INTELLIGENT ROBOTICS NOTES

October 9, 2019

School of Computer Science
University of Birmingham

Module Lead: Mohan Sridharan
Teaching Assistants: Laura Ferrante, Saif Sidhik

Contents

0.1 Assignment 1: Let's go for a walk!

For the first assignment, your task is to make your own map of the lower ground floor of the School of Computer Science, using your Pioneer Robot. To move the robot around you may use the joystick. Here below you can find some useful concepts and tutorials to guide you during your assignment.

0.2 Key stages for the assignment:

0.2.1 Get familiar with ROS: some useful tools.

Open a terminal window and type:

```
roscore
```

and, in another terminal window:

```
roslaunch socspioneer p2os_laser.launch
```

If you then open a thirdwindow and type:

```
rostopic list
```

you will get a complete list of all sensor and motor topics available to ROS when using the Pioneer robot.

0.2.2 Teleoperate your Pioneer robot using the Joystick

- Pick up a Logitech gamepad, plug it into the USB hub and then plug the USB hub into the laptop. You may also need to plug the Serial converter into the port on the side of the Pioneer.
- Open one terminal window, navigate to your Catkin workspace and type (to save time typing, you can press <tab> after typing the first few characters of each word to auto-complete the word. For example, rosla<tab> socs<tab> p2<tab>):

  ```
  roslaunch socspioneer p2os_teleop_joy.launch
  ```

- This tells ROS to use the p2os_teleop_joy launch script in the socspioneer package (p2os stands for Pioneer 2 Operating System). You can examine this script later if you want to know what it does, by typing in a terminal:

  ```
  roscd socspioneer
  ```

  (this puts you into the socspioneer package directory)

  ```
  gedit p2os_teleop_joy.launch
  ```

  A roslaunch will automatically start roscore if it detects that it is not already running, so in this case you do not need to run the following command:

  ```
  roscore
  ```

- It would be a good idea to fix the laptop to the Pioneer using the Velcro pads.

- At this point, the green 'stat' light should be flashing. Now you need to enable the motors: press the white "Motors" button to start the motor controller, and the 'stat' light should flash even more rapidly.
- Now, whilst holding down button 5 on the controller (left bumper button), you can use the joystick to control the Pioneer. Be careful though, if you bump it into things the robot will complain!
- You can kill the p2os driver by pressing Ctrl+C in the terminal.

0.2.3 Make your own map

At some point, you may want to create your own map of the lower ground floor as part of your project. Making a map is fairly simple:

1. Take control of the robot:

```
roslaunch socspioneer p2os_laser.launch
roslaunch socspioneer teleop_joy.launch
```

2. Record a bag file of data:

```
rosbag record –O <file> /base_scan /tf /odom
```

3. Now drive the robot around. Try to drive smoothly, and to make sure the laser can 'see' every part of the area (trying to avoid 'laser shadows' behind walls and obstacles in particular). You may want to go back and forth over certain areas a couple of times.
4. Try to bring the robot back to its starting location at the end of the recording process, so it can form a complete loop from start to finish.
5. Kill the p2os_laser and teleop_joy processes using Ctrl+C.
6. Set simulation time to true:

```
rosparam set use_sim_time true
```

7. Run the gmapping software

```
rosrun gmapping slam_gmapping scan:=base_scan
```

8. Replay the data file

```
rosbag play <file>
```

   The gmapping process should start producing output. If it doesn't, make sure the gmapping command has the correct laser topic name (in this case base_scan). If you're unsure what the topic name should be, try rostopic list for a list.

9. Once the rosbag has finished playing, save the map:

```
rosrun map_server map_saver
```

10. There should be a file map.pgm in your directory which you can check visually. Ideally there should be very little grey 'unknown' data within the map, and the walls should all be straight. If not, you may want to try again!

11. It's probably a good idea to crop the map using gimp or some other image editing program and to tidy it up, maybe removing any flaws or 'unseen data' in open space which will prevent your robot from planning a route for example.

