

Contents

Lab 4 – Navigation.....	2
Global Planner.....	2
Local Planner.....	2
Cost Maps	3
Understanding the move_base Package.....	3
The MacBot Navigation Setup	4
Visualizing Path Planning in RViz.....	9
Next Steps	9

Lab 4 – Navigation

This lab will demonstrate path planning using the **amcl** and **move_base** nodes. The primary function of the **move_base** node is to traverse a robot from a current location to a goal location. This goal location will be set using RViz's **2D Nav Goal** tool.

Global Planner

In order to navigate a robot from point A to point B, there are different strategies that can be used to calculate how to traverse that path. These are called path planning algorithms. Some popular examples are Dijkstra, A*, D*, and RRT. Applications like Google Maps use similar algorithms in order to optimize path planning and return the shortest route possible.

The **move_base** node has three global planners:

1. **carrot_planner** – Synonymous with 'putting a carrot in front of the robot', this strategy draws a direct path to the goal and ignores obstacles.
2. **Navfn** – Makes use of the Dijkstra's algorithm to find a global path with a minimal cost.
3. **global_planner** – A more flexible version of navfn that implements a variety of navigation algorithms.

All of the SLAM algorithms could comprise an entire course of content. Although this lab will not cover these algorithms, the reader is advised to briefly learn about Dijkstra and A* specifically.

<http://ros-developer.com/2019/05/20/dijkstras-algorithm-with-c/>

<https://realitybytes.blog/2018/08/17/graph-based-path-planning-a/>

This lab will primarily focus on the **global_planner** global planning package. The files are preconfigured so that they will use Dijkstra's algorithm for global path planning.

Local Planner

Now that a global planner has been selected, one must strategize how to handle local obstacles. An example of this is a person walking in front of the robot or an obstacle being placed in front of it. A global planner would not be aware of the local obstruction and attempt to push through it in order to maintain the set trajectory. Alternately, a local planner working in conjunction with the global planner would be able to replan the blocked segment to navigate around the local obstacle.

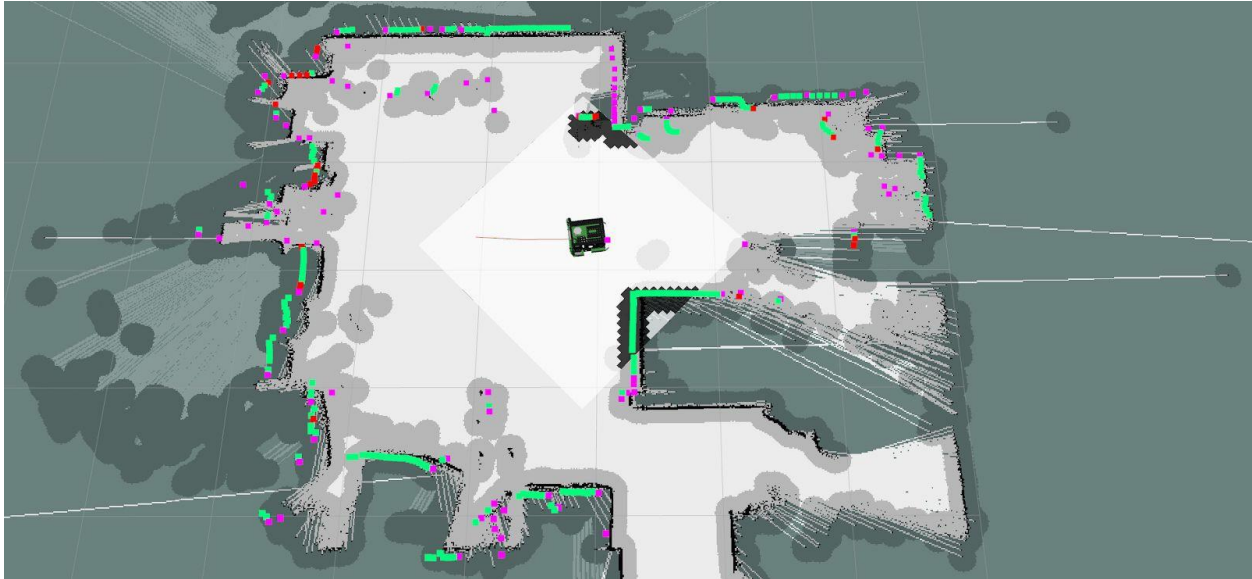
The **move_base** node has three local planners:

1. **dwa_local_planner** – Calculates the robot trajectory using velocity and angle. Evaluates which trajectory will keep it the most in-line with the global path.
2. **eband_local_planner** – Uses heuristics to model the change in trajectory as an elastic band.
3. **teb_local_planner** – 'timed elastic band' considers multiple trajectories in order to calculate the best solution.

