UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Controlli Automatici

Studio del sistema operativo ROS e approfondimento della libreria Crazyswarm

Advisor: **Prof. Giuseppe Notarstefano**

Co-advisor: **Ing. Lorenzo Pichierri**

Studente:

Lorenzo Venerandi

Anno accademico 2021/2022

Abstract

Il tirocinio ha come obiettivo lo studio e l'utilizzo di ROS, un software utilizzato per la programmazione ed il controllo dei robot. Durante lo studio di ROS ho approfondito strumenti come i Topic ed i Launch Files.
Insieme agli strumenti di ROS è stato studiato ed approfondito il linguaggio Python, utile per impartire comandi ai robot e studiarne i comportamenti tramite grafici e log.
Il tirocinio si è poi concentrato sullo studio di Crazyswarm, una libreria Open Source utilizzata per il controllo simultaneo di un insieme di droni con l'ausilio di sistemi di Motion Capture; in particolare verranno approfonditi la struttura a stack della libreria ed i protocolli di comunicazione tra il server ed i droni.
La parte finale del tirocinio è dedicata al controllo dei droni, in simulazione e con attrezzatura reale, ed allo studio del loro comportamento nelle varie configurazioni possibili.

Contents

List of Figures

Chapter 1

Introduzione a ROS

1.1 Introduzione

ROS è l'acronimo di Robot Operating System; esso infati è un insieme di software e librerie utilizzati per progettare, sviluppare e controllare dei robot. ROS è un sistema Open-Source e modulare, sviluppato in C++ ma programmabile in Python. Questo facilita notevolmente il lavoro deigli sviuppatori e consente una rapida diffusione di librerie personalizzate, interoperabilità tra sistemi differenti e creazione di implementazioni ad hoc per un dispositivo specifico. Ne è un esempio Crazyswarm [6], che utilizza il sistema ROS per ampliare la libreria di Bitcraze [3] per il controllo dei Crazyflie.

1.2 Strumenti di ROS

ROS è particolarmente utile perché mette a disposizione una serie di strumenti e strutture che facilitano notevolmente la cooperazione di più nodi (processi) o dispositivi.

1.2.1 Nodes

Un nodo di ROS è un processo che esegue delle operazioni, scambia messaggi con altri nodi tramite i Topic e si occupa del controllo dei robot.

1.2.2 Roscore

Roscore è il nodo principale di ROS, necessario per il funzionamento degli altri strumenti e responsabile del mantenimento di Topic e Services.

1.2.3 Topics

Un Topic è un bus dati a cui viene attribuito un nome. Viene creato quando un Nodo lo "pubblica" e all'interno di esso si può comunicare con un solo tipo di Messaggio (deciso dal Nodo che ha pubblicato il topic).

1.2.4 Messages

Un messaggio è una struttura dati utilizzata per comunicare all'interno dei Topics. ROS mette a disposizione dei messaggi predefiniti, ma è possibile definirne di nuovi partendo dai classici tipi primitivi (int, string, float ecc.).

In seguito verranno riportati degli esercizi svolti per prendere familiarità con il sistema ROS ed il liguaggio Python.

1.3 Risposta al gradino di un sistema LTI (ROS)

1.3.1 Introduzione

Lo scopo dell'esercitazione è quello di calcolare la risposta al gradino di un sistema lineare tempo invariante, ripreso dal progetto svolto per l'esame di Controlli Automatici T [7].
Il progetto consiste in un nodo `generator` che prima calcola la risposta al gradino e poi la stampa su un file `data[data-ora].csv` e su un ROS Topic (`StepResponse`).
Tutto viene poi letto dal nodo `plotter` che, mentre legge dal topic, grafica i dati ottenuti utilizzando `matplotlib`.

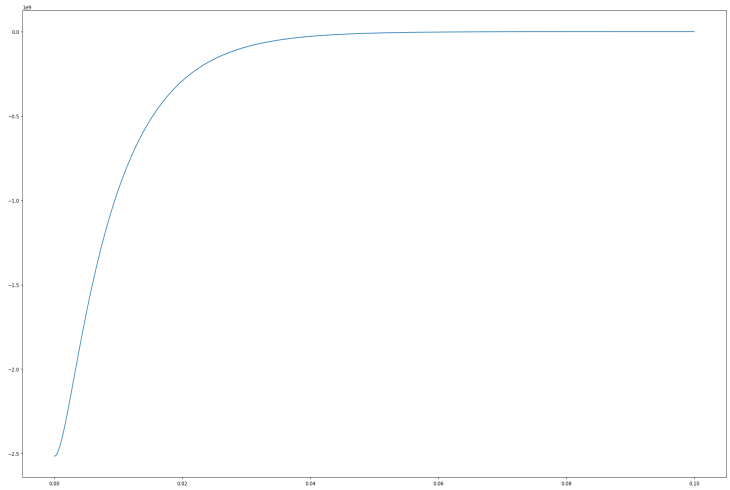

Figure 1.1: Step response di un sistema di un LTI

Il codice relativo a `generator.py`, `plotter.py` e `stepResponse.launch` si trova in appendice A.1.

1.4 3D Trajectory con Matplotlib

1.4.1 Introduzione

Questa esercitazione consiste nel disegnare la traiettoria di un oggetto con matplotlib.
La posizione dell'oggetto viene determinata utilizzando il Vicon, il quale la comunica al nodo plotter mediante un topic.

