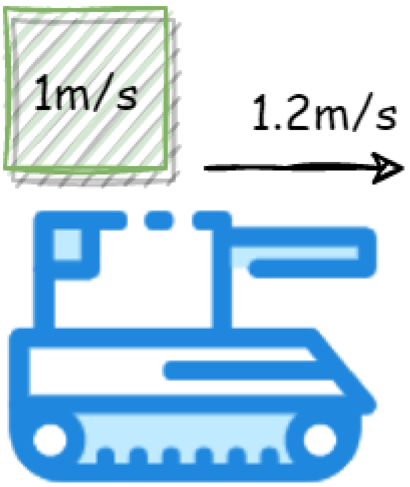
#1 控制算法——PID(入门级)

引入

在用电机驱动步兵车向前运动的时候，我们想使得它速度为 1m/s ，但实际可能有 1.2m/s.



那么为了使得它始终能都稳定在1m/s,而不随着路面的情况变化而速度不稳定，我们最简单的想法就是——车的速度快了就告诉电机慢一点，车慢了就告诉电机快一点,这其实就是 PID 的雏形了。

但上面这个想法存在许多问题。

- 如果车慢了，告诉电机快多少？
- 调整总时间如何？如果需要几十秒调节，那会对其他控制模块和操作带来巨大的不便。
- 算法没有通用性，只能针对部分场景。

为了解决上述问题，前人就总结了 PID 这种通用又好用的控制算法.

PID 描述

PID(proportion integration differentiation)，是指比例，积分，微分三个模块去调节。

$$I(t) = k_p err(t) + \frac{K_i}{T_i}\int err(t)dt + \frac{T_d derr(t)}{dt} \tag{2.1}$$

其中 $err(t)$ 是当前时刻的测量值和目标值之间的差, $derr(t)$ 是对 $err(t)$ 的微分, $I(t)$ 代表的是任何可以正向控制的量。举个例子，比如我们想让步兵的车速稳定的在 1m/s

