به نام خدا

دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر

اصول علم ربات

تمرین سری سوم

نام و نام خانوادگی	
شماره دانشجویی	
تاریخ ارسال گزارش	۱۴۰۲/۰۲/۲۵

فهرست گزارش سوالات

گام اول

بخش اول

در این بخش ربات را از نقطه (0,0) به نقطه مقصد (10,0) با استفاده از کنترلر PID می رسانیم. برای کنترل خطی و زاویه ای از PID استفاده شده است.

```
def linear_error(self):

    dis = (abs(self.odom_x-self.goal_x)**2 + abs(self.odom_y-self.goal_y)**2)**0.5
    return dis



def angular_error(self):
    x = self.goal_x-self.odom_x
    y = self.goal_y-self.odom_y
    goal_yaw = math.atan2(y,x)


    if self.odom_yaw > 0:
        sign = -1 if (self.odom_yaw - math.pi < goal_yaw < self.odom_yaw) else +1
    else:
        sign = +1 if (self.odom_yaw + math.pi > goal_yaw > self.odom_yaw) else -1

    return sign * (math.pi - abs(abs(self.odom_yaw - goal_yaw) - math.pi))
```

دو تابع برای محاسبه خطا خطی و زاویه ای وجود دارد که فاصله تا نقطه مقصد و اختلاف زاویه فعلی و زاویه هدف تا مقصد را محاسبه می کند.

```python
def control_tward_goal_pose(self):
    twist = Twist()
    sum_angular_error = 0
    sum_linear_error = 0
    prev_angular_error = 0
    prev_linear_error = 0
    while True:
        linear_error = self.linear_error()
        angular_error = self.angular_error()

        if linear_error < self.epsilon:
            return

        sum_angular_error += (angular_error * self.dt)
        sum_linear_error += (linear_error * self.dt)

        # PID for linear speed
        P = self.p_linear * linear_error
        I = self.i_linear * sum_linear_error
        D = self.d_linear * (linear_error - prev_linear_error)
        twist.linear.x =  P + I + D


        # PID for angular speed
        P = self.p_angular * angular_error
        I = self.i_angular * sum_angular_error
        D = self.d_angular * (angular_error - prev_angular_error)
        twist.angular.z =  P + I + D


        prev_angular_error = angular_error
        prev_linear_error = linear_error

        self.cmd_publisher.publish(twist)
        rospy.sleep(self.dt)
```

برای کنترل نیز مطابق ساختار کنترلر PID عمل می کنیم و در هر بازه زمانی مقادیر کنترلی جدید را تولید می کنیم.

همچنین ضرایب مناسب را در هر مرحله برای هر یک از پارامتر های کنترلر تعیین می کنیم.

```python
self.p_linear = 0.1
self.i_linear = 0.007
self.d_linear = 0.1

self.p_angular = 6.0
self.i_angular = 0.0001
self.d_angular = 0.6

self.epsilon = 0.1
self.dt = 0.005
```

برای بررسی کنترلر P، باقی ضرایب را برابر ۰ می گذاریم.

در این کنترلر ربات با سرعت زیاد شروع به حرکت می کند و در نزدیکی نقطه مقصد سرعت بسیار کمی پیدا می کند. همچنین تغییرات زاویه ای ربات نیز دارای نوسان است.

حال کنترلر PD را بررسی می کنیم. این کنترلر تغییر چندانی در سرعت خطی ربات ایجاد نمی کند (هرچند باعث می شود سرعت ابتدایی ربات کمی بیشتر شود). اما با اضافه کردن D به کنترل زاویه ای، نوسانات زاویه ربات بسیار کم می شود.

در کنترلر PID، سرعت خطی ربات در نزدیکی مقصد کمی بیشتر شده و مناسب تر می شود. فاکتور I در کنترل زاویه ای ربات کاربرد چندانی ندارد.

بخش دوم

در این بخش مشابه تمرین قبل برای ربات مقصد تولید می کنیم و بعد از کنترل ربات به آن مقصد و رسیدن به آن، مقصد جدید برای ربات تعریف می کنیم.

