CS3027/5095 Robotics

Practical II

2016

Introduction

In this practical we'll go into more depth into ROS, learning about launching multiple nodes, setting parameters on-the-fly, and start playing around with our first robot simulation environment.

Tasks

1. Launching turtlesim in the previous practical required us to open three terminal windows and run three separate commands. Imagine the case where you have to fire up 20 or 30 nodes; doing so manually is clearly impractical. You can use the `roslaunch` command to launch multiple ROS nodes. The command executes a *launch* file from some package, and the format for using it is `roslaunch package_name file.launch`. Here's an example launch file.

```
<launch>
  <node
    pkg="turtlesim"
    type="turtlesim_node"
    name="turtlesim"
    respawn="true"
   />
   <node
     pkg="turtlesim"
     type="turtle_teleop_key"
     name="teleop_key"
     required="true"
     launch-prefix="xterm -e"
    />
</launch>
```

   As you can see, the launch file is an XML file, containing `node` elements. In turn, a `node` identifies the package and node program to run (via the

pkg and type attributes respectively), and assigns the node a name. The respawn attribute will cause the node to relaunch if it crashes, while the presence of the required attribute will cause all other nodes to be shut down if the required node terminates. Finally, the launch-prefix is used to tell roslaunch how to run the node. If you use xterm -e, it'll run in its own window (in this case, allowing keyboard input to be provided to the node).

Create a launch file named hello.launch in your hello workspace containing the above, and run it with roslaunch hello hello.launch. Try close the window in which you use the arrow keys to steer the turtle, and see what happens. Roslaunch has many other powerful features, we're going to ignore most of them, if you're interested in finding out more, please refer to Chapter 6 of https://cse.sc.edu/~jokane/agitr/agitr-letter.pdf. However, you should familiarise yourself with Section 6.4, which might be useful later on in the course.

2. It's often useful to be able to pass parameters to a ROS program without having to recompile it. ROS provides a parameter server for this purpose. Parameters are key-value pairs, and you can use the rosparam command to view them from the command line. Running rosparam list will list all parameters present, and rosparam get parameterName will display the value for parameter parameterName. You can also use rosparam set name value to set parameter name to value value. Look at the tutorial on accessing parameters (http://wiki.ros.org/rospy/Overview/ParameterServer), and write a small ROS program to set and get some parameters.

   Note that you can have a launch file set parameters using XML of the form

   ```
   <param name="param-name" value="param-value" />
   ```

3. Stage is a 2D environment simulator, which we'll be using for part of the course. Try run the command rosrun stage_ros stageros `rospack find stage_ros`/world/willow-erratic.world which uses the stage_ros package and runs a stageros node, passing it the willow-erratic.world world description[1]. You can drive the robot around from the keyboard by running rosrun turtlesim turtle_teleop_key turtle1/cmd_vel:=/cmd_vel. Note that the parameter turtle1/cmd_vel:=/cmd_vel remaps the turtle1/cmd_vel topic to cmd_vel.

4. Once you're bored with navigating the robot, write a program to have the it move around randomly in the world. Start with a program similar to the one you created in the last practical. You should find that pretty quickly, your robot will crash into obstacles. To overcome this, we need

[1] If you're using a virtual machine to run ROS, it'll be useful if you add the following to your .bashrc: export LIBGL_ALWAYS_SOFTWARE=1 when running stage

to detect these obstacles through the robot's sensors. Press 'D' within the stage window, and you'll see a visualisation of the Laser Scan sensor found on the robot, and what it detects.

(a) If you do a `rostopic list` you'll see several useful topics, including `/base_scan`. Run `rostopic info /base_scan`, and you'll see that it is of type `sensor_msgs/LaserScan`.

(b) Now do a `rostopic echo /base_scan|more` and you'll see a whole bunch of useful info. Refer to `http://docs.ros.org/kinetic/api/sensor_msgs/html/msg/LaserScan.html` for documentation. If you know the start and end angles and the angle increment, you can work out the angle at which each element of the `range` array occurs at, and can work out where obstacles are.

(c) Write a program to get this data from the `base_scan` topic, and warn the user when an obstacle is closer than 0.5m away from the robot, and it'll collide with it. Assume that your robot is 0.35m wide. A sample solution is shown in Figure 1. This program doesn't use the parameter server, but you should ensure your program obtains the relevant information from the parameter store rather than having it hard coded. Note that you'll need to import the LaserScan message, using for example `from sensor_msgs.msg import LaserScan`.

(d) Now extend the program to have the robot move randomly in the world, stopping and rotating to avoid obstacles. To do this, you'll need to publish to the `/cmd_vel` topic.

(e) Given that the robot is square, what's the minimum distance it should stop away from an obstacle to allow it to turn? Try set this as the minimum distance, and take a look if the robot is stopping correctly (it shouldn't be). Try work out why this is the case by reading through the `willow-erratic.world` world file which you can find by running textttroscd stage_ros; cd world.

(f) If the above was too easy, take a look at the `base_pose_ground_truth` topic, and note that, among other things, it returns the robot's position. Write a program to have the robot move from one point on the map to another, assuming no obstacles. If that's too easy too, think about how you can do it with obstacles present (this is something we'll be discussing in class, it's not a trivial problem).

Recap

By now you should be able to write ROS nodes which both publish messages to other nodes (e.g., allowing the robot to move), and consume messages (e.g., allowing you to deal with sensor data). You've done this in the context of Stage, allowing a virtual robot to move around (albeit randomly), avoiding obstacles. You've also learned how to use the parameter server to store configuration parameters.

```
#!/usr/bin/env python
import rospy
import roslib
import math
from sensor_msgs.msg import LaserScan

class simplebot:
    def __init__(self):
        rospy.init_node('simplebot')
        self.min_range=0.5
        self.width=0.35

        self.subscriber=rospy.Subscriber('/base_scan',LaserScan,
                                         self.checkDistance)

    def checkDistance(self,laserScan):
        curAngle=laserScan.angle_min
        inc=laserScan.angle_increment

        for range in laserScan.ranges:
          x=range*math.cos(curAngle)
          y=range*math.sin(curAngle)

          if ((abs(y)<self.width/2) and (x<self.min_range)):
            print "Obstacle at",x," ",y,", be careful!"

          curAngle=curAngle+inc
robot=simplebot()
rospy.spin()
```

Figure 1: Detecting obstacles