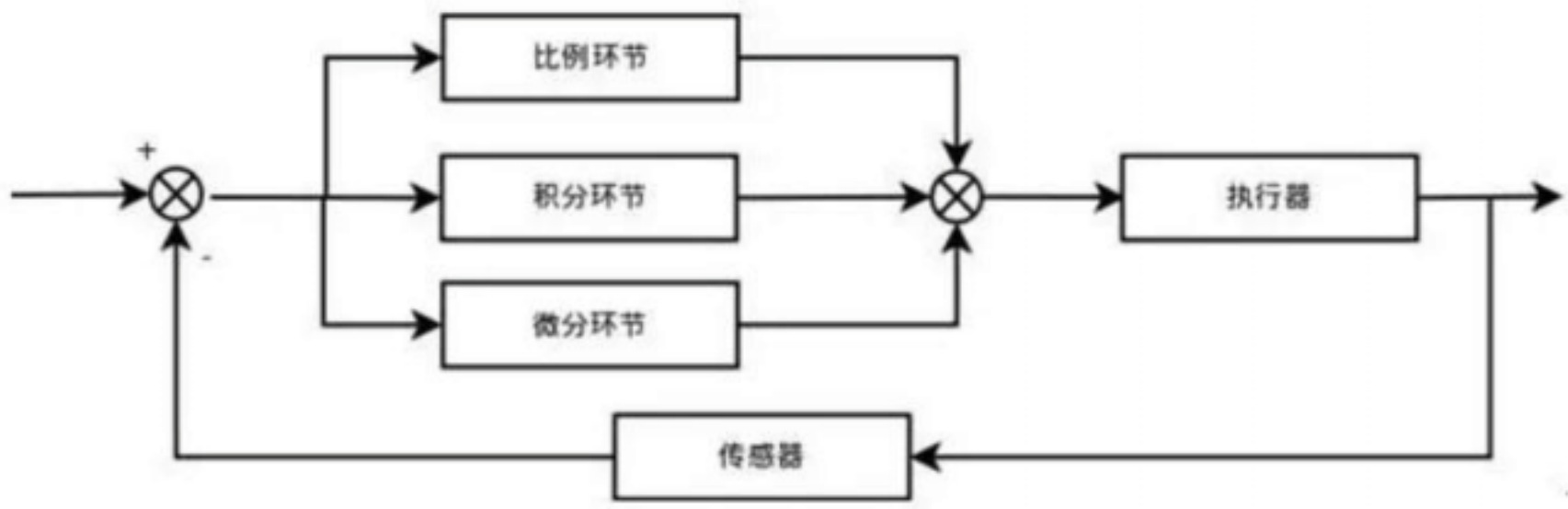
PID 控制算法一勺烩——从模型到实现再到口诀

一、 PID 的数学模型

在工业应用中 PID 及其衍生算法是应用最广泛的算法之一， 是当之无愧的万能算法，如果能够熟练掌握 PID 算法的设计与实现过程， 对于一般的研发人员来讲，应该是足够应对一般研发问题了，而难能可贵的是，在很多控制算法当中，PID 控制算法又是最简单，最能体现反馈思想的控制算法，可谓经典中的经典。经典的未必是复杂的，经典的东西常常是简单的，而且是最简单的。 PID 算法的一般形式：



PID 算法通过误差信号控制被控量，而控制器本身就是比例、积分、微分三个环节的加和。这里我们规定（在 t 时刻）：

1.输入量为 $i_{(t)}$

2.输出量为 $o_{(t)}$

3.偏差量为 $err_{(t)} = i_{(t)} - o_{(t)}$

$$u_{(t)} = k_p\left(err_{(t)} + \frac{1}{T_i}\cdot\int err_{(t)}d_t + \frac{T_D d_{err_{(t)}}}{d_t}\right)$$

