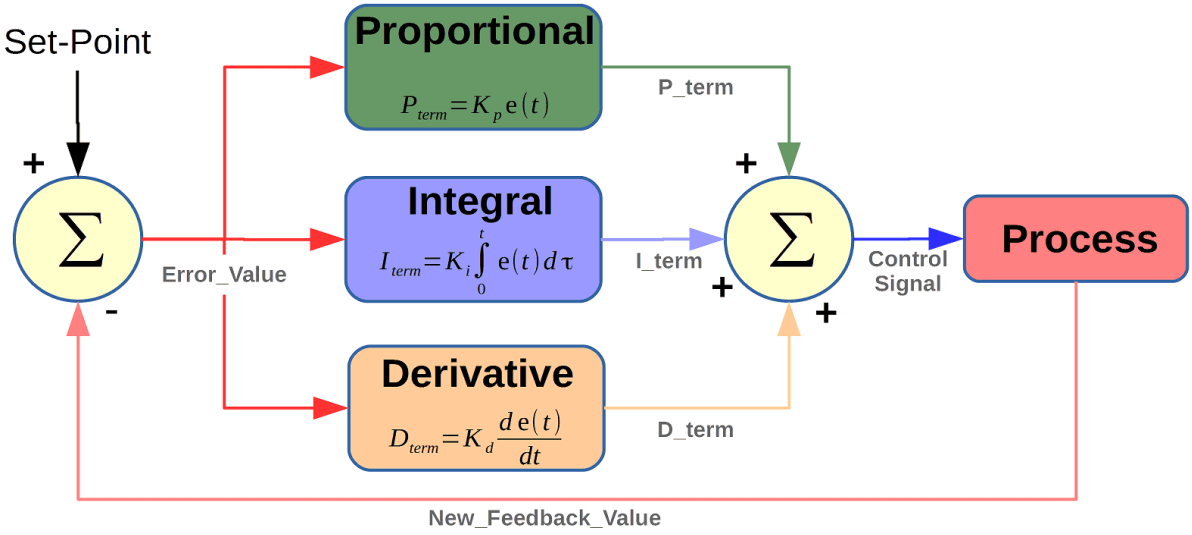
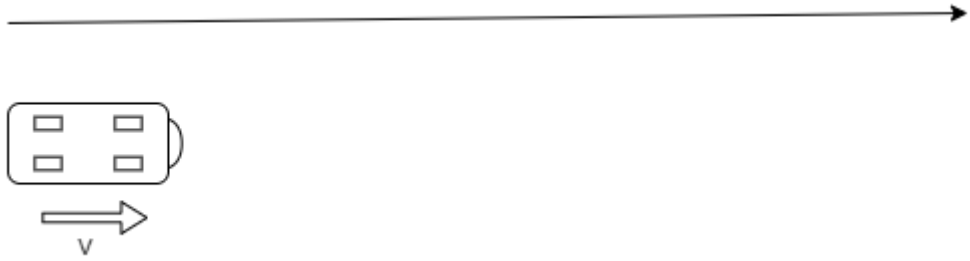
PID控制器

PID就是指 **比例（proportion）**、**积分（integral）**、**导数（derivative）**，这三项表示我们如何使用误差来产生控制指令，整个流程如下：



PID控制器是控制学中应用非常广泛的一个技术，让我们一起从0开始，实现并掌握PID控制器。



如上图，假如左边有一辆小汽车，车头向右。其前边两个轮子可以转向，后边两个轮子是固定的。我们希望让他沿着上边的直线，并按照箭头的方向前进。

假设其前进的速度恒定为 v，我们只可以控制其转向，那么如何完成我们的目标呢？

是使用一个固定的偏转角度呢，还是随机打方向，还是根据与目标路线的距离作为比例打方向呢？

接下来，我们学习如何通过PID相关的3个变量来进行控制。

环境准备

依赖安装：

确保Python的环境中包含以下依赖 `numpy`，`matplotlib`，如果没有，可以使用以下命令安装：

```
pip install numpy
```

```
pip install matplotlib
```

