به نام خدا

دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر

اصول علم ربات

تمرین سری دوم

نام و نام خانوادگی	
شماره دانشجویی	
تاریخ ارسال گزارش	۱۴۰۲/۰۱/۱۸

فهرست گزارش سوالات

گام اول

در این قسمت باید ربات در هر مرحله با دریافت موقعیت مقصد، به سمت آن موقعیت چرخیده و سپس مستقیم به سمت آن حرکت کند. برای این کار از یک سرویس GetNextDestination و دو نود Controller وMission استفاده 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

این نود را به عنوان یک کلاس تعریف می کنیم

```
class Controller:

    def __init__(self) -> None:

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

        self.odom_subscriber = rospy.Subscriber("/odom" , Odometry , callback=self.odom_callback)
        self.cmd_publisher = rospy.Publisher('/cmd_vel' , Twist , queue_size=10)

        # getting specified parameters
        self.linear_speed = rospy.get_param("linear_speed") # m/s

        self.angular_speed = 0.2
        self.epsilon_linear = 0.1
        self.epsilon_angular = 0.01
        self.odom_yaw = 0
        self.odom_x=0
        self.odom_y=0
        self.goal_x=5
        self.goal_y=5


        # defining the states of our robot
        self.GO, self.ROTATE = 0, 1
        self.state = self.ROTATE
```

instructor این کلاس در ابتدا مشخص می کند که این نود تاپیک odom را subscribe کرده تا موقعیت ربات را بدست آورد و به تاپیک cmd_vel، publish می کند تا ربات را حرکت دهد. سپس مقدار سرعت خطی را از پارامتر های فایل launch مربوطه مقدار دهی کرده و بقیه پارامتر ها را مقدار دهی اولیه می کند. منظور از epsilon_linear و epsilon_angular، مقدار حداقل خطای پذیرش شده است که برای کنترل حرکات ربات از آن استفاده خواهیم کرد. این کلاس دو ضعیت GO و ROTATE دارد که در هرکدام از آنها کار متناسب با آن را انجام می دهد. وضعیت ابتدایی ربات ROTATE است.

```python
def odom_callback(self, msg: Odometry):


    orientation = msg.pose.pose.orientation
    # convert quaternion to odom
    roll, pitch, yaw = tf.transformations.euler_from_quaternion((
        orientation.x ,orientation.y ,orientation.z ,orientation.w
    ))

    self.odom_yaw = yaw
    self.odom_x = msg.pose.pose.position.x
    self.odom_y = msg.pose.pose.position.y
```

تابع odom_callback زمانی که به تاپیک odom، پابلیش شود صدا زده می شود. این تابع موقعیت x وy ربات را از تاپیک odom خوانده و به عنوان موقعیت فعلی ربات آن را قرار می دهد. همچنین با استفاده از پکیج tf، مقادیر quaternion مربوط به زاویه ربات را به مقادیر euler تبدیل کرده و مقدار yaw را به عنوان زاویه فعلی ربات قرار می دهد.

```python
def recieve_goal_pose(self):
    rospy.wait_for_service('/GetNextDestination')
    resp = rospy.ServiceProxy('/GetNextDestination', GetNextDestination)(self.odom_x,self.odom_y)
    self.goal_x = resp.next_x
    self.goal_y = resp.next_y
```

تابع receive_goal_pose، در هربار صدا زده شدن، موقعیت فعلی ربات را در قالب سرویس GetNextDestination فرستاده و موقعیت جدید را از آن می گیرد و به عنوان موقعیت هدف جدید قرار می دهد.

```python
def calculate_goal_angle(self):
    x = self.goal_x-self.odom_x
    y = self.goal_y-self.odom_y
    angle = math.atan2(y,x)

    return angle
```

این تابع با محاسبه arctan تفاضل موقعیت های هدف و فعلی ربات، زاویه ای را که ربات برای رسیدن به هدف باید در آن قرار بگیرد را محاسبه می کند.

```python
def distance_calculator(self, x1 ,y1 ,x2 ,y2):
    dis = (abs(x1-x2)**2 + abs(y1-y2)**2)**0.5
    return dis
```