This lab will proceed with the **dwa_local_planner**, which uses the **Dynamic Window Approach (DWA)** algorithm.

Cost Maps

Cost maps represent the cost (difficulty) of navigating to areas of the map. Just as there are global and local planners, there are also global and local cost maps. Cost maps represent areas of tolerance a robot is able to smoothly navigate, and they are used to guide and route a safe while efficient path.



In the image above, the local cost map is within the white diamond / square surrounding the MacBot. The global cost map is overlaid on top of the map.

The global planner will use the global cost map to generate the global (long-term) plan. The local planner will use the local cost map to generate their local (short-term) plan. The local cost map will use local sensor data to build their obstacle (cost) map.

Understanding the move_base Package

Move_base is a ROS package that coordinates between global planner and the local planner. This is implemented as a ROS Action server, with structured goal, feedback, and result messages as well as the ability to pre-empt goals. If one looks within the `~/catkin_ws/src/macbot/macbot_navigation/config` directory, they will see a variety of parameter files. These files are for tuning the cost map and planners for both global and local maps. Knowing about the existence of these files becomes critical when tuning these planners for the robot's particular environment and on-board sensors specifications.

```

controller_frequency: 5.0
planner_frequency: 10.0
recovery_behaviour_enabled: true

GlobalPlanner:
  allow_unknown: true
  use_dijkstra: true
  visualize_potential: false
  use_quadratic: true
  use_grid_path: false
  old_navfn_behavior: false # Exactly mirror behavior of navfn, use defaults for
  cost_factor: 0.55
  neutral_cost: 66 #66

NavfnROS:
  allow_unknown: true # Specifies whether or not to allow navfn to create plans that
  default_tolerance: 0.1 # A tolerance on the goal point for the planner.

TrajectoryPlannerROS:
  # Robot Configuration Parameters
  acc_lim_x: 2.5
  acc_lim_theta: 3.2

  max_vel_x: 1.0
  min_vel_x: 0.0

  max_vel_theta: 1.0
  min_vel_theta: -1.0
  min_in_place_vel_theta: 0.2

  holonomic_robot: false
  escape_vel: -0.1

  # Goal Tolerance Parameters
  yaw_goal_tolerance: 0.1
  xy_goal_tolerance: 0.2
  latch_xy_goal_tolerance: false

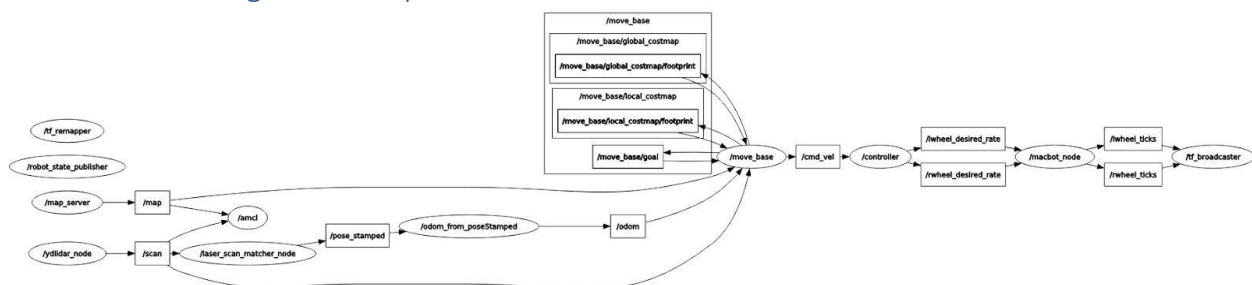
  # Forward Simulation Parameters
  sim_time: 5.0
  sim_granularity: 0.02
  angular_sim_granularity: 0.02
  vx_samples: 6
  vtheta_samples: 20
  # controller_frequency: 20.0

```

It is advised to read through the following guides before attempting to modify planning parameters. Changing these values could impact the expected movement of the robot. Ensure that the programmer has ample floor space to test.

<http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide>
<https://arxiv.org/pdf/1706.09068.pdf>

The MacBot Navigation Setup



Running **roslaunch rqt_graph rqt_graph** after the individual MacBot nodes have been started will display a high-level overview of the communication network between individual ROS nodes.

