INTELLIGENT ROBOTICS NOTES

October 22, 2019
School of Computer Science
University of Birmingham

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

Contents

Chapter 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.

1.1 Key stages for the assignment:

1.1.1 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.

1.1.2 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.

1.1.3 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
```

1.2 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.

1.3 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.

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.4 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.

Chapter 2

Assignment 2: Where am I?

Autonomous Mapping

The first part of the assignment will be to implement autonomous mapping for your robot. Your node should use the laser scan data to detect obstacles and perform appropriate motor actions to avoid / move around them. Your robot should build a map that covers as much area of the LG floor as possible, while moving autonomously without crashing or getting stuck.

Localisation using Particle Filter

For the second part of the assignment, you will implement and visualise particle filter localisation of the robot within the map that you built. The robot has to move using the autonomous exploration node written by you. This chapter will get you started on writing your own localisation node.

2.1 Localisation with AMCL

Before you write your own particle filter, it is good to test the AMCL library that comes built-in with ROS. To see how localisation within ROS works, first get the AMCL localisation software up and running:

1. In a terminal, run the Pioneer driver, as well as the joystick driver:

   ```
   roslaunch socspioneer p2os_teleop.launch
   ```

   Note: If you have implemented autonomous exploration, you can run the node here. In that case you do not need the joystick driver. You only need the laser and motor drivers ('p2os_ laser.launch'). Remember ROS Master has to be running before running any node (master is started using 'roscore' or by starting a persistent 'launch' file)

2. Run a map server in another terminal:

   ```
   rosrun map_server map_server <path_to_your_map_yaml_file>
   ```

   The *socspioneer* package has a 'yaml' file of the (outdated) LG floor map ('socspioneer/lgfloor.yaml').

3. Run the AMCL localisation node:

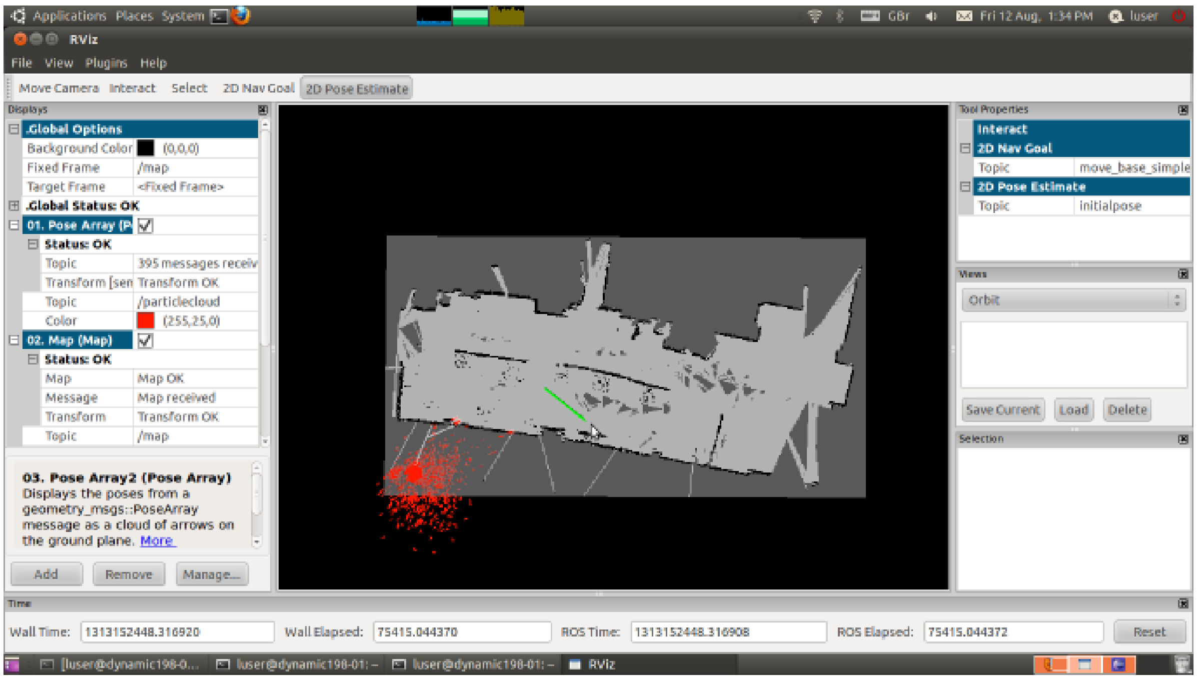
```
    rosrun amcl amcl scan:=base_scan
```

4. To visualise the localisation:

   - Run rviz:

   ```
       rosrun rviz rviz
   ```

   - Set the Fixed Frame to */map*:
   - Add a Pose Array view listening on the */particlecloud* topic
   - Add a Map view listening on the */map* topic
   - Finally, click the *2D Pose Estimate* button, then draw an arrow on the map showing the approximate location and direction in which your robot is facing (example in Figure 2.1).



.Global Options		
Background Color	■	(0,0,0)
Fixed Frame	/map	
Target Frame	<Fixed Frame>	
.Global Status: OK		
01. Pose Array (P	✓	
Status: OK		
Topic	395 messages receiv	
Transform [sen	Transform OK	
Topic	/particlecloud	
Color	🟥	(255,25,0)
02. Map (Map)	✓	
Status: OK		
Map	Map OK	
Message	Map received	
Transform	Transform OK	
Topic	/map	

03. Pose Array2 (Pose Array)
Displays the poses from a geometry_msgs::PoseArray message as a cloud of arrows on the ground plane. More

Interact	
2D Nav Goal	
Topic	move_base_simple
2D Pose Estimate	
Topic	initialpose

Wall Time: 1313152448.316920 Wall Elapsed: 75415.044370 ROS Time: 1313152448.316908 ROS Elapsed: 75415.044372

Figure 2.1

   - Enable the motors by pressing the white button on the Pioneer.
   - As you drive the Pioneer around using the joystick, AMCL will read in the laser data (topic */base_ scan*) and odometry data (topic */odom*) and continually update the particle cloud with the predicted location of the Pioneer.

2.2 PF Localisation Package: Writing your own localisation node

Link to package: `https://canvas.bham.ac.uk/files/8147779/download?download_frd=1`

For this exercise, you will write a replacement ROS node for AMCL in python which runs a basic particle filter and localises the robot within the supplied map. Unpack the *pf_ localisation* python package (provided) into your '*<catkin_ workspace>/src*' directory. The instructions to build the package and run the node can be found in 'README.md' file in the package.

This package contains a ROS node '*scripts/node.py*' which handles all the ROS backend stuff, and an abstract class PFLocaliserBase (*'src/pf_ localisation/pf_ base.py'*) which provides an easy way to interface with ROS and incorporate a SensorModel (*'src/pf_ localisation/sensor_ model.py'*) object (*'self.sensor_ model'*) that provides particle weight calculation. Your task for this exercise is to write a class called PFLocaliser which will extend PFLocaliserBase and provide localisation given a map, laser readings, and an initial pose estimate. A skeleton of this class is provided in *'src/pf_ localisation/pf.py'*, which you must complete. **You can successfully implement the particle filter localisation by editing JUST this file**.

The remainder of this section will the main parts of this class.

2.2.1 Constructor

The first thing your constructor should do, as when implementing any subclass, is to call the superclass constructor. Your constructor will also need to assign values for the odometry motion model:

```
# Set motion model parameters
self.ODOM_ROTATION_NOISE = ???? # Odometry model rotation noise
self.ODOM_TRANSLATION_NOISE = ???? # Odometry model x axis (forward) noise
self.ODOM_DRIFT_NOISE = ???? # Odometry model y axis (side-to-side) noise
```

These values will be used in the odometry update (in the method *'PFLocaliser:predict_from_ odometry()'* inherited from the superclass).

You will then need to implement the following three abstract methods:

2.2.2 initialise_particle_cloud(self, initialpose)

- Called whenever a new initial pose is set in rviz.
- This should instantiate and return a PoseArray object, which contains a list of Pose objects. Each of these Poses (corresponding to a particle in the particle cloud) should be set to a random position and orientation around the initial pose, e.g. by adding a Gaussian random number multiplied by a noise parameter to the initial pose.
- Orientation in ROS is represented as Quaternions, which are 4-dimensional representations of a 3-D rotation describing pitch, roll, and yaw ("heading"). This is more complex as the Pioneer only rotates around the yaw axis. To make it easier for you, you have been provided with some utility

methods for handling rotations in *'util.py'* imported in code as 'import pf_localisation.util':

```
rotateQuaternion(q_orig, yaw)
```

Takes an existing Quaternion q_orig, and an angle in radians (positive or negative), and returns the Quaternion rotated around the heading axis by that angle. So, for example, you can take the Quaternion from the Pose object, rotate it by Math.PI/20 radians, and insert the resulting Quaternion back into the Pose.

