Understanding PID Control

And (Very Basic) Control Theory

Tyler Tian

February 12, 2021

FRC Team Arctos 6135

Table of Contents

Introduction

What is a Controller?

Suppose we want to move an arm to a specific position, or spin a motor to an exact RPM for our shooter. However, we can't directly control these things; we can only modify the percentage output. So how do we do it?

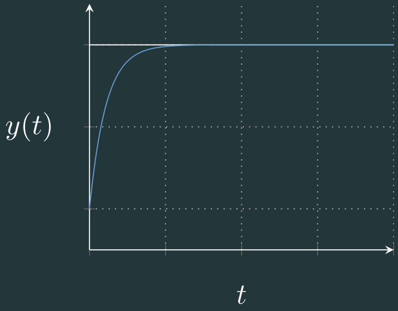
We can do this with a **controller**. In simple terms, a controller drives a variable (e.g. arm position) to a desired setpoint or reference point (e.g. the desired position).

Basic Concepts

- **Process Variable/Output** (PV) — The variable you're trying to control, e.g. arm position, shooter velocity, etc. A function of time commonly denoted $y(t)$.
- **Setpoint** (SP) or **Reference** — The desired value of the process variable. A function of time commonly denoted $r(t)$.
- **Control Output** — The output generated by the controller, e.g. percent output. A function of time commonly denoted $u(t)$.
- **Error** — The difference between the desired (setpoint) and actual values of the process variable. A variable of time commonly denoted $e(t) = r(t) - y(t)$.
- **Gain** — A constant scaling factor used to tune the controller's behaviour, usually determined through experimentation and denoted k.

Controllers Revisited

Ideal Controller Response



A controller attempts to drive the error to zero by generating a control output, based only on the setpoint (**Open-Loop Control**) or the setpoint and other variables (**Closed-Loop/Feedback Control**), often the value of the process variable.

Simple Open-Loop Controllers

For an open-loop controller, control output is *not* based on the current PV value. Consider controlling a motor's velocity with the following equation:

$$u(t) = k \cdot r(t)$$

We can choose our gain k to be $\frac{u_{max}}{v_{max}}$. However, the max velocity and curve shape are affected by many other factors (load, battery voltage, motor condition), making this an approximation at best.

Simple Closed-Loop Controllers (Bang-Bang Controllers)

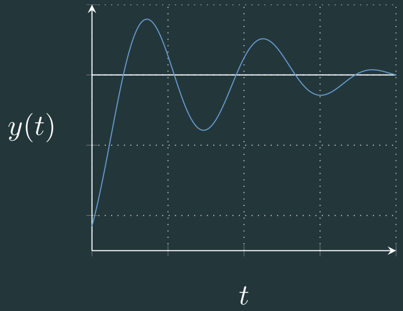
Consider controlling a robot's position with the simplest of feedback controller — if desired position is ahead, apply forward throttle; if it's behind, apply reverse throttle:

$$u(t) = \begin{cases} k & e(t) > 0 \\ 0 & e(t) = 0 \\ -k & e(t) < 0 \end{cases}$$

An appropriate gain k can be chosen by experimentation. It does the job, albeit not well. It will almost always overshoot and oscillate. For many simple systems it's good enough, but for driving in auto it's far from perfect.

Simple Closed-Loop Controllers (Bang-Bang Controllers)

Bang-Bang Controller Response



Notice the error quickly approaching zero, then overshooting and oscillating. This system takes a long time to stabilize and is not ideal.

Can We Do Better?

When a bang-bang controller doesn't do the job, we can use more complex controllers.

PID controllers are one such example. A properly tuned PID controller should reach the setpoint as fast as possible with almost no overshoot.

PID Basics

The PID Equation

If you search for ”PID controller” you might come across something like this:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t)$$

Sometimes there’s also a feedforward term:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t) + K_f f(t)$$

While correct, it looks daunting and not very intuitive. In reality, these terms come pretty intuitively, as you will soon see.

