**ROS\_Memo**

**Name**: Chengjie Lin

**Date**: 4/5/2016

**Introduction**

This memo includes 4 parts, data exchanging in mesh network, coordinated exploration, uncoordinated exploration and moving with the gradients.

1. **Data exchanging in mesh network**

**Purpose：**

With the adhoc\_communication package and olsrd package installed, we are able to create mesh network between multiple robots with multiple master so we don’t have to deal with the namespace conflicts issue of topics. The main use of the mesh network in this project I to exchange the probability distribution that represents the result of the intention functions between multiple Turtlebots.

**Terms:**

Intention function: The function to represent a certain intention in some area and the probability distribution that correspond to the intention, for example, the costmap can be intention function with the intention of avoiding obstacle.

Probability distribution: Distribution that represents the result of the different intention functions integrated together. The data structure is similar to OccupancyGrid, which is arrays and can be interpreted as the 2D grids like a map.

**Scene:**

Assuming that we have 5 robots running, each of them are running the TdmappingC.py (coordinated exploration) or TdmappingUC (uncoordinated exploration) in the background. Say the Ferdinand gets a probability distribution and encodes it, then it will use broadcast( ) function to broadcast the encoded probability distribution to its neighbors. The way it works is that the adhoc\_communication node running on the Ferdinand will serve as the middleman and it takes the encoded distribution then send it to the adhoc\_communication nodes running on Ferdinand’s neighbors. Its neighbor, Gonzalo, for example, will use its adhoc\_communication node to receive the encoded distribution from Ferdinand and the adhoc\_communication on Gonzalo will publish the message from Ferdinand to a user-made local topic on, /my3dMap for example., then the others nodes on Gonzalo can subscribe to the /my3dMap topic.

**How to send the array (data structure of probability distribution)**

In the adhoc\_commnucation package, there is a service called SendOccupancyGrid.srv that can be used to send the OccupancyGrid message, which is the array in fact. We can just call the sendMap() function in the adhoc\_communication.cpp of sending OccupancyGrid type message to send the probability distribution between nodes because the distribution is in fact arrays of value and is the same as the OccupancyGrid message. So to send array, we can just use the sendMap() which use the SendOccupancyGrid.srv.

SendOccuapncyGrid.srv contents:

string dst\_robot

string topic

nav\_msgs/OccupancyGrid map

---

uint8 status

**How to customize your own message type and send it**

If we want to use our own customized message type, we need to create the srv and implement the code to send it in the adhoc\_communication.cpp

1. Cd into the adhoc\_communication package

2. Make your own sendSomething.srv in the ./srv folder

3. In the ./src folder, edit the adhoc\_communication.cpp file

4. Find the sendMap() function

5. Write your sendSomething() function based on the sendMap() function, and we only need to make a few changes and called the sendSomething.srv

6. After we have our own sendSomething() function written, recompile the adhoc\_coomunication package.

7. The sendSomething() is ready to be called in other codes and help to send the customized message.

**2.Coordinated Exploration (TdmappingC.py)**

**Purpose:**

The purpose of coordinated exploration is to move the Turtlebot to explore the interesting area of the distribution in the space. For example, it will look around its surroundings and find the peak of the probability distribution. It will then set the pose of the peak as a goal, send the goal to the navigation stack and navigate to the goal and that means in the meantime the Turtlebot will also avoid the obstacles.

**Assumption:**

The probability distribution is viewed as a square of grids where the Turtlebot is in the center. The orientation of square of the probability distribution rotates with the orientation of the Turtlebot, which means it is aligned to the base\_link frame.

We also can create our own amcl launch file to reduce the size of costmap, so that it will be easier to process the OccupancyGrid data since we have less arrays to deal with. So we have the myamcl.launch as the launch file.

Myamcl.launch: Customized amcl launch file made for the Move with the Gradient package, in this launch file, we change the size of costmap to be 39\*39 instead of 79\*79, and of course, we can change the size of it to any appropriate number in mylocal\_costmap\_params.yaml, which defines the size of costmap used by myamcl.launch.

**Before running the code:**

1. Open the Terminal, run the ros\_mesh.py to start adhoc\_communication and mesh network
2. In the Turtlebot, set the ROS\_IP of every terminal to be the ip address of Turtlebot and run the myamcl.launch in the same terminal.
3. For each new terminal in the Turtlebot, we all need to set the ROS\_IP of every terminal to be the ip address of Turtlebot.
4. In the local computer, set the ROS\_IP of every terminal to be the ip address of Turtlebot running now, then run the turtlebot\_rivz\_launchers.
5. Run the TdmappingC.py.

**Scene:**

Here, for simplification, we will just use costmap as the probability distribution. We have three Turtlebot in the space, Ferdinand, Trinculo and Gonzalo. All of them have TdmappingC running in the background.

Let’s take Ferdinand for example, we have the following steps:

1. Get the position of the Ferdinand

2. We know that the costmap from Trinculo and Gonzalo will be published to the /my3dMap topic on Ferdinand, so we subscribe to the topic

3.Use a Queue to store the message from Gonzalo and Trinculo.