12. Have a look at one of the existing .world and .yaml files to see what details you may want to change, if any. At the very least, if you create a .world file based on one of the existing ones, you will need to change the map.pgm name, and if you rename the map at all, you will also need to update map.yaml.

13. Don't forget to set simulation time back to false, otherwise you won't be able to do anything with the robot

```
rosparam set use_sim_time false
```

0.2.4 Part B: Simulating using Stage and visualising using Rviz

Sometimes you will want to be able to test your code, but your robot won't be available: perhaps another team mate is using it, or the battery is charging, or you're working at home. For these situations, ROS provides a package called stage. Stage creates a simulated robot in a pre-defined world, and plugs into the ROS publish/subscribe mechanism, so you can control it and receive its simulated laser scan data just as if it was a real robot connected to your laptop. Remember that **your demonstration will require you to prove that your code is working on the physical robot**: if your code works in simulation, but not on the physical robots you will not receive marks.

- To run stage, open a terminal and type:

```
rosrun stage_ros stageros /data/private/ros/socspioneer/lgfloor.world
```

  This tells ROS to run the *stageros* program in the stage package, and to use the willow-erratic world description file.

- Remember, you will need to have a running roscore in another terminal if you don't already have one open.

- Try dragging the map (or the robot!) around by clicking and holding the mouse on it.

- If you hold down Ctrl whilst moving the mouse, the world will pan and spin.

- Use the scroll pad (on the right hand side of the mouse touchpad) or scroll-wheel on a mouse to raise and lower your altitude. Obviously, the robot is not receiving any messages to tell it to move, so it just sits in place on the map. However, its simulated laser is still scanning and publishing data on the topic */base_scan*. We can use a tool called rviz to get a 'robot's eye view' of the world by displaying the data being published by the laser.

- As always, make sure you have a roscore running in a terminal before you start!
- Open a terminal and type:

```
rosrun rviz rviz
```

- When rviz loads, it will show a blank screen. To display the laser data, click 'Add' and choose a 'Laser Scan' (if there are no display types available, go to Plugins -> Manage and tick 'Load built-in').
- In the 'Displays' pane on the left, expand the Laser Scan section and find the 'Topic' variable. If you click in the empty box next to it and on the grey button which appears, a window will pop up.
- Choose /base_scan (sensor_msgs/LaserScan).
- Now under 'Global Options', find the 'Fixed Frame' variable. Choose '/base_link' from the drop-down list. Now the laser data should be displayed, and you can drag-and-drop the robot around in the stage window to see how the laser data changes.

0.2.5 Part C: Writing a basic controller node

For letting the robot explore the floor you will need to send commands to the actuators. The example below show you how to write a simple node which published velocity commands and cause the (simulated) robot to move forward.

0.2.5.1 Publisher

- Create a node to send motion commands to the robot. An example of the scrips can be the following:

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist

def talker():
    pub = rospy.Publisher('cmd_vel', Twist, queue_size=100)
    rospy.init_node('Mover', anonymous=True)
    # rate = rospy.Rate(10) # 10hz

    while not rospy.is_shutdown():
        base_data = Twist()
            base_data.linear.x = 0.1
        pub.publish( base_data  )
        # rate.sleep()

if __name__ == '__main__':
```

```
try:
    talker()
except rospy.ROSInterruptException:
    pass
```

- If you run the node and look at the stage window (see part B if you need to get stage running again), you should see your robot travelling slowly forward. The laser data shown in rviz should also be changing as your simulated robot moves.
- You can type in another terminal the following command window to observe the commands your program is sending to the simulated robot

  ```
  rostopic echo /cmd_vel
  ```

  Try to get used to using

  ```
  rostopic list
  ```

  and

  ```
  rostopic echo
  ```

- as they will be very useful when debugging your robot's software. As always in Linux, you can press Ctrl+C to kill your program in the terminal.

0.2.5.2 Laser data

Now you have a Publisher which can command the robot to move in a certain direction, it would be helpful to make use of the available laser sensor data so that the robot can make informed decisions on its own about where and how it should move. First, it would help to know what format the laser data comes in:

- After ensuring stage is running, open a terminal and type