1.4.2 Struttura

La comunicazione fra le telecamere e il dispositivo in cui viene eseguito lo script avviene tramite un access-point connesso all'hub del Vicon.
Il pc si connette al wi-fi e riesce in questo modo ad accedere al ROS Topic `\vicon\cf1\cf1` (cf1 è il nome assegnato all'oggetto), in cui viene pubblicata la posizione dell'oggetto utilizzando il messaggio di ROS `TransformStamped`.

A questo punto si riesce ad accedere al Topic tramite un Subscriber, leggere i dati del messaggio (sono di nostro interesse i dati traslazionali, che descrivono appunto la posizione nello spazio) ed infine plottarli in un grafico animato utilizzando Matplotlib.
In particolare il grafico mostra la posizione in tempo reale, mantenendo la traiettoria delle ultime 2000 posizioni.

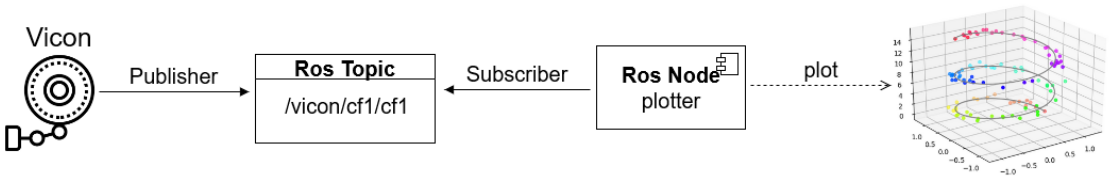

Figure 1.2: Struttura ROS 3d trajectory plot

Il codice è riportato in appendice A.2.

1.5 Posizione nello spazio con Rviz

1.5.1 Introduzione

In questa esercitazione verrà invece visualizzata la posizione dell'oggetto in tempo reale, utilizzando questa volta il tool `rviz`.

1.5.2 Strumenti

La comunicazione fra il Vicon ed il nodo di ROS è la medesima della scorsa esercitazione, la differenza sta nel fatto che la posizione dell'oggetto viene mostrata sul tool rviz mediante un messaggio di ROS `Marker`.
In questo caso infatti il nodo funge sia da Subscriber (al topic `\vicon\cf1\cf1`) che da Publisher (sul topic `visualizer`).
Rviz legge le informazioni dal topic `visualizer` e le mostra sull'interfaccia grafica.

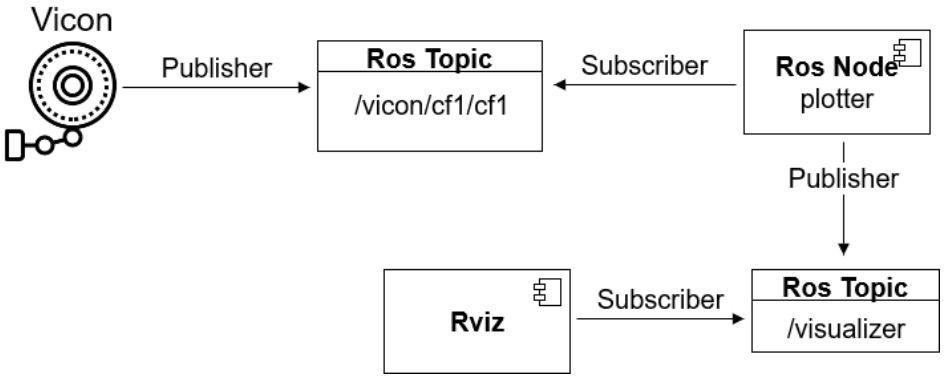

Figure 1.3: Struttura ROS 3d plot rviz

Il codice di `visualizer.py` è riportato in appendice A.3.

Chapter 2

Crazyswarm communication protocol

2.1 Introduzione a Crazyswarm

Crazyswarm [6] è una libreria Open-Source che estende le funzionalità dei Crazyflie (quadricotteri progettati e venduti da Bitcraze) in modo da poter controllare più droni simultaneamente.
In questo capitolo verrà spiegato il funzionamento dei protocolli che Crazyswarm adotta per la comunicazione fra il controller (tipicamente il pc) e i droni.

2.1.1 Architettura di Crazyswarm

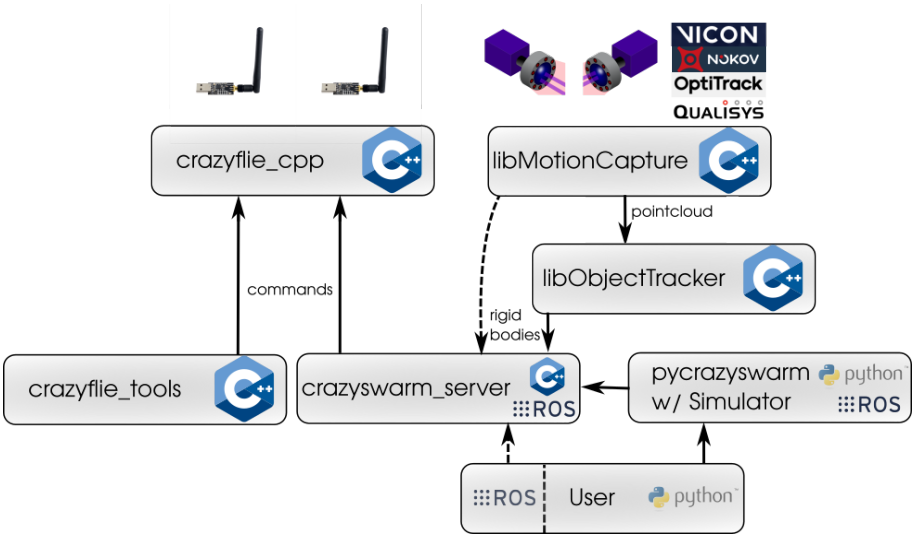

Figure 2.1: Architettura di Crazyswarm

Crazyswarm estende le funzionalità della libreria di Bitcraze integrandola con ROS:

- Implementazione degli script come ROS Nodes.
- Scambio di informazione fra i nodi mediante specifici ROS msg.
- Utilizzo di Topics.

