دانشگاه صنعتی امیرکبیر
(پلی‌تکنیک تهران)

# درس رباتیک
# گزارش تمرین سوم

نگارش
محمدرضا حیدری : **9926053**

استاد درس
دکتر جوانمردی

**خرداد ماه 1403**

## فهرست

## تعریف مسئله

پیاده سازی کنترلر PID بر روی ربات و شبیه سازی در ros , gazebo

حرکت روی مسیر های متفاوت – دنبال کردن یک ربات دیگر – دنبال کردن دیوار

## گام اول

### معرفی توابع و الگوریتم مورد استفاده

- Generate_PID_param

به ازای هر مسیر پارامترهای کنترلر را برای سرعت خطی و زاویه ای بر میگرداند.

- Generate_path

مسیر های مطلوب را میخواهیم ربات بگذراند به صورت لیست بر میگرداند.

- Current_posistion

مکان کنونی ربات را و جهت گیری آن را بر میگرداند.

- Go

هندلر اصلی است و از توابع مذکور استفاده کرده و ربات را به سمت مقصد حرکت می دهد.

```
#!/usr/bin/python3

import rospy
from geometry_msgs.msg import Twist
import tf
from nav_msgs.msg import Odometry
import numpy as np
from math import sqrt, atan2
from math import sqrt
from math import atan2
import  math
import matplotlib.pyplot as plt


class PathFollower():
    def __init__(self,path_number):
        rospy.init_node('path_fallower', anonymous=False)
        rospy.on_shutdown(self.on_shutdown)

        self.goal_x, self.goal_y = self.generate_path(path_number)
        # PID parameters
```

```
        self.k_i,self.k_p,self.k_d,self.k_i_a,self.k_p_a,self.k_d_a =
self.generate_PID_param(path_number)

        self.dt = 0.005
        self.v = 0
        self.D = 2
        rate = 1 / self.dt
        self.r = rospy.Rate(rate)
        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=5)
        self.errs = []
        self.errs1 = []
    def generate_PID_param(self,path_number):
        if path_number == 0:
           k_i = 0
           k_p = 2.5
           k_d = 0
           k_i_a = 0
           k_p_a = 2.5
           k_d_a = 30
        elif path_number == 1:
           k_i = 0.07
           k_p = 0.7
           k_d = 3
           k_i_a = 0.8
           k_p_a = 5
           k_d_a = 80
        elif path_number == 2:
           k_i = 0.01
           k_p = 0.1
           k_d = 1
           k_i_a = 0.4
           k_p_a = 4
           k_d_a = 40
        elif path_number == 3:
           k_i = 0.05
           k_p = 0.5
           k_d = 5
           k_i_a = 0.7
           k_p_a = 7
           k_d_a = 70
        return k_i,k_p,k_d,k_i_a,k_p_a,k_d_a
    def generate_path(self,path_number):
        rospy.loginfo('path:: %d',path_number)
        if path_number == 0:
            X = np.linspace(0, 4 , 100)
```

```
            Y = X
        elif path_number == 1:
            X1 = np.linspace(-1, 1 , 100)
            Y1 = np.array([3]*100)

            X2 = np.linspace(1, 1 + 2**(1/2) , 100)
            Y2 = - (2**(1/2)) * (X2 - 1) + 3

            Y3 = np.linspace(1, -1 , 100)
            X3 = np.array([1 + 2**(1/2)]*100)

            X4 = np.linspace(1 + 2**(1/2), 1, 100)
            Y4 = (2**(1/2)) * (X4 - 1 - 2**(1/2)) -1

            X5 = np.linspace(1, -1 , 100)
            Y5 = np.array([-3]*100)

            X6 = np.linspace(-1, -1 - 2**(1/2) , 100)
            Y6 = - (2**(1/2)) * (X6 + 1) - 3

            Y7 = np.linspace(-1, 1 , 100)
            X7 = np.array([- 1 - 2**(1/2)]*100)

            X8 = np.linspace(-1 - 2**(1/2), -1, 100)
            Y8 = (2**(1/2)) * (X8 + 1 + 2**(1/2)) + 1

            X = np.concatenate([X1, X2, X3, X4, X5, X6, X7, X8])
            Y = np.concatenate([Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8])
        elif path_number == 2:
            X1 = np.linspace(-6., -2 , 50)
            Y1 = np.zeros((50,))

            x_dim, y_dim = 2,2
            t = np.linspace(np.pi, 0, 100)
            X2 = x_dim * np.cos(t)
            Y2 = y_dim * np.sin(t)

            X3 = np.linspace(2, 6 , 50)
            Y3 = np.zeros((50,))

