

CS 5757: Optimization Methods for Robotics
Homework 1: Robotics Fundamentals

Due: Thursday, Feb. 12, 2026, at 11:59pm on Gradescope, Total Points: 50

Learning goals for this problem set:

- **Problem 1:** Gain insight into linearizing and discretizing continuous-time dynamics.
- **Problem 2:** Familiarization with posing robotics problems as optimizations, polynomial splines, and solving systems of linear equations.
- **Problem 3:** Learning how to implement numerical integrators on manifolds, and using proper rotation differences in control.

Submission Instructions: Submit your writeup as a PDF to the *Written Assignment* on Gradescope. For the *Programming Assignment*, submit *only* your completed `solutions.py`.





You are **strongly encouraged** to typeset your homework submissions using L^AT_EX (if you do not know L^AT_EX yet, now is as good a time as any to learn it – you will need it in grad school anyway). If you opt for handwritten submissions, make sure that they are legible – if we cannot read it, we cannot grade it!

1.1 Dynamics and Linearization

Consider a damped pendulum with dynamics:

$$m\ell^2\ddot{\theta} = -mg\ell \sin \theta - b\dot{\theta} + u,$$

where θ is the angle from vertical, $b > 0$ is the damping coefficient, and u is an input torque.

- (a)  (1 pts) Define the state $\mathbf{x} = [\theta \quad \dot{\theta}]^T$. Write the dynamics in the form $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, u)$.
- (b)  (2 pts) For simplicity, let $m, g, \ell = 1, b = 2$. Linearize the dynamics around $\mathbf{x}^* = [0 \quad 0]^T$, $u^* = 0$ (hanging down). Find \mathbf{A} and \mathbf{B} such that $\dot{\mathbf{x}} \approx \mathbf{A}\mathbf{x} + \mathbf{B}u$. Is this equilibrium stable?
- (c)  (2 pts) Repeat for $\mathbf{x}^* = [\pi \quad 0]^T$, $u^* = 0$ (inverted). Is this equilibrium stable?
- (d)  (2 pts) In class, we discussed *forward* Euler integration, which approximates the next state using the slope at the current time:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \mathbf{A}\mathbf{x}_k.$$

An alternative is the *backward* Euler method, an *implicit* integrator that instead evaluates the dynamics at the *next* time step:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \mathbf{A}\mathbf{x}_{k+1}.$$

Note that \mathbf{x}_{k+1} appears on both sides of the equation.

For each method, determine the range of step sizes $\Delta t > 0$ for which the discrete-time system remains stable (assuming $(\mathbf{I} - \Delta t \mathbf{A})$ is invertible). How do they compare? Finally, explain

briefly why backward Euler can be computationally expensive for general (nonlinear) systems.

Hint: If a matrix \mathbf{M} has eigenvalues λ_i , then the matrix $\alpha\mathbf{I} + \beta\mathbf{M}$ has eigenvalues $\alpha + \beta\lambda_i$, and the matrix $(\alpha\mathbf{I} + \beta\mathbf{M})^{-1}$ (assuming it exists) has eigenvalues $\frac{1}{\alpha + \beta\lambda_i}$.

Solution:

(a) We can write the dynamics in state-space form by rearranging the equations of motion:

$$\begin{aligned}\dot{\mathbf{x}} &= \frac{d}{dt} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}, \\ &= \begin{bmatrix} \dot{\theta} \\ -\frac{g}{\ell} \sin \theta - \frac{b}{m\ell^2} \dot{\theta} + \frac{1}{m\ell^2} u \end{bmatrix}, \\ &= \begin{bmatrix} 0 & 0 \\ -\frac{g}{\ell} & 0 \end{bmatrix} \sin x + \begin{bmatrix} 0 & 1 \\ 0 & -\frac{b}{m\ell^2} \end{bmatrix} x + \begin{bmatrix} 0 & \frac{1}{m\ell^2} \end{bmatrix} u, \\ &\equiv \mathbf{f}(\mathbf{x}, u).\end{aligned}$$

