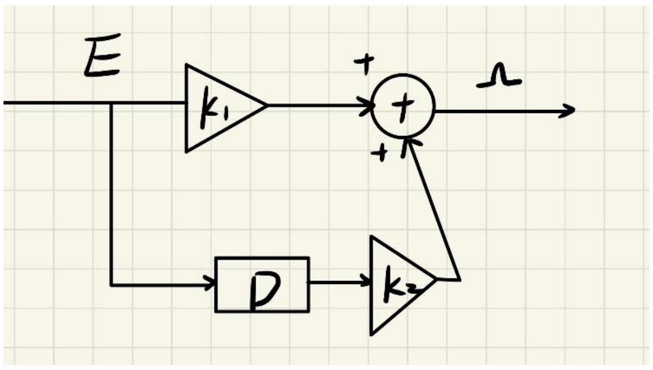
Check Yourself 1.

Here is the block diagram of the controller

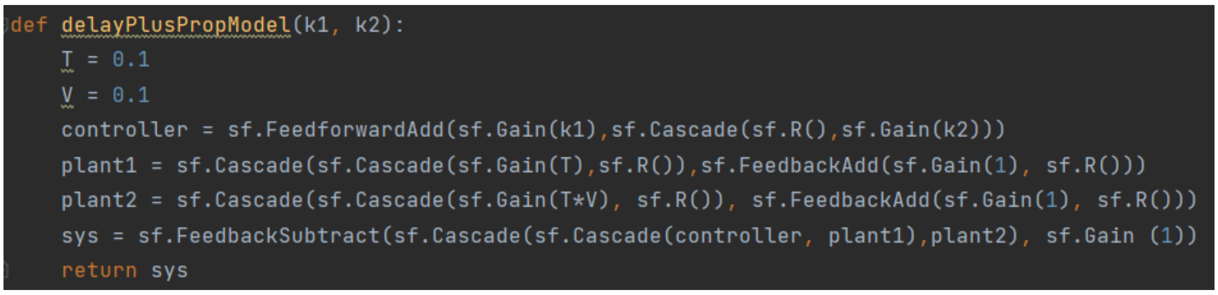


Step1

Wk.6.1.1 Part 1

Using the main body frame of 5.3.4 and SystemFunction class writes controller and realizes delayPlusPropModel.

omega[n] = k1 * e[n] + k2 * e[n-1]



```
def delayPlusPropModel(k1, k2):
    T = 0.1
    V = 0.1
    controller = sf.FeedforwardAdd(sf.Gain(k1),sf.Cascade(sf.R(),sf.Gain(k2)))
    plant1 = sf.Cascade(sf.Cascade(sf.Gain(T),sf.R()),sf.FeedbackAdd(sf.Gain(1), sf.R()))
    plant2 = sf.Cascade(sf.Cascade(sf.Gain(T*V), sf.R()), sf.FeedbackAdd(sf.Gain(1), sf.R()))
    sys = sf.FeedbackSubtract(sf.Cascade(sf.Cascade(controller, plant1),plant2), sf.Gain (1))
    return sys
```

Step2

Wk.6.1.1 Part 2

We present the results first and then explain the implementation.

K1	K2	Magnitude of dominant pole
10	-9.9	0.99
30	-29.8	0.98
100	-97.3	0.95
300	-271.7	0.77

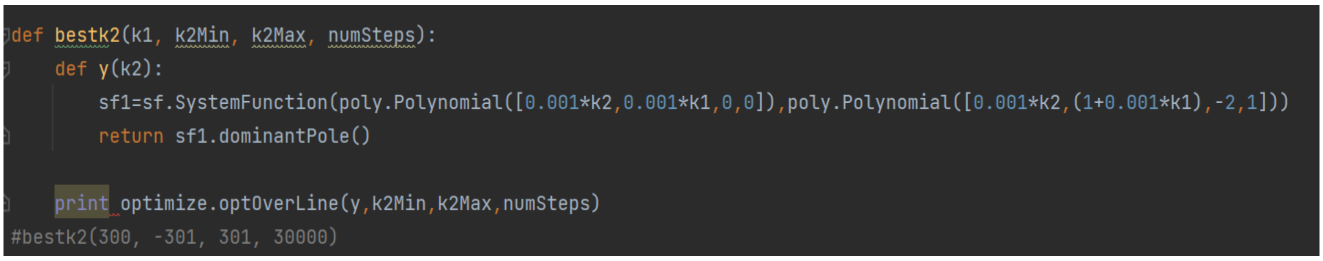
For this new system，we only change the pary of ‘controller’.

System function for delay plusproportional:

$$H = \frac{VT^2K_2R^3 + VTK_1R^2}{VT^2K_2R^3 + (1 + VT^2K_1)R^2 - 2R + 1}$$

We can find that it is a third order system function, which have huge amount of computation by cubic root formula.

Look back to homework2 ,we import Module sf and lib601 optimize module to structure the function 'bestk2' (T=0.1 seconds and V=0.1m/s)



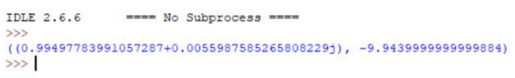
```python
def bestk2(k1, k2Min, k2Max, numSteps):
    def y(k2):
        sf1=sf.SystemFunction(poly.Polynomial([0.001*k2,0.001*k1,0,0]),poly.Polynomial([0.001*k2,(1+0.001*k1),-2,1]))
        return sf1.dominantPole()

    print optimize.optOverLine(y,k2Min,k2Max,numSteps)
#bestk2(300, -301, 301, 30000)
```

Function y(k2) definded directly to given the dominatPole.

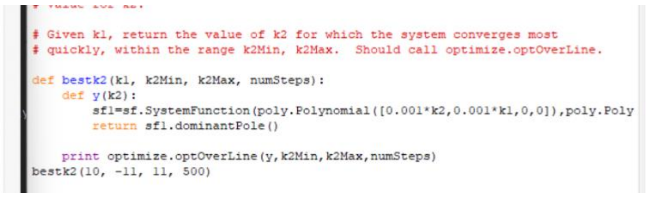
By the Hint: the magnitude of k2 should not be bigger than that of k1, we can ensure the range of k2Min、K2Max(-k1-1~k1+1) and number of sampling points(at least 20*(k1+1))

Picking gains:

When k1 = 10:

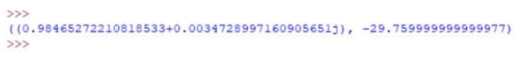


```
IDLE 2.6.6      ==== No Subprocess ====
>>>
((0.99497783991057287+0.0055987585265808229j), -9.9439999999999884)
>>> |
```

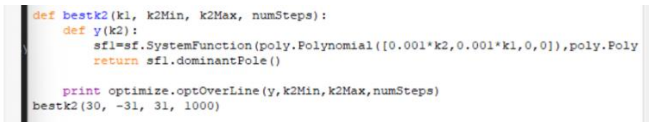


```python
# Given k1, return the value of k2 for which the system converges most
# quickly, within the range k2Min, k2Max.  Should call optimize.optOverLine.

def bestk2(k1, k2Min, k2Max, numSteps):
    def y(k2):
        sf1=sf.SystemFunction(poly.Polynomial([0.001*k2,0.001*k1,0,0]),poly.Poly
        return sf1.dominantPole()

    print optimize.optOverLine(y,k2Min,k2Max,numSteps)
bestk2(10, -11, 11, 500)
```

