Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering - Automation and Intelligent Cyber-Physical Systems

A.Y. 2024/2025

Graduation Session December 2025

Control Algorithms and Hardware Accelerations for Unmanned Aerial Vehicles

Implementation and Numerical Precision Analysis for MPPI Acceleration

Supervisors:
Prof. Paolo Bernardi

Candidate:
Luigi Graziosi

External Supervisor:
Prof. Marcello Traiola
Prof. Marco Tognon
Prof. Tommaso Belvedere

To my family,

for their love and strength.

And to those watching us from heaven.

Forever in my heart.

Abstract

This thesis investigates the real-time implementation of a Model Predictive Path Integral controller for quadrotor platforms, focusing on how numerical precision affects computational performance on embedded hardware. Although MPPI is well established for nonlinear optimal control, its deployment on resource-limited devices remains challenging because each control cycle requires evaluating thousands of trajectory rollouts. Surprisingly, no previous work has examined whether reduced-precision arithmetic can speed up MPPI on embedded systems without compromising the accuracy needed for stable flight.

To address this question, an MPPI controller was integrated into the TeleKyb3 framework and adapted to interface directly with the existing attitude controller by reformulating the system dynamics in terms of desired angular velocities. This avoided additional control layers and enabled real-time execution on an NVIDIA Jetson Orin Nano. The controller was first validated in Gazebo through different tests, then deployed retuning on the `mkquad5` platform in an indoor VICON environment. The real-world experiments confirmed stable behavior, with hovering accuracy within $\pm 5\,\mathrm{mm}$ and trajectory tracking errors around 5–10 cm.

A dedicated numerical study was then conducted using a custom MuJoCo implementation running directly on the Jetson, allowing a systematic comparison between `float32` and `float16` across thirty controller configurations. The results show that half-precision arithmetic consistently reduces computation time, typically by 7–15%, while maintaining tracking accuracy effectively identical to `float32`.

These findings indicate that MPPI naturally tolerates the rounding errors introduced by reduced precision, thanks to its sampling-based structure and receding-horizon updates. Based on this, the thesis outlines several guidelines for hardware-oriented implementations, including using 16-bit arithmetic as a sensible default and adopting mixed-precision strategies where necessary. Overall, the work shows that numerical precision can be safely reduced without loss of control quality, offering a practical pathway toward efficient and hardware-aware predictive controllers for autonomous aerial systems.

Acknowledgements

The completion of this thesis would not have been possible without the guidance, support, and encouragement of numerous people and institutions.
First, I am profoundly grateful to my supervisors, Marcello Traiola, Marco Tognon and Tommaso Belvedere for their expert guidance, constructive feedback, and constant availability throughout this research journey.
I also wish to thank professor Paolo Bernardi for their valuable advice and for fostering an inspiring academic environment.
I am deeply indebted to the members of the Rainbow team at IRISA/Inria, Rennes, whosev collaborative spirit and stimulating discussions significantly enriched this work.
I would like to express my heartfelt appreciation to my family and my friends for their patience, encouragement and unwavering belief in me during the challenges of this endeavor.

Table of Contents

List of Tables

List of Figures

Abbreviations

CNRS	Centre National de la Recherche Scientifique
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
D F	Differential Flatness
FP16	16-bit Floating Point Precision
FP32	32-bit Floating Point Precision
FPGA	Field-Programmable Gate Array
Gazebo	3D Robotics Simulator
GenoM	Generator of Modules
GPU	Graphics Processing Unit
HJB	Hamilton–Jacobi–Bellman
HPC	High Performance Computing
INRIA	I N Recherche en Informatique et en Automatique
ITC	Information-Theoretic Control
JAX	Just After eXecution
KL	Kullback–Leibler
LAAS-CNRS	Robotics Lab in Toulouse
LOG	Data Log File
LQR	Linear Quadratic Regulator

MBRL	Model-Based Reinforcement Learning
MC	Monte Carlo
MCS	Monte Carlo Sampling
MkQuad5	Quadrotor Platform
MPC	Model Predictive Control
MPPI	Model Predictive Path Integral
MuJoCo	Multi-Joint dynamics with Contact
NMPC	Nonlinear Model Predictive Control
OR	Optimization Routine
Orin Nano	NVIDIA Jetson Orin Nano Module
PDE	Partial Differential Equation
PI	Path Integral
PIC	Path Integral Control
PID	Proportional–Integral–Derivative
Pocolibs	Real-time middleware
ProxQP	Quadratic programming solver
PWM	Pulse Width Modulation
Quat	Quaternion representation for 3D orientation
QP	Quadratic Programming
Rainbow	Team for Robotics and Vision
RK4	Fourth Order Runge Kutta integration method
RMSE	Root Mean Square Error
ROTORCRAFT	low-level control module
RPM	Revolutions Per Minute
RT	Real-Time
SE(3)	Special Euclidean Group in Three Dimensions
SFD	System Functional Diagram
SSH	Secure Shell
Taran Team	FPGA research group
TeleKyb3	TeleKyb3 Framework for Aerial Robotics

UAV	Unmanned Aerial Vehicle
UAVAtt	Low level attitude control module
UKF	Unscented Kalman Filter
VICON	Motion Capture System

Chapter 1

Introduction

1.1 Context and Motivation

In recent years, real-time control of autonomous systems has become a central focus in robotics and automation research. In particular, the use of drones and autonomous aerial platforms requires the development of increasingly sophisticated and advanced controllers capable of combining precision, robustness, execution speed, and stability.

Model Predictive Path Integral (MPPI) control has emerged as one of the most promising techniques for optimal control in complex environments. **MPPI** is based on the idea of generating and evaluating a large number of candidate trajectories in parallel, selecting the one that minimizes a cost function in real time. However, the computationally intensive nature of this approach limits its applicability on embedded hardware, where resources and time margins are very limited.

This thesis, conducted at the **INRIA** research center in collaboration with the **Rainbow** and **TARAN** teams, examines precisely these scenarios and aims to explore the implementation of an **MPPI controller** on a quadrotor drone equipped with the **NVIDIA Jetson Orin** platform via **TeleKyb3**, and to evaluate the numerical precision that can be used to accelerate the controller on dedicated hardware. The motivation stems from the desire to combine two areas of great scientific interest: advanced drone control and performance optimization on accelerated hardware. The idea of being able to develop a truly operational control system, capable of running in real time on a real device, represents a fascinating challenge and a concrete contribution toward the integration of control theory and practical implementation.

During the project, the work involved both simulations in the **Gazebo** environment, with real-time and non-real-time tests, and experiments on a real drone controlled via a **VICON** motion capture system and integrated into the **TeleKyb3** framework. The main technologies used include **CUDA** for GPU parallelization and **JAX** for efficient management of numerical operations.

1.2 Problem Statement

The implementation of optimization-based control algorithms, such as **Model Predictive Path Integral (MPPI)**, poses significant computational challenges. These algorithms rely on the simultaneous generation and evaluation of numerous candidate trajectories, with the goal of selecting the one that minimizes a cost function in real time. Each control cycle therefore requires a large number of simulations, evaluations, and command updates, all executed within a matter of milliseconds. On systems with limited resources, such as the **Jetson** modules mounted on drones, this results in a computational load that often approaches the limit of the hardware's capabilities.

To make control truly executable in real time, it is necessary to maximize the potential of accelerated hardware by optimizing each phase of the computational process. In this context, one of the most critical aspects is the numerical precision used in the calculations. Floating-point representation directly impacts both processing speed and the numerical stability of the results. Using reduced precision speeds up operations, reducing execution times and improving parallelization capacity on **GPUs** and/or **FPGAs**. However, lower precision can introduce quantization and stability errors, compromising system control.

To the best of our knowledge, no prior work has systematically investigated how numerical precision influences MPPI performance when implemented on embedded hardware for real-time aerial control.

The central issue addressed in this thesis therefore concerns the trade-off between numerical precision and control performance in the context of real-time execution of an **MPPI controller** on a quadrotor drone. Specifically, the goal is to determine whether using reduced precision can offer sufficient computational speed to meet control requirements while maintaining an acceptable level of accuracy in trajectory tracking.

The implementation leverages the parallel computing capabilities offered by

the **NVIDIA Jetson Orin GPU**, using **CUDA** for simultaneous trajectory generation and **JAX** for efficient management of numerical operations and automatic differentiation.

The experiments focus on two main evaluation metrics:

- **Computation time per control iteration**, which determines real-time feasibility and the maximum frequency achievable by the system.
- **Tracking error**, which measures the drone's ability to accurately follow reference trajectories.

The results show that calculations performed with `float16` precision are significantly faster than those with `float32`, while maintaining comparable control performance. This suggests that using reduced precision may represent an effective solution for improving computational efficiency without compromising control quality. The findings also provide an opportunity to evaluate future implementations on **FPGAs**, where low precision arithmetic could be exploited to achieve higher operating frequencies and lower latencies, further improving the responsiveness of real-time control systems and halving the controller's computational time.

1.3 Objectives

The objectives of this thesis stem directly from the motivations and the identified research problem. The work aims to thoroughly analyze how numerical precision affects the real-time performance of the **Model Predictive Path Integral (MPPI)** controller when running on embedded hardware intended for quadrotor drone control.

Specifically, the research is structured around four main objectives:

1. **Develop and integrate an MPPI controller into the TeleKyb3 framework.**
   The algorithm was implemented using **JAX** and **CUDA** to parallelize computations on the **GPU**.
2. **Conduct a systematic analysis of numerical precision.**
   Comparative experiments were conducted between `float32` and `float16` representations on the **NVIDIA Jetson** platform, with the aim of evaluating the effects of arithmetic precision on computation times, tracking accuracy, and control stability.

3. **Verify real-time feasibility on embedded hardware.**
   We analyzed whether the speed increase achieved with reduced precision allows achieving high control frequencies while maintaining satisfactory dynamic performance and control quality consistent with the required specifications.

4. **Define guidelines for future FPGA implementations.**
   Based on the experimental results, we formulated considerations and recommendations for choosing numerical formats for the design of dedicated hardware.

In summary, the overall goal is to contribute to the development of **hardware-aware controllers**, i.e., controllers designed to consciously exploit the characteristics and limitations of the hardware on which they run. This approach aims to bridge the gap between control theory, embedded computing, and real-time implementation, providing a concrete basis for the creation of more efficient and responsive autonomous systems.

Relation to Sustainable Development Goals. Although this thesis focuses on a technical problem in real-time control and embedded numerical optimization, its outcomes are directly aligned with the **United Nations Sustainable Development Goal 9 (Industry, Innovation and Infrastructure)**. The work advances hardware-aware control strategies and computationally efficient algorithms that enable reliable autonomous systems to operate on resource-constrained platforms. Demonstrating that numerical precision can be reduced without degrading control quality provides concrete guidance for designing lighter, faster, and more efficient embedded controllers. These contributions support SDG 9 by promoting technological innovation and improving the efficiency of advanced robotic systems, which are increasingly used in industrial inspection, infrastructure monitoring, and other mission-critical applications where real-time performance and energy constraints are key considerations.

1.4 Thesis Structure

The thesis is divided into five main chapters, each dedicated to a specific aspect of the research, from the initial motivation to the conclusions and future perspectives.

Chapter 1 – Introduction presents the general context, the motivations that guided the project's development, the research problem addressed, and the objectives pursued.

Chapter 2 – Background and Related Work delves into the theoretical and technological foundations necessary to understand the work. It introduces the principles of **MPPI** control, drone dynamic modeling, and the operation of the **TeleKyb3** framework, providing a review of the most relevant contributions in the literature related to this work.

Chapter 3 – Methodology describes the controller design and implementation process in detail. It illustrates the mathematical formulation of the algorithm from a coding perspective, the methods of integration into the **TeleKyb3** framework, the development of the simulation environment, and the experimental methodology adopted to analyze the impact of numerical precision.

Chapter 4 – Experiments and Results reports the experimental evaluation of the proposed approach. The hardware and software configurations used are described, followed by a quantitative analysis of the results obtained in terms of computation time and tracking accuracy. The trade-off between computational speed and control quality is also discussed, with reference to the system's real-time implementation.

Finally, **Chapter 5 – Conclusions and Future Works** summarizes the main results achieved, highlights the original contributions of the work, and discusses the limitations encountered. The chapter concludes with some suggestions for future research, particularly regarding hardware-accelerated implementations dedicated to the **MPPI** controller.

Chapter 2

Background and Related Work

2.1 MPPI Theory

The MPPI (*Model Predictive Path Integral*) control system is a predictive control framework that applies the principles of PIC (*Path Integral Control*) and numerical approximation through MCS (*Monte Carlo Sampling*), leveraging the information-theoretic foundations of optimal control [1, 2, 3]. This approach allows direct handling of nonlinear systems and nonconvex cost functions, overcoming the limitations of classical controllers such as PID (*Proportional–Integral–Derivative*), LQR (*Linear Quadratic Regulator*), and MPC (*Model Predictive Control*).

In the optimal control problem, the objective is to compute a sequence of commands $u_{0:T}$ that minimizes a cost functional:

$$J = \mathbb{E}\left[\phi(x_T) + \int_0^T q(x_t, u_t)\, dt\right], \tag{2.1}$$

where $\phi(x_T)$ represents the terminal cost, and $q(x_t, u_t)$ is the instantaneous cost associated with the state and control. The HJB (*Hamilton–Jacobi–Bellman*) equation provides the formal solution to this problem by computing the optimal policy. However, it is intractable for most nonlinear or high-dimensional systems.

PIC is used to address this issue. This approach reformulates the problem using probabilistic methods, where the solution is expressed as a weighted average over a set of stochastic trajectories. In this context, MCS techniques can

be used to approximate the expected cost and the optimal control, enabling implementations based on the parallel sampling of multiple trajectories.

Given this, MPPI applies a sampling-based strategy: at each iteration, multiple trajectories are generated through random perturbations of the nominal control sequence. Each trajectory is evaluated according to its cost function and weighted based on its associated cost value. The optimal control is computed as a weighted average of the perturbations corresponding to the lowest-cost trajectories. By applying this method, the controller does not need to compute gradients or solve complex optimization problems, but simply calculates a weighted average of several control inputs.

From a theoretical point of view, MPPI can also be interpreted through the lens of information theory. In this formulation, the optimal control problem can be expressed as the minimization of the expected cost plus a regularization term that measures the information divergence between the controlled and uncontrolled trajectory distributions:

$$\mathcal{F} = \mathbb{E}_{\mathbb{Q}}[S(\tau)] + \lambda \, \mathrm{KL}(\mathbb{Q} \| \mathbb{P}), \tag{2.2}$$

where $\mathbb{P}$ and $\mathbb{Q}$ denote, respectively, the probability distributions of the uncontrolled and controlled trajectories, $S(\tau)$ is the trajectory cost, and $\mathrm{KL}(\mathbb{Q}\|\mathbb{P})$ is the Kullback–Leibler divergence between them [1].

Minimizing this functional leads to the optimal distribution of trajectories:

$$\frac{d\mathbb{Q}^*}{d\mathbb{P}} \propto \exp\left(-\frac{1}{\lambda} S(\tau)\right), \tag{2.3}$$

which shows that lower-cost trajectories are exponentially more likely under the optimal policy. This formulation provides a theoretical justification for the exponential weighting used in MPPI and clarifies the role of the temperature parameter λ, which balances exploration (through stochastic perturbations) and exploitation (selection of low-cost trajectories).

This information-theoretic formulation also highlights the flexibility of MPPI in handling nondifferentiable costs and complex constraints while maintaining real-time feasibility through efficient parallelization on modern GPU (*Graphics Processing Unit*) architectures.

In the following sections, the theoretical foundations of the two main components are presented: the formulation of the PIC and the numerical approximation through MCS.

2.1.1 Path Integral Control

Path Integral Control (PIC) provides the theoretical foundation of the MPPI method, offering a probabilistic formulation of optimal control that avoids the direct solution of the HJB equation. Originally introduced by Kappen [1, 4], the *path integral control* framework addresses nonlinear stochastic optimal control problems by estimating expectations over system trajectories, for example through MCS (*Monte Carlo Sampling*).

Consider the following dynamics:

$$dx_t = \Big[f(x_t, t) + G(x_t, t)u(x_t, t)\Big]dt + B(x_t, t)d\omega, \tag{2.4}$$

where $x_t = x(t) \in \mathbb{R}^N$ is the system state, $u(x_t, t) \in \mathbb{R}^m$ is the control input, and $d\omega_t \in \mathbb{R}^p$ is a Brownian disturbance. The system is affine in the control input.

Let $\phi(x_T, T)$ denote the terminal cost, $q(x_t, t)$ the running cost, and $R(x_t, t)$ a positive-definite control cost matrix. The value function of this stochastic optimal control problem is defined as:

$$V(x_t, t) = \min_u \mathbb{E}_Q \left[\phi(x_T, T) + \int_t^T \left(q(x_t, t) + \tfrac{1}{2}u(x_t, t)^\top R(x_t, t)u(x_t, t)\right)\right], \tag{2.5}$$

where $\mathbb{E}_Q[\cdot]$ denotes the expectation with respect to controlled trajectories, i.e., the stochastic process defined in (2.4).

The associated HJB equation is:

$$-\partial_t V = \min_u \left[(f + Gu)^\top \nabla_x V + \frac{1}{2}\mathrm{tr}(BB^\top \nabla_{xx} V) + q + \frac{1}{2}u^\top Ru\right], \tag{2.6}$$

with boundary condition $V(x_T, T) = \phi(x_T, T)$. The minimization is convex with respect to the control inputs u_t, so the optimum is obtained by setting the gradient with respect to u equal to zero:

$$u^* = -R^{-1}G^\top \nabla_x V. \tag{2.7}$$

Substituting (2.7) into (2.6) yields:

$$-\partial_t V = q + f^\top \nabla_x V - \frac{1}{2}\nabla_x V^\top GR^{-1}G^\top \nabla_x V + \frac{1}{2}\mathrm{tr}(BB^\top \nabla_{xx} V). \tag{2.8}$$

Since this nonlinear PDE is generally intractable, a desirability function $\psi(x, t)$ is introduced as:

$$V(x,t) = -\lambda \log(\psi(x,t)), \tag{2.9}$$

where λ is a positive constant. Substituting (2.9) into (2.8) and simplifying gives:

$$\partial_t \psi = \tfrac{1}{\lambda} q\psi - f^\top \nabla_x \psi + \tfrac{1}{2} \nabla_x^\top \psi G R^{-1} G^\top \nabla_x \psi - \tfrac{1}{2} \mathrm{tr}(BB^\top \nabla_x \psi x) + \tfrac{1}{2\psi} \nabla_x^\top \psi BB^\top \nabla_x \psi. \tag{2.10}$$

The quadratic terms in ψ cancel if and only if $BB^\top = \lambda G R^{-1} G^\top$, which constrains the choice of the matrix $\mathbf{R}$ but leads to a physically meaningful control cost structure. With suitable $\mathbf{R}$ and λ, the HJB equation becomes linear:

$$\partial_t \psi = \frac{1}{\lambda} q\psi - f^\top \nabla_x \psi - \frac{1}{2} \mathrm{tr}\left(BB^\top \nabla_x \psi x\right). \tag{2.11}$$

Applying the Feynman–Kac formula yields the solution:

$$\psi(x_t 0, t) = \mathbb{E}_{\mathbb{P}} \left[\exp\left(-\frac{1}{\lambda} S(\tau) \right) \right], \tag{2.12}$$

where

$$S(\tau) = \phi(x_T) + \int_t 0^T q(x_s, s)\, ds \tag{2.13}$$

is the cumulative cost (state-dependent cost-to-go) along the trajectory τ. Here, $\mathbb{E}_{\mathbb{P}}[\cdot]$ denotes the expectation over uncontrolled trajectories governed by:

$$dx = f(x_t, t)\, dt + B(x_t, t)\, d\omega. \tag{2.14}$$

The optimal control can then be expressed as:

$$u^* = \lambda R^{-1} G^\top \frac{\nabla_x \psi}{\psi}. \tag{2.15}$$

If the dynamics can be decomposed into directly and indirectly actuated components, the matrices can be written as:

$$G(x_t, t) = \begin{pmatrix} 0 \\ G_c(x_t, t) \end{pmatrix}, \quad B(x_t, t) = \begin{pmatrix} 0 \\ B_c(x_t, t) \end{pmatrix}. \tag{2.16}$$

In this case, the gradient can be computed analytically [1], and the optimal control takes the form:

$$u^*\, dt = R^{-1} G_c^\top (G_c R^{-1} G_c^\top)^{-1} B_c \, \frac{\mathbb{E}_{\mathbb{P}} \left[\exp\left(-\frac{1}{\lambda} S(\tau) \right) d\omega_t \right]}{\mathbb{E}_{\mathbb{P}} \left[\exp\left(-\frac{1}{\lambda} S(\tau) \right) \right]}. \tag{2.17}$$

If the dynamics cannot be easily decomposed in this form, a suitable transformation can be introduced to obtain an equivalent structure. For systems that are not affine in the control, an affine approximation can be obtained by linearizing the dynamics around a previously estimated sequence of optimal controls [5, 2].

