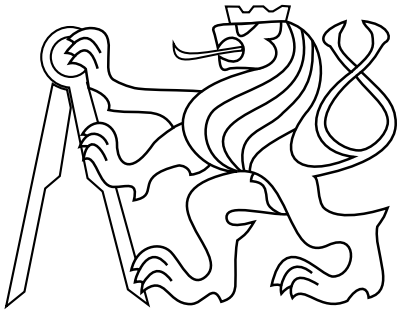
Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Cybernetics
Multi-robot Systems



Model Predictive Path Integral Control of a Drone Using a Database of Motion Primitives

Master's Thesis

Michal Minařík

Prague, January 2024

Study programme: Cybernetics and Robotics
Supervisor: Ing. Vojtěch Vonásek, Ph.D.

Acknowledgments

First and foremost, I would like to thank my supervisor Ing. Vojtěch Vonásek, Ph.D. and advisor Ing. Robert Pěnička, Ph. D. for their assistance and support. I would also like to thank my family and friends for supporting me during my studies and proofreading this thesis.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Minařík Michal** Personal ID number: **483465**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Model Predictive Path Integral Control of a Drone Using a Database of Motion Primitives

Master’s thesis title in Czech:

Řízení dronu metodou Model Predictive Path Integral za pomoci databáze manévrů

Guidelines:

1. Get familiar with methods for generating time-optimal trajectories for autonomous drones [1]. Propose and implement a method for generating time-optimal motion primitives, i.e., short trajectories containing both the states of drone dynamics and the control inputs. Consider environments without obstacles.
2. Design and implement a database containing the generated motion primitives which allows time efficient querying during the drone flight [2].
3. Implement MPPI controller [3] to fly through a predefined sequence of 3D waypoints using the motion primitives to guide the controller.
4. Verify the designed methods in a simulation environment [4].
5. (optional) Test the proposed algorithms on a physical drone.
6. (optional) Extend the method to environments with obstacles.

Bibliography / sources:

[1] P. Foehn, A. Romero, and D. Scaramuzza, “Time-optimal planning for quadrotor waypoint flight,” Science Robotics, vol. 6, July 2021.
[2] H. Samet, Foundations of Multidimensional And Metric Data Structures. Morgan Kaufmann, Aug. 2006.
[3] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, “Aggressive driving with model predictive path integral control,” in 2016 IEEE International Conference on Robotics and Automation (ICRA), (Stockholm), pp. 1433–1440, IEEE, May 2016.
[4] T. Baca, M. Petrlik, M. Vrba, V. Spurny, R. Penicka, D. Hert, and M. Saska, “The MRS UAV System: Pushing the Frontiers of Reproducible Research, Real-world Deployment, and Education with Autonomous Unmanned Aerial Vehicles,” Journal of Intelligent & Robotic Systems, vol. 102, p. 26, May 2021.

Name and workplace of master’s thesis supervisor:

Ing. Vojtěch Vonásek, Ph.D. Multi-robot Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **08.09.2023** Deadline for master's thesis submission: **09.01.2024**

Assignment valid until: **16.02.2025**

Ing. Vojtěch Vonásek, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 9. 1. 2024 .....

Michal Minařík

Abstract

In this thesis, the Model Predictive Path Integral control methodology is used to control a drone. The Model Predictive Path Integral method allows the use of a nonlinear model of the drone dynamics and a more general cost function at the cost of a high computational demand. To run the controller in real-time, the sampling-based optimization is performed in parallel on a graphics processing unit. An important part of the optimization is a correct initialization of the optimization variables — the control input sequence. A database containing short time-optimal trajectories is proposed to initialize the variables and guide the controller in real-time. The proposed methods are tested in two different drone dynamics simulators. These experiments demonstrate the ability of the controller to fly the drone along a specified sequence of 3D waypoints, even in an environment with obstacles.

Keywords Unmanned Aerial Vehicles, Automatic Control, Predictive Control, Model Predictive Path Integral

Abstrakt

Tato práce používá metodu Model Predictive Path Integral pro řízení dronu. Zmíněná metoda umožňuje použití nelineárního modelu dynamiky dronu a obecnější účelové funkce za cenu vysoké výpočetní náročnosti. Aby mohl řídící algoritmus běžet v reálném čase, optimalizace je prováděna paralelně na grafické kartě. Důležitou součástí optimalizace je správná inicializace optimalizačních proměnných — sekvence řídicích vstupů. Pro inicializaci proměnných a navádění regulátoru v reálném čase je navržena databáze obsahují krátké časově optimální trajektorie. Navržené metody jsou otestovány ve dvou různých simulátorech dynamiky dronu. Provedené experimenty demonstrují schopnost regulátoru řídit dron při letu podél sekvence bodů ve 3D, a to i v prostředí obsahující překážky.

Klíčová slova Bezpilotní Prostředky, Automatické Řízení, Prediktivní Řízení, Model Predictive Path Integral

Abbreviations

CTU Czech Technical University

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

GPU Graphics Processing Unit

KKT Karush-Kuhn-Tucker

LiDAR Light Detection and Ranging

MPC Model Predictive Control

MPCC Model Predictive Contouring Control

MPPI Model Predictive Path Integral

MRS Multi-robot Systems Group

NLP Nonlinear Program

NMPC Nonlinear Model Predictive Control

PID Proportional-Integral-Derivative controller

RRT Rapidly-exploring Random Tree

UAV Unmanned Aerial Vehicle

Contents

Chapter 1

Introduction

Over the past few decades, Unmanned Aerial Vehicles (UAVs), commonly referred to as drones, have witnessed a large rise in popularity. This can be attributed to the advances in integrated circuit production, making the hardware smaller, thereby decreasing the space and weight the UAV needs to provide and carry. Better availability and reduced costs allowed their massive adoption in several industries, such as surveillance, agriculture, and art [10], [14].

Despite the possible agility and maneuverability, the current applications are unable to use the full potential of the drones. The dynamic abilities are showcased during drone racing, where human pilots exhibit a great ability to control the drones while flying through complex tracks [7]. The challenges of controlling a drone during a fast and agile flight include modeling obstacles and handling constraints. Traditionally, the flight is planned on a high level as a path given to lower-level trackers that solve time allocation, and the trajectory is then tracked by low-level controllers such as Proportional-Integral-Derivative controller (PID). This hierarchical division allows safe and robust control of the drone. However, it falls short of fully exploiting the platform capabilities since satisfying the actual physical constraints (e.g., motor speed limits) is not integrated into the planning itself.

To introduce the constraints directly into the control task, advanced techniques such as Model Predictive Control (MPC) are used [8]. Researchers have shown that MPC can perform well but it also has multiple limitations. Namely, the traditional MPC requires a linear system of the model and imposes restrictions on the form of the cost function and the constraints.

Nonlinear Model Predictive Control (NMPC) is used to alleviate the linear model restrictions [4]. It allows the use of a more precise, nonlinear model of the drone dynamics. However, the other restrictions still hold — an example of this is universal avoidance of non-analytic[1], non-convex obstacles, which is almost impossible to introduce into the MPC framework.

In this thesis, we will explore the capabilities of another predictive control method, namely Model Predictive Path Integral (MPPI) [15], [19], and apply it to the task of flying through a sequence of 3D waypoints in a time-efficient and agile manner. Using sampling-based methods such as MPPI removes the restrictions on the cost function, model dynamics, and state constraints at the expense of a high computation demand. This demand is satisfied using the Compute Unified Device Architecture (CUDA) platform, allowing the parallelization of the calculations on a Graphics Processing Unit (GPU). We will cover multiple possibilities the framework offers, starting with the ability to track a constant velocity 3D trajectory reference while satisfying multiple constraints imposed on the controls and states. Furthermore, we will show the ability to add an arbitrary collision detection module into the control cost function.

[1] not described by well-known mathematical expressions, such as polynominals and trigonometric functions

As a predictive controller, MPPI takes advantage of having a good estimate of the future trajectory to track and a good initialization of the control inputs. Most current state-of-the-art approaches to time-optimal multi-waypoint flight split the task into two problems — offline planning task where a global, time-optimal trajectory is computed [7] and online tracking of the computed trajectory. However, computing the full time-optimal trajectory (which considers the full quadrotor model compared to a simpler model such as point mass) requires solving a complex time allocation problem through numerical optimization and is currently very computationally demanding (requiring minutes or even hours on full-scale computers [9]). This means that while tracking the preplanned trajectory, unexpected changes to the environment (e.g., obstacles not included in the planning) can have a detrimental effect on the controller's performance since the tracked reference cannot be updated online. In this thesis, we will try to analyze the possibility of employing the MPPI controller to iteratively select and track short trajectories (motion primitives) while adapting to the current scenarios in real-time.

This thesis should primarily serve as a proof of concept, showing the ability of the MPPI controller to control a drone under limited hardware capabilities. We cover reference tracking and obstacle avoidance, showcasing the advantages and discussing the disadvantages of using MPPI. Moreover, we propose a novel method of providing the controller with precomputed time-optimal trajectories as real-time guides to increase the controller's performance.

1.1 Thesis structure

Chapter 1 presents the motivation and the goals of our work. It serves as an introduction to the areas of concern and outlines the structure of this thesis.

The fundamentals of optimal and predictive control are covered in Chapter 2. We define the control problem and introduce methods used to tackle this problem. Most importantly, we present a sampling-based control framework (MPPI) which serves as a foundation of this thesis. Finally, we introduce the mathematical model of the controlled system (a drone).

In Chapter 3, we present work already published by other authors in the main areas related to this thesis. The chapter covers the works on reference trajectory generation and MPPI improvement, along with a discussion of our contributions and differences to the current state-of-the-art.

In Chapter 4, we cover generating a set of short time-optimal trajectories (motion primitives), storing them in a database that allows time-efficient querying, and using them to guide the MPPI controller. All implementation details are provided in this chapter.

The proposed methods are experimentally verified in Chapter 5. In this chapter, we show that the controller with all proposed modifications exhibits the expected behavior and works in multiple environments, even in environments with obstacles. Moreover, we reimplement our controller in third-party simulation software to ensure and further confirm its correct function.

Achieved results and possible improvements are further discussed in Chapter 6, which also serves as an overall summary of this thesis.

Chapter 2

Preliminaries

This chapter serves as the foundation of this master thesis, laying the groundwork for the following chapters. In Section 2.1, we define the problem of optimal and predictive control and introduce the main algorithm used in this thesis to tackle the problem. Section 2.2 introduces a mathematical model of the system dynamics used to simulate the system evolution.

2.1 Optimal and Predictive Control

The goal of optimal control is to formulate a control problem as a mathematical optimization problem and find the optimal values of the optimization variables that minimize a given cost function while satisfying given constraints. The cost function can range from stabilizing a system at a given state, tracking a given reference, or minimizing the time needed to transition from one state to another. The optimization can be formulated in either continuous or discrete time. In continuous time, the control signals are functions, and the optimization can be carried out using methods from functional analysis (such as calculus of variations). In discrete time, the signals are vectors (sequences of real numbers), which is easier to tackle since more tools are available [3].

One popular methodology is predictive control, which uses the knowledge about the system being controlled. Among the advantages of predictive control, contrary to controllers such as PID, is the possibility of leveraging information about future reference and acting accordingly.

In this thesis, we will consider a time-invariant dynamical system with dynamics

$$\boldsymbol{x}_{j+1} = f(\boldsymbol{x}_j, \boldsymbol{u}_j) \qquad j = 0, \ldots, N-1, \tag{2.1}$$

where $\boldsymbol{x} = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_N)$ is a sequence of the system states $\boldsymbol{x}_j \in \mathbb{R}^{n_x}$ and $\boldsymbol{u} = (\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{N-1})$ are the system inputs $\boldsymbol{u}_j \in \mathbb{R}^{n_u}$. Then, a discrete-time, finite horizon optimal control problem can be formulated as

$$\begin{aligned}
\textbf{optimize} \qquad & \boldsymbol{u}^* = \underset{\boldsymbol{u}}{\operatorname{argmin}} \sum_{j=0}^{N-1} L(\boldsymbol{x}_j, \boldsymbol{u}_j) + E(\boldsymbol{x}_N) \\
\textbf{subject to} \qquad & \\
& \boldsymbol{x}_{j+1} = f(\boldsymbol{x}_j, \boldsymbol{u}_j) && j = 0, \ldots, N-1 \\
& \boldsymbol{h}(\boldsymbol{x}_j, \boldsymbol{u}_j) \leq \boldsymbol{0} && j = 0, \ldots, N-1, \\
& \boldsymbol{x}_0 = \boldsymbol{x}_{\text{init}}
\end{aligned} \tag{2.2}$$

where $\boldsymbol{h}$ are the input and state constraints, $\boldsymbol{x}_{\text{init}}$ is the initial state of our system, $L(\boldsymbol{x}_j, \boldsymbol{u}_j)$ is the running cost function, and $E(\boldsymbol{x}_N)$ is the terminal state cost function. This type of formulation is called a *sequential approach* with a *fixed initial state*, since we fix the initial state to $\boldsymbol{x}_{\text{init}}$ and express the states $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ by forward simulation of the dynamics f from the initial state $\boldsymbol{x}_0 = \boldsymbol{x}_{\text{init}}$, applying the inputs $\boldsymbol{u}_j$. An alternative formulation is the *simultaneous approach*, where we use both the states $\boldsymbol{x}_j$ and the inputs $\boldsymbol{u}_j$ as optimization variables of the Nonlinear Program (NLP). Simultaneous approaches include *direct multiple shooting* and *direct collocation* [3].

A widely used approach to finding a solution to the problems mentioned above are *Newton-type optimization methods*, which apply a variant of the *Newton's method* to solve the nonlinear Karush-Kuhn-Tucker (KKT) conditions. However, describing these methods is out of the scope of this thesis, and we will rely on optimization software such as CasADi [13] or Acados [11] to solve the formulated problems.

In the rest of this chapter, we will concern ourselves with the sequential formulation. One of the simultaneous approaches, namely the multiple shooting method, will appear in Section 4.1, where the authors of [7] use this approach to formulate and solve a time-optimal control problem.

2.1.1 Model Predictive Control

Model Predictive Control (MPC) aims to control the system by solving an open-loop optimization problem over N discrete time steps into the future (*prediction horizon*). The first control input is then applied to the system, and the optimization is repeated at the next time step — either starting from scratch or reusing the results of the prior optimization as an initialization. This results in a closed-loop feedback controller [12].

MPC is widely used for regulation and reference tracking. A linear MPC regulation problem with constrained state and input values can be formulated as

$$
\begin{aligned}
&\textbf{optimize} && \boldsymbol{u}^* = \underset{\boldsymbol{u}}{\operatorname{argmin}}\ \boldsymbol{x}_N^T S \boldsymbol{x}_N + \sum_{j=0}^{N-1} \left[\boldsymbol{x}_j^T Q \boldsymbol{x}_j + \boldsymbol{u}_j^T R \boldsymbol{u}_j\right] && \\
&\textbf{subject to} && && \\
&\textit{system dynamics} && \begin{cases} \boldsymbol{x}_0 = \boldsymbol{x}_{init} & \\ \boldsymbol{x}_{j+1} = A\boldsymbol{x}_j + B\boldsymbol{u}_j & j = 0, \ldots, N-1 \end{cases} && (2.3)\\
&\textit{input and state constraints} && \begin{cases} \boldsymbol{x}_{\min} \leq \boldsymbol{x}_j \leq \boldsymbol{x}_{\max} & j = 0, \ldots, N \\ \boldsymbol{u}_{\min} \leq \boldsymbol{u}_j \leq \boldsymbol{u}_{\max} & j = 0, \ldots, N-1 \end{cases} &&
\end{aligned}
$$

where $\boldsymbol{x}_j$ and $\boldsymbol{u}_j$ are the system state and input at the discrete time step j, A and B are the matrices of the state-space system description, S, Q and R are positive semidefinite cost matrices, N is the prediction horizon, and $\boldsymbol{x}_{\min/\max}$ and $\boldsymbol{u}_{\min/\max}$ are the state and input limits.

Even though MPC has shown itself to be a very powerful tool and is used in numerous ap-

plications, it comes with multiple disadvantages that researchers are trying to tackle. Most of the traditional MPC variants require a convex approximation of the cost function and a low-order (first or second) approximation of the system dynamics [19]. This requirement of a convex cost function (or at least a smooth function with a few local minima) comes from the use of the gradient-based optimization methods, and can make the task formulation a time-consuming problem. It can also hinder the possibility of creating high-level, easily interpretable cost function representations, which can be desired in complex system control, where explainability is required. For example, including obstacle avoidance in the MPC framework is challenging without relying on convex and analytical representations of the obstacle regions. With the above-mentioned disadvantages of MPC and possible remedies in mind, we will present a Monte Carlo sampling approach to solving the optimization task, which allows for more general settings for the price of increased computation complexity.

