COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

Real - Time Trajectory Optimization of a Quadcopter Using Model Predictive Control

ME-493: Trajectory Optimization Algorithms

Author:
KHUSHANT KHURANA

Professor:
DR. LUCHTENBURG

May 15, 2025

CONTENTS

Real - Time Trajectory Optimization of a Quadcopter Using Model Predictive Control

ME-493: Trajectory Optimization Algorithms

I. Abstract

In this project, a dynamic programming-based approach is integrated with Model Predictive Controller (MPC) to perform real-time trajectory optimization on the Crazyflie 2.1 quadcopter. Unlike conventional methods such as A* that generate waypoint trajectories offline, this project seeks to continuously solve a nonlinear optimization problem to compute optimal control inputs for the quadcopter. Doing so enables the system to adapt to dynamic environments and correct its trajectory based on accumulated tracking errors. This MPC based approach is first validated in simulation using simple test cases and then a full-scale 6 DOF quadcopter model. Subsequently, hardware experiments are conducted on the Crazyflie platform, where Robot Operating System (ROS) is used to integrate all hardware components and interface with the MPC solver.

II. Introduction

Trajectory optimization is a fundamental tool used to compute the optimal behavior of dynamical systems over time. It finds widespread applications in robotics and aerospace, such as generating energy-efficient walking gaits for legged robots or determining fuel-optimal paths for spacecraft. The process involves minimizing a cost function—such as time, energy, or control effort—subject to the system's dynamic and physical constraints. Trajectory optimization lies within the broader scope of optimal control and can be tackled using various techniques, including dynamic programming, Pontryagin's Maximum Principle, and reinforcement learning. In this project, a dynamic programming-based approach is employed within a Model Predictive Controller (MPC) framework to optimize the trajectory of a quadcopter.

MPC is an optimization-based control strategy that forecasts the system dynamics over a prediction horizon by simulating multiple possible control input sequences. It then solves an optimization problem to find the optimal sequence, but only applies the first control input before repeating the process at the next time step. The prediction is governed by an embedded mathematical model for the dynamics which is usually given as a list of differential equations. In the context of trajectory optimization, MPC can be viewed as performing trajectory generation and tracking simultaneously at each step. Unlike other control strategies such as PID or Linear Quadratic Regulator (LQR), MPC can explicitly handle constraints and anticipate future behavior by leveraging predictions across the horizon.

In this project, the MPC is formulated as a nonlinear program and solved using an interior-point optimizer (IPOPT). The optimization problem is modeled using CasADi [1] in Python, and the Robot Operating System 2 (ROS 2) is used as the middleware for hardware integration and testing.

III. Related Work

Much of the prior work on trajectory optimization for the Crazyflie platform has focused on either tracking a pre-defined trajectory or generating a fixed trajectory offline using motion planning algorithms such as Rapidly Exploring Random Trees (RRT) and A*. These approaches typically decouple trajectory planning from execution, thereby limiting the system's ability to respond to dynamic changes in the environment.

A notable exception is TinyMPC [2], developed by graduate researchers at Carnegie Mellon University for the Crazyflie platform. It enables simultaneous trajectory generation and tracking using a linearized model around a hover configuration to achieve real-time performance. While this reduces computational load, it limits the controller's effectiveness to maneuvers near the linearization point, restricting its ability to handle aggressive or highly nonlinear trajectories.

IV. Problem Formulation

A. Formulating MPC as an Optimization Problem

The standard structure of an optimization problem consists of a cost function which is to be minimized while subjecting to a certain set of constraints. The stage cost for a single MPC computation is given as follows:

$$\ell(\mathbf{x}, \mathbf{u}) = \|\mathbf{x}_u - \mathbf{x}^r\|_Q^2 + ||\mathbf{u}^r||_R^2 \tag{1}$$

'Q' and 'R' in Equation 1 represent the cost associated with the error in state (the difference between the desired and current state) and the control input, respectively. Both are square matrices and define the penalties on the individual quantities. For example, a high value in Q represents a higher cost for the corresponding error state, and the solver will prioritize minimizing it more compared to others. Similarly, a higher value in R corresponds to a greater penalty on the control inputs.

While minimizing the cost function, optimization solvers compute its gradient and Hessian matrices. For efficient and stable convergence, it is preferable that the cost function has a strict global minimum (resembling the shape of a cone in

3D space). This can be ensured by making Q and R positive definite (PD) matrices, meaning their eigenvalues are greater than or equal to zero.

The finite-horizon optimal control problem over successive steps is defined in Equation 2. As explained earlier, MPC not only minimizes the cost function but also enforces constraints—primarily the system dynamics. Since MPC predicts across the horizon steps, it ensures that the predicted states are feasible according to the embedded mathematical model. Additionally, it respects any constraints on the control inputs and the bounds on the state vector.

$$\begin{aligned} \min_{\mathbf{u}} \; & J_N(\mathbf{x}_0, \mathbf{u}) = \sum_{k=0}^{N-1} \ell(\mathbf{x}(k), \mathbf{u}(k)) \\ \text{subject to} \; & \mathbf{x}_u(k+1) = \mathbf{f}(\mathbf{x}_u(k), \mathbf{u}(k)), \quad \text{(dynamics)} \\ & \mathbf{x}_u(0) = \mathbf{x}_0, \\ & \mathbf{u}(k) \in U, \quad \forall k \in [0, N-1], \\ & \mathbf{x}_u(k) \in X, \quad \forall k \in [0, N]. \end{aligned} \tag{2}$$