二、 PID 算法的数字离散化

假设采样间隔为 T，则在第 K 个 T 时刻：

偏差 $err_{(k)} = i_{(k)} - o_{(k)}$

积分环节用加和的形式表示，即 $err_{(k)} + err_{(k+1)} + ...$

微分环节用斜率的形式表示，即 $[err_{(k)} + err_{(k+1)}] / T$

PID 算法离散化后的式子：

$$u_{(k)} = k_p\left(err_{(k)} + \frac{T}{T_i}\cdot\sum err_{(j)} + \frac{T_D}{T}\left(err_{(k)} - err_{(k-1)}\right)\right)$$

则可表示成为：

$$u_{(k)} = k_p\left(err_{(k)} + k_i\sum err_{(j)} + k_d\left(err_{(k)} - err_{(k-1)}\right)\right)$$

其中式中：

比例参数 k_p：控制器的输出与输入偏差值成比例关系。 系统一旦出现偏差， 比例调节立即产生调节作用以减少偏差。特点：过程简单快速、比例作用大，可以加快调节，减小误差；但是使系统稳定性下降，造成不稳定，有余差。

积分参数 k_i：积分环节主要是用来消除静差，所谓静差，就是系统稳定后输出值和设定值之间的差值， 积分环节实际上就是偏差累计的过程， 把累计的误差加到原有系统上以抵消系统造成的静差。

微分参数 kd：微分信号则反应了偏差信号的变化规律， 或者说是变化趋势， 根据偏差信号的变化趋势来进行超前调节，从而增加了系统的快速性。

PID 的基本离散表示形式如上。目前的这种表述形式属于位置型 PID，另外一种表述方式为增量式 PID，由上述表达式可以轻易得到：

$$u_{(k-1)} = k_p(err_{(k-1)} + k_i \sum err_{(j)} + k_d(err_{(k-1)} - err_{(k-2)}))$$

那么：

$$\Delta u_{(k)} = k_p(err_{(k)} - err_{(k-1)}) + k_i err_{(k)} + k_d(err_{(k)} - 2err_{(k-1)} + err_{(k-2)})$$

上式就是离散化 PID 的增量式表示方式，由公式可以看出，增量式的表达结果和最近三次的偏差有关， 这样就大大提高了系统的稳定性。 需要注意的是最终的输出结果应该为：

输出量 =$u_{(k)}$ + 增量调节值

三、 PID 的 C 语言实现

1.位置式 PID 的 C 语言实现

上边已经抽象出了位置性 PID 和增量型 PID 的数学表达式，这里重点讲解 C 语言代码的实现过程。

第一步：定义 PID 变量结构体，代码如下：

```
struct t_pid{
    float SetSpeed;           //定义设定值
    float ActualSpeed;        //定义实际值
    float err;                //定义偏差值
    float err_last;           //定义上一个偏差值
    float Kp,Ki,Kd;           //定义比例、积分、微分系数
    float voltage;            //定义电压值（控制执行器的变量）
```

```
    float integral;             //定义积分值
}pid;
```

第二部：初始化变量，代码如下：

```
void PID_init(){
    pid.SetSpeed=0.0;
    pid.ActualSpeed=0.0;
    pid.err=0.0;
    pid.err_last=0.0;
    pid.voltage=0.0;
    pid.integral=0.0;
    pid.Kp=0.2;
    pid.Ki=0.015;
    pid.Kd=0.2;
}
```

统一初始化变量，尤其是 Kp,Ki,Kd 三个参数，调试过程当中，对于要求的控制效果，可以通过调节这三个量直接进行调节。

第三步：编写控制算法，代码如下：

```
float PID_realize(float speed){
    pid.SetSpeed=speed;
    pid.err=pid.SetSpeed-pid.ActualSpeed;
    pid.integral+=pid.err;
   pid.voltage=pid.Kp*pid.err+pid.Ki*pid.integral+pid.Kd*(pid.err-pid.err_last);
    pid.err_last=pid.err;
    pid.ActualSpeed=pid.voltage*1.0;
    return pid.ActualSpeed;
}
```