این تابع دو موقعیت را گرفته و فاصله این دو را به عنوان خروجی بر می گرداند.

```python
def run(self):
    rospy.loginfo("run started")

    count  = 0
    error_sum = 0.0
    while not rospy.is_shutdown():

        # check whether state is changed or not
        if self.state == self.GO:
            count +=1
            twist = Twist()
            twist.linear.x = self.linear_speed
            min = 40.0
            current = 50.0
            while((current := self.distance_calculator(self.odom_x,self.odom_y,self.goal_x,self.goal_y)) - min < self.epsilon_linear):
                rospy.loginfo("distance %f  ",current)
                if current < min:
                    min = current

                self.cmd_publisher.publish(twist)

            self.cmd_publisher.publish(Twist())
            rospy.sleep(2)
            error_sum += self.distance_calculator(self.odom_x,self.odom_y,self.goal_x,self.goal_y)
            self.state=self.ROTATE

        if count == 5:
            self.cmd_publisher.publish(Twist())
            error = error_sum/count
            rospy.loginfo("average error: %f" , error)
            rospy.sleep(2)
            break

        if self.state == self.ROTATE:
            self.recieve_goal_pose()
            goal_yaw = self.calculate_goal_angle()

            if (goal_yaw - self.odom_yaw <= math.pi and goal_yaw - self.odom_yaw >0) or goal_yaw - self.odom_yaw <= -math.pi:
                twist = Twist()
                twist.angular.z = self.angular_speed

            elif goal_yaw - self.odom_yaw == 0:
                twist = Twist()
            else :
                twist = Twist()
                twist.angular.z = -self.angular_speed


            while(abs(self.odom_yaw-goal_yaw) > self.epsilon_angular):
                self.cmd_publisher.publish(twist)

            self.cmd_publisher.publish(Twist())
            rospy.sleep(2)
            self.state=self.GO
```

این تابع در main، پس از ساختن یک instance از کنترلر،روی همان instance صدا زده می شود.

در این تابع کارکرد های دو وضعیت GO و ROATE پیاده سازی شده اند. در وضعیت GO ربات یک twist که فقط سرعت خطی در راستای x دارد تولید کرده و در یک حلقه while تا زمانی که فاصله ربات تا هدف به مقدار کمینه (به اضافه یک حد کوچک چون ممکن است odom در خواندن موقعیت خطا داشته باشد) نرسیده باشد، پابلیش می کند. مقدار کمینه نیز در صورت تغییر در هر اجرای حلقه به روزرسانی می شود. بعد از آن ربات متوقف شده؛ فاصله ربات تا مقصد را به عنوان خطا در نظر گرفته و وضعیت ربات را برابر ROTATE قرار می دهیم.

سپس بررسی می کنیم در صورتی که ۵ بار ربات به مقاصد تعریف شده حرکت کرده، خطای میانگین را محاسبه کرده و اجرا را متوقف می کنیم.

در وضعیت ROTATE ربات مقصد جدید را دریافت کرده، زاویه متناسب برای رسیدن به مقصد را محاسبه کرده، و جهت بهینه برای چرخش را تشخیص می دهد. سپس در یک حلقه while، تا رسیدن به

زاویه مورد نظر دوران میکند. بعد از رسیدن به زاویه هدف، ربات توقف کرده و وضعیت را برابر GO قرار می دهد.