我们一步步从“有什么用”的角度来来谈谈 PID 为什么是 (2.1) 公式表示的形式

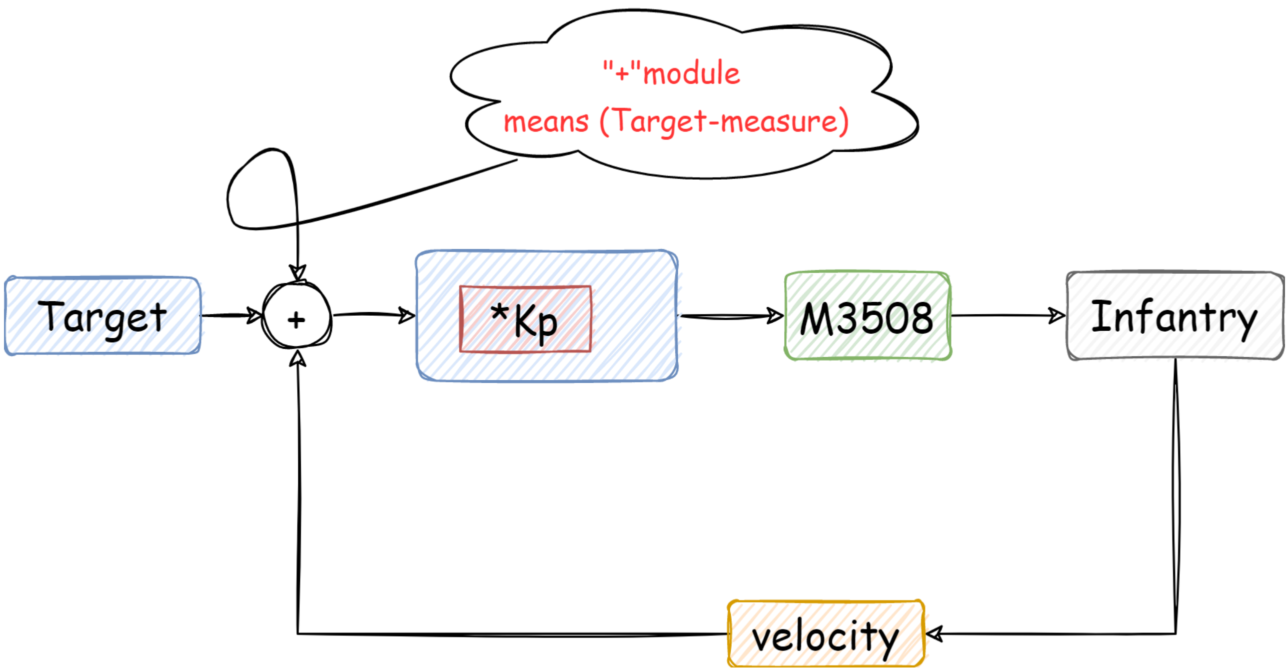
这里为了更加明了，我们设置一个具体的场景：

现在 A 板(我们的控制板)可以控制步兵的电机(型号 M3508)，电机可以带动步兵向前走，同时我们有一个测量速度的传感器可以换算出步兵的速度。我们要做的就是用 A 板控制电源给予 M3508 的电流大小，从而调节电机转速，使得步兵整体运行在我们的期望的速度上。

首先我们用最简单的思路，也就是给予电机的电流 $I(t)$：

$$I(t) = k_p(v_{target} - v_{meature}(t)) \tag{2.2}$$

$v_{target}, v_{meature}(t)$ 分别是目标速度和测量速度 ,k_p 是一个比例系数(单位 As/m)



如果我们将 $v_{target} - v_{meature}(t)$ 记为 $err(t)$ 那么就有如下式子

$$I(t) = k_p err(t) \tag{2.3}$$

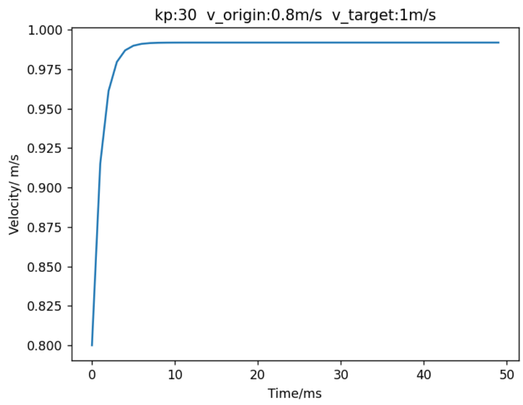
```
def pid1(kp,slop = 30,v_origin = 0.8,v_target = 1):
    dt = 0.001#模拟时间步长为 0.01s
    weight = 5 #假设步兵重 5kg
    V_list = np.arange(50,dtype = 'double')
    T_list = np.arange(50)
    v = v_origin
    g = 9.8 #重力加速度
    i2f = 100 #电流和电机力之间的转换系数
    V_list[0] = v
    T_list[0] = 0.0
    for i in range(1,50):
        v = v-dt*g*math.sin(math.radians(slop))+dt*kp*(v_target-
v)*i2f/weight
        V_list[i] = v
        T_list[i] = i
    #plt.ylim(0.7,1.05)
    plt.xlabel('Time/ms')
    plt.ylabel('Velocity/ m/s')
    plt.title('kp:10  v_origin:0.8m/s  v_target:1m/s')
    plt.plot(T_list,V_list)
    plt.show()
```

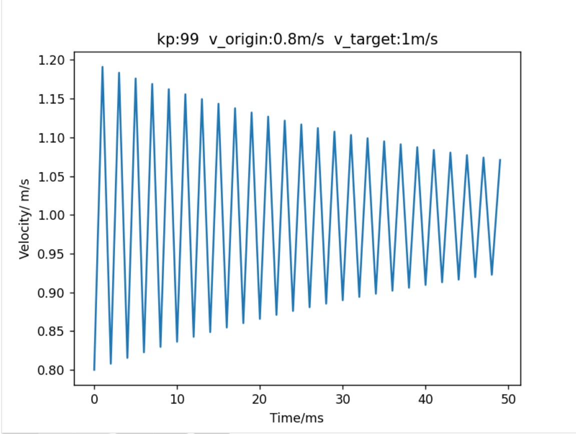
事实上，这部分就是 PID 的最基础也是最重要的部分，有了这一项也能胜任主要的控制工作.