2.1.2 Model Predictive Path Integral Control

Model Predictive Path Integral (MPPI) is a predictive control algorithm designed to control nonlinear systems [15], [19]. The core idea is similar to MPC, however, instead of employing an optimization algorithm, a Monte Carlo sampling approach is used. This shift to sampling-based optimization allows for general, non-convex cost criteria [17], [19]. Moreover, since no gradient-based optimization is used to find and improve the solution, we can utilize simple encodings of task descriptions with sparse (or nonexistent) gradients.

As MPPI is a predictive algorithm, we need to set a parameter N, which controls the number of discrete time samples to be evaluated into the future (*prediction horizon*). Another parameter is the number of rollouts that will be computed at each iteration, which will be denoted K. Let us denote the current state estimate as $\hat{x}$ and the nominal control sequence $\boldsymbol{u}^{\text{nom}}$ (sequence of N control actions obtained by initialization or previous optimization iterations). K disturbance sequences of length N are sampled from a normal distribution with a zero mean and a covariance matrix Σ. We will use lower index j to denote the discrete time index and upper index k to denote the rollout index

$$\left.\begin{aligned} \boldsymbol{x}^k &= (\boldsymbol{x}_0^k, \dots, \boldsymbol{x}_j^k, \dots, \boldsymbol{x}_{N-1}^k, \boldsymbol{x}_N^k) \\ \boldsymbol{u}^k &= (\boldsymbol{u}_0^k, \dots, \boldsymbol{u}_j^k, \dots, \boldsymbol{u}_{N-1}^k) \\ \boldsymbol{\delta}^k &= (\delta\boldsymbol{u}_0^k, \dots, \delta\boldsymbol{u}_j^k, \dots, \delta\boldsymbol{u}_{N-1}^k) \end{aligned}\right\} \quad k = 1, \dots, K. \tag{2.4}$$

From the initial state $\hat{x}$, K rollouts are computed by forward simulation of the system dynamics, applying the disturbed nominal control

$$\left.\begin{aligned} \delta\boldsymbol{u}_j^k &\in \mathcal{N}(0, \Sigma) \\ \boldsymbol{u}_j^k &= \boldsymbol{u}_j^{\text{nom}} + \delta\boldsymbol{u}_j^k \\ \boldsymbol{x}_{j+1}^k &= \boldsymbol{x}_j^k + \Delta t \cdot \boldsymbol{f}_{\text{RK4}}(\boldsymbol{x}_j^k, \boldsymbol{u}_j^k) \end{aligned}\right\} \quad \begin{aligned} k &= 1, \dots, K \\ j &= 0, \dots, N-1. \end{aligned} \tag{2.5}$$

After the rollouts are computed, each rollout is evaluated by a task-specific cost function (the lower the cost, the better the trajectory is considering the problem specification)

$$S_k = \text{ComputeCost}(\boldsymbol{x}^k, \boldsymbol{u}^k), \tag{2.6}$$

and the costs are transformed into weights $(\omega_1, \omega_2, \ldots, \omega_K)$

$$\omega_k = \frac{1}{\eta} \exp\left(-\frac{1}{\lambda}\left(S_k - \rho\right)\right), \qquad \eta = \sum_{k=1}^{K} \exp\left(-\frac{1}{\lambda}\left(S_k - \rho\right)\right), \qquad \rho = \min\{S_1, \ldots, S_K\}. \tag{2.7}$$

This procedure is essentially the softmax transform used heavily in neural networks to normalize a vector of K values into a probability distribution. Since softmax is invariant to shift in inputs, we can improve the numerical stability by subtracting $\rho = \min(\mathcal{S}_k)$ in softmax from all the costs without changing the resulting weights ω_k

$$\omega_k(\mathcal{S}_k - \rho) = \frac{e^{-\frac{1}{\lambda}(S_k - \rho)}}{\sum_{j=1}^{K} e^{-\frac{1}{\lambda}(S_j - \rho)}} = \frac{e^{-\frac{\mathcal{S}_k}{\lambda}} e^{\frac{\rho}{\lambda}}}{\sum_{j=1}^{K} e^{-\frac{\mathcal{S}_j}{\lambda}} e^{\frac{\rho}{\lambda}}} = \frac{e^{-\frac{\mathcal{S}_k}{\lambda}}}{\sum_{j=1}^{K} e^{-\frac{\mathcal{S}_j}{\lambda}}} = \omega_k(\mathcal{S}_k). \tag{2.8}$$

The weight computation is summarized in Alg. 1, and the whole rollout evaluation is visualized in Figure 2.1. After computing the weights, the nominal control actions are updated by a weighted average of the disturbances

$$\boldsymbol{u}_j^{\text{nom}} := \boldsymbol{u}_j^{\text{nom}} + \sum_{k=1}^{K} \omega_k \cdot \delta \boldsymbol{u}_j^k. \tag{2.9}$$

The parameter λ scales the contribution of the trajectory rollouts to the result based on their evaluated cost, ranging from taking only the best rollout to averaging all disturbances. When λ is low, the control disturbances resulting in the best rollout have a significant impact on the updated control (as λ approaches zero, the weight vector approaches $(0, \ldots, 0, 1, 0, \ldots, 0)$ with one at the place of the best rollouts). Conversely, when λ is high, the weight vector approaches $(\frac{1}{K}, \ldots, \frac{1}{K})$. The whole algorithm is presented in Alg. 2. To start the algorithm, we need to initialize the control over the control horizon N. In our case, we will set the input to zero desired body rates and a constant collective thrust, resulting in a hover state.

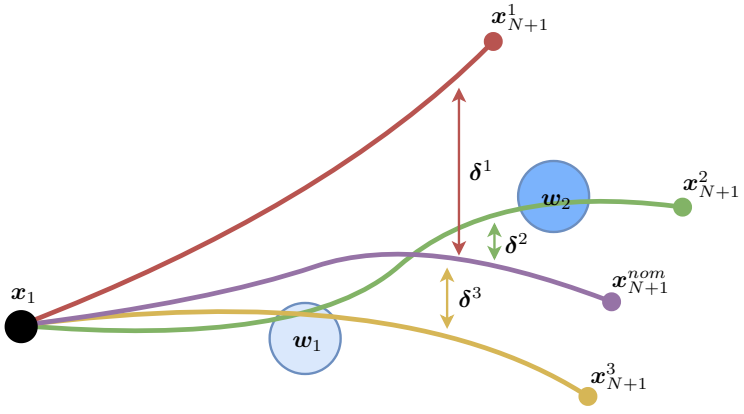


Figure 2.1: Illustration of MPPI rollout evaluation. The MPPI controller keeps nominal control (N consecutive control inputs) from the previous optimization iterations. When applied from the current state $\boldsymbol{x}_1$, the nominal rollout is computed (purple). The nominal control is then perturbed ($\boldsymbol{\delta}^1$, $\boldsymbol{\delta}^2$, $\boldsymbol{\delta}^3$), and by applying the perturbed actions we receive the perturbed rollouts (other colored curves). These rollouts are evaluated based on the cost function for the desired task. Here, the task is to fly through the waypoints $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ (in order). The rollouts are color-coded by their cost (green - best, red - worst).

Algorithm 1: Weight Calculation

Input: Rollout costs $(S_1, S_2, \ldots, S_K)$
Params: Parameter λ
Output: Weights $(\omega_1, \omega_2, \ldots, \omega_K)$

$\rho = \min\{S_1, \ldots, S_K\}$
$\eta = \sum_{k=1}^{K} \exp\left(-\frac{1}{\lambda}\left(S_k - \rho\right)\right)$
for k = 1, ..., K **do**
  $\omega_k = \frac{1}{\eta} \exp\left(-\frac{1}{\lambda}\left(S_k - \rho\right)\right)$
return $(\omega_1, \omega_2, \ldots, \omega_K)$

Algorithm 2: Model Predictive Path Integral Control

Input: Input initialization $\boldsymbol{u}_{\text{init}}$, Problem specific cost *ComputeCost*($\boldsymbol{x}^k$, $\boldsymbol{u}^k$), System dynamics $\boldsymbol{f}_{\text{RK4}}(\boldsymbol{x}_j^k, \boldsymbol{u}_j^k)$
Params: Number of rollouts K and time steps N, noise covariance Σ, time step Δt

1 **for** j = 0, ..., N-1 **do** // Initialize the control
2 $\boldsymbol{u}_j^{\text{nom}} = \boldsymbol{u}_{\text{init}}$
3 **while** *task not completed* **do**
4 $\hat{\boldsymbol{x}}$ = *CurrentStateEstimate*()
5 **for** k = 1, ..., K **do** // Simulate K rollouts
6 $\boldsymbol{x}_0^k = \hat{\boldsymbol{x}}$
7 $\boldsymbol{\delta}^k = (\delta\boldsymbol{u}_0^k, \ldots, \delta\boldsymbol{u}_{N-1}^k)$, $\delta\boldsymbol{u}_j^k \in \mathcal{N}(0, \Sigma)$
8 **for** j = 0, ..., N-1 **do** // N steps into the future
9 $\boldsymbol{u}_t^k = \boldsymbol{u}_j^{\text{nom}} + \delta\boldsymbol{u}_j^k$
10 $\boldsymbol{x}_{j+1}^k = \boldsymbol{x}_j^k + \Delta t \cdot \boldsymbol{f}_{\text{RK4}}(\boldsymbol{x}_j^k, \boldsymbol{u}_j^k)$
11 S_k = *ComputeCost*($\boldsymbol{x}^k$, $\boldsymbol{u}^k$) // and evaluate their cost
12 $(\omega_1, \omega_2, \ldots, \omega_K)$ = *ComputeWeights*$(S_1, S_2, \ldots, S_K)$ // Alg. 1
13 **for** j = 0, ..., $N-1$ **do** // Apply weighted average of
14 $\boldsymbol{u}_j^{\text{nom}} = \boldsymbol{u}_j^{\text{nom}} + \sum_{k=1}^{K} \omega_k \cdot \delta\boldsymbol{u}_j^k$ // control disturbances
15 **for** j = 0, ..., $N-2$ **do** // Shift nominal control
16 $\boldsymbol{u}_j^{\text{nom}} = \boldsymbol{u}_{j+1}^{\text{nom}}$ // one step forward
17 $\boldsymbol{u}_{N-1}^{\text{nom}}$ = *Initialize*$(\boldsymbol{u}_{N-1}^{\text{nom}})$

We will use the MPPI controller to fly a drone along a specified sequence of 3D waypoints. The mathematical model of the platform used to simulate the system evolution (line 10 in Alg. 2) is presented in the following section. The specific implementation of the other required functions, such as *ComputeCost* (line 11 in Alg. 2) and *Initialize* (line 17 in Alg. 2), will be covered in Chapter 4.

2.2 Mathematical Model

In this thesis, we want to apply the Model Predictive Path Integral control approach presented in Section 2.1.2 to fly a drone along a predefined sequence of 3D waypoints. To describe the state of the drone, we use position $\boldsymbol{p} \in \mathbb{R}^3$, unit quaternion rotation on the rotation group $\boldsymbol{q} \in \mathbb{SO}(3)$ with $\|\boldsymbol{q}\| = 1$, velocity $\boldsymbol{v} \in \mathbb{R}^3$ and body rates $\boldsymbol{\omega} \in \mathbb{R}^3$, resulting in the complete state $\boldsymbol{x} = [\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{v}, \boldsymbol{\omega}]$ with the following dynamics

$$\begin{aligned} \dot{\boldsymbol{p}} &= \boldsymbol{v} & \dot{\boldsymbol{q}} &= \frac{1}{2}\boldsymbol{q} \odot \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} \\ \dot{\boldsymbol{v}} &= \frac{1}{m}\mathbf{R}(\boldsymbol{q}) \begin{bmatrix} 0 \\ 0 \\ F_t \end{bmatrix} + \mathbf{g} & \dot{\boldsymbol{\omega}} &= \mathbf{J}^{-1}\left(\boldsymbol{\tau} - \boldsymbol{\omega} \times \mathbf{J}\boldsymbol{\omega}\right), \end{aligned} \tag{2.10}$$

where $\mathbf{R}(\boldsymbol{q})$ is the matrix representation of the quaternion $\boldsymbol{q}$, $\odot$ represents quaternion multiplication, m is the drone's mass, and $\boldsymbol{J}$ is the drone's inertia matrix. At first, we tried to control the drone on the level of single rotor thrusts $\boldsymbol{T} = [T_1, T_2, T_3, T_4]$, which can be converted to the collective thrust and body torques as

$$\begin{bmatrix} F_t \\ \boldsymbol{\tau} \end{bmatrix} = \mathbf{\Gamma}\boldsymbol{T}, \tag{2.11}$$

where $\mathbf{\Gamma}$ is the allocation matrix

$$\mathbf{\Gamma} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -l/\sqrt{2} & l/\sqrt{2} & l/\sqrt{2} & -l/\sqrt{2} \\ -l/\sqrt{2} & l/\sqrt{2} & -l/\sqrt{2} & l/\sqrt{2} \\ -c_{tf} & -c_{tf} & c_{tf} & c_{tf} \end{bmatrix} \tag{2.12}$$

with l being the drone's arm length and c_{tf} being the rotor's torque constant.

However, this resulted in a highly non-smooth motion of the drone since a disturbance to one of the single rotor thrusts changes all body rates simultaneously. Therefore, we have decided to control the body rates instead, leaving the motor speed control to the lower-level PIDs. This allows for a slower update rate (100 Hz for body rate control). However, controlling the body rates does come with some disadvantages. In addition to limiting the body rates and collective thrust, we also need to ensure that the commanded change in body rates is feasible (due to motor dynamics, we cannot expect the body rate to change precisely and arbitrarily as we command). Instead, we introduce *desired body rates* and *collective thrust*

$$\boldsymbol{u} = \begin{bmatrix} F_t \\ \omega_{xd} \\ \omega_{yd} \\ \omega_{zd} \end{bmatrix} = \begin{bmatrix} F_t \\ \boldsymbol{\omega}_d \end{bmatrix}, \tag{2.13}$$

from which we compute the needed change in body rates over the time step Δt

$$\dot{\boldsymbol{\omega}}_d = \frac{1}{\Delta t}\left(\boldsymbol{\omega}_d - \boldsymbol{\omega}\right), \tag{2.14}$$

compute the desired body torques

$$\boldsymbol{\tau}_d = \mathbf{J}\dot{\boldsymbol{\omega}}_d + \boldsymbol{\omega} \times \mathbf{J}\boldsymbol{\omega}, \tag{2.15}$$

and the desired single rotor thrusts generating the body torques $\boldsymbol{\tau}_d$ and collective thrust F_t

$$\boldsymbol{T}_d = \boldsymbol{\Gamma}^{-1} \begin{bmatrix} F_t \\ \boldsymbol{\tau}_d \end{bmatrix}. \tag{2.16}$$

We can clip these single rotor thrusts based on the real parameters of the drone motors

$$\boldsymbol{T}_{clip} = \text{clip}(T_{min}, \boldsymbol{T}_d, T_{max}), \tag{2.17}$$

and from the clipped rotor thrusts compute the clipped body torques, body rates, and collective thrust

$$\dot{\boldsymbol{\omega}}_{clip} = \mathbf{J}^{-1} \left(\boldsymbol{\tau}_{clip} - \boldsymbol{\omega} \times \mathbf{J}\boldsymbol{\omega} \right). \tag{2.18}$$

Finally, we reconstruct the feasible inputs as

$$\boldsymbol{\omega}_{\text{clip}} = \boldsymbol{\omega} + \dot{\boldsymbol{\omega}}_{\text{clip}} \cdot \Delta t \qquad \text{and} \qquad F_t = \sum_{i=1}^{4} T_{\text{clip, i}} \quad . \tag{2.19}$$

Chapter 3

Related Work

In this chapter, we will concern ourselves with work in three main areas related to the topic of this thesis. Section 3.1 addresses flying through a given sequence of waypoints with a drone in minimal time. More importantly, it discusses possible methods of reference trajectory generations suitable for this problem. Section 3.2 summarizes the works regarding the use of MPPI in real-world scenarios where disturbances are expected and need to be handled. We also discuss some of the advantages of MPPI, mainly the ability to use highly non-convex cost functions with multiple non-differentiable components. Finally, Section 3.3 contains a possible approach to advanced initialization of the MPPI controller, using the Rapidly-exploring Random Tree (RRT) algorithm to find initial estimates of the control input.

3.1 Model Predictive Contouring Control

In [9], the authors tackle the problem of flying through multiple waypoints with a drone in minimal time. A common approach is to solve the trajectory planning (path planning and time allocation) and control (tracking the trajectory) separately. Here, the authors utilize another approach, based on [23], which requires only a continuously differentiable 3D path as a reference (or a dense sequence of 3D waypoints from which the path is constructed). After the path is obtained, the Model Predictive Contouring Control (MPCC) controller minimizes a cost function reflecting the trade-off between tracking the 3D path and maximizing the progress in time along the path.

When a classical MPC approach is used for reference tracking, we need to specify the time allocation beforehand (i.e., know the desired state at each discrete time step of the optimization). In Contouring Control, the task is to minimize the 3D Euclidean distance to the reference path while minimizing the time needed to get to the path's end (or, equivalently, maximize the speed along the path). A *progress* variable θ is introduced, which is used to parametrize the desired path $\boldsymbol{p}^d : \mathbb{R} \to \mathbb{R}^3$. The controller then minimizes the distance between the position at the current time step $\boldsymbol{p}_j$ to the path while maximizing the progress variable θ.