B. Converting the Optimization Problem into a Non Linear Program (NLP)

Before framing MPC as an optimization problem, it is important to understand the type of optimization problem at hand. Since MPC involves nonlinear constraints due to the system dynamics, it is formulated as an NLP (NonLinear Program) with a quadratic cost function. A standard formulation of the NLP is given in Equation 3. Φ is the cost function that the optimization routine aims to minimize, while g_1 and g_2 represent the constraints applied to the problem.

$$\begin{aligned} \min_{\mathbf{w}} \quad & \Phi(\mathbf{w}) \\ \text{subject to} \quad & g_1(\mathbf{w}) \leq 0 \quad \text{(inequality constraints)} \\ & g_2(\mathbf{w}) = 0 \quad \text{(equality constraints)} \end{aligned} \tag{3}$$

In the context of MPC, the dynamic constraints are represented as equality constraints (g_1) and the constraints on the actuation and states are represented as inequality constraints (g_2).

C. Transcription Methods for Dynamics Constraints

Although MPC solves the problem in real-time, it is still formulated as a discrete problem. Consequently, the dynamics constraints cannot be represented as continuous-time differential equations and must be discretized. This process of converting the continuous problem into a discrete nonlinear program is called transcription. Various transcription methods exist, such as single shooting, multiple shooting, direct collocation etc. In this project, both single and multiple shooting methods are tested in simulation, while the latter is used for actual hardware testing.

In single shooting, only the control inputs are treated as decision variables, and the system dynamics are integrated forward from the known initial state. In multiple shooting, on the other hand, the dynamics at each horizon step are treated as equality constraints, dividing the time into multiple segments. This allows the state variables to be treated as decision variables as well, resulting in greater accuracy. The cost function for multiple shooting is given in Equation 4, where the optimization variables include both 'x' (state) and 'u' (control).

$$\min_{\mathbf{u},\mathbf{x}} J_N(\mathbf{x}_0, \mathbf{u}) = \sum_{k=0}^{N-1} \ell(\mathbf{x}(k), \mathbf{u}(k)) \tag{4}$$

Using multiple shooting increases the size of the problem (since there are more optimization variables), but it lifts the problem to a higher dimension, making convergence significantly easier. In addition, multiple shooting allows for the initialization of state variables, enabling better initial guesses for the solver, which simplifies the overall optimization process.

V. Simulation Results

Before conducting actual hardware testing, simulations are performed using a basic example of a toy car with a simplified 3-state model, and a quadcopter using the full 6-DOF state model (which is then later used for the hardware testing). The live example of all the simulation runs and the corresponding code can be found in my GitHub repository [3].

A. 2D Toy Car

In this simulation, trajectory optimization is performed for a non-holonomic mobile robot with three degrees of freedom (DOF). The kinematic model and control vector for the robot are provided in Equation 5. The state variables x, y, and θ represent the robot's position in the x-direction, position in the y-direction, and yaw angle, respectively. The control variables v and ω correspond to the linear and angular velocities of the robot.

$$\begin{aligned} \dot{x} &= v \cdot \cos(\theta) \\ \dot{y} &= v \cdot \sin(\theta) \\ \dot{\theta} &= w \\ \mathbf{u} &= \begin{bmatrix} \mathrm{v} \\ \mathrm{w} \end{bmatrix} \end{aligned} \tag{5}$$

The constraints for the control inputs are defined as $\omega = \pi/4$ rad/s and $v = 0.6$ m/s. The constraints on the state variables are given by $-2 \leq x, y \leq 2$ and $-\infty \leq \theta \leq \infty$. Note that these constraints on the state variables are enforced only when the multiple shooting method is used. With the initial state of the robot specified as $[x = 0\text{ m},\ y = 0\text{ m},\ \theta = 0^\circ]$ and the final state as $[x = 1\text{ m},\ y = 1\text{ m},\ \theta = \pi^\circ]$, the trajectory generated using a horizon of 5 steps is shown in Figure V.1, and the corresponding optimized control inputs are shown in Figure V.2.

As expected, the MPC implementation respects the constraints on the control variables by saturating them at $\pi/4$ rad/sec and 0.6 m/sec for w and v respectively.

