Comparison of Direct Methods for NMPC Applied to a Thrust-Vector-Controlled Drone

Isidore Mones and Heidi Dixon

October 14, 2025

Introduction

This paper is part of a larger project inspired by the small-scale rocket prototype presented by Linsen et al. (2022) which aims to study thrust vector control (TVC) and vertical takeoff and landing (VTOL) using a gimballed-thrust rocket prototype. VTOL milestones like NASA's Lunar Landing Research Vehicle (LLRV) which allowed Apollo astronauts to practice lunar landings, McDonnell Douglas's DC-X (Delta Clipper), and SpaceX's Falcon 9 landings, and the planned Artemis III mission that aims to land and return astronauts from the moon illustrate the value of propulsive VTOL with TVC. About the upright hover equilibrium, vertically flying rockets are open-loop unstable in attitude. They have highly nonlinear, coupled dynamics, and require fast control algorithms that respond quickly to disturbances and sensor noise. Unfortunately, validating control techniques on full-scale rockets is prohibitively costly and high-risk. There is a need for inexpensive test vehicles that allow experimentation with rocket guidance, navigation, and control. A gimballed-thrust drone with electric motors and propellers is a good choice for replicating rocket dynamics on a much smaller budget. A couple of low-cost designs have been proposed and implemented (Spannagl et al., 2021), (Linsen et al., 2022). The drone design by Linsen et al. (2022) allows experimentation with control algorithms for VTOL and TVC. We have designed a similar drone based on their images and design descriptions, and there remain many details and design choices to explore.

Our drone implements TVC and VTOL using a nonlinear model predictive control (NMPC) algorithm, allowing it to handle nonlinear rocket dynamics and manage constraints like gimbal angle range, servo rates, and thrust limits. NMPC optimizes control inputs across a finite time horizon at each time step. In this sense it optimizes a control sequence, allowing it to anticipate future control requirements. Revising the sequence at each time step allows it to respond to disturbances. The main downside of NMPC is its high computational load. NMPC solves a constrained Optimal Control Problem (OCP) that is transcribed into a finite dimensional nonlinear optimization problem (NLP) at every control step, and as a result it is much slower than simple control algorithms like proportional integral derivative (PID) controllers or linear quadratic regulators (LQR). There are a variety of ways to formulate an NMPC problem instance as a nonlinear programming problem. Different formulations may vary in their computation times and solution accuracy. The focus of this paper is experimenting with different choices of NLP formulations and evaluating their speed and accuracy. Our goal is to run our control algorithm on a Raspberry Pi 5 at 50 Hz so solution speed will be very important. We compare three approaches: a Runge-Kutta multiple-shooter method, an orthogonal collocation with Gauss-Radau nodes implemented in the **do-mpc** library (Fiedler et al., 2023), and a Chebyshev pseudospectral collocation approach. We compare the relative accuracy of these techniques and their relative efficiency.

Thrust Vector Drone Equations of Motion

The system state $\vec{x}$ includes translational position $\vec{p} = [x, y, z]$ and velocity $\vec{v} = [v_x, v_y, v_z]$ in the world frame, the attitude represented as a unit quaternion $\vec{q} = [q_x, q_y, q_z, q_w]$ and the angular velocity in the body frame

$\vec{\omega} = [\omega_x, \omega_y, \omega_z]$. The control variables include the two gimbal angles θ_1, θ_2, the average thrust between the two propellers $\overline{P}$ and the differential thrust between the propellers P_Δ to control the rotation about the body z axis.

$$\vec{x} = \begin{bmatrix} x & y & z & v_x & v_y & v_z & q_x & q_y & q_z & q_w & \omega_x & \omega_y & \omega_z \end{bmatrix}^T$$
$$\vec{u} = \begin{bmatrix} \theta_1 & \theta_2 & \overline{P} & P_\Delta \end{bmatrix}^T$$

The equations of motion are represented as a series of first-order vector differential equations:

$$\dot{\vec{p}} = \vec{v} \tag{1}$$
$$\dot{\vec{v}} = \frac{1}{m} R(\vec{q}) \vec{F_b} + \vec{g} \tag{2}$$
$$\dot{\vec{q}} = \frac{1}{2} Q(\vec{\omega}) \vec{q} \tag{3}$$
$$\dot{\vec{\omega}} = I^{-1} \left(\vec{M_b} - \vec{\omega} \times (I\, \vec{\omega}) \right) \tag{4}$$

where m is the mass of the drone and $\vec{g} = [0, 0, -9.81]^T$ is the acceleration due to gravity.

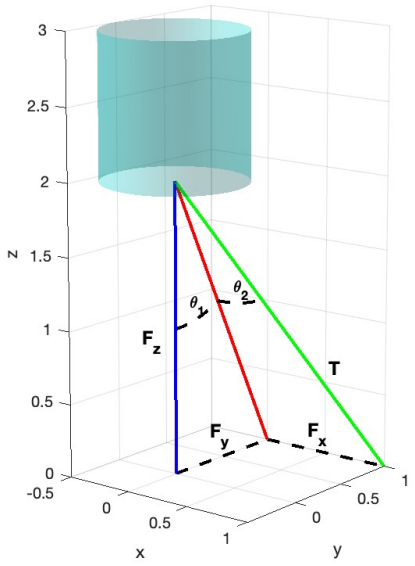
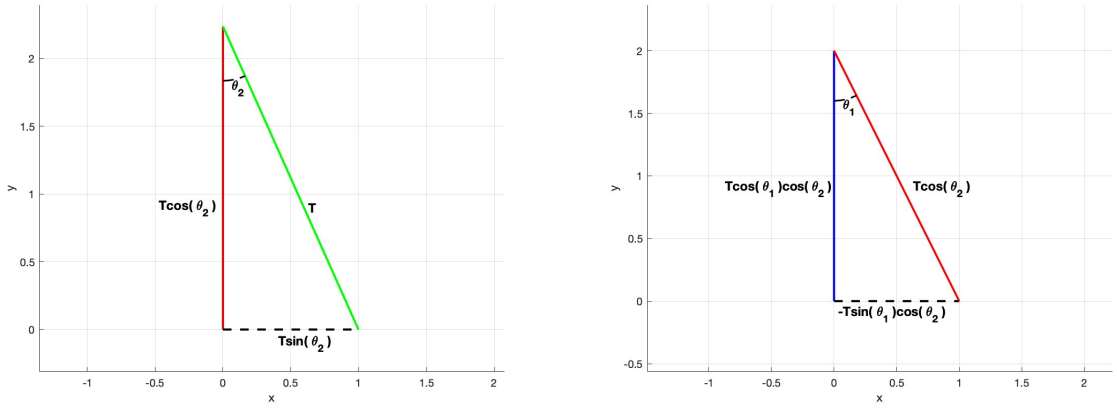


Figure 1: Three-dimensional thrust decomposition for a 2-axis gimballed drone.



(a) Thrust decomposition in the inner gimbal axis plane.

(b) Thrust decomposition in the x–z plane.

Figure 2: Two-dimensional thrust decompositions for a 2-axis gimballed drone.

Using vector decomposition we can find the thrust vector in body coordinates in terms of the gimbal angles θ_1 and θ_2 and magnitude of the thrust T acting on the drone.

$$\vec{F_b} = T \begin{bmatrix} \sin\theta_2 \\ -\sin\theta_1 \cos\theta_2 \\ \cos\theta_1 \cos\theta_2 \end{bmatrix}$$

$\vec{M_z}$ is the moment about the body z axis due to differential thrust.

$$\vec{M_z} = \begin{bmatrix} 0 \\ 0 \\ M_z \end{bmatrix}$$

The body frame moment vector is the sum of the torque due to thrust vector acting through the moment arm and the torque generated by the differential thrust. l is the length of the moment arm acting from the center of mass to the point $\vec{F_b}$ acts on the drone in body coordinates.

$$\vec{l} = \begin{bmatrix} 0 \\ 0 \\ -l \end{bmatrix}$$

$$\vec{M_b} = \vec{l} \times \vec{F_b} + \vec{M_z} = \begin{bmatrix} -lT\sin\theta_1\cos\theta_2 \\ -lT\sin\theta_2 \\ M_z \end{bmatrix}$$

The drone is designed to be symmetrical and the principal axes align with the body frame. The products of inertia vanish and inertia tensor can be written with only diagonal terms.

