Master Thesis

Dancing Quadrocopters: Trajectory Generation, Feasibility, and User Interface

Author(s):
Augugliaro, Federico

Publication Date:
2011

Permanent Link:

https://doi.org/10.3929/ethz-a-007328864 →

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for
Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme
und Regelungstechnik

Federico Augugliaro

Dancing Quadrocopters: Trajectory Generation, Feasibility, and User Interface

Master Thesis

Institute for Dynamic Systems and Control
Swiss Federal Institute of Technology (ETH) Zurich

Supervision

Angela Schoellig
Prof. Raffaello D'Andrea

September 2011

IDSC-RD-AS-07

Contents

Abstract

This Master Thesis presents various tools that enable the generation of quadrocopter choreograpies for the Flying Machine Arena. The goal of this thesis was to facilitate the design of multi-vehicle choreographies through the developement of a graphical user interface. The resulting software allows the user to easily define trajectories for the quadrocopters. In addition, the feasibility of a choreography can be assessed using the program.
In the first part of this work, we provide the mathematical foundation of the developed methods: Different motion primitives are introduced and their trajectories are defined. A method that determines the feasibility of arbitrary trajectories is demonstrated. Then, an algorithm that exploits convex optimization in order to plan collision-free trajectories for multiple quadrocopters is described. In the second part of this work, we introduce the software tool which was developed as a part of this thesis.

Introduction

This work started in 2009, when I was proposed two different Bachelor Thesis projects to choose between: a quadrocopter that balances a stick or a quadrocopter that dances to the music. I picked the latter. The Bachelor Thesis was mainly a proof-of-concept, in which we demonstrated that it is possible to precisely synchronize a periodic quadrocopter motion to the music beat [23].

This work was our first step into the world of dancing robots, and it was the very first step towards dancing aerial robots. Indeed, most of the research focuses on humanoid robots (see [5, 14, 17] for some of the various examples) or on robots for social interaction [20, 10]. Nevertheless, there are many groups working in the field of music and robotics as the good participation at the *IROS 2010 Workshop on Robots and Musical Expressions* (IWRME) [4] showed. The workshop was attended by both roboticists and members of the Music Information Retrieval (MIR) community. Our contribution to IWRME was the description of a platform for multi-vehicle quadrocopter dancing [22].

In this Master Thesis we further develop this idea into something usable; more precisely, into a software tool. The main goal is to create a tool which allows any user to design its own choreography for the quadrocopters of the ETH Flying Machine Arena (FMA) [19]: visual appealing trajectories for the quadrocopters are combined with music and synchronized to the music beat to achieve a dancing behaviour.

This thesis deals with several aspects that are key to the process of building such a choreography. Part I presents the major components of our system. Here, we lay emphasis on the mathematical aspects that are the basis of the actual software tool, which will be introduced in Part II.

In the first part, the following aspects will be considered: Chapter 1 introduces the reader to the music analysis part, which is marginal to this thesis, since we mainly focus on aspects related to the quadrocopters. Different motion primitives, that define the trajectory of the quadrocopters (and therefore their dancing behaviour) are presented in Chapter 2, which also summarizes some results obtained on synchronization of music beat and quadrocopter motion. Chapter 3 deals with the feasibility of such motion primitives. Indeed, due to the vehicle's physical constraints and the FMA's limited space not all of the conceivable trajectories can actually be flown. Finally, the subsequent chapter,

Chapter 4, copes with the planning of collision-free trajectories for multiple vehicles.

As mentioned above, the second part of this thesis introduces the reader to the software that was developed during this project. Chapter 5 explains how to use this tool. Chapter 6 contains a list of all parameters which are required to define suitable motion primitives. Chapter 7 explains how the class, which is in charge of planning collision-free trajectories for multiple vehicles, works. Finally, Chapter 8 presents the parameters used by the software.

Part I

The System

Chapter 1

Music Analysis

1.1 Music Structure

Introduction Classical choreographies for dancers combine the dancers' movements with the desired music piece. Our choreographies feature quadrocopters in place of dancers, but the music remains essential. Before starting with the design of the quadrocopters' trajectories, we first have to process the underlying soundtrack. In a music analysis step we collect information that helps us with the design of the choreography. Indeed, the desired quadrocopter movements depend on the music structure and will be synchronized to the music beat.

Time Structure A music piece has a well-defined time structure. *Beat times* are the most important information: they represent the tempo of the piece. We decided to use them as the basic unit of the music structure. Strictly related to the beat is the *measure*, which depends on the time signature of the music. In addition to the rhythmic part, we are also concerned about the organization of a song; e.g. we want to identify the refrain or a stanza. For that reason, we identify *sections* and *acts*, and *other special events*. These categories should facilitate the design of a choreography. Figure 1.1 illustrates a possible time structure.

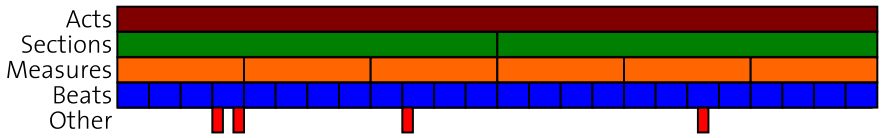

Figure 1.1: A sample time structure of a music piece.

Tools We rely on software for the beat extraction, since it is very difficult to accurately identify the beat times without any help. In the future, other components of the music structure (such as measure, sections, etc.) should be automatically extracted as well. Concerning this issue, Timon Heinis analyzed in his *Studies On Mechatronics* [16] different software tools which provide various

features, both low and high level, of a given music piece. In Section 1.2, we quickly summarize his results.

Currently we use *BeatRoot* [13] to extract the music beats. It is a Java-based software that allows the user to easily export beat times in .txt format. The identification of the other features is done by listening to the music and manually tagging it. The music structure is then stored in a .txt file which has the following structure:

```
...
0.00 B1 M1 S1 A1
1.00 B2
2.00 B3
3.00 B4
4.00 B5 M2
5.00 B6 O1
6.00 B7
7.00 B8
8.00 B9 M3
9.00 B10
10.00 B11 O2
11.00 B12
12.00 B13 M4 S2
13.00 B14
...
```

The first column contains the time information expressed in seconds. It is followed by the feature information (no matter in which order): B stands for beat, M for measure, S for section, A for act and O for other events. As an example, we observe that after four seconds the second measure begins, with the fifth beat. Later, we will use this structure to specify the starting time and the duration of a trajectory assigned to a quadrocopter.

Before advancing to the design of motion primitives for our vehicles, we quickly summarize Timon's results. The aim is to present an overview of software tools which can be used to automate the processing of new music tracks.

1.2 Software for Music Analysis

As mentioned before, beside the beat many other features characterize a music piece. Software to identify such features is already available, but is also part of current research. In the future, such programs will allow us to automatically reconstruct the music structure in a way that is similar to the one presented in Figure 1.1.

In [16], Timon Heinis investigated various software tools. His conclusions are as follows:

> "There are several opportunities to achieve the goal of extracting features for an automatic generation of quadrocopter choreography. If I have to choose a tool, I would vote for *Echonest* [2]. The automation of the process is realizable without any difficulty and the program returns a full set of useful features."

	EN	FMAK	ICRL	MAR	MTB	SV
input files	mp3, m4a, wav, aif,..	mp3	wav, aif	wav, aif, mp3	wav, mp3, aiff, au	wav, ogg, mp3
usability	cmd line	GUI, code kernel	C++ Lib	C++, cmd line	MAT-LAB	GUI
data interface	xml, jason	xml	xml, sdif, txt	arff	MAT-LAB	txt
diversity of features	+++	+++	+	++	++	+++
overall impression	+++	+++	+	+-	++	+++
availability	online	on request	on request	online	online	online
docu available	+	+	-	+	+	+

Table 1.1: Evaluation of high-level music feature extraction tools, from [16]. Legend: EN: Echonest, FMAK: FASTLab Music Analysis Kernel, ICRL: Icram Library, MAR: Marsyas, MTB: MIRtoolbox, SV: Sonic Visualiser.

Table 1.1 summarizes the results of the software tests he conducted during his work. In the future, these tools should be integrated in our software environment.

Chapter 2

Motion Primitives

2.1 Introduction

The Flying Machine Arena is the stage and the quadrocopters are the dancers; this is what a quadrocopter dancing performance is like. However, unlike human dancers, quadrocopters do not have limbs. Their expressiveness is therefore based on their trajectories solely, i.e. their position $x(t) \in \mathbb{R}^3$, and the corresponding velocity $\dot{x}(t)$ and acceleration $\ddot{x}(t)$ over time. Variedness can be extended by specifying the heading of the vehicle, given by the yaw angle and denoted by $\psi(t) \in \mathbb{R}$. Consequently, finding motion patterns which reflect the character of the music is not trivial. Humanoid robots can imitate the gestures of human dancers [21, 18], quadrocopters however cannot. The design of appropriate motions for quadrocopters requires creativity and a good understanding of the vehicle's dynamics and its constraints. In this section, we present some motion primitives we came up with.

Motion Primitive The *Motion Primitive* is the key building block of our choreographies: it represents a well-defined motion of a single quadrocopter during a period of time. The concatenation of many motion primitives defines the quadrocopter's trajectory during the whole performance. The main goal of this project is to provide a tool which enables the user to easily compose a choreography for flying robots in the FMA. We designed a collection of motion primitives which can be shaped to fit any user's need: the *motion library*. The motions can be readily assigned to the quadrocopters via scripting language and with the aid of a graphical interface. As an example: the following line of code

```
M1–M5, 1, CIRCLE | radius=1.5, center=[0;0;3], nrRounds=4 END;
```

assigns quadrocopter 1 a motion called *CIRCLE* for time M1 to time M5 (see Chapter 1 for time structure). In this case, the circle is characterized by three parameters: the radius, the center point and the total number of rounds. In Part II, we introduced the required syntax in more detail.

Categories In [22], we presented two categories of motion primitives, *synchronized motions* and *triggered motions*. Synchronized motions are movements

which are precisely synchronized to the music beat, while triggered motions are understood as movements which start and end at a precise time, but are not strictly related to the music beat in between. This distinction was made having the concept of synchronization in mind: if a synchronization algorithm could be added to the motion primitive, it was labeled as synchronized motion, otherwise as triggered motion. However, our latest considerations suggest that two other categories offer a more appropriate distinction: *Periodic Motions* and *Single-shot Motions*. Periodic motions repeat themselves many times during a given period of time, while single-shot motions are executed only once. These categories seem to better fit the needs of someone designing a choreography. Indeed, given a particular music section one of the first decisions to be made is the one of having or not having a repeating trajectory.
Moreover, the synchronization algorithm presented in [23] was recently extended in [29]. The method now works for any periodic motion primitive based on the framework presented in Section 2.2 (with some limitations). Therefore, only periodic motion primitives that fit some specific requirements are actually synchronizable in the current setup. We will introduce the synchronization framework in Section 2.4.

Motion Library Up to now, the motion library consists of the following entries:

1. Periodic Motions
    - Horizontal circle
    - Swing motion
    - Free periodic motion
2. Single-shot Motions
    - Go to
    - Flip

In the following sections, the motion primitives are described. In this chapter, only the mathematical description will be introduced. Part II deals with the actual implementation and use of the presented motion primitives.

Trajectory Following Controller The motion primitives considered in this chapter, are described by their position, velocity and acceleration. In addition, a yaw trajectory can be specified. These terms are input to an underlying trajectory following controller provided by the FMA infrastructure. This controller is called the Low Level Controller (LLC). Given a desired trajectory, the controller calculates the quadrocopter's input based on the current position of the vehicle. Alternatively, the user can also specifiy the quadrocopter inputs directly, namely the body rates and collective thrust [26]. Except for the special flip motion (where we directly control the inputs), we always provide trajectory information ($x(t),\ \dot{x}(t),\ \ddot{x}(t),\ \psi(t)$) to the LLC.

2.2 Periodic Motion Primitives

2.2.1 General Form

Fourier Series Periodic motion primitives can be represented within the framework of Fourier Series. Each periodic motion can be described by a (possibly) infinite sum of sines and cosines. The evolution of the motion is given by

$$x(t) = x_0 + \sum_{k=1}^{N} a_k \cos(k\Omega \cdot t + k\varphi_0) + b_k \sin(k\Omega \cdot t + k\varphi_0), \tag{2.1}$$

where $x_0 \in \mathbb{R}^3$ is the center point of the motion, $a_k \in \mathbb{R}^3$ and $b_k \in \mathbb{R}^3$ represent the amplitudes associated with the cosine and the sine terms, respectively. The angular frequency is denoted by Ω , and φ_0 is the desired phase. Differentiating with respect to t leads to the desired velocity,

$$\dot{x}(t) = \sum_{k=1}^{N} -a_k \sin(k\Omega \cdot t + k\varphi_0) \cdot k\Omega + b_k \cos(k\Omega \cdot t + k\varphi_0) \cdot k\Omega \ . \tag{2.2}$$

Eventually, the acceleration is given by

$$\ddot{x}(t) = \sum_{k=1}^{N} -a_k \cos(k\Omega \cdot t + k\varphi_0) \cdot (k\Omega)^2 - b_k \sin(k\Omega \cdot t + k\varphi_0) \cdot (k\Omega)^2. \tag{2.3}$$

Matrix Notation

We can convert expressions (2.1)-(2.3) to matrix notation,

$$x(t) = x_0 + A \cdot C(t) + B \cdot S(t) \tag{2.4}$$
$$\dot{x}(t) = A \cdot \dot{C}(t) + B \cdot \dot{S}(t) \tag{2.5}$$
$$\ddot{x}(t) = A \cdot \ddot{C}(t) + B \cdot \ddot{S}(t), \tag{2.6}$$

where

$$A = \begin{bmatrix} a_1 \dots a_N \end{bmatrix} \in \mathbb{R}^{3\times N},$$
$$B = \begin{bmatrix} b_1 \dots b_N \end{bmatrix} \in \mathbb{R}^{3\times N},$$
$$C(t) = \begin{bmatrix} \cos(\Omega t + \varphi) \\ \vdots \\ \cos(N\Omega t + N\varphi) \end{bmatrix} \in \mathbb{R}^{N\times 1},$$
$$S(t) = \begin{bmatrix} \sin(\Omega t + \varphi) \\ \vdots \\ \sin(N\Omega t + N\varphi) \end{bmatrix} \in \mathbb{R}^{N\times 1}.$$

Common Parameters

From Equations (2.4)-(2.6), we see that in order to fully characterize a periodic motion, we need to specify the following parameters:

- the matrices A and B;
- the angular frequency Ω;
- the center point x_0;
- the phase φ.

The matrices Through A and B the trajectories are shaped. The first row refers to the x–direction, the second to the y–direction and the third to z–direction. The i-th column drives the behaviour of the $(i \cdot \Omega)$ frequency component.

The angular frequency The angular frequency Ω defines the frequency of the periodic motion (and thus, its quickness). This value can be related to the time structure of the music (e.g. the user wants to start over the motion every 4 beats) or to the total duration of the motion (e.g. the total number of repetitions is specified). We have two possibilities:

- We define `beatMultiplier` which specifies the duration of one period of the motion by the number of beats it takes. A value of 2 indicates that a period lasts two beats.
- Another possibility is to define the total number of rounds (with *round* understood as *period*) to be executed during the duration of the motion primitive. This is specified by the parameter `nrRounds`.

The center point The center point $x_0 \in \mathbb{R}^3$ contains the coordinates x, y, and z in meters. The syntax follows the Matlab's one: `center = [1; 1; 5]` stays for $x_0 = [1,\ 1,\ 5]^T$.

The phase The phase φ_0 is given in radians. Some useful values:

Degrees	Radians
30	0.524
60	1.047
90	1.571
120	2.094
150	2.618
180	3.142
270	4.712

Table 2.1: Degrees to radians conversion.

Direction In addition, each periodic motion primitive has a travel direction (that is, e.g., clockwise and counterclockwise in a circle). The direction is specified within the description of each motion and can be reversed with `direction=-1`. Since $\cos(-x) = \cos(x)$ and $\sin(-x) = -\sin(x)$, reversing the direction results in $A_{rev} = A$ and $B_{rev} = -B$.

Start/End at Rest compensation

Let us consider a circular trajectory. According to Equation (2.5), the speed at $t = 0$ is greater than zero and is, indeed, constantfor all times. If the quadrocopter is starting the circle at rest, a step in the velocity vector is therefore expected. This is due to the fact that the term $\alpha(t) = \Omega \cdot t$ increases linearly with a constant slope. Thus, to avoid a discontinuity in the velocity, we aim for a zero slope at the beginning and at the end of our trajectory, such that the quadrocopter starts and finishes its trajectory with zero speed. We consider the start of a periodic trajectory. The profile of $\alpha(t)$ can be modified as shown in Figure 2.1.

