SIT310 – Task 8
Planning

Overview

This task focuses on using ROS planners with our Navigation stack. We will be learning how to properly setup the Navigation stack to use a planner, and how to interact with the planner.

Task requirements

a. Ensure you have already completed tasks 1-7 and assessments 1 - 3.
b. To complete this task, you will need to use ROS on the raspberry Pi as we will be using your physical robot
c. If you didn't attend the lecture this week, then either watch the lecture online and/or review the PowerPoint slides on the unit site.
d. Read the task instructions
e. Complete the activities in the task guide
f. Review your progress with your lab tutor or cloud tutor.

Task guide

So far, we have been either controlling the robot using turtlesim, manually using a keyboard program or automatically responding to scary things or walls. This lab will give you a taster of how to do proper planning by using the navigation stack and an adaptive Monte Carlo Localization (amcl) planner.

1. Planning

   1.1. As we have seen in the lectures, planning, and particularly planning in navigation, is an extremely important topic in robotics and robotic application development. There are many planners available, with a large number free.

   1.2. Have a look at the free book "Planning Algorithms" here, to see the range of planning options: http://planning.cs.uiuc.edu/

   1.3. For ROS, there are two main options:

      1.3.1. The Moveit! Toolkit, available here: https://moveit.ros.org/

      1.3.2. Planning in the Navigation stack, available here: http://wiki.ros.org/navigation

   1.4. We will be using the planning functionality in the navigation stack. There are multiple planners available:

      1.4.1. The *Carrot Planner* basically dangles a carrot in front of the robot and leads it towards the goal: http://wiki.ros.org/carrot_planner?distro=kinetic

      1.4.2. The *base local Planner* which implements the Trajectory Rollout and Dynamic Window approaches. It uses costmaps and will produce velocity commands to send a robot to a goal.

      1.4.3. The *amcl Planner*, which is an adaptive Monte Carlo Localization planner. See here: http://wiki.ros.org/amcl?distro=kinetic

      1.4.4. We will be using the amcl planner, mainly because it can be ran as a ROS node rather than programmed, and has good documentation.

2. Completing the Navigation Stack

   2.1. To use planners, we need to complete our Navigation stack. We'll do this from scratch so we have everything we need in our lab8 folder.

   2.2. Install new packages we need:

```
$ sudo apt-get install ros-kinetic-move-base ros-kinetic-map-server ros-kinet
ic-amcl
```

2.3. Create a new package as follows.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg lab8 move_base rospy
Check everything is fine.
$ cd ~/catkin_ws
$ catkin_make
Open a new terminal.
$ roscd lab8
$ mkdir scripts
$ cd scripts
```

2.4. Copy some files into the

```
$ cp ~/catkin_ws/src/lab6/scripts/zumo_tf_sensor.py .
$ cp ~/catkin_ws/src/lab5/scripts/zumo_controller.py .
$ cp /opt/ros/kinetic/share/turtle_tf2/rviz/turtle_rviz.rviz ../zumo.rviz
```

2.5. Edit the zumo.rviz as follows, and change “Plane Cell Count” to be 6 on line 48.

```
$ cd ..
$ gedit zumo.rviz
```

2.6. We need a modified zumo_tf_broadcaster.py which will periodically publish the robots pose in the /odom topic. We are also changing the topics generally to be more standard. Create the file with the following text. Note that this is going back to an older version of zumo_tf_broadcaster, rather than the compass version of lab 7.