سرویس GetNextDestination

این سرویس تبادل پیام بین دو نود Mission وController را انجام می دهد. Controller مطابق فرمت سرویس درخواست را فرستاده و Mission نیز با توجه به درخواست دریافت شده، پاسخ متناسب را مطابق فرمت سرویس می فرستد.

```
float64 current_x
float64 current_y
---
float64 next_x
float64 next_y
```

نود Mission

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

import rospy
import random

from part1.srv import GetNextDestination, GetNextDestinationResponse
def gnd_handler(req):
    current_x = req.current_x
    current_y = req.current_y
    next_x = random.uniform(-10,10)
    y_bound = (25 - (abs(next_x-current_x))**2)**0.5
    if abs(next_x-current_x) > 5:
        y_bound=0
    y_upper = current_y + y_bound
    y_lower = current_y - y_bound
    if y_lower < -10 :
        next_y = random.uniform(y_upper , 10)
    elif y_upper > 10:
        next_y = random.uniform(-10 , y_lower)
    else:
        next_y = random.choice([ random.uniform(-10 , y_lower) , random.uniform(y_upper , 10)])

    response = GetNextDestinationResponse()
    response.next_x = next_x
    response.next_y = next_y
    return response



def listener():
    rospy.init_node('mission_node', anonymous=True)
    s = rospy.Service('/GetNextDestination', GetNextDestination, gnd_handler)
    rospy.spin()

if __name__ == "__main__":
    listener()
```

این نود پس از دریافت موقعیت فعلی ربات، موقعیت مقصد جدید را به طور تصادفی تولیدکرده و پاسخ می دهد؛ منتها موقعیت جدید باید حداقل ۵ متر از موقعیت فعلی فاصله داشته باشد. برای این کار ابتدا یک مقدار برای x به طور تصادفی تولید می شود. سپس با توجه به مقدار تولید شده برای x، بازه هایی از y که انتخاب از آنها منجر به ایجاد فاصله حداقل ۵ متری میان موقعیت فعلی و موقعیت مقصد جدید می شود را مشخص کرده، و y را به طور تصادفی از آن بازه ها انتخاب می کنیم. در نهایت مقادیر x وy تولید شده را به عنوان پاسخ بر می گردانیم.

نود Controller

```python
def control_tward_goal_pose(self):

    sum_angular_error = 0
    sum_linear_error = 0
    prev_angular_error = 0
    prev_linear_error = 0
    while True:
        linear_error = self.linear_error()
        angular_error = self.angular_error()

        if linear_error < self.epsilon:
            return

        sum_angular_error += (angular_error * self.dt)
        sum_linear_error += (linear_error * self.dt)
        twist = Twist()
        if angular_error < math.pi/2 and angular_error > -math.pi/2:# if the goal is behind the robot, the robot shuold not move forward.
            # PID for linear speed
            P = self.p_linear * linear_error
            I = self.i_linear * sum_linear_error
            D = self.d_linear * (linear_error - prev_linear_error)
            twist.linear.x =  P + I + D


        # PID for angular speed
        P = self.p_angular * angular_error
        I = self.i_angular * sum_angular_error
        D = self.d_angular * (angular_error - prev_angular_error)
        twist.angular.z =  P + I + D


