Assignment #2 - TurtleBot the Ratatouille

Due Date: check Ninova

This is an individual assignment, group work is not allowed!

Any type of cheating is strictly prohibited. Please, submit your own work!

You can ask your questions via e-mail (bicer21@itu.edu.tr) or message board in Ninova

Summary

Objective:

The assignment focuses on following waypoints, and integrating fallback mechanisms to handle obstacles.

Assignment: The TurtleBot acts as a chef assistant in a restaurant. It must perform a series of tasks in sequence:

1. Go to Storage: Navigate to the storage area.
2. Pick up the vegetables: Simulate waiting for vegetables to be picked up (sleep for 5 seconds).
3. Bring to Chef: Deliver the vegetables to the chef in the kitchen.
4. Cook Meal: Simulate waiting for the meal to be cooked (sleep for 5 seconds).
5. Serve Meal: Deliver the meal plate to the guest.
6. Report to Manager: Navigate to the manager's room and inform them that the serving is complete.

Tasks:

- Waypoint Following:
  - Follow the waypoints extracted by the robot previously.
- Handling unexpected obstacles:
  - Waypoints are derived according to a map that has been acquired previously. Yet, new supplies are arrived recently to the restaurant and has not been arranged yet. These supplies can be along your pathway. You need to find a way to overcome such supplies and resume your waypoint following with the next waypoint (**your solution cannot be world-specific)**. Your approach should be a systematic approach that can also be utilized in a different world with different set of waypoints (i.e. motion planning).

Submission Type: Source code of your controller to be submitted is called **waypoint_follower.py**. A skeleton for the file can be found in an archive supplied with the assignment description. It should be possible to compile the package by **placing the file into a ros2 workspace** as described in further and running the colcon build. **Do not forget to add your id**'s to your code as a comment in the specified section.

A report (1-2 pages) that addresses the followings:

- **Waypoint Following:**
  - Explain the implemented waypoint following approach.
  - Explain how have you handled unexpected obstacles along the pathway. Think of an alternative approach to handle obstacles along the route and explain how it would affect the navigation trajectory and elapsed time for this assignment.
- **Results:**
  - Include overall elapsed time to finalize the tasks.
- **Challenges:**
  - Discuss any challenges faced during implementation.
  - Discuss how you have overcome challenges.

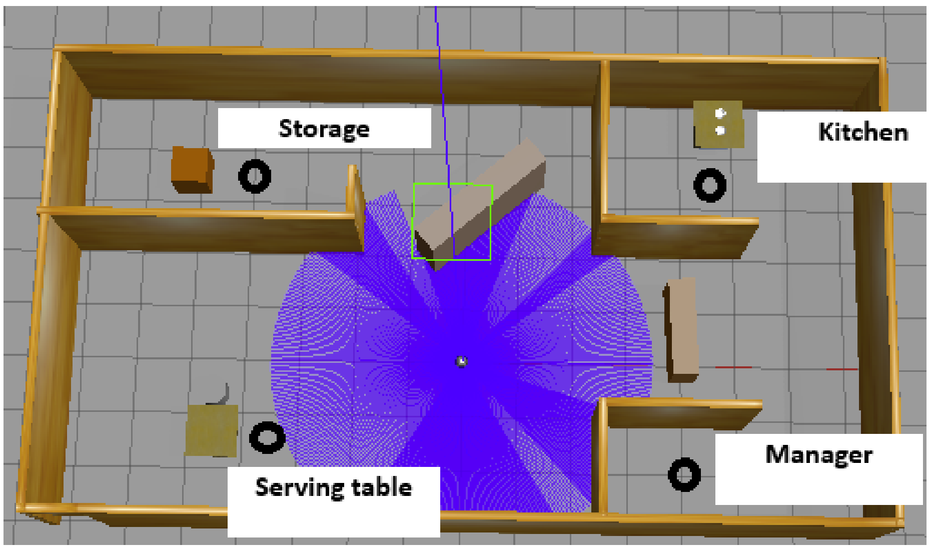


Fig. 1. Café Environment

Skeleton Code & Referee and Preparing Solution

“**referee**” and “**waypoint_follower**” are two ros nodes in this assignment. Referee checks followings:

- **Has robot reached the next waypoint?**
- **Does the robot have the desired yaw (orientation)? (Robot should be facing the same direction as the arrow shows)**
- **Elapsed time for the robot to reach the next waypoint?**
- **Elapsed time for the robot to finish all routes?**

Referee logs the answers for these questions in terminal and also in text files as **wpReport.txt and timeReport.txt in maps folder of package directory**. You can track your performance with these reports. wpReport includes waypoints that your robot reaches and how much time robot got late to that point as “WP0+,-2.30566668510437”. This says that first waypoint is achieved and the yaw of the robot is correct (i.e if there is no + sign, you only achieved that waypoint linearly). You got early to that waypoint by ~2.3 seconds as there is a “-” sign (i.e there would not be “-” if you reach to that waypoint late.). **It should be noted that even though your solution might be fast enough, lateness can be expected. Required time to reach those waypoints are selected arbitrarily. This report is given for you to track yourself and improve your solution iteratively.** timeReport will contain elapsed time and total lateness if all waypoints are finished.

Source code for answer node **will contain your implementation to move the robot through the waypoints as fast as possible**. A skeleton code is given for guidance. **Skeleton code already stores the route and waypoint variables. Map is also stored in variables (see Map in information section below).**

There is a launch file (“nav_launch.py”) that executes following:

- Gazebo
- MapServer for publishing map
- AMCL for localizing robot in the given map
- Nav2 Lifecycle to configure and launch MapServer and AMCL in coordination.
- Launch file of Robot State Publisher

- RViz (map should be already included)
- Referee Node

<u>To finalize the preparation of workspace:</u>

- Move the downloaded "**assignment2**" folder to **"~/ros2_ws/src"** folder in home.
- Check dependencies: within the **"~/ros2_ws" execute <u>"rosdep install -i --from-path src --rosdistro humble -y"</u>**
- Within the **"~/ros2_ws", execute <u>"colcon build"</u> command**
- If encountered: ignore bash warning in this terminal, it would disappear after you open new terminal.

Open up a new terminal and execute launch file by:

```
ros2 launch assignment2 a2_launch.py
```

Then, skeleton code can be executed by:

```
ros2 run assignment2 waypoint_follower
```