Since LiDAR is not a perfectly reliable source of odometry (its accuracy is affected by the material, colour, and shape of landmarks), getting autonomous path planning to work on the MacBot is challenging. In this case, a ROS node must be written to convert the laser scans from the LiDAR into usable odometry data.

laser_scan_matcher → geometry_msgs/PoseStamped → nav_msgs/Odometry

Create this node in the macbot_sensors package and name it pose_odom.py.

```
#!/usr/bin/env python

import rospy
import tf
import tf_conversions
import tf2_ros
import sys
import signal
from math import cos
from nav_msgs.msg import Odometry
from time import sleep
from geometry_msgs.msg import PoseStamped, Quaternion, Point, Pose, Quaternion, Twist, Vector3
```

```
"""
PoseStamped data structure:
```

```
header:
  seq: 1234
  stamp:
    secs: 123
    nsecs: 12345
  frame_id: "odom"
pose:
  position:
    x: 0.0
    y: 0.0
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 0.0
"""
```

```
class poseOdom():
    def __init__(self):
        self.pubOdom = rospy.Publisher("odom", Odometry, queue_size = 1)

        #self.r = rospy.Rate(5)

        self.x_prev = 0.01
        self.y_prev = 0.01
        self.z_prev = 0.01

        self.quat = [0.01, 0.01, 0.01, 0.01]
        self.quat_prev = [0.01, 0.01, 0.01, 0.01]

        self.time_now = rospy.Time.now()
        self.time_prev = rospy.Time.now()

        self.odom = Odometry()
        self.odom.header.frame_id = "odom"
        self.odom.child_frame_id = "base_link"

    def listener(self):
        self.subPoseStamped = rospy.Subscriber("pose_stamped", PoseStamped, self.callback)
        rospy.spin()
```

```

def callback(self, data):
    self.time_now = rospy.Time.now()
    self.odom.header.stamp = rospy.Time.now()

    self.delta_t = (self.time_now - self.time_prev).to_sec()

    self.quat[0] = (data.pose.orientation.x + self.quat_prev[0])/self.delta_t
    self.quat[1] = (data.pose.orientation.y + self.quat_prev[1])/self.delta_t
    self.quat[2] = (data.pose.orientation.z + self.quat_prev[2])/self.delta_t
    self.quat[3] = (data.pose.orientation.w + self.quat_prev[3])/self.delta_t

    self.quat_prev[0] = data.pose.orientation.x
    self.quat_prev[1] = data.pose.orientation.y
    self.quat_prev[2] = data.pose.orientation.z
    self.quat_prev[3] = data.pose.orientation.w

    self.rpy = tf.transformations.euler_from_quaternion(self.quat)
    self.vx = ((data.pose.position.x + self.x_prev)/self.delta_t)/cos(self.rpy[2])
    self.x_prev = data.pose.position.x

    self.odom.pose.pose = Pose(Point(data.pose.position.x, data.pose.position.y, data.pose.position.z),
        Quaternion(data.pose.orientation.x, data.pose.orientation.y, data.pose.orientation.z, data.pose.orientation.w))
    self.odom.twist.twist = Twist(Vector3(self.vx, 0, 0), Vector3(0, 0, self.rpy[2]))

    # Publish message
    self.pubOdom.publish(self.odom)
    #self.r.sleep()
    self.time_prev = self.time_now

```

```

def handler(signal, frame):
    # Program Cleanup
    print(' SIGINT CTRL+C received. Exiting.')
    sys.exit(0)

if __name__ == "__main__":
    signal.signal(signal.SIGINT, handler)

    rospy.init_node('odom_from_poseStamped')
    pose_odom_obj = poseOdom()

    while 1:
        pose_odom_obj.listener()

```

Odometry data is crucial for the **move_base** to work. The **move_base** package will later be sending **/cmd_vel** messages to the differential drive controller, which will send the desired wheel spin rates to the **macbot_node** from the previous lab.

Before getting started, there is a new flag that must be included when an attempt is made to SSH into the MacBot.

Old:

ssh [macbot@192.168.X.X](#)

New:

ssh -X [macbot@192.168.X.X](#)

This will enable X11 forwarding. An X-server is the display server that Ubuntu uses to display windows on a monitor. Forwarding this information allows RViz to be run graphically on a remote PC, which is an important tool for working with 2D pose data and navigation goals.

