Feedback Control

Alexander Koldy

Forest Hills Robotics League

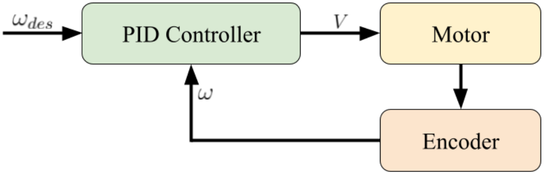
alexanderkoldy.ak@gmail.com

September 18, 2024

Introduction

In this lecture, we will discuss feedback control in the context of the *FIRST Tech Challenge* (FTC).

Feedback control is different from feedforward control because we use information from our sensors to our controller, rather than simply predicting what input we need to send to our system.



PID control

Let's go back to our motor velocity example from the feedforward lecture. One assumption we made is that our feedforward input produced the desired velocity/acceleration that we fed into the calculation. Let's now use an encoder to measure the motor's velocity. This will give us the feedback information that we will need to run a PID controller.

PID control - proportional

The "P" portion of a PID controller is the **proportional** term. This term is proportional to the error between our current state and our desired state. A **P-controller**, i.e., a PID controller where the "I" and "D" terms are set to zero, acts like a spring. When a spring is stretched significantly, it will quickly shoot back towards its equilibrium, eventually slowing down and settling at it. It may even oscillate about it's equilibrium a bit first. Using physics, we describe the spring's response with

$$F_s = kx$$

where F_s is the spring force k is the spring constant and x is it's distance for it's equilibrium. The P-controller takes a similar form:

$$u_p = k_p e$$

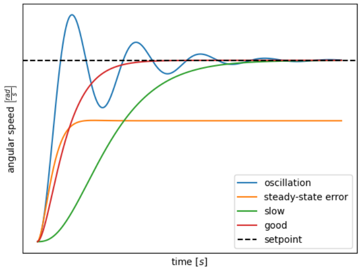
where u_p is an input to our system, k_p is a tune-able constant and e is the difference between our desired state and current state behaviors. For a motor velocity controller, the P-controller may look like:

$$u_p = k_p(\omega_{des} - \omega)$$

PID control - proportional

The figure below showcases some of the potential motor responses you may see depending on how well we have tuned the k_p value. We call the **steady-state** the area of curve where the system seems to have settled around constant value throughout time.

- **Oscillation** typically occurs with a k_p value that is **too high**
- **Steady-state error**, i.e., the system never settling near the setpoint, will typically occur with a k_p value that is **too low** or when the system experiences resistances from external factors such as friction
- **Slow** responses typically suffer from a k_p value that is **too low**.



Note: each system will have its own k_p values which are considered "good"!

PID control - integral

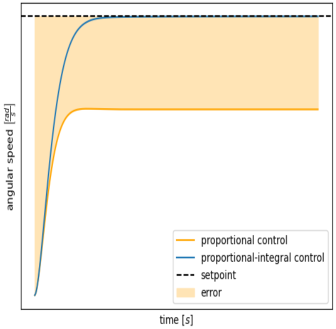
Sometimes, tuning a simple P-controller is not enough as steady-state error persists. The "I" portion of a PID controller is the **integral** term and it is responsible for minimizing steady-state error. The integral term continuously sums the error in the system in an attempt to push it towards the setpoint.

The input contribution from the integral term looks like:

$$u_i = k_i \int_0^{t_k} e(t)dt$$

where k_i is a new tune-able constant and the integral expression represents the sum of the error times time. Typically in code we simplify this expression to

$$u_i = k_i \sum_0^k e_k \Delta t, \quad \Delta t = t_k - t_{k-1}$$



Note: we can choose to ignore Δt and simply sum the error. We would simply just adjust our k_i tuning accordingly.

PID control - integral

The full **PI-controller** expression looks as follows (in the general sense):

$$u_{pi} = u_p + u_i = k_p e + k_i \int_0^{t_k} e(t)dt$$

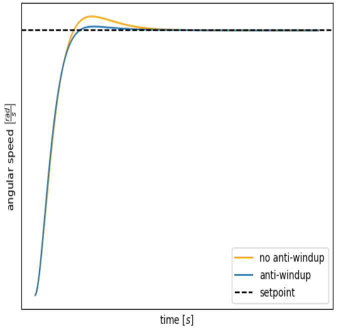
In code, we will typically keep a running error sum variable that gets updated every loop iteration.

