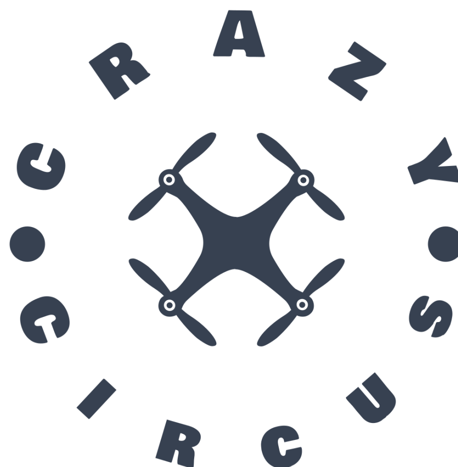
Technical Documentation

CrazyCircus-Group

December 27, 2023

Version 1.0



Status

Reviewed		
Approved		

CrazyCircus-Group
ellge955@student.liu.se

Project Identity

Group E-mail: ellge955@student.liu.se

Homepage: http://www.liu.se/

Orderer: Anton Kullberg, Reglerteknik/LiU
Phone: -
E-mail: anton.kullberg@liu.se

Customer: Daniel Axehill, Reglerteknik/LiU
Phone: +46 13 28 40 42
E-mail: daniel.axehill@liu.se

Supervisor: Joel Nilsson, Reglerteknik/LiU
Phone: -
E-mail: joel.nilsson@liu.se

Course Responsible: Daniel Axehill, Reglerteknik/LiU
Phone: +46 13 28 40 42
E-mail: daniel.axehill@liu.se

Project participants

Name	**Responsibility**	**E-mail**
Elliot Gestrin	Project manager (PM)	ellge955@student.liu.se
Martin Agebjär	Control technology (CT)	marag492@student.liu.se
Hugo Asplund	Hardware (HW)	hugas433@student.liu.se
Marcus Filipsson	Simulation (SIM)	marfi245@student.liu.se
Alvin Gustavsson Vester	Design (DES)	alvgu648@student.liu.se
Albin Helsing	Testing (TEST)	albhe896@student.liu.se
Tomas Röjder	Software (SW)	tomro614@student.liu.se
Adam Simon	GUI/Information (GUI/I)	adasi503@student.liu.se
Axel Stockhaus	Documentation (DOC)	axest416@student.liu.se

CONTENTS

1 Introduction 1
1.1 Purpose and Goal . . . 1
1.2 Concept Descriptions . . . 1
2 Overview of the System 3
2.1 Communication . . . 3
2.2 Hardware . . . 6
3 Modelling of the Crazyflie Drone 8
3.1 Position and Angle Representation . . . 8
3.2 State Space Model . . . 9
4 Description of Subsystems 12
4.1 Graphical User Interface . . . 12
4.2 Commander . . . 14
4.3 Simulation Environment . . . 14
4.4 Motion Planning System . . . 15
4.5 Control System . . . 20
4.6 Sensor System . . . 22
5 Further development 25
5.1 Parameter Estimation . . . 25
5.2 Model validation . . . 25

CrazyCircus-Group
ellge955@student.liu.se

DOCUMENT HISTORY

Version	Date	Changes made	Made by	Reviewed
0.1	2023-12-14	First version	CrazyCircus-Group	CrazyCircus-Group
0.2	2023-12-14	Updated based on feedback	CrazyCircus-Group	CrazyCircus-Group
0.3	2023-12-20	Updated based on feedback	CrazyCircus-Group	CrazyCircus-Group
0.3	2023-12-22	Updated version number	CrazyCircus-Group	CrazyCircus-Group

1 INTRODUCTION

In this document, the technical documentation for the project Crazyflies by CrazyCircus-Group can be found. The focus of this document is to describe how the developed systems and subsystems in the project are created, implemented, and integrated with each other.

1.1 Purpose and Goal

The purpose of the project is to investigate the possibilities of programming drones to perform acrobatic motions. Acrobatic motions can be flips (rotation around a horizontal axis, in place), loops (rotation around a horizontal axis, in a circular trajectory), spins (rotation around vertical axis, in place) or other nontrivially implemented or "awe-inspiring" motions.

The knowledge gained by this project can be used in future educational purposes, and will demonstrate the possibilities of regulating drones to perform specific motions.

1.2 Concept Descriptions

Table 1 and Table 2 presents definitions of terms and symbols used in this document, respectively.

Table 1: Definition of terms

Term	Description
Crazyflie	Crazyflie 2.1 drone developed by Bitzcraze [1].
Crazyradio	CrazyRadio PA developed by Bitzcraze [2].
GUI	Grafical User Interface to interact with the electronics easily.
IMU	Inertial Measurement Unit, combination of accelerometers, gyroscopes.
ROS	Robot Operating System used to build robot applications.
Visionen	Robotics lab at Linköping University.
Qualisys Camera System	A motion capture system inside Visionen, used for positioning of drones [3].
Waypoint	A specific location or point in space that is used in navigation.
Path	A route or course taken by the drone from one waypoint to another.
Trajectory	A predefined path with information of coordinates and angles for the drone to follow.
Flip	Spinning 360 degrees around its own roll and/or pitch axis in mid air.
Loop	Making a 360 degree turn with a given velocity, except it is in the vertical plane instead of the horizontal. Like a flip but in a circular motion.
Acrobatic trick	An acrobatic trick is a visually interesting motion performed by the drone, for example a loop or a flip.
Acrobatic sequence	An acrobatic sequence is a sequence of movements and tricks performed by the drone.
ROS Node / Node	A ROS node or just node is an isolated piece of Python or C++ code able to interact with other nodes and topics.
ROS Topic / Topic	A ROS topic or just topic is a channel where information is sent between ROS nodes. Each topic has a particular format for and information within the messages.

Subscriber / Subscribes	A subscriber is a term for a ROS node which receives information from a ROS topic. When new information is published to the subscriber it automatically performs some operation with this data. A subscriber is said to subscribe to the topic which it receives information from.
Publisher / Publishes	A publisher is a term for a ROS node which sends information over a ROS topic. A publisher is said to publish to the topic which it sends information to.

Table 2: Definition of symbols

Symbol	**Description**
x	Position in x-axis.
y	Position in y-axis.
z	Position in z-axis.
p	The vector $\begin{bmatrix} x & y & z \end{bmatrix}^T$
ψ	Rotation around z-axis (yaw).
θ	Rotation around y'-axis (pitch).
ϕ	Rotation around x''-axis (roll).
α	The vector $\begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^T$
ω	Angular velocity vector ($\dot{\alpha}$).
$\dot{a}$	Time derivative of a variable a.
a_x / a_y / a_z	Component of a vector a in the x / y / z -axis.

2 OVERVIEW OF THE SYSTEM

For a drone to successfully perform acrobatic tricks, a system capable of monitoring the drone, determining actions, and performing them is needed. This makes a complex system, but the entire system does not have to reside within the drone. The drone and its movements can be monitored by external cameras. For this project, the Qualisys motion capture system is used. The calculations needed to determine actions are computed with a computer, though the actions are of course executed by the drone itself. The entire system is therefore separated into multiple subsystems, where each subsystem is responsible for a piece of the process, and that communicates its information with the other subsystems. The system is divided into the following subsystems:

- Graphical User Interface
- Commander
- Simulation Environment
- Motion Planning System
- Control System
- Sensor System

2.1 Communication

The subsystems have to be able to communicate with each other even though they live on different hardware. This means that parts of the code for the communication can be implemented in various languages (e.g. C/C++ or Python) depending on what subsystem it is running on, and which subsystem that part is communicating with. For example, the communication with the control system is on the Crazyflie and is implemented in C/C++, while communication with the simulation environment is on the computer and is implemented in Python. In the latter case, the API for Robot Operating System 2 (ROS2) is used. ROS2 is the framework that is used to send information between subsystems that are not located at the Crazyflie itself. The flow of information between the subsystems is visualized in Figure 1.

