CS3027/5095 Robotics

Practical III

2016

Introduction

In this practical, we'll consider transformations between frames, and introduce the map server. Much of this practical is taken from the TF tutorials, found at `http://wiki.ros.org/tf/Tutorials`. As always, if you're using kinetic, you'll need to tweak the paths a bit.

Tasks

You may recall from the last practical that the sensors reported distance was sometimes incorrect. The reason for this was that the sensor was not found in the center of the robot, but was rather offset slightly. Therefore, the distances reported to obstacles were not reported with regards to the robot's center. While you could manually convert between the sensor's coordinates and the robot's center, this becomes increasingly complex when sensors are, for example, placed on a robot arm. The ROS tf package attempts to simplify the process of moving between different frames of reference.

1. After starting up roscore, fire up stage with `rosrun stage_ros stageros ‘rospack find stage_ros‘/world/willow-erratic.world`, and in another window, run `rosrun tf tf_echo /odom /base_footprint`. Drag the robot around with the mouse, and you should see numbers like the following appear:

   ```
   - Translation: [-0.308, 0.182, 0.000]
   - Rotation: in Quaternion [0.000, 0.000, 0.000, 1.000]
               in RPY (radian) [0.000, -0.000, 0.000]
               in RPY (degree) [0.000, -0.000, 0.000]
   ```

   What's going on here? `/odom` is a fixed frame of reference in the world, while `/base_footprint` is a frame of reference attached to the robot. As you move the robot around, `tf_echo` reports on the differences between the two frames *with respect to the parent (/odom) frame.*

2. To see what frames you have available to you, run `rosrun tf view_frames`, which will create a pdf file viewable with the command `evince frames.pdf`. You'll see there are 4 frames of reference available to you. Try work out where the laser is located by using `tf_echo` between the laser and robot's frame of reference.

3. Our main focus is going to be on transforming between points (or poses) in one frame to points (or poses) in another frame. Any program which does so must import the `tf` package. It's also useful to add the following imports.

   ```
   from geometry_msgs.msg import PointStamped
   from std_msgs.msg import Header
   from geometry_msgs.msg import Point
   ```

   We already know (from last week) how to move between laserscan information and (x, y) coordinates in the sensor's frame. The first thing we need to do is associate these coordinates with the sensor frame of reference, which is captured by the `PointStamped` datatype. This datatype stores the point itself, and a header listing the frame in which the point exists. Thus, for example, we can create a point in the laser's frame of reference as follows:

   ```
   ps=PointStamped(header=Header(stamp=rospy.Time.now(),frame_id="/base_laser_link"),
                               point=Point(5.0,5.0,0.0))
   ```

   This point was measured at the current point in time, is in the `base_laser_link` frame, and is at coordinate $x = 5$, $y = 5$.

4. The next step involves creating a *transform listener*, and looking up the transform. The commands to do this are

   ```
   listener=tf.TransformListener()
   ```

   and

   ```
   np=listener.transformPoint("/base_link",ps)
   ```

   The new point np will be the same point as `ps`, but in the `base_link` frame. Note that you only need to create a listener once in your program (usually during initialisation). Use this to print out the x,y coordinates of obstacles detected by the laser. A sample solution, which prints out both the x,y according to the transformation and the original x,y coordinates, can be found in Figure 1.

5. **Important:** It should be noted that you might end up with exceptions when tf isn't totally ready. When transforming points using tf, you

should always wrap your code in a `try ... except` block which continues, with except for the following exceptions: `tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException` which indicate a deeper problem in your code.

6. Another useful thing to do is to determine the robot's position in the world. This is achieved by considering the transformation between the robot's frame of reference (`/base_link`) and `/odom`. You can achieve this by again calling a method on the `listener`:

   ```
   listener.lookupTransform('/base_link', '/odom', rospy.Time(0))
   ```

   Write a program which prints out the results of this transformation as you drag the robot around the screen (this python program would be equivalent to running `rosrun tf tf_echo "/odom", "/base_link"`). Note that the result is with regards to the robot's original starting position.