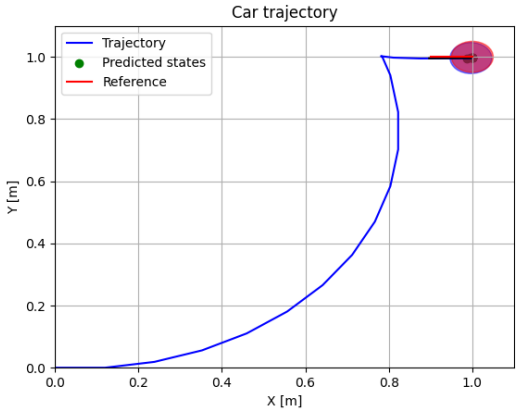

Fig. V.1: Optimized trajectory of the car

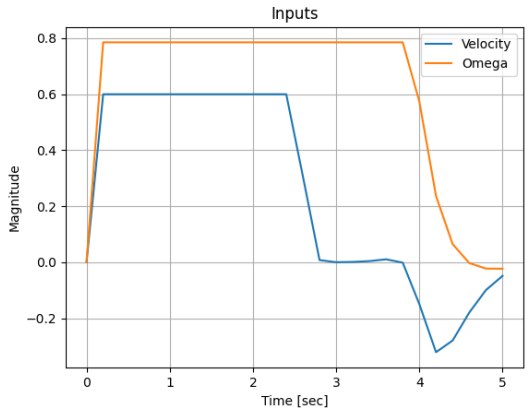

Fig. V.2: Optimized linear and angular velocity commands computed by the MPC

B. 2D Toy Car with Obstacles

The trajectory generation problem for the mobile robot is further extended to include obstacle avoidance, where the robot must reach its target state while avoiding collisions with obstacles. To ensure this, the optimization problem is constrained to maintain the signed distance between the robot and each obstacle to be greater than a prescribed minimum distance d, as shown in Equation 6.

$$d \geq \sqrt{(x_{\text{obs}} - x_{\text{robot}})^2 + (y_{\text{obs}} - y_{\text{robot}})^2} \tag{6}$$

However, it needs to be understood that formulating this constraint as such will make the optimization problem difficult to solve (the square root is making the problem more non-linear). Accordingly, the constraint is formulated in a different expression as shown in Equation 7:

$$d^2 - \left((x_{\text{obs}} - x_{\text{robot}})^2 + (y_{\text{obs}} - y_{\text{robot}})^2\right) \geq 0 \tag{7}$$

Similar initial and target states are used as in the previous example, but the constraint on the angular velocity ω is increased to $\pi/2$ rad/s to allow for quicker maneuverability. The obstacles are modeled as circles with a radius of 0.1 m, and the robot (car) is also assigned a radius of 0.1 m. The resulting trajectory, along with the optimized control inputs, are shown in Figure V.3 and Figure V.4.

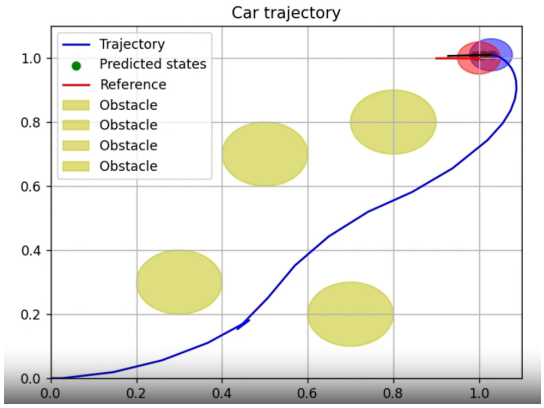

Fig. V.3: Optimized trajectory of the car to reach its final state without colliding into obstacles

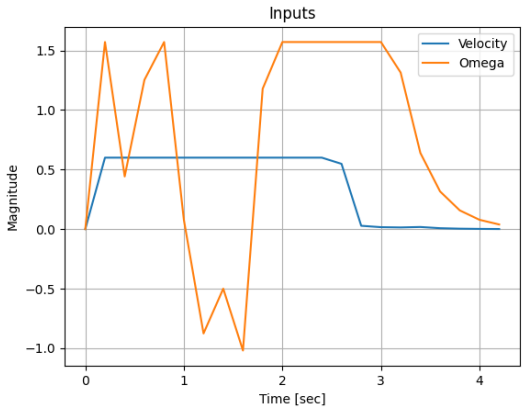

Fig. V.4: Optimized linear and angular velocity commands computed by the MPC

C. 6 DOF Quadcopter with Obstacles

After performing trajectory optimization on a simple system like the 3 DOF robot car, a similar procedure is performed on a full state 6 DOF quadcopter.

1) Mathematical Model for the Dynamics: To start designing the MPC, the mathematical model for the rigid body motion of the quadcopter is derived using Newton-Euler rigid body equations, given in Equation 8

$$\begin{bmatrix} \mathbf{F} \\ \mathbf{M} \end{bmatrix} = \begin{bmatrix} m\dot{\mathbf{v}} + m\omega \times \mathbf{v} \\ \mathbf{I}\dot{\omega} + \omega \times (\mathbf{I}\omega) \end{bmatrix} \tag{8}$$

The state vector for the 6 DOF rigid body motion is given in Equation 9 and the corresponding control vector for the quadcopter is given in Equation 10. Since the forces and moments require the second derivative of position, the state vector also contains the derivatives of the 6 DOF's.

$$\mathbf{x} = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & \phi & \theta & \psi & p \\ q & r & & & & & & & & \end{bmatrix} \tag{9}$$

$$\mathbf{u} = \begin{bmatrix} T & \tau_\phi & \tau_\theta & \tau_\psi \end{bmatrix} \tag{10}$$

The notation for the all the symbols in the state and control vector is given in Table I.