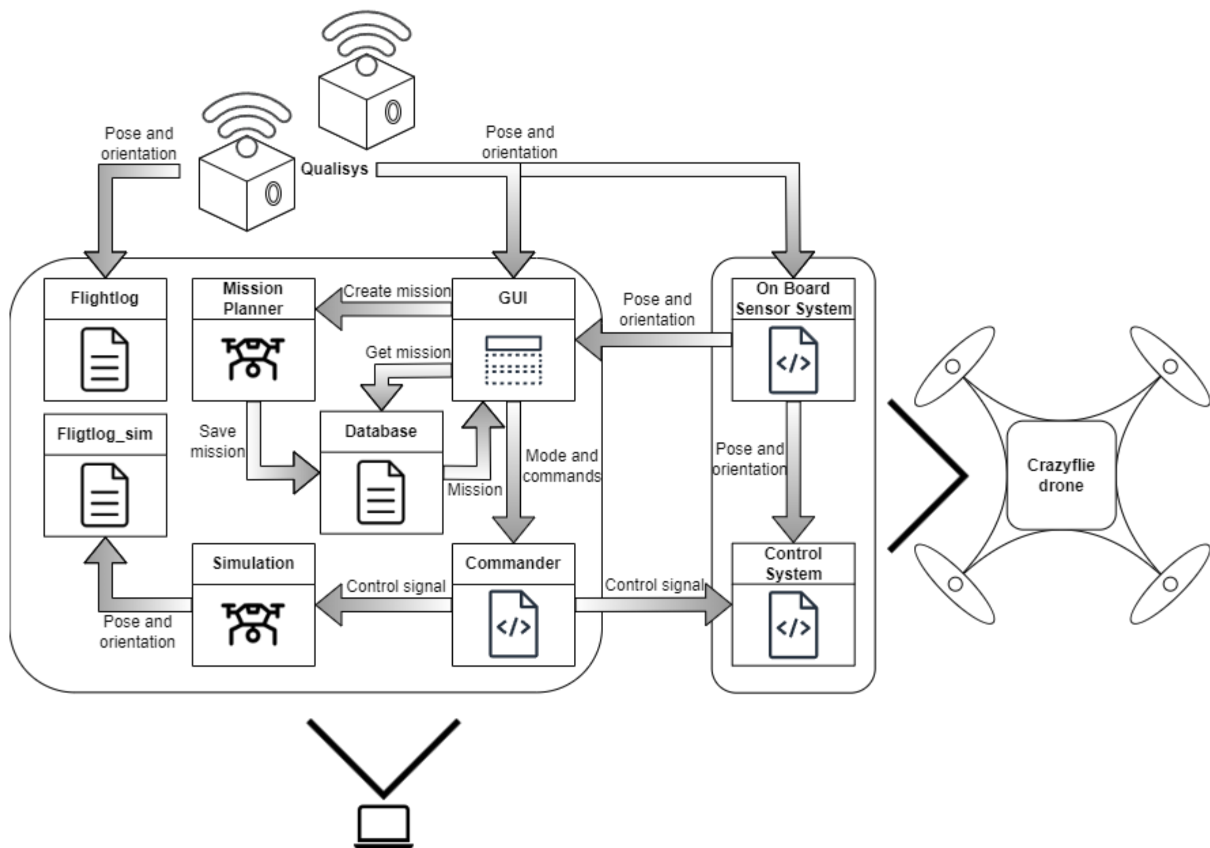


Figure 1: The flow of information between subsystems.

2.1.1 *Interface between subsystems*

This subsection defines the interface between subsystems. All messages, actions, and services are defined in `crazyswarm2/crazyflie_interface`. Each table is a service and contains 3 rows, the first row defines what was requested or what response the service wanted. The *Request* is information being sent to another node and the *Response* is what the sender receives once the receiver node is finished with the calculation. The second row defines the datatype. The third row defines variable names. If the table only defines a message type, the first row will correspond to the message name.

Fullstate is a message type used for both planning and executing commands on the drone. It contains all types of information that the drone uses and is defined according to Table 3. This way one single interface can be used regardless of what control signal is sent to the drone.

Table 3: Definition of FullState message

FullState															
position			orientation				vel			angular vel			acc		
x	y	z	q_1	q_2	q_3	q_4	$\dot{x}$	$\dot{y}$	$\dot{z}$	$\dot{\phi}$	$\dot{\theta}$	$\dot{\psi}$	$\ddot{x}$	$\ddot{y}$	$\ddot{z}$

A Mission message is defined according to Table 4. This message is used within the planner, the GUI, and the commander to execute a sequence of control signals.

Table 4: Definition of Mission message

Mission								
int32	FullState				float32			
rate	$p[0]$	$p[1]$	...	$p[n]$	$t[0]$	$t[1]$	...	$t[n]$

When planning a new mission information is sent between the GUI and the motion planner. Once the planning is initiated, the user can choose between 3 types of missions; circle, loop, and pirouette. This defines the type. For each mission the user is able to input key values that generate new points for the motion planner, these are called *info*. Once the motion planner is finished, it will return a mission. All types of missions use the PlanMission service and can be seen in Table 5.

Table 5: Definition on PlanMission service

Request					Response
string	float32				Mission
type	$info[0]$	$info[1]$	...	$info[n]$	*mission*

Once a mission is planned it can be selected through the GUI's `Choose Mission` button. This causes the GUI to use the GetMission service with the database node. The database node then converts it to a Mission message and returns it to the GUI where it is saved until it can be sent to the commander. The name that is sent in the service is the name of the file. The service is defined according to Table 6.

Table 6: Definition on GetMission service

Request	Response
string	Mission
name	*mission*

Once the `Start` button on the GUI is pressed this will start an "Action". What makes an action unique is that the initiated node will be able to receive feedback from the executing node during the execution of the task. Feedback is currently not implemented but could be useful for further development. Depending on what mode, environment, and mission is selected in the GUI, the information will be sent to the commander and it will then send the correct control signals to the selected environment. The action is defined according to Table 7

Table 7: Definition on RunDrone action

Request			Result	Feedback
string	*string*	*Mission*	*bool*	*int*32
mode	*env*	*mission*	*is_finished*	*current_num*

During operation, data from Qualisys is streamed by a ROS node to the GUI and the Crazyflie (more exactly to the onboard sensor system). The data is then sent according to Table 8. The fused data from the IMU on the crazyflie (see Section 4.6.2) is being sent to the control system, according to Table 9.

Table 8: Package of observed positions

pos			angular pos		
x	y	z	ϕ	θ	ψ

Table 9: Package of fused estimate of state from CF

pos			vel			angular pos			angular vel		
x	y	z	$\dot{x}$	$\dot{y}$	$\dot{z}$	ϕ	θ	ψ	$\dot{\phi}$	$\dot{\theta}$	$\dot{\psi}$

2.2 Hardware

The included hardware in the project is:

- 4 Crazyflie 2.1 (Figure 2)
- 1 Crazyradio 2.0 (Figure 3)
- Qualisys motion capture system (a camera is shown in Figure 4)
- Portable computer running Ubuntu 22.04 LTS and ROS2

Figure 2: Crazyflie 2.1 [1].

Figure 3: Crazyradio 2.0 [2].

Figure 4: Qualisys motion capture camera [3].

3 MODELLING OF THE CRAZYFLIE DRONE

To successfully do acrobatics with a Crazyflie drone, it is important to have a good mathematical model of the Crazyflie. The model is the foundation for both simulation and control. In this section the dynamic model of the Crazyflie will be presented.

3.1 Position and Angle Representation

There are plenty of options when choosing a dynamic model of a vehicle, and one aspect of it is that there are many ways to represent the three-dimensional position and rotation of a rigid body. To represent the position, the obvious choice is three-dimensional coordinate systems. We can define a global coordinate system, consisting of three orthogonal (right-handed) axes x, y and z. We can also define a body-fixed coordinate system, consisting of three orthogonal (right-handed) axes X, Y and Z. Origo, O, of the body-fixed coordinate system is set as the body's center of mass.