            x_dim, y_dim = 6,6
            t = np.linspace(np.pi*2, np.pi, 200)
            X4 = x_dim * np.cos(t)
            Y4 = y_dim * np.sin(t)
            X = np.concatenate([X1, X2, X3, X4])
```

```
            Y = np.concatenate([Y1, Y2, Y3, Y4])
        elif path_number == 3:
            # logarithmic spiral
            a = 0.17
            k = math.tan(a)
            X , Y = [] , []

            for i in range(150):
                t = i / 20 * math.pi
                dx = a * math.exp(k * t) * math.cos(t)
                dy = a * math.exp(k * t) * math.sin(t)
                X.append(dx)
                Y.append(dy)
            X = np.concatenate([X])
            Y = np.concatenate([Y])
        return X.tolist(), Y.tolist()

    def get_current_position(self):
        msg = rospy.wait_for_message("/odom" , Odometry)
        orientation = msg.pose.pose.orientation
        position_x = msg.pose.pose.position.x
        position_y = msg.pose.pose.position.y
        roll, pitch, yaw = tf.transformations.euler_from_quaternion((
            orientation.x, orientation.y, orientation.z, orientation.w
        ))
        rospy.loginfo('x,y == %f , %f',position_x,position_y)
        return yaw, position_x, position_y

    def go(self):
        sum_i_dist = 0
        prev_dist_error = 0

        sum_i_theta = 0
        prev_theta_error = 0

        move_cmd = Twist()
        move_cmd.angular.z = 0
        move_cmd.linear.x = self.v

        while not rospy.is_shutdown():
            for i in range(len(self.goal_x)):
                while not rospy.is_shutdown() and i<len(self.goal_x):
                    self.cmd_vel.publish(move_cmd)
                    theta_perv, x_perv, y_perv = self.get_current_position()
```

```
                    err = sqrt((self.goal_x[i] - x_perv)**2 + (self.goal_y[i] -
y_perv)**2)
                    if err < 0.1 :
                        move_cmd.angular.z = 0
                        move_cmd.linear.x = 0
                        self.cmd_vel.publish(move_cmd)
                        break

                    self.theta = atan2((self.goal_y[i] - y_perv), (self.goal_x[i]
- x_perv))
                    err_th = (self.theta - theta_perv)
                    while err_th > math.pi:
                        err_th -= 2 * math.pi
                    while err_th < -math.pi:
                        err_th += 2 * math.pi
                    self.errs.append(err)

                    sum_i_dist += err * self.dt
                    sum_i_theta += err_th * self.dt

                    P = self.k_p * err
                    I = self.k_i * sum_i_dist
                    D = self.k_d * (err - prev_dist_error)

                    Pa = self.k_p_a * err_th
                    Ia = self.k_i_a * sum_i_theta
                    Da = self.k_d_a * (err_th - prev_theta_error)

                    prev_dist_error = err
                    prev_theta_error = err_th
                    plant_input = P+I+D
                    rospy.loginfo('plant input : %f',plant_input)
                    plant_a_input = Pa+Ia+Da
                    move_cmd.linear.x = min(plant_input,0.5)
                    move_cmd.angular.z = min(plant_a_input,1.57)

                    self.r.sleep()


    def on_shutdown(self):
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        plt.plot(list(range(len(self.errs))),
                    self.errs, label='errs')
        plt.axhline(y=0,color='R')
```

```
        plt.draw()
        plt.show()

        rospy.sleep(1)
if __name__ == '__main__':
    try:
        path_number = 0
        path_follower = PathFollower(path_number)
        path_follower.go()
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation terminated.")
```

