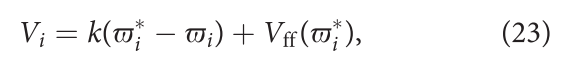
Control Documentation

Currently, the drone follows the control methodology described in Mahony, Kumar, and Corke, “Multirotor Aerial Vehicles: Modeling, Estimation, and Control of Quadrotor” in *IEEE Robotics & Automation Magazine*, 2023, with some adjustments. First, see Figure 5.

A diagram of a machine

Description automatically generated

We currently have a feedforward motor controller that accounts for the current battery voltage, but we have not implemented feedback at this level. Mahony’s suggested formula is



Where *w\** is the desired rotor speed, *w* is the actual speed, *V\_ff* is the feedforward voltage calculated from steady state tests, k is the proportional feedback gain, and *V* is the commanded voltage. We don’t know how to get the actual speed measurement, w. It may be possible through the ESC somehow using electromotive force (EMF) sensing, but we never figured it out. So we have *k*=0. The Vff term was calculated from steady state tests (see Modeling Documentation).

The desired speed, *w\**, of each motor is determined by the desired torques and lift. Mohony’s formula is

A diagram of a mathematical equation

Description automatically generated with medium confidence

Where *T\_Sigma* is the lift, *tau\_i* are the roll, pitch, and yaw torques, *c\_T* is the thrust coefficient of the rotor, *d* is the length of the arm (distance between center of rotor and center of mass), and *c\_Q* is the drag coefficient of the rotor. These parameters are also calculated from steady state tests (see Modeling Documentation). **Due to differing conventions between our model and Mohony’s model, the sign is flipped on the third row of *Gamma*.** To achieve a constant hover in place, one would set *T\_Sigma = mg* (where *m* is the mass of the drone and *g* is gravity), and *tau\_i = 0*, then invert *Gamma* to find the corresponding rotor speeds.

Correcting for errors in the feedforward lift and torque is explained in the sections “Attitude Control” and “Trajectory Control” in Mahony, but the result is somewhat obfuscated, so it’s explained differently here. The end result is essentially the same. Proportional-derivative (PD) control is used to adjust the lift and torques separately, as

Of the equations above, the first four implement PD control, where the *K*’s are chosen gains, is roll, is pitch, and is yaw. The remainder work together to determine the appropriate setpoints. The yaw and *z* set points are straightforward. The command value is given directly, and the difference is calculated. For now, we also assume that all the rate setpoints are 0, though this could be adjusted in the last six equations. For the rest, we have control over roll and pitch, but we really want to dictate *x* and *y* position. Therefore, the commanded *x* and *y* positions and rates are converted into commanded roll and pitch angles using PD control transformed to account for the yaw. The idea here is that if we want to make a small, slow adjustment in *x* and/or *y*, we’ll need a small roll and/or pitch angle, but if we want to move far and fast, we’ll need a steep pitch angle. The roll and pitch offset values account for the IMU not being perfectly level with respect to the plane of the drone. This is all implemented in Feedback.py’s PD function. The PD parameters are tuned by hand heuristically.

Once the desired rotor speeds are calculated, they are converted to pulsewidth in the Controller.py function Speed2PW. According to

Then, since the pulsewidth has maximum value 1900 and minimum value 1100, the Controller.py function RectifyControl enforces PW = max(1100,min(1900,PW)). The pulsewidth formula comes from the equilibrium force balance of the motor and rotor electro-mechanical dynamics, and the parameters are estimated from experiments (see Modeling Documentation).

For measurements, the x,y,z rates are estimated as a first order difference in the Controller.py function EstimateRates. Also, the yaw measurement is restricted between 0 and 2pi. To avoid jumps, the Controller.py function RectifyYaw adds or subtracts 2pi as necessary to eliminate these jumps. A more elegant solution would be to use quaternions instead of roll, pitch, and yaw, but that would greatly complicate things. I also wrote a function in Controller.py called FilterSignal, which implements a discrete first order filter of the raw data to remove noise. It has the form

,

Where is the filter state, is the measured state (for example, roll, pitch, y, z, etc), is the filtered estimation of that state, is the time step, is the time increment, and and are the tunable filter parameters. I stopped using this function when I thought it was the reason the LQR controller wasn’t working. The PD works ok without it, but I haven’t tried the PD controller with the filter. It might improve performance by reducing noise, or it might increase latency and make it harder to stabilize.

There’s also an LQR function in Controller.py that was a model-based design, but that model-based design does not yet work. Presumably, the model is not accurate enough.