To get started, please launch the differential drive node:

roslaunch macbot_physical diff_drive.launch

```
jetsonnano@jetsonnano-desktop: ~  
jetsonnano@jetsonnano-desktop: ~ 105x3  
jetsonnano@jetsonnano-desktop:~$ roslaunch macbot_physical diff_drive.launch
```

Next, the LiDAR sensor must be launched. Ensure to pass **true** for the **pub_tf** argument. Doing this will let the `laser_scan_matcher` node publish the TF link between `odom` and `base_link`.

Roslaunch `macbot_sensors lidar.launch pub_tf=true`

```
jetsonnano@jetsonnano-desktop: ~  
jetsonnano@jetsonnano-desktop: ~ 92x3  
jetsonnano@jetsonnano-desktop:~$ roslaunch macbot_sensors lidar.launch pub_tf=true
```

Next, please launch **`pose_odom.py`**. This node will be converting messages from `laser_scan_matcher` to odometry, as mentioned earlier. This is not the most accurate method of generating odometry data.

roslaunch `macbot_navigation pose_odom.py`

```
jetsonnano@jetsonnano-desktop: ~  
jetsonnano@jetsonnano-desktop: ~ 92x3  
jetsonnano@jetsonnano-desktop:~$ roslaunch macbot_navigation pose_odom.py
```

Lastly, the nodes **`move_base`** and **`amcl`** must be launched for path planning.

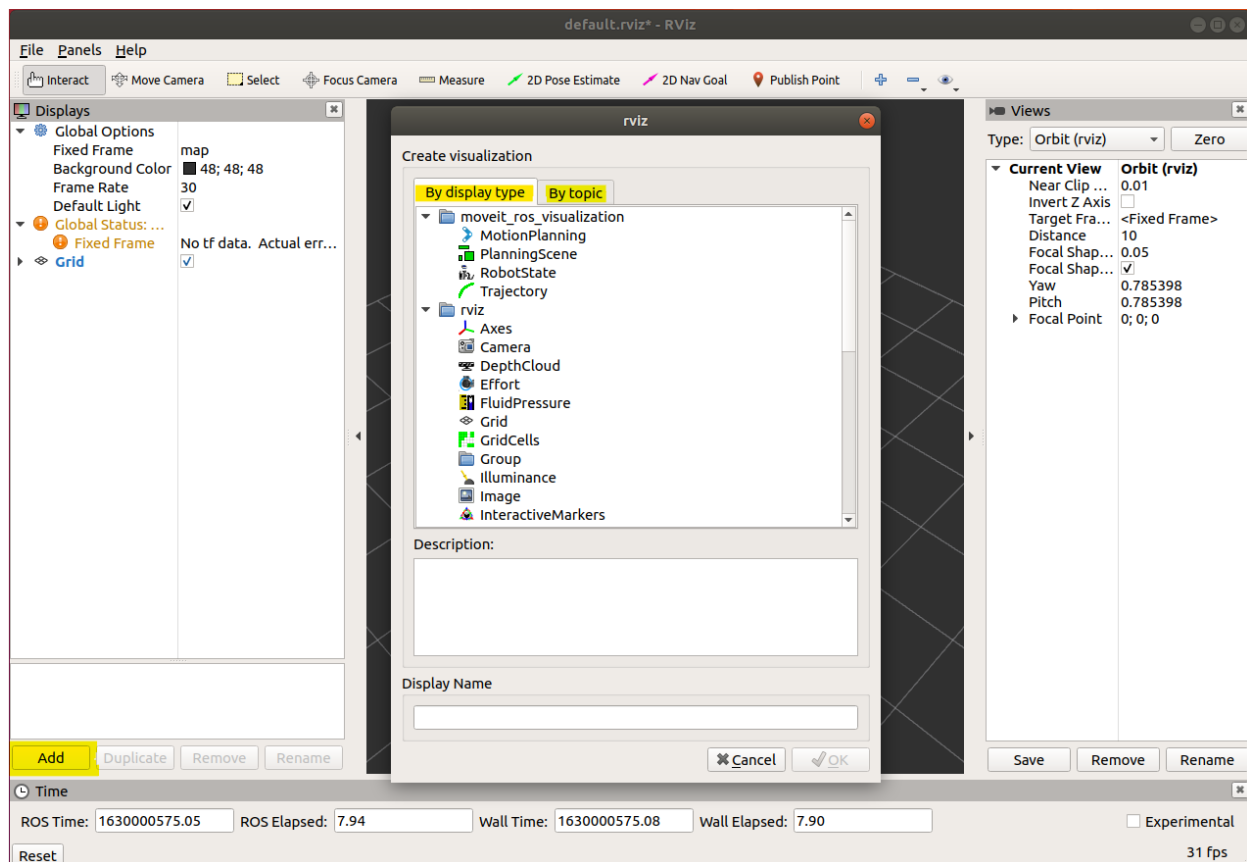
roslaunch `macbot_navigation amcl_macbot.launch`

```
jetsonnano@jetsonnano-desktop: ~  
jetsonnano@jetsonnano-desktop: ~ 92x3  
jetsonnano@jetsonnano-desktop:~$ roslaunch macbot_navigation amcl_macbot.launch
```