2.1.2 Monte Carlo Sampling

As introduced in Equation (2.9), the expectation in the path integral formulation must be computed with respect to the uncontrolled dynamics of the system. Since the exact evaluation of this expectation is generally intractable for nonlinear or high-dimensional systems, a numerical approximation through *MCS* is employed [1]. In this method, a large number of trajectories are generated by simulating the system under random disturbances, and the expectation is estimated as the weighted average over these trajectories. The accuracy of the estimation improves with the number of samples, but at the cost of increased computational effort, which makes efficient sampling strategies essential in practice.

Figure 2.1 illustrates a classical example from [1] of a particle moving with a constant horizontal velocity, while its vertical position evolves according to the following stochastic dynamics:

$$dx = u\,dt + d\omega, \tag{2.18}$$

where u is the control input and $d\omega$ represents a Brownian perturbation. In this scenario, the cost is quadratic around zero at the final time $t = 2$, and it becomes infinite when the particle collides with the lateral boundaries (depicted in blue). This configuration, often referred to as the *double-slit problem*, provides an intuitive visualization of how stochastic sampling behaves in the presence of hard constraints.

The figure clearly demonstrates the inefficiency of performing sampling directly from the uncontrolled dynamics. Most sampled trajectories violate the boundary conditions, leading to an infinite cost and thus providing no useful contribution to the estimation of the value function. Only a small fraction of trajectories, those that successfully pass through the slits, meaningfully contribute to the expected cost-to-go estimate. As a result, the computational cost required to obtain an accurate estimation becomes prohibitively high, especially as the dimensionality of the system increases.

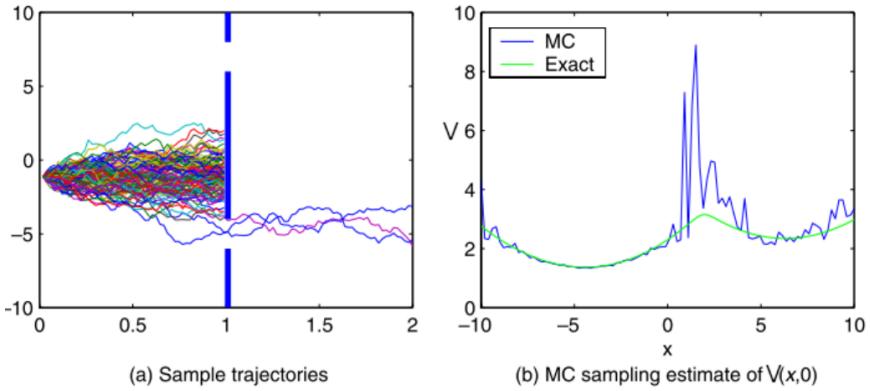


Figure 2.1: Example of Monte Carlo sampling of $V(x, t = 0)$ for the double-slit problem. (a) Sampled trajectories with initial point x used to estimate $V(x, t)$. Most trajectories encounter an infinite cost due to collisions with the walls. (b) Monte Carlo estimation of $V(x, t = 0)$ using $N = 100{,}000$ trajectories for each x. Figure adapted from [1].

This phenomenon is known as the *curse of dimensionality*, which severely limits the scalability of naive MCS methods.

Several strategies have been proposed in the literature to improve the efficiency of sampling [2, 6, 7]. These include *importance sampling* techniques, where trajectories are generated from a biased distribution that favors regions of the state space with lower cost, and their contribution is later reweighted to maintain unbiasedness. Such techniques reduce the variance of the estimator and concentrate computational resources on more relevant trajectories. Another class of improvements involves adaptive or iterative sampling, in which the nominal control sequence is updated based on the statistics of previously sampled trajectories, progressively steering the sampling distribution toward the optimal policy.

More recent approaches have focused on MPC-based formulations, among which the MPPI control algorithm has gained particular attention. In MPPI, at each control step, the algorithm generates samples around a nominal control sequence defined by the previously computed optimal control and shifts this sequence forward in time. This allows for an adaptive, receding-horizon implementation that effectively combines MCS estimation with real-time predictive optimization. By continuously updating the nominal

control and reusing information from past samples, MPPI achieves a balance between exploration and exploitation, maintaining real-time performance even in complex and nonlinear dynamical systems.

2.2 Drone Dynamics

The dynamic model used for simulation and real-world control through MPPI is that of a *quadrotor* [8, 9, 10]. The *quadrotor* is a platform consisting of four electric rotors arranged in a symmetric cross configuration. It is widely used in research due to its mechanical simplicity, high maneuverability, and versatility in control applications. The dynamics of the *quadrotor* exhibit a strongly nonlinear and coupled behavior, since the translational and rotational motions are closely interdependent [11]. The thrust produced by the rotors not only determines the linear acceleration but also generates rotational torques around the three principal body axes.

The system state is defined by the vector:

$$x = [\,\mathbf{p},\, \mathbf{q},\, \mathbf{v},\, \boldsymbol{\omega}\,],$$

where:

- $\mathbf{p} = [x,\, y,\, z]^\top$ is the position of the center of mass in the world frame;
- $\mathbf{q} = [q_w,\, q_x,\, q_y,\, q_z]^\top$ represents the orientation expressed in terms of unit quaternions;
- $\mathbf{v} = [v_x,\, v_y,\, v_z]^\top$ is the linear velocity of the center of mass in the world frame;
- $\boldsymbol{\omega} = [\omega_x,\, \omega_y,\, \omega_z]^\top$ is the angular velocity of the body expressed in the body frame.

The evolution of the system state is described by the following set of

differential equations:

$$\begin{cases} \dot{\mathbf{p}} = \mathbf{v}, \\ \dot{\mathbf{q}} = \dfrac{1}{2}\,\mathbf{Q}(\mathbf{q}) \odot \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix}, \\ \dot{\mathbf{v}} = \dfrac{1}{m}\,\mathbf{R}(\mathbf{q}) \begin{bmatrix} 0 \\ F_t \end{bmatrix} + \mathbf{g}, \\ \dot{\boldsymbol{\omega}} = \mathbf{J}^{-1}\left(\boldsymbol{\tau} - \boldsymbol{\omega} \times \mathbf{J}\boldsymbol{\omega}\right), \end{cases} \tag{2.19}$$

where:

- m is the mass of the drone;
- $\mathbf{g} = [0,\, 0,\, -g]^\top$ is the gravity acceleration vector;
- $\mathbf{J}$ is the inertia matrix of the rigid body, which is generally diagonal for symmetric configurations;
- $\mathbf{R}(\mathbf{q})$ is the rotation matrix derived from the quaternion $\mathbf{q}$;
- $\mathbf{Q}(\mathbf{q})$ represents the quaternion product operator, used to update the orientation over time.

From a control perspective, in these equations the system inputs are the thrust and torque actions generated by the motors, collected in the vector:

$$u = [\,\boldsymbol{\tau},\, F_t\,],$$

where $\boldsymbol{\tau} = [\tau_x,\, \tau_y,\, \tau_z]^\top$ represents the torques acting around the roll, pitch, and yaw axes, while F_t is the total thrust force along the vertical body axis. In the case of a *quadrotor*, these quantities are directly derived from the rotational speeds of the four rotors:

$$F_t = C_f \sum_{i=1}^{4} \omega_i^2, \qquad \boldsymbol{\tau} = \begin{bmatrix} lC_f(\omega_2^2 - \omega_4^2) \\ lC_f(\omega_3^2 - \omega_1^2) \\ C_\tau(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \end{bmatrix},$$

where l is the distance between the drone's center and each motor, C_f is the aerodynamic thrust coefficient, and C_τ denotes the aerodynamic torque

coefficient [9, 8]. Each rotor thus produces a thrust $F_i = C_f \omega_i^2$ and a reactive torque proportional to $C_\tau \omega_i^2$.

The model described above allows the complete dynamics of the drone to be accurately simulated, taking into account the coupling between rotational and translational motion. The equations (2.19) are valid under the rigid body assumption, neglecting secondary dynamic effects such as air resistance, turbulence, or variations in payload during flight [10]. These assumptions, however, are acceptable for control purposes, since they offer a good balance between accuracy and computational simplicity.

A particularly advantageous aspect of this formulation is the use of quaternions to represent the system orientation. Unlike Euler angles, quaternions overcome several issues such as singularities (*gimbal lock*) and allow stable numerical integration over time [12]. This is a fundamental requirement for the proper functioning of the MPPI controller in real time. Moreover, the compact structure of the equations allows efficient implementation on parallel architectures such as GPU, enabling the generation and simultaneous simulation of thousands of trajectories in very short computation times.

In conclusion, the dynamic model reported in (2.19) represents an effective compromise between physical realism and computational simplicity. It provides the mathematical foundation on which the MPPI controller builds its predictive evaluation, allowing the stochastic and parallel assessment of drone trajectories as a function of possible control input sequences.

2.3 TeleKyb3 Framework

The aerial system used in this work is based on the *TeleKyb3* software framework [13, 14], a modular open-source platform designed for the control and management of autonomous aerial robots, with particular attention to physical interaction applications. The framework was developed at the *LAAS-CNRS* laboratory in Toulouse as part of the *OpenRobots* ecosystem and represents the evolution of the previous TeleKyb control stacks. The idea behind the project was to create a flexible, easily extendable infrastructure based on the principles of modularity, interoperability, and reusability — all essential aspects for research in aerial robotics.

The framework is currently widely used by the robotics group at *LAAS-CNRS* for a variety of experiments, ranging from manipulation to human–robot physical interaction. A significant example is the work by Corsini et al. [15], who demonstrated the effectiveness of TeleKyb3 in implementing NMPC applied to human–drone handover scenarios. The architecture was explicitly designed to ensure deterministic real time performance, making it suitable for integrating advanced controllers such as the MPPI used in this work.

From an architectural point of view, TeleKyb3 is based on the GenoM3 formalism (*Generator of Modules*), a system that allows defining software components independent of the underlying middleware, with clearly specified interfaces. This enables the system to run on different robotic communication environments, such as *Pocolibs*, without losing compatibility among the modules. Each functionality (state estimation, control, perception) is implemented as a separate GenoM3 component, which greatly facilitates incremental development and modular validation.

For this thesis work, TeleKyb3 was chosen mainly for its real time nature and for the ease with which it allows the integration of stochastic predictive controllers such as the MPPI. Thanks to its modular interface, it is possible to replace or update individual functional blocks without modifying the overall structure of the control system.

2.3.1 Main Components

The control stack of TeleKyb3 is composed of several GenoM3 modules, each with a specific function. The main ones are:

- **rotorcraft-genom3**: handles the low-level interface with flight controllers, managing motors, PWM signals, and rotation speed (RPM);
- **nhfc-genom3**: a near-hovering flight controller for quadrotors, implemented using a cascaded PID-based control structure;
- **uavatt-genom3** and **uavpos-genom3**: dedicated respectively to attitude and position control for fully actuated UAVs;
- **maneuver-genom3**: manages trajectory generation and waypoint execution using polynomial interpolations to achieve smooth and continuous motions.

All these components can be operated independently and communicate through the deterministic channels provided by the *Pocolibs* middleware, which ensures low latency and predictable timing behavior.

2.3.2 Perception and Estimation Modules

In addition to the core control modules, TeleKyb3 integrates several components dedicated to perception and state estimation, which are fundamental for any closed-loop control system. Among them:

- **pom-genom3**: implements an UKF for sensor fusion, combining data from IMU, GPS, and motion capture systems;
- **optitrack-genom3** and **realsense-genom3**: provide interfaces for the *Optitrack* motion capture system and *Intel Realsense* depth sensors, respectively;
- **gps-genom3**: handles communication with GPS receivers (U-blox, Novatel, Tersus), ensuring accurate global localization.

The integration of these modules guarantees high accuracy in state estimation and effective synchronization between sensors and controller. This architecture is particularly relevant for implementing robust predictive control while maintaining response times compatible with real time operation.

2.3.3 UAVAtt Control Module

The *uavatt-genom3* module handles the drone's attitude control using a geometric approach formulated on the Lie group SE(3), as described in [16]. This type of controller allows the orientation and angular velocity of the vehicle to be regulated stably, avoiding the typical problems associated with Euler angle representation (singularities, *gimbal lock*).

From a practical perspective, the module takes the current drone state and compares it with the desired state generated by higher level modules (for instance, *uavpos-genom3* or *maneuver-genom3*). The desired state includes the reference quaternion $\mathbf{q}_d$, the angular velocity $\boldsymbol{\omega}_d$, and the total force $\mathbf{f}_d$ to be applied along the vertical direction of the body.

The orientation error is computed from the current rotation matrix $\mathbf{R}$ and the desired one $\mathbf{R}_d$ as:

$$\mathbf{e}_R = \frac{1}{2}\begin{bmatrix}(\mathbf{R}_d^\top\mathbf{R} - \mathbf{R}^\top\mathbf{R}_d)_{32}\\(\mathbf{R}_d^\top\mathbf{R} - \mathbf{R}^\top\mathbf{R}_d)_{13}\\(\mathbf{R}_d^\top\mathbf{R} - \mathbf{R}^\top\mathbf{R}_d)_{21}\end{bmatrix}, \tag{2.20}$$

while the angular velocity error, expressed in the body frame, is given by:

$$\mathbf{e}_\omega = \boldsymbol{\omega} - \mathbf{R}^\top\mathbf{R}_d\boldsymbol{\omega}_d. \tag{2.21}$$

The controller then computes the body-frame control torque $\boldsymbol{\tau}$ as a proportional–derivative combination:

$$\boldsymbol{\tau} = -\mathbf{K}_R\mathbf{e}_R - \mathbf{K}_\omega\mathbf{e}_\omega, \tag{2.22}$$

where $\mathbf{K}_R$ and $\mathbf{K}_\omega$ are diagonal matrices containing the proportional and derivative gains for the three axes. In the code, these gains are defined as:

$$\mathbf{K}_R = \mathrm{diag}(K_{q_{xy}}, K_{q_{xy}}, K_{q_z}), \qquad \mathbf{K}_\omega = \mathrm{diag}(K_{w_{xy}}, K_{w_{xy}}, K_{w_z}).$$

The resulting control action is combined with the total force $\mathbf{f}_d$ to form the *wrench* vector $\mathbf{w} = [\mathbf{f}_d,\, \boldsymbol{\tau}]^\top$, representing the overall force and torque required on the rigid body. The vector $\mathbf{w}$ is then mapped into the propeller space through the allocation matrix $\mathbf{G}^{-1}$, which distributes the contributions to each motor:

$$\boldsymbol{\omega}_p^2 = \mathbf{G}^{-1}\mathbf{w}, \tag{2.23}$$

where $\boldsymbol{\omega}_p$ denotes the propeller angular velocities.

Since the motors have physical limits on admissible speeds $(\omega_{\min},\, \omega_{\max})$, a *wrench saturation* procedure is applied. This procedure is implemented as a QP problem solved using the *ProxQP* library [17], which minimizes the deviation from the ideal wrench while respecting mechanical constraints:

$$\min_{\mathbf{x}} \sum_i k_i(x_i - 1)^2 \quad \text{s.t.} \quad \omega_{\min}^2 \le \mathbf{G}^{-1}\mathbf{w} \le \omega_{\max}^2. \tag{2.24}$$

The result provides a balance between control authority and actuator constraint compliance, improving stability even when the motors reach their limits.

To ensure robustness, the module continuously monitors the accuracy of attitude and angular velocity estimates. In the event of large deviations (for example, due to sensor noise or temporary loss of localization), the controller automatically switches to an emergency mode, maintaining the most recent desired orientation until reliable data are recovered.

Overall, the *uavatt-genom3* module provides precise, robust, and computationally efficient attitude control, natively integrated within the *TeleKyb3* framework. The modular structure based on GenoM3 also allows the controller to be replaced or extended with alternative logics without altering the communication pipeline or actuator management.

2.3.4 Summary TeleKyb3

In conclusion, *TeleKyb3* offers a complete and reliable infrastructure for managing autonomous UAVs, providing advanced tools for control, estimation, and perception. Thanks to its modular architecture and compatibility with multiple middleware systems, it represents an ideal environment for developing and testing advanced predictive controllers such as MPPI, which is employed in this work.

Resources:

- Project repository: `https://git.openrobots.org/projects/telekyb3`
- Official documentation: `https://git.openrobots.org/projects/telekyb3/pages/index`

2.4 Hardware Acceleration and FPGA Architecture Design

In recent years, *hardware acceleration* techniques have become increasingly important in the development of advanced control systems, particularly in applications that require high computational capability and extremely low response times. The term *hardware acceleration* refers to the use of dedicated hardware devices to perform the most computationally demanding operations in parallel, thereby reducing overall latency compared to purely software-based solutions. Unlike traditional CPUs, which execute instructions

sequentially, and GPUs, which provide parallelism but follow a general-purpose architecture, FPGAs (*Field Programmable Gate Arrays*) allow the design of customized logical circuits specifically optimized for the target application.

An FPGA consists of a network of reconfigurable logic blocks and a programmable interconnection structure. This makes it possible to implement hardware architectures capable of executing a large number of operations simultaneously. Thanks to this property, FPGAs are particularly suitable for real-time control applications, where it is essential to minimize the latency between sensor measurements and the application of control commands.

Figure 2.2 shows the general architecture of an FPGA–host control system. In this setup, sensors acquire the system state and send the data to the host unit, which processes the information and forwards it to the FPGA. The FPGA receives the system states, reference conditions, and nominal control inputs, then computes multiple control candidates in parallel and returns the optimal ones to the host, which finally transmits them to the actuators.

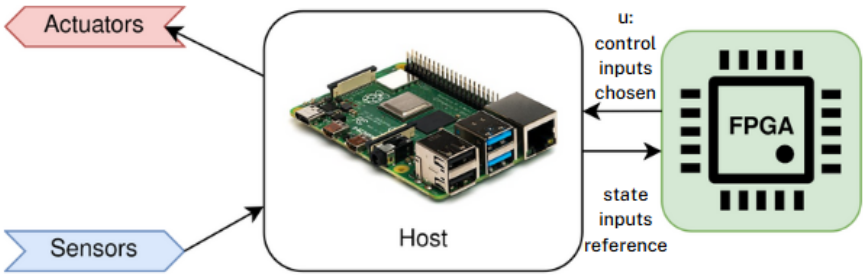


Figure 2.2: Host–FPGA architecture for real-time control. The host communicates with the FPGA, which performs parallel trajectory evaluations and returns the optimal control input.

2.4.1 FPGA-based MPPI Design

In the MPPI control framework, computing the optimal command requires evaluating a large number of stochastic trajectories (*rollouts*), each obtained by integrating the system dynamics over a prediction horizon H and computing the corresponding cumulative cost. This procedure is computationally intensive, especially when the number of rollouts is high. For this reason,

the *Taran Team* developed a dedicated FPGA-based architecture designed to accelerate the MPPI control algorithm by exploiting the possibility of executing multiple simulations in parallel.

The hardware implementation parallelizes the rollouts by employing multiple computation pipelines, each responsible for simulating an independent trajectory. In particular, the FPGA instantiates N parallel pipelines: for example, if the total number of rollouts is 1000 and $N = 200$, the FPGA is invoked five consecutive times, each time processing 200 trajectories. This approach significantly reduces computation time while preserving control accuracy.

Figure 2.3 illustrates the logical structure of the computation pipeline implemented on the FPGA. Each blue block represents an independent simulation pipeline, while the green blocks are responsible for computing the final trajectory cost. The architecture replicates N pipelines, each of which propagates the system dynamics across the prediction horizon H.