        prev_angular_error = angular_error
        prev_linear_error = linear_error

        self.cmd_publisher.publish(twist)
        rospy.sleep(self.dt)
```

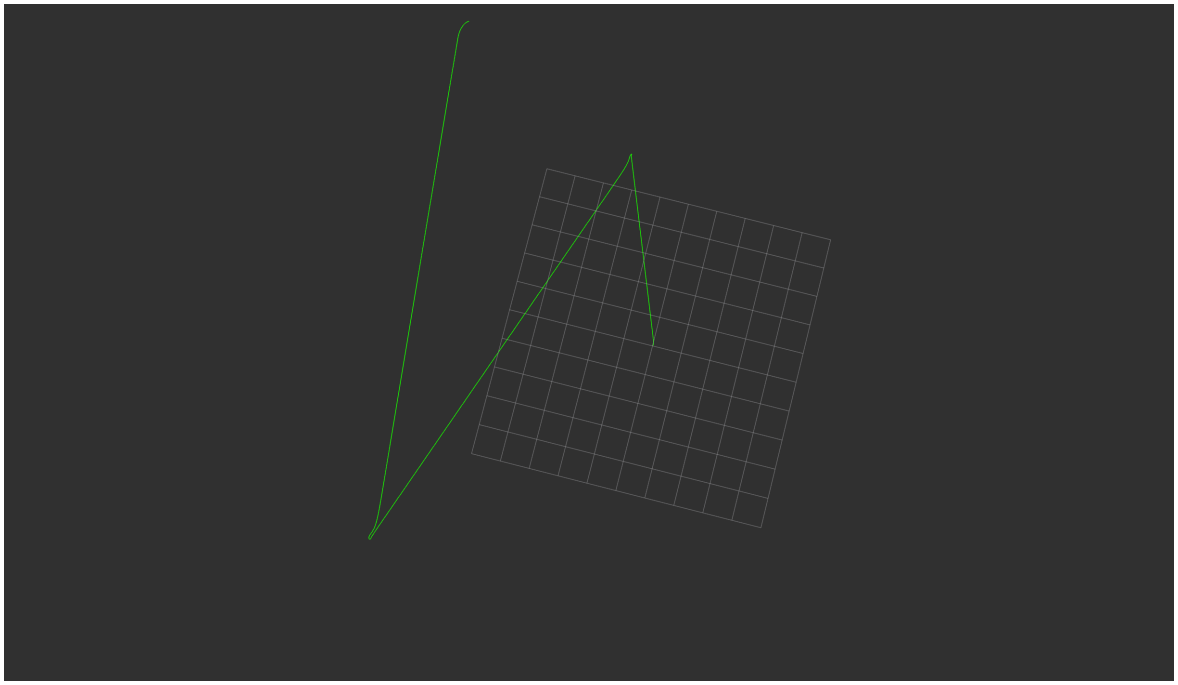
کنترلر این بخش نیز مشابه کنترلر بخش قبل است با این تفاوت که در صورتی که نقطه هدف پشت ربات قرار گرفته باشد؛ سرعت خطی را برابر ۰ قرار می دهد تا ربات به زاویه ای برسد که نقطه هدف پشت آن نباشد. اگر اینکار را انجام ندهیم در حالتی که نقطه هدف پشت ربات است، ربات به جلو حرکت می کند و خطایش را بیشتر می کند.

نود Monitor

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

import rospy
from nav_msgs.msg import Odometry, Path
from geometry_msgs.msg import PoseStamped


class PathMonitor:

    def __init__(self) -> None:

        rospy.init_node("monitor" , anonymous=False)

        self.path = Path()
        self.odom_subscriber = rospy.Subscriber("/odom" , Odometry , callback=self.odom_callback)
        self.path_publisher = rospy.Publisher("/path" , Path , queue_size=10)




    def odom_callback(self, msg : Odometry):
        self.path.header = msg.header
        pose = PoseStamped()
        pose.header = msg.header
        pose.pose = msg.pose.pose
        self.path.poses.append(pose)
        self.path_publisher.publish(self.path)


if __name__ == "__main__":
    path_monitor = PathMonitor()

    rospy.spin()
```

این نود برای نشان دادن مسیر در rviz اجرا می شود. برای این کار یک تاپیک با اسم path تعریف می کنیم که شامل یک آرایه از موقعیت ها است. سپس در هر بار پابلیش شدن موقعیت جدید توسط odom، موقعیت جدید به این آرایه اضافه شده و آن را پابلیش می کند.

نتایج اجرا



در این بخش نکته حائز اهمیت برای تعیین ضرایب مناسب برای پارامتر های Pو D خطی، این است که این پارامتر های باید به گونه ای تنظیم شوند که سرعت اولیه ربات خیلی زیاد نشود(زیرا کنترل زاویه آن دشوار می شود) همچنین در انتها نیز سرعت خیلی کم نشود تا زمان اجرا خیلی طولانی نشود.