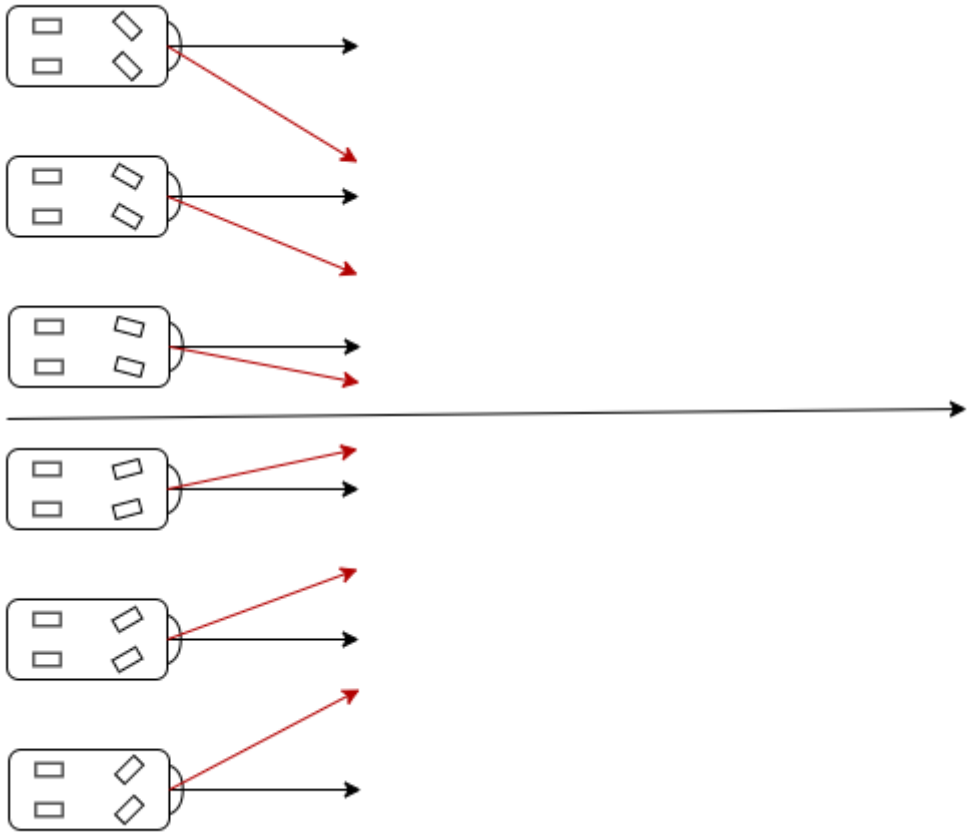
代码步骤：

1. 拷贝依赖文件 robot.py 到项目目录
2. 新建文件 `car_controller.py` 并编写如下内容：

```
# pid控制器 案例
from robot import Robot, show


def run(robot, n=100, speed=1.0):
    """
    运行多次运动并记录轨迹
    :param robot:    小车
    :param n:        循环次数
    :param speed:    小车速度
    """
    x_trajectory = []
    y_trajectory = []
    for i in range(n):
        # ---------------------------- start

        steer = 0

        # ---------------------------- end
        # 以steer为偏转角，speed为速度，执行一次运动
        robot.move(steer, speed)
        x_trajectory.append(robot.x)
        y_trajectory.append(robot.y)
        print(robot)
    return x_trajectory, y_trajectory


if __name__ == '__main__':
    robot = Robot()
    # 初始位置 x=0, y=-1, orient=0
    robot.set(0, -1, 0)

    # 运行并收集所有的x，y
    x_trajectory, y_trajectory = run(robot)

    # 可视化运行结果
    show(x_trajectory, y_trajectory, label='Car')
```

P 控制器

显而易见，我们应该根据**小车与目标路线的距离**来指定其前进的方向，即垂直距离的误差值，或称之为**横切误差（Cross Track Error）**简称**CTE**。我们用这个大小来决定前进的方向。偏差大的时候，我们偏转大的角度，当偏差小的时候，则偏转小一点的角度。



通过这个误差进行方向的修正就是P控制（Proportional），也即比例控制。

这时，问题来了，当我们使用一个系数 K_p 为比例进行方向修正的时候，其结果会是怎么样的？是会永远无法到达参考线还是会超过参考线?