TABLE I: Notation for 12-DOF States and Control Inputs

Symbol	Description
x, y, z	Position in inertial frame
$\dot{x}, \dot{y}, \dot{z}$	Linear velocity in inertial frame
ϕ, θ, ψ	Roll, pitch, and yaw angles
p, q, r	Angular velocity around body axes (roll, pitch, yaw rates)
T	Total thrust
$\tau_\phi, \tau_\theta, \tau_\psi$	Torques about roll, pitch, and yaw axes

The individual equations of motions (12) for translation and rotational velocities and accelerations are given below. As expected, these motions are highly non-linear and coupled. Although this risks the solver to increase the time complexity of the problem but it is regarded as a fair trade off for accuracy. It is to be noted that this mathematical model disregards a lot of other non-linear effects on the quadcopter, like ground effects, aerodynamics, motor curves etc. Accordingly, it is important to crank up the accuracy of this bare-bone rigid body model as much as possible.

$$\dot{\mathbf{x}} = \begin{bmatrix} (\cos\theta\cos\psi)\dot{x} - \sin\psi\dot{y} + (\sin\theta\cos\psi)\dot{z} \\ (\cos\theta\sin\psi)\dot{x} + \cos\psi\dot{y} + (\sin\theta\sin\psi)\dot{z} \\ -\sin\theta\dot{x} + \cos\theta\dot{z} \\ \frac{\text{thrust}}{m}(\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi) - (q\dot{z} - r\dot{y}) \\ \frac{\text{thrust}}{m}(\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi) + (p\dot{z} - r\dot{x}) \\ \frac{\text{thrust}}{m}(\cos\phi\cos\theta) - g + (p\dot{y} - q\dot{x}) \\ p + q\sin\phi\tan\theta + r\cos\phi\tan\theta \\ q\cos\phi - r\sin\phi \\ \frac{q\sin\phi + r\cos\phi}{\cos\theta} \\ \frac{I_y - I_z}{I_x}qr + \frac{\tau_\phi}{I_x} \\ \frac{I_z - I_x}{I_y}pr + \frac{\tau_\theta}{I_y} \\ \frac{I_x - I_y}{I_z}pq + \frac{\tau_\psi}{I_z} \end{bmatrix} \tag{11}$$

2) Parameters and Constraints: Since the hardware testing is to be performed on Crazyflie 2.1 quadcopter, the constraints for the MPC problem are derived for this specific drone, given in Table II

Parameter	Value	Parameter	Value
gravity	9.81	v_{max}	1
drone radius	0.1	v_{min}	-v_{max}
I_x	2.4×10^{-5}	w_{max}	10.47
I_y	I_x	w_{min}	-w_{max}
I_z	3.2×10^{-5}	mass	0.027
bounds	∞	thrust max	$1.9 \times m \times$ gravity
v_{max}	1	thrust min	0
τ_{max}	0.0097	τ_{min}	-τ_{max}

TABLE II: Parameters and values

All of these parameters are in SI units and are derived from Crazyflie documentation. The inertia of the quadcopter are derived from [4], velocity constraints from [5] and thrust to weight ratio from [6]. The torques are derived from a basic calculation of of force generated by a single motor ($thrust_{\text{max}}/4$) multiplied by the length of one of the arms (0.046 m).

3) Runge Kutta 4 Implementation: In order to integrate the MPC's model for the successive horizon steps, Runge Kutta 4 (RK4) integration method is implemented. Earlier iterations of the simulation testing relied on first order Euler integration which proved to be highly unstable. RK4, on the other hand, is generally considered superior to Euler's method due to its significantly higher accuracy and stability. While Euler's method uses a simple, single slope estimate to predict the next state, RK4 calculates multiple slopes (at the current state, and at points midway through the interval), and then takes a weighted average of those estimates. This leads to a much more accurate approximation of the system's behavior, especially for complex, nonlinear systems. The exact implementation is shown in Equation 12

$$\begin{aligned} k_1 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n, t_n), \\ k_2 &= \Delta t \cdot \mathbf{f}\left(\mathbf{y}_n + \frac{1}{2}k_1, t_n + \frac{1}{2}\Delta t\right), \\ k_3 &= \Delta t \cdot \mathbf{f}\left(\mathbf{y}_n + \frac{1}{2}k_2, t_n + \frac{1}{2}\Delta t\right), \\ k_4 &= \Delta t \cdot \mathbf{f}(\mathbf{y}_n + k_3, t_n + \Delta t), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right). \end{aligned} \tag{12}$$

While RK4 does provide more accuracy and stability in its forward integration, it does slow down the solver because of its four step computation process. This, however, is disregarded in building the MPC since the model's accuracy was required to be high, especially after neglecting a lot of real-world effects.

4) Results: The initial state of the quadcopter is defined to be a 12 x 1 vector of all 0's and the final state is defined as a vector of all 0's except $x = 1m, y = 1m, z = -1m$. The obstacles are defined to be sphere with radius of 1 m. The optimized trajectory, along with the orientation and optimized inputs are shown in Figure V.5, Figure V.6, and Figure V.7.