One of the issues with this simple PI-controller implementation is that the sum is never reset. Thus, the constant sum accumulation may result in u values that cause the system to overshoot the setpoint. If there is an overshoot, the system will take longer to stabilize, since the error sum is still positive and will need time to be reduced back to zero (it will start reducing once the setpoint is overshot, since the error will at that point be negative). This is called **integral windup** and is usually mitigated through the following:

- Reset the running sum once the setpoint is crossed (crossing the setpoint can be easily checked if the sign of the error at the previous loop iteration is different from the sign of the error at the current loop iteration, i.e., $\text{sign}(e_{k-1}) \neq \text{sign}(e_k)$).
- Cap the running sum with a tune-able maximum sum so that it cannot blow up. Ensure negatives are handled the same way.

PID control - integral

The figure below showcases the results of a PI-controller using **anti-windup**. We notice that the overshoot is reduced and that it settles quicker in relation to the controller not using anti-windup!



In general, using an integral term in FTC is *not recommended*. This is due to the prominent use of feedforward terms which typically handle many cases that an integral term would be useful for.

PID control - derivative

So, we have discussed reducing steady-state error in the context of PID. However, we have another issue to tackle - oscillation! The “D” term handles this for us using the rate of change (i.e., derivative) of the error:

$$u_d = k_d \dot{e}$$

Let’s think about this intuitively. If our setpoint is greater than our current state, our error is positive, but will decrease at each loop iteration since the P portion of our controller is trying to reduce error. This will give our error rate $\dot{e}$ a negative sign. Likewise, if our setpoint is less than our current state, our error is negative, but will increase at each loop iteration since the P portion of our controller is trying to make the error “less negative”. This gives our error rate a positive sign. Typically, the sign of $\dot{e}$ being opposite the sign of e will cause a damping effect to our response. We can think of this as “removing energy” from the system. This removal causes oscillation effects to reduce. The **PD-controller** look as follows:

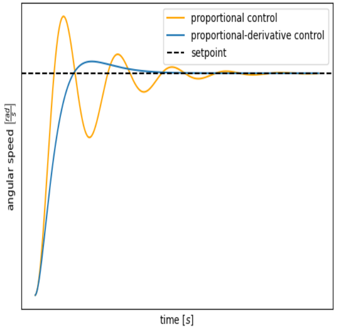
$$u_{pd} = u_p + u_d = k_p e + k_d \dot{e}$$

In code, we approximate $\dot{e}$ as

$$\dot{e} \approx \frac{e_k - e_{k-1}}{\Delta t}, \quad \Delta t = t_k - t_{k-1}$$

PID control - derivative

In the figure below, we notice that oscillations are reduced once we implement a derivative portion to our controller.



One potential issue with using the derivative is our approximation method. Approximating the derivative may lead to a noisy derivative signal we can call the **dirty derivative**. To overcome this, we can apply a low-pass filter to the derivative signal in an attempt to smooth it out!

PID control

For the sake of completion the full expression for a **PID-controller** is

$$u = u_p + u_i + u_d = k_p e + k_i \int_0^{t_k} e(t)dt + k_d \dot{e}$$

This is a general mathematical representation of the PID-controller. An actual implementation will have the approximations and improvements mentioned throughout the slides.

We can also construct a **PIDF-controller** which incorporates some feedforward (hence, the “F” term).

Tuning

There are many tuning methods for PID controllers and it is often system dependent. A general guide catered towards FTC can be found on this website. Starting will all tune-able constants at 0, following the process below should yield decent results.

1. Increase k_p until you eliminate as much steady-state error as possible
2. Increase k_i until your steady-state error is removed
3. Increase k_d until your oscillations are removed

Note: all tune-able constants should always be greater than or equal to zero. If you are using negative gains (constants), your implementation is wrong!

	Rise time	Overshoot	Settling Time	Steady-state error	Stability
k_p	Decrease	Increase	Minor change	Decrease	Degrade
k_i	Decrease	Increase	Increase	Eliminate	Degrade
k_d	Minor change	Decrease	Decrease	No effect*	Improve**

* in theory, ** if k_d is small