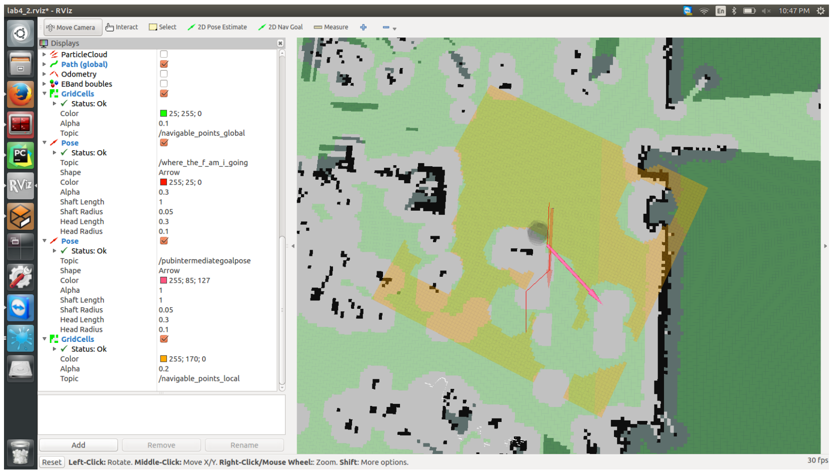
RBE 3002 Lab 4

Keshuai Xu, William Sullivan, Richard Eberheim

1. Rviz screenshot



ParticleCloud	☐
Path (global)	✓
Odometry	☐
EBand boubles	☐
GridCells	✓
✓ Status: Ok	
Color	🟩 25; 255; 0
Alpha	0.1
Topic	/navigable_points_global
Pose	✓
✓ Status: Ok	
Topic	/where_the_f_am_i_going
Shape	Arrow
Color	🟥 255; 25; 0
Alpha	0.3
Shaft Length	1
Shaft Radius	0.05
Head Length	0.3
Head Radius	0.1
Pose	✓
✓ Status: Ok	
Topic	/pubintermediategoalpose
Shape	Arrow
Color	🟥 255; 85; 127
Alpha	1
Shaft Length	1
Shaft Radius	0.05
Head Length	0.3
Head Radius	0.1
GridCells	✓
✓ Status: Ok	
Color	🟧 255; 170; 0
Alpha	0.2
Topic	/navigable_points_local

The key for the map shown above is as follows:

- The green cells are global navigable points.
- The orange cells are local navigable points.
- The red line is the global path planning.
- The blue line (not shown because no path was found) is the local path planning.

2. Occupancy grid size

The occupancy grid size was not modified because our a-star algorithm is efficient enough to handle the original number of cells.

The global and local costmap resolution is 0.05 m.

3. Updated heuristic

The new h-cost is the scaled-down weighed average of the Manhattan distance and the cost from the cost map.

```python
# fr and to are tuples (x,y)
def gethcost(fr, to, get_cost_map_cost=None):
    if get_cost_map_cost is not None:
        # magic coefficients. hopefully the added cost (0 to 10) does not exceed the actual
        return gethcost(fr, to) * 0.5 + get_cost_map_cost(to) * 0.1
    else:
        # manhattan distance
        return abs(to[0] - fr[0]) + abs(to[1] - fr[1])
```

4. Reading the cost map

In class `CostmapThing`, two callback functions were written to handle the cost map update. We subscribed to the `~costmap` and `~costmap_updates` of both local and global cost map to maintain updated copies of the cost maps. The cost maps were then binarized with a set threshold to determine the navigable area when a-star path planning is called.

5. A-star path planning

The path planning is done in array index unit in the corresponding map's frame. The array indicies were converted to the coordinate in the the map's frame when generating the waypoints.

The waypoints were transformed into `/map` frame before feeding into the move base function.

6. Path planning logic

The program does an initial planning in global map, then plan in local repetitively to go to the global waypoints until reaches the goal. If no path found in local, the program replan the path in global.

The local path planning handles dynamic obstacles.

- In a while loop, search in the global map with a-star. If there is a path, extract the first waypoint.
    - If it is the global goal, mark done and exit the loop after finishing rest of the code.
    - try to find a path in local map from the current pose to first waypoint in global map (local goal), with maximum distance of 1m.
        - If path is found, execute the path.
            - If next waypoint is local goal, navigate to it and replan in global.
        - If no path is found, plan in global again. If still no path found, give up.

- If no path found initially in global map, give up.

7. Move the base

We created a version of the `navToPose()` function that does not turn to the goal orientation when reaches the goal to use in the intermediate steps in local path following to save time. The robot only turn to goal orientation when it reaches the global goal.