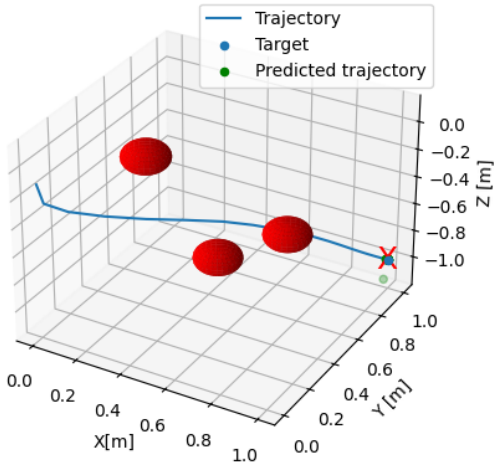

Fig. V.5: Optimized trajectory of the quadcopter through the obstacles

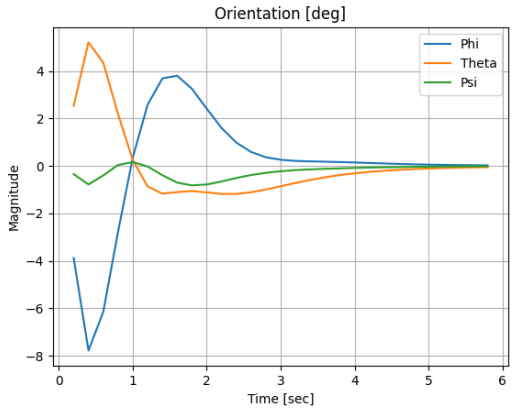

Fig. V.6: The orientation of the drone around its 3 axes as it flew across the obstacles and reached its target state

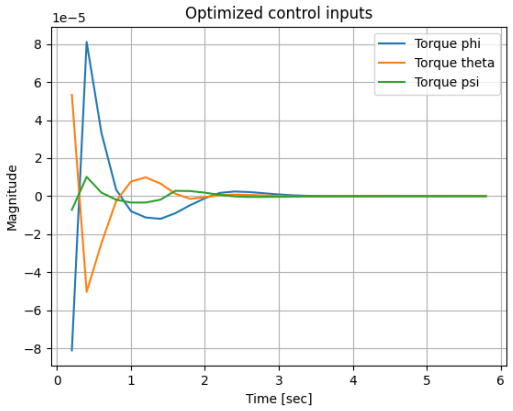

Fig. V.7: The optimized inputs calculated by the MPC

VI. Hardware Testing

After having success with the digital twin of Crazyflie 2.1, the focus is shifted to hardware testing on the actual quadcopter. All the corresponding code can be found in my Github Repository [3].

A. Motion Capture System

Unlike in simulation, where the mathematical model developed for the quadcopter in Equation 11 is used as the plant model, the true states of the Crazyflie must be continuously updated in real time as it flies toward the target. Although the Crazyflie is equipped with an Inertial Measurement Unit (IMU) and supports logging of position and orientation data, this data is often noisy and may lead to inaccuracies. To address this, the VICON tracking system [7] in the Dynamics and Controls Lab is used to obtain accurate measurements of the position ($[x, y, z]$) and orientation ($[\phi, \theta, \psi]$) of the quadcopter.

Four reflective markers are attached to both the quadcopter and the RC car (used to define the target state), allowing the motion capture system to track them precisely, as shown in Figure VI.5. The full setup of the motion capture system along with the tracking cameras is shown in Figure VI.6

Fig. VI.1: Crazyflie quadcopter with markers

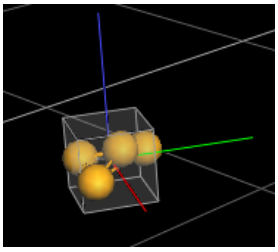

Fig. VI.2: Crazyflie's rigid body tracking configuration in the VICON system

Fig. VI.3: RC car (target) with markers

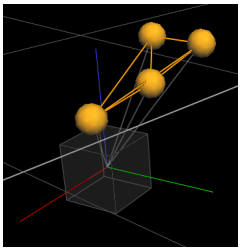

Fig. VI.4: RC Car's rigid body tracking configuration in the VICON system

Fig. VI.5: Images of the Crazyflie quadcopter and associated VICON system setup.

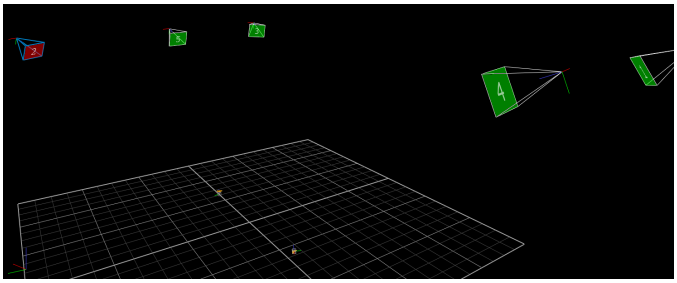

Fig. VI.6: The entire VICON setup with the crazyflie, RC Car, and the cameras

Despite giving accurate position and orientation of the Crazyflie, the VICON system does not track the velocities which are required for the true state calculations. Accordingly, dirty differentiation - finite step derivative with a low pass alpha filter - method with alpha value of 0.3 is used to find all the velocities, shown in Equation 13.

$$\dot{x}_{\text{est}}(t) = \alpha \cdot \dot{x}_{\text{est}}(t-1) + (1-\alpha) \cdot \frac{x(t) - x(t-1)}{\Delta t} \quad (13)$$

B. Middleware - Robot Operating System

The flow of information during hardware testing is shown in Figure VI.7. Executing all five steps within a single program is computationally intensive. Additionally, the Crazyflie requires a constant heartbeat (input command) to prevent shutdown, while the average lag of the MPC solver is approximately 0.012 seconds. As a result, both operations cannot be handled by a single program simultaneously.