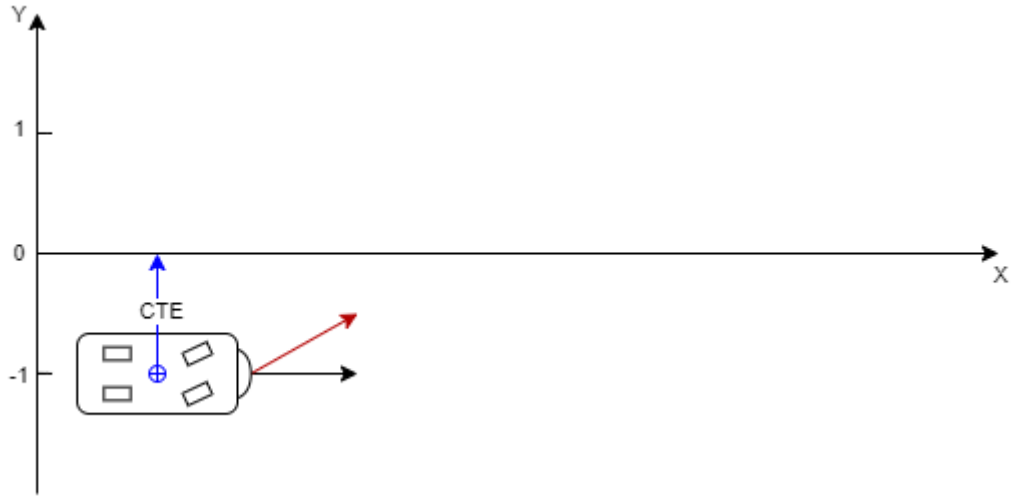
即角度修正的公式如下：

$$steer = K_p \cdot CTE$$

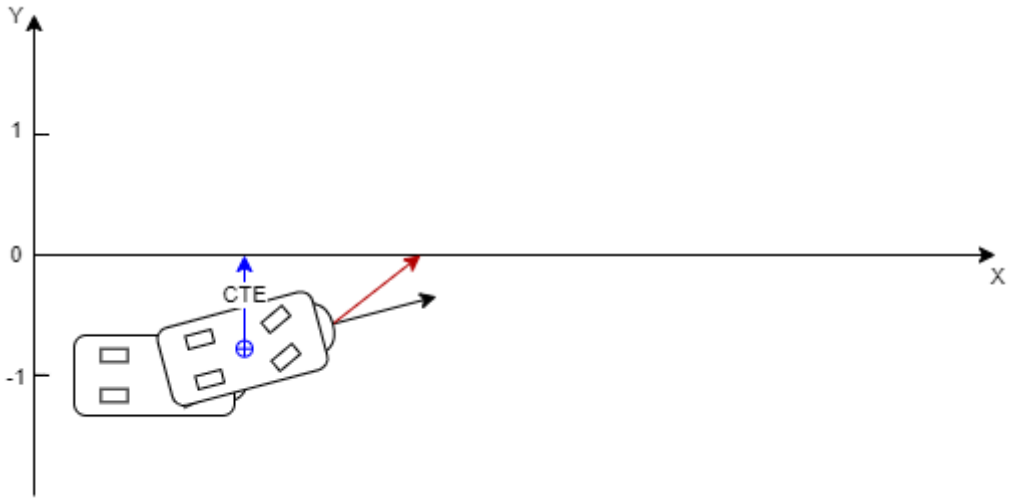
且 $CTE = target - current$

示例程序：

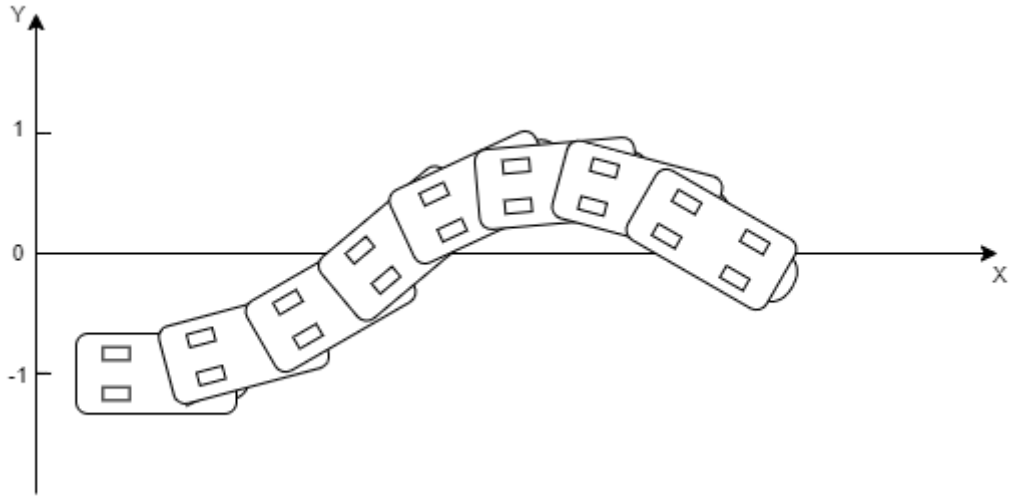
这里我们通过如下示例程序，对 P控制器 进行熟悉：



开始状态时，如上图，我们有一辆小车处于 $(0,-1)$的位置，我们希望其能够通过调整方向，沿着 $y=0$ 的 X 轴运动。此时可以认为其横切误差为 $CTE=0-y=1$ 表示（目标Y减去当前Y的差值），转向时，我们按照一定比例，例如此时设置系数 $K_p=0.1$，以作为P比例控制器的响应强度。



此时，如上图，由于方向的偏移量 $steer=K_p*CTE$ 为正值，则小车会进行轻微的逆时针偏转，进而缩短了下一次小车与 X 轴的距离。由于下一次的CTE减少了，则下一次的方向偏移量会相应的减少些。



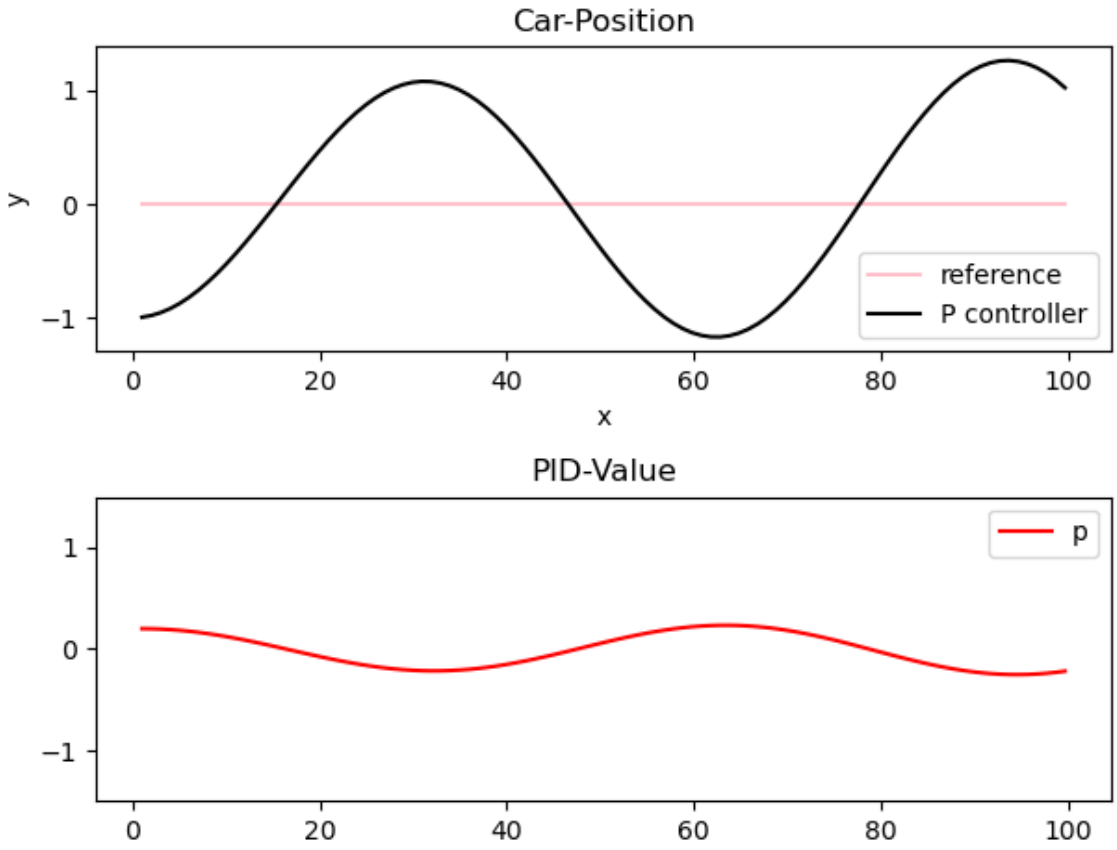
依次执行下去，在小车在越过X轴，处于Y正方向时，其 $steer$ 为负值，即小车会进行顺时针偏转，从而再次转向X轴的方向。这样，可以使小车在 X 轴上进行往复运动。

代码步骤：

1. 拷贝依赖文件 robot.py 到项目目录
2. 新建文件 `p_controller.py` 并编写如下内容：

```
# pid控制器 - P控制器
from robot import Robot, show

def run(robot, k_p, n=100, speed=1.0):
    """
    运行多次运动并记录轨迹
    :param robot:   小车
    :param k_p:     p系数
    :param n:       循环次数
    :param speed:   小车速度
    """
    x_trajectory = []
    y_trajectory = []
    p_arr = []   # 记录每一次的p
    for i in range(n):
        # ---------------------------- start
        cte = 0 - robot.y
        p = k_p * cte

        steer = p

        p_arr.append(p)
        # ---------------------------- end

        # 以steer为偏转角，speed为速度，执行一次运动
        robot.move(steer, speed)
        x_trajectory.append(robot.x)
        y_trajectory.append(robot.y)
        print(robot)
    return x_trajectory, y_trajectory, p_arr


if __name__ == '__main__':
    robot = Robot()
    # 初始位置 x=0, y=-1, orient=0
    robot.set(0, -1, 0)

    # 运行并收集所有的x，y，以及 p值
    x_trajectory, y_trajectory, p_arr = run(robot, k_p=0.1)

    # 可视化运行结果
    show(x_trajectory, y_trajectory, p_array=p_arr,  label='P')
```