4. In the code, we set if the number of maps in Queue reaches 5, Ferdinand will integrate the 5 costmaps from other Turtlebots and from itself and get a general understanding of the surrounding and we get an integrated distribution.

5. Ferdinand finds the interesting area in the integrated distribution, here the interesting area are areas without the obstacles, which means the value of the interesting area grids is 0.

6. Use Navigate stack to go to the interesting area, during the navigation it will avoid the obstacles as well.

**Why it is called coordinated exploration?**

The reason it is called coordinated exploration is the way we control the Turtlebot to move to the goal, and we use the baseline of Navigation Stack, which means we need the coordinate of the goal and send the coordinate of the goal to function MoveToTarget(), and the MoveToTarget( ) function will use the Navigation Stack to move the Turtlebot to the goal coordinate and avoid the obstacle. In the MoveToTarget() function, we need to specify the coordinate on the map we want to navigate to and the this function will navigate the Turtlebot to the specified coordinate.

**Problems:**

1. Since I used the costmap to test here, we know that there are multiple areas without obstacles surrounding the Ferdinand, which means that there are multiple interesting areas Ferdinand can choose from. So from testing point of view, to solve this problem, I divided the integrated distribution into four quadrants. Ferdinand starts with looking at the first quadrant, if it finds the interesting area (obstacle-free area) in this quadrant, it goes to the interesting area. If not, it will look at the second quadrants, and repeat the previous step.

If there is only one interesting area in the surroundings, we then don’t need to worry about the above issue and there will be no need dividing the costamp into four quadrants, all we need to do is find the interesting area and use the MoveToTarget() function to move the Turtlebot.

2. Sometimes as it moves to the targeted point, it will fail to find the path halfway, then we need to stop and start over to find the another interesting area again.

**3.Uncoordinated Exploration(TdmappingUC.py)**

Uncoordinated Exploration is similar to the Coordinated Exploration with a few modifications.

**Why it is called Uncoordinated Exploration?**

Compared to CE(Coordinated Exploration), UE(Uncoordinated Exploration) doesn’t use the Navigation Stack, which means it doesn’t need to know the coordinate of the goal. All it needs know is the distance between the goal point and the orientation. And we use the distance and orientation information to control the Twist message to move the Turtlebot to a certain distance with certain angle.

The second difference is that in the CE, the interesting areas are areas with the value of 0. However in UE, the interesting areas are areas with the value of 100.

It is the similar idea to Move with Gradients. It divided the probability distribution into four quadrants, and these four quadrants are the neighbor grids for Turtlebot. It checks every neighbor grids, find the point with the value of 100, add the point of value of 100 and its neighbor’s value and then it will assign the point with the max total to be the goal point. After determining the goal point, we get the index of the goal point to calculate the distance and orientation. Last, move the Turtlebot to the goal point and repeat the above steps over again.

Since the UC exploration doesn’t use the Navigation stack, instead it just published the Twist message based on the distance and orientation we calculate before.

**4. Moving with the gradients (MoveG1.py and MoveG2.py)**

For the moving with the gradient part, I am using the hall.yaml as the map file. And make a probability distribution on the map.

**Purpose:**

We have a probability distribution of the integrated intention function in the space, and we need to move the Turtlebot to move along with the gradients to reach the peak, which is the interesting area to explore for us.

**Terms:**

Global goal cell: The cell the goal locates in, which will be the peaks of the distribution function when it comes to probability distribution.

Current cell: The current cell the Turtlebot locates and it changes over time as the Turtlebot moves into new cells.

Neighbor cell: The neighboring cell surrounds current cell. For example, if the current cell is (0.0), the neighbor cells are (1,0), (1,1), (0,1), (-1,1), (-1,0), (-1,-1), (0,-1), (1,-1).

Local goal cell: The special neighbor cell with the max value out of all the neighbor cellslocates in.

**Before running the code:**

1. In the Turtlebot, run the myamcl.launch(optional).
2. In the local computer, set the ROS\_IP of every terminal to be the ip address of Turtlebot running now, then run the turtlebot\_rivz\_launchers.(optional, it is just used to make user more aware of what’s going on).
3. Run the MoveG1.py.

**Settings:**

1. The origin of the map: 0.0
2. The size of the grid: 0.5m\*0.5m (changeable)
3. Initial Pose of Turtlebot: Arbitrary

**Processes:**

1. Put Turtlebot in a random position in the space presented in the map.
2. Node subscribes to the “amcl\_pose” topic to get the starting pose, which includes the position and orientation.
3. Check the cell in which the starting point locates
4. Set the goal position and also check the cell in which the goal point locates, and that will be the global goal cell.
5. Turtlebot check its neighbors and find the cell with the largest value, which will be a local goal for the next movement.
6. Turtlebot starts rotating and moving.
7. As the Turtlebot is trying to move to the center of the local goal cell, it will constantly check its pose, once it finds itself in another cell, it will again, check the neighbors and update the local goal cell.
8. Turtlebot continues doing the above until it reaches the global goal cell.