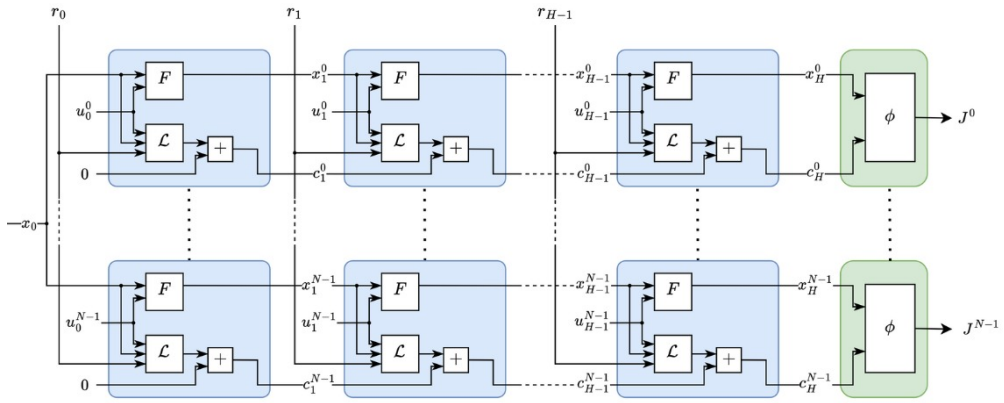


Figure 2.3: FPGA pipeline architecture for the parallel evaluation of MPPI rollouts. Each pipeline computes the dynamics, instantaneous costs, and total trajectory cost.

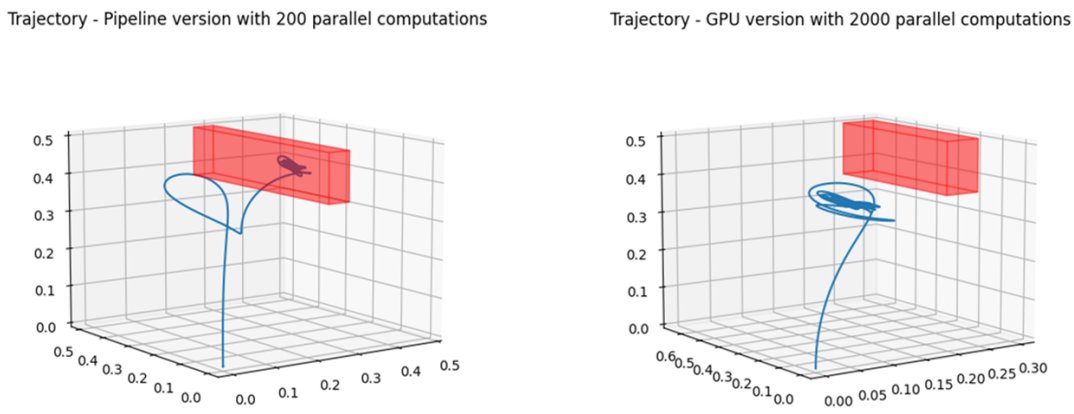
In detail:

- F: represents the system dynamics, which update the state x based on the control inputs u and the random disturbances c;
- L: computes the instantaneous cost associated with the current state;
- H: prediction horizon length;

- N: number of parallel pipeline instantiations;
- x: system state;
- c: random disturbance applied to each trajectory;
- u: control inputs;
- J: total cost of each trajectory computed in output.

The main advantage of this architecture is its ability to execute hundreds of independent trajectories simultaneously, reducing overall computation time and enabling real-time operation even on embedded platforms. The hardware implementation also removes the need for a general-purpose operating system, further minimizing communication latency and timing variability.

The *Taran Team* experimentally validated the FPGA implementation by comparing it to an equivalent GPU-based version. The results demon-



(a) FPGA pipeline – 200 parallel simulations.

(b) GPU version – 2000 parallel simulations.

Figure 2.4: Comparison between the trajectories computed by the MPPI controller executed on FPGA (a) and on GPU (b). The FPGA implementation achieves comparable performance with a smaller number of parallel computations.

strated that, despite using a smaller number of parallel computations, the FPGA achieved comparable performance, with better temporal efficiency and smoother control behavior.

In conclusion, the FPGA implementation of the MPPI controller proved to be both effective and suitable for systems requiring real-time predictive computation. The ability to parallelize rollouts and reduce latency makes this hardware solution a competitive alternative to GPU implementations, offering improved energy efficiency and a fully customizable architecture.

2.5 Related Work

In recent years, stochastic predictive control has attracted increasing interest within the scientific community, particularly in the fields of autonomous robotics and real-time control. The common goal of these studies is to develop strategies capable of combining computational efficiency, stability, and adaptability in complex dynamic scenarios. In this context, MPPI control represents one of the most promising solutions, as it combines the probabilistic formulation of optimal control with the numerical efficiency of MCS techniques.

Theoretical Foundations and Limitations. The origins of this approach trace back to the works of Kappen [1, 4], who introduced the concept of PIC. This formulation reinterpreted the stochastic optimal control problem in probabilistic terms, avoiding the need to directly solve the HJB equation. While Kappen's theoretical framework established the mathematical foundations of path integral control, it assumed continuous-time dynamics and *did not address practical implementation constraints* such as finite sampling, discrete-time integration errors, or limited computational precision. Subsequently, Theodorou et al. [2, 5] proposed a computational version based on MCS, where the optimal solution is approximated through the weighted average of a large number of randomly generated trajectories. However, their work primarily focused on systems with relatively low state dimensionality and did not systematically investigate the scalability to high-frequency control loops typical of aerial platforms. In our work, we extend these foundational results by explicitly analyzing how *numerical precision affects the stability and convergence* of the Monte Carlo approximation when deployed on embedded hardware with constrained arithmetic capabilities.

GPU Implementations and Numerical Precision Gaps. The practical implementation and experimental validation of the method were advanced by Williams et al. [7, 18], who demonstrated its effectiveness in real-time control of autonomous vehicles and drones. Their work showed that it is possible to

sample thousands of trajectories in parallel by exploiting GPU architectures, drastically reducing computation times. The algorithm thus combines the predictive optimization typical of MPC with the flexibility of stochastic sampling, overcoming many of the limitations of classical controllers such as PID and LQR. Despite these advances, Williams' implementation *exclusively used float32 precision* and did not investigate whether reduced-precision arithmetic could offer further efficiency gains without compromising control quality. This represents a significant gap, particularly for embedded systems where memory bandwidth and power consumption are critical constraints. Furthermore, their experiments focused on ground vehicles with relatively slow dynamics (≤ 2 m/s), whereas *our work targets highly dynamic aerial platforms* requiring higher control frequencies (> 100 Hz) and tighter real-time constraints. This motivated our systematic investigation of the *precision-performance trade-off* on the *Jetson Orin* platform—an aspect that has not been previously addressed in the MPPI literature.

Extensions and Application-Specific Challenges. Building on these results, numerous studies have proposed variations and improvements of the approach. Nagabandi et al. [19] integrated neural networks to learn system dynamics models, introducing the concept of MBRL. This combination of machine learning and predictive control opened the way to hybrid controllers capable of autonomously adapting to unforeseen conditions. However, learned models introduce additional sources of numerical error and may amplify the effects of reduced precision—an interaction that remains unexplored. At the same time, Schenk and Schilling [20] explored the use of MPPI on multirotor platforms for pursuit and dynamic planning scenarios, while Franchi et al. [21] demonstrated its applicability to drone–environment physical interaction tasks, maintaining stability even in the presence of actuator saturation or unexpected contact events. While these works successfully extended MPPI to challenging robotic scenarios, none of them *analyzed the sensitivity of the controller to arithmetic precision*, nor did they provide guidelines for selecting numerical formats in hardware-accelerated implementations. Our contribution addresses this gap by conducting controlled experiments comparing float16 and float32 representations, with the explicit goal of informing future FPGA designs.

Hardware Acceleration and FPGA Deployment Considerations. In parallel, research has focused on optimizing computational performance. The first versions of the algorithm were indeed limited by the number of

trajectories that could be evaluated in real time. To address this issue, several authors exploited parallel computation on GPU [22, 23], achieving a significant increase in speed and numerical stability. More recently, the *Taran Team* developed a dedicated FPGA-based architecture capable of executing hundreds of simulations in parallel, with reduced latency and greater energy efficiency compared to GPU implementations. These results demonstrate that MPPI is now fully compatible with the real-time control requirements of robotic systems. However, a critical limitation of existing hardware acceleration work is the *lack of systematic analysis regarding the minimal precision required* to maintain control performance. While the Taran Team's FPGA implementation achieved impressive latency reduction, it did not investigate how aggressively precision could be reduced without introducing instabilities or degrading tracking accuracy. This is a fundamental question for FPGA design, where arithmetic precision can be customized at the hardware level to optimize resource utilization and operating frequency. Unlike previous GPU-based studies that are constrained to standard floating-point formats (float32, float16), *our work provides quantitative guidelines* for selecting numerical representations by directly measuring the impact of precision reduction on both computation time and control quality. This analysis serves as a bridge between software prototyping and dedicated hardware implementation.

Software Frameworks and Integration Challenges. A complementary yet fundamental aspect concerns the software infrastructures that enable the integration of such controllers into complex systems. In this regard, the *TeleKyb3* framework [13, 14], developed at the LAAS-CNRS laboratory, represents one of the most advanced solutions. Based on the GenoM3 formalism, it allows the creation of modular components independent of the underlying middleware, facilitating deterministic communication and compatibility with real-time environments. Thanks to this architecture, predictive controllers such as MPPI can be implemented and tested seamlessly on autonomous aerial platforms, such as the drones used in this work. However, integrating sampling-based controllers like MPPI into existing low-level control architectures requires careful attention to interface design and timing synchronization—aspects that are often underrepresented in the literature. In our implementation, we explicitly addressed these challenges by modifying the system dynamics formulation to match the control inputs expected by the TeleKyb3 attitude controller, ensuring seamless integration without introducing additional latency or requiring invasive modifications to

the existing codebase.

Contribution and Positioning of This Work. In summary, while the literature demonstrates MPPI's effectiveness for real-time control, *three critical gaps remain unaddressed*: (1) the impact of reduced numerical precision on control stability and tracking accuracy has not been systematically studied, particularly for aerial platforms with fast dynamics; (2) existing GPU implementations do not provide actionable guidelines for FPGA deployment, where arithmetic precision can be explicitly tailored during hardware design; and (3) the interaction between precision reduction and controller hyperparameters (prediction horizon, number of samples, temperature) has not been characterized. **This thesis addresses all three gaps** by conducting a comprehensive numerical precision analysis of MPPI on a quadrotor drone using the *Jetson Orin* platform. Specifically, our contributions are: *(i)* a rigorous comparison of float16 and float32 precision across varying controller configurations, *(ii)* quantitative measurements of the precision-performance trade-off in terms of computation time and tracking error, and *(iii)* practical recommendations for future FPGA implementations based on empirical evidence from embedded hardware experiments. Unlike previous work that treated numerical precision as a fixed constraint, we demonstrate that *precision can be strategically reduced* to improve computational efficiency without sacrificing control quality—a finding that has direct implications for the design of next-generation hardware-accelerated control systems.

Chapter 3

Methodology

3.1 MPPI Algorithm Design

3.1.1 Model Predictive Path Integral (MPPI)

The *Model Predictive Path Integral* (MPPI) algorithm [24, 18] represents a practical formulation of the theoretical framework of *Path Integral Control* (PIC), which allows solving stochastic optimal control problems in real time. The main idea combines the probabilistic formulation of PIC with the *receding-horizon* structure typical of *Model Predictive Control* (MPC), resulting in a predictive controller capable of continuously adapting to the evolution of the system state.

The fundamental concept is to iteratively estimate the optimal control through *sampling around a nominal control sequence*. At each control step, a sequence of inputs is considered:

$$U = [u_0, u_1, \dots, u_{T-1}],$$

applied over a prediction horizon of length T. From this sequence, the algorithm generates K perturbed trajectories by adding Gaussian noise to the commands:

$$\delta u_t^{(k)} \sim \mathcal{N}(0, \Sigma),$$

thus obtaining, for each trajectory k, a perturbed control sequence:

$$u_t^{(k)} = u_t + \delta u_t^{(k)}, \qquad t = 0, \dots, T-1.$$

Each trajectory is then propagated forward in time according to the system dynamics, producing a sequence of states $x_0, x_1^{(k)}, \ldots, x_T^{(k)}$ and an associated cumulative cost:

$$S^{(k)} = \phi(x_T^{(k)}) + \sum_{t=0}^{T-1} \left[q(x_t^{(k)}, u_t^{(k)}) + \frac{1}{2} (u_t^{(k)})^\top R \, u_t^{(k)} \right]. \tag{3.1}$$

The specific cost function adopted in this work, as well as the modifications introduced in the system dynamics to match the low-level control inputs, will be discussed in Section 3.2.

The *path integral* principle states that the optimal control can be approximated as a *weighted average of the perturbations* $\delta u_t^{(k)}$, where each weight is proportional to the negative exponential of the corresponding trajectory cost:

$$w_k = \exp\left(-\frac{1}{\lambda} S^{(k)} \right), \tag{3.2}$$

where λ is the "temperature" parameter that regulates the balance between exploration and exploitation.

The updated control is then computed as:

$$u_t^* = u_t + \frac{\sum_{k=1}^{K} w_k \, \delta u_t^{(k)}}{\sum_{k=1}^{K} w_k}. \tag{3.3}$$

The temperature parameter λ plays a crucial role in shaping the behavior of the controller. It regulates how strongly the algorithm favors trajectories with lower cost, acting as a balance between exploration and exploitation. When λ takes small values, the exponential weighting in Equation (3.2) gives dominant importance to the best rollouts, causing the controller to focus almost exclusively on the lowest-cost trajectories. In contrast, a higher value of λ smooths the weight distribution, allowing a broader range of sampled trajectories to contribute to the control update. This leads to more exploratory behavior, which can improve robustness in uncertain or noisy environments, although at the expense of precision. Choosing an appropriate λ is therefore critical: too small a value may cause instability due to aggressive updates, while too large a value can make the control response sluggish. In practice, λ is tuned experimentally to achieve a trade-off between stability, responsiveness, and robustness.

In practice, MPPI is executed in a *receding-horizon* fashion, applying at each control cycle only the first optimized command u_0^*, while the prediction window is shifted forward in time:

$$U \leftarrow [u_1^*, u_2^*, \ldots, u_{T-1}^*, u_T^{\text{new}}].$$

This continuous update allows the controller to react in real time to new state measurements and to dynamically adapt to changes in the system.

The MPPI algorithm can be summarized in the following main steps:

1. Generate K sequences of perturbations $\delta u_t^{(k)} \sim \mathcal{N}(0, \Sigma)$;
2. Simulate the trajectories and compute the total cost $S^{(k)}$;
3. Compute the weights $w_k = \exp(-S^{(k)}/\lambda)$;
4. Update the control sequence according to Equation (3.3);
5. Apply the first control u_0^* and repeat the cycle at the next step.

Thanks to this formulation, MPPI does not require linearization of the system dynamics or gradient-based optimization methods. The approach relies solely on direct simulations and *Monte Carlo* sampling (MCS), making it particularly suitable for complex and highly nonlinear systems, such as autonomous drones and, more generally, dynamic robotic platforms.

To provide a clearer understanding of the operational flow, the next section presents a schematic representation of the algorithm in the form of pseudocode. This step-by-step description helps to visualize the sequence of operations performed at each control cycle and highlights the iterative logic at the core of the MPPI method.

3.1.2 Pseudocode Implementation

To provide a clear and concise overview of the algorithmic workflow, the implementation of the MPPI controller is summarized in Algorithm 1. The pseudocode describes the iterative structure of the control loop, showing how trajectories are sampled, evaluated, and used to update the control command at each iteration. This representation makes it easier to understand the sequence of operations involved in the real-time execution of the algorithm.

At the beginning, the nominal control sequence $U^{\text{nom}} = [u_0, u_1, \ldots, u_{T-1}]$ is initialized with a default value, usually corresponding to a hovering condition for aerial systems. During each control cycle, the current system state $\hat{x}$ is retrieved from the state estimator, and K trajectories (or *rollouts*) are simulated forward in time according to the system dynamics. Each trajectory is perturbed by Gaussian noise sampled from $\mathcal{N}(0, \Sigma)$, and its total cost is computed through the cost function defined in Section 3.1.

Once all trajectories have been evaluated, their associated weights are computed according to the exponential weighting rule described in Equation (3.2). The nominal control sequence is then updated as a weighted average of the applied perturbations, following Equation (3.3). Finally, the first control command u_0^* is applied to the system, and the prediction horizon is shifted forward for the next iteration.

This structure highlights the simplicity and efficiency of the MPPI algorithm. All computations — such as trajectory propagation, cost evaluation, and weight normalization — are easily parallelizable and can be distributed across multiple threads or GPU cores. This characteristic makes MPPI particularly suitable for real-time control on embedded platforms, where computational resources are limited but parallel execution is available.

3.2 Implementation on TeleKyb3

The mathematical formulation presented so far defines the theoretical structure of the MPPI controller, including its sampling strategy and optimization criteria. However, to achieve real-time execution on embedded hardware, this formulation must be carefully adapted to the specific architecture of the experimental framework used in this work. This section presents the integration of the MPPI controller within the TeleKyb3 framework and its connection with the low-level UAVAtt module. The following subsections describe the modifications introduced to the system dynamics, the communication interface between the modules, and the architectural design that enables real-time execution on embedded hardware.

3.2.1 Modified Dynamics

The MPPI controller described in the previous section was originally formulated using the system dynamics presented in Section 2.2. In that model,

Algorithm 1 Model Predictive Path Integral Control

Require: Initial control sequence u_{init}; cost function ComputeCost(x^k, u^k); system dynamics function f_{RK4}
Parameters: Number of rollouts K, time steps N, noise covariance Σ, time step Δt

Initialize nominal control: $u_j^{\text{nom}} \leftarrow u_{\text{init}}$, for $j = 0, \ldots, N-1$
while control task not completed **do**
  Obtain current system state: $\hat{x} \leftarrow$ CurrentStateEstimate()
  for $k = 1, \ldots, K$ **do** ▷ Simulate K rollouts
    Initialize $x_0^k \leftarrow \hat{x}$
    for $j = 0, \ldots, N-1$ **do**
      Sample noise: $\delta u_j^k \sim \mathcal{N}(0, \Sigma)$
      Compute perturbed control: $u_j^k = u_j^{\text{nom}} + \delta u_j^k$
      Propagate dynamics: $x_{j+1}^k = x_j^k + f_{\text{RK4}}(x_j^k, u_j^k, \Delta t)$
    end for
    Evaluate trajectory cost: $S^k =$ ComputeCost(x^k, u^k)
  end for
  Compute weights: $w_k = \exp(-S^k/\lambda)$
  for $j = 0, \ldots, N-1$ **do**
    Update control: $u_j^{\text{nom}} \leftarrow u_j^{\text{nom}} + \frac{\sum_{k=1}^K w_k \delta u_j^k}{\sum_{k=1}^K w_k}$
  end for
  Apply first control: ApplyToSystem(u_0^{nom})
  Shift horizon: $u_{N-1}^{\text{nom}} \leftarrow u_{\text{init}}$
end while

the control inputs were expressed as the total thrust and the body torques applied to the quadrotor:

$$u = \begin{bmatrix} F & \tau_x & \tau_y & \tau_z \end{bmatrix}^\top.$$

However, the low-level UAVAtt controller implemented in the experimental platform does not directly receive torques as input commands. Instead, it expects the total thrust and the desired angular velocity components in the body frame, $\omega_x, \omega_y, \omega_z$, which are then internally converted into torques by the attitude control layer.

To ensure full compatibility between the high-level MPPI policy and the existing control architecture, the system dynamics model was modified

accordingly. The input vector was redefined as:

$$u = \begin{bmatrix} F & \omega_x & \omega_y & \omega_z \end{bmatrix}^\top,$$

and a dedicated conversion function was introduced to compute the equivalent body torques required by the physical dynamics. This function, named `omega_to_torque()`, computes the torque vector corresponding to the desired body rates according to:

$$\tau = J\dot{\omega}_d + \omega \times (J\omega),$$

where J is the inertia matrix of the vehicle, and $\dot{\omega}_d$ is approximated as

$$\dot{\omega}_d = \frac{\omega_d - \omega}{\Delta t},$$

representing the change in angular velocity over the control timestep.

Through this modification, the model used within the MPPI simulation loop produces the same internal dynamics as the physical system governed by the low-level controller. This alignment ensures that the control commands generated by the MPPI algorithm can be applied directly to the UAVAtt module without further conversion or adaptation, maintaining a consistent relationship between simulation and real-world behavior.

From a numerical standpoint, this redefinition of the control inputs also improves the stability and precision of the trajectory propagation step. By avoiding the direct integration of torque inputs—which can amplify numerical errors in the attitude dynamics—the model achieves smoother transitions and better consistency with the real vehicle's response. This aspect is particularly relevant given the main goal of this work: enhancing numerical precision and real-time efficiency in sampling-based predictive control.