运行结果：



建议自行修改 `k_p` 值，然后观察小车轨迹及日志的变化。

我们会发现：

- 当 K_p 为0时，对方向没有任何修正，即对结果没有任何影响
- 当 K_p 较大时，小车的偏转也会比较大，进而导致小车大幅的在目标轨迹上下摆动，摆动周期较短
- 当 K_p 较小时，小车的偏转也会比较小，小车在方向上的变化较小，要很久才能靠近目标轨迹，摆动周期较长

PD 控制器

接下来，我们希望使用一种方式，可以避免小车频繁越界，进而避免在目标轨迹上下来回往复。

因而，我们在原先的 P控制器 基础上，添加 D控制，我们称之为 **PD 控制器**，即我们用到的偏转角 `steer` 将额外加上 K_d 与 CTE对时间的导数的乘积。即：

$$steer = K_p \cdot CTE + K_d \cdot \frac{d}{d_t} CTE$$

CTE对时间的导数 $\frac{d}{d_t} CTE = \frac{CTE_t - CTE_{t-1}}{\Delta t}$ ，由于我们每次运动其都是单位时间 $\Delta t = 1$ ，结果即为当前误差和上一次误差的差值 $CTE_t - CTE_{t-1}$ 。

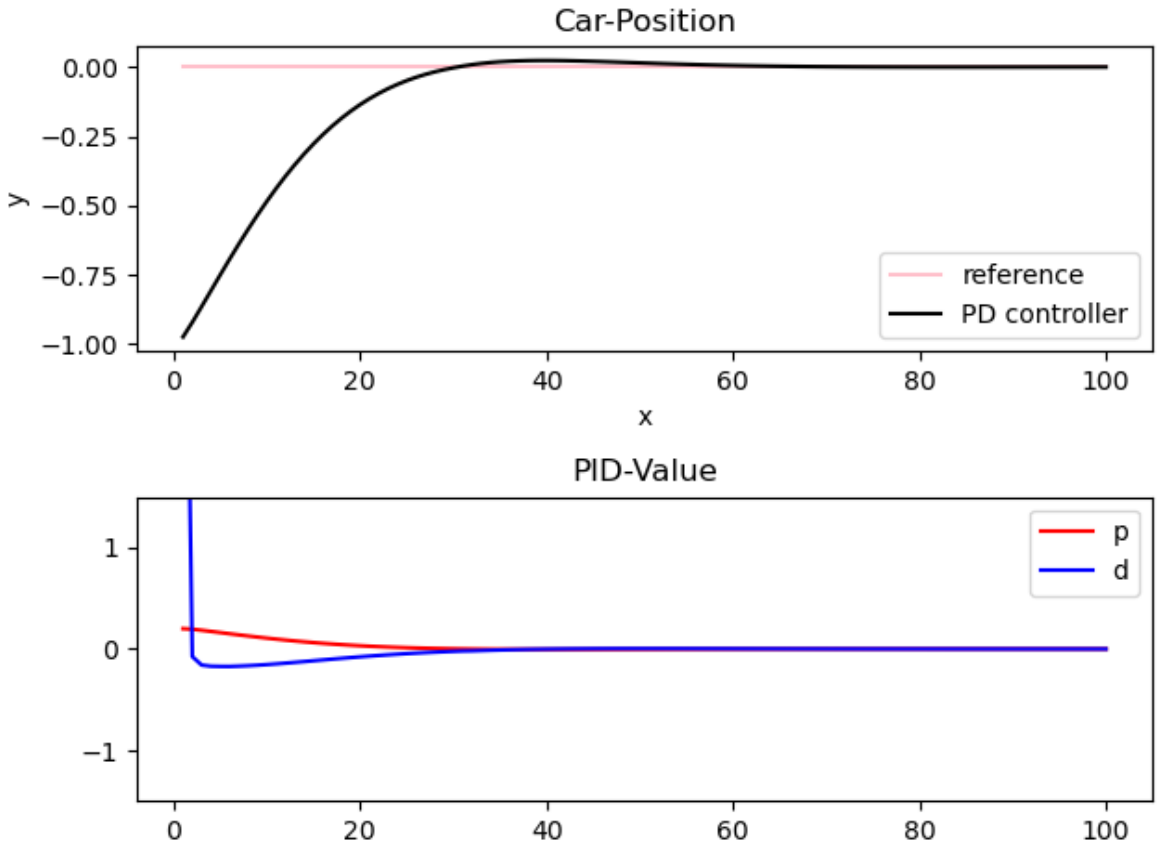
代码步骤：

1. 拷贝依赖文件 robot.py 到项目目录
2. 新建文件 `pd_controller.py` 并编写如下内容：

```
from robot import Robot, show
```

```
def run(robot, k_p, k_d, n=100, speed=1.0):
    """
    运行多次运动并记录轨迹
    :param robot:   小车
    :param k_p:     p系数
    :param k_d:     d系数
    :param n:       循环次数
    :param speed:   小车速度
    """
    x_trajectory = []
    y_trajectory = []
    p_arr = []  # 记录每一次的p
    d_arr = []  # 记录每一次的d

    prev_cte = 0 - robot.y
    for i in range(n):
        # ---------------------------- start
        cte = 0 - robot.y           # 目标 - 当前

        p = k_p * cte               # p

        d = k_d * (cte - prev_cte)  # d
        prev_cte = cte

        steer = p + d               # sum

        p_arr.append(p)
        d_arr.append(d)
        # ---------------------------- end

        # 以steer为偏转角，speed为速度，执行一次运动
        robot.move(steer, speed)
        x_trajectory.append(robot.x)
        y_trajectory.append(robot.y)
        print(robot)
    return x_trajectory, y_trajectory, p_arr, d_arr


if __name__ == '__main__':
    robot = Robot()
    # 初始位置 x=0, y=-1, orient=0
    robot.set(0, -1, 0)

    # 运行并收集所有的x，y，以及 p值
    x_trajectory, y_trajectory, p_arr, d_arr = run(robot, k_p=0.2, k_d=3.0)

    # 可视化运行结果
    show(x_trajectory, y_trajectory, p_array=p_arr, d_array=d_arr,  label =
'PD')
```

