione

Il server può servire un solo client alla volta: trattandosi di un server che esegue istruzioni di moto, non avrebbe senso assegnare la *risorsa robot* a due client diversi, come avviene, per esempio, per le stampanti.
Non appena il client si connette al server, viene visualizzato un messaggio sul server mediante l'istruzione

```
WRITE LUN_CRT ('Accettata comunicazione con successo', NL)
```

che visualizza sul terminale del robot il messaggio di comunicazione accettata.
A questo punto, il server si aspetta di ricevere dal client la funzione richiesta sulla socket e, non appena la riceve, invia un messaggio di conferma al client, sempre attraverso la socket, mediante il seguente codice:

```
READ vi_netlun (risp)
WRITE vi_netlun ('OK')
```

Il codice identificativo della funzione richiesta viene inserito nella variabile intera *risp*. La tabella 4.1 mostra l'elenco dei codici riconosciuti con la funzione associata.

4.1.3 Moto nella posizione di calibrazione

La prima funzione offerta dal server, identificata dal valore 1 della variabile *risp*, permette di muovere il robot nella posizione di calibrazione, corrispondente alla posizione nello spazio dei giunti $(0, 0, -90, 0, 90, 0)$. Per farlo, viene utilizzata questa porzione di codice:

```
WRITE LUN_CRT ('Robot in posizionamento, attendere... ')
MOVE TO $CAL_SYS
WRITE vi_netlun ('OK')
WRITE LUN_CRT ('Robot in posizione')
```

Codice	Funzione
1	Moto verso la posizione di calibrazione
2	Moto lineare verso un punto
3	Moto lineare lungo una traiettoria
4	Moto lineare relativo alla posizione corrente
5	Moto lineare lungo l'asse del braccio
6	Moto nello spazio dei giunti
7	Modifica della velocità del robot
8	Posizione attuale del robot nello spazio operativo
9	Posizione attuale del robot nello spazio dei giunti
10	Chiusura della connessione
99	Terminazione del server

Tabella 4.1: Possibili valori della variabile *risp* e funzione associata

dove, oltre ai messaggi *di servizio* e al messaggio di conferma inviato al termine del moto, l'istruzione che effettivamente provoca il moto del robot è la seconda, *MOVE TO $CAL_SYS*, composta dalla parola riservata *MOVE TO* e dalla variabile di sistema $CAL_SYS che contiene la posizione predefinita di calibrazione.

4.1.4 Moto lineare verso un punto

Questa funzione, che corrisponde al valore 2 della variabile *risp*, è la più utilizzata nelle applicazioni e consente di far muovere il robot in linea retta dalla posizione corrente ad un punto identificato da 6 coordinate di tipo *double* ($x1, y1, z1, x2, y2, z2$) inviate dal client al server attraverso la socket. L'istruzione fondamentale che permette questo moto è la seguente:

```
MOVEFLY LINEAR TO POS(x1, y1, z1, x2, y2, z2, '') ADVANCE
    WITH $LIN_SPD = 1.5
```

L'istruzione *MOVEFLY*, abbinata alla clausola *ADVANCE*, permette di rendere l'istruzione di moto non bloccante, che significa che il server esegue l'istruzione successiva anche se quella attuale non è terminata. Se l'istruzione successiva è una nuova istruzione di moto, il moto risultante consisterà nell'interpolazione tra i due moti.

La clausola *LINEAR* indica che il moto sarà di tipo lineare.
La clausola *TO POS* indica che si vuole muovere il robot verso una posizione dello spazio operativo.
Infine, la clausola *WITH* permette di impostare una velocità lineare massima per il moto, velocità che potrà poi essere modificata con la funzione 7 del server, illustrata successivamente.

4.1.5 Moto lungo una traiettoria

Una funzionalità molto importante per un manipolatore è quella di eseguire moti lungo traiettorie specificate dal programmatore. Questa funzione corrisponde al valore 3 della variabile *risp*.
Come prima cosa, il server legge dalla socket il numero di nodi che caratterizzano la traiettoria e lo memorizza nella variabile *nodes*.
Successivamente, vengono aggiunti *nodes* nodi ad una variabile *myPath* di tipo nodo mediante l'istruzione:

```
NODE_APP(myPath, nodes)
```

Il tipo nodo è stato definito all'inizio del programma in questo modo:

```
TYPE myNode = NODEDEF
    $MOVE_TYPE
    $MAIN_POS
ENDNODEDEF
```

Ogni nodo è caratterizzato da un tipo di moto e da una posizione.
Poi viene avviato un ciclo *REPEAT UNTIL*, svolto per *nodes* volte, che legge dalla socket le coordinate di un nodo e le memorizza dentro il cammino con le seguenti due istruzioni:

```
myPath.NODE[c].$MOVE_TYPE := LINEAR
myPath.NODE[c].$MAIN_POS := POS(x1, y1, z1, x2, y2, z2)
```

La prima istruzione imposta il tipo di moto del nodo (lineare), la seconda imposta la posizione nello spazio operativo del nodo (le sei coordinate del nodo).

A questo punto la traiettoria è memorizzata dentro alla variabile *myPath*. Per far eseguire al robot il moto lungo questa traiettoria si utilizza il seguente frammento di codice:

```
REPEAT
 MOVEFLY ALONG myPath[1..nodes] ADVANCE
 c := c - 1
UNTIL  c > 1
```

che utilizza ciclicamente l'istruzione di moto *MOVEFLY* con la clausola *ALONG*, che permette di specificare il cammino lungo il quale muoversi e il numero di nodi di quel cammino da attraversare. L'istruzione viene eseguita tante volte quanti sono i nodi del cammino. Al termine del moto, viene ovviamente inviato un messaggio al client di conferma di esecuzione della funzione richiesta.

4.1.6 Moto relativo

Questa funzionalità, corrispondente al valore 4 della variabile *risp*, permette di muovere il robot a partire dalla posizione corrente, specificando di quanto muoversi lungo i tre assi x, y, z.
Dopo aver letto le tre quantità $x1, y1, z1$ dalla socket, il server esegue l'istruzione:

```
MOVE LINEAR RELATIVE VEC(x1, y1, z1) IN BASE
```

che permette, attraverso la clausola *RELATIVE VEC*, di utilizzare l'istruzione di moto come istruzione di moto relativo lungo un vettore di coordinate $x1, y1, z1$.
La clausola *IN BASE* specifica che il moto relativo è riferito alla terna di base.

4.1.7 Moto lungo l'asse del braccio

La quinta funzione del server (*risp*=5) permette di far muovere il braccio del robot lungo il proprio asse di una certa quantità $x1$ letta dalla socket. L'istruzione che permette di eseguire questo moto è la seguente:

```
MOVE LINEAR AWAY x1
```

4.1.8 Moto nello spazio dei giunti

Un'altra funzione molto importante è quella che permette di far muovere il robot nello spazio dei giunti invece che nello spazio operativo, andando a specificare il valore dei sei giunti rotoidali. Questa funzione corrisponde al valore 6 della variabile *risp*. Dopo aver letto da socket i valori dei sei giunti, memorizzati in un array di sei elementi reali, il server esegue la seguente istruzione:

```
MOVE TO {giunti[1], giunti[2], giunti[3], giunti[4],
    giunti[5], giunti[6]}
```

che, attraverso le parentesi graffe e la mancanza della clausola *POS*, identifica il moto nello spazio dei giunti invece che in quello operativo.

4.1.9 Modifica della velocità

Questa funzionalità, corrispondente al codice 7 della variabile *risp*, permette di modificare la velocità del robot durante i suoi movimenti. La nuova velocità, memorizzata nella variabile intera *vel*, deve essere compresa tra 1 e 100 e rappresenta un numero percentuale rispetto alla velocità massima fissata dal server. Pertanto, *vel=50* corrisponderebbe a una velocità pari alla metà della velocità massima impostata.
La nuova velocità viene impostata modificando il valore di due variabili di sistema con la seguente porzione di codice:

```
$PROG_SPD_OVR := vel
$ARM_SPD_OVR := vel
```

4.1.10 Posizione attuale nello spazio operativo

La funzione corrispondente a *risp*=8 permette di recuperare la posizione attuale del robot nello spazio operativo, caratterizzata quindi da sei coordinate: x, y, z identificanti la posizione e a, e, r identificanti l'orientazione. La posizione attuale del robot nello spazio operativo si può recuperare mediante l'istruzione:

```
p1 := ARM_POS
```

dove *ARM_POS* è una funzione predefinita e *p1* è una variabile di tipo *POSITION* caratterizzata da sei campi reali corrispondenti alle sei coordinate x, y, z, a, e, r, a cui si può accedere mediante l'operatore $''.''$ (ad esempio, p1.X accede alla prima coordinata).

4.1.11 Posizione attuale nello spazio dei giunti

Questa funzione, corrispondente al valore 9 della variabile *risp*, è molto simile alla precedente come funzionamento, ma restituisce la posizione del robot nello spazio dei giunti, ovvero il valore attuale di ognuno dei sei giunti rotoidali del manipolatore.

```
j1 := ARM_JNTP
```

ARM_JNTP è una funzione predefinita che restituisce la posizione del robot nello spazio dei giunti e *j1* è una variabile di tipo *JOINTPOS* caratterizzata da sei campi reali corrispondenti alle posizioni dei sei giunti, a cui si può accedere specificando, tra parentesi quadre, il numero del giunto desiderato (ad esempio, j1[3] accede al valore attuale del terzo giunto).

4.1.12 Chiusura della connessione

Quando il client non ha più bisogno dei *servizi* del server, può chiudere la connessione con esso, permettendogli di *servire* altri utenti, inviando 10 come valore della variabile *risp*. Se ciò accade, il server torna ad eseguire l'istruzione di accettazione di una connessione mediante l'istruzione di salto incondizionato *GOTO connessione*, in cui *connessione* è un'etichetta.

4.1.13 Terminazione del server

L'ultima funzione offerta dal server riguarda la sua terminazione. Per terminare l'esecuzione del server, occorre che il client invii 99 come valore della variabile *risp*. Questo comporta la chiusura del canale di comunicazione (socket) da parte del server mediante l'istruzione:

```
CLOSE FILE vi_netlun
```

e la fine del programma, identificata dalla parola riservata *END* seguita dal nome del programma (*ServerComau*).

Al termine dell'esecuzione di una funzione, ad eccezione delle ultime due, il server si mette in attesa di una nuova richiesta da parte del client attualmente connesso alla socket e riesegue gli stessi passi descritti in precedenza.