3.1.1 *Euler Angles*

The orientation of the body can be represented by Euler angles, more exactly by Tait-Bryan angles or Proper Euler angles. In both cases, the complete rotation of the body is described by three sequential rotations around some axes. For Proper Euler angles, the first and third rotation is around the same axis (which can not be the same axis as for the second rotation). For Tait-Bryan angles, each of the three rotations are around a different axis. The rotations can be intrinsic or extrinsic. Intrinsic rotations are rotations around axes in the body-fixed coordinate system XYZ, and extrinsic rotations are around the fixed global coordinate system xyz. Figure 5 shows an example of intrinsic Proper Euler angles.

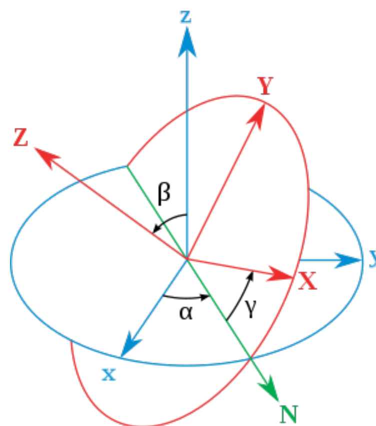


Figure 5: Intrinsic Proper Euler angles z-x'-z''. First, a rotation α around the local (which for the first rotation coincides with the global) z-axis is made. Then a rotation β around the new temporary local x-axis (denoted N in the figure) is made. Lastly a rotation γ around the local z-axis is made.

A common choice of the angles above for an aerial vehicle is the intrinsic Tait-Bryan angles z-y'-x'', meaning the first rotation *yaw* is around the z-axis, the second rotation *pitch* is around the y-axis and the third rotation *roll* is around the x-axis. This way of representing the orientation of a rigid body is used in the project.

3.1.2 *Quaternions*

Another way to represent the orientation of a rigid body is with quaternions. Quaternions are an extension of imaginary numbers to four dimensions. The subset consisting of quaternions of length one (which only has three degrees of freedom) can be used to represent orientation in three dimensions. The representation is not as intuitive as Euler angles but has its benefits. Computations using quaternions are cheaper than computations using rotation matrices with Euler angles. Also, in difference to the Euler angle representation, the quaternion representation does not suffer from Gimbal lock. The quaternion representation is used in the FullState mode, as seen in Table 3.

3.2 State Space Model

The following dynamic model is based on existing work [4][5]. The external force of gravity on the drone is given by

$$F_g = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} \tag{1}$$

where F_g is a force vector in the negative z-direction, m is the weight of the drone and g is the gravity acceleration. The upward thrust force in Newtons generated by each propeller is given by

$$f_i = C_T \omega_i^2, i \in \{1, 2, 3, 4\} \tag{2}$$

where C_T is a thrust coefficient and ω_i is the rotation speed of the i-th motor in revolutions per minute. Assuming the drone is in an upright position, one can sum up the thrust generated from all four propellers, f_i, to form a vector F_{tot} in the Z-axis,

$$F_{tot} = \begin{bmatrix} 0 \\ 0 \\ \sum_{i=1}^{4} f_i \end{bmatrix}. \tag{3}$$

However, the thrust generated by the propellers is not always in the (global) z-axis. The angles of the drone needs to be accounted for. The transformation matrix that connects the moving frame to the fixed frame can be found in three steps. First, a rotation around the z-axis by an angle ψ, followed by a rotation around the y-axis by an angle θ, and finally, a rotation around the x-axis by an angle ϕ (see Section 3.1.1). These rotations are defined as three rotation matrices,

$$R_z(\psi) = \begin{bmatrix} cos\psi & sin\psi & 0 \\ -sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}, \; R_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\phi & sin\phi \\ 0 & -sin\phi & cos\phi \end{bmatrix}. \tag{4}$$

The resulting transformation matrix R is defined as

$$R = R_x(\phi) \times R_y(\theta) \times R_z(\psi). \tag{5}$$

Conclusively equations 1, 2, 3 and 5 can be combined using Euler's first law, which results in

$$m\ddot{p} = RF_{tot} + F_g, \tag{6}$$

where $\ddot{p}$ is the drone's acceleration vector in the global coordinate system. The drone's translation has now been dealt with and the next stage is calculating the angular acceleration for the different angles. The momentum of the drone can be represented as

$$M = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} \frac{d}{\sqrt{2}} C_T \left(-\omega_1^2 - \omega_2^2 + \omega_3^2 + \omega_4^2\right) \\ \frac{d}{\sqrt{2}} C_T \left(-\omega_1^2 + \omega_2^2 + \omega_3^2 - \omega_4^2\right) \\ C_D \left(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2\right) \end{bmatrix}, \tag{7}$$

where M is the resulting momentum vector at the drone's center of mass, expressed in the body fixed coordinate system, d denotes the distance from the drone's center of gravity to the center of each motor and C_D denotes the aerodynamic drag coefficient. The inertia matrix for the drone can be represented and approximated as

$$I = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix} \approx \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}, \tag{8}$$

this matrix is in local coordinates and needs to be converted to global coordinates. The relation can be described as

$$W = \begin{bmatrix} 1 & 0 & -sin(\theta) \\ 0 & cos(\phi) & cos(\theta)sin(\phi) \\ 0 & -sin(\phi) & cos(\theta)cos(\phi) \end{bmatrix}. \tag{9}$$

The final inertia matrix can now be represented as

$$J = W^T \times I \times W, \tag{10}$$

and the Coriolis matrix as

$$C = \frac{d}{dt} J - \frac{1}{2} \frac{d}{d\alpha} \left(\alpha^T \cdot J \right), \quad \alpha = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}. \tag{11}$$

The angular acceleration can then be calculated using Euler's second law and equation 7, 10 and 11,

$$J \cdot \ddot{\alpha} = (M - C \cdot \dot{\alpha}). \tag{12}$$

Lastly the state space vector $x = \begin{bmatrix} p & \alpha & \dot{p} & \dot{\alpha} \end{bmatrix}^T$, can be constructed and updated using our known values $\dot{p}$ and $\dot{\alpha}$ joint with the calculated values $\ddot{p}$ and $\ddot{\alpha}$. The nonlinear state space vector is portrayed in equation 13 below,

$$\dot{X} = \begin{bmatrix} \dot{p} \\ \dot{\alpha} \\ \ddot{p} \\ \ddot{\alpha} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \frac{RF_{tot}}{m} + \frac{F_g}{m} \\ J^{-1} \cdot (M - C \cdot \dot{\alpha}) \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \frac{C_T \left(sin(\phi)sin(\psi) + cos(\phi)cos(\psi)sin(\theta) \right) \cdot (\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2))}{m} \\ \frac{C_T \left(cos(\psi)sin(\phi) - cos(\phi)sin(\psi)sin(\theta) \right) \cdot (\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2))}{m} \\ \frac{-gm + C_T cos(\phi)cos(\theta) \cdot (\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2))}{m} \\ J^{-1} \cdot (M - C \cdot \dot{\alpha}) \end{bmatrix}. \tag{13}$$

The resulting state space vector is in continuous form and needs to be discretized for application to real systems. In the simulation environment this is done by other tools, namely Webots, see Section 4.3. In the motion planning, Section 4.4, a Runge-Kutta method is used. The Runge-Kutta method chosen is RK4, also known as classical Runge-Kutta, which uses a four step linearization [6].

4 DESCRIPTION OF SUBSYSTEMS

The following section describes the different subsystems and their implementation.

4.1 Graphical User Interface

The Graphical User Interface (GUI) is where the user controls the Crazyflie drone. Everything from start, take off and selecting tricks to land and shutdown is done via the GUI. More about its functionality can be read below. The design of the GUI is meant to be user-friendly, however, the priority is functionality. The GUI is implemented in Rviz, a 3D visualization tool for ROS2, together with a control panel.