When k1 = 30:

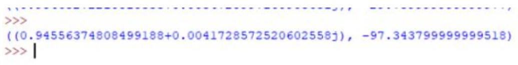


```
>>>
((0.98465272210818533+0.0034728997160905651j), -29.759999999999977)
>>>
```

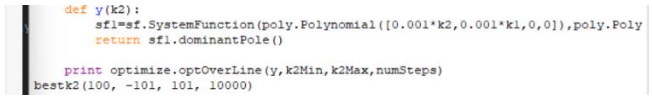


```python
def bestk2(k1, k2Min, k2Max, numSteps):
    def y(k2):
        sf1=sf.SystemFunction(poly.Polynomial([0.001*k2,0.001*k1,0,0]),poly.Poly
        return sf1.dominantPole()

    print optimize.optOverLine(y,k2Min,k2Max,numSteps)
bestk2(30, -31, 31, 1000)
```

When k1 = 100:

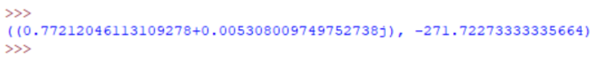


```
>>>
((0.94556374808499188+0.0041728572520602558j), -97.343799999999518)
>>> |
```

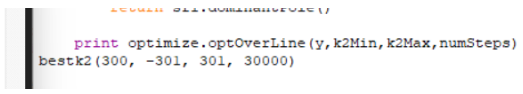


```python
    def y(k2):
        sf1=sf.SystemFunction(poly.Polynomial([0.001*k2,0.001*k1,0,0]),poly.Poly
        return sf1.dominantPole()

    print optimize.optOverLine(y,k2Min,k2Max,numSteps)
bestk2(100, -101, 101, 10000)
```

When k1 = 300:



```
>>>
((0.77212046113109278+0.0053080097497527383j), -271.72273333335664)
>>>
```

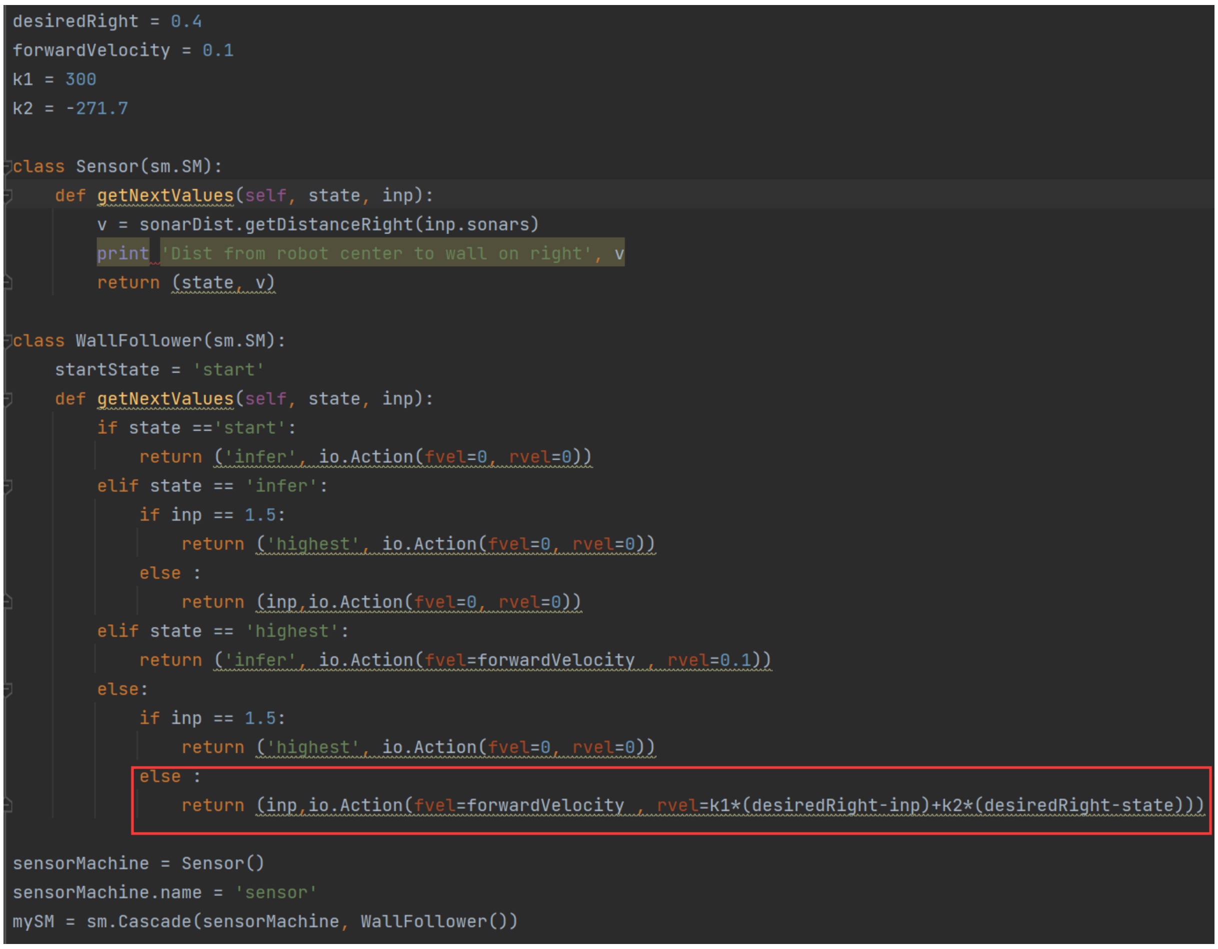


```
    print optimize.optOverLine(y,k2Min,k2Max,numSteps)
bestk2(300, -301, 301, 30000)
```

(The first result in a tuple indicates the dominant pole with the smallest value in the search range，The second result in the tuple represents the best gain in the search range k2)

Of course, the magnitude needs to be calculated by ourselves

Step3

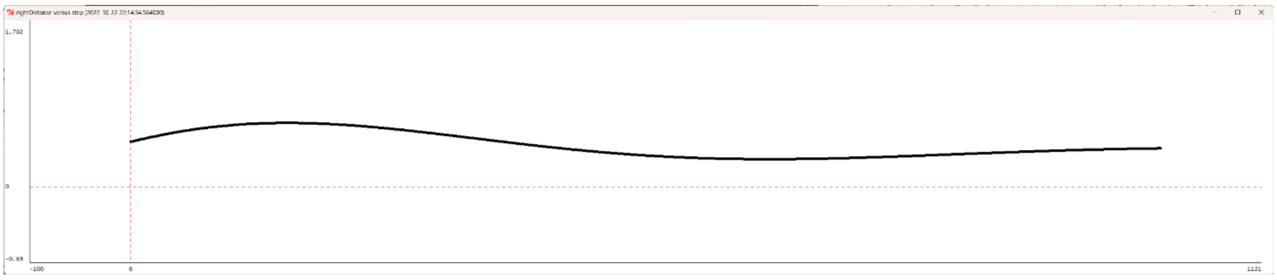


```
desiredRight = 0.4
forwardVelocity = 0.1
k1 = 300
k2 = -271.7

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)

class WallFollower(sm.SM):
    startState = 'start'
    def getNextValues(self, state, inp):
        if state =='start':
            return ('infer', io.Action(fvel=0, rvel=0))
        elif state == 'infer':
            if inp == 1.5:
                return ('highest', io.Action(fvel=0, rvel=0))
            else :
                return (inp,io.Action(fvel=0, rvel=0))
        elif state == 'highest':
            return ('infer', io.Action(fvel=forwardVelocity , rvel=0.1))
        else:
            if inp == 1.5:
                return ('highest', io.Action(fvel=0, rvel=0))
            else :
                return (inp,io.Action(fvel=forwardVelocity , rvel=k1*(desiredRight-inp)+k2*(desiredRight-state)))

sensorMachine = Sensor()
sensorMachine.name = 'sensor'
mySM = sm.Cascade(sensorMachine, WallFollower())
```