4.2 Il programma PDL2 CheckPosition

Il programma CheckPosition consente al robot di inviare periodicamente al client le coordinate della propria posizione, nello spazio operativo o nello spazio dei giunti a seconda delle richieste. Il programma è di tipo *NOT HOLDABLE* (vedi cap. 3) e, quindi, rimane in esecuzione anche durante l'esecuzione di altri programmi, in particolare del programma ServerComau che, invece, è di tipo *HOLDABLE*. Il programma CheckPosition viene avviato automaticamente all'avvio del programma ServerComau e termina in corrispondenza della terminazione di ServerComau.

L'implementazione del programma CheckPosition, ad eccezione dell'aggiunta della clausola *NOHOLD* nella dichiarazione del programma e del cambio di porta sulla quale aprire la socket, è una semplificazione del programma RobotComau, di cui si sono mantenute solo le funzioni:

- invio delle coordinate attuali nello spazio operativo (8);
- invio delle coordinate attuali nello spazio dei giunti (9);
- chiusura della connessione (10).

Solitamente, il programma viene utilizzato come ciclo continuo di una determinata funzione, ad esempio viene continuamente richiesta, da parte del client, la posizione del robot nello spazio operativo per monitorarne il moto.

Data la sostanziale uguaglianza delle caratteristiche tra questo programma e ServerComau descritto in precedenza, si rimanda alla sezione precedente per i dettagli implementativi (vedi par. 4.1.10).

4.3 Il programma PDL2 CheckError

Al fine di garantire la sicurezza per le persone e gli oggetti presenti nello spazio raggiungibile dal robot, è stato creato un secondo programma *NOT HOLDABLE*, cioè mantenuto sempre in esecuzione, che effettua alcuni controlli sulla posizione raggiunta dal robot, sia nello spazio operativo che in quello dei giunti, e blocca istantaneamente il moto nel caso si superino le posizioni limite impostate. Tale programma viene avviato automaticamente all'avvio del programma ServerComau e termina in corrispondenza della terminazione dello stesso programma.

I controlli sulla posizione vengono effettuati con una serie di istruzioni condizionali come la seguente:

```
IF (j1[5] > 100) OR (j1[5] < -100) THEN
 ERR_POST(44005, 'Posizione giunto 5 non consentita', 8)
ENDIF
```

La prima riga delimita l'intervallo di valori non ammissibili per il giunto 5: se, in un qualunque momento, una delle due condizioni diventa vera, viene eseguita l'istruzione successiva. La seconda riga è quella che genera un errore del controllore, causando l'arresto immediato del moto, mediante la primitiva PDL2 *ERR_POST* che ammette tre parametri obbligatori: un codice identificativo dell'errore, a scelta dell'utente e compreso tra 43008 e 44031, un messaggio da visualizzare sul terminale del robot che spieghi le cause dell'errore e, infine, il codice dell'azione da intraprendere come gestione dell'errore, in questo caso il robot viene messo in stato di *HOLD*.

Questo programma, come è facile vedere dalla sua semplicità, non comunica con il client, ma si preoccupa solo di intervenire sul moto del robot nel caso in cui si verifichino situazioni di emergenza. Per le necessità di questo lavoro di tesi, sono stati imposti dei limiti alla rotazione del quinto giunto, per evitare che la pinza andasse a sbattere contro il braccio durante la rotazione, e alla coordinata Z nello spazio operativo, per evitare che l'utensile sfondi il tavolo di lavoro. Rimane comunque possibile, in qualunque momento, aggiungere limiti di movimento o modificare quelli attuali, semplicemente andando ad intervenire sul programma CheckError.

4.4 La classe C++ RobotComau

La classe RobotComau, scritta in C++, fornisce numerosi metodi per l'utilizzo del robot *Comau Smart SiX*. Essa permette di inviare comandi al robot direttamente da un programma scritto in C++, mediante l'utilizzo di socket. Permette, inoltre, di monitorare la traiettoria seguita dal robot nei suoi movimenti, colloquiando via socket con il programma PDL2 Check Position (vedi par. 4.2). Il suo obiettivo principale rimane quello di far eseguire al robot manipolatore i comandi inviati dall'utente, facendo da tramite tra il programmatore e il linguaggio PDL2 del controllore C4G.

L'invio dei comandi al robot è facilitato dal fatto che il programma ServerComau.pdl2 è stato progettato come *menu di scelta*: pertanto, i metodi della classe non fanno altro che inviare, tramite socket, un codice identificativo della funzione che si vuole far svolgere al robot e, successivamente, i dati necessari al moto, come visto nel par. 4.1. Ad ogni comando inviato corrisponde un acknowledgment, ovvero un codice di risposta inviato dal server ai metodi della classe, che consente di identificare eventuali errori o di segnalare che l'istruzione è stata eseguita con successo.

Di seguito vengono analizzati in dettaglio i vari metodi offerti dalla classe RobotComau.

4.4.1 Dipendenze

La classe fa uso di funzioni e metodi presenti in alcune librerie standard. In particolare:

- **iostream**: per la gestione del'input/output con *cin* e *cout*;
- **time.h**: per le primitive di gestione del tempo, utili nel metodo *sleep*;
- **string.h**: per la gestione delle stringhe;
- **stdio.h**: per la gestione dei file di testo e dell'input/output;
- **fstream**: per la manipolazione dei file;
- **vector**: per l'utilizzo del tipo *vector*;

- **Point6d.h**: per l'utilizzo di oggetti della classe Point6d, definita appositamente per gestire punti composti da 6 coordinate;
- **winsock2.h**: per la comunicazione mediante socket.

4.4.2 Metodi pubblici

In questo paragrafo verranno illustrati i metodi pubblici a disposizione del programmatore e richiamabili da qualunque programma che utilizzi questa classe.

4.4.2.1 Il costruttore RobotComau

Il costruttore della classe ha come funzionalità principale quella di richiamare il metodo privato *connect_* per aprire il canale di comunicazione con il server, che può essere composto da una o due socket a seconda delle necessità: è stato infatti eseguito l'overload del costruttore, definendone una *versione* a due parametri, che consente di aprire solamente la socket di invio comandi, e una versione a tre parametri, che consente di aprire due socket, una per l'invio dei comandi e una per l'invio delle posizioni del robot. Questa seconda socket permette di monitorare la traiettoria seguita dal robot durante le istruzioni di moto, sfruttando il programma PDL2 CheckPosition descritto nel par. 4.2.

Il costruttore accetta, quindi, come parametri una stringa con l'indirizzo IP del controllore C4G sulla rete, un intero con il numero della porta su cui è attiva la socket dei comandi e, nella versione a tre parametri, un intero con il numero della porta su cui è attiva la socket delle posizioni. Ad esempio: *RobotComau r(160.78.28.98, 12345, 12344)* apre la comunicazione con il controllore sulla porta 12345 per i comandi e sulla porta 12344 per le posizioni (sola lettura).

Il codice del costruttore a due parametri è il seguente:

```
connect_(server_ip, server_port, lhSocket);
selsock=false;
```

mentre quello per la versione a tre parametri è il seguente:

```
connect_(server_ip, server_port, lhSocket);
connect_(server_ip, server_port_onlyREAD,
```

```
    lhSocket_OnlyREAD);
selsock=true;
```

in cui lhSocket indica il descrittore da assegnare alla socket comandi, mentre lhSocket_OnlyREAD il descrittore da assegnare alla socket posizioni, mentre la variabile booleana selsock, definita come attributo privato della classe, permette di stabilire quante socket sono state aperte (1 se falsa, 2 se vera), in modo da sapere quante chiuderne una volta terminato il programma.

4.4.2.2 Il distruttore ~RobotComau

La funzione del distruttore della classe, richiamato al termine del programma, è quella di chiudere tutte le socket aperte. Per fare questo, viene utilizzato il seguente codice:

```
closesocket(lhSocket);
if (selsock) closesocket(lhSocket_OnlyREAD);
WSACleanup();
```

in cui la prima riga chiude la socket dei comandi, che è stata sicuramente aperta; la seconda riga controlla il valore della variabile booleana selsock: se è vera, significa che era stata aperta anche la socket delle posizioni che, quindi, viene chiusa, altrimenti si passa alla terza riga che cancella eventuali dati rimasti in sospeso all'interno della socket, mediante la procedura predefinita della libreria *winsock2.h*.

4.4.2.3 void MoveCalibration(void);

Questo metodo permette di far muovere il robot nella posizione di calibrazione. Per farlo, è sufficiente inviare al robot il codice corrispondente a questa funzione, ovvero 1, con la seguente riga di codice:

```
SetFunction(1, lhSocket);
```

in cui *SetFunction* è un metodo privato della classe descritto in seguito, *1* è il comando da inviare e *lhSocket* è il descrittore della socket dei comandi aperta dal costruttore.

Successivamente, il metodo attende l'acknowledgment di avvenuta istruzione da parte del server con la seguente istruzione:

```
iResult=ReceiveData(data, lhSocket);
```

che utilizza il metodo privato *ReceiveData* descritto in seguito in cui *iResult* è un valore intero che restituisce il numero di caratteri letti, *data* è la stringa letta e *lhSocket* è il descrittore della socket.

4.4.2.4 void MoveLinearTo(char []);

Il metodo permette di muovere linearmente il robot verso una serie di punti dello spazio operativo letti da file; accetta come unico parametro una stringa con il percorso del file di testo contenente i punti da raggiungere. Ogni punto della traiettoria, nel file, viene memorizzato su una riga ed è composto da sei valori di tipo double separati da uno spazio.

Il suo funzionamento consiste nelle seguenti quattro righe di codice:

```
vector<Point6d> p;
p=ConvertFile(file);
for (unsigned int i=0; i<p.size(); i++)
    MoveLinearTo(p[i]);
```

La prima istruzione crea *p*, una variabile vettore di oggetti di classe Point6d, una classe appositamente creata che contiene sei coordinate di tipo double e i metodi per leggerle e modificarle. La seconda istruzione richiama il metodo *ConvertFile*, che restituisce un vettore di oggetti di classe Point6d contenenti i punti letti dal file di testo passatogli come argomento. Le ultime istruzioni eseguono un ciclo for con un numero di iterazioni pari alla dimensione della variabile *p*, ovvero al numero di punti letti dal file, in cui si richiama una seconda versione del metodo overloaded *MoveLinearTo* passandogli come parametro l'i-esimo punto del vettore *p*.