Crazyswarm si distingue dalla libreria di Bitcraze [3] anche nella modalità di diffusione delle informazioni: infatti mentre crazyflie utilizza dei pacchetti indirizzati a singoli droni (multicast) crazyswarm consente l'utilizzo di comandi diretti a tutti (broadcast), dove riesce ad inserire le informazioni relative a più droni contemporaneamente, riducendo così la latenza (esempio: comando di decollo `takeoff` per tutti i droni).

2.2 Crazyflie communication protocol

2.2.1 Introduzione

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	`Livelo fisico di USB e Radio`

I livelli `Link` e `Physical Medium` si occupano di definire il mezzo di comunicazione e i protocolli di basso livello ad esso correlati, astraendoli quindi per 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 [1]

2.2.2 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`

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.

2.2.3 Subsystems

I subsystem vengono utilizzati dai controllori dei quadricotteri 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 drone. 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, tra i quali:
  - getProtocolVersion(): versione del protocollo CRTP in uso
  - getFirmwareVersion(): versione del firmware installata
  - getDeviceTypeName(): tipo e nome del dispositivo (esempio CF20 -¿ Crazyflie 2.0)
  - App Channel: canale utilizzato per comunicare con altre applicazioni

2.3 Implementazione CRTP

2.3.1 Source Code

Lo stack CRTP viene implementato sulla libreria di `crazyflie_cpp` [2], più precisamente sui file `crtp.cpp` e `Crazyradio.cpp`.
Esempio di codice utilizzato per il controllo con un generico setpoint di un Crazyflie:

```
struct crtpVelocityWorldSetpointRequest
{
  crtpVelocityWorldSetpointRequest(
      float x, float y, float z, float yawRate)
      : header(0X07, 0), type(1), x(x), y(y), z(z), yawRate(yawRate)
  {
  }
```

```
  const crtp header;
  uint8_t type;
  float x;
  float y;
  float z;
  float yawRate;
}__attribute__((packed));Velocity
CHECKSIZE(crtpVelocityWorldSetpointRequest);
```

2.3.2 crasyflie_cpp wrapper

A livello pratico non si modifica mai il protocollo `crtp` in caso si voglia aggiungere un'implementazione personalizzata. Viene invece utilizzata la classe wrapper crazyflie.cpp, che si interfaccia automaticamente con i livelli più bassi (`crtp, crazyradio` ecc..):

- Definizione della classe personalizzata su `Crazyflie.h`, esempio:

```
void sendFullStateSetpoint(
  float x, float y, float z,
  float vx, float vy, float vz,
  float ax, float ay, float az,
  float qx, float qy, float qz, float qw,
  float rollRate, float pitchRate, float yawRate);
```

- Implementazione del metodo su `Crazyflie.cpp`:

```
void Crazyflie::sendFullStateSetpoint(
    float x, float y, float z,
    float vx, float vy, float vz,
    float ax, float ay, float az,
    float qx, float qy, float qz, float qw,
    float rollRate, float pitchRate, float yawRate)
{
    crtpFullStateSetpointRequest request(
        x, y, z,
        vx, vy, vz,
        ax, ay, az,
        qx, qy, qz, qw,
        rollRate, pitchRate, yawRate);
    sendPacket(request);
}
```

2.4 Astrazione con ROS

2.4.1 Astrazione crazyflie_cpp

La libreria `crazyflie_cpp` si occupa di astrarre la gestione dei protocolli di comunicazione e il controllo dei droni, in modo da renderli completamente trasparenti all'utente senza compromettere le prestazioni.
Ci sono però dei problemi nell'adottare questo metodo:

- Essendo `crazyflie_cpp` eseguito come un unico processo si creano problemi nel momento in cui si devono eseguire più operazioni in contemporanea, per esempio l'invio di un setpoint al drone mentre viene calcolato quello successivo.
- Python consente di eseguire `crazyflie_cpp` in un simulatore.

Questi problemi vengono risolti utilizzando ROS. Infatti ROS consente di eseguire `crazyflie_cpp` e i protocolli di comunicazione in modo concorrente. ROS fornisce inoltre un'infrastruttura che semplifica la comunicazione fra i nodi, indipendentemente dal linguaggio in cui sono scritti.

2.4.2 ROS msg

Dato che `crtp` ha un approccio stateless e connectionless simile ad UDP (quindi non necessita di stabilire una connessione e non tiene traccia dello stato delle trasmissioni precedenti) è necessaria un'astrazione che consenta il rapido invio di pacchetti senza preoccuparsi della qualità della trasmissione. Questo requisito viene soddisfatto dai messaggi di ROS, che consentono inoltre la comunicazione fra nodi scritti in linguaggi diversi (`C++` per crazyflie_cpp e `Python` per l'interfaccia utente).
È quindi consigliato definire un tipo di messaggio personalizzato in `crazyswarm/msg`, per esempio:

`FullState.msg`

```
Header header
geometry_msgs/Pose pose
geometry_msgs/Twist twist
geometry_msgs/Vector3 acc
```

Una volta definito il messaggio è sufficiente definire l'handler nella classe `CrazyflieROS`, dove è necessario definire il Subscriber, inserirlo all'interno del metodo `run()` e gestire la ricezione dei messaggi.
Un esempio di implementazione si può trovare in appendice A.4.