那么我们考虑一个问题，如果步兵遇到要上坡这类的情况：比如我们还是希望车速为 1m/s，但是上坡后车速只有 0.5m/s ,但是这时控制算法输出只有 $I(t) = k_p(1m/s - 0.5m/s)$, k_p 如之前所说是常数，所以这里可能电流输出还是没法使得步兵在上坡的时候到达 1m/s 甚至可能还会倒溜.

那你可能还会问那把 k_p 设置的足够大不就好了，那么就得保障 k_p 足够大，大到能够适应所有可能的坡度，甚至到近乎垂直的角度，这会有什么问题呢？

我们来看看不同的:





上方左图展示的是 $k_p = 30$ 的情况，可以观察到它收敛速度不错，但是无法到达我们给定的 v_{target} = 1m/s 的要求.而右图展示的是 k_p = 99 的情况，可以观察到它虽然最终会趋向目标速度，但是收敛起来实在太慢，而且变化速度非常快，步兵是无法实现这样的过程的.

这就会有一个问题，取的k_p 太小虽然稳定，但是没法达到我们的目标速度; k_p 太大虽然可以达到目标速度，但是又不稳定,而且 k_p 大意味着需要更大的加速度,这对电机的性能提出了不小的要求.

那么如何解决这个问题？我们再接着深入思考一下如果改善 k_p 小带来的输出能力不够的问题.其实如果我们的控制算法中，还能添加一个功能“当步兵速度长时间达不到目标速度时，就额外加大输出电流 $I(t)$”，这时候初学者可能会立刻联系到 C 语言课程中 if else 条件

```
if(测量速度长时间达不到目标速度){
    //额外加大 I(t)
}else{
    //正常控制
}
```

这虽然是常见的方法，但在目前的这个情景中，实施起来会有很多问题： if else 实现这里可能会导致在 if 条件满足和不满的临界状态疯狂跳转，步兵会出现“一顿一顿”的感觉，这个问题出现的本质原因就是 if else 离散程度太大。当然，又会问，那多套几个if else 不就好了，但这其实和接下来要介绍的方式是类似的，不过接下来的算法会更加简洁.