4.4.2.5 void MoveLinearTo(Point6d);

Questo metodo, molto spesso utilizzato come metodo di *basso livello* dal metodo precedente, richiede al server la funzione di moto lineare verso un punto dello spazio operativo inviandogli l'apposito codice (2) mediante l'istruzione seguente:

```
SetFunction(2, lhSocket);
```

Dopodiché, invia sulla socket comandi le sei coordinate del punto *p* passatogli come parametro, attendendo, ad ogni invio, la ricezione dell'acknowledgment di avvenuta ricezione. Il codice con cui si inviano e si ricevono dati sulla socket è il seguente:

```
iResult=SendData(data, lhSocket);
iResult=ReceiveData(data, lhSocket);
```

Facilmente intuibile il funzionamento dello stesso: la prima riga richiama il metodo *SendData* che invia sulla socket *lhSocket* la stringa *data* e mette in *iResult* il numero di dati scritti, mentre la seconda riga, come è già stato detto in precedenza, legge dati dalla socketo *lhSocket*, in particolare l'acknowledgment, e li memorizza nella stringa *data*, memorizzando in *iResult* il numero di dati letti.

4.4.2.6 void MoveTrajectory(char []);

La funzione offerta da questo metodo è il moto del robot lungo una traiettoria di punto dello spazio operativo specificata dall'utente in un file di testo, il cui percorso è passato come parametro del metodo. Le tre istruzioni che vengono eseguite da questa funzione sono le seguenti:

```
vector<Point6d> p;
p=ConvertFile(file);
MoveTrajectory(p);
```

Le prime due righe sono identiche a quelle del metodo *MoveLinearTo(char[])*, mentre la terza riga si limita a richiamare il metodo *MoveTrajectory* passandogli un vettore di punti come parametro.

4.4.2.7 void MoveTrajectory(vector<Point>);

Questo metodo consente di muovere il robot lungo una traiettoria di punti memorizzata nel vector di punti passato come parametro. Come prima cosa, viene selezionata la funzione di moto lungo una traiettoria inviando l'apposito comando (3) al server:

```
SetFunction(3, lhSocket);
```

Successivamente, si invia al server il numero di punti che compongono la traiettoria, ottenuti con queste istruzioni:

```
_itoa_s(p.size(),data,10);
strcat_s(data,"\n");
iResult=SendData(data, lhSocket);
```

in cui la prima istruzione converte la dimensione del vettore p (intera) in una stringa (data) mediante la funzione *_itoa_s*; la seconda istruzione aggiunge il carattere di fine riga alla stringa *data*; l'ultima riga invia la stringa *data* sulla socket, mediante il metodo *SendData*.

A questo punto, vengono inviate sulla socket, una ad una, le sei coordinate di ogni punto, attendendo la ricezione dell'acknowledgment dopo ogni invio, mediante un ciclo for che scorre tutto il vettore di punti. Terminato l'invio, si attende di ricevere dal server l'acknowledgment che avvisa che il robot si è posizionato correttamente seguendo la traiettoria.

4.4.2.8 void MoveRelative(double, double, double);

Questa funzione permette di muovere il robot a partire dalla posizione corrente nelle tre dimensioni dello spazio operativo x, y, z; accetta come parametri tre valori double che corrispondono, rispettivamente, a quanto spostarsi lungo l'asse x, lungo l'asse y e lungo l'asse z. Per fare ciò, viene inviato al server il comando corrispondente al moto relativo (4) e le tre coordinate x, y, z, si attende la ricezione dell'acknowledgment relativo al posizionamento corretto del robot e si restituisce il controllo al programma chiamante.

4.4.2.9 void MoveAxis(double);

Questo semplice metodo permette di muovere il robot lungo l'asse z della terna utensile (*TOOL*) di una certa quantità double passata come parametro. La sua implementazione consiste nel selezionare la funzione corrispondente sul server, inviando il codice 5, e a inviare il valore del moto lungo l'asse z.

4.4.2.10 void MoveJoint(char []);

Questo metodo è molto simile alla prima versione del metodo di moto lineare, con l'unica differenza che, invece di richiamare il metodo di *basso livello MoveLinearTo(Point6d p)* viene richiamato il metodo *MoveJoint(Point6d p*, che permette di interpretare i punti letti dal file come valori dei sei giunti rotoidali del robot. Il moto risultante sarà, pertanto, un moto nello spazio dei giunti. Per completezza, il codice della funzione è il seguente:

```
vector<Point6d> p;
p=ConvertFile(file);
for (unsigned int i=0; i<p.size(); i++)
MoveJoint(p[i]);
```

4.4.2.11 void MoveJoint(Point);

Questo metodo è molto simile alla seconda versione del metodo di moto lineare, quella di *basso livello*, con l'unica differenza che, invece di selezionare la funzione di moto lineare del server (5), seleziono la funzione di moto nello spazio dei giunti (6). L'implementazione dell'invio delle coordinate dei punti è del tutto identica alla precedente.

4.4.2.12 void SetSpeed(char[]);

Questo metodo non causa nessun moto del manipolatore, ma si limita ad impostare una velocità massima di movimento del robot. Accetta come parametro una stringa di testo corrispondente ad un valore percentuale di velocità: in sostanza, la funzione permette di ridurre di una certa quantità percentuale la velocità massima impostata dal server PDL2.

Dopo aver selezionato la funzione del server corrispondente (7), la velocità viene convertita in intero e controllata (deve essere compresa tra 0 e 100%). A questo punto, viene aggiunto il carattere di terminazione stringa alla velocità da impostare che, poi, viene inviata al server.

4.4.2.13 Point6d GetCurrentPosition(void);

Questo metodo restituisce in una variabile di tipo Point6d la posizione attuale del robot nello spazio operativo. A differenza dei metodi precedenti, questo metodo inverte il senso di comunicazione: infatti, il client invia al server il codice della funzione richiesta (8) e poi legge dal server le varie coordinate, che andrà a convertire in formato numerico e a memorizzarle nella variabile di tipo Point6d, ed invia un acknowledgment di avvenuta ricezione al server.

Per fare questo, viene utilizzato il seguente codice:

```
iResult=ReceiveData(data, lhSocket);
iResult=SendData("OK\n", lhSocket);
p.setX(atof(data));
```

La prima riga riceve la prima coordinata dal server tramite la socket comandi e la memorizza dentro alla variabile *data*. La seconda riga invia l'acknowledgment al server tramite la socket comandi. L'ultima istruzione converte la coordinata da stringa a numero reale mediante la funzione *atof* e il risultato lo passa come parametro al metodo *setX* di *p* che va a modificare la coordinata x di p.

4.4.2.14 Point6d GetCurrentPositionOR(void);

Questo metodo è identico al precedente ad eccezione della socket su cui legge e scrive, che questa volta è la socket delle posizioni. Questo permette di continuare a richiedere al programma *CheckPosition* la posizione del robot nello spazio operativo istante per istante, portando avanti *parallelamente* il moto e il controllo della posizione del robot.

4.4.2.15 Point GetJoint(void);

Anche questo metodo è identico al precedente, ad eccezione della funzione del server richiesta, che questa volta è la numero 9, quella corrispondente alla richiesta delle coordinate attuali nello spazio dei giunti. L'implementazione dell'invio delle coordinate dei punti è del tutto identica alla precedente.

4.4.2.16 Point6d GetJointOR(void);

Questo metodo è identico al precedente ad eccezione della socket su cui legge e scrive, che questa volta è la socket delle posizioni. Questo permette di continuare a richiedere al programma *CheckPosition* la posizione del robot nello spazio dei giunti istante per istante, portando avanti *parallelamente* il moto e il controllo della posizione del robot.

4.4.2.17 void CloseConnection(void);

Questo metodo di controllo della connessione permette di chiudere la connessione con il server, mantenendo tuttavia attivo il server, mettendolo nello stato di attesa di una nuova connessione da parte di un client. Per implementare questo, si svolgono le seguenti istruzioni:

```
close_(lhSocket);
if (selsock) close_(lhSocket_OnlyREAD);
```

Lo scopo di queste istruzioni è richiamare il metodo privato *close_* per ogni socket attiva, metodo che andrà a richiedere al server la funzione di chiusura connessione (10).

4.4.2.18 void StopServer(void);

Nonostante anche questo sia un metodo di controllo della connessione, a differenza del precedente questo metodo provvede a terminare il programma ServerComau.pdl2. Per implementare questo, si svolgono le seguenti istruzioni:

```
stop_(lhSocket);
if (selsock) stop_(lhSocket_OnlyREAD);
```

Lo scopo di queste istruzioni è richiamare il metodo privato *stop_* per ogni socket attiva, metodo che andrà a richiedere al server la funzione di terminazione (99).

4.4.2.19 void PrintMenu(void);

Questo semplice metodo non fa altro che visualizzare sullo schermo del client l'elenco delle funzioni disponibili sul server.

4.4.2.20 void Sleep(float);

L'ultimo metodo pubblico è il metodo che consente di sospendere l'esecuzione del programma per un certo numero reale di secondi. Questo può essere utile per far effettuare delle pause al robot tra un moto e l'altro. L'implementazione di questa semplice funzione si fa nel modo seguente:

```
time_t start;
time_t current;
time(&start);
do {
time(&current);
}
while(difftime(current,start) < delay);
```

Nella sostanza, si memorizza dentro alla variabile *start* l'istante di tempo nel quale si inizia ad eseguire la funzione, poi si ricalcola tante volte l'istante di tempo attuale, memorizzato nella variabile *current* finché la differenza *current-start* non diventa maggiore o uguale al ritardo (delay) passato come parametro.

4.4.3 Metodi e attributi privati

Nella sezione privata della classe, sono stati dichiarati tre attributi utilizzati nelle implementazioni dei metodi della classe. In particolare:

- *SOCKET lhSocket*: contiene il descrittore della socket per l'invio di comandi al robot;
- *SOCKET lhSocket_OnlyREAD*: contiene il descrittore della eventuale socket per la lettura della posizione del robot;
- *bool selsock*: risulta vera se sono state aperte entrambe le socket (comandi e posizione), falsa se è stata aperta solo la socket per l'invio di comandi al robot.

Nei prossimi paragrafi vengono, invece, descritti i metodi privati della classe, che vengono richiamati dai metodi pubblici per l'implementazione delle funzionalità offerte.