Further, the approaches to adding the waypoint passing to the optimization task are discussed. Adding the waypoints as hard constraints (i.e., enforcing that the minimal distance achieved to each waypoint is less than a given tolerance as a constraint on the optimization variables) quickly yields the problem infeasible. On the other hand, adding the waypoints as soft constraints introduces the need for slack variables with additional constraints and costs to tune. Instead, the authors opt to dynamically change the costs of the optimization task, focusing

on minimizing the tracking error when near the waypoint and focusing on maximizing the progress otherwise. This approach is illustrated in Figure 3.1.

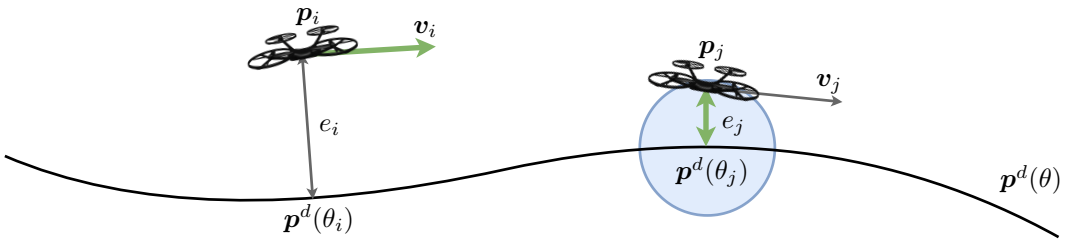


Figure 3.1: Illustration of the dynamic cost adjustment when following the continuously differentiable path $\boldsymbol{p}^d : \mathbb{R} \to \mathbb{R}^3$ as introduced in [23]. In between the waypoints (drone at $\boldsymbol{p}_i$), maximizing the velocity $\boldsymbol{v}_i$ is favored over tracking the reference precisely. When approaching the waypoint (blue circle), the controller gives higher priority to the path error e_j.

The authors further introduce three different methods to generate the nominal reference paths. The first method computes *Multi-Waypoint Minimum Snap*[1] trajectories [21]. Fourth-order polynomial trajectories are convenient to use with quadrotors since the continuous differentiability combined with the differential flatness property of quadrotors can be used to derive full state information at each point. A disadvantage of polynomial trajectories is that due to their inherent smoothness, they reach their maximal value at a single point and, therefore, cannot exploit the agility and aggressiveness of the drone to its potential. The authors use this approach to generate a continuously differentiable 3D path passing through the waypoints in a receding horizon fashion. This trajectory is then tracked by the MPCC controller (with the time allocation performed online).

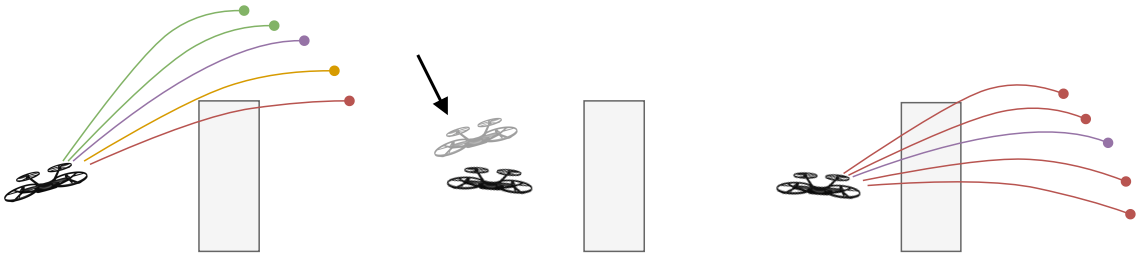
Another approach is the Time-Optimal Full Model. This method makes use of the full non-linear model of the quadrotor and formulates the time-optimal control problem as a non-linear optimization task. Compared to the polynomial trajectories, this approach allows to utilize the full rotor potential throughout the trajectory. This method is used in this thesis and is therefore discussed in detail in Section 4.1. The authors argue that the high generation time (up to hours of computation time per trajectory) is not worth spending since only a 3D path is sufficient (compared to full input and state information).

The last approach computes a time-optimal trajectory for a simpler, point-mass model with constrained acceleration and velocity. It can be shown that the time-optimal point-to-point control policy for fixed start and end position and velocity exhibits a bang-singular-bang behavior with a closed-form solution for the switching times [28]. To find the required multi-waypoint trajectory, a given number of velocities is sampled at each waypoint, the minimal-time policy is computed per each pair of velocities at successive waypoints, and the shortest path is found on the generated graph (e.g., using the Dijkstra's algorithm). Similarly to the minimum snap trajectory approach, the path is planned in a receding horizon fashion (planned only for a given number of future waypoints).

[1] **snap** — fourth time derivative of position

3.2 MPPI Extensions

In most standard settings, the iterative predictive control algorithms (MPC and MPPI) make use of a warm start - initialization of the solution based on previous optimization. This works around the implicit assumption that the actual next state of the system is close to the predicted (nominal) next state. However, in real-world settings with the presence of disturbances, this assumption may fail, as shown in Figure 3.2. In [16] (based on a nonlinear variant of the Tube-MPC algorithm [24]), authors try to remove this assumption and improve the robustness of the controller by running two instances of an MPPI controller (named *nominal*) in parallel. One instance uses the actual system state as the initial state, the other uses the nominal (predicted) state. If the nominal controller obtains a better solution using the actual state compared to the nominal state, the better solution is used and the nominal state is set to the actual system state. The nominal states are then sent to a controller designed for disturbance rejection (in the paper, the authors use an iterative Linear Quadratic Gaussian (iLQG) controller).



(a) (b) (c)

Figure 3.2: In Fig. 3.2a, the MPPI controller computes the nominal input and applies the first input to the system. However, due to a disturbace (Fig. 3.2b), the system evolution is vastly different to the predicted evolution (purple curve). If the nominal input is used to initialize the next iteration of MPPI (Fig. 3.2c), the system may become unable to find a good solution to reject the disturbance and fail.

The authors continue to demonstrate their approach on multiple problem instances. One is the problem of simulated helicopter landing, where the authors are able to land a helicopter by designing a highly non-convex cost function with multiple non-differentiable components (such as the maximum of a set), showing the abilities of MPPI to use such cost functions.

Methods in [2] and [5] further deal with systems where the state evolution is stochastic (due to modeled disturbances and model inaccuracies) compared to the deterministic model setting. Namely, they deal with a linear, time-variant stochastic system whose dynamics can be modelled as

$$\boldsymbol{x}_{j+1} = A_j \boldsymbol{x}_j + B_j \boldsymbol{u}_j + \boldsymbol{w}_j, \tag{3.1}$$

where the disturbance $\boldsymbol{w}_j$ is assumed to be zero-mean Gaussian (i.e., drawn from $\boldsymbol{w}_j \sim \mathcal{N}(\mathbf{0}, \Sigma_j)$) and $\mathbb{E}[\boldsymbol{w}_i \boldsymbol{w}_j]^T = \mathbf{0}$ for all $i \neq j$. The proposed method consists of three modules: MPPI controller, half-space generator, and Constrained Covariance Steering module. The MPPI controller provides a reference trajectory. Then, constraining half-spaces of admissible state values are computed, avoiding the obstacle regions. Finally, the CCS module is used to find a control policy minimizing the error from the reference trajectory while satisfying the safety constraints.

3.3 Guided MPPI methods

An important part of the controller's performance is a proper initialization of the nominal control $\boldsymbol{u}^{\text{nom}}$ (lines 2 and 17 in Alg. 2). Having a reasonable estimate of the control that results in a low-cost nominal trajectory is vital to the proper function of the controller. Conversely, when no initialization method is present (such as initializing $\boldsymbol{u}_N^{\text{nom}}$ to zero at each iteration), the controller may have problems finding a good value by only sampling disturbances from the nominal control. Researchers proposed multiple methods to address this issue. In [1], the researchers run Rapidly-exploring Random Tree (RRT) algorithm [25] to find the path. RRT is a sampling-based motion planning algorithm that builds a tree rooted at a given start state by expanding toward samples taken from the configuration space. The algorithm utilizes a local planner to connect the configurations in the free space and explores the configuration space until a specified subset of the free configuration space is reached. The solution found by the RRT algorithm is then used as a mean of the MPPI controller. However, due to computation complexity, the authors run the RRT algorithm offline prior to the control task, which makes it impossible to use onboard in an unknown environment.

3.4 Summary

The methods addressing the Contouring Control problem, such as MPCC, introduce an interesting idea of jointly optimizing the distance from a reference path and progress along it. This may prove advantageous in the setting of minimum-time flight with the use of precomputed paths. However, as we have discussed in Section 3.1, using non-linear solvers to solve the formulated problem requires an approximation of the distance to the reference path, since finding the nearest point on the path is itself an optimization task. This approximation could be alleviated by using MPPI, since the closest point on a given path could be computed with arbitrary precision (or even exactly if the path is given as a sequence of discrete points). Even though not used in this thesis, this idea which will be addressed in our future research.

The presented work on MPPI mainly discusses disturbance rejection methods. In this thesis, we will refrain from using stochastic systems and will use deterministic system dynamics (Section 2.2 covers the exact model used). Other works deal with better initialization methods, but they often rely on knowing the environment in advance, allowing environment-dependent time-heavy computations to be carried out off-line before deploying the robot (as discussed in Section 3.3).

Most of the presented approaches do not reasonably discuss the available computation power or work with systems where the weight of the full-scale GPUs and their power demand is not a problem. However, in our scenario, where the maximal weight is limited and using a high-end, full-scale gaming GPU is not possible, we need to keep those constraints in mind. This will be discussed in Section 5.4.

Chapter 4

Methodology

In this chapter, we will introduce our method of controlling a drone using the MPPI control methodology. In Section 4.1, we will cover the specification of the optimal control problem for our case, using approaches already presented by other authors. Solving the problem will give us a set of short time-optimal trajectories (motion primitives). Section 4.2 addresses the design of a database that stores the motion primitives and allows their retrieval in real-time. Finally, in Section 4.3, we describe the implementation of the MPPI controller, along with a discussion of the cost function and the tunable parameters.

4.1 Time-optimal Trajectory Computation

We use a method presented in [7] to compute the time-optimal trajectories. In the paper, single rotor thrusts are used as the control input $\boldsymbol{u} = [T_1, T_2, T_3, T_4]$. This allows to exploit the full potential of the platform (saturating the single rotor thrusts) instead of limiting the body rates. The problem of finding the time-optimal trajectory passing through a predefined sequence of M waypoints $\boldsymbol{\mathcal{W}} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_M)$ with tolerance d_{tol} is formulated as a non-linear discrete optimization problem. Progress variables $\boldsymbol{\lambda} \in \mathbb{R}^{M \times N}$ are introduced to measure the progress along a track, where λ_j^m defines the progress at time t_j towards completing the m-th waypoint. At the start of the track, all progress variables start at 1 ($\boldsymbol{\lambda}_0 = 1$) and have to reach 0 at the end ($\boldsymbol{\lambda}_N = 0$). Each sequence $\boldsymbol{\lambda}^m$ has to be non-increasing, decreasing at time step j only if the states $\boldsymbol{x}_j$ are near the m-th waypoint $\boldsymbol{w}_m$. A progress change μ and tolerance slack ν are introduced to measure and encode the proximity to the waypoints — the meaning of the optimization variables is visualized in Figure 4.1 and the full problem formulation is stated in Eq. 4.1.

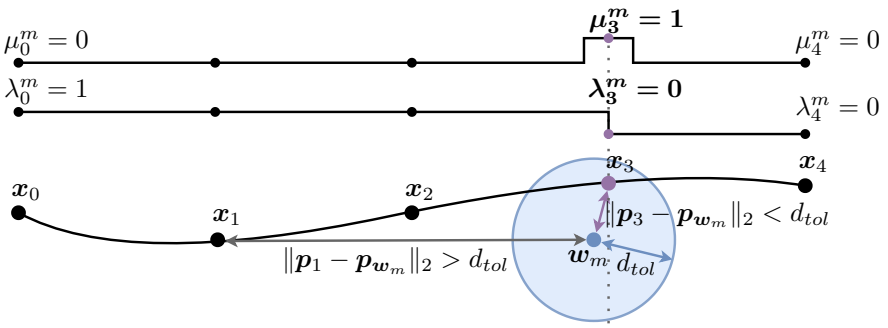


Figure 4.1: Illustration of time-optimal trajectory computation from [7]. The progress variable λ^m changes by a non-zero change μ^m only when the drone is in the neighborhood of waypoint $\boldsymbol{w}_m$.

$$
\begin{aligned}
&\textbf{optimize} && (\boldsymbol{x}^*, \boldsymbol{u}^*, \boldsymbol{\lambda}^*, \boldsymbol{\mu}^*, \boldsymbol{\nu}^*) = \underset{(\boldsymbol{x},\boldsymbol{u},\boldsymbol{\lambda},\boldsymbol{\mu},\boldsymbol{\nu})}{\operatorname{argmin}} \ \Delta t \\
&\textbf{subject to} \\
&\textit{system dynamics} && \begin{cases} \boldsymbol{x}_0 = \boldsymbol{x}_{init} & \\ \boldsymbol{x}_{j+1} - \boldsymbol{x}_j - \Delta t \cdot \boldsymbol{f}_{RK4}(\boldsymbol{x}_j, \boldsymbol{u}_j) = 0 & 0 \le j \le N \end{cases} \\
&\textit{input constraints} && \begin{cases} \boldsymbol{u}_{min} - \boldsymbol{u}_j \le 0 & 0 \le j \le N \\ \boldsymbol{u}_j - \boldsymbol{u}_{max} \le 0 & 0 \le j \le N \end{cases} \\
&\textit{progress evolution, boundary, and sequence constraints} && \begin{cases} \boldsymbol{\lambda}_{j+1} - \boldsymbol{\lambda}_j + \boldsymbol{\mu}_j = \boldsymbol{0} & 0 \le j \le N-1 \\ \boldsymbol{\lambda}_0 - 1 = \boldsymbol{0} & \\ \boldsymbol{\lambda}_N = \boldsymbol{0} & \\ \boldsymbol{\mu}_j \ge 0 \quad \lambda_j^m - \lambda_j^{m+1} \le 0 & 0 \le j \le N \ , \ 1 \le m \le M \end{cases} \\
&\textit{progress constraint with tolerance} && \begin{cases} \mu_j^m \cdot (\|\boldsymbol{p}_j - \boldsymbol{p}_{\boldsymbol{w}_j}\|_2^2 - \nu_j^m) = 0 & 0 \le j \le N \ , \ 1 \le m \le M \\ -\nu_j^m \le 0 & 0 \le j \le N \ , \ 1 \le m \le M \\ \nu_j^m - d_{tol}^2 \le 0 & 0 \le j \le N \ , \ 1 \le m \le M \end{cases}
\end{aligned}
\tag{4.1}
$$

Even in this formulation, some approximations needed to be introduced. The trajectory is discretized and divided into a fixed number of nodes between each pair of waypoints. Another approximation can come from the model mismatch (even more precise methods exist, such as blade element momentum theory for modeling the aerodynamics of the propellers [18]).

The problem is then solved using a numeric optimization framework, such as CasADi [13]. After the optimization is performed, the solution contains the inputs (single rotor thrusts) and full dynamic states (position, velocity, rotation, and angular velocity) of the time-optimal trajectory at each time step.

In order for our method to be usable during a real flight where many different situations can be encountered, we need to generate various training scenarios for which the motion primitives will be computed. We do so by randomly generating tracks consisting of a start and goal configuration and three waypoints to be flown through. From the start configuration, we generate a random 3D waypoint in a spherical shell centered at the start configuration with radii $r_{\min}$ and $r_{\max}$. To ensure a uniform distribution of directions, we generate a 3D vector consisting of independent samples from three standard normal distributions and normalize it, which results in a uniform distribution of points on a unit sphere. This vector is then multiplied by a length sampled uniformly between $r_{\min}$ and $r_{\max}$ (visualized in Figure 4.2). This procedure is repeated until a specified number of waypoints is generated, with the spheres centered at the last generated point.

After the tracks are created, we compute the time-optimal trajectories for them using the method presented above. Three of the tracks and computed time-optimal trajectories are visualized in Figure 4.3.