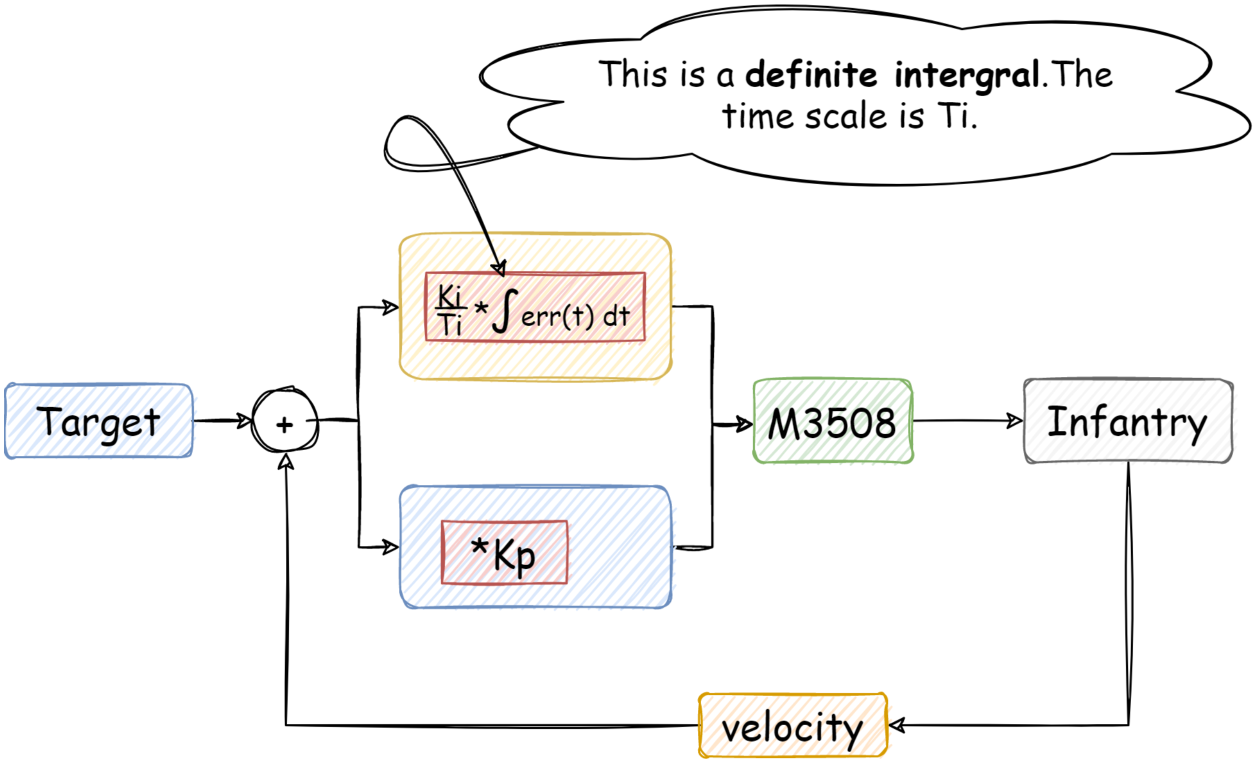
前面的需求是 “长时间达不到目标速度则额外增大 $I(t)$ ” 这说明我们需要一个记忆单元，而积分正是解决这个需求的好办法.

其实 PID 的积分项 (proportion ***integration*** differentiation)作用就是如此.