注意：这里用了最基本的算法实现形式，没有考虑死区问题，没有设定上下限，只是对公式的一种直接的实现，后面的介绍当中还会逐渐的对此改进。

到此为止， PID 的基本实现部分就初步完成了。下面是测试代码：

```
int main(){
    PID_init();
    int count=0;
    while(count<1000)
```

```
    {
        float speed=PID_realize(200.0);
        printf("%f\n",speed);
        count++;
    }
return 0;
}
```

2.增量型 PID 的 C 语言实现

上一节中介绍了最简单的位置型 PID 的实现手段，这一节讲解增量式 PID 的实现方法。

```
#include<stdio.h>
#include<stdlib.h>

struct t_pid{
    float SetSpeed;             //定义设定值
    float ActualSpeed;          //定义实际值
    float err;                  //定义偏差值
    float err_next;             //定义上一个偏差值
    float err_last;             //定义最上前的偏差值
    float Kp,Ki,Kd;             //定义比例、积分、微分系数
}pid;

void PID_init(){
    pid.SetSpeed=0.0;
    pid.ActualSpeed=0.0;
    pid.err=0.0;
    pid.err_last=0.0;
    pid.err_next=0.0;
    pid.Kp=0.2;
    pid.Ki=0.015;
    pid.Kd=0.2;
}

float PID_realize(float speed){
```

```
    pid.SetSpeed=speed;
    pid.err=pid.SetSpeed-pid.ActualSpeed;
    float
incrementSpeed=pid.Kp*(pid.err-pid.err_next)+pid.Ki*pid.err+pid.Kd*(pid.err-2
*pid.err_next+pid.err_last);
    pid.ActualSpeed+=incrementSpeed;
    pid.err_last=pid.err_next;
    pid.err_next=pid.err;
    return pid.ActualSpeed;
}

int main(){
    PID_init();
    int count=0;
    while(count<1000)
    {
        float speed=PID_realize(200.0);
        printf("%f\n",speed);
        count++;
    }
    return 0;
}
```

3.积分分离的　PID 控制算法

在普通 PID 控制中，引入积分环节的目的，主要是为了消除静差，提高控制精度。但是在启动、 结束或大幅度增减设定时， 短时间内系统输出有很大的偏差，会造成 PID 运算的积分积累，导致控制量超过执行机构可能允许的最大动作范围对应极限控制量， 从而引起较大的超调， 甚至是震荡，这是绝对不允许的。

为了克服这一问题，引入了积分分离的概念，其基本思路是　　当被控量与设定值偏差较大时，取消积分作用 ；当被控量接近给定值时，引入积分控制，以消除静差，提高精度。其具体实现代码如下：

```
pid.Kp=0.2;
    pid.Ki=0.04;
    pid.Kd=0.2;    //初始化过程
if(abs(pid.err)>200)
    {
    index=0;
    }else{
    index=1;
    pid.integral+=pid.err;
    }
    pid.voltage=pid.Kp*pid.err+index*pid.Ki*pid.integral+pid.Kd*(pid.err-pid.err
_last);     //算法具体实现过程
```

4.抗积分饱和的 PID 控制算法 C 语言实现

所谓的积分饱和现象是指如果系统存在一个方向的偏差，PID 控制器的输出由于积分作用的不断累加而加大，从而导致执行机构达到极限位置，若控制器输出 U(k) 继续增大，执行器开度不可能再增大，此时计算机输出控制量超出了正常运行范围而进入饱和区。一旦系统出现反向偏差，u(k) 逐渐从饱和区退出。进入饱和区越深则退出饱和区时间越长。在这段时间里，执行机构仍然停留在极限位置而不随偏差反向而立即做出相应的改变，这时系统就像失控一样，造成控制性能恶化，这种现象称为积分饱和现象或积分失控现象。