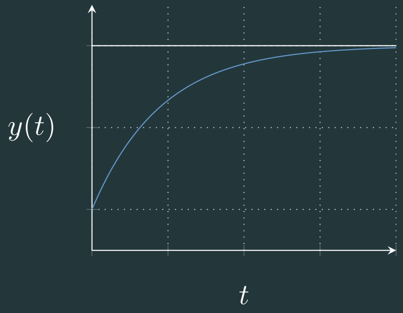
The Proportional Term

$$u(t) = \boxed{K_p e(t)} + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t)$$

The proportional term generates a control output *proportional to the current error*. It drives the error to zero and generates less output as the error decreases. Slowing down near the setpoint avoids overshooting.

P Controller Response

Theoretical P Controller Response



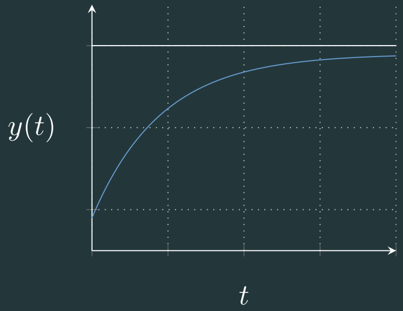
Assuming a constant setpoint and a linear relationship between $u(t)$ and $\frac{\mathrm{d}y}{\mathrm{d}t}$ (rate of change of PV), we have a differential equation:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = k(r - y(t))$$

If you know your functions well you'll recognize this as an exponential decay driving $y(t)$ to r or $e(t)$ to 0.

Steady-State Error

Actual P Controller Response



Since $u(t)$ approaches zero as $e(t)$ approaches zero, the error *asymptotes* at 0 (i.e. it approaches but never reaches 0).

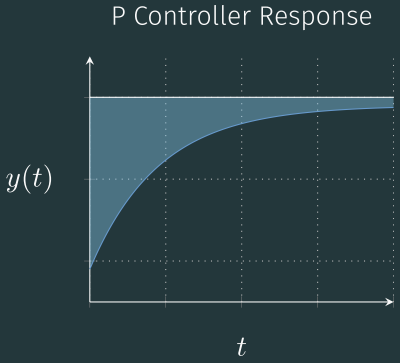
In a real system if $u(t)$ is too small $y(t)$ might not be affected at all. This creates a **steady-state error**, i.e. the system stabilizes before reaching the setpoint.

The Integral Term

$$u(t) = K_p e(t) + \boxed{K_i \int_0^t e(\tau)\,\mathrm{d}\tau} + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t)$$

To eliminate steady-state error we can use an integral term. The integral term produces a control output based on the *error accumulated over time*.

Definition of Integrals



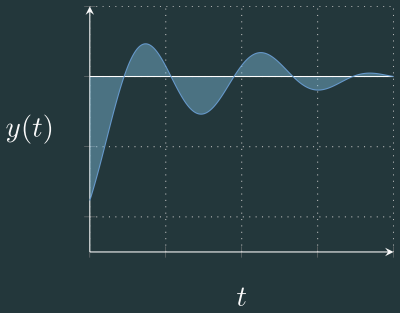
Recall that an integral is the area under a curve:

$$\int_0^t e(\tau)\,\mathrm{d}\tau$$

is the area under $e(\tau)$ for τ from 0 to t. As long as $e(t)$ is nonzero, this area will keep accumulating and generating a control output. This eliminates steady-state error as $u(t)$ will keep getting larger if there is any error.

Integral Windup

PI Controller Response



However, as $\int_0^t e(\tau)\,\mathrm{d}\tau$ and thus $u(t)$ are not necessarily zero when $e(t) = 0$, we face the problem of overshooting.

If a large change in the setpoint creates a significant initial error, the integral term can only decrease by overshooting, causing a large overshoot and oscillations. This is referred to as **integral windup**.

Saturation

Physical components have limits; you can't run a motor at 1000% output even if that's what the controller outputs. When a part of the controller exceeds its physical limit, the controller is **saturated**.