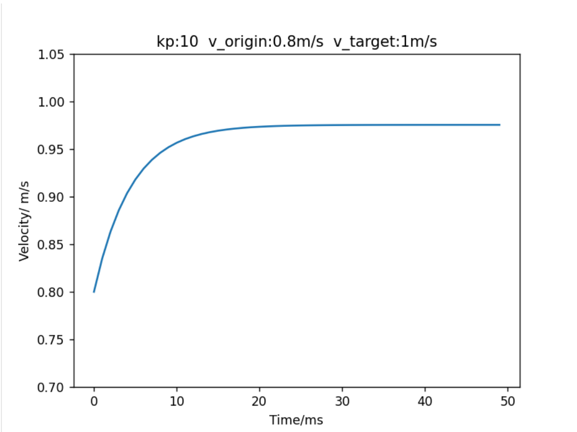
那么我们就可以写出带有积分项的控制算法

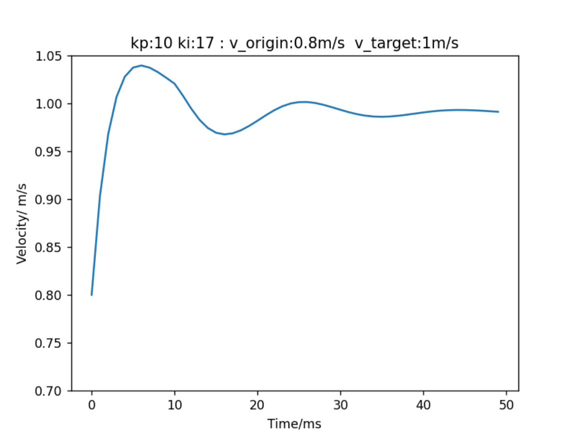
$$I(t) = K_p err(t) + \frac{K_i}{T_i} \int err(t)dt \tag{2.4}$$

```python
def pid2(kp,ki,slop = 30,v_origin = 0.8,v_target = 1.0):
    dt = 0.001
    weight = 5
    Num = 50
    E_num = 0
    E_num_max = 10
    E_sum = 0
    V_list = np.arange(Num,dtype = 'double')
    E_list =  np.zeros(1,dtype = 'double')
    T_list = np.arange(Num)
    v = v_origin
    g = 9.8 #重力加速度
    i2f = 100 #电流和电机力之间的转换系数
    V_list[0] = v
    T_list[0] = 0.0
    for i in range(1,50):
        if E_num < E_num_max:
             E_num = E_num+1
        E_list = np.insert(E_list,0,v_target-V_list[i-1])
        if np.size(E_list)<E_num_max:
           E_sum = np.sum(E_list)
        else:
             E_sum = np.sum(E_list[0:E_num_max])
        v = v-dt*g*math.sin(math.radians(slop))+dt*(kp*(v_target-
v)+ki*E_sum/E_num)*i2f/weight
        V_list[i] = v
        T_list[i] = i
    #plt.ylim(0.7,1.05)
    plt.xlabel('Time/ms')
    plt.ylabel('Velocity/ m/s')
    plt.title('kp:10 ki:17 : v_origin:0.8m/s  v_target:1m/s')
    plt.plot(T_list,V_list)
    plt.show()
```



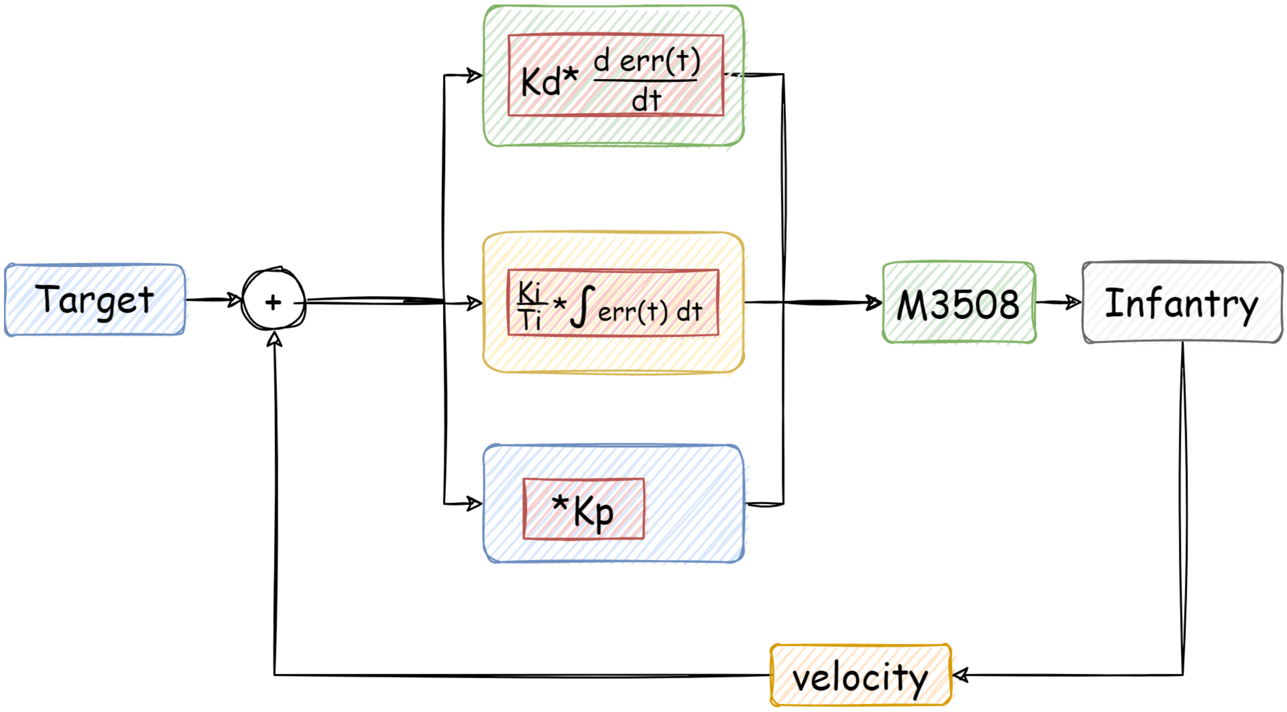
这会有什么效果呢？为了展示更加明显，我们将K_p 调整至 10 来，来做 (2.3)式和 (2.4)式的比较