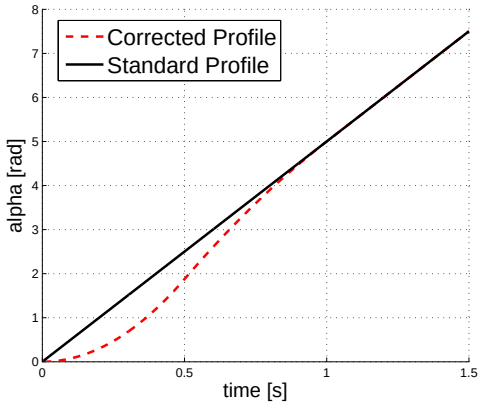

Figure 2.1: Standard profile and corrected profile of the term $\alpha(t)$ at the beginning of a periodic motion primitive.

In Figure 2.1, the compensation lasts $T_{comp} = 1$ second. The term $\alpha(t)$ is modified similar as shown later in Section 2.3.1 where the following boundary conditions apply:

$$\begin{aligned} \alpha(t = 0) &= 0 \\ \dot{\alpha}(t = 0) &= 0 \\ \alpha(t = T_{comp}) &= \Omega \cdot T_{comp} \\ \dot{\alpha}(t = T_{comp}) &= \Omega. \end{aligned} \tag{2.7}$$

The compensation procedure at the end is done similarly.
Be careful when using these compensation terms, since they cause the trajectory to be faster than usual at times where the slope of the modified $\alpha(t)$ is bigger than the original one.

Next, we introduce some particular periodic motion primitives for which we developed a separate interface in our software tool, see Part II.

2.2.2 Horizontal Circle

During the design of the first choreographies [7, 8], we found that the circle is a very useful motion primitive. When combined with a varying radius or center point, appealing figures such as ascending or descending helicoidal trajectories can be obtained (see Section 2.2.5 for details). Moreover, the circle is particularly attractive when flying with multiple quadrocopters, since all vehicles can for example be placed on the same circle by selecting different phases for each vehicle.

A circle in the xy–plane is defined by its radius r and center point x_0. The matrices A and B are fixed and defined as follows:

$$A = \begin{bmatrix} r & 0 & 0 \end{bmatrix}^T$$
$$B = \begin{bmatrix} 0 & r & 0 \end{bmatrix}^T.$$

This convention results in a circle depicted as in Figure 2.2.

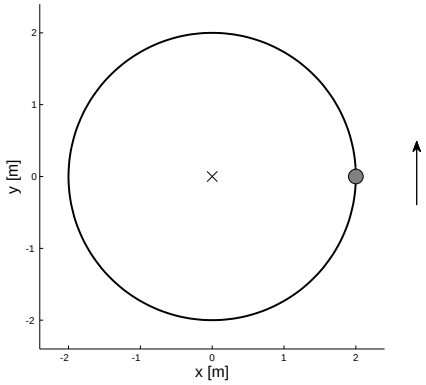

Figure 2.2: Horizontal Circle. The grey dot represents the initial position. The cross is the center point. The arrow indicates the direction of the motion.

2.2.3 Swing Motion

The swing motion (depicted in Figure 2.3) was the very first motion primitive that we investigated. The results were published in [23]. It is a motion primitive which looks a lot lice dancing. It is particularly suitable to precisely track the music beat. The synchronization algorithm works great on this motion primitive and precise synchronization can thus be achieved. The swing motion is defined as a back–and–forth movement of amplitude L in the xy–plane along a given axis (defined by α) around the center point x_0. The matrices A and B are given

Figure 2.3: Swing Motion

as follows:

$$A(t) = \begin{bmatrix} L\cos\alpha & L\sin\alpha & 0 \end{bmatrix}^T,$$
$$B(t) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T.$$

The resulting trajectory is shown in Figure 2.4.

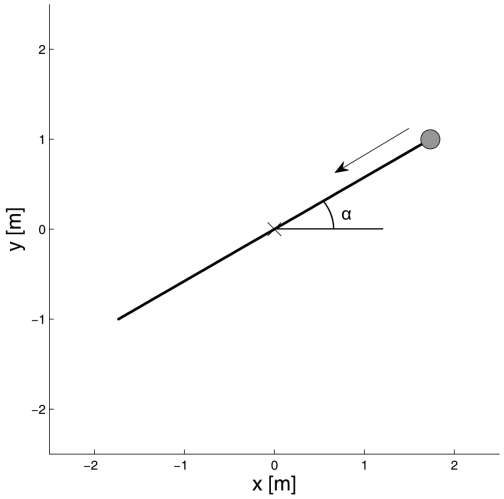

Figure 2.4: Swing Motion. The grey dot represents the initial position. The cross is the center point. The arrow indicates the direction of the motion. The angle α defines the orientation.

2.2.4 Free Periodic Motion

In Section 2.2.1, we presented a general framework for defining periodic motion primitives a sum of sines and cosines. The previous motion primitives are special cases, where the matrices A and B of the Equations (2.4)–(2.6) were predefined. However, to guarantee maximal expressiveness, we provide the user with the possibility of fully defining $A \in \mathbb{R}^{3\times N}$ and $B \in \mathbb{R}^{3\times N}$ for a chosen N.

In this section, we provide some insights in how to choose these matrices. First of all, the matrices A and B must have the same size (though, they can contain zeros). The elements of the matrices determine the amplitude of a movement for a particular direction and frequency. The three rows refer to the three directions x, y and z, respectively. The columns correspond to different frequency components: the i-th column refers to the frequency $i \cdot \Omega$. Next,

some examples and the corresponding trajectories are illustrated. Note that the following motions do not have their own software interface, but are included through the *Free Motion* interface.

Bounce Motion

The bounce motion is depicted in Figure 2.5. Note that the Figure shows the xz-plane. The z–coordinate changes two times faster than the x–coordinates. This behaviour is obtained by the following matrices

$$A = \begin{bmatrix} \Delta x & 0 \\ 0 & 0 \\ 0 & \Delta z \end{bmatrix}, \qquad B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

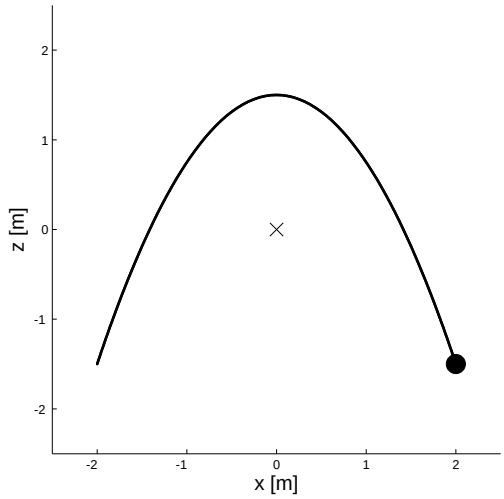

Figure 2.5: Bounce Motion. The grey dot represents the initial position. The cross is the center point. Here $\Delta z = 1.5$, $\Delta x = 2$.

Eight

A motion that resembles an eight is depicted in Figure 2.6. Similar to the bounce motion, the y–coordinate changes two times faster than the x–coordinate. This time however, it acts on the sine term:

$$A = \begin{bmatrix} A_x & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 0 & 0 \\ 0 & B_y \\ 0 & 0 \end{bmatrix}.$$

2.2.5 Some Extensions

Above, we introduced the evolution of periodic motion primitives. In Equation (2.1), the amplitudes a_k and b_k and the center point x_0 are constant. To allow the user to create appealing behaviours from a basic periodic motion primitive

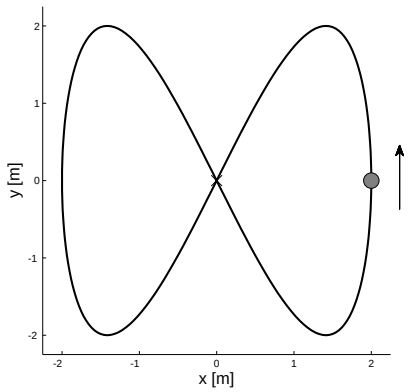

Figure 2.6: Eight Motion. The grey dot represents the initial position. The cross is the center point. The arrow indicates the direction of the motion. Here $A_x = B_y = 2$.

(such as a spiral from a circle), we decided to also allow for time-varying parameters. The user can define time-varying amplitudes $a_k(t)$ and $b_k(t)$, and center point $x_0(t)$. Accordingly, equations (2.4)–(2.6) are modified as follows:

$$x(t) = x_0(t) + A(t) \cdot C(t) + B(t) \cdot S(t) \tag{2.8}$$

$$\dot{x}(t) = \dot{x}_0(t) + A(t) \cdot \dot{C}(t) + B(t) \cdot \dot{S}(t) + \dot{A}(t) \cdot C(t) + \dot{B}(t) \cdot S(t) \tag{2.9}$$

$$\begin{aligned}\ddot{x}(t) =& \ddot{x}_0(t) + A(t) \cdot \ddot{C}(t) + B(t) \cdot \ddot{S}(t) + 2\dot{A}(t) \cdot \dot{C}(t) \\ &+ 2\dot{B}(t) \cdot \dot{S}(t) + \ddot{A}(t) \cdot C(t) + \ddot{B}(t) \cdot S(t),\end{aligned} \tag{2.10}$$

where

$$A(t) = \begin{bmatrix} a_1(t) \dots a_N(t) \end{bmatrix} \in \mathbb{R}^{3\times N},$$

$$B(t) = \begin{bmatrix} b_1(t) \dots b_N(t) \end{bmatrix} \in \mathbb{R}^{3\times N},$$

$$C(t) = \begin{bmatrix} \cos(\Omega t + \varphi) \\ \vdots \\ \cos(N\Omega t + N\varphi) \end{bmatrix} \in \mathbb{R}^{N\times 1},$$

$$S(t) = \begin{bmatrix} \sin(\Omega t + \varphi) \\ \vdots \\ \sin(N\Omega t + N\varphi) \end{bmatrix} \in \mathbb{R}^{N\times 1}.$$

Note that these equations no longer guarantee a periodic motion primitives. However, for convenience we still term the motions presented in the previous section as *periodic*.

Time-varying center point The user can specify a time varying center point $x_0(t)$ in any of the previously defined periodic motion primitives. For convenience, we defined some functions which allow to apply a ramp or a sinusoidal behaviour to a parameter (see Section 5.2.2 for details). Currently, this is the only way to specify a time-varying parameter.

Time-varying matrices In our software, the matrices A and B can be time-varying only when using the special interface for a horizontal circle or for a swing motion. There, the radius $r(t)$ and the amplitude $L(t)$ can be defined following Section 5.2.2.

2.3 Single-shot Motions

As said before, with *single-shot motions* we indicate motion primitives that do not repeat their trajectory. Up to now, this section consists of the *Go To* motion and of the *Flip* motion.

2.3.1 Go To Motion

This motion has been designed to provide smooth transitions (continuous in position and velocity) between motion primitives. The *Go To* motion accepts as inputs an initial position $p_0 \in \mathbb{R}^3$ and velocity $v_0 \in \mathbb{R}^3$, and a final position $p_T \in \mathbb{R}^3$ and velocity $v_T \in \mathbb{R}^3$. The idea of the trajectory design is to generate a bang-bang behavior on the acceleration. In order to meet the feasibility requirements (e.g. bounded third derivative, see Chapter 3), we use a sinusoid instead of a step to switch between accelerations, as Figure 2.7 shows.

Trajectory Our desired trajectories are defined as:

$$\ddot{x}(t) = \begin{cases} A + B & \text{for } 0 \leq t < t_1 \\ A\cos\big(\omega(t - t_1)\big) + B & \text{for } t_1 \leq t < t_2 \\ -A + B & \text{for } t_2 \leq t < T \end{cases} \tag{2.11}$$

$$\dot{x}(t) = \begin{cases} (A + B)t + c_1 & \text{for } 0 \leq t < t_1 \\ \frac{A}{\omega}\sin\big(\omega(t - t_1)\big) + Bt + c_2 & \text{for } t_1 \leq t < t_2 \\ (-A + B)t + c_3 & \text{for } t_2 \leq t < T \end{cases} \tag{2.12}$$

$$x(t) = \begin{cases} \frac{1}{2}(A + B)t^2 + c_1 t + c_4 & \text{for } 0 \leq t < t_1 \\ -\frac{A}{\omega^2}\cos\big(\omega(t - t_1)\big) + \frac{B}{2}t^2 + c_2 t + c_5 & \text{for } t_1 \leq t < t_2 \\ \frac{1}{2}(-A + B)t^2 + c_3 t + c_6 & \text{for } t_2 \leq t < T \end{cases}, \tag{2.13}$$

where

$$t_1 = kT - \frac{T_s}{2},$$
$$t_2 = kT + \frac{T_s}{2},$$
$$\omega = \frac{\pi}{T_s},$$

with $k \in (0, 1)$, T total motion time and T_s (Step time, $\to 0$ for a step) given. The integration constants c_i and the acceleration values A and B are found by

solving for the boundary and continuity constraints. The result is

$$
\begin{aligned}
A &= \frac{2\omega^2(T(v_0+v_T)+2(p_0-p_T))}{-8+\pi^2+4(-1+k)kT^2\omega^2},\\
B &= \frac{AT-2AkT-v_0+v_t}{T},\\
c_1 &= v_0,\\
c_2 &= AkT+v_0-\frac{A\pi}{2\omega},\\
c_3 &= AT-BT+v_T,\\
c_4 &= p_0,\\
c_5 &= \frac{-A\left(-8+(\pi-2kT\omega)^2\right)+8\omega^2 p_0}{8w^2}, \text{ and}\\
c_6 &= \frac{1}{2}(-A+B)T^2-Tv_T+p_T.
\end{aligned}
$$

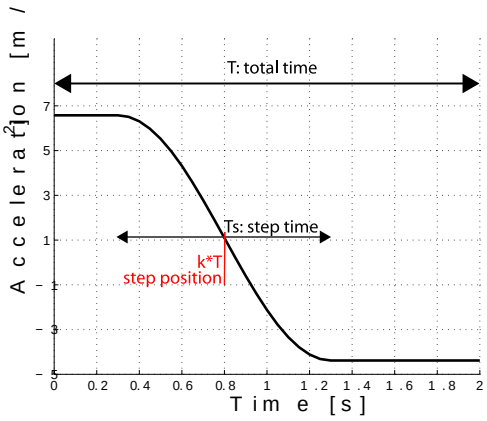

Figure 2.7: The acceleration profile shaped by the following parameters: $k = 0.4,\ T_s = 1,\ T = 2$.

To shape this motion, the step duration T_s and the step position specified by k can be adjusted. Limitations on the parameters are as follows:

$$0 < k < 1 \tag{2.14}$$

$$0 < T_s \leq T(1 - 2 \cdot |0.5 - k|) \tag{2.15}$$

Figure 2.7 shows how the parameters affect the acceleration profile. Figure 2.8 depicts the *Go To* motion along a single direction.

2.3.2 Flip

A particular motion primitive we decided to add, is the flip. From initial point x_0, the vehicle performs a flip reaching $x_0 + [0,\ 0,\ h]^T$ at the end of the motion, where the height h is a parameter. During the flip phase, which lasts T_{flip} seconds, we directly control the desired angle rates w_x and w_y (see Figure 3.2). The ω–profile during the flip phase is depicted in Figure 2.9.

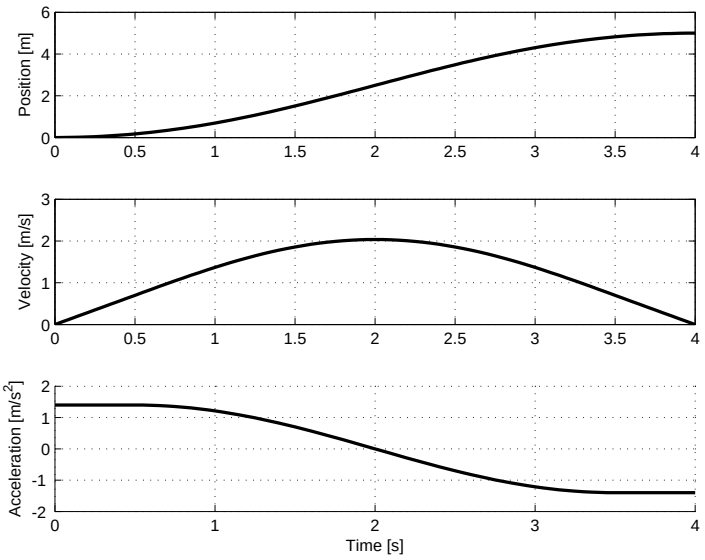

Figure 2.8: The Go To motion in one dimension. Parameters: $k = 0.5,\ T_s = 3,\ T = 4, p_0 = 0,\ v_0 = 0,\ p_T = 5,\ v_T = 0$.