نود Monitor

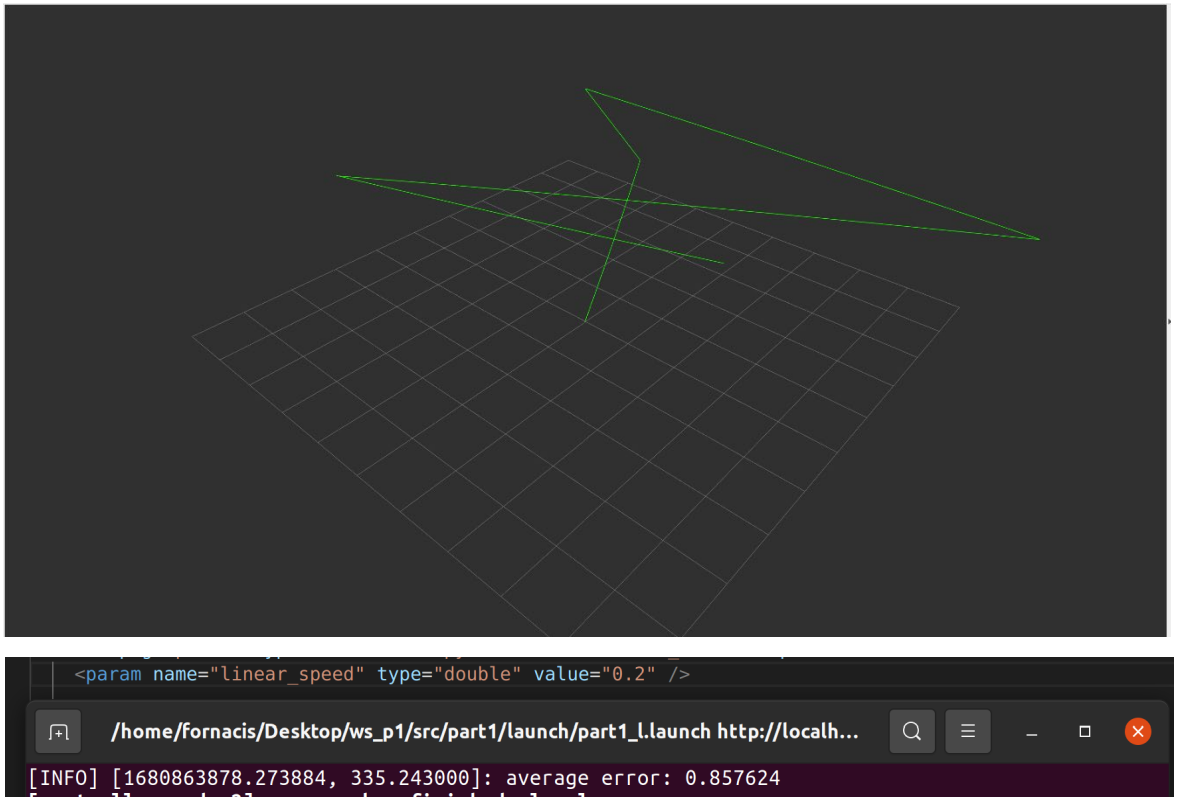
```python
#!/usr/bin/python3

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.

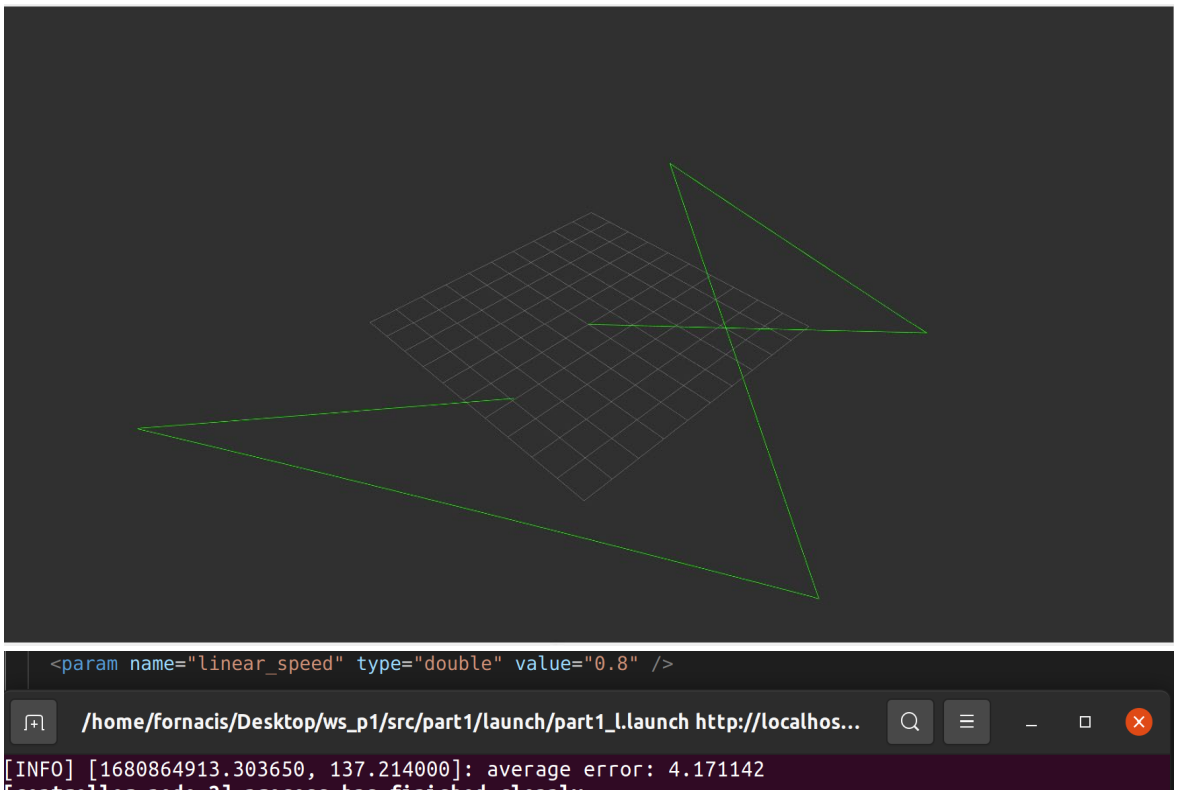
نتایج اجرا

حرکت ربات، هنگام شروع به حرکت و هنگام ایستادن با خطای زیادی همراه است. همچنین به علت تغییر سرعت ربات به مقادیر ۰.۲، ۰.۴، و ۰.۸ مدت زمان ویدیو های ضبط شده از اجرای گام اول برای ۵ بار حرکت به مقصد بسیار بیشتر از یک دقیقه شد.

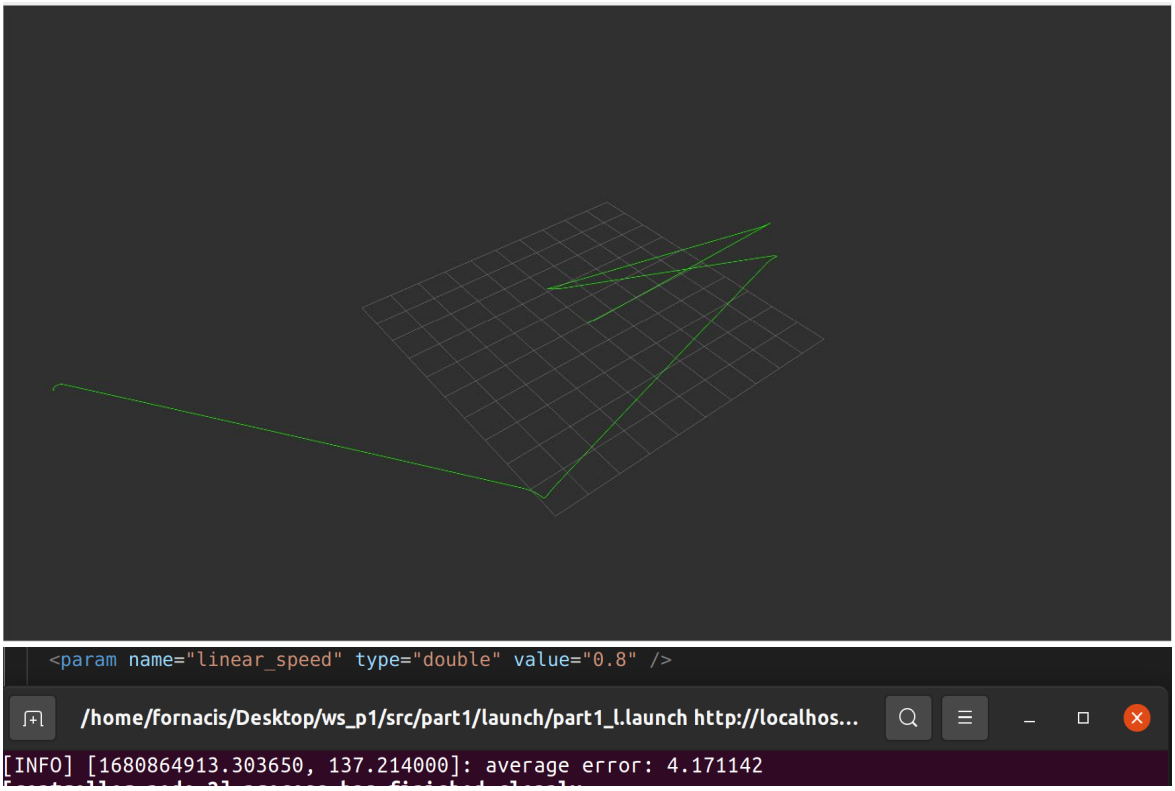
نتایج اجرای شبیه سازی به ازای سرعت های ۰.۲، ۰.۴، و ۰.۸ به شکل زیر است:



مسیر و میانگین خطای ربات به ازای سرعت ۰.۲



مسیر و خطای ربات به ازای سرعت ۰.۴



مسیر و خطای ربات به ازای سرعت ۰.۴

گام دوم

در این گام به طور عادی به سمت جلو حرکت می کنیم و با استفاده از سنسور LiDAR، فاصله از اجسام را اندازه گرفته و به محض رسیدن به مانعی با فاصله کمتر از ۲ متر، به آن پشت کرده و به حرکت ادامه می دهیم.

برای این کار از دو نود Sensor و Controller و پیام ClosestObstacle استفاده می کنیم.

پیام ClosestObstacle

src > part2 > msg > ClosestObstacle.msg

