ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Project in space technology (EE-589)

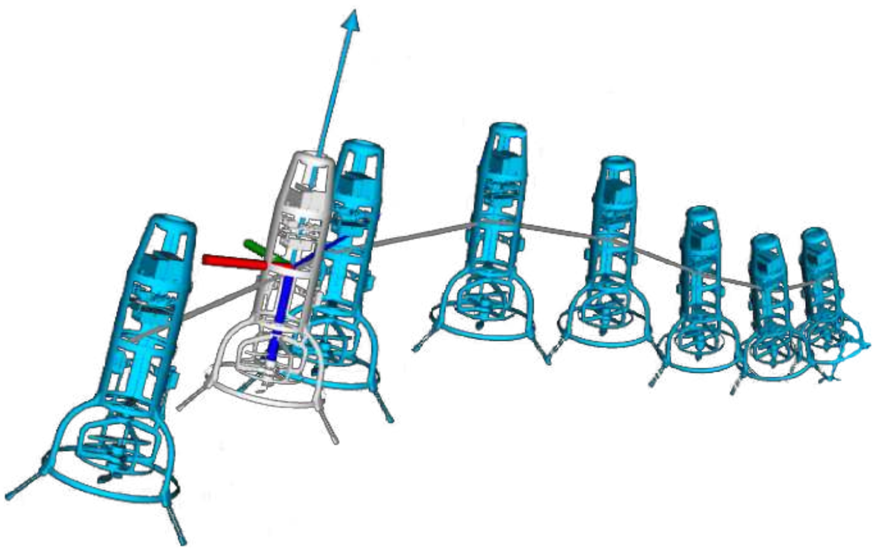
Energy-optimal embedded Nonlinear MPC for a thrust-vector-controlled drone

Automatic Control Laboratory

Author:
Tom Fähndrich (289106)
(Robotics master student)

Supervisors:
Johannes Waibel
(*Doctoral Assistant*)

Colin Jones
(*Associate Professor*)



Spring 2023

Contents

Acronyms

CoM	Center of Mass
EKF	Extended Kalman Filter
HiL	Hardware in the Loop
IMU	Inertial Measurments Unit
LAMI	LaOPT, Multiple shooting, IPOPT
LAMP	LaOPT, Multiple shooting, PIQP
MOMI	MX, OptiStack, Multiple Shooting, IPOPT
MPC	Model Predictive Control
ms	multiple shooting
NED	North, East, Down
NLP	Nonlinear Program
NMPC	Nonlinear Model Predictive Control
OCP	Optimal Control Problem
radau	Radau collocation
ROS	Robot Operating System
SOC	State Of Charge
TVC	Thrust Vector Controlled

1 Introduction

This project explores the development of an embedded Nonlinear Model Predictive Control (NMPC) for the second iteration of a thrust-vector-controlled (TVC) drone developed by the Automatic Control Laboratory. The unique control challenges of TVC rockets, marked by nonlinear dynamics and the need for high precision maneuvering, demand innovative control strategies. NMPC, with its inherent capacity to manage nonlinear dynamics and trajectory planning, offers a promising solution.
However, the implementation of NMPC on embedded systems is not straightforward due to the computational requirements of real-time optimization in optimal control for resources-constrained robotics application. This project aims to address this issue, developing a C++/ROS NMPC algorithm that balances computational efficiency, control performance, and hardware constraints. The focus is on achieving complex maneuvers with robustness, precision, and real-time responsiveness.
The second aspect of this project is to attempt to implement a strategy enabling the controller to minimize the energy consumption associated with the various trajectories. An effective implementation could have a major impact for space applications by minimizing fuel consumption, and prolonging mission life, thereby enhancing the feasibility and cost-effectiveness of space missions.

Note: This project was carried out in collaboration with Pietro Mello Rella, and most of the implementations and results are common to both.

2 Model identification

2.1 Rocket dynamics

The aim of this section is to provide a summary of the most important elements of the rocket model and dynamics, which will be used in the identification, NMPC implementation and simulations phases.

2.1.1 States

The drone's physical behavior is defined by its state space representation which is composed of its absolute position, velocity angular rate and attitude as follows:

- **Position** of the drone is expressed in the NED (North, East, Down) world inertial reference frame (w) $\mathbf{x}^w = \begin{bmatrix} x_\mathrm{N} & x_\mathrm{E} & x_\mathrm{D} \end{bmatrix}^T$ (m)
- **Translational velocity** $\mathbf{v}^w = \begin{bmatrix} v_\mathrm{N} & v_\mathrm{E} & v_\mathrm{D} \end{bmatrix}^T$ (m/s)
- **Rotation rate** of the body frame (b) with respect to the world frame $\boldsymbol{\omega}^b = \begin{bmatrix} \omega_x^b & \omega_y^b & \omega_z^b \end{bmatrix}^T$ (rad/s)
- **Attitude** expressed in quaternion $q = \begin{bmatrix} q_w & q_x & q_y & q_z \end{bmatrix}^T$ (-)

The complete state vector representation is the following :

$$\mathbf{x} = \begin{bmatrix} \mathbf{v}^w & \boldsymbol{\omega}^b & \mathbf{x}^w & q \end{bmatrix}^T =$$
$$\begin{bmatrix} v_\mathrm{N} & v_\mathrm{E} & v_\mathrm{D} & \omega_x^b & \omega_y^b & \omega_z^b & x_\mathrm{N} & x_\mathrm{E} & x_\mathrm{D} & q_w & q_x & q_y & q_z \end{bmatrix}^T \tag{1}$$

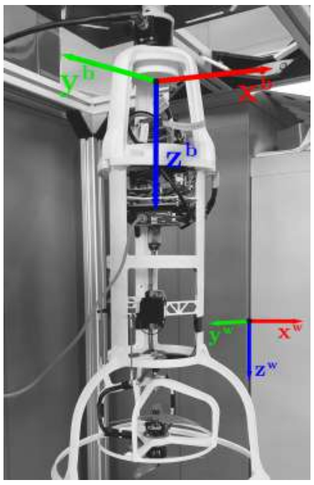


Figure 1: Body and world frame of the TVC drone

It is also important to note that the following Euler angles have been defined for this project and are used as references throughout this report: *Roll* around the x-axis, *Pitch* around the y-axis and *Yaw* around the z-axis.

2.1.2 Inputs

The drone is controlled by a gimbal oriented around 2 axes by 2 servo-motors on which a set of coaxial counter-rotating propellers is mounted. The movement of the servo-motors (dR, dP) controls the *Roll* and *Pitch* by orienting the force vector generated by the throttle (dF) due to rotation of the propellers, while the difference in speed between them (dF_{diff}) controls the rocket's *Yaw*. The input commands are contraint as follows:

$$dR, dP \in [-1, 1] \qquad (-)$$

$$dF \in [0,1] \qquad \text{(-)}$$

$$dF_{diff} \in [-1,1] \qquad \text{(-)}$$

With the input vector:

$$\mathbf{u} = \begin{bmatrix} dR & dP & dF & dF_{diff} \end{bmatrix}^T \tag{2}$$

The relations between these input commands and the resulting physical behavior of the drone in terms of gimbal orientation, generated thrust or rotation moment will be identified and discussed in the Section 2.2.

2.1.3 Thrust vectoring

To enable the drone to change its attitude and move sideways, the gimbal on which the propellers are mounted can be oriented. This orientation will modify the thrust vector components and thus generate lateral accelerations in addition to the ability to climb, descend or maintain altitude. The referential of the gimbal is also NED and defined as $(x^g,\ y^g,\ z^g)$.
dR input commands will result in a rotation θ_x of the gimbal around the x^g-axis and modify the *Roll* of the drone. dP input commands will result in a rotation θ_y of the gimbal around the y^g-axis and modify the *Pitch* of the drone. The setup can be observed in Figure 2. The orientation vector of the gimbal with respect to the body frame is the following:

$$\boldsymbol{o}_F^b = \begin{bmatrix} \sin\theta_y \cdot \cos\theta_x \\ -\sin\theta_x \\ \cos\theta_y \cdot \cos\theta_x \end{bmatrix} \tag{3}$$

The thrust vector and moment can now be obtained:

$$\mathbf{F}^b_{\text{thrust}} = -\boldsymbol{o}_F^b \cdot F_{\text{thrust}} \tag{4}$$

$$M^b_{thrust} = -\boldsymbol{o}_F^b \cdot T_z + \begin{bmatrix} 0 \\ 0 \\ r_z \end{bmatrix} \times \mathbf{F}^b_{\text{thrust}} \tag{5}$$

with $r_z = 17.8$ cm the distance between the gimbal joint and the center of mass (CoM) of the rocket, T_z the torque due to dF_{diff} inputs.

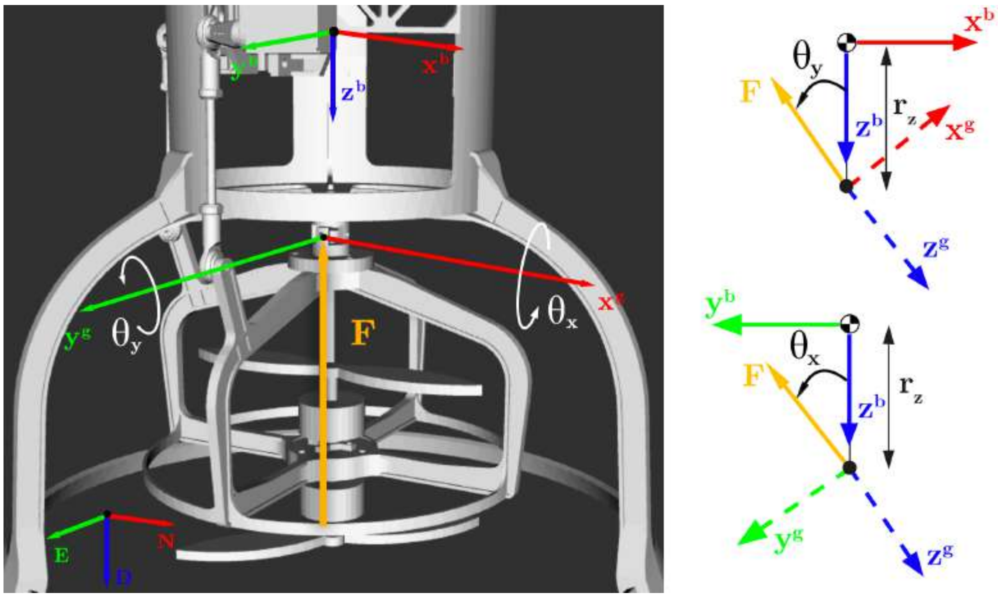


Figure 2: Gimbal thrust vectoring mechanism

2.1.4 Dynamics

In order to develop a controller that can predict the model's responses to different inputs, and thus perform complex maneuvers, in addition to perform computer simulations, the drone's dynamics needs to be modeled as accurately as possible. The following equations will govern the hopper dynamics during this project and are implemented in the `Rocket.m` and, `RocketDynamicsCasadi.hpp` or `Eigen.hpp` files, in MATLAB and C++ respectively.

$$\dot{\mathbf{x}} = f(\boldsymbol{x}, \boldsymbol{u}) = \begin{bmatrix} \dot{\mathbf{v}}^w \\ \dot{\boldsymbol{\omega}}^b \\ \dot{\mathbf{x}}^w \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \frac{1}{m}\left[\mathbf{F}^w_{\text{aero}} + \mathbf{F}^w_{\text{thrust}}\right] + \mathbf{g}^w \\ J^{-1}\left[M^b_{\text{aero}} + M^b_{\text{thrust}} - \boldsymbol{\omega}^b \times \left(J\boldsymbol{\omega}^b\right)\right] \\ \vec{v}^w \\ \frac{1}{2} q \circ \left(0, \boldsymbol{\omega}^b\right) \end{bmatrix} \tag{6}$$

Note that $\mathbf{F}^w_{\text{aero}}$ and $\mathbf{F}^w_{\text{thrust}}$ need to be transformed from the body frame to the world frame using the attitude of the rocket (q).