In summary, the modification of the dynamics serves a dual purpose: it guarantees hardware-level compatibility with the TeleKyb3 control framework and, at the same time, improves numerical accuracy in the high-frequency simulation loop used by the MPPI algorithm.

This refinement directly contributes to the main objective of this work, namely improving numerical precision and the overall stability of the predictive control loop.

3.2.2 Cost Function Definition

The design of the cost function is a key aspect of the MPPI controller, since it defines the criteria according to which each trajectory is evaluated during the optimization process. In this work, the cost is formulated to guide the quadrotor toward a desired reference state while maintaining smooth and dynamically feasible control actions.

At each time step, the algorithm computes a *running cost* based on the difference between the current state of the drone and the reference trajectory. The system state is represented as

$$x = \begin{bmatrix} p & q & v & \omega \end{bmatrix}^\top,$$

where $p \in \mathbb{R}^3$ denotes the position, $q \in \mathbb{R}^4$ the orientation expressed as a quaternion, $v \in \mathbb{R}^3$ the linear velocity, and $\omega \in \mathbb{R}^3$ the angular velocity.

In general, the reference quantities x_{ref} and u_{ref} can originate from any external source, such as a predefined trajectory, a human-operated command, or another higher-level planner. In the present implementation, the references are computed through a *differential flatness*-based trajectory generator, which provides smooth and dynamically consistent profiles for position, velocity, and orientation. This approach ensures that the reference states are always physically realizable by the quadrotor dynamics, preventing discontinuities and improving numerical stability during trajectory propagation. Moreover, the differential flatness formulation allows the computation of all state and input references directly from a set of flat outputs, resulting in a compact and computationally efficient representation of the desired motion.

In this implementation, the control input vector is defined as

$$u = \begin{bmatrix} F_t & \omega_x & \omega_y & \omega_z \end{bmatrix}^\top,$$

where F_t is the total thrust along the body-frame z axis, and $\omega_x, \omega_y, \omega_z$ are the desired angular velocity components expressed in the body frame. These quantities correspond directly to the control inputs expected by the low-level UAVAtt controller. Within the simulation model, these desired angular velocities are internally converted into body torques through the function `omega_to_torque()`, ensuring that the simulated dynamics remain consistent with the physical response of the real platform.

The instantaneous cost penalizes deviations from the reference state in terms of position, attitude, velocity, and angular velocity. To properly account for the orientation error, the difference between the current and desired quaternions is computed through quaternion algebra:

$$q_{\text{err}} = q^{-1} \otimes q_{\text{ref}},$$

from which the vector part of the quaternion is extracted to represent the rotational discrepancy. The running cost can therefore be expressed as:

$$\begin{aligned} \ell(x, u, x_{\text{ref}}, u_{\text{ref}}) = w_p \|p - p_{\text{ref}}\|^2 + w_q \|q_{\text{err}}\|^2 \\ + w_v \|v - v_{\text{ref}}\|^2 + w_\omega \|\omega - \omega_{\text{ref}}\|^2 \\ + (u - u_{\text{ref}})^\top R (u - u_{\text{ref}}), \end{aligned} \tag{3.4}$$

where w_p, w_q, w_v, w_ω are scalar weights that balance the contribution of each term, and R is a diagonal matrix penalizing large variations in the control inputs.

The cumulative cost of a trajectory is obtained by summing the running cost along the prediction horizon, optionally including a final cost term:

$$S^{(k)} = \phi(x_T^{(k)}) + \sum_{t=0}^{T-1} \ell(x_t^{(k)}, u_t^{(k)}, x_{\text{ref}}, u_{\text{ref}}), \tag{3.5}$$

where $\phi(\cdot)$ denotes the terminal cost, which in this implementation is set to zero since the objective is to achieve steady-state regulation rather than point-to-point trajectory tracking.

Additional constraints can be introduced to account for environmental interactions, such as obstacle avoidance or workspace limitations. In this work, soft constraints were defined to penalize proximity to obstacles within a given radius, ensuring that the sampled trajectories respect feasible flight conditions.

The weighting coefficients of Equation (3.4) were tuned empirically to achieve a good compromise between tracking accuracy, stability, and smoothness of the generated control signals. The exact numerical values used in the experiments are presented in Section 4.1.

Finally, it is important to highlight that the combination of the differential-flatness-based reference generation and the modified dynamics described

in Section 3.2 directly supports the main goal of this thesis — improving numerical precision and stability in real-time predictive control. This integration ensures that the MPPI controller operates coherently with the low-level architecture of the drone, which will be detailed in the following subsection.

3.2.3 Integration with the Low-Level Controller

Within the TeleKyb3 framework, the flight control architecture is organized into several interconnected modules, each responsible for a specific layer of the control hierarchy. Among these components, the `UAVAtt` and `Rotorcraft` modules play a central role in handling attitude stabilization and motor-level control, respectively. The MPPI controller developed in this work was integrated into this structure as a high-level control node, replacing the standard position controller while remaining fully compatible with the existing low-level layers.

In the default configuration of TeleKyb3, the `UAVPos` module generates desired attitude and thrust commands based on a reference trajectory, typically computed through differential flatness. These commands are then passed to the `UAVAtt` module, which converts them into individual rotor speed references through the `Rotorcraft` interface. However, in this implementation, the MPPI controller directly replaces the `UAVPos` block, providing control commands in the form of:

$$u_{\text{input}} = [F_z, \omega_x, \omega_y, \omega_z]^\top,$$

which correspond to the total thrust and the desired angular velocity components expressed in the body frame.

This choice was made to ensure seamless compatibility with the `UAVAtt` module, which expects precisely these inputs to compute the required body torques and generate rotor speed commands. In this configuration, the MPPI controller receives the estimated vehicle state directly from the onboard state estimator and produces the desired control inputs at each iteration. These inputs are then processed by `UAVAtt`, which internally performs the necessary transformations to translate angular velocity and thrust commands into the corresponding motor signals through the `Rotorcraft` component.

The resulting structure, illustrated in Figure 3.1, highlights how the proposed controller integrates into the existing control pipeline without altering the underlying communication mechanisms. By maintaining the

same message interfaces used by TeleKyb3, the implementation preserves full compatibility with the middleware and can be executed in real time on the embedded hardware.

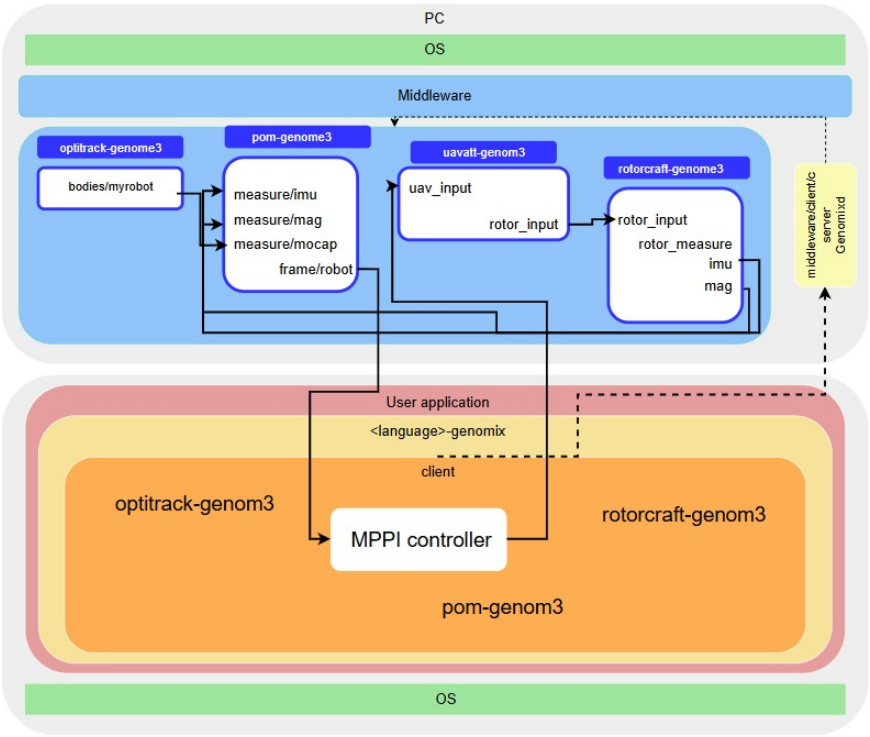


Figure 3.1: Integration of the MPPI controller within the TeleKyb3 architecture. The MPPI replaces the `UAVPos` component, providing desired thrust and angular rates to `UAVAtt`, which computes the corresponding rotor commands through `Rotorcraft`.

This modular integration approach ensures that the MPPI controller can exploit the existing attitude and motor control loops, reducing implementation complexity and avoiding redundancy in software design. Furthermore, by maintaining a direct interface with `UAVAtt`, the controller benefits from precise attitude stabilization and motor feedback, which are essential for ensuring real-time performance and improving the numerical precision of the predictive control computations.

3.3 Drone Simulation Environment

Simulation environments play a fundamental role in the development and validation of control algorithms for aerial robotics. Modern simulators have evolved significantly over the past decade, becoming increasingly accurate and capable of reproducing the complex physics of the real world. They incorporate high-fidelity aerodynamic models, realistic sensor noise, and communication delays, allowing researchers to evaluate the performance of control strategies under conditions that closely resemble those encountered in flight experiments. For these reasons, simulation represents a crucial step between the theoretical formulation of a controller and its deployment on an actual platform, ensuring safety, repeatability, and cost efficiency.

In this work, two complementary simulation environments were employed, each serving a specific purpose within the overall validation process. The first environment, based on **Gazebo** and integrated with the TeleKyb3 framework, was used to evaluate the real-time behavior of the controller within the complete system architecture. The second environment, implemented in **Python** using the **MuJoCo** physics engine, was designed to analyze numerical precision and computational performance directly on the embedded **Jetson Orin** platform. Together, these two setups provided a complete validation framework, enabling both system-level and numerical-level evaluation of the proposed MPPI algorithm.

3.3.1 Simulation in Gazebo with TeleKyb3

The TeleKyb3 framework natively supports the **Gazebo** simulator, providing an environment in which both high-level and low-level control components can be executed in real time. Gazebo offers a physics engine capable of simulating rigid-body dynamics, aerodynamic interactions, and sensor feedback with sufficient accuracy for aerial robotics applications. The integration between TeleKyb3 and Gazebo enables seamless communication between the simulated drone and the software modules through the same middleware used in the real system. As a result, every control component operates exactly as it would on a physical vehicle.

In this setup, the MPPI controller interacts with the low-level UAVAtt module, which converts the received commands (total thrust and desired

angular velocities) into rotor speed references for the ROTORCRAFT component. This structure mirrors the real control pipeline and allows testing the proposed approach in a *software-in-the-loop* configuration, where all timing constraints and communication interfaces are faithfully preserved. The resulting simulation provides a reliable assessment of the controller's stability, responsiveness, and robustness under realistic operating conditions.

Quadrotor Simulation Model. All Gazebo-based experiments in this thesis used an SDF (Simulation Description Format) model that replicates the mechanical structure and inertial properties of the `mkquad5` platform employed in the physical tests. The model represents the drone as a single rigid body with mass 1.3 kg and an inertia matrix derived from the real platform's geometry. Four rotors are attached via revolute joints positioned according to the measured arm layout, with their aerodynamic behavior (thrust and drag) handled by the `mrsim` Gazebo plugin. This plugin also introduces realistic effects such as motor noise and first-order rotor dynamics, which help reproduce the response characteristics observed in flight.

To maintain computational efficiency while preserving physical accuracy, the visual and collision geometries were simplified: the body is represented as a box, and the rotors as cylinders. This simplification ensures stable contact detection and real-time simulation performance without significantly affecting the dynamics relevant to control evaluation. Overall, the SDF configuration provides a reasonable balance between fidelity and computational cost, making it well-suited for testing the MPPI controller within the TeleKyb3 framework before transitioning to actual hardware.

3.3.2 Simulation in MuJoCo for Numerical Precision Analysis

While the Gazebo-based environment offers an accurate and integrated representation of the full control architecture, it is computationally demanding and not fully compatible with the embedded **Jetson Orin** platform used in this work. To overcome these limitations, a second simulation setup was developed in **Python**, leveraging the **MuJoCo** physics engine. MuJoCo provides a highly efficient and differentiable simulator, well suited for high-frequency control and detailed numerical studies.

In this environment, the MPPI controller and the drone dynamics were

implemented entirely in Python, using the same mathematical formulation described in Section 2.2. The simulation ran directly on the Jetson hardware, allowing precise measurement of execution times and the impact of different floating-point precisions (`float32` and `float16`) on control performance. Although the simulation was not strictly real-time, a timing synchronization mechanism was introduced to maintain an update rate comparable to that of the real system. This approach made it possible to emulate the temporal behavior of an onboard control loop while retaining full control over the numerical and computational aspects of the algorithm.

Overall, the MuJoCo-based setup provided a lightweight and flexible testing framework that complemented the Gazebo simulation. It allowed isolating the numerical properties of the controller from the rest of the system and directly observing how different precision levels affected convergence and stability. The results obtained from this environment were essential to validate the core objective of the thesis — improving numerical precision and computational efficiency in real-time predictive control.

3.4 Numerical Precision Analysis for FPGA

A central challenge in implementing real-time predictive control algorithms on hardware accelerators such as FPGAs lies in balancing numerical precision and computational performance. While higher-precision arithmetic improves numerical stability and robustness, it also increases latency, memory usage, and power consumption. Conversely, lower-precision formats enable faster computation and reduced resource utilization but may compromise the accuracy of the controller's internal computations. This trade-off is particularly relevant for sampling-based methods such as MPPI, where thousands of trajectory rollouts are simulated at every control cycle. In this context, identifying the lowest viable precision that preserves control performance is essential for efficient hardware deployment.

The investigation conducted in this work aims to answer the following research question:

> *"What numerical precision should be adopted on FPGA to improve the control performance and computational efficiency of the MPPI algorithmy?"*

To explore this question, a set of experiments was carried out using the **MuJoCo**-based simulation environment described in the previous section. The tests were executed on the **Jetson Orin** platform, which allows controlled manipulation of numerical precision through the JAX computation framework. Although the experiments were not performed directly on an FPGA, they were designed to reproduce the arithmetic constraints of a fixed-point or low-precision implementation. By explicitly controlling the numerical representation of all computations, the behavior observed in these tests provides a realistic prediction of the expected performance once deployed on reconfigurable hardware.

Within JAX, the smallest supported floating-point format is FP16, while `float8` is not available. The default format FP32 was therefore used as a reference to evaluate the effects of precision reduction. In each test, all quantities in the computation pipeline—state variables, control inputs, and constant parameters such as inertia and weight matrices—were consistently represented using either FP16 or FP32. This ensured a uniform numerical environment and avoided hybrid-precision inconsistencies that could otherwise mask the true influence of arithmetic resolution.

Each experiment consisted of running the MPPI controller under varying configurations of two key hyperparameters: the **prediction horizon** (T) and the **number of sampled trajectories** (K). These parameters were selected because they directly affect both the computational load and the statistical properties of the controller. A longer prediction horizon increases accuracy but also amplifies numerical accumulation errors, whereas a larger number of samples improves convergence at the cost of higher computational effort.

N samples	**H=11**	**H=17**	**H=20**	**H=25**	**H=35**
100					
1000					
2000					
3000					
10000					
50000					

The cost function used in these tests remained unchanged across all configurations, since the MPPI controller is highly dependent on task-specific tuning. The same structure and weights defined in Section 3.2 were employed

for all runs, ensuring that any observed differences originated solely from the numerical precision or configuration parameters. The reference trajectory consisted of a smooth circular path centered at the origin, with a radius of 2 m and altitude of 1 m, generated through D F to guarantee continuity of position, velocity, and attitude. This trajectory provided a repeatable and moderately dynamic scenario, suitable for evaluating both tracking accuracy and numerical stability.

For each configuration, the controller was executed ten times to account for the stochastic nature of the sampling process. At every iteration, the computation time required to generate a single control command was recorded, and the mean value across all iterations and repetitions was computed. This approach allowed a robust estimation of the average per-cycle computational time, independent of transient system effects.

Two performance indicators were analyzed:

- **Tracking Error (RMSE)** — representing the root mean square deviation between the reference and actual drone position;
- **Computation Time per Control Step** — representing the mean execution time required for one full MPPI optimization cycle.

Preliminary tuning experiments had established that, with $T = 20$ and $K = 2000$, the cost function achieved a steady tracking error below 10 cm. This threshold was therefore adopted as a practical benchmark: all tested configurations maintaining an average tracking error below this limit were considered acceptable from a control-performance standpoint. By comparing the achieved precision and execution times under FP16 and FP32, it was possible to evaluate how much computational efficiency could be gained without exceeding this accuracy threshold.

The experimental results revealed that the reduced precision format (FP16) did not introduce any noticeable instability or oscillations in the control response. The system maintained consistent behavior across all tested horizons, and the overall tracking accuracy remained comparable to that obtained with FP32. However, the average computation time per iteration decreased significantly, highlighting the potential of low-precision arithmetic for real-time embedded implementation. This finding indicates that, at least for the tested configuration and trajectory, the FP16 precision provides an excellent trade-off between numerical reliability and computational performance.

In conclusion, these tests confirm that lowering numerical precision can be an effective strategy to enhance computational efficiency in sampling-based predictive controllers, provided that the precision remains sufficient to capture the key dynamics of the system. The insights gained from this analysis will directly inform the future FPGA implementation, where arithmetic precision can be explicitly tailored to balance resource utilization and control performance.

Chapter 4

Experiments and Results

4.1 Experimental Setup

This chapter presents the experimental validation of the proposed MPPI controller through three complementary test campaigns, each designed to address a specific aspect of our research question: *Can reduced numerical precision improve computational efficiency for real-time MPPI control without compromising tracking accuracy?* The validation strategy progresses systematically from controlled simulation to physical deployment, culminating in targeted numerical analysis on embedded hardware.

4.1.1 Overview of Test Environments

Our experimental methodology employed three distinct environments, each serving a specific validation purpose while contributing to the overall research objective.

Environment 1: Real-Time Simulation with Gazebo and TeleKyb3. Initial testing took place entirely in simulation using the TeleKyb3 framework integrated with Gazebo. The MPPI controller operated within a physics engine that replicates the aerodynamic behavior of our mkquad5 quadrotor with sufficient fidelity for control development. We executed all standard TeleKyb3 components (UAVAtt attitude control, ROTORCRAFT motor allocation, and state estimation modules) in real time using the same pocolibs middleware employed on the physical platform. The simulation

host was a workstation running Ubuntu 24.04 with genom3 communication infrastructure.

This environment served multiple critical purposes beyond basic algorithm validation. First, it provided a *safe development space* where aggressive control policies could be tested without risk of hardware damage. Second, it established a *controlled baseline* with known dynamics and minimal measurement noise, allowing us to isolate algorithmic behavior from environmental confounds. Third, the deterministic nature of simulation enabled *reproducible experiments* where parameter variations could be studied systematically.

Environment 2: Real-World Flight Tests with mkquad5. After confirming stable behavior in simulation, we transitioned to flight tests using the actual mkquad5 platform. The drone carries an **NVIDIA Jetson Orin Nano** (16 GB) that executes the identical control stack validated in simulation, preserving software consistency between environments. We connected remotely via SSH for system monitoring and parameter adjustment, while a VICON motion capture system provided state feedback at 100 Hz with sub-millimeter accuracy.

The physical experiments served to validate sim-to-real transfer and assess robustness to unmodeled dynamics. Real flight introduces numerous effects absent from simulation: blade flapping dynamics, aerodynamic coupling between rotors, motor response delays, communication latency, and environmental disturbances. We conducted experiments including hovering, waypoint navigation, and continuous circular trajectories, with each scenario repeated multiple times to assess consistency.

Environment 3: MuJoCo-Based Numerical Precision Analysis. While Gazebo cannot run on the on the Jetson Orin hardware and we want to evaluate the impact of the numerical precision of the mppi, We consider a custom simulation using the **MuJoCo** physics engine implemented entirely in Python with JAX for automatic differentiation and CUDA acceleration.

This lightweight environment can run on the Jetson Orin hardware to replicate actual computational constraints. Unlike Gazebo, the MuJoCo–JAX implementation gave us *explicit control over numerical precision* throughout the entire computation pipeline, enabling rigorous comparison of `float32` versus `float16` effects on both computational performance and control quality.