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

The body to world matrix is used to compute the force in world coordinates due to the body-centric thrust vector given the quaternion orientation.

$$R(\vec{q}) = \begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_wq_z) & 2(q_xq_z + q_wq_y) \\ 2(q_xq_y + q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_wq_x) \\ 2(q_xq_z - q_wq_y) & 2(q_yq_z + q_wq_x) & 1-2(q_x^2+q_y^2) \end{bmatrix}$$

The quaternion propagation matrix computes the change in quaternions given a set of angular velocities.

$$Q(\vec{\omega}) = \begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix}$$

The final extended dynamic equations are expressed as:

$$\dot{x} = v_x$$

$$\dot{y} = v_y$$

$$\dot{z} = v_z$$

$$\dot{v_x} = \frac{\sin\theta_2 * T(1 - 2(q_y^2 + q_z^2)) - \sin\theta_1 \cos\theta_2 * 2T(q_x q_y - q_w q_z) + \cos\theta_1 \cos\theta_2 * 2T(q_x q_z + q_w q_y)}{m}$$

$$\dot{v_y} = \frac{\sin\theta_2 * 2T(q_x q_y + q_w q_z) - \sin\theta_1 \cos\theta_2 * T(1 - 2(q_x^2 + q_z^2)) + \cos\theta_1 \cos\theta_2 * 2T(q_y q_z - q_w q_x)}{m} + g$$

$$\dot{v_z} = \frac{\sin\theta_2 * 2T(q_x q_z - q_w q_y - \sin\theta_1 \cos\theta_2 * 2T(q_y q_z + q_w q_x) + \cos\theta_1 \cos\theta_2 * T(1 - 2(q_x^2 + q_y^2))}{m}$$

$$\dot{q_x} = \frac{\omega_z q_y - \omega_y q_z + \omega_x q_w}{2}$$

$$\dot{q_y} = \frac{-\omega_z q_x + \omega_x q_z + \omega_y q_w}{2}$$

$$\dot{q_z} = \frac{\omega_y q_x - \omega_x q_y + \omega_z q_w}{2}$$

$$\dot{q_w} = \frac{-\omega_x q_x - \omega_y q_y - \omega_z q_z}{2}$$

$$\dot{\omega_x} = \frac{-lT \sin\theta_1 \cos\theta_2 - \omega_y \omega_z I_{zz} + \omega_y \omega_z I_{yy}}{I_{xx}}$$

$$\dot{\omega_y} = \frac{-lT \sin\theta_2 - \omega_x \omega_z I_{xx} + \omega_x \omega_z I_{zz}}{I_{yy}}$$

$$\dot{\omega_z} = \frac{M_z - \omega_x \omega_y I_{yy} + \omega_x \omega_y I_{xx}}{I_{zz}}$$

Note that these equations use the variable T representing the magnitude of the thrust vector acting on the drone. The actual control variables are defined as the average $\overline{P}$ and difference P_Δ between the two propeller thrust values T_1 and T_2.

$$\overline{P} = \frac{T_1 + T_2}{2}$$

$$P_\Delta = T_1 - T_2$$

T_1 and T_2 denote the normalized brushless motor command signals, ranging from 0 (no throttle) to 1 (full throttle).

$$T_1 = \overline{P} + P_\Delta/2$$

$$T_2 = \overline{P} - P_\Delta/2$$

To map the average thrust command signal value to the actual thrust force value in Newtons used in the thrust-vector decomposition a second-order polynomial fit is experimentally determined. Similarly, a linear relationship is used to map the differential thrust to the moment about the body frame z axis.

$$T = a\overline{P}^2 + b\overline{P} + c$$

$$M_z = dI_{zz}P_\Delta$$

Nonlinear Model Predictive Control (NMPC)

Model predictive control (MPC) (Garcia et al., 1989) is a control algorithm that solves an optimization problem at each time step. An MPC solution yields a series of control inputs over a finite time horizon, creating a *plan* of actions that can anticipate and adjust to future issues. For example, the optimal speed for a car at a given instance might change if the algorithm knew that it was entering a curve in the road in the next few time steps. The control values for the first step of the plan is applied to the system and then the plan is revised at the next time step, allowing the method to react to disturbances and noise. Nonlinear model predictive control (NMPC) is a variant of MPC that can handle nonlinear dynamics.

Generally, given a set of continuous time nonlinear differential equations $\dot{x} = f(x(t), u(t))$ and an initial state x_0, the finite horizon NMPC optimization problem is

$$\begin{aligned} \min_{u(t)} \quad & \int_0^{t_f} \ell(x(t), u(t))\, dt \\ \text{s.t.} \quad & \dot{x} = f(x(t), u(t)) \\ & x(0) = x_0 \end{aligned}$$

The solution $u(t)$ is a real valued function that minimizes the cost function over time interval $[0, t_f]$. The constraints require that $u(t)$ and $x(t)$ obey the system dynamics $\dot{x} = f(x(t), u(t))$ and the initial state $x(0)$ must be equal to the current state x_0. Given this framework, it is easy to add additional constraints that enforce system limitation such as minimum or maximum control values.

Equations (1-4) are strongly nonlinear, but can be linearized around the fixed point in which the drone is vertically balanced as in LQR and Linear MPC. However, the linearized equations quickly become inaccurate if the drone moves away from the fixed point by tilting or rotating. Working directly with the nonlinear equations allows more accurate predictions of the drone behavior. The NMPC framework also allows us to enforce the mechanical limitations of our thrust motors and gimbal servos. Libraries like CasADi (Andersson et al., 2018) and **do-mpc** (Fiedler et al., 2023) make NMPC easy to implement in practice.

An NMPC instance for the thrust vector drone is a minimization problem of the form

$$\begin{aligned} \min_{u(t)} \quad & \int_0^{t_f} \ell(x(t), u(t))\, dt \;+\; \phi(x_f) && (5) \\ \text{s.t.} \quad & \dot{x} = f(x(t), u(t)) \\ & x(0) = x_0 \\ & -\theta_{max} \le \theta_1 \le \theta_{max} && (6) \\ & -\theta_{max} \le \theta_2 \le \theta_{max} && (7) \\ & T_{min} \le \overline{P} + P_\Delta/2 \le T_{max} && (8) \\ & T_{min} \le \overline{P} - P_\Delta/2 \le T_{max} && (9) \\ & 0 \le z. && (10) \end{aligned}$$

The cost function (5) minimizes both the squared error between the current state $x(t)$ and the goal state x_{ref} weighted by diagonal matrix Q and the error between the current control $u(t)$ and the goal control u_{ref} weighted by diagonal matrix R.

$$\ell(x(t), u(t)) = (x(t) - x_{ref})^T Q (x(t) - x_{ref}) + (u(t) - u_{ref})^T R (u(t) - u_{ref})$$

We also add a terminal cost

$$\phi(x_f) = (x(t_f) - x_{ref})^T Q_f (x(t_f) - x_{ref})$$

The problem instance also enforces minimum and maximum gimbal angles (6,7), minimum and maximum values for each thrust motor (8, 9) and restrictons on the drone's vertical height (10).

NLP Formulations

To solve the continuous time optimal NMPC instance, it is formulated as a discrete time finite dimensional nonlinear problem (NLP) and solved by a standard NLP solver. NLP solvers are well established, highly optimized methods that originated in the field of operations research. Unfortunately, despite the availability of efficient solvers for these problems, NLP solutions for NMPC problem instances may be too slow to produce control inputs for a thrust vector drone which requires a fast response time. This is a prime drawback of MPC and NMPC algorithms over LQR and PID methods which can be computed very quickly. Ideally, we'd like to run our control algorithm at 50Hz on a Raspberry Pi 5. To meet this timing restriction we need to build an NMPC instance that can be solved efficiently.

An NLP problem over a finite set of decision variables w has the form

$$\min_{w \in \mathbb{R}^n} \; F(w) \tag{11}$$

$$\text{s.t.} \;\; G(x_0, w) = 0 \tag{12}$$

$$H(w) \leq 0 \tag{13}$$

It has an optimization function (11), a set of equality constraints (12) and a set of inequality constraints (13). In practice, the distinction between equality and inequality constraints is moot since an equality constraint can be written as two inequality constraints and an inequality constraint can be written as an equality constraints with the introduction of a slack variable.

To turn our NMPC problem into an NLP problem the continuous time functions $x(t)$ and $u(t)$ are parameterized into a finite set of discrete variables over the control trajectory. Then differential equations for the system dynamics are discretized using a numerical integration method. There are a many ways to do this and the choices we make will affect the size of the problem, the speed at which it can be solved, and the accuracy of the numerical approximation. Typically there is a tradeoff between speed and accuracy. Next we explore three different ways to formulate our NMPC problem as an NLP and compare and contrast their ability produce viable control trajectories and their relative efficiency. These three methods are in no way exhaustive. We chose to implement pseudospectral collocation because it was shown to be effective in an earlier paper on thrust vector drones (Linsen et al., 2022) and we chose multiple shooting and orthogonal collocation because they were very easy to implement.