(b) To linearize the system, we take the Jacobian of the dynamics \mathbf{f} with respect to each of its arguments. The linearized system matrix is given by the Jacobian with respect to \mathbf{x} ,

$$\begin{aligned}\mathbf{A} &= \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x}^*, u^*), \\ &= \begin{bmatrix} 0 & 1 \\ -\frac{g}{\ell} \cos \theta & -\frac{b}{m\ell^2} \end{bmatrix} \bigg|_{(\mathbf{x}^*, u^*)}, \\ &= \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix},\end{aligned}$$

for the parameter values given and $\mathbf{x}^* = [0 \ 0]^T$. Similarly, the linearized input matrix \mathbf{B} is the Jacobian with respect to u ,

$$\mathbf{B} = \begin{bmatrix} 0 \\ \frac{1}{m\ell^2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad (1)$$

for the parameter values given.

This equilibrium (\mathbf{x}^*, u^*) is stable if and only if the eigenvalues of \mathbf{A} have negative real parts. These eigenvalues are the roots of the equation:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = \det \left(\begin{bmatrix} -\lambda & 1 \\ -1 & -2 - \lambda \end{bmatrix} \right) = \lambda(2 + \lambda) + 1 = 0.$$

This yields the characteristic polynomial $\lambda^2 + 2\lambda + 1 = 0$, which has roots

$$\lambda = \frac{-2 \pm \sqrt{4 - 4}}{2} = -1$$

meaning the matrix has repeated real roots at -1 . Thus it is stable.

(c) If we repeat for the “up” equilibrium, the analysis remains the same:

$$\mathbf{A} = \left[\begin{array}{cc} 0 & 1 \\ -\cos \theta & -2 \end{array} \right] \Big|_{(\mathbf{x}^*, u^*)} = \left[\begin{array}{cc} 0 & 1 \\ 1 & -2 \end{array} \right].$$

The eigenvalues of this matrix are given by

$$\det(\mathbf{A} - \lambda \mathbf{I}) = \det \left(\left[\begin{array}{cc} -\lambda & 1 \\ 1 & -2 - \lambda \end{array} \right] \right) = \lambda(2 + \lambda) - 1 = 0.$$

This characteristic polynomial is then $\lambda^2 + 2\lambda - 1$, which has roots

$$\lambda = \frac{-2 \pm \sqrt{4 + 4}}{2} = -1 \pm \sqrt{2}$$

So, this system matrix has two real eigenvalues, and since $\sqrt{2} > 1$, the positive branch of this equation produces a real, positive eigenvalue. Thus, this equilibrium is unstable.

(d) Recall that a discrete-time linear system $\mathbf{x}_{k+1} = \mathbf{A}_d \mathbf{x}_k$ is stable if and only if all eigenvalues of \mathbf{A}_d lie strictly inside the unit circle, i.e., $|\mu_i| < 1$.

Forward Euler: This method gives the discrete-time system

$$\mathbf{x}_{k+1} = (\mathbf{I} + \Delta t \mathbf{A}) \mathbf{x}_k \equiv \mathbf{A}_d \mathbf{x}_k.$$

Using the hint, if λ is an eigenvalue of \mathbf{A} , then $\mu = 1 + \Delta t \lambda$ is an eigenvalue of \mathbf{A}_d . Substituting our eigenvalues:

$$\mu = 1 - \Delta t$$

The squared magnitude is

$$\begin{aligned} |\mu|^2 &= (1 - \Delta t)^2, \\ &= \Delta t^2 - 2\Delta t + 1. \end{aligned}$$

For stability, we require $|\mu|^2 < 1$:

$$\Delta t^2 - 2\Delta t = \Delta t(\Delta t - 2) < 0$$

Since $\Delta t > 0$, this requires $\Delta t < 2$. Forward Euler is stable only for $\boxed{\Delta t < 2}$.