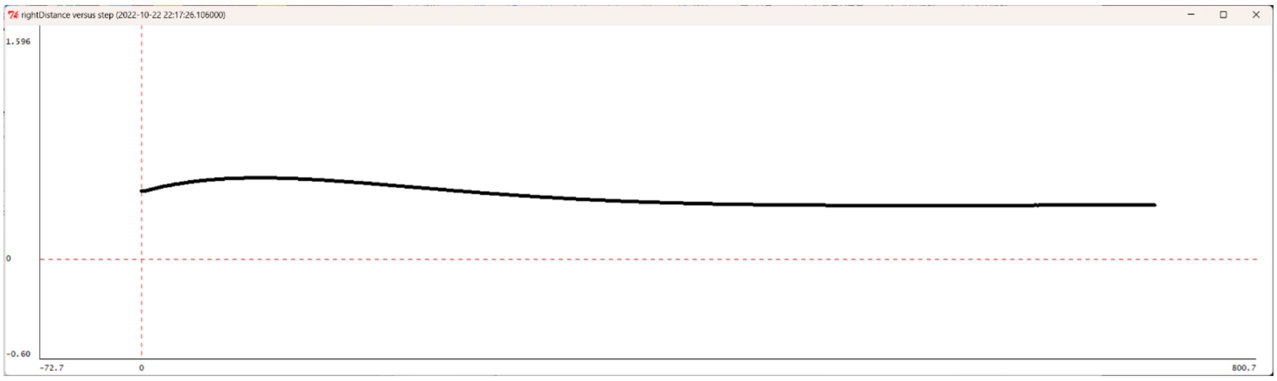
For originanl state ,we intend to only to collect location information and output nothiong.In this progress, we ensure the robot gets valid n-1 moment information which avoid getting error output.

The main body of this code is marked by red line.Virtually, it comes from `omega[n] = k1 * e[n] + k2 * e[n-1].`

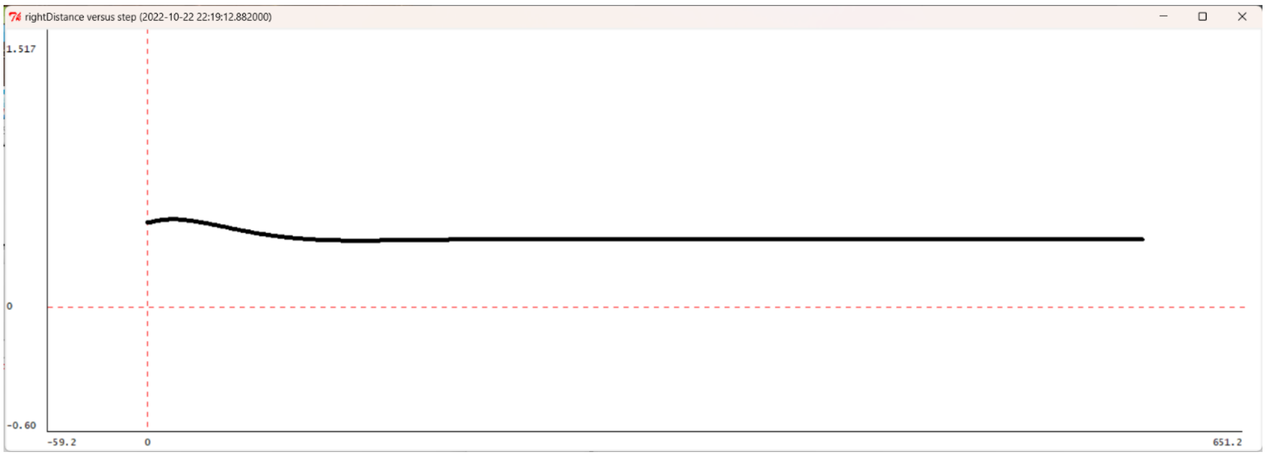
Step4



K1 = 10、k2 = -9.9

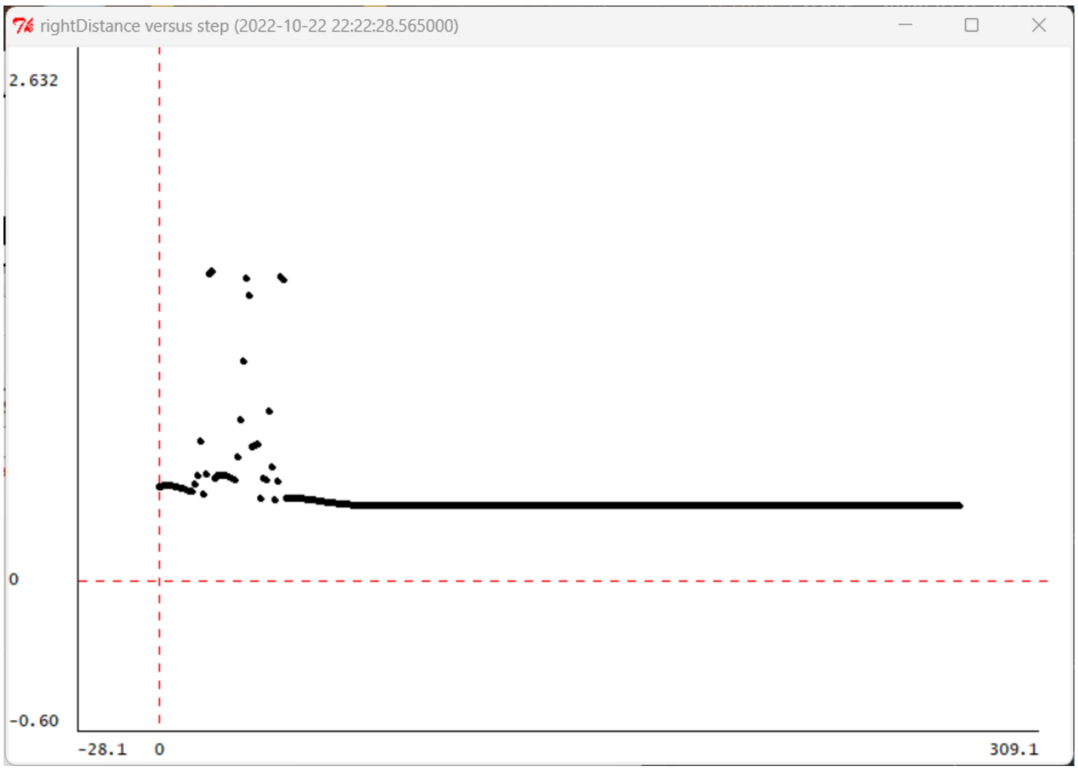


K1 = 30、k2 = -29.8



K1 = 100、k2 = -97.3

For this three situations, As the value of k1 increases, our system stabilizes significantly faster, and at the same time the amplitude of the shock at the target position decreases。



K1 = 300、k2 = -271.3

Obviously, our robot twitched in its initial state.The reason is that on the premise of no output in the initial state, the angular velocity of io output exceeds the acceptable error range of the whole system due to the large value of k1.Cannot get valid correct output.However, we can see that because the system is processing and judging the data all the time, the convulsions of the system can always find a suitable exit, and thanks to the large k1 gain, the robot quickly stabilized at the target position.