Actuator saturation is a main cause of integral windup (as the actuator is saturated, the system cannot respond fast enough to decrease the error). Sometimes we can avoid it by avoiding sudden changes to the setpoint (e.g. with a motion profile), but sometimes it is unavoidable.

Solutions to Integral Windup

When actuator saturation is unavoidable, we can use one of the following strategies to reduce integral windup:

- Disabling the integral until $e(t)$ is within a certain zone (sometimes referred to as the *Integral-Zone or I-Zone*);
- Preventing the integral from accumulating above a certain bound;
- Zeroing the accumulated error when $e(t)$ crosses 0;
- Filtering the setpoint so it doesn't move faster than the system can respond.

Proper tuning and a derivative term can also help.

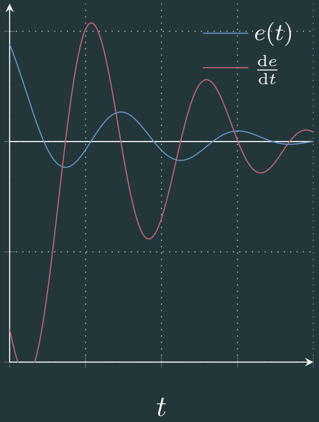
The Derivative Term

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + \boxed{K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t)}$$

Finally, the derivative term can act as a *dampener*. For a constant setpoint, the derivative term counteracts the control effort and dampens the response.

Definition of Derivatives

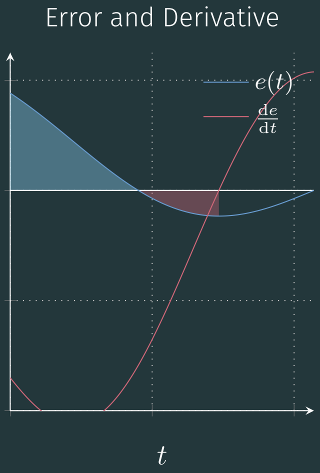
Error and Derivative



The derivative of a function $f(t)$, denoted $\frac{\mathrm{d}f}{\mathrm{d}t}$ or $\frac{\mathrm{d}}{\mathrm{d}t}f(t)$, is the rate of change (slope) of the function.

For a constant setpoint, the derivative always acts *against* the current trend of the process variable. This can produce a dampening effect or slow down overshoot.

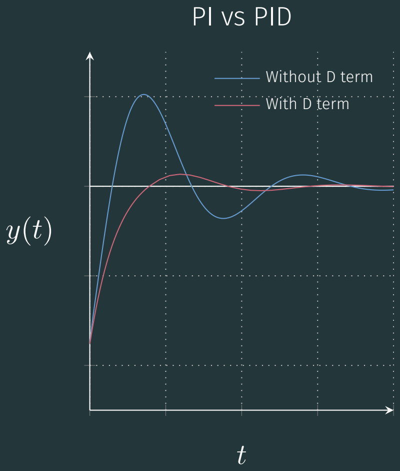
Dampening Effect Example



Consider the blue shaded region; $e(t) > 0$ so a positive control output is needed $(r(t) > y(t))$. $e(t)$ decreases, creating a negative $\frac{\mathrm{d}e}{\mathrm{d}t}$ acting *against* the control effort.

As we overshoot (red region) $e(t)$ continues to decrease. As $r(t) < y(t)$ we need a negative output to correct the overshoot. The negative derivative term now acts against the overshooting.

Dampening Visualization



With optimal tuning, the derivative term can significantly dampen overshoot, at the cost of initially crossing the setpoint later.

Alternative Interpretation

Consider if the setpoint is non-constant ($\frac{\mathrm{d}y}{\mathrm{d}t} \neq 0$):

$$\begin{aligned}
\frac{\mathrm{d}e}{\mathrm{d}t} &= \lim_{h\to 0} \frac{e(t+h) - e(t)}{h} \\
&= \lim_{h\to 0} \frac{(r(t+h) - y(t+h)) - (r(t) - y(t))}{h} \\
&= \lim_{h\to 0} \frac{(r(t+h) - r(t)) - (y(t+h) - y(t))}{h} \\
&= \lim_{h\to 0} \frac{r(t+h) - r(t)}{h} - \lim_{h\to 0} \frac{y(t+h) - y(t)}{h} \\
&= \frac{\mathrm{d}r}{\mathrm{d}t} - \frac{\mathrm{d}y}{\mathrm{d}t}
\end{aligned}$$

Alternative Interpretation

We showed that $\frac{\mathrm{d}e}{\mathrm{d}t} = \frac{\mathrm{d}r}{\mathrm{d}t} - \frac{\mathrm{d}y}{\mathrm{d}t}$, i.e. the rate of change of the error is equal to the difference between the rates of change of the setpoint and the process variable.

When we drive $\frac{\mathrm{d}e}{\mathrm{d}t}$ to 0, we're actually driving $\frac{\mathrm{d}r}{\mathrm{d}t} - \frac{\mathrm{d}y}{\mathrm{d}t}$ to 0, i.e. we're driving the difference between rates of change of the setpoint and the process variable to 0.

In effect, the derivative term attempts to make the rate of change of the process variable match the rate of change of the setpoint.

The Feedforward Term

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t) + K_f f(t)$$

In some systems, a feedforward term is also used. The feedforward term is an *open-loop term* added to increase the effectiveness of the system based on prior knowledge of the way the system works.

The definition of $f(t)$ depends on the specific system (”prior knowledge”), but being an *open-loop* quantity, it does not depend on the value of $e(t)$ or $y(t)$.

Feedforward Term Example

Consider two different feedforward terms for two different applications:

- Controlling the speed of a motor: Faster speeds lead to more friction/drag, so a higher setpoint will always require more output. We can define $f(t) = r(t)$ so increasing the setpoint increases the control output.
- Controlling the angle of an arm: A more horizontal position means more torque is needed to counteract gravity. We can define $f(t) = \sin(r(t))$ so angles further from vertical generate more output.

In both cases the feedforward term is defined in terms of $r(t)$ based on the way the system operates.

Putting PIDF Together

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t) + K_f f(t)$$

- The *proportional term* $K_p e(t)$ corrects for ***current error***.
- The *integral term* $K_i \int_0^t e(\tau)\,\mathrm{d}\tau$ corrects for ***past error***.
- The *derivative term* $K_d \frac{\mathrm{d}}{\mathrm{d}t} e(t)$ corrects for ***future error***.
- The *feedforward term* $K_f f(t)$ is a ***system-dependent open-loop*** term that aids the closed-loop terms.

Implementation

Example Implementation

In the next slides there will be an example implementation in Java.

This example is for demonstration purposes only. In reality you'll rarely have to implement a PID controller yourself (typically already done at the hardware level or in another library).

For the sake of simplicity, we skipped certain features like resetting the integral to prevent windup.

Example Implementation i

```java
/**
 * An example PID(F) controller implementation.
 */
public class PID {
    // Gains and setpoint
    private double kP, kI, kD, kF, setpoint;
    // Timestamp of the last control loop run, used to calculate time difference between runs
    private long lastTime;
    // Error in the last control loop run (used for derivative and integral) and current integral value
    private double lastError, integral;

    /**
     * Create a new controller object.
     *
     * @param p Proportional gain
     * @param i Integral gain
     * @param d Derivative gain
     */
    public PID(double p, double i, double d) {
        this(p, i, d, 0);
    }

```

Example Implementation ii

```java
    /**
     * Create a new controller object with feedforward gain.
     *
     * @param p Proportional gain
     * @param i Integral gain
     * @param d Derivative gain
     * @param f Feedforward gain
     */
    public PID(double p, double i, double d, double f) {
        kP = p;
        kI = i;
        kD = d;
        kF = f;
    }

    /**
     * Set the setpoint of the controller.
     *
     * @param r The new setpoint
     */
    public void setSetpoint(double r) {
        setpoint = r;
```

Example Implementation iii

```java
    }

    /**
     * Initialize the controller.
     *
     * Setpoint should be set before this using {@link #setSetpoint(double)}.
     * Needs to be called before {@link #run(double)}.
     *
     * @param y The current value of the process variable
     */
    public void init(double y) {
        integral = 0;
        lastError = setpoint - y;
        lastTime = System.nanoTime();
    }

    /**
     * Run the control loop for one iteration.
     *
     * @param y The current value of the process variable
     * @return The control output
     */
```

Example Implementation iv

```java
    public double run(double y) {
        // Calculate error, current time, and change in time
        double e = setpoint - y;
        long t = System.nanoTime();
        double dt = (t - lastTime) / 1.0e9;
        // Approximate integral using a riemann sum
        integral += lastError * dt;
        // Approximate derivative using the slope of a secant
        double derivative = (e - lastError) / dt;

        lastTime = t;
        lastError = e;
        return kP * e + kI * integral + kD * derivative + kF * setpoint;
    }
}
```

Tuning PIDs

Tuning PIDs

PIDs only work if your gains are chosen well. The process of choosing and adjusting gains is called **tuning**. This is what you'll actually be doing a lot of the time when working with PIDs.

Many advanced techniques and tools for PID tuning exist, but often manual tuning suffices and sometimes you have no choice. It comes with experience but the series of steps listed are one common way to do it.

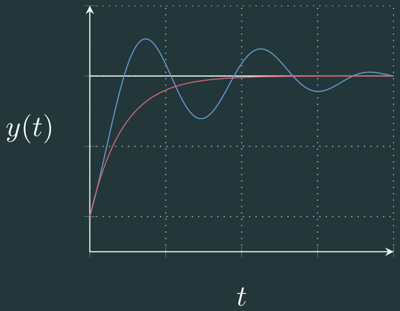
Manual Tuning Steps

We'll be tuning the system by analyzing its response to a disturbance or sudden change in the setpoint. Always begin by setting all gains to 0.

If using a feedforward term, start with K_f first. You can often start with a calculated value, e.g. for a velocity control with $K_f r(t)$ as feedforward you can use $\frac{u_{max}}{v_{max}}$.

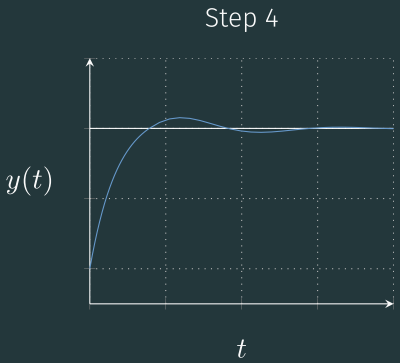
Manual Tuning Steps

Steps 1-3



1. Increase K_p until the system oscillates upon disturbance.
2. Increase K_d until oscillations go away.
3. If desired, repeat the previous two steps until increasing K_d no longer stops oscillations, and set it to the largest stable value.

Manual Tuning Steps



4. If there is steady-state error or response is slow, increase K_i until steady-state error is eliminated or response is fast enough.

Magnitudes of Gains

The magnitudes of your gains depends on the units of $y(t)$, time, and other factors. In general, most of the time these will be true:

- K_p will be a few orders of magnitude below 1.
- K_i will be much smaller than K_p.
- K_d depends on the unit of time and usually has an order of magnitude between that of K_p and K_i.
- All gains are almost always positive.

Effects of Adjusting Gains

When tuning, keep in mind the effect of adjusting each gain:

- Increasing K_p will decrease the response time if the controller is not saturated. Too much K_p can cause oscillations as a result of momentum.
- Increasing K_i will also decrease the response time and eliminate steady-state error. Too much K_i leads to integral windup and oscillations worse than K_p.
- Increasing K_d will dampen oscillations but also increase the response time. Too much K_d leads to slow responses (overdamping) and very high K_d values also cause oscillation.

The End

Thank You for Listening!

By now you should have learned the basics of PID theory and applications.

PIDs are a part of the much larger branch of applied mathematics called **Control Theory**. These are only the basics of the basics, but sufficient for FRC. Check out other control theory topics if you’re interested.