4.4.3.1 void connect_ (const char[], int, SOCKET&);

Questo metodo consente di aprire una socket di comunicazione tra il PC e il robot. La funzione accetta tre parametri: l'ip del robot, la porta sulla quale aprire la socket e il descrittore a cui associare la socket. La procedura di creazione di una nuova socket è piuttosto standard, grazie alla libreria *winsock2*. Le istruzioni fondamentali sono le seguenti:

```
sock = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
memset(&lSockAddr,0, sizeof(lSockAddr));
lSockAddr.sin_family = AF_INET;
lSockAddr.sin_port = htons(server_port);
lSockAddr.sin_addr.s_addr = inet_addr(server_ip);
lConnect = connect(sock,(SOCKADDR *)&lSockAddr,
      sizeof(SOCKADDR_IN));
```

La prima riga richiama la primitiva *socket* che consente di creare la socket di comunicazione, specificando la famiglia di indirizzi da utilizzare (*INET*), il tipo di socket (*STREAM*) e il protocollo utilizzato (*TCP*). La seconda riga crea la struttura che conterrà i parametri di configurazione della socket. Le tre righe successive consentono di memorizzare nella struttura di configurazione la famiglia di indirizzi, la porta sulla quale aprire la socket e l'indirizzo del server. La funzione *htons* consente di convertire in formato TCP il numero della porta. L'ultima riga mette la socket in attesa di una connessione, utilizzando la primitiva *connect* della libreria *winsock2*.

4.4.3.2 void SetFunction(int, SOCKET&);

Questa funzione consente di inviare al server il codice della funzionalità richiesta, specificato nel primo parametro passato al metodo. Il secondo parametro è, invece, il descrittore della socket di comunicazione.

L'implementazione di questo metodo è molto semplice e sfrutta i due metodi privati *SendData* per l'invio del codice e *ReceiveData* per la ricezione dell'acknowledgment.

4.4.3.3 vector<Point6d> ConvertFile(char[]);

Questo metodo consente di convertire un file di testo contenente dei punti in un vettore di oggetti *Point6d*. Il codice che permette l'implementazione di questa funzione è il seguente:

```
vector<Point6d> l;
int num = 0;
char t[80];
ifstream fin(file);
while (fin.getline(t,80)) {
Point6d p;
p.setX(atof(strtok(t," ")));
p.setY(atof(strtok(NULL," ")));
p.setZ(atof(strtok(NULL," ")));
p.setA(atof(strtok(NULL," ")));
p.setB(atof(strtok(NULL," ")));
p.setC(atof(strtok(NULL," ")));
l.push_back(p);
num++;
}
fin.close();
return l;
```

Le prime righe servono per dichiarare le variabili utilizzate nel metodo: un vettore di Point6d vuoto di nome *l*, una variabile intera *num* che conterrà il numero di elementi del vettore e una variabile stringa *t*. Successivamente, viene aperto il file di testo, passato come parametro della funzione, mediante la primitiva *ifstream*, assegnando al file il descrittore *fin*. Poi viene avviato un ciclo while che legge una riga del file ad ogni iterazione con la primitiva *fin.getline* e la memorizza nella variabile *t*. Il ciclo termina quando non ci sono più righe da leggere. Ad ogni iterazione, vengono lette le sei coordinate, separate utilizzando la funzione *strtok*, e memorizzate in un oggetto *p* di classe *Point6d* che poi viene inserito nel vector mediante il metodo predefinito *push_back* prima di incrementare la numerosità del vettore stesso

num. Infine, a ciclo terminato, il file viene chiuso e viene restituito come valore del metodo il vector *l*.

4.4.3.4 void close_ (SOCKET&);

Questo metodo consente di chiudere la socket di comunicazione, limitandosi ad inviare il codice 10 corrispondente alla funzione di fine comunicazione del server e ad attendere l'acknowledgment dal server.

4.4.3.5 void stop_ (SOCKET&);

Questo metodo consente di terminare il server PDL2, limitandosi ad inviare il codice 99, che corrisponde alla funzione di terminazione del server, e ad attendere l'acknowledgment dal server.

4.4.3.6 int SendData(char[], SOCKET&);

Questo metodo rappresenta la primitiva fondamentale per l'invio di comandi al robot, consentendo al programma C++ di scrivere sulla socket. Accetta due parametri: la stringa da scrivere sulla socket e l'identificativo della socket. Il metodo è tuttavia molto semplice da implementare: infatti, l'istruzione fondamentale è la seguente:

```
iResult = send(sock, data, strlen(data), 0);
```

Questa istruzione utilizza la primitiva *send* della libreria *winsock2* per la scrittura di dati su socket. Alla primitiva vengono passati quattro parametri: l'identificativo della socket, la stringa da scrivere, il numero di caratteri da scrivere e alcuni flag configurazionali (non utilizzati in questa implementazione). La primitiva restituisce il numero di caratteri effettivamente scritti, che vengono memorizzati nella variabile intera *iResult*, poi restituita come valore di ritorno dal metodo.

4.4.3.7 int ReceiveData(char[], SOCKET&);

Questo metodo rappresenta la primitiva fondamentale per la lettura di dati dalla socket. Accetta due parametri: una stringa su cui memorizzare i dati letti e l'identificativo della socket. Il metodo è tuttavia molto semplice da implementare: infatti, l'istruzione fondamentale è la seguente:

```
iResult = recv(sock, data, 80, 0);
```

Questa istruzione utilizza la primitiva *recv* della libreria *winsock2* per la lettura di dati su socket. Alla primitiva vengono passati quattro parametri: l'identificativo della socket, la stringa su cui memorizzare i dati letti, il numero di caratteri da leggere e alcuni flag configurazionali (non utilizzati in questa implementazione). La primitiva restituisce il numero di caratteri effettivamente letti, che vengono memorizzati nella variabile intera *iResult*, poi restituita come valore di ritorno dal metodo.

4.4.4 Utilizzo

Per utilizzare la classe, sul robot deve essere in esecuzione il programma ServerComau.cod, memorizzato nella memoria di massa del controllore. Per far eseguire tale programma al robot, si veda il par. 2.1.2. Per poter utilizzare la classe nel proprio programma C++, è sufficiente includerla con l'istruzione *include (RobotComau.h)* e creare un oggetto di classe RobotComau. Successivamente, sarà possibile richiamare tutti i metodi pubblici dell'oggetto a seconda delle esigenze. Una volta terminato il programma, sarà possibile scegliere se terminare la connessione, mantenendo attivo il server per eventuali connessioni future, o terminare il server, con i due appositi metodi. In quest'ultimo caso, sarà poi necessario riavviare il server nel caso si voglia utilizzare nuovamente il robot.

Nel prossimo capitolo vengono presentate alcune prove sperimentali condotte nel laboratorio di robotica in cui si vedrà nei dettagli l'applicazione di tale libreria con alcuni frammenti di codice.

4.5 La classe C++ Gripper

La classe Gripper, scritta in C++, permette di gestire in maniera molto semplice il gripper robotico Schunk Pg70. Le funzioni fondamentali sono quelle di apertura e chiusura della pinza, in cui è possibile impostare una serie di parametri ben specifici caratterizzanti il moto di apertura e chiusura.

Il controllo della pinza si svolge, sostanzialmente, seguendo i passi dell'algoritmo di fig. 4.2.

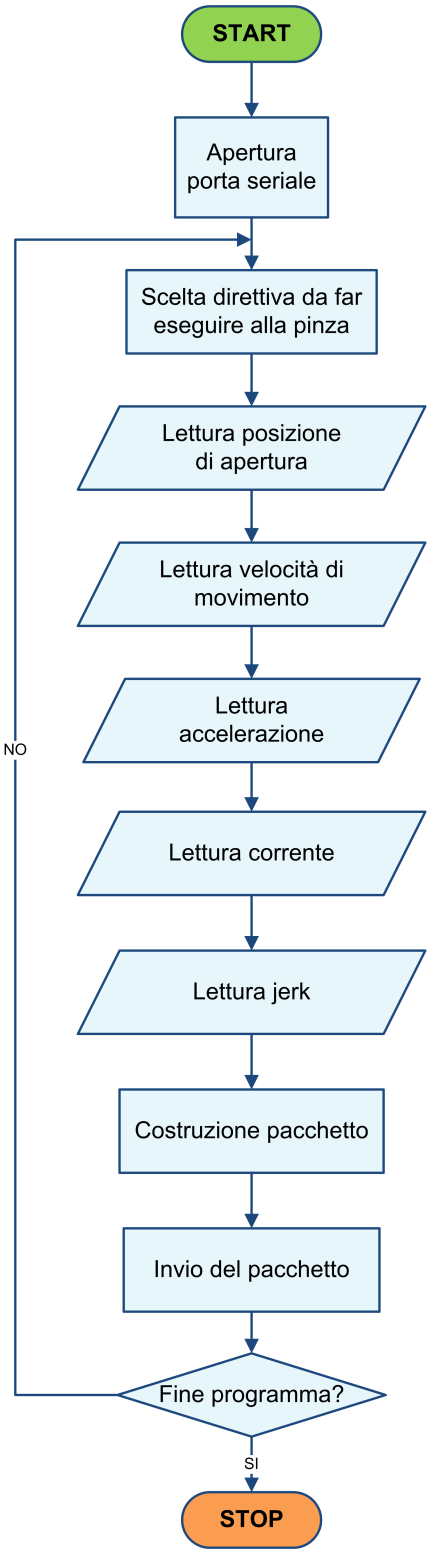

Figura 4.2: Passi da eseguire per il controllo del gripper Schunk Pg70

4.5.1 Dipendenze

La classe fa uso di funzioni e metodi presenti in alcune librerie standard. In particolare:

- **iostream**: per la gestione del'input/output con *cin* e *cout*;
- **time.h**: per le primitive di gestione del tempo, utili nel metodo *sleep*;
- **windows.h**: per l'utilizzo di primitive di gestione della comunicazione su porta seriale;
- **SchunkSerialProtocol.hpp**: per l'utilizzo di oggetti della classe SchunkSerialProtocol, sviluppata da Marco Tarasconi [1] come driver del gripper Schunk Pg70.

4.5.2 Metodi pubblici

Di seguito vengono illustrati i metodi pubblici della classe.

4.5.2.1 Il costruttore Gripper(char[],int baud=9600);

Il costruttore della classe ha come compito principale quello di creare la comunicazione seriale, che permetterà di inviare i pacchetti contenenti i comandi al gripper. Inoltre, si preoccupa anche di inizializzare gli attributi (variabili) della classe. Accetta come parametri una stringa contenente il nome della porta seriale (ad esempio COM2) e la velocità di invio di dati (in baud), che ha come valore predefinito 9600. L'istruzione utilizzata per aprire la connessione seriale è la seguente:

```
serial = CreateFile(com, GENERIC_READ | GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, 0, NULL);
```

in cui *com* indica il nome della porta seriale passato come parametro al costruttore, il secondo parametro indica una comunicazione bidirezionale, su cui è possibile sia leggere che scrivere, il terzo parametro indica la condivisione della comunicazione, il quarto eventuali attributi di sicurezza, il quinto indica di non creare la connessione nel caso fosse già esistente ma di aprire quella esistente e gli ultimi due indicano

ulteriori flag o attributi non utilizzati in questo caso. La funzione *CreateFile* restituisce un oggetto di tipo *HANDLE* che viene memorizzato nella variabile *serial* dello stesso tipo, che servirà poi per accedere alla seriale stessa, utilizzandola come una sorta di *descrittore*.