2.2 Identification experiments

The first stage of the project is to identify the drone's physical model, in order to be able to correlate the various inputs to a dynamic reaction. This will then enable realistic simulations to be carried out, but above all will provide the future NMPC with a model enabling behavior to be predicted over a time horizon. The commands given to the 2 motors running the propellers and to the 2 servo-motors controlling the gimbal are centralized in the Pixhawk 4 [1] autopilot system. The latter acts as the brain of the drone and then collects data from various sensors, including accelerometers, gyroscopes, magnetometers, barometers, and GPS receivers, to determine the vehicle's states and can execute commands to control the hopper's actuators.

The first step before being able to carry out the different model identification experiments is to create a simple ROS node that can publish commands to the actuators by reading the required inputs into an entry file and record the data published by the Bota Systems force/torque sensor [2] placed on the drone's fairing. The experiment setup can be observed on Figure 3, The drone is securely fixed from above to its test bench, on which the sensor is mounted. The given commands as well as the force vector ($\mathbf{F} = \begin{bmatrix} F_\text{N} & F_\text{E} & F_\text{D} \end{bmatrix}^T$) and torques ($\mathbf{T} = \begin{bmatrix} T_\text{x} & T_\text{y} & T_\text{z} \end{bmatrix}^T$) recorded are finally stored in an output `.bag` timetable. The data are then extracted into `.csv` files using the `rosbag_parser.py` script and analyzed with MATLAB. The various physical parameters identified are the following: the drone's thrust (2.2.2), the differential thrust (2.2.3), the gimbal orientation (2.2.5), the inertia (2.2.6) and the command's time delay (2.2.7).

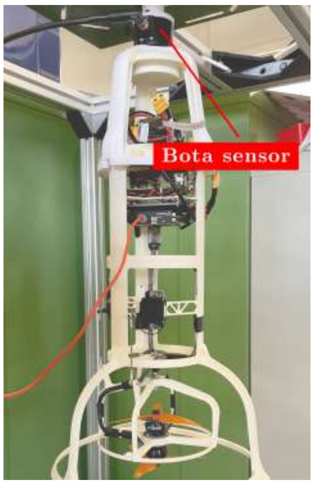


Figure 3: Identification experiments setup

The purpose of this identification is the creation of the `tvc2.yaml` file containing the coefficients needed to associate the given inputs with the actual system responses, which can then be read by the class modeling the drone's dynamics.

2.2.1 Data preprocessing

Before being able to accurately analyze the results obtained during this identification phase, it is important to identify whether any biases are present in the force/torque sensor used. To do this, data without the drone being placed on the test bench were collected and the average bias for each measured quantity was calculated. Table 1 lists the different biases measured for the forces and torques. The results of the identification experiments were then adjusted accordingly.

	F_N (N)	F_E (N)	F_D (N)	T_x (Nm)	T_y (Nm)	T_z (Nm)
bias	0.5742	-3.1334	-3.3331	-0.0123	-0.0570	0.1219

Table 1: Measured sensor biases

In addition to correcting the bias, it was necessary to synchronize the time stamp of the inputs with the time stamp of the data gathered by the sensor, in order to have common time steps at a frequency of 200Hz. To do so, the MATLAB function `retime()` with the `'pchip'` parameter was used.

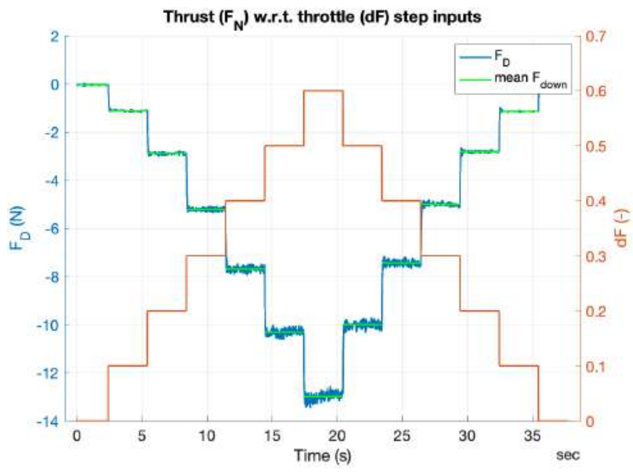
Given that the drone has a mass of $m =$**1.04** kg and is hanging on the sensor, the weight of the drone has also been corrected in the F_D measurement, so that only the vertical thrust generated by the propellers can be measured.

Finally, the last remaining minor bias still observed in the data before each experiment began, was also corrected before starting the analysis, so that only the quantities caused by the actuators were taken into account.

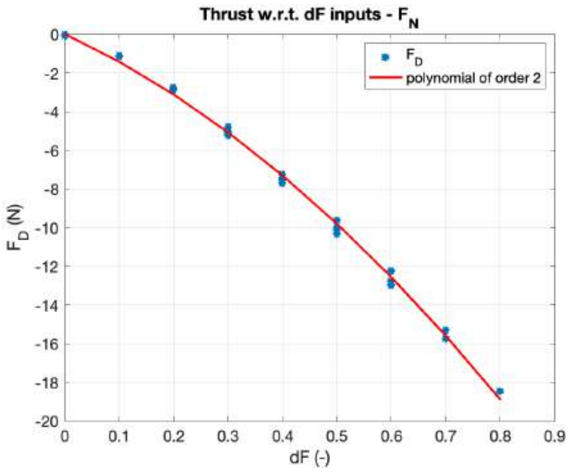
2.2.2 Thrust identification

In order to identify the thrust generated by the drone's 2 counter-rotating propellers when a certain input dF is given, the experiment carried out is as follows: gradual stepwise increase in throttle of $dF = 0.1$ every 3 seconds from $dF = 0$ until $dF = 0.6$ is reached, then gradual decrease until dF returns to 0. The resulting force down (F_D) is recorded and plotted in Figure 4a with the mean force value in each step printed in green. This mean force value with respect to each step input is then plotted in Figure 4b and finally an interpolation of order 2 is performed to obtain an equation relating throttle (dF) and generated thrust (F_{thrust}).

$$F_{\text{thrust}} = c_2 \cdot dF^2 + c_1 \cdot dF + c_0 = -13.3795 \cdot dF^2 - 12.9215 \cdot dF + 0.0013 \quad \text{(N)} \tag{7}$$



a) Throttle step inputs experiment



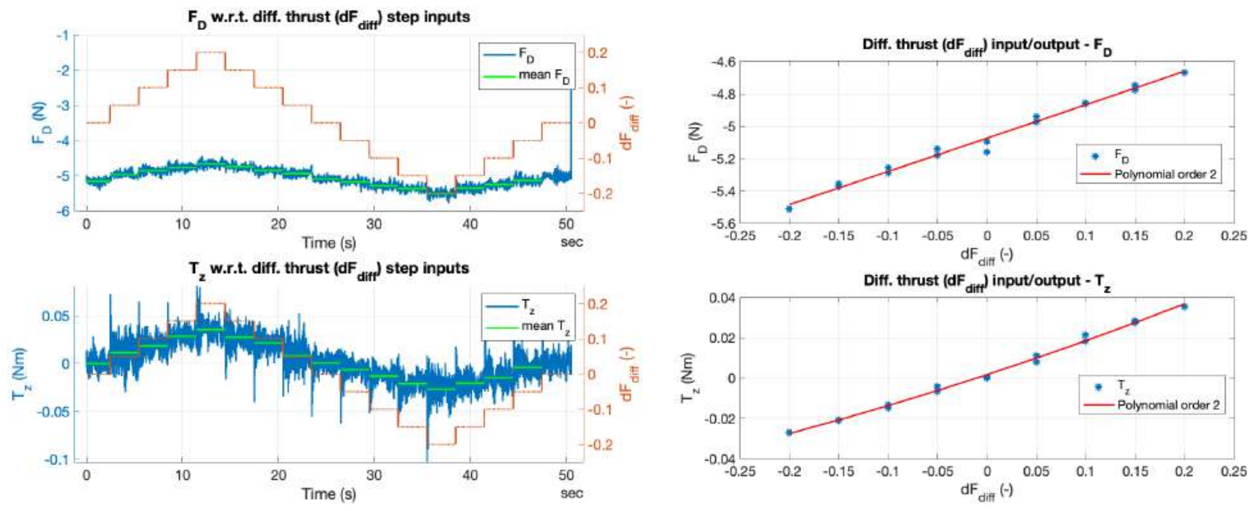
b) Order 2 polynomial interpolation

Figure 4: Thrust identification experiment - Generated F_D with respect to throttle input command (dF)

2.2.3 Differential thrust identification

In a very similar way to what was done to identify the drone's thrust, the experiment to identify the torque T_z generated by the dF_{diff} command is as follows: stepwise increment of $dF_{diff} = 0.05$ every 3 seconds from no input until $dF_{diff} = 0.2$ then gradually until $dF_{diff} = -0.2$ and finally returns to 0. The recorded changes in F_D and T_z are plotted in Figure 5a and the mean value for each step is highlighted in green. It can be seen that giving a differential thrust command in addition to generating a rotational moment also produces a small change in the measured F_D meaning a change in the thrust developed. However, this shift seems to be minor.
Similar to what was done for thrust identification, the average F_D and T_z for each dF_{diff} steps are shown in Figure 5b and an interpolation to order 2 is performed. Given that only the one concerning the torque is truly relevant, the latter is the sole one presented here.

$$T_z = c_2 \cdot dF_{diff}^2 + c_1 \cdot dF_{diff} + c_0 = 0.0753 \cdot dF_{diff}^2 + 0.1608 \cdot dF_{diff} + 0.0017 \quad \text{(Nm)} \tag{8}$$



a) Differential thrust step inputs b) Order 2 polynomial interpolation

Figure 5: Differential thrust identification experiment - Resulting torque (T_z) and force (F_D) with respect to the differential throttle input command (dF_{diff})

2.2.4 Thrust vectoring experiment

It is now interesting to look at the modification of the force vector and the torques generated when modifiying the gimbal orientation by giving successively dR and dP stepwise increasing/decreasing inputs of magnitude 0.1 every 3 seconds while maintaining a constant thrust ($dF = 0.5$). Figure 6 illustrates the result of these changes in gimbal orientation in F_N, F_E, T_x and T_y magnitude.
As expected, and with reference to Figure 2, increasing dP leads to a decrease of θ_y (see section 2.2.5), an increase of F_N magnitude as well as T_y. On the contrary, increasing dR decreases θ_x and therefore leads to a decrease of F_E magnitude as well as an increase of T_x.