```
getHeading(q)
```

This function performs the reverse conversion and gives you the heading (in radians) described by a given Quaternion q.

2.2.3 update_particle_cloud(self, scan)

- Called whenever a new LaserScan message is received. This method does the actual particle filtering.
- The PFLocaliserBase will already have moved each of the particles around the map approximately according to the odometry readings coming from the robot. But odometry measurements are unreliable and noisy, so the particle filter makes use of laser readings to confirm the estimated location.
- Your method should get the likelihood weighting of each Pose in self.particlecloud using the *self.sensor_model.get_weight()* method. This weighting refers to the likelihood that a particle in a certain location and orientation matches the observations from the laser, and is therefore a good estimate of the robot's pose.
- You should then resample the particlecloud by creating particles with a new location and orientation depending on the probabilites of the particles in the old particle cloud, for example by doing roulette-wheel selection to choose high-weight particles more often than low-weight particles. Each new particle should have resampling noise added to it, to keep the particle cloud spread out enough to keep up with any changes in the robot's position.
- The new particle cloud should be assigned to self.particlecloud to replace the existing one.

2.2.4 estimate_pose()

- *self.particlecloud* should return the estimated position and orientation of the robot based on the current particle cloud. You could do this by finding the densest cluster of particles and taking the average location and orientation, or by using just the selected 'best' particle, or any other method you prefer – but make sure you test your method and justify it! Taking the average position of the entire particle cloud is probably not a good solution... can you think why not?

- To find the average orientation of a set of particles, you will encounter a problem because angles increase from 0 to pi radians then continue from -pi back to 0. For example, if you have two particles, one facing at -179 degrees and one facing at 179 degrees (only 2 degrees apart in reality), the mean orientation will be -179 + 179 / 2 = 0 degrees, not 180 degrees.

- This situation is exactly what quaternions were created for. Instead of calculating the heading of each particle and finding the mean, you can simply take the mean of each of the x, y, z and w values directly from the Quaternions (using getW() etc.) before creating a new Quaternion with these mean values. This new quaternion represents the average heading of all the particles in your set. Instead of using the whole set of points in your cloud, you might want to do some filtering to choose the ones you want to rely on for orientation.

2.3 Tips!

- As well as seeing your code running as your Pioneer explores your map, we will look for evidence that you have experimentally (i.e. in the form of records in some log book) investigated how the particle filter behaves, including (but not limited to): adding noise to the particle cloud, visualising the sensor model function, etc. *self.sensor_ model.predict(obs_ range, map_ range)* by plotting a graph of probabilities for various observation and prediction ranges, and measuring the overall location error over time.

- Remember to go through the 'README.md' file in the *'pf_ localisation'* package for more information about the package and its usage.

- Remember to use *'rostopic echo <topic>'* and/or rviz to listen to the messages which are being passed between all nodes, including yours for debugging, testing etc.

- Your node, unlike AMCL, also publishes the estimated pose as a message which can be displayed by rviz. Add an rviz display for messages of type 'PoseStamped' on topic *'estimatedpose'* to see the exact position estimate for your robot.

- The node doesn't call your PFLocaliser update methods until odometry update messages have been recieved (i.e. it doesn't update automatically whenever there is a laser scan, only when the robot has actually moved). So you will need to run *'teleop_joy'*, *'keyboard_ teleop'* or the autonomous exploration node, and make the robot move before your particle cloud will appear on the map in rviz.

- Rotations around the compass are in radians. For example, 0 degrees is North and -3/4 pi degrees is South-West. You may find it easier to keep everything in radians as *math.cos()* and *math.sin()* take their arguments in radians, but if you want to work in degrees you can make use of the *math.radians()* method. For example, you could use *rotateQuaternion(heading, math.radians(90))* to rotate by 90 degrees, or *rotateQuaternion(heading, math.pi/2)* to do the same rotation in radians.

Appendix A

Additional Notes

A.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()
```

A.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):

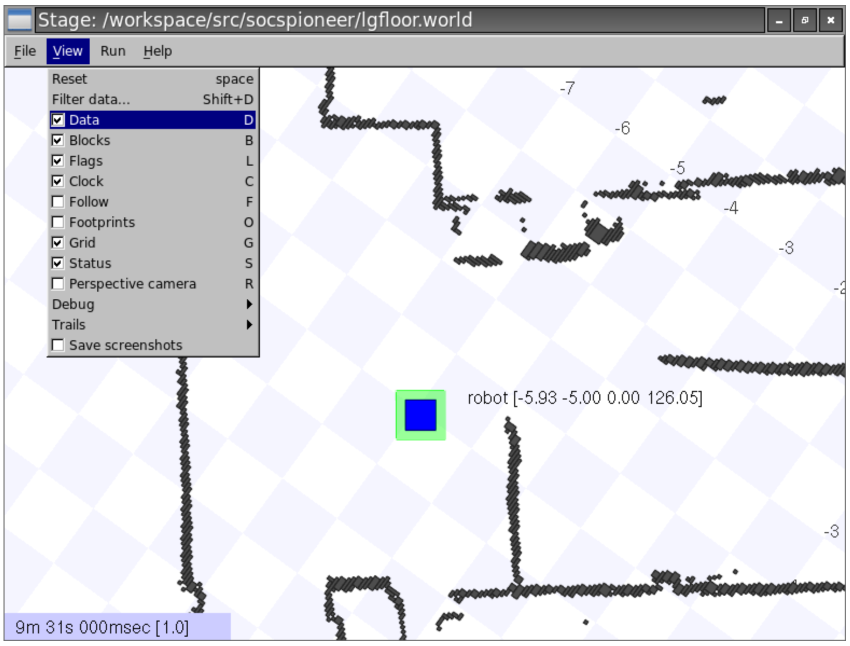
```
rosrun map_server map_server "socspioneer/lgfloor.yaml" &>/dev/null &
rosrun stage_ros stageros "socspioneer/lgfloor.world" &>/dev/null &
rosrun rviz rviz -d "/workspace/src/tools/docker/stage.rviz"
```