Backward Euler: Rearranging the implicit equation to solve for \mathbf{x}_{k+1} :

$$\begin{aligned} \mathbf{x}_{k+1} - \Delta t \mathbf{A} \mathbf{x}_{k+1} &= \mathbf{x}_k, \\ (\mathbf{I} - \Delta t \mathbf{A}) \mathbf{x}_{k+1} &= \mathbf{x}_k, \\ \mathbf{x}_{k+1} &= (\mathbf{I} - \Delta t \mathbf{A})^{-1} \mathbf{x}_k \equiv \mathbf{A}_d \mathbf{x}_k. \end{aligned}$$

Using the hint, if λ is an eigenvalue of \mathbf{A} , then $\mu = \frac{1}{1 - \Delta t \lambda}$ is an eigenvalue of \mathbf{A}_d . Substituting

$\lambda = -1$ into this expression yields

$$\mu = \frac{1}{1 + \Delta t},$$

which is less than one for any $\Delta t > 0$. Backward Euler is *unconditionally stable*—it remains stable for any positive step size.

Backward Euler’s numerical stability is a significant advantage over forward, especially for “stiff” systems where forward Euler would require prohibitively small time steps. When using backward Euler, the origin retains its stability no matter the timestep used to integrate the system.

The hard part of using this integrator for more complex systems is that it requires us to solve a system of nonlinear equations to find our next state \mathbf{x}_{k+1} . Specifically, our next state is only implicitly defined as the root of the equation

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \mathbf{f}(\mathbf{x}_{k+1}, \mathbf{u}_{k+1}),$$

which requires us to solve a system of equations each time we simulate a discrete-time step. Solving systems of nonlinear equations can be quite difficult in general, which means we often resort to explicit numerical integrators which are easier to implement.



1.2 Spline-based trajectory generation

You are planning a trajectory for a drone to pass through two gates at positions $\mathbf{g}_1, \mathbf{g}_2 \in \mathbb{R}^3$ at times t_1, t_2 , with an initial position \mathbf{x}_{init} at time t_0 . Since quadrotors are *differentially flat* in position [2], they can track arbitrary smooth trajectories $\mathbf{p}(t) = (x(t), y(t), z(t))$ in 3D. So we’ll solve the 1D problem below, then use `jax.vmap` to plan all three dimensions.


We represent the trajectory as a cubic spline with two segments:

$$x_i(\tau) = a_i + b_i\tau + c_i\tau^2 + d_i\tau^3, \quad \tau \in [0, \Delta t_i],$$

for $i = 0, 1$, where $\tau = t - t_i$ is the local time and $\Delta t_i = t_{i+1} - t_i$. Define the coefficient vector $\mathbf{c} = [a_0 \ b_0 \ c_0 \ d_0 \ a_1 \ b_1 \ c_1 \ d_1]^T \in \mathbb{R}^8$.


- (a)  (3 pts) Write $x_0(\tau)$, $\dot{x}_0(\tau)$, and $\ddot{x}_0(\tau)$ as inner products $\phi(\tau)^T \mathbf{c}_0$, $\dot{\phi}(\tau)^T \mathbf{c}_0$, $\ddot{\phi}(\tau)^T \mathbf{c}_0$ where $\mathbf{c}_0 = [a_0 \ b_0 \ c_0 \ d_0]^T$. Give expressions for $\phi(\tau)$, $\dot{\phi}(\tau)$, $\ddot{\phi}(\tau)$.
- (b)  (2 pts) The trajectory must satisfy:
- Initial condition $x_0(0) = x_{\text{init}}$
 - Waypoint: $x_i(\Delta t_i) = g_{i+1}$ for $i = 0, 1$
 - Continuity: $x_0(\Delta t_0) = x_1(0)$
 - Smoothness: $\dot{x}_0(\Delta t_0) = \dot{x}_1(0)$
 - Boundary: $\dot{x}_0(0) = 0$ and $\dot{x}_1(\Delta t_1) = 0$

Each constraint can be written as $\mathbf{a}^T \mathbf{c} = b$ for some $\mathbf{a} \in \mathbb{R}^8$. Using your answer from (a), write out (\mathbf{a}, b) for the waypoint constraint on segment 0 and the smoothness constraint.


- (c)  (3 pts) In `solutions.py`, implement `build_constraints(x0, gates, dt)` which returns

(**A**, **b**) encoding all seven constraints. Here **x0**, shape (1,) gives the initial position, **gates**, shape (2,), contains $[g_0, g_1]$, **dt**, shape (2,), contains $[\Delta t_0, \Delta t_1]$, and the outputs have shapes (7, 8) and (7,).