防止积分饱和的方法之一就是抗积分饱和法，该方法的思路是在计算 u(k) 时，首先判断上一时刻的控制量 u(k-1) 是否已经超出了极限范围：如果 u(k-1)>umax，则只累加负偏差；如果 u(k-1)<umin，则只累加正偏差。从而避免控制量长时间停留在饱和区。

```
struct t_pid{
    float SetSpeed;             //定义设定值
    float ActualSpeed;          //定义实际值
```

```
float err;                  //定义偏差值
float err_last;              //定义上一个偏差值
float Kp,Ki,Kd;              //定义比例、积分、微分系数
float voltage;               //定义电压值（控制执行器的变量）
float integral;              //定义积分值
float umax;
float umin;
}pid;

void PID_init(){
    pid.SetSpeed=0.0;
    pid.ActualSpeed=0.0;
    pid.err=0.0;
    pid.err_last=0.0;
    pid.voltage=0.0;
    pid.integral=0.0;
    pid.Kp=0.2;
   pid.Ki=0.1;          //注意，和上几次相比，这里加大了积分环节的值
    pid.Kd=0.2;
    pid.umax=400;
    pid.umin=-200;
}
float PID_realize(float speed){
    int index;
    pid.SetSpeed=speed;
    pid.err=pid.SetSpeed-pid.ActualSpeed;
  if(pid.ActualSpeed>pid.umax)      //灰色底色表示抗积分饱和的实现
    {
      if(abs(pid.err)>200)          //蓝色标注为积分分离过程
{
    index=0;
}else{
    index=1;
    if(pid.err<0)
    {
    pid.integral+=pid.err;
```

```
        }
    }
}else if(pid.ActualSpeed<pid.umin){
    if(abs(pid.err)>200)        //积分分离过程
    {
        index=0;
    }else{
        index=1;
        if(pid.err>0)
        {
        pid.integral+=pid.err;
        }
    }
}else{
    if(abs(pid.err)>200)                    //积分分离过程
    {
        index=0;
    }else{
        index=1;
        pid.integral+=pid.err;
        }
    }
    pid.voltage=pid.Kp*pid.err+index*pid.Ki*pid.integral+pid.Kd*(pid.err-pid.err
_last);
    pid.err_last=pid.err;
    pid.ActualSpeed=pid.voltage*1.0;
    return pid.ActualSpeed;
}
```

5.梯形积分的　PID 控制算法

先看一下梯形算法的积分环节公式

$$\int_0^T e_{(t)} d_t = \sum_{i=0}^{k} \frac{e_{(i)} + e_{(i-1)}}{2} . T$$

作为 PID 控制律的积分项，其作用是消除余差，为了尽量减小余差，应提高积分项运算精度，为此可以将矩形积分改为梯形积分，具体实现的语句为：

```
pid.voltage=pid.Kp*pid.err+index*pid.Ki*pid.integral/2+pid.Kd*(pid.err-pid.err_last);
//梯形积分
```

6.变积分的 PID 控制算法

变积分 PID 可以看成是积分分离的 PID 算法的更一般的形式。在普通的 PID 控制算法中，由于积分系数是常数，所以在整个控制过程中，积分增量是不变的。但是，系统对于积分项的要求是，系统偏差大时，积分作用应该减弱甚至是全无，而在偏差小时，则应该加强。积分系数取大了会产生超调，甚至积分饱和，取小了又不能短时间内消除静差。因此，根据系统的偏差大小改变积分速度是有必要的。

变积分 PID 的基本思想是设法改变积分项的累加速度，使其与偏差大小相对应：偏差越大，积分越慢；偏差越小，积分越快。

这里给积分系数前加上一个比例值 index：

当 abs(err)<180 时，index=1;

当 180<abs(err)<200 时，index=（200-abs(err)）/20;

当 abs(err)>200 时，index=0;

最终的比例环节的比例系数值为 ki*index;

具体 PID 实现代码如下：

```
pid.Kp=0.4;
pid.Ki=0.2;      //增加了积分系数
pid.Kd=0.2;
float PID_realize(float speed){
 float index;
 pid.SetSpeed=speed;
```

```
pid.err=pid.SetSpeed-pid.ActualSpeed;
if(abs(pid.err)>200)                    //变积分过程
{
        index=0.0;
}
  else if(abs(pid.err)<180)
  {
        index=1.0;
        pid.integral+=pid.err;
}
  Else
  {
        index=(200-abs(pid.err))/20;
        pid.integral+=pid.err;
}
pid.voltage=pid.Kp*pid.err+index*pid.Ki*pid.integral+pid.Kd*(pid.err-pid.err_last);
pid.err_last=pid.err;
pid.ActualSpeed=pid.voltage*1.0;
    return pid.ActualSpeed;
}
```

