山东大学

崇新学堂

2025－2026 学年第一学期

实 验 报 告

课程名称：　电子信息工程导论

实验名称：　Sizable Following

专　业　班　级　崇新学堂

学　生　姓　名　高子轩，钱竹玉，吕思洁，徐亚骐

实　验　时　间　2025 年 10 月 30 日

Introduction

During the process of moving the robot in parallel with the proportional controller, we found that there was no appropriate k value that made the robot perform well, especially when the robot was not parallel to the wall at the initial Angle.

In this experiment, we will develop two new types of controllers to achieve better performance; The first controller depends on the previous distance to the wall and the current distance, as well as the second one. The controller depends on the current Angle from the wall and the current distance.

Remembrance of Things Past

We plan to adjust the angular velocity by means of $e[n]$ in the two states. That is:

$$\omega[n] = k_1 e[n] + k_2 e[n-1]$$

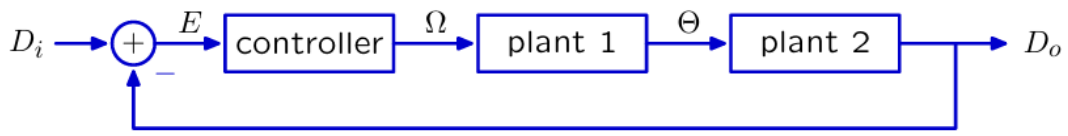


Figure 1 The system for last lab

The system still has the same form as the one from last week, Only the controller has changed.

Bulid the delayPlusPropModel

Check Yourself 1 : Draw a block diagram of the controller.

The system consists of two components: one with gain k_1 enters the adder, while the other with gain k_2 and a one-time delay enters the adder.

Here is our answer

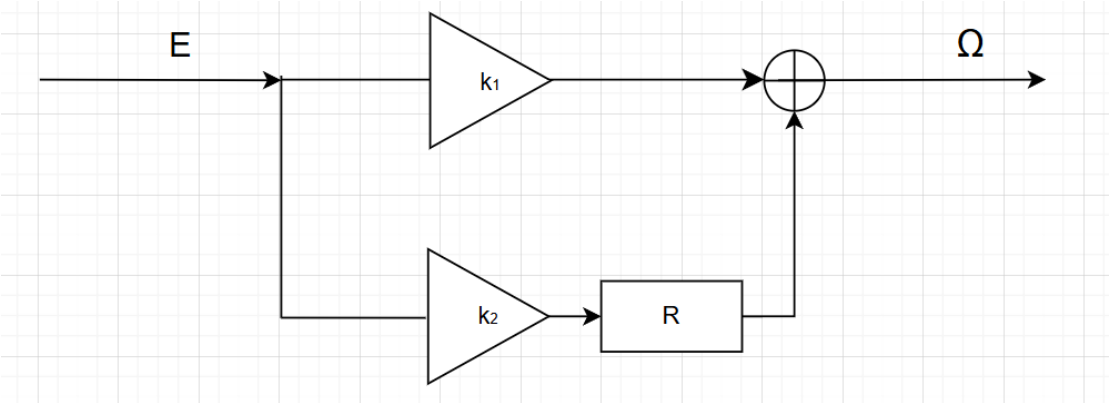


Figure 2 The system of Controller

Step 1: Code for delayPlusPropModel

According to the system diagram, the Controller is composed of two paths added together, one with a gain of k_1 and the other with a gain of k_2, followed by a one-unit delay.

Therefore, sf.FeedforwardAdd is used to connect the upper and lower paths. Since the overall framework remains the same as before, only this part needs to be changed and then connected according to the previous framework to complete it.

Here is the key code

```
def delayPlusPropModel(k1, k2):
    T = 0.1
    V = 0.1

    controller = sf.FeedforwardAdd(sf.Gain(k1), sf.Cascade(sf.R(), sf.Gain(k2)))
    plant1_sys = plant1(T)
    plant2_sys = plant2(T, V)
    total_plant = sf.Cascade(plant1_sys, plant2_sys)
    forward_path = sf.Cascade(controller, total_plant)
    sys = sf.FeedbackSubtract(forward_path,sf.Gain(1))
    return sys
```

Step 2: Look for the value of k_2

Aiming to find the best k_2 to improve system convergence, we should start by understanding a few things:

1. The convergence performance of a system depends primarily on the dominant pole, which is the pole with the largest amplitude, so we have to calculate it first.
2. The smaller the amplitude of the dominant pole, the better the system's convergence performance. Therefore, we need to determine the value of k_2 that minimizes its amplitude.