Hint: You may want to implement helper functions for calculating $\phi(\tau)$, $\dot{\phi}(\tau)$, $\ddot{\phi}(\tau)$. Make sure any helper functions are included in **solutions.py**.

- (d)  (5 pts) In **solutions.py**, implement **evaluate_spline_1d(coeffs, t, dt)** which evaluates the spline at a given time **t**. Here **coeffs**, shape (8,) are the spline coefficients, **t**, shape () is the query time, and **dt**, shape (2,) contains $[\Delta t_0, \Delta t_1]$. Your function should evaluate which segment of the spline you are in (assuming $t \in [0, \Delta t_0 + \Delta t_1]$), and return the corresponding segment evaluated at its corresponding value of τ .


Note: For full credit, your implementation must be compatible with **jax.vmap**, meaning that it cannot use native Python **if** statements or **while** loops (since these are not compatible with the “traced” datatypes used by **vmap**). We recommend you check out our short guide here on how to write **jit**- and **vmap**-compatible functions in JAX.

- (e)  (6 pts) We minimize integrated squared acceleration $J(\mathbf{c}) = \mathbf{c}^T \mathbf{Q} \mathbf{c}$ subject to $\mathbf{A} \mathbf{c} = \mathbf{b}$. We provide **build_Q(dt)**. Implement **solve_spline_1d(x0, gates, dt)** that solves:

$$\begin{bmatrix} 2\mathbf{Q} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{c}^* \\ \boldsymbol{\lambda}^* \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b} \end{bmatrix}$$

and returns \mathbf{c}^* , shape (8,).

Note: For full credit, your implementation must be compatible with **jax.vmap**.

- (f)  (2 pts) The notebook **drone_racing.py** implements a small, interactive Viser visualization of your spline-based planner which uses **jax.vmap** to plan a 3D path ($\mathbf{x}(t)$, $\mathbf{y}(t)$, $\mathbf{z}(t)$) from an initial condition through two gates which you can move around in the visualizer.

In your writeup, discuss some limitations of our approach. Are the trajectories your planner produces guaranteed to be feasible for the drone? What other constraints might you need to add for a real-world racing drone?

Solution:

- (a) A very nice property of polynomial splines is that they (and their derivatives) can be expressed as linear functions of polynomial basis functions $\phi(t)$.

For the first segment, we can simply write

$$\begin{aligned} x_0(t) &= \begin{bmatrix} 1 & \tau & \tau^2 & \tau^3 \end{bmatrix} \begin{bmatrix} a_0 \\ b_0 \\ c_0 \\ d_0 \end{bmatrix}, \\ &\equiv \phi(\tau)^T \mathbf{c}_0, \end{aligned}$$

where the basis function $\phi(\tau) = \begin{bmatrix} 1 & \tau & \tau^2 & \tau^3 \end{bmatrix}^T$. The time derivatives of x_i can just be

expressed using time derivatives of $\phi(\tau)$,

$$\begin{aligned}\dot{x}_0(\tau) &= \dot{\phi}(\tau)^T \mathbf{c}_0 = [0 \quad 1 \quad 2\tau \quad 3\tau^2]^T \mathbf{c}_0, \\ \ddot{x}_0(\tau) &= \ddot{\phi}(\tau)^T \mathbf{c}_0 = [0 \quad 0 \quad 2 \quad 6\tau]^T \mathbf{c}_0.\end{aligned}$$

(b) An implication of this linearity is that constraints on the value of $x_i(\tau)$ or its derivatives are also linear in the spline parameters \mathbf{c}_i . We derive each constraint below:

- Initial condition: since $x_0(\tau) = \phi(\tau)^T \mathbf{c}_0$, the initial condition constraint can be expressed:

$$x_0(0) = [\phi(0)^T \quad \mathbf{0}^T] \mathbf{c} = x_{\text{init}}.$$

$$\text{So } \mathbf{a} = \begin{bmatrix} \phi(0) \\ \mathbf{0} \end{bmatrix}, b = x_{\text{init}}.$$