2.5 Pycrazyswarm

2.5.1 Implementazione messaggio

L'ultimo livello di astrazione avviene mediante la libreria `Pycrazyswarm`, scritta in `Python`.
Per terminare l'implementazione del messaggio è sufficiente abilitare il Publisher all'interno della classe `pycrazyswarm`:

```
from crazyswarm.msg import ..., FullState

class Crazyflie:

        def __init__(...):
                ...
                self.cmdFullStatePublisher = rospy.Publisher(
                        prefix + "/cmd_full_state", FullState, queue_size=1)
                self.cmdFullStateMsg = FullState()
                self.cmdFullStateMsg.header.seq = 0
                self.cmdFullStateMsg.header.frame_id = "/world"
                ...

        ...

        def cmdFullState(self, pos, vel, acc, yaw, omega):
                self.cmdFullStateMsg.header.stamp = rospy.Time.now()
                self.cmdFullStateMsg.header.seq += 1
                self.cmdFullStateMsg.pose.position.x    = pos[0]
                ...
                self.cmdFullStateMsg.twist.angular.z    = omega[2]
                self.cmdFullStatePublisher.publish(self.cmdFullStateMsg)
```

2.5.2 Pycrazyswarm API

Fortunatamente non è obbligatorio definire il proprio messaggio, dato che Crazsyswarm mette a disposizione una ricca lista di comandi utilizzabili per controllare uno o più droni,situati all'interno della API Python.

2.6 References

- CRTP - Communication with the Crazyflie
- Crazyflie c++
- Implementazione di Crazyflie con ROS

- Pycrazyswarm wrapper
- Pycrazyswarm API

Chapter 3

Test con i Crazyflie

3.1 Log Informazioni di un Crazyflie (ROS)

3.1.1 Introduzione

Questa esercitazione consiste nel creare uno script in Python che effettui il log delle informazioni ricevute dal Crazyflie e che le plotti in un grafico.

L'implementazione avviene utilizzando due nodi di ROS, collegati al topic `crazyflie` e che comunicano con un tipo di messaggio personalizzato.

Verranno analizzati sia i log dello stabilizzatore (componente situato all'interno del drone che ricava thrust, roll, pitch e yaw tramite dei sensori) che quelli del controller, calcolati tramite il controllore Mellinguer.
I parametri di Logging sono definiti nelle API Python sul sito di Crazyflie.

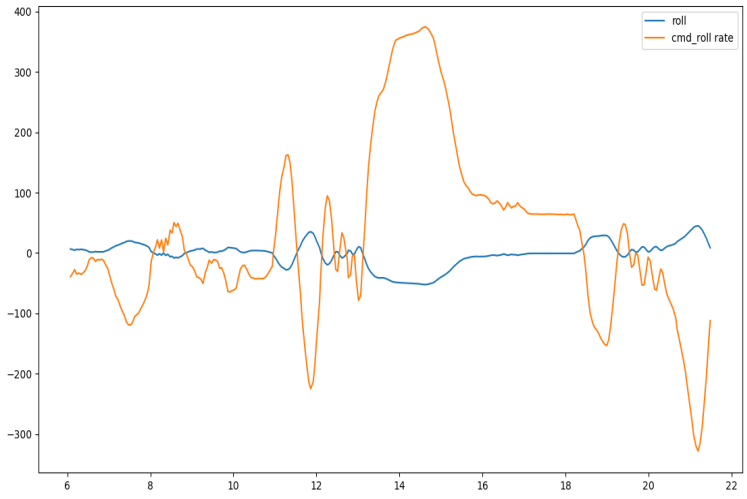

Figure 3.1: Cmd_roll e roll_estimate di un crazyflie single-marker

3.1.2 Mellinguer.msg

```
float32 thrust
float32 roll
float32 rollController
float32 pitch
float32 pitchController
float32 yaw
float32 yawController
```

Nel messaggio è possibile passare entrambi i tipi di log (stabilizzatore e controller)

3.1.3 crazyflie_logger.py

Questo nodo ha molte funzioni:

- Configura il collegamento con il crazyflie.
- Configura il nodo Publisher di ROS.
- Si mette in ascolto per i log del crazyflie.
- Invia i comandi per muovere il crazyflie.
- Pubblica i log sul topic `crazyflie` tramite messaggio `Mellinguer`.

```
import logging
import time
from threading import Event
import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.positioning.motion_commander import MotionCommander
from cflib.utils import uri_helper
import rospy
from laboratorio.msg import Mellinguer


URI = uri_helper.uri_from_env(default='radio://0/110/2M/E7E7E7E708')
DEFAULT_HEIGHT = 0.1
position_estimate = [0, 0]
deck_attached_event = Event()
logging.basicConfig(level=logging.ERROR)

def move_linear_simple(scf):
    with MotionCommander(scf, default_height=DEFAULT_HEIGHT) as mc:
        time.sleep(1)
        mc.forward(0.5)
        time.sleep(1)
        mc.turn_left(180)
        time.sleep(1)
        mc.forward(0.5)
        time.sleep(1)


def log_pos_callback(timestamp, data, logconf):
    msg.roll = data['stabilizer.roll']
    msg.rollController = data['controller.rollRate']

    pub.publish(msg)


def main():

    global pub,rate
    pub = rospy.Publisher('crazyflie', Mellinguer, queue_size=15)
    rospy.init_node('generator', anonymous=True)
    rate = rospy.Rate(3000)
```

```
    global msg
    msg = Mellinguer()



def take_off_simple(scf):
    with MotionCommander(scf, default_height=DEFAULT_HEIGHT) as mc:
        time.sleep(3)
        mc.stop()

if __name__ == '__main__':

    try:
        main()
    except rospy.ROSInterruptException:
        pass

    cflib.crtp.init_drivers()

    with SyncCrazyflie(URI, cf=Crazyflie(rw_cache='./cache')) as scf:

        logconf = LogConfig(name='Position', period_in_ms=50)
        logconf.add_variable('stabilizer.roll', 'float')
        logconf.add_variable('controller.rollRate', 'float')
        scf.cf.log.add_config(logconf)
        logconf.data_received_cb.add_callback(log_pos_callback)

        logconf.start()
        move_linear_simple(scf)
        logconf.stop()
```

3.1.4 Crazyflie_plotter.py

Questo nodo si occupa di leggere i log tramite un Subscriber sul topic `crazyflie` e di plottarli tramite matplotlib.

```
#!/usr/bin/env python3

import rospy
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
from laboratorio.msg import Mellinguer
```

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

def getCords(data):

    roll.append(data.roll)
    rollController.append(data.rollController)

    timeline.append(time.time()-start_time)

    rospy.loginfo(data)

def animate(i,xs, ys, yr):

    ax.clear()
    ax.plot(xs, ys,xs,yr)


def main():

    global roll,start_time,rollController,timeline
    rollController = []
    roll = []
    timeline = []

    start_time = time.time()

    rospy.init_node('plotter', anonymous=True)
    rospy.Subscriber("crazyflie", Mellinguer, getCords)

    ani = animation.FuncAnimation(fig, animate, fargs=(timeline,roll,rollController)
    plt.show()

    rospy.spin()

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

3.2 Test con droni multi-marker e problemi di Crazyswarm

3.2.1 Introduzione

In questa sezione vengono riportati i problemi che sono stati osservati nel testing di droni multi-marker con Crazyswarm. Viene quindi effettuata un'analisi sulle possibili cause di tali problemi confrontando le differenze di implementazione fra i droni con un solo marker e quelli con più marker.

3.2.2 Configurazione strumenti

Per il controllo dei droni viene utilizzata la libreria Crazyswarm, un'estensione di quella adottata da bitcraze che utilizza ROS.
Per individuare il drone viene utilizzato Vicon, un sistema di telecamere ad infrarossi che riesce ad individuare la posizione dei marker (delle sfere riflettenti che riflettono gli infrarossi delle telecamere).
La comunicazione fra il sistema Vicon e il dispositivo in cui è in funzione Crazyswarm avviene tramite una connessione wi-fi. Crazyswarm si connette al Vicon e pubblica i dati relativi alle posizioni nel topic `\tf`.

Configurazione file hover_swarm.launch

```
motion_capture_type: "vicon"
motion_capture_host_name: "192.168.10.1"
enable_parameters: True
enable_logging: True
genericLogTopics: ["log1"]
genericLogTopicFrequencies: [10]
```

Nella configurazione è stato abilitato il log dei parametri del crazyflie (pubblicati sul topic `\log1`).
Inoltre i droni vengono configurati per utilizzare il controllore Mellinguer [5] invece di quello PID in quanto il primo è più adatto a gestire i movimenti rapidi del quadricottero.

3.2.3 Differenze fra single-marker e multi-marker

Crazyswarm implementa tre possibili configurazioni per i marker:

- Single-marker
- Multi-marker (stessa configurazione per tutti i droni)
- Multi-marker (configurazione univoca)

Configurazione single-marker

In questo caso è presente un solo marker al centro del drone. Per utilizzare questa configurazione è necessario modificare alcuni parametri su Crazyswarm, in particolare nel file `hover_swarm.launch`:

```
motion_capture_type: "vicon"
motion_capture_host_name: "192.168.10.1"
object_tracking_type: "libobjecttracker"
```

In questo modo Crazyswarm, invece di recuperare l'oggetto rigido dal Vicon, riceve soltanto la nuvola dei punti (marker) individuati e li associa ai crazyflie utilizzando una libreria esterna (`libobjecttracker`).