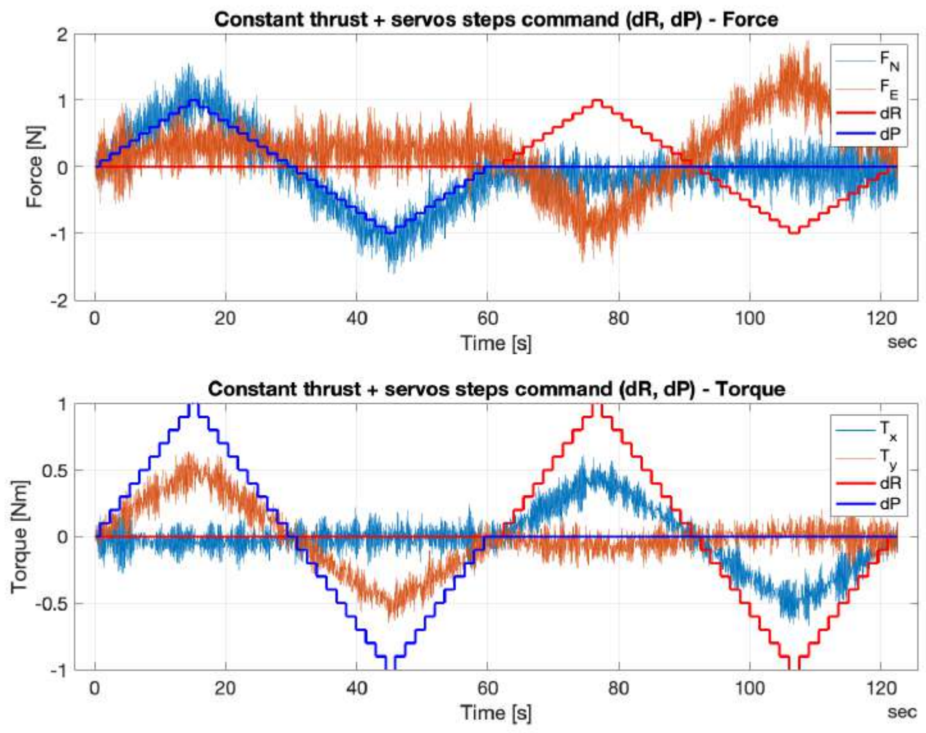


Figure 6: Resulting forces (F_N, F_E) and torques (T_x, T_y) with restpect to constant throttle ($dF = 0.5$) + *Roll* (dR) and *Pitch* (dP) steps command experiment

Finally, as in the previous sections, an interpolation is performed, plotted in Figure 7, in order to relate inputs and system responses. The following results are obtained:

$$F_N = c_1 \cdot dP + c_0 = 1.1613 \cdot dP - 0.0734 \quad \text{(N)} \ ; \quad F_E = c_1 \cdot dR + c_0 = -1.1348 \cdot dR + 0.2347 \quad \text{(Nm)} \quad (9)$$

$$T_y = c_1 \cdot dP + c_0 = 0.5238 \cdot dP - 0.0135 \quad \text{(N)} \ ; \quad T_x = c_1 \cdot dR + c_0 = 0.4940 \cdot dR - 0.0235 \quad \text{(Nm)} \quad (10)$$

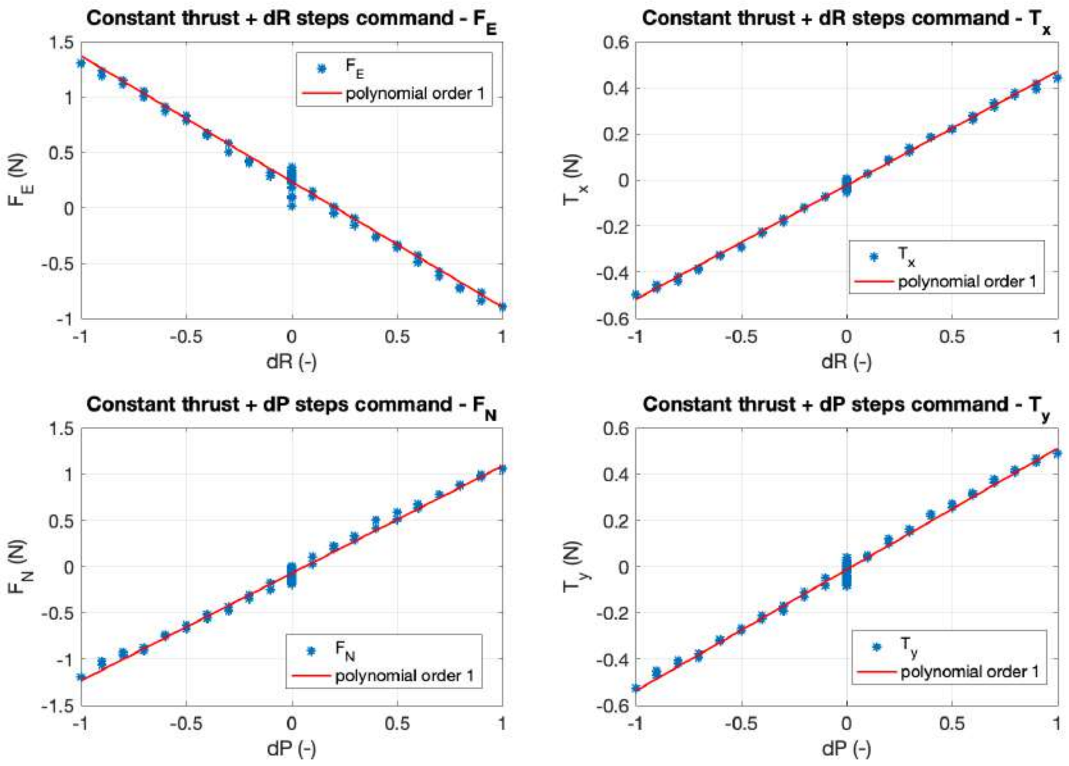


Figure 7: Polynomial interpolation of the resulting forces and torques generated by *Roll* (dR) and *Pitch* (dP) inputs

2.2.5 Gimbal orientation identification

The method chosen to identify the relationship between *Roll* and *Pitch* inputs (dR and dP) and the actual inclination of the gimbal, respectively θ_x and θ_y (in degrees), is to gradually decrease $dR \in [0; -1]$ by steps of -0.1 and take a picture of the resulting angle (θ_x in this case) of the gimbal where a rod has been attached in order to increase post-processing measurement accuracy. The images are then straightened in Adobe Photoshop, and the angle is precisely measured. Figure 8 showcases an example of 4 images (for $dR \in \{0, -0.3, -0.6, -0.9\}$) in which the measurement was carried out.

The second method, which was used to check the consistency of the visual observations, is to trigonometrically back-calculate the gimbal angle using the variations in F_N, F_E and F_D measured when moving the gimbal with a constant thrust during the experiment discussed in the previous section 2.2.4. The trigonometric relations used are the following:

$$\theta_x = \arctan\left(\frac{F_E}{-F_D}\right) \quad \text{and} \quad \theta_y = \arctan\left(\frac{F_N}{F_D}\right) \tag{11}$$

The results obtained by both methods are presented in Figure 9 for comparison. It can be seen that the results obtained are fairly similar, with slight differences due to the inaccuracies specific to each method. The general trend appears to be very consistent. The first-order interpolation was performed on the data collected from the images, which are probably less noisy. The equation relating dR and dP to θ_x and θ_y respectively is as follows:

$$\theta_x = c_1 \cdot dR + c_0 = -7.2364 \cdot dR + 0.5727 \quad (^\circ) \tag{12}$$

Note that due to the symmetry, the experiment was carried out on one axis only.

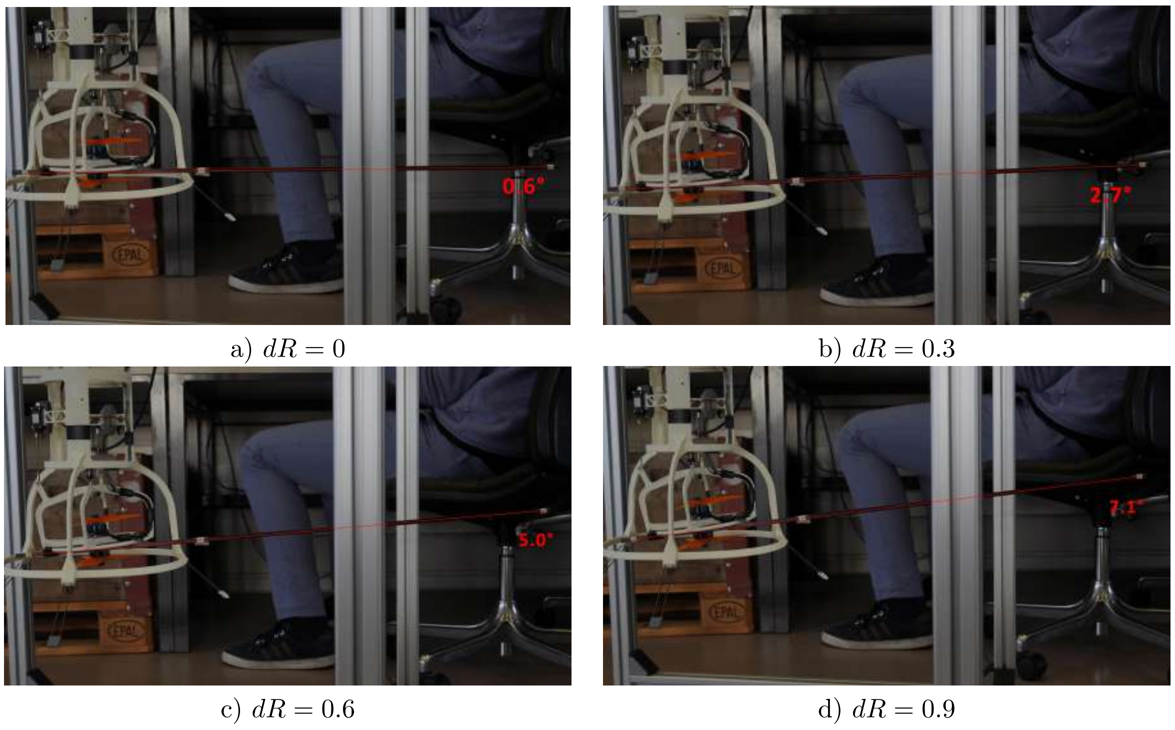


Figure 8: Gimbal orientation identification experiment - Pictures of the resulting gimbal inclination (θ_x) with respect to $dR \in \{0, -0.3, -0.6, -0.9\}$

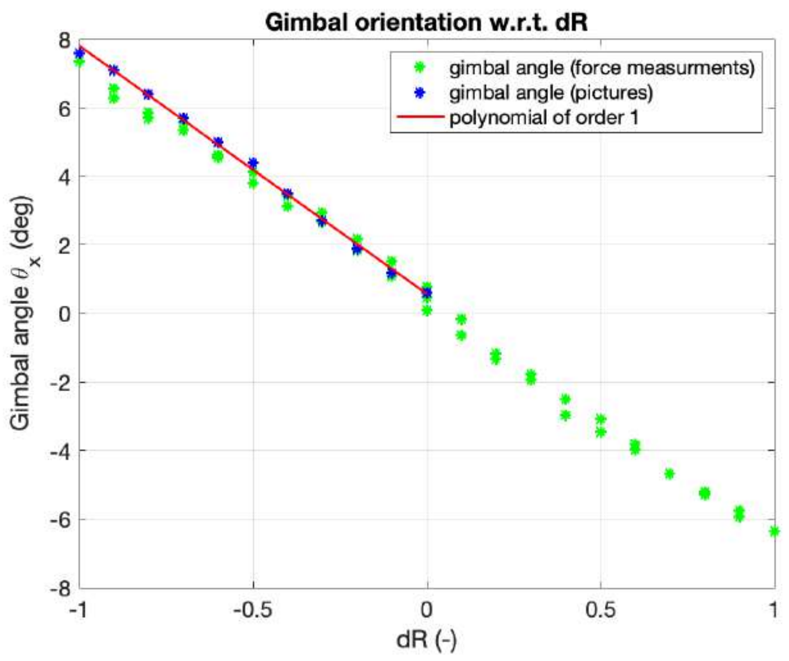


Figure 9: Resulting gimbal angle with respect to dR input commands - comparison between pictures identification (blue) and changes in the measured force vector (green) with the 1st order interpolation

2.2.6 Inertia identification

Identification of the drone's inertia is performed using the pendulum method [3]. The idea is to hang the drone by two strings symmetrically placed to the left and right of the center of mass (CoM) and attached to the ceiling (see Figure 10) in order to make the drone oscillate around its CoM for each axis and measure the mean period (T). Knowing the mass of the drone ($m = 1.04$ kg), the length of the ropes (h) and the distance between the two attachment points (D), the inertia (I) can be derived as follows:

$$I = \frac{mgD^2T^2}{16\pi^2 h} \tag{13}$$

The results were obtained by averaging the mean period over 5 experiments of 10 and 20 oscillations for $I_{xx} = I_{yy}$ and I_{zz} respectively. The following results are obtained.

$$I_{xx} = I_{yy} = 0,0265 \text{ kg m}^2 \ ; \quad I_{zz} = 0,0037 \text{ kg m}^2$$

It can be noted that the value obtained for I_{zz} seems quite small - it might be worth running the experiment again to double-check the results.