4.5.2.2 Il distruttore ∼Gripper(void)

La funzione del distruttore della classe, richiamato al termine del programma, è quella di riportare la pinza nella posizione di riferimento e di chiudere la comunicazione seriale. Per fare questo, viene utilizzato il seguente codice:

```
reference();
CloseHandle(serial);
```

in cui la prima riga riporta il gripper alla posizione di riferimento mediante il metodo della classe stessa *reference*, descritto più avanti, mentre la seconda riga chiude la comunicazione seriale mediante la primitiva *CloseHandle* della libreria *windows.h*.

4.5.2.3 void move(float, float speed=-1, float acceleration=-1, float current=-1, float jerk=-1);

Questo è il metodo fondamentale della classe, ovvero quello che permette di far muovere la pinza in una certa posizione, specificando eventualmente alcuni parametri del moto. Accetta cinque parametri di tipo float: la posizione da raggiungere, compresa tra 0 (pinza chiusa) e 65,9 mm (pinza completamente aperta), la velocità di movimento (compresa tra 0 e 82 mm/s) che, se non specificata, viene impostata a 10 mm/s, l'accelerazione (compresa tra 0 e 328 mm/s^2) che, se non specificata, viene impostata a 10 mm/s^2, la corrente impiegata che è indice della forza applicata dalla pinza sugli oggetti che afferra (compresa tra 0 e 6,5 A), valore di default 2 A, e il jerk (compreso tra 0 e 10000 mm/s^3) che, se non specificato, viene assunto pari a 0 mm/s^3. Si noti che la posizione di apertura massima della pinza è limitata a 65,9 mm, quindi un limite più stringente rispetto alle caratteristiche della pinza (68 mm), in quanto la libreria *SchunkSerialProtocol* [1] limita ulteriormente l'apertura della pinza per evitare di sforzare troppo i motori interni.

L'implementazione di questo metodo è la seguente:

```
physicValueOptionsOut[0]=speed;
physicValueOptionsOut[1]=acceleration;
physicValueOptionsOut[2]=current;
physicValueOptionsOut[3]=jerk;
directive = POSITION_MOVEMENT_DIR;
elements = createPacketToBeWritten(toBeWritten,directive,
    crc8_1,crc8_2,&crc16,position,hex,&stopFlag,
    physicValueOptionsOut,stringIn,input);
WriteFile(serial, toBeWritten, sizeof(toBeWritten),
    &iBytesWritten, NULL);
```

Le prime quattro righe memorizzano all'interno di un array di float, dichiarato come attributo della classe, i valori dei parametri *speed*, *acceleration*, *current*, *jerk*.
La quinta riga memorizza all'interno della variabile *directive* il codice corrispondente al comando della pinza di moto verso una particolare posizione, codice memorizzato dentro a una costante definita nella libreria *SchunkSerialProtocol*.
La sesta riga provvede alla creazione del pacchetto che verrà poi inviato sulla seriale alla pinza dall'istruzione successiva. Viene quindi richiamata la funzione *createPacketToBeWritten* della libreria *SchunkSerialProtocol* passandogli come parametri la stringa di destinazione del pacchetto da inviare, il comando richiesto alla pinza, tre variabili corrispondenti al controllo di ridondanza (CRC), la posizione voluta, una variabile *stopFlag* utilizzata dalla funzione al suo interno, l'array dei parametri caratterizzanti il moto e, infine, due variabili non utilizzate in questo caso (per approfondimento, si consulti il testo [1] in bibliografia).

4.5.2.4 void open(float speed=10.0, float acceleration=10.0, float current=2.0, float jerk=0.0);

Questo metodo è stato definito come *scorciatoia* per il comando di apertura della pinza. La sua implementazione, perciò, risulta una semplice chiamata al metodo *move* in cui si passa, come valore della posizione da raggiungere, la posizione massima (65,9 mm). Risulta sempre possibile specificare eventuali parametri di configurazione del moto di apertura.

4.5.2.5 void close(float speed=10.0, float acceleration=10.0, float current=2.0, float jerk=0.0);

Questo metodo è stato definito come *scorciatoia* per il comando di chiusura della pinza. La sua implementazione, perciò, risulta una semplice chiamata al metodo *move* in cui si passa, come valore della posizione da raggiungere, la posizione minima ($0\ mm$). Risulta sempre possibile specificare eventuali parametri di configurazione del moto di apertura.

4.5.2.6 void reference(void);

Questo metodo consente di far posizionare il gripper nella posizione di riferimento, ovvero la posizione in cui la pinza è chiusa. Per questioni di sicurezza, è sempre bene riportare il gripper nella posizione di riferimento prima di staccargli la corrente, al fine di non sforzare inutilmente i motori. L'implementazione di questa funzione è la seguente:

```
directive = REFERENCE_MOV_DIR;
elements = createPacketToBeWritten(toBeWritten,directive,
    crc8_1,crc8_2,&crc16,physic,hex,&stopFlag,
    physicValueOptionsOut,stringIn,input);
WriteFile(serial, toBeWritten, sizeof(toBeWritten),
    &iBytesWritten, NULL);
```

in cui la prima riga specifica la direttiva da richiedere alla pinza, data dal codice contenuta nella costante *REFERENCE_MOV_DIR*, la seconda riga crea il pacchetto (in questo caso, l'array di configurazione del moto è vuoto) e l'ultima riga *scrive* il pacchetto sulla seriale, inviandolo al gripper.

4.5.2.7 void exitError(void);

In alcune situazioni, ad esempio a seguito di una mancanza di tensione di alimentazione, è possibile che la pinza entri nello stato di errore e non risponda più ai comandi. Questo metodo consente di inviare alla pinza uno speciale comando per farla uscire da questo stato, riportandola nello stato normale. L'implementazione di questa funzione è la seguente:

```
directive = ERROR_ACK_DIR;
elements = createPacketToBeWritten(toBeWritten,directive,
    crc8_1,crc8_2,&crc16,physic,hex,&stopFlag,
    physicValueOptionsOut,stringIn,input);
WriteFile(serial, toBeWritten, sizeof(toBeWritten),
    &iBytesWritten, NULL);
```

in cui la prima riga specifica la direttiva da richiedere alla pinza, data dal codice contenuta nella costante *ERROR_ACK_DIR*, la seconda riga crea il pacchetto (in questo caso, l'array di configurazione del moto è vuoto) e l'ultima riga *scrive* il pacchetto sulla seriale, inviandolo al gripper.

4.5.3 Attributi privati

Nella sezione *private* vengono dichiarati alcuni attributi utili nell'implementazione dei metodi pubblici. In particolare:

- *unsigned char toBeWritten[30]*: contiene il pacchetto da inviare al gripper;
- *unsigned int directive*: contiene il comando richiesto al gripper;
- *unsigned int crc16*: contiene il codice CRC a 16 bit del pacchetto;
- *unsigned char crc8_1, crc8_2*: contengono le due parti a 8 bit del codice CRC del pacchetto;
- *float physic*: variabile necessaria alla costruzione del pacchetto, non utilizzata in questo caso;
- *unsigned long hex*: variabile necessaria alla costruzione del pacchetto, non utilizzata in questo caso;
- *float physicValueOptionsOut[8]*: contiene gli eventuali parametri caratterizzanti il moto (velocità, accelerazione, corrente e jerk);
- *char stringIn[40]*: variabile necessaria alla costruzione del pacchetto, non utilizzata in questo caso;

- *char input[300]*: variabile necessaria alla costruzione del pacchetto, non utilizzata in questo caso;
- *bool stopFlag*: variabile necessaria alla costruzione del pacchetto, non utilizzata in questo caso;
- *unsigned int elements*: contiene il numero di pacchetti restituiti dall'invocazione del metodo *createPacketToBeWritten* della libreria *SchunkSerialProtocol*, sempre 1 in questo caso;
- *DWORD iBytesWritten*: contiene il numero di bytes scritti sulla seriale;
- *HANDLE serial*: identifica l'handler della connessione seriale creata (*descrittore*);
- *COMMTIMEOUTS m_CommTimeouts*: variabile utilizzata per impostare un tempo di timeout alla connessione seriale;
- *DCB PortDCB*: struttura che contiene le impostazioni con le quali la connessione seriale viene aperta.

4.5.4 Utilizzo

Per utilizzare la classe, occorre ovviamente collegare il gripper al PC e all'alimentazione. Inoltre, è bene sapere il nome che il sistema operativo ha assegnato alla porta seriale a cui è stata collegata la pinza. All'interno del programma C++, è sufficiente includere la classe con l'istruzione *include (Gripper.h)* e creare un oggetto di classe Gripper, specificando il nome della porta seriale e, eventualmente, la velocità di invio di dati. Successivamente, sarà possibile richiamare tutti i metodi pubblici dell'oggetto a seconda delle esigenze.

Nel prossimo capitolo vengono presentate alcune prove sperimentali condotte nel laboratorio di robotica in cui si vedrà nei dettagli l'applicazione di tale libreria con alcuni frammenti di codice.

Capitolo 5

Prove sperimentali

In questo capitolo verranno presentati i risultati di alcune prove sperimentali condotte nel mese di Novembre 2010 nel laboratorio di robotica dell'Università di Parma per testare la libreria sviluppata.

5.1 Moto a onda quadra

La prima prova consiste nel far muovere il robot *Comau Smart SiX* lungo una traiettoria a forma di onda quadra. Questa prova, pertanto, non coinvolge la classe Gripper, ma solamente la classe RobotComau.

L'implementazione di questo programma consiste nel far muovere il robot in una posizione iniziale mostrata in fig. 5.1 e memorizzata dentro il file di testo *inizio1.txt* con il metodo *MoveLinearTo* e, da lì, farlo muovere lungo la traiettoria a forma di onda quadra mediante l'utilizzo di una serie di chiamate al metodo *MoveRelative*. Infine, viene chiusa la connessione con il server e il programma termina. Per questioni di sicurezza e per un'analisi più precisa, la velocità è stata impostata al 20% della velocità massima.

