I use TurtleBot3 , so I use map.pgm to finish my lab.

1. Start the map server node:

```
Summary: 5 packages finished [1.67s]
rcpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$ source ~/Xue_Yanyaobo_ws/install/setup.ba
sh
rcpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$ ros2 run nav2_map_server map_server --ros
-args -p yaml_filename:=~/Xue_Yanyaobo_ws/src/TurtleBot/maps/map.yaml
[INFO] [1717220794.543385456] [map_server]:
        map_server lifecycle node launched.
        Waiting on external lifecycle transitions to activate
        See https://design.ros2.org/articles/node_lifecycle.html for more inform
ation.
[INFO] [1717220794.543695010] [map_server]: Creating
```

2. Start the PID controller node:

```
rcpsl@rcpsl-ros2-vm: ~/Xue_Yanyaobo_ws
cpsl@rcpsl-ros2-vm:~$ cd ~/Xue_Yanyaobo_ws
cpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$ source ~/Xue_Yanyaobo_ws/install/setup.ba
h
cpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$ ros2 run TurtleBot pid_controller
```

3. Start the motion planner node:

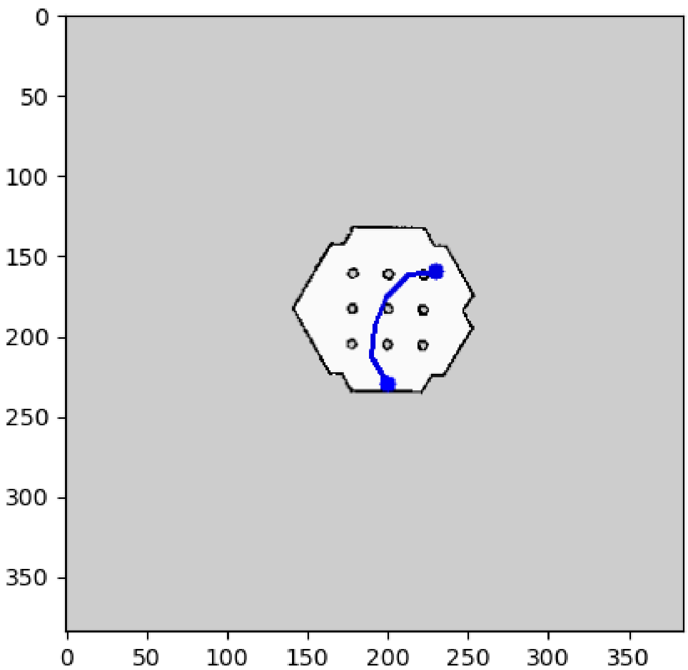
```
rcpsl@rcpsl-ros2-vm: ~/Xue_Yanyaobo_ws
rcpsl@rcpsl-ros2-vm:~$  cd ~/Xue_Yanyaobo_ws
rcpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$ source ~/Xue_Yanyaobo_ws/install/setup.ba
sh
rcpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$ ros2 run TurtleBot motion_planner
```

4. ros2 run TurtleBot rrt_node :