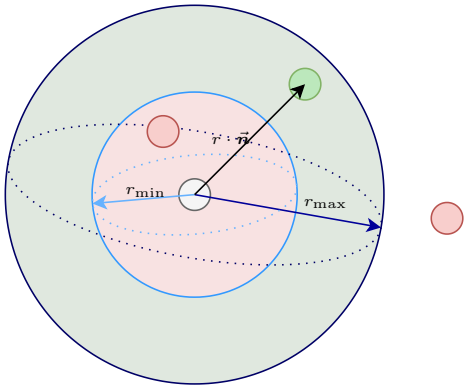
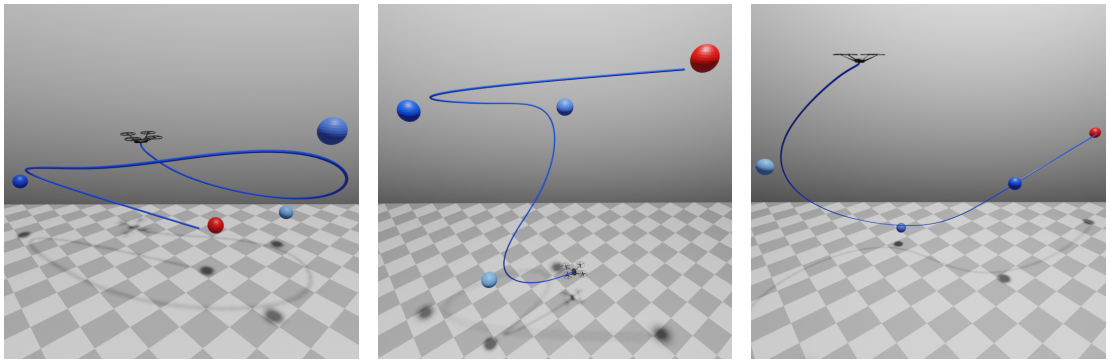


Figure 4.2: Visualization of the training tracks generation. From the current waypoint (grey), the next waypoint (green) is created in a random direction $\vec{n}$ generated uniformly on a unit sphere at a random uniform distance between $r_{\min}$ and $r_{\max}$.



(a) (b) (c)

Figure 4.3: Time-optimal trajectories for three tracks. The drone has to fly through the three blue waypoints (light to dark blue, in order) and end at the final (red) waypoint. The formulation of the optimization task admits a tolerance and does not require flying exactly through the waypoints.

Algorithm 3: Time-optimal trajectory computation

Input: Number of scenarios to generate N
Params: Waypoint distance limits $r_{\min}$ and $r_{\max}$
Output: N time-optimal motion primitives $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_N)$

for $i = 1, \ldots, N$ **do**
  $(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3) = \mathit{GenerateWaypoints}()$ // Figure 4.2
  $\boldsymbol{\pi} = \mathit{ComputeOptimalTrajectory}(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3)$ // Solution to Eq. 4.1
  $j = \mathit{ClosestState}(\pi, \boldsymbol{w}_1)$ // Index of state closest to $\boldsymbol{w}_1$
  $\boldsymbol{\pi}_i = \mathit{MoveToOrigin}(\boldsymbol{\pi}[j : \text{end}])$ // Ensure $\boldsymbol{\pi}$ starts at $(x, y, z) = (0, 0, 0)$
return $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_N)$

It is important to note that we do not expect that the waypoints used for querying during the realtime flight will be distributed by the same laws as the waypoint used to generate the motion primitives. The querying system is designed in such a way that a motion primitive passing through the query waypoints is returned regardless of the waypoints that generated the motion primitive. This is covered in detail in the following section.

4.2 Database of Motion Primitives

In this section, we propose a database-based method for storing and online retrieval of a precomputed motion primitives. A motion primitive is a short trajectory computed by the time-optimal planner described in Section 4.1. Since the initital state of the drone is at zero velocity, we save the trajectory from the first waypoint onwards.

4.2.1 Trajectory storing

Since we have all the data available prior to the flight and can determine the range of the data, we can use binning metric space-partitioning methods, that is, splitting the space into bins and saving each data element to a corresponding bin. This approach allows for retrieval of the relevant bins in a constant time, compared to a logarithmic time complexity of data structures such as k-d trees or R-trees. Moreover, balancing the trees is needed to ensure the logarithmic complexity of the search, which brings an additional overhead [26].

To design the database, we first need to analyze the symmetries of the system. First, we assume translation invariancy of the gravitational field[1] and can, therefore, freely translate the trajectories through $\mathbb{R}^3$. Second, we assume rotation invariancy around the world z-axis (i.e., if a sequence of inputs is applied to a drone rotated around the world z-axis, the trajectory will be the same as if we applied the input to a drone that was not rotated and rotated the resulting trajectory by the same amount instead). This results in bins being hollow cylinders with a given width α_{xy} and height α_z. To build the database index, we move all trajectories so that they start at the origin of the global coordinate frame (done in Alg. 3) and bin the trajectory states accordingly. The bins and binned trajectories are visualized in Figure 4.4, and the algorithm for creating the index is outlined in Alg. 4.

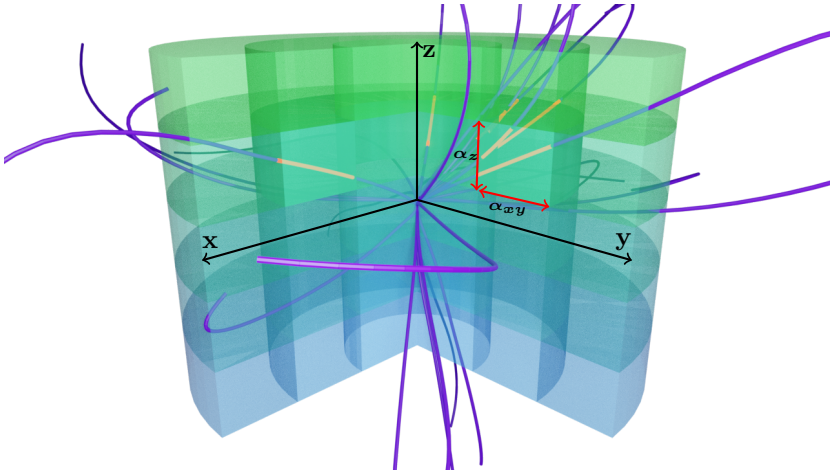


Figure 4.4: Visualization of the database structure. Each bin is a hollow cylinder with height α_z and width α_{xy}. States of trajectory inserted into the database (purple curves) are assigned to respective bins by their 3D coordinates. Parts of trajectories assigned to the highlighted bin are shown in red.

[1] While not entirely true, the variance of the gravitational acceleration is small enough to be omitted.

Algorithm 4: DB - Build Index

Input: N time-optimal motion primitives $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_N)$
Params: Partitioning resolutions α_{xy} and α_z
Output: Database index $\mathcal{D}$

$min_z, max_z, max_{xy} = ComputeDataRange(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_N)$ // min_{xy} is 0
$bins_{xy} = \lceil \frac{max_{xy}}{\alpha_{xy}} \rceil$
$bins_z = \lceil \frac{max_z - min_z}{\alpha_z} \rceil$
$\mathcal{D} = Init((\boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_N), bins_{xy}, bins_z)$
for $i = 1, \ldots, N$ **do** // For each trajectory
  for $j = 1, \ldots, |\boldsymbol{\pi}_i|$ **do** // and each state
    $(x, y, z) = \boldsymbol{\pi}_i[j]$
    $d_{xy} = \sqrt{x^2 + y^2}$
    $bin_{xy} = \lfloor \frac{d_{xy}}{\alpha_{xy}} \rfloor$ // bin by distance from origin
    $bin_z = \lfloor \frac{z - min_z}{\alpha_z} \rfloor$ // and height
    $AddToBin(\mathcal{D}, bin_{xy}, bin_z, (i, j, d_{xy}, z))$
return $\mathcal{D}$

4.2.2 Resampling the generated trajectories

The time step of the time-optimal trajectory (Section 4.1) cannot be defined prior to the optimization and can vary between resulting trajectories. Therefore, we use linear interpolation to resample all trajectories to the same time step when inserting them into the database[2]. The interpolated states are shown in Figure 4.5.

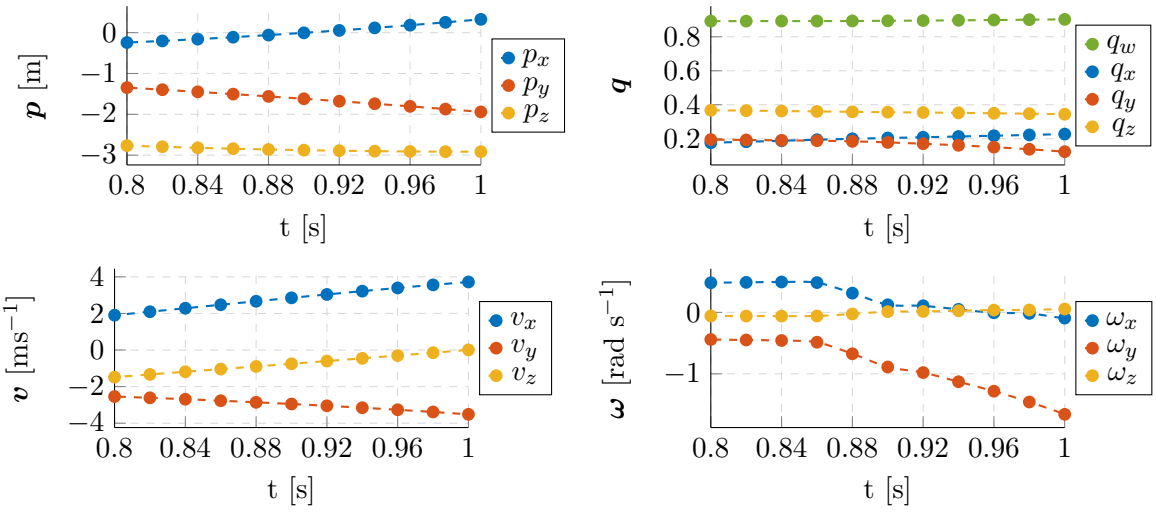


Figure 4.5: The time-optimal trajectory is linearly interpolated (dashed lines) and the resampled points are generated on the interpolation with a new time step (points, here $\Delta t = 0.02\,\mathrm{s}$).

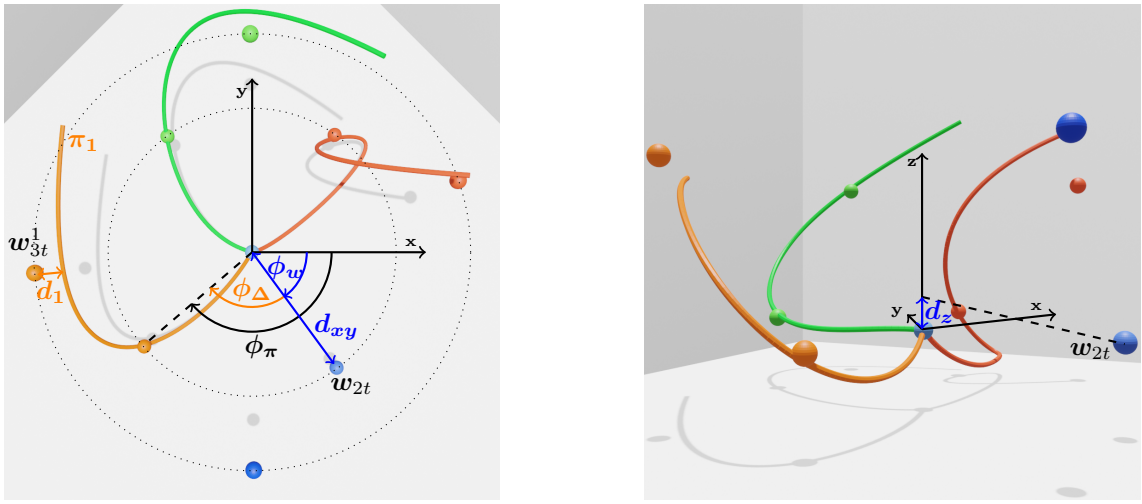
[2]Even though linear interpolation is not precisely correct for quaternions, we assume that the time-optimal trajectory is dense enough that the changes between two consecutive states are small. Otherwise spherical interpolation (slerp) should be used.

4.2.3 Trajectory retrieval

The next three waypoints $(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3)$ will be used to query the database $\mathcal{D}$ for a relevant motion primitive $\boldsymbol{\pi}$. We motivate this decision as follows — we expect that the controller already has a plan to reach the first waypoint $\boldsymbol{w}_1$ obtained by the previous optimization runs. Therefore, our main goal is to find a trajectory connecting the first waypoint $\boldsymbol{w}_1$ with the second waypoint $\boldsymbol{w}_2$. However, selecting an arbitrary trajectory between the waypoints $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ regardless of the position of the future waypoint $\boldsymbol{w}_3$ could result in highly non-smooth and aggressive maneuvers. By including the third waypoint $\boldsymbol{w}_3$ in the query, we can leverage the power of predictive control and prepare to steer the drone towards the waypoint $\boldsymbol{w}_3$ earlier, making the flight smoother and faster. More than three waypoints could be considered indeed. However, that would make the database sparser (we would need to generate a lot more motion primitives to be able to handle most of the possible combinations) and may not result in substantial performance improvements. An example of this can be found in [9].

To find a suitable motion primitive $\boldsymbol{\pi}$, we compute the position of the second waypoint $\boldsymbol{w}_2$ relative to the first waypoint $\boldsymbol{w}_1$. Then, we compute the bin in database $\mathcal{D}$ to which the transformed point $\boldsymbol{w}_{2t}$ belongs and retrieve the motion primitives that the bin contains. Since the bin can contain multiple states belonging to the same motion primitive, we select the state which is closest to the transformed $\boldsymbol{w}_{2t}$. This procedure is outlined in Alg. 5.

For each retrieved motion primitive, we find a rotation around the z-axis which aligns the motion primitive with the second waypoint $\boldsymbol{w}_{2t}$. From the rotated motion primitives, we select the one closest to the third waypoint $\boldsymbol{w}_3$[3]. The querying process is visualized in Figure 4.6 and the algorithm is outlined in Alg. 6.



(a) Top view (b) Side view

Figure 4.6: For a query $(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3)$ (blue spheres), three trajectories are found in the database (colored curves). Each curve passes through a bin containing $\boldsymbol{w}_2$, characterized by the norm of the xy-plane projection d_{xy} and height d_z. The transformed waypoints are rotated by the angle ϕ_Δ, aligning the transformed waypoint $\boldsymbol{w}_{2t}$ onto the trajectory $\boldsymbol{\pi}_1$. Then, we compute the distance of the transformed waypoint $\boldsymbol{w}_{3t}$ to the primitive $\boldsymbol{\pi}_1$ (d_1). The trajectory resulting in the smallest distance is returned.

[3]In the implementation, we transform the third waypoint instead of the whole motion primitive for efficiency.

Algorithm 5: Get Unique Trajectories

Input: Database index $\mathcal{D}$, Bins bin_{xy}, bin_z, Waypoint $\boldsymbol{w}$
Params: Partitioning resolutions α_{xy} and α_z
Output: List of motion primitives and states $[(\pi_1, s_1), \dots]$

$d_{xy} = \|\boldsymbol{w}_{2t}\|$
$z = \boldsymbol{w}_{2t,z}$
$result = [\,]$
for $(i, j, d'_{xy}, z') \in GetBin(\mathcal{D}, bin_{xy}, bin_z)$ **do** // State $\pi_i[j]$ with distances d'_{xy} and z'
  $\pi_i = GetTrajectory(\mathcal{D}, i)$
  if trajectory i in $result$ **and** $\pi_i[j]$ is better **then**
    $Update(result, (\pi_i, j))$
  else
    $Add(result, (\pi_i, j))$
return $result$

Algorithm 6: DB - Query

Input: Database index $\mathcal{D}$, Waypoint triplet $(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3)$
Params: Partitioning resolutions α_{xy} and α_z
Output: Motion primitive π

$\boldsymbol{w}_{2t} = \boldsymbol{w}_2 - \boldsymbol{w}_1$ // Transform everything relative to $\boldsymbol{w}_1$
$d_{xy} = \|\boldsymbol{w}_{2t}\|$
$bin_{xy} = \lfloor \frac{d_{xy}}{\alpha_{xy}} \rfloor$
$bin_z = \lfloor \frac{\boldsymbol{w}_{2t,z} - z_{min}}{\alpha_z} \rfloor$
$U = GetUniqueTrajectories(\mathcal{D}, bin_{xy}, bin_z, \boldsymbol{w}_{2t})$ // Alg. 5
$\phi_{\boldsymbol{w}} = \mathrm{atan2}(\boldsymbol{w}_{2t,y}, \boldsymbol{w}_{2t,x})$
$d^* = \infty$
for $(\pi_i, j_2) \in U$ **do** // $\boldsymbol{s} = \pi_i[j_2]$ is a state on π_i from bin
  $\boldsymbol{s} = \pi_i[j_2]$ // (bin_{xy}, bin_z) closest in $\mathbb{R}^3$ to $\boldsymbol{w}_{2t}$
  $\phi_\pi = \mathrm{atan2}(\boldsymbol{s}_y, \boldsymbol{s}_x)$
  $\phi_\Delta = \phi_\pi - \phi_{\boldsymbol{w}}$
  $\boldsymbol{w}_{3t} = R_{\phi_\Delta}(\boldsymbol{w}_3 - \boldsymbol{w}_1)$
  $(j_3, d_i) = ClosestState(\pi_i, \boldsymbol{w}_{3t})$ // Index and distance of the closest state
  if $j_3 \leq j_2$ **then** // Discard trajectories where the third waypoint
    continue // is reached before the second waypoint
  if $d_i < d^*$ **then** // No valid trajectory found
    $d^* = d_i$
    $\phi^*_\Delta = \phi_\Delta$
    $\pi^* = \pi_i$
if $d^* = \infty$ **then**
  return $LinearInterpolation(\boldsymbol{w}_1, \boldsymbol{w}_2, \boldsymbol{w}_3)$
return $Transform(\pi^*, R_{\phi^*_\Delta}, \boldsymbol{w}_1)$ // Rotate π^* by ϕ^*_Δ and move to $\boldsymbol{w}_1$

4.3 Model Predictive Path Integral Control

As described in Section 2.2, the MPPI controller will control the drone on the level of collective thrust and body rates $\boldsymbol{u} = (F_t, \boldsymbol{\omega}_d)$. We expect the existence of a higher-level mission planner, which supplies our controller with the next waypoints $\boldsymbol{\mathcal{W}}$ (and optionally with a representation of the obstacle region $\mathcal{O}$, which will be addressed in Section 5.2.4). The controller then computes the desired control and sends it to lower-level controllers, which try to track this control. After applying the control and processing the available measurements $\boldsymbol{y}_{\text{meas}}$, a state estimator will provide our controller with an estimate of the current state $\hat{\boldsymbol{x}}$. This process is visualized in Figure 4.7.