Il codice del programma viene presentato di seguito.

```
#include "RobotComau.h"
#include <pthread.h>
RobotComau r("160.78.28.98",12345,12344);
bool stop=false;
```

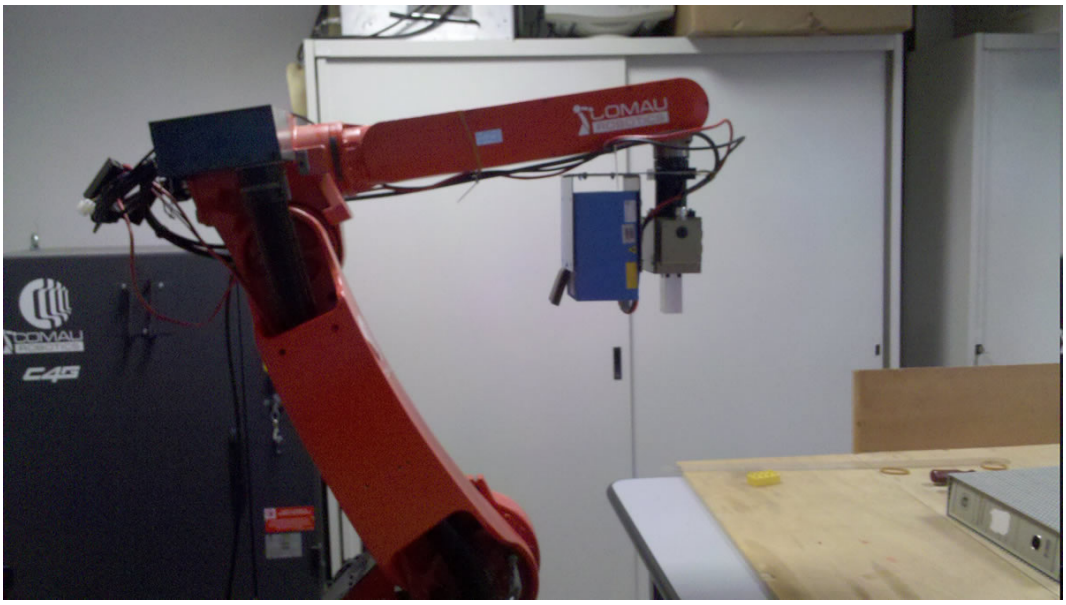

Figura 5.1: Posizionamento iniziale del robot durante la prima prova

L'intestazione del programma si preoccupa di includere le librerie necessarie al funzionamento dello stesso, prima fra tutte la *RobotComau*. Successivamente, viene creato un oggetto globale *r* di classe *RobotComau* aprendo due socket diverse sul robot, una sulla porta 12345 (la socket dati) e una sulla porta 12344 (la socket delle posizioni). Infatti, al fine di monitorare le posizioni assunte, viene creata anche la seconda socket.

```
int main () {
  pthread_t thread1, thread2;
  pthread_create(&thread2,NULL,motion,NULL);
  pthread_create(&thread1,NULL,getPos,NULL);
  pthread_join(thread2,NULL);
  pthread_join(thread1,NULL);
}
```

Il programma crea pertanto due threads, *thread1* e *thread2*, e le mette in esecuzione, associandole rispettivamente alla funzione *getPos* e *motion* e, infine, mediante l'istruzione *pthread_join*, sospende la thread *main* finché *thread1* e *thread2* non sono terminate.

```
void * motion (void * arg) {
  r.SetSpeed("20");
  r.MoveLinearTo("inizio1.txt");
  r.MoveRelative(0,0,-200);
  r.MoveRelative(0,100,0);
  r.MoveRelative(0,0,200);
  r.MoveRelative(0,100,0);
  r.MoveRelative(0,0,-200);
  r.MoveRelative(0,100,0);
  r.MoveRelative(0,0,200);
  r.MoveRelative(0,100,0);
  r.MoveRelative(0,0,-200);
  r.MoveRelative(0,100,0);
  r.MoveRelative(0,0,200);
  stop=true;
  r.CloseConnection();
  return NULL;
}
```

La funzione *motion* è quella che invia i comandi di moto al robot. La prima istruzione della funzione consente di impostare la velocità al 20% della velocità massima, la seconda istruzione permette di far muovere il robot in una posizione iniziale memorizzata nel file di testo *inizio1.txt* contenente un punto nello spazio (ad esempio il punto di coordinate $0, 500, 850, 90, 180, 90$, mentre le istruzioni successive permettono di far muovere il robot di una certa quantità (in mm) lungo i tre assi cartesiani. Ad esempio, spostare il robot di -200 lungo l'asse Z significa far abbassare il braccio di 200 mm, mentre spostarlo di 100 lungo l'asse Y significa far avanzare il braccio di 100 mm.

Si noti che lo stesso tipo di moto è anche realizzabile semplicemente utilizzando l'istruzione:

```
r.MoveTrajectory("traiettoria.txt");
```

in cui il file *traiettoria.txt* contiene i punti attraversati dalla traiettoria. Tuttavia, l'obiettivo di questa prova è di provare le primitive di moto verso un singolo punto,

mentre l'istruzione di moto lungo una traiettoria specificata dall'utente verrà testata nella terza prova.

Infine, la funzione impone vera la variabile globale *stop* e chiude la connessione con il server, facendo ritornare la funzione.

```
void * getPos (void * arg) {
  Point6d p1;
  ofstream fout("traccia.txt");
  while (!stop) {//Condizione di stop
    p1 = r.GetCurrentPositionOR();
    fout<<p1;  }
  fout.close();
  return NULL;
}
```

La funzione *getPos*, invece, ha il compito di leggere periodicamente la posizione raggiunta dal robot e di memorizzarla all'interno del file *traccia.txt*. Per fare questo, viene aperto il file e, finché la variabile globale *stop* è falsa, viene letta la posizione del robot mediante il metodo *GetCurrentPositionOR* dell'oggetto globale *r* di classe *RobotComau* e scritta sul file della traccia. Infine, quando *stop* diventa vera, viene chiuso il file e la funzione ritorna.

Il risultato del moto è visibile in fig. 5.2: la linea continua nera mostra l'andamento ideale del moto, mentre i puntini blu mostrano una serie di *campioni* delle posizioni realmente attraversate dal robot.

Questa prova mostra anche la notevole precisione nel moto da parte del robot.

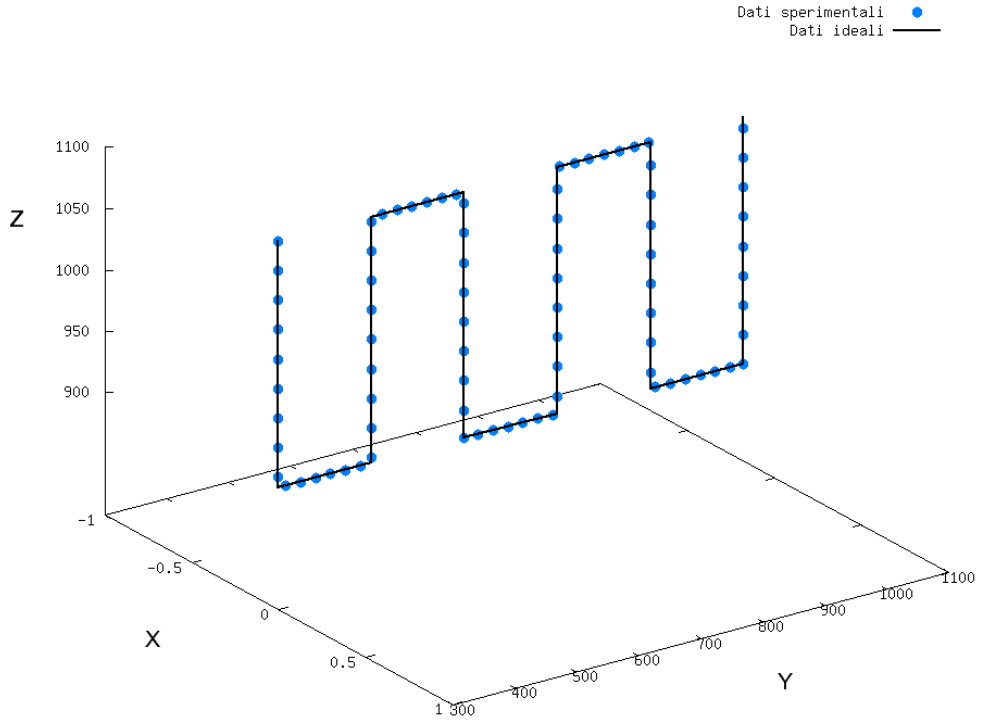

Figura 5.2: Grafico del moto del robot durante la prima prova nelle tre dimensioni X, Y, Z (realizzato con gnuplot)

5.2 Spostamento di un oggetto

La seconda prova, che richiede anche l'utilizzo del gripper, consiste nell'afferrare un oggetto e spostarlo dalla posizione attuale in una posizione vicina, ruotandolo di 90 gradi.

L'implementazione di questa prova consiste nel muovere il robot in una posizione iniziale, corrispondente alla posizione sopra all'oggetto da afferrare, con il metodo *MoveLinearTo*, aprire la pinza con il metodo *open*, far scendere il robot con il metodo *MoveRelative*, chiudere la pinza con il metodo *close*, far risalire il robot fino alla posizione iniziale con il metodo *MoveRelative*, muovere il robot nella posizione di destinazione con il metodo *MoveLinearTo* e riaprire la pinza. Infine, la connessione con il server viene chiusa. Per questioni di sicurezza e per un'analisi più precisa, la velocità è stata impostata al 30% della velocità massima. In fig. 5.3 viene riportata una sequenza di immagini del moto del robot.