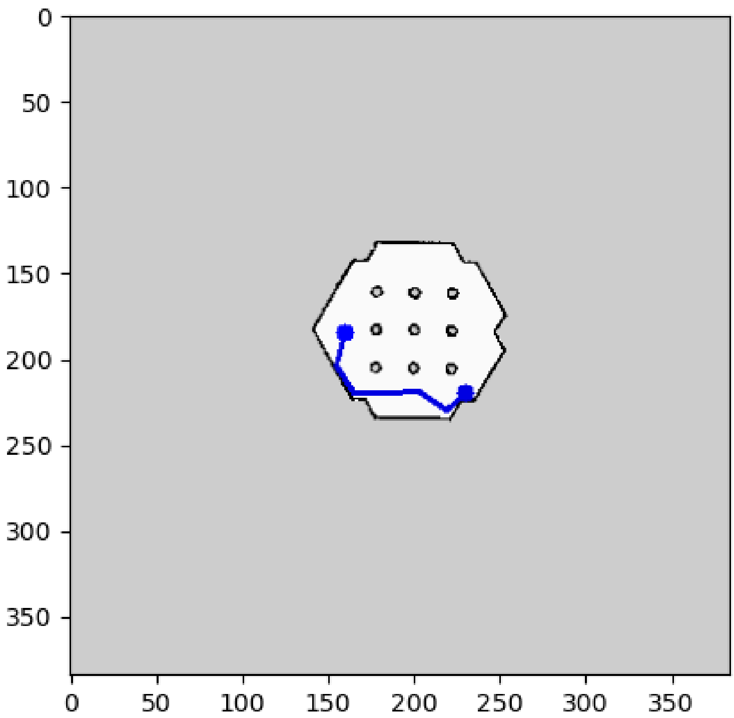
```
rcpsl@rcpsl-ros2-vm: ~/Xue_Yanyaobo_ws
rcpsl@rcpsl-ros2-vm:~$ cd ~/Xue_Yanyaobo_ws
rcpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$  source ~/Xue_Yanyaobo_ws/install/setup.b
ash
rcpsl@rcpsl-ros2-vm:~/Xue_Yanyaobo_ws$ ros2 run TurtleBot rrt_node
```

5. We can see few result here:

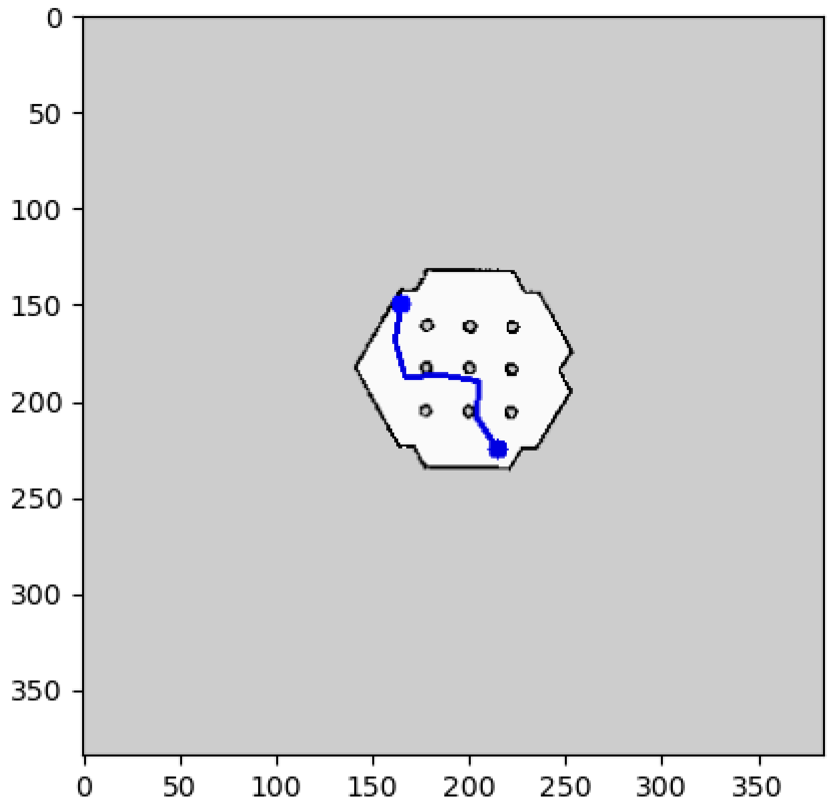
   start(200,230)
   end (230,160):



start(160,185)
end (230,220):



start(165,150)
end (215,225):



6. Code explanation:

    a. Motion_Planner.py:

        In this Motion_Planner.py script, I have implemented a motion planning node for a TurtleBot in ROS2. My goal was to design a node that handles receiving the current robot position, the target pose, and the computed trajectory to guide the robot towards its goal effectively.

        First, I created the MotionPlanner class, which inherits from Node. This class handles the subscriptions and publications needed for motion planning. I initialized subscriptions to three key topics: /odom to get the current position of the robot, /trajectory to receive the path computed by the RRT algorithm, and /target_pose to obtain the desired goal position. Additionally, I set up publishers for /start_goal and /reference_pose, which are essential for interacting with the RRT node and the PID controller node, respectively.

