PID

WORKING WITH HARDWARE

Code for hardware is a little bit different than standalone code. If you set a variable to a specific value, it doesn't change. That isn't the same with hardware. If you set a motor to go 1 m/s, it may not go 1 m/s. The robot might hit a wall. It may be going uphill, and be trying to go that fast but really be going slower. It may be going downhill and in fact being going faster. It may go over really rough or really slippery surfaces causing it to go slower and then faster. One side of the robot may go faster than the other because one side may have more friction than the other, which would make it turn. It may be speeding up or slowing down. There may not be a motor plugged in. All you know is that you tried to make it go 1 m/s.

So we work with sensors. You need to get information back from the system in order to tell you what speed that motor is actually going. Then you can adjust accordingly, to make it go faster or slower.

Keep in mind that a sensor can only read (called sampling) every so often. So you may only have a reading every 0.1 seconds or something like that. That can mean if you start going too fast or too slow, it will take some time to react and fix it. So even if you find out you are going too fast, you will keep going too fast, and maybe even get faster for some period of time before you can fix it.

PROPORTIONAL CONSTANT

Let's think about a system where you have a robot and you want to get it up to speed. Now if we're thinking about it as a normal variable we'd have something like

```
double speed = 5; // m/s
```

But there are several reasons this doesn't work. First, it would usually be a setter function rather than a variable. Second, we don't want to go straight up to speed, any real system is going to take a while to speed up. Third, like we said above it won't necessarily stay at that speed (or even get there) without help, we need a sensor. So we have to do something a little bit different. But first we need to define a few variables and functions that will help us.

- n the sensor time step you are on. This is going to be an integer for our purposes. If we got a reading from our sensor every 0.1 seconds. Then at 0.1 seconds n would be 1, and at 1 second n would be 10.
- $w[n]$ the speed you want to robot to be at time n, This can also be called the setpoint.
- $v[n]$ the speed you get from the sensor at time n. This is our input function.
- $c[n]$ the current that is being output at time n. This is our output.
- $e[n] = w[n] - v[n]$ This is our error function. The goal is to get $v[n]$ to be as close to $w[n]$ as possible, to get the real speed of the robot to as closely mimic the desired speed as possible. So it is useful to know how far our real speed is from our setpoint.

In an idealized situation, $e[n]$ starts at $w[n]$ and goes steadily to 0. By "idealized", we mean none of the situations above, no friction, no obstacles… impossible in reality but a good start for analysis. When we have things like obstacles or friction changes, this doesn't break the math at all. This math actually helps us overcome most problems like that (I mean it won't help us drive through a wall) but we won't look at that just yet.

So let's relate the error to the output.

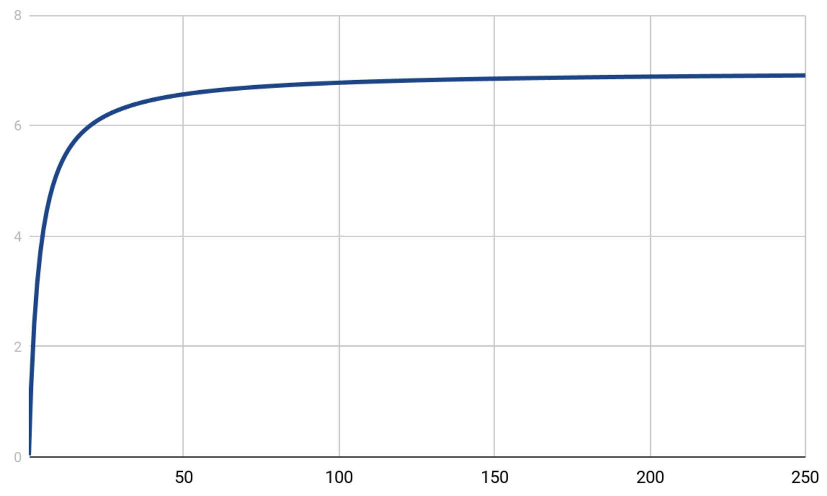
$$c[n] = K_p e[n]$$

The p here stands for proportional since it relates to the term that is directly proportional to the error. This makes sense because the further away you are from the setpoint, the more current you want to accelerate the motor so it gets closer to the right speed.