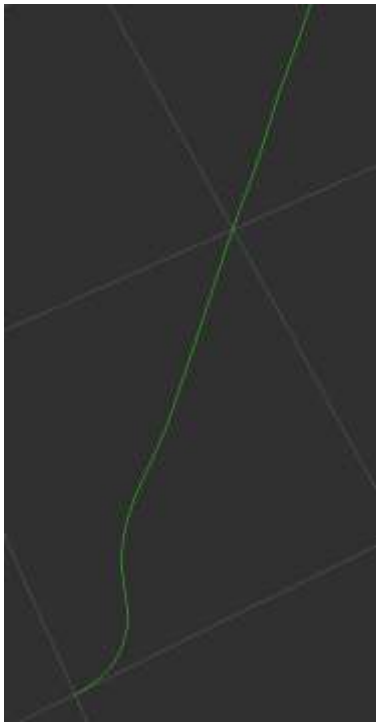
بخش اول – مستقیم الخط

نقطه شروع از مبدا و حرکت به سمت نقطه (4و4) با کنترلر های PD و P و PID

| K_d | K_i | K_p | |
|---|---|---|---|
| 0 | 0 | 2.5 | P |
| 30 | 0 | 2.5 | PD |
| 30 | 0.22 | 2.5 | PID |

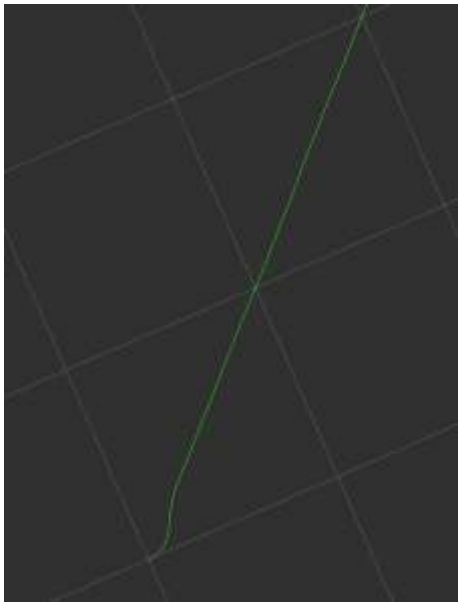
کنترلر تناسبی

با افزایش ضریب پاسخ نوسانی تر و نهایتا ناپایدار می شود.



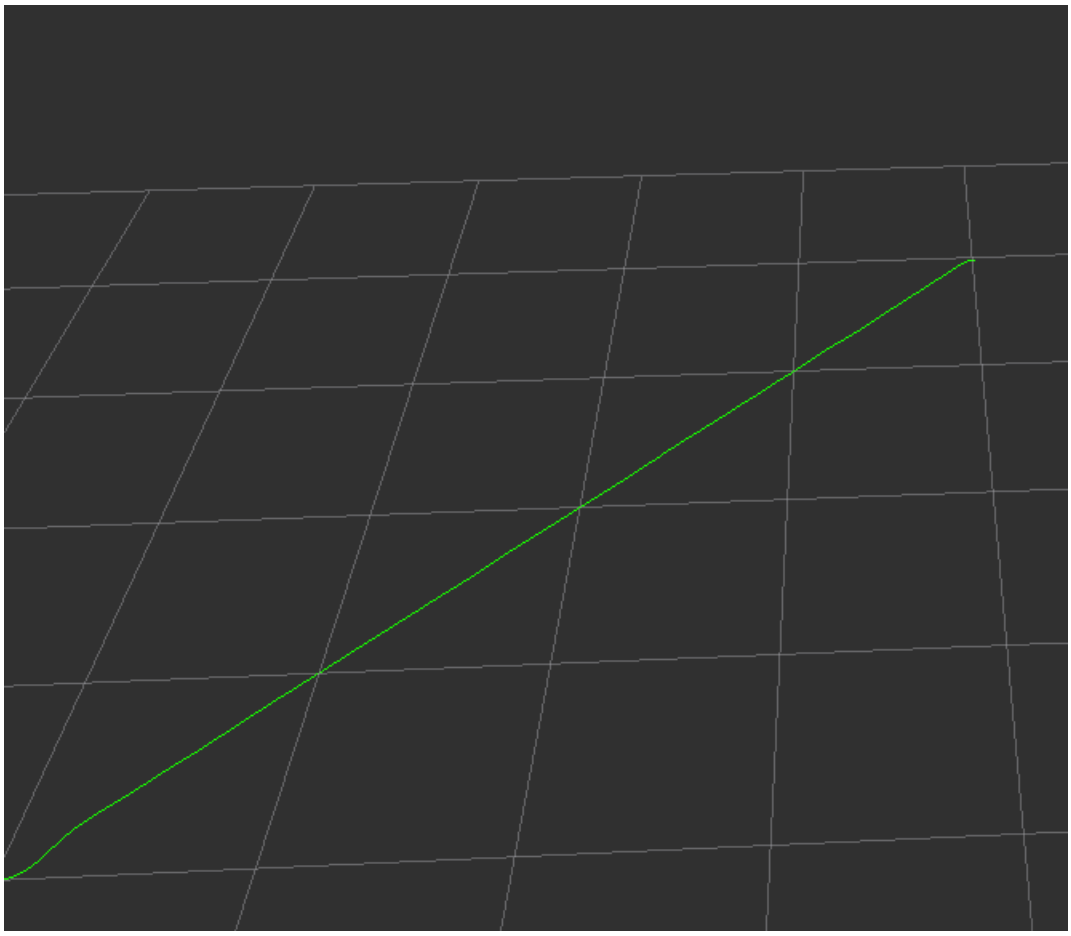
کنترلر تناسبی – مشتق گیر

با افزایش ضریب مشتق گیر پاسخ اورشوت کمتر تر و سرعت سیستم بیشتر می شود.