```
$ roscd lab8
$ cd scripts
$ gedit zumo_tf_broadcaster.py
$ chmod +x zumo_tf_broadcaster.py
```

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

import tf_conversions
import tf2_ros
import geometry_msgs.msg
from geometry_msgs.msg import Twist, Pose, Point, Quaternion, Vector3
from nav_msgs.msg import Odometry

#for calculating position
```

```
import math
from math import sin, cos, pi

x=0
y=0
th=0.0

odom_pub = rospy.Publisher("odom", Odometry, queue_size=50)

def handle_zumo_pose(vel_msg):
    global x
    global y
    global th

        #for rotation, 5 degrees is 0.0872665 rads
    if(vel_msg.linear.y==1.0):  th+= 0.0872665
    elif(vel_msg.linear.y==-1.0): th-=0.0872665
    elif(vel_msg.linear.x==1.0): #this is a movement
        vx = 0.1 #velocity of x, how much we move
        # compute odometry
        delta_x = vx * cos(th)
        delta_y = vx * sin(th)
        delta_th = 0  #no change in angle

        #add the changes to the values
        x += delta_x
        y += delta_y
        th += delta_th

    elif(vel_msg.linear.x==-1.0): #this is a movement
        vx = 0.1 #velocity of x, how much we move
        # compute odometry
        delta_x = vx * cos(th)
        delta_y = vx * sin(th)
        delta_th = 0  #no change in angle

        #add the changes to the values
        x -= delta_x
        y -= delta_y
        th -= delta_th

def publish_transform():
    br = tf2_ros.TransformBroadcaster()

    t = geometry_msgs.msg.TransformStamped()

    #    y+=vel_msg.linear.y
    t.header.stamp = rospy.Time.now()
    t.header.frame_id = "odom"
    t.child_frame_id = "base_link"

    #set x,y,z
    t.transform.translation.x = x
    t.transform.translation.y = y
    t.transform.translation.z = 0.0

    #set rotation
    q = tf_conversions.transformations.quaternion_from_euler(0, 0, th)
    t.transform.rotation.x = q[0]
    t.transform.rotation.y = q[1]
    t.transform.rotation.z = q[2]
    t.transform.rotation.w = q[3]

    br.sendTransform(t)


def loop():
    # next, we'll publish the odometry message over ROS
    odom = Odometry()

    #set rotation
    q = tf_conversions.transformations.quaternion_from_euler(0, 0, th)

    odom.header.frame_id = "odom"

    # set the position
    odom.pose.pose = Pose(Point(x, y, 0.), Quaternion(*q))

    # set the velocity
    odom.child_frame_id = "base_link"
    odom.twist.twist = Twist(Vector3(0, 0, 0), Vector3(0, 0, 0))

    current_time = rospy.Time.now()
    odom.header.stamp = current_time

    # publish the message
    odom_pub.publish(odom)
    publish_transform()

if __name__ == '__main__':
    rospy.init_node('zumo_tf_broadcaster')
    rospy.Subscriber('/zumo/cmd_vel',
                     Twist,
                     handle_zumo_pose)

    while not rospy.core.is_shutdown():
        loop()
        rospy.rostime.wallsleep(0.01)
        #rospy.spin()
```

2.7. To use the navigation stack, we need a map server. A map server is normally configured with an image which contains a heat map – the darker the colours, the higher the objects. Grab an image from online and create a new file as follows

```
$ roscd lab8

$ wget https://raw.githubusercontent.com/pirobot/ros-by-example/master/rbx_vol_1/rbx1_nav/maps/blank_map.pgm

$ gedit mymap.yaml
```

```
image: blank_map.pgm
resolution: 0.01
origin: [0, 0, 0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

2.8. As we are now using a lot of nodes, we are going to use ROS launch files, which are like batch files. Create a file zumo_nav.launch, as follows, with the following text.

```
$ roscd lab8

$ gedit zumo_nav.launch
```

```
<launch>
<!-- Run the map server -->
     <node name="map_server" pkg="map_server" type="map_server" args="$(find
lab8)/mymap.yaml"/>
   <node pkg="rosserial_python" type="serial_node.py" name="rosserial"
output="screen">
    <param name="port" value="/dev/ttyACM0"/>
    </node>
   <node pkg="lab8" type="zumo_tf_broadcaster.py" name="zumo_pose"
output="screen"></node>
   <node pkg="lab8" type="zumo_tf_sensor.py" name="zumo_sensor"
output="screen"></node>
</launch>
```

2.9. Test as follows, each line in a different window or tab. Note that you will be using the latest Arduino program from Lab5-4.2. In rviz, change the fixed frame from world to odom.

