

## DRONE PID CONTROL

### INTRODUCTION TUTORIAL

At its simplest, an autonomous drone is one that moves in all autonomy. It would have to plan a trajectory, for everything leading up to the creation of a trajectory, and then follow the plan. **This project is about Trajectory Following.**

Is it really that hard to follow a trajectory?

First, a word about Trajectory Following. In this tutorial, we are just given a sequence of points, waypoints for the drone, so it just needs to move from one point to the next. It's easy to make a drone move from a to b if all it takes is a very precise motion forward. As long as the drone is +/- 5cm from the target point, we're satisfied.



But now what if we want to build up speed? There is a number of things that can go wrong. Perhaps the inertia of the drone slows it down, because the motor thrust is calculated on an estimation of the drone's inertia and not the actual inertia (a model error). If it slows down unexpectedly it won't end up very precisely on the target. Another problem: after tuning the drone to hover straight, it still has a slight rotation as it moves forward (another model error). Over 10m, the drone might arc away even when we ask it to move straight.

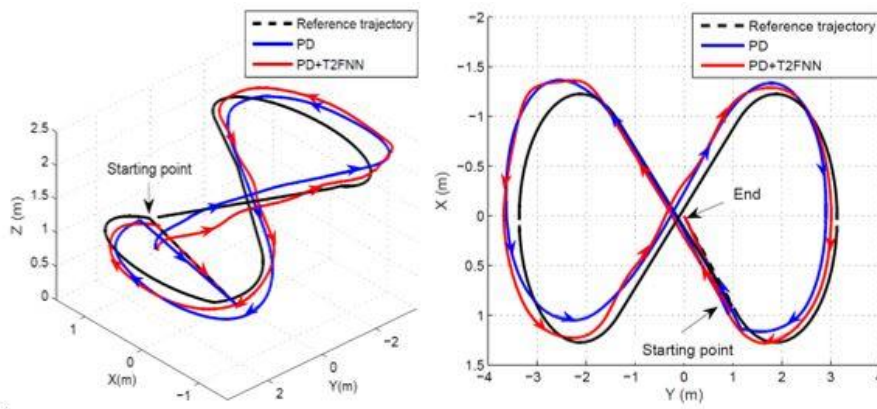
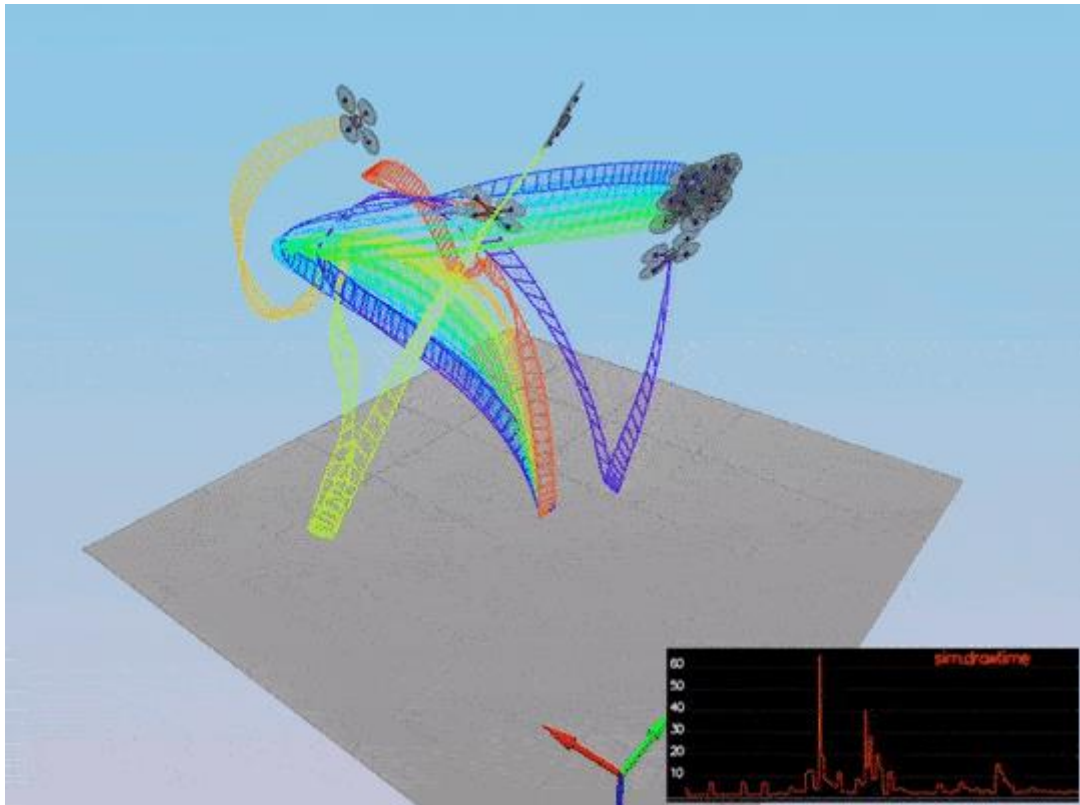


Figure 1: using different control methods to follow a trajectory

It makes sense that these are hiccups between our virtual model and the real world. We can **eliminate** many of these hiccups with Control Theory. In this case, negative feedback loops. The basic principle here is that the drone can observe its motion and grade specific information against the required motion. For example: if the camera tilts to the right, it knows to apply a yaw velocity of  $\gamma$  rad/s to the left. If the motion is slow enough, all it takes is moving the same amount as the difference between the target point and where on the camera we expect the target point to be. Basing ourselves on this **position error** and transforming it to the required thrust gives us a **proportional controller**. You can imagine a slow drone moving from a to b in this way.