گام دوم

در این گام روی مسیری که توسط یک آرایه از موقعیت ها به ما داده می شود حرکت می کنیم.

```python
def shape_rectangle(self):
    X1 = np.linspace(-3, 3 , 100)
    Y1 = np.array([2]*100)

    Y2 = np.linspace(2, -2 , 100)
    X2 = np.array([3]*100)

    X3 = np.linspace(3, -3 , 100)
    Y3 = np.array([-2]*100)

    Y4 = np.linspace(-2, 2 , 100)
    X4 = np.array([-3]*100)

    self.X = np.concatenate([X1,X2,X3,X4])
    self.Y = np.concatenate([Y1,Y2,Y3,Y4])


def shape_star(self):
    X1 = np.linspace(0, 3 , 100)
    Y1 = - (7/3) * X1  + 12

    X2 = np.linspace(3, 10 , 100)
    Y2 = np.array([5]*100)

    X3 = np.linspace(10, 4 , 100)
    Y3 = (5/6) * X3  - (10/3)

    X4 = np.linspace(4, 7 , 100)
    Y4 = -(3) * X4  + 12

    X5 = np.linspace(7, 0 , 100)
    Y5 = -(4/7) * X5  - 5

    X6 = np.linspace(0, -7 , 100)
    Y6 = (4/7) * X6  - 5

    X7 = np.linspace(-7, -4 , 100)
    Y7 = 3 * X7  + 12

    X8 = np.linspace(-4, -10 , 100)
    Y8 = -(5/6) * X8  - (10/3)

    X9 = np.linspace(-10, -3 , 100)
    Y9 = np.array([5]*100)

    X10 = np.linspace(-3, 0 , 100)
    Y10 = (7/3) * X10  + 12

    self.X = np.concatenate([X1,X2,X3,X4,X5, X6, X7, X8, X9, X10])
    self.Y = np.concatenate([Y1,Y2,Y3,Y4,Y5, Y6, Y7, Y8, Y9, Y10])



def shape_spiral(self):
    a = 0.17
    k = math.tan(a)
    self.X , self.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)
        self.X.append(dx)
        self.Y.append(dy)
```

در ابتدا ۳ تابع برای تعریف کردن آرایه مختصات هر یک از این ۳ شکل تعریف می کنیم و در هر اجرا قبل از شروع حرکت ربات، تابع مربوط به مسیر مورد نظرمان را صدا می زنیم.

کنترلر این بخش مشابه بخش قبل است.

دو تابع برای تنظیم هدف این بخش تعریف می کنیم:

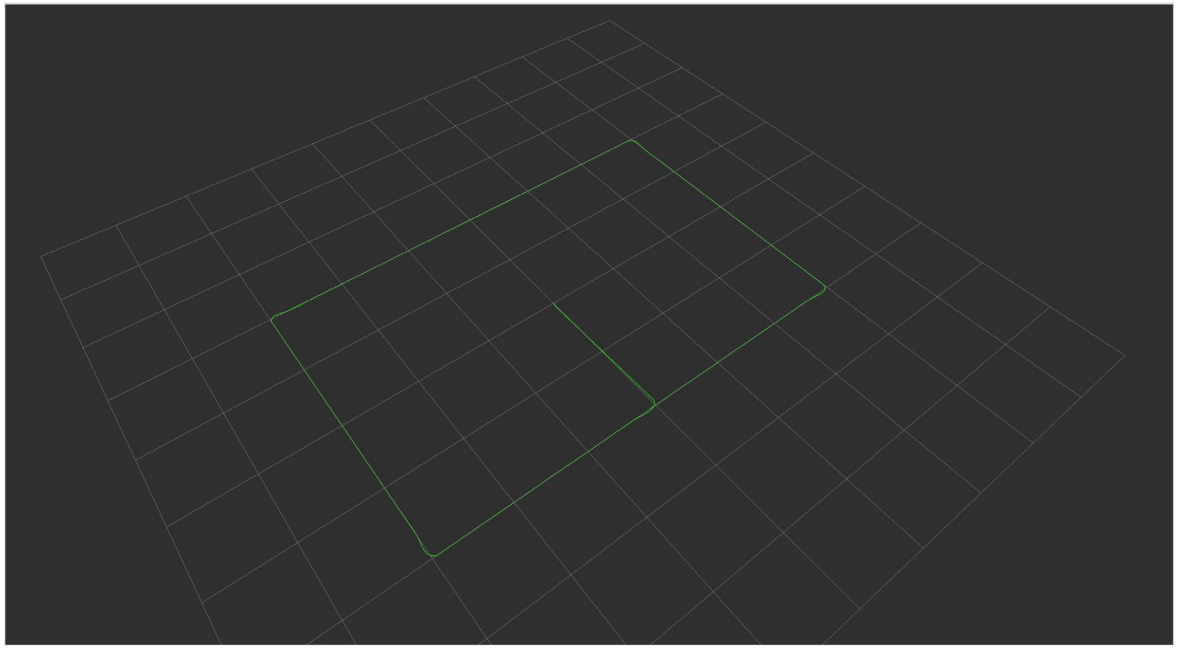
```python
def set_closest_as_goal(self):
    min = 1000
    min_x = 0
    min_y = 0
    min_i=0

    for i in range(len(self.X)):
        distance = self.calculate_distance(self.odom_x, self.odom_y, self.X[i], self.Y[i])
        if distance < min:
            min = distance
            min_x = self.X[i]
            min_y = self.Y[i]
            min_i = i

    self.goal_x = min_x
    self.goal_y = min_y
    self.first_index = min_i
    self.current_index = min_i
```

