1) Add ‘random\_walk’ and ‘a3\_help’ packages to your catkin\_ws

2) Compile these packages with ‘catkin\_make’

3) In a terminal, run ‘roscore’

4) In a terminal start stage simulation with

rosrun stage\_ros stageros $(rospack find a3\_help)/worlds/uoa\_robotics\_lab.world

5) In a terminal start component which will create local\_map

rosrun local\_map local\_map /local\_map/scan:=/base\_scan\_1 \_map\_width:=300 \_map\_height:=300 \_map\_resolution=0.1

OGMAP creator makes the center of this map coinciding with robots current position, so i am using 300x300 pixel map to avoid overflow the boundaries of local map, when robot moves towards max min position. So actual map has a size of “6.270x15.700” (see ./uoa\_robotics\_lab.world) . With 0.1m resolution it has 62-63/ 157 pixel at all. Its easy to cover this map with 300x300pixel local\_map

6) In a terminal start component which will create image from OGMAP

rosrun a3\_help map\_to\_image\_node /map:=/local\_map/local\_map /pose:=/odom

7) In two different terminals start two rqt\_image\_viewer with,

rosrun rqt\_image\_viewer rqt\_image\_viewer

These viewers will be used to monitor first\_query and second\_query of PRM algorithm. If you want to visualize current OGMAP also, open a third rqt\_image\_viewer and subscribe to ‘/map\_image/full’

8) In a terminal, start random\_walk component to discovery configuration space

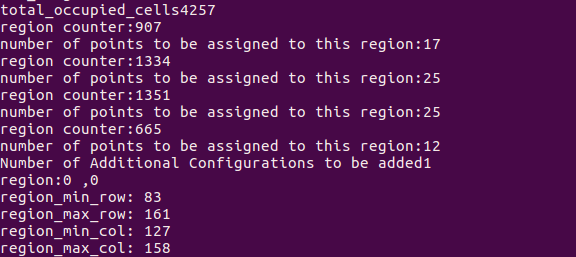
rosrun random\_walk random\_walk

This will start a random walk in map and discovered percentage of the map (percentage of free and occupied cells to total number of cells) . See line 130 of ‘gazebo\_retrieve.cpp’, when discovered percentage is higher than %80, random\_walk stops and this component request a random goal\_state from ‘sample’ component. If this random goal\_state is not on free configuration space ‘sample’ component doesnt respond ack to the request and ‘random\_walk’ component makes another RequestGoal on following imagecallback(until it finds a suitabl goal state). When it finds a suitable goal state(i.e. ack respond from RequestGoal service) it publishes this goal state with ‘/chatter’ topic.