Rispetto alla prova precedente, viene inclusa anche la libreria per il controllo della pinza mediante l'istruzione:

```
#include "Gripper.h"
```

posta nell'intestazione del programma. Inoltre, viene modificata la funzione *motion*, mentre il *main* e la funzione *getPos* rimangono identiche all'esempio precedente.

```
void * motion (void * arg) {
  Gripper pinza("COM2", 9600);
  r.SetSpeed("30");
  r.MoveLinearTo("inizio2.txt");
  pinza.open(60,50); //apre
  r.MoveRelative(0,0,-255); //scende
  r.Sleep(1);
  pinza.close(40,30,1); //chiude
  r.Sleep(2);
  r.MoveRelative(0,0,255); //sale
  Point6d p = r.GetCurrentPosition();
  p.setX(p.getX()-400);
  p.setY(p.getY()+400);
```

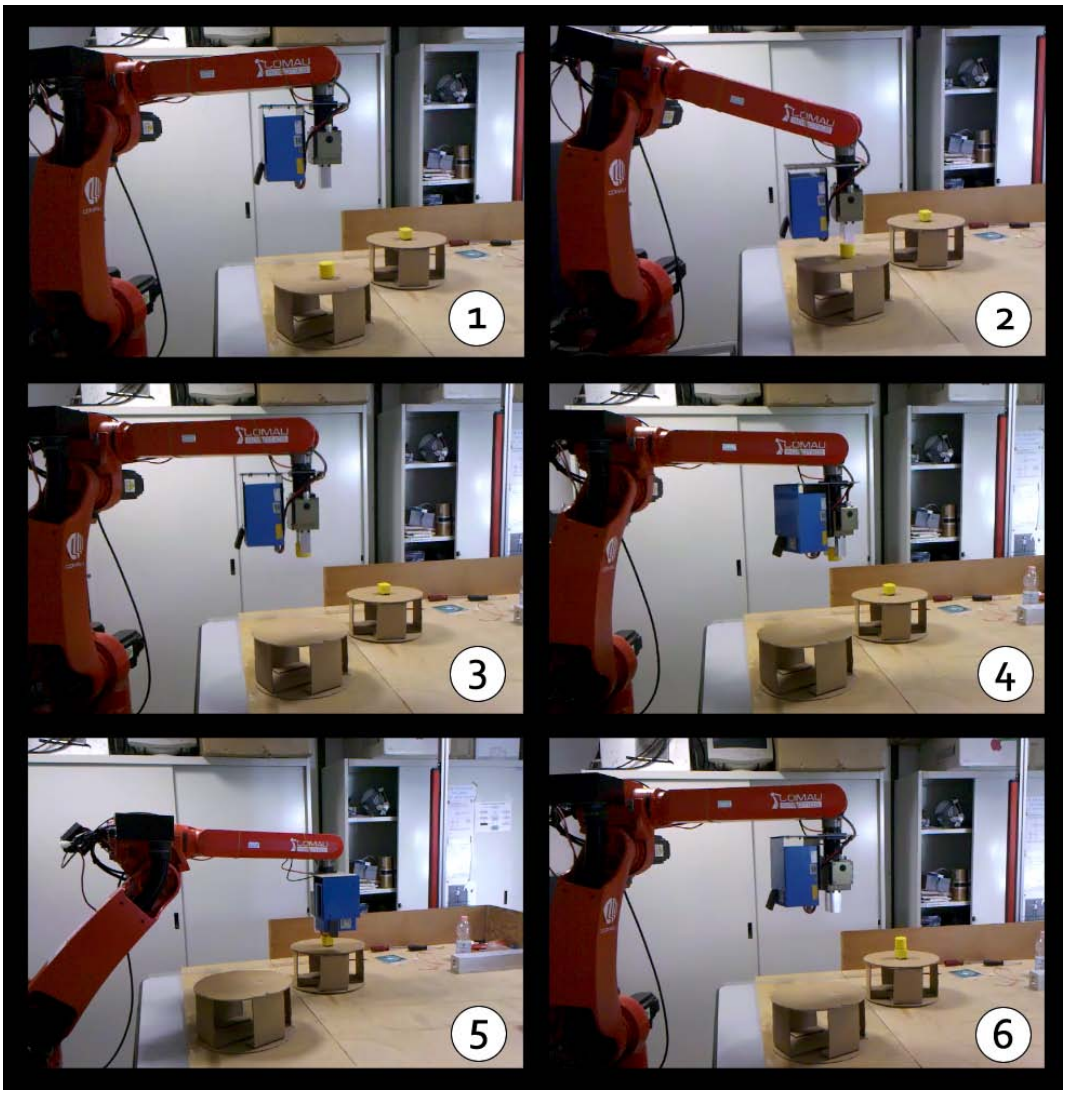

Figura 5.3: Esecuzione della seconda prova, con spostamento di un oggetto

```
  p.setZ(p.getZ()-210);
  p.setC(p.getC()-90);
  r.MoveLinearTo(p);
  r.Sleep(5);
  pinza.open(60,50); //apre
  r.Sleep(3);
  r.MoveRelative(0,0,50);
  r.Sleep(1);
  pinza.close(10,50,1); //chiude
  r.MoveLinearTo("inizio2.txt");
  r.Sleep(5);
  stop=true;
  r.Sleep(1);
  r.CloseConnection();
  system("PAUSE");
  return NULL;
}
```

La funzione *motion* per prima cosa crea un oggetto *pinza* di classe *Gripper* che consente poi di inviare comandi alla pinza. Inoltre, imposta la velocità del robot al 30% della velocità massima e muove il robot nella posizione iniziale, memorizzata nel file di testo *inizio2.txt*, che in questo caso è il punto $300, 800, 850, 90, 180, 90$. Successivamente, con l'istruzione nella riga 5, la pinza viene aperta e, con la riga 6, il robot si sposta in basso di 255 mm per arrivare ad afferrare l'oggetto. Dopo un secondo di pausa, che permette al robot di terminare il moto, la pinza viene chiusa e, dopo altri due secondi, il robot risale di 255 mm con l'oggetto tra le dita della pinza. Successivamente, viene calcolato il punto nel quale andare a depositare l'oggetto: a partire dalla posizione corrente del robot, memorizzata nell'oggetto p di classe *Point6d*, si sposta il punto di -400 mm lungo l'asse X, di 400 mm lungo l'asse Y, di -210 mm lungo l'asse Z e si fa compiere una rotazione al braccio di -90°; infine, si muove il robot verso questo punto mediante l'istruzione *r.MoveLinearTo(p)*. Arrivati a destinazione, la pinza viene riaperta, per dare modo all'oggetto di posarsi sul supporto, il robot si rialza di $50mm$ e torna alla posizione iniziale chiudendo la

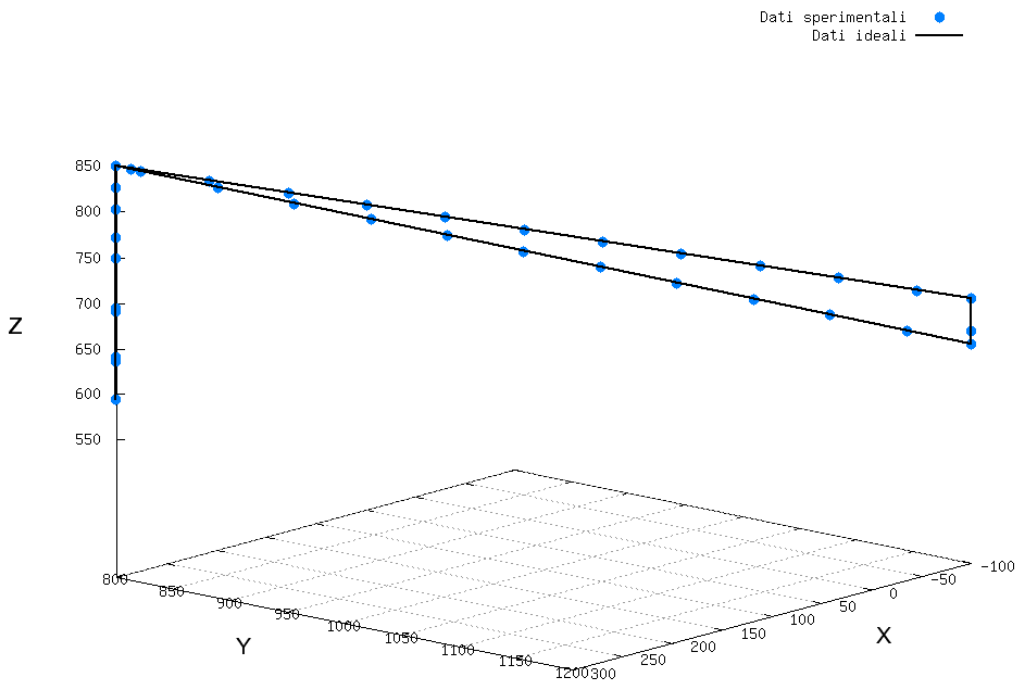

Figura 5.4: Grafico del moto del robot durante la seconda prova nelle tre dimensioni X, Y, Z (realizzato con gnuplot)

pinza. Poi, il programma termina, chiudendo la connessione con il robot.

Il risultato del moto dell'estremità del robot è visibile in fig. 5.4: la linea continua nera mostra l'andamento ideale del moto, mentre i puntini blu mostrano una serie di *campioni* delle posizioni realmente attraversate dal robot. La posizione iniziale corrisponde al punto di coordinate $300, 800, 1100$.

5.3 Spostamento di una bottiglia di plastica

La terza e ultima prova consiste nell'afferrare un oggetto, nello specifico una bottiglietta di acqua, non più dall'alto come nella prova precedente, ma di lato. Una volta afferrata, la bottiglietta viene spostata seguendo una traiettoria triangolare e, successivamente, viene ruotata, simulando di versare acqua in un bicchiere. Infine,

la bottiglietta viene rimessa nella posizione di partenza. Il tutto viene svolto ad una velocità pari al 20% della velocità massima. In fig. 5.5 viene riportata una sequenza di immagini del moto del robot.