Check Yourself 3.

K1 = 100 and K2 = -97.3 works best in simulation.

K1 = 300 and K2 = -271.3 causes bad behavior.

Step5

We did the experiment offline, and the relevant data will be given in Checkoff1 and Check Yourself4

Check Yourself 4.

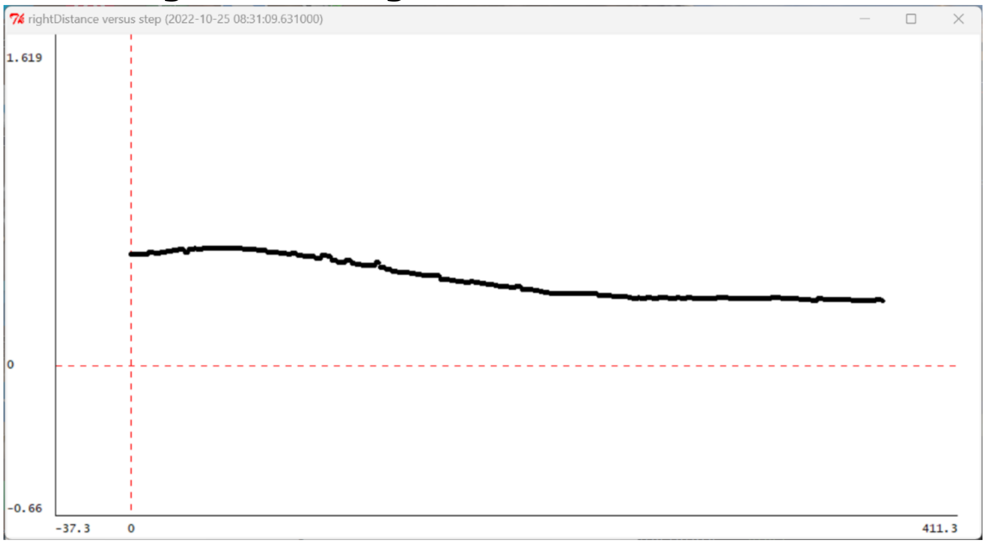
K1 = 30 and K2 = -29.8 works best on a robot.

It is not as same as simulation.

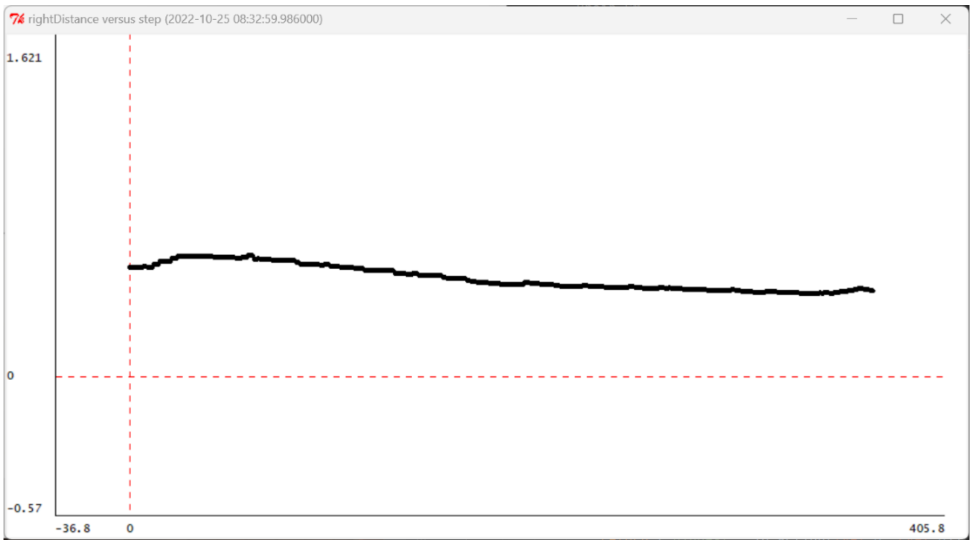
K1 = 300 and K1 = 100 causes bad behavior.

Checkoff 1.

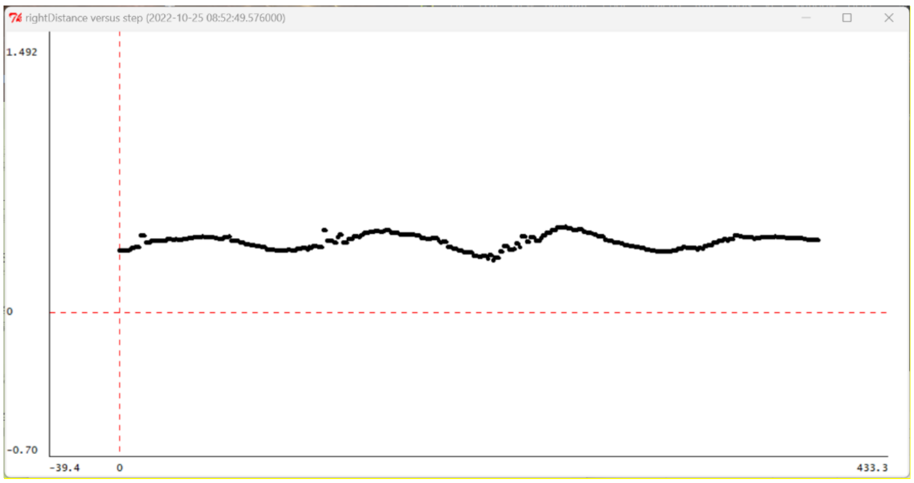
Here is the image of convergence on a robot:



K1 = 10、k2 = -9.9

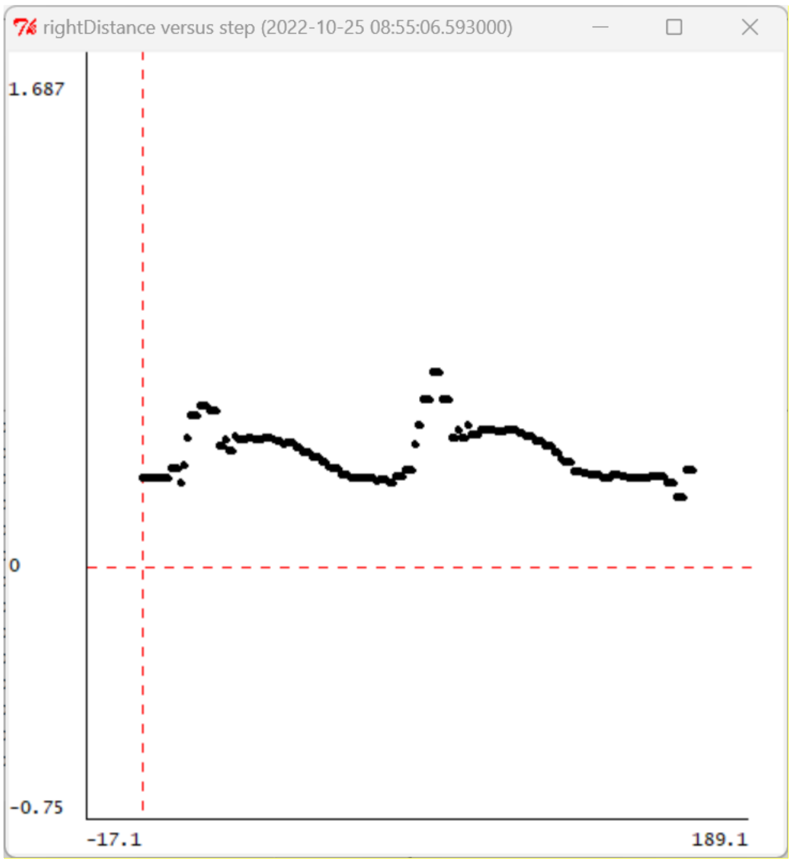


K1 = 30、k2 = -29.8



K1 = 100、k2 = -97.3

For this situation ,we changed original position (make the robot closer to the wall)



