An Introduction to PID Controllers

Written by George Gillard

Published: 22-July-2017

Updated: 10-November-2017

Introduction

A PID Controller, if created and tuned well, is a powerful tool in programming for incredibly efficient and accurate movements. There are three key components behind the PID Controller – Proportional, Integral, and Derivative, from which the acronym PID originates from. However, you don't strictly need to use all three together – for example, you could create just a P controller, a PI controller, or a PD controller.

In this guide, we'll learn about each component, how they work together, and how to put it all into practice. There will be some pseudocode examples to help out along the way. PID by nature involves calculus, however don't be put off if you haven't learned calculus yet, as I've attempted to design this guide to be easy enough for anyone to understand.

Sections:

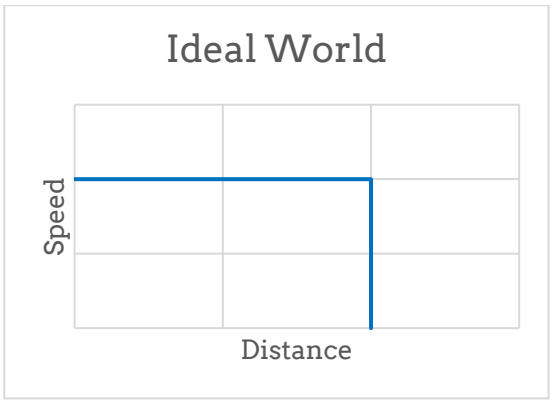
1. Background Information

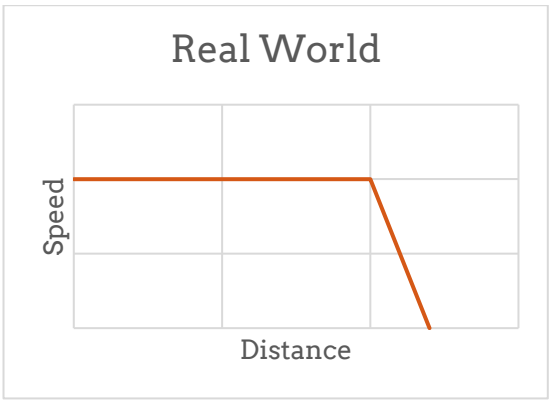
Before cracking into the nitty-gritty of a PID controller, a bit of history into the development provides some useful insight:

> *It was not until 1922 that PID controllers were first developed using a theoretical analysis, by Russian American engineer Nicolas Minorsky for automatic ship steering. Minorsky was designing automatic steering systems for the US Navy and based his analysis on observations of a helmsman, noting the helmsman steered the ship based not only on the current course error, but also on past error, as well as the current rate of change; this was then given a mathematical treatment by Minorsky. His goal was stability, not general control, which simplified the problem significantly. While proportional control provides stability against small disturbances, it was insufficient for dealing with a steady disturbance, notably a stiff gale (due to steady-state error), which required adding the integral term. Finally, the derivative term was added to improve stability and control.*
>
> *Trials were carried out on the USS New Mexico, with the controller controlling the angular velocity (not angle) of the rudder. PI control yielded sustained yaw (angular error) of ±2°. Adding the D element yielded a yaw error of ±1/6°, better than most helmsmen could achieve.*

\- Wikipedia

Typical basic programming used on a robot is along the lines of "run at a constant power until you reach a certain point and then stop". In an ideal world, we'd be able to do this, and stop exactly on the spot. However, in the real world, there are additional and largely unpredictable factors that will cause our system to overshoot the "setpoint" (ideal target), such as momentum (influenced by speed and hence battery voltage) or other external influences.





2. P: Proportional

The Proportional component provides the bulk of the power for controlling your system. The key objective is to give a large amount of power when there is a long way to go, but only a small amount of power when you're nearly at your setpoint. This results in a smooth deceleration through the movement as you approach the setpoint.

2.1 The Error

Firstly, a variable is created, called the error. The error is really simple – just the difference between the current sensor value, and what you want that sensor value to reach (the setpoint). For example, the error could be the distance remaining to be travelled, the height remaining to be lifted to, etc. If your phone battery was at 30% and you were charging it to reach 100%, the error would be 70% - it's just the difference between where you're at, and where you want to be.