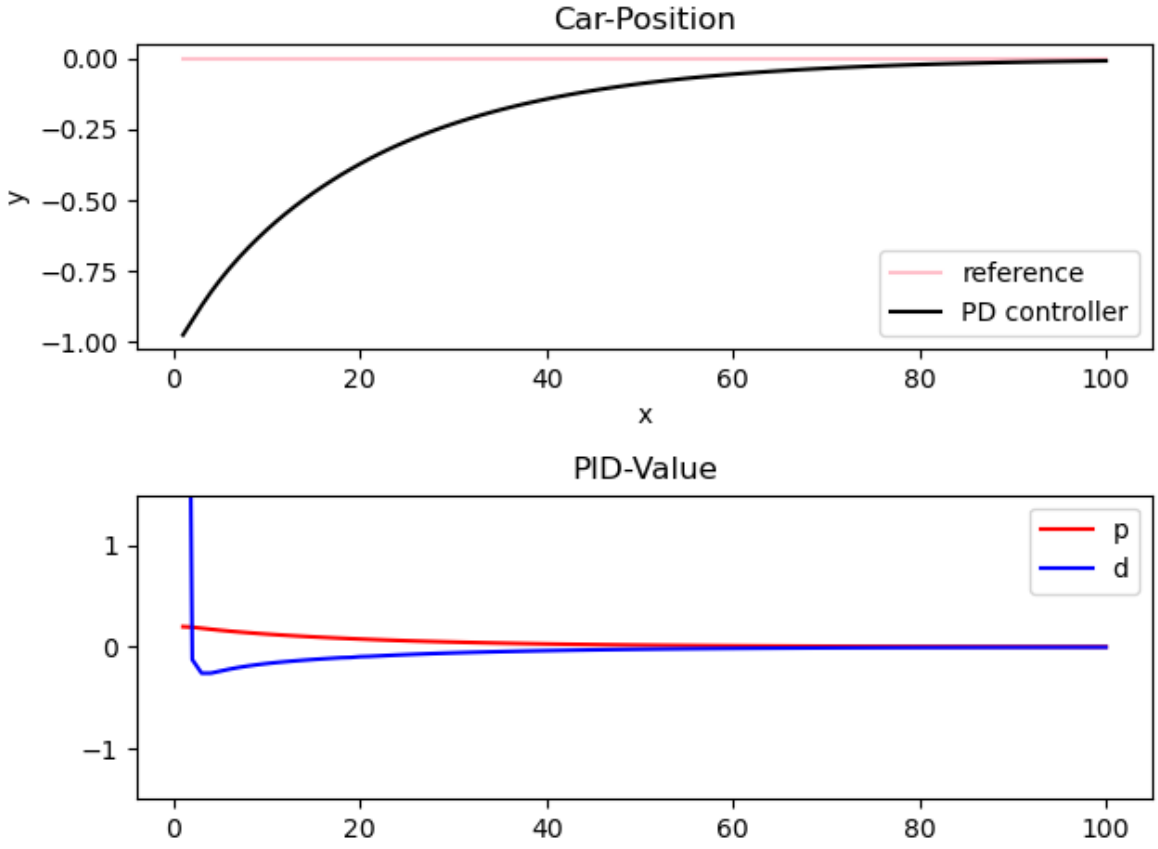
运行结果：



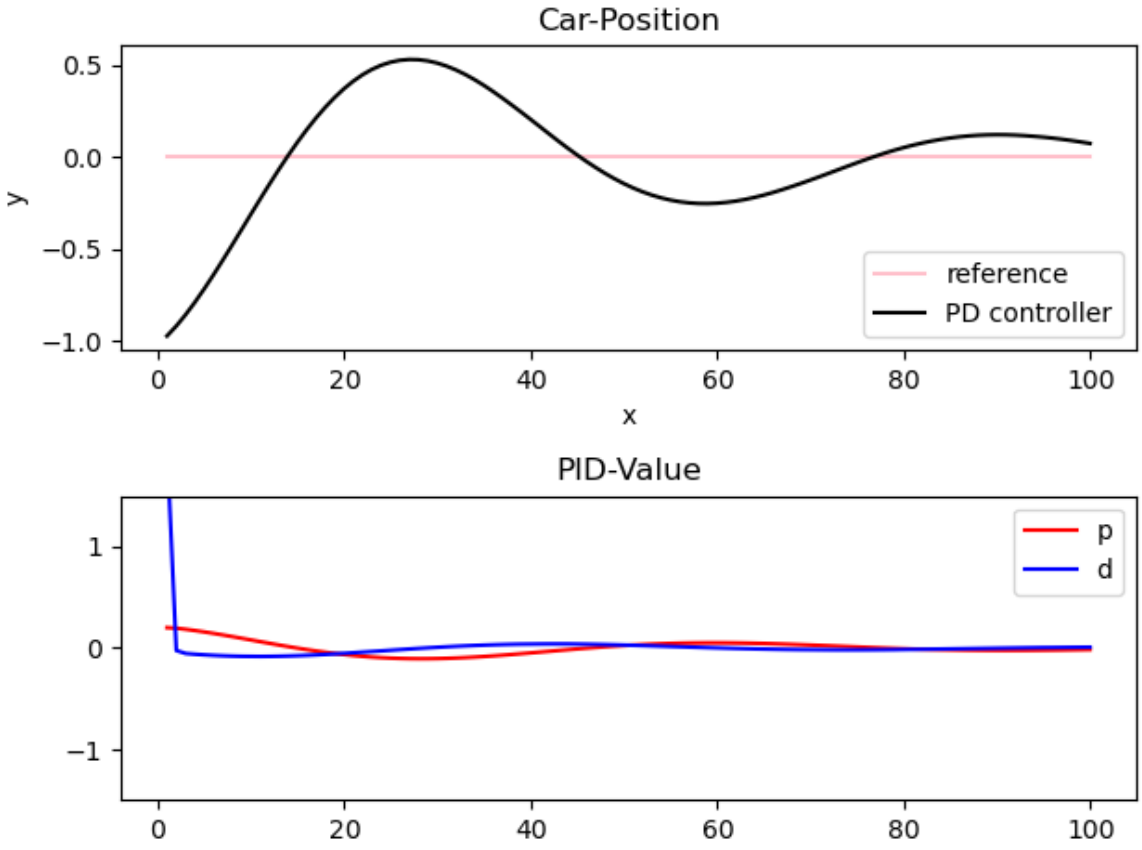
建议自行修改 `k_d` 值，然后观察小车轨迹及日志的变化。

我们发现，在相同的 K_p 系数情况下：

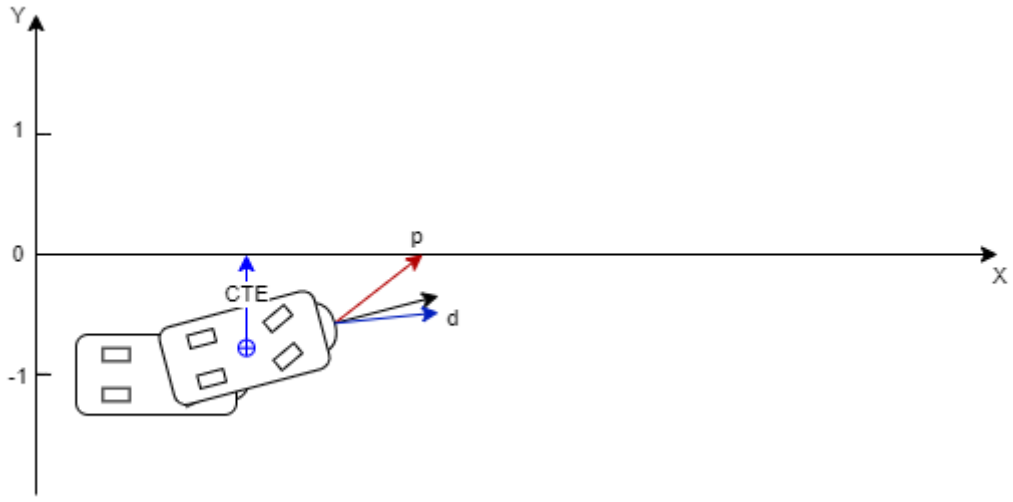
- 当 K_d 适中时，小车的轨迹可以很快与目标轨迹重叠
- 当 K_d 过大时，相当于 **阻尼（Damped）** 过大，导致小车需要较长的时间才能与目标轨迹重叠



- 当 K_d 较小时，相当于 **阻尼（Damped）**较小，导致小车仍然在目标轨迹上下震动，只不过振幅越来越小



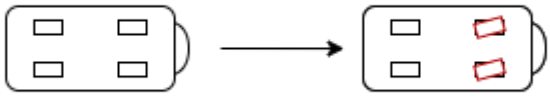
我们可以认为 K_d 项是用于帮我们抵消纠正过多的 K_p 变化。如下图的小车前进方向的蓝色箭头d：