K1 = 300、k2 = -271.3

As for two bad bahavior, limited by the sonar sampling speed in the actual situation, the robot position information we get at each moment is not so accurate, and there is a delay, which leads to the

smaller k1 gain we can accept compared with the simulation.

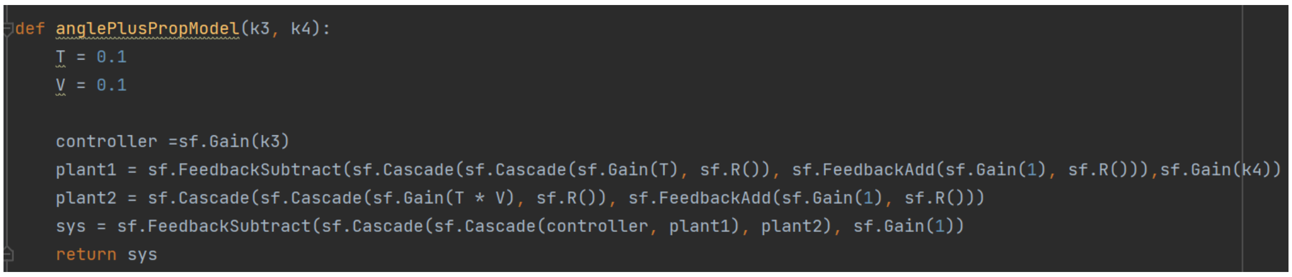
The rest of the report is written in a similar style as above

Step6

Wk.6.1.1 Part 1

Rewritting controller and realizes anglePlusPropModel.

$$\omega[n] = k_3 e[n] - k_4 \theta[n]$$



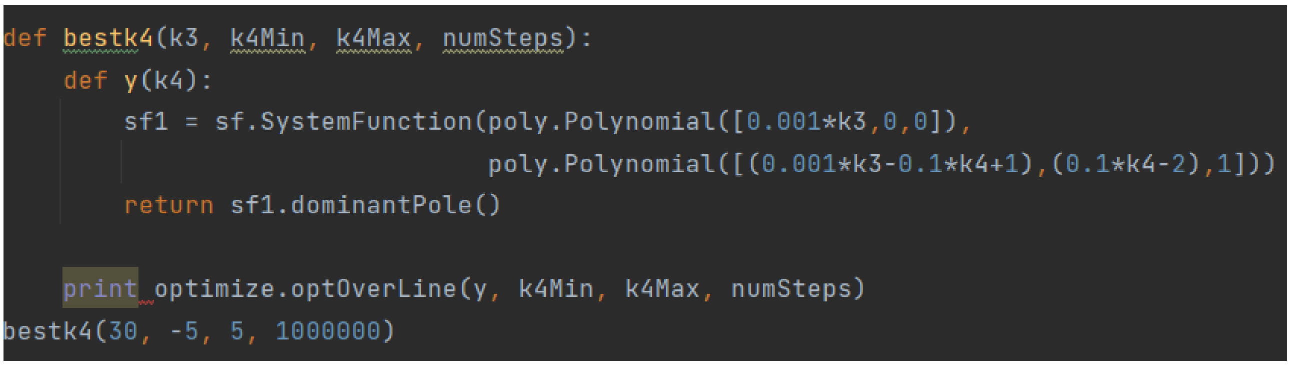
```
def anglePlusPropModel(k3, k4):
    T = 0.1
    V = 0.1

    controller =sf.Gain(k3)
    plant1 = sf.FeedbackSubtract(sf.Cascade(sf.Cascade(sf.Gain(T), sf.R()), sf.FeedbackAdd(sf.Gain(1), sf.R())),sf.Gain(k4))
    plant2 = sf.Cascade(sf.Cascade(sf.Gain(T * V), sf.R()), sf.FeedbackAdd(sf.Gain(1), sf.R()))
    sys = sf.FeedbackSubtract(sf.Cascade(sf.Cascade(controller, plant1), plant2), sf.Gain(1))
    return sys
```

Step7

System function for delay angleproportional:

$$H = \frac{VT^2K_3R^2}{(1 - K_4 + VT^2K_3)R^2 + (K_4T - 2)R + 1}$$

Using the same method of step2 .We can define the bestk4:

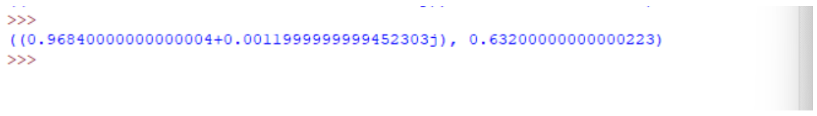


```
def bestk4(k3, k4Min, k4Max, numSteps):
    def y(k4):
        sf1 = sf.SystemFunction(poly.Polynomial([0.001*k3,0,0]),
                                poly.Polynomial([(0.001*k3-0.1*k4+1),(0.1*k4-2),1]))
        return sf1.dominantPole()

    print optimize.optOverLine(y, k4Min, k4Max, numSteps)
bestk4(30, -5, 5, 1000000)
```

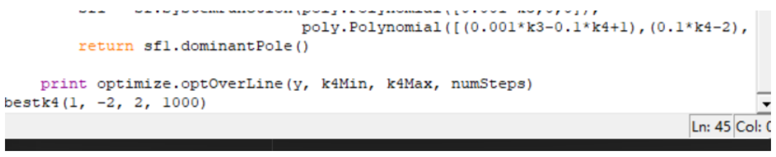
The overall sampling rules remain the same, only change ‘numSteps’. After a certain range of sampling we found that k2 is stable in the single digits.