Based on this, we can calculate the value of k_2 by building a *program*

Code logic:

• First of all, we need to calculate the amplitude function of the dominant pole. Here, the system_function.poles method is used for the calculation.

• Then, through sampling, the k_2 with the smallest amplitude was calculated using optimize.optOverLine.

Here is the key code

```
def bestk2(k1, k2Min, k2Max, numSteps):
    def objective(k2):

        system_function = delayPlusPropModel(k1, k2)
        poles = system_function.poles()
        max_pole_magnitude = max(abs(pole) for pole in poles)
        return max_pole_magnitude

   (bestObjValue, bestK2) = optimize.optOverLine(
        objective,
        k2Min,
        k2Max,
        numSteps,
        operator.lt
    )
return bestK2
```

Here are the best k_2 for every k_1

k_1	k_2	*magnitude of dominant pole*
10	*-9.974*	*0.995*
30	*-29.769*	*0.985*
100	*-97.348*	*0.946*
300	*-271.731*	*0.772*

Step 3: Code for delayPlusPropBrainSkeleton.py

Explanation for code

We define the values of gain in this design lab, through Soar, we found that when k_1 = 100, k_2 = -97.348, the robot can achieve the best performance.

```
k1 = 100
desiredRight = 0.4
forwardVelocity = 0.1
k2 = -97.348
```

This part is defining the sensor ,we get the sensor's input (distance to the wall on right) and print the values.

```
class Sensor(sm.SM):
    def getNextValues(self, state, inp):
        v = sonarDist.getDistanceRight(inp.sonars)
        print 'Dist from robot center to wall on right', v
        return (state, v)
```

We define the WallFollower part to receive the values of sensors and calculate the value of rotation speed.

```
class WallFollower(sm.SM):
    startState = [desiredRight,None]
    def getNextValues(self, state, inp):
        error = desiredRight - inp
        error_diff = desiredRight - state[0]
        rvel = k1*error + k2*error_diff
        return ([inp,state],io.Action(forwardVelocity,rvel))
```

Combine those two state machines, using sensor state machine's

output as WallFollower's input to achieve the goal.

```
sensorMachine = Sensor()

sensorMachine.name = 'sensor'

mySM = sm.Cascade(sensorMachine, WallFollower())
```

Step 4: Simulation in soar

We successively substituted each group of k_1 and k_2 and obtained the simulated operation diagram.

The results as follows

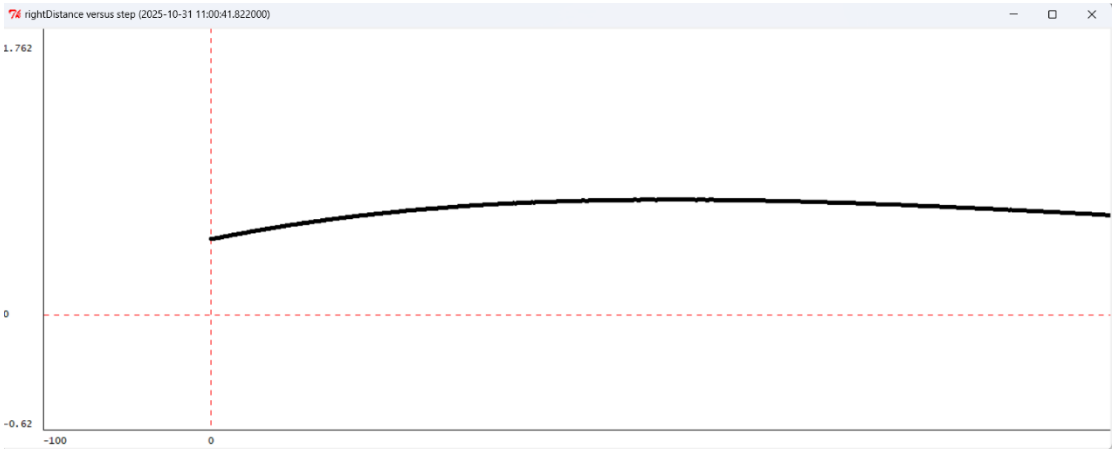


Figure 3 The situation $k_1 = 10, k_2 = -9.974$

The amplitude of the car in the simulation is too large, the period is too long, and the convergence speed is too slow.