4.1.1 *Functionality*

The GUI can visualize both the trajectory of the drone and the simulation in real-time. In addition to that, the planned trajectory can also be visualized. The control panel functionality is to control the drone. Here different missions can be loaded and started in different modes, both in real-time and in simulation. It's also possible to plan a completely new mission. Manual control can be started together with a manual controller connected to the computer. There are a few acrobatic tricks that can be started directly from the GUI, Flip and Throwing start. These tricks use an external PID-controller, see section 4.5.2. Lastly, there is also functionality to land the drone at any time (except for during tricks), cancelling future actions, and emergency stop the drone, shutting off all power.

4.1.2 *Communication with other subsystems*

The GUI serves as the primary interface between the user and the system, responsible for capturing user input and translating it into actions. Through the GUI, the user can plan mission by pressing the plan mission button. In the following pop-up windows the user can choose what type of mission and key values to design the mission. This will sent to the Motion Planning system through the /PlanMission service.

When choosing a mission the /GetMission serivce will be called and a Mission will be returned from the database-node. The resulting_mission will be published on the /planned_mission topic.

During the flight, the RVIZ also interacts with the PathPublisher node to displays the current measured position by Qualisys and the current state estimation by the drone. This data is used for visualization. The ROS topics and services can be seen in Table 11.

Table 10: GUI interface

	Name	**Format**
Subscribe	/path_topic	nav_msgs/msg/Path
Subscribe	/pose_topic	geometry_msgs/PoseArray
Subscribe	/planned_path_topic	nav_msgs/msg/Path
Subscribe	/planned_pose_topic	geometry_msgs/PoseArray
Publish	/planned_mission	Table 4
Request	PlanMission_srv	Table 5
Request	GetMission_srv	Table 6
Request	RunDrone_action	Table 7

4.1.3 *Design*

An overview of the GUI design can be seen in Figure 6.

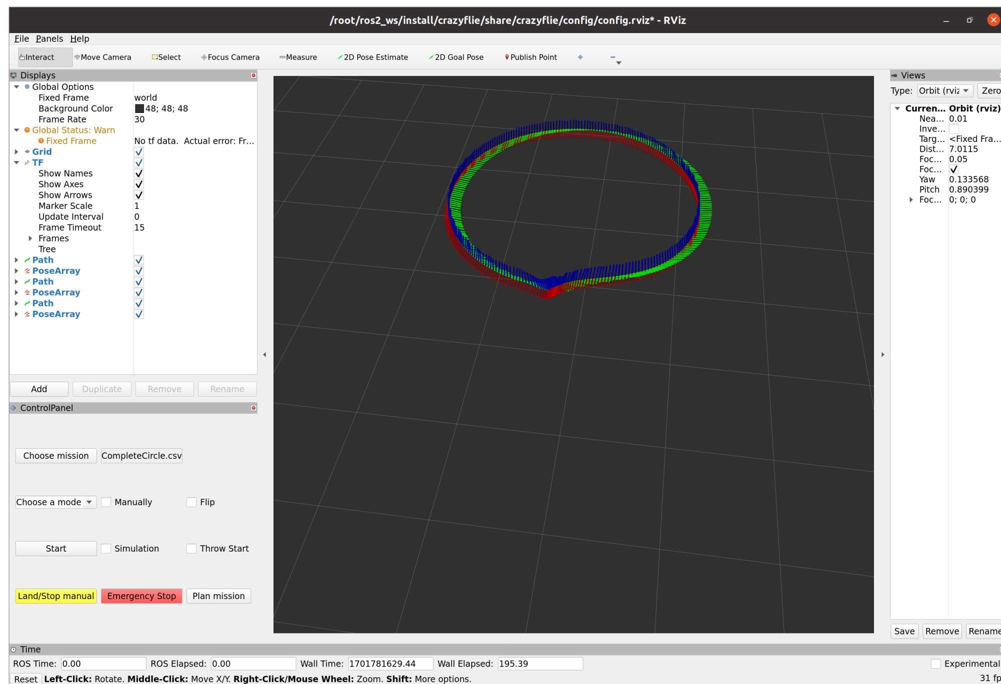


Figure 6: Design of the Graphical User Interface.

4.2 Commander

Commander is an auxiliary node used when the GUI sends a mission to either the Simulation Environment or the drone. The commander initiates the drones on start up. If testing of the system is done without a drone connected the variable Drone needs to be set to false in main() function of commander node. By default the commander sets the drone into rate mode which changes the behavior of control signal crazyswarm/cmdVel. Now the drone will assume only angle rates for this control signal. When the GUI initiates a mission the interface between the GUI and Commander is always the same. This way every mission can be used with every control signal. Therefore the commander handles all the logic and send the correct signals to the chosen environment. The Commander can, for example, choose operational modes such as manual operation, simulation or autonomous operation. If the simulation is selected, based on the chosen mode, the Commander sends the stream of control signals by publishing them on the respective topic.

4.2.1 *Communication with other subsystems*

The commander communicates with the GUI, simulation and the drone. Depending on what is chosen in the GUI the commander will send different type of signals to the environment.

Table 11: GUI interface

	Name	**Format**
Publish	/goTo	crazyswarm/goTo
Publish	/cmdFullState	Table 3
Request	/cmdRate	crazyswarm/cmdVel
Response	RunDrone_action	Table 7

4.3 Simulation Environment

The simulation environment is built in Webots - an open-source robot simulator. The dynamic model used in the simulation is taken from Bitcraze [7]. The model parameters include mass, inertia matrix, and thrust coefficients, and are taken from a system identification of the Crazyflie 2.0 [8]. The mass and inertia matrix are taken directly, and the thrust coefficients have been set according to the knowledge that the Crazyflie can precisely hover with double its weight when using max thrust, as is stated by the system identification.

It was planned to take advantage of "software-in-the-loop" and its benefits for the controller of the Crazyflie in simulation. This would be done by using the actual controller in the firmware, on the real Crazyflies, with Python bindings. Much effort was spent in trying to make such a solution work. Unfortunately, the result was an unstable simulation, probably due to timing issues in the controller caused by the fact that the controller is developed for the actual Crazyflie hardware.

A separate implementation of the controller was therefore created for the simulation, translated from the firmware C-code to a Python script. More specifically, the angular velocity PID-controller part of the firmware controller was implemented. The simulation environment is shown in Figure 7.

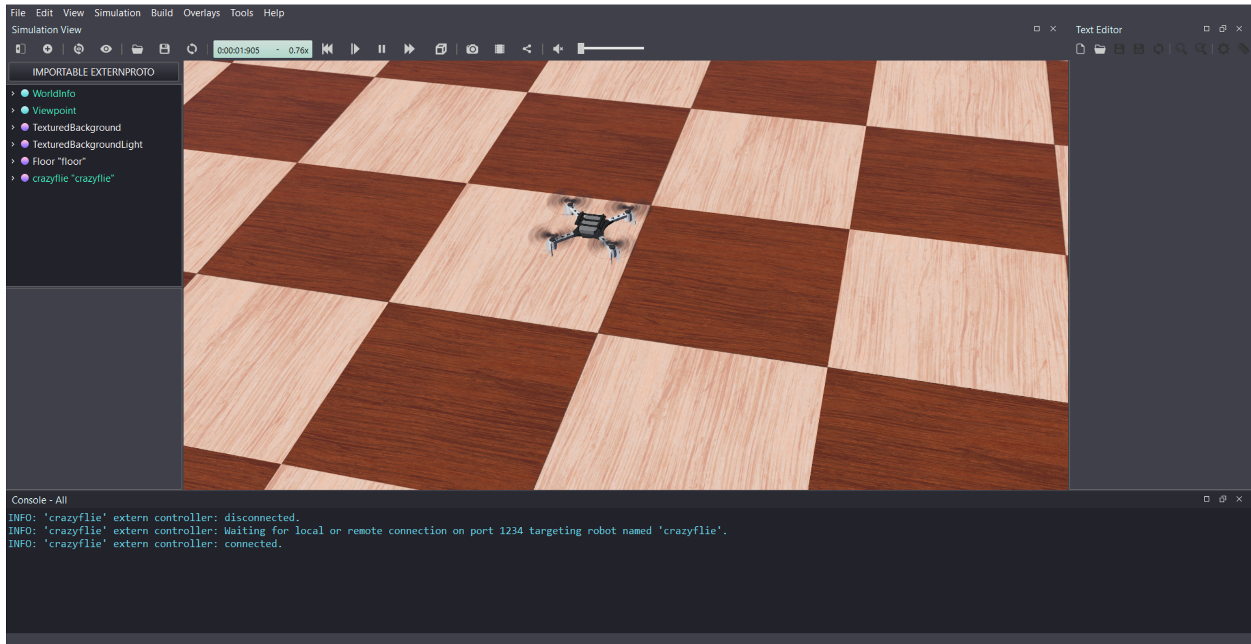


Figure 7: The simulated Crazyflie in the simulation environment.

