Drone modeling, dynamics, and control

Outline

These are some notes from Dr. Guanya Shi's aerial mobility lectures at CMU. Learn more about him at gshi.me.

1. Modeling aerial vehicles

Let x be our robot's state and u be our control input. We describe our robot's change in state as a function of its current state and some input:

$$\dot{x} = \frac{dx}{dt} = f(x, u) \tag{1}$$

Modeling concerns f, how f is found, and how our input u influences it.

1.1. The rotor lift model

Let's consider the motion of a single rotor, where:

- T is thrust (in Newtons)
- τ is torque
- ρ is air density

- n is rotation speed
- D is diameter

Under the **rotor lift model**:

$$
\begin{aligned}
T &= k_T \rho n^2 D^4 \\
\tau &= k_\tau \rho n^2 D^5
\end{aligned}
\tag{2}
$$

In practice, engineers typically use a **thrust to torque ratio**, $k_{T2\tau}$, and simplify (2) as:

$$
\begin{aligned}
T &\propto n^2 \\
\tau &= k_{T2\tau} T
\end{aligned}
\tag{3}
$$

This ratio is typically very small, such that its much harder to control torque than it is to control thrust.

1.2. Modeling a 2D (planar) drone

Let:

- $p \in \mathbb{R}^2$ be position in the world frame
- $v \in \mathbb{R}^2$ be velocity in the world frame
- $\theta \in \mathcal{S}^1$ be the angle
- $\omega \in \mathbb{R}$ be the angular rate
- $T_1, T_2 \in \mathbb{R}$ be the thrust per rotor
- $\vec{g} = [0; -g] \in \mathbb{R}^2$ be the gravity vector
- $m, j, d \in \mathbb{R}$ be mass, inertia, and arm length

Any motion model contains a system of four equations: the translational kinematics and dynamics, and the rotational kinematics and dynamics. As a reminder, *dynamics* refers to physical forces and moments.

We can model a 2D drone as:

$$
\begin{aligned}
\dot{p} &= v \\
m\dot{v} &= m\vec{g} + (T_1 + T_2) \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} \\
\dot{\theta} &= \omega \\
J\dot{\omega} &= (T_2 - T_1)d
\end{aligned}
\tag{4}
$$

Let's make some simplifications. First, we define total thrust $T = T_1 + T_2$ and torque $\tau = (T_2 - T_1)d$.

Let's then define an **actuation matrix**, which combines total thrust and torque into a single equation:

$$
\begin{bmatrix} T \\ \tau \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -d & d \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \end{bmatrix}
\tag{5}
$$

Finally, we define state and input as:

$$
\begin{aligned}
x &= [p; v; \theta; \omega] \in \mathbb{R}^6 \\
u &= [T; \tau] \in \mathbb{R}^2
\end{aligned}
\tag{6}
$$

These additional formulas allow us to simplify (4) as:

$$
\begin{aligned}
\dot{p} &= v \\
m\dot{v} &= m\vec{g} + T \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} \\
\dot{\theta} &= \omega \\
J\dot{\omega} &= \tau
\end{aligned}
\tag{7}
$$

1.3. Modeling a quadrotor

Our 3D model will operate in two coordinate frames:

- $\{A\}$, the right-handed world frame (also called the inertial frame)
- $\{B\}$, the right-handed body frame

1.3.1. Orientation

b_1, b_2, b_3 are the axes of $\{B\}$ w.r.t. the world frame $\{A\}$. The the orientation of our quadrotor is given by the rotation matrix $R = [b_1 b_2 b_3] \in SO(3)$. (The appendix briefly explains what $SO(3)$ means.)

We can also model orientation as a quaternion or in Euler parameterization.

1.3.2. Actuation matrix

Recall that an actuation matrix is a combined representation of the total thrust and torque of a drone, as a function of the individual thrusts of its rotors. In 2D this was a 2×2, but in 3D it's a 4×4 matrix:

$$
\begin{bmatrix} T \\ \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & d & 0 & -d \\ -d & 0 & d & 0 \\ k & -k & k & -k \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix}
\tag{8}
$$

where $k = k_{T2\tau}$, the thrust-to-torque ratio specific to each drone.