It can be seen from the image that the system has poor convergence characteristics and a too slow convergence speed. Even in the end, it still failed to converge to the target distance.

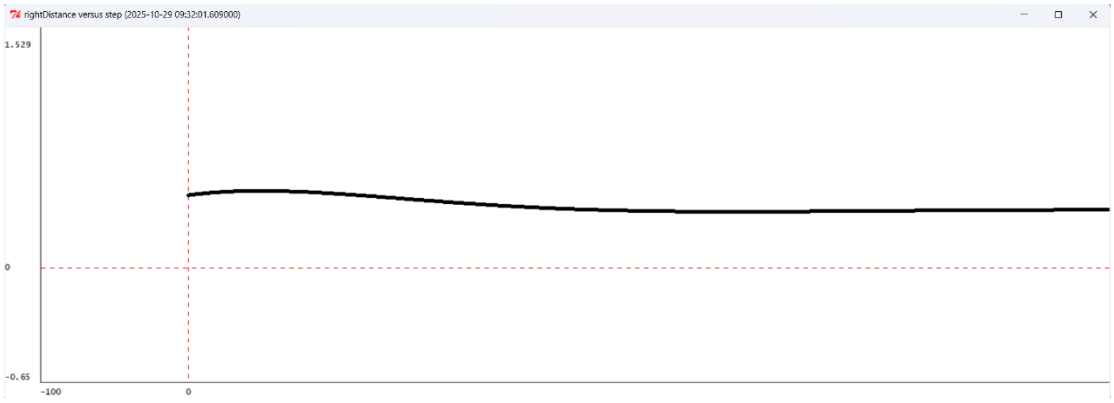


Figure 4 The situation $k_1 = 30, k_2 = -29.769$

In this case, the trolley can converge relatively quickly and well, and the system is stable with good convergence characteristics

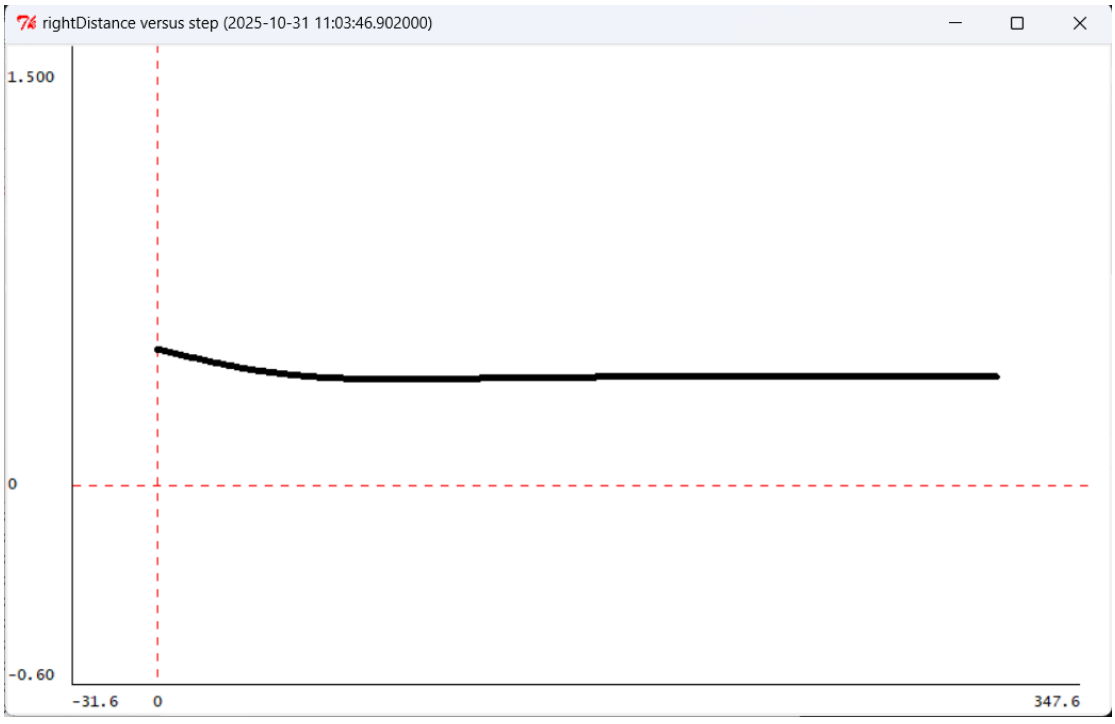


Figure 5 The situation $k_1 = 100, k_2 = -97.348$

The convergence speed has been further accelerated, and the convergence performance of the system is very good.