L'implementazione di questo *task* è molto simile alla precedente: ciò che cambia è, come prima, la funzione *motion*. Questa volta, per buona parte dei moti, viene utilizzata la funzione di moto lungo una traiettoria precalcolata memorizzata in un file di testo, *traiettoria.txt*.

```
void * motion (void * arg) {
  Gripper pinza("COM2", 9600);
  r.SetSpeed("20");
  r.MoveLinearTo("inizio3.txt");
  pinza.open(50,50); //apre
  r.MoveRelative(0,0,-100); //scende
  r.Sleep(1);
  r.MoveRelative(0,90,0); //avanza
  r.Sleep(1);
  pinza.close(40,30,0.5); //chiude
  r.Sleep(2);
  r.MoveRelative(0,0,100); //sale
  r.MoveTrajectory("traiettoria.txt");
  r.Sleep(5);
  r.MoveLinearTo("inizio3.txt");
  r.MoveRelative(0,0,-100); //scende
  r.Sleep(1);
  r.MoveRelative(0,90,0); //avanza
  r.Sleep(1);
  pinza.open(60,60);
  r.Sleep(3);
  r.MoveLinearTo("inizio3.txt");
  r.Sleep(2);
  pinza.close(30,50,1); //chiude
  stop=true;
```

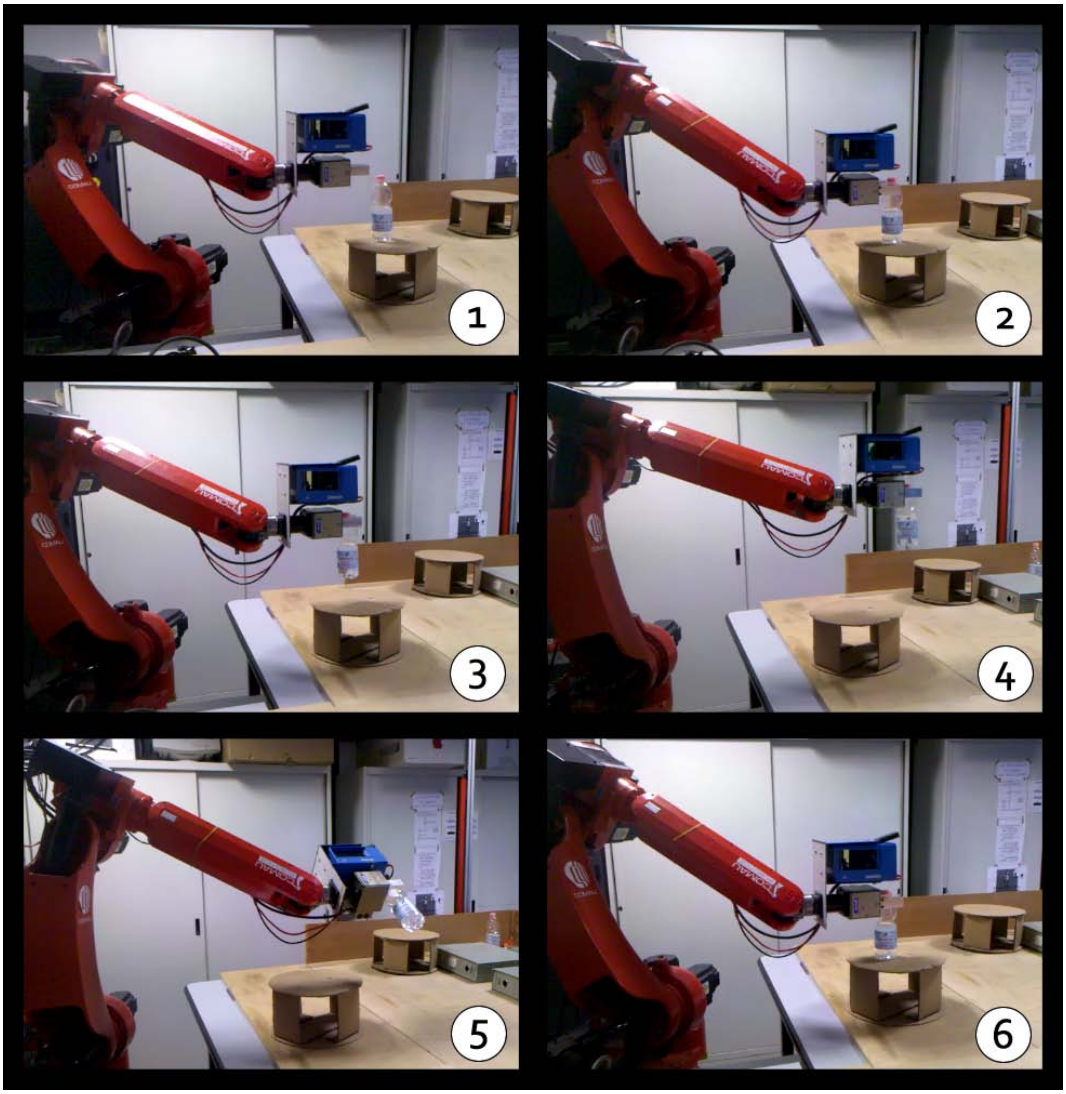

Figura 5.5: Esecuzione della terza prova, con spostamento di una bottiglia di plastica

```
  r.CloseConnection();
  return NULL;
}
```

Dopo le istruzioni di apertura della connessione con la pinza e di impostazione della velocità, il robot viene spostato nella posizione iniziale memorizzata nel file di testo *inizio3.txt* e corrispondente al punto di coordinate $250, 700, 850, 90, 90, -90$.

Successivamente, la pinza viene aperta e il robot viene spostato in basso, in corrispondenza del collo della bottiglietta di plastica, e infine in avanti per poterla afferrare. La pinza viene quindi chiusa e il robot viene fatto risalire di 100 mm. In seguito, viene fatto muovere il robot lungo la traiettoria di punti formante un triangolo mediante l'istruzione *r.MoveTrajectory(traiettoria.txt)*. Dopo alcuni secondi di pausa, il robot viene riportato alla posizione iniziale e, da qui, riportato nella posizione in cui ha preso la bottiglia per permettergli di rilasciarla aprendo la pinza. Infine, la pinza viene chiusa, la connessione al server chiusa e il programma termina.

Con questa prova, si sono testate le funzioni di moto lungo una traiettoria e di rotazione della pinza. La particolarità più complessa di questa prova risulta essere la modifica dell'orientazione della pinza al fine di afferrare la bottiglia come la prenderebbe un uomo per versare da bere. Una volta posizionata la pinza in modo corretto, le azioni svolte sono analoghe a quelle della prova precedente.

In questo caso, a differenza dei precedenti, si può notare un errore nella traiettoria, dato dai due puntini blu fuori dalla traiettoria teorica (linea nera). Tuttavia, se si analizza la scala, si nota che si tratta di errori dell'ordine del millesimo di millimetro, assolutamente ininfluenti nel moto del robot. Pertanto, si può affermare che anche in questa prova il robot ha dimostrato un'ottima precisione e che la libreria descritta in questa tesi funziona a dovere.

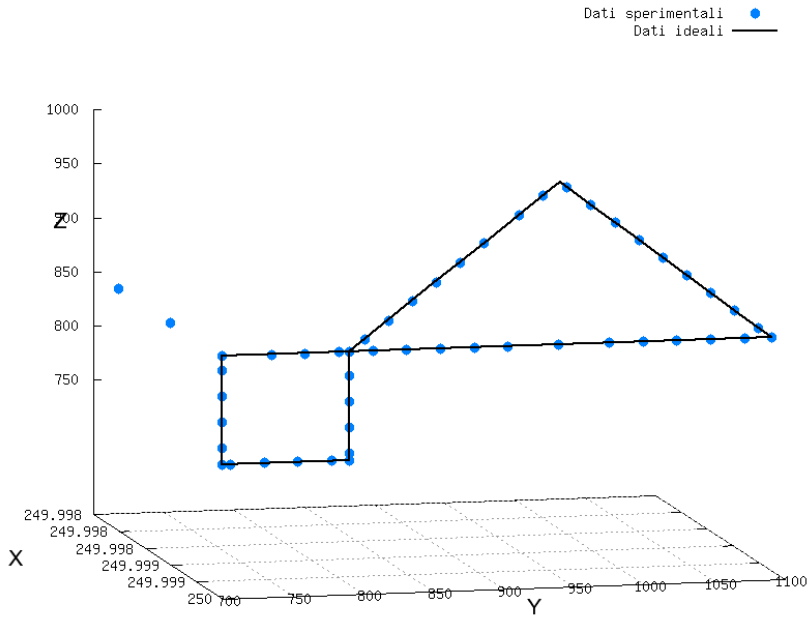

Figura 5.6: Grafico del moto del robot durante la terza prova nelle tre dimensioni X, Y, Z (realizzato con gnuplot)

Capitolo 6

Conclusioni

Il lavoro che ha portato alla realizzazione di questa tesi ha riguardato inizialmente lo studio dei manuali del linguaggio di programmazione PDL2, al fine di raccogliere le competenze necessarie per la scrittura di programmi che facessero muovere il robot. Inoltre, ampio spazio è stato dedicato allo studio del gripper Schunk Pg70.

Lo sviluppo della libreria si è articolato in vari passi:

1. Codifica del programma PDL2 ServerComau
2. Codifica della classe C++ RobotComau
3. Codifica della classe C++ Gripper
4. Codifica del programma PDL2 CheckPosition
5. Codifica del programma PDL2 CheckError

A questo, è seguita la fase di test dei vari componenti e le prove sperimentali.

I risultati ottenuti dalle prove sperimentali sono stati soddisfacenti: la precisione del manipolatore e la semplicità di utilizzo hanno consentito fin da subito l'utilizzo della libreria da parte di altri utenti.

Il risultato di questo lavoro di tesi, ovvero la libreria per la programmazione del robot manipolatore Comau Smart SiX e della pinza Schunk Pg70, consente molte applicazioni future. Infatti, questa libreria semplifica molto la codifica di programmi per il controllo del robot, in quanto scritta in un linguaggio standard come il C++.

Nonostante questo, il progetto presenta ancora molti margini di sviluppo e miglioramento. Di seguito ne vengono riportati alcuni:

- Un primo sviluppo che si potrebbe rendere necessario nell'immediato futuro è quello di adattare questa libreria al sistema operativo opensource Linux, in modo da consentire l'utilizzo semplice del robot e della pinza anche a dispositivi con quel sistema operativo installato.
- Al fine di misurare più accuratamente la precisione del moto del robot, si potrebbero effettuare prove sperimentali ad una velocità più elevata.
- Utilizzando questa libreria, si potranno scrivere programmi con evolute interfacce grafiche per il controllo del robot in tempo reale. Ogni PC avrà, quindi, a disposizione una sorta di telecomando virtuale per pilotare il robot e la pinza.
- Vista la diffusione sempre più ampia di smartphone e palmari dalle ottime prestazioni, un'idea sarebbe quella di adattare la libreria ai sistemi operativi di questi dispositivi, come Android o iPhone, per permettere un controllo del robot manipolatore per mezzo dei dispositivi mobili.
- Il robot potrebbe essere collegato ad un pianificatore di traiettorie per l'esecuzione di compiti complessi che richiedono, ad esempio, di evitare ostacoli nell'ambiente circostante. Per questo, occorrerebbe integrare la libreria con il pianificatore di traiettorie.

Bibliografia

[1] Marco Tarasconi. *Sviluppo di un driver in ambiente Orchestra per il controllo di un gripper robotico*. Università di Parma, 2009.

[2] COMAU Robotics. *PDL2 Programming Language Manual - System Software Rel. 3.2x*. COMAU Robotics, June 2008.

[3] COMAU Robotics. *C4G Programmazione del Movimento - Software di Sistema Rel. 3.2x*. COMAU Robotics, June 2008.

[4] Corrado Guarino Lo Bianco. *Cinematica dei manipolatori*. Pitagora, 2004.

[5] http://www.pdl2.info.

[6] http://miktex.org.

[7] http://www.guit.sssup.it.

[8] http://www.gnuplot.info.