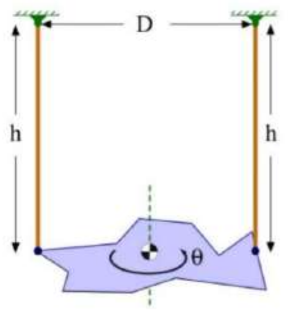


Figure 10: Inertia identification setup [3]

2.2.7 Time delay identification

Finally, the last parameter identified during this project is the time delay between the moment an input is given and the first observable reaction of the system. This could be useful to get an idea of how fast the system is reacting and to be able to anticipate the time required to adjust the trajectory. This identification is based on data collected during the thrust identification (section 2.2.2) but here the time delay between a decrease in throttle input (dF) and a drop in thrust (F_{up} for convenience and only in this case) is measured. This leads to a resulting time delay around **28 ms** as shown in Figure 11. However, this delay seems rather high, which could be explained by the inertia of the 2 propellers, which requires a certain amount of time before being able to effectively adjust speed and therefore reduce thrust. This behavior is certainly something to keep in mind. This time delay has been subtracted from the results in order to carry out the analysis of the identification results.

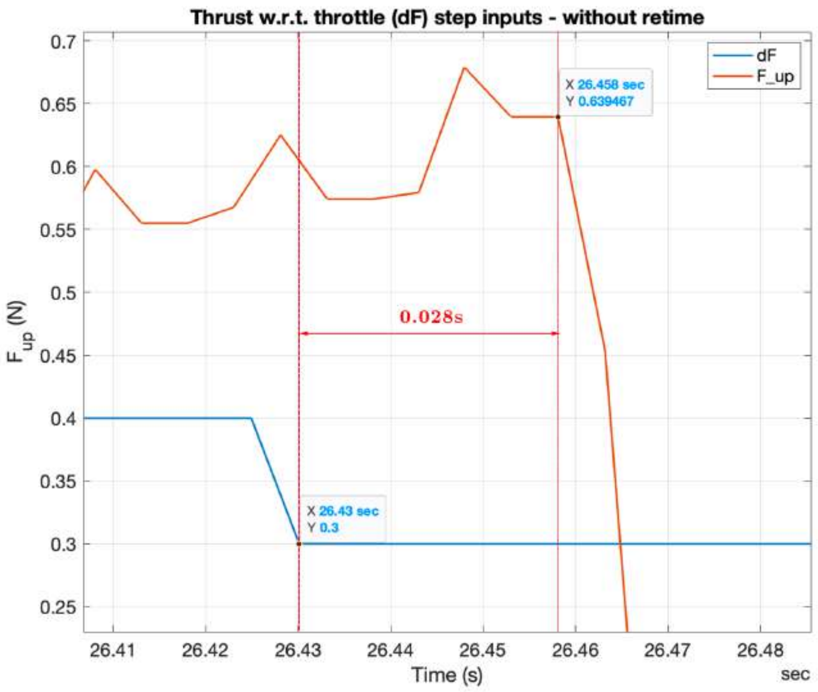


Figure 11: Time delay identification between a decrease of -0.1 in throttle input (dF) and the thrust (F_{up}) response

3 NMPC design

Now that the model has been identified and the drone's dynamics are known, it's possible to develop a NMPC that will make use of this model to predict its future state and optimize its trajectory over a defined horizon (N). The goal of the controller is to track a reference state ($\mathbf{x}_{ref} = \begin{bmatrix} \mathbf{v}^w_{ref} & \boldsymbol{\omega}^b_{ref} & \mathbf{x}^w_{ref} & q_{ref} \end{bmatrix}^T$)

knowing its current state ($\mathbf{x} = \begin{bmatrix} \mathbf{v}^w & \boldsymbol{\omega}^b & \mathbf{x}^w & q \end{bmatrix}^T$) and the available inputs ($\mathbf{u} = \begin{bmatrix} dR & dP & dF & dF_{diff} \end{bmatrix}^T$). To achieve this, the cost function (eq. 14) will be solved at each time step in order to optimize the control inputs over the finite prediction horizon (N) while considering system dynamics and constraints (eq. 15). The role of the Lagrange cost is to track the residuals and penalize the error between the current state $\mathbf{x}$ and the reference state $\mathbf{x}_{ref}$ at time t. The Mayer cost in eq. 14 represents the cost associated with the final state of the system at the end of the prediction horizon (t_f) and helps achieving the desired control objectives while ensuring stability and constraint satisfaction. The constraints (eq. 15) are mainly those associated with the inputs and explained in section 2.1.2, as well as the one constraining the drone not to go below the ground. The gain matrices Q, R and Q_f weight the contributions of the state and input variables in the prediction horizon and allow a trade-off between control performance and stability. The Optimal Control Problem (OCP) is the following:

$$\min_{\boldsymbol{u}(t)} \int_{t_0}^{t_f} \underbrace{(\underbrace{(\boldsymbol{x}-\boldsymbol{x}_{ref}(t))^T\,\mathbf{Q}\,(\boldsymbol{x}-\boldsymbol{x}_{ref}(t))}_{\text{non-control cost}} + \underbrace{\boldsymbol{u}^T\mathbf{R}\boldsymbol{u}}_{\text{control cost}}\ )}_{\text{Lagrange cost}}\,dt + \underbrace{(\boldsymbol{x}-\boldsymbol{x}_f)^T\,\mathbf{Q}_f\,(\boldsymbol{x}-\boldsymbol{x}_f)}_{\text{Mayer cost}} \tag{14}$$

$$\begin{aligned} \text{s.t.} \quad \dot{\boldsymbol{x}} &= f(\boldsymbol{x},\boldsymbol{u}) \\ -1 &\leq dR \leq 1 \\ -1 &\leq dP \leq 1 \\ 0 &\leq dF \leq 1 \\ -1 &\leq dF_{\text{diff}} \leq 1 \\ z &\leq 0 \end{aligned} \tag{15}$$

In view of the non-linear and time-variant nature of the system dynamics (eq. 6), the OCP has to be transformed into a Nonlinear Program (NLP) using discretization methods such as multiple shooting [4] or collocation [5] [6] which can then be solved using optimization solvers. During this project, IPOPT [7] which implements an interior point line search filter method and PIQP (Proximal interior-point quadratic programming) [8] will mainly be used. The goal is to compare the results obtained by these different methods and solvers in order to find the one best suited to the project's needs.

3.1 MATLAB implementation

The first step was to carry out an initial implementation of the OCP and rocket parameters identified in the previous section 2.2 on MATLAB. In this MATLAB framework, the transcription classes (`LaMultipleShootingUSX.m`, `RadauCollocationUSX.m`), the solving part, the rocket dynamics class (`Rocket.m`, `RocketBase.m`), the simulation class (`Simulator.m`) and visualization class (`RocketVis.m`) of results were already pre-implemented.

The OCP is constructed in the `RocketUOcp.m` class, which contains all the elements shown in equation 14. The first thing is to implement the Lagrange cost defined by `get_non_control_cost()` and `get_control_cost()` functions that computes the elements presented in eq 16.

$$l(\boldsymbol{x},\boldsymbol{u},t) = (\boldsymbol{x}-\boldsymbol{x}_{ref}(t))^T\,\mathbf{Q}\,(\boldsymbol{x}-\boldsymbol{x}_{ref}(t)) + \boldsymbol{u}^T\mathbf{R}\boldsymbol{u} \tag{16}$$

The non control cost fucntion compute and penalize the state errors by subtracting the reference to the current state. The penalization in term of attitude in quaternions is computed as follows :

$$q_w^{err} = q_w \cdot q_w^{ref} - \begin{bmatrix} q_x^{ref} & q_y^{ref} & q_z^{ref} \end{bmatrix} \cdot \begin{bmatrix} -q_x \\ -q_y \\ -q_z \end{bmatrix} \tag{17}$$

$$q_{x,y,z}^{err} = \begin{bmatrix} q_x^{ref} \\ q_y^{ref} \\ q_z^{ref} \end{bmatrix} \times \begin{bmatrix} -q_x \\ -q_y \\ -q_z \end{bmatrix} + q_w^{ref} \begin{bmatrix} -q_x \\ -q_y \\ -q_z \end{bmatrix} + q_w \begin{bmatrix} q_x^{ref} \\ q_y^{ref} \\ q_z^{ref} \end{bmatrix} \tag{18}$$

$$q^{err} = \left(q_w^{err}, q_{x,y,z}^{err}\right) - \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T \tag{19}$$

```
function non_control_cost = get_non_control_cost(obj, x, t)
    non_control_cost = 0;
    v_err = obj.sym.params.x_ref(1:3) - x(1:3);
    w_err = obj.sym.params.x_ref(4:6) - x(4:6);
    p_err = obj.sym.params.x_ref(7:9) - x(7:9);
    q_err = (quat_multiply(obj.sym.params.x_ref(10:13)', [x(10);-x(11:13)]') - [1 0 0 0])';

    non_control_cost = non_control_cost ...
        +  v_err' * diag(obj.sym.params.W_err_diag(1:3)) * v_err ...
        +  w_err' * diag(obj.sym.params.W_err_diag(4:6)) * w_err ...
        +  p_err' * diag(obj.sym.params.W_err_diag(7:9)) * p_err ...
        +  q_err' * diag(obj.sym.params.W_err_diag(10:13)) * q_err ;
end
```

The `get_control_cost()` function computes the control cost which assigns a weight to each given inputs through the diagonal coefficients of the R matrix as follows:

```
function control_cost = get_control_cost(obj, u, t)
    control_cost = 0;
    us = [0; 0; 0.5; 0];
    du = u - us;
    control_cost = control_cost + du' * diag(obj.sym.params.R_diag) * du;
end
```

Finally, the Lagrange cost sums the result of these two functions, while the Mayer cost (see eq. 14) multiplies the non-control cost at final time t_f with the Mayer multiplier coefficient which defines the weight of $\mathbf{Q}_f$. Note also the substraction of the $\mathbf{u}_s = \begin{bmatrix} 0 & 0 & 0.5 & 0 \end{bmatrix}$ vector to correct the fact that the minimum throttle input cost is defined by making the drone hover, i.e. not going up or down.