To solve our non-linear programming (NLP) problems we used the `ipopt` solver (Wächter and Biegler, 2006) that comes installed with CasADi 3.7.1 (Andersson et al., 2018). This solver requires an additional subroutine to solve sparse matrix systems. We experimented with the `mumps` solver that comes with CasADi, and also tried using the `ma27` and `ma57` solvers from HSL (Harwell Subroutine Library) (HSL, 2013). All solvers were compiled with Apple clang version 17.0.0.

The `mumps` solver was slower than both `ma27` and `ma57` on all instances. The `ma57` solver was substantially slower than `ma27` for our orthogonal collocation instances and only a tiny bit slower than `ma27` on our Chebyshev pseudospectral collocation instances. The performance of `ma27` and `ma57` on the multiple shooter instances was about the same. The `ma57` solver is considered a better solver than the older `ma27` solver, but our particular problem instances seem to be in the small class of problems that do better with `ma27`. We suspect that our problems are not large enough to benefit from the additional performance optimizations in `ma57`. As a result, all of our experiments were run with `ma27`. All solvers and experiments were run with the following solver settings.

```
ipopt_settings = {
    'ipopt.max_iter': 100,
    'ipopt.tol': 1e-3,
    'ipopt.acceptable_tol': 3e-2,
    'ipopt.linear_solver': 'ma27',
}
```

To compare these three methods, we need to first find optimal settings for each method. Then we can compare the methods in terms of solution time and trajectory accuracy. While solution times are easy to compare, trajectory accuracy is harder to assess. Good trajectories show state values that converge promptly to the goal state. How promptly they need to converge will depend on the requirements of the application. We want the simplest and smoothest state trajectories that efficiently achieve the desired goal. Unneccesary spikes or irregularities should be avoided. Unfortunately, good trajectories are hard to characterize mathematically. It was difficult to find metrics that reliably identified good trajectories. To assess accuracy we looked at the following metrics

- We stored the status result of the IPOPT solver. A result of `Solver_Succeeded` means that the NLP converged properly and all constraints were satisfied.
- We found it useful to graph the objective function cost for the returned NLP solution. Trajectory spikes often occurred with a simultaneous spike in objective function cost suggesting the solver had converged properly, but to a local minimum. NLP solvers are not guaranteed to produce a globally optimal solutions. Tracking the objective function cost helped identify and quantify these poor solutions.
- When appropriate we used reference trajectories to assess accuracy. To create a reference trajectory we use an NLP encoding that we know is highly accurate but typically too expensive to compute in practice. Faster NLP encodings can then be compared to the reference trajectory through an appropriate distance metric. We found reference trajectories useful when tuning individual methods, but less useful for comparing different methods. You can have multiple good trajectories that are not close in terms of distance metrics. This limits the value of reference trajactories as they can penalize trajectories that are actually good.
- We found that full accuracy assessments couldn't be done in a purely analytic fashion. All trajectories needed a visual inspection of the graphs to ensure that weird or unsuitable trajectories were noticed and avoided.

We selected four test cases from a larger set of test cases to run simulations on. These cases were chosen to reflect the variety of types of situations we expected the drone to encounter.

1. The first was a simple hover simulation where the drone begins in the goal state and is expected to hover in that state for the duration of the simulation. You would think that a hover simulation would be the easiest for an implementation to handle, but we found that when the drone is very close to the optimal state there is less direction for the NLP problem. This makes the IPOPT solver prone to finding local minimums.
2. The drone begins with a 45° rotation about the vertical z axis and must unwind itself to achieve the goal state.
3. The drone begins at the (x, y, z) point $(1, 0, 1)$ with an initial velocity in x direction. The drone must return to the point $(0, 0, 0)$.
4. The drone begins at the point $(0, 1, 0)$ with a 15° rotation about the x axis and must return to the point $(0, 0, 0)$.

Multiple Shooting (with Runge–Kutta discretization)

For multiple shooting, we divide our time horizon into N evenly spaced nodes $\{t_0, t_1, \ldots, t_N\}$ of length Δt and we make copies of our state x_k and control u_k variables for each time step. The initial state is initialized to the current state

$$x_0 - X_0 = 0.$$

Using the system dynamics $\dot{x} = f(x(t), u(t))$ and Runge-Kutta within each time step $[t_k, t_{k+1}]$ we forward integrate our differential equations

$$\begin{aligned} k_1 &= f(x_k, u_k, t_k), \\ k_2 &= f\big(x_k + \tfrac{\Delta t}{2} k_1,\, u_k,\, t_k + \tfrac{\Delta t}{2}\big)\,, \\ k_3 &= f\big(x_k + \tfrac{\Delta t}{2} k_2,\, u_k,\, t_k + \tfrac{\Delta t}{2}\big)\,, \\ k_4 &= f(x_k + \Delta t\, k_3,\, u_k,\, t_k + \Delta t)\,, \end{aligned}$$

$$x_{k+1} - \Big(x_k + \tfrac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Big) = 0, \qquad k = 0, \ldots, N-1. \tag{14}$$

to create constraints (14) that enforce the change of state between time steps.

The cost function is defined to penalize deviation from reference state as well as deviation from reference control input. There is also a terminal cost that incentivizes long term stability by weighting the final state more heavily.

$$x_r = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}^T, \qquad u_r = \begin{bmatrix} 0 & 0 & T_{\text{hover}} & 0 \end{bmatrix}^T.$$

$$\sum_{k=0}^{N-1} \Big[(x_k - x_r)^T Q (x_k - x_r) + (u_k - u_r)^T R (u_k - u_r) \Big] + (x_N - x_r)^T Q_f (x_N - x_r) \tag{15}$$

Constraints on gimbal angles and thrust motors are encoded by adding a copy of the appropriate inequalities at each time step. To improve performance of our multiple shooter method we provide an initial guess for the NLP by using the solution from the previous time step.

Multiple shooting considers the state and control variables at every time step to be optimization variables so the full NLP vector is:

$$\begin{bmatrix} x_0 & u_0 & x_1 & u_1 & \ldots & x_{N-1} & u_{N-1} & x_N \end{bmatrix}$$

This can easily become computationally demanding for long horizons and high-dimensional systems. For a thrust vector drone system with a horizon length of 100 time steps there are $13 * (100 + 1) + 4 * 100 = 1713$ decision variables. While the number of variables is very high, the constraint matrix is typically very sparse since each constraint references only a small number of variables.

Multiple shooting is easy to implement using CasADi. While our final NMPC implementation will run at 50Hz, the time steps of our multiple shooting formulation don't need to be 0.02 seconds. To find the optimal settings we tried different sized time steps. The horizon, N was selected to form a full two second time horizon given the time step. Experiements were run on a Macbook Pro 3.49GHz Apple M2 chip with 8GB RAM. For each time step we ran all four of our simulations and evaluated both time and accuracy.

Table (1) shows average CPU time for each choice of time step. The larger the time step, the faster the solution time. Time steps of size 0.2 or larger gave times fast enough to run on the Raspberry Pi 5 at 50Hz.

The fourth simulation was the only simulation that caused the NLP to fail to converge and all choices of time step experienced failures on this instance. We did compare each solution to a reference trajectory that was produced with a time step of 0.02 and $N = 100$. To compute the accuracy of a time step we look at the squared error of the trajectory summed over each iteration. If x_i is the simulated state at iteration i and r_i is the reference run state at iteration i then the accuracy is computed as

$$\text{accuracy} = \sum_{i=1}^{200} (x_i - r_i)^T (x_i - r_i)$$

However, we excluded the fourth simulation since the reference trajectory was irregular on that simulation. Time steps from size 0.02 all the way to 0.4 all produced good and remarkably similar trajectories. For larger time steps the rate of convergence to the goal state increased and trajectories failed to converge for a time step of 1 second. Based on these experiments we can chose a time step of size 0.2 seconds with a horizon $N = 10$ for this multiple shooting implementation.

time step, horizon	time (sec)	NLP fails	ref distance
$ts = 0.02, N = 100$	0.054	4	0.0
$ts = 0.025, N = 80$	0.043	3	0.006
$ts = 0.04, N = 50$	0.027	3	0.087
$ts = 0.05, N = 40$	0.022	3	0.14
$ts = 0.1, N = 20$	0.01	3	0.50
$ts = 0.2, N = 10$	0.006	4	1.8
$ts = 0.4, N = 5$	0.004	40	8.3
$ts = 0.5, N = 4$	0.003	46	14.3
$ts = 1, N = 2$	0.002	76	60

Table 1: Comparison of average CPU time in seconds and number of failed NLP solutions for a series of multiple shooting simulations for different size time steps within a two second horizon.

Orthogonal Collocation using do-mpc

Of all the methods we implemented, orthogonal collocation was the easiest. We used the open source **do-mpc** Python library for model predictive control (Fiedler et al., 2023) which implements MPC and NMPC using orthogonal collocation with Radau nodes (Biegler, 2010). The **do-mpc** library is ideal for prototyping MPC and NMPC algorithms and requires minimal understanding of MPC and no understanding of orthogonal collocation to build functioning algorithms.