Picking gains:

When k1 = 1:

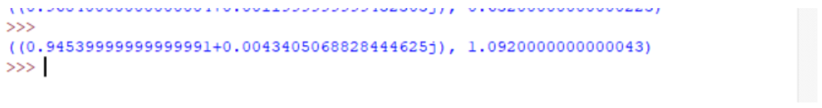


```
>>>
((0.96840000000000004+0.0011999999999452303j), 0.63200000000000223)
>>>
```

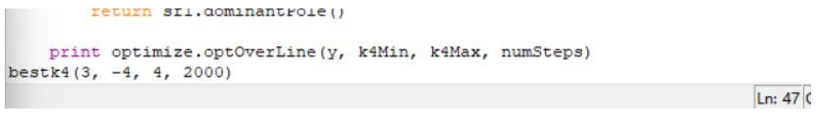


```
                              poly.Polynomial([(0.001*k3-0.1*k4+1),(0.1*k4-2),
        return sfl.dominantPole()

    print optimize.optOverLine(y, k4Min, k4Max, numSteps)
bestk4(1, -2, 2, 1000)
```

When k1 = 3:

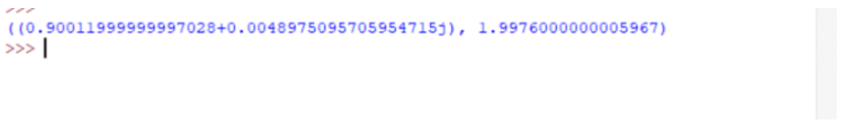


```
>>>
((0.94539999999999991+0.0043405068828444625j), 1.0920000000000043)
>>> |
```

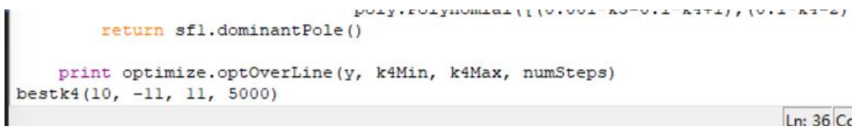


```
    print optimize.optOverLine(y, k4Min, k4Max, numSteps)
bestk4(3, -4, 4, 2000)
```

When k1 = 10:

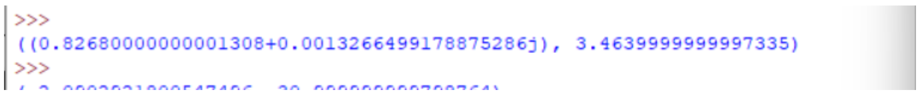


```
((0.90011999999997028+0.0048975095705954715j), 1.9976000000005967)
>>> |
```

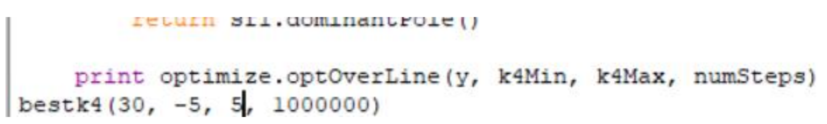


```
        return sfl.dominantPole()

    print optimize.optOverLine(y, k4Min, k4Max, numSteps)
bestk4(10, -11, 11, 5000)
```

When k1 = 30:



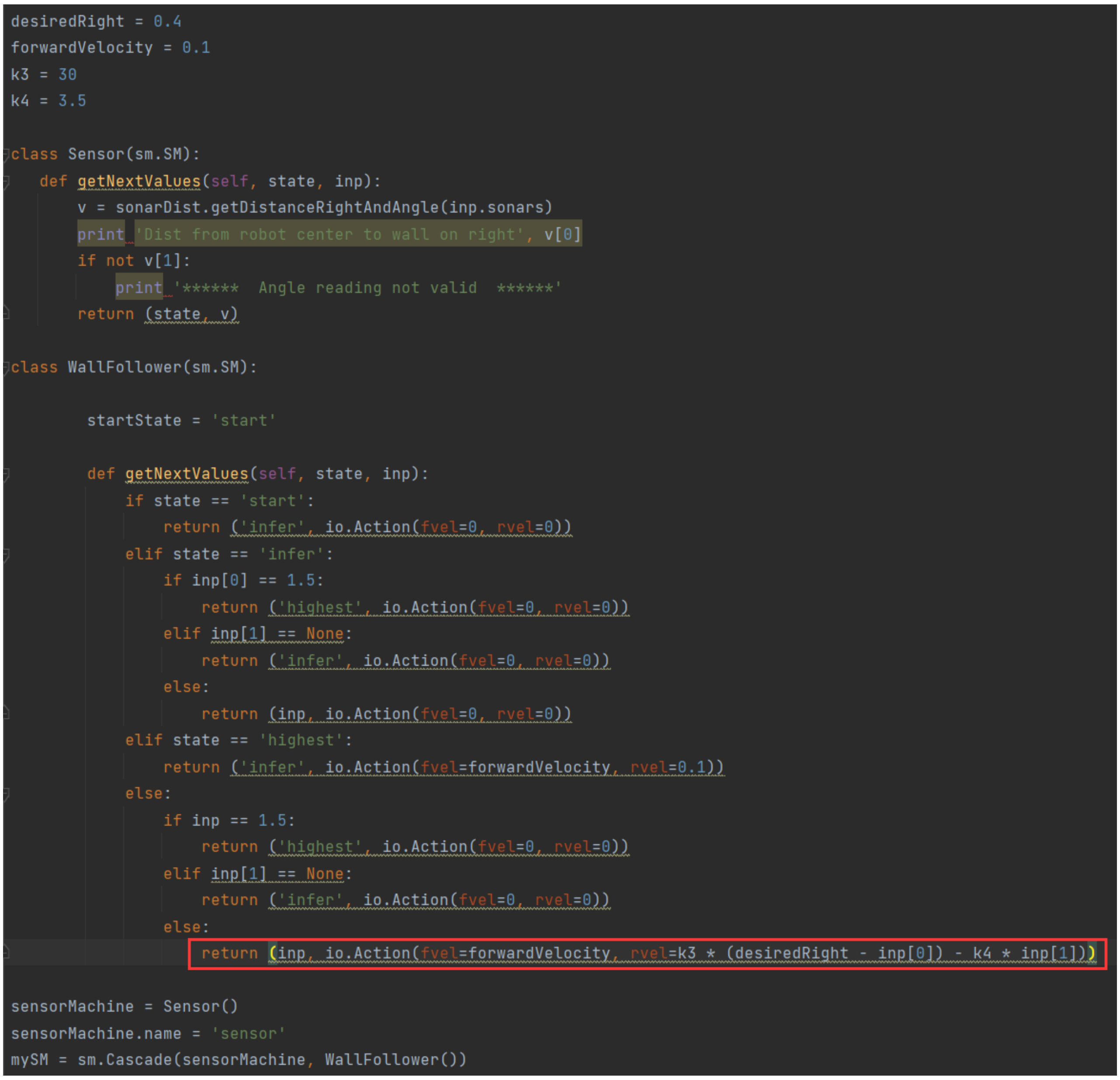
```
>>>
((0.82680000000001308+0.0013266499178875286j), 3.4639999999997335)
>>>
```



```
    print optimize.optOverLine(y, k4Min, k4Max, numSteps)
bestk4(30, -5, 5, 1000000)
```