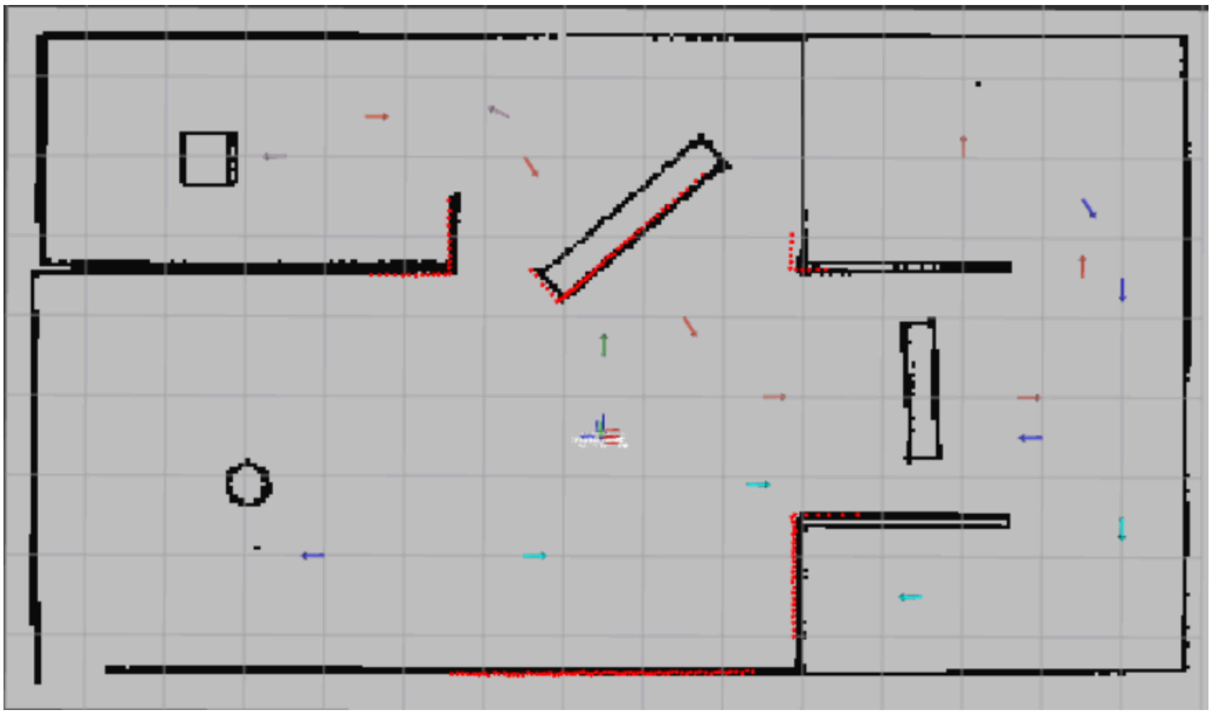


Fig. 2. RViz (each route is indicated with a different color)

Gazebo (see Fig.1) and RViz (see Fig. 2) will open up and you will see our world after executing launch file. **Goal is to reach to the end of routes. Elapsed time is being measured after robot starts the movement.** In RViz, you can see the **map with the occupancy grid, robot, TF2 frames, routes, next waypoint and LaserScan points** (see Fig. 2). **Next waypoint will be appeared with a different color.** Each route has a different color.

Marking Criteria

- Publishing smooth movements as much as possible (do not stop unless you need to).
- Publishing movements that get the robot to the target position.
- Publishing movements that get the robot to the target position and orientation (desired yaw).
- Responding quickly to changes in waypoint (after you achieved a waypoint, new waypoint will
- be assigned.).
- Publishing movements that can pass the obstacles and resume waypoint following.

- Elapsed time will be evaluated. Your robot should be fast as much as possible without risking the robot to stumble (you are encouraged to discuss the elapsed time performance among yourselves).
- Obstacle/Wall collision avoidance
- **Modular code:** divide operations into functions to make code seem more elegant.
- **Code with comments** (good practice).
- Sufficient explanation for specified questions in Report document.

Hints:

- Using teleoperation to get the sense of robot navigation.
- Implementation of a controller like Proportional (P) control or another advanced controller.
- Figuring out how to utilize /route1, /route2, /route3, /route4 topics provided by the referee.

Information regarding messages, topics and so on

You may check docs.ros2.org for more than below.

How to use scan data: Laser scan data will be available within the callback function that your program registers when it subscribes to the /scan topic by creating a subscription object with “create_subscription” function. A LaserScan message is published through this topic, and detail about the data structure of this message can be found.

This message basically tells us how far are the objects that surround our robot via 360 laser sensors on its body. In skeleton code, helper code is left to guide you how to manage this message.

How to navigate the robot: You can send Twist messages to the robot to say how much velocity it should have in linear or angular aspect. This message can be used for three dimensional objects. As we are only concerned with two dimensions, you will only need to set the **x component of the linear** velocity and the **z component of the angular velocity**. As with the laser scan, you can access the fields of the **geometry_msgs/Twist** message, which will be “linear” and “angular”, which themselves are messages of type **geometry_msgs/Vector3**. **Skeleton code includes helper code in scan callback for sending movement messages.** To see what fields are available in a Vector3 message, see the message definition.