左图是只有比例项的图，而右边是加了积分项的图，可以比较得到加上积分项之后不仅保留了比例项在 K_p 较小时稳定的优点，而且可以看到加了积分项之后更加接近目标速度.

但是积分项的加入又有另外一个小缺点，就是相比只有比例项变化的幅度变大了，那么解决这个问题就需要 PID 的微分项(proportion integration ***differentiation***)



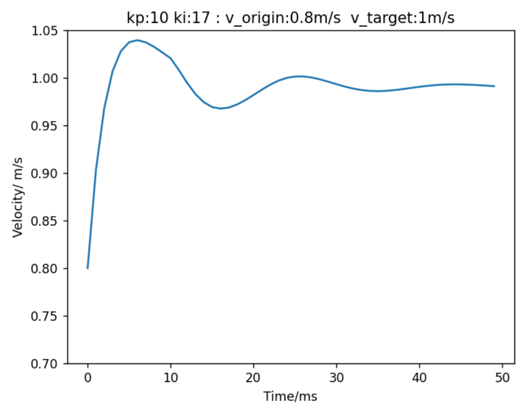
这就是完整的 PID 形式：

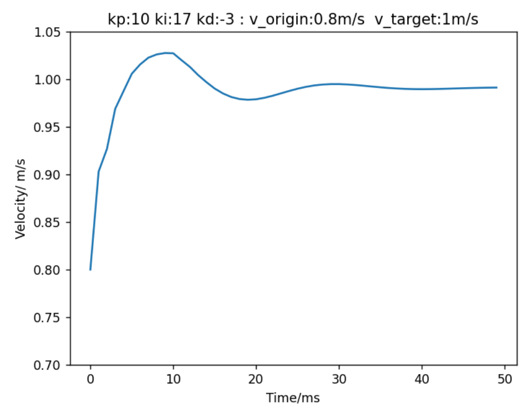
$$I(t) = k_p err(t) + \frac{K_i}{T_i} \int err(t) dt + \frac{T_d derr(t)}{dt} \tag{2.5}$$

```
def pid3(kp,ki,kd,slop = 30,v_origin = 0.8,v_target = 1.0):
    dt = 0.001
    weight = 5
    Num = 50
    E_num = 0
    E_num_max = 10
    E_sum = 0
    E_d = 0
    V_list = np.arange(Num,dtype = 'double')
    E_list =  np.zeros(1,dtype = 'double')
    T_list = np.arange(Num)
    v = v_origin
    g = 9.8 #重力加速度
    i2f = 100 #电流和电机力之间的转换系数
    V_list[0] = v
    T_list[0] = 0.0
    for i in range(1,50):
        if E_num < E_num_max:
            E_num = E_num+1
        E_list = np.insert(E_list,0,v_target-V_list[i-1])
        if np.size(E_list)<E_num_max:
```

```
            E_sum = np.sum(E_list)
        else:
            E_sum = np.sum(E_list[0:E_num_max])
        if i>1:
            E_d = V_list[i-1]-V_list[i-2]
        v = v-dt*g*math.sin(math.radians(slop))+dt*(kp*(v_target-v)+ki*E_sum/E_num+kd*E_d)*i2f/weight
        V_list[i] = v
        T_list[i] = i
    plt.ylim(0.7,1.05)
    plt.xlabel('Time/ms')
    plt.ylabel('Velocity/ m/s')
    plt.title('kp:27 ki:13 kd:-6 : v_origin:0.8m/s  v_target:1m/s')
    plt.plot(T_list,V_list)
    plt.show()
```