A general overview of the MATLAB framework is shown in Figure 12. The implementation proceeds as follows: First, the creation of the `optimization_problem` object including all the elements defining the problem (gain matrices ($\mathbf{Q}$, $\mathbf{R}$, $\mathbf{Q}_f$), constraints (`lbu`, `ubu`), horizon (H), actual state and reference ($\mathbf{x}$, $\mathbf{x}_{ref}$), OCP (`RocketUOcp.m`)). This object is then transcribed using either multiple shooting or Radau collocation. Finally, the problem is solved using the CasADI [9] framework and IPOPT at each time step (dt) up to the final time (t_f) in order to find the optimal input sequence over the horizon. The first optimal input is then at each time step applied to the system using the `simulate_step()` function related to the dynamics. The resulting new state becomes the current state ($\mathbf{x}$) and serves as the starting point for the next solving.

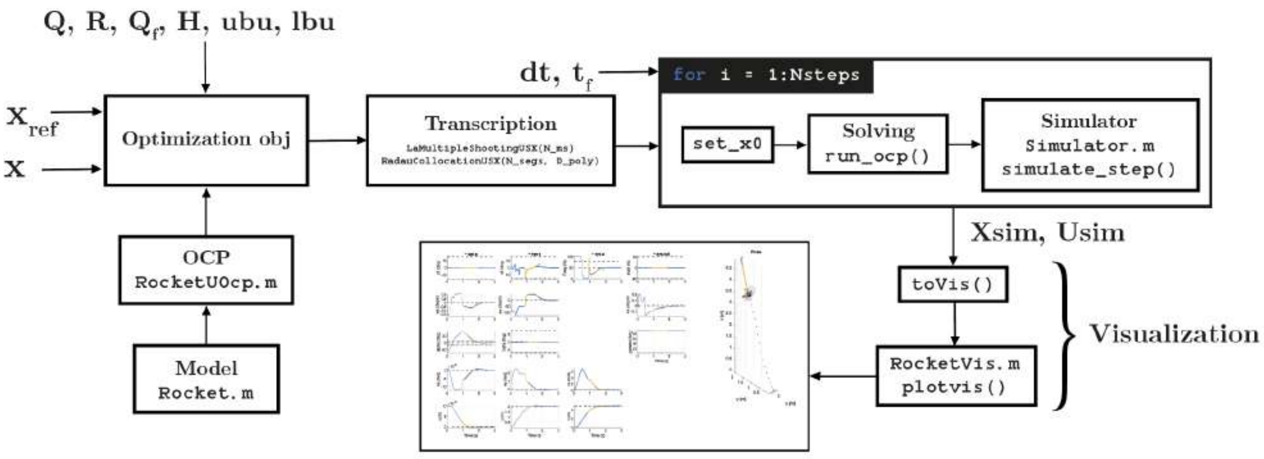


Figure 12: MATLAB implementation scheme

Figures 13 and 14 present the results obtained for simulations using multiple shooting (N_{segs} = 25 segments) and Radau collocation (N_{segs} = 6 and D_{poly} = 4) respectively with H = 2s, t_f = 3s, dt = 0.05s recovering from an initial Euler angle of $-30°$ in y (converted to quaternion) and tracking the state $\mathbf{x}_{ref}$.

$$\mathbf{x}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9659 & 0 & -0.2588 & 0 \end{bmatrix}^T$$

$$\mathbf{x}_{ref} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 3 & 3 & -5 & 1 & 0 & 0 & 0 \end{bmatrix}^T$$

$$\mathbf{Q} = \text{diag}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 5 & 5 & 5 & 10 & 10 & 10 & 10 \end{bmatrix}^T; \quad \mathbf{Q}_f = 10 \cdot \mathbf{Q}$$

$$\mathbf{R} = \text{diag}\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T$$

The choice of the $\mathbf{Q}$ coefficients is done by assigning more weights to the states we care the most about. In the specific case of this project, errors in drone position and attitude are more critical to vehicle control than errors in speed or angular rate. For this reason, a higher wheigt is applied to increase the cost related to these errors, and better track these states. In the two Figures (13 and 14), the yellow curve on each graph represents the open-loop trajectory for the given state (i.e. the optimal trajectory predicted at time $t = 0.814$s), which can be compared with the resulting final trajectory (blue). Indeed, only the first optimal input calculated by the NMPC is applied to the system at each time increment, which explains why these curves are very similar, but differ slightly after all. The small slider below the rocket pose graph allows the user to go back in time at the end of the simulation and analyze the trajectory predicted by the MPC at a given time step. This tool was implemented during the project to verify whether the trajectory predicted by the NMPC throughout the simulation was consistent or not.
Finally, we can see that the results obtained using both multiple shooting and Radau collocation are consistent and rather similar, although the second one seems a little less " smooth ".

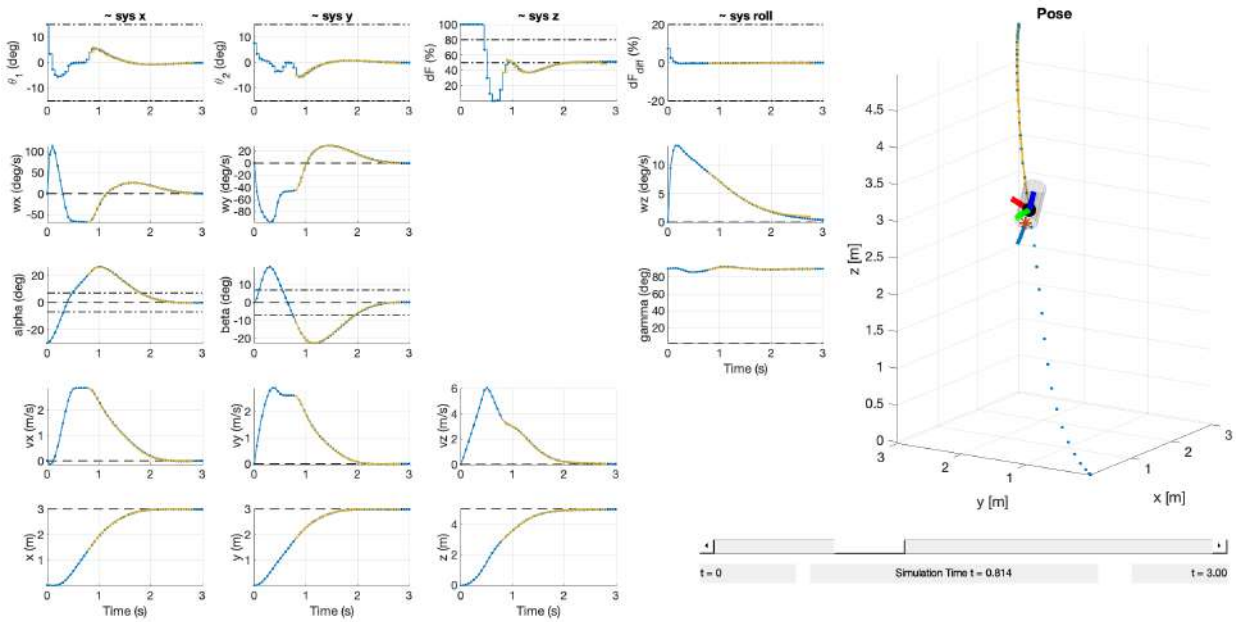


Figure 13: Simulation using multiple shooting ($N_{segs} = 25$)

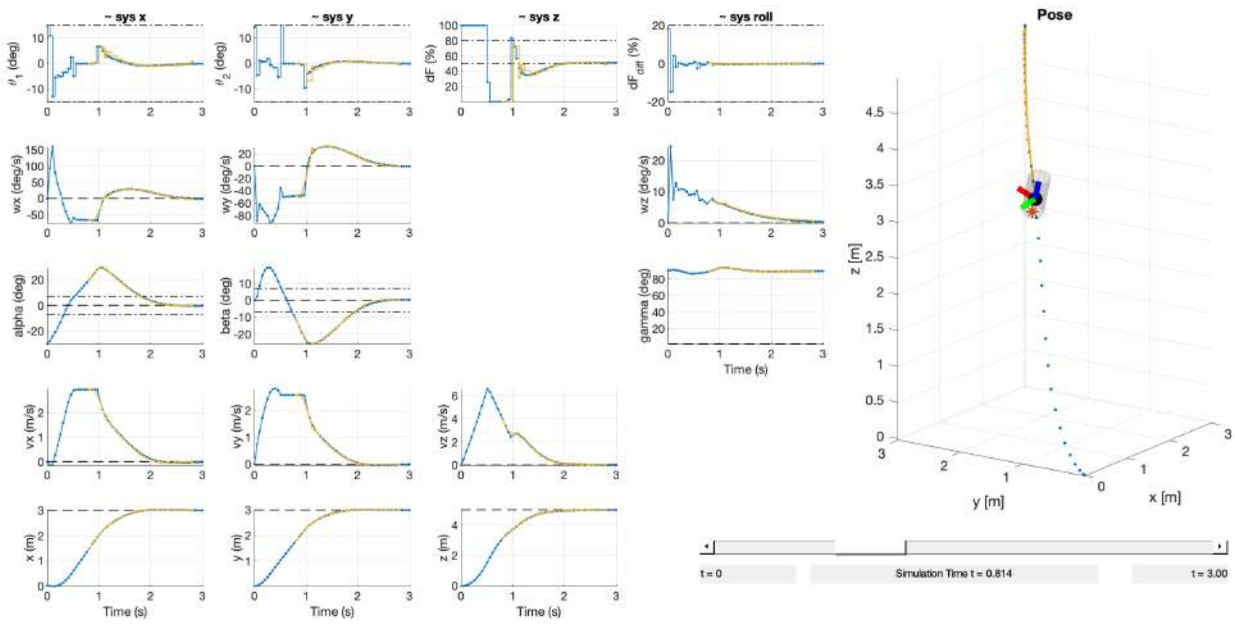


Figure 14: Simulation using Radau collocation ($N_{segs} = 6$ and $D_{poly} = 4$)

3.2 C++/ROS implementation

Now that the NMPC implementation and the identified dynamics have been tested on MATLAB, it's time to implement quite the same setup on the C++/ROS framework, which already contains the simulation and visualization nodes and the optimization routine. This architecture will enable to perform more realistic simulation tests by for example, adding disturbances or moving the tracked reference in real time, just as it would be possible in the reality. Furthermore, if the results obtained are convincing, quite the same implementation can later be embedded in the drone's flight computer.
To achieve this, and to be able to compare the results obtained (in particular the solving time), two different implementations are carried out in parallel: I implement the NMPC in the CasADI framework with multiple shooting as transcription and IPOPT as solver (MOMI), while Pietro implements it in the Eigen3 [10] framework using LaOPT with multiple shooting as transcription and IPOPT or PIQP as solver (LAMI and LAMP respectively). Only the implementation using CasADI will be discussed in this report.

The three main elements of the NMPC that had to be implemented are: the `FlightOCP` class defining the optimal control problem to be solved, the `PositionTracking_MOMI_Optimizer` class bringing together all the elements used to solve the OCP, and the `FlightOcpParams` structure used to define the parameters and constraints which are read from an external file (`constraints.yaml`) that can be modified to refine or modify the problem.
With regard to the OCP, the implementation is very similar to the one carried out on MATLAB, including the same functions for calculating non-control, control, Mayer and Lagrange costs (as defined by eq. 14), the difference being perhaps that all the variables are symbolic as required by CasADI, making the calculations more heavy. The `update_config_params()` and `update_tuning_params()` functions are used to update the parameters needed by the OCP. The difference is that the first function (`update_config_params()`) handles updating the parameters that change at each time step, i.e. mainly $\mathbf{x}_{ref}$ while the config parameters are the gain matrices ($\mathbf{Q}$, $\mathbf{R}$ and Mayer factor) and the constraints on the inputs (lbu, ubu) which are not meant to change very often. Finally, the `setup_discretized_trajectory_constraints()` function, in addition to setting the first state to be the starting point, sets the states and controls constraints defined in eq. 15 as follows:

```cpp
for (int i = 0; i < U_sym.size2(); ++i)
{
    opti->subject_to(u_lb_sym(0) <= U_sym(0, i) <= u_ub_sym(0));
    opti->subject_to(u_lb_sym(1) <= U_sym(1, i) <= u_ub_sym(1));
    opti->subject_to(u_lb_sym(2) <= U_sym(2, i) <= u_ub_sym(2));
    opti->subject_to(u_lb_sym(3) <= U_sym(3, i) <= u_ub_sym(3));
}

// Constraints on states
casadi::Slice all;
opti->subject_to(X_sym(8, all) <= 0); // z value only negative (NED)

// Initial state
opti->subject_to(X_sym(all, 0) == x0_sym);
```

With regard to the optimizer, the latter is mainly dedicated to construct functions that call the elements required (pre-defined in the `OptiOcp` class) before and after solving the OCP, such as the transcription method, compute the optimal controls, extract the optimal states, etc.