1.3.3. Kinematics and dynamics

$$
\begin{aligned}
\dot{p} &= v \\
m\dot{v} &= m\vec{g} + Re_3T \\
\dot{R} &= RS(\omega) \\
J\dot{\omega} &= J\omega \times \omega + \tau
\end{aligned}
\tag{9}
$$

where $S(\omega)$ is a special matrix that maps a vector to its skew-symmetric matrix form:

$$
S(\omega) = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}
\tag{10}
$$

2. Setpoint and trajectory tracking controllers

2.1. The goal

Recall that modeling concerns the function $\dot{x} = f(x, u)$, the change in our robot's state given its current state and some input. Specifically, modeling studies how our state responds to some input u.

By contrast, **control** is about designing some feedback controller or **policy** $\pi(x)$ that leads to some desired trajectory or setpoint $x_d(t)$.

2.2. The trajectory tracking controller

It's common for drone controllers to have what we call "double integrator dynamics." That means that our input $u = \ddot{x}$. Our tracking controller takes the form:

$$u = -K_p(x - x_d) - K_d(\dot{x} - \dot{x}_d) + \ddot{x}_d \tag{11}$$

where $\ddot{x}_d$ is our **feedforward term**. If our goal is setpoint tracking, then $\dot{x}, \ddot{x} = 0$.

What's the point of the feedforward term, anyway? Well, our tracking error is $e = x - x_d$. Then we can rework (11) as:

$$\begin{aligned} \ddot{x} &= -K_P e - K_D \dot{e} + \ddot{x}_d \\ \ddot{e} + K_P e + K_D \dot{e} &= 0 \end{aligned} \tag{12}$$

The above only works if we add the feedforward term. The intuition is that this term allows us to predict and brace for future error states.

2.3. Cascaded trajectory tracking

In this approach, the nonlinear trajectory tracking task is separated into two linear controllers.

The first calculates a desired force vector that would align the drone with the target trajectory. The second calculates the desired rotor torques that would align the drone with the force vector.

This separation leads to two sets of gains. In the typical case where PD controllers are used, this means a K_P, K_D for the force controller and $K_{P,\tau}, K_{D,\tau}$ for the torque controller.

Here's some code for cascaded control:

```
def cascaded_control(self, p_d, yaw_d):
    # Helpful stuff
    e3 = np.array([0., 0., 1.])
    z = self.R @ e3

    # position control
    K_P = 35.0
    K_D = 6.0

    # attitude control
    K_Ptau = 200.0
    K_Dtau = 40.0
```

```
        f_d = -np.array([0., 0., -self.g]) - K_P * (self.p - p_d) - K_D * (self.v - v_d) + a_d
        T = (f_d.T @ z) * self.m
        z_d = f_d/norm(f_d)
        # R_d is the combination of z_d and yaw_d
        R_d = z_d + yaw_d
        n = np.cross(e3, z_d)
        rho = np.arcsin(norm(n))
        if norm(n) == 0: # Check for divide by zero
            R_EB = np.eye(3)
        else:
            R_EB = Rotation.from_rotvec(rho * (n/norm(n))).as_matrix()
        R_AE = Rotation.from_euler('z', yaw_d).as_matrix()
        R_d = R_AE @ R_EB
        R_e = R_d.T @ self.R

        # Skew symetric form
        R_ess = R_e-R_e.T
        # Vectorized
        R_ev = np.array([R_ess[2,1], R_ess[0,2], R_ess[1,0]])
        alpha = -K_Ptau * R_ev - K_Dtau * self.omega

        # Map alpha to tau
        Ji = self.J_inv
        J = self.J
        A = Ji @ np.cross(J@self.omega, self.omega)
        tau = J @ (alpha - A)
        Ttau = np.concatenate(([T], tau))

        # Map [T, tau] to individual thrusts
         # u = [T1, T2, T3, T4], one per rotor
        u = self.B_inv @ Ttau

        return u
```

And here are (some of) the equations used.