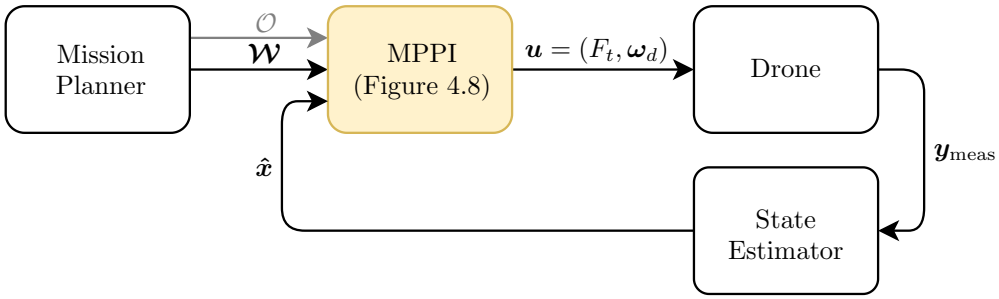


Figure 4.7: Illustration of the main control loop. A high-level mission planner specifies the waypoints $\boldsymbol{\mathcal{W}}$ to fly through and (optionally) the obstacle region $\mathcal{O}$ to avoid. The MPPI controller takes those commands along with an estimate of the current state $\hat{\boldsymbol{x}}$ and computes the next action input $\boldsymbol{u} = (F_t, \boldsymbol{\omega}_d)$ to be applied by the system. The state $\hat{\boldsymbol{x}}$ is estimated by a state estimator from measurements $\boldsymbol{y}_{meas}$.

Designing and implementing the state estimators and a realistic simulator of a drone is out of the scope of this thesis. Therefore, for initial testing, we will assume that the real state is known and the drone is able to apply the commands precisely, resulting in an ideal simulation that makes the system evolution identical to the mathematical model described in Section 2.2. Section 5.2 covers the verification of our proposed method under these ideal conditions. However, if applied on a real platform, the controller will not have an exact knowledge of the dynamical state and the system will not evolve precisely according to the mathematical model. To test our method under more realistic conditions, we will implement the proposed methods in yet another, more advanced simulator — this is covered in Section 5.3.

At each iteration, the controller samples disturbances from the nominal control sequence and computes the predicted trajectory by a forward simulation of the system dynamics (as described in Section 2.2). After the costs of the rollouts are evaluated (Eq. 2.6) and transformed into weights (Eq. 2.7), the nominal control is updated, and the first control of the control sequence is applied to the system. A single iteration is illustrated in Figure 4.8.

The dynamics integration and rollout evaluation are independent over the rollouts, which allows a parallel approach (Figure 4.9). By implementing the algorithms using CUDA architecture, we are able to leverage a GPU to achieve the required computation speeds.

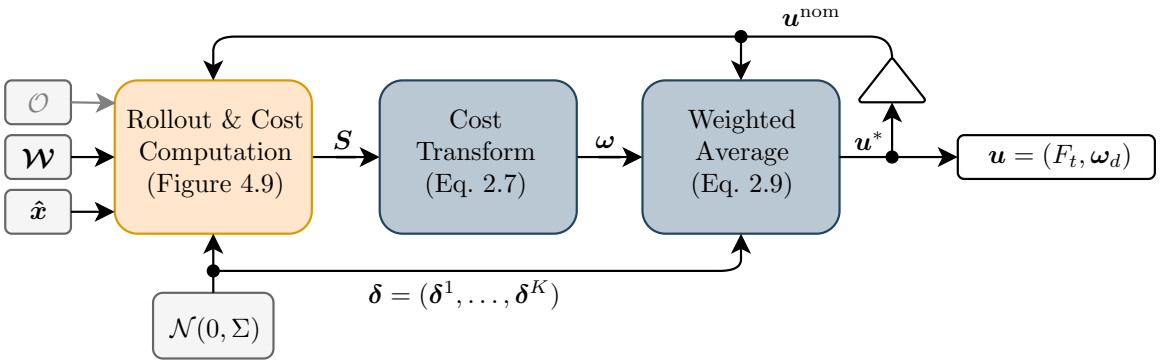


Figure 4.8: Illustration of one MPPI iteration. As input, the controller receives the waypoint sequence $\boldsymbol{\mathcal{W}}$, the current state estimate $\hat{\boldsymbol{x}}$, and (optionally) the obstacles $\mathcal{O}$. These inputs, together with the nominal control sequence $\boldsymbol{u}^{\text{nom}}$ and the control noise $\boldsymbol{\delta}$ are sent to the GPU, where the rollouts and costs $\boldsymbol{S} = (S_1, \ldots, S_K)$ are computed. The costs are then transformed to weights $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_K)$ and used to update the control ($\boldsymbol{u}^*$). The first control of the control sequence is applied to the system, and the whole control sequence is used to initialize the next iteration.

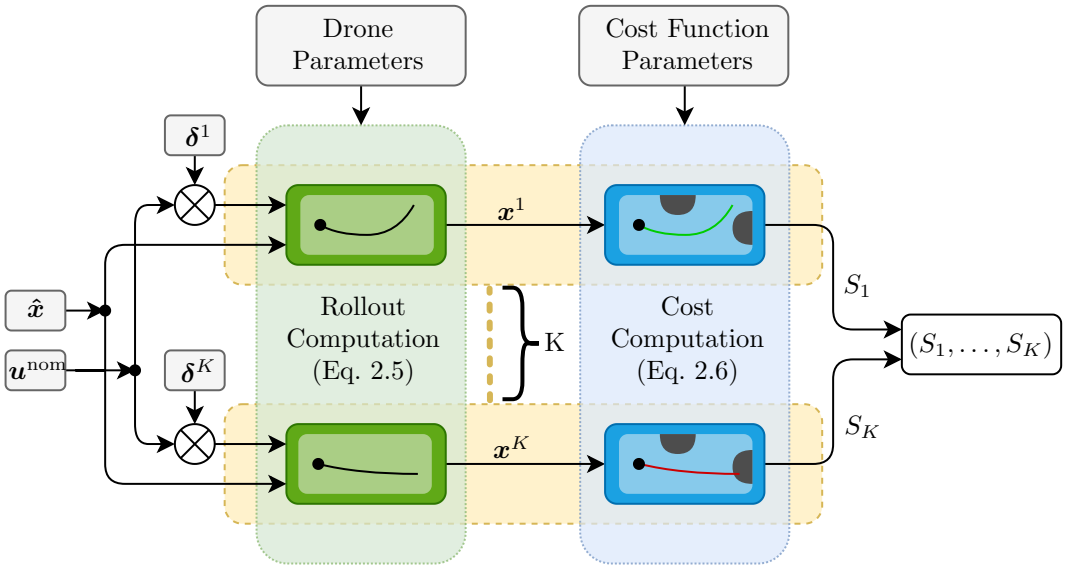


Figure 4.9: Illustration of MPPI rollout and cost computation on GPU. From the initial state $\hat{\boldsymbol{x}}$, nominal control $\boldsymbol{u}_{nom} = (\boldsymbol{u}_0^{\text{nom}}, \ldots, \boldsymbol{u}_{N-1}^{\text{nom}})$ disturbed by $\boldsymbol{\delta}^k = (\boldsymbol{\delta}_0^k, \ldots, \boldsymbol{\delta}_{N-1}^k)$ is applied (green) in parallel over K CUDA cores (yellow). Resulting rollouts $(\boldsymbol{x}^1, \ldots, \boldsymbol{x}^K)$ are evaluated by the cost function (blue), and costs $(S_1, \ldots, S_K)$ are returned from the GPU.

To design the cost function which assigns a real value to each rollout $\boldsymbol{x}^k$ with inputs $\boldsymbol{u}^k$, we start with the standard weighing of input and input change

$$S_k = \sum_{j=0}^{N} \|\boldsymbol{u}_j^k\|_R^2 + \sum_{j=0}^{N-1} \|\Delta \boldsymbol{u}_j^k\|_{R_\Delta}^2, \tag{4.2}$$

where R and R_Δ are positive semidefinite cost matrices, $\Delta \boldsymbol{u}_j^k = \boldsymbol{u}_{j+1}^k - \boldsymbol{u}_j^k$ denotes the change of input, and $\|\cdot\|_P^2$ denotes the weighted Euclidean inner product $\|\boldsymbol{u}\|_P^2 = \boldsymbol{u}^T P \boldsymbol{u}$. The states $\boldsymbol{x}_j^k$ are not used for weighing, since we only expect to penalize high angular velocities, which can be done by weighing the desired body rates in $\boldsymbol{u}_j^k$.

To enforce tracking the reference, we introduce a term in the form

$$\sum_{j=0}^{N} \rho_{\text{ref}}(\boldsymbol{x}_j^k, \boldsymbol{x}_j^{\text{ref}}), \tag{4.3}$$

where ρ_{ref} is a metric (or at least an approximation thereof) on $\mathbb{R}^3 \times \mathbb{SO}(3) \times \mathbb{R}^3 \times \mathbb{R}^3$

$$\rho_{\text{ref}}(\boldsymbol{x}_j^k, \boldsymbol{x}_j^{\text{ref}}) = \|\boldsymbol{p}_j^k - \boldsymbol{p}_j^{\text{ref}}\|^2_{c_{\boldsymbol{p}}^{\text{ref}}} + c_{\boldsymbol{q}}^{\text{ref}} \cdot d_{\boldsymbol{q}}(\boldsymbol{q}_j^k, \boldsymbol{q}_j^{\text{ref}})^2 + \|\boldsymbol{v}_j^k - \boldsymbol{v}_j^{\text{ref}}\|^2_{c_{\boldsymbol{v}}^{\text{ref}}} + \|\boldsymbol{\omega}_j^k - \boldsymbol{\omega}_j^{\text{ref}}\|^2_{c_{\boldsymbol{\omega}}^{\text{ref}}}, \tag{4.4}$$

with the weighted Euclidean metric used for $\mathbb{R}^3$ and weighing coefficients $c_{\boldsymbol{p}}^{\text{ref}}$, $c_{\boldsymbol{q}}^{\text{ref}}$, $c_{\boldsymbol{v}}^{\text{ref}}$ and $c_{\boldsymbol{\omega}}^{\text{ref}} \in \mathbb{R}$. The Euclidean norm is not a proper metric on $\mathbb{SO}(3)$ (quaternion $\boldsymbol{q}$ representing the rotation) — this can be most prominently seen for quaternions $\boldsymbol{q}$ and $-\boldsymbol{q}$, which represent the same rotation in $\mathbb{R}^3$, but the Euclidean metric will yield a non-zero result. Therefore, we need to address this inconvenience when adding the reference tracking part to be able to handle distances between quaternions correctly. There exist two commonly used options for the function $d_{\boldsymbol{q}} : \mathbb{SO}(3) \times \mathbb{SO}(3) \to \mathbb{R}$. The first one is computing the angle of rotation required to get from one orientation to the other, which is given by

$$\theta = \cos^{-1}(2\langle \boldsymbol{q}_1, \boldsymbol{q}_2 \rangle^2 - 1), \tag{4.5}$$

where $\langle \cdot, \cdot \rangle : \mathbb{SO}(3) \times \mathbb{SO}(3) \to \mathbb{R}$ denotes the quaternion inner product

$$\langle \boldsymbol{q}_1, \boldsymbol{q}_2 \rangle = w_1 w_2 + x_1 x_2 + y_1 y_2 + z_1 z_2. \tag{4.6}$$

However, the evaluation of this function is computationally demanding, and we will use a common alternative, which roughly corresponds the exact angle (upto a multiplicative constant)

$$d_{\boldsymbol{q}}(\boldsymbol{q}_1, \boldsymbol{q}_2) = 1 - \langle \boldsymbol{q}_1, \boldsymbol{q}_2 \rangle^2. \tag{4.7}$$

The advantage of both the exact angle and its approximation over the Euclidean metric is that it better respects the behavior of quaternions, mainly $d_{\boldsymbol{q}}(\boldsymbol{q}, \boldsymbol{q}) = d_{\boldsymbol{q}}(\boldsymbol{q}, -\boldsymbol{q}) = 0$. The comparison can be seen in Figure 4.10.

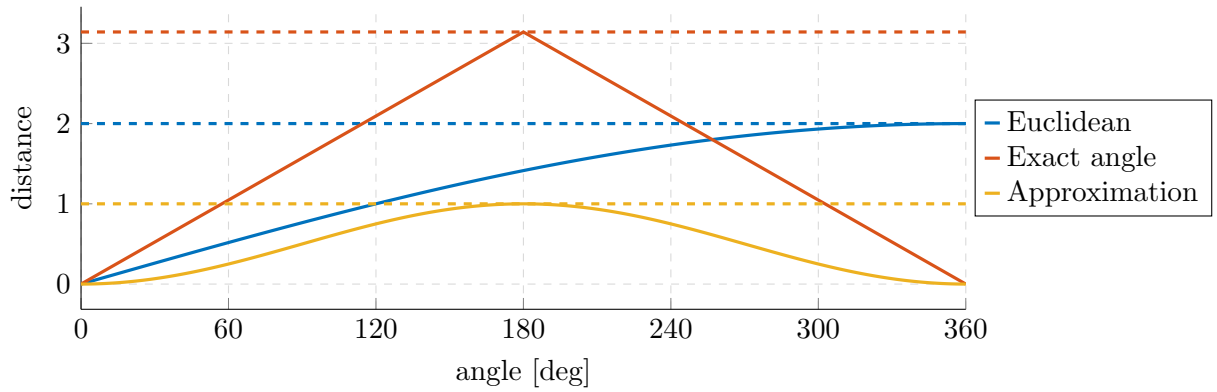


Figure 4.10: Comparison of three quaternion distance methods — Euclidean, exact angle (Eq. 4.5), and approximation thereof (Eq. 4.7). An object makes a full 360-degree rotation around a single axis, and we compute the distance to a unit quaternion $(1, 0, 0, 0)$. We see that the Euclidean metric yields a value of 2 when the object rotates fully (and returns to its initial position). In comparison, the angle and its approximation reach maximum (π and 1, respectively) at 180° and report a zero distance after the full rotation is completed.

Furthermore, we can make the controller favor trajectories passing in the neighborhood of the waypoints by adding

$$-c_{\boldsymbol{w}} \cdot \mathbb{1}_{\boldsymbol{x}^k \in \mathcal{N}_{d_{\boldsymbol{w}}}(\boldsymbol{w}_i)}, \tag{4.8}$$

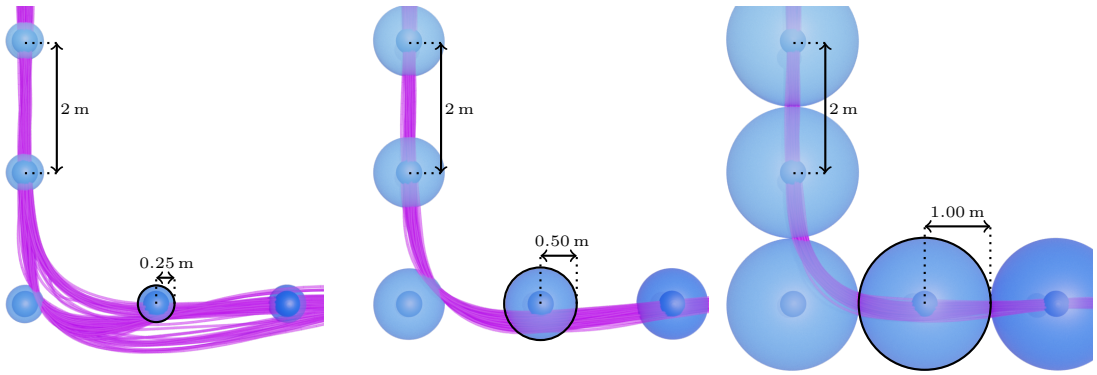
where $\mathcal{N}_{d_{\boldsymbol{w}}}(\boldsymbol{w}_i)$ is a neighborhood around $\boldsymbol{w}_i$ with radius $d_{\boldsymbol{w}}$ and $c_{\boldsymbol{w}}$ is the waypoint completion coefficient. Since we want to reward the drone for flying through the neighborhood, the value is subtracted from the total cost. Because flying through the neighborhood is not a hard constraint, the drone can miss a waypoint if it results in a lower cost (e.g., during an aggressive turn, where passing through the waypoint would result in an inability to track the reference). This behavior, along with the effects of the parameter $d_{\boldsymbol{w}}$, are illustrated in Figure 4.11.