系统性偏差

什么是系统性偏差Systematic Bias，又是如何产生的?

当我们把小车组装好后，理想情况下，我们的前轮应当是与前进的方向对齐的。但是，由于微小的硬件问题或是安装问题，两个前轮又不可避免的会产生一些偏转。这种偏转也可能在小车开一段时间后产生。



对于人来说，这不是一个大问题。当我们注意到这个问题时，我们就会在另一个方向控制得更强一些。比如我们骑的自行车把头是歪的情况下，我们也可以通过控制多打一些方向，正常骑行。那么，对于我们的 PD 控制器来 说，会发生什么事情?

- 在PD-Controller控制器中，添加系统性偏差

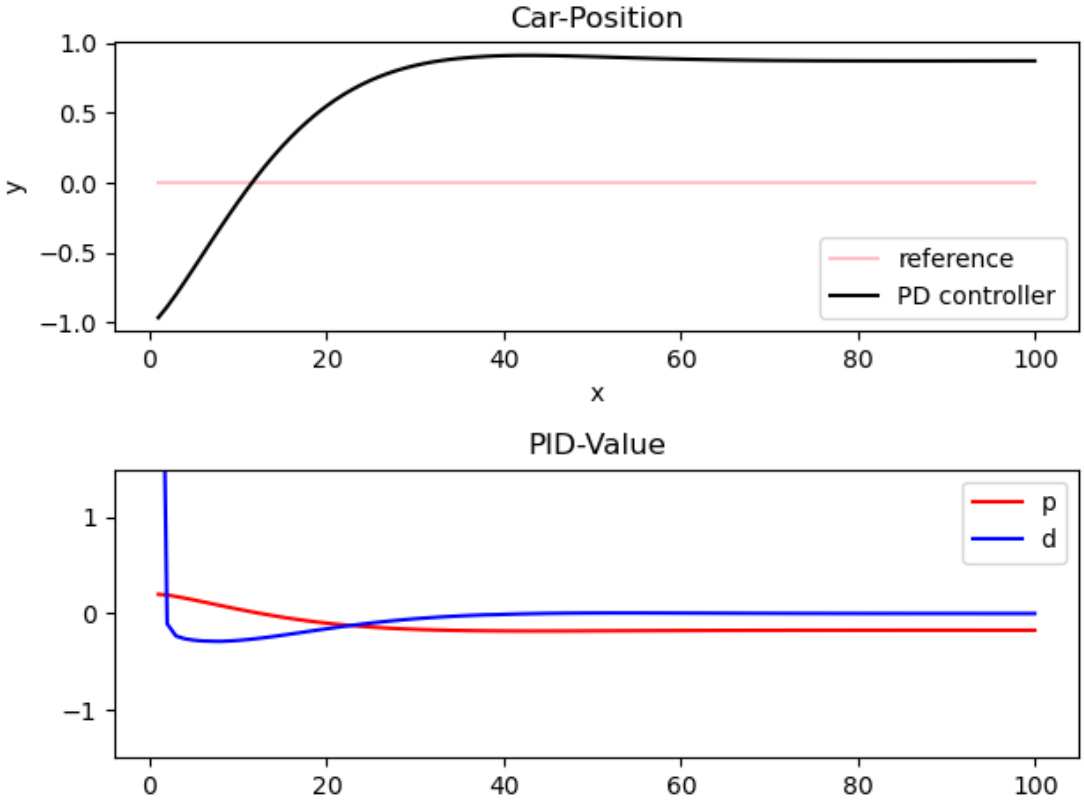
  `robot.set_steering_drift(10. / 180. * np.pi)  # 10 degrees of steer drift`