Critically, this environment addresses a gap in previous MPPI literature: while prior work demonstrated GPU acceleration using standard FP32 arithmetic [18, 22], no studies have systematically characterized how aggressively precision can be reduced before control performance degrades.

Summary of Experimental Strategy. Our three test environments were:

1. **Gazebo simulation:** Validates complete control architecture under real-time constraints with known dynamics
2. **Indoor flight tests:** Demonstrates robustness and confirms sim-to-real transfer on physical hardware
3. **MuJoCo simulations:** Provides quantitative data on precision-performance trade-offs for embedded systems

This multi-environment strategy ensures that our conclusions rest on converging evidence rather than single source validation .

4.2 MPPI Performance Evaluation

We structured the performance evaluation to progressively increase dynamic complexity, beginning with equilibrium conditions (hovering) and advancing toward demanding trajectory tracking tasks with obstacle avoidance. Throughout all experiments, we focused on four key metrics: position tracking error, attitude consistency, control signal smoothness, and numerical stability.

4.2.1 Simulation-Based Validation (Gazebo)

The Gazebo experiments served as our primary vehicle for controller characterization under controlled conditions. This environment replicates the real system's feedback loops and dynamics with sufficient fidelity to reveal potential instabilities or tuning issues before committing to physical flights.

Hovering Performance

Hovering serves as the canonical benchmark for multirotor control since it reveals how well the system maintains equilibrium despite sensor noise and

stochastic sampling effects inherent to MPPI. The drone was commanded to hold a fixed position at (0, 0, 1) m for 60 seconds while we monitored position error, attitude stability, and control signal characteristics.

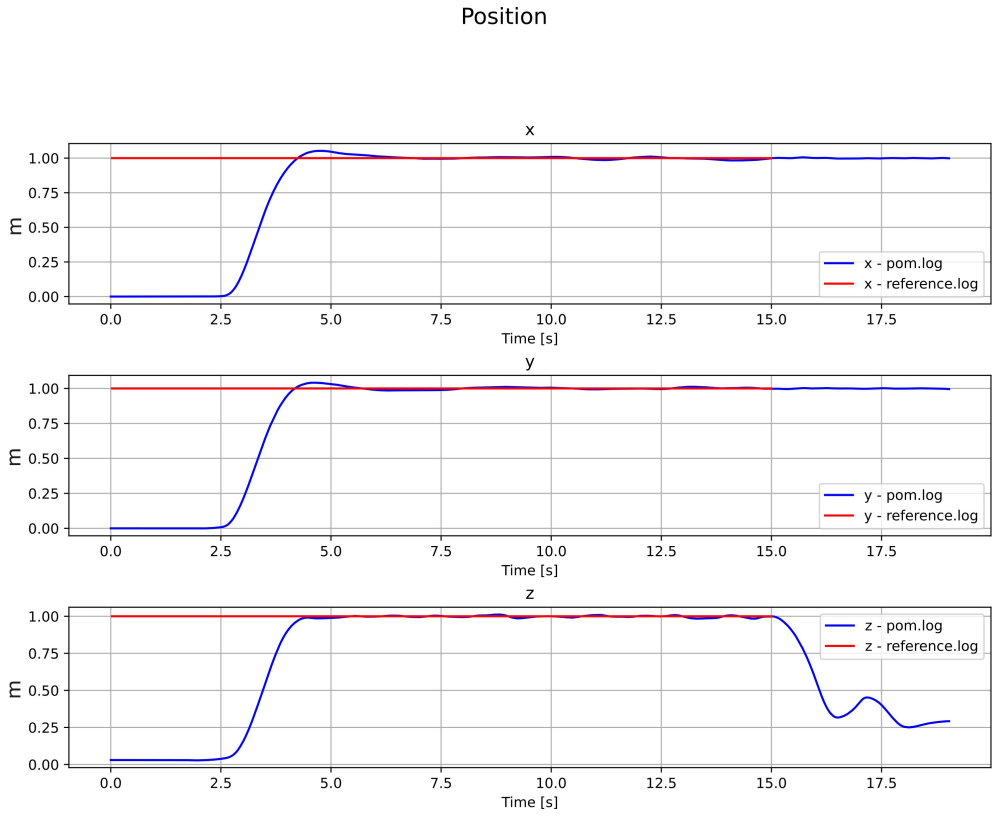


Figure 4.1: Position tracking during simulated hovering. After a short transient phase with a maximum overshoot of approximately 3 cm, the system stabilizes with residual oscillations below 1 cm, demonstrating an effective balance between the exploratory sampling and the exploitation of low-cost trajectories inherent to MPPI.

Figure 4.1 demonstrates rapid convergence to the target position with settling time below 2 seconds. The observed overshoot (approximately 3 cm) reflects MPPI's characteristic exploration behavior. The controller inherently trades slight transient aggression for faster convergence. Once settled, position error remains below 3cm throughout the remainder of the test, with no observable drift or sustained oscillations.

This stability is particularly noteworthy given that MPPI *continuously resamples* control inputs rather than converging to a fixed policy. The absence

of high-frequency oscillations confirms that our temperature parameter λ successfully balances exploration and exploitation, and that the stochastic sampling process does not introduce control noise.

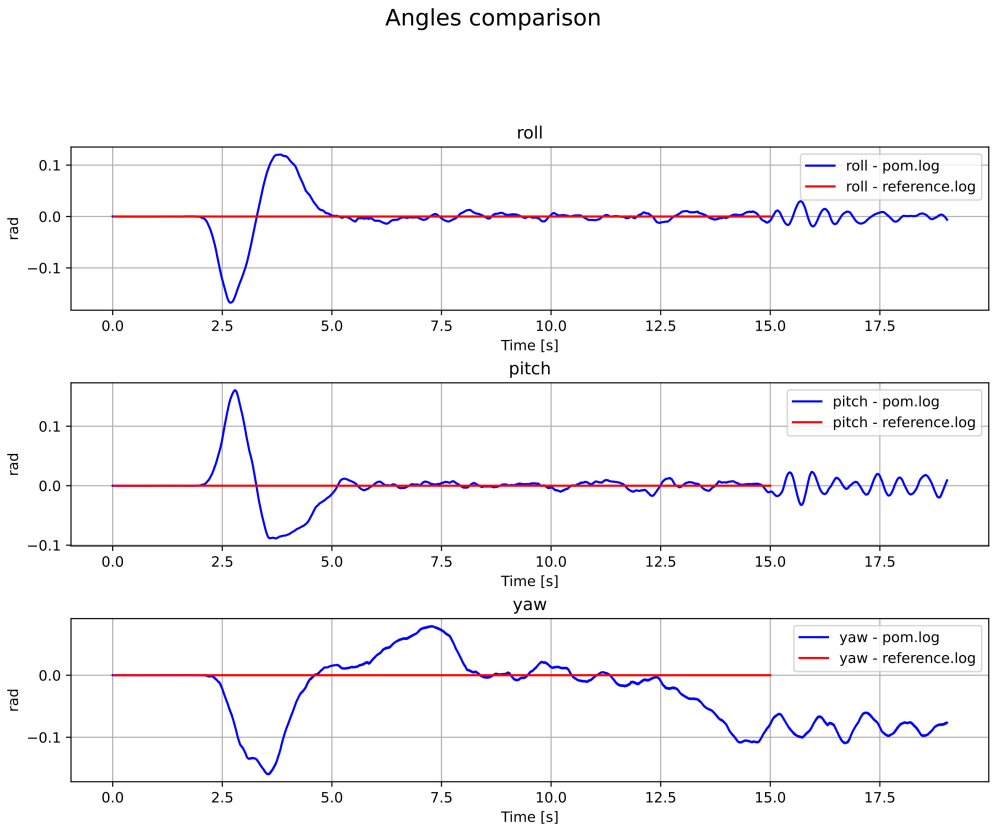


Figure 4.2: Attitude evolution during hovering. Roll and pitch track reference commands with precision better than 0.5°, while yaw exhibits bounded variations around 0.1 rad (5.7°), consistent with weak coupling to position control during hovering.

Attitude evolution (Figure 4.2) confirms smooth, coordinated orientation control. Roll and pitch track reference values precisely (better than 0.5° RMS), demonstrating effective coordination between our high-level MPPI policy and the underlying UAVAtt attitude stabilization layer. Yaw shows slightly larger fluctuations (approximately 0.1 rad) but remains stable and bounded, which is expected since yaw is weakly coupled to position control during hovering.

Figures 4.3 and 4.4 present control inputs and corresponding body torques.

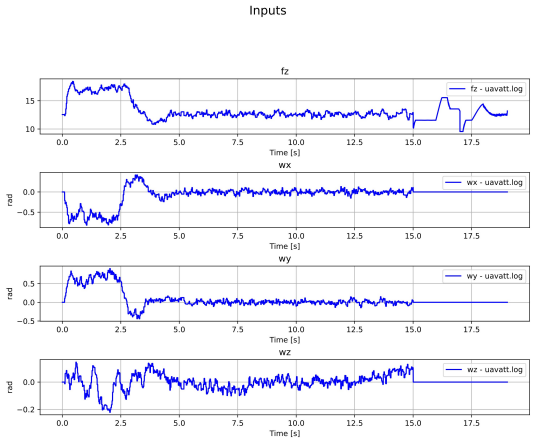


Figure 4.3: Control inputs generated by MPPI during hovering.

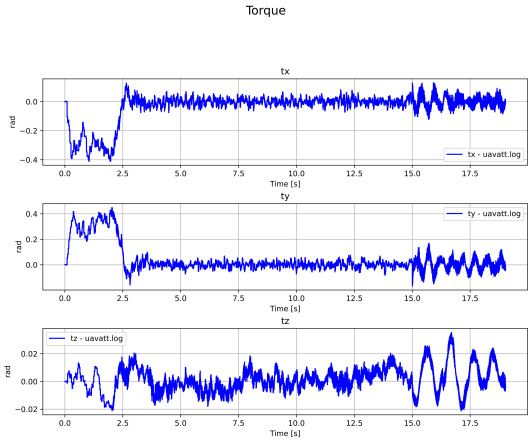


Figure 4.4: Body torques computed by the UAVAtt controller, confirming consistent actuation without saturation or discontinuities.

Both signals remain continuous and well-behaved, confirming absence of numerical instabilities. Thrust stabilizes near the nominal gravity compensation value (12.7 N for our 1.3 kg platform), while angular rate commands stay well within actuator limits (peak values < 0.5 rad/s).

Trajectory Tracking Performance

Having established baseline performance during hovering, we evaluated dynamic flight capabilities through progressively complex trajectory-tracking scenarios.

Continuous Linear Trajectory. The first dynamic test employed consecutive point to point trajectories forming a continuous spatial path with multiple linear segments. While geometrically C^0 continuous, the velocity profile includes C^1 discontinuities at connection points where direction changes abruptly, challenging predictive controllers to adapt across velocity discontinuities.

Figure 4.5 demonstrates excellent tracking with average position error below 2 cm. Importantly, the controller handles velocity discontinuities at segment junctions without overshoot, indicating that MPPI's predictive horizon successfully anticipates direction changes. Attitude evolution (Figure 4.6)

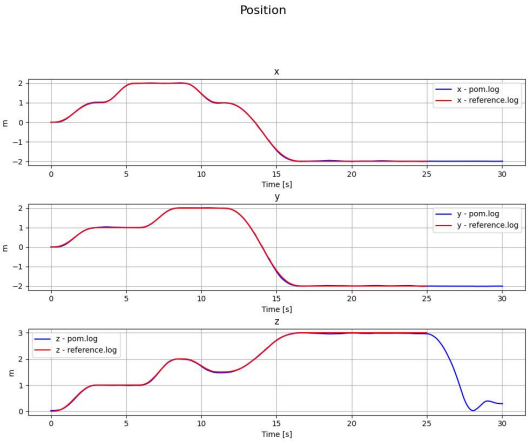


Figure 4.5: Position tracking during continuous trajectory following.

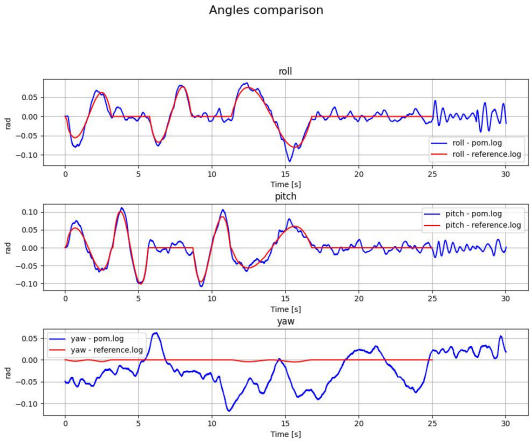


Figure 4.6: Attitude evolution during trajectory tracking.

exhibits well coordinated dynamics, with peak angles reaching approximately 8° during aggressive segments.

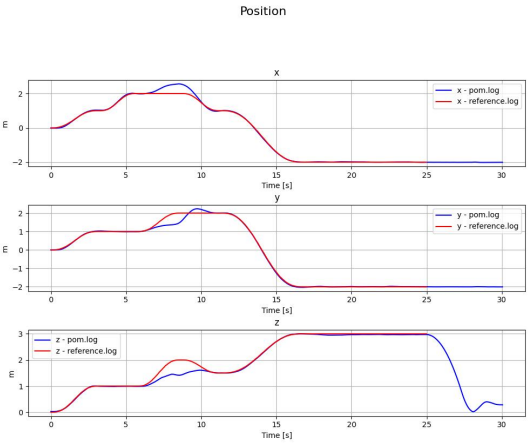


Figure 4.7: Position tracking with obstacle present. The vehicle executes a smooth lateral deviation (0.4 m) to maintain safe clearance.

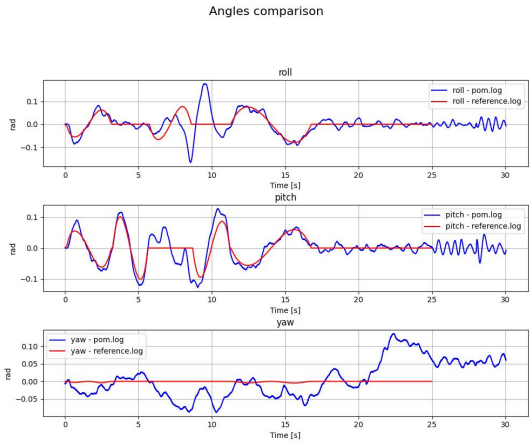


Figure 4.8: Attitude response during obstacle avoidance. Peaks (up to 12°) corresponding to lateral deviation.

Continuous Trajectory with Obstacle Avoidance. To evaluate constraint handling, we introduced a static spherical obstacle (0.3 m radius) along the reference trajectory, forcing lateral deviation while maintaining

forward progress. The obstacle was penalized through an exponential barrier term in the cost function.

Figure 4.7 shows successful avoidance with smooth lateral deviation (0.4 m) and maximum error below 5 cm. Attitude evolution (Figure 4.8) exhibits transient peaks consistent with avoidance motion, with angles remaining within safe operating limits.

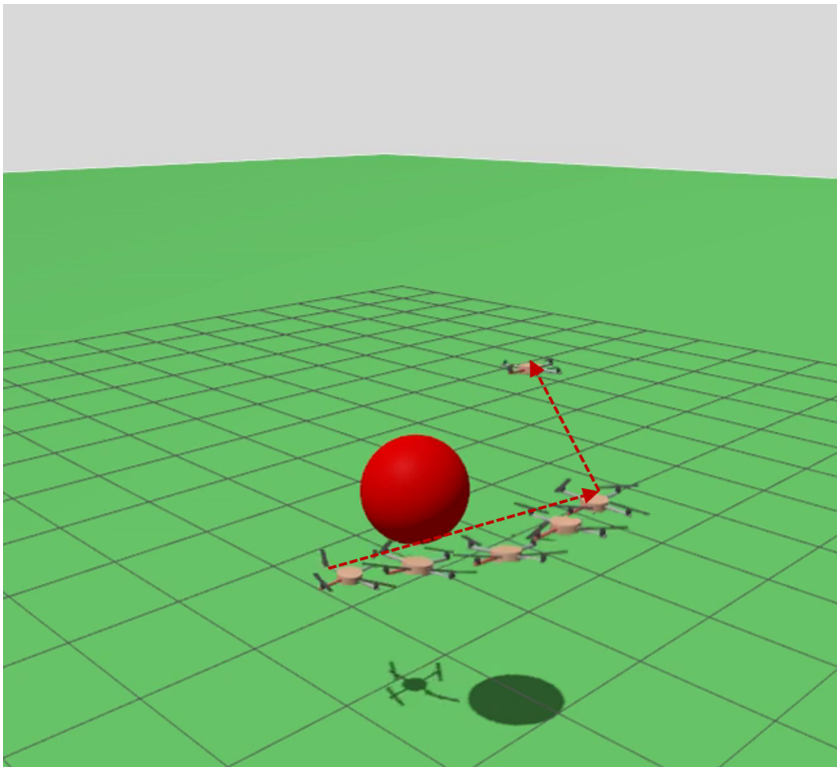


Figure 4.9: Visualization of obstacle avoidance in Gazebo. Overlaid frames illustrate smooth deviation trajectory and subsequent return to nominal path.

Importantly, this avoidance behavior emerged purely from cost function structure without separate planning modules or safety filters, demonstrating a key advantage of sampling-based methods.

Circular Trajectory Tracking. The final dynamic scenario involved continuous circular motion (2 m radius, 1 m altitude, 0.5 m/s tangential velocity), selected to evaluate long term numerical stability and coupled dynamics. The drone completed three full revolutions.

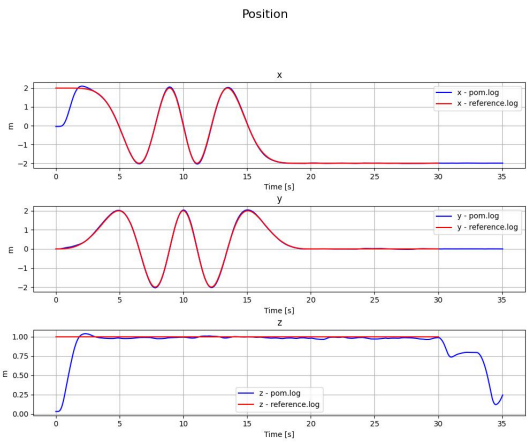


Figure 4.10: Position tracking along circular trajectory. The executed path follows the reference.

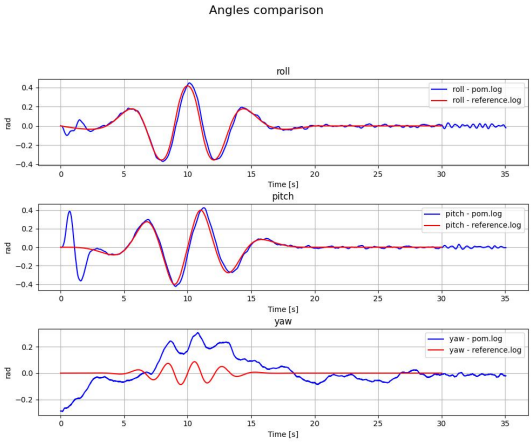


Figure 4.11: Attitude evolution during circular motion. Roll and pitch follow the reference, while yaw remains stable within 0.2 rad.

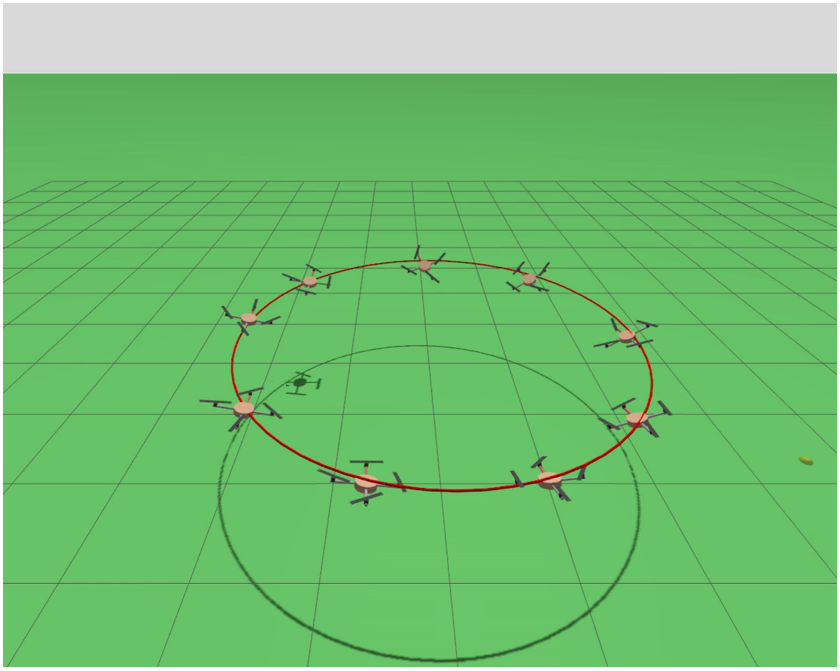


Figure 4.12: Visualization of circular trajectory execution in Gazebo. Uniformly spaced frames confirm constant velocity maintenance throughout motion.