Figure 3.2: Crazyflie configurazione single-marker

Configurazione multi-marker

Questa configurazione consiste nel piazzare 4 marker sul drone. In questo modo il sistema di MotionCapture riesce ad ottenere, oltre alla posizione, anche roll, pitch e yaw dell'oggetto (in relazione alla configurazione dei marker salvata nel sistema ViconTraker).
La disposizione dei marker può essere la stessa per tutti i crazyflie, in tal caso Crazyswarm si occupa del riconoscimento dei droni (essi vanno disposti correttamente nello spazio secondo quanto descritto nel file `crazyflies.yaml`).
Se si decide invece di disporre i marker in modo univoco per ogni crazyflie è possibile assegnare ogni disposizione ad un particolare oggetto direttamente

dall'applicazione ViconTraker.
In questo caso sarà Vicon ad occuparsi dell'identificazione dei droni, che potranno quindi essere disposti nello spazio indipendentemente dalle posizioni di partenza definite in `crazyflies.yaml`.
Il secondo metodo è più affidabile e consente di evitare dei problemi legati all'identificazione dei droni di Crazyswarm (per esempio dopo un veloce scambio di posizione di due droni, se la disposizione dei marker è la stessa Crazyswarm potrebbe invertire i due droni).
Le informazioni relative ai droni vengono ottenute dal Vicon utilizzando la libreria `motionCapture`.
Per poter utilizzare la configurazione con più marker devono essere modificate delle righe sul file `hover_swarm.launch`:

```
motion_capture_type: "vicon"
motion_capture_host_name: "192.168.10.1"
object_tracking_type: "motionCapture"
```

Figure 3.3: Crazyflie configurazione multi-marker (univoca)

3.2.4 Comportamento dei droni

Test con drone single-marker

I test sono stati effettuati con crazyflie single marker (cf5) e i log vengono ricavati e graficati utilizzando lo script descritto al punto 3.1.4.
Nel seguente grafico vengono mostrati i log all'esecuzione dello script `hello_world.py`, che consiste nel far decollare il drone fino all'altezza di 20 cm, farlo rimanere in hover per 20 secondi e quindi farlo atterrare.

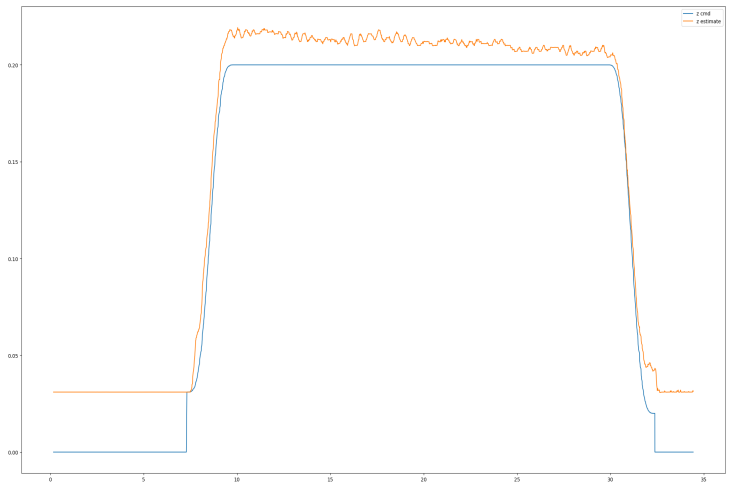

Figure 3.4: cmd_z e z_estimate di un crazyflie single-marker

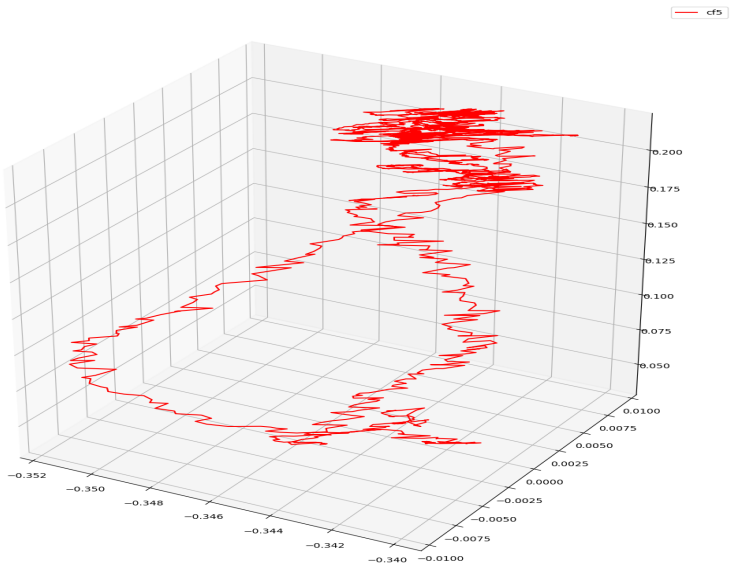

Figure 3.5: Traiettoria reale di un crazyflie - hello_world

Il test ha successo e, come si può notare dal grafico, il drone non ha problemi a mantenere la traiettoria che il controller gli fornisce.
Lo stesso comportamento viene osservato all'esecuzione di altri script.
Di seguito invece i grafici relativi all'esecuzione dello script `figure8_csv.py`, che consiste nel far decollare un drone, fargli seguire una traiettoria a forma di 8 per due volte e quindi farlo atterrare.