  即任何偏转都会多出10度的系统性偏差

```
if __name__ == '__main__':
    robot = Robot()
    # 初始位置 x=0, y=-1, orient=0
    robot.set(0, -1, 0)
    robot.set_steering_drift(10. / 180. * np.pi)  # 10 degrees of steer
drift

    # 运行并收集所有的x，y，以及 p值
    x_trajectory, y_trajectory, p_arr, d_arr = run(robot, k_p=0.2, k_d=3.0)

    # 可视化运行结果
    show(x_trajectory, y_trajectory, p_array=p_arr, d_array=d_arr, label =
'PD')
```

  运行结果:

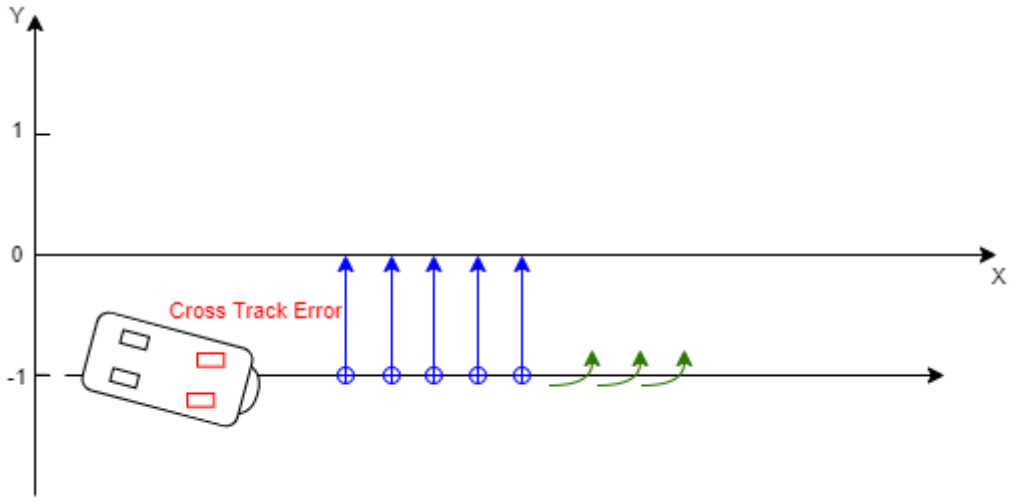
  可以发现其 y 值的范围在 0.8 附近，随后始终没有靠近目标轨迹



也就是说，此时 K_p 产生的比例偏移量 p 和 K_d 产生的阻尼偏移量 d 相互抵消（一正一负）。导致小车始终无法运动到目标轨迹上。 那么我们如何解决这个问题呢?

PID 控制器

首先，为了解决上边产生的问题，想象一下我们平车开车或骑自行车，如果我们按照正常的经验，开了一会儿，发现这段时间离目标路线一直保持的距离，并且没有接近目标，此时可能是轮子有所偏差。如下图:



这时候我们会尝试把方向盘再多向左打一些（逆时针），以靠近目标路线，进而抵消这种系统偏差。那么这个量到底如何计算呢?

为了解决系统性偏差，我们引入 **I控制**，其原理是累计所有历史的CTE横切误差，即使用CTE基于时间的积分 ΣCTE 和 K_i 的乘积作为转向角度增益。

将之前的所有项合并相加后，我们得到完整的 PID控制器 公式:

$$steer = K_p \cdot CTE + K_d \cdot \frac{d}{d_t} CTE + K_i \cdot \Sigma CTE$$

那么，可以预见的是，最后一项由于我们将每次的 CTE 都进行累加，那么 $K_i \cdot \Sigma CTE$ 会越来越大，即使 K_i 比较小，随着CTE数值的累加，也会影响到最终的结果，进而实现对小车运行轨迹的修正。

代码步骤:

1. 拷贝依赖文件 robot.py 到项目目录
2. 新建文件 `pid_controller.py` 并编写如下内容:

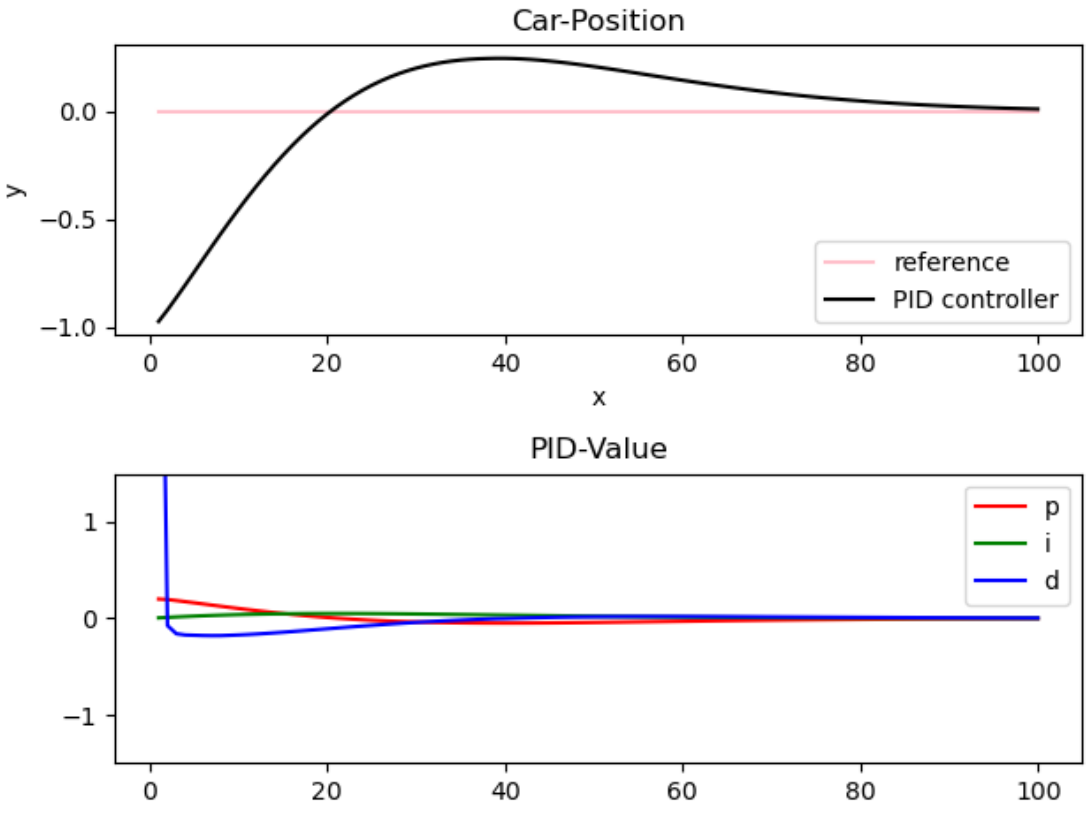
```
from robot import Robot, show
import numpy as np


def run(robot, k_p, k_d, k_i, n=100, speed=1.0):
    """
    运行多次运动并记录轨迹
    :param robot:    小车
    :param k_p:      p系数
    :param k_d:      d系数
    :param n:        循环次数
    :param speed:    小车速度
```

```
    """
    x_trajectory = []
    y_trajectory = []
    p_arr = []  # 记录每一次的p
    d_arr = []  # 记录每一次的d
    i_arr = []  # 记录每一次的i

    prev_cte = 0 - robot.y
    sum_cte = 0
    for i in range(n):
        # ---------------------------- start
        cte = 0 - robot.y
        p = k_p * cte               # p

        d = k_d * (cte - prev_cte)  # d
        prev_cte = cte

        sum_cte += cte
        i = k_i * sum_cte           # i

        steer = p + d + i           # sum

        p_arr.append(p)
        d_arr.append(d)
        i_arr.append(i)
        # ---------------------------- end

        robot.move(steer, speed)

        x_trajectory.append(robot.x)
        y_trajectory.append(robot.y)
        print(robot)
    return x_trajectory, y_trajectory, p_arr, d_arr, i_arr


if __name__ == '__main__':
    robot = Robot()
    # 初始位置 x=0, y=-1, orient=0
    robot.set(0, -1, 0)
    robot.set_steering_drift(10. / 180. * np.pi)  # 10 degrees of steer drift

    # 运行并收集所有的x, y, 以及 p值
    x_trajectory, y_trajectory, p_arr, d_arr, i_arr = run(robot, k_p=0.2,
k_d=0.0, k_i=0.0)
    # 可视化运行结果
    show(x_trajectory, y_trajectory, p_array=p_arr, d_array=d_arr,
i_array=i_arr, label = 'P')
    # 重置小车位置
    robot.reset()

    # 运行并收集所有的x, y, 以及 p值
    x_trajectory, y_trajectory, p_arr, d_arr, i_arr = run(robot, k_p=0.2,
k_d=3.0, k_i=0.0)
    # 可视化运行结果
    show(x_trajectory, y_trajectory, p_array=p_arr, d_array=d_arr,
i_array=i_arr, label = 'PD')
    # 重置小车位置
    robot.reset()
```