As you'd suspect, to calculate the error, you'd create something as simple as this:

```
error = setpoint – sensor value
```

To solidify the understanding of the error, have a look at the following table. The goal is for a robot to drive a total of 1000 units:

Target value (setpoint)	Current sensor reading	Error
1000	0	1000
1000	200	800
1000	400	600
1000	600	400
1000	800	200
1000	1000	0

Similarly, if the robot then overshot the setpoint, the error would begin to become negative (indicating the robot now needs to go in reverse):

Target value (setpoint)	Current sensor reading	Error
1000	600	400
1000	800	200
1000	1000	0
1000	1200	-200
1000	1400	-400

2.2 Assigning an Output Power

To achieve the nice smooth deceleration that the Proportional Controller provides, we could simply set the power of our motors to be equal to the error, like so:

error = setpoint – sensor value

power = error

However, you may find that the power values don't seem to be scaled right. The robot may be a bit too gentle approaching the target, and may in fact not have enough power at all to reach the setpoint when the error becomes small. Or alternatively, the robot might be a bit aggressive, and it might significantly overshoot, and then overcorrect, in a never-ending cycle.

To combat this issue, we introduce another value, the proportional constant (kP). Simply put, we multiply the error by kP when we assign the error to the output power. Later we'll tune this value to get the desired output, but for now here's how we'd implement it:

error = setpoint – sensor value

*power = error*kP*

Up until now, we've skipped over a fairly critical part of the PID (or P, so far) controller. Currently, we could run the code and it would perform the calculations once. However, we'd obviously need to keep recalculating these values as our robot moves, otherwise our error and powers will never update. To fix this, we put everything in a loop.

Here's some slightly more realistic pseudocode, which with some completion would work absolutely fine for many situations:

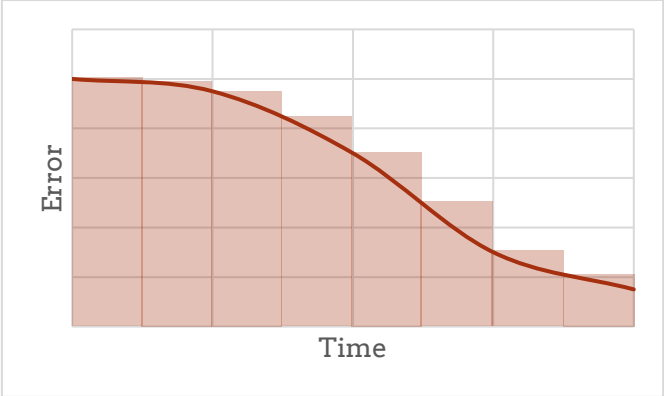
```
void myPID(int setpoint)
{
  while ( some condition )
  {
    error = setpoint - sensor value;
    power = error*kP;
  }
}
```

3. I: Integral

You may recall from the short bit if history covered in the introduction that the proportional term was determined to be sufficient for small disturbances, but not much else. The integral term is going to give us some versatility when it comes to other disturbances (e.g. a "stiff gale" was a problem in the case of the automatic ship steering). You'll also find with the proportional component that once the error becomes small you have very little power, and might see some significant remaining error that just isn't eliminated – the integral will get rid of this for you by slowly increasing the power. The integral is going to be concerned with looking back in time over all the errors your system has calculated. By definition, an integral is an area under a curve in calculus. For our purposes, calculating the integral using standard calculus isn't really feasible, so we do it the easy way. We're going to work out the total area by summing the area of many thin slices.

3.1 The Maths

The example below shows a curve with the area estimated using rectangular slices – much like what we're going to do with our code. Each slice is as tall as the error, and has a constant width we're going to call "dT". As we can see, it isn't perfect, but gives a decent estimation of the area. A smaller value for dT (thinner slices) gives us a more accurate estimation of the area.



The "area under the curve" for each cycle of our loop is going to be the current error, multiplied by the time it takes for that cycle of the loop. It's a rough approximation, but it works fine for us with such slim slices of time.

```
area = error * dT
```

The integral is equal to the sum of all of these areas. At any instant, it is the sum of the areas of all the previous cycles, so we create a variable ("integral"), and add on the new slice of area in each cycle of our loop:

```
integral = integral + error*dT
```

Since dT is normally a constant delay that we ourselves set (e.g. wait 15 milliseconds per cycle of the loop), we can factor it out later and hence tend to ignore its existence. Hence, we'll use this to calculate our integral:

```
integral = integral + error
```