After brief initial transient, the drone settles into steady tracking with radial error below 3 cm (Figure 4.10). Crucially, no systematic drift or error accumulation appears over three revolutions. Attitude behavior (Figure 4.11) exhibits clear periodic structure with roll and pitch oscillating at revolution frequency ($\pm 10°$), while yaw shows bounded variations within 0.2 rad.

Circular Trajectory with Obstacle Avoidance. As final simulation validation, we combined continuous circular motion with repeated obstacle avoidance. The obstacle was positioned such that the drone encounters it twice per revolution, forcing repeated avoidance maneuvers.

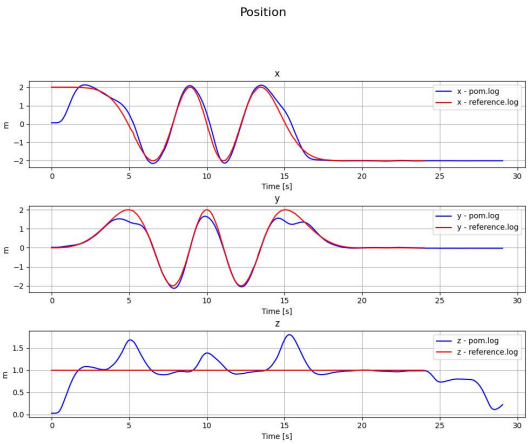


Figure 4.13: Position tracking during circular motion with obstacle avoidance. The drone executes deviations (15 cm lateral, 5 cm vertical) maintaining safe clearance.

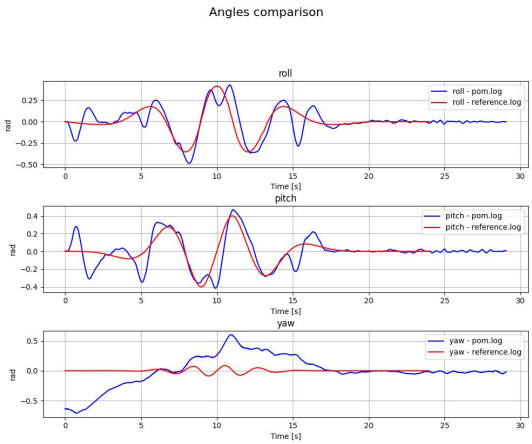


Figure 4.14: Attitude response during repeated avoidance. Roll and pitch show transient peaks (15°).

The drone successfully executes repeated avoidance (Figure 4.13) with consistent behavior across encounters. Attitude evolution (Figure 4.14) reveals transient peaks superimposed on periodic oscillations, demonstrating robust coordination.

Summary of Simulation Results. The Gazebo experiments establish:

1. **Baseline stability:** MPPI maintains smooth control across static and dynamic scenarios with position errors below 5 cm

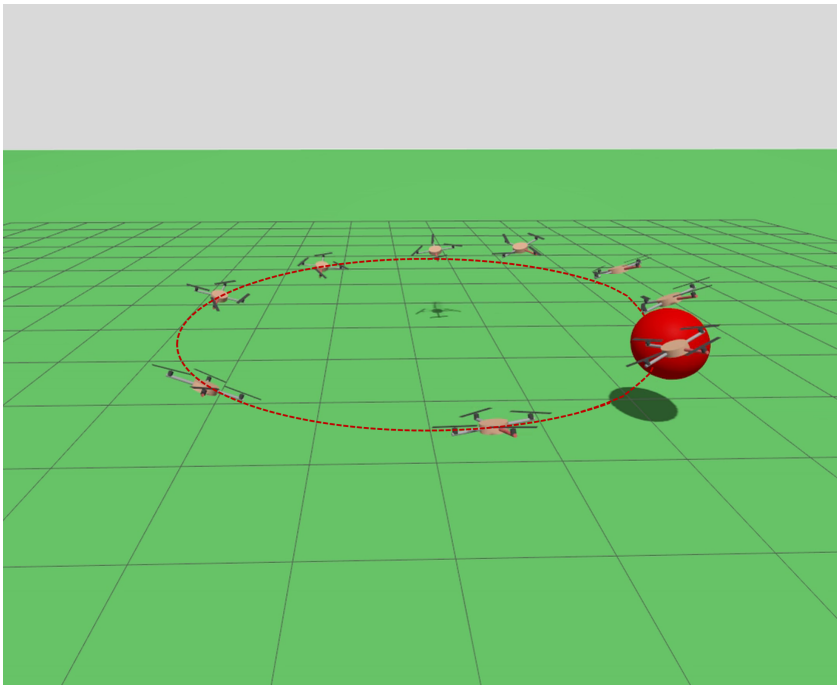


Figure 4.15: Visualization of repeated obstacle avoidance during circular tracking, showing consistent avoidance behavior across multiple encounters.

2. **Constraint handling:** Obstacle avoidance emerges naturally from cost function structure
3. **Long-term consistency:** Extended tests show no error accumulation or numerical drift
4. **Numerical conditioning:** Control signals remain smooth, suggesting good numerical conditioning

4.2.2 Real-World Flight Experiments

Following successful simulation validation, we transitioned to physical flight tests using the mkquad5 platform with Jetson Orin Nano. All flights occurred indoors with VICON motion capture providing state estimates at 1000 Hz.

Hovering Performance

We commanded the drone to maintain fixed position at approximately (-0.027, 0, 1.05) m, with each test lasting 20 seconds followed by controlled landing.

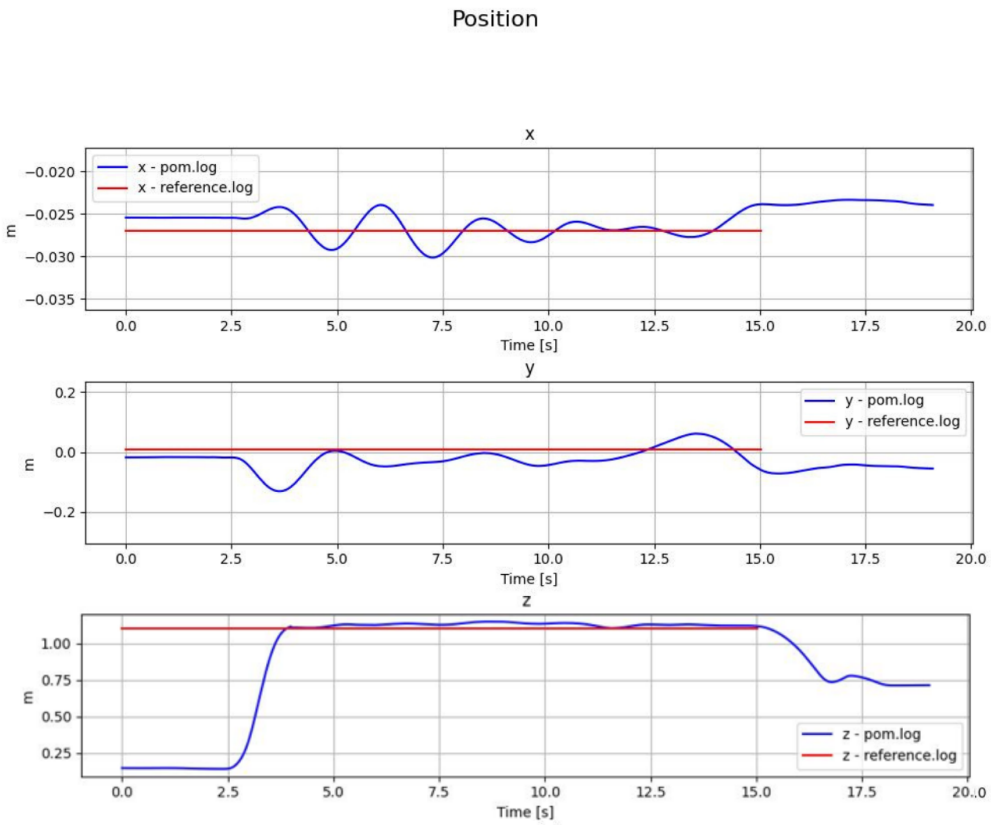


Figure 4.16: Position tracking during physical hovering and landing. Horizontal coordinates show oscillations within ±5 mm during hovering. Z coordinate follows correctly the hovering point during the hovering phase, followed by controlled landing sequence (t > 15s).

Figure 4.16 shows excellent horizontal stability (±5 mm) for the hovering position. The controlled landing phase (t > 15s) demonstrates smooth descent without abrupt drops, confirming stable terminal behavior. Importantly, despite altitude drift, horizontal positioning remains stable, demonstrating that MPPI successfully decouples slow thrust model mismatch from faster position control dynamics.

Attitude behavior (Figure 4.17) shows coordinated control with roll and pitch oscillating (±0.05 rad) during hovering, coupled to position corrections. During landing, both angles converge smoothly toward zero. Yaw exhibits gradual drift (0.025 rad) but remains bounded, consistent with lower priority in the control objective.

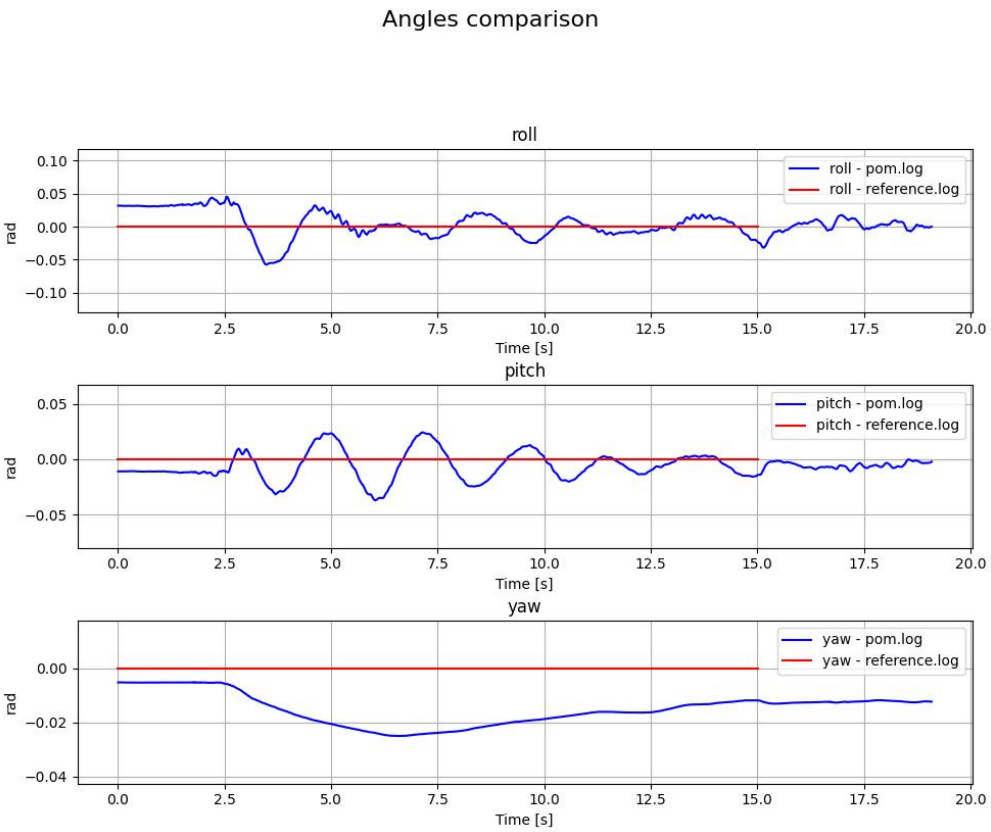


Figure 4.17: Attitude evolution during physical hovering and landing. Roll and pitch exhibit periodic oscillations (±0.05 rad) during hovering. During landing (t > 15s), attitudes converge smoothly toward zero.

Comparing with simulation, real platform maintains horizontal accuracy (±5 mm) comparable to Gazebo during hovering, confirming successful sim-to-real transfer for core control objectives.

Continuous Trajectory Tracking

We evaluated dynamic tracking using connected linear segments over 25 seconds including takeoff, tracking, and landing phases.

During active tracking (Figure 4.18), horizontal coordinates demonstrate good adherence with typical errors below 10 cm. Initial segments show higher error (10-12 cm) reflecting adaptation period, but quality improves significantly once adapted (errors < 5 cm). Z coordinate maintains excellent altitude tracking with deviations below 5 cm.

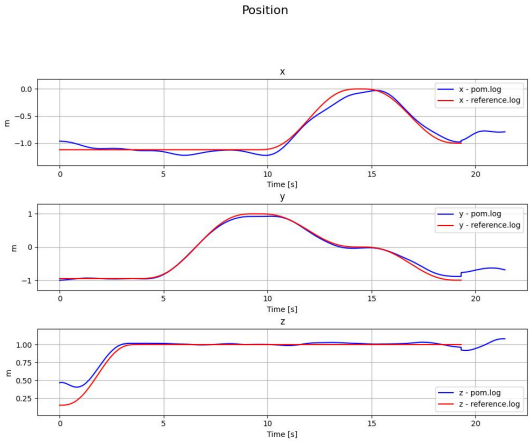


Figure 4.18: Position tracking during physical trajectory following through takeoff (t < 5s), tracking (5s < t < 18s), and landing (t > 18s) phases. Tracking error during active phase remains below 10 cm.

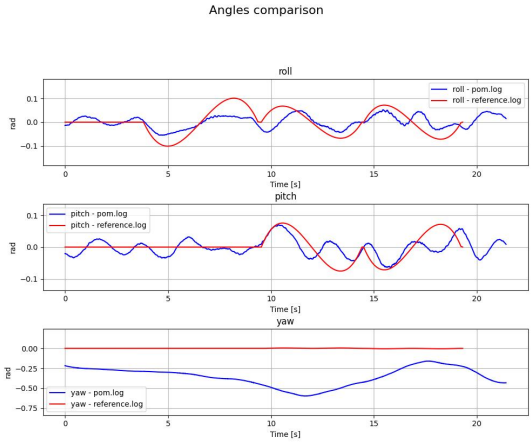


Figure 4.19: Attitude evolution during physical trajectory tracking. Roll and pitch exhibit periodic modulation with amplitudes up to ±0.1 rad, corresponding to acceleration demands. Yaw shows gradual drift (up to -0.6 rad) but remains stable.

Attitude behavior (Figure 4.19) shows coordinated dynamics with roll and pitch oscillating up to ±0.1 rad. Yaw exhibits larger drift (0.6 rad) than hovering, which is expected during dynamic flight where horizontal tracking demands priority.

Real-world tracking error (5-10 cm during steady segments) is approximately 2-3× larger than Gazebo (2 cm), aligning with expected simu-to-real gap. Crucially, the controller maintains stability throughout without parameter retuning.

Trajectory Tracking with Obstacle Avoidance

As final validation, we tested collision avoidance during during tracking test. A physical obstacle (two stacked cardboard cubes, 60 cm sides) was positioned at (0, 0, 0.0) m, the center of the lower box was in (0, 0, 0.31) m position, while upper box had center in (0, 0, 0.91) m. The drone navigated from (-1.5, 0, 0) to (1.5, 0, 1) m, with reference passing directly through the obstacle.

Figure 4.20 clearly demonstrates successful avoidance. X coordinate shows

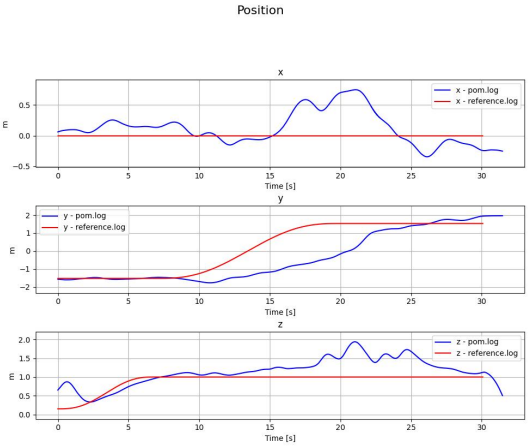


Figure 4.20: Position tracking during obstacle avoidance. Lateral deviation in X (peak 0.7 m at t $\approx$ 20s) and delayed Y progression demonstrate active avoidance, while Z maintains stable altitude tracking.

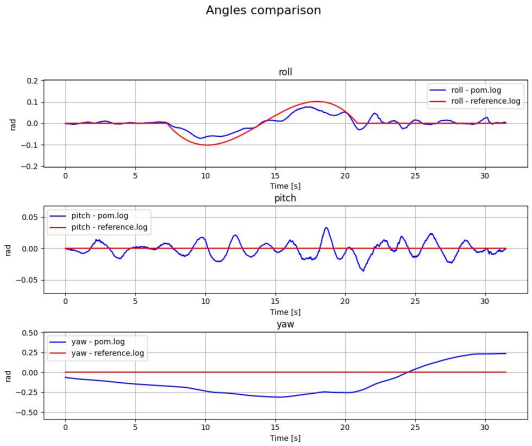


Figure 4.21: Attitude evolution during real-world obstacle avoidance. Roll exhibits pronounced oscillations (peak 0.1 rad) during lateral deviation. Yaw drift (0.25 rad) stays within acceptable bounds.

significant lateral deviation (peak 0.7 m) precisely when encountering the obstacle, while Y shows delayed progression during avoidance phase. Altitude tracking remains excellent (<10 cm deviation) throughout maneuver.

Attitude behavior (Figure 4.21) reflects dynamic demands with pronounced roll activity ($\pm$0.1 rad) during lateral deviation. The oscillatory nature indicates continuous replanning rather than open-loop execution.

Real-world avoidance behavior is qualitatively similar to simulation but quantitatively more conservative (0.7 m vs 0.4 m clearance), appropriately accounting for real-world uncertainties.

These experiments validate complete MPPI control architecture under most demanding real-world scenario: dynamic motion with real-time constraint satisfaction.

Figure 4.22: Multi-exposure photograph showing overlaid drone positions during avoidance maneuver. Point A marks start, structure B represents target point, and frame sequence illustrates lateral deviation trajectory.

4.3 Numerical Precision Analysis on Embedded Hardware

Having validated MPPI functionality through Gazebo simulations and real-world flights, we now address the central thesis question:
what impact does numerical precision have on computational efficiency and control quality when implementing MPPI on embedded hardware?
This section presents systematic investigation comparing float32 and float16 arithmetic across broad controller configuration spectrum, conducted using MuJoCo simulation running directly on Jetson Orin platform.

Unlike previous experiments evaluating overall system behavior, these precision-focused tests isolate specific effects of arithmetic resolution on both computational performance (execution time) and control quality (tracking accuracy), directly informing future FPGA implementations where precision can be customized at hardware level.

4.3.1 Experimental Methodology

We designed controlled test campaign isolating precision effects from confounding factors while maintaining realistic control conditions. All experiments used identical trajectory references, cost function parameters, and initial conditions. The variables that changed during the test were controller hyperparameters (prediction horizon H, sample count K).

Test Environment and Hardware Configuration. Experiments employed custom MuJoCo-based simulation implemented entirely in Python using JAX for automatic differentiation and CUDA acceleration. This lightweight environment runs directly on Jetson Orin Nano (16 GB), replicating embedded deployment constraints. Unlike Gazebo, JAX allows configuring all operations (state propagation, cost evaluation, weight computation, control updates) to consistently use either float32 or float16 throughout entire pipeline.

Quadrotor dynamics correspond to Section 2.2 formulation, with modified input representation (Section 3.2). While not strictly real-time due to Python overhead, we implemented temporal synchronization maintaining update frequencies comparable to physical system **(target 100 Hz)**.

Controller Configuration Space. We systematically varied two key MPPI hyperparameters directly influencing both computational load and control quality:

- **Prediction horizon** $H \in \{11, 17, 20, 25, 35\}$ time steps (0.11–0.35 s lookahead). Longer horizons improve anticipatory behavior but increase integration error accumulation and computational cost
- **Number of samples** $K \in \{100, 1000, 2000, 3000, 10000, 50000\}$ rollouts per control cycle. More samples improve statistical quality but scale computation time linearly

This produced 30 distinct configurations (5×6 grid). Each configuration was tested with both float32 and float16 (60 experimental conditions total). Each condition involved 10 repetitions accounting for stochastic MPPI sampling variation, yielding 600 total experimental runs.