K3	K4	Magnitude of dominant pole
1	0.6	0.97
3	1.1	0.95
10	2.0	0.90
30	3.5	0.83

Step8

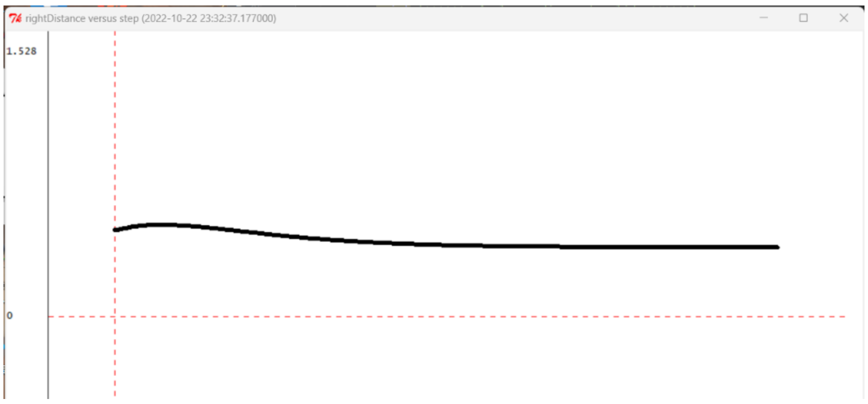


```
desiredRight = 0.4
forwardVelocity = 0.1
k3 = 30
k4 = 3.5

class Sensor(sm.SM):
    def getNextValues(self, state, inp):
        v = sonarDist.getDistanceRightAndAngle(inp.sonars)
        print 'Dist from robot center to wall on right', v[0]
        if not v[1]:
            print '******  Angle reading not valid  ******'
        return (state, v)

class WallFollower(sm.SM):

        startState = 'start'

        def getNextValues(self, state, inp):
            if state == 'start':
                return ('infer', io.Action(fvel=0, rvel=0))
            elif state == 'infer':
                if inp[0] == 1.5:
                    return ('highest', io.Action(fvel=0, rvel=0))
                elif inp[1] == None:
                    return ('infer', io.Action(fvel=0, rvel=0))
                else:
                    return (inp, io.Action(fvel=0, rvel=0))
            elif state == 'highest':
                return ('infer', io.Action(fvel=forwardVelocity, rvel=0.1))
            else:
                if inp == 1.5:
                    return ('highest', io.Action(fvel=0, rvel=0))
                elif inp[1] == None:
                    return ('infer', io.Action(fvel=0, rvel=0))
                else:
                    return (inp, io.Action(fvel=forwardVelocity, rvel=k3 * (desiredRight - inp[0]) - k4 * inp[1]))

sensorMachine = Sensor()
sensorMachine.name = 'sensor'
mySM = sm.Cascade(sensorMachine, WallFollower())
```

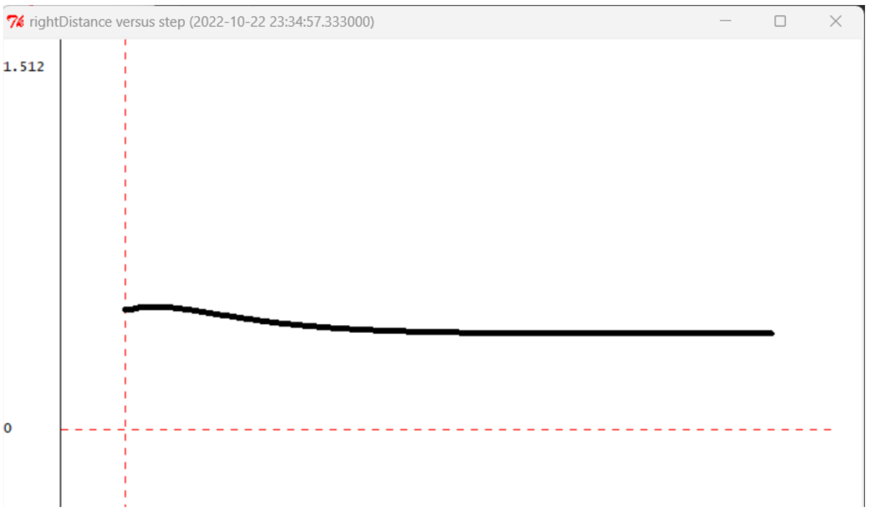
For the failure of sonar 6 and 7, we specifically define the extra state.

The main body of this code is marked by red line.Virtually, it comes from $\omega[n] = k_3 e[n] - k_4 \theta[n]$.

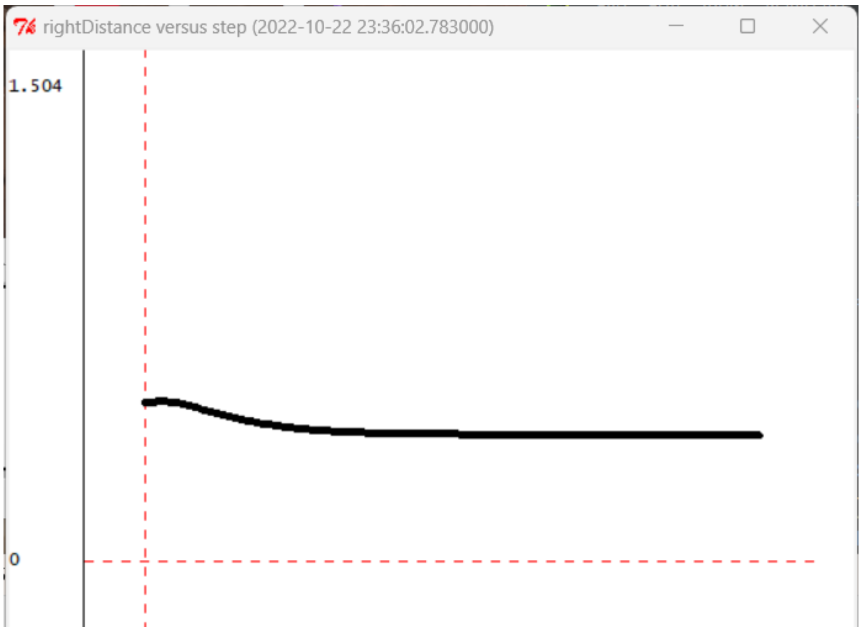
Step9



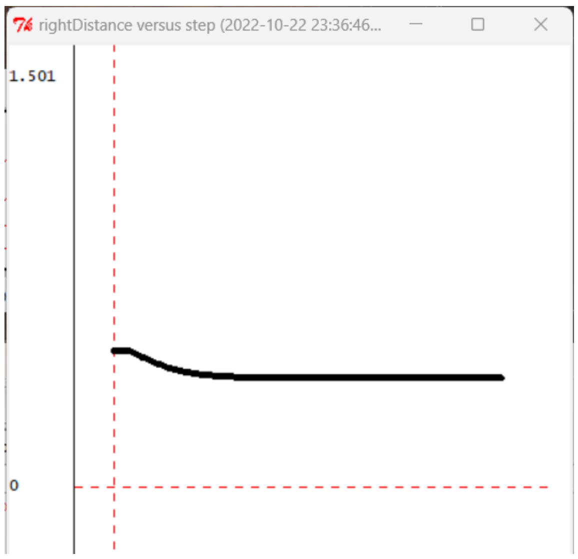
K3 = 1、k4 = 0.6



K3 = 3、k4 = 1.1



K3 = 1、k4 = 2.0



K3 = 30、k4 = 3.5

All the given parameters completed the set task well. We found that with the increase of k3, the convergence speed of the whole system increased faster, and compared with the previous program, we did not need a large gain to get a better effect, which broadened the space for the improvement of the convergence speed of the whole program.

Check Yourself 5.

K3 = 30 and K4 = 3.5 works best in simulation.

NO gain cause bad behavior.

Step9

We did the experiment offline, and the relevant data will be given in Checkoff2 and Check Yourself6

Check Yourself 6.