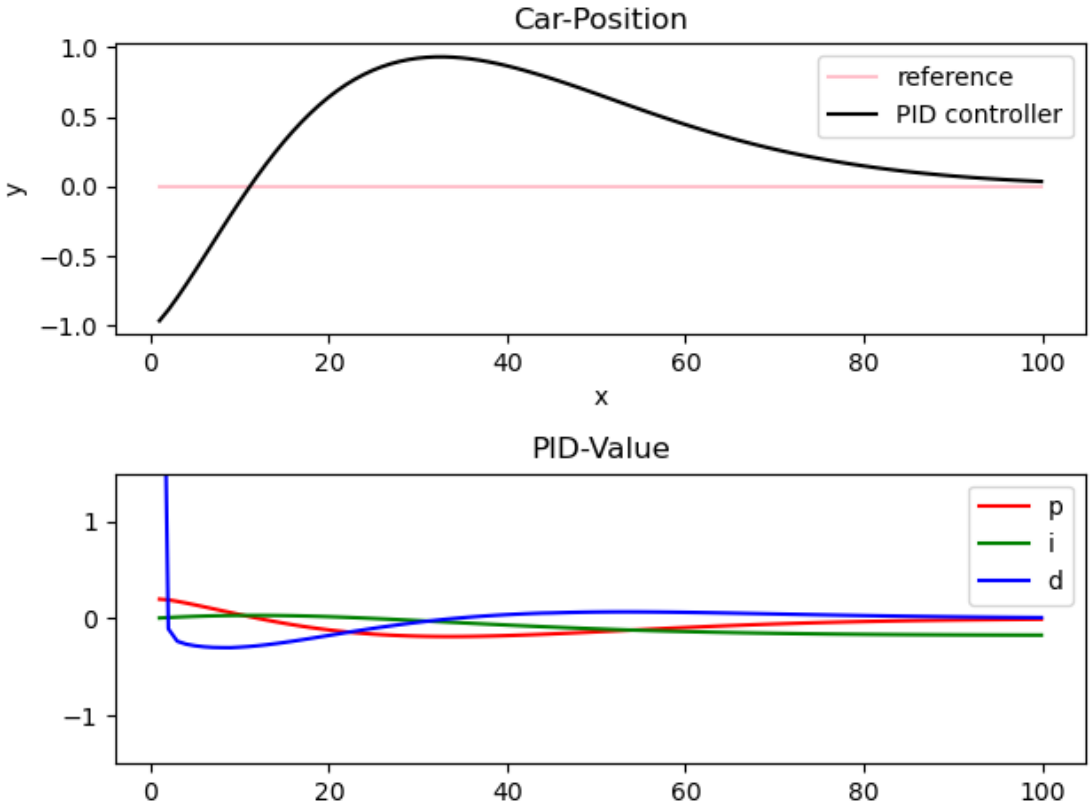
```
    # 运行并收集所有的x，y，以及 p值
    x_trajectory, y_trajectory, p_arr, d_arr, i_arr = run(robot, k_p=0.2,
k_d=3.0, k_i=0.005)
    # 可视化运行结果
    show(x_trajectory, y_trajectory, p_array=p_arr, d_array=d_arr,
i_array=i_arr, label = 'PID')
```

运行结果：

- 在没有系统偏差时：



- 在有系统偏差时：



可以发现，积分 **I控制** 可以帮我们在有一定系统偏差的情况下，帮助小车回到目标轨迹上去。

目前看来，不管是有没有系统偏差，只要添加了 **I控制**， 就会导致结果需要比较久的时间才能到达稳定状态，**PID控制器** 效果似乎还没有直接使用 **PD控制器** 的好，那么如何优化这个问题呢?

即这样的PID值虽然可以帮我们实现需求，但是什么样的PID值才是最优的？这就需要用到接下来的Twiddle调优技术啦。