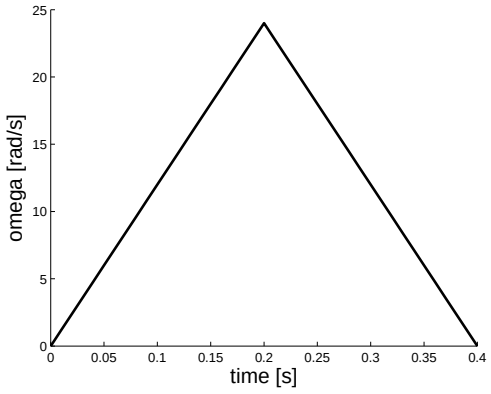

Figure 2.9: The angle rate during the flip phase. Here we chose $\omega_{max} = 24$ and $T_{flip} = 0.4$. These values work well on the real system.

Ideally, the total angle commanded $\frac{T_{flip} \cdot \omega_{max}}{2}$ should equal 2π in order to have a full turn. In practice, however, these values need to be tuned.

2.4 Synchronization

Our system, or more precisely, the combination of quadrocopter and Low Level Controller (LLC), is not always able to exactly follow a given trajectory. This fact is particularly noticeable when trying to synchronize the motion of the

quadrocopter to the music beat. Any temporal inaccuracy of the quadrocopter's response can be easily perceived by a human spectator. The phase lag of the quadrocopter's response was first identified and corrected for in [23] for the unidirectional case of a swing motion.

Recently, the semester project by Clemens Wiltsche [29] identified and compensated errors which occur when flying a periodic trajectory using the LLC for trajectory following. In addition to a shift in the phase, also a considerable error in the amplitude was noticed. The proposed control strategy provides correction terms which are obtained from an offline identification and a method to identify and correct for any residual errors online, while performing the motion. Below, we summarize the main result of this work which are incorporate in our software tool.

2.4.1 Important Results

Limitations

Up to now, the method only works with a single frequency component per direction. It is also important to point out that this method works only for periodic motions, and thus cannot be used in combination with the extensions (e.g. time varying radius) presented in Section 2.2.5.

Corrections Terms

We denote with the subscript $_{ff}$ the feed-forward terms, that are identified offline during a test run. The subscript $_{fb}$ represents the online feedback correction. Considering a single direction and frequency, Equation (2.1) is modified as follows

$$\begin{aligned} x(t) = x_0 &+ (\alpha_{ff} \cdot \alpha_{fb}) a \cos\big(\Omega \cdot t + (\varphi_0 + \varphi_{ff} + \varphi_{fb})\big) \\ &+ (\alpha_{ff} \cdot \alpha_{fb}) b \sin\big(\Omega \cdot t + (\varphi_0 + \varphi_{ff} + \varphi_{fb})\big), \end{aligned} \tag{2.16}$$

with $(\alpha_{ff} \cdot \alpha_{fb})$ and $(\varphi_{ff} + \varphi_{fb})$ being the amplitude and the phase correction terms, respectively. In [29], the online corrector term for the amplitude α_{fb} is additive. We have thus the following relationship,

$$\alpha_{fb} = 1 + \frac{\Delta a}{\alpha_{ff} \cdot a}, \tag{2.17}$$

where Δa is from the online correction integral and a is the desired amplitude.

Claims

The main results of [29] are summarized by the following statements:

- *Decoupled directions.* The feed-forward correction terms in each direction are independent of the motion's components in the other directions. Moreover, as expected from the quadrocopter's symmetry, the x and y directions exhibit the same behavior.
- *Linear behavior.* Considering the motion component in one direction i, the feed-forward correction terms depend only on Ω_i, the desired frequency for that direction.

Chapter 3

Feasibility

3.1 Introduction

A choreography is developed by concatenating many motion primitives resulting in complex and appealing movements. Before carrying out the resulting performance on the real system (and even before testing it in simulation), we want to assure that the chosen trajectories are feasible. Feasibility has to be guaranteed both on the single-vehicle level and on the multi-vehicle level. First of all, the physical constraints of the vehicle must be met. Secondly, the transitions between different motion primitives must be doable with the LLC. Finally, the trajectories must not intersect and must respect the FMA boundaries. In this chapter, we develop tools that allow us to validate the choreographies ahead of time.

3.2 Single Trajectory Feasibility

In this section, we extend the work presented in [24] to arbitrary trajectories $x(t) \in \mathbb{R}^3$ and yaw profiles $\psi(t)$. The goal is to test a given trajectory, characterized by $x(t)$ and $\psi(t)$, for feasibility with respect to vehicle constraints. We derive the required single motor thrusts and rotational rates of such a trajectory, and we check if they are within feasible bounds (see Section 3.2.4). It is important to point out that, in order to be feasible, $\dddot{x}(t)$ (or $x^{(4)}(t)$, see Section 3.2.5 for a discussion) and $\ddot{\psi}(t)$ must be bounded.

3.2.1 Dynamics

FMA Rotation Convention

First, it is important to explain how rotations are expressed in the FMA. We refer to [25] for a detailed description. The vehicle attitude is represented by the Euler angles yaw ψ, pitch θ, and roll ϕ which define the body-fixed frame $\mathbf{V}$, see Figure 3.1. The transformation from the inertial coordinate system $\mathbf{O}$ to $\mathbf{V}$ follows the $z-y-x$ convention and is described by the rotation matrix

$$\overset{V}{O}R(\phi,\theta,\psi) = R_x(\phi)R_y(\theta)R_z(\psi), \tag{3.1}$$

which describes three successive single-axis rotations.

Interesting to note, is that after the first rotation about the z-axis: the direction of the projection of the body-fixed x-axis onto the inertial xy-plane does not change anymore (for $-\frac{\pi}{2} < \theta < \frac{\pi}{2}$). Thus, the yaw angle, which corresponds to the first rotation, can be viewed as the quadrocopter's heading.

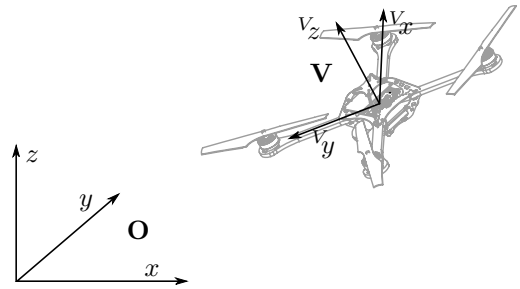

Figure 3.1: The inertial coordinate system **O** and the vehicle coordinate system **V**, from [24].

Translational Dynamics

First, note that from now on, the time and angle dependency of the rotation matrix and other variables are omitted for notational convenience. We consider the vehicle position in the inertial frame ${}^{O}x \in \mathbb{R}^3$. The translational dynamics are given by

$$ {}^{O}\ddot{x} = {}^{O}_{V}R \begin{bmatrix} 0 \\ 0 \\ a \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}, \tag{3.2} $$

where ${}^{O}_{V}R$ is the rotation matrix from the vehicle coordinate frame **V** to the inertial frame **O**, g is the gravity constant, and $a \geq 0$ is the mass-normalized collective thrust, where a is the sum of the mass-normalized rotor forces,

$$ f_i = \tfrac{F_i}{m}, \; i = 1, \ldots, 4, \tag{3.3} $$

where m is the quadrocopter's mass, see Figure 3.2. From Equation (3.2), we can define the thrust vector in the inertial frame as

$$ {}^{O}f := {}^{O}_{V}R \begin{bmatrix} 0 \\ 0 \\ a \end{bmatrix} = {}^{O}\ddot{x} + \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix}. \tag{3.4} $$

By defining $\overline{f} := \frac{f}{\|f\|} = \frac{f}{a}$, we get

$$ {}^{O}\overline{f} = {}^{O}_{V}R \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{3.5} $$

Note that the normalized thrust vector ${}^{O}\overline{f}$ is given by the trajectory that we are analyzing and that this also defines the third column of ${}^{O}_{V}R$. We will exploit this relationship later, in order to reconstruct the attitude of the quadrocopter.

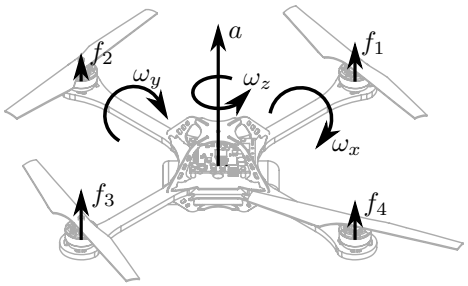

Figure 3.2: The control inputs of the quadrocopter are the body rates ω_x, ω_y, and ω_z and the collective thrust a. These inputs are converted by an onboard controller into motor forces $f_i, i = 1, 2, 3, 4$. Figure taken from [24].

Rotational Dynamics

While spinning, in addition to the force F_i, perpendicular to the quadrocopter base structure, each rotor also causes a reaction torque $M_i = kF_i$, where k is a motor constant (see [24]). We denote the rotational angle rates about the vehicle body axes by $\omega = [\omega_x, \omega_y, \omega_z]$, as shown in Figure 3.2. The rotational dynamics of the body-fixed frame are given by

$$I\dot{\omega} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ k(F_1 - F_2 + F_3 - F_4) \end{bmatrix} - \omega \times I\omega, \tag{3.6}$$

where L is the distance from each motor to the center of the quadrocopter and I is the diagonal inertia matrix of the quadrocopter.

Linking Thrust and Angle Rates

We now go back to (3.5). Differentiating with respect to time and multiplying with ${}^V_O R$ leads to

$${}^V_O R\, {}^O_V\dot{R}\, {}^V n = {}^V_O R(\dot{{}^O\overline{f}}), \tag{3.7}$$

where ${}^V n = [0,\ 0,\ 1]^T$.
Let ${}^V_V S := {}^V_O R\, {}^O_V\dot{R}$ be the rotational velocity of the vehicle relative to the inertial frame, measured in the vehicle frame. It can be shown (see e.g. Chapter 3 in [27]) that

$${}^V_V S = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}. \tag{3.8}$$

With (3.7) and (3.8), we eventually have the following relationship between the angle rates and the derivative of the normalized thrust vector:

$$\begin{bmatrix} \omega_y \\ -\omega_x \\ 0 \end{bmatrix} = ({}^O_V R)^T (\dot{{}^O\overline{f}}), \tag{3.9}$$

where

$$\left({}^{O}\dot{\overline{f}}\right) = \frac{{}^{O}\dddot{x}}{\|{}^{O}f\|} - \frac{{}^{O}f^{T}\,{}^{O}\dddot{x}}{\|{}^{O}f\|^{3}}\,{}^{O}f$$

is known given the desired trajectory $x(t)$ (see Equation (3.4)). We are still missing the entries of the rotation matrix ${}^{O}_{V}R$. In the next section, we reconstruct the attitude of the quadrocopter.

3.2.2 Attitude Reconstruction

We recall from above, that the yaw angle defines the heading of the quadrocopter. Moreover, Equation (3.4) shows that the thrust vector is parallel to the body-fixed z-axis, expressed in the inertial coordinate frame. Therefore, ψ and ${}^{O}f$ fully describe the quadrocopter's attitude. We exploit this fact to reconstruct ${}^{O}_{V}R$.

Rotation Matrix

According to the FMA conventions, the rotation matrix from **V** to **O** reads as

$${}^{O}_{V}R = \left({}^{V}_{O}R\right)^{T} = \begin{bmatrix} c\psi c\theta & c\psi s\theta s\phi - s\psi c\phi & c\psi s\theta c\phi + s\psi s\phi \\ s\psi c\theta & s\psi s\theta s\phi + c\psi c\phi & s\psi s\theta c\phi - c\psi s\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix}, \tag{3.10}$$

where c and s denote cosine and sine. Recall that from (3.5), the third column of the rotation matrix equals the known normalized thrust vector ${}^{O}\overline{f} = [f_x,\ f_y,\ f_z]^{T}$. Therefore,

$$f_x = c\psi s\theta c\phi + s\psi s\phi \tag{3.11}$$
$$f_y = s\psi s\theta c\phi - c\psi s\phi \tag{3.12}$$
$$f_z = c\theta c\phi \Leftrightarrow c\phi = \frac{f_z}{c\theta} \tag{3.13}$$

By inserting (3.13) into (3.11) and (3.12), and then combining the latter two, we get

$$\theta = \arctan\left(\frac{f_x \cos\psi + f_y \sin\psi}{f_z}\right). \tag{3.14}$$

Knowing both θ and ψ, we can thus reconstruct also the first column of ${}^{O}_{V}R$ from Equation (3.10).

We will now exploit the orthogonality of rotation matrices. Therefore, with r_i denoting the i-th column of ${}^{O}_{V}R$, we have that

$$r_2 = r_3 \times r_1. \tag{3.15}$$

At this point, ${}^{O}_{V}R$ is fully known.

3.2.3 Single Motor Forces

Knowing the rotation matrix at all times t, we obtain the angle rates ω_y and ω_x from Equation (3.9). Notice that ω_z is simply the time derivative of ψ (this is not the case for ω_y and ω_x, see e.g. [24], Equation (9)). By numerical differentiation we also get $\dot{\omega}_x$, $\dot{\omega}_y$, and $\dot{\omega}_z$. From the rotational dynamics, we retrieve the required single motor thrust given the angle rates. Solving (3.6) and (3.3) for f_i, we obtain

$$\begin{aligned} f_1 &= \frac{1}{4m}\left(m \cdot a - \frac{2b_2}{L} + \frac{b_3}{k}\right), \\ f_2 &= \frac{1}{4m}\left(m \cdot a + \frac{2b_1}{L} - \frac{b_3}{k}\right), \\ f_3 &= \frac{1}{4m}\left(m \cdot a + \frac{2b_2}{L} + \frac{b_3}{k}\right), \\ f_4 &= \frac{1}{4m}\left(m \cdot a - \frac{2b_1}{L} - \frac{b_3}{k}\right), \end{aligned} \tag{3.16}$$

where

$$\begin{aligned} b_1 &= \dot{\omega}_x I_x + \omega_y \omega_z I_z - \omega_z \omega_y I_y, \\ b_2 &= \dot{\omega}_y I_y - \omega_x \omega_z I_z + \omega_z \omega_x I_x, \\ b_3 &= \dot{\omega}_z I_z + \omega_x \omega_y I_y - \omega_y \omega_x I_x. \end{aligned}$$

Afterwards, $\dot{f}_i,\ i = 1, \ldots, 4$ are found by numerical differentiation.

3.2.4 Vehicle Constraints

In the previous sections, we provided a numerical method to calculate the single motor forces based on the acceleration profile $\ddot{x}(t)$ of a desired trajectory $x(t)$ and on the desired yaw angle $\psi(t)$. This result allows us to establish if a trajectory is feasible or not. Indeed, the quadrocopter's agility is constrained by the minimum and maximum force of a single rotor,

$$f_{i,min} \leq f_i \leq f_{i,max} \quad i = 1, \ldots, 4. \tag{3.17}$$

Note that $f_{i,min} > 0$, since the rotors are able to spin in one direction only and cannot be completely switched off during flight.

Although pretty fast, the motors cannot instantaneously change their rotational speed. We have

$$\left|\dot{f}_i\right| \leq \dot{f}_{i,max} \quad i = 1, \ldots, 4. \tag{3.18}$$

In addition, we also have bounds on the maximal rotational velocities

$$|\omega_k| \leq \omega_{k,max} \quad k = x, y, z, \tag{3.19}$$

3.2.5 Discussion

Above, we derived the single motor forces and the rotational rates of the quadrocopter associated to a desired trajectory. Equation (3.16) shows that the single motor forces f_i depend on the body rate derivatives $\dot{\omega}_k$ (angular acceleration). Furthermore, Equation (3.9) relates the jerk $\dddot{x}(t)$ to the body rates. This indicates that the body rate derivatives depends on the snap $x^{(4)}(t)$, the fourth derivative of the position. Summarizing, we have

$$f_i = \text{FUNCTION}\big(\dot{\omega}_k\big), \tag{3.20}$$

$$\omega_k = \text{FUNCTION}\big(\dddot{x}\big), \tag{3.21}$$

$$\Rightarrow f_i = \text{FUNCTION}\big(x^{(4)}\big). \tag{3.22}$$

This relation implies that, in order for a trajectory to meet the previous constraints we derive the following necessary conditions:

- To satisfy constraint (3.18), the fifth derivative must be bounded.
- To meet constraint (3.17), the fourth derivative must be bounded.

However, in practice the conditions can be relaxed. Indeed, for a 20 ms discretization timestep (as currently used for sending commands in the FMA) and a step in the jerk, the resulting numerical derivative remains bounded. Moreover, in Equation (3.16) the angular accelerations are multiplied with very small inertia terms. Therefore, the influence of steps in jerk on the single motor forces f_i is negligible. Figure 3.3 shows the single motor forces which result from the depicted jerk trajectory. Although a step in the jerk, the peak in the f_i is small and satisfies the thrust limit $f_{i,max}$. In the current setup, the condition of having a bounded third derivative represents a quite good approximation of the true constraints.