```
1   float64 distance
2   float64 direction
```

این پیام که توسط نود Sensor پابلیش می شود، حاوی دو متغیر برای مقدار کمترین فاصله و جهتی که مانع با کمترین فاصله در آن قرار دارد است.

نود Sensor

```python
class Sensor:

    def __init__(self):
        rospy.init_node("sensor_node", anonymous=True)
        self.pub = rospy.Publisher("/sensor",ClosestObstacle,queue_size=10)
        self.sub = rospy.Subscriber("/scan" , LaserScan , callback=self.laser_callback)

    def laser_callback(self, msg:LaserScan):
        min_distance = 999.0
        min_degree = 0
        for i in range(360):
            if msg.ranges[i] < min_distance:
                min_degree = i
                min_distance = msg.ranges[i]

        r = radians(min_degree)
        if r > pi:
            r = r - 2*pi

        sensor_msg = ClosestObstacle()
        sensor_msg.distance = min_distance
        sensor_msg.direction = r
        self.pub.publish(sensor_msg)



    def run(self):
        rospy.spin()

if __name__ =="__main__":
    sensor = Sensor()
    sensor.run()
```

این نود تاپیک scan را که مربوط به داده های سنسور LiDAR است subscribe کرده و به تاپیک sensor که حاوی پیام ClosestObstacle است پابلیش می کند. در هر بار پابلیش شدن تاپیک مربوط به LiDAR، یک تابع اجرا می شود تا مانع با کمترین فاصله را پیدا کند؛ سپس زاویه مربوط به این مانع را، به زاویه اویلری تبدیل کرده و آن را از pi- تا pi+ قرار داده و همراه با کمترین فاصله یافت شده، به تاپیک sensor پابلیش می کند.

نود Conroller

این قسمت اشتراک زیادی با نود Controller مربوط به گام اول دارد لذا فقط تفاوت های این دو در این بخش توضیح داده می شود.