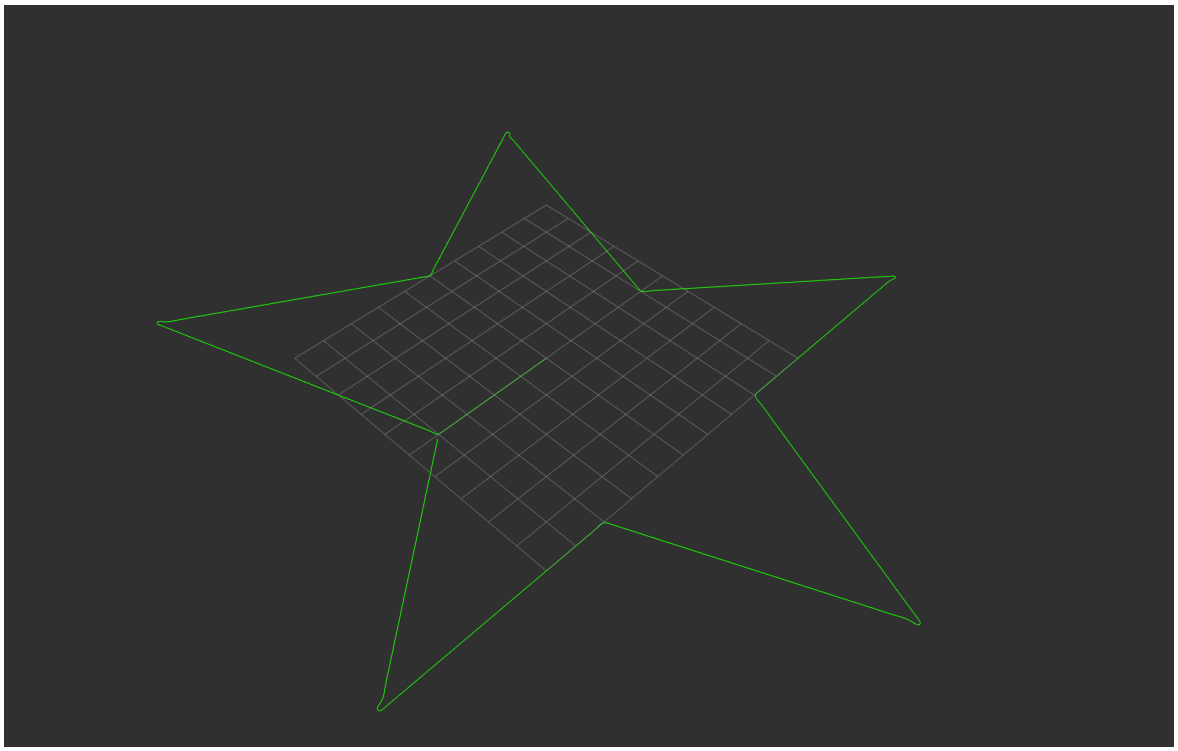
این تابع در ابتدا اجرا صدا زده می شود و نقطه هدف را نزدیکترین نقطه مسیر به ربات قرار می دهد. همچنین اندیس این نقطه را دخیره می کند تا هنگامی که در طول اجرا دوباره به این نقطه رسیدیم؛ اجرا را متوقف کنیم.

```python
def set_next_goal(self):
    self.current_index = (self.current_index + 1) % len(self.X)
    if self.current_index == self.first_index:
        return False

    self.goal_x = self.X[self.current_index]
    self.goal_y = self.Y[self.current_index]
    return True
```