To make the control from the database usable, we also need to push the drone towards a dynamic state that matches the starting state $\boldsymbol{x}_0^{\boldsymbol{\pi}}$ of the motion primitive $\boldsymbol{\pi}$. This can be achieved by adding another term to the cost function, which penalizes the distance between the required state and the nearest state (in 3D) on the rollout trajectory

$$d(\boldsymbol{x}^k, \boldsymbol{x}_0^{\boldsymbol{\pi}}) = \rho_{\boldsymbol{\pi}}(\boldsymbol{x}_{j^*}^k, \boldsymbol{x}_0^{\boldsymbol{\pi}}), \qquad j^* = \operatorname*{argmin}_{0 \le j \le N} \|\boldsymbol{p}_j^k - \boldsymbol{p}_0^{\boldsymbol{\pi}}\|. \tag{4.9}$$

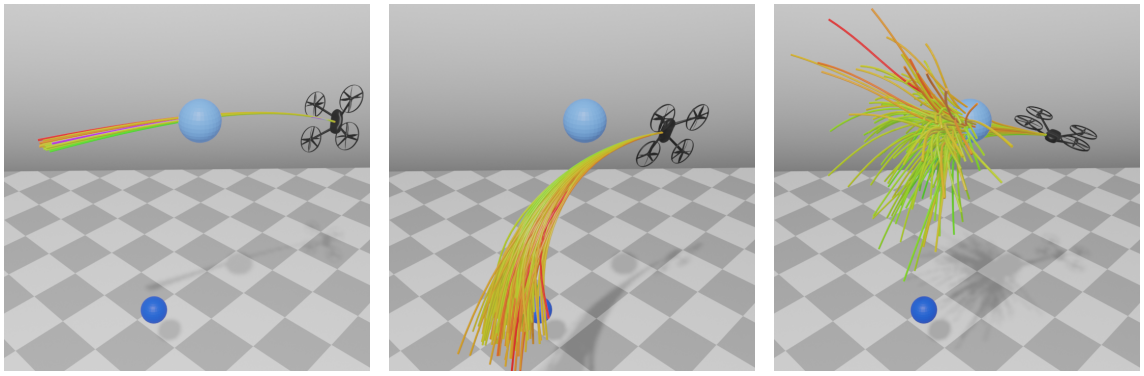
For $\rho_{\boldsymbol{\pi}}$, we will use a function identical to Eq. 4.4 but will allow different weighing coefficients for more versatility.

Another tunable parameter is the control noise, defining the covariance matrix Σ (Eq. 2.5). By tuning the covariance, we are choosing between exploration (high noise allows to explore a larger part of the input space) and exploitation (lower noise allows faster convergence to a local minima). The effects of changing the noise variance are shown in Figure 4.12.



(a) $d_{\boldsymbol{w}} = 0.25\,\mathrm{m}$ (b) $d_{\boldsymbol{w}} = 0.50\,\mathrm{m}$ (c) $d_{\boldsymbol{w}} = 1.00\,\mathrm{m}$

Figure 4.11: Illustration of the parameter's $d_{\boldsymbol{w}}$ effect. The task is to track a reference generated as a linear interpolation between the waypoints moving at $8\,\mathrm{m\,s^{-1}}$. When passing through the neighborhood requires an aggressive maneuver penalized heavily by the cost function, the controller may decide to miss it to better track the reference (Fig. 4.11a). This behavior can be suppressed by increasing the neighborhood radius (Fig. 4.11b and 4.11c), however, the upper limit depends on the task.



(a) $\Sigma = \text{diag}(1.0, 0.5, 0.5, 0.3)$ (b) $\Sigma = \text{diag}(4.0, 3.0, 3.0, 1.5)$ (c) $\Sigma = \text{diag}(6.0, 8.0, 8.0, 6.0)$

Figure 4.12: With increasing variance of the control noise, we need to find a balance between exploitation (low noise, Fig. 4.12a) and exploration (high noise, Fig. 4.12c). The drone should pass through the waypoints (light and dark blue spheres, in order). When the noise is too low (left), the controller is unable to change the direction of movement. By increasing the noise, rollouts get more variable.

4.4 Summary

In this chapter, we introduced our proposed method for controlling a drone using the MPPI controller and split the task into three subproblems. First, we compute multiple short time-optimal trajectories (motion primitives) for our target drone by specifying the task as a non-linear program and solving it using an available NLP solver. The NLP formulation was presented in Section 4.1, along with the track generation algorithm and examples of the motion primitives. The motion primitives are saved into a database that bins the trajectory states based on their distance relative to the origin in the z-axis and xy-plane. Building this database and resolving the queries was covered in Section 4.2. An MPPI controller is then used to control the body rates and collective thrust of the drone, using the precomputed trajectories to guide the controller. Section 4.3 discussed the implementation of the MPPI controller and introduced all parts of the cost function used by the predictive controller.

Chapter 5

Results

In this chapter, we will verify the proposed methods and present results of the performed experiments. All tests were run on a computer equipped with an *Intel i7-1165G7* CPU and a *GeForce MX450* GPU, running Ubuntu 20.04.

5.1 Database Creation

To fill the database, 1000 motion primitives were generated (as described in Alg. 3). The waypoints were generated with parameters $r_{min} = 3\,\text{m}$ and $r_{max} = 6\,\text{m}$ in a map with a bounding box of 20x20x20 m. Since computing one trajectory can take up to 10 minutes, the trajectories were computed on MetaCentrum[1], which allows computing the trajectories in parallel. The computed motion primitives are shown in Figure 5.1.

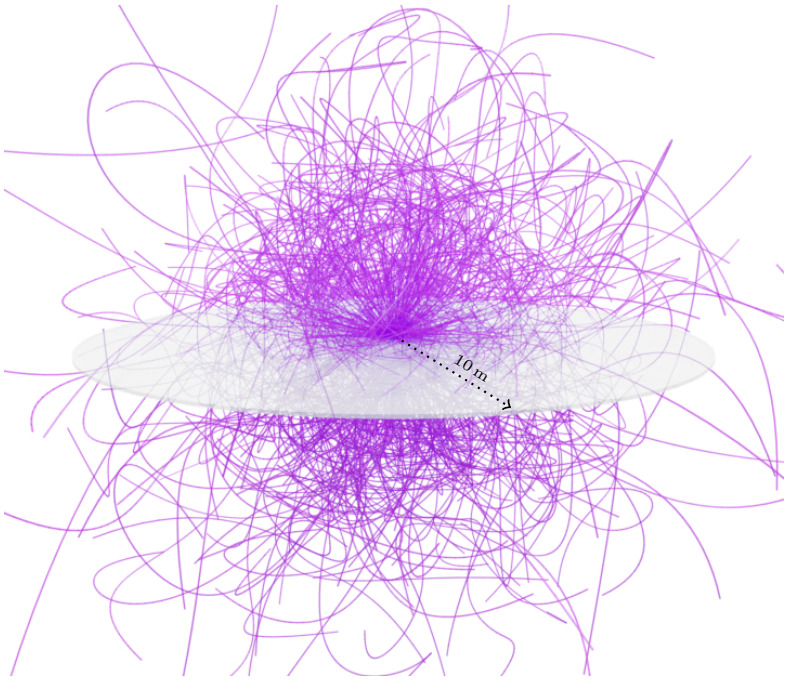


Figure 5.1: Visualization of the 1000 motion primitives computed. A disc with a radius of 10 m is added for scale.

[1]Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

5.2 Proposed Method Verification

At first, all the proposed algorithms are tested in a simple environment where we know the state precisely and the evolution of the system is given by the same forward dynamics simulation as the prediction of MPPI, making the prediction perfect. We will generate multiple tracks (sequences of 3D waypoints) using the same method as in Section 4.1 (Figure 4.2) and use the proposed methods to fly along them. The values of the parameters used during the simulations are presented in Table 5.1. Two additional parameters $\omega_{xy,max}$ and $\omega_{z,max}$ are introduced that limit the desired control applied to the system[2].

Parameter	Value	Unit
m	0.85	kg
l	0.15	m
c_{tf}	0.05	m
J	$\mathrm{diag}(0.001, 0.001, 0.0017)$	$\mathrm{kg\,m^2}$
T_{min}	0.0	N
T_{max}	6.88	N
$\omega_{xy,max}$	15.0	$\mathrm{rad\,s^{-1}}$
$\omega_{z,max}$	0.3	$\mathrm{rad\,s^{-1}}$

Table 5.1: Drone parameters (Section 2.2).

5.2.1 Speed of computation

An important value is the time needed for one iteration of the control loop. For us, the upper limit is 10 ms per iteration, corresponding to the update rate of controllers in the MRS UAV system (discussed in Section 5.3) working on the collective thrust and body rates level (100 Hz). Even though it is possible to use slower update rates in the simulations, we want to satisfy this limit since our ultimate goal is to test the controller on a real drone. Figure 5.2 shows how the number of rollouts K and number of prediction steps N affect the iteration time.

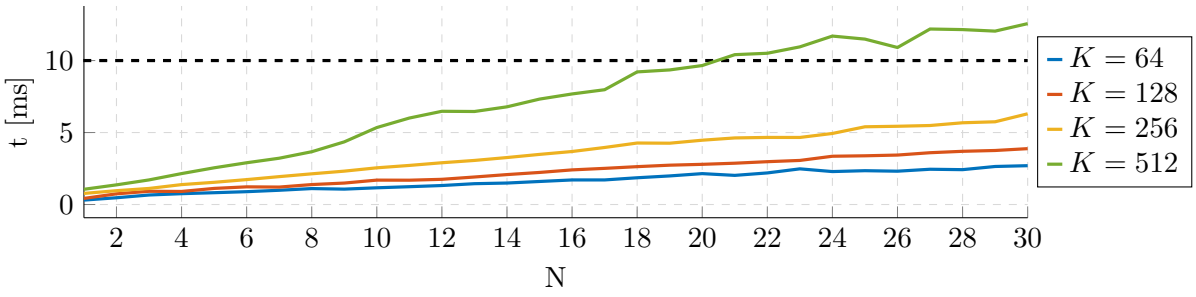


Figure 5.2: Average time of one iteration of our MPPI control algorithm depending on the number of rollouts K and number of prediction steps N. To run the controller at 100 Hz, we need to keep the iteration time below 10 ms (black dashed line).

[2]Approach commonly used in other framework that implement MPPI, such as the ROS navigation stack `Nav2`, publicly available at `https://github.com/ros-planning/navigation2/tree/main/nav2_mppi_controller`.

5.2.2 Control input limits

Another important requirement imposed on the controller is to respect the limits on desired collective thrust, body rates, and single rotor thrusts. In Figure 5.3 and Figure 5.4, we show that the limits are satisfied and often reached, showing the ability of the MPPI controller to saturate the inputs and, therefore, use the full agile capabilities of the platform.

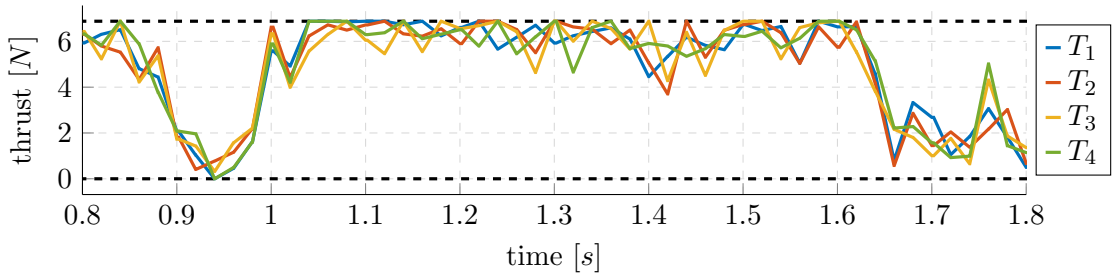


Figure 5.3: An example of the desired single rotor thrusts. The limits are shown as black dashed lines with values $T_i \in [0, 6.88]$ $i = 1, \ldots, 4$.

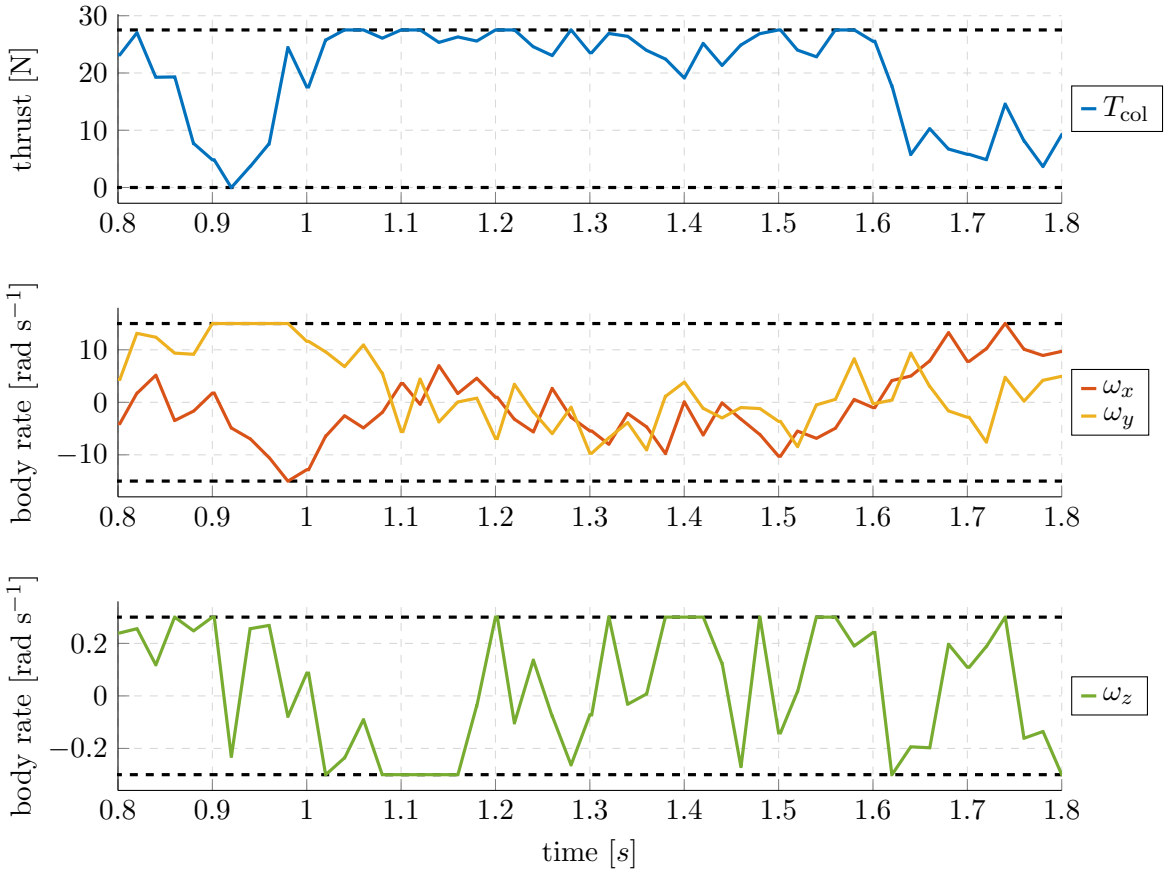
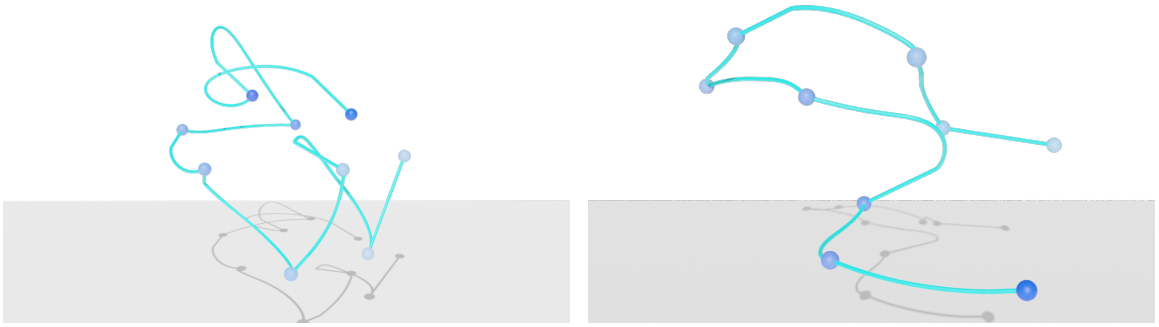


Figure 5.4: An example of the controller's output. The limits are shown as black dashed lines with values $T_{\text{col}} \in [0.0, 27.52]$, ω_x and $\omega_y \in [-15.0, 15.0]$, and $\omega_z \in [-0.3, 0.3]$.