4.3.1 *Communication with other subsystems*

The simulation environment subscribes to the topic /cmdFullState_sim from where it can receive a stream of angular velocity states. The ROS topic is listed in Table 12.

Table 12: Simulation interface

	Topic	**Format**
Subscribe	/cmdFullState_sim	Table 3

4.4 Motion Planning System

By modeling the Crazyflie's physical characteristics, feasible trajectories can be calculated, along with corresponding control signals. For details on the physical model used, see Section 3.2. The motion planning system implemented in the CrazyCircus project is based on the work by Oliver Ljungberg [4].

To plan a trajectory, a sequence $\tilde{X}$ of P full state points, $\tilde{x}_p \in \tilde{X}$, is given. This sequence is what the motion planner will try to replicate. Each such point defines the following twelve parameters: $[x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi}]$. In addition to this, a time step $\tilde{T}$ between the points is specified. This means that the goal is for the drone to be close to point $\tilde{x}_p$ after $p\tilde{T}$ seconds. The target time for point $\tilde{x}_p$ is denoted $\tilde{x}_p^t$.

Based on the target trajectory $\tilde{X}$ and the physical transfer function f a sequence of sub-optimisation problems is created.

$$\begin{aligned} &\min_{X,U} \; g(X, U, \tilde{X}, i, N) \\ &\text{subject to } x_{k+1} = f(x_k,\; u_k),\; \forall\, k \in i, ..., i+N. \\ &\qquad\qquad x_i = x_i' \\ &\qquad\qquad u_k \in [0, u_{max}],\; \forall\, k \in i, ..., i+N. \end{aligned} \tag{14}$$

The optimization variables in (14) are the sequence of states X and the sequence of control signals U, for which $x_k \in X$ and $u_k \in U$ are the corresponding samples. Observe that U aren't the PWM signals but are instead the approximated motor RPMs, though these are connected by an injective function [9]. The samples are simulated with a time step of T, resulting in the k:th state and control signal occurring after kT seconds. The time for the state x_k and control signal u_k are denoted x_k^t and u_k^t respectively. By default, $T = \frac{1}{50}s$, which matches the sample frequency of $50Hz$ from the Qualisys system and results in $|X| = |U| = P * \frac{\tilde{T}}{T}$ simulated points. Note that f is the Runge-Kutta estimation of the dynamics from 3.2 and exactly specifies the simulated, physical behavior of the system. Therefore, the constraint $x_{k+1} = f(x_k, u_k)\ \forall\ k \in i, ..., i+N.$ together with $x_i = x_i'$, where x_i' is the initial state for the subproblem, restricts the optimization to one over only the sequence U. This initial state is either an input, for the first subproblem, or a result of previously iterated subproblems. Observe further that u_k is bounded to the physical limits of the Crazyflie by the u_{max} term, the value of which is approximately $4.7e8$.

Further note that (14) only considers a window of N samples, where by default $N = 50$. The entire sequence U, and thereby X, is then solved through the following iterative process. After solving the j:th sub-optimisation problem, with $i = j$, the system fixes $x'_{j+1} = x_{i+1}$. Then it solves the $j+1$:th sub-optimisation problem by setting $i = j+1$ in (14). As each sub-problem is only able to alter the variables within its corresponding window, the solution to the entire sequences X and U are iteratively built with each sub-problem providing a single chosen x_i and u_i which is fixed for all future iterations. Therefore, (14) is solved $|X|$ times, once for each simulated point. Each iteration also uses the results from the last iteration to initiate the solver, providing a large speed up as this initial guess is likely good.

An example of a trajectory generated by iteratively solving sub-problems as described above is shown in Figure 8.

Figure 8: A planned trajectory. The target path is shown in orange and the planned trajectory in blue.

g in (14) is the goal or cost function that is minimized. There are currently two options implemented in the CrazyCircus project. The first is the so-called Legacy goal function which is shown in (15). The second is the Interpolated goal function which is shown in (16).

$$g_{Legacy}(X, U, \tilde{X}, i, N) = \sum_{k=i}^{i+N} \Big[(x_k - \tilde{x}_p)^T Q (x_k - \tilde{x}_p) + (u_k - u_{hover})^T R (u_k - u_{hover}) \Big] \tag{15}$$

Where:

$p \in \{1, ..., P\}$ minimises $|x_i^t - \tilde{x}_p^t|$

The Legacy goal function shown in (15) consists of two parts. The first part is the state loss, which aims to minimize the distance between $\tilde{x}_p$ and all the considered $x_k \in X$. The point $\tilde{x}_p$ is the point in $\tilde{X}$ which is closest in time to the point x_i, and all $x_k \in X$ aim to be close to this same $\tilde{x}_p$. Q is a matrix which specifies how heavily deviation in X from $\tilde{x}_p$ is punished. The default content thereof is found in Table 13. The second part is a regularising term. u_{hover} is the constant control value which, if held for all motors, leads to a constant, stationary hover for the drone. Deviation from this is punished at a rate controlled by the matrix R, the default content of which are found in Table 14.

Note that the Legacy cost function results in a system that doesn't consider future objective points. The objective is only for all considered x_k to be close to the chosen $\tilde{x}_p$ which can change suddenly. As such, the planned path might halt at a point in $\tilde{X}$ if it reaches it "ahead of time", to then suddenly rush off toward the next point when the objective function switches. This behavior can be mitigated through the use of more frequent points.

$$g_{Interpolate}(X, U, \tilde{X}, i, N) = \sum_{k=i}^{i+N} \left[\frac{\sum_{p=1}^{P} w_{k,p} (x_k - \tilde{x}_p)^T Q (x_k - \tilde{x}_p))}{\sum_{p=1}^{P} w_{k,p}} + (u_k - u_{hover})^T R (u_k - u_{hover}) \right] \tag{16}$$

Where:

$$w_{k,p} = 4\sigma_{k,p}(1 - \sigma_{k,p})$$

$$\sigma_{k,p} = \frac{1}{1 + e^{-h(x_k^t - \tilde{x}_p^t)}}$$

The Interpolated cost function is displayed in (16). The difference to the Legacy function shown in (15) is that each point $\tilde{x}_p \in \tilde{X}$ is always considered, and the subsequent addition of the weights $w_{k,p}$. Note that h is a hyperparameter with a default value of 15. Larger values result in a reduced weight for points further in time from x_k.

The weights $w_{k,p}$ lead to an interpolation between all the points $\tilde{x}_p \in \tilde{X}$. The largest weight for point x_k is given to the objective points that are the closest to it in time, but all points get some effect. Due to this, the optimization looks ahead to future objectives, and smoother plans are made. Generally, these plans also use less deviating controls than the Legacy version, since the goal function doesn't suddenly change. However, the optimization process is far slower than when using the Legacy function.

A comparison between the trajectories planned using the Legacy goal function from (15) and the Interpolated goal function from (16) is shown in Figure 9. Note that only some states are displayed.