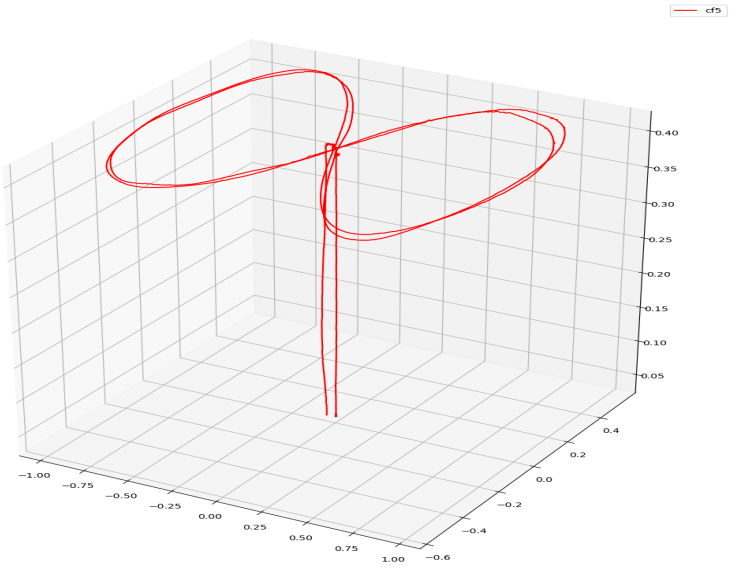

Figure 3.6: Traiettoria reale di un crazyflie - figure8

Come si può notare dal grafico, il drone segue la traiettoria desiderata e non presenta particolari problemi.
Si presentano problemi invece al momento in cui il drone esegue movimenti particolarmente rapidi, in quanto il server di crazyswarm sembra perdere traccia del crazyflie. Probabilmente è un problema collegato alla libreria `libobjecttracker`, utilizzata al posto di `motioncapture` per identificare il drone in mezzo alla nuvola di punti.

Test con drone multi-marker

In questo caso i test sono stati effettuati con un crazyflie (cf8) con una configurazione di 4 marker:

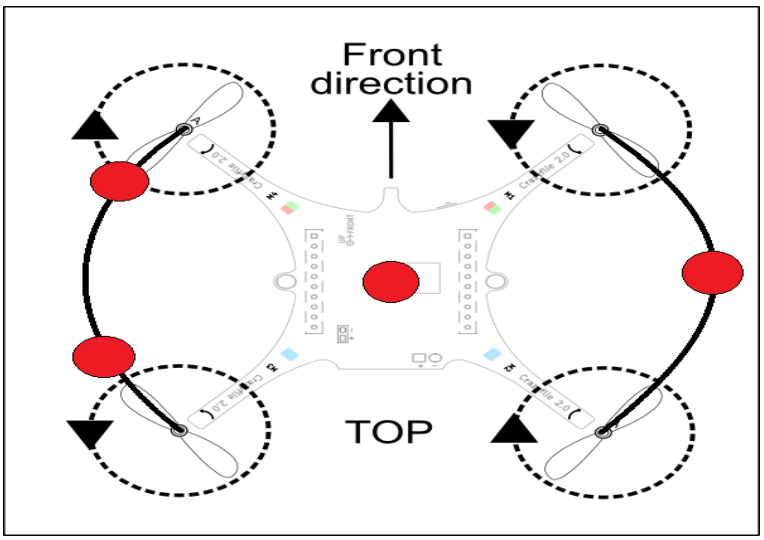

Figure 3.7: Disposizione marker cf8

La posizione e l'orientamento del crazyflie ricavati dal Vicon (utilizzando in questo caso la libreria `motionCapture`) molto spesso presentano valori non consistenti che inevitabilmente causano problemi al controllore.
Questo si manifesta in due modi:

- Il crazyflie si riavvia e smette di rispondere ai comandi
- Il crazyflie termina la sua esistenza contro una parete

Dopo aver effettuato un'analisi delle possibili cause ho individuato il problema nell'implementazione della libreria `motionCapture` [4] nel file `crazyswarm_server.cpp`. Infatti utilizzando la libreria `vicon_bridge` come alternativa a `motionCapture` per ottenere la posizione del crazyflie non si notano errori nel tracciamento del drone.

Purtroppo Crazyswarm non prevede un modo rapido per sostituire `motionCapture` con altre librerie di tracciamento, in quanto essa viene utilizzata direttamente all'interno di `crazyswarm_server.cpp`.

```
std::map<std::string, libmotioncapture::RigidBody>* pMocapRigidBodies,
```

Questo approccio "rigido" semplifica il codice ed aumenta le performance, ma rende complicato un aggiornamento dello stesso e quasi impossibile la sostituzione di un componente senza una modifica di grandi porzioni di programma.

Appendix A

Codice

A.1 Risposta al gradino di un sistema LTI (ROS)

A.1.1 Generator

```
#!/usr/bin/env python3

import rospy
import csv
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt
import control

from laboratorio.msg import Coordinates

def get_step_response():

    xe = [3*(10**7),0,0]
    ue = [364.63]

    W = 8*(10**-5);

    T = np.arange(0,0.1,time_step)

    G=control.tf([3.33*(10**-8),10**-8,7.38*(10**-16)],
                 [1,0.4,0.03,7.386*(10**-10)])
    R = control.tf([3.022*(10**9),10**9],[0.0012,1,0])

    F = control.feedback(G*R,1)

    int,y = control.step_response(F,T,ue+xe,W)

    return int,y


def main():

    pub = rospy.Publisher('stepResponse', Coordinates, queue_size=10)
    rospy.init_node('generator', anonymous=True)
    rate = rospy.Rate(3000)

    # Ottengo la risposta al gradino del sistema
    global time_step
    time_step = 0.00001
```

```
    t,func = get_step_response()

    global msg
    msg = Coordinates()

    # Creazione ed apertura file .csv
    now = datetime.now()
    nomeFile = "data"+now.strftime("[%d-%m-%Y-%H:%M:%S]")+".csv"
    file = open('/home/lore/catkin_ws/src/laboratorio/src/'+
                nomeFile,'w',encoding='UTF8')
    w = csv.writer(file)
    w.writerow(['t','step_response'])


    i = 0

    for value in func:

        i=i+time_step

        # Print to csv
        w.writerow([i,value])

        if not rospy.is_shutdown():
            # print in Topic
            msg.x = i
            msg.y = value
            rospy.loginfo(msg)
            pub.publish(msg)

            rate.sleep()


if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

A.1.2 Plotter