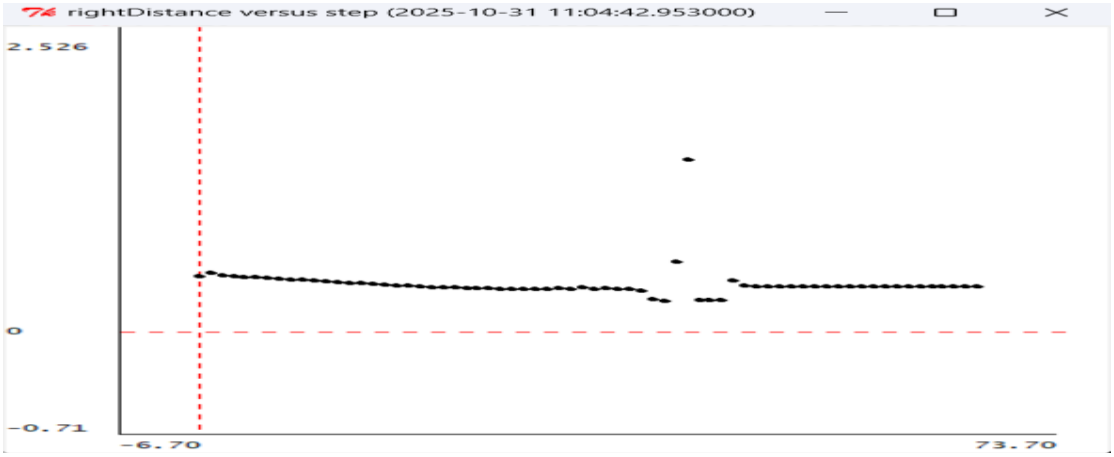


Figure 6 The situation $k_1 = 300, k_2 = -271.731$

The amplitude of this situation is too large. It directly hit the wall and the effect is not good.

Comparison result

By comparing the above four pictures, it can be seen that the fourth plot has a collision and the amplitude is too large. The first plot hasn't finished its task and keeps out of the desired distance.

Check Yourself 3. Which of the four gain pairs work best in simulation?

The best

- k_1=100 k_2=-97.35

The convergence speed is fast and stable

The bads

- k_1 = 10 k_2 = -9.974

The amplitude is too large, the period is too long, the convergence speed is too slow

- k_1 = 300 k_2 = -271.731

The amplitude was too large and it hit the wall directly

Step5：Real robot debugging

Check Yourself 4. Which of the four gain pairs work best on a robot?

The best

- k_1= 30 k_2=-29.769

The convergence speed is fast and stable

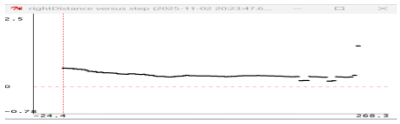
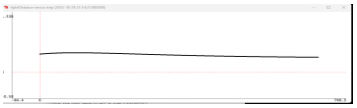
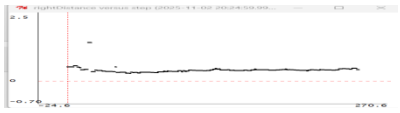
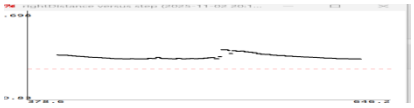
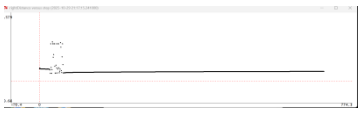
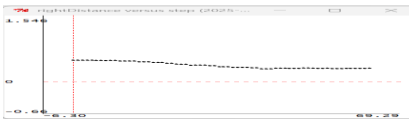
The bads