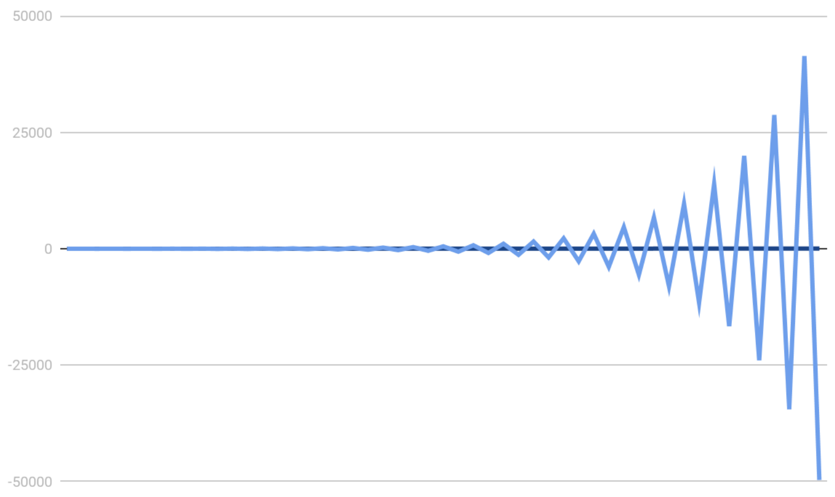
The current output is directly related to how far away from the desired speed you are at time n. K_p serves a few different purposes:

- Converts between units. $e[n]$ is in units of speed and $c[n]$ is in units of current so some conversion factor between the two is needed
- Change the acceleration: a low K_p means that the speed is going to change more slowly and a high K_p means that the speed will change faster. You may want to speed up quickly or slowly. Too fast may knock over your robot. Too slow might not even spin your motor or may just not get you there fast enough. Note: the speed will always change in relationship to the error, but the constant will change how much.

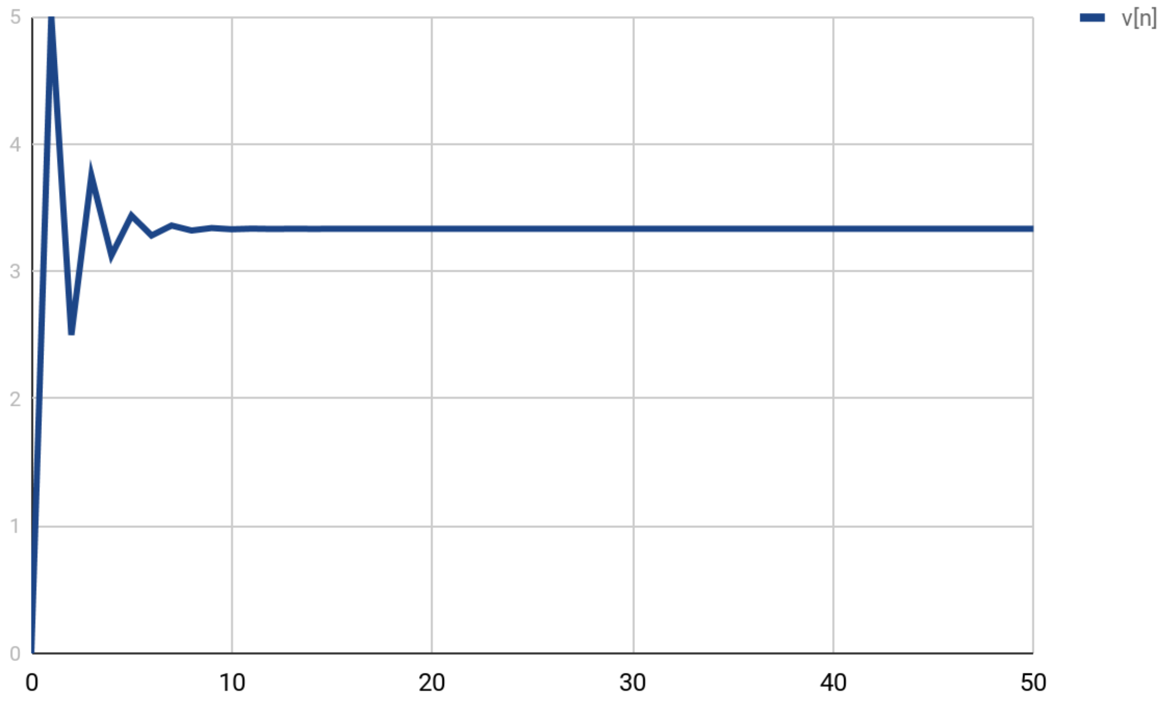
You can graph time versus the speed. This graph can take a few different shapes. It can get to an asymptote where the error starts to get so small, that the K_p changes in velocity start to get really really small. And we may get really really close to a the right value. But we don't actually get to the right speed. It may get close fast. It may take a while to get close.