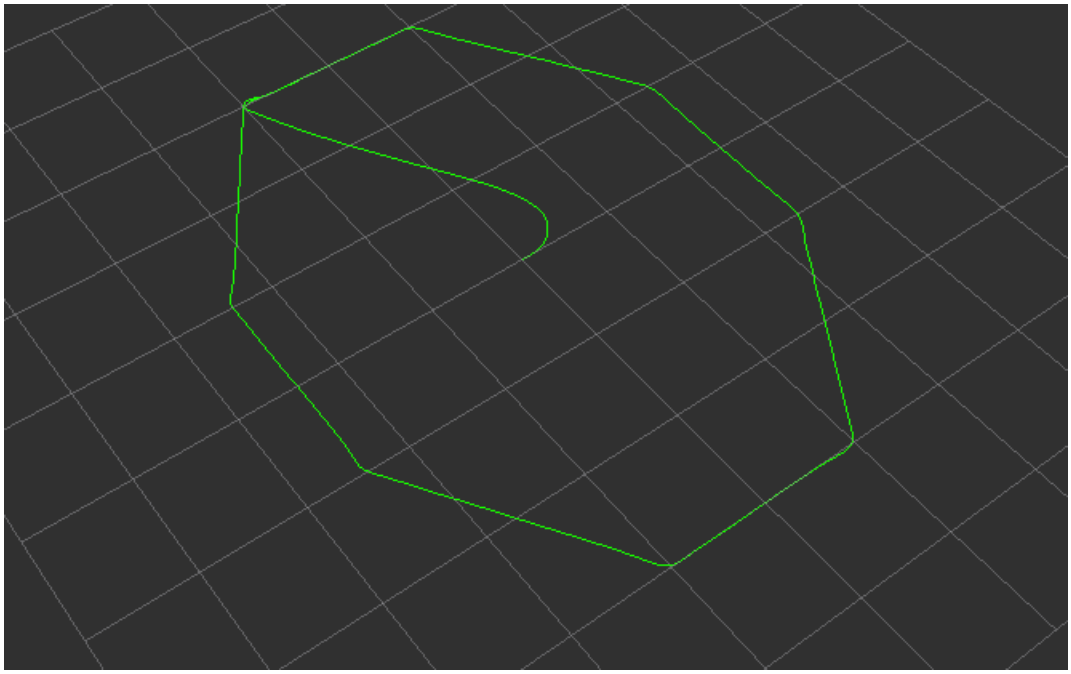
## کنترلر PID

افزایش ضریب انتگرال گیر خطای ماندگار کمتر می شود.