```python
class Controller:

    def __init__(self) -> None:

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

        self.odom_subscriber = rospy.Subscriber("/odom" , Odometry , callback=self.odom_callback)
        self.sensir_subsciber = rospy.Subscriber("/sensor" , ClosestObstacle , callback=self.sensor_callback)
        self.cmd_publisher = rospy.Publisher('/cmd_vel' , Twist , queue_size=10)

        # getting specified parameters
        self.linear_speed = rospy.get_param("linear_speed") # m/s

        self.angular_speed = 0.4
        self.epsilon_linear = 0.1
        self.epsilon_angular = 0.02
        self.odom_yaw = 0
        self.odom_x=0
        self.odom_y=0
        self.min_distance = 10.0
        self.min_direction  = 1.0


        # defining the states of our robot
        self.GO, self.ROTATE = 0, 1
        self.state = self.GO
```

```python
    def sensor_callback(self, msg: ClosestObstacle):
        self.min_distance = msg.distance
        self.min_direction = msg.direction
```

کلاس controller دو فیلد min_distance و min_direction دارد که با هر پابلیش شدن تاپیک sensor به روزرسانی می شود. همچنین این نود تاپیک sensor را subscribe می کند. همچنین وضعیت ابتدایی ربات GO است.

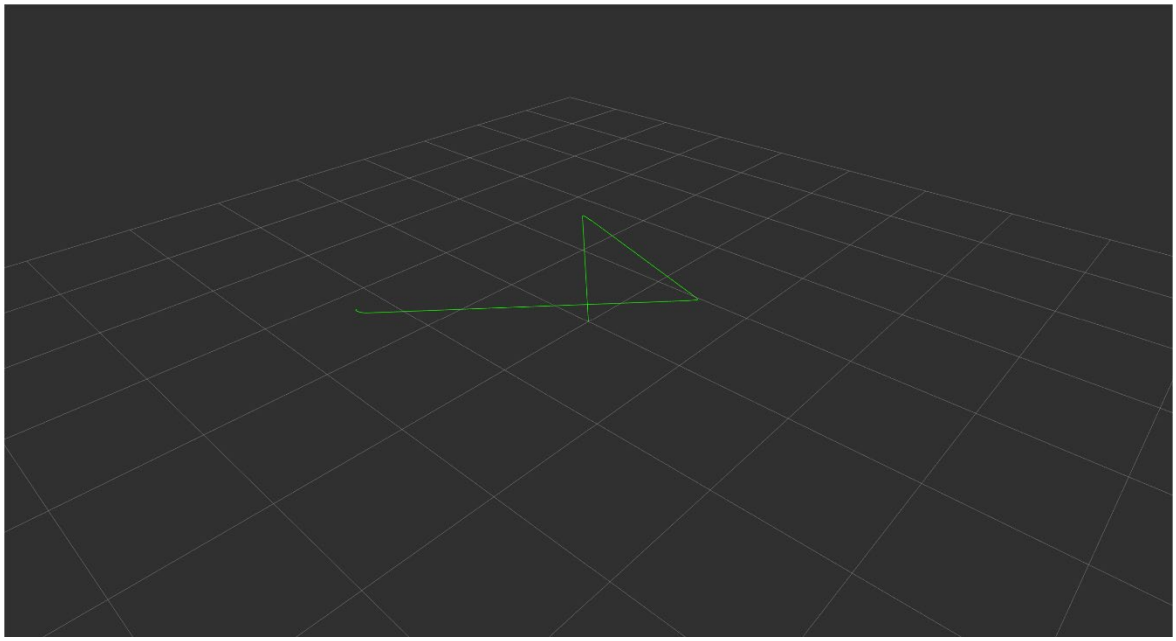
```python
def calculate_goal_angle(self):
    goal_angle = (self.odom_yaw + self.min_direction + math.pi) % (2* math.pi)
    if goal_angle > math.pi:
        goal_angle -= 2* math.pi

    return goal_angle
```

این تابع با استفاده از زاویه کنونی ربات و زاویه نزدیکترین مانع، زاویه هدف ربات به منظور قرارگیری مانع در پشت ربات را محاسبه می کند.