  ```
  rostopic list
  ```

- These are all the topics (or channels) on which a node can listen.
- The laser data is being published on the topic base_scan. You can 'listen' to it by typing

  ```
  rostopic echo /base_scan
  ```

- Press Ctrl+C to stop listening. If you examine one of the messages, you will notice that it is simply a header and a long array of numbers, each of which is the range reported by the laser in a particular direction during its spin.

0.2.5.3 Subscribing to laser messages and publishing to robot controller

We can now use those ranges when we come to write a program to control the robot.

- This page (`http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)`) will show you how to write a node in Python which subscribes to a topic and publishes on another topic. Use this example to write node subscribe to the LaserScan messages on the /base_scan topic, and publish movement command messages to the robot on the /cmd_vel topic.
- Use the ranges[] field in the LaserScan messages to obtain the laser data. You can find here `http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html` the API.
- Test your code using stage.
- Hint 1: Use rostopic echo /base_scan to examine the laser messages coming in from the robot. What sort of ranges do you get when the robot is in front of an obstacle?
- Hint 2: The field in the Twist object which causes the robot to drive forward is twist.linear.x (positive values to drive forward; negative values to drive in reverse). Click here (`http://wiki.ros.org/mini_max/Tutorials/Moving%20the%20Base` for a simple tutorial on how to move your base in Python.
- Hint 3: The program rqt_graph (`http://wiki.ros.org/rqt_graph`) gives you a very helpful interactive display of all the current ROS nodes, including which topics they are publishing on and subscribing to.
  Once it's working in simulation, try it on an actual robot:
- Plug the USB to serial converter and the laser sensor into the USB hub on the robot, and plug this into the computer. Make sure the robot is switched on.
- Open two terminals. In one, run:

  ```
  roslaunch socspioneer p2os_laser.launch
  ```

  This will start up the Pioneer drivers and the laser driver. The Pioneer should beep. In the second terminal, compile and run your node.
- Now firmly attach the laptop to the Pioneer using the Velcro pads.
- Once you're sure everything is safe, press the white 'Motor' button to enable the Pioneer's motors. This button also acts as an emergency kill switch to stop the motors but keep the Pioneer running.

0.2.6 Hints!

1. It may help to group the laser ranges into a number of blocks (e.g. 'left', 'right', 'straight on'), then find the average range in each block, and simply turn the robot left, right, or carry on straight ahead, depending on which laser block has the longest average range.
2. The fields for sending commands to the robot are twist.linear.x (forward motion; negative to reverse) and twist.angular.z (z-axis anti-clockwise rotation; negative to go clockwise).

3. You may want to test your code using stage before trying it out on the actual robot.
4. Movement commands only last for a short duration (less than a second) to prevent the robot driving into danger if one of the nodes crashes. For continued motion, you should publish the movement commands repeatedly in a for-loop, using sleep command (or a similar length of time), or alternatively calculate and publish appropriate movement commands every time you receive a LaserScan message.
5. The Pioneers not only have a laser sensor, but also 8 sonar sensors (the gold discs on the front). It is these which make the repeated clicking sound when you start the robot. Sonar may or may not be useful for you during navigation (the laser is certainly likely to be more accurate), but it may still provide you with usable data if you choose to try and use it. Sonar data comes in on topic /sonar, so create a subscriber for this in your ROS node. You will also need to import SonarArray.

.1 Rviz

Rviz (`http://wiki.ros.org/rviz`) is a visualisation tool which is used to view the robot's position in the map and can also be used for drawing your own shapes onto the map. Below are the various different things that can be added to an rviz 'scene' (by clicking the 'add' button in the sidebar) The following is a bare-bones node which adds a spherical Marker at the origin of the */odom* frame. Functions can be created to add arbitrary shapes or text for debugging or display purposes. Note that lots of identical markers are better processed as a MarkerArray.