3.3 Implementation comparison

Now it's time to check the consistency of this implementation against the MATLAB framework, which acts as a sanity check. The case study carried out in the section 3.1 (Figures 13 and 14) is also performed on the C++ framework, and the optimal controls and states obtained by optimization on a single iteration with an horizon $H = 2$ are superimposed on Figures 15 and 16 respectively. The $\mathbf{x}_0$, $\mathbf{x}_{ref}$, $\mathbf{Q}$, $\mathbf{Q}_f$, $\mathbf{R}$ and multiple shooting transcription settings are exactly the same as those presented in the previous section.

It can be observed that the optimal controls and states obtained using multiple shooting as transcription and IPOPT as solver (ms and MOMI), are perfectly superimposed, which implies that the solution is as expected similar. As for the solution using Radau collocation (MATLAB only), this differs a bit due to the differences in the algorithms involved, but still looks very similar to the solutions obtained using multiple shooting.

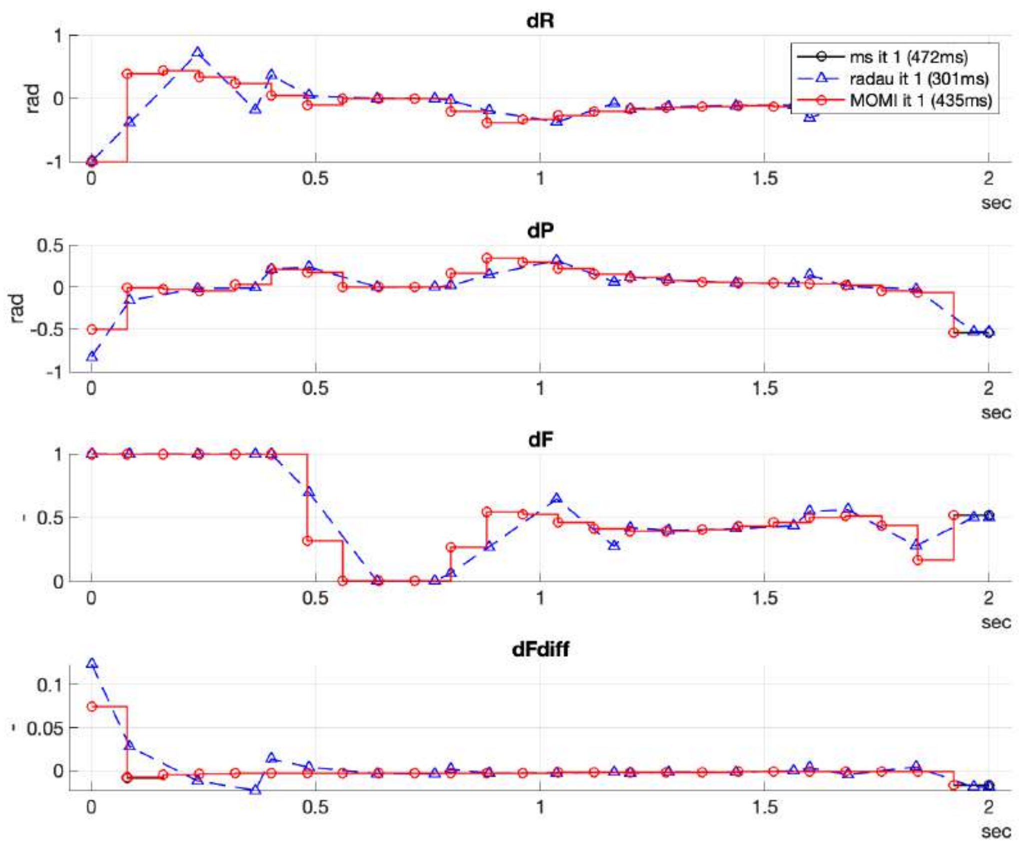


Figure 15: Optimal control trajectory comparison: MATLAB (ms(N_{segs} = 25), radau(N_{segs} = 6 and D_{poly} = 4)) and C++ MOMI(N_{segs} = 25) for a horizon of H = 2s

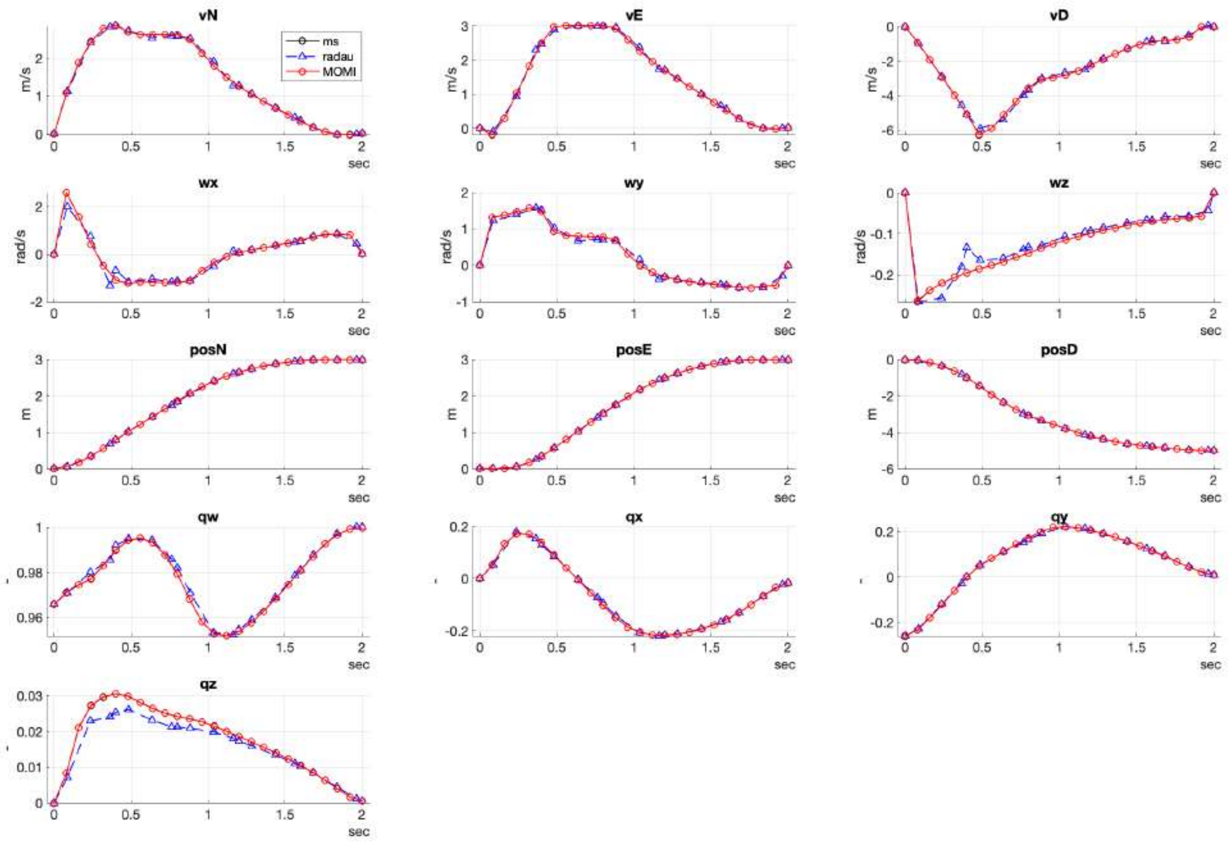


Figure 16: Optimal state trajectory comparison: MATLAB (ms(N_{segs} = 25), radau(N_{segs} = 6 and D_{poly} = 4)) and C++ MOMI(N_{segs} = 25) for a horizon of H = 2s

4 Simulations

Now that the MPC's implementation has been validated, it's time to test its behavior on simulations by integrating its operation into the complete ROS model. This will emulate the dynamics and the drone behavior in order to have more realistic simulations and to test the robustness against distrurbances. The Figure 17 illustrates the architecture of the different ROS nodes used to test the TVC drone through simulations. The `simulation_flight_ros` node returns the states of the object according to the controls given by the `main_control node`, which can either control the vehicle using its PID or, if the `_arm_mpc:=true` condition is met and the `position_tracking_mpc_MOMI_node` node (for example) is running, control the rocket using the NMPC running at 50Hz. Finally, by connecting a Playstation controller and launching the `joy_node`, it is possible to move the drone by changing the controller's reference states in real time. All this, is visualized through Rviz [11], which provides a graphical interface.

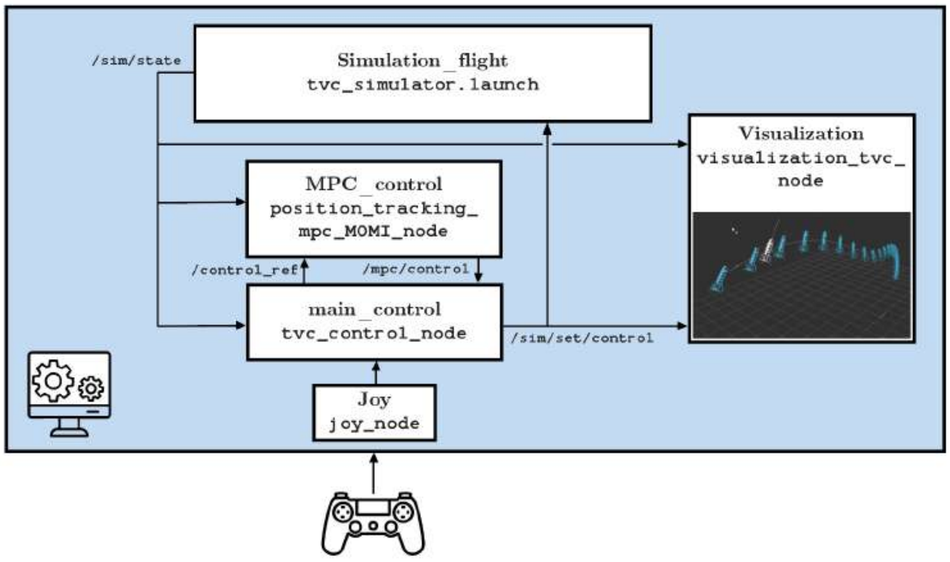


Figure 17: Scheme of the ROS simulations framework

The drone's behavior during the simulations is visualized as shown in Figure 18. Note in blue the MPC states prediction on the $H = 2$s horizon. On the left of the figure, the vehicle is successfully relocating, as the reference has been moved by the user. On the right, the drone is stabilized around the tracked target.