- k_1= 100 k_2 = -97.35
- k_1= 300 k_2 = -271.731

The ampltitude is too large and it hit the wall directly

Checkoff 1. See real car effects

Comparison of simulation and experimental results

simulation	*real*	k_1, k_2
		k_1=10 k_2=-9.974
		k_1=30 k_2=-29.769
		k_1=100 k_2=-97.348
		k_1=300 k_2=-271.731

The simulated curve is smoother and more even, while the actual vehicle curve shows fluctuations. Since the simulation is an idealized environment, the results are as expected; in contrast, the real vehicle environment is subject to a series of errors due to interference from sensors and other environmental factors.

Relationship between robot's behaviour and dominant pole's value

The smaller the dominant extremum, the faster the oscillation converges, and the faster the car can approach the target

The explanation of Startstate

The first element represents the distance to the right wall, and the second element is a boolean value that is in an unused state. The first element stores the previous partition distance, which facilitates the calculation of $e[n-1]$—this is precisely why it is defined.

Bulid the anglePlusPropModel

Step 6 Code for anglePlusPropModel

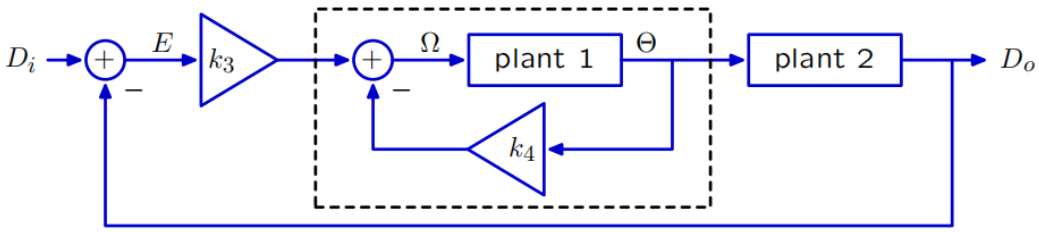


Figure 7 The new system for anglePlusPropModel

Code logic

Here, Plant1 becomes a negative feedback system, where the forward path is the original Plant1 and the feedback path is the gain k_4, so sf.FeedbackSubtract is used.

Here is the key code

```
def anglePlusPropModel(k3, k4):
    T = 0.1
    V = 0.1
```

```
    # plant 1 is as before
    plant1_sys = plant1(T)
    # plant2 is as before
    plant2_sys = plant2(T, V)
    new_plant1 = sf.FeedbackSubtract(plant1_sys,sf.Gain(k4))
    new_total_plant = sf.Cascade(new_plant1, plant2_sys)
    forward_path = sf.Cascade(sf.Gain(k3), new_total_plant)
    # The complete system
    sys = sf.FeedbackSubtract(forward_path,sf.Gain(1))

    return sys
```

Step 7 Look for the value of k_4

Similar to Step2, through the same procedural sampling, we can find k_4 for each k_3

k_3	k_4	*magnitude of dominant pole*
1	*0.630*	*0.969*
3	*1.094*	*0.945*
10	*1.998*	*0.900*
30	*3.462*	*0.827*

Step 8 Code for anglePlusPropBrainSkeleton.py

Explanation for code

To implement the proportional plus angle controller, we edit two parts connected in cascade. Firstly, through testing in Soar, we found the best performance when k_3=1, k_4=0.6. We also set the desired right distance and forward velocity.

```
K3=1
K4=1
desiredRight = 0.4
forwardVelocity = 0.1
```

This sensor part is the same as the sensor in delayPlusPropBrainSkeleton.py, which can print the current distance value in real time for easy debugging and issue a warning when the angle data is invalid.