- Waypoints: Similarly, the waypoint constraint for segment 0, $x_0(\Delta t_0) = g_1$, can be expressed as a linear function of \mathbf{c} , we can write

$$x_0(\Delta t_0) = [\phi(\Delta t_0)^T \quad \mathbf{0}^T] \mathbf{c} = g_1.$$

$$\text{So } \mathbf{a} = \begin{bmatrix} \phi(\Delta t_0) \\ \mathbf{0} \end{bmatrix}, b = g_1.$$

- Continuity: This constraint can be rewritten $x_0(\Delta t_0) - x_1(0) = 0$, which can be expressed as:

$$x_0(\Delta t_0) - x_1(0) = [\phi(\Delta t_0)^T \quad \mathbf{0}^T] \mathbf{c} - [\mathbf{0}^T \quad \phi(0)^T] \mathbf{c} = 0,$$

$$\text{which simplifies to } \mathbf{a}^T \mathbf{c} = b \text{ for } \mathbf{a} = \begin{bmatrix} \phi(\Delta t_0) \\ -\phi(0) \end{bmatrix}, b = 0.$$

- Smoothness: Following similar reasoning, we can write this constraint as $\mathbf{a}^T \mathbf{c} = b$ for $\mathbf{a} = \begin{bmatrix} \dot{\phi}(\Delta t_0) \\ -\dot{\phi}(0) \end{bmatrix}, b = 0$
- Boundary Conditions: Finally, we can write the velocity boundary conditions as:

$$\begin{aligned}\dot{x}_0(0) &= [\dot{\phi}(0)^T \quad \mathbf{0}^T] \mathbf{c} = 0, \\ \dot{x}_1(\Delta t_1) &= [\mathbf{0}^T \quad \dot{\phi}(\Delta t_1)^T] \mathbf{c} = 0\end{aligned}$$

Note that full credit for this question requires only providing the waypoint and smoothness conditions; we write all constraints here for completeness.

- (f) This setup does not always produce dynamically feasible trajectories for the drone - since the drone has limited actuation, it cannot arbitrarily track trajectories for any values of Δt_i . Moreover, a typical constraint for drones is collision avoidance - as formulated, this spline-based method does not let us encode any sort of collision costs or constraints.

1.3 Drone dynamics and control on $SO(3)$

Now we will implement a simulation of our drone and a controller to track the paths we plan.

The drone's state consists of a pose (world-frame position $\mathbf{p}^W \in \mathbb{R}^3$ and orientation $\mathbf{R} \in SO(3)$) and a twist (world-frame linear velocity $\dot{\mathbf{p}}^W$ and body-frame angular velocity $\boldsymbol{\omega}^B$). The control inputs are a scalar thrust $f \in \mathbb{R}$ along the body z -axis and a body-frame torque $\boldsymbol{\tau}^B \in \mathbb{R}^3$.

The equations of motion are:

$$\dot{\mathbf{R}} = \mathbf{R}(\boldsymbol{\omega}^B)^\wedge, \quad (2)$$


$$m\ddot{\mathbf{p}}^W = -g\mathbf{e}_3 + (\mathbf{R}\mathbf{e}_3)f, \quad (3)$$

$$\mathcal{J}\dot{\boldsymbol{\omega}}^B = -(\boldsymbol{\omega}^B)^\wedge \mathcal{J}\boldsymbol{\omega}^B + \boldsymbol{\tau}^B, \quad (4)$$


where $\mathbf{e}_3 = [0 \ 0 \ 1]^T$ and the “wedge” operator maps vectors in \mathbb{R}^3 to their skew-symmetric representation,

$$\mathbf{v}^\wedge = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix},$$

for $\mathbf{v} \in \mathbb{R}^3 = [v_1 \ v_2 \ v_3]^T$.


- (a)  (3 pts) In `solutions.py`, implement `f_drone` which computes the state derivative. We provide Python constants in the module, `M_DRONE`, a scalar, representing the drone's mass m and `INERTIA`, an (invertible) array shape `texttt(3,3)` representing the drone's inertia matrix \mathcal{J} , which you should use in your solution.

The state is represented as `state = (p, R, dp, w)` where `p`, shape `(3,)`, is position, `R`, shape `(3,3)`, is orientation, `dp`, shape `(3,)`, is velocity, and `w`, shape `(3,)`, is angular velocity. The function should return `dstate = (dp, dR, ddp, dw)` where `dR`, shape `(3,)`, is the tangent vector $\boldsymbol{\omega}^B$ (not a matrix), and `ddp`, `dw`, shape `(3,)`, are the accelerations from (3)-(4).