Orthogonal collocation is a local method that divides the time horizon into finite elements and fits polynomials to each interval. The dynamics are enforced throughout each interval at a set of K Gauss-Radau nodes: the $K-1$ interior nodes, which are the roots of a Jacobi polynomial $P_{K-1}^{(\alpha,\beta)}$, and one additional endpoint node, in this case the right endpoint, $\tau = 1$. The Jacobi parameters α and β determine the distribution of nodes throughout the interval, and a right-Radau formulation corresponds to $\alpha = 1$, $\beta = 0$.

$$\gamma_0 = 1, \qquad \gamma_j = \gamma_{j-1}\frac{(K-j+1)(K+j+\alpha+\beta)}{j(j+\beta)}, \qquad j = 1, \ldots, K$$

When P_K are the Gauss-Jacobi polynomials:

$$P_K^{(\alpha,\beta)}(\tau) = \sum_{j=0}^{K}(-1)^{K-j}\gamma_j\tau^j$$

Lagrange polynomials, $\ell_j(\tau)$, are used to interpolate the state trajectory $z^K(\tau)$, where z_{ij} is the state at the ith interval and the jth node:

$$\ell_j(\tau) = \prod_{\substack{m=0 \\ m\neq j}}^{K}\frac{\tau-\tau_m}{\tau_j-\tau_m}$$

$$z^K(\tau) = \sum_{j=0}^{K} z_{ij}\ell_j(\tau)$$

By differentiating the state trajectory and equating it with the dynamics $\dot{x} = f(x(t), u(t))$, a dynamics constraint is formed:

$$h_i = t_i - t_{i-1}$$

$$\sum_{j=0}^{K} z_{ij}\frac{d\ell_j(\tau_k)}{d\tau} = h_i f(z_{ik}, t_{ik}), \qquad k = 1, \ldots, K.$$

Continuity constraints are used to fit an interval polynomial's endpoint with the first point of the next interval. There are two additional continuity constraints for enforcing the overall horizon's start and endpoint continuity.

$$z_{i+1,0} = \sum_{j=0}^{K} \ell_j(1) z_{ij}$$

$$z_f = \sum_{j=0}^{K} \ell_j(1) z_{Nj}, \qquad z_{1,0} = z_0$$

The continuous Bolza cost function is defined as:

$$\min_{x(t),u(t)} J = \Phi(z(t_f), t_f) + \int_{t_0}^{t_f} L(z(t), u(t), t) dt$$

Where $\Phi(z(t_f), t_f)$ is the Mayer term or terminal cost. From here onward the t_f input will not be included because our system has a fixed horizon. $L(z(t), u(t), t)$ is the running cost in continuous time. We will first discretize the system and then use a change of variables to convert from t to τ.

$$h_i = t_i - t_{i-1} \qquad t = t_{i-1} + h_i \tau \qquad dt = h_i d\tau$$

$$\int_{t_0}^{t_f} L(z(t), u(t), t) dt = \sum_{i=1}^{N} \int_{t_{i-1}}^{t_i} L(z(t), u(t), t) dt = \sum_{i=1}^{N} h_i \int_0^1 L(z(t_{i-1} + h_i\tau), u(t_{i-1} + h_i\tau), t_{i-1} + h_i\tau) d\tau$$

Using the Gauss–Radau quadrature rule (a variant of Gaussian quadrature, exact for polynomials up to degree $2K - 1$, we can substitute a quadrature sum of the cost at each given node for the integral of the running cost over the interval.

$$\int_a^b f(\tau)\, d\tau \;\approx\; \sum_{k=1}^{K} \omega_k f(\tau_k)$$

$$t_{ik} = t_{i-1} + h_i \tau_k \qquad z_{ik} = z_i^K(\tau_k) \qquad u_{ik} = u_i^K(\tau_k)$$

$$J \approx \Phi(z_f) + \sum_{i=1}^{N} h_i \sum_{k=1}^{K} \omega_k L(z_{ik}, u_{ik}, t_{ik})$$

The weights ω_k are determined by Radau quadrature, which integrates all polynomials up to $2K - 1$ with zero error.

Primal warm starts in which the previous iteration's solution is used as an initial guess for the current iteration's NLP problem are done automatically in do-mpc. To use **do-mpc**'s orthogonal collocation, we need to determine the optimal size for our finite element intervals and the degree K. To do this, we fixed the length of our time horizon to 2 seconds and we considered time intervals with sizes $[0.1, 0.2, 0.3, 0.4, 0.5]$ and polynomial degrees $c = 1, 2, 3$.

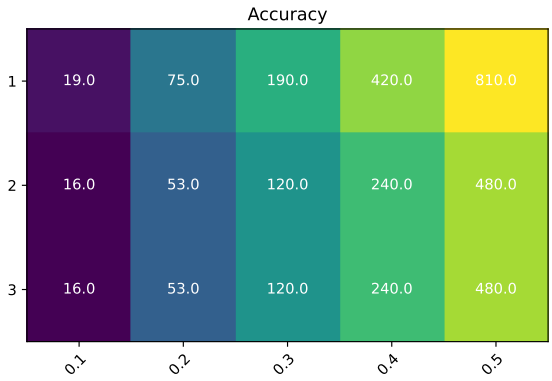
We ran a simulation that started with the drone in a perturbed state and with a the goal state being vertically balanced and motionless at the point $(x, y, z) = (0, 0, 0)$ in three dimensional space. Figure (3b) shows the average CPU time for each combination of time interval and degree.

To formally measure the accuracy of each combination, we first ran the simulation with a time interval of 0.02 seconds and a degree of 3. This is the most fine grained simulation we ran and should be the most accurate. On visual inspection, the state and control curves produced by this run were smooth and converged promptly to the goal state. We then used the state data from this run as a reference to compare accuracy of the other runs.

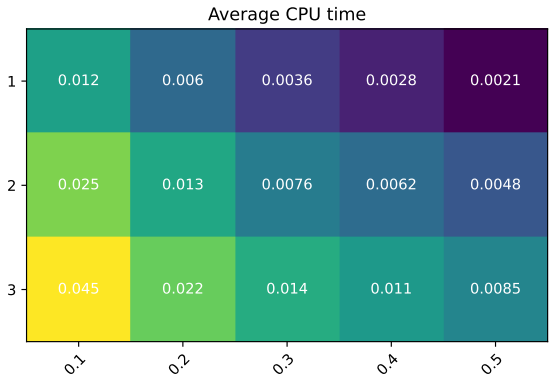
To compute the accuracy of a run we look at the squared error of the run summed over each iteration. If x_i is the simulated state at iteration i and r_i is the reference run state at iteration i then the accuracy is computed as

$$\text{accuracy} = \sum_{i=1}^{200} (x_i - r_i)^T (x_i - r_i)$$

Visual inspection of all the graphs showed that accuracy measures less then 120 produced smooth graphs that converged promptly. Above this accuracy level, graphs may containt irregularities, converge slowly or fail to converge at all. Accuracy measures for all runs are shown in Figure (3a). Combining the data from both graphs we see that a good choice of time interval and degree is a 0.3 time interval with 2 collocation points. With optimal choices for our time interval and polynomial degree we are ready to compare this method with our other methods.



(a) Accuracy metric for orthogonal collocation with different size time steps and collocation degrees.



(b) Average CPU time for orthogonal collocation with different size time steps and collocation degrees.