PoseArray, PoseStamped, Pose and Related Topics:
Pose: Pose is a representation of pose in free space, composed of position and orientation. Position is represented with geometry_msgs/Point type (x,y,z) and orientation is in Quaternion format (x,y,z,w).
PoseStamped: PoseStamped is a Pose with reference coordinate frame and timestamp. It consists of a header additionally which includes timestamp information and associated frame (map frame in our case). We used timestamp to add expected approximate elapsed time for reaching the waypoint.
PoseArray: An array of poses with a header for global reference.
Related Topics: /route1 returns a PoseArray of waypoints along the route 1. **/route2** returns a PoseArray of waypoints along the route 2. **/waypoint** returns the next waypoint with the type of PoseStamped.

TF2 and getting Robot’s Position: In order to obtain the robot's current pose, the TF package can be used to get the transform between the robot's original pose (which also happens to be the map frame as long as the robot’s odometry agrees with the localization subsystem) and its currently known pose according to the odometry. These poses are known to the TF transform manager as frames /odom and /base_footprint. In order to use TF well, you can examine carefully the output of the following commands to view what is going on:

```
ros2 run tf2_tools view_frames

evince frames.pdf
```

Also:
`ros2 run tf2_ros tf2_echo /base_footprint /odom`

More information

Map: If you want to use it, the information in this section can help. The map is provided in an array with occupancy_grid variable. Provided map is updated using the robot's laser sensors. Since the robot does not have the map information beforehand, the map will be updated as the robot navigates and senses the environment using laser sensors.

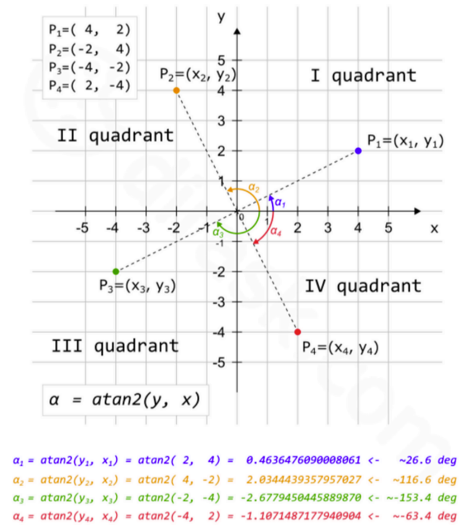
In order to access map information you can use the following expressions:

- Value (at X,Y): `occupancy_grid[X+ Y*map_width]`
  - Meaning of value:
    - 0 = fully free space.
    - 100 = fully occupied.
    - -1 = unknown (not yet explored).
- Coordinates of cell 0,0 in /map frame (bottom left corner of the map):`map_origin`
- The size of each grid cell:`map_resolution`

`For values between 0-100, by default, values greater than 0.65 are considered as completely occupied, values less than 0.25 are considered as completely free space.`

If, in the skeleton code, you set the value of the (const bool) variable **chatty_map to true**, it will print the map as a list. Also, **the rviz session shows the map.**

atan2: It is used in the "euler_from_quaternion" method in skeleton code. The math.atan2() method returns the arc tangent of y/x, in radians. Where x and y are the coordinates of a point (x,y). The returned value is between PI and -PI. **It is utilized to find the angle to face towards a specified point!**



```
α1 = atan2(y1, x1) = atan2( 2,  4) =  0.4636476090008061 <-   ~26.6 deg
α2 = atan2(y2, x2) = atan2( 4, -2) =  2.0344439357957027 <-  ~116.6 deg
α3 = atan2(y3, x3) = atan2(-2, -4) = -2.6779450445889870 <- ~-153.4 deg
α4 = atan2(y4, x4) = atan2(-4,  2) = -1.1071487177940904 <-  ~-63.4 deg
```

Fig. 2. atan2 function