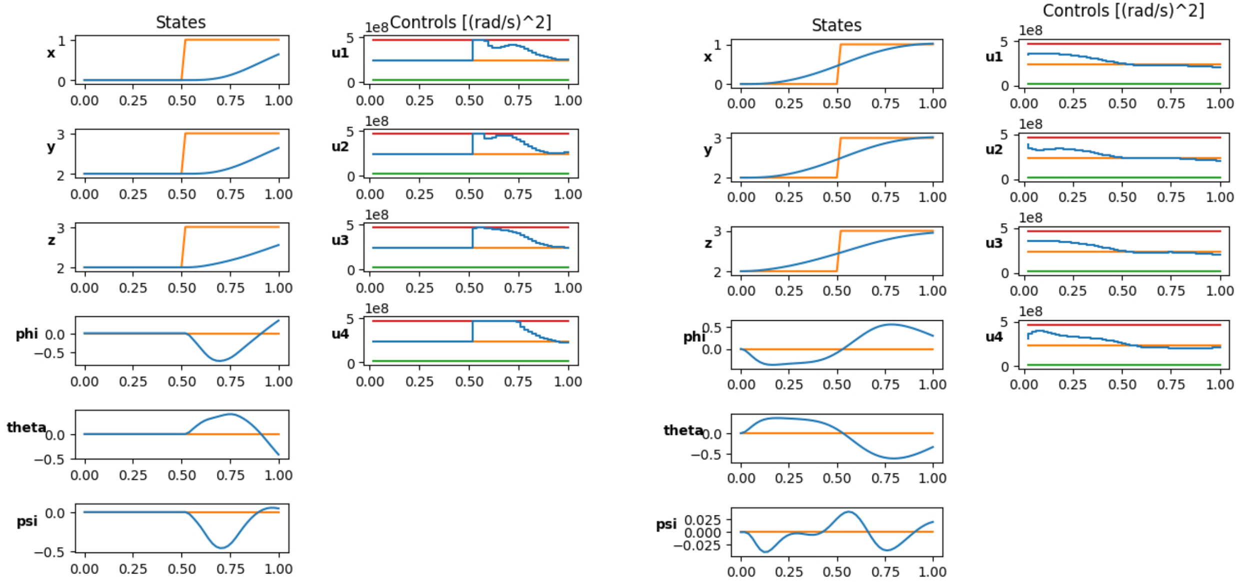


Figure 9: A comparison between the result of using "Legacy" and "Interpol" goal functions to solve a trajectory for travel between two points.

It's also noteworthy that there exists a third implemented way of finding the optimal trajectory. Instead of solving the optimization problem from (14) for each point $x_k \in X$ the so-called Interpolated-Skip method skips solving equation (14) for the last $N-1$ points in X. Instead it uses the prediction from the last iteration for these points. This means that instead of solving (14) $|X|$ times, it's solved $max(|X|-N+1,1)$ times. This gives a roughly linear reduction in time spent, and allows for short plans with $|X| \leq N$ to be solved in a single iteration, far faster than even the Legacy method above. However, the Interpolated-Skip method can only be used if the Interpolation goal function from (16) is used however. This is due to Legacy not considering future points, as discussed above. The results are also somewhat worse than just using the Interpolated cost function as the final sub-problem is harder to solve well than the $N-1$ subproblems avoided.

Table 13: Default Q Values.

	x	y	z	ϕ	θ	ψ	$\dot{x}$	$\dot{y}$	$\dot{z}$	$\dot{\phi}$	$\dot{\theta}$	$\dot{\psi}$
Value	1e10	1e10	1e10	8e8	8e8	8e8	1e2	1e2	1e2	1e2	1e2	1e2

Table 14: Default R Values.

	u_1	u_2	u_1	u_4
Value	2.13e-8	2.13e-8	2.13e-8	2.13e-8

4.4.1 *Communication with other subsystems*

The motion planning can be used in one of two ways.

It can be used directly in Python code with full control of the target state defined. For details on how to use this, see the CrazyCircus User Manual. When used in this way, it doesn't receive anything directly from the other subsystems.

The alternative is to use the motion planner through ROS, though this only allows for the creation of simpler trajectories based on motion primitives. See Table 15 for the topics used.

After the optimization process, the motion planner saves a CSV file, containing the optimized state and control signals to disk. This can later be read into the GUI, see Section 4.1.

Table 15: Motion planing interface

	Service	**Format**
Response	/GetMission_srv	Table 5

4.5 Control System

The control system regulates the Crazyflie drone to follow predefined trajectories. The control system utilized in the Crazy Circus project consists of both an onboard controller and a custom-made controller running on an external computer.

4.5.1 *On board controller*

The default controller on the Crazyflie is a cascaded Proportional-Integral-Derivative (PID) controller. Four types of states are measured and used in control; position, velocity, attitude, and attitude rate. These are controlled in cascade, where the innermost and fastest loop controls the attitude rates, resulting in the desired thrusts for roll, pitch, yaw, and height. These thrusts are handled by the power distribution of the motors, as shown in Figure 10.

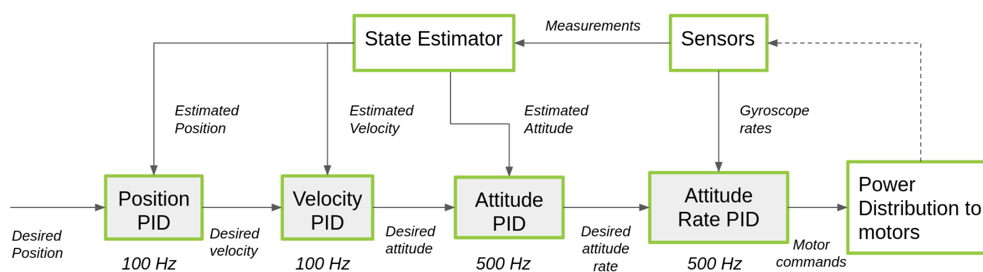


Figure 10: Cascaded PID controller.

Desired state setpoints are sent to the onboard controller using a few different high-level commands, together with a selected mode. While the trajectory generated by the motion planning system includes position, velocity, acceleration, attitude, and attitude rate, the default onboard controller cannot intelligently utilize the extra information the full-state

adds. The types of states (i.e., position, attitude, or attitude rate) used as target setpoints in the controller are selected based on the type of higher-level command used and the selected mode. Depending on the types of setpoints sent, the setpoints enter different parts of the cascaded PID.

The default PID does not utilize the extra information the full-state adds, but another controller called the Mellinger controller is also available in the default firmware for that purpose. The Mellinger controller is also a PID controller but considers most of the entire full-state vector. The full-state setpoint is particularly feasible for smooth maneuvers where feedforward inputs for acceleration and angular velocity play a vital role in achieving precise tracking performance. However, the potential of the Mellinger controller for executing acrobatic maneuvers was not fully investigated.

By selecting a high-level command (i.e., *cmdfullstate* or *cmdvelocity*) and a mode (i.e., *absolute* or *velocity*), the target setpoints returned from the motion planning system (see Section 4.4) are streamed to the cascaded PID controller. This enabled the controller to follow a predefined trajectory. However, it turned out that the onboard controller is not robust or tuned enough to follow aggressive or acrobatic maneuvers.

4.5.2 *Acrobatic PID Controller*

To enable the drone to execute an acrobatic flip, a custom-made controller is developed in Python on an external computer. The controller's input is its horizontal position obtained from Qualisys, along with its orientation acquired from the drone's internal Kalman estimation. The controller outputs a desired attitude rate that the onboard controller will attempt to achieve, along with a fixed thrust selected by the user. When enabled, the controller stabilizes the drone by controlling the roll and pitch angles (ϕ and θ) to zero. To eliminate any horizontal drift caused by the drone not reaching the exact desired attitude, a horizontal position (x- and y-position) PID can be enabled as well. The position PID will adjust the desired attitude based on the horizontal position error, resulting in a cascaded PID shown in Figure 11.