Consider a case where our error is decreasing at a nice constant rate (not realistic), ignoring dT. The following table describes how the integral would be calculated:

Cycle No.	Error	Integral
1	1000	1000
2	800	1800
3	600	2400
4	400	2800
5	200	3000

Now, consider what would happen if we had some external influence that caused our error to reduce more slowly. In the above example it decreased 200 units per cycle. The next table considers what would happen if that was 100:

Cycle No.	Error	Integral
1	1000	1000
2	900	1900
3	800	2700
4	700	3400
5	600	4000

As we can see, in the first example our integral was 3000 after 5 cycles. Now, with a slower deceleration, it's 4000. This increase in value is our indicator of some external influence and will help create some more versatile control for our system.

3.2 Assigning an Output Power

A higher value of the integral indicates that there is some external influence slowing down our system. Hence, to combat this we add some extra power to give a bit more of a "boost", and to accomplish this we add the integral to the existing output. To account for the scaling issues just like we saw for the proportional term, we introduce another constant – kI. As you can probably tell, our integral is likely going to be a huge number, thus entirely useless as an output power as-is, so we often expect kI to be quite small compared to kP, especially in cases where we are neglecting dT in our equations.

```
power = error*kP + integral*kI
```

We typically don't bother with brackets to separate each term because our order of operations (BEDMAS/BODMAS) takes care of that for us, but feel free to include them if it makes it easier to read, like so:

```
power = (error*kP) + (integral*kI)
```

Our code so far would now look something like this pseudocode:

```
void myPID(int setpoint)
{
  while ( some condition )
  {
    error = setpoint - sensor value;
    integral = integral + error;
    power = error*kP + integral*kI;
    wait 15 mSec;
  }
}
```

The 15 milliseconds wait at the end is really important – this is our dT. Without it, your integral value will skyrocket to some huge number as numbers get added together without any break, and 15 milliseconds is quite fast enough for an accurate integral for most of our purposes. This delay will also prove to be critical for the derivative term soon, as I'm sure you'll realise when we reach that section. If you don't consider a constant dT, you will need to measure the time per cycle and consider that in your integral calculation.

3.3 Issues

3.3.1 Problem No. 1:

When the error reaches zero, that is you've made it to the setpoint, the integral is most likely going to be significant enough to keep the output power high enough to continue. This can be a nuisance in situations where you don't need any additional power to hold the position – e.g. if this is for a drive train on a flat surface, if your wheels continue turning that's somewhat of an issue!

In these cases, we can reset the integral to zero once our error passes the setpoint, like so:

```
void myPID(int setpoint)
{
  while ( some condition )
  {
    error = setpoint - sensor value;
    integral = integral + error;
    if (error = 0 or passes setpoint)
        integral = 0;
    power = error*kP + integral*kI;
    wait 15 mSec;
  }
}
```

Note: if this PID controller is for a system that needs a bit of help to hold its position (e.g. an arm lifting up some weight), **you absolutely should not try this**. When your error passes the setpoint, the integral value will gradually be diminished and it will still settle. This is only suitable for systems that maintain their sensor value with zero power (e.g. wheels on a flat surface).

3.3.2 Problem No. 2:

Integral windup is an issue where a large change in setpoint occurs (e.g. 0 to 1000), causing the integral to start calculating for huge error values. This then results in an unusably high value for the integral when you really want it (i.e. near the setpoint). There's a few ways to combat this problem:

Solution #1: Limit the value that the integral can reach.

```
if (integral is huge)
   integral = maximum value;
```

Solution #2: Limit the range in which the integral is allowed to build up in (i.e. once the error is below a certain value, or once the current output power is less than a certain value, e.g. 100%).

```
if ( error is big )
    integral = 0;
```

Solution #3: Gradually increase the error between it's previous value (e.g. 0) and it's new value (e.g. 1000). This is a little more complicated to code, and a little out of the scope of this guide, but is still very possible.

For this guide, we'll use the second solution using the error as a limiting factor, as it's a little more useful than the first but still simple to program.