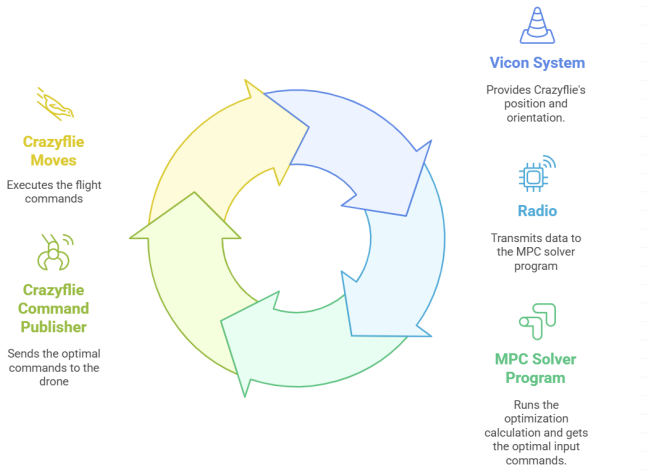

Fig. VI.7: Chronological order of the flow of information between hardware and software components

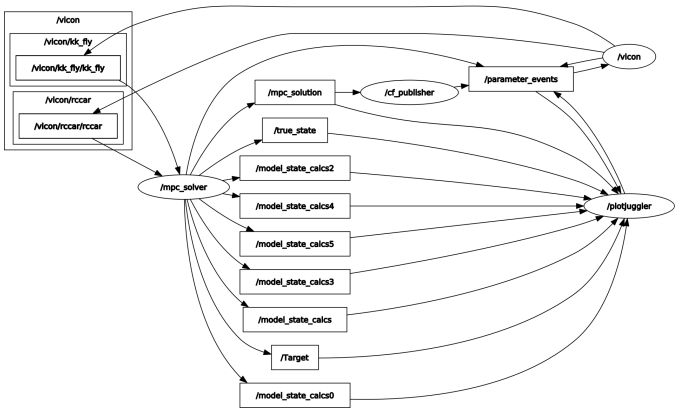

Fig. VI.8: RQT graph of the ROS2 architecture: nodes, topics, and the flow of information between the two

To enable simultaneous operation of all five components without interference, Robot Operating System 2 (ROS2) is used as the middleware to connect each task. ROS2, written in C++, uses a node-based architecture where each node represents a part of the program, and communication between nodes is handled via a publisher-subscriber model [8]. Four ROS2 packages are created for the following purposes:

- *vicon_receiver*: converts the VICON radio messages to ROS2 topics. These topics can be then subscribed to for getting the position and orientation data.
- *traj_opt*: contains the two nodes:
  - *mpc_solver*: This node subscribes to the motion capture system and solves the optimization problem. Once an optimal solution has been found it sends it to the *cf_publisher* node.
  - *cf_publisher*: This node sends the MPC solver's solution to the crazyflie at constant frequency in order to maintain the heartbeat.
- *custom_msgs*: Defines custom topics and messages to allow interaction between the two nodes.
- *launcher*: Creates the launch file for starting the ROS2 architecture.

In order to plot all the data, PlotJuggler - ROS2 custom package for plotting - is used. The RQT graph (ROS2 architecture with the nodes and topics) is shown in Figure VI.8.

C. Experimental Results

Through the initial tests, the MPC solver is able to produce correct results, but the optimized thrust solution keeps fluctuating between extremes. Upon reviewing the data, it is discovered that the MPC's integration time step is smaller than its solve time. As a result, the predicted trajectory consistently overshoots, causing the quadcopter to exhibit a hopping motion toward the target, since the MPC keeps re-initializing. A video of this test can be found in my GitHub repository [3].

This issue is fixed in later iterations. Another problem is identified with the initial guess for the optimization problem. Initially, the guess vector is set to all zeros, which prevents the solver from finding a feasible trajectory. To address this, the position components (x, y, z) of the initialization are determined by discretizing the parametric path between the initial position and the target. Additionally, the cost function is simplified by including only the position states (x, y, z), which significantly improves solver speed. With these adjustments, the Crazyflie is able to execute a very smooth maneuver toward the target state. A video of this test is also available in my GitHub repository [3].

The XY trajectory graph of the Crazyflie during the test run is shown in Figure VI.9. The red point signifies the target, RC car here. The true state of the Crazyflie along with the estimated state of the 5 horizon steps is also shown. As expected, the trajectory labeled */model_state_calcs5/state[0:1]* is closest to the target because it is the trajectory of the 5th horizon step.

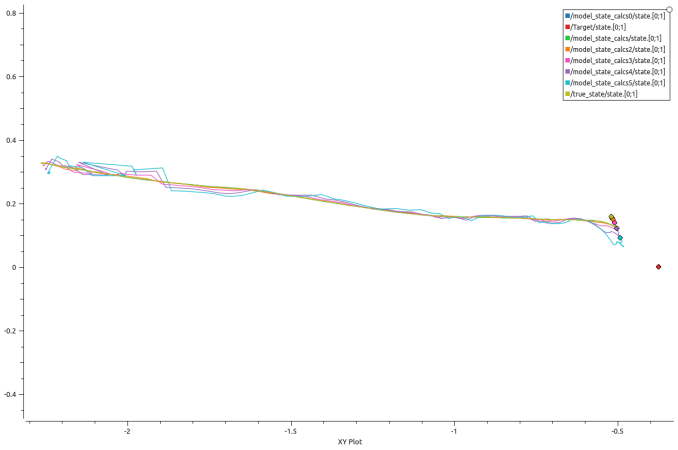

Fig. VI.9: A XY graph showing the 6 predicted trajectories of the MPC, the true trajectory generated from the VICON system, and the target. As seen, all the trajectories are converging to the target hence adding validation to the solutions generated by the MPC solver.