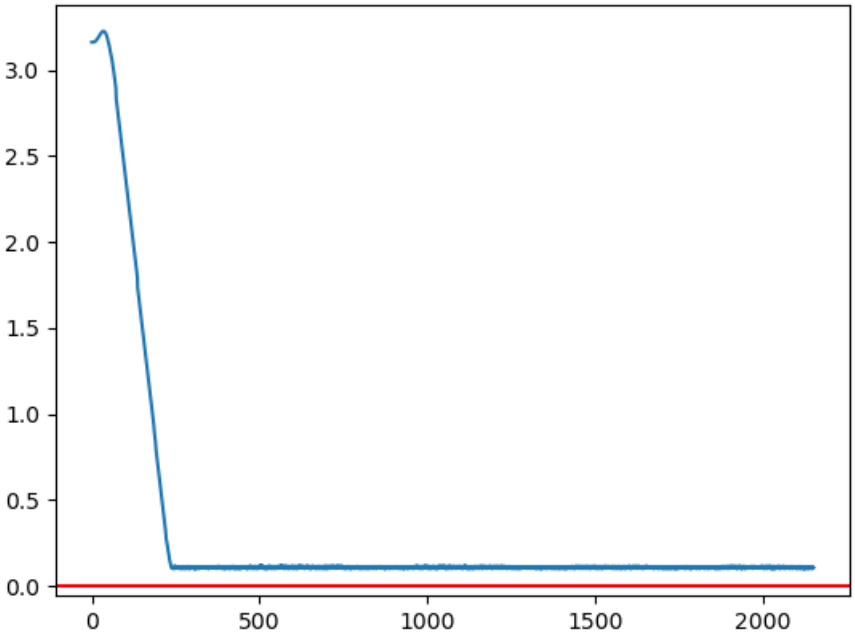


بخش دوم – مسیر هشت ضلعی

| K_d | K_i | K_p | |
|---|---|---|---|
| 5 | 0.05 | 0.5 | PID |



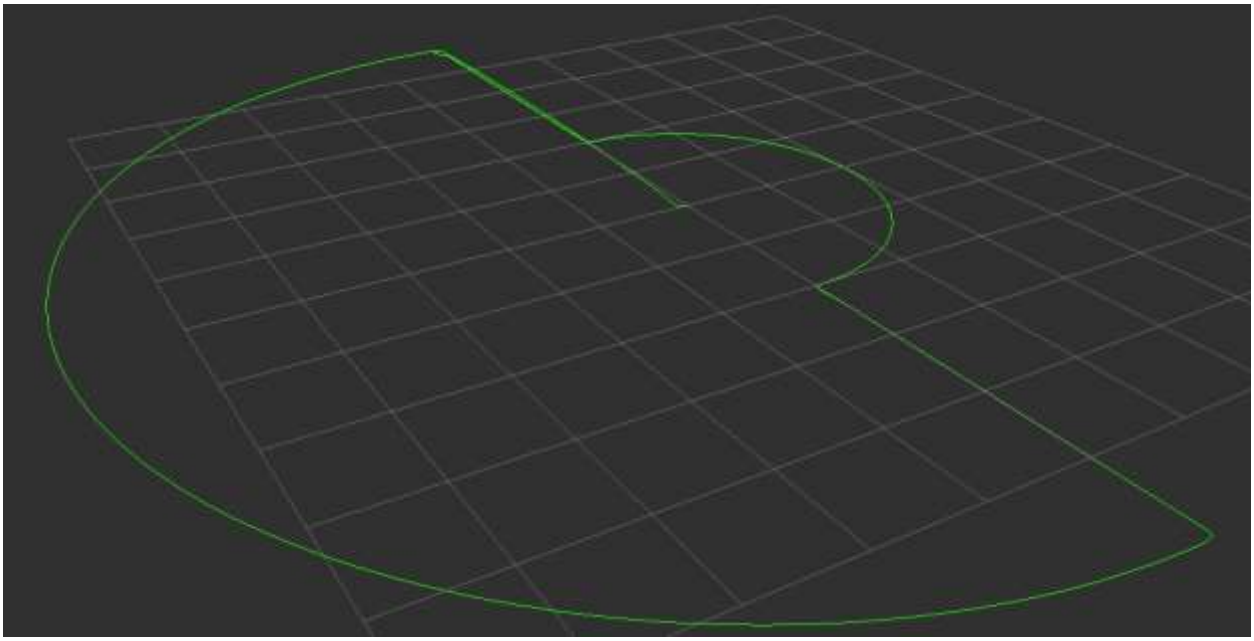
نمای مسیر




نمودار خطا

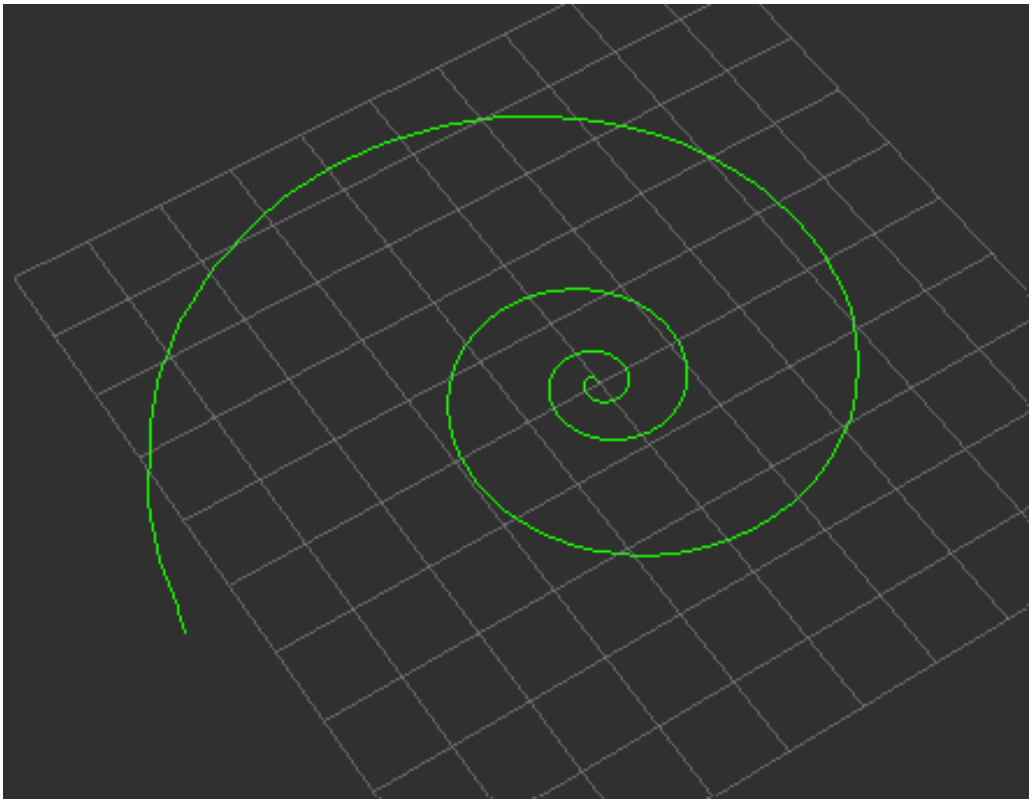
بخش سوم – مسیر دو نیم دایره

| K_d | K_i | K_p | |
|---|---|---|---|
| 0.1 | 0.01 | 0.1 | PID |



بخش چهارم – مسیر لگاریتمی

| K_d | K_i | K_p | |
|---|---|---|---|
| 5 | 0.05 | 0.5 | PID |



## گام دوم

در این گام میخواهیم یک ربات با استفاده از teleop و با keyboard کنترل شود و ربات دیگر به دنبال ربات اول حرکت کند و فاصله نیم را از ربات اول حفظ کند.

### معرفی توابع مورد استفاده

- Odom_data_callback

مختصات هدف که همان ربات اول است را بر میگرداند.

- Get_current_pose

مختصات و جهت گیری کنونی ربات دنبال کننده را بر میگرداند.