3.3 Transition Feasibility

3.3.1 Error in Simulation

The designed choreography consists of the concatenation of many motion primitives. For a single motion primitive the states are continuous up to the acceleration (consequently, the third derivative remains bounded, see Section 3.2.5). However, when combining two different motion primitives it may happen that the initial and final velocities (or acceleration) do not match. To partially avoid that, one could easily design a choreography where the trajectories are continuous up to velocity, exploiting for example the *Go To Motion*, see Section 2.3.1.

However, one may want to drastic transition between motion primitives. For these cases, we decided to only require continuity in the position. The LLC must deal with possible jumps in the velocity and in the attitude of the quadrocopter. Before performing such an aggressive trajectory on the real system (which is unfeasible according to what we presented in Section 3.2), we need to approximate how the LLC reacts. We therefore simulate the choreography in our simulation

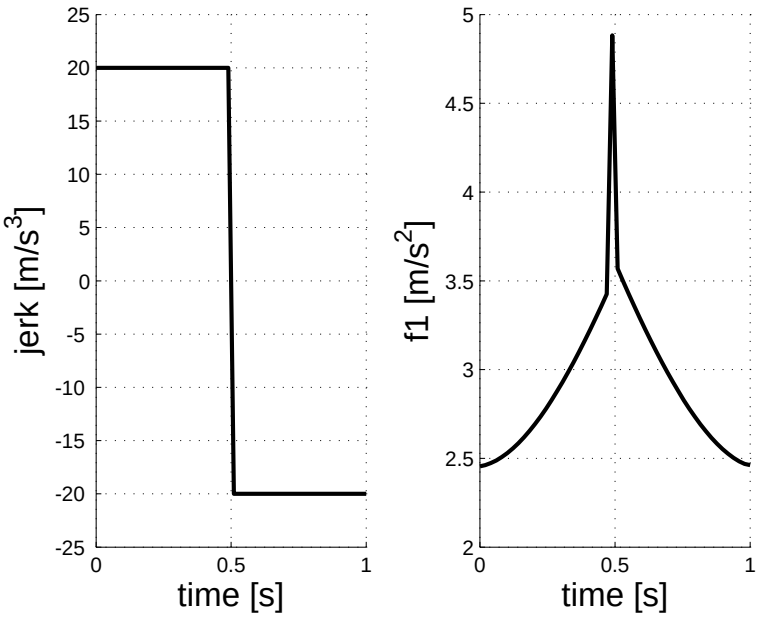

Figure 3.3: On the right, a step in the jerk trajectory for the x-coordinate is depicted. The resulting single motor force f_1 is plotted on the left. Despite the big step in the jerk, the peak is small and is within the acceptable values (which are around 5 $\frac{m}{s^2}$) assuming a sampling time of 20 ms.

environment (which includes the LLC), and measure of how much the actual trajectory $x_a(t)$ deviates from the desired one $x_d(t)$. If

$$\|x_d(t) - x_a(t)\| \leq \Delta x_{max}, \quad \forall t, \tag{3.23}$$

where Δx_{max} indicated the maximal error permitted, the transition is labeled *feasible*.

3.3.2 Drawbacks

We are aware of the approximative character of this method. Strictly speaking, a necessary condition for a motion primitives to be feasible is a continuous acceleration (or even continuous jerk, as discussed in Section 3.2.5) . In that case, we could employ the tools of Section 3.2 to verify feasibility. In practice however, the design of a choreography demands some artistic choices that contradict the mathematical model. We decided to allow these inaccuracies and test for their practical feasibility, rather than for the theoretical one.

3.4 Position Constraints

3.4.1 Intersections

The designed choreographies consist of many quadrocopters flying in the FMA. Therefore, we must assure that the individual trajectories do not intersect. Let

N be the total number of quadrocopters. The following condition has to be satisfied:

$$\|x_i(t) - x_j(t)\| \geq R_{min}, \quad i \neq j,\ i,j = 1,\dots,N,\ \forall t, \tag{3.24}$$

where R_{min} is the minimum possible distance between vehicles. This check is also performed in simulation, since some transitions may result in strange behaviours (as discussed in Section 3.3).

3.4.2 FMA Boundaries

The FMA is a 10x10x10 meter space. However, not all of the space can be used. Indeed, the motion capture cameras cover the arena only partially and, at the bottom of the space, the water channel occupies a part of the space. The need to account for possible inaccuracies of the LLC response further shrinks the usable area. Therefore, we have the following constraints for the arena walls

$$k_{min} \leq x_{i,k} \leq k_{max} \qquad k = x,\ y,\ z, \tag{3.25}$$

and for the water channel,

$$\|x_{yz} - x_{yz,wc}\|_\infty \leq R_{wc} \tag{3.26}$$

where $x_{yz} \in \mathbb{R}^2$ indicates that only the y-coordinate and z-coordinate are considered.

3.5 Preliminary Tests

Besides feasibility, we also want to have an indication of how well the LLC performs. To analyze the performance, we flew a diagonal trajectory, first upwards then downwards, three times, always decreasing the duration of the motion. According to the feasibility checks of Section 3.2, the first iteration was feasible, the second was near the boundaries, while the third one was labeled unfeasible. In Figure 3.4 the trajectories for each coordinate are shown.
We observe that the LLC has problems when stopping the quadrocopter (even if that is most probably due to the discontinuity in the attitude, caused by the *Go To* motion we used). However, we note that the precision during the diagonal trajectories is always worse. This is obvious for the third trajectory, which was labeled as unfeasible. This is only a preliminary test. Further analysis, which also takes into account motor saturation and the general behaviour of the LLC should be conducted. However, even this simple test shows, as expected, a correlation between the physical feasibility and a good LLC tracking response.

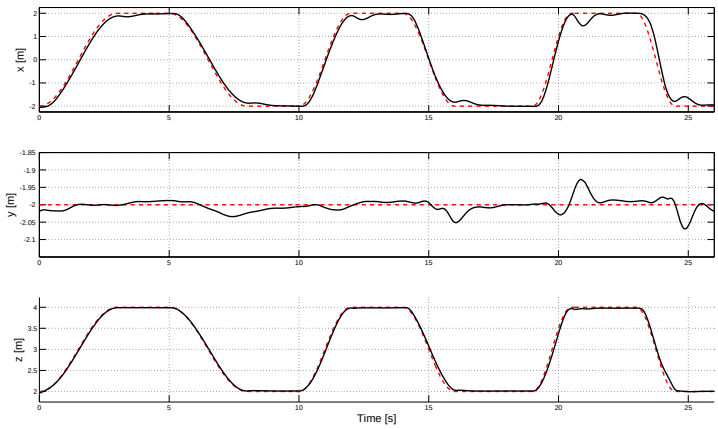

Figure 3.4: Solid black line: Actual Trajectory. Dashed Red Line: Desired Trajectory.

Chapter 4

Collision-Free Motion

In Chapter 3, we showed that all predefined trajectories are checked for feasibility and potential intersections. By exectuing these tests, we can check if a choreography is feasible. More troublesome are the takeoff and landing phases which precede, respectively follow, the choreography. Before a performance at the FMA, the quadrocopters are randomly placed in the space: it is difficult to place the vehicles at the right position, so that no crashes occur during takeoff and landing. In the future, the quadrocopters will even take off from charging stations, which will be located at fixed spots.

As a consequence, we need a tool to generate trajectories for multiple vehicles on-the-fly, that are feasible and do not intersect. The following work was inspired by a talk of Yang Wang [28] from the Information Systems Laboratory of Stanford University. During a visit at the IDSC, he presented three applications of convex optimization in control. The use of convex optimization to generate optimal trajectories for multiple vehicles that meet avoidance constraints seemed to be suitable for our needs. With the aid of the talk's slides, his work was implemented and adapted to our system.

4.1 Trajectory Optimization Problem with Avoidance Constraints

4.1.1 Problem Formulation

According to [12], a standard form for optimization problems is

$$\begin{aligned} &\text{minimize } f_0(x) \\ &\text{subject to } f_i(x) \leq 0, i = 1, \ldots, m \\ &\qquad\quad h_i(x) = 0, i = 1, \ldots, p \end{aligned} \tag{4.1}$$

where $x \in \mathbb{R}^n$ is the optimization variable, $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the objective function, $f_i : \mathbb{R}^n \to \mathbb{R},\ i = 1, \ldots, m$, are the inequality constraint functions and $h_i : \mathbb{R}^n \to \mathbb{R}$ are the equality constraints functions. If $f_0, f_1, \ldots, f_m$ are convex, and the equality constraints are affine we call (4.1) a *convex* optimization problem. Convexity is important since any locally optimal point of a convex problem is

also globally optimal [12]. Our multi-vehicle trajectory planning problem will be formulated as quadratic convex optimization problem:

$$\begin{aligned} &\text{minimize } x^T H x + f^T x \\ &\text{subject to } A_{in} x \leq b_{in} \\ &\qquad\qquad A_{eq} x = b_{eq} \\ &\qquad\qquad L_b \leq x \leq U_b. \end{aligned} \tag{4.2}$$

Given N quadrocopters, the goal is to generate trajectories to transition from the set of initial states to the final states, while ensuring a minimal distance between vehicles.

System Dynamics

The trajectories are described by the discretized dynamics equations:

$$v_i(t+1) = v_i(t) + h\ a_i(t), \tag{4.3}$$

$$p_i(t+1) = p_i(t) + h\ v_i(t) + \frac{h^2}{2} a_i(t), \tag{4.4}$$

where $p_i(t) \in \mathbb{R}^3$ is the position of vehicle i at time t, whereas $v_i(t) \in \mathbb{R}^3$ indicates its velocity and $a_i(t) \in \mathbb{R}^3$ its acceleration. The variable h is the discretization time step between discrete time steps denoted by $t = 1,\ 2, \ldots,\ T$.

Optimization Variable

The optimization variable $x \in \mathbb{R}^{3NT}$ consists of the quadrocopters' accelerations at each time step t and is defined as follows:

$$x = \begin{bmatrix} a_{1,x} \\ a_{1,y} \\ a_{1,z} \\ a_{2,x} \\ a_{2,y} \\ a_{2,z} \\ \vdots \\ a_{N,x} \\ a_{N,y} \\ a_{N,z} \end{bmatrix} \in \mathbb{R}^{3NT},$$

where $a_{i,k} \in \mathbb{R}^T$ is the acceleration of quadrocopter i along the direction k for $t = 1, \ldots, T$.
According to (4.1), the constraints must be formulated with respect to the optimization variable. However, some of our constraints are specified on the position and velocity. From the system dynamics (4.3) and (4.4), we can derive the expressions for $p_i(t)$, $v_i(t)$, for $t = 1, \ldots, T$, and $i = 1, \ldots, N$. For a single

quadrocopter the velocity can be deduced as follows:

$$
\begin{aligned}
v(1) &= v^{start} \\
v(2) &= v(1) + h\ a(1) = v^{start} + h\ a(1) \\
v(3) &= v(2) + h\ a(2) = v^{start} + h\ a(1) + h\ a(2) \\
&\vdots \\
v(t) &= v^{start} + h\,[\ a(1) + \ a(2) + \ldots + a(t-1)]\,.
\end{aligned}
$$

In matrix form, this is

$$
\begin{bmatrix} v(1) \\ \\ \vdots \\ \\ v(T) \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & \cdots & \cdots & \cdots & 0 \\ h & 0 & \cdots & \cdots & 0 \\ h & h & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ h & \cdots & \cdots & h & 0 \end{bmatrix}}_{H_1 \in \mathbb{R}^{T \times T}} \begin{bmatrix} a(1) \\ \\ \vdots \\ \\ a(T) \end{bmatrix} + \begin{bmatrix} v^{start} \\ \\ \vdots \\ \\ v^{start} \end{bmatrix}.
$$

Similarly, for the position we obtain

$$
p(t) = p^{start} + h\ (t-1)\ v^{start} + \frac{h^2}{2}\left[(2t-3)a(1) + (2t-5)a(2) + \ldots + a(t-1)\right],
$$

which can be written as:

$$
\begin{bmatrix} p(1) \\ \\ \vdots \\ \\ p(T) \end{bmatrix} = \underbrace{\frac{h^2}{2}\begin{bmatrix} 0 & \cdots & \cdots & \cdots & 0 \\ 1 & 0 & \cdots & \cdots & 0 \\ 3 & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ 2T-3 & \cdots & 3 & 1 & 0 \end{bmatrix}}_{H_2 \in \mathbb{R}^{T \times T}} \begin{bmatrix} a(1) \\ \\ \vdots \\ \\ a(T) \end{bmatrix} + h \underbrace{\begin{bmatrix} 0 \\ 1 \\ \vdots \\ T-2 \\ T-1 \end{bmatrix}}_{H_3 \in \mathbb{R}^{T \times 1}} v^{start} + \begin{bmatrix} p^{start} \\ \\ \vdots \\ \\ p^{start} \end{bmatrix}.
$$

We can now express the velocities and the positions with respect to the optimization variable x:

$$
\begin{bmatrix} v_{1,x} \\ v_{1,y} \\ v_{1,z} \\ \vdots \\ v_{N,x} \\ v_{N,y} \\ v_{N,z} \end{bmatrix} = \begin{bmatrix} H_1 & 0 & \cdots & \cdots & 0 \\ 0 & H_1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & & \\ 0 & \cdots & \cdots & \cdots & H_1 \end{bmatrix} x + \begin{bmatrix} v_{1,x}^{start} \cdot \mathbf{1} \\ v_{1,y}^{start} \cdot \mathbf{1} \\ v_{1,z}^{start} \cdot \mathbf{1} \\ \vdots \\ v_{N,x}^{start} \cdot \mathbf{1} \\ v_{N,y}^{start} \cdot \mathbf{1} \\ v_{N,z}^{start} \cdot \mathbf{1} \end{bmatrix}, \tag{4.5}
$$

$$
\begin{bmatrix} p_{1,x} \\ p_{1,y} \\ p_{1,z} \\ \vdots \\ p_{N,x} \\ p_{N,y} \\ p_{N,z} \end{bmatrix} = \begin{bmatrix} H_2 & 0 & \cdots & \cdots & 0 \\ 0 & H_2 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ & & & & \\ 0 & \cdots & \cdots & 0 & H_2 \end{bmatrix} x + \begin{bmatrix} p_{1,x}^{start} \cdot \mathbf{1} \\ p_{1,y}^{start} \cdot \mathbf{1} \\ p_{1,z}^{start} \cdot \mathbf{1} \\ \vdots \\ p_{N,x}^{start} \cdot \mathbf{1} \\ p_{N,y}^{start} \cdot \mathbf{1} \\ p_{N,z}^{start} \cdot \mathbf{1} \end{bmatrix} + \begin{bmatrix} v_{1,x}^{start} \cdot H_3 \\ v_{1,y}^{start} \cdot H_3 \\ v_{1,z}^{start} \cdot H_3 \\ \vdots \\ v_{N,x}^{start} \cdot H_3 \\ v_{N,y}^{start} \cdot H_3 \\ v_{N,z}^{start} \cdot H_3 \end{bmatrix}, \tag{4.6}
$$

where $\mathbf{1} \in \mathbb{R}^T$ is a vector of length T containing ones.