```
$ roslaunch lab8 zumo_nav.launch

$ rosrun rviz rviz -d zumo.rviz

$ rosrun lab8 zumo_controller.py
```

You should be presented with a 6*6 grid, with a robot that you can control as usual with the zumo_controller.py program. If not, debug the problem! You will need these 3 commands running for the remainder of the task – you might need to restart them if you have problems.

3. Using the amcl planner.

   3.1. We need to configure the amci planner to use it. There is a lot of information about the configuration options for the planner – see here http://wiki.ros.org/amcl?distro=melodic

   3.2. Create the following files in the lab8 folder using gedit:

costmap_common_params.yaml

```
robot_radius: 0.5

observation_sources: point_cloud_sensor
point_cloud_sensor: {sensor_frame: odom, data_type: PointCloud, topic:
/zumo/objectcloud, marking: true, clearing: true}
```

local_costmap_params.yaml

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 0.5
  publish_frequency: 0.1
  static_map: false
  rolling_window: true
  width: 10.0
  height: 10.0
  resolution: 0.05
  transform_tolerance: 10
```

global_costmap_params.yaml

```
global_costmap:
   global_frame: odom
   robot_base_frame: base_link
   update_frequency: 5.0
   publish_frequency: 5.0
   resolution: 0.05
   static_map: true
   rolling_window: false
   transform_tolerance: 10
   width: 10.0
   height: 10.0
   resolution: 0.05
```

base_local_planner_params.yaml

```
TrajectoryPlannerROS:
  max_vel_x: 0.45
  min_vel_x: 0.1
  max_vel_theta: 1.0
  min_in_place_vel_theta: 0.4

  acc_lim_theta: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5

  holonomic_robot: true
```

move_base.launch

```
<launch>
<master auto="start"/>

<!--- Run AMCL -->
    <include file="$(find amcl)/examples/amcl_diff.launch" />

          <param name="tf_prefix" type="string" value=""/>

<node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
          <rosparam file="$(find lab8)/costmap_common_params.yaml"
command="load" ns="global_costmap" />
          <rosparam file="$(find lab8)/costmap_common_params.yaml"
command="load" ns="local_costmap" />
          <rosparam file="$(find lab8)/local_costmap_params.yaml"
command="load" />
          <rosparam file="$(find lab8)/global_costmap_params.yaml"
command="load" />
          <rosparam file="$(find lab8)/base_local_planner_params.yaml"
command="load" />
</node>
</launch>
```

3.3. To run the planner, use the following.

```
$ roslaunch lab8 move_base.launch
```

4. Testing the planner

   4.1. There are two ways to test the planner. By setting goals in rviz, and by setting goals using code within a node.

   4.2. We need to see what is going on within ROS topics when we test the planner, so first, has a look at what topics now exist.

```
$ rostopic list
```

   4.3. There is a lot of topics, but we can monitor the topics we are interested in. Create the following in terminal windows. The first command are the goals set for the planner. The second command is the velocity output from the planner – this is the result of the planning process.

```
$ rostopic echo /move_base/goal
$ rostopic echo /cmd_vel
```

   4.4. In the menu bar of viz, click "2D Nav Goal". Then select and hold the mouse button near the robot and drag it towards the top left of the grid. This is positive x and y.

   4.5. You will see in the /move_base/goal echo that a goal has been set with x and y positions and a pose.

   4.6. You will also see that in /cmd_vel echo, a velocity is being requested. This should be a x (move forward or back) and an orientation – as we are using amcl_diff for differential robotics. This should be constantly requested. Try pressing enter a few times in the window to see it rapidly move.

   4.7. To illustrate that this is live and dynamic – move the robot using the keyboard controller which should be running already. Put the marker in front of the robot. As you move the robot around, you should see the x and orientation values change as the planner is constantly re-planning.