$$
\begin{aligned}
f_d &= -\vec{g} - K_p(p - p_d) - K_d(v - v_d) + a_d \\
T &= m(f_d \cdot z) \\
z_d &= \frac{f_d}{||f_d||_2} \\
R_d &= z_d + \psi_d \\
\rho &= \arcsin(||e_3 \times z_d||_2) \\
R_{EB} &= \text{matrix}(\rho \frac{n}{||n||_2}) \\
R_{AE} &= \text{matrix}(\psi_d) \\
R_d &= R_{AE} \cdot R_{EB} \\
R_e &= R_d^T \cdot R
\end{aligned}
\tag{13}
$$

2.4. Linear systems and LQR

Given some linear state space model $\dot{x} = Ax + Bu$, where A is the dynamics matrix and B is the control matrix, we can design a policy $u = -Kx$, where K is our gain matrix.

Our closed-loop system then becomes:

$$\dot{x} = (A - BK)x = A_{CL}x \tag{14}$$

Our system is exponentially stable if A_{CL} is a Hurwitz matrix: all eigenvalues must have strictly negative real parts.

We can determine an optimal K using LQR, pole placement, or related techniques.

3. Trajectory generation and optimization

3.1. Overview

We've studied methods by which drones can follow given trajectories. But where do these trajectories come from? How can we generate reasonable trajectories that link a drone's current pose to some goal?

Let's divide a drone's tasks into three sections. First, **modeling** gives our drone a sense of how its state will change over time, especially given some input u. After that, **control** leverages our model to move the drone toward a goal pose, or along some trajectory. Finally, **planning** concern how our desired poses or trajectories are generated.

Given two poses, there are an infinite number of trajectories that connect them. How can we narrow our options down? We consider **feasibility** and **optimality** (cost).

Let's express trajectory optimization mathematically, with the assumption that our time window is *fixed*. We won't consider time-optimal control here.

$$\begin{gathered} \min_{x,u} \sum_{t=1}^{T} c_t(x_t, u_t) \\ \text{such that:} \\ x_{t+1} = f_t(x_t, u_t) \\ x_t \in X_T \\ u_t \in U_t \end{gathered} \tag{15}$$

3.2. Differential flatness

The goal of differential flatness is to relate (x, u) to a *flat output*. This allows us to transform nonlinear systems into linear controllable ones. The idea was expressed in this paper by Fliess, Lévine, Martin, and Rouchon in 1992.

3.2.1. Wheel on flat surface

As an example, consider a wheel of radius r on a flat surface, where p is the wheel's position, ω is its angular velocity, and τ is the torque applied to it. Then $\dot{p} = \omega r$ and $\dot{\omega} = \tau$, and we can express position p as a *flat output*, where:

$$\begin{aligned} p &= p \\ \omega &= \dot{p}/r \\ \tau &= \ddot{p}/r \end{aligned} \tag{16}$$

In this scenario, we can easily generate smooth trajectories $p(t)$.

3.2.2. The 2D case

Recall that in the 2D drone scenario, $x = [p, v, \theta, \omega]$ and $u = [T, \tau]$. Let's define the drone's *body axes* as:

$$\begin{aligned} \vec{x} &= [\cos\theta, \sin\theta] \\ \vec{y} &= [-\sin\theta, \cos\theta] \end{aligned} \tag{17}$$

Let jerk be $j = \dot{a} = \dddot{p}$ and let snap be $s = \ddddot{p}$ (yes, four dots!).

Our goal: build a 1-1 mapping between (a, js) and $(\theta, \omega, T, \tau)$, which would make p a flat output:

$$(p, v, a, j, s) \leftrightarrow (p, v, \theta, \omega, T, \tau) \tag{18}$$

Steps:

1. $\vec{y} = \frac{a - \vec{g}}{||a - \vec{g}||}, \theta = -\arctan\vec{y}, T = (a - \vec{g})^T \vec{y}$
2. $\omega = -(j^T \vec{x})/T$
3. $\tau = -(s^T \vec{x} + 2j^T \vec{y}\omega)/T$

This means that, lacking any bounds on our drone's rotors, *we can track any smooth trajectory* using only (p, v, a, j, s)!