Chebyshev Pseudospectral Method

Multiple shooting is useful for its robustness to discontinuities and nonlinearity and its conceptual simplicity. However, thrust-vectoring dynamics are smooth and continuous, so many of the benefits of multiple shooting are unnecessary. Chebyshev pseudospectral collocation (Fahroo and Ross, 2000) is a global method that represents the trajectory of each state with a single interpolating polynomial of degree N over the entire horizon, defined by its values at a set of $N + 1$ nodes. It achieves spectral accuracy, with error decreasing exponentially as the number of nodes increases, while requiring far fewer decision variables than multiple shooting. The dynamics are enforced by constraining the derivative of the interpolant at each node to the dynamic equations. Chebyshev–Gauss–Lobatto nodes are used to cluster more of the nodes towards the beginning and end of the horizon, mitigating the Runge Phenomenon, where polynomial interpolation develops large oscillations near the endpoints. The cosine distribution ensures accuracy and stability. Nodes τ_j are defined as

$$\tau_j = \cos\left(\frac{j\pi}{N}\right), \qquad j = 0, 1, ..., N$$

These node values are defined from 1 to -1 instead of on the time horizon $[t_0, t_f]$. To determine the horizon time for a given τ a linear mapping is used:

$$t(\tau) = \frac{t_f - t_0}{2}\tau + \frac{t_f + t_0}{2}, \qquad \frac{dt}{d\tau} = \frac{t_f - t_0}{2}.$$

$$\frac{dx}{dt} = \frac{d\tau}{dt}\frac{dx}{d\tau} = \frac{2}{t_f - t_0}\frac{dx}{d\tau}$$

At each time step the dynamics are checked with:

$$\frac{dx}{d\tau}(\tau_j) \;\approx\; \sum_{j=0}^{N} D_{ij}\, x(\tau_j), \qquad i = 0, \ldots, N.$$

where D_{ij} is the Chebyshev differentiation matrix that will be defined below. Using the chain rule the τ and time derivatives can be related allowing the node differentiation matrix to be checked against the system dynamics $f(x(t), u(t))$.

$$\dot{x}(t_j) = \frac{2}{t_f - t_0} \sum_{j=0}^{N} D_{ij}\, x(\tau_j),$$

$$\sum_{j=0}^{N} D_{ij}\, x(\tau_j) = \frac{t_f - t_0}{2}\, f(x(t), u(t)), \qquad i = 1, \ldots, N-1.$$

Where D is the Chebyshev differentiation matrix defined by:

$$D_{N_{00}} = \frac{2N^2 + 1}{6}$$

$$D_{N_{jj}} = \frac{-x_j}{2(1 - x_j^2)}$$

$$D_{N_{ij}} = \frac{c_i}{c_j}\frac{(-1)^{i+j}}{x_i - x_j}$$

$$D_{N_{NN}} = -\frac{2N^2 + 1}{6}$$

$$c_{i/j} = \begin{cases} 2 & i/j = 0 \text{ or } N, \\ 1 & \text{otherwise.} \end{cases}$$

For $N = 6$ the differentiation matrix is:

$$D_6 = \begin{bmatrix} \frac{73}{6} & -(8+4\sqrt{3}) & 4 & -2 & \frac{4}{3} & 4\sqrt{3}-8 & \frac{1}{2} \\ 2+\sqrt{3} & -\sqrt{3} & -(1+\sqrt{3}) & \frac{2}{\sqrt{3}} & 1-\sqrt{3} & \frac{1}{\sqrt{3}} & \frac{1-\sqrt{3}}{1+\sqrt{3}} \\ -1 & 1+\sqrt{3} & -\frac{1}{3} & -2 & 1 & 1-\sqrt{3} & \frac{1}{3} \\ \frac{1}{2} & -\frac{2}{\sqrt{3}} & 2 & 0 & -2 & \frac{2}{\sqrt{3}} & -\frac{1}{2} \\ -\frac{1}{3} & \sqrt{3}-1 & -1 & 2 & \frac{1}{3} & -(1+\sqrt{3}) & 1 \\ \frac{1-\sqrt{3}}{1+\sqrt{3}} & \sqrt{3} & \sqrt{3}-1 & -\frac{2}{\sqrt{3}} & 1+\sqrt{3} & \sqrt{3} & -(2+\sqrt{3}) \\ -\frac{1}{2} & 8-4\sqrt{3} & -\frac{4}{3} & 2 & -4 & 8+4\sqrt{3} & -\frac{73}{6} \end{bmatrix}$$

The Chebyshev cost function is:

$$J = \frac{t_f - t_0}{2} \int_{t_0}^{t_f} \ell(x(t), u(t), t)\, dt \;+\; \phi(x(t_f))$$

$$J \;\approx\; \frac{t_f - t_0}{2} \sum_{j=0}^{N} w_j\, \ell(x_j, u_j, t_j) \;+\; \phi(x_N), \qquad t_j = \frac{t_f - t_0}{2}\, \tau_j + \frac{t_f + t_0}{2}.$$

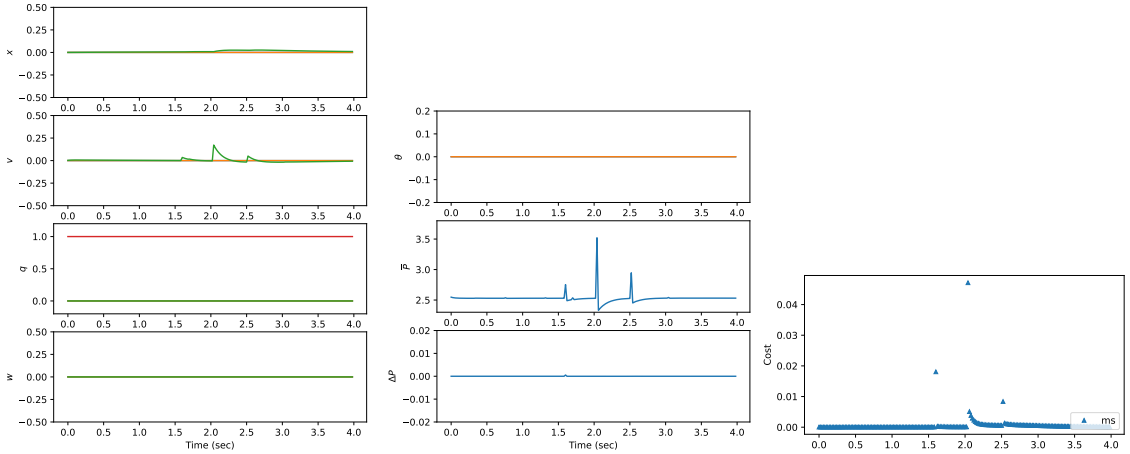
where $\phi(x(t_f))$ is the terminal cost and w_j are the Clenshaw-Curtis quadrature weights:

$$w_{\mathrm{j_{even}}} = \begin{cases} \dfrac{2}{N}\left(1 - \displaystyle\sum_{n=1}^{\frac{N}{2}-1} \dfrac{2}{4n^2-1}\cos\left(\frac{2nj\pi}{N}\right) - \dfrac{(-1)^j}{N^2-1}\right), & 1 \le j \le N-1, \\ \dfrac{1}{N^2-1}, & j = 0 \text{ or } j = N \quad (\text{since } N \text{ even}). \end{cases}$$

$$w_{\mathrm{j_{odd}}} = \begin{cases} \dfrac{2}{N}\left(1 - \displaystyle\sum_{n=1}^{\frac{N}{2}-1} \dfrac{2}{4n^2-1}\cos\left(\frac{2nj\pi}{N}\right)\right), & 1 \le j \le N-1, \\ \dfrac{1}{N^2}, & j = 0 \text{ or } j = N. \end{cases}$$

collocation points	time (sec)	NLP fails
6	0.003	0
8	0.008	2
10	0.013	3
12	0.021	4
14	0.03	2

Table 2: Comparison of average CPU time in seconds and number of failed NLP solutions for a series of Chebyshev pseudospectral simulations for different numbers of collocation points.



(a) state trajectory (b) control trajectory (c) cost function

Figure 4: Simulation of a hover using Chebyshev pseudospectral collocation with 10 collocation points.

We ran our Chebyshev pseudospectral implementation on our four test cases. We used a two second time horizon and varied the number of collocation points between 2 and 14. When the number of collocation points was less than 6 the trajectories failed to converge. The average CPU time for different numbers of collocations points is seen in Table (2) along with the number of NLP failures. Solution times increase as the number of collocation points increase.

Assessing the accuracy of the graphs was more complicated. This formulation would sometimes converge successfully on an iteration, but fail to find a good solution. We would typically see a corresponding jump

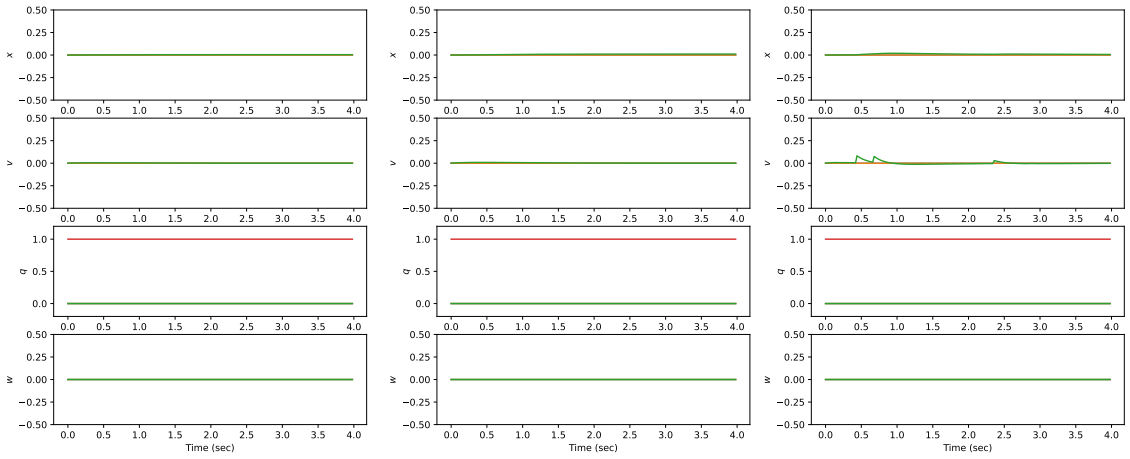
in the objective function cost at the same iteration. An example of this can be seen in Figure (4) where three poor solutions are seen in the objective function cost graph and we also see corresponding spikes in the control and state trajectories. These poor solutions were most apparent on the hover simulation but were also seen in the final simulation where there was a rotation about the x axis. These trajectory spikes persisted even when the number of collocation points were increased. It's possible the addition of rate constraints for the thrust control value might give the NLP more direction away from these local minimums and toward a more globally optimal solution. Based on these experiments, we chose to run our Chebyshev pseudospectral collocation implementation with 6 collocation points.