```python
    def run(self):
        rospy.loginfo("run started")

        while not rospy.is_shutdown():

            # check whether state is changed or not
            if self.state == self.GO:
                twist = Twist()
                twist.linear.x = self.linear_speed
                while self.min_distance > 2.0 or (self.min_direction > (math.pi*0.8) or self.min_direction < (math.pi* -0.8)):
                    rospy.loginfo("distance %f  ",self.min_distance)
                    self.cmd_publisher.publish(twist)

                self.cmd_publisher.publish(Twist())
                rospy.sleep(3)
                self.state=self.ROTATE

            if self.state == self.ROTATE:
                goal_yaw = self.calculate_goal_angle()
                if (goal_yaw - self.odom_yaw <= math.pi and goal_yaw - self.odom_yaw >0) or goal_yaw - self.odom_yaw <= -math.pi >0:
                    twist = Twist()
                    twist.angular.z = self.angular_speed
                    rospy.loginfo(twist.angular.z)

                elif goal_yaw - self.odom_yaw == 0:
                    twist = Twist()
                else :
                    twist = Twist()
                    twist.angular.z = -self.angular_speed
                    rospy.loginfo(twist.angular.z)

                while(abs(self.odom_yaw-goal_yaw) > self.epsilon_angular):
                    self.cmd_publisher.publish(twist)

                self.cmd_publisher.publish(Twist())
                rospy.sleep(3)
                self.state=self.GO
```

در تابع run، ربات تا زمانی که فاصله نزدیکترین مانع به آن کمتر از ۲ شود به حرکت ادامه می دهد (در صورتی که مانع در محدوده پشت ربات قرار داشته باشد، ربات همچنان به حرکت خود ادامه می دهد در غیر اینصورت، ربات بعد از هر بار مشاهده مانع با فاصله کمتر از ۲ متر و دوران برای قرارگیری مانع در پشت خود، باز هم حرکت نمی کرد چون مانع با فاصله کمتر از ۲ متر در پشت سر آن قرار گرفته بود). بعد از این کار وضعیت را به ROTATE تغییر داده و تا رسیدن به زاویه هدف دروان می کند. بعد نیز دوباره وضعیت به GO تغییر پیدا می کند.

نتایج اجرا



نمای مسیر ربات

گام سوم

در این گام با visualize کردن داده های مربوط به LiDAR آشنا می شویم

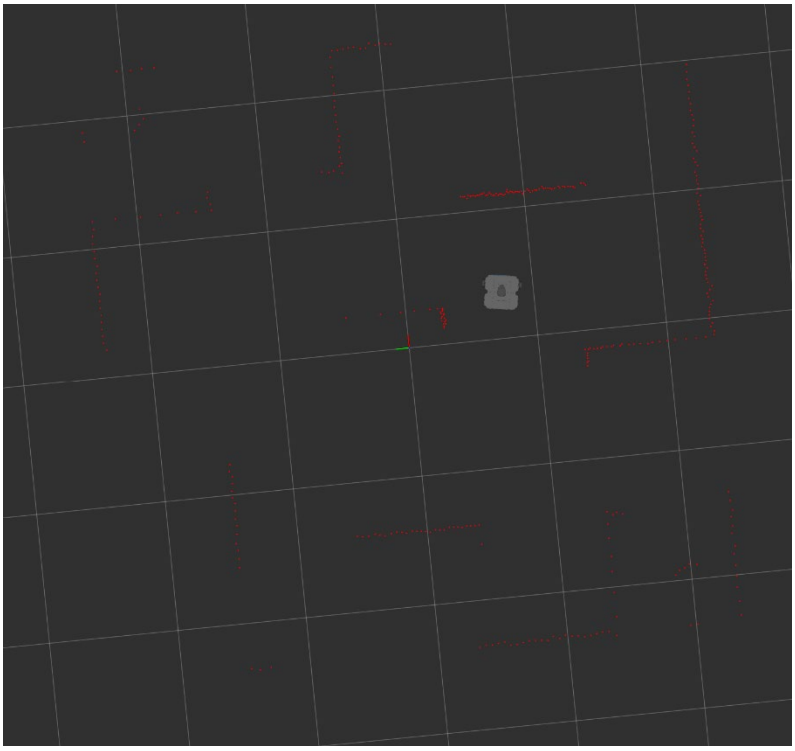
بخش اول

مطابق با توضیحات دستورکار، فایل launch زیر را ساخته و آن را اجرا می کنیم.

src > pcd_view > launch > part3.launch