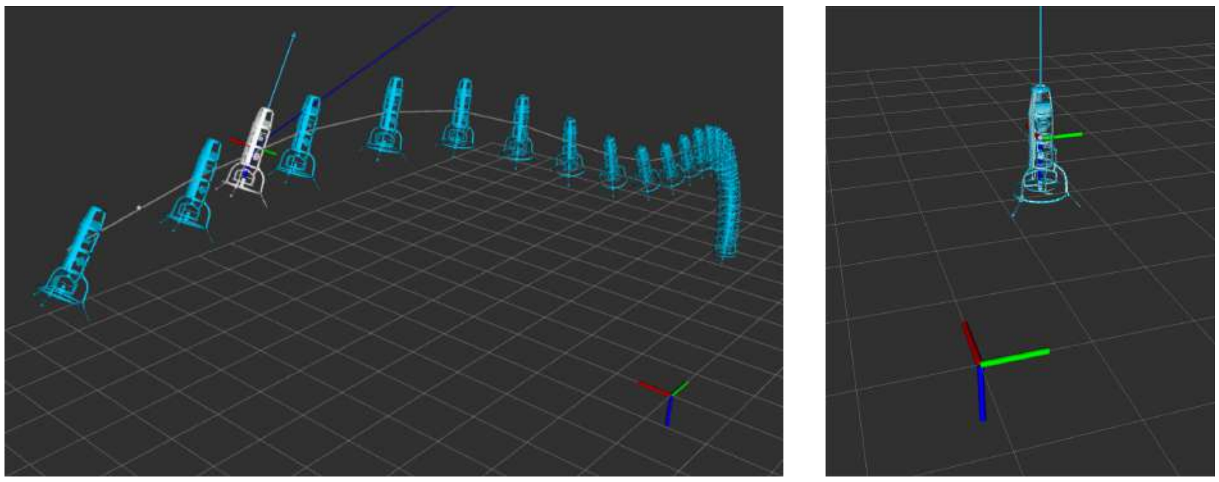


Figure 18: Visualization resulting from simulations of the drone controlled by the NMPC when the reference is relocated (left) and stabilized around its reference (right) without disturbances

One of the first observation made during the simulations, was the significant difference in solving time between the different OCP formulations when solving the OCP in real time within the complete ROS simulation architecture. The MOMI (with $N_{segs} = 25$) formulation presented in previous 3.2, is particularly slow, with a solving time varying between **80**ms and **200**ms depending on computer performances. This

delay is particularly high given that simulations run at a frequency of 200Hz and that the NMPC try to solve the OCP at 50Hz. This latency tends to completely destabilize the controller, which is unable to compute optimal controls in time, resulting in a drone crash and make testing almost impossible. A formulation using Radau collocation instead of multiple shooting (MORI with $N_{segs} = 5$ and $D_{poly} = 4$) has been tried out but abandoned due to the lack of significant improvements. In order to still evaluate the behavior of the drone controlled by the MOMI formulation of the OCP, simulations were carried out with the speed divided by a factor of two (`simSpeed:=0.5` as parameter of the simulation node). This led to the conclusion that this formulation was fully consistent with a drone that is stable and capable of relocating exactly as desired when commanded by the user. However, due to the excessively slow NMPC resolution time, there is no way this OCP can be embedded in the real TVC drone's onboard computer. As the initial aim of the project was to produce an NMPC capable of being integrated into the model rocket, the future developments have been focused on the other formulations which are showing much more promising results.
Indeed, for the LAMP formulation implemented by Pietro, the resolution time is much more reasonable, with results between **30**ms and **45**ms depending on the computer. This can be appreciated in simulations, where flying the drone by moving the reference provides much more stable and smooth behavior, enabling more complex maneuvers, and seems considerably more robust. It is this formulation that has stood out from the others in terms of performance, and has been chosen as the reference for the rest of the project, with the aim of being embedded in the prototype drone.

4.1 Robustness to disturbances

To further test the limits of the controller, it is important to observe its behavior when confronted to external disturbances. The aim of these tests is to analyze the NMPC's reactions to the various events it might encounter during a real test flight. The two major types of disturbance that can destabilize the drone are, firstly, unanticipated external forces such as a gust of wind or, secondly, an error during the identification phase, leading to a model mismatch and thus unpredictable drone reactions.

4.1.1 Wind disturbances

In order to handle the unexpected conditions of a real-life flight, with all the elements that can disrupt the vehicle's behavior, notably external aerodynamics forces, the NMPC must be sufficiently robust to absorb these disturbances, while still being able to track its references consistently. To simulate the effect of random gusts of wind on the drone, the disturbance parameters (`windFrom_deg` and `windSpeed`) can be modified when launching the simulation node. Figure 19 shows the rocket able to relocate to its new reference position (left) and stabilize around its fixed reference (right) with a 5 m/s wind oriented at 45°. More generally, the drone's behavior when flying is reliable, even with disturbances as significant as a 5 m/s wind, which the controller seems to be fully able to handle.

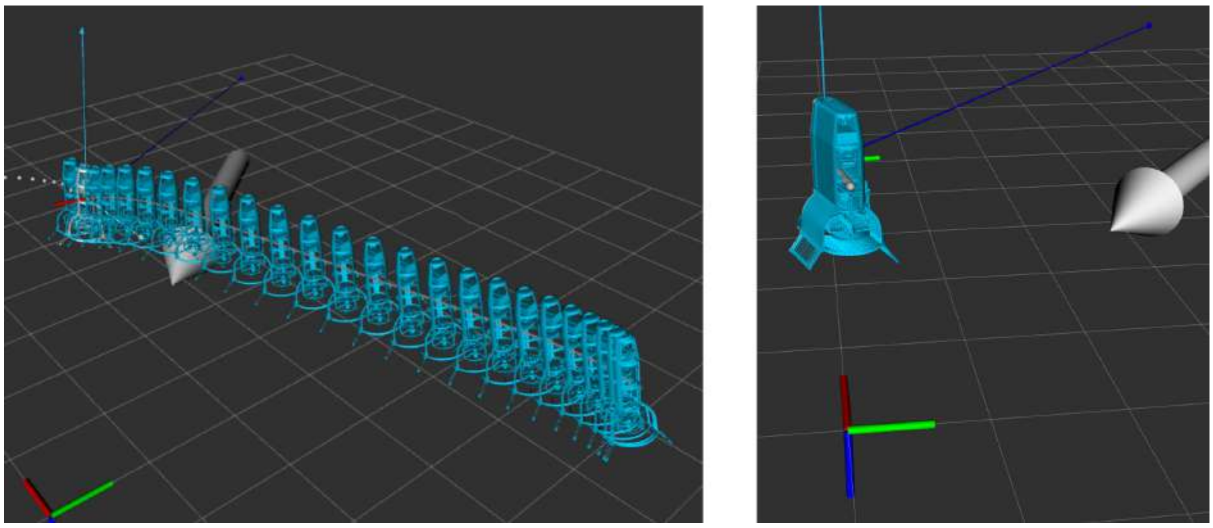


Figure 19: Visualization resulting from simulations of the drone controlled by the NMPC when the reference is relocated (left) and stabilized around its reference (right) with 5 m/s wind disturbances

4.1.2 Model mismatch

The other main source of disturbance that could be present and destabilize the drone are the problems related to model identification discussed in section 2.2. Indeed, if one of the identified components turns out to be wrong, this means that the drone dynamics on which the NMPC bases its prediction of the future state will be biased. This could result in an inappropriate choice of inputs by the controller and unpredicted reactions by the rocket model. The NMPC must therefore be able to correct problems arising from a difference in dynamics between the one used for prediction and the one that actually drives the drone in order to still be able to track a reference. To evaluate this ability, various simulations were carried out after deliberately modifying certain components of the file containing the identified parameters on which the dynamics is based (`tvc2.yaml`) and used by the controller for predictions. The following model mismatchs were investigated:

- Increase/decrease the drone's weight
- Increase/decrease the drone's inertia
- Randomly increase/decrease the c_2, c_1 and c_0 thrust coefficients (eq. 7)
- Randomly increase/decrease the c_1 and c_0 gimbal orientation coefficients (eq. 12)
- Randomly increase/decrease the c_2, c_1 and c_0 diffrential thrust coefficients (eq. 8)

For each of these disturbances, and sometimes even with several combined, the NMPC has always been able to compensate and make the drone flyable in simulation. It was always able to track its reference, stabilize itself around it and even perform fairly complex maneuvers. Nevertheless, for example, when the drone's mass is modified and when the drone is stable around its reference frame, even though it does not move, the controller still seems to predict a small future displacement due to the fact that the dynamics are altered and therefore do not correspond to actual behavior. In its scheme, the NMPC predicts an action that does not happen because of the model mismatch. This behavior was expected.

5 Real-world framework and embedded NMPC

5.1 Hardware in the loop (HiL)

The last step before potentially being able to test flying the drone in its full experimental framework, with the controller embedded in the on-board computer, is to verify its proper functioning on the Raspberry Pi [12]. Nevertheless, this still has to be done inside the simulation loop. Basically, this means that the controller (main control and MPC) operates within the on-board computer (*slave*), but that the simulation and visualization functions still run on the *master* computer. So it is still the simulation that emulates the vehicle's states and dynamics, with the hardware integrated into the simulation loop. This implies that the *master* computer must communicate the drone states determined by the simulation at each time step, and that the on-board *slave* computer must send the controls once the NMPC has computed them. This communication is achieved via the wifi connection between the Raspberry Pi and the computer, which enables the ROS topics running on the *slave* computer to be streamed to the *master* computer.
This stage enabled to observe 2 main things. The first was that the NMPC running on the Raspberry Pi was slightly slower than on the computer (which was to be expected), but that overall the computation time, although still too slow, remained reasonable. The measured solving time resulting from these simulations varied between **45**ms and **55**ms (LAMP formulation at 50Hz, $N_{segs} = 25$, $H = 2$s), compared with **30-45**ms on the computers. The second observation is that, despite the above, simulations requiring real-time wifi state transfer between the Rapsberry Pi and the computer were very unstable due to delays. These significantly slow down the control loop and destabilize the drone and making it almost every time uncontrollable.

5.2 Full experimental setup

The final stage of this project is to finally attempt to operate all the elements discussed, and in particular the NMPC, within the drone, so that if all the lights are green, being able to carry out a test flight. The drone must be autonomous and capable to track its real-world reference by itself, using the NMPC controller. To achieve this, several elements need to be adapted. Firstly, and for the first time in this project, it is no longer the simulations that are responsible for returning the vehicle's states to the controller. Indeed, the drone must be able to determine its state via its IMUs, but above all via the GPS coordinates it receives. For experimental reasons, which require the drone to be tested indoors in a secure environment, this means that the GPS coordinates estimating its position must be replaced by the OptiTrack [13] system. This system uses infrared cameras on the ceiling and markers on the drone to determine the drone's position and attitude in real time.

The general structure of the system shown in the Figure 20 is organized as follows. Firstly, the Raspberry Pi is responsible for receiving position and orientation information from the OptiTrack via a connection to the VRPN server, which streams these information in real time. It must then transform these coordinates into the desired format and frames in order to emulate real GPS data. It then communicates these coordinates to the second component of the on-board computer, the Pixhawk 4. The Pixhawk has several tasks, the first being to fuse the information provided by the OptiTrack (or the GPS if outdoors) with the data measured by its own IMUs using a Extended Kalman Filter (EKF). It can then return the ROS topic containing the drone's states $\mathbf{x}$ (`/state`) to the Raspberry Pi. The second task of this Pixhawk 4 is to receive commands from the remote operator. The remote controller can then modify the coordinates of the state to be tracked ($\mathbf{x}_{ref}$) and fly the drone as desired. This reference state is also transmitted to the Raspberry Pi via the ROS topic `/control_ref`. The Raspberry's main role is to compute the controls in order to track the desired reference. This can be achieved via the main control or, if enabled, via the MPC. In this case, the NMPC will compute the optimal controls over the horizon according to $\mathbf{x}$, $\mathbf{x_{ref}}$ and its internal parameters. It will then forward the first computed control (`/mpc/control`) to the main control node and to the Pixhawk (`/set/control`), which will be responsible for relaying this control to the actuators. Interactions between the Raspberry Pi and the Pixhawk are handled by the MAVROS [14] package, which provides the communication interface.
Finally, all this can be displayed on the laptop in real time, as the OptiTrack system and the Raspberry stream their topics to the visualization node, allowing to see both the drone state evolving according to its real-life movements in the environment and the NMPC prediction updated in real time in the classic visualization scope. It is also important to note that the pilot can manually switch from main control/MPC mode to the PID controller included in the Pixhawk when needed.