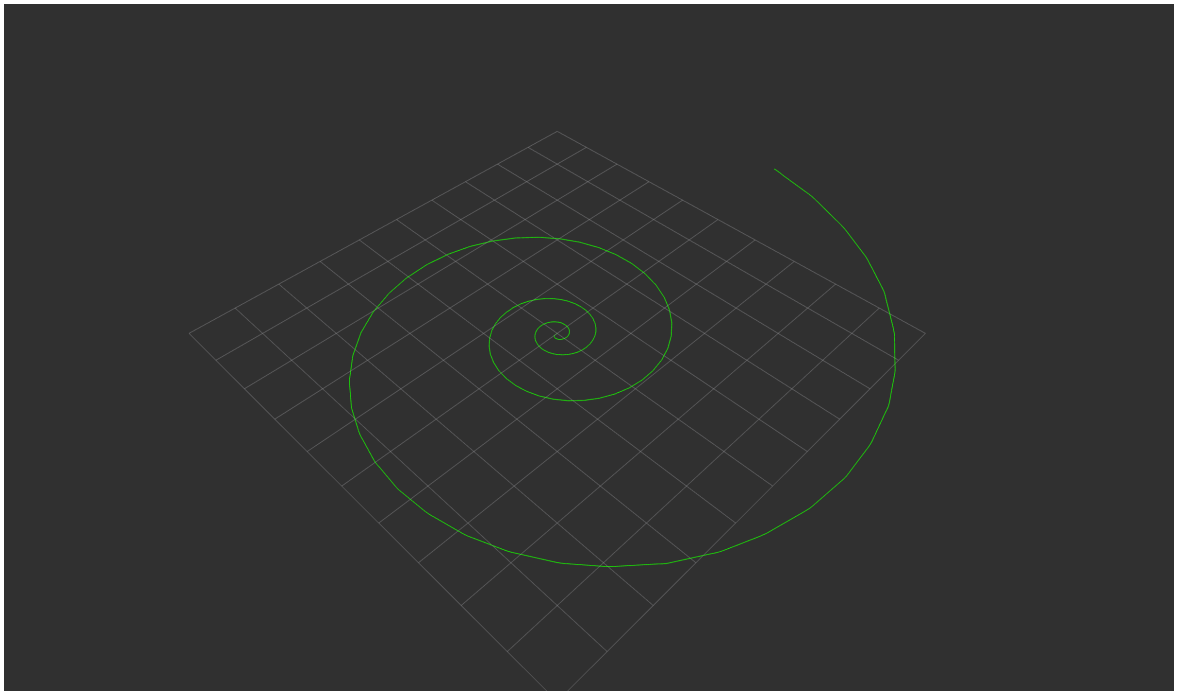
این تابع پس رسیدن به نقطه هدف، برای تعیین نقطه هدف بعدی صدا زده می شود و نقطه با اندیس بعدی را به عنوان مقصد جدید قرار می دهد. در صورتی که هدف جدید وجود داشته باشد، مقدار True و در غیر اینصورت False بر می گرداند تا ربات متوقف شود.