- Normalize_angle

جهت نرمال سازی زوایای بیش از 360 و کمتر از 360 استفاده شده است

- Go

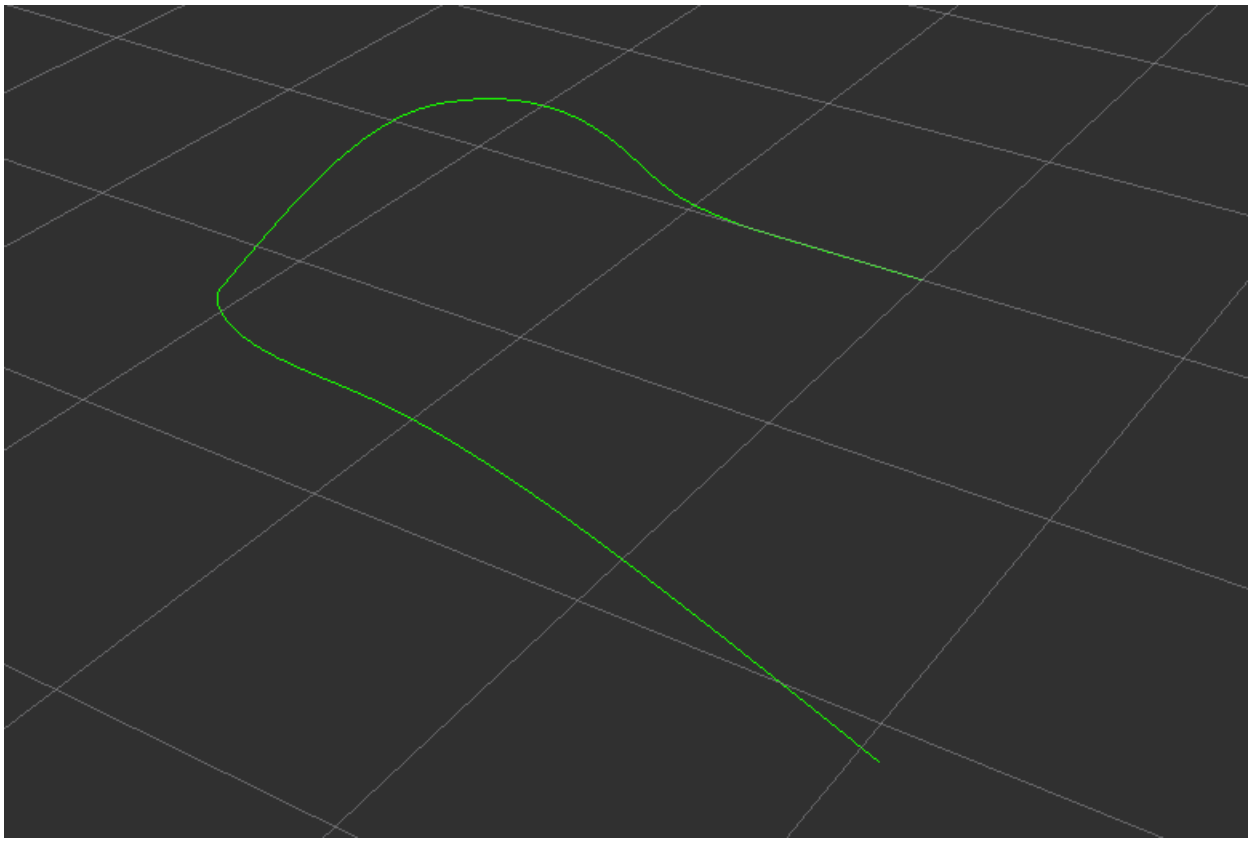
هندلر اصلی ربات است که هدف را با استفاده از توابع مذکور میسر می سازد.

```python
#!/usr/bin/python3

import rospy
from geometry_msgs.msg import Twist
import matplotlib.pyplot as plt
import tf
from nav_msgs.msg import Odometry
from math import sqrt, atan2, pi

class RobotFollower():
    def __init__(self):
        rospy.init_node('robot_follower', anonymous=False)
        rospy.on_shutdown(self.shutdown_procedure)
        rospy.Subscriber("/tb3_0/odom", Odometry, self.odom_data_callback)

        self.target_x = 0
        self.target_y = 0
        self.current_yaw = 0
        self.error_threshold = 0.005

        self.k_i = 0
        self.k_p = 0.2
        self.k_d = 0

        self.k_i_a = 0
        self.k_p_a = 2
        self.k_d_a = 0

        self.dt = 0.005
        self.v = 0
        rate = 1 / self.dt

        self.r = rospy.Rate(rate)
        self.cmd_vel = rospy.Publisher('/tb3_1/cmd_vel', Twist, queue_size=5)
        self.errs = []
        self.angle_errors = []

    def odom_data_callback(self, msg):
        self.target_x = msg.pose.pose.position.x
        self.target_y = msg.pose.pose.position.y
        rospy.loginfo('Target coordinates: x = %f, y = %f', self.target_x,
self.target_y)

    def get_current_pose(self):
        msg = rospy.wait_for_message("/tb3_1/odom", Odometry)
```

```
        orientation = msg.pose.pose.orientation
        position_x = msg.pose.pose.position.x
        position_y = msg.pose.pose.position.y
        roll, pitch, yaw = tf.transformations.euler_from_quaternion((
            orientation.x, orientation.y, orientation.z, orientation.w
        ))
        rospy.loginfo('x,y == %f , %f',position_x,position_y)
        return yaw, position_x, position_y


    def normalize_angle(self, angle):
        while angle > pi:
            angle -= 2 * pi
        while angle < -pi:
            angle += 2 * pi
        return angle

    def go(self):
        sum_i_dist = 0
        prev_dist_error = 0

        sum_i_angle = 0
        prev_angle_error = 0

        move_cmd = Twist()
        move_cmd.angular.z = 0
        move_cmd.linear.x = self.v
        error = 100

        while not rospy.is_shutdown():
            self.cmd_vel.publish(move_cmd)
            rospy.loginfo(move_cmd)
            current_yaw, current_x, current_y = self.get_current_pose()

            error = sqrt((self.target_x - current_x) ** 2 + (self.target_y -
current_y) ** 2)
            rospy.loginfo(f"Target X: {self.target_x}")
            rospy.loginfo(f"Target Y: {self.target_y}")
            if error < 0.5:
                move_cmd.angular.z = 0
                move_cmd.linear.x = 0
                self.cmd_vel.publish(move_cmd)
            else:
                target_angle = atan2((self.target_y - current_y), (self.target_x
- current_x))
```

```
                angle_error = self.normalize_angle(target_angle - current_yaw)

                self.errs.append(error)
                self.angle_errors.append(angle_error)

                sum_i_dist += error * self.dt
                sum_i_angle += angle_error * self.dt
                P = self.k_p * error
                I = self.k_i * sum_i_dist
                D = self.k_d * (error - prev_dist_error)
                Pa = self.k_p_a * angle_error
                Ia = self.k_i_a * sum_i_angle
                Da = self.k_d_a * (angle_error - prev_angle_error)

                prev_dist_error = error
                prev_angle_error = angle_error

                move_cmd.linear.x = self.v + min(P + I + D, 0.7)
                move_cmd.angular.z = min(Pa + Ia + Da, 1.5)

                self.r.sleep()

    def shutdown_procedure(self):
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        plt.plot(list(range(len(self.errs))), self.errs, label='Distance Errors')
        plt.axhline(y=0.5, color='R')
        plt.draw()
        plt.show()
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        controller = RobotFollower()
        controller.go()
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation terminated.")
```