```
#!/usr/bin/env python
import rospy
# documentation: http://wiki.ros.org/rviz/DisplayTypes/Marker
from visualization_msgs.msg import Marker, MarkerArray
from geometry_msgs.msg import Point

rospy.init_node(name='marker_demo')
mark_pub = rospy.Publisher('visualization_marker', Marker, queue_size=10)
id_counter = 0

# place a point
m = Marker()

# specify reference frame (options in RViz Global Options > Fixed Frame)
# markers relative to 0,0 in odometry
m.header.frame_id = '/odom'
m.header.stamp    = rospy.Time.now()

# marker with same namespace and id overrides existing
m.ns = 'my_markers'
m.id = id_counter
```

```
id_counter += 1

m.type = Marker.SPHERE
m.action = m.ADD

m.pose.position.x = 0
m.pose.position.y = 0
m.pose.position.z = 0
m.pose.orientation.x = 0
m.pose.orientation.y = 0
m.pose.orientation.z = 0
m.pose.orientation.w = 1
m.scale.x = 1
m.scale.y = 1
m.scale.z = 1
m.color.r = 1
m.color.g = 0
m.color.b = 0
m.color.a = 1

m.lifetime = rospy.Duration() # forever
#m.lifetime = rospy.Duration(...)

mark_pub.publish(m)
rospy.spin()
```

.2 Stage Simulator

Stage is the name of the robot simulator. It is fed a description of the world and the robot in a .world text file, and an occupancy map in the form of a .pgm image. Below is a script to start the stage simulator, map server and rviz (using the world description given in the socspioneer directory which should be present on the provided laptops):