9) In a terminal, start ‘sample’ component to execute PRM algorithm

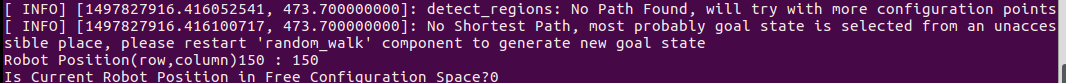
rosrun a3\_help sample

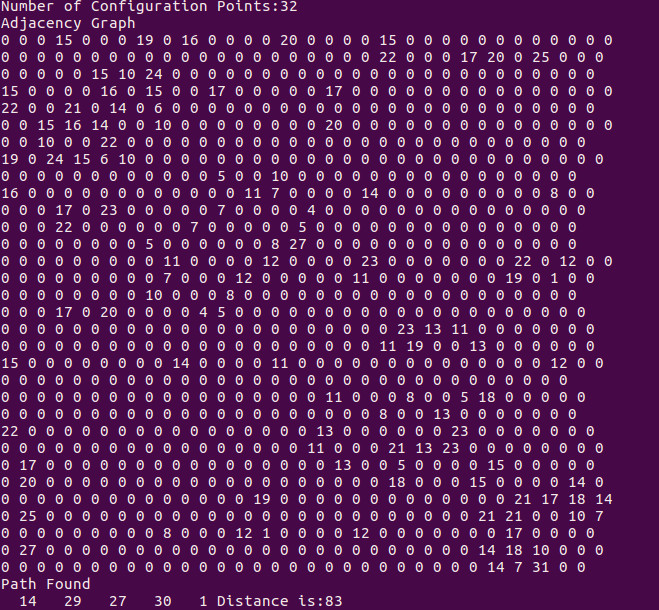
This component, wait for a ‘/chatter’ message from ‘random\_walk’ component. When it gets a suitable goal state, it divides current map into different regions. See ‘detect\_regions’ function in ‘sample.cpp’. This function partitions map into multiple regions (with ‘number\_of\_regions\_in\_col’ and ‘number\_of\_regions\_in\_row’ arguments; i use 2 but you can try different number of regions.) Then it counts for occupied and unknown pixels in each region. And it tries to assign more configuration point where occupied and unknown pixels have more density with a weighting algorithm. With this approach its easier to find paths within narrow passages and unknown parts of configuration space. After this process it assigns given number of configuration points at each region. For this purpose it randomly selects configuration points within these regions and checks whether that point is in free configuration space or not. If its not in free configuration space then process tries to assign another random point in that region(until it assigns given number of points at each region successfully)



After this process,it starts to execute PRM algorithm. See ‘add\_new\_goal\_and\_calculate\_shortest\_path’ function in ‘sample.cpp’ . If it cannot find a shortest path between start state(i.e. robots current position) and goal state, it adds more points to current roadmap and searches again. If it cannot find a shortest path between these points even if with 100 configuration points then process terminates because most probably the current position of robot is in occupied pixel, or selected goal point is at an unaccesible place. Process throws these kind of messages. If current position of robot is at a occupied pixel, its not possible to create a path from this point to any goal state, since there will be no path from an occupied cell, and you need to restart stage, local\_map and random\_walk. If selected goal state is at an unaccessible place, all you need is to restart random\_walk component to assign new goal state. This is the output messages for these kind of situations.

