



Department of  
Computer Engineering

به نام خدا



Amirkabir University of Technology  
(Tehran Polytechnic)

دانشگاه صنعتی امیرکبیر  
دانشکده مهندسی کامپیوتر  
اصول علم ربات

تمرین سری سوم

نام و نام خانوادگی	حمیدرضا همتی
شماره دانشجویی	۹۶۳۱۰۷۹
تاریخ ارسال گزارش	

## فهرست گزارش سوالات

- سوال ۱ - سوال تئوری اول ..... ۳
- سوال ۲ - سوال تئوری دوم ..... ۴
- سوال ۳ - سوال تئوری سوم ..... ۵
- سوال ۴ - سوال تئوری چهارم ..... ۵
- سناریو اول - نزدیک ترین مانع ..... ۶
- سناریو دوم - دنبال کردن دیوار ..... ۱۵

## سوال ۱ – سوال تئوری اول

سنسورهای compass، gyroscope سنسورهای passive هستند.

سنسورهای ultrasonic، Doppler radar سنسورهای active هستند.

GPS یک نوع سنسور active هست. نحوه کار سنسورهای active به این صورت است که از بازتاب یا دریافتی که از محیط پس از ساطع کردن انرژی یا سیگنال دارند sensing انجام میدهند. نحوه کار gps و تعیین موقعیت مکانی به همین صورت است پس یک نوع سنسور active هست.

## سوال ۲ – سوال تئوری دوم

نمیتوان از GPS استفاده کرد زیرا که سیگنال هایی که GPS با آن کار میکند، از ماهواره ها گرفته میشوند پس از محیط خارج از فضای ساختمانی ما ساطع شده اند ولی مشکل این است که این سیگنال ها به خاطر مشخصات طول موج و فرکانسی که دارند نفوذ کمی دارند و نمیتوانند از دیوارهای ساختمان به درستی گذر کنند به همین دلیل کارکرد GPS در فضاهای ساختمانی مختل میشوند.

### سوال ۳ – سوال تئوری سوم

از سنسور gyroscope استفاده میشود. به وسیله این سنسور میتوانیم جهت حرکت و heading خود را تشخیص دهیم.

### سوال ۴ – سوال تئوری چهارم

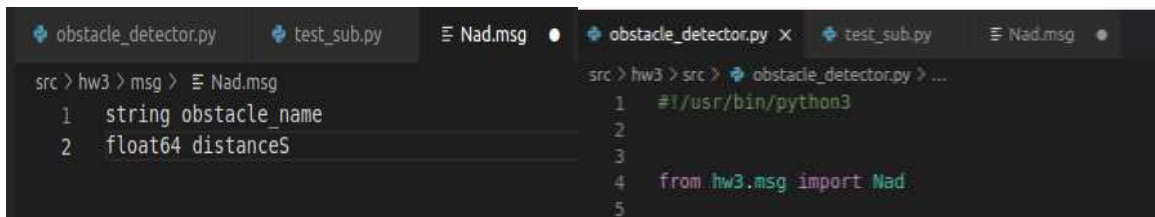
با داشتن قطب نما میتوانیم heading خود را تشخیص دهیم. روش آن به این صورت است که ب compass را به یک سمت set میکنیم. سپس خروجی قطب نما یک عدد است که با آن میتوانیم انحراف خود را از جهت set شده تشخیص دهیم. اما مشکلی که وجود دارد این است که قطب نما در اثر میدانهای مغناطیسی دچار ایراد میشود و اعداد اشتباهی را نشان میدهد. پس در مکان هایی که سیمهای فشارقوی برق یا آهن وجود دارد نمیتوان از آن به درستی استفاده کرد.

## سناریو اول – نزدیک ترین مانع

( الف )

در این بخش ابتدا یک فایل پایتون به نام `obstacle_detector` میسازیم. در این فایل قرار هست که موقعیت هر لحظه ربات به طور `real time` گرفته شود و فاصله آن با همه موانع محاسبه شود. در نهایت باید مانعی که نزدیک تر هست انتخاب شده و اسم و مقدار فاصله اش در تاپیک `closest_obstacle` قرار داده شود.

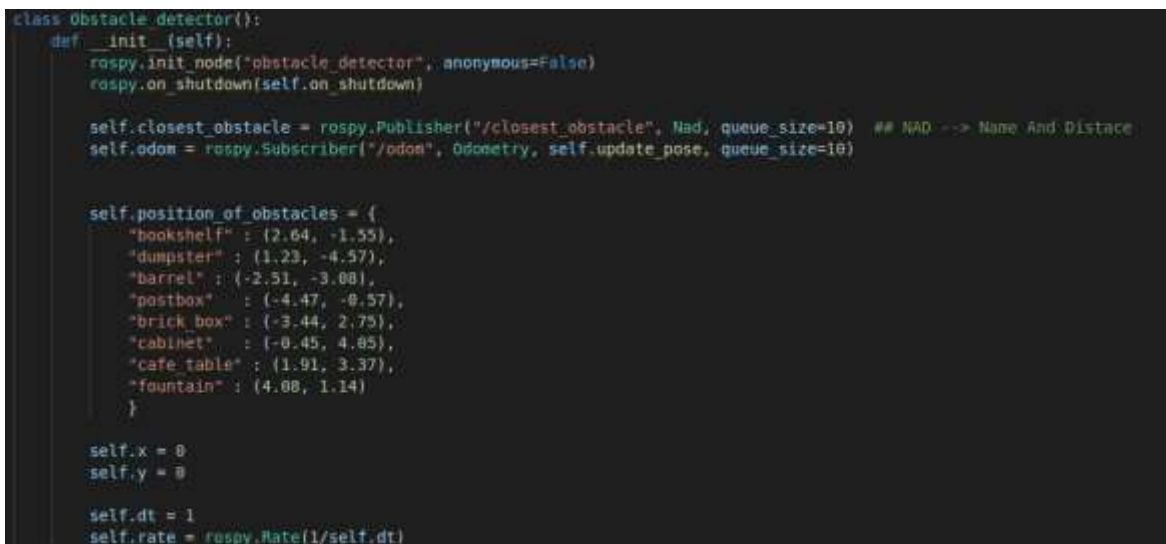
به دلیل اینکه متغیر استاندارد یک مرکب از یک `string` و یک `float` وجود ندارد ابتدا به صورت زیر یک `message` کاستوم ساخته و آنرا در فایل کد `import` میکنیم. از این مسیج برای ذخیره سازی نام و فاصله نزدیک ترین مانع به ربات و در نهایت `publish` کردن آن از طریق تاپیک `closest_obstacle` استفاده میکنیم.



```
obstacle_detector.py test_sub.py Nad.msg
src > hw3 > msg > Nad.msg
1 string obstacle_name
2 float64 distanceS

obstacle_detector.py x test_sub.py Nad.msg
src > hw3 > src > obstacle_detector.py > ...
1 #!/usr/bin/python3
2
3
4 from hw3.msg import Nad
5
```

در شکل زیر تابع `init` کلاس `obstacle_detector` آمده است. همانطور که قابل مشاهده هست در تابع `init` خود `node` و همچنین `publisher` و `subscriber` تعریف شده اند. همچنین یک دیکشنری متشکل از اسم و موقعیت مراکز موانع تعریف شده است که بعدا برای محاسبه فاصله ربات از موانع از آن استفاده میکنیم.



```
class ObstacleDetector():
    def __init__(self):
        rospy.init_node('obstacle_detector', anonymous=False)
        rospy.on_shutdown(self.on_shutdown)

        self.closest_obstacle = rospy.Publisher("/closest_obstacle", Nad, queue_size=10) ## NAD --> Name And Distance
        self.odom = rospy.Subscriber("/odom", Odometry, self.update_pose, queue_size=10)

        self.position_of_obstacles = {
            "bookshelf" : (2.64, +1.55),
            "dumpster" : (1.23, -4.57),
            "barrel" : (-2.51, -3.08),
            "postbox" : (-4.47, -0.57),
            "brick box" : (-3.44, 2.75),
            "cabinet" : (-0.45, 4.85),
            "cafe_table" : (1.91, 3.37),
            "fountain" : (4.08, 1.14)
        }

        self.x = 0
        self.y = 0

        self.dt = 1
        self.rate = rospy.Rate(1/self.dt)
```

تابع زیر که در subscriber شکل بالا صدا زده شده بود وظیفه update کردن موقعیت فعلی ربات را دارد. msg ایی که در ورودی این تابع قرار دارد مقدار جدیدی است که هر بار از تاپیک odom خوانده میشود.

```
def update_pose(self, msg):
    self.x = msg.pose.pose.position.x
    self.y = msg.pose.pose.position.y
```

تابع زیر لوپ اصلی کد هست که در آن فاصله ربات از موانع محاسبه شده و مشخصات نزدیک ترین مانع به طور مداوم بر روی تاپیک closest\_obstacle قرار داده میشود.

نحوه کار کلی این بخش به این صورت است که هر بار بر روی دیکشنری که در تابع init توضیح داده شد چرخیده و فاصله تمام موانع را از موقعیت فعلی ربات (که به وسیله تابع update\_pose به روز رسانی شده است) محاسبه میکنیم. در اینجا یک دیکشنری local جدید ساخته و اسم موانع را به همراه فاصله آنها در آن ذخیره میکنیم (خطهای ۵۲ و ۵۳).

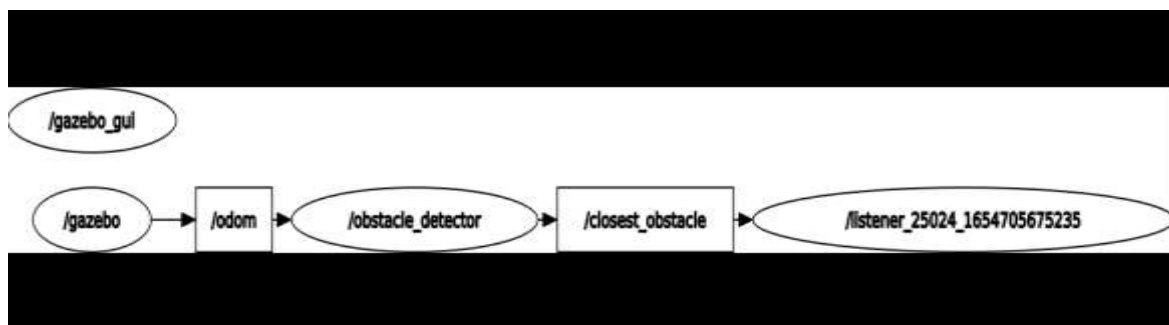
سپس یک متغیر از جنس NAD ساخته (متغیر custom که ایجاد کرده بودیم) و پس از پیدا کردن نزدیک ترین مانع در خط های ۵۷ و ۵۸ این متغیر را مقدرا دهی میکنیم. در نهایت متغیر ساخته شده را در خط ۶۱ publish میکنیم.

```
48 def detect_closest_obstacle(self):
49     rospy.sleep(2)
50     temp = {}
51     while not rospy.is_shutdown():
52         for key, value in self.position_of_obstacles.items():
53             temp[key] = self.distance_from_obstacle(value[0], value[1])
54             # print(key, " dist: ", self.distance_from_obstacle(value[0], value[1]))
55
56         nad = Nad()
57         nad.obstacle_name = min(temp, key=temp.get)
58         nad.distance = min(temp.values())
59         print(type(nad))
60         print(nad)
61         self.closest_obstacle.publish(nad)
62
63         print(" ----- ")
64
65         self.rate.sleep()
```

برای تست کردن این که آیا پیام ها به درستی در topic مورد نظر یعنی closest\_obstacle قرار میگیرند یک node ساده صرفا برای تست کردن به صورت زیر ایجاد کردم.

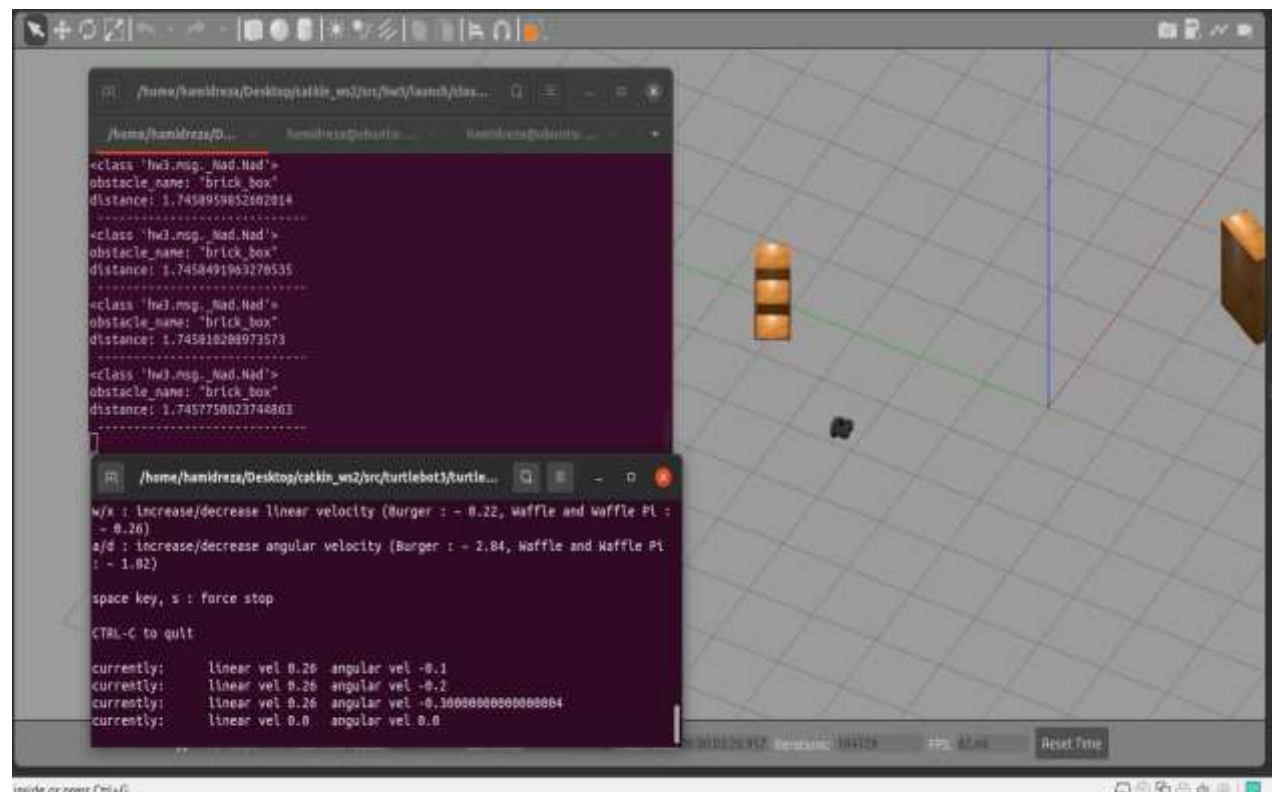
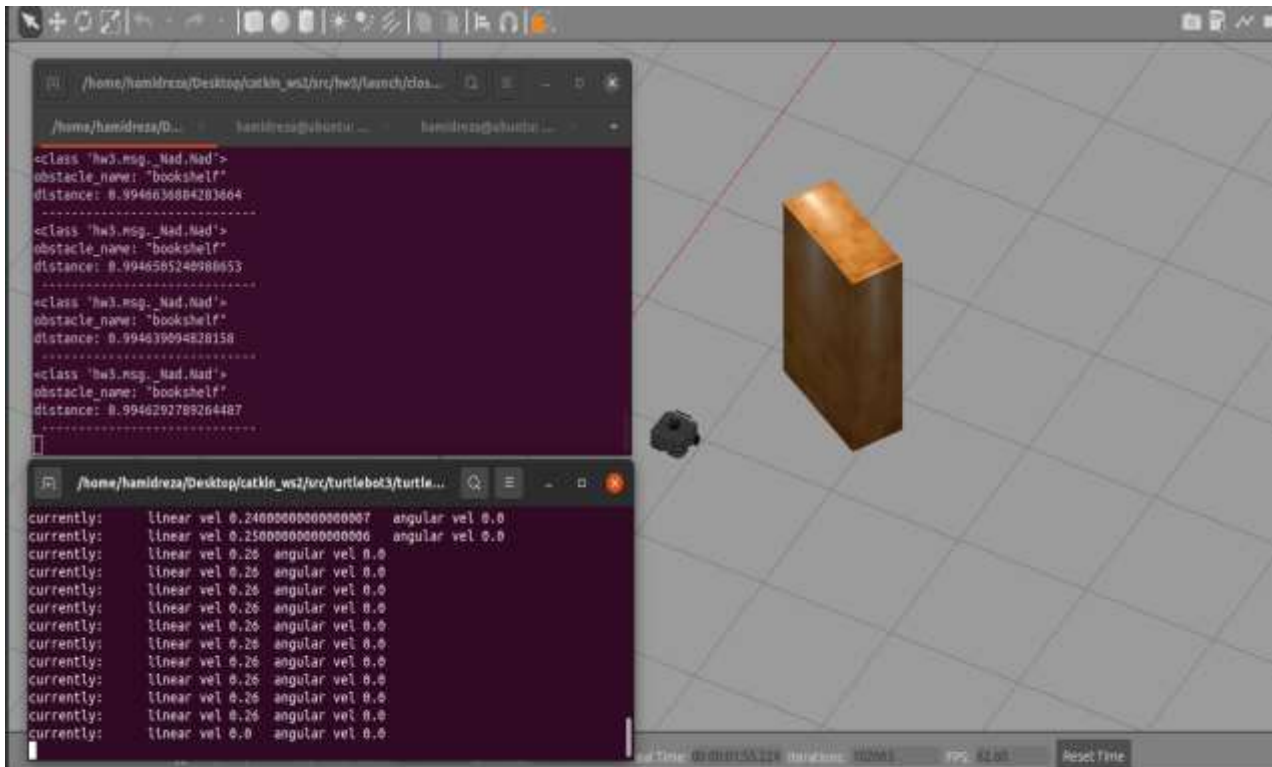
```
obstacle_detector.py  test_sub.py X  Nad.msg
src > hw3 > src > test_sub.py > ...
1  #!/usr/bin/python3
2
3  from hw3.msg import Nad
4  import rospy
5
6
7  def display(data):
8      rospy.loginfo('{} {}'.format(data.obstacle_name, data.distance))
9
10
11  def hardware_departement():
12      rospy.init_node('listener', anonymous=True)
13      rospy.Subscriber('closest_obstacle', Nad, callback=display)
14      rospy.spin()
15
16  if __name__ == '__main__':
17      hardware_departement()
```

پس از run کردن هر دو نود گراف زیر رسم شد که نشان میدهد مسیج ها درست ساخته و publish میشوند.

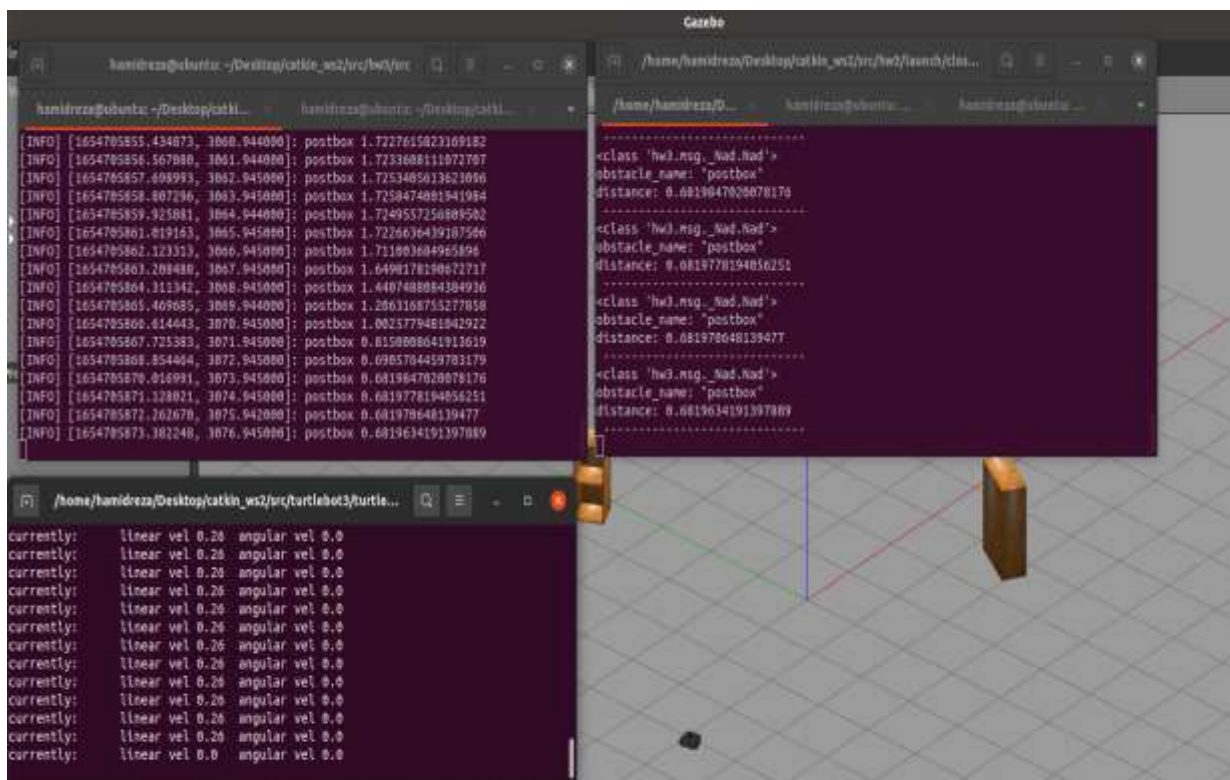


در شکل های زیر تصاویری از تست های انجام شده را میتوانید مشاهده نمایید.  
توجه: فایل word ایی که من داشتم فقط دوتا bookshelf رو generate میکرد و بقیه موانع رو نمایش نمیدهد. ولی چون که موقعیت های موانع hardcoded شده هستن مشکلی برای بخش الف به وجود نمیآورد.



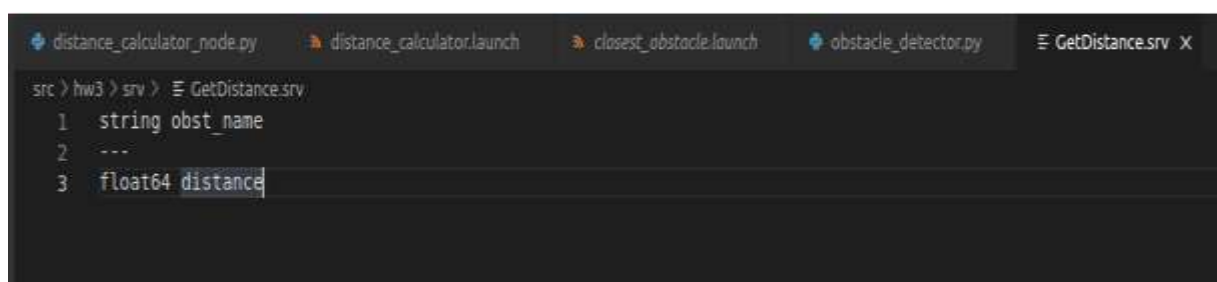






(ب)

در این بخش باید یک سرویس ایجاد شود که نام مانع را دریافت کرده و فاصله از آن مانع را به عنوان خروجی بازگرداند. در شکل زیر فایل srv تولید شده را میتوانید مشاهده کنید.



ورودی این سرویس یک string و خروجی آن یک عدد float است.

سپس باید کد این سرویس را بنویسیم. در شکل زیر تابع listener مشخص شده است، نحوه کار این تابع به این صورت است که کلاس مورد نظر را ایجاد میکند و سپس برای update کردن موقعیت ربات و همچنین فاصله موانع از ربات تابع update\_distances در کالک subscriber به تاپیک odom صدا میشود. سپس تابع get\_distance هم به عنوان کالک service استفاده میشود.

```

72
73 def listener():
74     rospy.init_node('distance_calculator_node', anonymous=True)
75     dc = Distance_calculator()
76     rospy.Subscriber("/odom", Odometry, dc.update_distances)
77     s = rospy.Service('/get_distance', GetDistance, dc.get_distance)
78     rospy.spin()
79
80
81 if __name__ == '__main__':
82     listener()

```

در شکل های زیر بخش init کلاس که دیکشنری های مهم distances که برای نگهداری فواصل به روز رسانی شده ربات از موانع هست و همچنین دیکشنری position\_of\_obstacles که position موانع را نگه میدارد مشاهده میکنید. همچنین توابع update\_distance و update\_pose که موقعیت ربات و فاصله ربات تا موانع را به روز رسانی میکند هم مشخص شده است.

```

8
9 class Distance_calculator():
10     def __init__(self) -> None:
11
12         self.position_of_obstacles = {
13             "bookshelf" : (2.64, -1.55),
14             "dumpster" : (1.23, -4.57),
15             "barrel" : (-2.51, -3.08),
16             "postbox" : (-4.47, -0.57),
17             "brick_box" : (-3.44, 2.75),
18             "cabinet" : (-0.45, 4.05),
19             "cafe_table" : (1.91, 3.37),
20             "fountain" : (4.08, 1.14)
21         }
22
23         self.distances = {
24             "bookshelf" : -1.0,
25             "dumpster" : -1.0,
26             "barrel" : -1.0,
27             "postbox" : -1.0,
28             "brick_box" : -1.0,
29             "cabinet" : -1.0,
30             "cafe_table" : -1.0,
31             "fountain" : -1.0
32         }
33
34         self.x = 0
35         self.y = 0
36
37         self.dt = 1
38         self.rate = rospy.Rate(1/self.dt)
39
40     def update_pose(self, msg):
41         self.x = msg.pose.pose.position.x
42         self.y = msg.pose.pose.position.y
43

```



```

43
44
45     def distance_from_obstacle(self, obstacle_x, obstacle_y):
46         dif_x = (self.x - obstacle_x) ** 2
47         dif_y = (self.y - obstacle_y) ** 2
48         return (dif_x + dif_y) ** 0.5
49
50
51     def update_distances(self):
52         self.update_pose()
53         for key, value in self.position_of_obstacles.items():
54             self.distances[key] = self.distance_from_obstacle(value[0], value[1])
55

```

توابعی که در بالا آورده شده اند هربار پس از آمدن پیام جدیدی در تایپیک odom به صورت سلسله مراتبی صدا میشوند. ابتدا تابع update\_distance صدا میشود و این تابع update\_pose را فراخوانی میکند که این تابع وظیفه به روز رسانی موقعیت ربات را بر عهده دارد. سپس با استفاده از دیکشنری position\_of\_obstacles که موقعیت موانع را نگه داری میکند فاصله ربات با تمام موانع را محاسبه میکنیم و در دیکشنری distances ذخیره میکنیم.

حال هروقت سرویس صدا میشود تابع تابع get\_distance که در شکل زیر آمده است صدا میشود. این تابع خودش هیچ پردازش خاصی ندارد و همه پردازش ها در توابع دیگر انجام شده اند. تنها کاری که این تابع میکند اسم مانع که در ورودی سرویس آمده است را برمیدارد و از دیکشنری distances که قبلا توضیح داده شد مقدار value مربوط به آن را بازیابی میکند.

```

56
57     def get_distance(self, req):
58         distance = -1
59         obst_name = req.obst_name
60         print("obstacle name: ", obst_name)
61         for key, value in self.distances.items():
62             if key == obst_name:
63                 distance = value
64                 res = GetDistanceResponse()
65                 res.distance = distance
66                 print("distance: ", distance)
67                 return res
68         rospy.logerr(f'direction_name is not valid: {obst_name}')
69         return None
70
71

```

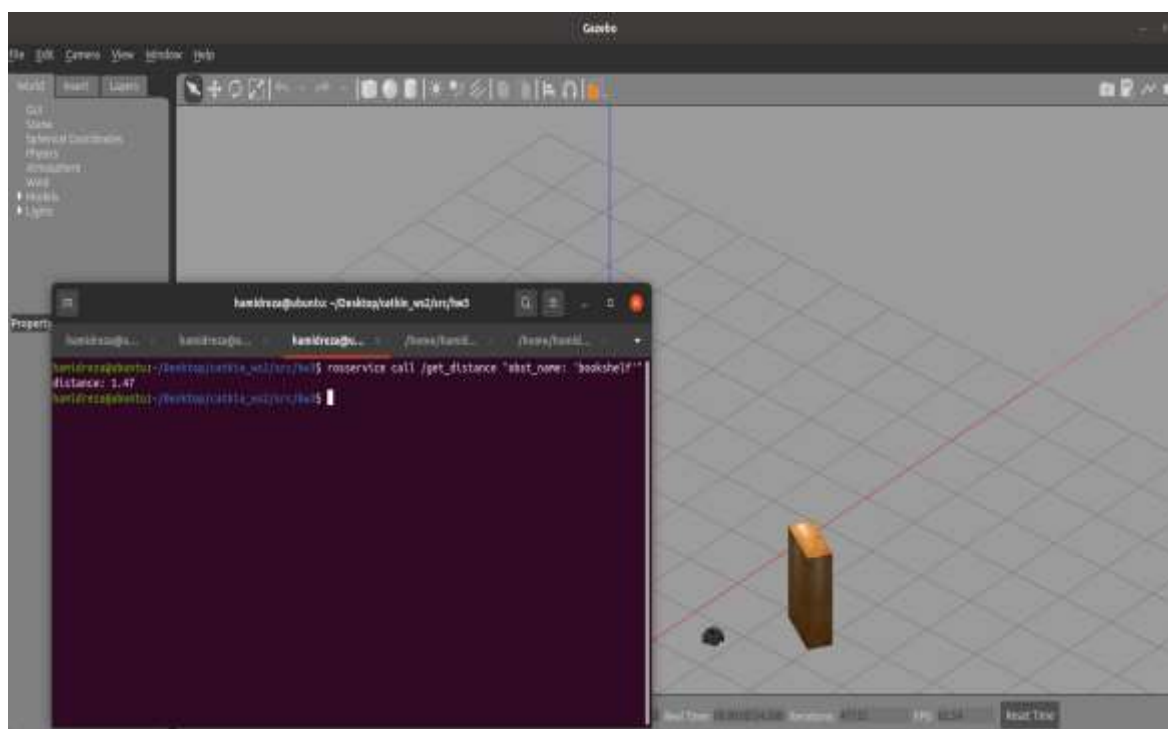
شکل زیر کامند مربوط به نمایش لیست service ها را نشان میدهد که مشخص است service مورد نظر درست ایجاد شده و همچنین وقتی که سرویس صدا میشود بدون آن که گزبو بالا باشد مقدار دیفالت یا منفیک را برمیگرداند.

```

hamidreza@ubuntu: ~/Desktop/catkin_ws2/src/hw3
hamidreza@ubuntu:~/Desktop/catkin_ws2/src/hw3$ cd ..
hamidreza@ubuntu:~/Desktop/catkin_ws2/src/hw3$ rosservice list
/distance_calculator_node_7273_1655222648378/get_loggers
/distance_calculator_node_7273_1655222648378/set_logger_level
/get_distance
/rosout/get_loggers
hamidreza@ubuntu:~/Desktop/catkin_ws2/src/hw3$ rosservice call /get_distance "obst_name: 'bookshelf'"
ERROR: Unable to load type [hw3/GetDistance].
Have you typed 'make' in [hw3]?
hamidreza@ubuntu:~/Desktop/catkin_ws2/src/hw3$ cd ..
hamidreza@ubuntu:~/Desktop/catkin_ws2/src$ cd ..
hamidreza@ubuntu:~/Desktop/catkin_ws2$ . devel/setup.bash
hamidreza@ubuntu:~/Desktop/catkin_ws2$ roscd hw3
hamidreza@ubuntu:~/Desktop/catkin_ws2/src/hw3$ rosservice call /get_distance "obst_name: 'bookshelf'"
distance: -1.0
hamidreza@ubuntu:~/Desktop/catkin_ws2/src/hw3$ rosservice call /get_distance "obst_name: 'bookshelfff'"
ERROR: service [/get_distance] responded with an error: b'service cannot process request: service handler returned N
one'
hamidreza@ubuntu:~/Desktop/catkin_ws2/src/hw3$

```

شکل زیر هم خروجی سرویس بعد از اجرای فایل launch و حرکت دادن ربات درون محیط gazebo به وسیله کیبورد است.



### الف)

در بخش کد این سوال از سه تابع زیر برای update کردن موقعیت فعلی ربات و همچنین تشخیص فاصله از دیوار سمت چپ استفاده میکنیم. تابع distance\_from\_left\_wall از تاپیکی که مقدار های laserScan در آن ریخته میشود استفاده میکند.

```
37 def update_pose(self, msg):
38     self.x = msg.pose.pose.position.x
39     self.y = msg.pose.pose.position.y
40     self.yaw = self.quaternion_to_euler(msg)
41
42
43 def quaternion_to_euler(self, msg):
44     orientation = msg.pose.pose.orientation
45     # convert quaternion to odom
46     roll, pitch, yaw = tf.transformations.euler_from_quaternion((
47         orientation.x ,orientation.y ,orientation.z ,orientation.w
48     ))
49     return yaw
50
51
52 def distance_from_left_wall(self):
53     laser_data = rospy.wait_for_message("/scan", LaserScan)
54     left_side = laser_data.ranges[10:180]
55     return min(left_side)
56
```

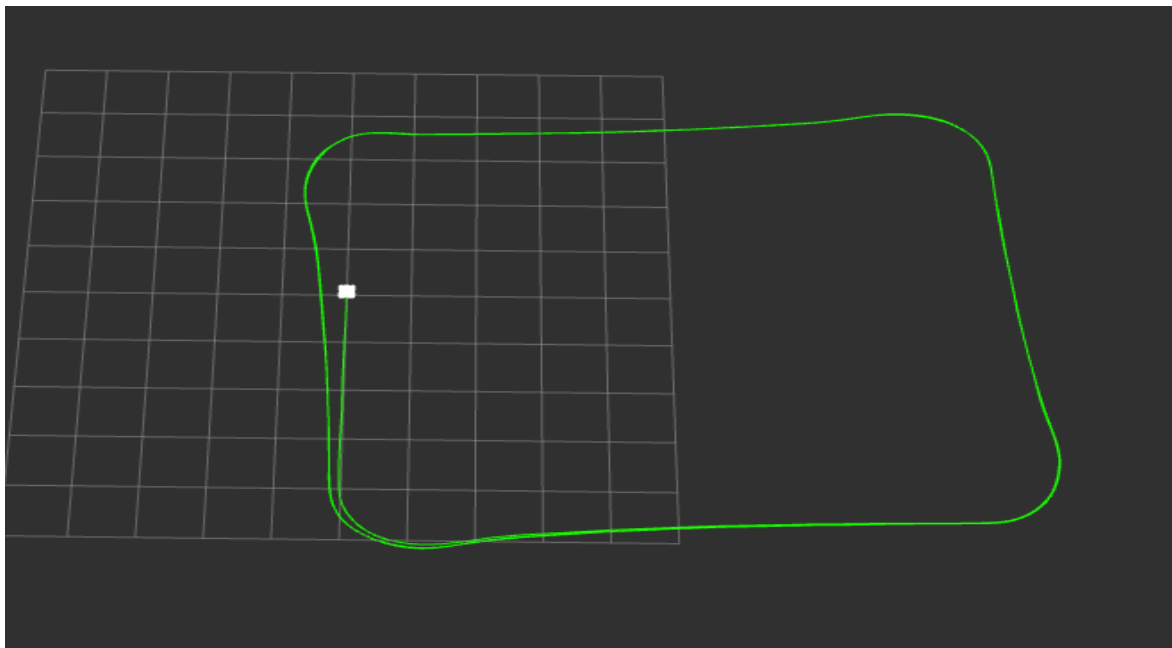
در کد زیر که لوپ اصلی کد است برای کنترل ربات به صورتی که دیوار سمت چپ را دنبال کند استفاده میکنیم. در این لوپ ابتدا مقدار error را محاسبه میکنیم و سپس مقادیر PID را با توجه به error به روزرسانی کرده و مقدار حاصل را که سرعت زاویه ایی ربات حول محور z است در تاپیک cmd\_vel پابلیش میکنیم. نحوه کار این کنترلر به این صورت است که اگر ربات از یک حد مشخصی که با distance\_margin مشخص شده است مثلا ۱ متر، به دیوار نزدیک تر بود با توجه به فاصله آن PID یک مقدار منفی و اگر از آن حد مشخص دورتر بود با توجه به میزان فاصله یک عدد مثبت میگیرد. درنهایت ربات بر روی خط با فاصله دلخواه ما از دیوار حرکت خواهد کرد. تمام این مراحل در کد شکل زیر مشخص است.

```

70 while not rospy.is_shutdown():
71
72     error = left_dist - self.distance_margin
73
74     P = self.kp * error
75     I = self.ki * integral
76     D = self.kd * derivative
77
78     speed.angular.z = P+I+D
79     speed.linear.x = self.linear_Speed
80
81     self.cmd_publisher.publish(speed)
82
83
84     ## update controller
85     integral += error * self.dt
86     derivative = error - privous_error
87     privous_error = error
88
89     left_dist = self.distance_from_left_wall()
90
91     self.rate.sleep()

```

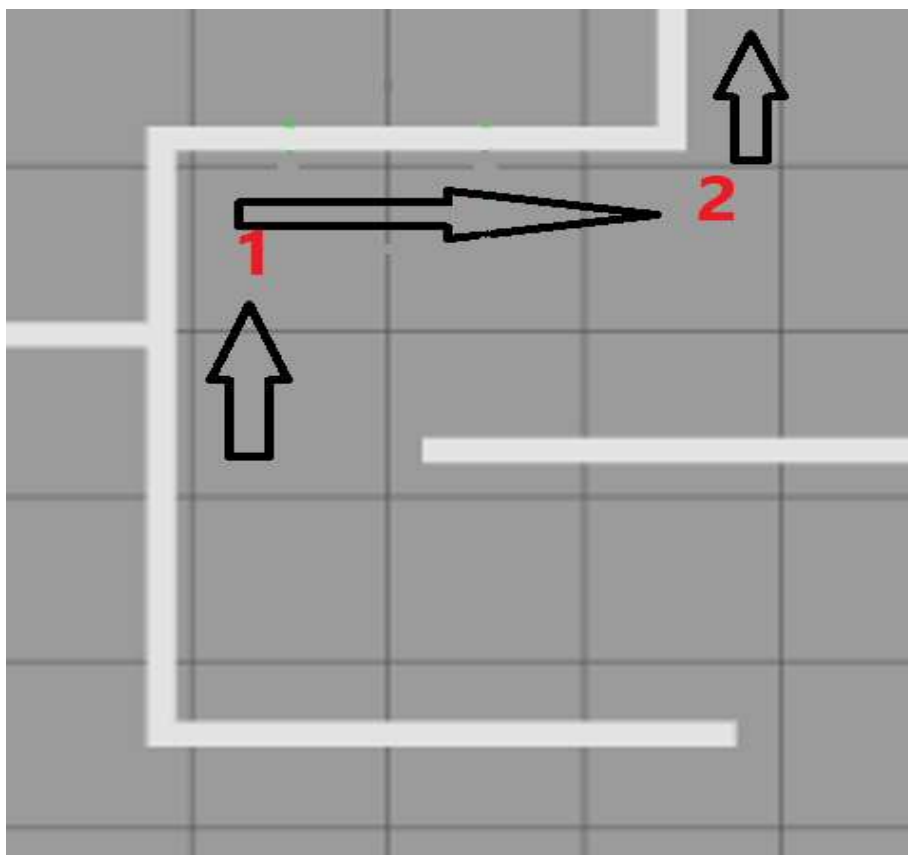
در شکل زیر مسیر ربات حول دیوار را میتوانید مشاهده نمایید.





## (ب)

اکثریت این بخش همانند بخش الف است زیرا که ربات قرار است دیوار سمت چپ را آنقدر دنبال کند تا از maze بیرون بیاید. پس کنترلی که برای دنبال کردن دیوار و رد کردن گوشه ها در سوال قبل درست کردیم برای این بخش هم به کار میرود. اما چون که در نقشه maze ما کنج هایی مانند شکل زیر داریم نمیتوان دقیقا کد بخش قبل را اجرا کرده و انتظارداشته باشیم که ربات بتواند دیوار را دنبال کند. برای توضیح بیشتر شکل زیر را در نظر بگیرید. اگر ربات دیوار سمت چپ را دنبال کند با توجه به کنترلر بخش الف نمیتواند کنج که با شماره ۱ مشخص شده است را رد کند در حالی که گوشه که با شماره ۲ مشخص شده است را میتواند رد کند و آن قدر بچرخد تا دیوار در سمت چپ آن قرار بگیرد. تفاوتی که بین ۱ و ۲ وجود دارد این است که در حالت ۲ که مانند دنبال کردن دیوار در بخش الف این سناریو است وقتی دیوار در سمت چپ نباشد مقدار error زیاد میشود و این باعث میشود که ربات به سمت چپ بچرخد. اما در حالت ۱ دیوار در سمت چپ وجود دارد و اگر با کنترلر بخش الف رباتی را در این مسیر قرار دهیم نمیتواند به سمت راست گردش کند و مستقیم به دیوار برخورد میکند.



برای حل مشکلی که در بالا توضیح داده شده است دوتغییر مهم در در کد بخش الف ایجاد کرده ام تا ربات بتواند هم از کنج های مسیر رد شود و هم از گوشه دیوار ها.

تغییر اول این است که تابعی به شکل زیر برای تشخیص فاصله از دیوار جلو ایجاد میکنیم.

```
56 def distance_from_left_wall(self):
57     laser_data = rospy.wait_for_message("/scan", LaserScan)
58     left_side = laser_data.ranges[20:135]
59     return min(left_side)
60
61
62 def distance_from_front_wall(self):
63     laser_data = rospy.wait_for_message("/scan", LaserScan)
64     l_front = laser_data.ranges[0]
65
66     # r_front = laser_data.ranges[355:360]
67     front = l_front
68     return front
69
```

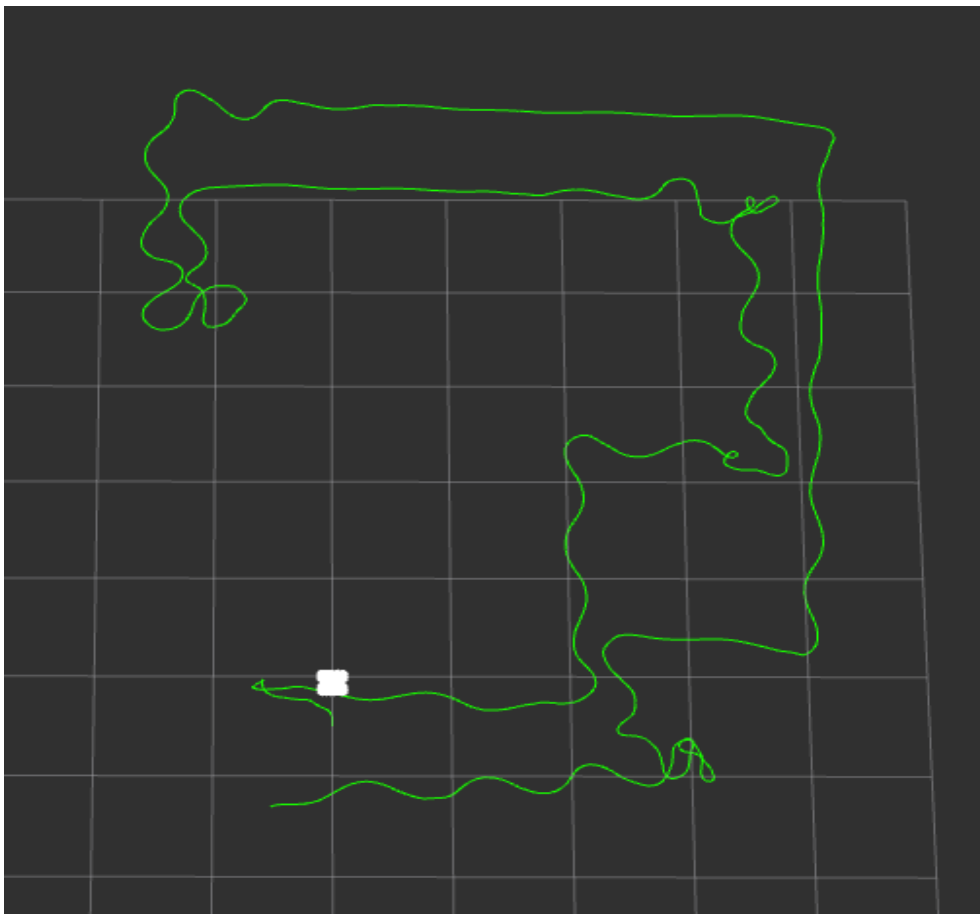
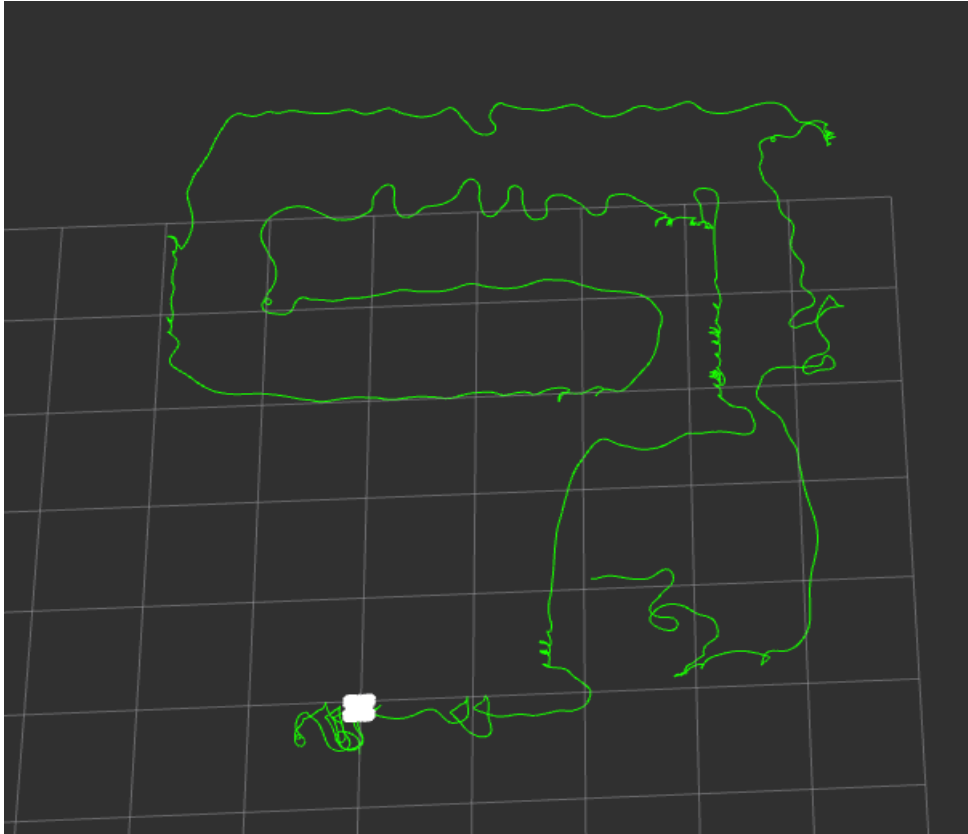
تغییر مهم بعدی این است که مقدار error در بخش قبلی صرفا اختلاف بین موقعیت ربات و مارجینی بود که از دیوار set کرده بودیم ولی در این سوال من یک معادله ساده به شکل زیر نوشتم که با فاصله از دیوار روبرو هم رابطه دارد.

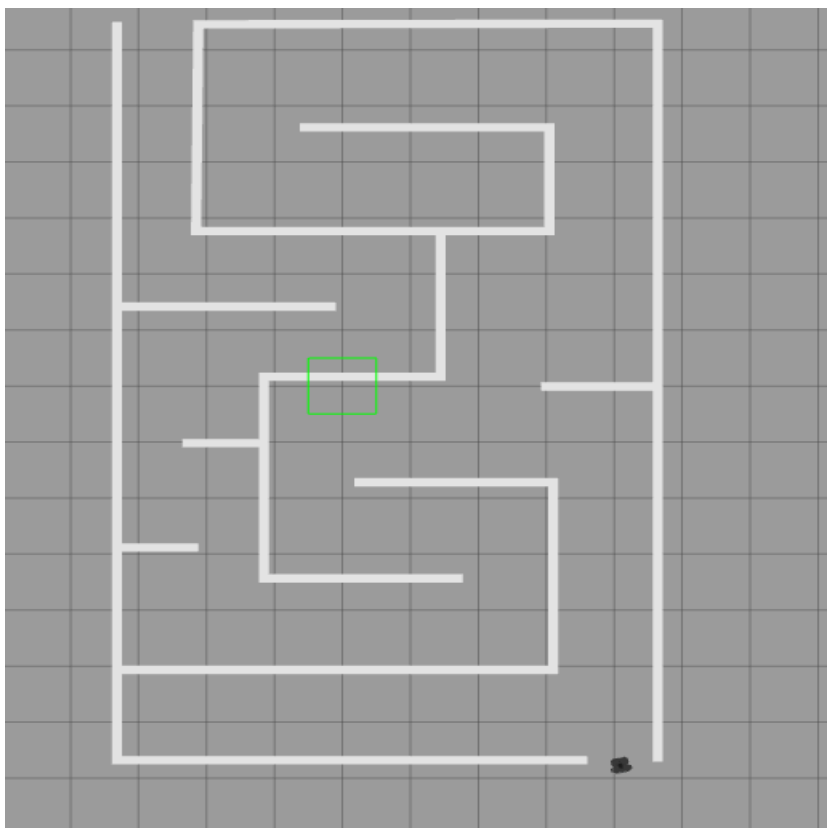
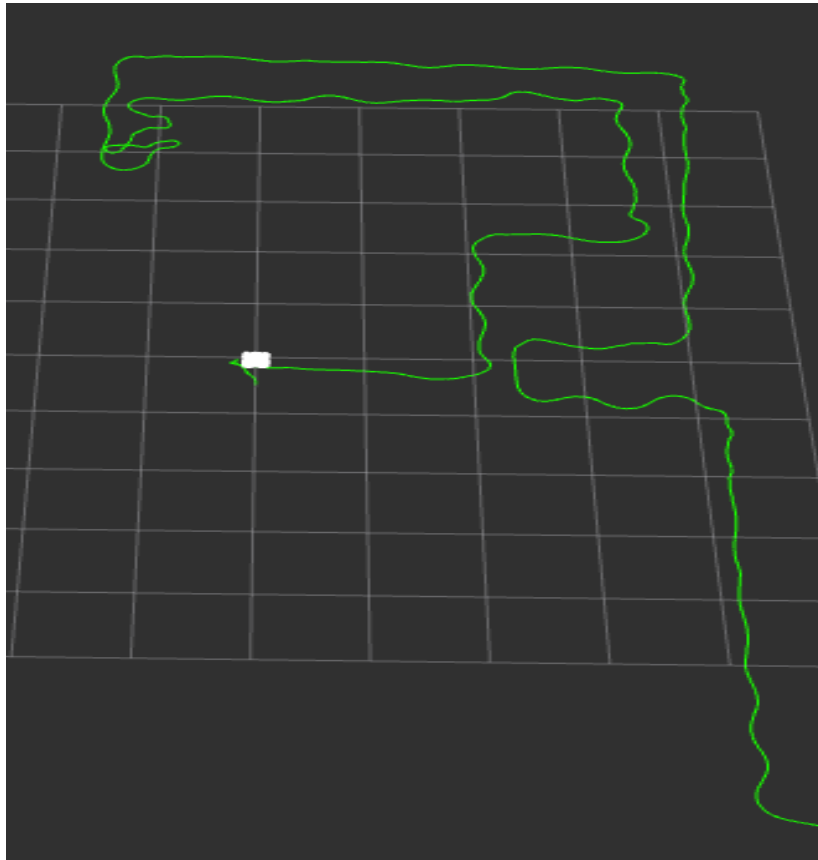
```
error = left_dist - self.distance_margin - (1/front_dist) * self.distance_margin * self.tuner
```

نحوه کار این معادله به این صورت است که فاصله از دیوار روبرو هم دخیل میکند. اگر فاصله از دیوار روبرو زیاد باشد این معادله به خاطر کسر ۱ تقسیم بر front\_dist به left\_dist - distance\_margin که همان فرمول محاسبه error بخش قبل بود تبدیل میشود. حال هرچقدر این فاصله کمتر باشد باعث میشود که ربات در نزدیکی دیوار به سمت راست متمایل شود.

در انتها ضریب tuner هم صرفا برای tune کردن تاثیر فاصله از دیوار جلو در این معادله کنترلی است.

در شکل های زیر مسیرهای طی شده توسط ربات را میتوانید ببینید. دوشکل اول ربات نتوانسته است که مسیر را درست پیمایش کند. در شکل سوم با تغییر ضرایبی مانند kd, kp و distance\_margin و tuner ربات توانست maze را حل کند.





(ج)

در این بخش برای حرکت مستقیم به سمت هدف از تابع زیر استفاده میکنیم. این تابع ابتدا heading ربات را به سمت هدف قرار میدهد و سپس در آن راستا شروع به حرکت میکند.

```
91 def go_to_point(self):
92     speed = Twist()
93
94     inc_x = self.goal.x - self.x
95     inc_y = self.goal.y - self.y
96
97     angle_to_goal = atan2(inc_y, inc_x)
98
99     # print(degrees(angle_to_goal), degrees(self.yaw))
100
101     if abs(angle_to_goal - self.yaw) > 0.15:
102         print("rotating")
103         speed.linear.x = 0
104         speed.angular.z = -0.3
105     else:
106         print("moving to point")
107         speed.linear.x = 0.175
108         speed.angular.z = 0
109
110     self.cmd_publisher.publish(speed)
111
112     return speed
```

نحوه کار این بخش به این صورت است که سه بولین تعریف شده است. این بولین ها فاصله ربات از دیوار روبرو، upper left و downer left است. اگر فاصله از هر کدام از این دیوار ها از یک حد مشخصی کمتر باشد بولین مربوط true و در غیر این صورت false میشود. حال اگر دوتا یا بیشتر از این بولین ها true باشد در این صورت ربات باید دیوار را دنبال کند و در غیر این صورت باید مستقیم به سمت هدف حرکت کند. دو شکل زیر لوپ اصلی کد آمده که توضیحات آن گفته شد.

```

131 while not rospy.is_shutdown():
132     print("debug 1")
133
134     error = left_dist - self.distance_margin
135
136     P = self.kp * error
137     I = self.ki * integral
138     D = self.kd * derivative
139
140     if sum(self.is_obstacle) >= 2:
141         speed.angular.z = P+I+D
142         speed.linear.x = self.linear_Speed
143         print(" follow walll ")
144     else:
145         speed = self.go_to_point()
146
147     self.cmd_publisher.publish(speed)

```

```

148
149     ## update controller
150     if front_dist > 0.7:
151         self.is_obstacle[0] = False
152     else:
153         self.is_obstacle[0] = True
154
155     if upper_left > 0.7:
156         self.is_obstacle[1] = False
157     else:
158         self.is_obstacle[1] = True
159
160     if downer_left > 0.7:
161         self.is_obstacle[2] = False
162     else:
163         self.is_obstacle[2] = True
164
165     derivative = error - privous_error
166     privous_error = error
167
168     left_dist = self.distance_from_left_wall()
169     upper_left = self.distance_from_upper_left_wall()
170     downer_left = self.distance_from_downer_left_wall()
171     front_dist = self.distance_from_front_wall()
172
173     self.rate.sleep()

```

شکل های زیر مسیر پیمایش ربات و همچنین ربات در هنگام رسیدن به هدف را نشان میدهد.