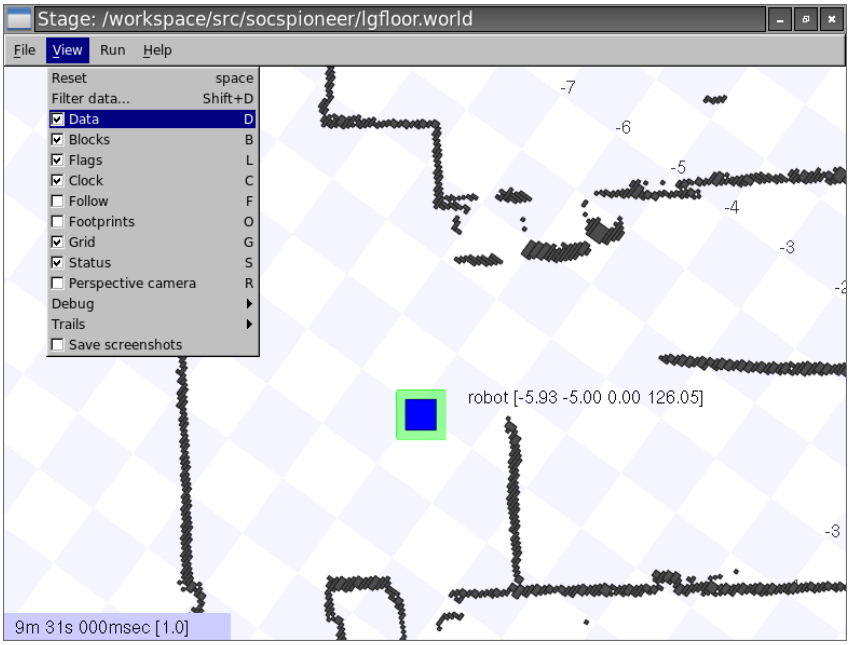
```
rosrun map_server map_server "socspioneer/lgfloor.yaml" &>/dev/null &
rosrun stage_ros stageros "socspioneer/lgfloor.world" &>/dev/null &
rosrun rviz rviz -d "/workspace/src/tools/docker/stage.rviz"
```

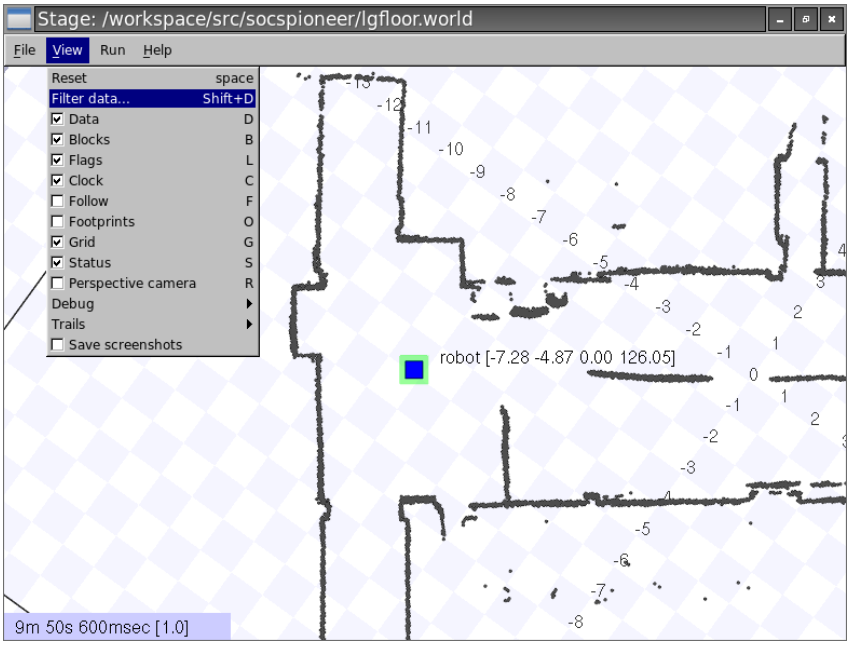
The map server is configured with a .yaml file and in this case uses the same occupancy map as the simulator. The map server allows you to view the map in rviz. The following node can be used to drive the simulated robot using the keyboard:

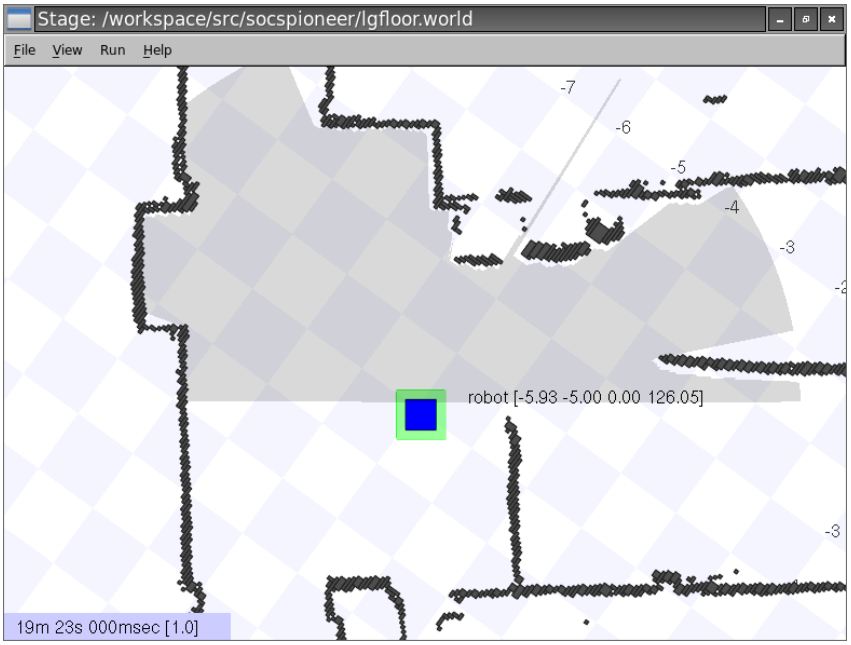
```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

.3 Viewing Laser Ranges

To view the laser scanner data in stage perform the following steps:






Stage: /workspace/src/socspioneer/lgfloor.world
File View Run Help
-7
-6
-5
-4
-3
robot [-5.93 -5.00 0.00 126.05]
-3
19m 23s 000msec [1.0]

.4 RQT

Rqt (`http://wiki.ros.org/rqt`) is a collection of various GUI tools (plugins) for working with ROS. It can be run with:

```
$ rqt &
```

Initially the window will be blank, but using the toolbar, various 'plugins' can be loaded into panels. For example, rviz is actually a plugin of rqt.