Comparison of Methods

To compare our different formulations for the NLP problem, we ran simulations on all four of our test cases. All experiements were run on a Macbook Pro 3.49GHz Apple M2 chip with 8GB RAM. We initially ran a set of seven different simulations. For each simulation, we computed the average time required to solve the NLP problem at each iteration. The quality of the solutions produced by each method was assessed by comparing the state and control graphs through time.

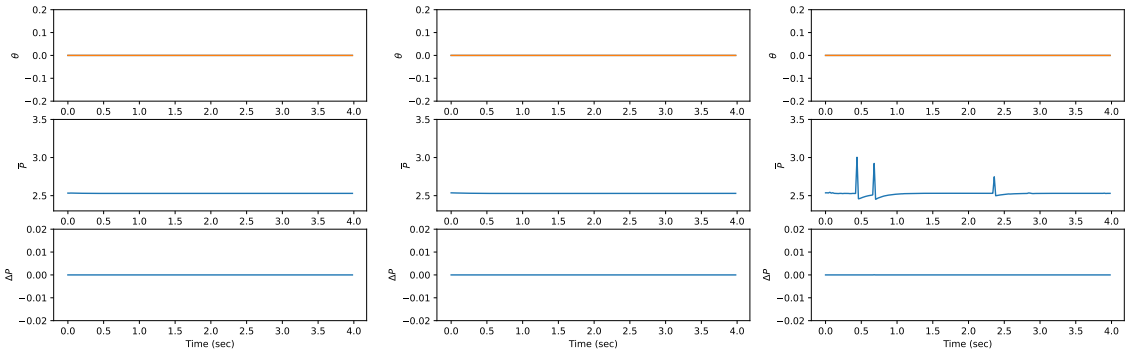
Results

We begin with our first simulation which requires the drone to hover in the goal state. Figure (5) shows a graph of the drone's state through time for each of our methods: multiple shooter, orthogonal collocation, and Chebyshev pseudospectral. The figures (11), (14) and (5) show similar graphs for our other three simulations. Comparison graphs of the control variables are given in figures (9), (12), (15) and (6). Finally, graphs comparing the CPU time for NLP calls throughout the simulations are given in figures (10), (13), (16) and (7).



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 5: State data on a simulation with initial state equal to goal state.



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 6: Control data on a simulation with initial state equal to goal state.

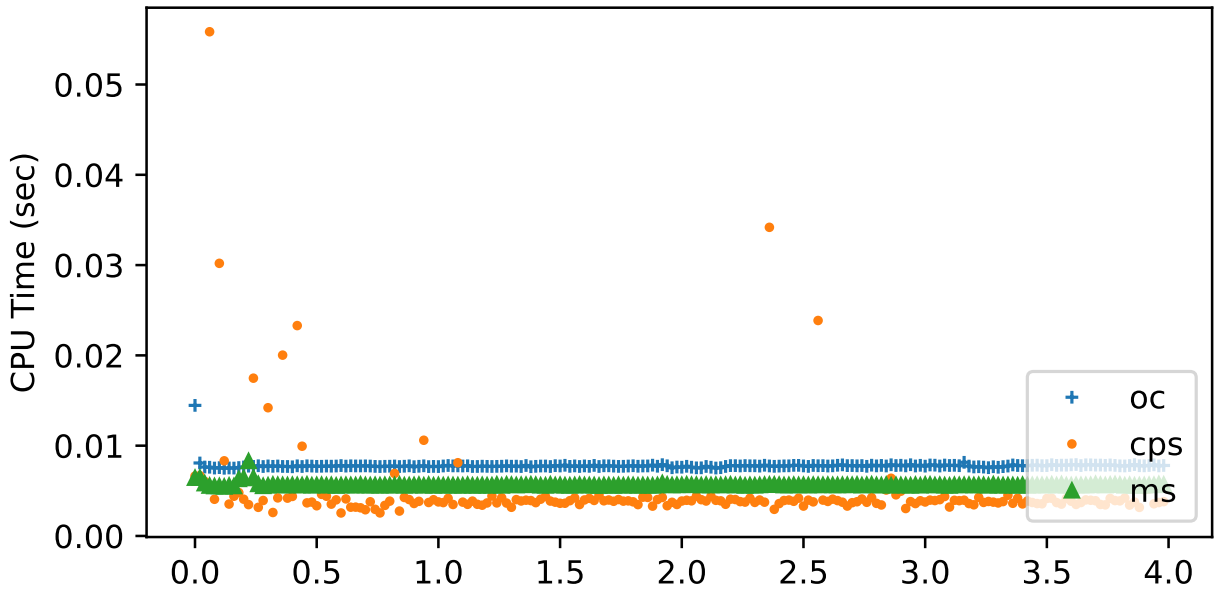
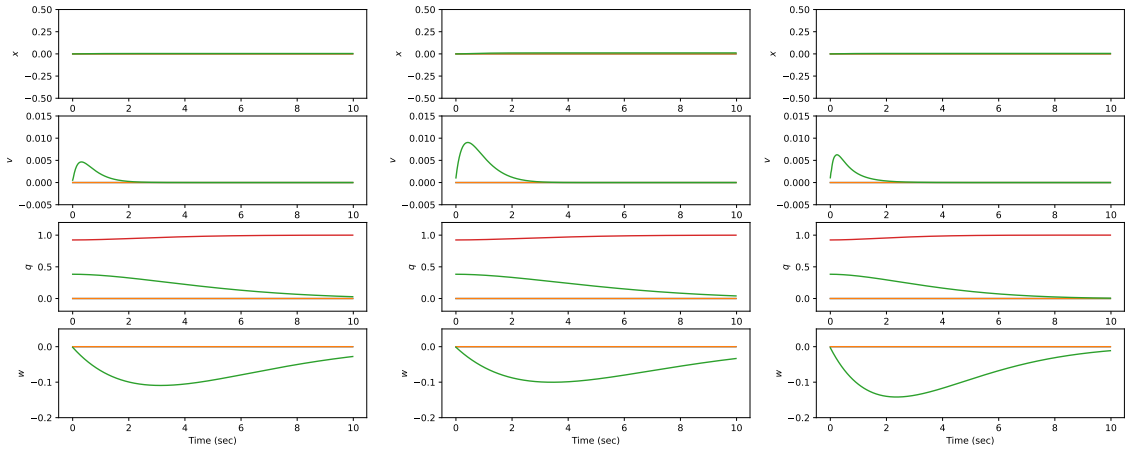
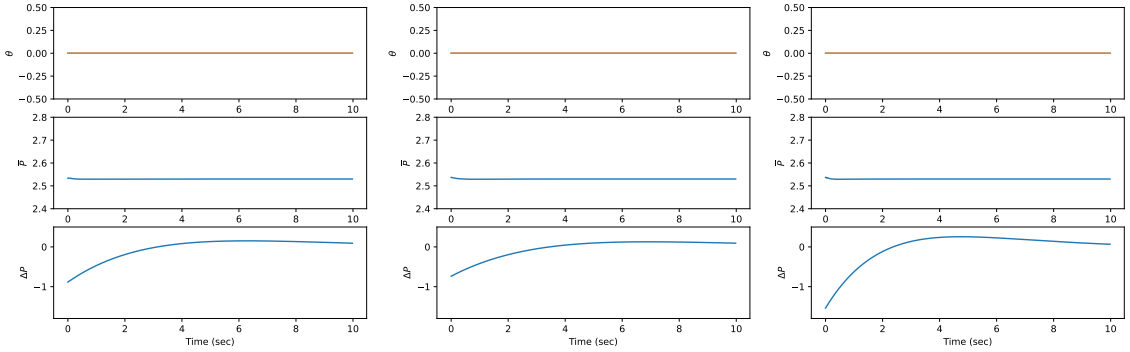


Figure 7: CPU time for simulation with initial state equal to goal state.



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 8: State data on a simulation requiring a 45° rotation about the z axis.