I then defined a fixed start position at coordinates (200.0, 230.0). I chose this fixed start position to simplify testing and ensure the robot always begins from the same location.

In the odom_callback method, I update the robot's current pose whenever a new Odometry message is received. This information is critical for following the trajectory. The target_pose_callback method processes the target pose message, and once both the current pose and target pose are available, it calls send_start_goal to publish these positions to the RRT node.

The send_start_goal method constructs a Float64MultiArray message containing the current and target positions and publishes it to the /start_goal topic. This triggers the RRT node to compute a trajectory.

Once the trajectory is received in trajectory_callback, I store it and initiate the process of sending reference poses to the PID controller node. The follow_trajectory method continuously monitors the robot's progress along the trajectory and sends the next reference pose whenever the robot gets close to the current target.

The send_reference_pose method publishes the next point in the trajectory to the /reference_pose topic, guiding the PID controller to adjust the robot's movement.

Finally, the get_distance method calculates the Euclidean distance between the robot's current position and the target point to determine when to move to the next point in the trajectory.

b. PID_Controller.py:

In this PID_Controller.py script, I have implemented a PID controller node for a TurtleBot in ROS2. My objective was to design a node that listens to the reference pose and current robot position to compute and publish velocity commands that guide the robot to the desired pose.

First, I created the PIDController class, inheriting from Node. In the constructor, I set up two subscriptions: one to the /reference_pose topic to receive the target position and orientation, and another to the /odom topic to get the current robot pose. I also established a publisher to the /cmd_vel topic to send velocity commands to the robot.

I defined the PID constants (kp, ki, kd) for both linear and angular velocities. Initially, I set the integral (ki) and derivative (kd) terms to zero, planning to adjust them later if needed.

In the listener_callback method, I updated the reference pose whenever a new message was received. Similarly, in the odom_callback method, I updated the current pose. When both the reference pose and current pose were available, I called the control_loop method.

The control_loop method was crucial. It calculated the distance error between the current position and the reference position, and the angular error between the current orientation and the target orientation. Using these errors, I computed the linear and angular velocities based on the proportional control (kp). I then created and published a Twist message with these computed velocities.

To handle the robot's orientation, I implemented the get_yaw_from_quaternion method, which extracted the yaw angle from the quaternion representing the robot's orientation. I also created the normalize_angle method to ensure the angular error stayed within the range of -pi to pi.

c. RRT_Node.py:

In this RRT_Node.py script, I have implemented an RRT (Rapidly-exploring Random Tree) node for path planning in ROS2. My goal was to create a node that subscribes to the occupancy grid map and the start and goal positions, then uses the RRT algorithm to compute and publish the trajectory for the robot to follow.

First, I created the RRTNode class, inheriting from Node. In the constructor, I set up two subscriptions: one to the /map topic to receive the occupancy grid map, and another to the /start_goal topic to get the start and goal positions. I also established a publisher to the /trajectory topic to send the computed path.

In the map_callback method, I processed the occupancy grid map message to extract the resolution and origin of the map, and I reshaped the map data into a 2D array. This data is essential for the RRT algorithm to understand the environment.

The start_goal_callback method was triggered whenever a new start and goal position was received. If the map was already available, I called the compute_rrt_path method to generate the path from the start to the goal position.

In the compute_rrt_path method, I converted the real-world coordinates of the start and goal positions into map indices. I then converted the map into an image format that the RRT algorithm could work with. Using the find_path_RRT function from the rrt module, I computed the path in terms of map indices. After obtaining the path, I converted it back into real-world coordinates.

The coord_to_index and index_to_coord methods handled the conversion between real-world coordinates and map indices. The map_to_image method transformed the occupancy grid map into a binary image where free space was white, occupied space was black, and unknown space was gray.

Finally, the publish_trajectory method published the computed path as a Float64MultiArray message to the /trajectory topic. This trajectory could then be followed by the robot.