| K_d | K_i | K_p | |
|---|---|---|---|
| 0 | 0 | 0.2 | PID |

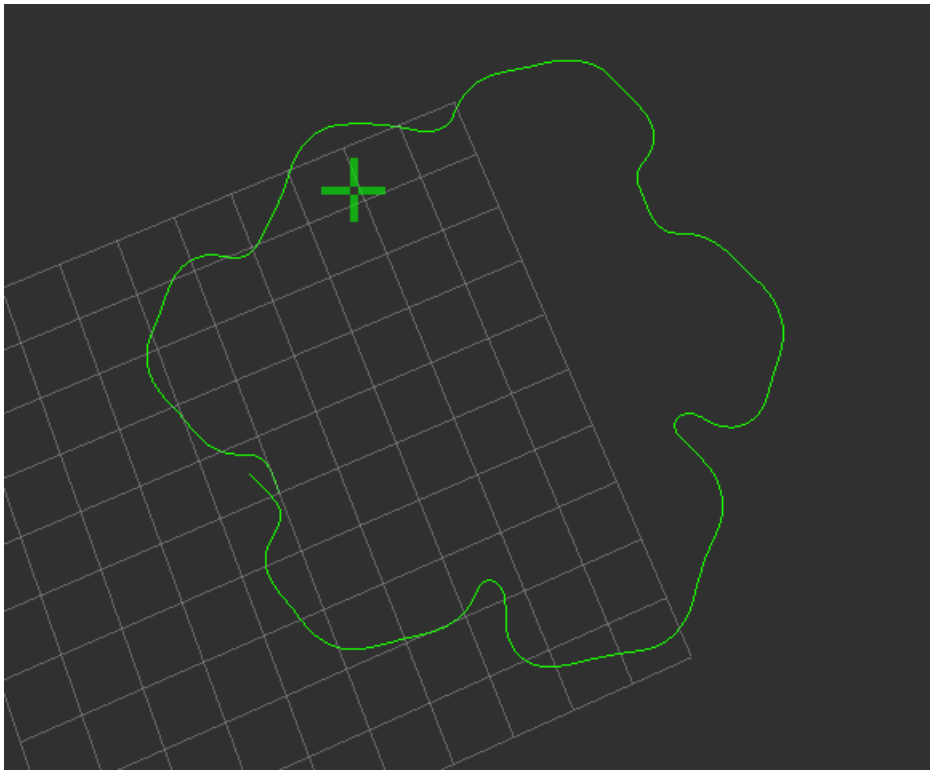


نمای مسیر حرکت ربات دوم

## گام سوم

مطلوب است ربات با اسفاده از الگوریتم دنبالگر دیوار راست یا دیوار چپ و سنسور Laser scan، مسیر دیوار را طی کند.

| K_d | K_i | K_p | |
|---|---|---|---|
| 15 | 0.07 | 0.7 | PID |



نمای مسیر

الگوریتم مورد استفاده و کد دقیقا مشابه کد تدریسیار بوده منتها ضرائب pid برای سرعت زاویه ای و خطی تعیین شده اند.

```
#!/usr/bin/python3

import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
import matplotlib.pyplot as plt

class PIDController():


    def __init__(self):

        rospy.init_node('wall_follower', anonymous=False)
        rospy.on_shutdown(self.on_shutdown)

        self.k_i = 0.07
        self.k_p = 0.7
        self.k_d = 15
        self.k_i_a = 0.8
        self.k_p_a = 5
        self.k_d_a = 80
```

```
        self.dt = 0.005
        self.v = 0.2
        self.D = 1
        rate = 1/self.dt

        self.r = rospy.Rate(rate)
        self.cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=5)
        self.errs = []


    def distance_from_wall(self):
        laser_data = rospy.wait_for_message("/scan" , LaserScan)
        rng = laser_data.ranges[:180]
        d = min(rng)
        return d


    def follow_wall(self):

        d = self.distance_from_wall()
        sum_i_theta = 0
        prev_theta_error = 0

        move_cmd = Twist()
        move_cmd.angular.z = 0
        move_cmd.linear.x = self.v


        while not rospy.is_shutdown():
            self.cmd_vel.publish(move_cmd)

            err = d - self.D
            self.errs.append(err)
            sum_i_theta += err * self.dt

            P = self.k_p * err
            I = self.k_i * sum_i_theta
            D = self.k_d * (err - prev_theta_error)

            rospy.loginfo(f"P : {P} I : {I} D : {D}")
            move_cmd.angular.z = P + I + D
            prev_theta_error = err
            move_cmd.linear.x = self.v
```

```
            rospy.loginfo(f"error : {err} speed : {move_cmd.linear.x} theta :
{move_cmd.angular.z}")

            d = self.distance_from_wall()

            self.r.sleep()

    def on_shutdown(self):
        rospy.loginfo("Stopping the robot...")
        self.cmd_vel.publish(Twist())
        plt.plot(list(range(len(self.errs))),
                    self.errs, label='errs')
        plt.axhline(y=0,color='R')
        plt.draw()
        plt.legend(loc="upper left", frameon=False)
        plt.savefig(f"errs_{self.k_p}_{self.k_d}_{self.k_i}.png")
        plt.show()

        rospy.sleep(1)


if __name__ == '__main__':
    try:
        pidc = PIDController()
        pidc.follow_wall()
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation terminated.")
```