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 9: Control data on a simulation requiring a 45° rotation about the z axis.

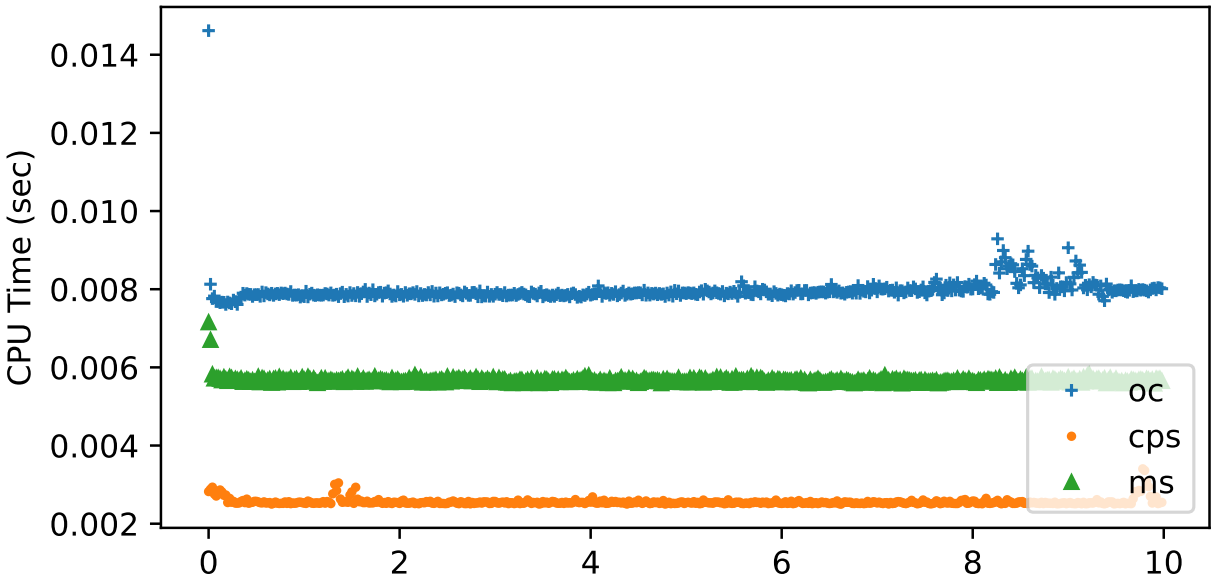
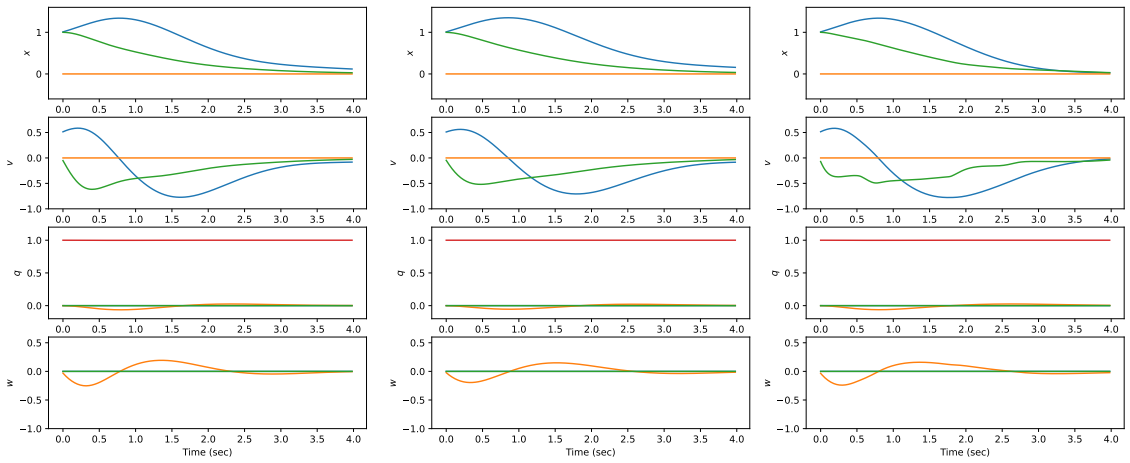
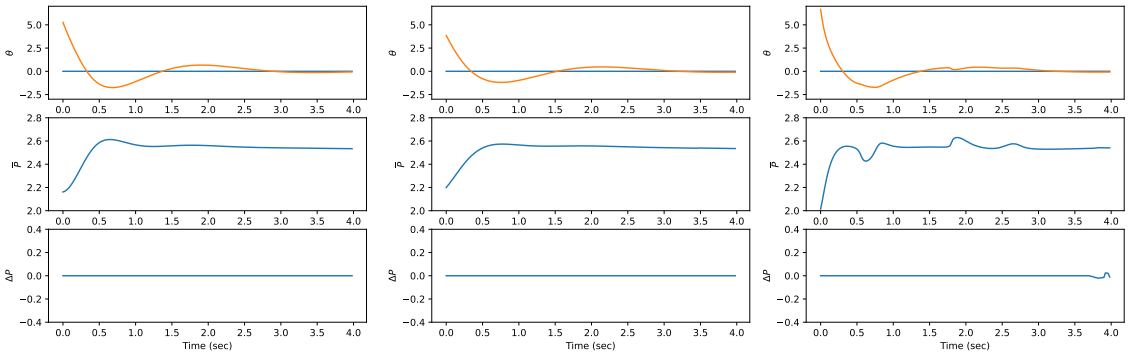


Figure 10: CPU time for simulation requiring a 45° rotation about the z axis.



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 11: State data on a simulation with initial conditions: $x = 1$, $z = 1$, $v_x = 0.5$



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 12: Control data on a simulation with initial conditions: $x = 1$, $z = 1$, $v_x = 0.5$

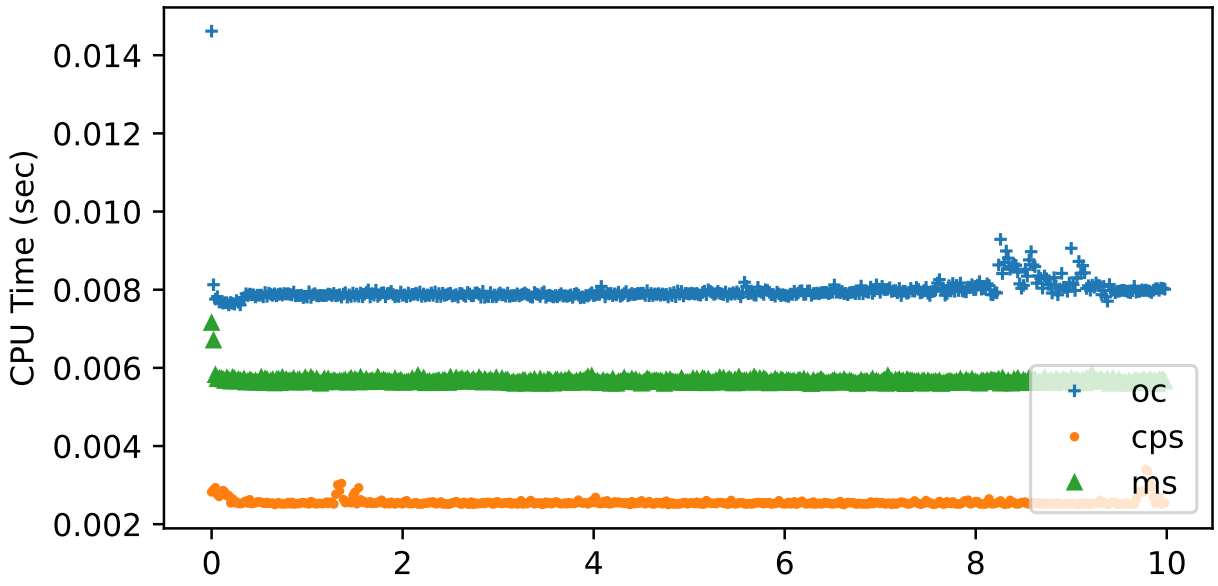
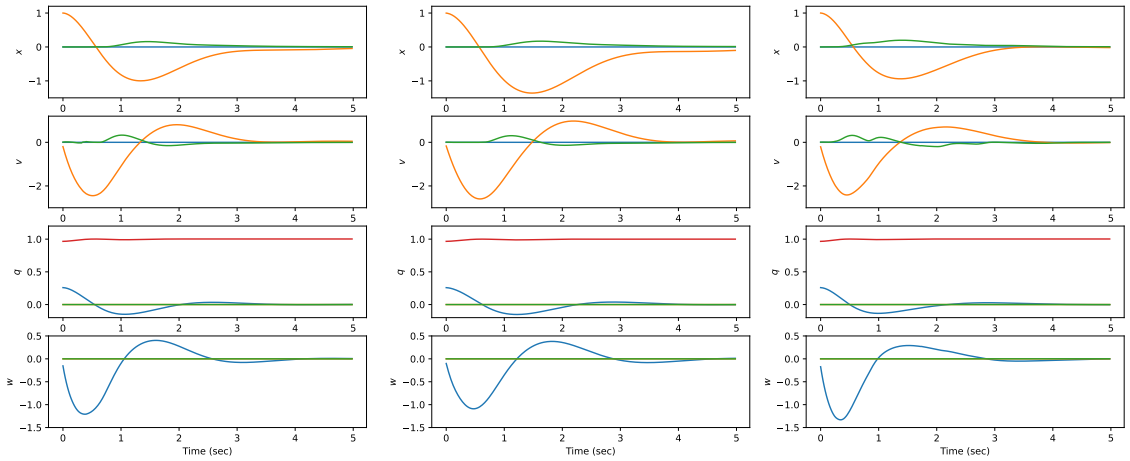
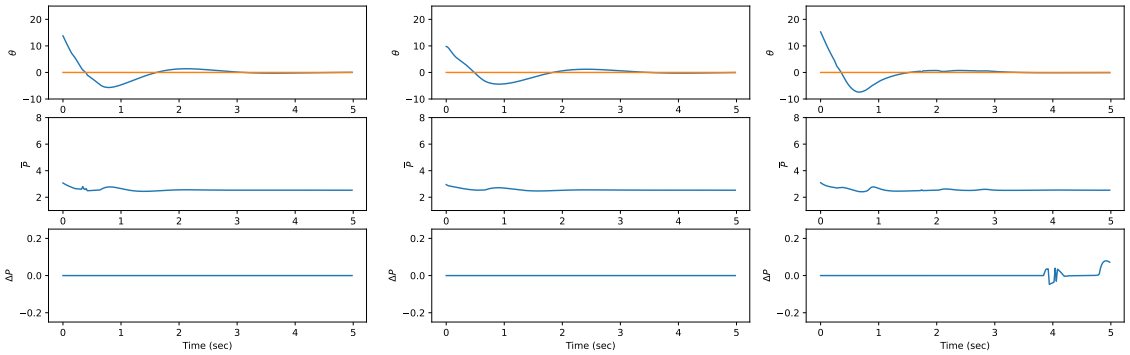