But if we K_p too big, we can easily overshoot. And it can become unstable very quickly, meaning it quickly goes larger and large and far away from the desired value.



It can also stabilize with the right values, but keep in mind this is all idealized.



Problems with a P only system

But the biggest problem we have is what happens if it gets to the right speed. The error goes to zero, so the current goes to zero. And you do need some current to keep the motor running.

So you need to keep the value high enough to keep your motor moving. If the error were small, you would have to keep the constant high. But if you make the K_p too high the whole thing gets unstable. And the only other way to make the current high is to keep a high error, which means you aren't getting up to the speed you wanted which isn't a great solution.

ADDING AN I TERM

So we need something else. The formula ends up looking like this:

$$c[n] = K_p e[n] + K_i \sum_{m=0}^{m=n} e[m]$$

This term also has a constant for the same reasons we had one on the first term. For those of you that know calculus, you'll note that the summation here looks a lot like an integral over the error function, which is where the i comes from. For those of you that don't, don't worry as we'll deal with the summation here.

We are going to be summing up all the errors over time. This should give us the boost we need to have a base level of current even when the error gets really small. And when it gets really really small, the summation stops changing much, so this base level of current is going to keep pretty constant.

Problems with PI system

In most cases these two terms are enough, and you can find a way to make your system stable with just these two terms, but just to get a complete understanding, let's think about the case where they aren't.

Remember that the current needs to be enough to keep the motor going but also to accelerate it, so when we are starting at zero, which is the main case we've been looking at, the current is going to have to be larger than the value that it needs to be at to hold the motor at its final value. It has to get it up to speed. Also remember that K_p can't get too big or the system becomes unstable. So if K_p is really small, most of your current is going to come from the i term. Let's imagine a case where it is so small that we can think of it as being

$$c[n] \approx K_i \sum_{m=0}^{m=n} e[m]$$

So you would need to make a K_i that gives you big enough values that you can get up to speed. But then eventually you have to give it less current, because you would just need to keep it at speed, so the output of this equation has to get smaller. And the only way it gets smaller is you overshoot your setpoint, in which case your error gets to be negative, and your summation gets a bit smaller, and the whole equation gets a bit smaller. So you end up with a few different choices for how to set K_i. You can set it small, and it takes a while to get to a desired speed. Or you can set it large, and have it way overshoot, and hopefully come back to the right number. Unfortunately, in the worst case, it can become unstable and way overshoot in both directions and go back and forth between a 0 and a max speed until it breaks (or until you shut if off, which should come first).

ADDING A D TERM

So we add another term right? Let's see what that looks like:

$$c[n] = K_p e[n] + K_i \sum_{m=0}^{m=n} e[m] + K_d \frac{(e[n]-e[n-1])}{T}$$

Let's break this down a little more so we can understand what the different pieces are.

$$e[n] - e[n-1]$$

So this is the current error now minus the previous error, so if we're doing our job correctly the error is getting smaller and this is a negative number. Especially in the case where you set K_p and K_i large and you are getting close to your answer fast.

Now we divide that by T, which is the amount of time between samples. So we get the rise in the error graph over the run of the error graph…sound like a slope? It should…because it is. Again, aside for those of you that know calculus…the D in the constant name comes from derivative since this is a slope. If you don't want to think about the calculus here, you can think of this as the delta term.

So with the slope you have an idea of where the graph is going. And if it is going super fast, chances are you are going to overshoot, so this term gives you a chance to slow things down. The faster you are going towards the answer, the larger of a negative this is so this is a larger constant here helps you slow down the rate at which you approach the setpoint so you don't overshoot as easily. This is the term that looks into the future and makes sure that you don't go too far too fast.

Problems using PID

The reason this term doesn't get used too often is that it can be a bit messy. The real world has things that change your graph around, what we usually call noise. Any time the speed changes, this can look like a big spike in your graph. And a big spike has a big slope. Big slope means lots of contribution from this part of the formula…and that may not be so great. So maybe unless the PI version is just way too slow…just leave this part out.

PID FOR OTHER VALUES

Other applications

We've mostly been focusing on PID for speed, but PID can be used for any input/output pair where we have a feedback loop like this. Most notably in the case of a robot, it can be used for position tracking, where the value we want to track is a particular position. But this is also how thermostats keep at the right temperature.