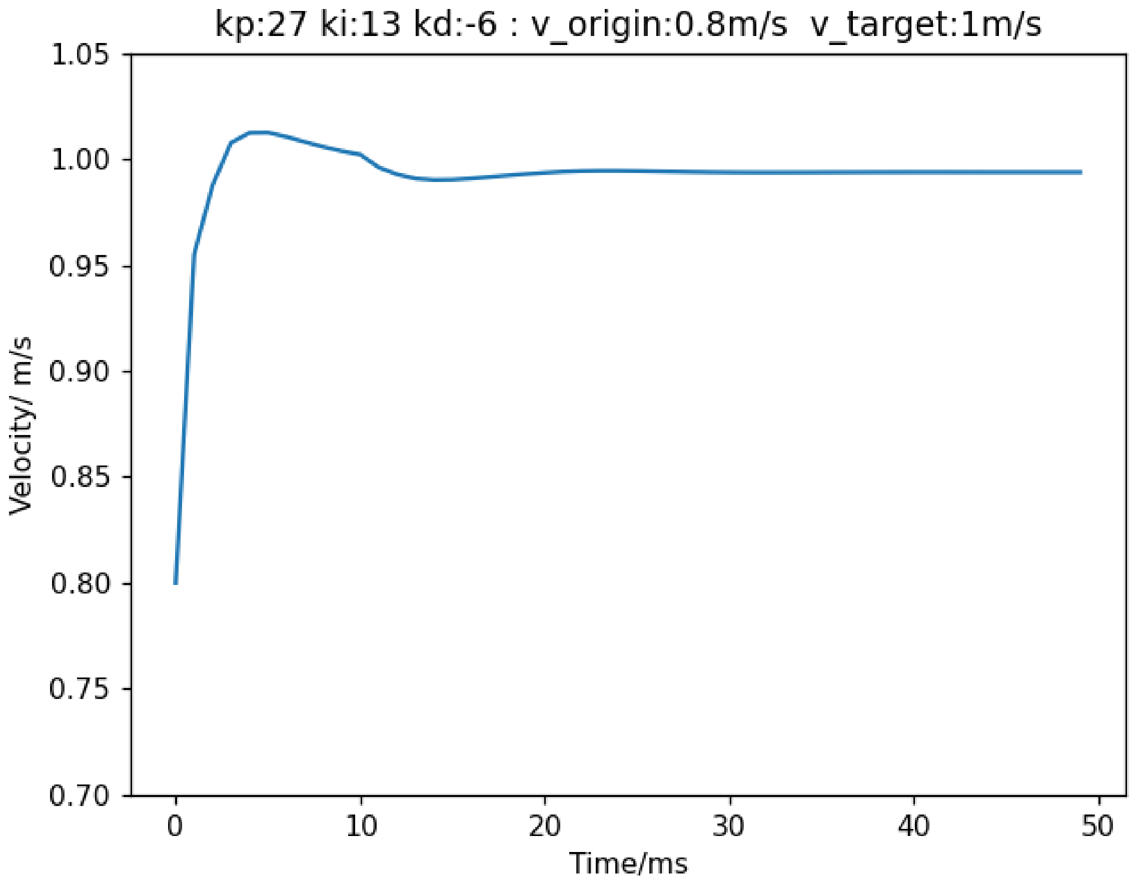
这得到的效果就是





左图是有积分项没有微分项的，右图是加入微分项的，加入微分项之后变化幅度明显变小了.

最后给出一个调参相对完善的 PID .



以上就是完整的 PID 介绍，但是K_p, K_i, Kd的常数调节还是需要不断尝试和调整. 所以经典的 PID 模型还是有许多不足. 但作为入门教程还是值得学习的. 后辈的工程师又针对不同的问题又提出了新的 PID 模型, 比如针对电机功率有限这一条件提出了限幅PID，针对调参麻烦这个问题又提出了神经元 PID 等等.