Objective Function

The trajectories generated by this method are required to be smooth are optimal with respect to an objective. Following Wang's slides, we decided to minimize the energy consumption and therefore the total thrust. Taking gravity into account (see Equation (3.4)), we define the thrust at time t by

$$f_i(t) = a_i(t) + [0,\ 0,\ 9.81]^T . \tag{4.7}$$

The discrete-time objective function is therefore

$$\min_x \sum_{i=1}^{N} \sum_{t=1}^{T} \|f_i(t)\|_2{}^2. \tag{4.8}$$

The objective function (4.8) can be reformulated to fit the desired form

$$\min_x x^T H x + f^T x. \tag{4.9}$$

4.1.2 Equality Constraints

Initial and Final States

The position, velocity and acceleration of the quadrocopter at $t = 1$ and $t = T$ are precisely defined. When working with quadrocopters, the acceleration is important since it defines (together with the yaw angle which is assumed to be constant) the vehicle's attitude. The constraints are:

$$\begin{aligned} p_i(1) &= p_i^{start}, & p_i(T) &= p_i^{end}, \\ v_i(1) &= v_i^{start}, & v_i(T) &= v_i^{end}, \\ a_i(1) &= a_i^{start}, & a_i(T) &= a_i^{end}. \end{aligned} \tag{4.10}$$

The initial positions and velocities were already taken into account in Equations (4.5) and (4.6). With the aid of these expressions and of (4.10), we can write down the remaining equality constraints with respect to x as in (4.2). The resulting matrices are

$$\begin{aligned} A_{eq} &\in \mathbb{R}^{4NT \times 3NT}, \\ b_{eq} &\in \mathbb{R}^{4NT \times 1}. \end{aligned}$$

4.1.3 Convex Inequality Constraints

Jerk Constraints

The *jerk* is the third derivative of the position. It indicates the rate of change of acceleration. Again, for quadrocopters this is equivalent to a change in the attitude, which cannot be instantaneous and thus needs to be constrained. The jerk constraint is properly defined through the vector norm

$$\|\dot{a}\| = \|J\|_2 \leq J_{max} \tag{4.11}$$

or can be restrained to the single directions

$$|\dot{a}_k| = |J_k| \leq J_{k,max}, \quad k = x,\ y,\ z. \tag{4.12}$$

A method to define a limit value for the jerk is found in [15], Equation (37). The use of quadratic constraints makes the optimization problem slower. Therefore, we decided to constrain the single directions. This can be described by affine functions as the formulation (4.2) requires. The resulting matrices are

$$A_{in,1} \in \mathbb{R}^{2(T-1)3N \times 3NT},$$
$$b_{in,1} \in \mathbb{R}^{2(T-1)3N \times 1}.$$

Acceleration Constraints

Due to the physical constraints of our vehicles and their motors, the total thrust required to fly a trajectory cannot exceed F_{max},

$$\|f\|_2 \leq F_{max}. \tag{4.13}$$

This constraint is accurate, but it is quadratic. To make the problem easier and faster to solve we decided to introduce upper and lower bounds on the single accelerations only,

$$a_{k,min} \leq a_k \leq a_{k,max}, \quad k = x,\ y,\ z. \tag{4.14}$$

Note that the actual acceleration along the z-direction must take into account the gravity vector g. Therefore, $a_{z,max} = a_{max} - g$. These bounds are specified in

$$L_b \in \mathbb{R}^{3NT},$$
$$U_b \in \mathbb{R}^{3NT}.$$

Arena Constraints

The flying space is constrained by the FMA boundaries adding the following position constraints

$$k_{min} \leq p_{i,k} \leq k_{max} \qquad k = x,\ y,\ z,$$

which result in

$$A_{in,2} \in \mathbb{R}^{6NT \times 3NT},$$
$$b_{in,2} \in \mathbb{R}^{6NT \times 1}.$$

4.1.4 Nonconvex Inequality Constraints

In addition to the above convex constraints, this particular problem encloses also nonconvex constraints as described below. In the next section, we will explain how we those constraints.

Collision Avoidance Constraints

The quadrocopters must maintain a minimum distance R_{min} from each other and cannot optimize their own trajectory only. The corresponding constraints read as follows:

$$\|p_i(t) - p_j(t)\|_2 \geq R_{min}, \quad i \neq j, \quad t = 1, \ldots, T. \tag{4.15}$$

This constraint guarantees a minimum distance of R_{min} between any two quadrocopters.

Water Channel Constraints

The water channel cannot be described by affine constraints. We can model it by fixing a minimum distance between the quadrocopters and the water channel center line.

$$\|p_{i,yz}(t) - p_{wc,yz}(t)\|_2 \geq R_{wc,min}, \quad t = 1, \ldots, T. \tag{4.16}$$

where $p_{i,yz} \in \mathbb{R}^2$.

4.2 The Algorithm

4.2.1 Sequential Convex Programming

Sequential Convex Programming (SCP) is a local optimization method for nonconvex problems [11]. The idea is to replace the nonconvex constraints f with linear approximations $\hat{f}$ around a previous solution x^k and repeatedly solve the convex problem. The method will then iterate until a stop condition is satisfied. SCP is a *heuristic* method and may fail to find the optimal solution and the result depends on the starting point $x^{(0)}$. Nevertheless, according to [11] it often works well and finds a feasible point with good, if not optimal, objective value. This fact was confirmed in our experiments.

Collision Avoidance Constraints

The collision avoidance constraint (4.15) is linearized around the previous solution $x^{(k)}$ using a first-order Taylor expansion

$$\hat{f}(x) = f(x^{(k)}) + \nabla f(x^{(k)})^T (x - x^{(k)}) \tag{4.17}$$

where $\nabla f(x^{(k)})$ denotes the gradient at $x^{(k)}$. This results in

$$\left\|p_i^k(t) - p_j^k(t)\right\|_2 + g^T \left[\left(p_i(t) - p_j(t)\right) - \left(p_i^k(t) - p_j^k(t)\right)\right] \geq R_{min} \tag{4.18}$$

with

$$g = \frac{p_i^k(t) - p_j^k(t)}{\left\|p_i^k(t) - p_j^k(t)\right\|_2}.$$

The total number of constraints is

$$N_c = \frac{N(N-1)}{2}. \tag{4.19}$$

This results in the following matrices

$$A_{in,3} \in \mathbb{R}^{N_c T \times 3NT},$$
$$b_{in,3} \in \mathbb{R}^{N_c T \times 1}.$$

Water Channel Constraints

The water channel constraint (4.16) is linearized in a similar way:

$$\left\| p^k_{i,yz}(t) - p_{wc,yz}(t) \right\|_2 + g^T \left[p_{i,yz}(t) - p^k_{i,yz}(t) \right] \geq R_{wc,min}, \tag{4.20}$$

with

$$g = \frac{p^k_{i,yz}(t) - p_{wc,yz}(t)}{\left\| p^k_{i,yz}(t) - p_{wc,yz}(t) \right\|_2}.$$

The corresponding matrices are

$$A_{in,4} \in \mathbb{R}^{NT \times 3NT},$$
$$b_{in,4} \in \mathbb{R}^{NT \times 1}.$$

4.2.2 Solving the Algorithm

Figure 4.1 presents an overview of the algorithm. The main points are explained below.

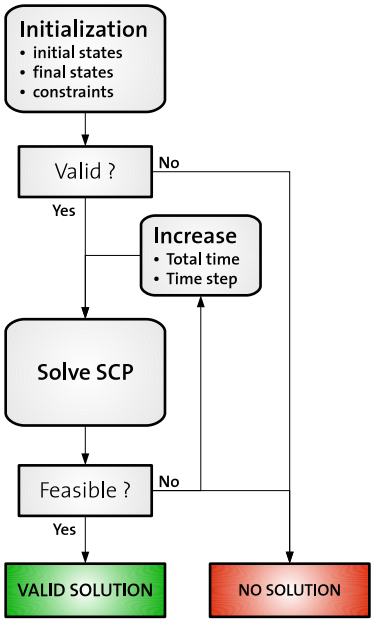

Figure 4.1: Solution strategy to the SCP.

Checking Initial and Final Positions

Before starting the Sequential Convex Programming algorithm, we check if p_i^{start} and p_i^{end} are valid, i.e. if the minimum distance R_{min} between quadrocopters and the FMA constraints are satisfied.

Total Time and Timestep

First of all, initial values for the total time T_{tot} and the number of time steps per seconds, denoted by N_{ts}, are chosen. The actual timestep h is chosen according to

$$h = \frac{T_{tot}}{\text{floor}(T_{tot} \cdot N_{ts})} \tag{4.21}$$

The smaller N_{ts}, the faster the solution to our problem is computed since its dimension is smaller. Surprisingly, we found that even very few time steps per second (e.g. $N_{ts} = 5$) often produce good results. Initially, the total time T_{tot} is chosen to be a low value (around 2 seconds). Both values will be increased if the SCP either exits without a solution or finds an unfeasible solution (see below).

Starting Point

The starting point $x^{(0)}$ for the SCP is the solution to the optimization problem (4.2) without the nonconvex avoidance constraints. The trajectories generated are often close to the final ones and thus are a good starting point. Figure 4.2 illustrates the resulting trajectory along a single dimension.

Regularization

The use of the above starting point solution has problems in some cases. For example, consider the situation where we want two quadrocopters to switch their positions. The trajectories resulting from the unconstrained problem will cross each other. Therefore, at the intersection time $t = t_{cross}$, $p_1^0(t) - p_2^0(t) = 0$. At that point the linearized avoidance constraint (4.18) is undefined.

To solve this problem, we add a small noise to the initial and final positions,

$$p_{i,k}(1) = p_{i,k}^{start} + \eta, \qquad p_{i,k}(T) = p_{i,k}^{end} + \eta, \tag{4.22}$$

where η is drawn from the uniform distribution $\mathcal{U} = (-\overline{\eta}, \overline{\eta})$. In our case, $\overline{\eta} = 0.005$ m solves the problem.

Stopping Condition

The SCP iterates until a stopping condition is satisfied. For this particular problem, the following conditions must be met:

- The nonlinearized constraints (4.15) and (4.16) must be satisfied.
- $x^{(k)}$ must be an optimal solution of the approximate convex problem.
- The objective value changed less than a given tolerance (around 1%) from the previous iteration (we assume that convergence is achieved).

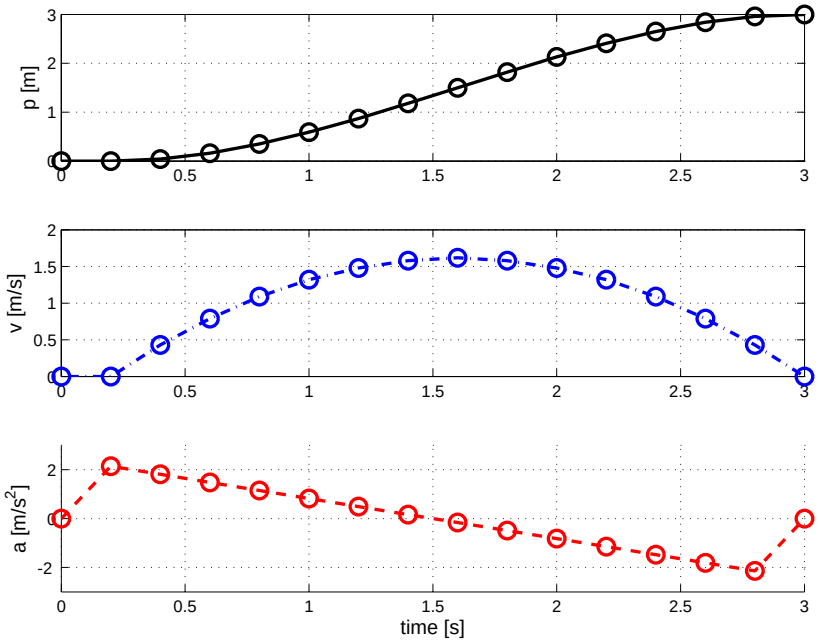

Figure 4.2: Position, velocity and acceleration after the first iteration along one dimension. Initial and final velocity and acceleration were chosen to be zero. The circles illustrate the points at the discrete times $t = 1, \dots, T$. Here, $T_{tot} = 3$, and $N_{ts} = \frac{1}{h} = 5$ were chosen.

Feasibility Checks

When the SCP returns a solution, this is tested for feasibility (according to Section 3.2). Indeed, the above constraints (4.12) and (4.14) do not precisely describe the vehicle's physical limits. They only help the optimization routine to return a solution that is likely to be feasible. By inserting the solution x^* in (4.6) and (4.5), we eventually get $p_i(t)$, $v_i(t)$ and $a_i(t)$ at $t = 1, \dots, T$. Between time periods, we get the corresponding position, velocity and acceleration value by linear interpolation,

$$x(t) = p(t_0) + (t - t_0)\frac{p(t_1) - p(t_0)}{t_1 - t_0}, \quad t_0, t_1 \in \{1, \dots, T\}, \quad t \in [t_0, t_1]. \tag{4.23}$$

where t_0 and t_1 are consecutive discrete times. We perform the feasibility checks on the interpolated trajectories according to Chapter 3, i.e.

- single vehicle trajectory feasibility,
- minimum distance between quadrocopters,
- FMA constraints (wall and water channel).

The minimum distance and the FMA constraints were already included in the formulation of (4.2), but we only guarantee that they are satisfied at the discrete times $t = 1, \dots, T$. In between, these conditions are not necessarily met. This behaviour is explained in Figure 4.3.

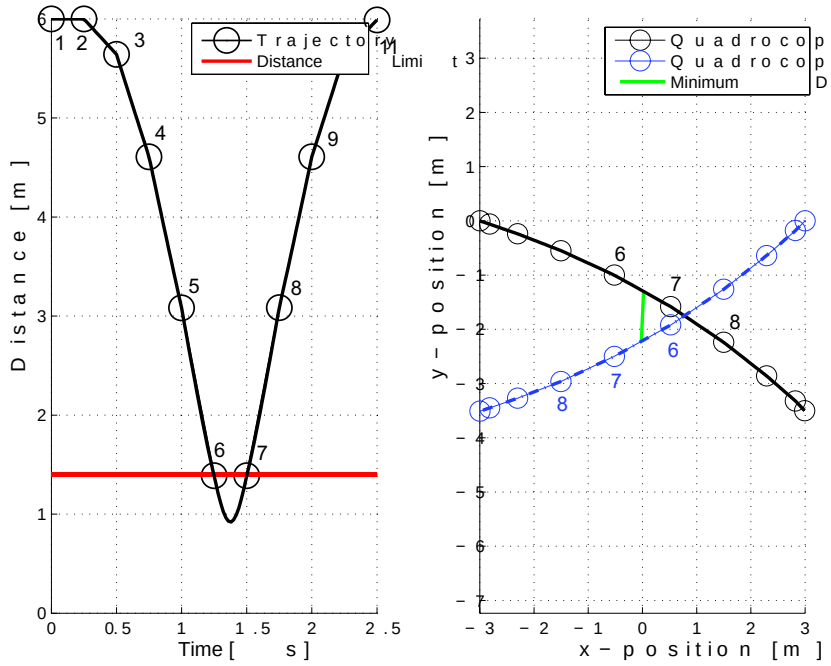

Figure 4.3: We consider two intersecting trajectories (solid black and dashed blue on the right). On the left, the distance between the two vehicles is plotted versus time. The numbered circles indicates the discrete time points. The left plot shows that the algorithm computed valid positions which satisfy the limit distance indicated by the horizontal red line. However, between the time steps 6 and 7, the actual distance is smaller. This behaviour can be seen on the right plot, where the trajectories of the two quadrocopters are plotted in the x-y plane. The green line indicates their minimum distance.

If between timesteps the trajectories appear to be unfeasible, we increase N_{ts} and restart the SCP with a finer resolution. Similarly, when a single trajectory does not meet some physical constraint, we increase T_{tot} before restarting the SCP. Eventually, a feasible solution is found.

4.3 Results

4.3.1 Trajectories

As shown in Figure 4.2, the objective function (4.8) results in smooth single-vehicle trajectories. This is the case, even when the collision avoidance constraints are active. Figure 4.4 shows how the algorithm performs in two dimensions.

Note, that despite the fact that no optimality is guaranteed, the resulting trajectories seems to be meaningful and effectively solve our problem.

4.3.2 Applications

Whenever we want to move various quadrocopters at the same time in the FMA, we need to be sure that their trajectories do not intersect. The method presented is very handy in those cases. In particular, we were motivated by the fact that a choreography shall be able to start and end with the quadrocopters at arbitrary locations. Thus, a safe takeoff and landing procedure was needed. This method turned out to be very suitable for this problem. Indeed, the resulting trajectories are fast, smooth and visual appealing. For this reason, this algorithm can also be used to transition between different motion primitives.

4.3.3 Collision Free Transitions in a Choreography

This algorithm turned out to be very useful when creating choreographies. It allows for fast and safe transitions between different motion primitives. A transition motion is defined by its ID (`transitionID`, starting at 1). All the participating quadrocopters must be labeled with an internal ID (`quadroID`) starting at 1 and their starting time and end time must be the same.

The algorithm requires initial and end position to be specified. In addition, also velocity and acceleration can be handed to the function. Velocity and acceleration are particularly useful to create smooth transitions. For implementation details see Section 5.3.4.

4.3.4 Demonstration

To demonstrate this algorithm, a demonstration on the real system was developed. More details can be found in Chapter 7. A video showing the quadrocopters in action is found online [9].