   4.8. The second way to test the planner is using a program. Create the python program test_goal.py as follows, with the following text.

```
$ roscd lab8
$ cd scripts
$ gedit test_goal.py
$ chmod +x test_goal.py
```

```
#!/usr/bin/env python
# license removed for brevity
```

```
import rospy

# Brings in the SimpleActionClient
import actionlib
# Brings in the .action file and messages used by the move base action
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

def movebase_client():
    # Create an action client called "move_base" with action definition file
"MoveBaseAction"
    client = actionlib.SimpleActionClient('move_base',MoveBaseAction)

    # Waits until the action server has started up and started listening for
goals.
    client.wait_for_server()

    # Creates a new goal with the MoveBaseGoal constructor
    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "odom"
    goal.target_pose.header.stamp = rospy.Time.now()
    # Move 0.5 meters forward along the x axis of the "map" coordinate frame
    goal.target_pose.pose.position.x = 2
    goal.target_pose.pose.position.y = 2
    # No rotation of the mobile base frame w.r.t. map frame
    goal.target_pose.pose.orientation.w = 1.0

    # Sends the goal to the action server.
    client.send_goal(goal)
    # Waits for the server to finish performing the action.
    wait = client.wait_for_result()
    # If the result doesn't arrive, assume the Server is not available
    if not wait:
        rospy.logerr("Action server not available!")
        rospy.signal_shutdown("Action server not available!")
    else:
        # Result of executing the action
        return client.get_result()

# If the python node is executed as main process (sourced directly)
if __name__ == '__main__':
    try:
        # Initializes a rospy node to let the SimpleActionClient publish and
subscribe
        rospy.init_node('test_goal_py')
        result = movebase_client()
        if result:
            rospy.loginfo("Goal execution done!")
    except rospy.ROSInterruptException:
        rospy.loginfo("Navigation test finished.")
```

4.9. If your robot is around 0,0. Then when you run the script as follows. You can move the robot around as before and /cmd_vel will be updated.

```
$ rosrun lab8 test_goal.py
```

4.10. So far the Zumo subscribes to /zumo/cmd_vel, and the planner issues commands on /cmd_vel. The Zumo expects either 1.0 or -1.0 values, whereas the planner can issue fractions. To get the planner working with the Zumo, you need to either adapt the Arduino program, or write a bridge between the two that will convert the values.

5. Assessment

This week's lab assessment is based on the output of this lab as follows. You should start this now. Depending on what you achieve, you should upload a short paragraph of explanation of what you have done, your code in a zip and a short video to the quiz "Lab Assessment 4" on the unit site. Upload your video using DeakinAir – a mobile phone video is fine. Put your zip on your onedrive, grab a link. And insert this link in the text box.

Leave the quiz questions for more advanced functionality blank. Don't worry, there is plenty of time to get great grades in the unit!

- P – Demonstrate this week's task working on with both rviz and program-based goal setting.
- C – Get the Zumo working with the commands sent to/cmd_vel by the planner. Either adapt the Arduino program to act on /cmd_vel messages directly or write a node that converts /cmd_vel to /zumo/cmd_vel.
- D – Demonstrate goal tracking working with the compass based zumo_tf_broadcaster.py – you will have to modify the program and you will need to let the compass settle down.
- HD – Demonstrate object avoidance working. This should be already supported – but will require debugging.

6. Additional resources

6.1. Learn about linux

- https://www.computerworld.com/article/2598082/linux/linux-linux-command-line-cheat-sheet.html
-

6.2. Learn about ROS.

- http://wiki.ros.org
- A Gentle Introduction to ROS: https://www.cse.sc.edu/~jokane/agitr/
- http://wiki.ros.org/kinetic/Installation/Ubuntu
- http://wiki.ros.org/ROS/Tutorials
- https://www.pololu.com/docs/0J63/4
- http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup
- http://wiki.ros.org/rosserial_arduino/Tutorials/Hello%20World