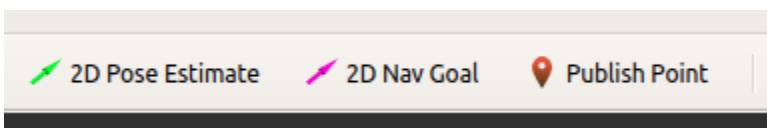
Looking back at the VMWare Ubuntu desktop, please launch RViz. Select **Add** in the bottom-left area of the window. Navigate through the menu pop-up to select the desired visualization formats.

Take some time to try out and experiment with the various display types. They can be further configured in the left **Displays** column of RViz.

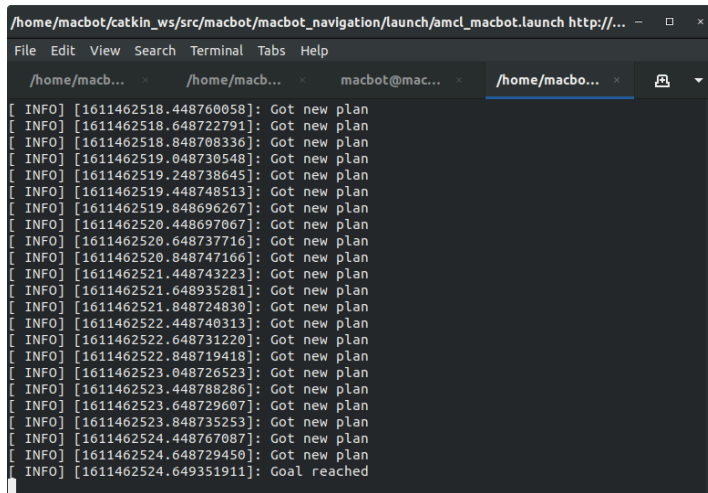
Feel free to include information regarding the global planners, local planners, and cost maps.



Depending on the MacBot's position within the map, publish a 2D pose update from the top toolbar so that the amcl is able to localize the robot within the generated map that was previously made.



Visualizing Path Planning in RViz



```
/home/macbot/catkin_ws/src/macbot/macbot_navigation/launch/amcl_macbot.launch http://... - □ ×
File Edit View Search Terminal Tabs Help

/home/macb... /home/macb... macbot@mac... /home/macbo...

[ INFO] [1611462518.448760058]: Got new plan
[ INFO] [1611462518.648722791]: Got new plan
[ INFO] [1611462518.848708336]: Got new plan
[ INFO] [1611462519.048730548]: Got new plan
[ INFO] [1611462519.248738645]: Got new plan
[ INFO] [1611462519.448748513]: Got new plan
[ INFO] [1611462519.648758667]: Got new plan
[ INFO] [1611462520.048697067]: Got new plan
[ INFO] [1611462520.248737716]: Got new plan
[ INFO] [1611462520.448747166]: Got new plan
[ INFO] [1611462521.048743223]: Got new plan
[ INFO] [1611462521.248935281]: Got new plan
[ INFO] [1611462521.448724830]: Got new plan
[ INFO] [1611462522.048740313]: Got new plan
[ INFO] [1611462522.248731220]: Got new plan
[ INFO] [1611462522.448719418]: Got new plan
[ INFO] [1611462523.048726523]: Got new plan
[ INFO] [1611462523.248788286]: Got new plan
[ INFO] [1611462523.448729607]: Got new plan
[ INFO] [1611462523.648735253]: Got new plan
[ INFO] [1611462524.048767087]: Got new plan
[ INFO] [1611462524.248729450]: Got new plan
[ INFO] [1611462524.449351911]: Goal reached
```

After publishing the navigation goal, a global plan will be generated within RViz. Soon after, the local plan will be generated as the robot begins detecting obstacles.

Next Steps

The next step is to attempt to optimize this navigation process. It is currently configured to navigate in a straight line. Extend this exercise by trying to optimize it so that it may navigate turns and smaller spaces more efficiently.