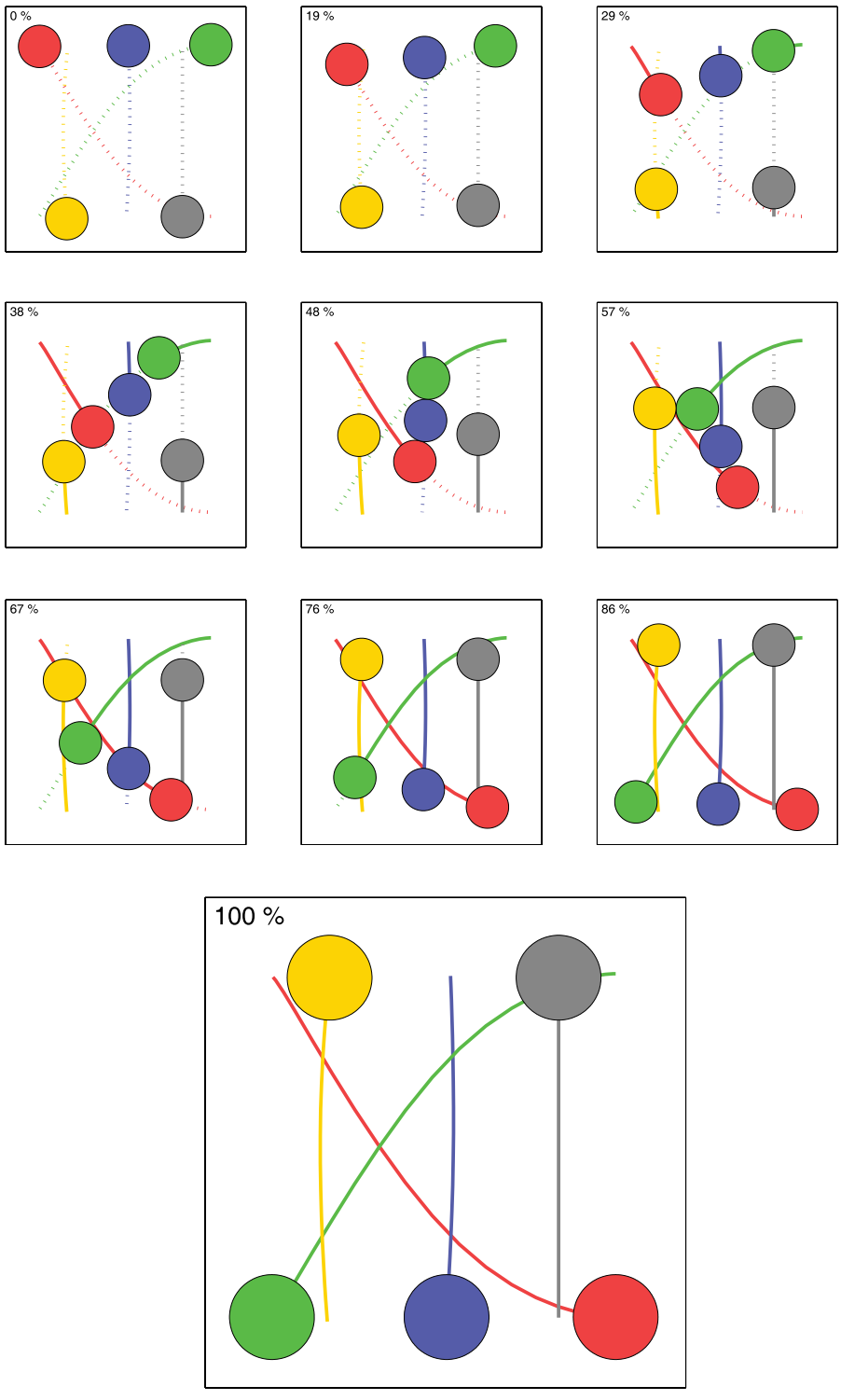

Figure 4.4: The performance of the algorithm demonstrated for a 2D scenario. The figures show the positions of the vehicles (and the required minimum distance) in the xy-plane at subsequent times starting at the top-left corner. The lines indicate the resulting trajectories.

Part II

The Software

Chapter 5

Choreography Generator

5.1 Introduction

The software developed plays an important role in this Master Thesis. Indeed, the purpose was to create a tool which allows a user to easily design a choreography for multiple quadrocopters. We developed an user-friendly interface enabling the user to readily assign the motion primitives presented in Chapter 2 to the vehicles. The software is written in C++, using Qt [3] for the graphical interface. In this chapter, we explain how to compose a choreography starting from scratch.

Folder Structure

First, let us introduce the folder structure. The project is located in the SVN FMA repository under

`\Users\Federico\ChoreographyGeneratorGUI`.

If you want to compile the project, Cplex [1] and Qt must be installed on your computer (according to the FMA Wiki page). You need to update the .pro and .bat files with you Cplex and Qt local path, respectively, and run the .bat file, which creates the Visual Studio project file `ChoreographyGeneratorGUI.vcproj`. Other relevant folders, are located in

`\bin\ChoreographyGenerator`.

These folders contain different files that are needed by the software:

- `ChoreographyFiles` – txt files defining the choreographies.
- `MusicAnalysis` – txt files with the time structure of music pieces.
- `MusicFiles` – wav music files.
- `Corrector` – identified feed-forward terms for the synchronization algorithm, see Section 2.4.

Quick Start

How to start using the software without compiling it.

- Update your SVN FMA.
- ChoreographyGeneratorGUI.exe is in the bin directory.
- If Cplex is not installed on your computer, place the Cplex libraries, ILOG.Concert.dll , ILOG.CPLEX.dll and cplex112.dll in the bin directory. In that case, you are not able to use the takeoff and landing procedure (and other motions that need the collision-free class, see Figure 5.4).

5.2 Choreography Design

The first choreography developed with this tool is found under

`bin\ChoreographyGenerator\ChoreographyFiles\ArmageddonChoreography.txt`

A video showing the results is available online, see [6].

The process of designing a choreography involves the following steps:

- First of all, you have to process a music file, extract its time structure and write it in a txt file.
- Then, motion primitives can be assigned to the quadrocopters for the different parts of the music.
- The resulting choreography has to be checked for feasibility and tested in simulation.
- Finally, the dancing performance can be performed in the FMA.

Below, we present a simple example of a choreography.

5.2.1 Music Structure

Let us assume for simplicity, that we have a music piece where beats occur every second. Following the structure described in Chapter 1, the txt file describing the music structure may look like this:

```
MUSIC="music.wav"

BEAT TIMES

0.00 B1 M1 S1 A1
1.00 B2
2.00 B3
3.00 B4
4.00 B5 M2
5.00 B6 O1
6.00 B7
7.00 B8
8.00 B9 M3
9.00 B10
10.00 B11 O2
```

```
11.00 B12
12.00 B13 M4 S2
13.00 B14
14.00 B15
15.00 B16
16.00 B17 M5
17.00 B18
18.00 B19
19.00 B20
20.00 B21 M6
21.00 B22
22.00 B23
23.00 B24
24.00 B25 M7
25.00 B26
26.00 B27 O3
27.00 B28
28.00 B29 M8  S3 A2 E1
29.00 E2
```

Recall that B stays for beat, M for measure, S for section, A for act, and O for other events. The first line indicates that the corresponding music file is found under

`bin\ChoreographyGenerator\MusicFiles\music.wav` .

If no music file is associated with the time structure, simply set `MUSIC="NONE"`.

5.2.2 Choreography File

Syntax

To assign a motion to a quadrocopter we write

```
TIME1–TIME2, ID, MOTION_TYPE | parameter=value, parameter=value, .. END;
```

We later introduce the parameters specific to each motion. For now, just note that they can be of different type: doubles, integers, matrices, or strings. To define a matrix, we use the Matlab convention where `;` ends a row. Thus, `[1; 1; 1]` represents a 3×1 vector (the transpose convention of Matlab is not supported). Also note, that the text following `#` is considered as a comment, and that `END;` ends a line:

```
#This is a comment
M1–M2, 1, CIRCLE | radius = 1,     ..    END; #This one, too!
```

Example

A script that describes a choreography of three quadrocopters that fits the above music structure may look as follows:

```
#Initial Information
MUSICFILE = "testBeat.txt"
TITLE = "This Choreography Title"
START CHOREOGRAPHY #Let this tag here!
#Start with a goto motion
M1–M2 , 1, GOTO | endPosition=[1.8;0;3.5], endVelocity=[0;2.36;0], k=0.5,
                  stepTime =1, startPosition=[1.5;–3;3.5],
                  startVelocity=[0;0;0], yaw0 = 0.0, yaw1 =20.0  END;
M1–M2 , 2, GOTO | endPosition=[0;0.0;4], endVelocity=[–0.79;–0.08;0.25],
                  k=0.5, stepTime =1, startPosition=[2;0;3.5],
```

```
                startVelocity=[0;0;0]  END;
M1-M2 , 3, GOTO | endPosition=[-1.8;0;3.5], endVelocity=[0;-2.36;0],
                 k=0.5, stepTime =1, startPosition=[-2;-2;3.5],
                 startVelocity=[0;0;0]  END;

#Then, circles with changing radius
M2-M5 , 1, CIRCLE | radius=1.8, center=[0;0;3.5], phi=0,
                    nrRounds=3, direction=1 END;
M2-M5 , 2, CIRCLE | radius0=0, radius1=1, center0=[0;0;4], center1=[0;0;6],
phi=1.57, nrRounds=3, direction=-1 END;
M2-M5 , 3, CIRCLE | radius=1.8, center=[0;0;3.5], phi=3.14,
                    nrRounds=3, direction=1 END;

#Again, use the goto motion to transition
M5-B19 , 1, GOTO | endPosition=[0.5;1;4], endVelocity=[0;0;0],
                   k=0.5, stepTime =1 END;
M5-B19 , 2, GOTO | endPosition=[0.5;-1;4], endVelocity=[0;0;0],
                   k=0.5, stepTime =1 END;
M5-B19 , 3, GOTO | endPosition=[0.5;-3;4], endVelocity=[0;0;0],
                   k=0.5, stepTime =1 END;

#Swing motion
B19-B23 , 1, SWING | amp=0.5, center=[0;1;4], angle=0,
                     beatMultiplier=2, FFCorr = 1 END;
B19-B23 , 2, SWING | amp=0.5, center=[0;-1;4], angle=0,
                     beatMultiplier=2, direction=-1, FFCorr = 1  END;
B19-B23 , 3, SWING | amp=0.5, center=[0;-3;4], angle=0,
                     beatMultiplier=2, FFCorr = 1 END;

#Stay in place, till the end
B28-E2 , 1, GOTO | END;
B28-E2 , 2, GOTO | END;
B28-E2 , 3, GOTO | END;
```

This file can be found under

`bin\ChoreographyGenerator\ChoreographyFiles\sampleChoreography.txt`

The first lines contain three tags that must always be present: only change the string between `"  "`. This choreography consists of four parts: from M2 to M5, the vehicles perform circles. Before that, we tell them to reach the circles' starting points and velocities using the `GOTO` motion (which is applied from M1 to M2). This motion also helps us to transition between the circles and the final swing motions, which are assigned to the quadrocopters from B19 to B28.

End Position

The choreography is supposed to end with the quadrocopters hovering in the space. Indeed, at the end of the choreography (i.e. after the last beat in the music file), the land action is triggered (see Section 5.5.2). We recommend using the *Go To* motion without parameters, as in the above example.

Time Varying Parameters

As you can see above, we allow some parameters to vary over time (see Section 2.2.5). This allows for a bigger variety of motions. For example, by defining a circular motion with an increasing radius and an increasing height, we obtain a spiral. A spiral is realized by the following code:

```
.. CIRCLE | radius0=0, radius1=2, center0=[0; 0; 3], center1=[0;0;6] ..
```

Currently, two different options for varying the parameters are implemented.

Ramp In the above example, a ramp function was applied to both parameters, `radius` and `center`. If we want to apply a ramp on a parameter `p`, we define the starting value and the end value by

`p0` and `p1`,

respectively. Then, the parameter value will change according to:

$$p(t) = p0 + \frac{p1 - p0}{T} \cdot t, \quad t \in (0, T)\,,$$

where T is the total duration of the motion primitive. In the motion library (Chapter 6), parameters are indicated with ^{r} (e.g. $\texttt{center}^r$), if they allow the application of a ramp function.

Sinusoidal Another possibility is to vary a parameter sinusoidally. We define

`p0`, `pAmp`, and `pN`,

offset, amplitude and total number of periods periods, respectively. The parameter's behaviour is then given by

$$p(t) = \mathrm{p0} + \mathrm{pAmp} * \sin\left(\frac{2\pi \cdot \mathrm{pN}}{T} \cdot t\right),$$

where T is the total duration of the motion primitive. Similarly to the ramp function, parameters which accepts the sinusoidal function are indicated with ^{s} (e.g. $\texttt{center}^s$).

Yaw

The motion primitives that were presented in Chapter 2, are described by their position over time $x(t) \in \mathbb{R}^3$. In addition, the user can define the quadrocopter heading through the yaw angle $\psi(t)$. Yaw is defined similar to the other time varying parameters (see above). There are four different possibilities:

- Constant yaw: `yaw`.
- Ramp: `yaw0` and `yaw1`.
- Ramp: `yaw0` and `yawN`, such that $\mathrm{yaw1} = \mathrm{yawN} \cdot 2\pi$.
- Sinusoidal: `yaw0`, `yawAmp`, `yawN`.

FMA Coordinate System

Figure 5.1 shows the inertial coordinate system of the Flying Machine Arena. The FMA is a 10x10x10 meters space, but not all the space is actually usable. See Section 3.4.2 for more details and Chapter 8 for the actual values.
We now explain how to use the provided software.

5.3 Text Editor

The interface layout is based on the layout of `Inconvenience.exe`. It has four different applications: *Text Editor*, *Feasibility Checks*, *Fast Simulation*, and *Flying Mode*. When starting the software, we are in *Text Editor* mode as shown in Figure 5.2.

With the left bar, we can navigate through the different applications. In the middle, we write our choreographies, whereas on the right the music structure file is shown to help the design. Note that any change made to the music structure within this GUI, is not saved. At the top, some useful buttons are placed.

When clicking *New*, a template file appears. There, you should first change the strings `MUSICFILE` and `TITLE` and save the file. It automatically loads the associated music structure into the right panel. Then you can start writing the choreography. To ease the process, some helpful tools were developed.

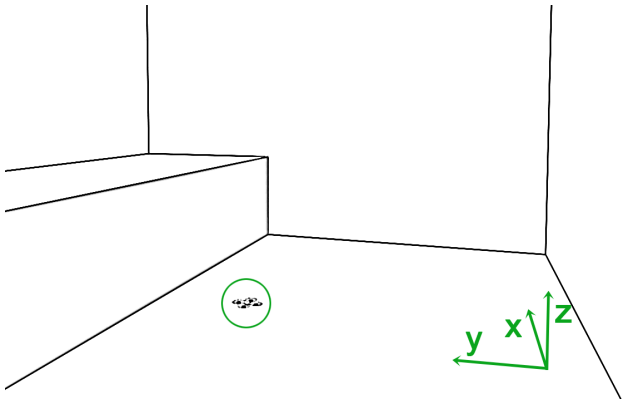

Figure 5.1: Inertial coordinate system of the FMA as viewed from the control room. The quadrocopter (encircled) is sitting at the origin.

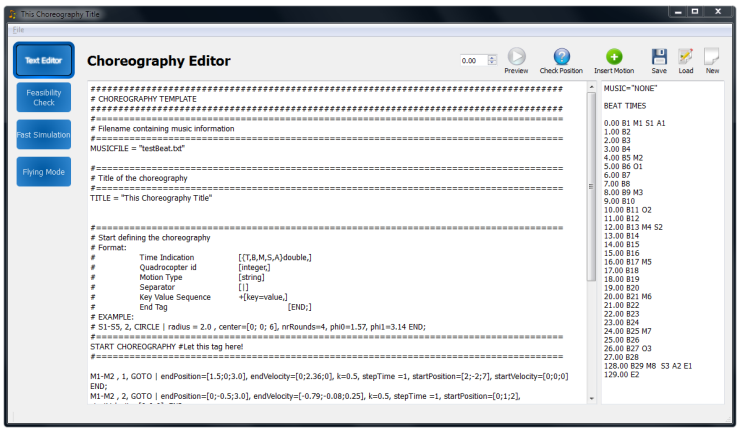

Figure 5.2: The Text Editor.

5.3.1 Insert Motion

If you click on the button *Insert Motion*, the window shown in Figure 5.3 will appear. On the left, you can choose between different motion primitives. Then, on the right, you can specify all the necessary motion parameters. By clicking *Insert Motion*, the text describing the motion primitives will be automatically added at the cursor location.

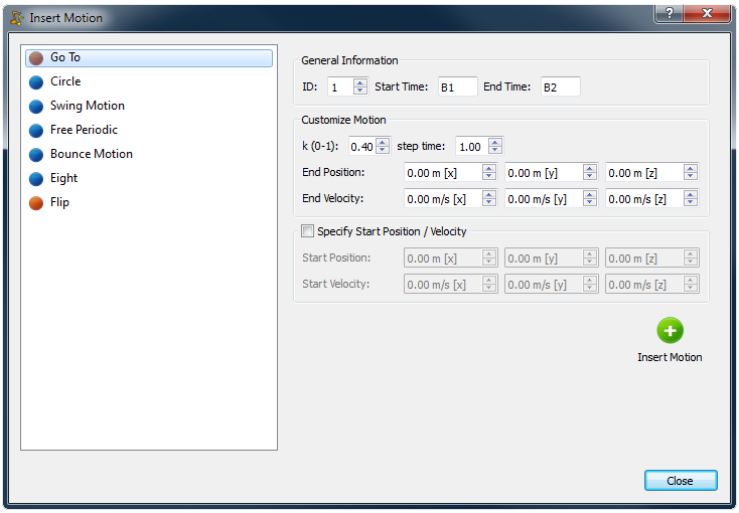

Figure 5.3: The insert motion dialog.