Cost Function and Reference Trajectory. Cost function remained identical across tests, using Equation (3.4) structure with weights tuned during Gazebo validation. Reference trajectory consisted of circular path identical to previous experiments (2 m radius, 1 m altitude, 0.5 m/s tangential velocity), generated through D F.

Performance Metrics. We evaluated each configuration using two complementary metrics capturing speed-accuracy tradeoff:

1. **Computation Time per Control Cycle:** Mean execution time from state input to control output, recorded using CUDA events for precise GPU kernel execution measurement, excluding Python overhead to isolate core computational cost
2. **Tracking Error (RMSE):** Root mean squared deviation between reference and actual position:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \|\mathbf{p}_i - \mathbf{p}_{\text{ref},i}\|^2}$$

Preliminary tuning established that with $H = 20$ and $K = 2000$, the controller achieves steady tracking error below 3 cm in simulation. Based on

sim-to-real comparison showing 50% degradation, we established practical acceptance threshold of **10 cm RMSE**. Configurations maintaining error below this limit are considered satisfactory, making computation time the primary selection criterion among acceptable configurations.

4.3.2 Quantitative Configuration Summary

Tables 4.1–4.4 present complete quantitative performance summary across entire controller configuration space, providing exact numerical values for all 30 tested configurations under both precision formats.

Table 4.1: Tracking error (RMSE in cm) for MPPI configurations using float32 precision. Tuned configuration (H=20, K=2000, bold) achieves optimal performance at 2.03 cm.

N samples	H=11	H=17	H=20	H=25	H=35
100	8.56	3.02	2.71	3.63	8.05
1000	6.68	3.05	1.90	2.21	5.03
2000	6.78	2.81	**2.03**	2.10	4.20
3000	6.36	2.87	2.01	1.94	4.09
10000	6.44	2.80	1.79	2.01	3.71
50000	6.11	2.76	1.69	1.89	3.68

Table 4.2: Tracking error (RMSE in cm) for MPPI configurations using float16 precision. Tuned configuration (H=20, K=2000, bold) achieves 1.77 cm.

N samples	H=11	H=17	H=20	H=25	H=35
100	8.84	3.54	2.71	3.11	8.49
1000	6.59	2.70	2.05	2.67	4.56
2000	6.67	2.96	**1.77**	2.19	3.93
3000	6.18	2.82	1.98	2.06	4.28
10000	6.21	2.79	1.77	1.81	3.93
50000	6.27	2.69	1.73	1.90	3.52

While these tables provide precise numerical data essential for reproducibility, patterns and trends emerging from 60 data points are more readily comprehensible through graphical visualizations. Following sections present

Table 4.3: Computation time per control cycle (ms) for MPPI configurations using float32 precision. Tuned configuration (H=20, K=2000, bold) requires 10.97 ms, enabling control frequencies up to 91 Hz.

N samples	H=11	H=17	H=20	H=25	H=35
100	6.66	7.31	7.91	8.72	10.36
1000	7.11	8.58	9.42	10.50	13.16
2000	8.30	10.17	**10.97**	12.76	16.56
3000	9.41	11.64	13.06	15.35	20.00
10000	18.10	24.73	27.79	32.76	38.03
50000	43.50	52.91	53.67	59.65	73.82

Table 4.4: Computation time per control cycle (ms) for MPPI configurations using float16 precision. Tuned configuration (H=20, K=2000, bold) requires only 10.20 ms, 7% faster than float32.

N samples	H=11	H=17	H=20	H=25	H=35
100	6.46	7.16	7.45	8.36	9.76
1000	6.94	8.20	9.04	10.31	12.59
2000	8.05	9.08	**10.20**	11.62	14.85
3000	9.25	10.80	11.65	13.84	17.70
10000	14.74	20.25	23.01	26.87	34.42
50000	37.31	43.52	48.43	52.46	61.04

this data as heatmaps, scatter plots, and bar charts, revealing complementary insights into precision and performance tradeoffs.

4.3.3 Horizon Sensitivity Analysis

Before examining global performance, we analyze how prediction horizon influences tracking accuracy. A critical factor interacting subtly with numerical precision.

Figures 4.23 and 4.24 reveal critical insight: *numerical precision is not the primary MPPI performance bottleneck*. Rather, *alignment between prediction horizon and cost function tuning* dominates tracking accuracy. Both figures show identical patterns:

1. **Valley at H=20:** All curves exhibit sharp local minimum at H=20, precisely the horizon used during cost function tuning in Gazebo

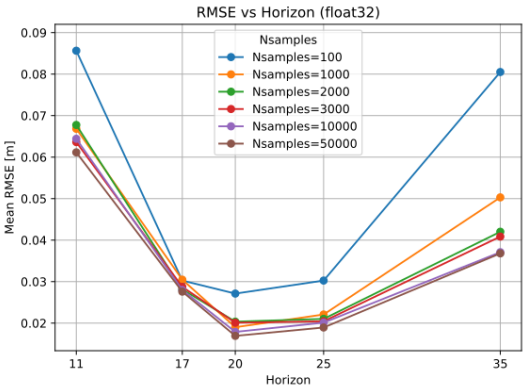


Figure 4.23: RMSE vs prediction horizon for float32. All curves show minimum at H=20 (tuned configuration), with degradation for both shorter and longer horizons.

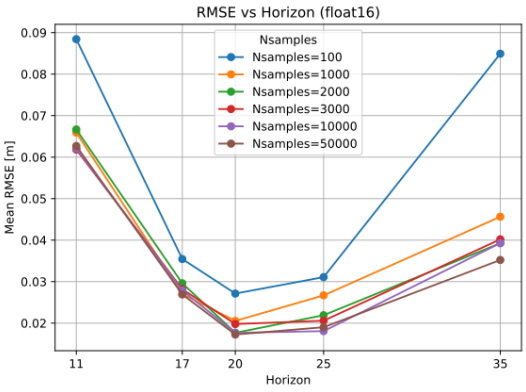


Figure 4.24: RMSE vs prediction horizon for float16. Pattern nearly identical to float32, confirming cost function tuning dominates precision effects.

2. **Asymmetric degradation:** Short horizons (H=11) show worst degradation (∼8-10 cm RMSE, +400% vs H=20), while long horizons (H=35) show moderate degradation (∼4-5 cm, +150%)

3. **More samples cannot compensate:** Increasing K from 2000 to 50000 provides minimal RMSE improvement when H deviates from tuned value, demonstrating that *horizon misalignment is structural, not statistical*

4. **Precision equivalence:** Float16 and float32 curves are virtually indistinguishable, with differences typically below 0.5 cm

This behavior emerges because cost function tuning with H=20 established specific weights balancing immediate vs future costs under 2-second lookahead assumption. Changing horizon breaks this balance: H=11 provides insufficient lookahead for anticipating trajectory curvature; H=35 accumulates integration errors and may induce overly cautious behavior.

These findings have important FPGA design implications: hardware implementations should prioritize **flexible horizon configuration** over brute and force sample scaling, since former enables task-specific tuning while latter offers diminishing returns beyond $K \approx 2000$.

4.3.4 Computational Performance Results

We now examine computational performance across entire configuration space, revealing how numerical precision affects execution time and real-time control feasibility.

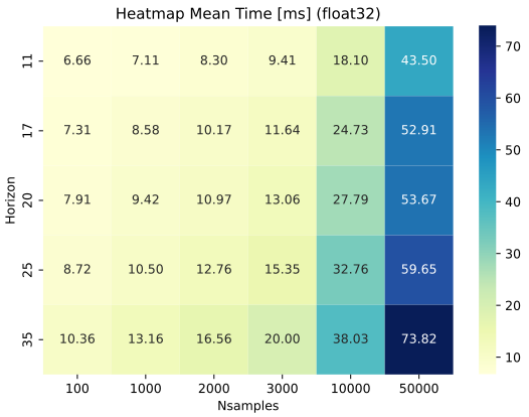


Figure 4.25: Computation time heatmap (ms) for float32. Times scale approximately linearly with K and sublinearly with H, ranging from ∼7 ms to ∼74 ms.

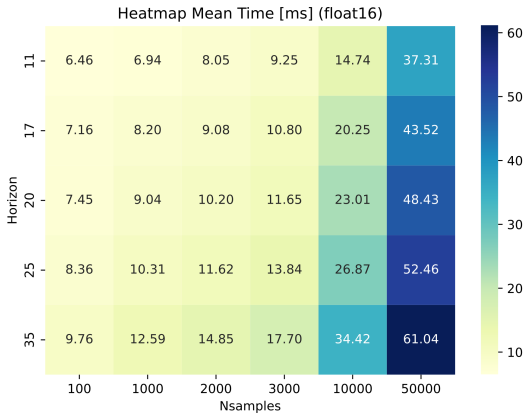


Figure 4.26: Computation time heatmap (ms) for float16. Scaling patterns identical to float32 but with consistently lower values—most cells show 7-17% reduction.

Figures 4.25 and 4.26 provide panoramic performance view. Key patterns:

1. **Consistent float16 advantage:** Comparing cell by cell, float16 produces systematically lower execution times across *all* 30 configurations. Universal advantage confirming precision reduction provides real computational benefits

2. **Linear scaling with K:** Times increase approximately linearly along columns, confirming trajectory propagation dominates computational cost and GPU parallelization effectively distributes load

3. **Sublinear scaling with H:** Times grow less than proportionally along rows (H=11 to H=35: 3.2× increase yields typical 1.5-2× time growth), likely reflecting better GPU utilization

4. **Load-dependent speedup:** Float16 speedup increases with computational load. Small configurations show modest gains (3-5%) while heavy configurations reach 17% reduction

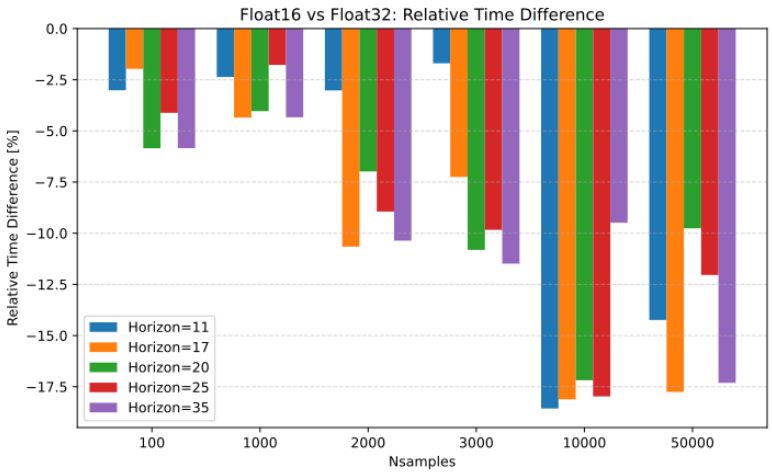


Figure 4.27: Relative time difference (%) between float16 and float32. Negative bars indicate float16 faster.

Figure 4.27 quantifies speedups directly, confirming:

- **100% success:** All 30 bars negative. float16 never fails providing computational benefit
- **Scaling behaviour:** Increasing the number of samples generally leads to larger speedups when using `float16`, but the trend is not strictly monotonic within each horizon.
- **Practical average:** For reasonable configurations (K=1000-10000, H=17-25), speedup averages 8-13%

The tuned configuration ($H = 20$, $K = 2000$) achieves 10.20 ms with `float16` vs 10.97 ms with `float32` (98 Hz vs 91 Hz maximum frequencies). While a 7% gain may seem modest, it enables comfortable operation above the 50 Hz stability threshold for quadrotor control while maintaining timing margins. For aggressive configurations ($K \geq 5000$) needed in complex scenarios, `float16` becomes essential for maintaining real-time feasibility.

4.3.5 Tracking Accuracy Results

Having established computational advantages, we examine whether these gains sacrifice control quality.

Figures 4.28 and 4.29 reveal surprising result: *reducing numerical precision does not degrade control quality.* Heatmaps show virtually identical patterns:

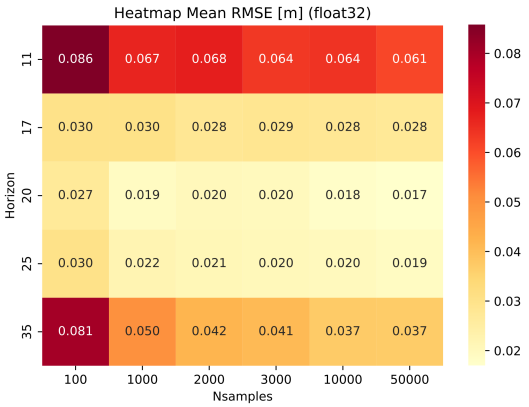


Figure 4.28: Tracking error heatmap (RMSE in cm) for float32. Error ranges from ∼1.7 cm (optimal: $H = 20$, $K \geq 10000$) to ∼8.6 cm (poor: $H = 11$, $K = 100$). Optimal region clusters around tuned configuration.

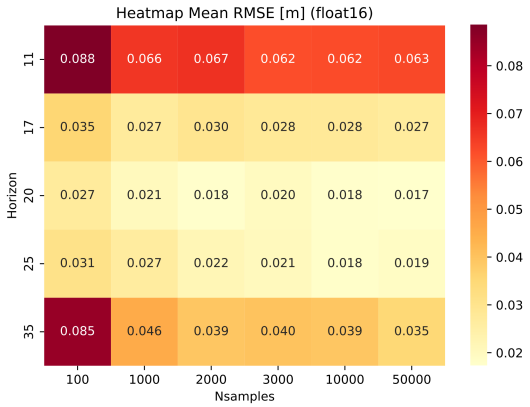


Figure 4.29: Tracking error heatmap (RMSE in cm) for float16. Spatial pattern nearly identical to float32, with differences typically <0.5 cm—well within experimental noise.

1. **Same performance topology:** Both maps show an optimal region centered on $H = 20$ with $K \geq 2000$, worst error in the upper-left corner ($H = 11$, small K), and moderate degradation in the lower-right corner ($H = 35$, large K).

2. **Minimal value-by-value differences:** Comparing corresponding cells, RMSE differences typically 0.2-0.5 cm, with some cells showing float16 slightly *better* (e.g., tuned configuration: 1.77 cm vs 2.03 cm)

3. **Threshold maintained:** Both precisions maintain RMSE < 5 cm in well-tuned region ($H = 17$–25, $K \geq 2000$) and both exceed 6 cm in poorly-tuned region ($H = 11$).

4. **Convergence pattern:** Along each column, error decreases rapidly with increasing K until ∼2000-3000, then stabilizes with diminishing returns—pattern identical for both precisions

Figure 4.30 quantifies differences directly:

- **Zero-centered distribution:** Bars scattered above and below zero without systematic pattern

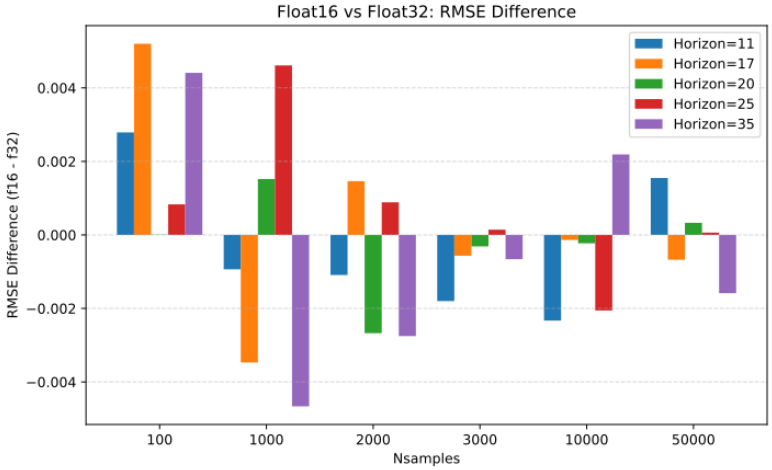


Figure 4.30: Relative RMSE difference (%) between float16 and float32. Bars oscillate around zero (typical range: ±2%), indicating accuracy equivalence without systematic bias.

- **Narrow range:** Most bars fall within ±3%, with extreme cases reaching ±5%
- **No instability:** Absence of large positive bars confirms robust numerical stability

This counterintuitive result emerges from MPPI properties:

1. **Stochastic regularization:** The MPPI algorithm inherently relies on sampling trajectories with stochastic perturbations. The rounding errors introduced by the float16 arithmetic (approximately 10^{-4} relative) are therefore minor compared to deliberate sampling noise and are effectively absorbed within the stochastic process itself.
2. **Weighted averaging:** The control command in MPPI is computed as a weighted average across all perturbed trajectories. This procedure naturally mitigates the effect of individual numerical inaccuracies: even if some trajectories experience float16 integration drift, their impact is diluted when aggregated over thousands of samples.
3. **Receding horizon:** The Receding horizon formulation continuously reinitializes the prediction horizon at each control step, which prevents long-term accumulation of numerical errors.

4. **Sufficient dynamic range:** The numerical values typically involved in MPPI—such as states, control inputs, and cost terms comfortably fall within the representable range of float16 ($\pm 6.5 \times 10^4$). With appropriate cost normalization, this ensures that neither overflow nor underflow occurs during computation.

4.3.6 Pareto Trade-off Analysis

Having examined time and accuracy separately, we synthesize through scatter plots revealing precision-performance tradeoff.

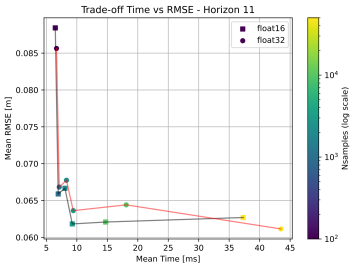


Figure 4.31: Scatter H=11: Short horizon produces poor accuracy regardless of precision or samples.

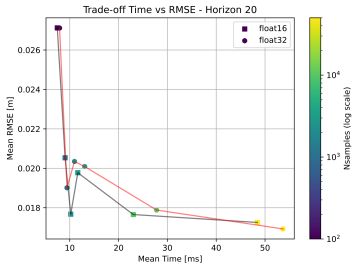


Figure 4.32: Scatter H=20: Tuned configuration shows float16 dominance—nearly all orange points left/below blue points.

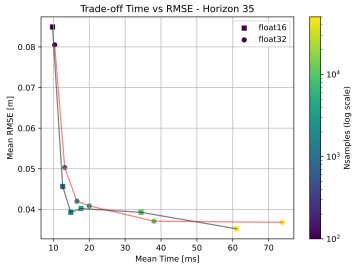


Figure 4.33: Scatter H=35: Float16 speedup maximum here (17%) but accuracy degraded for both precisions.

Scatter plots (Figures 4.31–4.35) provide richest tradeoff view, revealing how precision interacts with architectural choices:

H=11 (Figure 4.31): Short horizon produces worst accuracy (∼6-9 cm) across all sampling configurations. Even with K=50000, RMSE remains above 6 cm. No amount of samples compensates for structurally inadequate horizon. Float16 provides modest speed gains (3-14%) but advantage is academic given unacceptable control quality.

H=20 (Figure 4.32): Most important scatter plot showing tuned configuration. Observations:

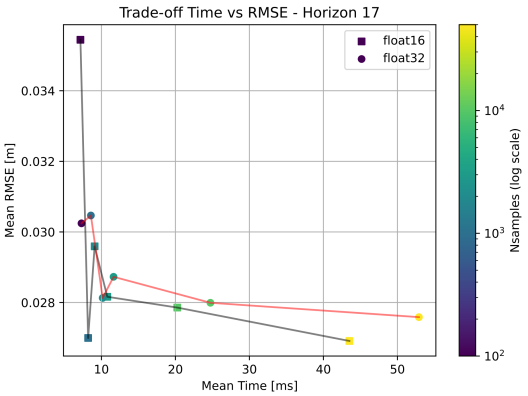


Figure 4.34: Scatter H=17: Good accuracy (3 cm) with moderate float16 speed advantage and comparable accuracy.

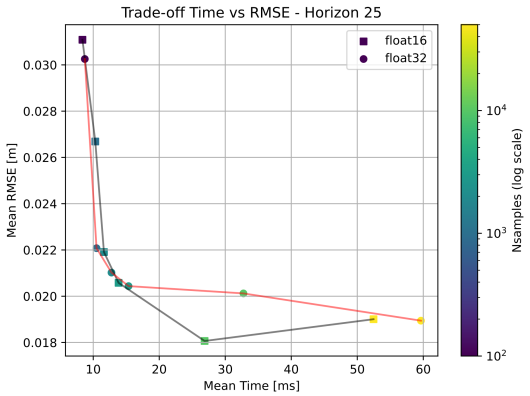


Figure 4.35: Scatter H=25: Accuracy degradation (2-4 cm) vs H=20, with less pronounced float16 advantage.