7. Challenge exercise: It's time to put together everything you've learned so far. Identify a series of points in the map which can be driven to without encountering obstacles. Write a program which takes in this set of points, finds the closest one with respect to the robot's position and drives to it. Once it reaches this point, it finds the next closest one, drives to it, and so on. Now repeat this process, but have the path between a pair of points be blocked by an obstacle. The robot should try drive, and print out the (x,y) position of the obstacle to the screen.

8. While you're able to view the map on the screen, the robot (at the moment) has no awareness of the environment apart from what's being fed into its sensors. ROS provides a *map server* which allows your robot to obtain an array based map representation, and we'll now proceed to set up the map server and have the robot communicate with it.

   (a) The map server is configured through the use of a `yaml` file. Here is an example:

   ```
   image: "/opt/ros/indigo/share/stage_ros/world/willow-full.pgm"
   resolution: 0.1
   origin: [-27.0,-26.35,0.0]
   occupied_thresh: 0.5
   free_thresh: 0.3
   negate: 0
   ```

   These parameters are described in detail in `http://wiki.ros.org/map_server`. The important parameters for us are the path to the image, the resolution in meters per pixel, and the origin of the lower left pixel of the map). The threshold values are used to determine whether a pixel is counted as occupied or free by the map server. Create a `map.yaml` file in your directory with the above values.

(b) Now copy `/opt/ros/indigo/share/stage_ros/world/willow-erratic.world` and `willlow-full.pgm` from that directory to your own directory, and edit the `.world` file. Go down to the bit which states "`#load an environment bitmap`" and below it, set pose to 0 0 0 0. This will align the orientation of the mapserver and stage maps.

(c) To start stage now, you'll need to use your new mapfile - run `rosrun stage_ros stageros ./willow-erratic.world` from your current home directory.

(d) To run the map server, use `rosrun map_server map_server map.yaml`

(e) Now write a node that will listen to the `map` topic. This is a *latched* topic, meaning it will send a single message when subscribed to, and provides an occupancy grid representing the map (see http://docs.ros.org/api/nav_msgs/html/msg/OccupancyGrid.html). You can use the `info` part of the message to obtain the map resolution, and then access the `data` field to obtain the underlying map. The best way to get the map data is probably with a code fragment similar to the following, which halts node execution until the map data is obtained:

```
rospy.wait_for_message('/map',OccupancyGrid,timeout=None)
```

(f) If you do a `rostopic echo map`, you'll see the type of output that the map server provides.

(g) There are several ways to work out where you are in the world — there's a topic called `base_pose_ground_truth` for example. However, it's better to utilise a transform with origin at the 0,0 pixel of the map. For this, you can use the `fake_localization` node, which can be run using `rosrun fake_localization fake_localization`. If you do a `rosrun tf tf_echo base_link map`, and drag the robot around, you should see the translation parameter change in response to the robot's position.

(h) Now repeat the exercise from (7) above, but this time, use the absolute coordinates in the world.

9. Challenge exercise: if the above was too easy, try write a program which will use the map server to determine what the robot's range sensor will report back whenever the robot is dropped somewhere on the map.

```
import roslib
import math
import rospy
import tf
from geometry_msgs.msg import PointStamped
from std_msgs.msg import Header
from geometry_msgs.msg import Point
from sensor_msgs.msg import LaserScan


def transformToBase(x,y):
  try:
    ps=PointStamped(header=Header(stamp=rospy.Time.now(),
                                  frame_id="/base_laser_link"),
                                  point=Point(x,y,0))
    print listener.transformPoint("/base_link",ps)," ",Point(x,y,0)
  except (tf.LookupException):
    return

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

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

rospy.init_node('anodename')
listener=tf.TransformListener()
laserSub=rospy.Subscriber('/base_scan',LaserScan,checkDistance)

rospy.spin()
```

Figure 1: Code to transform laser readings between the laser and robot base frames.