5.3.2 Check Position

We have seen in Section 3.3 that the trajectory should be as smooth as possible. Most important, at least the position must be continuous. Therefore, we added the check position tool which reports the quadrocopters states at the start and at the end of each motion primitives. It is the user's responsibility to develop trajectories that are continuous in the position (and preferably also in the velocity and acceleration). In order to achieve this goal, we suggest the use of the `GOTO` motion or the transition motion.

5.3.3 Preview

Preview offers the possibility to quickly take a look at your freshly designed trajectories. Just enter the time (in seconds) at which you want to start the performance and press the button. The simulation will start. Just look at the 3D GUI in *Inconvenience* to see the results.

5.3.4 Optimize

To generate the special transition motions described in Section 4.3.3, we need to run an optimization routine. This is done by clicking on *Optimize*. The trajectories are then stored in the txt file `yourFile_trajectories.txt` and loaded when flying.

The transitions are assigned as follows:

```
M5_B19 , 1, TRANSITION |p0=[ 1.8 ; 0 ; 3.50] ,v0=[ 0; 2.83 ; 0],
                        a0=[ -4.44 ; -0.01 ; 0.00] , pT=[ 0.50 ; 1.00 ; 4.00],
                        vT=[ 0.00 ; 0.00 ; 0.00], aT=[ -4.93 ; 0.00 ; 0.00],
                        transitionID=1, quadroID=1, END;
M5_B19 , 2, TRANSITION | p0=[ 0.5 ; -1 ; 4], v0=[ -1.57 ; -0.08 ; 0.17],
                        a0=[ -0.26 ; 2.47 ; 0], pT=[ 0.50 ; -1 ; 4] ,
                        aT=[ -4.93 ; 0 ; 0], transitionID=1, quadroID=2, END;
```

5.4 Feasibility Checks

In Chapter 3, we discussed the feasibility question. Those tools were implemented and are ready to use.

5.4.1 Feasibility

In the application *Feasibility Check*, you can simply press the different buttons to check if your choreography is feasible or not. We check for:

- Single vehicle feasibility (see Section 3.2). It reports which constraint is violated or that everything is ok.
- Arena boundaries (3.4.2): Are your trajectories inside the allowable space?
- Multiple vehicles (3.4.1): Here we check for possible intersections. You do not want to destroy any quadrocopter, do you?

5.4.2 Fast Simulation

Another useful application is the *Fast Simulation*. It is designed to test if transitions and trajectories are followed well by the LLC (see Section 3.3 for a discussion). By clicking start, the whole choreography is simulated as fast as possible and any deviation from the desired trajectory is reported. Also the minimum distance between quadrocopters is investigated.

5.5 Flying Mode

The *Flying Mode* is used to actually fly the choreography in the FMA. You first load the choreography file and then can change to this mode. You have to insert the real IDs of the quadrocopters. If you do not want to fly the trajectories of a specific vehicle (ID as defined in the txt choreography file), just enter the *real* ID 99.

After having inserted the IDs, you can start the bridge. This means that the software starts to communicate with the Copilot. The control interface appears.

5.5.1 Architecture

Figure 5.4 depicts the different states in which your vehicles can be. The quadrocopter's behaviour in the various phases is driven by the *AirSupervisor* class. With the graphical interface you can takeoff, land, go to a specific time in the choreography, and start it. The figure also shows in which cases the help class *Collision Free* is used to manage the trajectories for multiple vehicles. Before flying, assure to calibrate the quadrocopters with the Copilot.

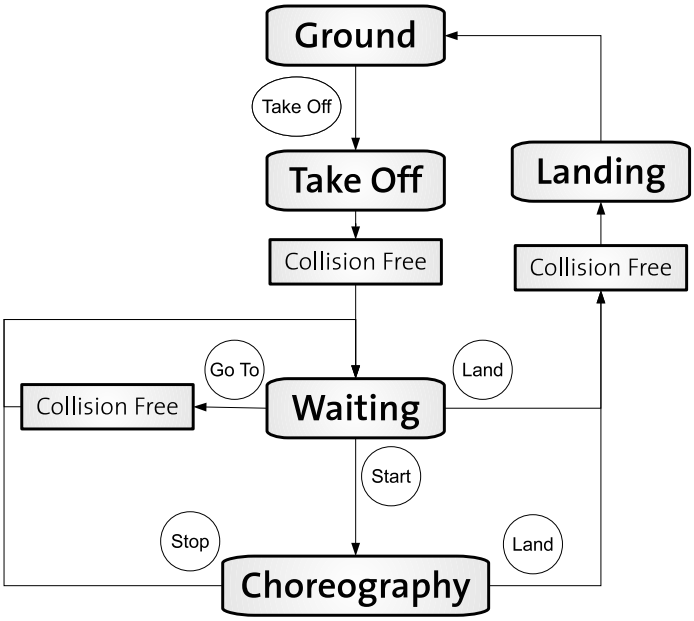

Figure 5.4: The different states of the AirSupervisor class. The round boxes indicate a click on the button of the graphical interface, which triggers a transition to the next state. The help class *Collision Free* plans trajectories for multiple vehicles. Obviously, the first phase is the *Ground*.

5.5.2 Takeoff and Landing Procedures

As shown above, the takeoff and landing procedures use the *Collision Free* class (see Chapter 7). When you press the takeoff button, the Copilot starts the takeoff for all quadrocopters. In the meantime, the software calculates the trajectories from the takeoff/landing points to the start points of the choreography. As soon as the trajectories are calculated, the quadrocopters start flying towards their goal points.

The landing procedure works similarly. It computes the trajectories from the actual points to the takeoff/landing points while the quadrocopters are hovering, and then lands the quadrocopters automatically.

5.6 Extensions

Any extension to the C++ code should be easy. The code is modular and includes comments.

Inserting a New Motion Primitive

A new motion primitive is inherited from the abstract class `MotionPrimitive` and contains some virtual functions as defined in `motionPrimitives.h`. Given the current time, the motion primitive class returns the associated position, velocity and acceleration of the trajectory. When a new motion primitive is added, the file `DanceSupervisor.cpp` is to be updated as well. Indeed, it is responsible for assigning the motion primitives to the quadrocopters.

Updating the Insert Motion Dialog

To do that, you should be familiar with Qt. The different components of the *Insert Motion* dialog, are found in the project's folder under `\insertMotion`. `InsertMotionDialog.cpp` contains the main layout. The other files are the dialogs of the various motion primitives.

Chapter 6

Motion Library

In this section, we provide an overview on the interfaces of the various motion primitives, i.e. all parameters that can, or must, be specified are summarized in a table. The mathematical description of each motion can be found in Chapter 2. Some parameters have a small uppercase letter (^{r} or ^{s}). They can be defined as time-varying parameters, see Section 5.2.2 for an explanation. Many parameters have a default value which is assigned if the parameter is not specified. Make sure that this default value actually exists (e.g. *lastPosition* does not exist at time zero.)

6.1 Common Parameters

Parameters common to all motions are:

Parameter	Default	Description
`yaw`^{rs}	0 [rad]	Allows to specify the yaw profile (see Section 5.2.2).
`feasibilityCheck`	1 [-]	Indicates if the motion primitives should be considered when checking for feasibility (0 = no).

6.2 Periodic Motion Primitives

The next table contains the parameters common to all periodic motion primitives (see Section 2.2.1 for a detailed description). Please read Section 2.2.5 for a discussion about time-varying parameters.

Parameter	Default	Description
`center`^{rs}	- [m]	Is a 3×1 vector. Indicates the center point $x_0(t)$.
`direction`	1 [-]	Choose -1 to reverse the direction.
`phi`^{r}	- [rad]	Allows to modify the phase $\phi_0(t)$.
`nrRounds`	-	Specify the angular frequency Ω. The use of `nrRounds` excludes the use of `beatMultiplier`.
`beatMultiplier`	-	Decide the angular frequency Ω. The use of `beatMultiplier` excludes the use of `nrRounds`.
`FFCorr`	0	Use the feed-forward correction for phase and amplitude (1: yes/0: no)
`FBCorr`	0	Use the feedback correction for phase and amplitude (1: yes/0: no)
`tCompStart`	0 [s]	Start at rest compensation time, see Section 2.2.1.
`tCompEnd`	0 [s]	End at rest compensation time, see Section 2.2.1.
`kComp`	0.5 [-]	The position of the step (0.1-0.9), see Section 2.2.1.

6.2.1 Circle

See Section 2.2.2.

Parameter	Default	Description
`CIRCLE`	-	Motion name.
`radius`^{rs}	- [m]	The radius of the circle.

6.2.2 Swing

See Section 2.2.3.

Parameter	Default	Description
`SWING`	-	Motion name.
`angle`	0 [deg]	The orientation of the swing motion (0-360).
`amp`^{rs}	- [m]	The amplitude L.

6.2.3 Free

See Section 2.2.4.

Parameter	Default	Description
`FREE`	-	Motion name.
`A`	- [-]	The matrix A.
`B`	- [-]	The matrix B.

6.3 Single-shot Motion Primitives

6.3.1 Go To

See Section 2.3.1.

Parameter	Default	Description
`GOTO`	-	Motion name.
`endPosition`	last position	Vector with the desired end position.
`startPosition`	last position	Vector with the desired start position.
`startVelocity`	last velocity	Vector with the desired start velocity.
`endVelocity`	[0; 0; 0]	Vector with the desired end velocity.
`k`	0.5 [-]	The position of the step (0.1-0.9).
`stepTime`	T [s]	The duration of the sinusoidal step.

6.3.2 Flip

See Section 2.3.2.

Parameter	Default	Description
`FLIP`	-	Motion name.
`startPosition`	last position	Vector with the desired start position.
`h`	3 [m]	The total height. Between 2 and 4 meters.

6.4 Collision Free Transitions

See Section 4.3.3.

Parameter	Default	Description
`TRANSITION`	-	Motion name.
`transitionID`	- [-]	The transition ID. Is the same for all the quadrocopters participating in the same transition. Overall, must start at 1.
`quadroID`	- [-]	The internal quadrocopter ID. For every transition, it must start at 1.
`p0`	- [m]	Vector with the desired start position.
`pT`	- [m]	Vector with the desired end position.
`v0`	[0; 0; 0] [m]	Vector with the desired start velocity.
`vT`	[0; 0; 0] [m]	Vector with the desired end velocity.
`a0`	[0; 0; 0] [m]	Vector with the desired start acceleration.
`aT`	[0; 0; 0] [m]	Vector with the desired end acceleration.

Chapter 7

Collision-Free Class

As shown in Chapter 4, we developed a method that allows the safe and fast transition of multiple quadrocopters. In this chapter, we will present how the corresponding class works and the related demonstration that was developed.

7.1 Shared Class

The class is implemented as a `QThread` class. This means, that it will not block your program, since it runs in a different thread. Call `isDone()` to know the state of the computations.

7.1.1 Public Functions

```
CollisionFree(vector<V3D> initPos, vector<V3D> endPos, vector<V3D> initVel,
              vector<V3D> endVel, vector<V3D> initAcc, vector<V3D> endAcc )

CollisionFree(vector<V3D> initPos, vector<V3D> endPos,
              vector<V3D> initVel, vector<V3D> endVel)

CollisionFree(vector<V3D> initPos, vector<V3D> endPos)
```

The constructor needs the desired initial and final states. There are three overloaded constructors. If the velocities or the accelerations are not specified they are initialized to zero.

```
void runIt()
```

Once initialized, you have to run the thread by calling this function.

```
int isDone()
```

Returns the state of the thread. *0* means that it is still running, *1* that a valid solution was found, *2* no solution found, *3* that the initial positions are not valid, *4* indicates invalid end positions, and *5* says that the input vectors are not consistent (different dimensions).

```
vector<vector<V3D>> getPos()
vector<vector<V3D>>  getVel()
```

```
vector<vector<V3D>>  getAcc()
vector<double> getTimes()
```

You can call these functions if `isDone()==1`: you get the position, the velocity and the acceleration for each time step.

```
void setNrTimeSteps(double)
void setTotalTime(double)
```

Define manually (instead of reading from the XML file) the initial number of timesteps pro second and the initial total time.

7.1.2 Example

This is an example on how to use this class (it also contains pseudo-code):

```
#include "../Common/CollisionFree.h"

//Define pointer
shared_ptr<CollisionFree> collisionFreePtr;

...
//prepare the initial and final states
vector<V3D> initPos, initVel, endPos, endVel;
for (i=0;i<N;i++ ) {
        initPos.push_back(startPoints[i]);
        endPos.push_back(endPoints[i]);
}

//initialize the class and run it
collisionFreePtr.reset(new CollisionFree(initPos, endPos));
collisionFreePtr->runIt();


//once the optimization is done, interpolate the trajectories at tCurrent
while(someCondition == true) {
        if (collisionFreePtr->isDone()!=1) {
                //keep controlling the vehicle
                }
        else {
                for (i=0;i<NrQuad;i++ ) {
                    vector<vector<double>> timestamp = CollisionFree->getTimes();
                    vector<vector<V3D>> pos = CollisionFree->getPos();
                    vector<vector<V3D>> vel = CollisionFree->getVel();
                    vector<vector<V3D>> acc = CollisionFree->getAcc();

                    int N = timestamp.size();
                    if (tCurrent>timestamp[N-1]) {
                            tCurrent = timestamp[N-1];
                    } else if (tCurrent<timestamp[0]) {
                            tCurrent = timestamp[0];
                    }

                    //1) find index time into timestamp
                    int index0, index1;
                    for (int i = 1; i<N; i++) {
                            if (timestamp[i]>=tCurrent) {
                                    index0 = i-1;
                                    index1 = i;
                                    break;
                           }
                    }

                    //2) Interpolate position, velocity, acceleration
                    V3D pos0 = pos[i][index0];
                    V3D pos1 = pos[i][index1];
                    V3D vel0 = vel[i][index0];
                    V3D vel1 = vel[i][index1];
                    V3D acc0 = acc[i][index0];
                    V3D acc1 = acc[i][index1];
                    double t0 = timestamp[index0];
```

```
                    double t1 = timestamp[index1];

                    //the trajectories for the ith quadrocopter
                    V3D posVec = pos0 + (tCurrent-t0) * (pos1-pos0)/(t1-t0);
                    V3D velVec = vel0 + (tCurrent-t0) * (vel1-vel0)/(t1-t0);
                    V3D accVec = acc0 + (tCurrent-t0) * (acc1-acc0)/(t1-t0);

                    //Fly
                    doSomething(posVec, velVec, accVec);
                }
            }
}
```

7.2 Demo

A demo was developed to demonstrate the use of this class. The software is located in

`\Users\Federico\CollisionFreeMotion`

The created file should be saved in

`\bin\CollisionFree`.

To use this software you need Cplex [1] and Qt [3] on your computer. If you want to compile it, update the .pro and .bat files with you Cplex and Qt local path, respectively. After you run the software you can choose between two different modes: *Predefined Points* and *Random Points*.
All the necessary parameters with an appropriate explanation are found in Chapter 8.

7.2.1 Predefined Points

The objective of this demonstration, is to show how the trajectories resulting from our algorithm look like. You can define different sets of points and tell the

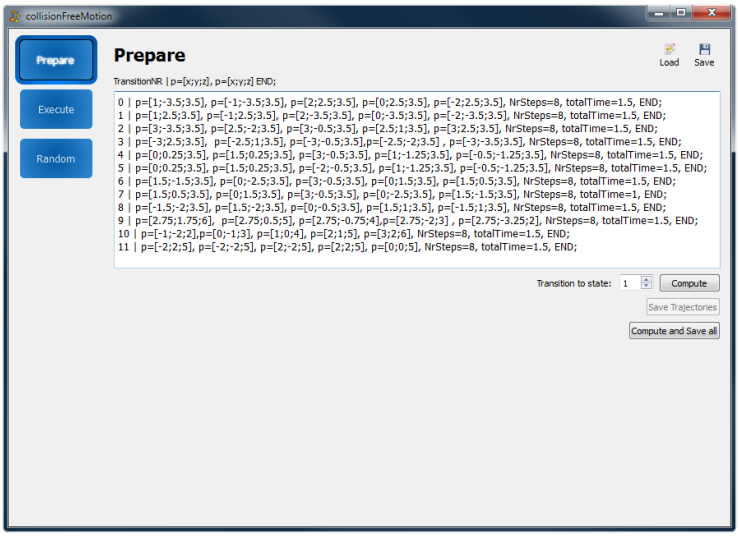

Figure 7.1: The software in *prepare* Mode.

program to create trajectories to transition between them. On the left of the graphical interface (see Figure 7.1), there are three buttons: *Prepare*, *Execute*, and *Random*.

