

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea in Ingegneria Informatica

Sviluppo di un'interfaccia grafica per realizzazione di traiettorie di quadrirotori

Relatore:
Chiar.mo Prof.
Giuseppe Notarstefano

Correlatori:
Ing. Lorenzo Picherri

Candidato:
Lorenzo Venerandi

Sessione 3
Anno Accademico 2021-2022

Abstract

Negli ultimi decenni si è assistito ad un enorme aumento di interesse nello sviluppo di UAV (*Unmanned Aerial Vehicles*). In particolare la ricerca si è indirizzata nei quadrirotori di piccole e medie dimensioni, dato l'ampio range di scenari in cui possono trovare applicazione: un esempio è quello del progetto Crazyflie, un nanodrone sviluppato da Bitcraze. Esso è uno dei principali soggetti di questo elaborato, nel quale verranno approfonditi gli strumenti, le architetture ed i software utilizzati per il controllo dei quadrirotori, partendo da ROS 2 (*Robot Operating System*) fino ad arrivare all'interfacciamento fra esso e i vari dispositivi. Il focus principale della tesi è lo sviluppo di un'applicazione dotata di interfaccia grafica che consenta di pianificare traiettorie ed analizzarne le caratteristiche, come velocità ed accelerazione. Verranno approfonditi e descritti alcuni passaggi necessari per lo sviluppo di un software, partendo dallo studio ed analisi dei requisiti fino ad arrivare alla scelta delle tecnologie ed alla progettazione ed implementazione dell'applicazione. L'ultimo capitolo dell'elaborato si concentrerà sulla parte sperimentale fornendo una descrizione della strumentazione utilizzata durante i test, effettuati prima con il simulatore e poi con il quadrirotore.

Indice

1 Motivazioni	3
2 Introduzione alle tecnologie	4
2.1 Struttura del progetto	4
2.2 Python	5
2.2.1 Introduzione	5
2.2.2 Caratteristiche	6
2.3 ROS	8
2.3.1 Introduzione	8
2.3.2 Architettura di ROS	9
2.4 ROS 2	11
2.4.1 Caratteristiche principali	11
2.4.2 Architettura	12
2.5 Dispositivi ed hardware	12
2.5.1 Quadrirotore	12
3 Design applicazione	14
3.1 Introduzione	14
3.2 Analisi dei requisiti	14
3.2.1 Raccolta dei requisiti	14
3.2.2 Tabella dei requisiti	15
3.3 Casi d'uso	17
3.3.1 Scenari	18
3.4 Diagramma delle classi	20
4 Realizzazione applicazione	24
4.1 Progettazione Architettonale	24
4.1.1 Requisiti non funzionali	24
4.1.2 Scelta tecnologie	25
4.1.3 Persistenza	27
4.2 Implementazione	28
4.2.1 Interfaccia grafica	29
4.2.2 Interpolazione di una traiettoria	33
4.2.3 Calcolo spline di una traiettoria	35

4.2.4	Interfacciamento con ROS	37
5	Sperimentazione	41
5.1	Strumenti utilizzati	41
5.1.1	Vicon	41
5.1.2	Crazyflie	42
5.2	Simulazione percorrenza traiettoria	45
5.2.1	Disegno traiettoria	45
5.2.2	Analisi spline	45
5.2.3	Simulazione e volo	46
	Conclusioni	48
	Elenco delle figure	49
	Elenco delle tabelle	50
	Bibliografia	51
	Ringraziamenti	52

1 Motivazioni

Le motivazioni principali che hanno ispirato questo progetto nascono dai problemi riscontrati nell'utilizzo di sistemi di controllo per quadrirotori: questi molto spesso, soprattutto in ambito di ricerca, risultano poco immediati in quanto è necessario agire sul codice e lanciare degli script. Si riscontra infatti la necessità di un'implementazione più efficace nella definizione di traiettorie e nell'impartizione di comandi rapidi, come l'hovering¹ e l'atterraggio. L'obiettivo principale di questa tesi è progettare e sviluppare un'applicazione dotata di interfaccia grafica che faciliti il controllo dei quadrirotori, integrandola con l'architettura messa a disposizione dal laboratorio.

Durante il mio percorso di formazione ho svolto l'attività di tirocinio presso il laboratorio di automazione CASY (*Center for Research on Complex Automated Systems*), dove ho potuto studiare ed approfondire gli strumenti utilizzati per il controllo di robot e quadrirotori. Questa esperienza ha facilitato notevolmente il design e lo sviluppo dell'applicativo e mi ha fornito conoscenze che mi hanno aiutato nella stesura di questo elaborato.

¹Posizione in volo in cui viene mantenuta la stessa altezza da parte di un quadrirotore

2 Introduzione alle tecnologie

In questa sezione verranno introdotte le tecnologie principali utilizzate durante lo sviluppo del progetto, così come il loro ruolo all'interno della struttura dello stesso.

2.1 Struttura del progetto

L'architettura del sistema è strutturata su tre livelli:

- **Crazydraw**, l'applicazione dotata di interfaccia grafica e scritta utilizzando il codice Python [7]. È l'obiettivo centrale della tesi e viene utilizzata per:
 - disegnare, salvare e gestire traiettorie
 - interpolare la traiettoria desiderata per generare una curva polinomiale di terzo grado, così da poter ricavare posizione, velocità ed accelerazione rispetto ad un istante t
 - impartire comandi base al quadrirotore, come decollo, hover ed atterraggio e farlo muovere seguendo una traiettoria
- **ROS 2** [6], un sistema di software e librerie utilizzato per il controllo di robot. Quest'ultimo viene utilizzato per mettere in comunicazione l'applicazione con il controllore dei quadrirotori, il simulatore e le telecamere.
- I dispositivi fisici utilizzati durante il testing e la loro interfaccia con il sistema di ROS 2. Essi verranno descritti ed approfonditi durante la fase di Sperimentazione (capitolo 5).

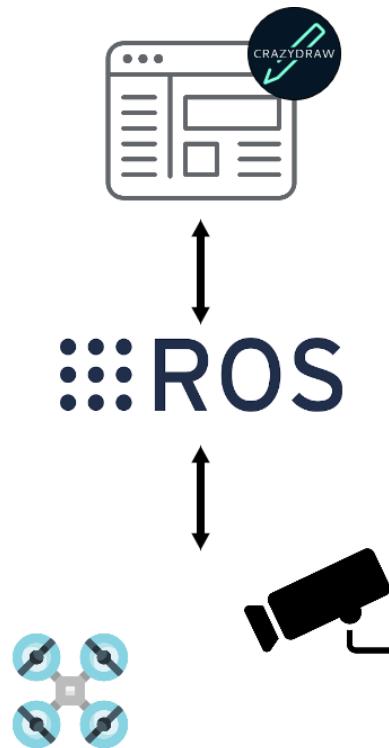


Figura 2.1: Struttura del progetto

In seguito verranno approfondite le due principali tecnologie utilizzate durante lo sviluppo del progetto, cioè il linguaggio Python ed il sistema ROS.

2.2 Python



Figura 2.2: Logo di Python

2.2.1 Introduzione

Python è un linguaggio di alto livello rilasciato per la prima volta nel 1991 dal suo creatore Guido van Rossum. Lo sviluppo di Python attualmente viene gestito dall'organizzazione *Python Software Foundation*.

Python segue i principi dell'open source: il codice è disponibile online ed è liberamente modificabile e riutilizzabile. Sia l'interprete di Python che la sua documentazione sono gratuiti, così come la licenza per poterlo utilizzare nelle proprie applicazioni.

2.2.2 Caratteristiche

Python è un linguaggio che supporta diversi paradigmi di programmazione, come quello object-oriented (con supporto ad ereditarietà multipla) e quello imperativo.

Uno dei suoi punti di forza è sicuramente l'essere un linguaggio comodo da utilizzare e molto semplice da imparare.

Python è caratterizzato infatti da una sintassi molto minimale dotata di:

- costrutti semplici
- semantica pulita (è per esempio assente il classico ";" utilizzato nei linguaggi come C [5] e Java [2] per terminare un'istruzione)
- blocchi logici definiti soltanto dall'allineamento del codice, invece che da parentesi graffe

Questa caratteristica verrà evidenziata in seguito, durante l'analisi delle prestazioni di uno script in Python ed in C. Un altro punto di forza di Python è la sua estrema portabilità, data dal fatto di essere un linguaggio semi-interpretato. Infatti, a differenza di C, non è dotato di un compilatore e quindi non crea un file eseguibile.

Python utilizza un interprete che, a tempo di esecuzione, traduce il codice in istruzioni in linguaggio macchina. Questo consente al codice di essere compatibile universalmente, a patto che sia installato l'interprete Python giusto. Tutto questo avviene però a scapito delle prestazioni, infatti la velocità di esecuzione di un codice interpretato è molto inferiore a quella di un programma eseguibile già compilato.

Questa differenza viene evidenziata nel seguente esempio, in cui viene eseguito un semplice algoritmo per un certo numero di volte. Viene confrontato il tempo di esecuzione dello stesso script scritto in C ed in Python.

```
int main(void){
int sum = 0;
int n_iterations = 10000;
long unsigned int start_time= micros();
for(int i=0;i<n_iterations;i++)
    sum = sum + sum%(i+1);
printf("Finish time is %f
→ milliseconds\n",
→ (float)(micros()-start_time)/1000);
return 0;
}
```

```
start_time = time.time()
n_iterations = 10000
sum = 0
for i in range(n_iterations):
    sum += sum % (i+1)

print(f"Finish time
→ {(time.time()-start_time)*1000}
→ milliseconds")
```

Dopo aver eseguito entrambi i programmi più volte (per avere una stima realistica del tempo di esecuzione indipendente dal sistema operativo in cui viene eseguito), sono stati ottenuti i seguenti valori:

- **Programma in C** $\simeq 0.18$ millisecondi
- **Programma in Python** $\simeq 1.5$ millisecondi

Come si può notare dai risultati la differenza di prestazioni è significativa, quasi di un ordine di grandezza. Per far fronte a questo problema, Python offre un ampio catalogo di librerie già pre-compilate (scritte in linguaggio C), tra le quali *numpy* [3] e *tensorflow* [1]. Il supporto all'open-source ha contribuito all'enorme diffusione di Python in tutto il mondo: al giorno d'oggi risulta infatti il linguaggio più utilizzato, anche da aziende come Google, Microsoft e molte altre.

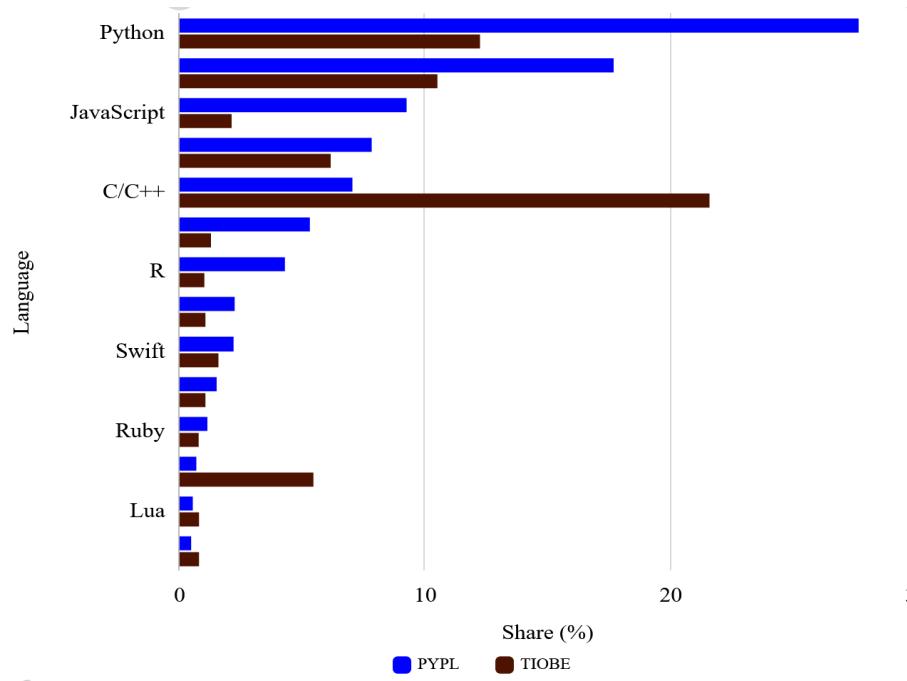


Figura 2.3: Top Computer Languages (June 2022)

Il grafico è stato tracciato dal sito *StatisticTimes*¹ seguendo due indici:

- TIOBE – calcolato dalla omonima azienda in base al numero di ricerche del linguaggio in questione e prendendo in considerazione i dati di 25 motori di ricerca differenti
- PYPL² – calcolato in base al numero di ricerche di tutorial del linguaggio preso in considerazione sul motore di ricerca Google

¹<https://statisticstimes.com/tech/top-computer-languages.php>

²Popularity of Programming Language

2.3 ROS

2.3.1 Introduzione

ROS è un insieme di software e di librerie utilizzato per la gestione e il controllo di robot. È l'acronimo di *Robot Operating System*, esso può essere infatti definito un meta-sistema operativo.