### What is wrong with just using position error?

However once again, we want to build up speed. We want to avoid obstacles. We want to see the drone falling to the ground and saving itself at the last possible moment. If we just use position control to affect the motor thrusts, the drone might actually adjust itself too slowly to its target. Scaling up the P controller will make things move faster. But we hit a limitation of position: either we go too fast or too slowly toward the target – leading to oscillations or slow settling time. Accumulate this slight error over all degrees of freedom, and the drone actually spins out of control. Sometimes, a sweet spot for the P controller doesn't even exist. [*drone spins out of control*]



PID Control doesn't just use position error, but also velocity error, and acceleration error (P, I and D). At its simplest, each next level helps smoothen out the previous level. PID control is explored on a wheeled robot in [this tutorial](#). Our implementation of PID control is on a drone: a different model entirely from wheeled robots.

## What's the big difference between controlling drones and wheeled robots?

It all depends on how many moving directions there are, vs what mechanism we use to move the robot. Let's look at the drone case:

- Moving directions: You can move up/down (1), left/right (2), forward/backwards (3). But you can also rotate in roll (4), pitch (5), yaw (6). Contrast this with a wheeled vehicle: forward/backwards (1), left/right (2 - most cars don't...), and no up/down. As well as rotating in roll(3), and sometimes pitch(4).
- Motion mechanism: for a quadcopter-type drone: thrust motion is achieved with 4 propellers attached to rotating motors. This means our baseline is rotational velocities  $\omega_1, \omega_2, \omega_3, \omega_4$ . And we need to associate these to forces (thrust) so that it links up to the drone motion in all 6 directions above. How do we do that? Mathematical equations.

### Equations of Movement

We assume a common factor of proportionality  $k$  and  $F = \sqrt{T}$   
(we will see that such an assumption is not a problem!):

$$\begin{aligned}\dot{\phi} &= k((\omega_1 + \omega_4) - (\omega_2 + \omega_3)) = k\omega_1 - k\omega_2 - k\omega_3 + k\omega_4 \\ \dot{\theta} &= k((\omega_1 + \omega_2) - (\omega_3 + \omega_4)) = k\omega_1 + k\omega_2 - k\omega_3 - k\omega_4 \\ \dot{\psi} &= k((\omega_1 + \omega_3) - (\omega_2 + \omega_4)) = k\omega_1 - k\omega_2 + k\omega_3 - k\omega_4 \\ F &= k((\omega_1 + \omega_2 + \omega_3 + \omega_4)) = k\omega_1 + k\omega_2 + k\omega_3 + k\omega_4\end{aligned}$$


or, using matrices:

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ F \end{pmatrix} = \begin{pmatrix} k & -k & -k & k \\ k & k & -k & -k \\ k & -k & k & -k \\ k & k & k & k \end{pmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{pmatrix}$$

Now we hit a problem: only 4 inputs, and 6 outputs that we want to control. With a wheeled vehicle, things are different: 4 inputs, 4 outputs. This leads to a well-known problem in science: this system is **statistically indeterminate**. Equations, normally used to relate  $n$  inputs to  $n+m$  outputs, now rely heavily on limited inputs for simultaneous outputs.

There is actually a way to do this for 6 axes of freedom. But certain variables have to become **dependent** on one another. Why does this make it more difficult? Now some axis of freedom become very fragile.

Phase



**EQUATIONS OF MOTION**

$$\ddot{z} = g - \frac{u_1}{m} \cdot \cos(\phi)$$

$$\ddot{y} = \frac{u_1}{m} \cdot \sin(\phi)$$

$$\ddot{\phi} = \frac{u_2}{I_{xx}}$$

- $m = 0.2 \text{ kg}$
- $I_{xx} = 0.1 \text{ units}$

In this case, our yaw depends on the 7<sup>th</sup> derivative of our position. Small differences in yaw control will eventually lead to overcompensation and undercompensation:

[yaw control issue]

Less precise yaw tuning begins to affect the rest of the system.

Can't we avoid this issue by rearranging the equations?

Exactly, welcome to the project!

Since there's no perfect PID controller, we build up a controller iteratively using different scenarios. It can take into account:

- The nature of the trajectory
- The number of axes we choose to control
- The completion speed
- The hardware limitations of the mechanisms used (e.g. maximum thrust of motors vs inertia of the drone). This has a huge effect too.
- Unaccounted variations in the physical model parameters (shift of center of gravity, of drone mass...)
- We might know time-dependant boundaries for the drone, in case a drone will collide with another drone. See it as a new trajectory that changes with any one element in the previous information.



PID Controller tuning now becomes a series of tests for the best sweet spot given the variability of our conditions. It all depends what criteria we wish to reach. Note: there are other types of controllers out there that can be more useful for different cases.

## Where do I start?

This introduction has expanded upon drone control for the precise purpose of following a trajectory. I tried to simplify everything to their root causes, but I skipped some steps. If you leave a like, I will make tutorials that teach this from scratch.

- How do I create my own control equations? Theory tutorial
- Now I have a controller, how hard is it really to tune it?

A set of scenarios that help build up the control architecture iteratively

- How do I code my C++ to actually programming this logic?

Techniques to optimise drone control algorithms

- What simulator can I use that will help me iterate this controller architecture?

Realtime feedback to connects to the simulator in real-time during C++ code development

If you leave a like, I might explore these topics in further tutorials.

## **Going further with this concept**

- How do I make my own trajectories?
- How do I compute trajectories that keep my drones from colliding into one another?
- How do I do this whole procedure with a real drone?