Prepare

First of all, you need to *Prepare* the points, e.g. the code

```
0 | p=[1;−3.5;3.5], p=[−1;−3.5;3.5], NrSteps=8, totalTime=1.5, END;
1 | p=[1;2.5;3.5], p=[−1;2.5;3.5], NrSteps=8, totalTime=1.5, END;
2 | p=[3;−3.5;3.5], p=[2.5;−2;3.5], NrSteps=8, totalTime=1.5, END;
3 | p=[−3;2.5;3.5],  p=[−2.5;1;3.5], NrSteps=8, totalTime=1.5, END;
```

indicates four sets of two points (the i-th column always refers to the i-th quadrocopter). Two quadrocopters change their positions three times. By clicking *Compute and Save All*, a file with the corresponding trajectories is created. If you want to change only a single transition, you can select its number (1,2 or 3 in the above example) and click *Compute* and then *Save Trajectories*. The trajectories are written to a text file called
`yourFilename_trajectories.txt`.

Execute

To execute the precomputed trajectories, you have to select the waiting time between trajectories and then load the corresponding file
`yourFilename_trajectories.txt`. Afterwards, insert the vehicle IDs and press *Start*. The vehicles will take off, fly the trajectories and then safely land.

7.2.2 Random Points

This demo shows that the algorithm can also be used in real time. The optimization procedure usually finds a valid solution within a few seconds. To run this demo, just click on *Random*, select the total number of transitions, the waiting time in between and the number of quadrocopters. Then, insert their IDs and press *Start*. The vehicles will take off, fly the trajectories and then safely land.

Chapter 8

Parameters

In this section, we describe all parameters needed by the code relevant to this thesis. Between square brackets you find good values. Some of them are empirically obtained (i.e. they work well), others are from tests or from measurements.

8.1 ChoreographyFeasibility.xml

Parameter	Default	Unit	Description
`FMABoundaries`			
`xMin`	-3.6	[m]	Limit on negative x for the quadrocopter position.
`xMax`	3.6	[m]	Limit on positive x for the quadrocopter position.
`yMin`	-3.8	[m]	Limit on negative y for the quadrocopter position.
`yMax`	2.6	[m]	Limit on positive y for the quadrocopter position.
`zMin`	0.5	[m]	Limit on negative z for the quadrocopter position.
`zMax`	6.75	[m]	Limit on positive z for the quadrocopter position.
`waterChannelY`	1.30	[m]	Start on the y-coordinate for the water channel boundary (till yMax).
`waterChannelZ`	2.4	[m]	End on the z-coordinate for the water channel boundary.
`VehicleParameters`			
`m`	0.468	[kg]	Mass of a quadrocopter.
`k`	0.016	[-]	Efficiency motor constant.
`L`	0.17	[m]	Arm length.
`Ix`	0.0023	[kg m^2]	Term of the diagonal intertia matrix.
`Iy`	0.0023	[kg m^2]	Term of the diagonal intertia matrix.
`Iz`	0.0046	[kg m^2]	Term of the diagonal intertia matrix.

VehicleConstraints			
fiMax	4.1	[m/s^2]	Normalized maximal thrust allowable for a single rotor.
fiMin	0.6	[m/s^2]	Normalized minimal thrust allowable for a single rotor.
dfiMax	40	[m/s^3]	Normalized maximal thrust derivative value for a single rotor.
omegaMax	25	[rad/s]	Maximum angle rate in pitch and roll.
omegaZMax	5.24	[rad/s]	Maximum angle rate in yaw.

8.2 CollisionFreeCplex.xml

Parameter	Default	Unit	Description
`Setup`			
`totalTime`	2	[s]	The total duration of the transition used for the first iteration.
`timeSteps`	6	[-]	The number of timesteps per second for the first iteration.
`maxIter`	20	[-]	Total maximum of iterations in a single SCP step (max k).
`tolerance`	0.015	[-]	Bound on the percentual change of objective function to tell that convergence is achieved
`printTimeInformation`	false	[-]	Display time information.
`timestepIncrease`	1	[-]	Increase in the number of timesteps per second if the first SCP results unfeasible.
`timeIncrease`	0.5	[s]	Increase in the total duration of the transition if the first SCP results unfeasible.
`maxTime`	50	[s]	Maximum time for the algorithm to find a solution.
`ArenaBoundaries`			
`xMin`	-3.5	[m]	Limit on negative x for the quadrocopter position in the collision-free algorithm. Should be smaller than the arena boundaries.
`xMax`	3.5	[m]	Limit on positive x for the quadrocopter position in the collision-free algorithm. Should be smaller than the arena boundaries.
`yMin`	-3.7	[m]	Limit on negative y for the quadrocopter position in the collision-free algorithm. Should be smaller than the arena boundaries.
`yMax`	2.5	[m]	Limit on positive y for the quadrocopter position in the collision-free algorithm. Should be smaller than the arena boundaries.
`zMin`	0.5	[m]	Limit on negative z for the quadrocopter position in the collision-free algorithm. Should be smaller than the arena boundaries.
`zMax`	6.75	[m]	Limit on positive z for the quadrocopter position in the collision-free algorithm. Should be smaller than the arena boundaries.

`RandomPoints`

`xMin`	-3.5	[m]	Limit on negative x for the quadrocopter position in the choice of random points. Should be smaller than the boundaries in the collision-free algorithm.
`xMax`	3.5	[m]	Limit on positive x for the quadrocopter position in the choice of random points. Should be smaller than the boundaries in the collision-free algorithm.
`yMin`	-3.7	[m]	Limit on negative y for the quadrocopter position in the choice of random points. Should be smaller than the boundaries in the collision-free algorithm.
`yMax`	2.5	[m]	Limit on positive y for the quadrocopter position in the choice of random points. Should be smaller than the boundaries in the collision-free algorithm.
`zMin`	0.5	[m]	Limit on negative z for the quadrocopter position in the choice of random points. Should be smaller than the boundaries in the collision-free algorithm.
`zMax`	6.75	[m]	Limit on positive z for the quadrocopter position in the choice of random points. Should be smaller than the boundaries in the collision-free algorithm.

`WaterChannel`

`Rwc`	2.1	[m]	The radius for the ellipsoid boundary on the water channel.
`centerY`	2.8	[m]	The y-position of the center for the ellipsoid boundary on the water channel.
`centerZ`	1.25	[m]	The z-position of the center for the ellipsoid boundary on the water channel.
`zCompression`	1,25	[-]	The compression along the z direction. (see Matlab files on the CD).

`Distance`

`Rmin`	2.1	[m]	The minimum distance between quadrocopter in the collision-free algorithm .
`zCompression`	0.66	[-]	The compression along the z-direction. (see Matlab files on the CD).
`trueMinDistance`	1.25	[m]	The minimum distance between quadrocopters for the interpolated trajectories.

`VehicleConstraints`

`maxCollThrust`	0	[m/s^2]	Maximum normalized collective thrust. Set 0, to only use the affine constraint on the acceleration.
`maxJerk`	23	[m/s^3]	Maximum jerk in a single direction.
`maxAccXY`	15	[m/s^2]	Maximum acceleration (absolute value) in the x and y directions.
`maxAccZ`	12	[m/s^2]	Maximum acceleration in the z-direction.
`minAccZ`	-8	[m/s^2]	Maximum acceleration in the z-direction.

8.3 ChoreographyGenerator.xml

Parameter	Default	Unit	Description
`Logging`			
`logging`	true	[-]	If the position should be loged.
`logName`	test	[-]	The log filename stored in \bin\ChoreographyFiles\Logs.
`PeriodicMotions`			
`ampGain`	0.01	[-]	Amplitude gain for the feedback correction of the synchronization algorithm.
`phaseGain`	0.15	[-]	Phase gain for the feedback forrection of the synchronization algorithm.
`N`	1	[-]	Number of periods for the feedback forrection of the synchronization algorithm.
`Flip`			
`flipDuration`	0.4	[s]	The duration of the flip phase.
`maxOmega`	0.24	[rad/s]	The maximal value of the angle rate.

Conclusions

This Master Thesis presented a tool for creating quadrocopter choreography at the ETH Flying Machine Arena. Part I treated the mathematical aspects of this work, whereas Part II introduced the reader to the developed software.

This project could possibly be a never-ending one. Especially, the software could be improved a lot in all aspects related to its usability. For example, Drag and Drop of motion primitives could be added, and the scripting language can be replaced by graphical objects.

Future Work

Beside the software, also the overall system can be improved. New directions may be explored and some parts need refinement. I think that the most important steps for the future are:

- *Automated generation of a choreography based on human rating* – The idea is to create a system that combines motion primitives in a meaningful way given the music structure. The decision process can be improved by exploiting human rating and evaluation.

- *Interaction* – A possible new direction is real-time interaction. A quadrocopter could be told what to do by a human instructor in the space. This could possibly mimic the human process of testing and learning a choreography.

- *Feasibility* – The method which was developed to test the feasibility of a trajectory was not experimentally validated. We only ran some preliminary tests which supported the theory.

- *Automated music analysis* – In order to achieve a fully automated choreography generation, also the music structure should be provided automatically. Software should be tested and, possibly, a collaboration with the Music Information Retrieval community should be sought.

- *Software improvements* – As mentioned, there is great room for improvement on the software side. Once the final structure of the project and most of the tools are developed, we could outsource this task.

Acknowledgments

After six months of work, some acknowledgments are due. First of all, I would like to thank my girlfriend Francesca, for the many suggestions for the choreographies and for her constant support. I am also grateful to my family, which keeps asking for a personalized dancing robot.

On the academic level, I thank Angela Schoellig for her supervision during this project and the helpful comments. I am also grateful to the whole FMA team for the support. Finally, I say thank you to Professor Raffaello D'Andrea who supported and believed in this project.

Appendix A

Errata

This appendix corrects for errors and inaccuracies in this Master Thesis. If you find any error in the original document, please contact me at faugugliaro@ethz.ch. An updated version of this document is found online at www.f-augugliaro.ch.

Chapter 3 - Feasibility

- Section 3.2.3: The statement $\omega_z = \dot{\psi}$ is a valid approximation only for very small ϕ and θ. The full expression is:

$$\omega_z = \frac{\dot{\psi}\cos\phi\cos\theta - \omega_x\cos\phi\sin\phi}{1-\sin\phi\sin\theta}. \tag{A.1}$$

Bibliography

[1] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/. Software. Accessed on 08/30/2011.

[2] The Intelligent Music Application Platform – The Echo Nest. http://the.echonest.com/. Software. Accessed on 09/30/2011.

[3] Qt - Cross-platform Application and UI Framework. http://qt.nokia.com/. Library. Accessed on 08/30/2011.

[4] *The Proceedings of IROS 2010 Workshop on Robots and Musical Expressions (IWRME-2010)*, 2010. http://winnie.kuis.kyoto-u.ac.jp/RMEWS/proc/. Accessed on 09/25/2011.

[5] J.J. Aucouturier. Cheek to Chip: Dancing Robots and AI's Future. *Intelligent Systems, IEEE*, 23(2):74–84, 2008.

[6] F. Augugliaro, A. Schoellig, and R. D'Andrea. Dance of the Quadrocopters – Armageddon. http://youtu.be/7r281vgfotg. Video. Accessed on 09/30/2011.

[7] F. Augugliaro, A. Schoellig, and R. D'Andrea. Dance Together! http://youtu.be/NPvGxIBt3Hs. Video. Accessed on 08/30/2011.

[8] F. Augugliaro, A. Schoellig, and R. D'Andrea. Dance with Three. http://youtu.be/DrHlgxf0oQw. Video. Accessed on 08/30/2011.

[9] F. Augugliaro, A. Schoellig, and R. D'Andrea. Fast Transitions of a Quadrocopter Fleet Using Convex Optimization. http://youtu.be/wwK7WvvUvlI. Video. Accessed on 09/30/2011.

[10] E. Avrunin, J. Hart, A. Douglas, and B. Scassellati. Effects Related to Synchrony and Repertoire in Perceptions of Robot Dance. In *Proceedings of the 6th International Conference on Human-robot Interaction*, HRI '11, pages 93–100, New York, NY, USA, 2011. ACM.

[11] S. Boyd. Sequential Convex Programming. http://www.stanford.edu/class/ee364b/lectures/seq_slides.pdf, 2011. Lecture Slides. Accessed on 09/30/2011.

[12] S.P. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge Univ Pr, 2004. Available Online: http://www.stanford.edu/ boyd/cvxbook/. Accessed 09/25/2011.

[13] S. Dixon. BeatRoot: an Interactive Beat Tracking and Visualisation System. http://www.eecs.qmul.ac.uk/~simond/beatroot. Software. Accessed on 08/30/2011.

[14] D. Grunberg, R. Ellenberg, Y. Kim, and P. Oh. Creating an Autonomous Dancing Robot. In *Proceedings of the International Conference on Hybrid Information Technology (ICHIT)*, pages 221–227, 2009.

[15] M. Hehn and R. D'Andrea. Quadrocopter Trajectory Generation and Control. In *IFAC World Congress*, pages 3355–3360, 2011.

[16] T. Heinis. Exploring Software Tools for Music Analysis. Studies on Mechatronics, IDSC, ETH Zurich, March 2011.

[17] Y.E. Kim, A.M. Batula, D. Grunberg, D.M. Lofaro, J. Oh, and P.Y. Oh. Developing Humanoids For Musical Interaction. In *10th IEEE-RAS International Conference on Humanoid Robots*, 2010.

[18] S. Kudoh, T. Okamoto, T. Shiratori, S. Nakaoka, and K. Ikeuchi. Towards a Dancing-to-music Humanoid Robot: Temporal Scaling Model of Whole Body Motion for a Dancing Humanoid Robot. In *Intelligent Robots and Systems – Workshop on Robots and Musical Expressions, International Conference on*, 2010.

[19] S. Lupashin, A. Schoellig, M. Hehn, and R. D'Andrea. The Flying Machine Arena as of 2010. In *Robotics and Automation (ICRA), International Conference on*, pages 2970 –2971. IEEE, may 2011.

[20] M. P. Michalowski, S. Sabanovic, and H. Kozima. A Dancing Robot for Rhythmic Social Interaction. In *Proceedings of the ACM/IEEE International Conference on Human-robot Interaction*, HRI '07, pages 89–96, New York, NY, USA, 2007. ACM.

[21] J.L. Oliveira, L. Naveda, F. Gouyon, M. Leman, and L.P. Reis. Synthesis of Dancing Motions Based on a Compact Topological Representation of Dance Styles. In *Intelligent Robots and Systems – Workshop on Robots and Musical Expressions, International Conference on*. IEEE/RSJ, 2010.

[22] A. Schoellig, F. Augugliaro, and R. D'Andrea. A Platform for Dance Performances with Multiple Quadrocopters. In *Intelligent Robots and Systems – Workshop on Robots and Musical Expressions, International Conference on*, pages 1–8. IEEE/RSJ, 2010.

[23] A. Schoellig, F. Augugliaro, S. Lupashin, and R. D'Andrea. Synchronizing the Motion of a Quadrocopter to Music. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3355–3360, 2010.

[24] A. Schoellig, M. Hehn, S. Lupashin, and R. D'Andrea. Feasiblity of Motion Primitives for Choreographed Quadrocopter Flight. In *Proceedings of the American Control Conference (ACC)*, pages 3843–3849, 2011.

[25] M. Sherback. *Coordinate Systems for the FMA: Decisions, Fundamentals, and Helpful Formulas*. IDSC, ETH Zurich, December 2009. Internal Report.

[26] M. Sherback. *Single Process and Simulator Documentation*. IDSC, ETH Zurich, Spring 2010. Internal Report.

[27] B. Siciliano, L. Sciavicco, and L. Villani. *Robotics: Modelling, Planning and Control*. Springer Verlag, 2009.

[28] Y. Wang. Applications of Convex Optimization in Control, May 2011. Talk.

[29] C. Wiltsche. Synchronized Motion in Three Dimensions Based on Feed Forward Compensation. Semester Project, IDSC, ETH Zurich, July 2011.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control
Prof. Dr. R. D'Andrea, Prof. Dr. L. Guzzella

Title of work:

Dancing Quadrocopters: Trajectory Generation, Feasibility, and User Interface

Thesis type and date:

Master Thesis, September 2011

Supervision:

Angela Schoellig
Prof. Raffaello D'Andrea

Student:

Name: Federico Augugliaro
E-mail: faugugli@student.ethz.ch
Semester: 10