.5 Transforms/Transform Tree

The transform tree viewer is useful for diagnosing problems with the ROS transforms which convert between reference frames (coordinate systems). For example there is a reference frame fixed at the laser scanner on the robot (usually called base_link, and another called base_laser_link only in the stage simulator) and one which is fixed at the origin of the map (usually called map, there is also a similar frame called odom which is based on the odometry information). Coordinates can be provided relative to any of these reference frames, but sometimes they have to be represented in another frame, which requires the ROS transform tree to tell the software how to transform between these frames. You don't have to interact with the transform tree directly, but various parts of ROS depend on the tree being correct.

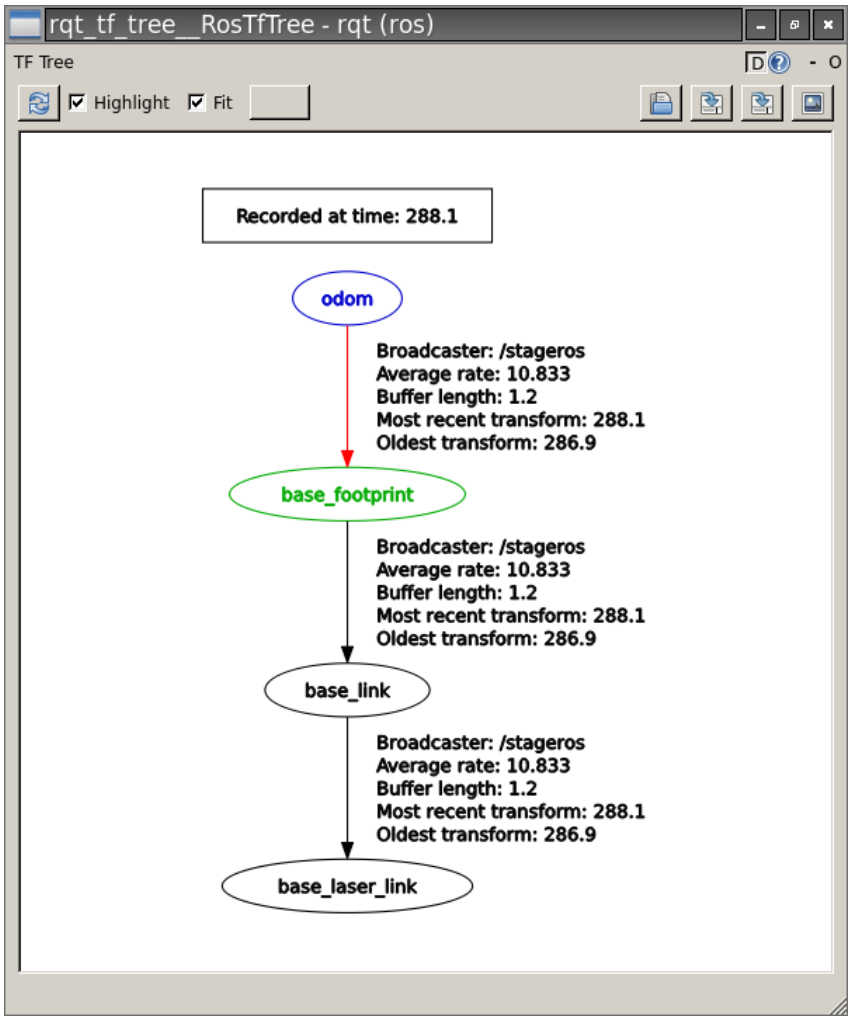
```
$  rosrun rqt_tf_tree rqt_tf_tree
```