7.专家 PID 和模糊 PID

从前面的讲解中不难看出， PID 的控制思想非常简单， 其主要问题点和难点在于比例、 积分、微分环节上的参数整定过程， 对于执行器控制模型确定或者控制模型简单的系统而言， 参数的整定可以通过计算获得， 对于一般精度要求不是很高的执行器系统，可以采用拼凑的方法进行实验型的整定。

然而，在实际的控制系统中， 线性系统毕竟是少数， 大部分的系统属于非线性系统，或者说是系统模型不确定的系统， 如果控制精度要求较高的话， 那么对于参数的整定过程是有难度的。专家 PID 和模糊 PID 就是为满足这方面的需求而设计的。专家算法和模糊算法都归属于智能算法的范畴， 智能算法最大的优点就是在控制模型未知的情况下， 可以对模型进行控制。 这里需要注意的是， 专家

PID 也好，模糊 PID 也罢，绝对不是专家系统或模糊算法与 PID 控制算法的简单加和，他是专家系统或者模糊算法在 PID 控制器参数整定上的应用。也就是说，智能算法是辅助 PID 进行参数整定的手段。

专家系统、模糊算法，需要参数整定就一定要有整定的依据，也就是说什么情况下整定什么值是要有依据的，这个依据是一些逻辑的组合，只要找出其中的逻辑组合关系来，这些依据就再明显不过了。下面先说一下专家 PID 的 C 语言实现。正如前面所说，需要找到一些依据，还得从 PID 系数本身说起。

(1).比例系数 K_p 的作用是加快系统的响应速度，提高系统的调节精度。K_p 越大，系统的响应速度越快，系统的调节精度越高，但是容易产生超调，甚至会使系统不稳定。K_p 取值过小，则会降低调节精度，使响应速度缓慢，从而延长调节时间，是系统静态、动态特性变差；

(2).积分作用系数 K_i 的作用是消除系统的稳态误差。K_i 越大，系统的静态误差消除的越快，但是 K_i 过大，在响应过程的初期会产生积分饱和的现象，从而引起响应过程的较大超调。若 K_i 过小，将使系统静态误差难以消除，影响系统的调节精度；

(3).微分系数 K_d 的作用是改善系统的动态特性，其作用主要是在响应过程中抑制偏差向任何方向的变化，对偏差变化进行提前预报。但是 K_d 过大，会使响应过程提前制动，从而延长调节时间，而且会降低系统的抗干扰性。

反应系统性能的两个参数是系统误差 e 和误差变化律 e_c，这点还是好理解的：

首先我们规定一个误差的极限值，假设为 Mmax ；规定一个误差的比较大的值，假设为 Mmid ；规定一个误差的较小值，假设为 Mmin ；

当 abs（e）>Mmax 时，说明误差的绝对值已经很大了，不论误差变化趋势如何，都应该考虑控制器的输入应按最大（或最小）输出，以达到迅速调整误差的效果，使误差绝对值以最大的速度减小。此时，相当于实施开环控制。

当 e*ec>0 时，说明误差在朝向误差绝对值增大的方向变化，此时，如果 abs(e)>Mmid ，说明误差也较大，可考虑由控制器实施较强的控制作用，以达到扭转误差绝对值向减小的方向变化，并迅速减小误差的绝对值。此时如果 abs(e)<Mmid ，说明尽管误差是向绝对值增大的方向变化， 但是误差绝对值本身并不是很大，可以考虑控制器实施一般的控制作用， 只需要扭转误差的变化趋势，使其向误差绝对值减小的方向变化即可。

当 e*err<0 且 e*err(k-1)>0 或者 e=0 时，说明误差的绝对值向减小的方向变化，或者已经达到平衡状态，此时保持控制器输出不变即可。

当 e*err<0 且 e*err(k-1)<0 时，说明误差处于极限状态。如果此时误差的绝对值较大，大于 Mmin , 可以考虑实施较强控制作用。 如果此时误差绝对值较小，可以考虑实施较弱控制作用。