نتایج اجرا



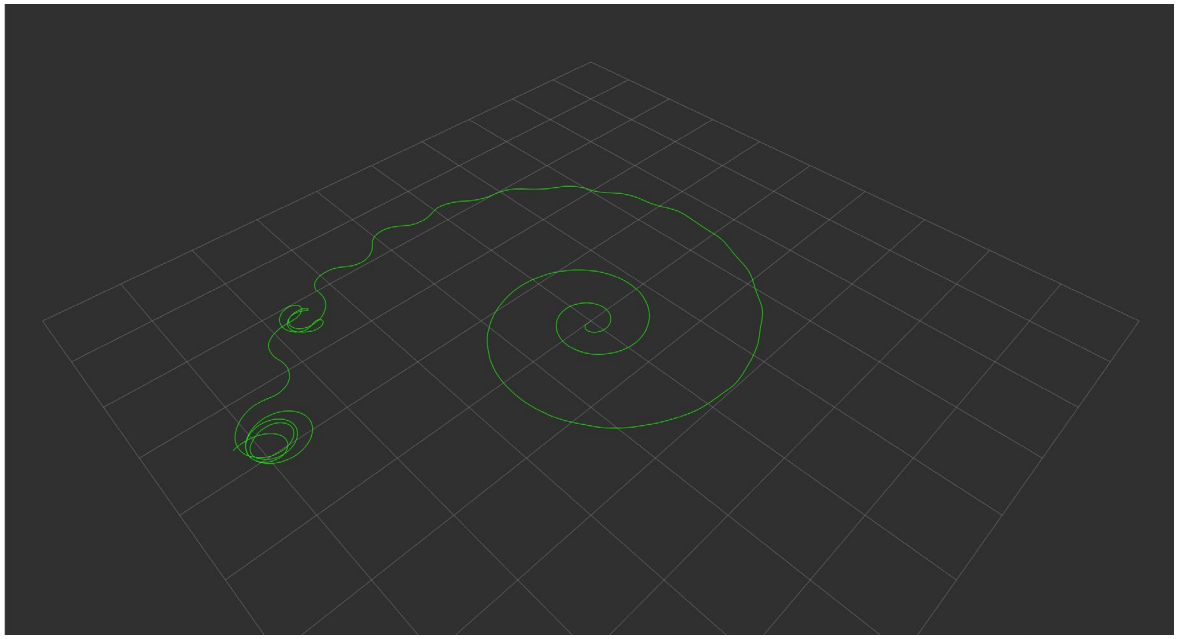


در این شکل خطای odometry در نشان دادن مسیر مشهود است.



نکته حائز اهمیت برای تعیین ضرایب پارامتر های کنترلی در این بخش این است که ضرایب مربوط به P و I برای سرعت خطی را نسبت به گام قبلی بزرگتر قرار دهیم، زیرا فاصله نقاط در این بخش بسیار کم است و در غیر اینصورت؛ اجرا بسیار طولانی می شود. همچنین تعیین ضرایب مناسب P و D برای کنترل زاویه ای نیز تاثیر در دقت در پیچ ها دارد به طوری که در صورتی که این ضرایب را زیاد قرار دهیم، ربات در طول مسیر خط صاف خود نوسانات شدید تری دارد اما در شکستگی ها دقیق تر می پیچد(کمتر گرد می شود) و در صورتی که این ضرایب را کم قرار دهیم، ربات حرکت نرم تری در طول خط صاف دارد و در نقاط شکستگی گرد تر می پیچد.

برای مسیر مارپیچ، ضرایب کنترل خطی را باید کمی کمتر از سایر مسیر ها قرار دهیم زیرا، در بخش هایی از این مسیر فاصله نقاط از هم زیاد شده و در نتیجه ربات از مسیر خارج می شود. همچنین ضرایب کنترل زاویه ای را نیز باید بیشتر از مسیر های قبلی قرار دهیم زیرا ربات باید بعد از رسیدن به هر نقطه از مسیر، سریع زاویه اش را مطابق با نقطه بعد تغییر دهد تا این انحراف زاویه به نقاط بعدی مسیر منتقل نشود.



نمونه اجرای نامناسب در صورت زیاد قرار دادن ضرایب مربوط به کنترلر خطی و کم بودن ضرایب کنترل زاویه ای

گام سوم

در این گام ربات ما باید در راستای یک دیوار حرکت کرده و یک فاصله ثابت با دیوار را حفظ کند.