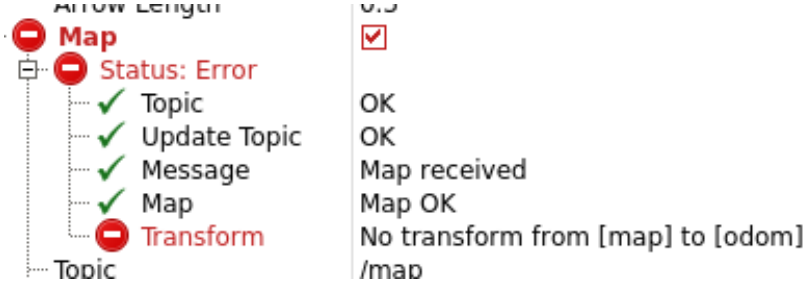
Below is an example transform tree when running in the stage simulator: Here are just some examples of transform issues that you might run into (in rviz).

Problem 1

There is no transform between the map (as provided by the map server) and odometry (provided by stage) frames. This is because the robot is not localised on the map and so doesn't know where it is. A hacky 'fix' is to introduce a node which supplies a fixed/static transform (which should be configured based on your particular situation). The better approach would be to run a node which localises the robot.

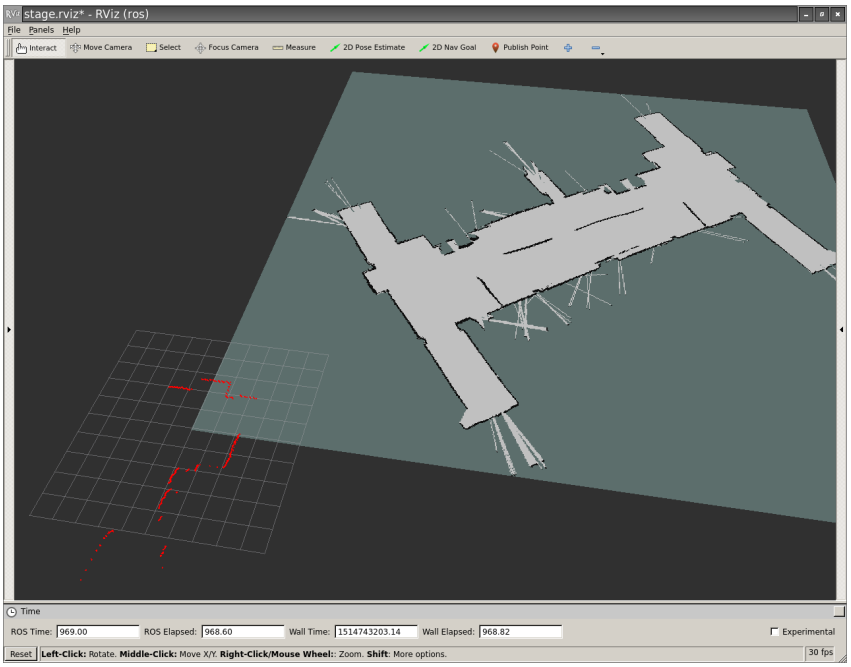
```
$ rosrun tf static_transform_publisher 0 0 0  0 0 0 1  /map /odom 100
or add to a launch file

<launch>
    <!-- http://wiki.ros.org/tf#static_transform_publisher -->
    <!-- arguments are:
        x y z qx qy qz qw frame_id child_frame_id  period_in_ms
    -->
    <node pkg="tf" type="static_transform_publisher" name="odom_to_map"
        args="0 0 0  0 0 0 1  /map /odom 100" />
</launch>
```





Problem 2
Rviz is currently configured to base the global view around the */map* frame, however it doesn't exist in the transform tree. The map server provides an occupancy map in the /map frame but doesn't explain how it relates to the rest of the scene. When the robot is localised, e.g. using the navstack, then the localisation node publishes the transform between */base_link* (the robot) and */map*.

Problem 3 If transforms are present, but wrong, situations like this are common, where things don't line up correctly or move in the right direction. Here you can see the laser scan data



does not line up with the map.