5.2.3 Flying through the tracks

In the following section, we will present results obtained by flying 100 times through four of the tracks. Since our task was neither flying as fast as possible nor flawlessly passing all waypoints, the results serve mainly informational purposes, and the behavior could be changed by tuning the cost function parameters differently (e.g., not passing a waypoint could be penalized more to push the controller towards a higher waypoint completion). The references (motion primitives) are retrieved from the database during the flight and are used to guide the MPPI controller. An example of the reference trajectories is shown in Figure 5.5. We see that the results from the database guide the controller reasonably in most of the cases. However, in the case of the last three waypoints in `Track 3`, we can also see an example of retrieving an unsuitable database result, where the drone is forced to change its velocity in an unnecessary way (but even then is the controller able to track the reference and reach the last waypoint). The parameter values were tuned by hand to obtain reasonable results — the values used in the tests are presented in Table 5.2.



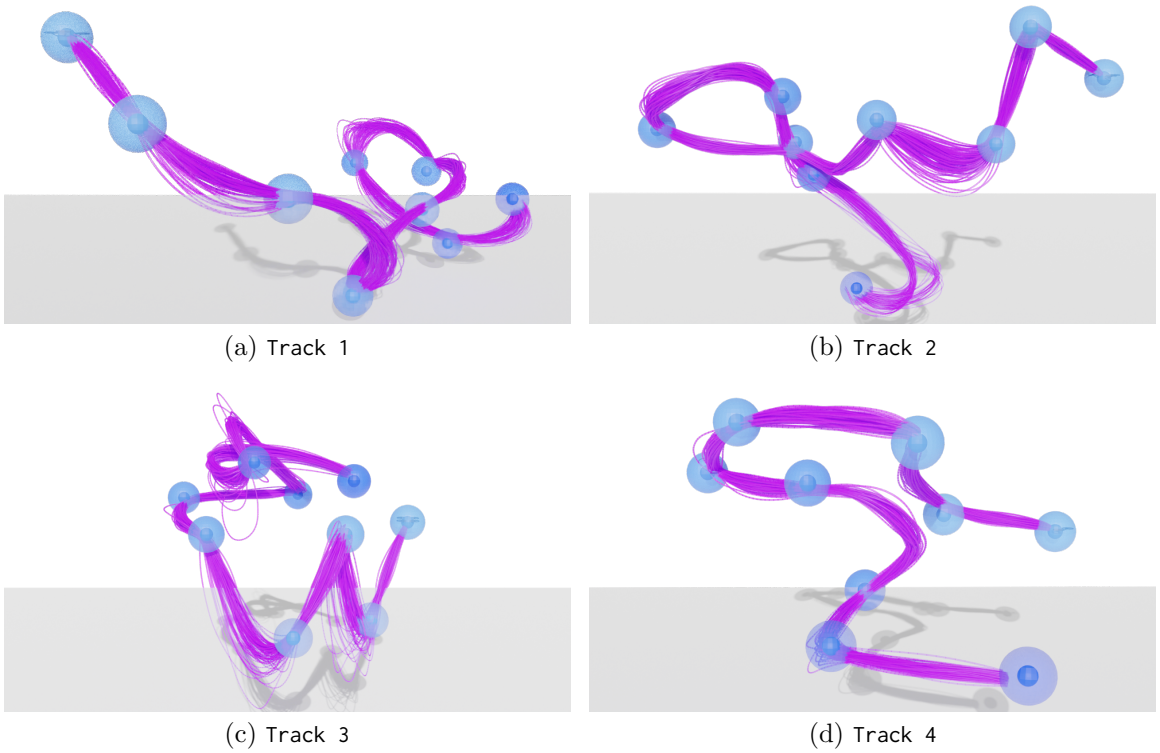
(a) `Track 3` (b) `Track 4`

Figure 5.5: Reference trajectories retrieved from the database online during the flight.

Parameter	Value	Parameter	Value
K	512	$c_{\boldsymbol{w}}$	10000.0
N	20	$d_{\boldsymbol{w}}$	0.5
λ	10^{-5}	Σ	$\mathrm{diag}(4.0, 3.5, 3.5, 1.5)$
R_T	0.01	$R_{\Delta T}$	0.01
$R_{\boldsymbol{\omega}_{xy}}$	0.2	$R_{\Delta\boldsymbol{\omega}_{xy}}$	0.2
$R_{\boldsymbol{\omega}_z}$	0.2	$R_{\Delta\boldsymbol{\omega}_z}$	0.2
R	$\mathrm{diag}(R_T, R_{\boldsymbol{\omega}_{xy}}, R_{\boldsymbol{\omega}_z})$	R_Δ	$\mathrm{diag}(R_{\Delta T}, R_{\Delta\boldsymbol{\omega}_{xy}}, R_{\Delta\boldsymbol{\omega}_z})$
$c^{\mathrm{ref}}_{\boldsymbol{p}}$	100.0	$c^{\boldsymbol{\pi}}_{\boldsymbol{p}}$	800.0
$c^{\mathrm{ref}}_{\boldsymbol{v}}$	0.1	$c^{\boldsymbol{\pi}}_{\boldsymbol{v}}$	5.0
$c^{\mathrm{ref}}_{\boldsymbol{q}}$	0.01	$c^{\boldsymbol{\pi}}_{\boldsymbol{q}}$	5.0
$c^{\mathrm{ref}}_{\boldsymbol{\omega}}$	0.01	$c^{\boldsymbol{\pi}}_{\boldsymbol{\omega}}$	5.0

Table 5.2: Parameters of the MPPI controller used in our simulation environment

In Table 5.3, we list the percentage of runs where the drone passed through all waypoints in the given tolerance, the distance traveled, the time of the flight, and information about the velocity. The controller is able to pass most of the tracks consistently, as can be seen in Figure 5.6. The only exception is `Track 1`, where the controller misses the last waypoint in approximately one out of five runs (shown in Figure 5.7).



(a) `Track 1` (b) `Track 2`

(c) `Track 3` (d) `Track 4`

Figure 5.6: Visualization of 100 runs on each of the four testing tracks. The trajectories flown by the drone are shown as purple curves and the task is to fly through neighborhoods of the waypoints (blue spheres, from light to dark blue). For `Track 1`, only trajectories that went through all waypoints are shown in Fig. 5.6a, the unsuccessful runs are shown in Figure 5.7.

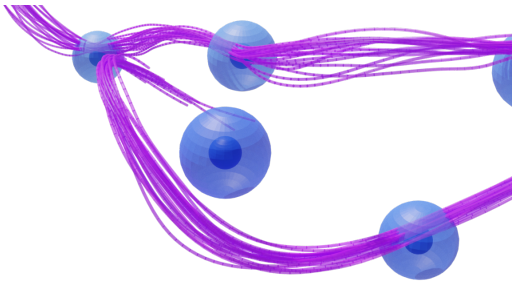


Figure 5.7: On `Track 1` (Fig. 5.6a), the drone missed the last waypoint in 21 out of 100 tries (all 21 trajectories are visualized here). This can be attributed to an aggressive maneuver needed to change the direction of flight and could be resolved by tuning the parameters (as discussed in Figure 4.11).

To test the influence of having a full dynamic state as the reference, we repeated the test with zero weight on velocity, rotation and angular velocity ($c_{\boldsymbol{v}}^{\boldsymbol{\pi}}$, $c_{\boldsymbol{q}}^{\boldsymbol{\pi}}$ and $c_{\boldsymbol{\omega}}^{\boldsymbol{\pi}}$), pushing the controller only to track the position information of the motion primitive. The results are presented in Table 5.4. Finally, we test the parameters in Table 5.2, but change the initialization method (line 17 in Alg. 2) to duplicate the last input instead of using the control input retrieved along with the motion primitive from the database. The results are presented in Table 5.5.

Track	Successful runs [%]	Distance [m] (avg ± std)	Time [s] (avg ± std)	Velocity [$\mathrm{m\,s^{-1}}$] (avg ± std)	Max velocity [$\mathrm{m\,s^{-1}}$]
1	72	43.75 ± 5.74	7.06 ± 1.09	6.43 ± 2.11	11.28
2	100	35.77 ± 1.61	6.69 ± 0.33	5.31 ± 1.86	9.91
3	100	38.69 ± 1.34	7.02 ± 0.19	5.47 ± 2.33	12.09
4	100	33.37 ± 0.45	5.26 ± 0.05	6.40 ± 2.38	9.80

Table 5.3: Statistical results of 100 runs on each of the four testing tracks - all reference states used.

Track	Successful runs [%]	Distance [m] (avg ± std)	Time [s] (avg ± std)	Velocity [$\mathrm{m\,s^{-1}}$] (avg ± std)	Max velocity [$\mathrm{m\,s^{-1}}$]
1	79	41.73 ± 5.67	6.49 ± 1.20	6.60 ± 2.07	10.96
2	99	35.57 ± 1.58	6.51 ± 0.27	5.39 ± 1.81	9.56
3	99	38.93 ± 1.67	6.94 ± 0.32	5.83 ± 2.49	12.14
4	100	33.25 ± 0.44	5.20 ± 0.07	6.26 ± 2.37	9.55

Table 5.4: Statistical results of 100 runs on each of the four testing tracks - reference position used only.

Track	Successful runs [%]	Distance [m] (avg ± std)	Time [s] (avg ± std)	Velocity [$\mathrm{m\,s^{-1}}$] (avg ± std)	Max velocity [$\mathrm{m\,s^{-1}}$]
1	56	44.04 ± 7.08	7.29 ± 1.32	6.03 ± 2.33	11.32
2	79	37.28 ± 4.11	7.45 ± 0.91	5.33 ± 1.94	10.05
3	100	39.14 ± 1.65	7.15 ± 0.21	5.56 ± 2.39	11.10
4	97	33.00 ± 1.93	5.39 ± 0.60	6.60 ± 2.17	9.62

Table 5.5: Statistical results of 100 runs on each of the four testing tracks - all reference states, without using input from the database.

There seems to be no significant difference between tracking the full dynamic state along the motion primitive and tracking the position only. This could have been expected even from the original parameter values used, where the coefficients $c_{\boldsymbol{v}}^{\boldsymbol{\pi}}$, $c_{\boldsymbol{q}}^{\boldsymbol{\pi}}$ and $c_{\boldsymbol{\omega}}^{\boldsymbol{\pi}}$ had to be set small compared to the position coefficient $c_{\boldsymbol{p}}^{\boldsymbol{\pi}}$ to obtain good performance. Increasing the values had a detrimental effect on the results and resulted in an unstable flight.

On the other hand, using the input from the database shows a great improvement in the performance over a different input initialization method (such as duplicating the last input). This serves as a confirmation of the assumption that the input initialization has a large impact on the controller's performance.

5.2.4 Obstacles

One of the key advantages of the MPPI method is the ability to include obstacles in the cost function and force the controller to avoid them. This can be achieved by adding a term to the cost function that heavily penalizes states where the drone collides with the environment

$$S'_k = S_k + \sum_{j=0}^{N} c_{\text{obs}} \cdot \mathbb{1}_{\boldsymbol{x}_j^k \in \mathcal{C}_{\text{obs}}} \tag{5.1}$$

where S_k is the original cost of the rollout, c_{obs} is the cost coefficient for the collision term, and $\mathbb{1}_{\boldsymbol{x}_j^k \in \mathcal{C}_{\text{obs}}}$ is an indicator function, which is 1 if the drone at state $\boldsymbol{x}_j^k$ is in collision with the environment. Collision detection can be then performed in multiple ways (e.g., using mesh collision detection such as *Rapid* [27] or discrete grid-based methods, such as *OctoMap* [20]), and the exact implementation depends on the platform capabilities (e.g., Light Detection and Ranging (LiDAR)). For our tests, we have used value of $c_{\text{obs}} = 10000$. By adding collision detection into the simulation environment, we can show that the MPPI controller is able to handle and successfully avoid obstacles, as visualized in Figure 5.8 and Figure 5.9.

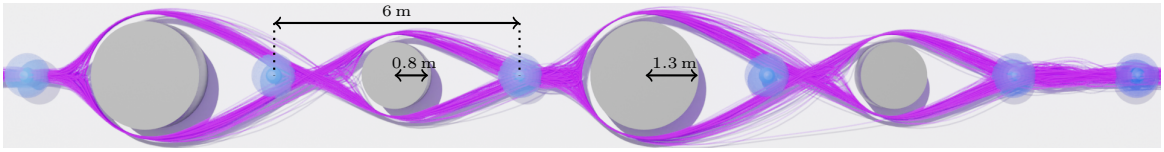


Figure 5.8: The drone should follow a reference moving at a constant speed of $6\,\mathrm{m\,s^{-1}}$ from left to right and pass through the neighborhoods of the waypoints (translucent blue spheres). However, due to the obstacles (grey cylinders), the drone cannot fly directly straight. Trajectories from 100 runs are shown as purple curves — we can see that sometimes the drone does not pass through the waypoint neighborhood. That is caused by the cost function not enforcing passing through the neighborhood, and it can happen that missing a waypoint results in a better cost (e.g., due to aggressive maneuvers).

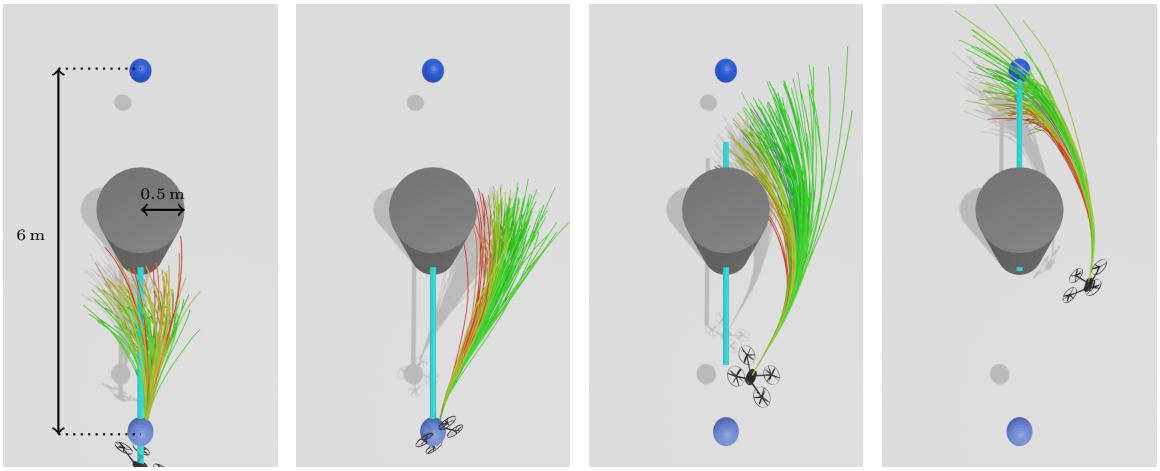


Figure 5.9: The objective of the drone is to track a straight line moving in the y-direction at a constant speed of $6\,\mathrm{m\,s^{-1}}$ (cyan) and to fly through the waypoints (blue spheres). However, the reference passes through an obstacle (gray cylinder), which the drone needs to avoid. The rollouts that result in a collision are heavily penalized (red curves) in comparison to the non-colliding rollouts (green).

5.3 MRS UAV system

In the previous section, we have shown that the algorithms work under ideal conditions. However, this is far from the real world, where we can only estimate the system state and the evolution of the system is more complex (with effects such as drag and disturbances). To simulate some of the mentioned deviations and to get closer to the real world, we reimplemented and integrated the control algorithm into the multirotor Unmanned Aerial Vehicle (UAV) control and estimation system developed by the Multi-robot Systems Group (MRS) at the Czech Technical University (CTU) [6]. Visualizations from the system are shown in Figure 5.10.

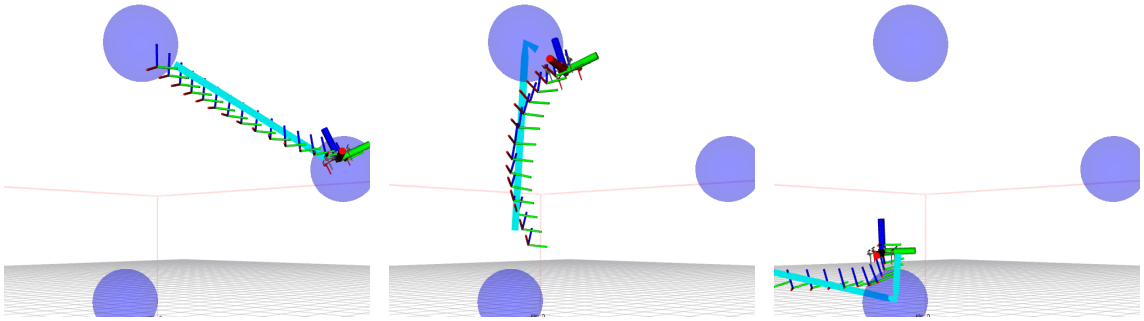


Figure 5.10: Visualization of the MPPI controller controlling a drone in the MRS UAV system. The images show the waypoint neighborhoods (blue spheres), reference to track (cyan line), the drone, and its predicted nominal trajectory.