ربات باید ابتدا مستقیم حرکت کند تا دیوار را پیدا کرده و به فاصله مشخص از دیوار برسد. سپس در راستای دیوار حرکت کند. لذا حرکت ربات به دو بخش تقسیم می شود.

در بخش اول فاصله جلوی ربات تا دیوار را محاسبه می کنیم و تا رسیدن ربات به فاصله مشخص، به حرکت ادامه می دهیم. برای رسیدن به فاصله مشخص از دیوار از PID روی سرعت خطی استفاده می کنیم.

```python
def error_from_front(self):
    laser_data = rospy.wait_for_message("/scan" , LaserScan)
    rng = laser_data.ranges[0:90] + laser_data.ranges[270:360]
    d = min(rng)
    return d - self.distance_from_wall


def control_toward_wall(self):
    sum_linear_error = 0
    prev_linear_error = 0
    while True:
        linear_error = self.error_from_front()
        if linear_error < self.epsilon:
            return

        sum_linear_error += (linear_error * self.dt)
        twist = Twist()
        # PID for linear speed
        P = self.p_linear * linear_error
        I = self.i_linear * sum_linear_error
        D = self.d_linear * (linear_error - prev_linear_error)
        twist.linear.x =  P + I + D

        prev_linear_error = linear_error

        self.cmd_publisher.publish(twist)
        rospy.sleep(self.dt)
```

بعد از رسیدن به دیوار، بخش دوم حرکت که حرکت در امتداد دیوار است را آغاز می کنیم. توجه شود که می توانستیم قبل از شروع این بخش، ۹۰ درجه بچرخیم(تا ربات در امتداد دیوار قرار بگیرد) و سپس این بخش را آغاز کنیم. با این حال الگوریتم کنترل ما در این بخش باید بتواند در حالتی ربات رو به روی دیوار قرار دارد نیز، کنترل مناسب را انجام دهد تا در نهایت ربات در امتداد دیوار قرار گرفته و با فاصله مشخص از دیوار حرکت کند؛ لذا این چرخش را انجام ندادیم تا این قابلیت الگوریتم مشخص باشد.

خطای ربات در حالت تعقیب دیوار با استفاده از (حداقل) فاصله ربات تا دیوار و فاصله هدف تا دیوار تعیین می شود. همچنین در صورتی که فاصله بیش از ۲۰ گزارش شود، همان مقدار ۲۰ را گزارش می کنیم تا از تغییر زاویه های شدید جلوگیری کنیم.

```python
def error_from_left(self):
    laser_data = rospy.wait_for_message("/scan" , LaserScan)
    rng = laser_data.ranges[:100]
    d = min(rng)
    error =  d - self.distance_from_wall
    if error < 20.0 :
        return error
    else:
        return 20.0
```

همچنین برای کنترل ربات برای تعقیب دیوار، سرعت خطی را برابر مقدار ثابتی قرار می دهیم و برای سرعت زاویه ای از PID استفاده می کنیم.

```
def control_wallFollow(self):

    sum_angular_error = 0
    prev_angular_error = 0
    while True:
        angular_error = self.error_from_left()
        sum_angular_error += (angular_error * self.dt)
        twist = Twist()
        twist.linear.x =  self.linear_speed  # constant speed during wall following

        # PID for angular speed
        P = self.p_angular * angular_error
        I = self.i_angular * sum_angular_error
        D = self.d_angular * (angular_error - prev_angular_error)
        twist.angular.z =  P + I + D


        prev_angular_error = angular_error

        self.cmd_publisher.publish(twist)
        rospy.sleep(self.dt)
```

نتایج اجرا

تعیین ضرایب مناسب برای کنترلر در این بخش بسیار حساس تر است. برای اینکه ربات پیچ ها را به خوبی بپیچد (زیاد گرد نکند) باید ضریب P را نسبتا زیاد قرار دهیم. این کار باعث می شود ربات در طول حرکت خط صاف خود، نوسان زاویه زیاد داشته باشد؛ برای حل این موضوع ضریب D را نیز باید بیشتر کنیم تا از نوسانات جلوگیری شود.