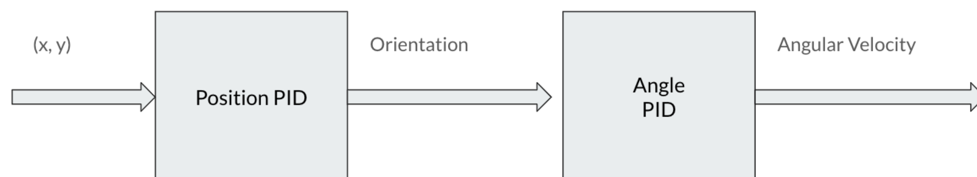


Figure 11: Custom acrobatic PID controller.

The acrobatic PID controller is tuned differently for various purposes. Below are our use cases listed.

- **Launching the drone:** When the drone is launched straight up as preparation before a flip, both the position and angle PID are activated and tuned to keep its horizontal position fixed.
- **Making the flip:** The actual rotation around the drone's x-axis is accomplished by setting a certain angular rate through the command *cmdvelocity* that the onboard controller will attempt to achieve. The rotation is interrupted when a threshold roll angle is detected, indicating that the drone has completed almost a full rotation.
- **Catching the drone:** After a flip, the acrobatic PID controller is again activated to catch the drone. In the catching stage, the position PID is deactivated, and the angle controller is tuned to eliminate overcorrections

and unstable behavior. This implies that some drifting can be expected. The catching stage is interrupted when a velocity in the positive z-direction is detected, indicating that the drone is sufficiently stable and caught.

- **Executing a throwing start:** During the execution of a throwing start, the drone first enters a standby mode where the motors turn slowly while being held, indicating that the drone is ready. Once a threshold acceleration in the negative z-direction is detected (indicating that the drone has been thrown), the acrobatic PID is activated to stabilize the drone in the air. The same configuration is used as when catching the drone after a flip.

4.5.3 *Communication with other subsystems*

The controller requires knowledge of the current position and the target state to calculate a control signal. During real-time flight, the controller runs on the Crazyflie and receives its current measured state directly from the onboard sensor system, see Section 4.6.2.

While following a trajectory, the control system also receives a continuous stream of control signals by subscribing to the corresponding topic. This stream provides the current reference position. The ROS topics can be found in Table 16.

Table 16: Controller interface

	Topic	**Format**
Subscribe	/goTo	crazyswarm/goTo
Subscribe	/cmdFullState	Table 3
Subscribe	/cmdRate	crazyswarm/cmdVel

4.6 Sensor System

The sensor system is split into two parts, the Qualisys system (the outer sensor system) and the on board sensor system (the inner sensor system).

4.6.1 *Qualisys*

The system that is used for measuring the position of the drone is a Qualisys Motion Capture system. The Qualisys system uses 20 Qualisys IR cameras. The Crazyflie is equipped with active IR markers that emit LED light. These markers are designed to be detected by the Qualisys cameras and the Crazyflie can, if needed, also be equipped with passive markers. The passive markers work by reflecting IR flashes from the cameras which can then be detected to estimate position and orientation. The Qualisys system is capable of capturing low-latency 6DOF positioning data in real-time to estimate the state of the Crazyflie. The data is streamed in real-time using the Qualisys Track Manger software and published on the topic /poses. The measured position is received by the GUI as well as the onboard control system on the Crazyflie by subscribing on /poses. There are two ways to obtain the pose from the Qualisys. One with a rotation matrix and the other one that represents the orientation as Euler angles. Both have their advantages and disadvantages. Euler angles are intuitive and provide an uncomplicated representation of orientation. However, they have a problem with singularity. Rotation matrices are a better choice for computational purposes since it does not have problems with singularity.

The Qualisys-node publishes its measured positions to a ROS topic according to Table 17.

Table 17: Qualisys interface

	Topic	Format
Subscribe	/poses	Table 8

4.6.2 *On Board Sensor System*

The onboard sensor system fetches measurements from the Inertial Measurement Unit (IMU) located on the Crazyflie. It fuses the received position estimate (acquired from Qualisys through /poses) with the sensor data from the IMU to make a final estimate of its state. It then sends that estimate directly to the control system, and to the external control system, the acrobatic pid, by submitting on the topic /est_states. The ROS topics can be found in Table 18.

Table 18: On board sensor system interface

	Topic	Format
Subscribe	/poses	Table 8
Publish	/est_states	Table 9

4.6.3 *Sensor Fusion*

Qualisys is an exceptionally accurate and reliable system for estimating position, and therefore it is used when estimating both position and orientation. However, in situations where the data from Qualisys is likely to be flawed, the IMU data is used as the only source for estimating orientation. An example of this is when a flip is performed. In the top of the flip where the drone is upside down, the Qualisys system loses track of the markers of the drone. Therefore the IMU is solely responsible for the orientation estimate and this is the only source of orientation used to control the main part of the flip. The filter that is responsible for the fusion of the IMU data and the state estimation from the Qualisys system is an Extended Kalman filter. The extended Kalman filter is an improved version of the traditional Kalman filter that can handle non-linear filtering. The extended Kalman filter linearizes non-linear functions using Taylor series around the mean. The Kalman filter used for this project is shipped with the Crazyflie drone and uses both the drone's internal sensors, mainly the IMU, and external sensors, for this project the Qualisys system, to estimate the various states. See Figure 12.

4.6.4 *Firmware updates*

The goal was to do minimal changes to the firmware, but some still needed to be done. The firmware has a safety-system which allow for emergency signals to be received which kills power and locks the drone for further commands until a reboot is made. This functionality is incredibly important when testing to prevent big crashes and burned motors. The firmware itself triggered this emergency state with a function called isTumbledCheck. The function essentially checks when the drone is tumbling based on tilt angle and acceleration over time, then it triggers the emergency stop if these conditions are fulfilled. This limits the drone from performing acrobatic flying such as flips and loops when the drone is upside down. In the project this is disabled by modifying the isTumbledCheck function to always returning false, however this leads to a need for manual trigger of the emergency stop and vigilant users.

Another tweak made to the firmware is in the power distribution logic. When power to each motor is calculated the base thrust is based on the desired acceleration and a term is then added or subtracted from each thrust to provide the desired rotation. In the edge case where the calculated thrust to a motor is above max thrust (2^{16}), all motor

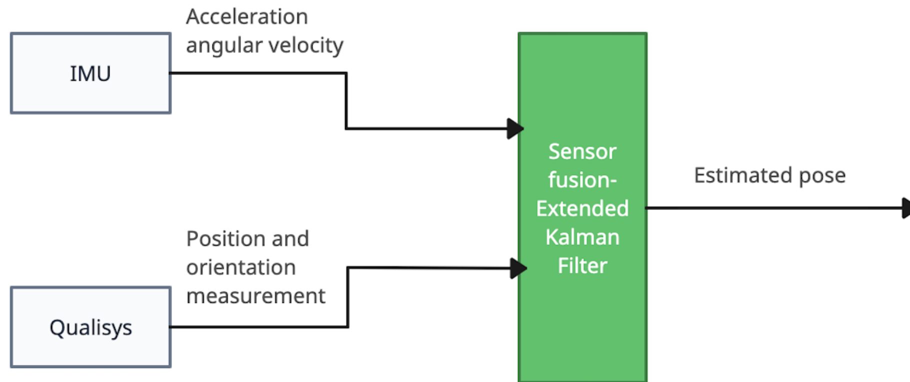


Figure 12: The sensor fusion process.

thrust values are compensated down with the amount above max thrust, r, see figure 13. This means that the power distribution prioritises desired rotational acceleration rather than general thrust. A similar edge case can also be reached when at low thrusts, for instance if we give a low thrust value but still want to rotate fast. This is the case when trying to do a loop. In this case the default firmware will simply cut each thrust at a minimum value, the idle thrust, which limits the possibility to rotate fast with low thrust. The firmware was changed so that it applies the same principle of compensating the thrust on all motors when one is above the max to the case where a thrust is below the minimum. This allows for fast rotation regardless of thrust, but leads to an increase in total thrust in these edge cases.