- (b)  (8 pts) In `solutions.py`, implement `rk4_step` which performs one step of RK4 integration. The update is:

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k), \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{x}_k \oplus \frac{\Delta t}{2}\mathbf{k}_1, \mathbf{u}_k), \\ \mathbf{k}_3 &= \mathbf{f}(\mathbf{x}_k \oplus \frac{\Delta t}{2}\mathbf{k}_2, \mathbf{u}_k), \\ \mathbf{k}_4 &= \mathbf{f}(\mathbf{x}_k \oplus \Delta t \mathbf{k}_3, \mathbf{u}_k), \\ \mathbf{x}_{k+1} &= \mathbf{x}_k \oplus \frac{\Delta t}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \end{aligned}$$

where \oplus adds tangent vectors to the state. For the rotation component, use `jaxlie.SO3.exp()` to compute $\mathbf{R}' = \mathbf{R}\exp(\delta\boldsymbol{\omega}^\wedge)$. The function takes `state`, `u` shape `(4,)`, and `dt`, and returns `next_state`.

- (c)  (5 pts) We provide `compute_outer_loop(state, p_des, dp_des, ddp_des)` which computes the thrust magnitude f and desired orientation \mathbf{R}_d from position tracking error. Your task is to implement `compute_attitude_control(R, R_d, w)` which computes body-frame torques. The attitude error is the tangent vector from identity to $\mathbf{R}_d^T \mathbf{R}$:


$$\mathbf{e}_R = \log(\mathbf{R}_d^T \mathbf{R})^\vee.$$

Use `jaxlie.S03.from_matrix(...).log()` to compute this. The torque command is:

$$\boldsymbol{\tau}^B = -k_R \mathbf{e}_R - k_\omega \boldsymbol{\omega}^B + \boldsymbol{\omega}^B \times \mathcal{J} \boldsymbol{\omega}^B.$$

The function takes `R`, shape `(3,3)`, `R_d`, shape `(3,3)`, and `w`, shape `(3,)`, and returns `tau`, shape `(3,)`.

In `solutions.py`, complete `compute_control` by implementing the attitude error and torque computation. The function takes `state`, `R_d` shape `(3,3)`, `f` a scalar, and `gains` with keys `kR`, `komega`, and returns `u`, shape `(4,)`.

- (d)  (3 pts) In `drone_racing.ipynb`, run the simulation of your drone tracking a trajectory from Problem 2.

In your writeup describe the results of a few experiments:


- Try increasing the sim timestep `sim_dt`. At some point, the drone begins to behave erratically. What could explain this? Does it happen consistently for all values of k_R , k_ω ?
- Try playing with the Δt_i values in `delta_t`. How do these affect tracking?

Solution:

- (d) When we increase `sim_dt` past a certain point, we see the drone destabilize and crash. One explanation of this could be numerical error in the dynamics (which could make sense for very large values). However, even small values cause the drone to crash. Another explanation is that increasing `sim_dt` causes the drone's controller to run more slowly (since we hold the controls constant over the simulation interval) which causes it to not be able to respond quickly to disturbances or command changes.

If we decrease Δt_i too much, the tracking becomes quite poor - this is because the time segments are too short for the drone to reasonably track with its current gains. This motivates ideas like time rescaling [1] which first plan geometric paths, and then decide on the time intervals Δt_i which are feasible for the quadrotor to track. Dynamic feasibility is one reason we solve full trajectory optimizations instead of just doing geometric path planning.

1.4 AI Usage Survey

-  (2 pts) Please describe how you used AI (if at all) while completing this homework. Note that you will not be penalized for AI usage, regardless of how extensively you used it, as long as you acknowledge it. We are generally curious about how we can improve the course material and better help with your learning!

References

- [1] Philipp Foehn, Angel Romero, and Davide Scaramuzza. Time-optimal planning for quadrotor waypoint flight. *Science Robotics*, 6(56):eabh1221, 2021.
- [2] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011.