- **Pareto dominance:** Orange float16 points form evident lower-left envelope. Nearly every float16 configuration is both faster and more accurate or equivalent to float32 counterpart
- **Tight clustering:** Configurations with $K \geq 2000$ cluster in lower-left corner (time <15 ms, RMSE <2.5 cm)
- **Diminishing returns:** Distance between K=2000 and K=50000 points smal: 7 ms additional cost for only 0.5 cm RMSE improvement

H=17 and H=25 (Figures 4.34, 4.35): Adjacent horizons show interesting transitions. H=17 maintains good accuracy (2.5-3.5 cm) with float16 speed advantage (8-12%). H=25 shows accuracy degradation (2-4 cm) with greater spread, suggesting that as we move from tuned horizon, other factors (integration errors, model approximation) begin dominating precision effects.

H=35 (Figure 4.33): Long horizon produces maximum computational explosion (times up to 75 ms float32, 61 ms float16) and largest relative speedup (17%). However, accuracy degrades (3.5-8.5 cm) with high variance. This graphically illustrates why extreme computational loads don't always improve performance. Cost is high but benefits degrade.

Across all horizons, consistent themes emerge: (1) float32 points are consistently shifted rightward (slower) compared to their float16 counterparts,

(2) the best trade-off occurs at $H = 20$, where tuning, precision, and computational cost align optimally, and (3) no single global pattern emerges across horizons—each H value defines a distinct performance landscape, reinforcing that *architectural choices dominate precision effects*. Overall, these observations demonstrate that **float16 not only provides measurable computational advantages but also preserves, and in some cases slightly improves, control accuracy**, making it the preferred precision format for real-time MPPI on embedded hardware.

4.3.7 Discussion and Implications for FPGA Design

Experimental results establish several key conclusions with direct implications for future hardware implementations of MPPI control.

Key Experimental Findings.

1. **Float16 provides consistent speedup (7-17%)** without compromising control quality. Across all 30 tested configurations, reduced precision delivered measurable computational advantages while maintaining tracking accuracy within ±2% of float32
2. **Architectural tuning dominates precision effects.** Prediction horizon choice (H=20 vs H=11) creates 400% accuracy differences, while precision choice (float32 vs float16) produces <3% variations, clarifying design priorities

FPGA Implementation Guidelines. Based on these results, we formulate specific recommendations for future FPGA designs:

1. **Adopt 16-bit arithmetic as baseline.** Data conclusively demonstrates 16-bit precision maintains full control quality while preserving stability margins. FPGAs should implement 16-bit datapaths for trajectory propagation, cost evaluation, and weight computation, enabling doubled throughput vs 32-bit designs for given silicon area
2. **Prioritize horizon configurability over extreme sample scaling.** Since H-tuning alignment dominates performance, FPGA architectures should support easily adjustable horizon lengths (ideally 10-40 steps) through hardware parametrization. Conversely, designing for K>10000 offers limited utility given observed saturation

3. **Explore mixed precision.** While results show uniform float16-float32 equivalence, some operations might benefit from higher precision. Candidates for 32-bit arithmetic: (a) weight normalization (where underflow could concentrate weights on few trajectories), (b) cost accumulation (where summing thousands of terms might amplify errors), (c) state integration (where error accumulation over H steps might degrade predictions). Mixed strategy could allocate 32-bit datapaths only for these sensitive components

4. **Consider even lower precisions for cost evaluation.** Since float16 introduces no observable degradation, room exists exploring 8-bit integer or fixed-point formats for cost function terms, potentially reducing resources further since cost evaluation scales with K and thus dominates area in parallel implementation. As written in the previuos section, this is not possible using JAX, beacuse the less numerical precision that can be adopted in JAX is FP16

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

The work presented in this thesis focused on the design and evaluation of a Model Predictive Path Integral (MPPI) controller for quadrotors, with particular attention to how numerical precision influences real-time performance. Although MPPI has been widely explored in the recent literature, especially in simulation, several practical questions remained open when moving toward embedded deployment. In particular, we wanted to understand whether reduced numerical precision could be used safely on a real aerial platform, how this choice interacts with key controller parameters, and whether these observations could help inform future hardware implementations such as FPGA-based architectures. The thesis addressed these questions through a combination of simulation studies, real experiments, and a large set of numerical tests.

Contribution 1: Hardware-Aware MPPI Implementation. A first major contribution is the complete implementation of an MPPI controller inside the TeleKyb3 framework. This required adapting the standard MPPI formulation so that its outputs were compatible with the existing control chain. In TeleKyb3, the low-level `UAVAtt` module expects desired angular velocities, not body torques, so the system dynamics had to be reformulated accordingly. Although this adjustment is conceptually straightforward, it

has important consequences: it avoids inserting extra transformation blocks (which would add latency or numerical inconsistencies) and keeps the entire control pipeline coherent. With this adaptation, the controller was able to run at almost 100 Hz on the Jetson Orin platform while maintaining compatibility with TeleKyb3's modular structure.

Contribution 2: Multi-Stage Experimental Validation. The second contribution concerns the experimental methodology. Instead of relying solely on simulation, we followed a staged approach. We first tested the controller in Gazebo, using the full TeleKyb3 stack, which allowed us to verify stability and tuning under controlled conditions. After that, we moved to indoor experiments with the mkquad5 platform, using a VICON motion capture system. The transition from simulation to real hardware was smoother than expected: hovering, trajectory tracking, and obstacle avoidance worked reliably without retuning parameters. The increase in tracking error observed in real flights (typically between 5 and 10 cm, compared to about 2 cm in simulation) was in line with typical sim-to-real differences and confirmed that the controller could deal with unmodeled effects such as aerodynamic disturbances or actuation delays. Finally, to isolate the role of numerical precision, we ran a set of dedicated tests using MuJoCo and JAX directly on the Jetson hardware. This allowed us to evaluate precision effects without introducing the additional delays present in the real system.

Contribution 3: Precision–Performance Analysis. A central part of the thesis is the systematic analysis of how numerical precision influences MPPI performance. We considered 30 different controller configurations obtained by varying the prediction horizon (from 11 to 35 time steps) and the number of samples (from 100 to 50000), and we repeated each test under both `float32` and `float16` arithmetic. Each configuration was executed multiple times to average out the inherent randomness of MPPI.

Several trends emerged clearly. First, the use of `float16` consistently reduced computation time, with speedups ranging from about 7% for moderate workloads to more than 15% for the most demanding settings. Interestingly, this reduction in precision did not worsen tracking performance: the difference in RMSE between `float16` and `float32` was typically within a few percent and often smaller than the run-to-run variability of the algorithm itself.

Another important observation is that controller design has a much stronger impact on performance than numerical precision. For example, using a prediction horizon that was not aligned with the cost tuning could increase tracking error by several hundred percent, whereas switching between `float32` and `float16` produced comparatively minor changes. We also found that increasing the number of samples beyond roughly 2000–3000 provided very limited gains compared to the increase in computation time, an insight that is particularly relevant when designing hardware-oriented implementations.

Lastly, the advantage of `float16` became more pronounced for increasingly large workloads. This suggests that the benefits observed on embedded GPUs could be even greater on FPGA platforms, where control of data paths and memory bandwidth is more explicit.

Contribution 4: Guidelines for FPGA Implementation. Based on the experimental results, the thesis also proposes several practical recommendations for future hardware implementations of MPPI:

1. **Use 16-bit arithmetic as the default format.** The experiments show that `float16` is sufficient for reliable MPPI control and offers clear computational gains. On FPGA hardware, where resource usage is critical, 16-bit datapaths would be a natural starting point.

2. **Provide flexibility in the prediction horizon.** Since the horizon plays a central role in achieving good performance, FPGA architectures should make it easy to adjust this parameter rather than fixing it at synthesis time.

3. **Consider a mixed-precision approach.** While 16-bit arithmetic works well overall, certain operations, such as weight normalization or cost accumulation, may benefit from higher precision to avoid overflow or underflow. Using 32-bit precision only where necessary could provide a balanced solution.

4. **Explore sub-16-bit representations for specific components.** Since the cost evaluation stage did not appear sensitive to precision in our tests, fixed-point or 8-bit formats might be viable in some cases, potentially reducing hardware resources further.

These observations complement existing FPGA-based work on MPPI, which mainly focuses on parallelization, by highlighting how precision choices affect performance and where hardware optimizations are likely to be most effective.

Summary. To summarise, the thesis provides four main contributions: the implementation of a real-time MPPI controller tailored to an embedded platform; a set of simulation and real-world experiments validating the approach; a systematic study of how numerical precision affects computation time and control quality; and a set of practical guidelines for future hardware implementations. Overall, the results indicate that reduced numerical precision can be used effectively in sampling-based predictive control without compromising accuracy, and that this choice can significantly improve computation efficiency on embedded systems. These findings contribute to bridging the gap between advanced control algorithms and their deployment on resource-constrained hardware.

Beyond its technical contributions, this work supports the broader development of efficient and reliable autonomous aerial systems. By improving computational efficiency without degrading control performance, the thesis contributes to the advancement of innovative and resource-aware robotic technologies, directly aligning with the United Nations Sustainable Development Goal 9 (Industry, Innovation and Infrastructure).

5.2 Limitations

While the proposed MPPI controller demonstrated reliable real-time performance in simulation and in physical experiments, several aspects of this work should be interpreted with caution. Some limitations arise from practical constraints encountered during the experimental campaign, while others reflect intrinsic trade-offs of the methodology.

Hyperparameter Sensitivity. One of the clearest findings is that controller performance is highly sensitive to the choice of prediction horizon. As shown in Chapter 4, deviations from the tuned configuration ($H = 20$) led to substantial degradation in tracking accuracy, whereas switching between `float32` and `float16` produced only marginal differences. This highlights that MPPI requires careful task-specific tuning before deployment. In this

thesis, the tuning process was performed manually and iteratively, which can be time-consuming and does not guarantee global optimality. Furthermore, a controller tuned for one trajectory may not generalize to other maneuvers without additional adjustment.

Simplified Dynamics in the Precision Study. The numerical precision experiments relied on a custom MuJoCo-based model running directly on the Jetson Orin. While this environment provided fine-grained control over arithmetic formats, it omitted several nonlinear effects. These simplifications mean that the strong numerical stability observed with `float16` reflects the behavior of an idealized model. Physical experiments did reveal additional dynamics—such as gradual altitude drift—that were absent from the simplified simulation. It remains possible that reduced precision may interact differently with these unmodeled effects.

Structured Indoor Testing. All real-world experiments were conducted indoors under VICON motion capture. This setup removes the uncertainties associated with outdoor flight nd provides state estimates with sub-millimeter accuracy. While appropriate for initial validation, the results therefore reflect performance in a structured environment with high-quality feedback. Additional work is required to determine how the controller behaves outdoors or under degraded sensing conditions.

Restricted Exploration of Low-Precision Formats. Because JAX does not natively support sub-`float16` formats, this thesis could only explore half-precision arithmetic as the lowest available option. Since `float16` produced no degradation in control performance, it is plausible that certain operations could operate reliably at even lower precision. However, evaluating 8-bit or fixed-point representations would require custom kernels or hardware prototypes, which were beyond the scope of this work.

Lack of Hardware-in-the-Loop Validation. Although Chapter 4 proposed guidelines for FPGA implementation, these recommendations were derived entirely from software-based experiments. No FPGA prototype was tested. Real hardware introduces constraints that are difficult to anticipate fully in simulation, including memory bandwidth limits, routing overhead, and timing closure. Thus, the reported benefits of reduced precision should

be considered indicative rather than definitive until validated on an actual FPGA.

Single Trajectory Type in Precision Experiments. The numerical precision study used a circular trajectory as the sole reference. While this trajectory is representative of smooth motion, it does not encompass high-jerk maneuvers or abrupt transitions that may stress the controller more aggressively. Whether reduced precision remains robust across such scenarios remains an open question.

5.3 Future Work

The findings of this thesis suggest several promising directions for future research. Some extend the current work to more realistic scenarios, while others explore how reduced-precision numerical strategies can be pushed further or adapted to different hardware platforms.

Automated and Adaptive Tuning. Given the strong dependence on hyperparameter choices, future work should explore automated tuning methods such as Bayesian optimization or gradient-free search. Another possibility is online adaptation, where the controller adjusts its parameters in real time based on observed performance. Adaptive horizon selection, for instance, could improve responsiveness during aggressive maneuvers and stability during steady flight.

Higher-Fidelity Simulation Models. To bridge the remaining gap between numerical studies and real-world performance, future work should incorporate more detailed dynamic models that include rotor lag, motor dynamics, aerodynamic drag, and voltage-dependent thrust. These models would provide a more reliable testbed for evaluating precision effects before conducting physical experiments.

Exploration of Sub-16-Bit Arithmetic. Since `float16` performed reliably, the next logical step is to investigate whether certain stages of the MPPI pipeline can operate at 8-bit or fixed-point precision. This would require analyzing the dynamic ranges of all variables and designing appropriate

scaling strategies. Ultimately, such evaluations may need to be performed on actual hardware.

Mixed-Precision Architectures. Another promising direction is to assign different precisions to different components of the algorithm. Trajectory propagation and cost evaluation—where most computation occurs—might run at low precision, while weight computation or control averaging could use higher precision to avoid numerical issues. This staged approach could yield better performance-per-resource than uniform `float16`.

Direct FPGA Implementation and Benchmarking. A key step forward is implementing MPPI on an FPGA and directly measuring latency, throughput, and resource utilization. Such experiments would clarify how many rollouts can be processed in parallel, how precision choices impact clock frequency, and whether the controller can sustain update rates beyond 100 Hz. This would also verify whether the precision-related speedups observed on GPU transfer to reconfigurable hardware.

Multi-Agent and Cooperative Control. Extending MPPI to multi-robot systems is another interesting direction. Reduced-precision computation could be especially beneficial in decentralized control architectures, where each agent must operate within strict timing constraints.

Closing Remarks. Taken together, these extensions point toward a broader research agenda aimed at making sampling-based predictive control practical for real-time aerial robotics. The central insight of this thesis—that numerical precision can be strategically reduced without compromising control quality—provides a strong foundation for both software-based improvements and hardware-accelerated implementations.

Bibliography

[1] H. J. Kappen. «Path integrals and symmetry breaking for optimal control theory». In: *Journal of Statistical Mechanics: Theory and Experiment* 11 (Nov. 2005), P11011–P11011 (cit. on pp. 6–11, 22).

[2] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. «A Generalized Path Integral Control Approach to Reinforcement Learning». In: *Journal of Machine Learning Research* 11 (2010), pp. 3137–3181 (cit. on pp. 6, 10, 11, 22).

[3] Grady Robert Williams. «Model Predictive Path Integral Control: Theoretical Foundations and Applications to Autonomous Driving». Accepted: 2020-05-20T16:57:06Z. PhD thesis. Atlanta, GA: Georgia Institute of Technology, Mar. 2019 (cit. on p. 6).

[4] Hilbert J. Kappen. «An introduction to stochastic control theory, path integrals and reinforcement learning». In: *AIP Conference Proceedings*. Vol. 887. AIP, Feb. 2007, pp. 149–181 (cit. on pp. 8, 22).

[5] Evangelos Theodorou, Yuval Tassa, and Emo Todorov. «Stochastic Differential Dynamic Programming». In: *Proceedings of the 2010 American Control Conference (ACC)*. IEEE, 2010, pp. 1125–1132 (cit. on pp. 10, 22).

[6] Grady Williams, Andrew Aldrich, and Evangelos Theodorou. «Model Predictive Path Integral Control using Covariance Variable Importance Sampling». In: *CoRR* (Sept. 2015) (cit. on p. 11).

[7] Grady Williams, Nolan Wagener, Brian Goldfain, Paul Drews, James M. Rehg, Byron Boots, and Evangelos A. Theodorou. «Information Theoretic MPC for Model-Based Reinforcement Learning». In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1714–1721 (cit. on pp. 11, 22).

[8] Robert Mahony, Vijay Kumar, and Peter Corke. «Multirotor Aerial Vehicles: Modeling, Estimation, and Control of Quadrotor». In: *IEEE Robotics & Automation Magazine* 19.3 (2012), pp. 20–32 (cit. on pp. 12, 14).

[9] Samir Bouabdallah. «Design and Control of Quadrotors with Application to Autonomous Flying». PhD thesis. École Polytechnique Fédérale de Lausanne (EPFL), 2007 (cit. on pp. 12, 14).

[10] G. M. Hoffmann, H. Huang, S. Waslander, and C. Tomlin. «Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment». In: *AIAA Guidance, Navigation and Control Conference*. 2007 (cit. on pp. 12, 14).

[11] Randal W. Beard and Timothy W. McLain. *Small Unmanned Aircraft: Theory and Practice*. Princeton University Press, 2008 (cit. on p. 12).

[12] Joan Sola. «Quaternion kinematics for the error-state Kalman filter». In: *arXiv preprint arXiv:1711.02508* (2017) (cit. on p. 14).

[13] G. Corsini. *TeleKyb3 – A Research Framework for Aerial Physical Interaction and Manipulation*. Presentation, Journée Drones 2024. Accessed from internal documentation. 2024 (cit. on pp. 14, 24).

[14] LAAS–CNRS. *telekyb3 Project Page*. `https://git.openrobots.org/projects/telekyb3`. Accessed: 2025-07-24. 2024 (cit. on pp. 14, 24).

[15] G. Corsini, M. Jacquet, H. Das, A. Afifi, D. Sidobre, and A. Franchi. «Nonlinear Model Predictive Control for Human-Robot Handover with Application to the Aerial Case». In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2022, pp. 7597–7604. DOI: `10.1109/IROS47612.2022.9981045`. URL: `https://ieeexplore.ieee.org/document/9981045` (cit. on p. 15).

[16] T. Lee, M. Leok, and N. H. McClamroch. «Geometric Tracking Control of a Quadrotor UAV on SE(3)». In: *49th IEEE Conference on Decision and Control (CDC)*. 2010, pp. 5420–5425. DOI: `10.1109/CDC.2010.5717652` (cit. on p. 16).

[17] G. Pannetier, P. Sopasakis, and M. Schlegel. *ProxSuite: High-performance Quadratic Programming for Robotics*. `https://github.com/Simple-Robotics/proxsuite`. 2023 (cit. on p. 17).

[18] Grady Williams, Paul Drews, and Evangelos Theodorou. «Model Predictive Path Integral Control: From Theory to Parallel Computation». In: *Journal of Guidance, Control, and Dynamics* 41.12 (2018), pp. 2453–2469. DOI: 10.2514/1.G003776 (cit. on pp. 22, 26, 44).

[19] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. «Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model Predictive Control». In: *IEEE International Conference on Robotics and Automation (ICRA)* (2018), pp. 7559–7566. DOI: 10.1109/ICRA.2018.8463189 (cit. on p. 23).

[20] Michael Schenk and Raphael Schilling. «Real-Time Model Predictive Path Integral Control for Quadrotors». In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 6871–6878. DOI: 10.1109/LRA.2020.3010845 (cit. on p. 23).

[21] Antonio Franchi, Giovanni Corsini, Maik Jacquet, Hamidreza Das, and Didier Sidobre. «Nonlinear Model Predictive Control for Human–Robot Handover with Application to the Aerial Case». In: *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2022, pp. 7597–7604. DOI: 10.1109/IROS47612.2022.9981045 (cit. on p. 23).

[22] Hyung Lee, Minsoo Kim, and Jinwoo Lee. «GPU-Accelerated Model Predictive Path Integral Control for Quadrotor Systems». In: *IEEE Access* 9 (2021), pp. 112345–112356. DOI: 10.1109/ACCESS.2021.3103274 (cit. on pp. 24, 44).

[23] Pranav Bhardwaj, Karthik Elamvazhuthi, and Evangelos Theodorou. «Fast and Robust Sampling-Based Model Predictive Control Using GPU Acceleration». In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pp. 4325–4332. DOI: 10.1109/ICRA48891.2023.10161345 (cit. on p. 24).

[24] Grady Williams and Evangelos Theodorou. «Model Predictive Path Integral Control using Covariance Variable Importance Sampling». In: *IEEE Robotics and Automation Letters* (2016) (cit. on p. 26).