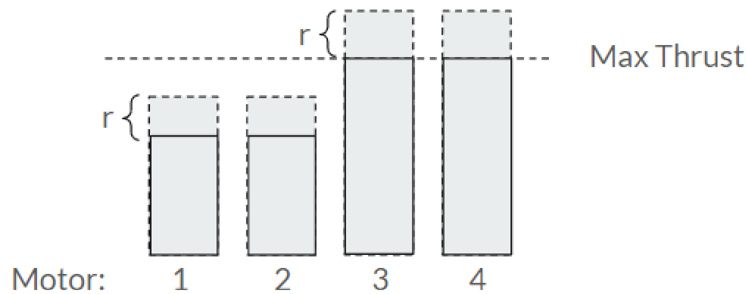


Figure 13: Thrust cap compensation.

5 FURTHER DEVELOPMENT

Since the project focuses on one single Crazyflie drone, further development could be made by extending the project with several drones. It could for example be multiple drones doing synchronized flips and loops as part of an acrobatic performance.

It would also be of interest to improve the motion planning. The following are some things that could be done:

- Place the objective points in time based on distances between each other, rather than at set intervals.
- After solving the optimisation problem, use the solution to better estimate when each objective point should be placed in time and re-solve the optimisation problem.
- Improve the efficiency of the Interpolated cost function by parameterising it as a function of w, similarly to how the Legacy version is parameterised over $\tilde{X}_p$.

Another development would be to validate the dynamic model used. This is discussed further in Section 5.2

5.1 Parameter Estimation

Given a physical model of the Crazyflie with parameters Θ, there are multiple ways to determine these parameters. Since parameter estimation takes a lot of time and effort, and has already been made in other studies, the model in this project uses parameters already calculated. However, the Crazyflies used in this project has an active marker shield for use with Qualisys. Therefore its dynamics should be a little bit different from just the Crazyflie 2.1. Because of that, parameter estimation of the Crazyflies used in this project (including the marker shield) is of interest.

One way to estimate model parameters is to perform physical experiments where the parameters are measured. Such experiments can be to directly measure the weight of the Crazyflie, or to indirectly measure the components of the inertia matrix by rotating the body around certain angles by applying certain momentum.

Another way to determine the parameters is by collecting data from direct use of the system. That is, by collecting data of the state of the Crazyflie while it is operating. The hope is then to be able to estimate the parameters using machine learning techniques. The challenge with this method is instead to properly excite the system. For the purpose of the model, data should be collected while doing flips and fast movements.

5.2 Model validation

It was planned to validate the model used in the simulation environment (and motion planner), and confirm that the model performs similarly compared to measurements of real flight. Unfortunately, as time ran out, the dynamic model in simulation did not perform close enough to real flight to an extent that model validation would be of significance. Therefore, model validation was never performed. However, model validation scripts were developed (but never actually used).

If the simulation was to be improved, model validation would be of interest. Although the parameters are taken from other papers, it is still important to validate the obtained model. This is especially true since the parameters have been

tweaked a little, and if completely new parameters were estimated using techniques in Section 5.1, model validation would be even more important. The data-driven method described in Section 5.1 is used for validation of the model. For a given mission to use for validation, the states of the Crazyflie during real operation, $\bar{X}(t)$, and the states of the Crazyflie in simulation, $\tilde{X}(t;\Theta)$, are measured. The simulated states are then compared to the measured states, the goal being to get the simulated states as close to the measured states as possible. In other words, the goal is to get the error $\bar{e}(t;\Theta) = \tilde{X}(t;\Theta) - \bar{X}(t)$ close to zero for all t, or to minimize the cost function

$$I(\Theta) = \int_{t_0}^{t_f} \bar{e}(t;\Theta)^2 \, dt = \int_{t_0}^{t_f} (\tilde{X}(t;\Theta) - \bar{X}(t))^2 \, dt \tag{17}$$

where t_0 and t_f is the start time and stop (final) time measuring/simulating. In practice, both measured and simulated states are given in discrete time t_k, $k = 1, 2, 3, ..., n$, where n is the number of measurements (and simulated states). The cost function becomes

$$J(\Theta) = \sum_{k=1}^{n} \bar{e}(t_k;\Theta)^2 = \sum_{k=1}^{n} (\tilde{X}(t_k;\Theta) - \bar{X}(t_k))^2. \tag{18}$$

This is called the *Mean Squared Error (MSE)*. When using this method to find better parameters, this function is to be minimized with respect to the parameters Θ. The evaluation of $J(\Theta)$ can serve as a measurement of the accuracy of the simulation and the parameters Θ, i.e., it can be used as a measure during model validation.

The simulated states are sampled with the same frequency as the real states are recorded. However, the recordings are not necessarily of equal length in time and number of samples. A bigger problem is, they might not be synchronized. When measuring the real states, the recording might be started, say, 2 s before the drone executes the mission. The recording of the simulated states might start at 0.5 s before the drone executes the mission. These timings might not be easy to see by just looking at the recordings, but even a small error in timing can make a significant difference in the evaluation of the Mean Squared Error. Of course, the absolute times of the recordings have no meaning - it is only the time difference between corresponding motions in the recordings that is of value. The time difference of the two recorded signals Δt corresponds to an offset in recorded samples Δk, since the sample times are generated by a discrete function $t(k)$ of the sample parameter k. The offset Δk between two discrete signals a and v can be estimated by calculation of the correlation between the signals as

$$c_k = \sum_n a_{n+k} \cdot v_n \tag{19}$$

where $a_k = a(t_k)$ and $v_k = v(t_k)$. Here, a is chosen as the signal with more samples of $\tilde{X}$ and $\bar{X}$, and v as the other. The offset Δk between the signals is then found as

$$\Delta k = \underset{k}{\operatorname{argmax}}\, c_k \tag{20}$$

which corresponds to the time difference of the signals Δt.

Note that all of this has been implemented (but not properly used) during the project, from recording of real and simulated flight to time delay pre-processing and calculation of the mean squared error. The model validation implementation also includes plotting the positional trajectories from the recordings, for visual comparison, and calculating the mean and max positional error (distance) between the recordings.

REFERENCES

[1] Bitcraze, "Crazyflie 2.1," https://www.bitcraze.io/products/crazyflie-2-1/, 2023, [Online; accessed September 12, 2023].

[2] ——, "Crazyradio pa," https://www.bitcraze.io/products/crazyradio-pa/, 2023, [Online; accessed September 12, 2023].

[3] Qualisys, "Motion capture camera for mri scanners," https://www.qualisys.com/cameras/oqus-mri/, 2023, [Online; accessed September 20, 2023].

[4] O. Ljungberg, "Crazyflie accrobatics," https://gitlab.liu.se/olilj86/crazyflie_accrobatics, 2023, [Online; accessed September 14, 2023].

[5] C. Luis, "Design of a trajectory tracking ccontroller for a nanoquadcopter," 2016, [Online; accessed October 4, 2023].

[6] Wikipedia, "Runge–kutta methods," https://en.wikipedia.org/wiki/RungeâĂŞKutta_methods, 2023, [Online; accessed December 15, 2023].

[7] Bitcraze, "Simulation," https://www.bitcraze.io/tag/simulation/, 2023, [Online; accessed december 13, 2023].

[8] J. Förster, "System identification of the crazyflie 2.0 nano quadrocopter," https://www.research-collection.ethz.ch/handle/20.500.11850/214143, 2015, [Online; accessed December 13, 2023].

[9] C. Luis, *DESIGN OF A TRAJECTORY TRACKING CONTROLLER FOR A NANOQUADCOPTER*. ÉCOLE POLYTECHNIQUE DE MONTRÉAL, 2016.