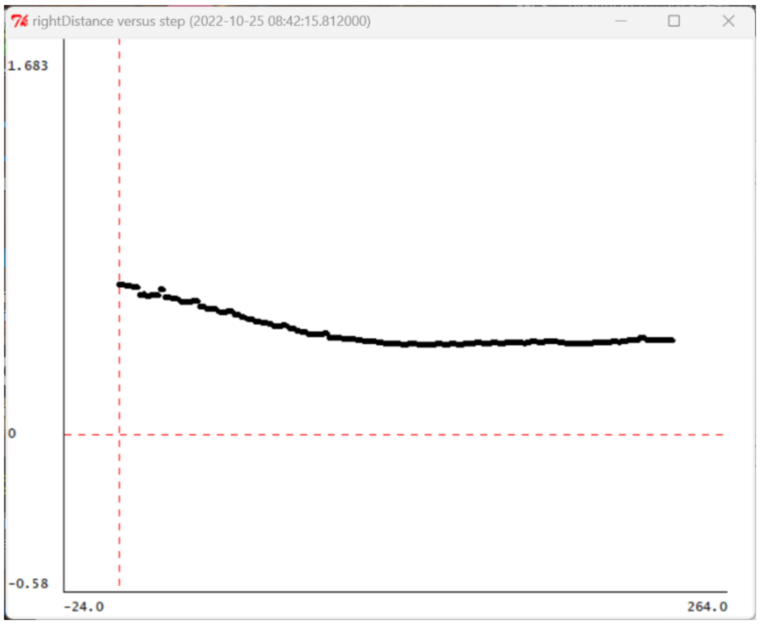
K3 = 10 and K4 = 2.0 works best on a robot.

It is not as same as simulation.

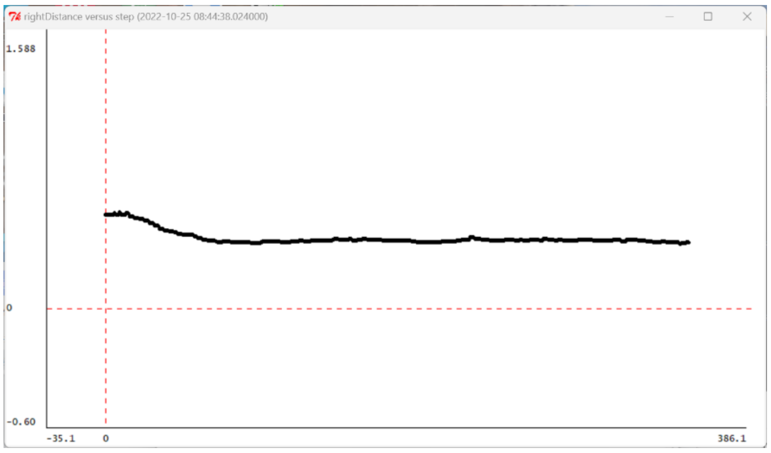
K1 = 30 and K1 = 3.5 causes bad behavior.

Checkoff 2.

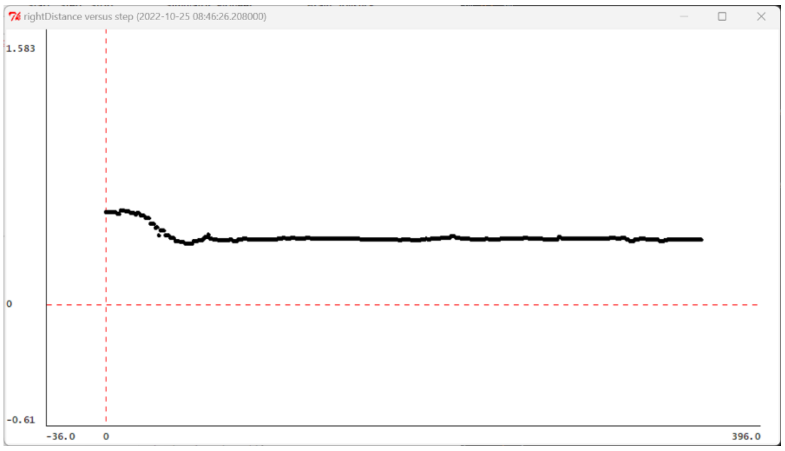
Here is the image of convergence on a robot:



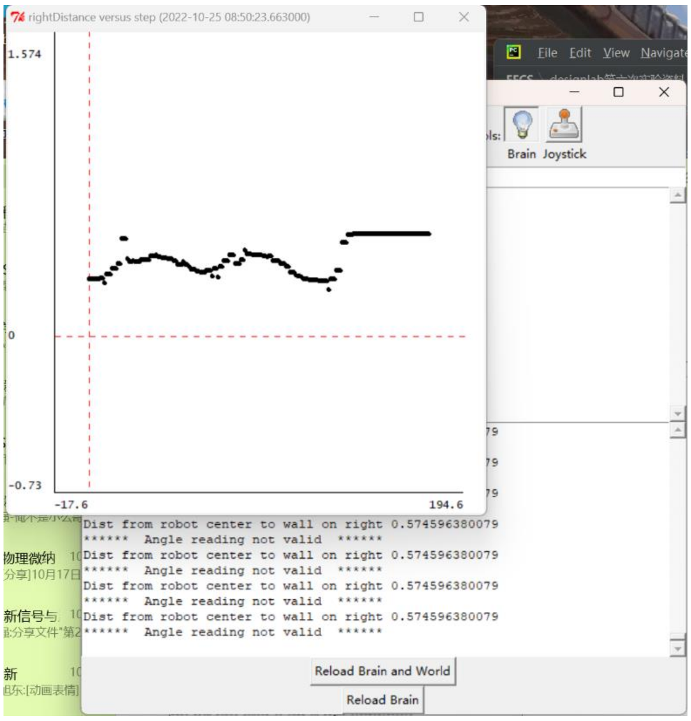
K3 = 1、k4 = 0.6



K3 = 3、k4 = 1.1



K3 = 10、k4 = 2.0



```
Dist from robot center to wall on right 0.574596380079
******  Angle reading not valid  ******
Dist from robot center to wall on right 0.574596380079
******  Angle reading not valid  ******
Dist from robot center to wall on right 0.574596380079
******  Angle reading not valid  ******
Dist from robot center to wall on right 0.574596380079
******  Angle reading not valid  ******
```

K3 = 30、k4 = 3.5

We can see angle-plus-proportional performs better. Angle-plus-proportional system. The accumulation of errors in the last sampling time is avoided, and only the position information at this time is used to ensure the freshness of the position. At the same time, the differential part is introduced to effectively attenuate the oscillation in the process of convergence.

Summary

For delay-plus-proportional we find $K1 \approx -k2$ **,which make**

`omega[n] = k1*e[n] + k2*e[n-1]` **into** `omega[n] = k1 *(e[n]-e[n-1])`

Using PID(proportion integration differentiation) to explain why:

For `omega[n] = k1 *(e[n]-e[n-1])`, **this is a system combined the proportion** and **differentiation, So when using the same scale coefficient to get the best effect(ensuring the unification of the differential) because we are using the amount of time delay differential control, we inevitably affected by the sampling error**

caused by delay, actual sonar in appropriate proportion coefficient can still complete because of the existence of differential control can be realized in certain jitter.

For delay-plus-proportional, we separate the proportion term from the differential phase, and use different physical quantities to control them respectively, so as to achieve their relative independence and improve the stability of the system as a whole. The most important thing is to use only the physical parameters at the present moment, which gets rid of the unstable situation in large proportion control due to insufficient sampling speed.