当 abs(e)<Mmin 时，说明误差绝对值很小， 此时加入积分， 减小静态误差。

上面的逻辑判断过程，实际上就是对于控制系统的一个专家判断过程。

实际上模糊算法属于智能算法， 智能算法也可以叫非模型算法， 智能算法包含了专家系统、模糊算法、遗传算法、神经网络算法等。其实这其中的任何一种算法都可以跟 PID 去做结合，而选择的关键在于，处理的实时性能不能得到满足。当我们处理器的速度足够快速时， 我们可以选择更为复杂的、 精度更加高的算法。但是，控制器的处理速度限制了我们算法的选择。当然，成本是限制处理器速度最根本的原因。模糊 PID 适应一般的控制系统是没有问题。

模糊算法其实并不模糊。 模糊算法其实也是逐次求精的过程。 这里举个例子说明。我们设计一个倒立摆系统， 假如摆针偏差＜ 5°, 我们说它的偏差比较 “小 ”; 摆针偏差在 5°和 10°之间，我们说它的偏差处于 “中 ”的状态；当摆针偏差＞ 10°的时候，我们说它的偏差有点儿 “大 ”了。对于 “小 ”、 “中 ”、 “大 ”这样的词汇来讲，他们是精确的表述，可问题是如果摆针偏差是 3°呢，那么这是一种什么样的状态呢。我们可以用 “很小 ”来表述它。如果是 7°呢，可以说它是 “中 ”偏 “小 ”。那么如果到了 80°呢，它的偏差可以说 “非常大 ”。而我们调节的过程实际上就是让系统的偏差由非常 “大 ”逐渐向非常 “小 ”过度的过程。当然，我们系统这个调节过程是快速稳定的。 通过上面的说明， 可以认识到， 其实对于每一种状态都可以划分到大、中、小三个状态当中去，只不过他们隶属的程度不太一样，比如 6°隶属于小的程度可能是 0.3，隶属于中的程度是 0.7，隶属于大的程度是 0。这里实际上是有一个问题的， 就是这个隶属的程度怎么确定？这就要求我们去设计一个隶属函数。详细内容可以查阅相关的资料， 这里没有办法那么详细的说明了。 那么，知道了隶属度的问题， 就可以根据目前隶属的程度来控制电机以多大的速度和方向转动了， 当然，最终的控制量肯定要落实在控制电压上。 这点可以很容易的想想，我们控制的目的就是让倒立摆从隶属 “大 ”的程度为 1 的状态，调节到隶属 “小 ”的程度为 1 的状态。当隶属大多一些的时候， 我们就加快调节的速度， 当隶属小多一些的时候，我们就减慢调节的速度，进行微调。可问题是，大、中、小的状态是汉字，怎么用数字表示，进而用程序代码表示呢？其实我们可以给大、中、小三个状态设定三个数字来表示，比如大表示用 3 表示，中用 2 表示，小用 1 表示。那么我们完全可以用 1*0.3+2*0.7+3*0.0=1.7 来表示它，当然这个公式也不一定是这样的， 这个公式的设计是系统模糊化和精确化的一个过程， 读者也可参见相关文献理解。 但就 1.7 这个数字而言， 可以说明， 目前 6°的角度偏差处于小和中之间， 但是更偏向于中。 我们就可以根据这个数字来调节电机的转动速度和时间了。 当然，这个数字与电机转速的对应关系， 也需要根据实际情况进行设计和调节。

前面一个例子已经基本上说明了模糊算法的基本原理了。 可是实际上， 一个系统的限制因素常常不是一个。 上面的例子中， 只有偏差角度成为了系统调节的

参考因素。而实际系统中，比如 PID 系统，我们需要调节的是比例、积分、微分三个环节， 那么这三个环节的作用就需要我们认清， 也就是说， 我们需要根据超调量、 调节时间、 震荡情况等信息来考虑对这三个环节调节的比重， 输入量和输出量都不是单一的， 可是其中必然有某种内在的逻辑联系。 所以这种逻辑联系就成为我们设计工作的重点了。