3.4 The New Code

The PI controller we've created so far would be something like this:

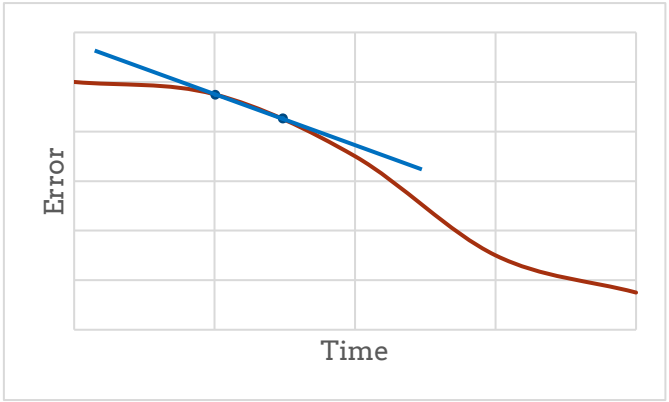
```
void myPID(int setpoint)
{
  while ( some condition )
  {
    error = setpoint - sensor value;
    integral = integral + error;
    if (error = 0 or passes setpoint)
        integral = 0;
    if (error is outside useful range)
        integral = 0;
    power = error*kP + integral*kI;
    wait 15 mSec;
  }
}
```

4. D: Derivative

So far we've looked at controlling the output power based on the current error, and past errors. Now, we're going to look a bit into the future. The idea of the derivative is to look at the rate of change of our error – how fast are we approaching the setpoint? Do we need to slow down a bit? The contribution from the derivative will be in the direction opposite to your current direction of travel, with a larger magnitude for a greater speed. It typically will be outweighed by the proportional and integral components, but if there's some deviation from "normal" and the robot is going faster for some reason, the derivative component will become larger and your output power will be reduced. Similarly, if your robot is going slower than normal, the derivative component will be smaller in magnitude and hence the output power will become greater.

4.1 The Maths

As described above, the derivative is the rate of change. In calculus, this is also the gradient (slope) of your curve. If your curve is steeper, you will have a bigger gradient (and thus a larger derivative).



In general, to find the gradient of a curve, you'd do something like this:

$$Gradient = \frac{change\ in\ Y}{change\ in\ X} = \frac{Y2 - Y1}{X2 - X1}$$

For us, our Y axis is our error, and our change in X is dT, so our derivative is:

$$Derivative = \frac{current\ error - previous\ error}{dT}$$

Just like with the integral, if we treat dT as a constant, we can ignore its effect in our calculations and merge it in with our constants later on.

We can use the existing error we have for our "current error", but for our "previous error" we need to add in a new variable. All we'll make this variable do is keep track of that our error was in the previous cycle.

So, to calculate our error, we'll use something like this:

```
derivative = error - prevError;
prevError = error
```

4.2 Assigning an Output Power

Just like for the Proportional and Integral components, we add a constant ("kD") to account for issues with scaling for our derivative. A value that is too high will cause instability - with the power contribution from the derivative overpowering the total power momentarily. A value that is too small will cause a derivative component that may as well not exist.

To incorporate to our existing output power, we add our derivate component onto the end, like as follows:

```
power = error*kP + integral*kI + derivative*kD
```

Our PID controller now looks something like this:

```
void myPID(int setpoint)
{
  while ( some condition )
  {
    error = setpoint - sensor value;
    integral = integral + error;
    if (error = 0 or passes setpoint)
        integral = 0;
    if (error is outside useful range)
        integral = 0;
    derivative = error - prevError;
    prevError = error;
    power = error*kP + integral*kI + derivative*kD;
    wait 15 mSec;
  }
}
```

5. Tuning

The most time and labour intensive, as well as generally the most frustrating and tedious part of a PID controller is tuning the constants to make it all work well. There are different ways of tuning these constants, from using computers to mathematics and just trial & error. I'm going to discuss the last two options, the most popular for most educational/competition robots being trial & error. At all times when tuning, it's worthwhile keeping an eye on the value of the error or sensor, instead of just relying on sight to judge what "looks" right.

First of all, we'll look into some factors that determine the behaviour and performance of our PID controller in reality:

- Rise time – the time it takes to get from the beginning point to the target point
- Overshoot – how far beyond the target your system goes when passing the target
- Settling time – the time it takes to settle back down when encountering a change
- Steady-state error – the error at the equilibrium, when it's stopped moving
- Stability – the "smoothness" of the motion

Now, let's check out how these are effected by an increase in our three constants:

Parameter	Rise Time	Overshoot	Settling Time	Steady-State Error	Stability
kP	Decrease (faster)	Increase (further)	N/A	Decrease (more precise)	Worsens
kI	Decrease (faster)	Increase (further)	Increase (takes longer)	Decrease (more precise)	Worsens
kD	N/A	Decrease (closer)	Decrease (quicker)	N/A	Improves*