ROS fornisce infatti i servizi che vengono comunemente associati ad un sistema operativo, come:

- astrazione dell'hardware
- controllo dei dispositivi a basso livello e gestione dei driver
- gestione dei processi e dei canali di comunicazione fra di essi
- management dei package

ROS è un sistema Open-Source e modulare sviluppato in C++ [4]; ciò gli conferisce notevoli capacità prestazionali, indispensabili nell'ambito della robotica data l'esigenza di processare notevoli quantità di dati in tempo reale.

ROS è un sistema estendibile e programmabile: è possibile aggiungere dei componenti programmati in C++ oppure in Python.

Questo facilita notevolmente il lavoro degli sviluppatori e consente una rapida diffusione di librerie personalizzate, garantisce una maggiore interoperabilità tra sistemi differenti e la creazione di implementazioni ad hoc per un dispositivo specifico.

Per quanto riguarda la compilazione di programmi scritti in C++ ROS utilizza CMake³, un software multi-piattaforma utilizzato per facilitare la compilazione dei file. Per utilizzarlo è necessario definire dipendenze e file eseguibili all'interno del file *CMakeLists.txt*. Una volta fatto questo è sufficiente eseguire i seguenti comandi:

```
mkdir build  
cd build  
cmake ..  
make
```

In questo modo verrà creata una directory `build` e CMake si occuperà di compilare tutti i file ed inserirli in essa. Questo tool risulta molto comodo in progetti di grandi dimensioni come ROS in cui sono presenti numerosi file, la cui gestione risulterebbe notevolmente complessa in assenza di un build-manager come CMake.

³Hoffman, W., & Martin, K. (2003). The CMake Build Manager. Dr. Dobb's Journal: Software Tools for the Professional Programmer, 28(1), 40-43.

2.3.2 Architettura di ROS

In questa sezione verranno approfonditi i componenti fondamentali di ROS, partendo dal *file system* fino ad arrivare alla gestione dei processi e delle *pipe*.

File System

Il file system di ROS è basato sui *Package*, essenzialmente una directory contenente i file che costituiscono un modulo. Un package può contenere i seguenti elementi:

- **msg/** – cartella in cui vengono memorizzati i messaggi definiti dal package
- **svc/** – cartella in cui vengono descritti i servizi
- **scripts/** – dove sono situati i codici eseguibili relativi al modulo
- **CMakeLists.txt** – file utilizzato da CMake per compilare il codice scritto in C++.

Un altro elemento essenziale per la composizione del package è il manifest (file `package.xml`), che contiene informazioni come:

- Nome – nome del package
- Versione – versione attuale del package
- Descrizione – descrizione del package
- Maintainer – utente che si occupa di mantenere ed aggiornare il package
- Licenza – licenza attribuita all'utilizzo pubblico o meno del package
- Dipendenze esterne – librerie e package necessari al funzionamento degli script contenuti all'interno del package

Un altro elemento del file system è la Repository, essa è utilizzata per la pubblicazione di moduli e può includere uno o più package; quelli inclusi nella Repository che hanno la stessa versione possono essere rilasciati insieme.

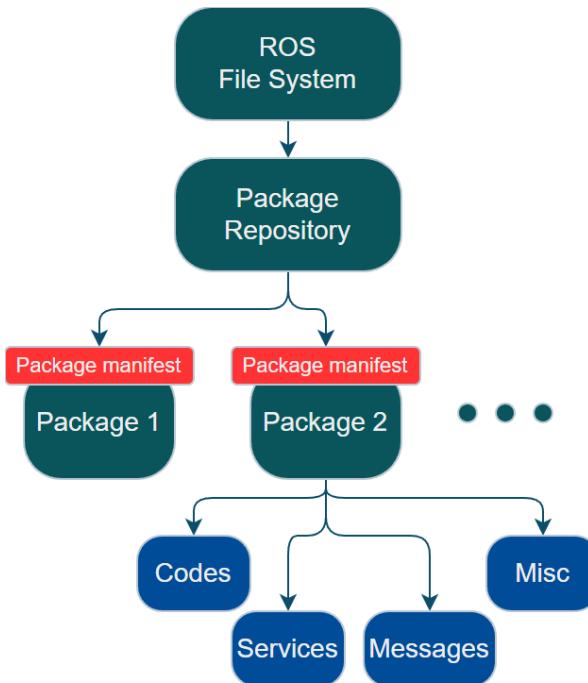


Figura 2.4: Struttura del File System di ROS

Gestione processi

In letteratura informatica un processo è un programma che viene eseguito e caricato in memoria. In particolare quello definito “pesante”, oltre ad allocare lo spazio necessario all'esecuzione (come le memorie Stack ed Heap) copia il codice da eseguire e le variabili utilizzate all'interno di esso. Questo tipo di modello viene adottato da ROS, dove il processo prende il nome di Nodo.

Un **Nodo** infatti è un processo, compilato ed eseguito partendo da uno script contenuto in un package, che può eseguire operazioni di calcolo e comunicare con altri processi e dispositivi esterni. Solitamente il processo di controllo per un robot comprende numerosi Nodi. L'elemento che si occupa di gestire i Nodi e le risorse è **roscore**, esso stesso composto da un insieme di Nodi. Roscore si occupa anche della creazione di un nameservice detto Master, cioè un servizio utilizzato per associare ad ogni risorsa (Nodi, Topics, Services etc.) un nome. In questo modo ogni Nodo può accedere ad ogni risorsa registrata al sistema di ROS.

Comunicazione tra i Nodi

Uno dei punti forti di ROS è proprio la facilità con cui i Nodi possono interoperare e scambiarsi messaggi, questo principalmente può avvenire in due modi:

- Utilizzando un **Topic** – un bus dati (molto simile ad una pipe⁴ di un comune sistema operativo) a cui viene associato un nome. Un Topic viene registrato al Master nel momento in cui un Nodo si iscrive a tale Topic come *Publisher*. Un Nodo Publisher può aggiungere dei messaggi ad un Topic. Se un Nodo ha intenzione di leggere le informazioni contenute dal Topic deve iscriversi ad esso in modalità *Subscriber*.
Generalmente ad ogni Topic è associato un tipo di **messaggio**. Il messaggio è una struttura dati che viene utilizzata dai Nodi per comunicare attraverso un Topic. ROS mette a disposizione dei messaggi predefiniti, ma è possibile definirne di nuovi partendo dai classici tipi primitivi (int, string, float etc.).
- Utilizzando un **Service** – un servizio è uno strumento più versatile di un topic nel caso si voglia attuare un meccanismo di domanda-risposta tra due Nodi. Esso infatti agisce come una chiamata a procedura remota (RPC).

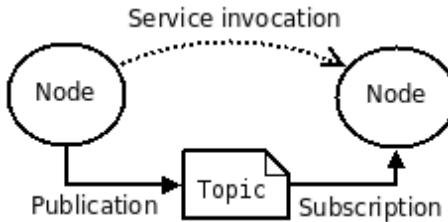


Figura 2.5: Comunicazione fra più Nodi

2.4 ROS 2

2.4.1 Caratteristiche principali

ROS 2 è l'update di ROS rilasciato negli ultimi anni e tra i principali cambiamenti possiamo trovare:

- Aggiornato lo standard di C++ utilizzato per la scrittura del sistema (da C++03 a C++11)
- Maggior supporto a Python: infatti non è più necessario specificare gli script di Python da includere nel package, il tool di compilazione ora si riferirà direttamente ai Python package definiti dai file setup.py.
- Possibilità di definire i nodi estendendo una classe, sia in C++ che in Python.
- File di launch scritti in Python (invece che in XML).

⁴Canale di comunicazione fra processi utilizzato comunemente nei sistemi Unix

2.4.2 Architettura

Il cambiamento più significativo sta nella modifica dell'architettura alla base del funzionamento di ROS:

- **ROS 1** utilizza un modello client-server: è presente il nodo principale *roscore* che funge da server e fornisce una serie di servizi agli altri nodi, come già spiegato nella sezione 2.3.2.
- **ROS 2** adotta un modello distribuito, infatti non è presente alcun nodo principale ed ogni servizio viene gestito dai singoli nodi.

Sebbene il modello distribuito adottato da ROS 2 possa risultare più complesso consente di risolvere molte problematiche, tra cui quella del *single point of failure*.

Per *single point of failure* si intende il caso in cui il funzionamento complessivo di un sistema è vincolato da quello di un componente specifico. Questo tipo di problematica è comune del modello client-server adottato da ROS, in quanto il nodo *roscore* ha un ruolo determinante al funzionamento dell'intero sistema.

ROS 2 utilizza invece un modello distribuito, in cui la comunicazione fra i nodi è peer-to-peer, cioè un tipo di comunicazione in cui tutti i partecipanti hanno lo stesso ruolo; inoltre la risoluzione dei nomi di Topic e Services è affidata al servizio DDS (*Data Distribution Service*), anch'esso distribuito fra tutti i nodi.

2.5 Dispositivi ed hardware

Come già introdotto in precedenza (sezione 2.1) il terzo livello del progetto è costituito dai dispositivi fisici utilizzati e dal loro link con il sistema di ROS 2. Questi principalmente sono di due tipi:

- Quadrirotori e strumenti utilizzati per comunicare con essi
- Sistema di Motion Capture utilizzato per individuare la posizione dei quadrirotori

Questi dispositivi verranno approfonditi in fase di Sperimentazione (5.1).

2.5.1 Quadrirotore

Per quadrirotore (o quadricottero) si intende un veivolo che viene sollevato e messo in movimento da quattro rotorì. Si differenzia dal comune elicottero in quanto le pale sono dette "a passo fisso", infatti il loro angolo di attacco non varia durante il volo.

Il cambio di direzione, la velocità e l'altezza vengono modificati dal controllore agendo sulla velocità di rotazione dei singoli rotorì.

Modellazione del quadrirotore

Si definisce f_i la forza prodotta da ogni rotore; essa è proporzionale al quadrato della sua velocità angolare $f_i = k_f \cdot \omega_i^2$.

Inoltre, dato che i rotori 2 e 4 ruotano in senso orario (mentre i rotori 1 e 3 in senso anti-orario) la coppia τ_i viene calcolata come $\tau_i = -k_\tau \cdot f_i$ per i primi due e $\tau_i = k_\tau \cdot f_i$ per gli ultimi due.

È quindi possibile determinare il valore della forza risultante f , così come quello delle coppie agenti in ogni asse:

$$\begin{aligned} f &= f_1 + f_2 + f_3 + f_4 \\ \tau_x &= \frac{\sqrt{2}}{2} \cdot l(-f_1 - f_2 + f_3 - f_4) \\ \tau_y &= \frac{\sqrt{2}}{2} l(-f_1 + f_2 + f_3 - f_4) \\ \tau_z &= k_\tau(f_1 - f_2 + f_3 - f_4) \end{aligned}$$

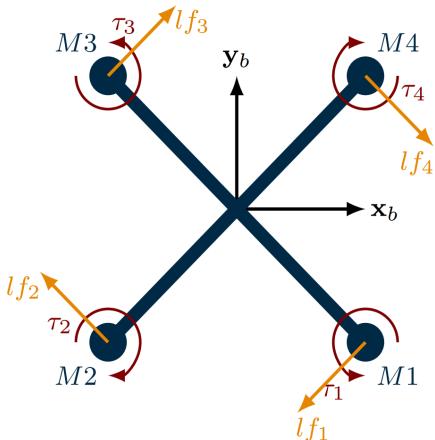


Figura 2.6: Forze generate da un quadrirotore

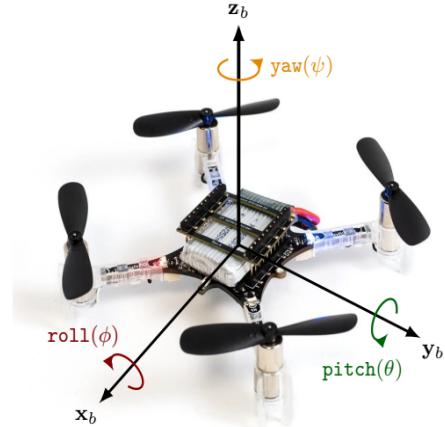


Figura 2.7: Roll, pitch e yaw di un quadrirotore

Il modello dinamico di un quadrirotore è definito come segue

$$\dot{p} = v$$

$$m\dot{v} = -mge_3 + fR(\Phi)e_3$$

$$\dot{\Phi} = J(\Phi)\omega$$

$$I\dot{\omega} = -S(\omega)I\omega + \tau$$

dove g è la costante di accelerazione gravitazionale, m è la massa del quadrirotore, $e_3 = [0 \ 0 \ 1]$, $J(\Phi)$ è la matrice Jacobiana ed $I = \text{diag}(i_x, i_y, i_z)$ la matrice di inerzia.