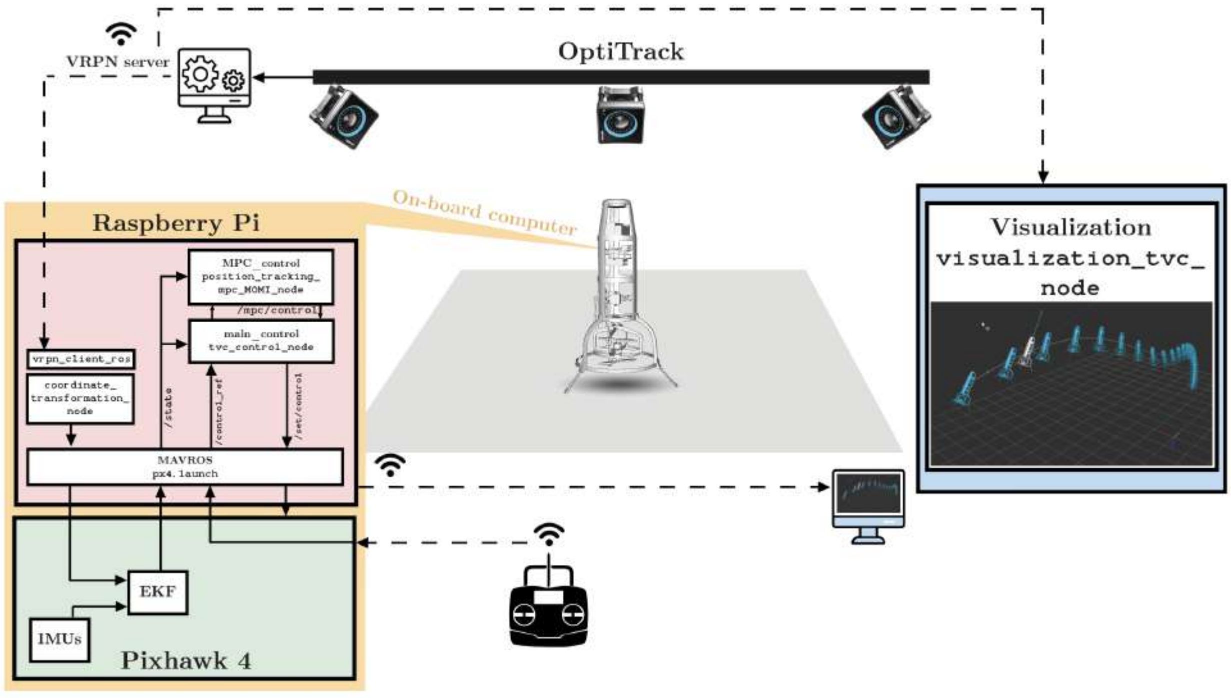


Figure 20: Full experimental framework

One of the main problems was to find a way to "trick" the Pixhawk into providing positional feedback from the OptiTrack and not directly from the GPS signal, as is supposed to be the case. These data had to respect a fairly precise formating.

After sorting out a number of small problems linked to the activation of the various nodes required on the Raspberry and network communication between the devices, we were able to test the various components together. This consisted in moving the drone around the room with the complete structure activated, MPC mode armed, but without supplying power to the motors. The goal was to check whether the movements were correctly mapped into the visualization frame and whether the prediction provided for the NMPC

was valid at all time.
The conclusion is rather positive: the NMPC prediction seems quite consistent and always seems to track the reference position, even if the latter is modified by the remote control. Another point already partly discussed in the previous section 5.1 is the OCP's solving time. Observations during this experiment show that this delay is around **60**ms in this full configuration (LAMP formulation). This seems to be quite a critical issue when it comes to the drone's stability in flight, and raises questions about the possibility of controlling the drone at real speed.
Another issue that unfortunately could not be resolved due to the lack of time was the mechanism allowing the NMPC controller to be disabled if it compromises the drone's stability or is unable to compute optimal controls in time. This point is a central element of flight safety which needs to be implemented in the futur, and raises the question of whether the MPC is deactivated for a reason, for how long, and how to reactivate it.
In view of all these grey areas and uncertainties, it was judged too unreliable at this stage to attempt a test flight in the full configuration and with motors powered.

6 Further improvements

As discussed in the previous section, the main improvements to be made to go further would be to add reliability to the control architecture by enabling better switching between the main control mode and the NMPC in an intelligent manner when the situation becomes problematic. It would be a plus, if possible, to be able to then re-arm the NMPC mode during flight. In addition to this, adding more restrictive constraints on inputs (for example $dF \in [0.3, 0.7]$) would offer greater safety and less aggressive maneuvers. Finally, if conditions allow, it would be a great opportunity to test several different flight experiments to analyze the real-world drone's behavior and get precious feedback.

In addition to this, identifying a more complete model of the drone, including an aerodynamic aspect for example, could be a great asset and provide more accurate dynamics.

6.1 Towards energy optimality

As mentioned in the title, in addition to developing an NMPC embedded into the drone's on-board computer, the second objective of this project is to minimize manoeuvers energy consumption. Implementing an "energy-optimal" NMPC for rocket control brings significant benefits. By optimizing energy consumption throughout the rocket's flight trajectory, it enables a reduction in the required fuel mass for the mission. This reduction not only decreases costs but also increases the available payload mass that the rocket can carry. By maximizing the energy efficiency of the rocket's operations, we can achieve greater mission capabilities, explore farther destinations, deliver larger payloads, or land the vehicle using less fuel consumption and ultimately pushing the boundaries of space exploration by enhancing efficiency. As an example, the Jet Propulsion Laboratory (JPL) of NASA has developed landing algorithms aimed at finding the optimal trajectory that minimizes fuel consumption [15].

In this project, the drone's role is to mimic the behavior of a real rocket. The orientation of the gimbal is similar to that of the nozzle, and the propellers replicate the thrust generated by a jet engine. In the context of energy optimality, which can be understood as minimizing the fuel used for a given maneuver in the real rocket, this is quite similar to minimizing the battery's electrical consumption in the drone which in a way emulates the fuel tank. The idea would be to add the battery's state of charge (SOC) [16] as an additional system state, and therefore as part of the OCP to minimize the current consumption when planning a trajectory.

$$\mathrm{SOC}(t) = 100\left(SOC_0 - \int_0^t \frac{I}{Q}dt\right) \tag{20}$$

With I being the current, SOC_0 the initial state of charge and Q the battery capacity.

For this system to work, it would be necessary to implement current consumption as a function of a certain thrust command (dF) within the dynamics, so that the NMPC can predict the future SOC. This implies an additional identification experiment which should, as for the other parameters identified in section 2.2, enable the dF commands to be related to the current flowing out of the battery (I) using a polynomial interpolation. To carry out this identification experiment, it would probably be sufficient to simply connect a current probe to the battery output while performing the thrust identification experiment discussed in the section 2.2.2 and record the results for polynomial fitting.

Unfortunately, due to the lack of time available and the decision to focus on embedding the controller and solving the inherent problems to possibly carry a test flight, this strategy could not be implemented during this project.

7 Conclusion

The initial purpose of this project was to develop an NMPC embedded in a TVC drone, simulating the behavior of a rocket and providing an approach that minimized energy consumption during maneuvers. The first step in the project was to identify the physical model of the drone by carrying out a number of experiments to relate the inputs given to the system to its dynamic response. This resulted in a relatively simple model that captured the main physical components of the vehicle and enabled simulations to be carried out. This model remains simple and fairly basic, and could easily be made more complex to better capture certain factors such as the aerodynamics forces acting on the drone. The second stage of the project was to develop the architecture of the NMPC, defining the OCP and the key elements of its resolution first in MATLAB and then in the full C++/ROS framework. This stage revealed the difficulties inherent in solving this kind of problem in real time, pointing out the issues linked to the consistency of the solution obtained and to the solving time, which must be kept as low as possible. The final step was to adapt the controller and all its components to the on-board computer for embedding. Although this stage has not yet led to autonomous drone flight, it has shown significant progress in this direction, with all components interacting in real time. Once again, this stage highlighted the time needed to resolve the OCP once embedded in a Raspberry PI, which is unfortunately still too slow, and the difficulties involved in developing a safe and reliable control architecture. These two main elements will need to be improved in the future to be able to carry out a test flight in proper conditions. Finally, the energy-optimal aspect could unfortunately not be implemented and tested during this project. Integrating this aspect would be beneficial in the future, since its potential application for real space missions is not in doubt.
For me, this project has been very enriching in terms of the diversity of elements and areas involved, and in terms of demonstrating the difficulties inherent in developing and embedding optimal controllers in a real robotic system.

I'd like to thank Johannes and Pietro for their helpful collaboration throughout the project, which was always very productive and enjoyable.

References

[1] PX4 Autopilot. Pixhawk 4, 2023. URL: `https://docs.px4.io/main/en/flight_controller/pixhawk4.html`.

[2] Bota Systems. Bota systems medusa (bft-meds-ecat-m8). URL: `https://www.botasys.com/force-torque-sensors/medusa`.

[3] K. Michael. The experimental determination of the moment of inertia of a model airplane. *Honors Research Projects. 585.*, 2017.

[4] M. Diehl. Real-time optimization for large scale nonlinear processes. *PhD Thesis, University of Heidelberg*, 2001.

[5] L. T. Biegler, A. M. Cervantes, and A. Wächter. Advances in simultaneous strategies for dynamic process optimization. *Chemical Engineering Science*, 2002.

[6] C. Magherini L. Brugnano, F. Iavernaro. Efficient implementation of radau collocation methods. *Applied Numerical Mathematics, vol. 87, p. 100-113*, 2015. `doi:https://doi.org/10.1016/j.apnum.2014.09.003`.

[7] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming 106(1), pp. 25-57*, 2006.

[8] Roland Schwan, Yuning Jiang, Daniel Kuhn, and Colin N. Jones. Piqp: A proximal interior-point quadratic programming solver, 2023. `arXiv:2304.00290`.

[9] CasADi. Casadi, 2023. URL: `https://web.casadi.org`.

[10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[11] Open Robotics. rviz. URL: `http://wiki.ros.org/rviz`.

[12] Raspberry Pi Ltd. Raspberry pi 4. URL: `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/`.

[13] NaturalPoint Inc. Optitrack. URL: `https://optitrack.com`.

[14] Open Robotics. Mavros. URL: `http://wiki.ros.org/mavros`.

[15] B. Acikmese and S. R. Ploen. Convex programming approach to powered descent guidance for mars landing. *Journal of Guidance, Control, and Dynamics, vol. 30, no. 5, pp. 1353–1366*, 2007. `doi:10.2514/1.27553`.

[16] R. Głębocki M. Jacewicz, M. Żugaj and P. Bibik. Quadrotor model for energy consumption analysis. *Energies, vol. 15, no. 19, p. 7136*, 2022. `doi:10.2514/1.27553`.