- Wikipedia

** If kD is small enough. Too much kD can make it worse! Since the derivative term acts in the opposite direction to the proportional and integral components, if the power produced by the derivative term is too great it will outweigh the proportional and integral components, causing the robot to slow down and potentially stop when it shouldn't. When the robot slows down, the derivative component will weaken and the robot will once again be able to continue, only until the derivative term becomes strong enough once again to slow the robot down unnecessarily. The resulting motion looks jumpy, or jittery.*

Note: for the steady-state error, it's important to remember that reliability is generally more important than accuracy. By that, if you're always exactly 10 units over the target, that's typically better than having a steady-state error ranging between -5 to +5, because you can then just predict you'll be 10 units over and adjust as appropriate.

5.1 Trial & Error (Manual Tuning)

This method of tuning your PID controller is done entirely by you, with no extra tools to help you out other than your fundamental knowledge and understanding. It can be the most repetitive and tedious process, but often considered the simplest.

First of all, you set all three constants (kP, kI, kD) to zero. This "disables them". We'll tune them one by one, rather than jumping straight in. We generally tune in the order of Proportional, Derivative, Integral, that is, we tune in the order of kP, kD and finally kI. This entire process relies on making a prediction for your constant ("trial"), and then adjusting it when it doesn't go to plan ("error"). It's important to be prepared to stop your robot (both by disabling it from your program or a switch, and by physically catching it if necessary), as you'll likely make a prediction that is far off an appropriate value. So long as you're ready, there typically isn't too much harm in just experimenting.

1. Increase kP until the robot oscillates just slightly, once or twice. We're interested in achieving a fast motion to the target here, but not too violent – it needs to settle, and in a reasonable amount of time!
2. Start adding kD until the steady-state error starts to decrease until something suitable. This will allow us to maintain the fast motion from the Proportional component, whilst minimising the overshoot. You may need to go back to adjusting kP a little.
3. Start adding kI until any minor steady-state error and disturbances are accounted for. You may need to adjust kD when doing this.
4. Using the knowledge from the table on the previous page, keep adjusting the constants until you end up with a nice, quick but smooth motion that you're happy with.

This can be very frustrating and difficult the first few times, but it gets a lot better with practice, and you'll be able to guess fairly accurate values for your constants with a bit of experience.

5.2 Mathematical Tuning

Mathematics can be used to provide a decent estimate for your PID constants. Typically, you'll need to make some manual adjustments afterwards, but mathematical models can do a reasonable amount of the bulk of the work for you. For this, we'll look at the Ziegler-Nichols method, which also requires a little bit of manual work at the beginning for the calculations.

Just as for the trial & error method, begin by disabling all three constants (set them to zero).

1. Increase kP until you get steady continuous oscillations. These need to be stable and consistent. Record this value as "kU" (ultimate or critical gain).
2. Measure the period of the oscillations caused by step 1. That is, the time it takes to do a full cycle from a point back to itself. You can measure this with a bit of programming, or with a stopwatch. It's a good idea to measure many oscillations and then divide the time by the number of oscillations to get a good average. Record this number as "pU" (period for ultimate or critical gain).
3. Calculate the approximate constant values from the following table, depending on the type of your controller:

	kP	**kI**	**kD**
P	0.5*kU	0	0
PI	0.45*kU	0.54*kU/pU	0
PD	0.8*kU	0	0.1*kU*pU
PID	0.6*kU	1.2*kU/pU	0.075*kU*pU

- Wikipedia

As previously mentioned, you'll most likely need to make some adjustments through trial & error to perfect the motion. The values in the tables above for kI and kD do not account for any assumption for ignoring dT, so you would need to accommodate that as well in your calculations.

6. Conclusion

This document covers the basics of creating a PID controller. We've looked at all three components, and how they can help to create a reliable autonomous movement for your system.

It's important to note that there are other features you may need to implement to your code to improve the controller. It's also important to note that you don't need all three components to create a good controller – depending on your situation, a P, PI, or PD controller might be just as good, if not more appropriate.

Understanding the fundamentals behind how and why a PID controller works will aid you tremendously with your programming. It is worthwhile searching for some examples online for various applications to broaden your knowledge.

I hope this guide has helped you, and I wish you the best of luck with your programming!