Figure VI.10 shows the true state (x and y position) from the graph and the predicted state from the MPC. Both states

overlay each other which adds proof to the mathematical model embedded into the MPC.

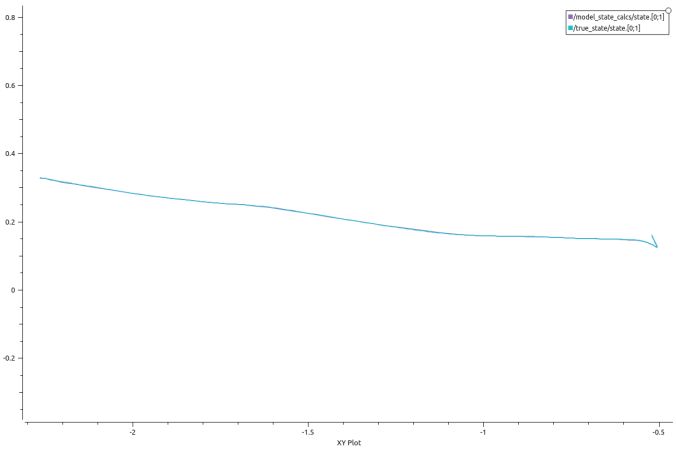

Fig. VI.10: A comparison of the true XY trajectory and the predicted XY trajectory. As expected, the trajectories completely overlay each other, adding validation to the mathematical model embedded into the MPC solver.

D. Discussion

The time delay between the MPC solver and the Crazyflie's movement proved to be really challenging. Despite using a simpler cost function with only the positions as the error states, the average solve time is about 0.011 sec. Accordingly, as the MPC would start solving the optimization problem for Crazyflie's current state, the quadcopter would move by the time next optimized solution is received. An example of this time delay can be seen in Figure VI.11 where the predicted x velocity tries to follow the true x velocity but has much less precision and overshoots the true velocity.

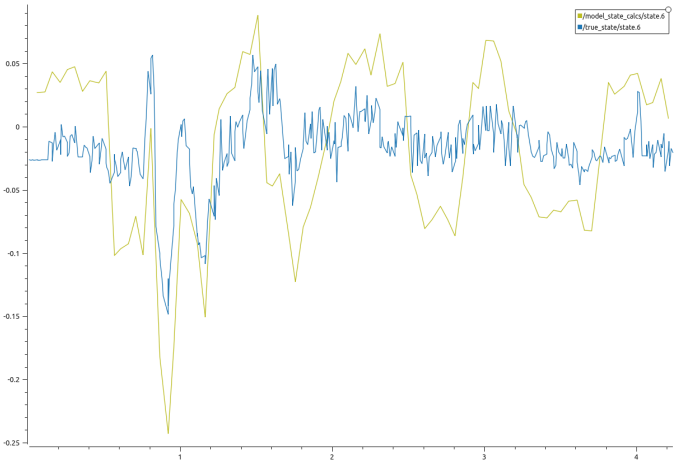

Fig. VI.11: A comparison of the true and estimated x velocity of the crazyflie.

After iterating through system constraints such as linear and angular velocity, a suitable compromise is found: the quadcopter moves slowly enough to stay within the prediction horizon without overshooting, allowing the MPC to compute optimal solutions in time.

VII. Future Work

For future work, simpler models should be considered to improve MPC solver speed. While the current 6-DOF model allows the Crazyflie to reach its target, it is not fast enough to handle agile maneuvers such as obstacle avoidance. To address this limitation, the following directions can be explored:

A. Optimization Through Non-Linear Control

The bulk of the lag in the MPC comes from its simultaneous work on trajectory generation and tracking. Usually, these two processes are separated by generating trajectory offline (no time constraints) and then a different controller is put to work for following the point trajectory generated. A similar approach can be applied here. A common method for performing trajectory generation is using the discrete Hamilton-Jacobi-Bellman (HJB) model, which runs a recursive routine on the discrete time steps and minimizes the value function, shown in Equation 14

$$V_k(\mathbf{x}) = \min_{\mathbf{u}\in\mathcal{U}} \left[\ell(\mathbf{x},\mathbf{u}) + V_{k+1}\left(\mathbf{f}(\mathbf{x},\mathbf{u})\right)\right] \tag{14}$$

Trajectory generation using HJB is tried in simulation for the 2D robot car and the optimized trajectory along with the inputs are shown in Figure VII.1 and Figure VII.2, respectively. As expected, the optimized inputs match that of the MPC simulation.