```
<launch>


  <include file="$(find turtlebot3_gazebo)/launch/turtlebot3_stage_4.launch"/>
  <include file="$(find turtlebot3_gazebo)/launch/turtlebot3_gazebo_rviz.launch"/>


</launch>
```

سپس teleop را نیز اجرا کرده و در rviz، نمای مربوطه را مشاهده می کنیم.



علت لرزش نقاط خطای سنسور LiDAR است. این سنسور در هر بار از بازتاب امواجی که ارسال کرده استفاده می کند تا فاصله را در هر جهت تشخیص دهد. منتها به علت خطای سنسور، حتی در صورتی که ربات ثابت باشد در هر بار پردازش امواج بازتاب شده، فاصله را دقیقا مطابق دفعه قبل تشخیص نمی دهد و این مقدار اندکی تفاوت دارد. این جا به جایی اندک نقاط، لرزش به نظر می رسد.

بخش دوم

در این بخش داده های LiDAR در تمامی زمان ها را کنار هم قرار می دهیم تا تصویری از محیط درست کنیم.

src > pcd_view > src > laser2pc.py > ...

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

import rospy
from laser_assembler.srv import AssembleScans2
from sensor_msgs.msg import PointCloud2

rospy.init_node("assemble_scans_to_cloud")
rospy.wait_for_service("assemble_scans2")
assemble_scans = rospy.ServiceProxy("assemble_scans2", AssembleScans2)
pub = rospy.Publisher ("/laser_pointcloud", PointCloud2, queue_size=1)

r = rospy.Rate (1)


while (True):
    rospy.loginfo("in while")
    try:
        resp = assemble_scans(rospy.Time(0,0), rospy.get_rostime())
        rospy.loginfo(rospy.Time(0,0))
        rospy.loginfo(rospy.get_rostime())
        rospy.loginfo("meeeeeeeee Got cloud with %u points" % len(resp.cloud.data))
        pub.publish (resp.cloud)

    except rospy.ServiceException as e:
        rospy.loginfo("meeeeeeeeee Service call failed: %s"%e)

    r.sleep()

```

برای این کار داده های laser جمع آوری شده را از سرویس assemble_scans2 گرفته و در تاپیک laser_poincloud پابلیش می کنیم تا در rviz بتوانیم آن را مشاهده کنیم.

همچنین فایل launch زیر را مطابق توضیحات دستور کار می سازیم:

src > pcd_view > launch > part3.launch

```xml
<launch>


  <include file="$(find turtlebot3_gazebo)/launch/turtlebot3_stage_4.launch"/>
  <include file="$(find turtlebot3_gazebo)/launch/turtlebot3_gazebo_rviz.launch"/>

  <node type="laser_scan_assembler" pkg="laser_assembler"
    name="my_assembler">

  <param name="max_scans" type="int" value="400" />
  <param name="fixed_frame" type="string" value="odom" />
  </node>
  <node type ="laser2pc.py" pkg="pcd_view" name="laser2pc" output="screen"/>



</launch>

```

شکل نهایی بدست آمده از نقشه: