

توجه: برای بخش امتیازی قسمت **map generation** انجام شده است.

گام اول: اضافه کردن نقشه مورد نظر به سیستم عامل ربات

بدین منظور فایل funky-maze.world باید به دایرکتوری مربوط به پکیج شبیه سازی ربات اضافه شود. محل دایرکتوری مورد نظر به کمک دستورات زیر بدست می آید.

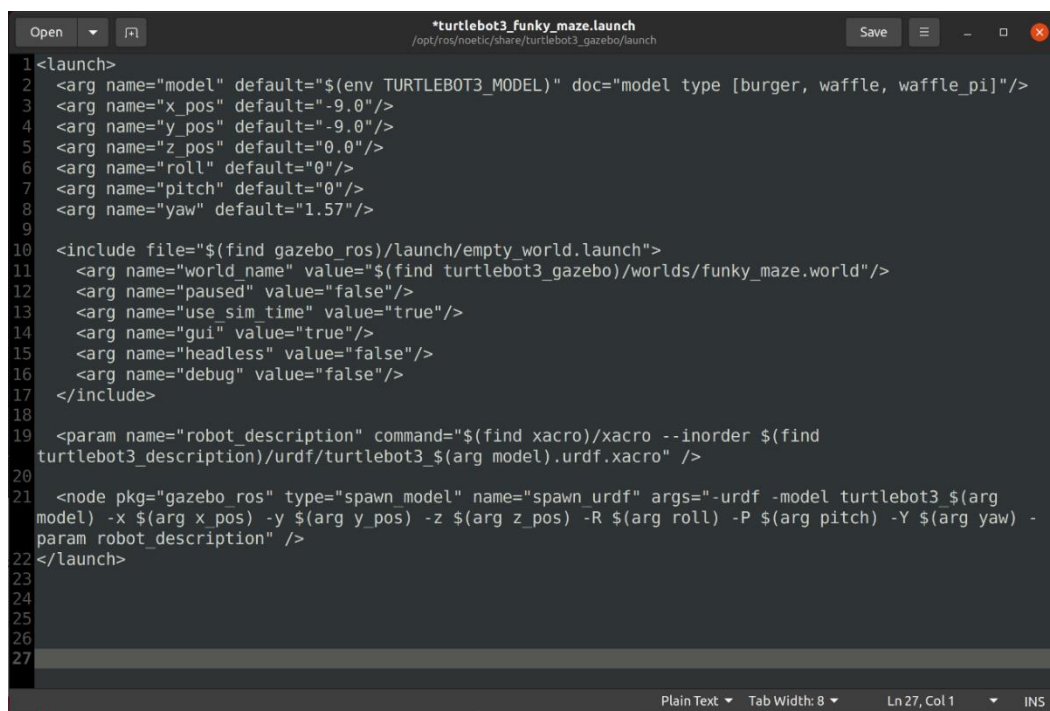
```
roscd turtlebot3_gazebo/
```

```
cd worlds
```

سپس یک فایل launch مخصوص این نقشه به دایرکتوری پکیج شبیه سازی اضافه می شود.

```
cd ../launch
```

```
sudo touch turtlebot3_funky_maze.launch
```

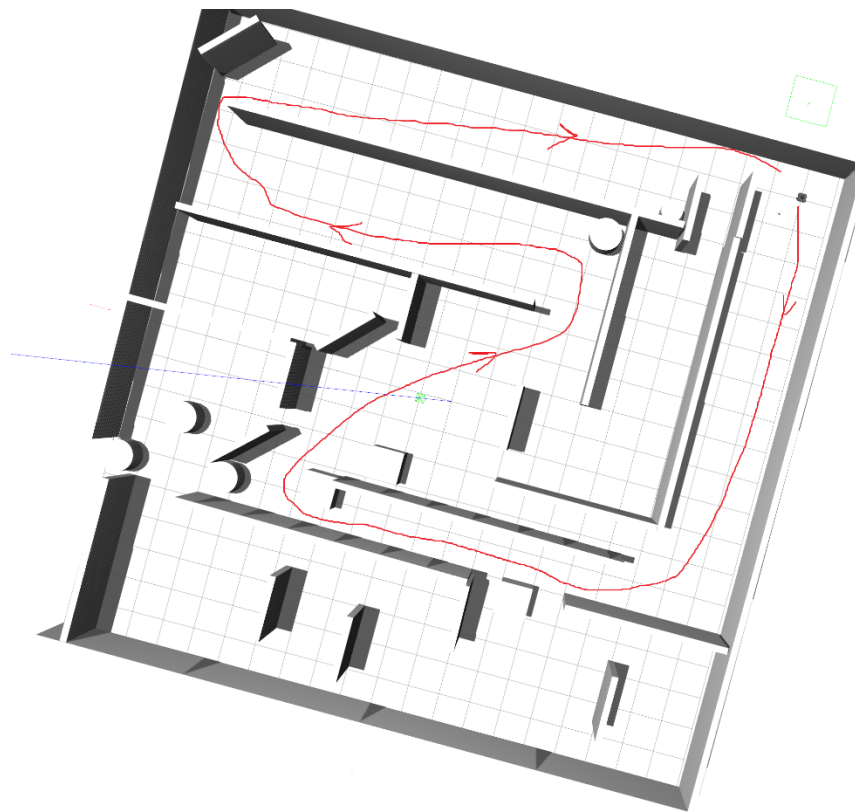


```
<?xml version="1.0"?>
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="x_pos" default="-9.0"/>
  <arg name="y_pos" default="-9.0"/>
  <arg name="z_pos" default="0.0"/>
  <arg name="roll" default="0"/>
  <arg name="pitch" default="0"/>
  <arg name="yaw" default="1.57"/>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find turtlebot3_gazebo)/worlds/funky_maze.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>
  <param name="robot_description" command="$(find xacro)/xacro --inorder $(find
turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />
  <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-urdf -model turtlebot3_$(arg
model) -x $(arg x_pos) -y $(arg y_pos) -z $(arg z_pos) -R $(arg roll) -P $(arg pitch) -Y $(arg yaw) -
param robot_description" />
</launch>
```

با توجه به شکل فوق، مختصات محل اولیه ربات در $x=-9$, $y=-9$, $z=0$ و هم چنین جهت ربات با نود درجه (1.57 رادیان) دوران نسبت به محور z تعریف شده است.

اکنون می توان با اجرای دستورات زیر، شبیه سازی را اجرا کرد:

```
source /opt/ros/noetic/setup.bash
roslaunch turtlebot3_gazebo turtlebot3_funky_maze.launch
```



گام دوم: ساخت پکیج اصلی

در این مرحله مانند تمرین قبل عمل کرده و دستورات زیر اجرا می شوند تا پکیج اصلی پروژه ساخته شود.

```
mkdir -p catkin_ws/src
cd src
source /opt/ros/noetic/setup.bash
catkin_init_workspace
catkin_create_pkg funky_maze std_msgs rospy
cd ..
catkin_make
cd src/funky_maze/src
```

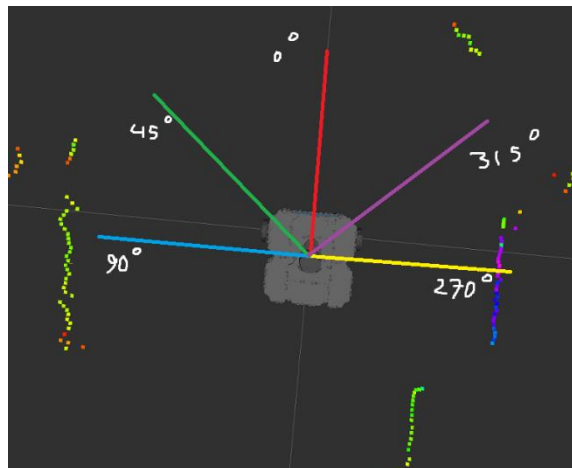
این پکیج متشکل از سه فایل اصلی به نام های زیر خواهد بود:

-1 obstacle_detector: یک نود با همین نام می سازد که تایپیک /scan را سابسکرایب کرده و سپس داده های مورد نیاز پروژه را به تایپیک /obstacle_detected پابلیش می کند.

```
touch obstacle_detector.py
chmod +x obstacle_detector.py
```

```
obstacle_detector.py > callback
1  #!/usr/bin/env python
2
3  import rospy
4  from sensor_msgs.msg import LaserScan
5
6  rospy.init_node('obstacle-detector')
7  pub = rospy.Publisher('/obstacle_detected', LaserScan, queue_size = 10)
8  laser = LaserScan()
9
10
11 def callback(msg):
12     current_time = rospy.Time.now()
13     laser.header.stamp = current_time
14     laser.header.frame_id = 'laser'
15     laser.angle_min = 0.0
16     laser.angle_max = 6.28318977355957
17     laser.angle_increment = 0.017501922324299812
18     laser.time_increment = 0.0
19     laser.range_min = 0.11999999731779099
20     laser.range_max = 3.5
21     laser.ranges = [msg.ranges[0],msg.ranges[45],msg.ranges[90],msg.ranges[270],msg.ranges[315]]
22     laser.intensities = [0.0,0.0,0.0,0.0,0.0]
23     pub.publish(laser)
24
25
26 if __name__ == '__main__':
27     sub = rospy.Subscriber('/scan', LaserScan, callback)
28     rospy.spin()
```

با توجه به شکل فوق، داده های مورد نیاز پروژه، رنج های تولیدی سنسور لیدار در زوایای 0، 90، 270 و 315 هستند که توسط این نود به نود کنترل کننده ربات ارسال می شود.



2- velocity_controller.py: یک نود با همین نام می سازد که تایپیک /obstacle_detected را سابسکرایب کرده و سرعت خطی و زاویه مورد نظر را به تایپیک /cmd_vel ارسال می کند.

```
touch velocity_control.py
chmod +x velocity_control.py
```

```

1 #!/usr/bin/env python
2 import rospy
3 from sensor_msgs.msg import LaserScan
4 from geometry_msgs.msg import Twist
5 import time
6
7 def callback(dt):
8     print('-----')
9     print('Range data in front side: {}'.format(dt.ranges[0]))
10    print('Range data in front-left side: {}'.format(dt.ranges[1]))
11    print('Range data in left side: {}'.format(dt.ranges[2]))
12    print('Range data in right side: {}'.format(dt.ranges[3]))
13    print('Range data in front-right side: {}'.format(dt.ranges[4]))
14    print('-----')
15
16    if dt.ranges[0]>0.5:
17        if dt.ranges[1]>0.5 and dt.ranges[4]>0.5:
18            move.linear.x = 0.3
19            move.angular.z = 0.0
20        elif dt.ranges[1]<0.5:
21            move.linear.x = 0.0
22            move.angular.z = -0.3
23        elif dt.ranges[4]<0.5:
24            move.linear.x = 0.0
25            move.angular.z = 0.3
26        elif dt.ranges[1]<0.5 and dt.ranges[4]<0.5:
27            move.linear.x = -0.3
28            move.angular.z = 0.0
29    else:
30        if dt.ranges[2]>dt.ranges[3]:
31            move.linear.x = 0.0
32            move.angular.z = 0.3
33        if dt.ranges[2]<dt.ranges[3]:
34            move.linear.x = 0.0
35            move.angular.z = -0.3
36    pub.publish(move)
37
38    move = Twist()
39    rospy.init_node('velocity-controller')
40    pub = rospy.Publisher("/cmd_vel", Twist, queue_size=10)
41    sub = rospy.Subscriber("/obstacle_detected", LaserScan, callback)
42
43    rospy.spin()

```

با توجه به شکل فوق، این فایل ابتدا داده های سنسور را روی ترمینال نمایش می دهد سپس الگوریتم کنترلی به صورت زیر اجرا می شود:

در صورتی که رنج تولیدی توسط سنسور در زاویه صفر (سمت روبروی ربات) بیشتر از نیم متر باشد و هم چنین رنج های مربوط به زوایای 45 و 315 نیز بیشتر از نیم متر باشد ربات حرکت مستقیم رو به جلو خواهد داشت.

در صورتی که رنج تولیدی توسط سنسور در زاویه صفر بیشتر از نیم متر باشد ولی رنج مربوط به زاویه 45 کمتر از نیم متر باشد ربات حرکت دورانی در جهت ساعتگرد خواهد داشت تا زمانی که این فاصله به بیشتر از نیم متر برسد.

در صورتی که رنج تولیدی توسط سنسور در زاویه صفر بیشتر از نیم متر باشد ولی رنج مربوط به زاویه 315 کمتر از نیم متر باشد ربات حرکت دورانی در جهت پادساعتگرد خواهد داشت تا زمانی که این فاصله به بیشتر از نیم متر برسد.

در صورتی که رنج تولیدی توسط سنسور در زاویه صفر کمتر از نیم متر باشد داده های مربوط به رنج های زوایای 90 و 270 مقایسه می شوند. اگر رنج زاویه 90 بیشتر باشد ربات حرکت دورانی پادساعتگرد، و در غیر این صورت حرکت دورانی ساعتگرد خواهد داشت.

3- foot_print.py: از این فایل به منظور ایجاد یک تاپیک برای نمایش مسیر پیموده شده توسط ربات در محیط rviz استفاده می شود.

گام سوم: پیاده سازی حلقه کنترلی

پس از اجرای فایل شبیه سازی دستورات زیر به ترتیب اجرا می شوند تا حلقه کنترلی کامل شود.

```
source ../../devel/setup.bash
```

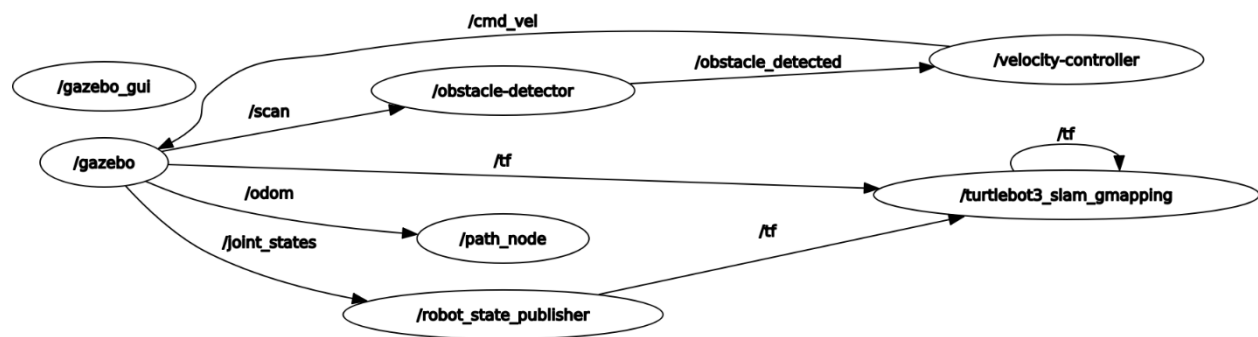
```
roslaunch funky_maze obstacle_detector.py
```

```
roslaunch funky_maze velocity_control.py
roslaunch funky_maze footprint.py
```

هم چنین جهت اجرای محیط rviz و مشاهده نقشه برداری توسط ربات می توان از پکیج turtlebot3_slam استفاده کرد.

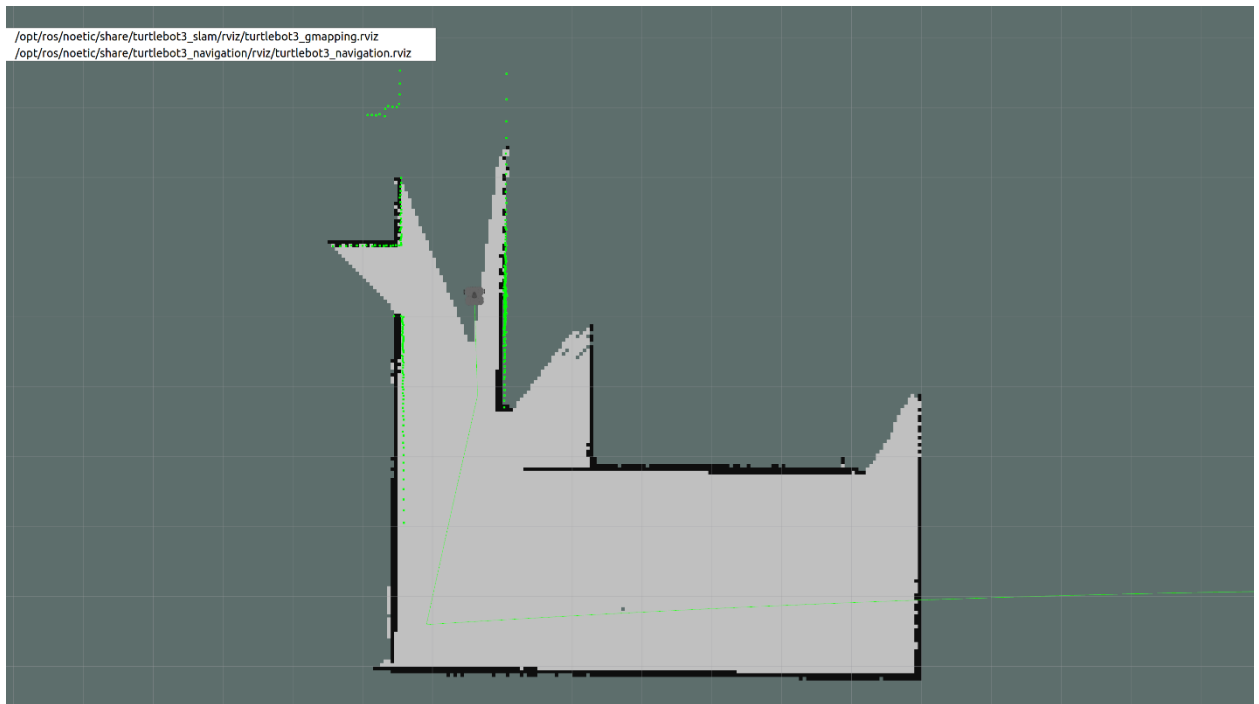
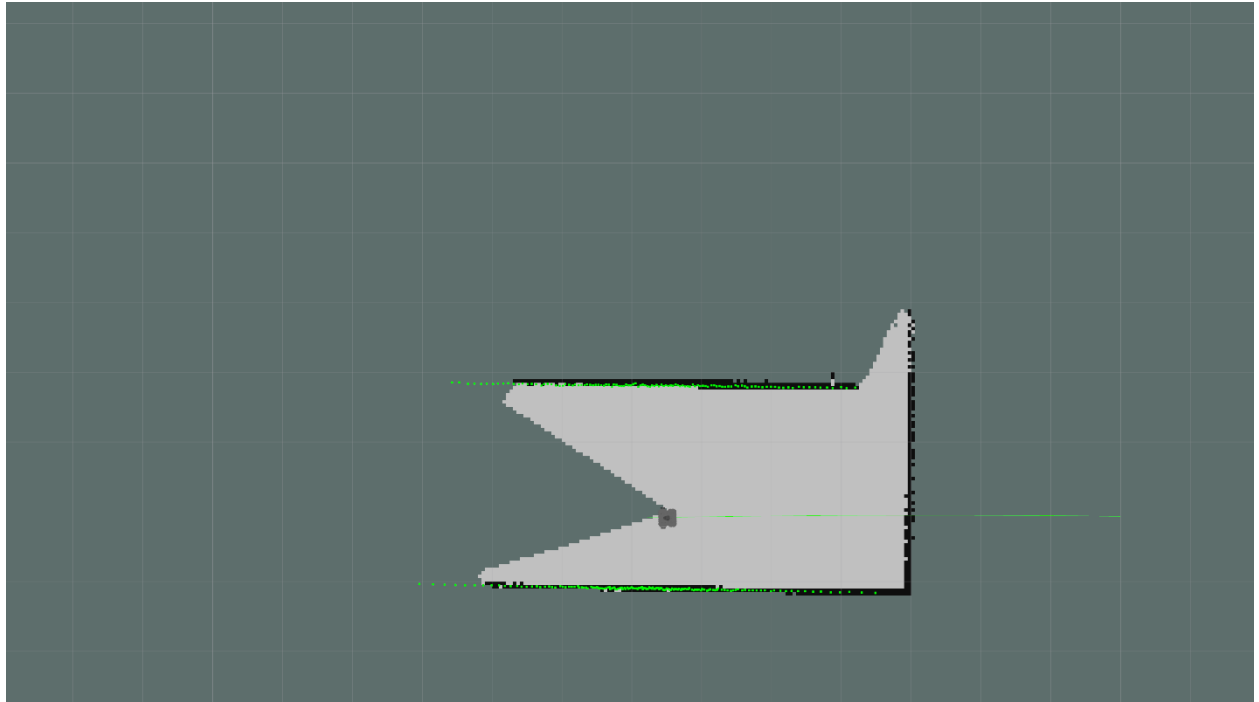
```
roslaunch turtlebot3_slam turtlebot3_slam.launch
```

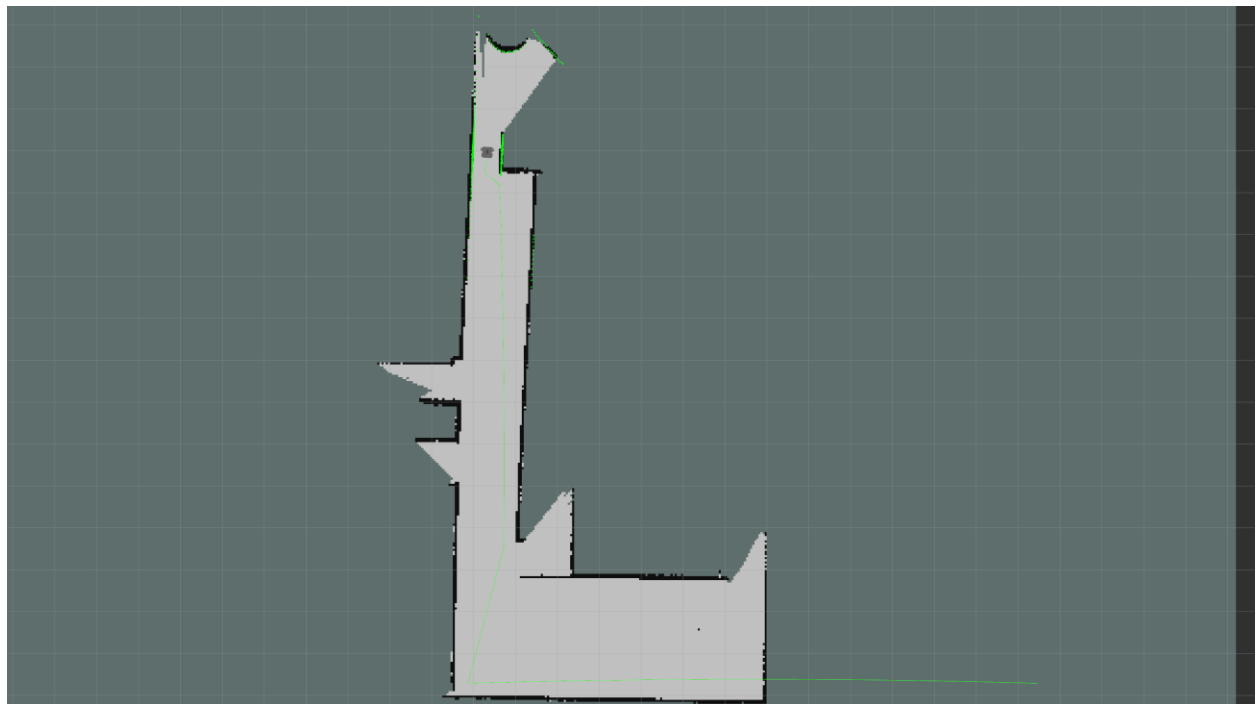
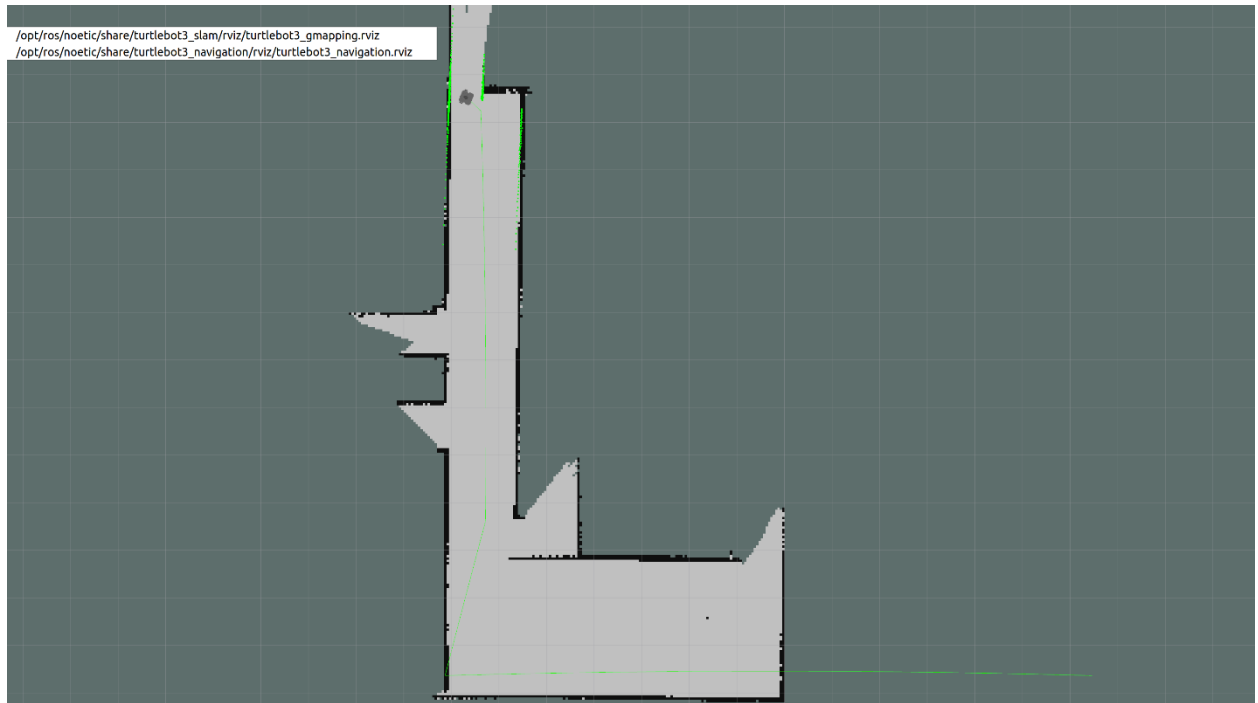
در نهایت شبکه نود ها و تاپیک ها به صورت زیر خواهد بود:

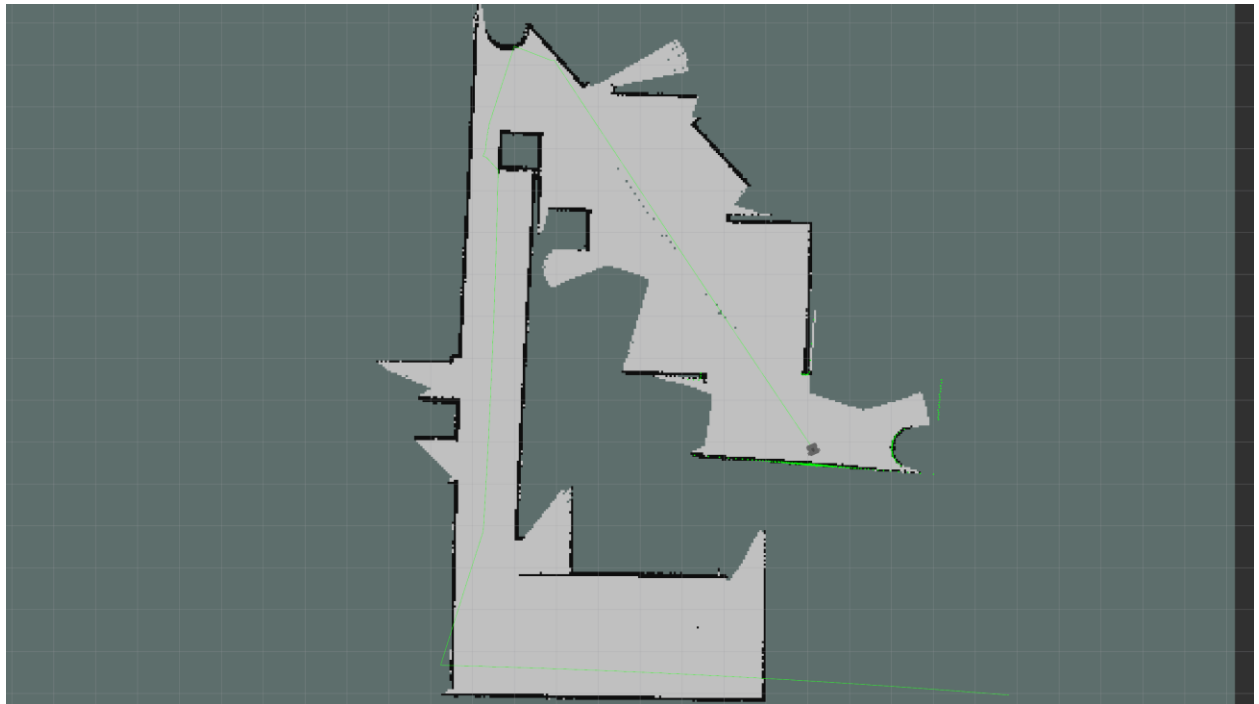
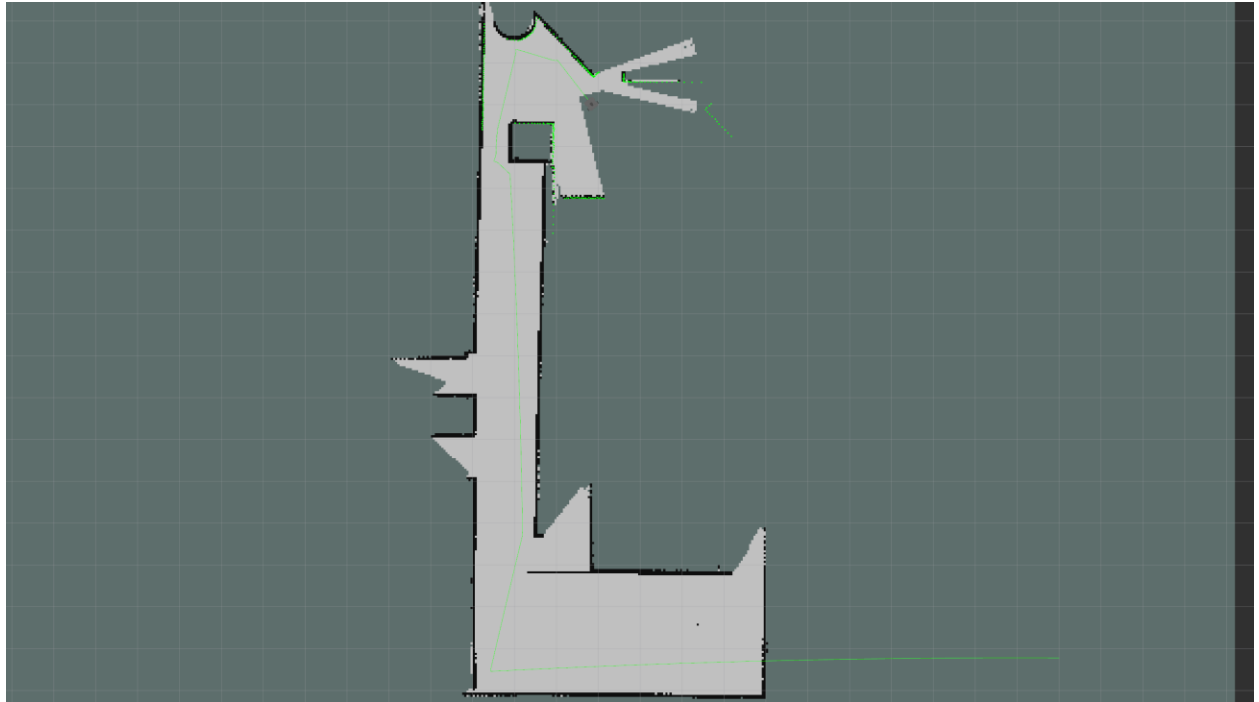


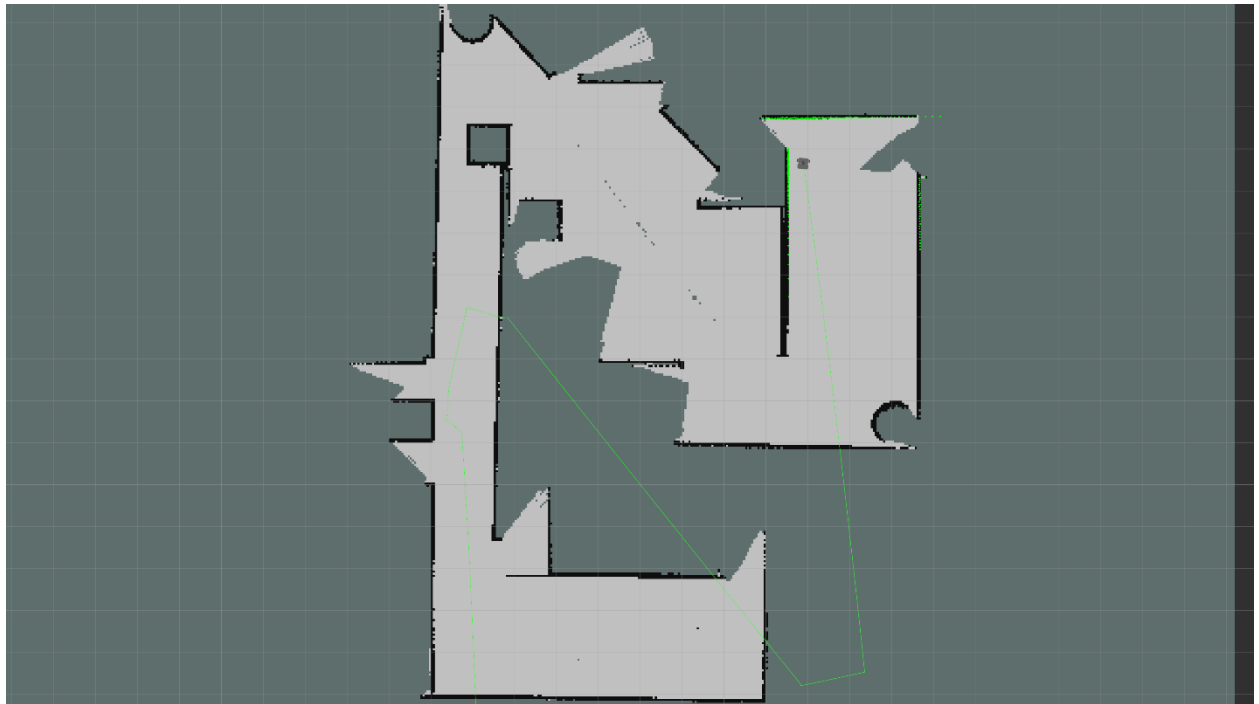
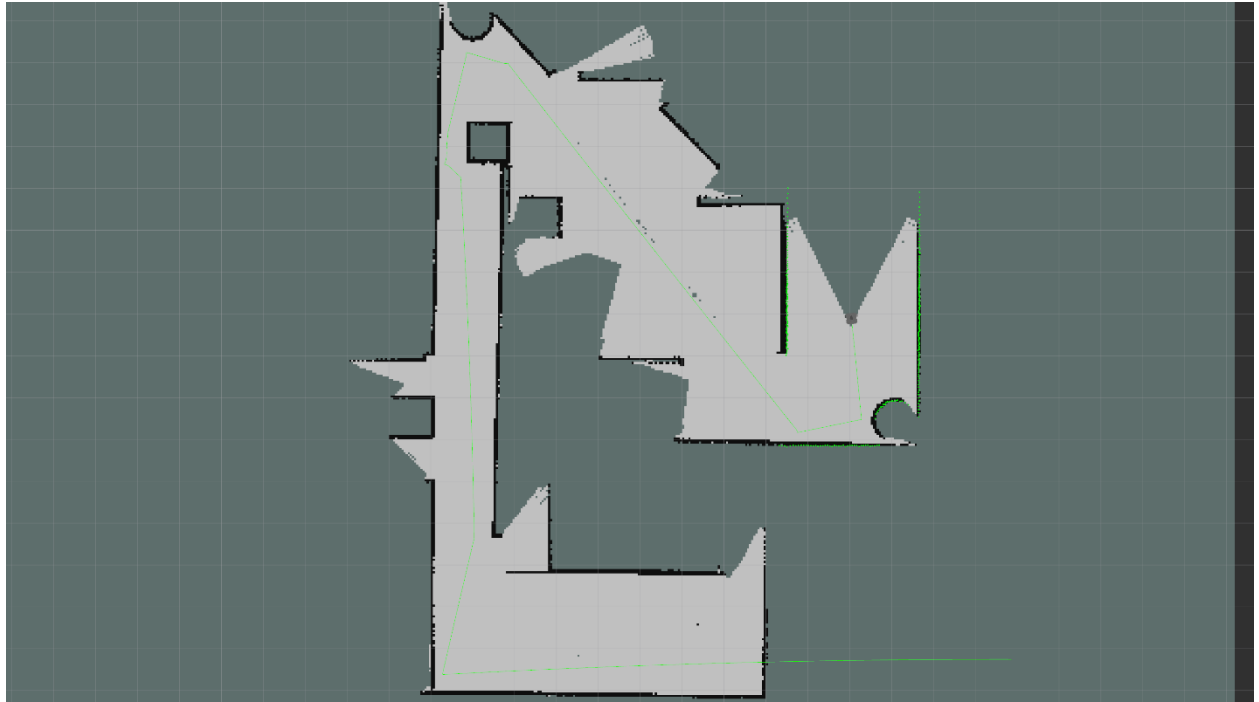
گام چهارم: اجرای شبیه سازی

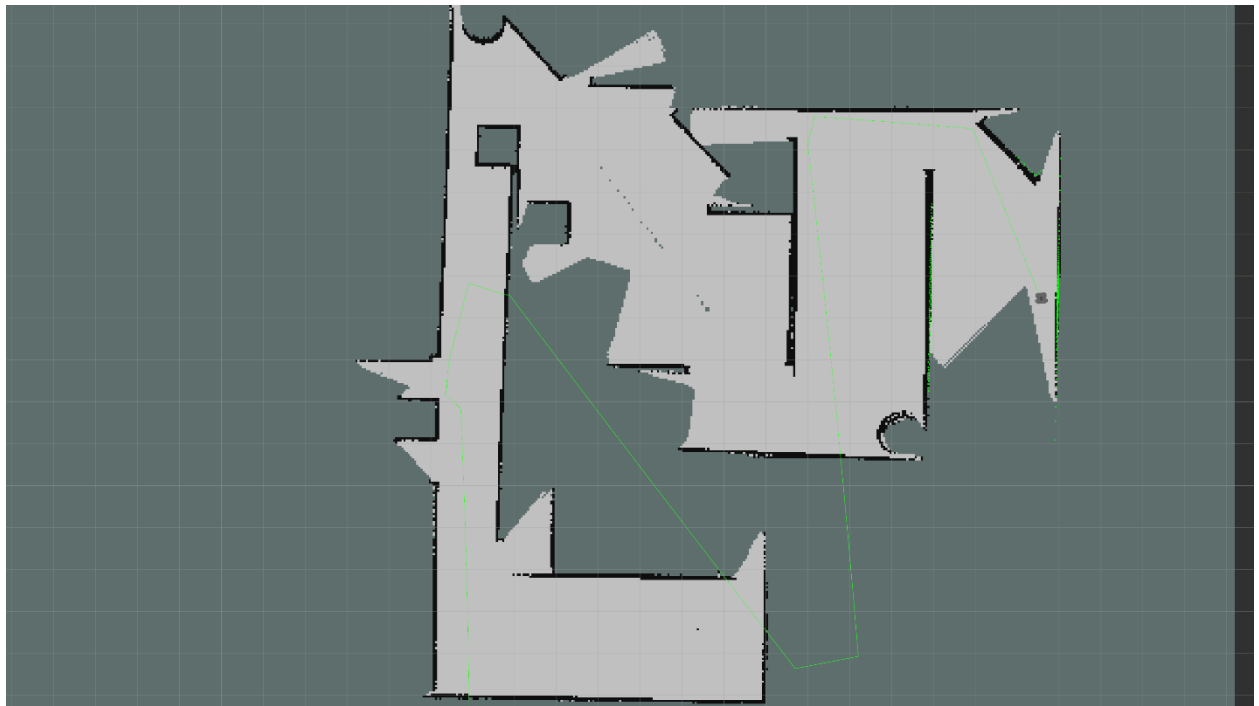
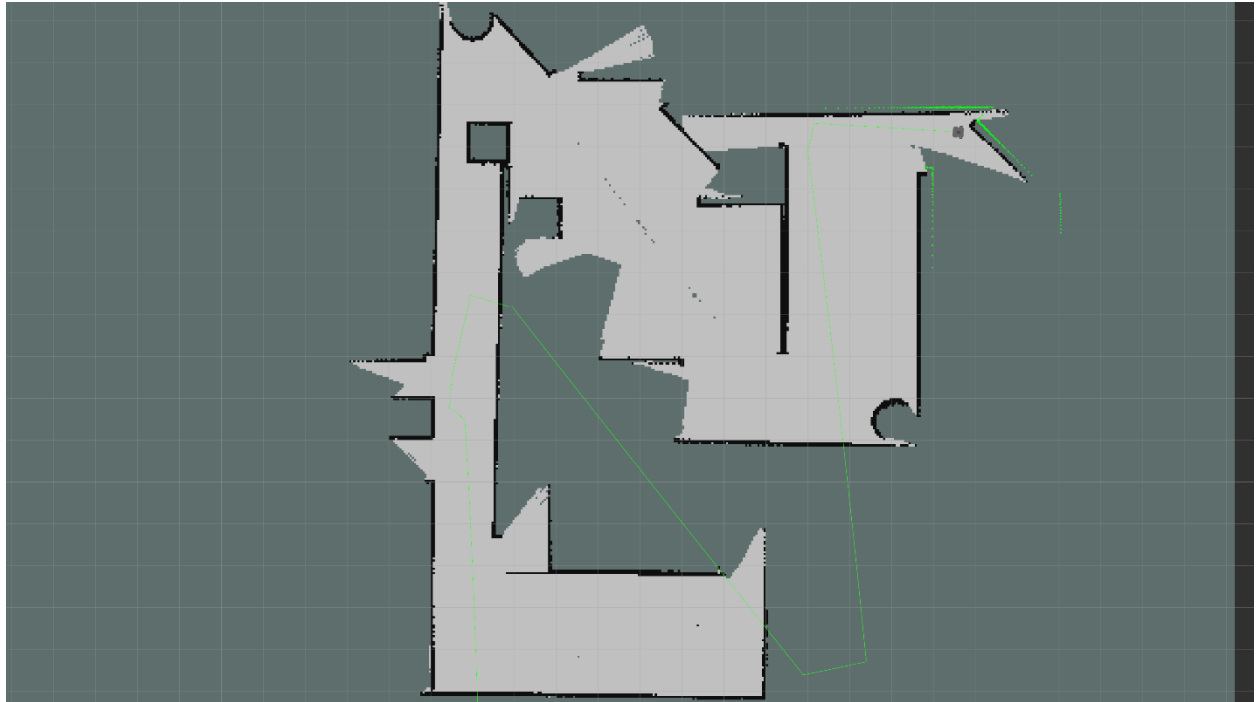
در نهایت حلقه کنترلی پیاده سازی شده و نتایج شبیه سازی به ترتیب صورت زیر خواهند بود:

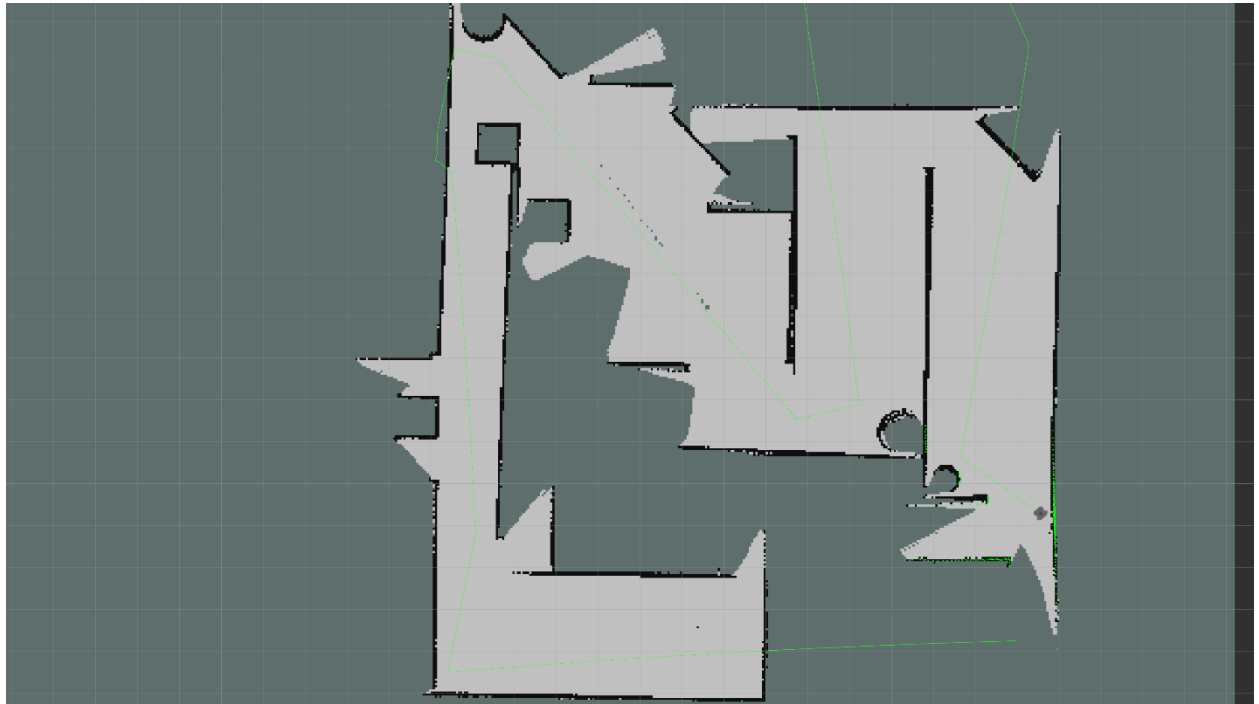
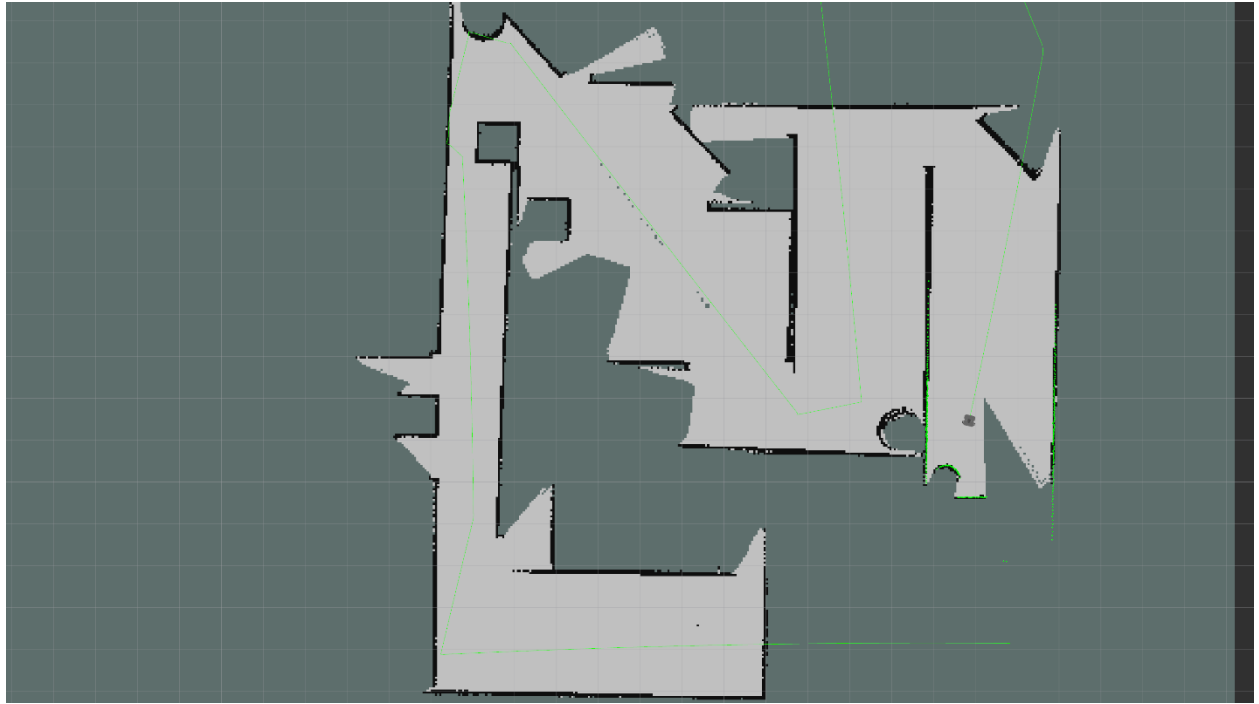


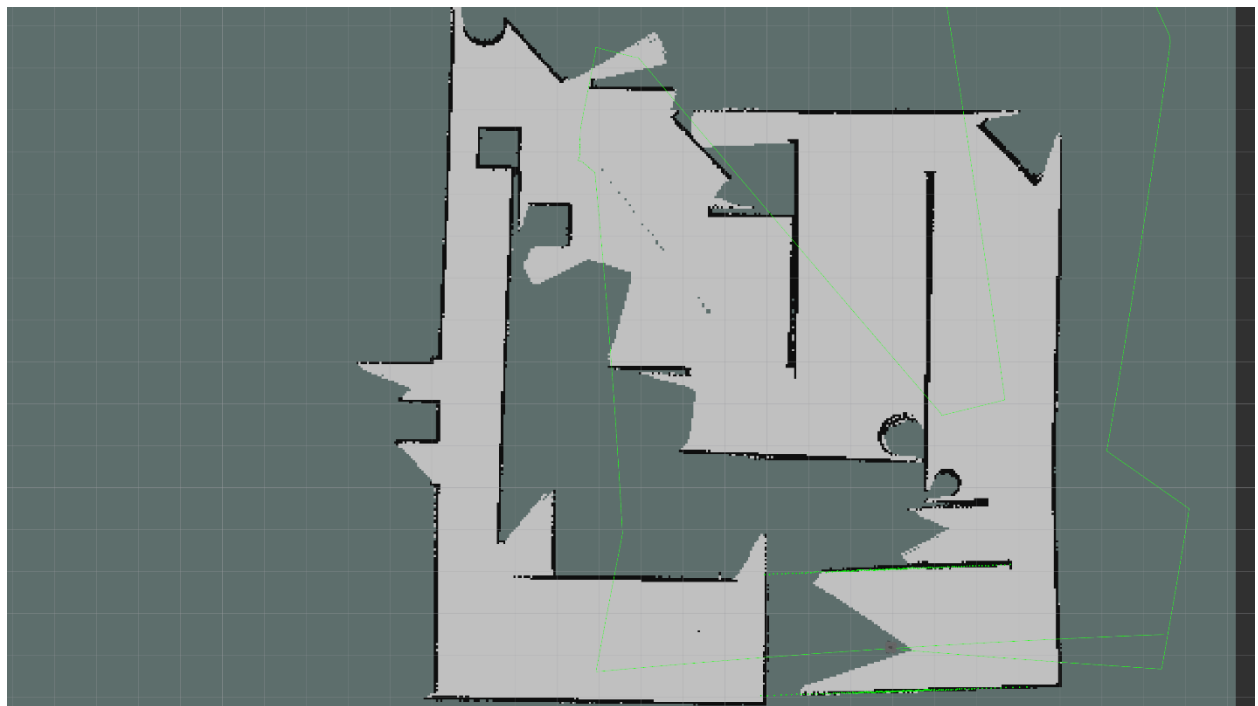
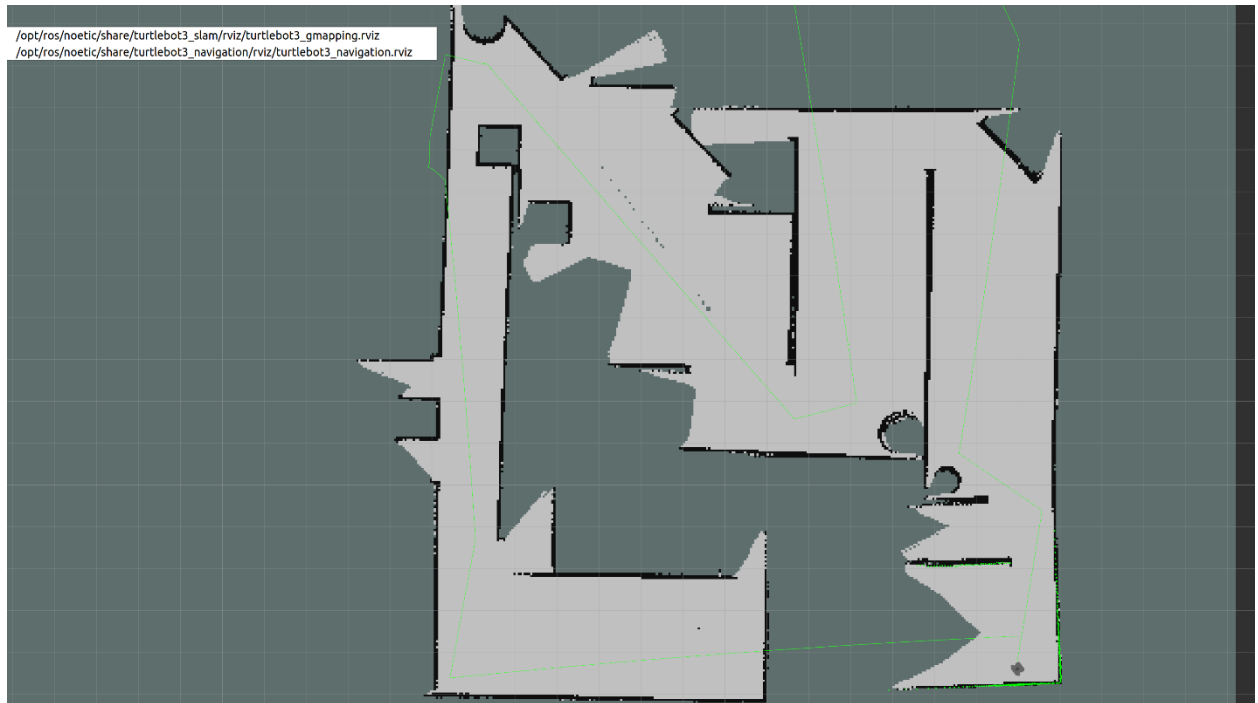


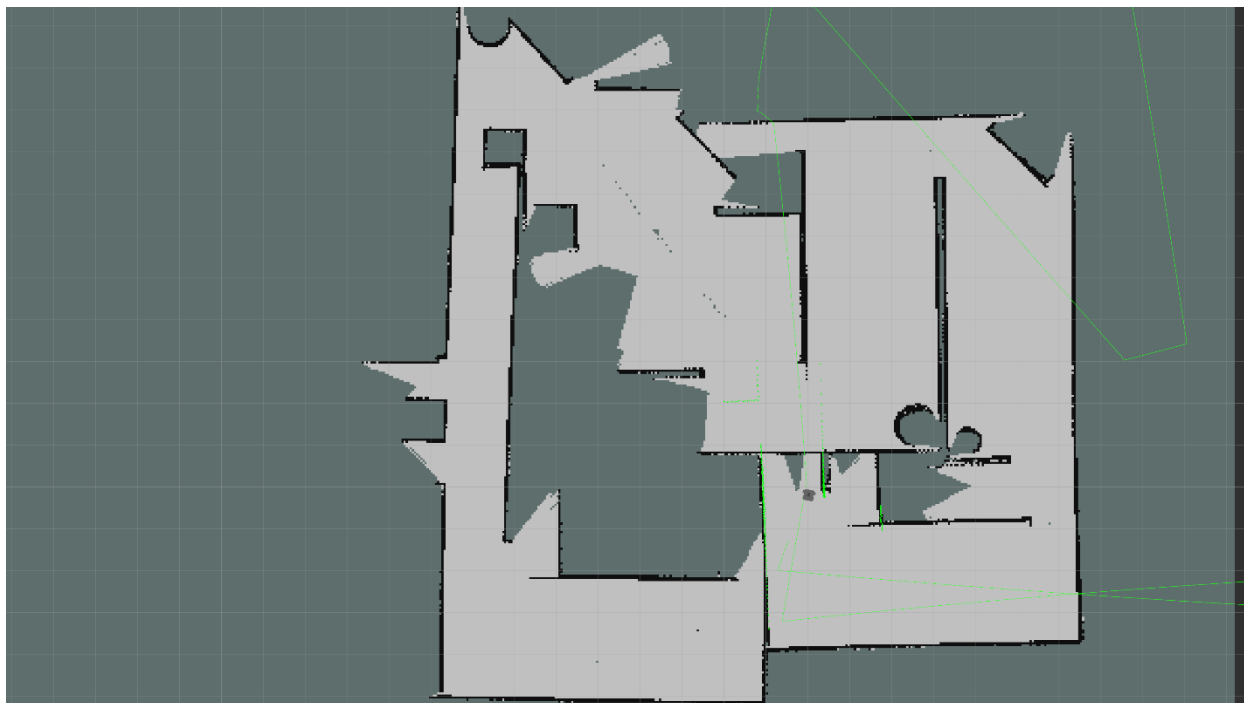






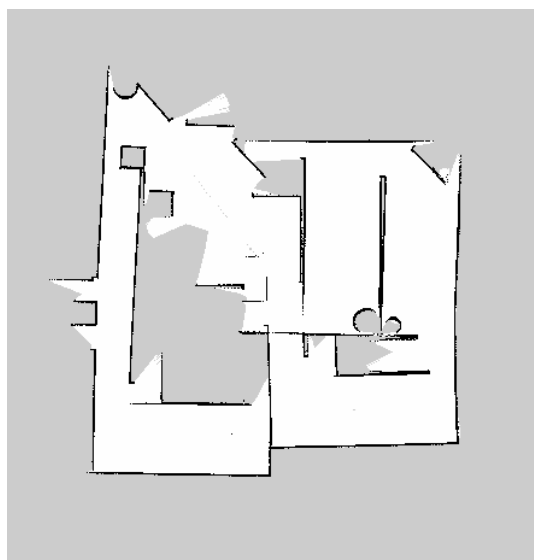






در نهایت می توان با اجرای دستور زیر در یک ترمینال دیگر، نقشه برداری را در قالب یک فایل yaml ذخیره کرد

```
roslaunch map_server map_saver -f ~/map
```



هم چنین با استفاده از نسخه کامل شده ی این فایل و پکیج turtlebot3_navigation بهترین مسیر را در محیط rviz برای رسیدن به مقصد پیدا کرد.