3 Design applicazione

3.1 Introduzione

In questo capitolo verrà approfondito il processo di design dell'applicazione, partendo dall'analisi dei requisiti fino alla scelta dell'architettura e definizione di pattern e classi. Questo progetto è stato realizzato seguendo i modelli e principi dell'ingegneria del software, tra i quali:

- suddivisione del progetto in sotto-progetti (moduli), così da avere componenti facilmente gestibili
- utilizzo di componenti già pronti
- standardizzazione dei componenti

Questo tipo di approccio, benché allunghi il processo di design, conferisce all'applicazione modularità, affidabilità e mantenibilità, caratteristiche essenziali nello sviluppo moderno di software.

3.2 Analisi dei requisiti

Il primo passo nel processo di sviluppo è proprio la raccolta ed analisi dei requisiti; questi vengono generalmente divisi in due tipi:

- **Requisiti funzionali (RF)** – quei requisiti che riguardano un comportamento del sistema (esempio: deve essere possibile memorizzare una traiettoria).
- **Requisiti non funzionali (RNF)** – quei requisiti che costituiscono un vincolo del sistema ma non influenzano necessariamente il comportamento dello stesso (esempio: interfaccia grafica semplice ed intuitiva).

3.2.1 Raccolta dei requisiti

Innanzitutto è necessario stilare l'elenco dei requisiti che l'applicazione dovrà soddisfare:

- Interfaccia grafica per pianificare le traiettorie

- Interfaccia semplice e comoda
- Gestore delle traiettorie salvate
- Conversione dei waypoint¹ in polinomiali di almeno terzo grado
- Conversione di traiettoria 2D in traiettoria 3D
- Processi di conversione efficienti
- Comunicazione della traiettoria 3D a ROS 2 (Foxy)
- Integrazione con sistema Vicon, quadrirotori Crazyflie e simulatore

3.2.2 Tabella dei requisiti

Basandoci sulla raccolta dei requisiti dividiamo i requisiti in funzionali e non funzionali. I requisiti funzionali saranno poi utilizzati immediatamente nella definizione dei casi d'uso, quelli non funzionali saranno tenuti in considerazione in fase di realizzazione.

Requisiti non Funzionali	
Tipo	Requisito
R1NF	Interfaccia grafica multi-window semplice da utilizzare
R2NF	Conversione rapida da waypoint a polinomiale
R3NF	Comandi rapidi per il quadrirotore immediati da utilizzare
R4NF	Gestore delle traiettorie comodo
R5NF	Editor grafico delle impostazioni

Tabella 3.1: Requisiti non funzionali

¹Traiettoria a cui viene associato un istante temporale. Un waypoint bi-dimensionale sarà quindi composto da x, y, t

Requisiti Funzionali	
Tipo	Requisito
R1F	Interfaccia per definire una traiettoria, possibilità di salvarla
R2F	Manager delle traiettorie, possibilità di rimuoverne
R3F	Possibilità di visualizzare le traiettorie salvate in precedenza
R4F	Capacità di convertire la traiettoria salvata da serie di waypoint a polinomiale di terzo grado
R5F	Grafico che mostri la polinomiale generata insieme alle derivate fino al secondo ordine (quindi posizione, velocità ed accelerazione)
R6F	Interfacciamento con ROS, interfaccia con comandi rapidi per i quadrirotori
R7F	Possibilità di inviare una traiettoria al quadrirotore tramite ROS
R8F	Possibilità di cambiare le impostazioni dell'applicazione

Tabella 3.2: Requisiti funzionali

3.3 Casi d'uso

Lo scopo del diagramma dei casi d'uso è quello di comprendere meglio i requisiti funzionali, così da definire al meglio:

- **Attori** – cioè un ruolo interpretato da un utente
- **Caso d'uso** – cioè un servizio che il sistema offre per un attore o un altro caso d'uso

Il diagramma dei casi d'uso è utile per comprendere il limite del sistema, suddividerlo in sotto-moduli e definire le relazioni tra di essi e gli attori; è rappresentato secondo le regole della sintassi UML, tra cui i seguenti componenti:

- **Persona**: rappresenta un attore
- **Ellisse**: rappresenta un caso d'uso
- **Freccia “extends”** – relazione fra due casi d'uso. Si utilizza quando il caso d'uso da cui parte la freccia aggiunge delle funzionalità a quello in cui essa termina.
- **Freccia “include”** – similmente alla freccia “extends”, si utilizza quando il caso d'uso da cui parte utilizza almeno una volta quello in cui essa termina.

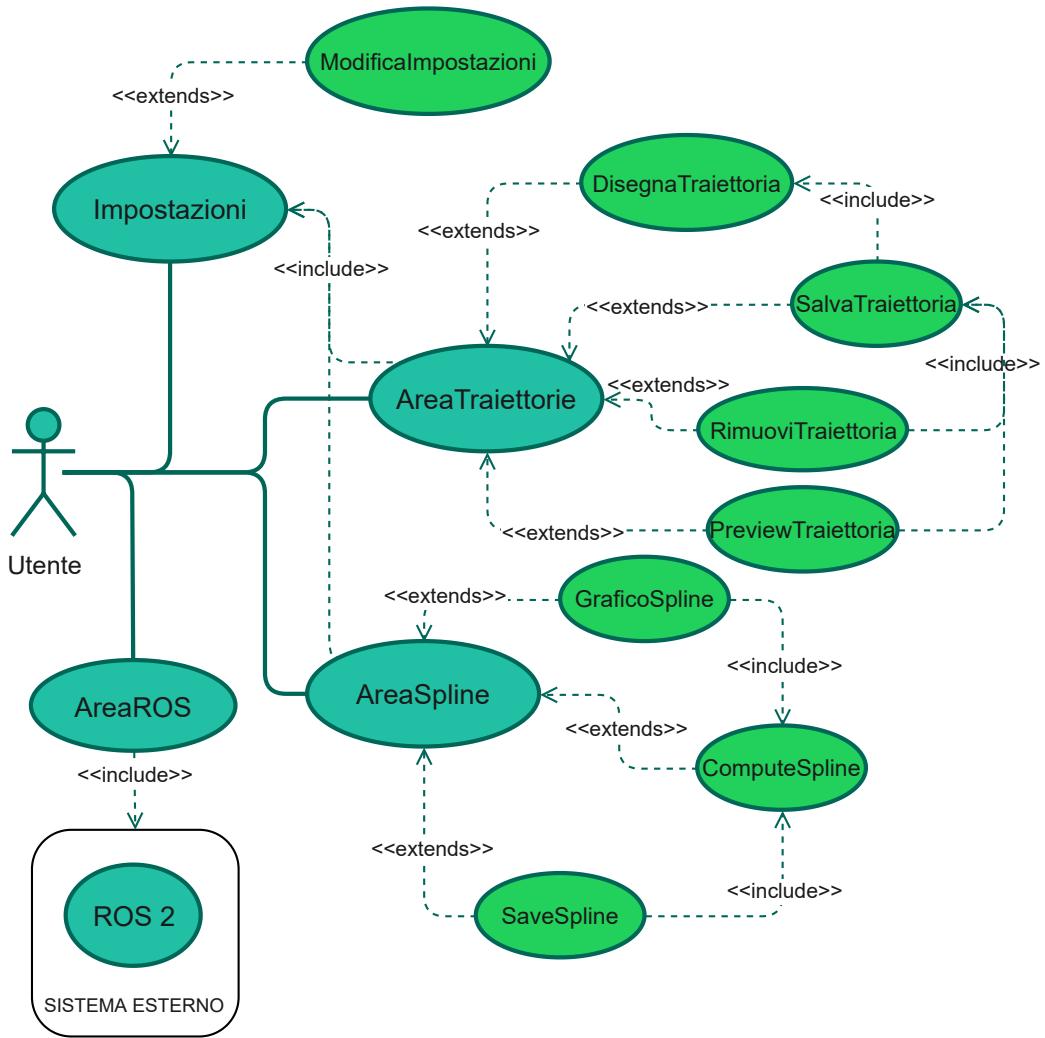


Figura 3.1: Diagramma dei casi d'uso

3.3.1 Scenari

In questa sezione verranno approfonditi gli scenari collegati ad ogni caso d'uso.

Impostazioni

L'utente ha la possibilità di modificare le impostazioni relative all'applicazione, come:

- Dimensione dell'area di disegno
- Percorso di salvataggio delle traiettorie

- Percorso di salvataggio degli spline²

Per applicare i cambiamenti l'utente dovrà premere sul pulsante di salvataggio; questo causerà un riavvio dell'applicazione.

Area Traiettorie

L'utente ha a disposizione un'area di disegno con una serie di pulsanti ed un file manager. L'utente può:

- Disegnare una traiettoria nell'area di disegno
- Cancellare la traiettoria disegnata premendo il pulsante “CLEAR”
- Salvare la traiettoria disegnata (pulsante “SAVE”), apparirà un prompt dove verrà richiesto il nome del file
- Visualizzare l'elenco delle traiettorie salvate, con possibilità di eliminarne
- Visualizzare una preview di una traiettoria salvata

Area Spline

L'utente visualizza un file manager con le traiettorie salvate, una toolbar con dei pulsanti ed un area con i grafici.

L'utente può selezionare una traiettoria e premere il pulsante “SPLINE”, questo genererà uno spline di terzo grado e ne mostrerà il grafico.

Area ROS

L'utente visualizza un file manager con le traiettorie salvate ed una serie di pulsanti utilizzabili per controllare il quadrirotore.

Questo scenario si interfaccia direttamente con il sistema esterno di ROS.

²Tipo di interpolazione che consiste nel dividere il dominio della funzione in più intervalli ed approssimare la funzione ad una curva polinomiale ad ogni intervallo

3.4 Diagramma delle classi

Una volta analizzati i casi d'uso e gli scenari relativi è possibile definire l'architettura logica dell'applicativo nello specifico: in particolare è possibile individuare le classi associate ad attori e funzionalità e definirne le interazioni. Questo approccio facilita notevolmente la fase di realizzazione: la struttura delle classi implementata nel codice rispecchierà quasi completamente quella individuata in questa fase.

Il diagramma delle classi è rappresentato seguendo la sintassi UML insieme ad alcune convenzioni che ne facilitano la comprensione:

- **Classe di colore Rosso:** classe appartenente al layer di presentazione (questo comprende interazione con utenti, sistemi esterni ed esposizione di API)
- **Classe di colore Verde:** classe che funge da controller (interazione con layer di presentazione, manipolazione dati e gestione altri controller)

Area impostazioni

Il *MainController* si occupa di creare un'istanza di *Settings* e di metterla a disposizione degli altri controller.

Settings una volta istanziato carica le impostazioni da un file e le salva in un dizionario. *SettingsView* mette a disposizione dell'utente un'interfaccia per modificare le impostazioni dell'applicazione.

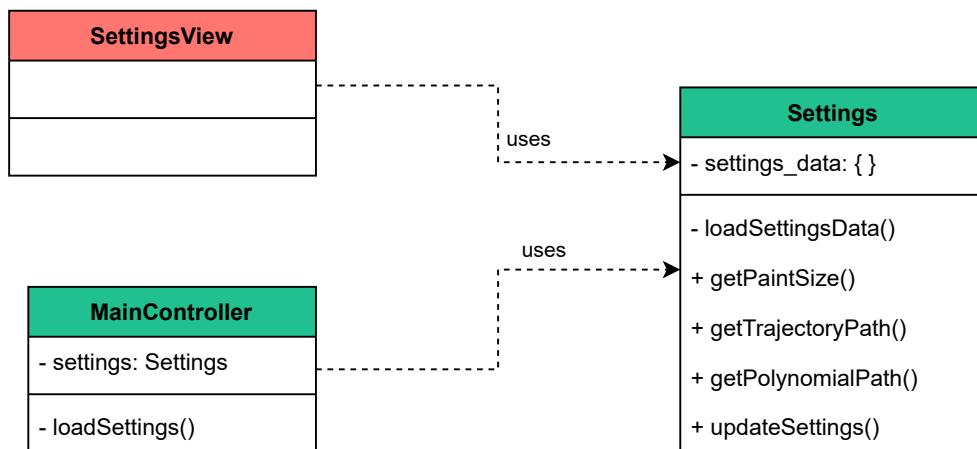


Figura 3.2: Diagramma delle classi: Impostazioni

Area traiettorie

L'interfaccia *TrajectoryView* è composta da:

- *FileManagerView*, che mostra un file manager con la lista delle traiettorie salvate. La logica è gestita da *FileManagerController*, che consente di eliminare un file e di mostrarne un’anteprima (visualizzata in *TrajectoryPreview*)
- *PaintTrajectoryView*, che mette a disposizione un’area dove disegnare la traiettoria, un pulsante per pulire la traiettoria disegnata ed uno per salvarla. *PaintController* si occupa di implementare i servizi offerti dall’interfaccia.

Sia *FileManagerController* che *PaintController* caricano le impostazioni correnti da *MainController*.

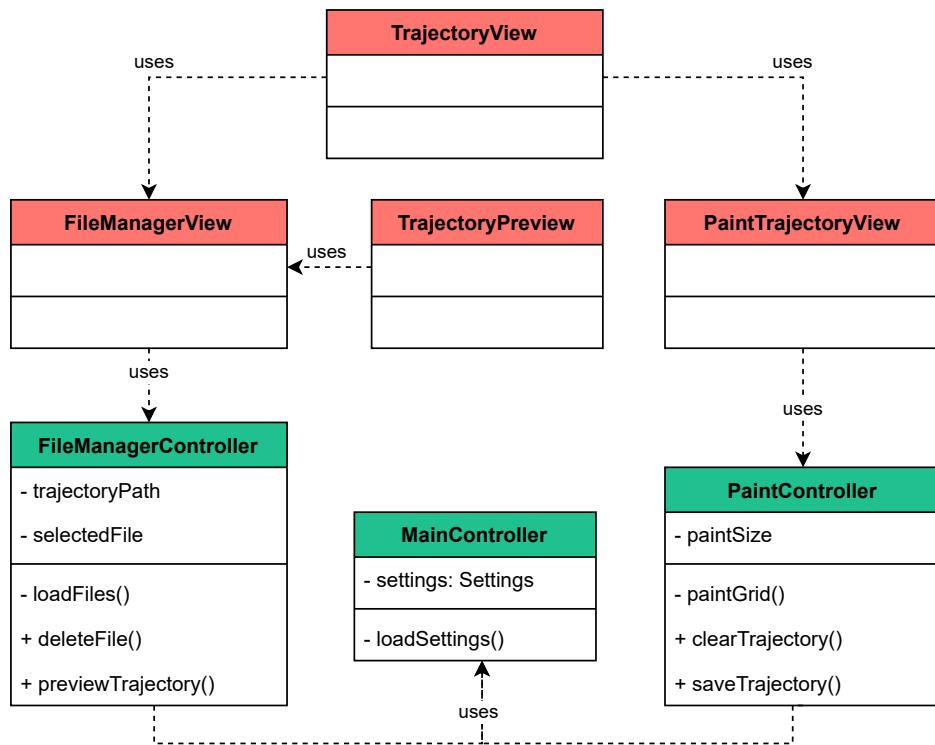


Figura 3.3: Diagramma delle classi: Traiettorie

Area Spline

SplineView è composta da un file manager e da un’area per la visualizzazione dei grafici. Il file manager mostra la lista delle traiettorie salvate e consente di:

- eliminare un file
 - generare una spline a partire da una traiettoria.
- Questo scatena una chiamata di funzione a *SplineController*, che si occupa di calcolare la spline e generare i due grafici (uno per l’asse x ed uno per l’asse y). I grafici vengono mostrati nell’interfaccia *SplineGraphView*.

- Salvare in un file la spline generata

Anche in questo caso *FileManagerController* carica le impostazioni da *MainController*.

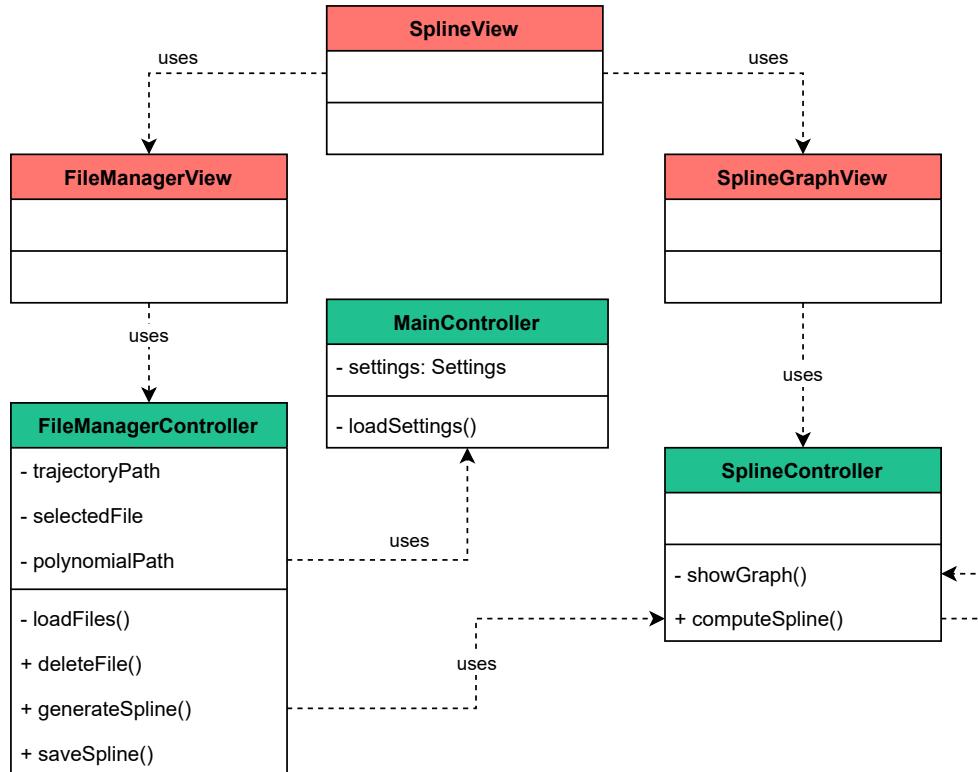


Figura 3.4: Diagramma delle classi: Spline

Area ROS

L’interfaccia *ROSView* mette a disposizione un file manager ed una toolbar (*ToolsView*) con dei comandi rapidi per i quadrirotori, come Hover e Land.

FileManagerView mostra la lista delle traiettorie salvate e permette di:

- eliminare un file
- inviare una traiettoria al quadrirotore.

Quest’ultima funzionalità, così come Hover e Land forniti dalla toolbar, viene gestita da *ROSController*, che si interfaccia direttamente con il sistema esterno di ROS.

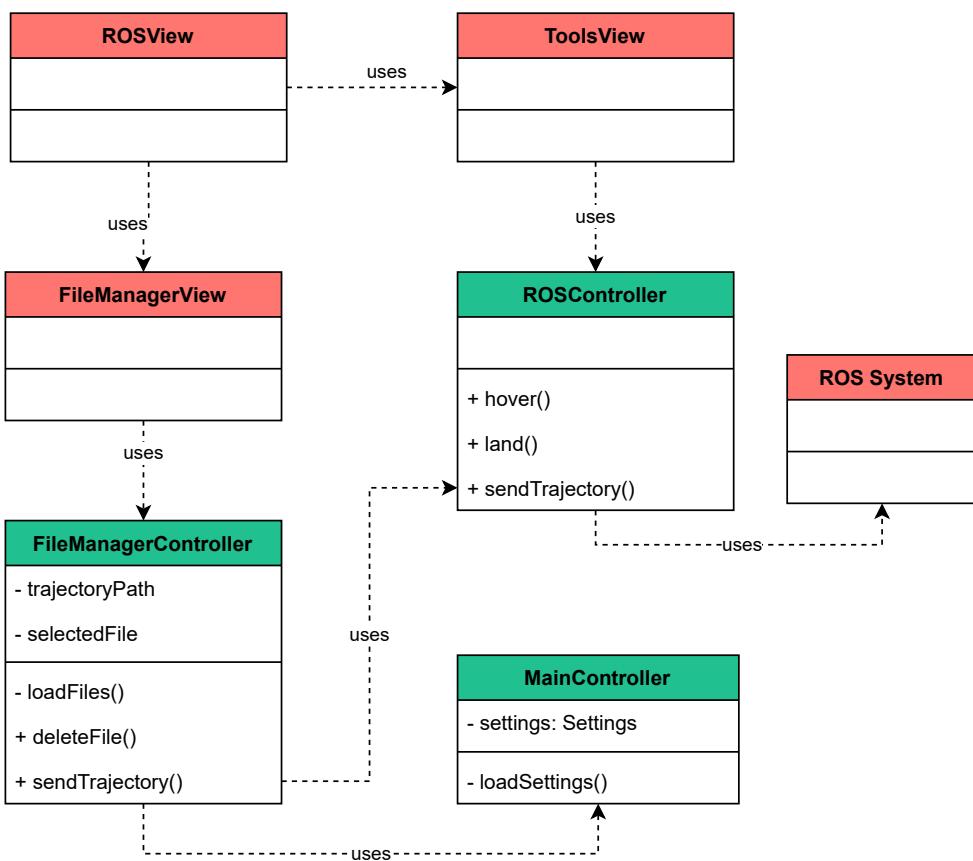


Figura 3.5: Diagramma delle classi: ROS

4 Realizzazione applicazione

4.1 Progettazione Architetturale

Il primo step del processo di realizzazione dell'applicazione è quello della progettazione dell'architettura.

Questo include:

- Analisi dei requisiti non funzionali (sezione 3.1)
- Scelta opportuna dei linguaggi di programmazione
- Identificazione dell'architettura più adatta allo scopo del progetto.

Una volta terminati questi passaggi è possibile utilizzare il diagramma delle classi definito nella sezione 3.4 per aiutarsi nell'implementazione dell'applicativo.

4.1.1 Requisiti non funzionali

In questa fase vengono presi in considerazione i requisiti non funzionali precedentemente individuati, che verranno considerati come “vincoli” in fase di implementazione:

- Interfaccia grafica multi-window semplice da utilizzare
- Comandi rapidi per il quadrirotore immediati da utilizzare
- Gestore delle traiettorie comodo
- Editor grafico delle impostazioni
- Conversione rapida da waypoint a polinomiale

Si nota come la maggior parte dei RNF riguarda l'ambito dell'interfaccia grafica, che deve essere veloce da utilizzare ed intuitiva.

È presente un vincolo sulla velocità di esecuzione della conversione da waypoint a spline, che verrà affrontato in seguito.

4.1.2 Scelta tecnologie

Come già anticipato in precedenza la scelta riguardante il linguaggio di programmazione è ricaduta su Python. Questo per una serie di motivi:

- Linguaggio diffuso e facilmente utilizzabile; questo facilita un'eventuale estensione o modifica dell'applicativo.
- Forte compatibilità con ROS 2. I Nodi vengono infatti implementati estendendo la classe **Node** dalla libreria *rclpy.node* di ROS 2.
- Presenza di numerose librerie dedicate al calcolo numerico e matriciale, come *numpy* [3].

Qt framework

Per soddisfare i requisiti relativi alla UI¹ si è scelto di utilizzare Qt², un framework multi piattaforma indirizzato allo sviluppo di applicazioni grafiche.

Qt è scritto in C++, questo garantisce delle ottime performance anche se utilizzato da linguaggi interpretati come Python.

La comunicazione fra Qt e l'applicazione in Python avviene tramite la libreria *pyqt*³: questa si interfaccia con delle API esposte dal framework di Qt ed offre un'ampia gamma di componenti grafici utilizzabili per comporre la UI.

Componenti di Qt

Tra i componenti messi a disposizione dal framework possiamo trovare:

- **Window**, è una finestra in cui è possibile disporre tutti gli altri componenti. Ogni applicazione Qt possiede una *QMainWindow*, cioè la finestra principale dell'applicazione; è possibile inoltre generare altre finestre, per esempio come avviene nei Dialogs.
- **Widget**, è il componente base del framework con cui l'utente può interagire. La UI è composta da Widgets posizionati all'interno di una o più Windows. Qt offre una vasta gamma di Widget già implementati, come pulsanti, caselle di testo, slider e molto altro.
- **Layout**, uno strumento utilizzato per disporre i Widget all'interno delle Window. Principalmente vengono usati due tipi di layout: *QVBoxLayout* e *QHBoxLayout*,

¹User Interface

²Qt documentation at https://wiki.qt.io/About_Qt

³Qt for Python, documentation at <https://doc.qt.io/qtforpython/>

che dispongono i componenti rispettivamente impilati verticalmente ed orizzontalmente. È possibile combinare più layout all'interno di uno stesso Widget o Window, in questo modo risulta molto facile creare anche delle interfacce elaborate.

- **Dialogs**, sono delle finestre pop-up⁴ che vengono utilizzate in più casi:

- messaggi di conferma o errori
- anteprima di un elemento
- richiesta di immissione dati (simile ad un Form)

Ogni componente di Qt può essere esteso, ridefinito e riutilizzato in più situazioni differenti; questo aumenta notevolmente la modularità del sistema e consente uno sviluppo (e una eventuale estensione) del sistema molto più rapido.

Esempio implementazione Qt: File Manager

In questa sezione viene riportato un esempio di come la gestione a componenti faciliti l'implementazione dell'applicazione, cioè il caso della progettazione del File Manager. Come è stato evidenziato in fase di Design, è necessario che l'applicazione possieda un File Manager in tre aree differenti, cioè in *Area Traiettorie*, *Area Spline* ed *Area Ros*. Dato che in tutti e tre i casi gran parte del codice relativo ad esso sarebbe lo stesso, si è optato per una struttura multi-classe modulare:

- Una classe `FileManagerMain` che estende `QWidget`.
Questa rappresenta il File Manager principale con i componenti comuni a tutti e tre i casi ed è essa stessa composta da due componenti: un `FileManager` (la lista effettiva dei file) ed una toolbar contenente il pulsante per eliminare un file.
- Altre tre classi che estendono `FileManagerMain`: `FileManagerDraw`, `FileManagerSpline` e `FileManagerROS`.
Ognuna di esse aggiunge delle funzionalità a `FileManagerMain` a seconda delle esigenze dell'area in cui verranno utilizzate.

```
# Lista dei file
class FileManager(QWidget):
    def __init__(self, trajectory_path):
        ...

# File Manager principale, contiene la lista dei file
# ed una toolbar con pulsante per eliminare un file
class FileManagerMain(QWidget):
```

⁴Finestra, solitamente di piccole dimensioni, che compare automaticamente durante l'utilizzo di un'applicazione

```
def __init__(self, trajectory_path):
    super().__init__()
    ...
    self.bttm_delete = QPushButton("DELETE")
    self.bttm_delete.clicked.connect(self.delete_file)
    self.toolbar = QtWidgets.QToolBar()
    self.toolbar.addWidget(self.bttm_delete)
    self.file_manager_widget = FileManager(trajectory_path)
    self.pagelayout = QVBoxLayout()
    self.pagelayout.addWidget(self.toolbar)
    self.pagelayout.addWidget(self.file_manager_widget)
def delete_file(self):
    ...

# Estensioni del File Manager principale
class FileManagerDraw(FileManagerMain):
    def __init__(self, trajectory_path):
        super().__init__(trajectory_path)
        self.bttm_preview = QPushButton("PREVIEW", self)
        self.bttm_preview.clicked.connect(self.preview_trajectory)
        self.toolbar.addWidget(self.bttm_preview)
    ...
    def preview_trajectory(self):
        dlg = QDialog(self)
        preview = GraphPreview(self.selected_file)
        ...
        dlg.exec()
    ...
```

4.1.3 Persistenza

Una parte fondamentale in fase di sviluppo è l'opportuna progettazione della persistenza dei dati. Nel caso specifico di questo progetto essa è molto limitata: gli unici dati rilevanti sono quelli relativi ad impostazioni, traiettorie e spline; non è quindi necessario appoggiarsi ad un Database e sarà sufficiente salvare le informazioni all'interno di file.

Impostazioni

Le impostazioni vengono memorizzate nel file `settings.yaml`; questo è codificato secondo la sintassi yaml⁵ per aumentarne la leggibilità.

Ecco un esempio di file di configurazione dell'applicazione:

```
area_settings:  
  paint_square_size:  
    - 2  
    - 2  
data_storage:  
  polynomials_directory: /home/lore/Documents/Crazydraw/saves/poly  
  trajectory_directory: /home/lore/Documents/Crazydraw/saves/trajectory  
enable_ros: true
```



Figura 4.1:
Logo
Yaml

Traiettorie e spline

La memorizzazione delle traiettorie e degli spline avviene nella path specificata nel file di configurazione.

I file sono memorizzati in formato CSV (*Comma Separated Values*) secondo questa forma:

- Le traiettorie vengono memorizzate come serie di waypoints:

```
x, y, time  
2137.1428571428573, 1360.0, 0.001010894775390625  
...  
...
```

- Per quanto riguarda gli spline, vengono memorizzati i coefficienti dei polinomi generati (sia per x che per y) specificando l'intervallo temporale in cui valgono:

```
t_start, t_stop, ax^3, bx^2, cx, d, ay^3, by^2, cy, d  
0.005, 0.165, -91435.82218070257, ...  
...
```

4.2 Implementazione

In questa sezione verrà approfondita l'implementazione delle varie parti che compongono l'applicazione, come:

- interfaccia e componenti grafiche

⁵Yaml official Web Site – <https://yaml.org/>

- algoritmi per registrare la traiettoria ed eseguire l’interpolazione
- interfacciamento con il sistema di ROS 2

4.2.1 Interfaccia grafica

Innanzitutto verranno descritte le finestre che compongono l’interfaccia grafica, insieme alle funzionalità messe a disposizione dell’utente.

Area pianificazione traiettorie

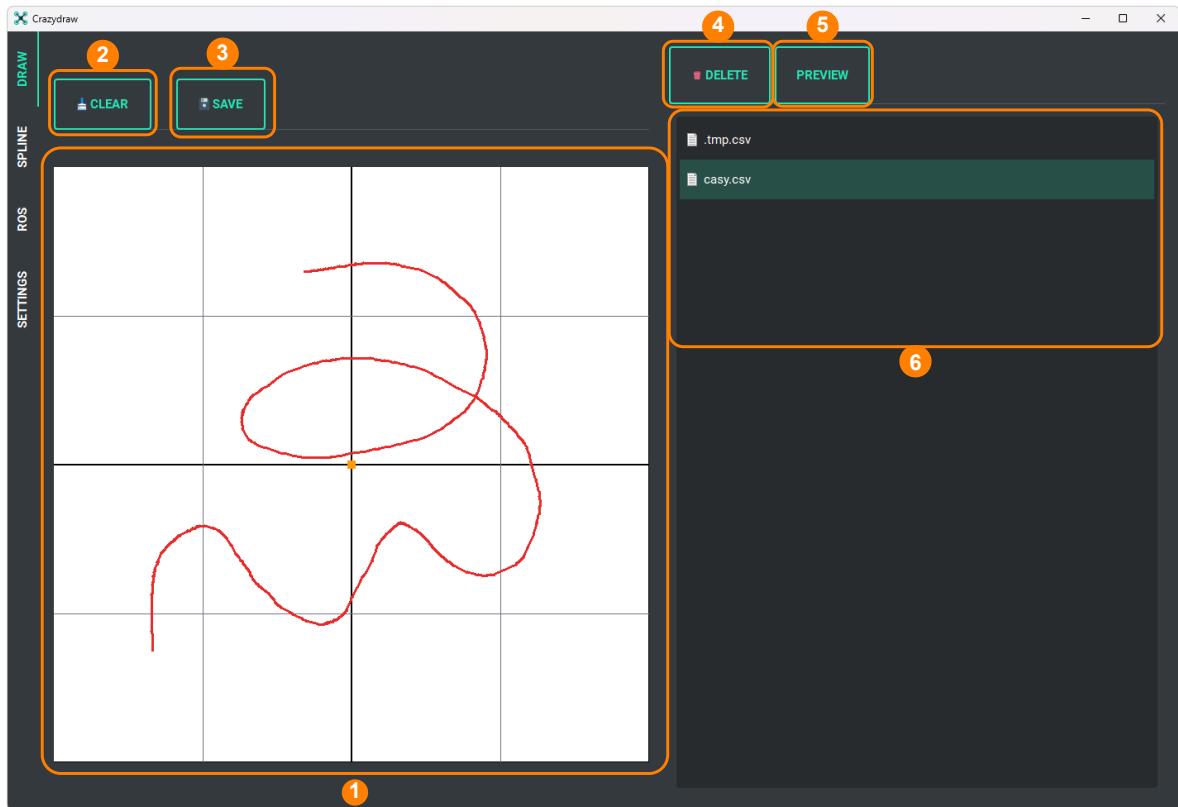


Figura 4.2: Interfaccia grafica: Area traiettorie

1. Area in cui è possibile disegnare la traiettoria. Un quadrato della griglia principale corrisponde ad un metro nella realtà (in questo caso la dimensione totale dell’area è di 2m x 2m). La dimensione dell’area può essere modificata nelle impostazioni.
2. Pulsante “CLEAR”, pulisce l’area di disegno

3. Pulsante “SAVE”, salva la traiettoria disegnata. Presenta un prompt in cui richiede il nome del file e, se non viene fornito, inserisce il timestamp
4. Pulsante “DELETE”, elimina il file selezionato
5. Pulsante “PREVIEW”, presenta un’anteprima della traiettoria selezionata
6. File Manager, mostra la lista delle traiettorie salvate (la directory di salvataggio può essere modificata nelle impostazioni)

Area visualizzazione spline

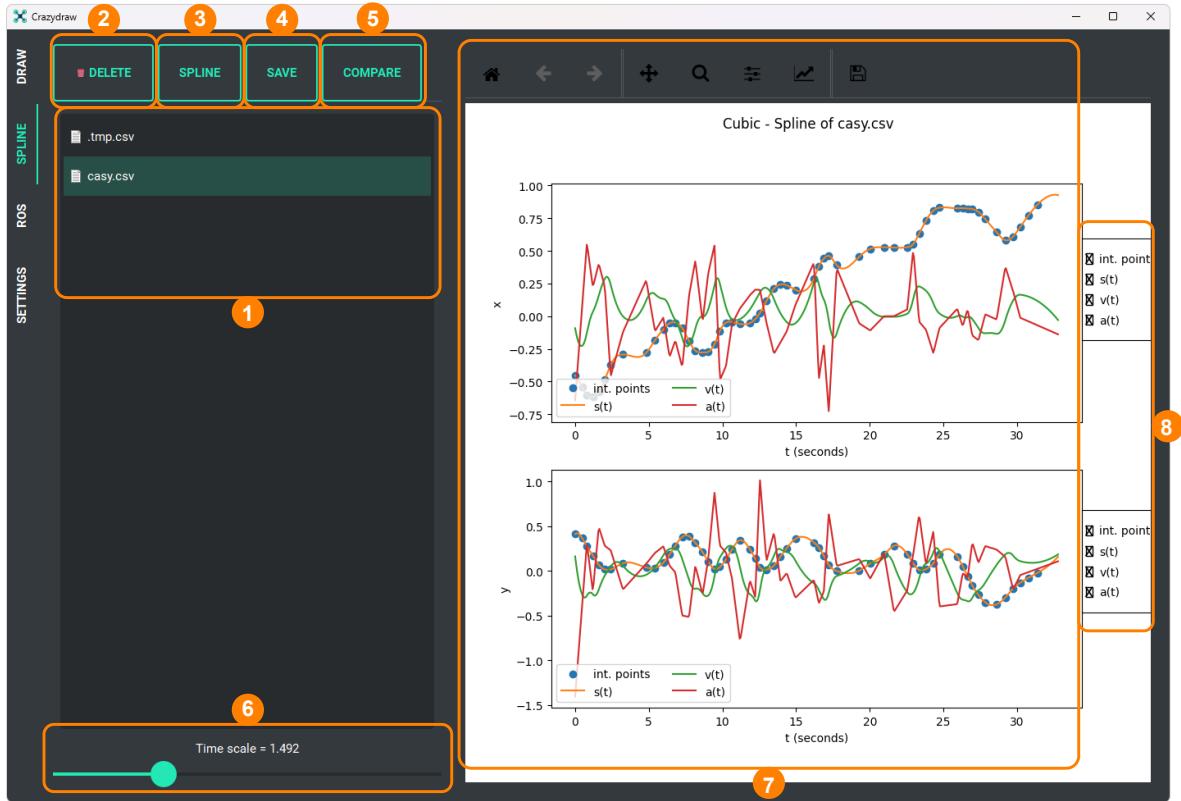


Figura 4.3: Interfaccia grafica: Area spline

1. File Manager, mostra la lista delle traiettorie salvate (la directory di salvataggio può essere modificata nelle impostazioni)
2. Pulsante “DELETE”, elimina il file selezionato
3. Pulsante “SPLINE”, calcola la spline relativa alla traiettoria selezionata e ne mostra il grafico sull’area apposita

4. Pulsante “SAVE”, calcola la spline relativa alla traiettoria selezionata e la salva sulla directory preimpostata. Presenta un prompt in cui richiede il nome del file e, se non viene fornito, inserisce il timestamp
5. Pulsante “COMPARE”, plotta la traiettoria e la spline calcolata a partire da essa, così da evidenziare le differenze
6. Slider che permette di aumentare la durata di una traiettoria
7. Toolbar e grafici e relativi alle spline degli assi x ed y rispetto al tempo. Vengono mostrati i punti di interpolazione, la posizione, la velocità e l’accelerazione.
8. Legenda che consente di scegliere cosa visualizzare nei grafici

Area ROS

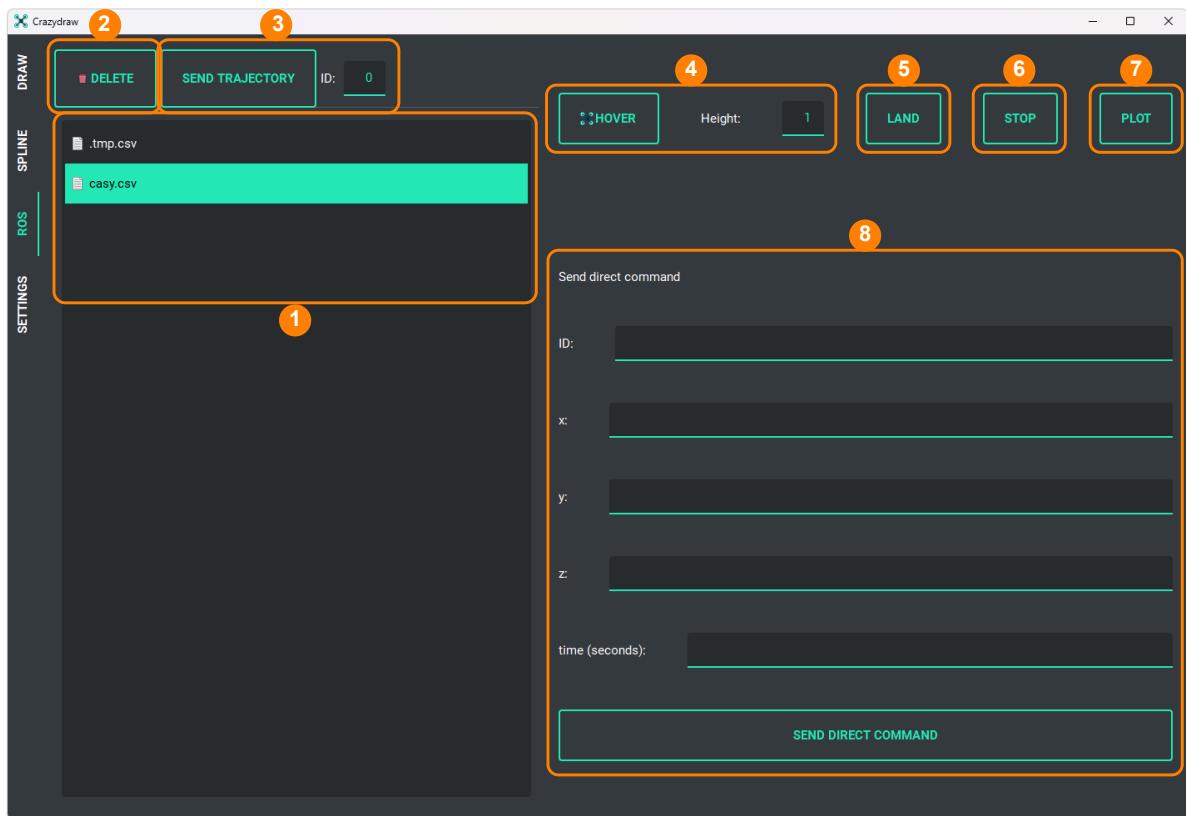


Figura 4.4: Interfaccia grafica: Area ROS

1. File Manager, mostra la lista delle traiettorie salvate (la directory di salvataggio può essere modificata nelle impostazioni)

2. Pulsante “DELETE”, elimina il file selezionato
3. Pulsante “SEND TRAJECTORY” con campo per specificare l’ID, invia tramite ROS la traiettoria al quadrirotore
4. Pulsante “HOVER”, fa eseguire un hover all’altezza specificata al quadrirotore
5. Pulsante “LAND”, invia un comando che fa atterrare il quadrirotore
6. Pulsante “STOP”, invia un comando che fa spegnere il quadrirotore
7. Pulsante “PLOT”, mostra un grafico con la traiettoria percorsa, registrata dal controller del quadrirotore
8. Area “Direct command”, è possibile inviare un comando “go to” ad un quadrirotore (selezionato tramite l’ID): viene specificata la posizione di destinazione nei tre assi ed il tempo che deve impiegare il quadrirotore per arrivarci.

Area Impostazioni



Figura 4.5: Interfaccia grafica: Impostazioni

1. Directory di salvataggio per le traiettorie
2. Directory di salvataggio per gli spline generati
3. Dimensione dell'area di disegno in metri
4. Abilitazione della comunicazione con ROS
5. Pulsante di salvataggio delle impostazioni. Se premuto aggiorna il file *settings.yaml* e riavvia l'applicazione

4.2.2 Interpolazione di una traiettoria

Memorizzazione waypoint

Come già specificato durante la progettazione della persistenza (sezione 4.1.3), la traiettoria disegnata viene salvata come waypoint bi-dimensionale.

Di conseguenza durante la realizzazione della traiettoria l'applicazione, oltre a salvare il singolo punto bi-dimensionale, deve registrare l'istante temporale associato a tale punto. La memorizzazione della traiettoria disegnata sull'area apposita avviene utilizzando un file temporaneo *.tmp.csv*, che verrà poi rinominato in caso la traiettoria debba essere salvata.

```
# First event.
if self.last_x is None:
    self.last_x = e.x()
    self.last_y = e.y()
    self.start_time = time.time()
    return # Don't paint first time
# Update the painting area
painter = QtGui.QPainter(self.pixmap())
...
painter.drawLine(self.last_x, self.last_y, e.x(), e.y())
painter.end()
self.update()
# Update the origin for next time.
self.last_x = e.x()
self.last_y = e.y()
# Append waypoint in temporary file
self.csv_file.write(f'{(self.last_x - self.settings.get_paint_size_scaled()[0]/2) / self.scale_size}, {((self.settings.get_screen_res().height()*2/3 - (self.last_y + self.settings.get_paint_size_scaled()[1]/2)) / self.scale_size}, {(time.time() - self.start_time)*2.5}\n')
```

Punti di interpolazione

La prima operazione da eseguire per il calcolo dello spline è quella di ricavare i punti di interpolazione, cioè quei punti che fungeranno da estremi per gli intervalli in cui verrà calcolato ogni singolo polinomio.

Nel programma sono implementate due funzioni che ottengono la lista dei punti da interpolare: *distance_interpolation* e *time_interpolation*; entrambe accettano come parametri i tre vettori contenenti i dati dei waypoint (x , y , t) e il numero di punti di interpolazione da ottenere.

L'algoritmo di *distance_interpolation* calcola la distanza fra i punti e ne aggiunge uno quando essa è maggiore dell'intervallo desiderato.

```
def __distance_interpolation__(x, y, t, line_count, interval):
    # Initialise lists
    tmp_x, tmp_y, tmp_t = [], [], []
    tmp_x.append(x[0])
    tmp_y.append(y[0])
    tmp_t.append(t[0])
    # Compute interpolation points
    tmp_dist = 0
    for i in range(line_count - 1):
        if math.dist([tmp_x[-1], tmp_y[-1]], [x[i + 1], y[i + 1]]) + tmp_dist >
           interval:
            tmp_x.append(x[i])
            tmp_y.append(y[i])
            tmp_t.append(t[i])
            tmp_dist = 0
        else:
            tmp_dist += math.dist([x[i], y[i]], [x[i + 1], y[i + 1]])
    # Return numpy array
    return np.array(tmp_x), np.array(tmp_y), np.array(tmp_t)
```

L'algoritmo *time_interpolation* invece utilizza il vettore degli istanti temporali t per ottenere gli indici dei punti da restituire.

```
def __time_interpolation__(x, y, t, line_count, num_interpolation):
    tmp_x, tmp_y, tmp_t = [], [], [] # temporary arrays
    # get index of interpolations
    xs = np.arange(0, line_count - 1, int((line_count - 1) / num_interpolation))
    for i in xs:
        tmp_x.append(x[i])
        tmp_y.append(y[i])
        tmp_t.append(t[i])
    # return numpy arrays
    return np.array(tmp_x), np.array(tmp_y), np.array(tmp_t)
```

4.2.3 Calcolo spline di una traiettoria

Spline e punti di interpolazione

Una volta ottenuti i punti da interpolare viene utilizzata una funzione della libreria `scipy`⁶, più precisamente del modulo `scipy.interpolate`, cioè `CubicSpline`.

`CubicSpline`⁷ accetta come parametri i due vettori (ascissa ed ordinata) dei valori da interpolare e restituisce un oggetto contenente lo spline.

Una volta creato l'oggetto è possibile ottenere i coefficienti tramite il parametro `c`, rappresentati come matrice di dimensioni [4, n-1], in cui n è il numero di interpolazioni.

È inoltre possibile ottenere la derivata n-esima dello spline utilizzando una chiamata a funzione `cubic_spline(t, n)`, in cui t è l'array delle ascisse su cui calcolare la funzione.

```
# Get interpolations point
x, y, t = DrawSpline.distance_interpolation(x, y, t, line_count, num_interpolation)
# Compute spline
spline_x = CubicSpline(t, x)
spline_y = CubicSpline(t, y)
# Plot derivatives of spline_x
plots, ax = plt.subplots(2)
ax[0].plot(t, x, 'o', label='data')
ax[0].plot(t_full, spline_x(t_full), label="s(t)")
ax[0].plot(t_full, spline_x(t_full, 1), label="v(t)")
ax[0].plot(t_full, spline_x(t_full, 2), label="a(t)")
...
...
```

⁶Virtanen, P., Gommers, R., Burovski, E., Oliphant, T. E., Weckesser, W., Cournapeau, D., ... & Feng, Y. (2021). `scipy/scipy`: SciPy 1.5. 3. Zenodo.

⁷CubicSpline documentation at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html>

Esempio interpolazione e calcolo spline di una traiettoria

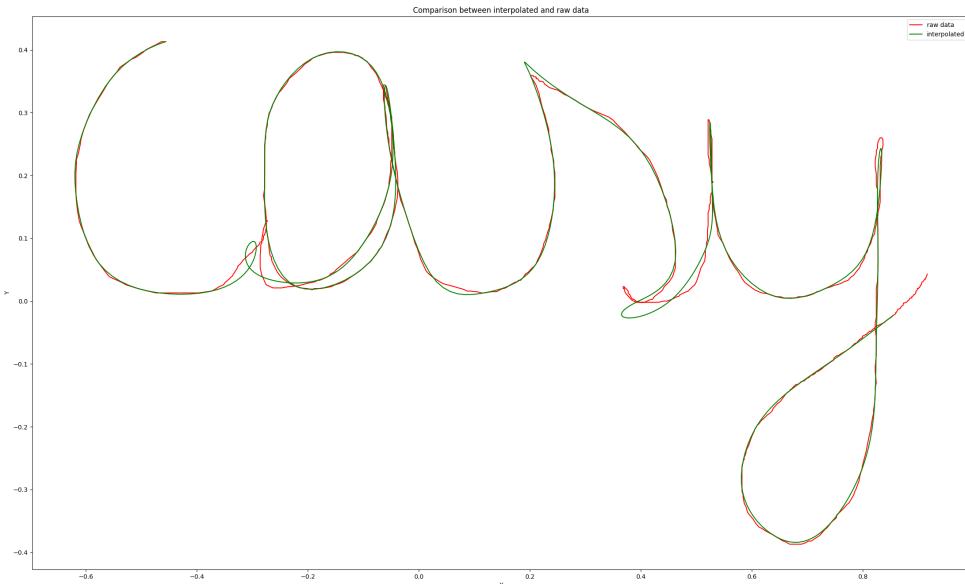


Figura 4.6: Preview traiettoria disegnata e spline

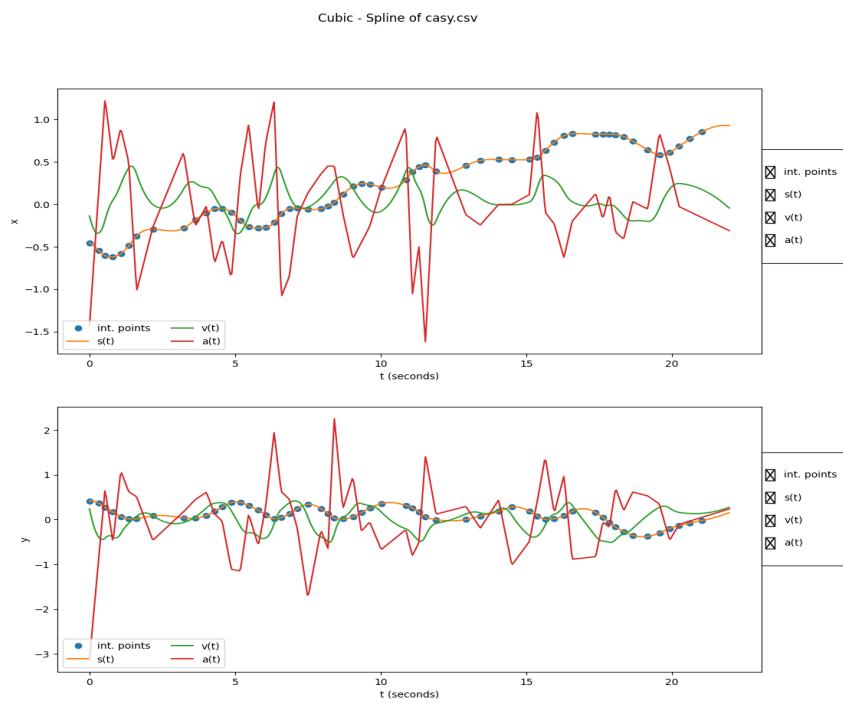


Figura 4.7: Interpolazione con algoritmo *distance_interpolation*

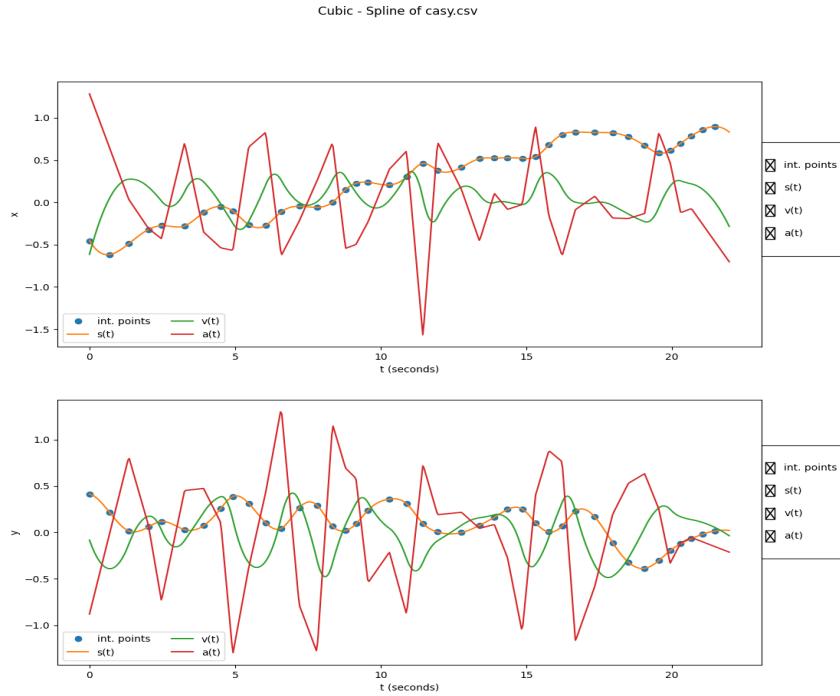


Figura 4.8: Interpolazione con algoritmo *time_interpolation*

4.2.4 Interfacciamento con ROS

Per far sì che l'applicazione comunichi con i quadrirotori e il simulatore è necessario configurare un collegamento con il sistema di ROS 2 installato nella macchina; questo avviene tramite l'implementazione del controller come nodo di ROS.

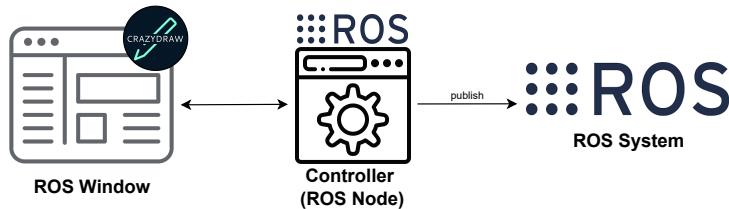


Figura 4.9: Schema comunicazione con ROS 2

ROS controller

All'interno del controller avviene la creazione dei Publisher sui topic utilizzati per comunicare i comandi, tra i quali:

- /land, un semplice publish in questo topic invia un comando di atterraggio a tutti i quadrirotori. Non utilizza nessun tipo di messaggio specifico.

- `/agent_n/traj_params`, utilizzato per comunicare una serie di waypoint tridimensionali. Utilizza il messaggio *Trajectory*

In particolare il topic `/agent_n/traj_params` viene utilizzato sia per il comando di hover che per la comunicazione di una traiettoria.

Il comando “hover” infatti genera una traiettoria, con origine il punto (0, 0, 0) e destinazione (0, 0, height), rispetto agli assi *x*, *y*, *z*.

```
def hover(self, height=1.0, duration = 5.0, id = 0):
    # Inizializzazione
    x = numpy.zeros(10)
    y = numpy.zeros(10)
    z = numpy.linspace(0, height, 10)
    t = numpy.linspace(0.2, duration, 10)
    msg = Trajectory()
    # Append waypoints to trajectory
    for i in range(len(t)):
        point = TrajectoryPoint()
        point.positions = [float(x[i]), float(y[i]), float(z[i])]
        point.time_from_start.sec = int(t[i])
        point.time_from_start.nanosec = int((t[i]-int(t[i]))*1e9)
        ...
        msg.points.append(point)
    # Publish trajectory
    self.publishers_traj_params[id].publish(msg)
```

Per quanto riguarda la traiettoria, il programma si limita ad inviare i punti di interpolazione ricavati con gli algoritmi approfonditi nella sezione precedente (4.2.3).

```
def send_trajectory(self, id, filename):
    # Get interpolation points
    x, y, t, line_count = DrawSpline.get_cords(file_name)
    x, y, t = DrawSpline.__distance_interpolation__(x, y, t,
    ↳ line_count, num_interpolations=30)
    # Build message
    msg = Trajectory()
    for i in range(len(t)):
        point = TrajectoryPoint()
        point.positions = [float(x[i]), float(y[i]), float(z)]
        point.time_from_start.sec = int(t[i])
        point.time_from_start.nanosec = int((t[i]-int(t[i]))*1e9)
        ...
        msg.points.append(point)
    # Publish trajectory
    self.publishers_traj_params[id].publish(msg)
```

Handler traiettoria

All'interno del sistema di ROS è presente un nodo che si occupa di leggere i waypoint pubblicati sul topic `/agent_n/traj_params`, utilizzarli per calcolare lo spline e fornire quindi posizione, velocità ed accelerazione relative ad un istante t al controllore dei quadrirotori. Innanzitutto il nodo si iscrive al topic `/agent_n/traj_params` ed aggiorna una variabile locale ogni volta che qualcuno pubblica una traiettoria.

Quando questo succede viene impostata la flag `self.first_evaluation = True`, in questo modo il programma calcolerà nuovamente lo spline.

```
def traj_params_callback(self, msg):
    traj_params = msg.points
    # Initialise trajectory
    self.start_pose = np.copy(self.current_pose.position)
    self.traj_params['time'].append(0)
    self.traj_params['position'].append(self.current_pose.position)
    self.traj_params['velocity'].append(self.current_pose.velocity)
    self.traj_params['acceleration'].append(np.zeros(3))
    self.traj_params['jerk'].append(np.zeros(3))
    # Copy trajectory points
    for i in range(len(traj_params)):
        self.traj_params['time'].append(traj_params[i].time_from_start.sec +
            → traj_params[i].time_from_start.nanosec*1e-9)
        self.traj_params['position'].append(traj_params[i].positions)
        self.traj_params['velocity'].append(np.array(traj_params[i].velocities))
        self.traj_params['acceleration'].append(np.array(traj_params[i].accelerations))
        self.traj_params['jerk'].append(np.array(traj_params[i].effort))
    self.first_evaluation = True
```

Ogni volta che il controllore effettua una richiesta al nodo per ottenere i dati viene invocata la funzione `evaluate_reference()`; questa (se il flag `first_evaluation` è impostato a `True`) si occuperà di calcolare lo spline e poi di restituire i dati relativi a posizione, velocità ed accelerazione.

```
def evaluate_reference(self):
    if self.first_evaluation:
        self.start_time_sec = self.get_time() # Get startings time
        # Coefficient computation
        self.compute_coefficients()
        self.first_evaluation = False
    # Get execution time
    time = self.get_time() - self.start_time_sec
    # position
    ref[0] = self.multi_spline[0](time)
```

```
ref[1] = self.multi_spline[1](time)
ref[2] = self.multi_spline[2](time)
# attitude
attitude_reference = np.zeros(3) # Roll, Pitch, Yaw
ref[3:7] = R.from_euler('xyz', attitude_reference).as_quat()
# velocity
ref[7] = self.multi_spline[0](time, 1)
ref[8] = self.multi_spline[1](time, 1)
ref[9] = self.multi_spline[2](time, 1)
# acceleration
ref[10] = self.multi_spline[0](time, 2)
ref[11] = self.multi_spline[1](time, 2)
ref[12] = self.multi_spline[2](time, 2)
return ref
```

La funzione `compute_coefficients()` si occupa di calcolare i tre spline relativi agli assi (x, y, z), utilizzando la funzione *CubicSpline* di `scipy.interpolate` già approfondita in precedenza (sezione 4.2.3).

5 Sperimentazione

In questo capitolo verrà approfondita la fase di testing dell'applicazione, insieme alla descrizione degli strumenti utilizzati durante la sperimentazione. Verranno quindi riportati i risultati ottenuti ed un'analisi degli stessi.

5.1 Strumenti utilizzati

Come già spiegato nella sezione 2.1 il progetto è composto da tre componenti principali; i primi due (applicazione e sistema ROS 2) sono già stati approfonditi in precedenza. Il terzo livello è composto dall'interfacciamento di ROS 2 con l'hardware e gli strumenti del laboratorio, che nel caso di questo progetto sono principalmente due:

- Il sistema di tracking **Vicon**
- I quadrirotori **Crazyflie**

Entrambi i dispositivi e la loro cooperazione verranno approfonditi in questa sezione.

5.1.1 Vicon

Vicon è un sistema di Motion Capture¹ sviluppato dall'azienda omonima. Esso consiste in un sistema di telecamere ad infrarossi disposte appositamente e connesse ad un hub Vicon tramite cavo Ethernet; l'hub viene poi connesso ad un computer in cui è installato il software che, utilizzando le informazioni di tutte le telecamere, riesce a ricavare la posizione degli oggetti o delle persone tramite l'utilizzo dei marker.

Il marker è una sfera di piccole dimensioni (diametro di circa 1 cm) ricoperta di un materiale che riflette i raggi infrarossi; in questo modo il marker è facilmente individuabile dalle telecamere.

Questo sistema di tracking garantisce una precisione molto elevata (± 0.1 mm), essenziale nell'ambito della robotica in quanto anche un piccolo errore di rilevamento può causare un'errata impartizione dei comandi da parte del controllore.

Per tutte queste caratteristiche Vicon viene utilizzato non solo nell'ambito della ricerca, ma anche come strumento in ambito di produzione cinematografica e videoludica.

¹Processo di registrazione dei movimenti di oggetti o persone



Figura 5.1: Telecamera ad infrarossi Vicon

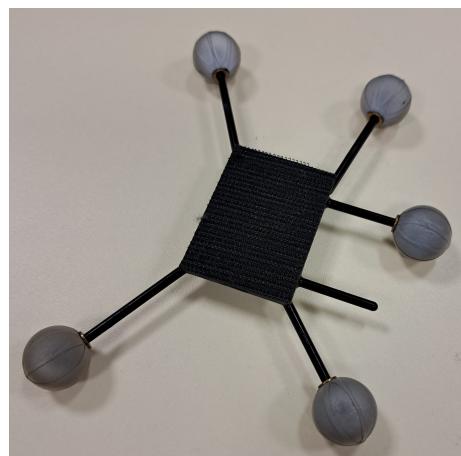


Figura 5.2: Prototipo di quadrirotore con marker

5.1.2 Crazyflie

Introduzione

Crazyflie è il nome dato al quadrirotore sviluppato e prodotto da BitCraze². Si tratta di un UAV leggero e di piccole dimensioni (peso 29 grammi, diametro 92 mm) e con hardware e software associato completamente open-source progettato principalmente per l'utilizzo da parte di ricercatori e sviluppatori.

Crazyflie è una piattaforma composta da:

- Quadrirotore Crazyflie 2.0
- Crazyradio PA, un USB dongle personalizzato con un'antenna a 2.5 GHz e firmware programmabile. Viene utilizzata per comunicare con i Crazyflie.
- Firmware ed architettura software basata su ROS

²<https://www.bitcraze.io/>



Figura 5.3: Quadrirotore Crazyflie 2.0



Figura 5.4: Crazyradio PA

Protocollo di comunicazione

Il protocollo di comunicazione di Crazyflie si basa su uno stack di quattro livelli indipendenti fra di loro:

Subsystems	Log, Comandi, Parametri, Memoria
CRTP	[porta, canale, payload]
Link	Radio, USB abstract link
Physical medium	Livello fisico di USB e Radio

Tabella 5.1: Stack protocollo di comunicazione Crazyflie

I livelli **Link** e **Physical Medium** si occupano di definire il mezzo di comunicazione e i protocolli di basso livello ad esso correlati, astraendoli in modo da renderli “invisibili” ai livelli superiori.

Attualmente sono supportati due tipi di mezzo di comunicazione:

- **USB link:** connessione diretta tramite cavo e protocollo USB 2.x
- **Radio link:** connessione via radio compatibile con nRF24 [?]

CRTP

Il nome sta per Crazy RealTime Protocol, il quale pacchetto è così formato:

port	channel	payload
(0-15) --> 4 bits	(0-3) --> 2 bits	31 bytes

Tabella 5.2: Struttura pacchetto CRTP

La porta viene utilizzata per selezionare il Subsystem a cui ci si vuole rivolgere, ognuno dei quali ha a disposizione 4 canali diversi per comunicare. Il payload contiene il vero e proprio corpo del comando, relativo al subsystem selezionato nel campo port.

Subsystems

I subsystem vengono utilizzati dai controllori dei quadrirotori per gestire più facilmente la suddivisione delle richieste che vengono eseguite dal client. Ci sono 15 subsystem, tra cui 0-Console, 3-Commander, 7-Setpoint Generico, 13-Platform di particolare rilievo per il controllo del quadrirotore. In particolare:

- Il Subsystem 3(Commander) ci consente di impartire ordini direttamente al “controllore di volo”, cioè di agire direttamente su parametri come roll, pitch, yaw e thrust.
- Il Subsystem 7(Generic Setpoint) consente invece un controllo più di alto livello, come la modalità hover o il controllo della velocità sui tre assi cartesiani.
- Il Subsystem 13(Platform Services) offre una lista di comandi relativi alla piattaforma in uso.

Crazyflie URI

Ogni Crazyflie viene identificato da un URI univoco, nel caso della connessione radio tramite USB Dongle l'URI (esempio URI: `radio://8/100/2M/E7E7E7E705`) è diviso in quattro sezioni:

- Interfaccia utilizzata – `radio`
- Numero antenna USB – 8
- Canale radio – 100
- Velocità di collegamento – 2M
- Identificativo del Crazyflie – E7E7E7E705

5.2 Simulazione percorrenza traiettoria

In questa sezione verrà descritto il processo di utilizzo dell'applicazione, a partire dalla definizione della traiettoria fino alla simulazione di volo. Il simulatore è stato messo a disposizione dal laboratorio *CASY* ed ottiene i comandi iscrivendosi allo stesso topic di ROS 2 utilizzato dai Crazyflie.

5.2.1 Disegno traiettoria

Innanzitutto è necessario disegnare la traiettoria nell'apposita area e, se soddisfatti del risultato, salvarla.



Figura 5.5: Disegno traiettoria “Unibo”

5.2.2 Analisi spline

Una volta definita la traiettoria è consigliato utilizzare lo strumento per la visualizzazione del grafico dello spline. In questo modo è possibile individuare valori critici nella velocità ed accelerazione e, se necessario, aumentare la durata totale della traiettoria utilizzando lo slider apposito.

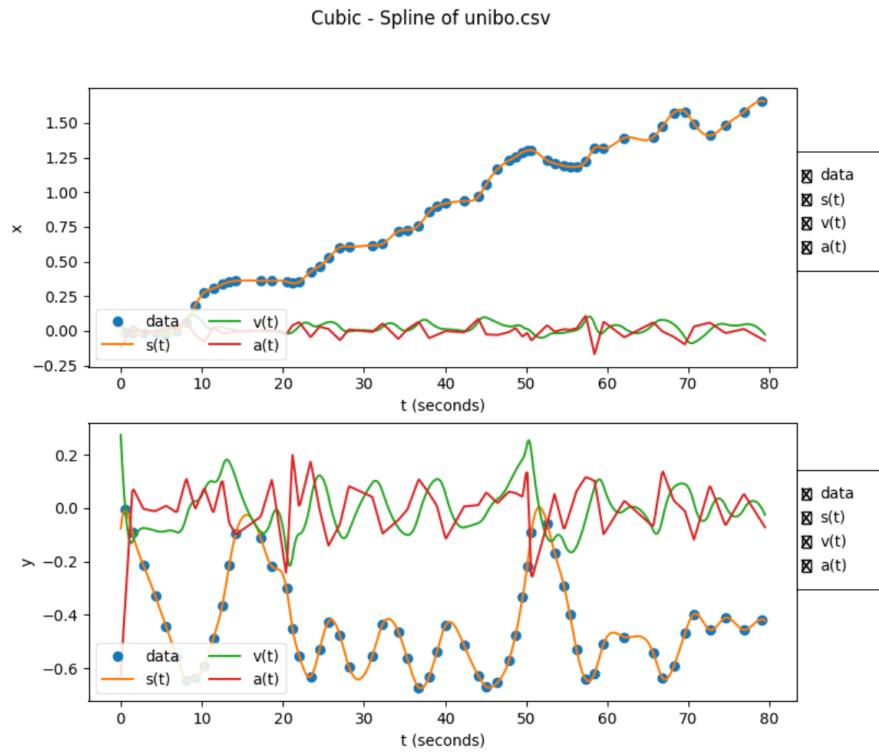


Figura 5.6: Spline traiettoria “Unibo”

5.2.3 Simulazione e volo

A questo punto è possibile inviare i comandi a ROS 2, quindi ai quadrirotori ed al simulatore. Innanzitutto è necessario far eseguire un hover al dispositivo, così che si trovi all’altezza giusta per seguire la traiettoria.

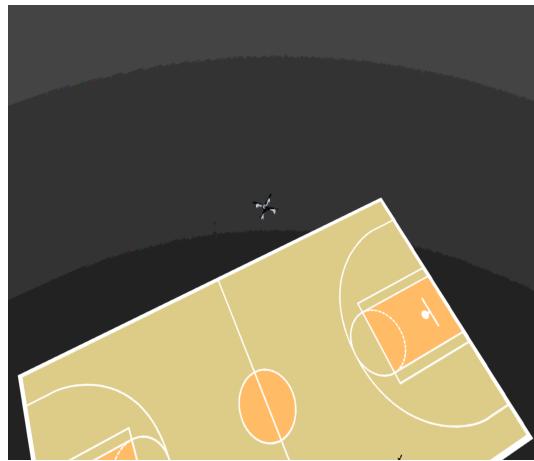


Figura 5.7: Simulazione di un quadrirotore in *Hover*

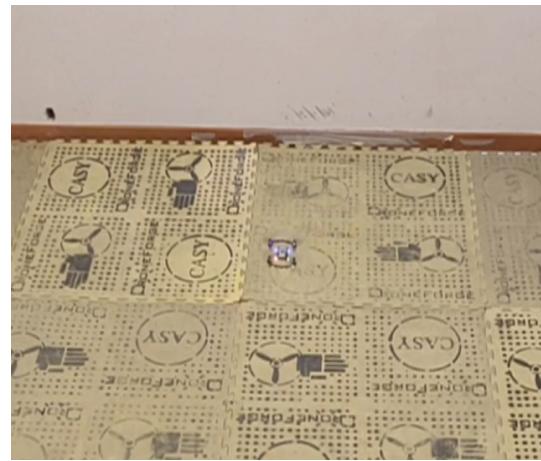


Figura 5.8: Quadrirotore Crazyflie in *Hover*

Il passo successivo è selezionare la traiettoria desiderata ed inviarla a ROS. Tramite il tool di *plotting* è possibile visualizzare il percorso reale seguito dal quadrirotore, registrato dal controllore dello stesso.

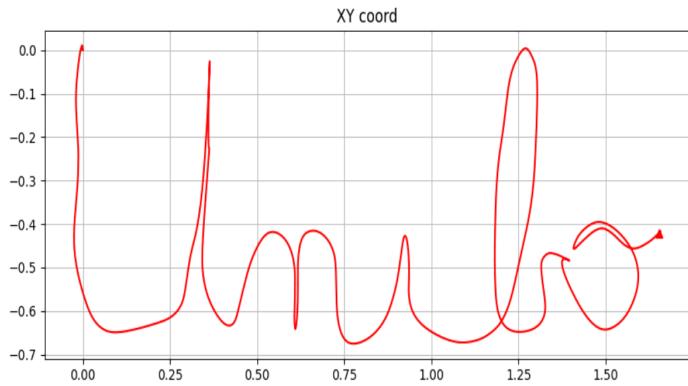


Figura 5.9: Percorso seguito dal quadrirotore

Conclusioni

In fase di sperimentazione ha avuto un'influenza rilevante l'utilizzo dell'applicazione sviluppata come obiettivo principale di questa tesi al posto dei sistemi convenzionali per manovrare i quadrirotori, come l'impartire i comandi tramite uno script di Python. Avere a disposizione un'interfaccia grafica in cui è possibile definire una traiettoria tracciandone il percorso con il puntatore del mouse e il poter impartire i comandi più frequenti (come l'hovering e l'atterraggio) semplicemente premendo dei pulsanti risulta notevolmente più comodo e consente di risparmiare del tempo prezioso. Inoltre, lo strumento di analisi delle spline interpolate risulta molto utile per determinare la fattibilità della traiettoria stessa: è possibile infatti verificare la presenza di valori critici nel grafico delle accelerazioni e delle velocità, in modo da prevenire errori ed incidenti che potrebbero verificarsi durante la percorrenza della traiettoria da parte del quadrirotore.

In futuro potrebbe essere considerato l'utilizzo di un algoritmo più preciso nel calcolo dello spline: infatti la libreria attualmente in uso `scipy.interpolate.CubicSpline` restituisce polinomi di terzo grado, mentre sarebbe preferibile ottenerne di quinto o settimo grado. Un altro possibile miglioramento potrebbe essere l'aggiunta di un tool che consenta di definire le traiettorie in assi diversi da quelli x ed y, per esempio definendone una negli assi x e z (in questo modo il quadrirotore percorrerebbe la traiettoria spostandosi anche verso l'alto ed il basso, invece di rimanere alla stessa altezza). Con uno studio più approfondito potrebbe essere possibile combinare più traiettorie definite in assi diversi, in modo da ottenerne una propriamente tridimensionale. Infine il supporto dell'applicazione potrebbe essere esteso anche a sistemi multi-robot eterogenei, per esempio includendo un pianificatore di traiettorie per robot di terra o estendendo l'attuale tool in modo che consenta la gestione di più robot in contemporanea.

Elenco delle figure

2.1	Struttura del progetto	5
2.2	Logo di Python	5
2.3	Top Computer Languages (June 2022)	7
2.4	Struttura del File System di ROS	10
2.5	Comunicazione fra più Nodi	11
2.6	Forze generate da un quadrirotore	13
2.7	Roll, pitch e yaw di un quadrirotore	13
3.1	Diagramma dei casi d'uso	18
3.2	Diagramma delle classi: Impostazioni	20
3.3	Diagramma delle classi: Traiettorie	21
3.4	Diagramma delle classi: Spline	22
3.5	Diagramma delle classi: ROS	23
4.1	Logo Yaml	28
4.2	Interfaccia grafica: Area traiettorie	29
4.3	Interfaccia grafica: Area spline	30
4.4	Interfaccia grafica: Area ROS	31
4.5	Interfaccia grafica: Impostazioni	32
4.6	Preview traiettoria disegnata e spline	36
4.7	Interpolazione con algoritmo <i>distance_interpolation</i>	36
4.8	Interpolazione con algoritmo <i>time_interpolation</i>	37
4.9	Schema comunicazione con ROS 2	37
5.1	Telecamera ad infrarossi Vicon	42
5.2	Prototipo di quadrirotore con marker	42
5.3	Quadrirotore Crazyflie 2.0	43
5.4	Crazyradio PA	43
5.5	Disegno traiettoria “Unibo”	45
5.6	Spline traiettoria “Unibo”	46
5.7	Simulazione di un quadrirotore in <i>Hover</i>	46
5.8	Quadrirotore Crazyflie in <i>Hover</i>	46
5.9	Percorso seguito dal quadrirotore	47

Elenco delle tabelle

3.1 Requisiti non funzionali	15
3.2 Requisiti funzionali	16
5.1 Stack protocollo di comunicazione Crazyflie	43
5.2 Struttura pacchetto CRTP	44

Bibliografia

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [2] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.
- [3] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.
- [4] Nicolai M Josuttis. The c++ standard library: a tutorial and reference. 2012.
- [5] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.
- [6] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.
- [7] Guido vanRossum. Python reference manual. *Department of Computer Science [CS]*, (R 9525), 1995.

Ringraziamenti

In primis, vorrei ringraziare i mio relatore Prof. Notarstefano; un ringraziamento particolare anche al mio correlatore Lorenzo ed a tutto lo staff del laboratorio CASY.

Vorrei inoltre ringraziare i miei amici e la mia famiglia, per avermi sostenuto durante questo percorso.

Infine un ringraziamento speciale alla mia ragazza Chiara, che mi ha sopportato e costretto a studiare durante questi tre lunghi anni.