Then, we define the the WallFollower class , whose input is a pair of the perpendicular distance to the wall on the right and the angle to the wall and whose output is an instance of the class io.Action. It can understand the current relationship with the wall, calculate the best steering strategy and control the robot to move smoothly along the wall.

```
class WallFollower(sm.SM):
   startState = None
   def getNextValues(self, state, inp):
      (distanceRight,angle)=inp
      e1 = desiredRight - distanceRight
      e2 = k4*angle
      w = k3*e1 -k4*e2
      return (None,io.Action(fvel = forwardVelocity,rvel = w))
```

Step 9 Simulation in soar

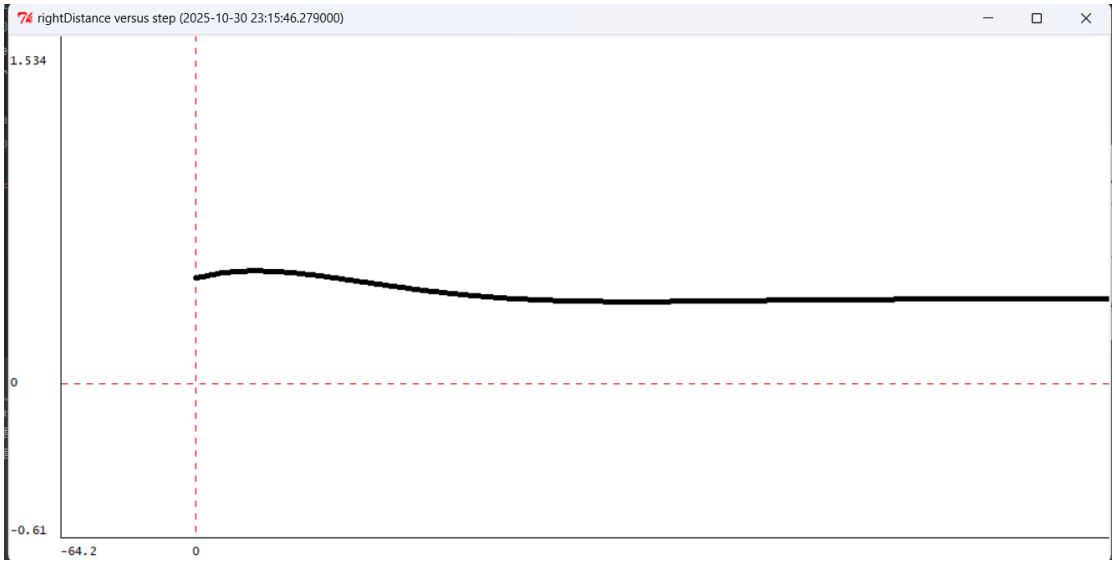


Figure 8 The situation $k_3 = 1, k_4 = 0.630$

It converges stably after a slight oscillation once, and the convergence effect is relatively good

Figure 9 The situation $k_3 = 3, k_4 = 1.094$

The oscillation is slighter and the convergence is improved.

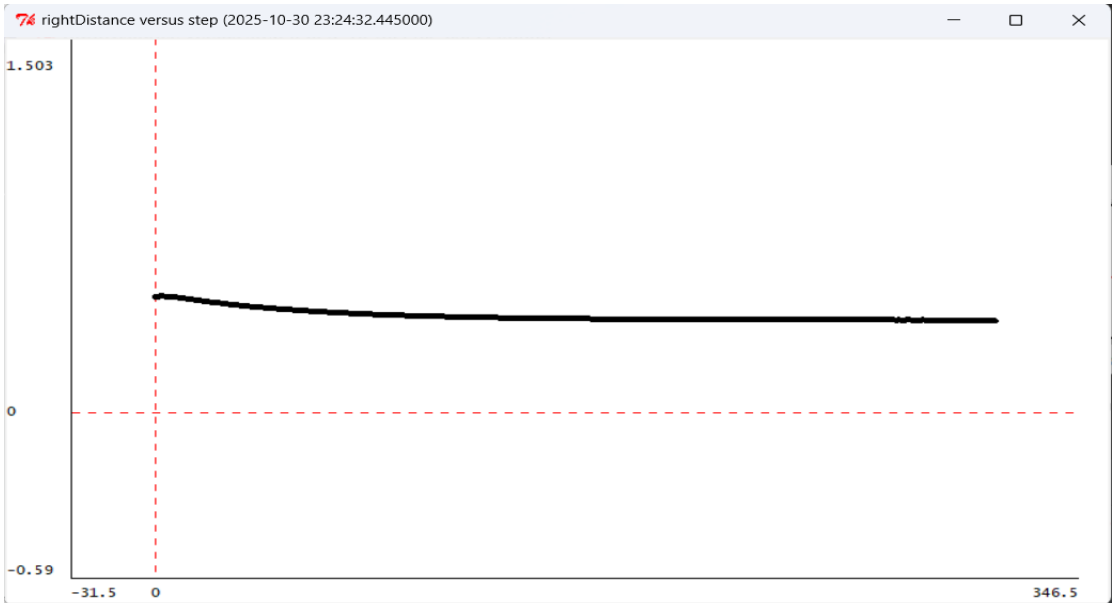


Figure 10 The situation $k_3 = 10, k_4 = 1.998$

It hardly oscillates anymore and has a very good convergence effect.