The different systems may behave a little bit differently. For example, one of the issues with a P-only system for speed was that in order to maintain a certain speed, you need a certain amount of current, so you can't have zero current and maintain that speed. But if you have zero speed, you do keep a certain position, so a P-only system might work okay sometimes for position.

Other setpoints

You can also use PID not just starting from 0. You can go from a certain speed to a new higher or lower speed. You can even use PID to cleanly come to a stop.

Non-idealized scenarios

Let's also think about some non-ideal cases a bit.

What would happen if you go too slow for a little while?

The friction on the carpet is high, there's a bump, something makes it harder. It may start going slower than before or even come to a stop. The proportional term wouldn't change because the error value isn't changing or isn't changing much. If you are doing PID for speed, the error may change initially as it slows down, which may give you enough current to get past the obstacle. If you are going for position, the error is going to stay constant for a while. Either way once you are stuck, the error isn't going to be changing much. A delta term wouldn't change because the error isn't changing so the delta would be close to 0. What would happen is that the error would add up. The longer you are further away from the setpoint, the more the summation is going to increase. So your integral term is going to build. And the longer you are at that point, the more that term is going to build up and send more power to the motor. Eventually that should give you enough power to overcome the friction in the situation.

What happens if you are going too fast?

Say you are going downhill and you start going really fast. This ends up being the same as an overshoot. As long as you are getting the real value that the robot is moving fed into the system, the motors can compensate for the extra speed and back off on the current, even turning off the motors (or even reversing!) if it helps get to the right speed. This system doesn't care why the motors are going the speed they are, they just care what speed they are going and help to make them go the right speed.

What if the friction is different on both sides?

Usually you have motors on both sides and each side is compensating independently so you can keep each side going the right speed. So you probably want to track both sides. But that doesn't necessarily fix the problem completely. During the period of time that the robot is stuck on one side, the robot will turn a little, and once it gets up to speed on both sides, it isn't going to straighten out. In order to go straight, you would need to track its angle using a gyro, and keep gyro at 0 (relative to the starting point). There are other ways to fix this once you get deeper into control systems but for now we'll leave it at that.

What if you hit a wall?

OK, PID can't solve every problem. It will treat this scenario just like any other and keep trying to get to a setpoint. It may continue to try to overcome it like the high friction situation. It will continue to try harder and harder as the integral term builds up, until it shuts down, either manually or due to burn out. Since the integral term actually goes from time 0 until now, it will stick around, even if the robot now tries to go away from the obstacle. So it will still be trying to fight that wall even when it is going in a slightly different direction. This is called integrator windup.

WHAT CONSTANTS TO USE

So we've got this great idea for a system but it needs constants…so what constants do you use? How do you know what values for K_p, K_i and K_d to use? There are several different methods that can be used we'll talk about two here.

Manual tuning

The four main things in the system you want to get right are

1. Rise time: the time it takes to get past 90% of the right value for the first time
2. Overshoot: the difference between the peak level and the steady state

3. Settling time: the time it takes to get to a steady state
4. Steady-state error: the difference between steady-state output and a desired value

So let's look at the impact the increasing each of the three different values have on the four different system dynamics

	Rise Time	Overshoot	Settling TIme	Steady State Error
K_p	Decrease	Increase	Minor change	Decrease
K_i	Decrease	Increase	Eliminate	Decrease
K_d	Minor change	Decrease	No effect	Improve if K_d is small

So how do you use this to pick constants (aka tune the system)?

1. Start by increasing K_p to decrease the rise time, with K_i and K_d at 0

2. Use K_i to eliminate the steady state error.

3. Use K_d to reduce overshoot and settling time.

But that doesn't really tell you what to set the values to. Or how to pick values. Just to start trying things until they work. So maybe there's a better way than that.

Ziegler-Nichols

The Zielger-Nichols method gives you a little bit more structure in how you tune. First set all constants to 0. Then, only increasing K_p, find a steady oscillation point, where it goes back and forth consistently around the setpoint. Keep track of the constant value used at that point, we'll call that K_u and the oscillation period, we'll call that T_u.

Then based on those two values we have rules of how we calculate our constants in different situations

Control Type	K_p	K_i	K_d
P	0.5 K_u	0	0
PI	0.45 K_u	0.54 K_u / T_u	0
PID	0.6 K_u	1.2 K_u / T_u	3 K_u T_u /40
Some overshoot	0.33 K_u	0.66 K_u / T_u	0.1089 K_u T_u
No overshoot	0.2 K_u	0.4 K_u / T_u	0.66 K_u T_u

Remember that these are heuristics, not rules. They will not always work for every case, that is why there are multiple of them.