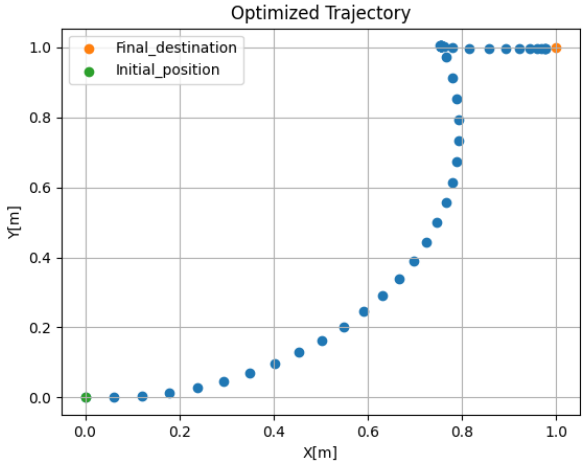

Fig. VII.1: Optimized trajectory generated by the solving the Hamilton-Jacobi-Bellman equation.

B. Using Lumped Models for MPC Dynamics

In addition to MPC computation time, a significant source of lag arises from the Crazyflie's motor dynamics. When the MPC requests a specific thrust, the motors exhibit an inherent delay in reaching the commanded setpoint. To account for this, a first-order transfer function can be used to model the lag in the rotational acceleration dynamics. This simplification approximates the system's response as a single exponential, neglecting the coupling between the roll, pitch, and yaw axes.

To estimate the time constant of this first-order lag, a test script was developed: the drone hovers for 2 seconds, then

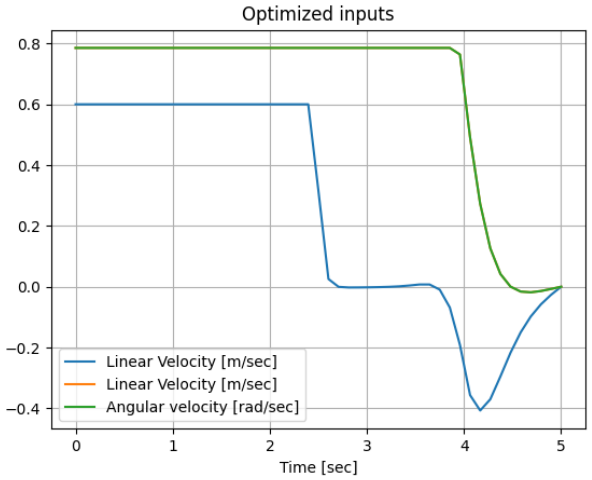

Fig. VII.2: The optimized inputs determined by HJB solver.

receives a 5-degree pitch step command. The response, shown in Figure VII.3, indicates a time constant of approximately 300 msec for the roll axis, which is assumed to be similar for pitch.

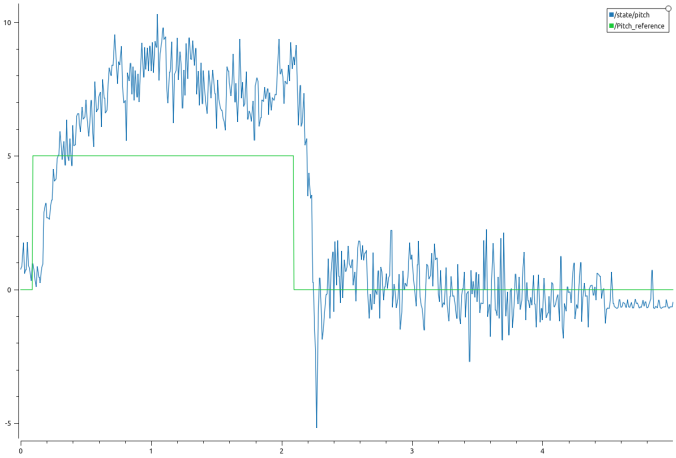

Fig. VII.3: Determination of the time constant through a step response

Using this lumped model wouldn't make the solver faster but introduce the time constant lag into the MPC's mathematical model; hence constraining the MPC from overshooting while determining its prediction steps.

References

[1] J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "Casadi: A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019.

[2] K. Nguyen, S. Schoedel, A. Alavilli, B. Plancher, and Z. Manchester, "Tinympc: Model-predictive control on resource-constrained microcontrollers," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2024.

[3] K. Khurana, *Trajectory-optimization*, https://github.com/khushant2001/Trajectory-optimization, Accessed: 2025-04-12, 2025.

[4] P. Landry *et al.*, "Planning and control for quadrotor flight through cluttered environments," *Robotics Research Center, Massachusetts Institute of Technology*, 2015. [Online]. Available: https://groups.csail.mit.edu/robotics-center/public_papers/Landry15.pdf.

[5] Bitcraze, *Bitcraze forum*, Accessed: 2025-04-11, 2025. [Online]. Available: https://forum.bitcraze.io/.

[6] Y. Chen and N. Perez-Arancibia, "Nonlinear adaptive control of quadrotor multi-flipping maneuvers in the presence of time-varying torque latency," Jan. 2019. DOI: 10.1109/IROS.2018.8594265.

[7] Vicon Motion Systems, *Vicon motion capture systems*, Accessed: 2025-05-03, 2024. [Online]. Available: https://www.vicon.com/.

[8] Open Source Robotics Foundation, *Robot operating system 2 (ros 2)*, Accessed: 2025-05-03, 2024. [Online]. Available: https://docs.ros.org/en/ros2/.