```
#!/usr/bin/env python3

import rospy
```

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from laboratorio.msg import Coordinates

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

y= []
t = []

def animate(i,xs, ys):
    # Disegna dinamicamente xs ed xy
    ax.clear()
    ax.plot(xs, ys)

def getCords(data):
    y.append(data.y)
    t.append(data.x)

def main():

    rospy.init_node('plotter', anonymous=True)
    rospy.Subscriber("stepResponse", Coordinates, getCords)

    # Setup per grafico animato
    ani = animation.FuncAnimation(fig, animate, fargs=(t, y), interval=500)
    plt.show()

    # Per non far terminare il processo
    rospy.spin()

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

A.1.3 Launch file

```
<launch>

  <node pkg="laboratorio" name="generator" type="generator.py"/>
  <node pkg="laboratorio" name="plotter" type="plotter.py"/>

</launch>
```

A.2 3D Trajectory con Matplotlib

A.2.1 3dplot.py

```
#!/usr/bin/env python3

import rospy
import matplotlib.animation as animation
import time
from geometry_msgs.msg import TransformStamped
from matplotlib import pyplot as plt

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.set_xlim(-3,3)
ax.set_ylim(-3,3)

ax.autoscale(enable=None, axis="x", tight=True)

def getCords(data):

    x.append(data.transform.translation.x)
    y.append(data.transform.translation.y)
    z.append(data.transform.translation.z)

    if(len(x) > 2000):
        x.pop(0)
        y.pop(0)
        z.pop(0)

def animate(i,x, y, z):

    ax.clear()
    ax.plot3D(x, y, z, 'red')
    ax.legend(['cf1'])

def main():

    global x,y,z
    x = []
    y = []
    z = []

    rospy.init_node('plotter', anonymous=True)
```

```
    rospy.Subscriber("/vicon/cf1/cf1", TransformStamped, getCords)

    ani = animation.FuncAnimation(fig, animate, fargs=(x,y,z), interval=25)
    plt.show()

    rospy.spin()

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

A.3 Posizione nello spazio con Rviz

A.3.1 visualizer.py

```
#!/usr/bin/env python3

import rospy
from geometry_msgs.msg import TransformStamped
from visualization_msgs.msg import Marker

def getCords(data):

    marker.header.frame_id = 'my_frame'
    marker.header.stamp = rospy.Time.now()

    marker.type = Marker.SPHERE

    marker.pose.position.x = data.transform.translation.x
    marker.pose.position.y = data.transform.translation.y
    marker.pose.position.z = data.transform.translation.z

    marker.action = Marker.ADD
    marker.ns = 'agents'

    marker.id = 32

    scale = 0.2
    marker.scale.x = scale
    marker.scale.y = scale
    marker.scale.z = scale

    color = [1.0, 0.0, 0.0, 1.0]

    marker.color.r = color[0]
```

```
    marker.color.g = color[1]
    marker.color.b = color[2]
    marker.color.a = color[3]

    pub.publish(marker)


def main():

    global marker,pub

    marker = Marker()

    rospy.init_node('plotter', anonymous=True)

    rospy.Subscriber("/vicon/cf1/cf1", TransformStamped, getCords)
    pub = rospy.Publisher("/visualizer",Marker,queue_size=10)

    rospy.spin()

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

A.4 Crazyswarm communication protocol

A.4.1 ROS msg

```
class CrazyflieROS
{
public:
...
        void cmdFullStateSetpoint(
                const crazyswarm::FullState::ConstPtr& msg)
        {
                if (!m_isEmergency) {
                        float x = msg->pose.position.x;
                        ...
                        float yawRate = msg->twist.angular.z;

                        m_cf.sendFullStateSetpoint(x, ..., yawRate);

                        m_sentSetpoint = true;
                }
        }
```

```
...
        void run()
        {
                ros::NodeHandle n;
                ...
                m_subscribeCmdFullState = n.subscribe(
                        m_tf_prefix + "/cmd_full_state",
                        1,
                        &CrazyflieROS::cmdFullStateSetpoint,
                        this);
        }

private:
...
        ros::Subscriber m_subscribeCmdFullState;
...
}
```

Bibliography

[1] nrf24. 2022. Custom Driver per chip radio nRF24. https://nrf24.github.io/RF24/index.html.

[2] Bitcraze. Crtp protocol. 2022. Code at https://github.com/whoenig/crazyflie_cpp/blob/25bc72c120f8cea6664dd24e334eefeb7c9606ca/src/crtp.cpp.

[3] Bitcraze. System overview. 2022. Software available at https://www.bitcraze.io/documentation/repository/.

[4] IMRClab. Libmotioncapture. 2022. Code at https://github.com/IMRCLab/libmotioncapture.git.

[5] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011.

[6] James A. Preiss*, Wolfgang Hönig*, Gaurav S. Sukhatme, and Nora Ayanian. Crazyswarm: A large nano-quadcopter swarm. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3299–3304. IEEE, 2017. Software available at https://github.com/USC-ACTLab/crazyswarm.

[7] Lorenzo Venerandi, Patrick Di Fazio, and Giorgio Mastrotucci. Controllo satellite in orbita intorno alla terra. 2022. Progetto https://github.com/FlippaFloppa/Controlli-Automatici-T.