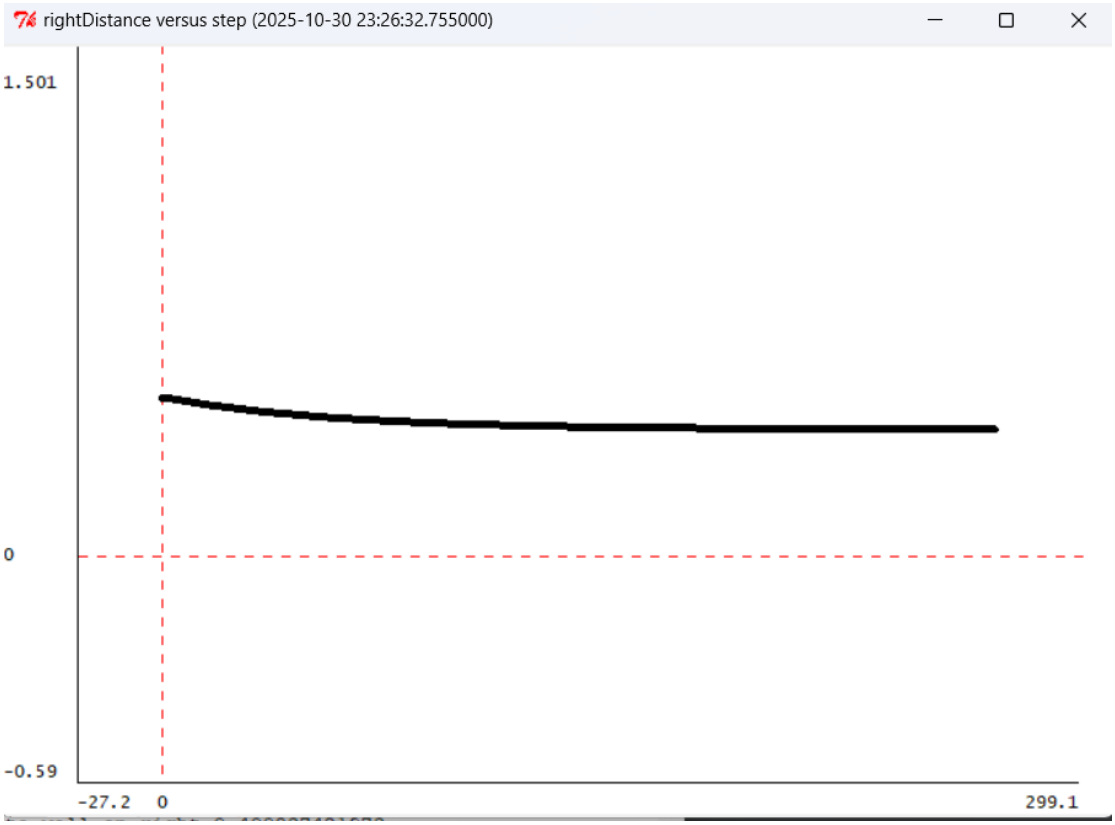


Figure 11 The situation $k_3 = 30, k_4 = 3.462$

No longer oscillating, directly monotonically converging, stable and

rapid

Comparison result

The stability and convergence speed of the four systems are all excellent, and the best one is the fourth one.

Check Yourself 5. Which of the four gain pairs work best in simulation?

The best

$k_3 = 30$, $k_4 = 3.462$

The convergence speed is fast and stable

Others

The convergence effects of other values are also quite good. Among the four values in this group, there are no values that are obviously very poor.

The oscillation is relatively large and the performance is normal.

Step 10: Real robot debugging

Check Yourself 6. Which of the four gain pairs work best on a robot?