**Algorithms:**

1. How to get the cell value of the neighbors: In this simple, we just the use reciprocal of the distance from the neighbor cells to the global goal cell, in this way, we will get a cone presenting the interesting area in the map instead of a pit.
2. How do we label the orientation as the next local goal cell: The next local goal cell will be indexed based on the its orientation. Let’s take (0,0) as an example, it has eight neighbor cells: (1,0), (1,1), (0,1), (-1,1), (-1,0), (-1,-1), (0,-1), (1,-1). Starting from the very right one (1,0), we label these neighbor cells in a counter-clockwise direction. So the eight cells will be indexed as 0,1,2,3,4,5,6,7. This index will be useful in the control of the Twist message published to control the Turtlebot.
3. How Turtlebot uses Twist message: There code keeps on publishing Twist messages to the /cmd\_vel\_mux/input/navi topic. That means the Turtlebot is moving all the time. If the Turtlebot detects that it is in a new cell and the local goal cell changes, the Twist message will be changed too, if nothing is detected, the Twist message will not be changed.
4. How to set the twist.angular.z and twist.linear.x to move the Turtlebot with the tendency to reach its local goal cell, here we assume the cell with the highest value is the upperleft one, which will be indexed as 3.
   1. First, get the current pose of Turtlebot and extract the x1,y1 coordinate and orientation (presented as ).
   2. Get the x2,y2 coordinate of the center point in the local goal cell.
   3. Calculate the distance between the (x1,y1) and (x2,y2).
   4. Convert the Quartenion to Euler angle, so we get the orientation of (x1,y1) and (x2,y2) in Euler angle formatting as YAW1 and YAW2
   5. Calculate the angle difference between YAW1 and YAW2
   6. Set the linear.x to be 0.2
   7. Calculate how much time it will take to allow Turtlebot to travel through the distance between the (x1,y1) and (x2,y2), here we annotate it to be T
   8. Having known the T and YAW1 and YAW2, we set angular.z =(YAW1-YAW2)/T. In this way, we assume that when the Turtlebot travels through the required distance, it will turn the angle of (YAW1-YAW2).
   9. Once the Turtlebot detects that it is another cell, it will start over from step.a to step.i again.

**Output on terminal:**

There will be some output in the Terminals.

**Problems:**

1. Considering the following situation, the robot is in the same row as the global goal cell, so the neighbor cell with the max value is the left one in our case. But the Turtlebot’s orientation is currently -135 degree for example, which means it is facing towards the lower left cell but the local goal cell is the left one. There are two options here. First one, we stop going forward and start turning to -180 so the Turtlebot can face the global goal cell. The second one, the Turtlebot can keep on going forward and at the same time turning. But that will make the trace more like a curve. Here I implemented both of the above, MoveG1.py for the first option and MoveG2.py, MoveG3,py for the second option.

Better solution: Use a PID controller to control the angle. So if the global goal cell is far away from the Turtlebot, it will turn slower. If it is close to the Turtlebot, it will turn faster. So as the Turtlebot approaches the global goal cell, it is closer to the global goal cell and turning faster.

2. Since we are using amcl and the odom frame. So the Turtlebot drifts over time during its moving. What you see in the real space might have a little difference from what you see in Rviz.

**Difference between MoveG1.py, MoveG2.py, MoveG3.py**

The MoveG2 is created to solve the Problem1 mentioned before. So when the Turtlebot is aligned with the target in the same row or the same column, in MoveG2, it will starts turning to face the goal, then it will move towards the goal. In this way, it will reduce the path to get the goal.

MoveG3 and MoveG2, is of the same idea, the difference is the way the turn. MoveG3 uses publishing time of Twist message to control the angle turned. MoveG2 uses the odom frame and an error checking controller to control the angle turned.

**Improvements that can be made:**

1. A PID controller can be used to control the angular,z more accurately and solve the problem.1

2. Instead of using amcl localization, we can use a more accurate localization tool.

3. More advanced algorithms.

**5.Conclusion and feedbacks:**

It is a very meaningful semester for me and I learnt a lot and get a better understanding of myself. The ROS lectures are really enlightening and gave a very good insight into the robotic programming. I cannot be more grateful for this opportunity to do some work in the lab and granted to be in the ROS class because this experience makes me realize what I am really interested and what I really want to go into for my grad school.

Despite the efforts put into this project, I feel that I could have done better. The start is good, we know what we need to do and me and James works out the Adhoc mesh network. However, when it comes to moving with the gradients part, it didn’t go that smoothly because the misunderstanding of the requirements, so in fact I only got two weeks to work on the actual moving with the gradients part and it would be much better if I have more time.

Anyway, many thanks to everyone in the lab and especially Dr. Crick for teaching me a lot. I am leaving this summer but I will be back next fall for my last semester. So if you need a hand or someone to help to do some lab work, I will be very glad to volunteer.

Best regards!