四、 PID 算法参数整定方法

1.临界比例度法

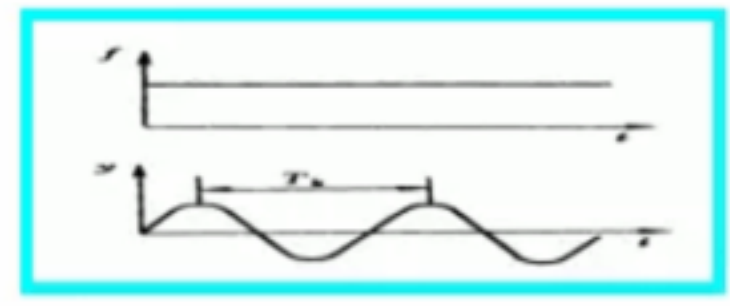
（1）将调节器的积分时间置于最大，微分时间置零，比例度 δ适当，平衡操作一段时间，把系统投入自动运行。

（2）将比例度 δ逐渐减小，得到等幅振荡过程，记下临界比例度 δ_k 和临界振荡周期 T_k 值。

（3）根据和值，采用经验公式，计算出调节器各个参数，即 δ、、的值。

（4）按 ‘先 P 后 I 最后 D”的操作程序将调节器整定参数调到计算值上。若还不够满意，可再作进一步调整。

参数		(min)	(min)
P	2		
P/I	2.2		
P/I/D	1.6	0.5	0.25



临界振荡整定计算公式

2.衰减曲线法

在纯比例作用下， 由大到小调整比例度以得到具有衰减比 （4：1）的过渡过程，记下此时的比例度及振荡周期， 根据经验公式， 求出相应的积分时间和微分时间。

参数		(min)	(min)
P			
P/I	1.2	0.5	
P/I/D	0.8	0.3	0.1

衰减曲线法控制器参数计算表

Note:(1). 反应较快的控制系统，要认定　4：1 衰减曲线和读出　Ts 比较困难，此时，可用记录来回摆动两次就达到稳定作为　4：1 衰减过程。 (2). 在生产过程中，负荷变化会影响过程特性。 当负荷变化较大时， 必须重新整定调节器参数值。 (3). 若认为 4：1 衰减太慢，宜应用　10：1 衰减过程。对于　10：1 衰减曲线法整定调节器参数的步骤与上述完全相同，仅仅采用计算公式有些不同。

3.经验法

根据经验先将控制器参数放在某些数值上，　直接在闭合的控制系统中通过改变给定值以施加干扰，看输出曲线的形状，以　δ、、，对控制过程的规律为指导，调整相应的参数进行凑试，直到合适为止。

系统	参数		
	δ (%)	T_I ( )	T_D ( )
温度	20～60	3 ～10	0.5～3
流量	40～100	0.1～1	
压力	30～70	0.4～3	
液位	20～80		

长期的生产实践中总结出来的参数表

4.经验口诀

参数整定找最佳，从小到大顺序查。
先是比例后积分，最后再把微分加。
曲线振荡很频繁，比例度盘要放大。
曲线漂浮绕大湾，比例度盘往小扳。
曲线偏离回复慢，积分时间往下降。
曲线波动周期长，积分时间再加长。
曲线振荡频率快，先把微分降下来。

动差大来波动慢。微分时间应加长。
理想曲线两个波，前高后低　4 比 1。
一看二调多分析，调节质量不会低。

原文转自：
http://mp.weixin.qq.com/s?__biz=MjM5NDQ0NjM5Mg==&mid=2650417333&idx=1&sn=41d9ce4c7e77cff2ab742144a62be7ff&chksm=be8925e289feacf4856d84c7dcb8bd64a1123e42a8b33e6fa94e797aa1e4f74c59f9941e53b3&mpshare=1&scene=23&srcid=0224KxRA7kYg2T65IAL0aMvW#rd