3.2.3. Adding differential flatness to cascaded control

Using the equations derived above, our cascaded control equations become:

$$\begin{aligned} f_d &= -\vec{g} - K_p(p - p_r) - K_D(v - v_r) - K_I \int_t (p - p_r) + a_r \\ \theta_d &= -\arctan\left(\frac{f_{d,x}}{f_{d,y}}\right) \\ T &= f_d^\intercal [-\sin\theta, \cos\theta]^T \\ \tau &= -K_{P,\tau}(\theta - \theta_d) - K_{D,\tau}(\omega - \omega_d) - K_{I,\tau} \int_t (\theta - \theta_d) + \tau_d \\ \omega_d &= -\frac{j_r^\intercal \vec{x}}{T}, \tau_d = \frac{s_r^\intercal \vec{x} + 2j_r^\intercal \vec{y}\omega}{T} \end{aligned} \tag{19}$$

3.3. Trajectory planning with polynomials

Now that we can track any smooth trajectory within a bounded (a, j, s), let's use polynomials to link waypoints. For simplicity we'll consider the case of a 2D drone such that state $x = [p_x, p_y, v_x, v_y, \theta, \omega]$.

Our goal is to generate a polynomial $p(t)$ that links two waypoints A, B within a final time T under the following constraints:

$$\begin{aligned} p(0) &= p_A \\ p(T) &= p_B \\ p'(0) &= p'(T) = 0 \\ p''(0) &= p'(T) = 0 \\ p'''(0) &= p'''(T) = 0 \\ p'''' &= 0 \end{aligned} \tag{20}$$

Since we have nine constraints, we'll need to design an 8th-order polynomial of the form:

$$\begin{aligned} p(t) &= a_8t^8 + a_7t^7 + a_6t^6 + a_5t^5 + a_4t^4 + a_3t^3 + a_2t^2 + a_1t + a_0 \\ p'(t) &= 8a_8t^7 + 7a_7t^6 + 6a_6t^5 + 5a_5t^4 + 4a_4t^3 + 3a_3t^2 + 2a_2t + a_1 \\ p''(t) &= 56a_8t^6 + 42a_7t^5 + 30a_6t^4 + 20a_5t^3 + 12a_4t^2 + 6a_3t + 2a_2 \\ p'''(t) &= 336a_8t^5 + 210a_7t^4 + 120a_6t^3 + 60a_5t^2 + 24a_4t + 6a_3 \\ p''''(t) &= 1680a_8t^4 + 840a_7t^3 + 360a_6t^2 + 120a_5t + 24a_4 \end{aligned} \tag{21}$$

In this 2D case, we can separate the trajectory into two equations, one for x and one for y.

As an example, suppose we wish to move the drone from position $(0, 0)$ to $(1, 0)$. Referencing (21), we can write x's constraints in matrix form as:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ T^8 & T^7 & T^6 & T^5 & T^4 & T^3 & T^2 & T & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 8T^7 & 7T^6 & 6T^5 & 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 56T^6 & 42T^5 & 30T^4 & 20T^3 & 12T^2 & 6T & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 336T^5 & 210T^4 & 120T^3 & 60T^2 & 24T & 6 & 0 & 0 & 0 \\ 1680^4 & 840T^3 & 360T^2 & 120T & 24 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_8 \\ a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{22}$$

y's constraints are identical except that the rightmost matrix is all zeros ($y_{des} = 0$).

Once we have both systems in matrix form, then we can take the inverse of the leftmost matrix to find the polynomials' coefficients.

Appendix

Wait, what's $SO(3)$?

The group of all possible rotations in 3D is called the **special orthogonal group**, denoted as $SO(3)$. This is the set of all 3×3 real matrices such that:

1. $R^T R = I$ and
2. $\det R = 1$.

Similarly, the set of all 2D rotation matrices is called $SO(2)$.

Recall that in linear algebra, a group is a set of elements, where the product of any two elements is just another element in the group, and the inverse of any element is also in the group.

In other words, given *any* two matrices $R, T \in SO(3)$, then $R^{-1} \in SO(3)$ and $RT \in SO(3)$.