Since the system is slower than the one used in Section 5.2 (with parameters listed in Table 5.6), we wanted to increase the prediction horizon while satisfying the controller update rate fixed at 100 Hz. However, it is not as straightforward as increasing the number of predicted time steps (parameter N). The prediction horizon would be increased indeed, but it would negatively impact the computational complexity. Firstly, it directly increases the number of iterations of the dynamics simulation needed. Secondly, it increases the number of optimization parameters (N control inputs $\boldsymbol{u}$), which means we would need to do more rollouts to determine which input change decreases the value of the cost function.

Another option is to compute the rollouts only every n-th iteration of the controller and increase the controller time step to $n \cdot \Delta t$, applying a constant control in the other iterations. However, that would mean no state feedback would be used during the other iterations, hindering the controller's performance in real-world scenarios. To make use of the state feedback,

Parameter	Value	Unit	Parameter	Value	Unit
m	2.0	kg	T_{min}	0.371	N
l	0.25	m	T_{max}	16.48	N
c_{tf}	0.07	m	$\omega_{xy,max}$	3.0	$\mathrm{rad\,s^{-1}}$
J	$\mathrm{diag}(0.033, 0.033, 0.063)$	$\mathrm{kg\,m^2}$	$\omega_{z,max}$	0.1	$\mathrm{rad\,s^{-1}}$

Table 5.6: Drone parameters used in the MRS UAV system (Section 2.2).

we employ an interpolation scheme, and instead of shifting the nominal control by one each time (lines 16 in Alg. 2), we interpolate between them and run the optimization task every iteration. In our case, we will use $n = 10$, resulting in MPPI having the prediction time step 0.1 s but still being able to update at 100 Hz.

Since the drone used by the UAV system has different parameters than those used in Section 5.2, we would need to recompute the motion primitives imported to the database. Otherwise, the trajectories may not be feasible, and the control inputs would result in different time evolutions of the system. Due to time constraints and implementation complexity, we have decided to test the MPPI controller without the database. We used a linear interpolation of the waypoints instead of the motion primitive as the reference (using parameter $c_{\boldsymbol{p}}^{\text{ref}}$ to penalize the 3D position distance to the desired position) and penalized the minimal 3D distance to the next waypoint for each rollout (introducing parameter $c_{\boldsymbol{p}}^{\boldsymbol{w}}$).

Values of the parameters used are shown in Table 5.7. The parameters were again tuned by hand to obtain reasonable results. One notable change is the need to increase the λ value (from 10^{-5} used in our simulator to 10^{-3}), which improved the stability of the drone in the MRS UAV simulator. This means more averaging of the control noise is done, compared to using only the input that yields the best rollout cost (as discussed at the end of Section 2.1.2).

In Table 5.8, we present the results from flying along a sequence of 11 waypoints (a track similar to those used in Section 5.2). To generate the reference trajectory, we use linear interpolation between each consecutive pair of waypoints, with n_{steps} steps per pair. We measured the distance to the reference d_{ref} and the velocity $\|\boldsymbol{v}\|$ during the flight. We show that the controller is able to track the reference and visit the waypoints reliably until approximately $5\,\text{m}\,\text{s}^{-1}$, where the drone is no longer able to both keep up with the reference and reach the waypoint neighborhoods.

Parameter	Value	Parameter	Value
K	512	$c_{\boldsymbol{w}}$	10000.0
N	15	$d_{\boldsymbol{w}}$	1.0
λ	10^{-3}	Σ	$\text{diag}(1.7, 0.4, 0.4, 0.2)$
R_T	0.01	$R_{\Delta T}$	0.05
$R_{\boldsymbol{\omega}_{xy}}$	0.05	$R_{\Delta\boldsymbol{\omega}_{xy}}$	0.1
$R_{\boldsymbol{\omega}_z}$	0.1	$R_{\Delta\boldsymbol{\omega}_z}$	0.3
R	$\text{diag}(R_T, R_{\boldsymbol{\omega}_{xy}}, R_{\boldsymbol{\omega}_z})$	R_Δ	$\text{diag}(R_{\Delta T}, R_{\Delta\boldsymbol{\omega}_{xy}}, R_{\Delta\boldsymbol{\omega}_z})$
$c_{\boldsymbol{p}}^{\text{ref}}$	100.0	$c_{\boldsymbol{p}}^{\boldsymbol{w}}$	100.0

Table 5.7: Parameters of the MPPI controller used in the MRS UAV system

n_{steps}	Completed waypoints	d_{ref} [m] (avg ± std)	d_{ref} [m] (max)	$\|\boldsymbol{v}\|$ [m s^{-1}] (avg ± std)	$\|\boldsymbol{v}\|$ [m s^{-1}] (max)
200	11/11	0.11 ± 0.06	0.42	2.20 ± 0.67	3.91
170	11/11	0.14 ± 0.08	0.51	2.49 ± 0.70	4.41
140	11/11	0.15 ± 0.10	0.73	2.95 ± 0.70	4.89
110	11/11	0.22 ± 0.13	0.96	3.53 ± 0.94	6.14
80	10/11	0.31 ± 0.16	1.03	4.47 ± 1.27	7.78
50	5/11	0.53 ± 0.40	2.00	5.21 ± 1.61	8.45

Table 5.8: Results of flying through a track in the MRS UAV system with increasing speed of the reference.

5.4 Discussion

All tests were run on a laptop with a dedicated graphics card (*GeForce MX450* GPU with 896 cores operating at frequency 720 MHz), which brings up the question whether a sufficient computational power could be made available on a real drone. Thanks to the recent demand for AI application in robotics and to the advances in mobile graphical computing, it is possible to meet the requirements by using energy-efficient graphical modules. One such example are the *Jetson Orin Nano Series* modules developed by *NVIDIA*, offering 512 cores running at frequency 625 MHz in the lowest configuration[3]. In our experiments, we have used upto 512 rollouts, therefore, the number of cores is sufficient to run the tests. Moreover, even though the frequency is lower by approximately 15 %, we believe that it is possible to safely meet the required 100 Hz update rate by further code optimization[4]. We plan to test the algorithms on a real drone in the near future.

Multiple other tricks and ideas from the realm of predictive control could be used to improve the performance of our algorithms. An example would be the so called *Delay compensation by prediction* [3]. The time of the computation of each iteration can be expected to be very consistent thanks to a fixed number of computations performed per iterations, in contrast to iterative optimization algorithms, that run until convergence (or the time limit). Therefore, instead of addressing the problem starting at the current state, we can forward-simulate the system and start the optimization at a state at which the system will be when we will have computed the optimization iteration. We did not implement this idea in order to first test and validate our original method. However, for the real-world deployment, implementing such a trick is planned to increase the performance of the algorithms.

[3] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/ , accessed at Nov. 20, 2023

[4] A lot of data was being saved solely for the purpose of later visualization, which would not be needed in real deployment.

Chapter 6

Conclusion

This thesis dealt with the task of flying a drone along a given sequence of 3D waypoints using a sampling-based model predictive methodology to contol the drone at the level of body rates and collective thrust. In Chapter 2, we introduced the problem of optimal control and presented the main control algorithm — Model Predictive Path Integral (MPPI). We presented the mathematical model of a drone controlled by collective thrust and body rates and discussed a way to include low-level constraints (single rotor thrusts) into the task, thereby increasing the ability of the controller to fully exploit the platform potential.

A vital part of the controller's function is a good initialization of the nominal control, which is then adjusted by a weighted average of the sampled disturbances based on the task-specific cost function. In Chapter 4, we covered a method of computing short time-optimal trajectories (motion primitives) and designed a database able to store them and allow efficient retrieval in real-time. We discussed the design of the cost function and used the motion primitives to guide the controller while flying through generated tracks.

In Chapter 5, we verified the correct behavior of the proposed method. We showed the performance of the controller in multiple setups and demonstrated the importance of correct control input initialization. Moreover, we confirmed the ability of the controller to easily include obstacle avoidance in the planning, which is one of the most prominent features of the sampling-based methodologies. Finally, we implemented the MPPI algorithm into a third-party simulator and shown that the controller is able to work under more realistic conditions (namely, when lower-level dynamics and constraints are considered).

A part of Section 5.4 discussed the computational resources needed to run the proposed algorithms in real-time and compared the resources used to ones available on a real drone. We expect that our implementation of MPPI will be able to run on an actual drone in real-time after a few minor modifications, and we intend to conduct such experiments for a planned journal publication. In this thesis, we showed that the MPPI methodology may become a viable addition to other predictive control methods used for agile drone control, such as Nonlinear Model Predictive Control (NMPC) — mostly thanks to the ability to add almost arbitrary objective terms and constraints without a large increase in the computational complexity. A lot of research is yet to be concluded, however, we believe this thesis may serve as the groundwork for further work on MPPI in the Multi-robot Systems Group (MRS) at CTU Prague. Moreover, the implementation of the algorithms presented in this thesis required us to design and combine Compute Unified Device Architecture (CUDA) code with other systems that are already in use by other researchers in the group (e.g., Eigen [22]) in a memory-efficient manner. This contribution could serve as a foundation for other GPU powered applications, since this is the first controller implementation in the group using the GPU directly.

References

[1] C. Tao, H. Kim, and N. Hovakimyan, *RRT Guided Model Predictive Path Integral Method*, en, arXiv:2301.13143 [cs, eess], Jan. 2023. [Online]. Available: http://arxiv.org/abs/2301.13143.

[2] I. M. Balci, E. Bakolas, B. Vlahov, and E. Theodorou, *Constrained covariance steering based tube-mppi*, 2022. arXiv: 2110.07744 [math.OC].

[3] S. Gros and M. Diehl, "Numerical optimal control," Apr. 2022. [Online]. Available: https://www.syscop.de/files/2020ss/NOC/book-NOCSE.pdf.

[4] L. F. Recalde, B. S. Guevara, C. P. Carvajal, V. H. Andaluz, J. Varela-Aldás, and D. C. Gandolfo, "System identification and nonlinear model predictive control with collision avoidance applied in hexacopters uavs," *Sensors*, vol. 22, no. 13, 2022, ISSN: 1424-8220. DOI: 10.3390/s22134712. [Online]. Available: https://www.mdpi.com/1424-8220/22/13/4712.

[5] J. Yin, Z. Zhang, E. Theodorou, and P. Tsiotras, "Trajectory distribution control for model predictive path integral control using covariance steering," in *2022 International Conference on Robotics and Automation (ICRA)*, IEEE, May 2022. DOI: 10.1109/icra46639.2022.9811615. [Online]. Available: http://dx.doi.org/10.1109/ICRA46639.2022.9811615.

[6] T. Baca, M. Petrlik, M. Vrba, *et al.*, "The MRS UAV System: Pushing the Frontiers of Reproducible Research, Real-world Deployment, and Education with Autonomous Unmanned Aerial Vehicles," en, *Journal of Intelligent & Robotic Systems*, vol. 102, no. 1, p. 26, May 2021, ISSN: 0921-0296, 1573-0409. DOI: 10.1007/s10846-021-01383-5. [Online]. Available: https://link.springer.com/10.1007/s10846-021-01383-5.

[7] P. Foehn, A. Romero, and D. Scaramuzza, "Time-optimal planning for quadrotor waypoint flight," en, *Science Robotics*, vol. 6, no. 56, Jul. 2021, ISSN: 2470-9476. DOI: 10.1126/scirobotics.abh1221. [Online]. Available: https://www.science.org/doi/10.1126/scirobotics.abh1221.

[8] H. Nguyen, M. Kamel, K. Alexis, and R. Siegwart, "Model predictive control for micro aerial vehicles: A survey," in *2021 European Control Conference (ECC)*, 2021, pp. 1556–1563. DOI: 10.23919/ECC54610.2021.9654841.

[9] A. Romero, S. Sun, P. Foehn, and D. Scaramuzza, "Model predictive contouring control for near-time-optimal quadrotor flight," *CoRR*, vol. abs/2108.13205, 2021. arXiv: 2108.13205. [Online]. Available: https://arxiv.org/abs/2108.13205.

[10] M. Sivakumar and N. Tyj, "A literature survey of unmanned aerial vehicle usage for civil applications," *Journal of Aerospace Technology and Management*, vol. 13, Nov. 2021. DOI: 10.1590/jatm.v13.1233.

[11] R. Verschueren, G. Frison, D. Kouzoupis, *et al.*, "Acados – a modular open-source framework for fast embedded optimal control," *Mathematical Programming Computation*, Oct. 2021, ISSN: 1867-2957. DOI: 10.1007/s12532-021-00208-8. [Online]. Available: https://doi.org/10.1007/s12532-021-00208-8.

[12] J. B. Rawlings, D. Q. Mayne, and M. M. Diehl, *Model Predictive Control: Theory, Computation, and Design*, 2nd ed. USA: Nob Hill Publishing, 2020, ISBN: 978-0-9759377-5-4. [Online]. Available: https://sites.engineering.ucsb.edu/~jbraw/mpc/.

[13] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019. DOI: 10.1007/s12532-018-0139-4.

[14] H. Shakhatreh, A. H. Sawalmeh, A. Al-Fuqaha, *et al.*, "Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges," *IEEE Access*, vol. 7, pp. 48 572–48 634, 2019. DOI: 10.1109/ACCESS.2019.2909530.

[15] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Information-Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving," en, *IEEE Transactions on Robotics*, vol. 34, no. 6, pp. 1603–1622, Dec. 2018, ISSN: 1552-3098, 1941-0468. DOI: 10.1109/TRO.2018.2865891. [Online]. Available: https://ieeexplore.ieee.org/document/8558663/.

[16] G. Williams, B. Goldfain, P. Drews, K. Saigol, J. Rehg, and E. Theodorou, "Robust sampling based model predictive control with sparse objective information," Jun. 2018. DOI: 10.15607/RSS.2018.XIV.042.

[17] G. Williams, A. Aldrich, and E. A. Theodorou, "Model predictive path integral control: From theory to parallel computation," *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 2, pp. 344–357, 2017. DOI: 10.2514/1.G001921. [Online]. Available: https://doi.org/10.2514/1.G001921.

[18] M. Bangura, M. Melega, R. Naldi, and R. Mahony, *Aerodynamics of rotor blades for quadrotors*, 2016. arXiv: 1601.00733 [physics.flu-dyn].

[19] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," en, in *2016 International Conference on Robotics and Automation (ICRA)*, Stockholm: IEEE, May 2016, pp. 1433–1440, ISBN: 978-1-4673-8026-3. DOI: 10.1109/ICRA.2016.7487277. [Online]. Available: https://ieeexplore.ieee.org/document/7487277/.

[20] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, 2013, Software available at http://octomap.github.com. DOI: 10.1007/s10514-012-9321-0. [Online]. Available: http://octomap.github.com.

[21] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 International Conference on Robotics and Automation (ICRA)*, IEEE, 2011, pp. 2520–2525. DOI: 10.1109/ICRA.2011.5980409.

[22] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[23] D. Lam, C. Manzie, and M. Good, "Model predictive contouring control," in *49th IEEE Conference on Decision and Control (CDC)*, 2010, pp. 6137–6142. DOI: 10.1109/CDC.2010.5717042.

[24] D. Q. Mayne and E. C. Kerrigan, "Tube-based robust nonlinear model predictive control," *IFAC Proceedings Volumes*, vol. 40, no. 12, pp. 36–41, 2007, 7th IFAC Symposium on Nonlinear Control Systems, ISSN: 1474-6670. DOI: https://doi.org/10.3182/20070822-3-ZA-2920.00006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667016354994.

[25] S. M. LaValle, *Planning Algorithms*. USA: Cambridge University Press, 2006, ISBN: 0521862051.

[26] H. Samet, *Foundations of Multidimensional And Metric Data Structures*. Morgan Kaufmann, Aug. 2006, ISBN: 9780123694461.

[27] S. Gottschalk, M. Lin, and D. Manocha, "Obbtree: A hierarchical structure for rapid interference detection," *Computer Graphics*, vol. 30, Oct. 1997. DOI: 10.1145/237170.237244.

[28] H. Maurer, "On optimal control problems with bounded state variables and control appearing linearly," *SIAM Journal on Control and Optimization*, vol. 15, no. 3, pp. 345–362, 1977. DOI: 10.1137/0315023. eprint: https://doi.org/10.1137/0315023. [Online]. Available: https://doi.org/10.1137/0315023.

Appendix

Attachments

The attached file `multimedia.zip` contains videos of the flights described in Section 5.2 and Section 5.3.