The best

• k_3= 1 k_4=0.630

The convergence speed is fast and stable

The bads

•k_3= 10 k_4 = 1.998

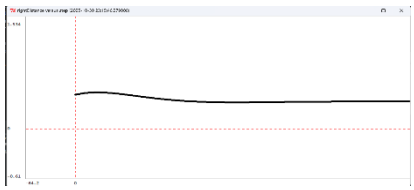
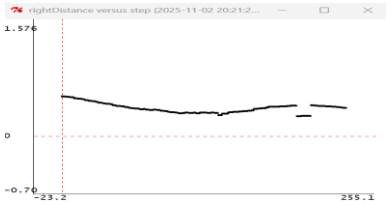
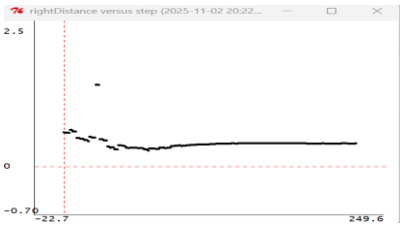
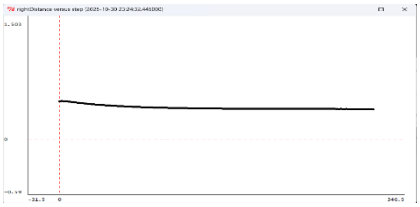
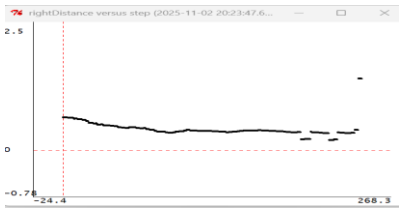
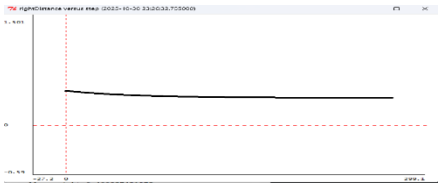
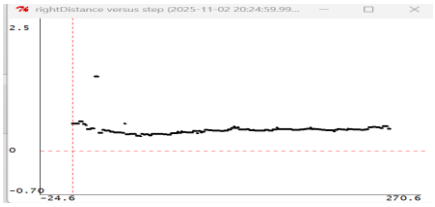
•k_3= 30 k_4 = 3.462

The amplitude of the correction was too large, causing violent oscillations. This resulted in an unstable rotational velocity, making the robot prone to spinning in place or colliding with the wall.

Checkoff 2 See real car effects

Comparison of simulation and experimental results

simulation	*real*	k_3,k_4

		$k_3 = 1$ $k_4 = 0.630$
		$k_3 = 3,$ $k_4 = 1.094$
		$k_3 =$ $10, k_4 =$ 1.998
		$k_3 =$ $30, k_4 =$ 3.462

The simulated trajectory is significantly smoother and more stable. In contrast, the actual vehicle's trajectory exhibits noticeable and persistent jitter. This discrepancy is expected, as the simulation operates in an idealized environment. The real-world experiment, however, is subject to a range of errors caused by sensor noise and various environmental interferences, leading to less satisfactory performance. Consequently, while this configuration yields the best results in simulation, its performance degrades in real-world testing

Relationship between robot's behaviour and dominant pole's value:

The smaller the dominant extremum, the faster the oscillation converges, and the faster the car can approach the target distance

Analysis of the better performance of the angle-plus-proportional controller

The angle-plus-proportional controller outperforms the delay-plus-proportional controller in achieving parallel alignment with the wall more rapidly and ensuring greater driving stability.

While the delay-plus-proportional controller determines the rotational velocity (rvel) based solely on the error between the actual and desired distance from the wall, the angle-plus-proportional controller calculates rvel by integrating both the angular deviation from parallel alignment and the distance error. This integrated approach provides a more comprehensive consideration of the vehicle's state, allowing for finer and more nuanced adjustments.

In addition, the angle gain controller calculates the angular deviation, denoted as "angle," between the current orientation and the wall-parallel direction. This approach is more concise and faster compared to the delay gain controller, which computes the difference between the ideal and actual distances from the wall.

As a result, it significantly enhances system stability and reduces the need for constant corrective maneuvers during movement.

Appendix: The Description of AI Usage in the Report

In this experiment, we used AI exclusively for English translation, along with partial code naming optimization and verification.