Figure 13: CPU time for a simulation with initial conditions: $x = 1$, $z = 1$, $v_x = 0.5$



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 14: State data on a simulation with initial conditions: $y = 1{,}15°$ rotation about x axis.



(a) multiple shooter (b) orthogonal collocation (c) Chebyshev pseudospectral

Figure 15: Control data on a simulation with initial conditions: $y = 1{,}15°$ rotation about x axis.

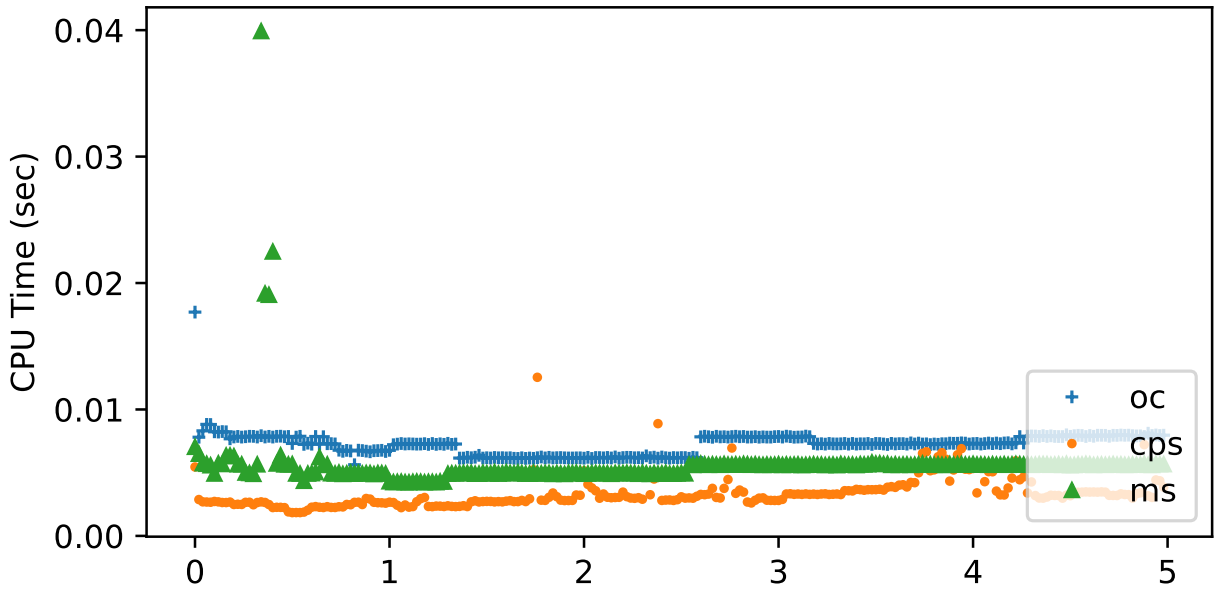


Figure 16: CPU time for simulation with initial conditions: $y = 1{,}15°$ rotation about x axis.

All of the methods provide good quality solutions with state values that behaved smoothly and converged to the desired goal state. The control graphs produced by Chebyshev's pseudospectral method were somewhat irregular and produced chatter when it attempted to hover. These experiments were run on a Macbook Pro with a 3.49GHz CPU while the Raspberry Pi 5 has a CPU speed of 2.4GHz with 8GB of RAM. However, in practice we find the Macbook Pro is two times faster than the Pi. A 50Hz implementation on the Pi requires a 0.02 second solution on the Pi and a 0.01 solution on the Macbook Pro. All the methods met this requirement. Table (3) summarizes the average CPU time for solution of the NLP problem for each simulation and each method.

initial state	MS	OC	CPS
hover	0.006	0.008	0.005
45° rotation about z axis	0.006	0.008	0.003
$x = 1$, $z = 1$, $v_x = 0.5$	0.005	0.006	0.003
$y = 1{,}15°$ rotation about x axis	0.005	0.007	0.003

Table 3: Comparison of average CPU time in seconds for solution of NLP for a series of simulations on the Raspberry Pi 5. Methods include multiple shooter (MS), orthogonal collocation (OC), and Chebyshev pseudospectral method (CPS).

Conclusions

We implemented three different formulations of the NPL problem for an NMPC algorithm controlling a thrust vector drone and compared their solution quality and relative efficiency. All methods produced good quality solutions that were efficient enough to control a thrust vector drone using NMPC on a Raspberry Pi 5. Chebyshev's pseudospectral method was slightly faster but produced graphs that were at times irregular. All of these methods are good candidates for control of our thrust vector drone, but further real-world

experiments are required to determine if the solutions produced here will provide good control in practice.

References

Raphaël Linsen, Petr Listov, Albéric de Lajarte, Roland Schwan, and Colin N. Jones. Optimal thrust vector control of an electric small-scale rocket prototype. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 1996–2002, 2022. doi: 10.1109/ICRA46639.2022.9811938.

Lukas Spannagl, Elias Hampp, Andrea Carron, Jerome Sieber, Carlo Pascucci, Aldo Zgraggen, Alexander Domahidi, and Melanie Zeilinger. Design, optimal guidance and control of a low-cost re-usable electric model rocket, 03 2021.

Felix Fiedler, Benjamin Karg, Lukas Lüken, Dean Brandner, Moritz Heinlein, Felix Brabender, and Sergio Lucia. do-mpc: Towards fair nonlinear and robust model predictive control. *Control Engineering Practice*, 140:105676, 2023. ISSN 0967-0661. doi: https://doi.org/10.1016/j.conengprac.2023.105676. URL `https://www.sciencedirect.com/science/article/pii/S0967066123002459`.

Carlos E. Garcia, David M. Prett, and Manfred Morari. Model predictive control: Theory and practice - a survey. *Autom.*, 25:335–348, 1989. URL `https://api.semanticscholar.org/CorpusID:3443742`.

Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 2018.

Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, March 2006. ISSN 0025-5610.

HSL. A collection of fortran codes for large scale scientific computation., 2013. URL `http://www.hsl.rl.ac.uk`.

Lorenz T. Biegler. *Nonlinear Programming*. Society for Industrial and Applied Mathematics, 2010. doi: 10.1137/1.9780898719383. URL `https://epubs.siam.org/doi/abs/10.1137/1.9780898719383`.

Fariba Fahroo and I. Ross. Direct trajectory optimization by a chebyshev pseudospectral method. volume 25, pages 3860 – 3864 vol.6, 02 2000. ISBN 0-7803-5519-9. doi: 10.1109/ACC.2000.876945.