47510W Phoenix Horizon Programming

2024-2025

Contents

1 PID Controller

A PID Controller (Proportional–Integral–Derivative Controller) is a control system that brings the system softly to a target, preventing overshoots and making for a more accurate program. PID controllers are used within our program to make more accurate turns and allow fine control over the amount the robot moves.

1.1 How Does a PID Controller Work?

Each term, P I D, each serve a specific role in creating and controlling the input which is used to reach the target. By adding up the values of the terms we get the input. We will analyze the function of each term individually.

P Term The P (Proportional) increases the input proportionally to the distance to the target. In other words, the farther you are from your target the larger the P Term is. P is the error, or distance from the target. We calculate the value of P with the following formula.

$$P = T - M$$

Where T is target and M is the measured current value

The P Term is the primary way of tuning the PID and is always the first step. The specifics of tuning and how we tune each term will be discussed in a later section.

I Term The I (Integral) is the cumulative error. The I term increases as the program runs and increases faster if farther away from the target. Which allows the controller to power over an obstacle, most often friction close to the target. We calculate the value of I with the following formula.

$$I = I_0 + P$$

Where I_0 is the I term from the previous cycle, and P is the current error(The P term)

D Term The D (Derivative) is how fast the error is changing. The D term helps bring the controller out of a oscillating state caused by repeated overshoots, as are often induced by the P term. We calculate the D term with the following formula.

$$D = P - P_0$$

Where P is the error(P term) and P_0 is the previous error (last P Term)

Putting It Together To the get the input that we feed to our system, usually a motor for our purposes, we add together our three terms. Each of which multiplied by its tuning information. Tuning specifics will be covered in the next section. Our input formula is as follows.

$$R = P \cdot K_P + I \cdot K_I + D \cdot K_D$$

Where R is the output(or return), P I and D are the calculated values, and K_P K_I and K_D are the tuning values

1.2 Tuning A PID

For a PID to properly reach the target it must be tuned. Inadequate tuning can lead to failure to reach the target or aggressive oscillations. We tune a PID by changing tuning parameters, these parameters are called K_P K_I and K_D. The first step is to set all tuning parameters to 0. By setting a paramerter to zero we can cut out thier influence for the time being. As in the previous section we will look at the tuning of each term individually.

Tuning the K_P Term The first step to tuning a PID is to set the K_P term. The first step to setting K_P is to increses it until we get an overshoot and a settle either on the target or settles into an undershoot. If it settles on the target we are done with tuning and PID should be ready to use. If we settle into an undershoot we must move onto setting the K_I term. If we overshoot or enter oscillations without settling we need to decrese the K_P term.

Tuning the K_I Term The purpose of changing the K_I term is to fix steady state error after the control settles. As a result of the behavior of the I term to increses over time, it can add input to make the final push to the target even after the distance is not great enough for the P term to have an effect. The K_I does not need to be changed in all application. The K_I term should be incresed with caution because incresing it even by small ammounts can introduce instability and overshoot.

Tuning the K_D Term The purpose of the K_D term is to control oscillations in the system. We should change K_D when the system has small but unacceptable oscillations. If the oscillations are large, it points to error in the K_P term. Changing K_D is often unnecessary in the context of robotics. We should change K_D to refine out small errors when working with more percise systems such as driving and turning. However systems which do not need to be super percise, like the speed of a flywheel, we can forgo tuning this term. Tuning of K_D is a final refinement step to add extra stability or to counteract instability added by tuning K_I.

1.3 Coding a PID for Robotics

When using a PID for an actual application we can use the information described above to create a foundation, but we also need to add some extra parts to help with the imperfections of the real world. The change that must be made is to tell the program when it is "Close Enough" and it can stop. There is also the problem of feeding the input to the application. First we will look at the parts which are simmilar to the PID theory above.

Initalizing Variables and Setting Up The first step to creating any new code system is to create an object (See the "Overview of Object Oriented Programming" to get information on this concept), and create variables we need. For now we will only create variables related to the theory and not the modifactions required for a code implementation. This is all done in the header file (Also see the "Overview of Programing Concepts" for more information). So far, our code looks like the following.

```
in pid.hpp:

class PID_OBJ
{
    public:
        double p;
        double i;
        double d;
        double kp;
        double ki;
        double kd;

        double lastErr;

        PID_OBJ(double kp, double ki, double kd);

        double pidCalc(double target, double current);
        void reset(double target);
};
```

Let's break down that chunk of code a piece at a time. The first thing we see is the class declaration and the public keyword. These are explained in depth in the "Overview of Object Oriented Programing". For our purposes of explaining PID, they are unimportant. Then we see a list of variables all declared as floating point(decimal) values with double percision(hence the name 'double'). After that we declare 3 functions which we will look at in a moment. First, however, let us analyze the variables.

Variables. We can see that we have many famillar variables which are reminicent of the equations in the previous section. We also have a new variable

`double lastErr` which will keep track of what the error was during the last cycle of the control loop. If you recall from the section of the D term, having access to the last error is important to the calculation of the term.

Functions. Now we will look at the functions we have declared. The first fucntion `PID_OBJ` is a contructor and is called on the creation of a new PID. It takes three arguments, `kp ki kd`, which should be fammilar from both the section on tuning and our variables above. These arguments are how we will tune our system. The `pidCalc` function does the operations described in section 1.1. Finnaly the reset function allows us to reset the terms without creating a new PID. This is important between tasks to prevent interference. First we will look at the constructor and reset functions, then we will look at the `pidCalc` function.

Constructor and Reset Functions The constructor and reset functions serve to initilize the PID before an external control loop. We will first look at the constructor.

Constructor. The constructor serves to build a new PID object and set tuning information. The code within the constructor is as follows.

```
in pid.cpp:

    this->kp = kp;
    this->ki = ki;
    this->kd = kd;
```

We are simply setting the passed values we saw in the header to internal values in the object which we can access later to do calculations with. This allows us to tune once. However, to use the PID multiple times we need to look at the rest function.

Reset. The reset function sets the values of the PID to be ready for operation. It is absolutly necessary that this function is called before attempting to use the PID. We write the reset function like so.

```
in pid.cpp:

    this->p = 0;
    this->i = 0;
    this->d = 0;
    this->lastErr = target;
```

We set each term to zero so the calculation has a clean slate. We set `lastErr` to our target value because before the robot has moved, we can assume that the error(distance to the target) is equal to the target. This code could also be placed in the constructor, but this creates a "Disposable PID" which can only be used once. By moving the reset code into it's own function, we can reuse our PID.