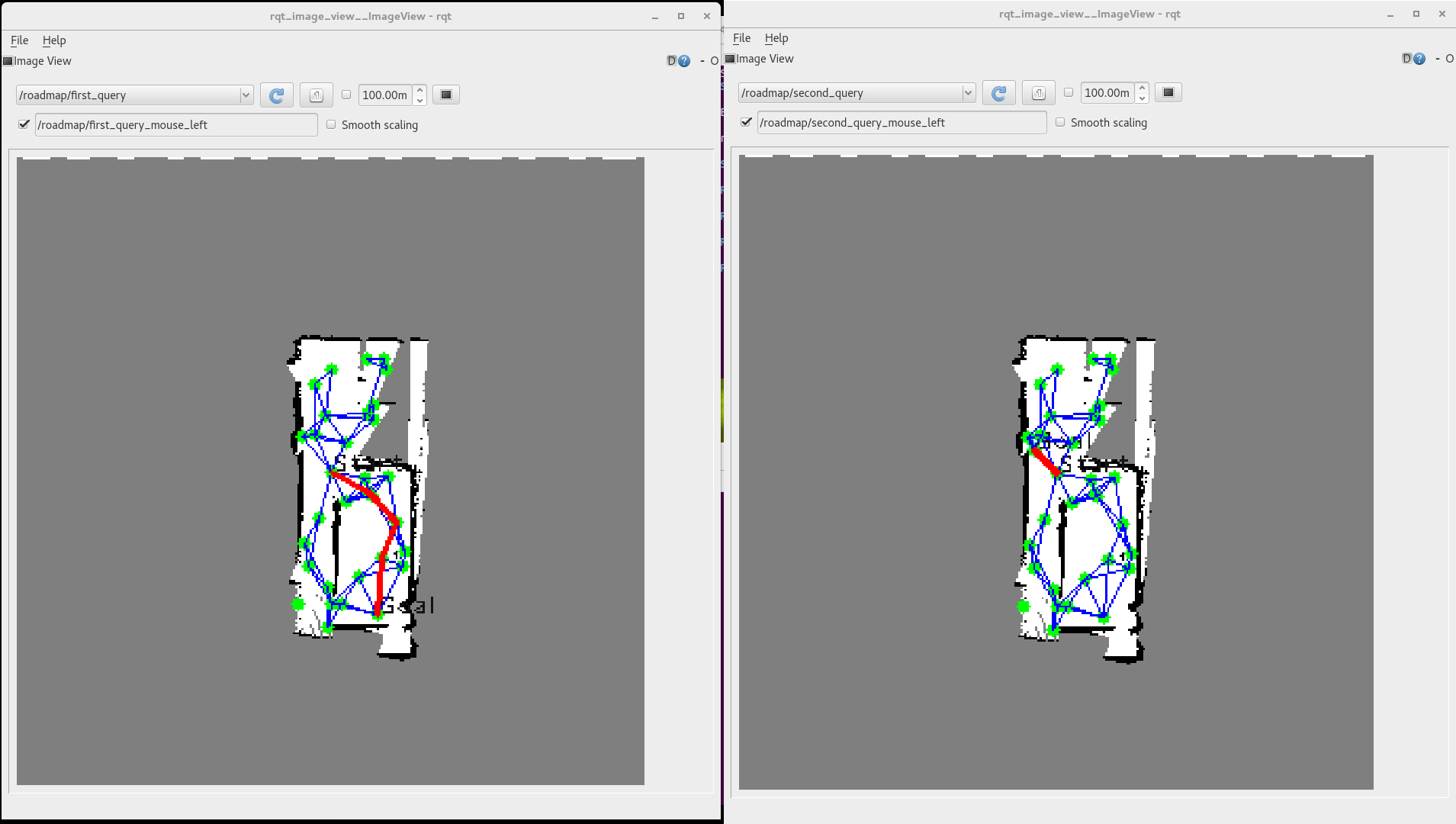


If everything is ok, and a shortest path is found between start and goal states, then ‘sample’ component publishes the resultant map with ‘roadmap/first\_query’ . You can visualize it with rqt\_image\_viewer (Step 7). Here is an adjacency graph to be used in shortest path algorithm



Then, ‘sample’ component randomly selects another goal state and add it to current PRM to demonstrate that constructed map is reusable for multiple queries. If process can find a shortest path from start state to this new goal state it published the resultant map with ‘roadmap/second\_query’ . You can visualize it with rqt\_image\_viewer (Step 7). If it cannot find a path between start state and this new goal state, by using current roadmap, there is nothing to do, you need restart random\_walk for new trials.

Here is a sample image to show the outputs of both queries.



Since images have only 300x300 pixels(because of 0.1m resolution stated in support packages) its really hard to visualize lines between configuration points, lines on shortest path, circles on configuration points etc. I tried to put green circle on configuration points, blue lines for roadmap, red lines for shortest path. I also tried to add ‘Goal’ and ‘Start’ texts on goal and start states but its a little bit hard to see them since opencv doesnt have so much flexibility with the parameters of ‘cv::putText’ function

Note: To check a point whether if its in free configuration space or not, its needed to augment this point with Minkowski sum. Since we have a circular robot, with a 20cm diameter and map resolution of 10cm (0.1m) , then we need to augment a point with upper, lower, left, right neighbors to be sure there is no collision even when robot is on these neighbors. This is how i implement Minkowski sums. Similary, for line segment checks, i created line segment between start a end points and sweep all points on this line segment and check each of them free or not.See ‘isConfigurationFree’ and ‘check\_line\_segment’ functions in ‘sample.cpp’

For automatic documentation, you need to execute

rosdoc\_lite src/a3\_help

rosdoc\_lite src/random\_walk

It will generate doc in ~/catkin\_ws/doc.I write to comments suitable for auto documentation with rosdoc\_lite. But dont forget to save /doc folder before executing rosdoc\_lite for other package, since rosdoc\_lite will overwrite that folder for different packages.

I also write unit tests for image2global and global2image function as suggested in support documents. It randomly tests these 2 functions with 5000000 iterations. See ‘src/a3\_help/test’ folder. To compile tests,

Compiling the test

*catkin\_make tests*

Running the test

*./devel/libs/a3\_help/a3\_help-test*