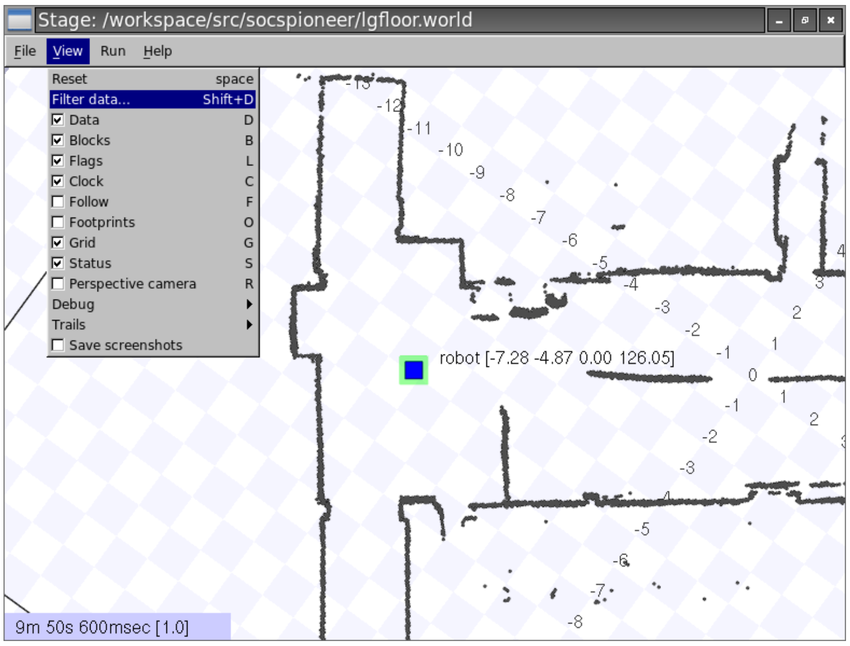
The map server is configured with a .yaml file and in this case uses the same occupancy map as the simulator. The map server allows you to view the map in rviz. The following node can be used to drive the simulated robot using the keyboard:

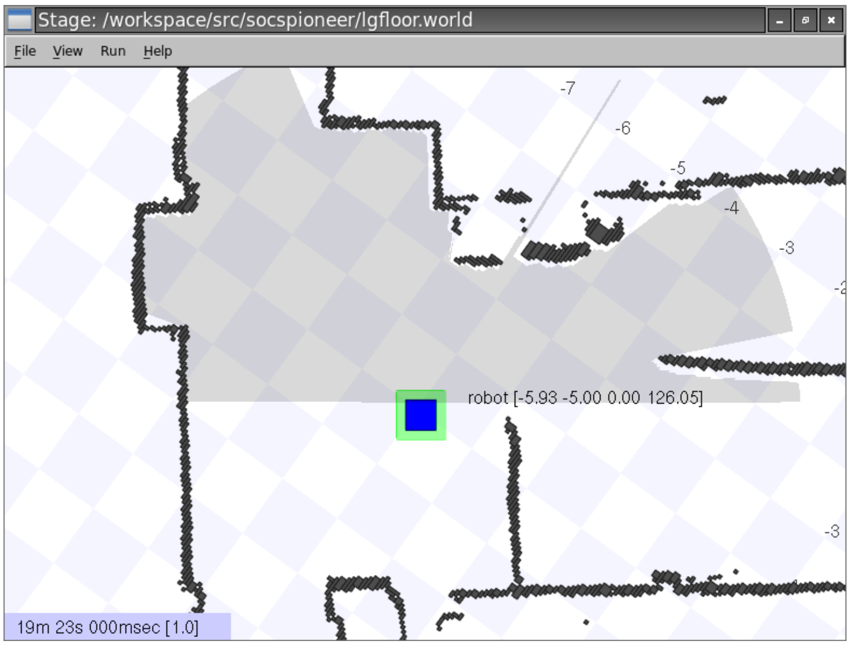
```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

A.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
-3
robot [-5.93 -5.00 0.00 126.05]
19m 23s 000msec [1.0]

A.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.

A.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
```

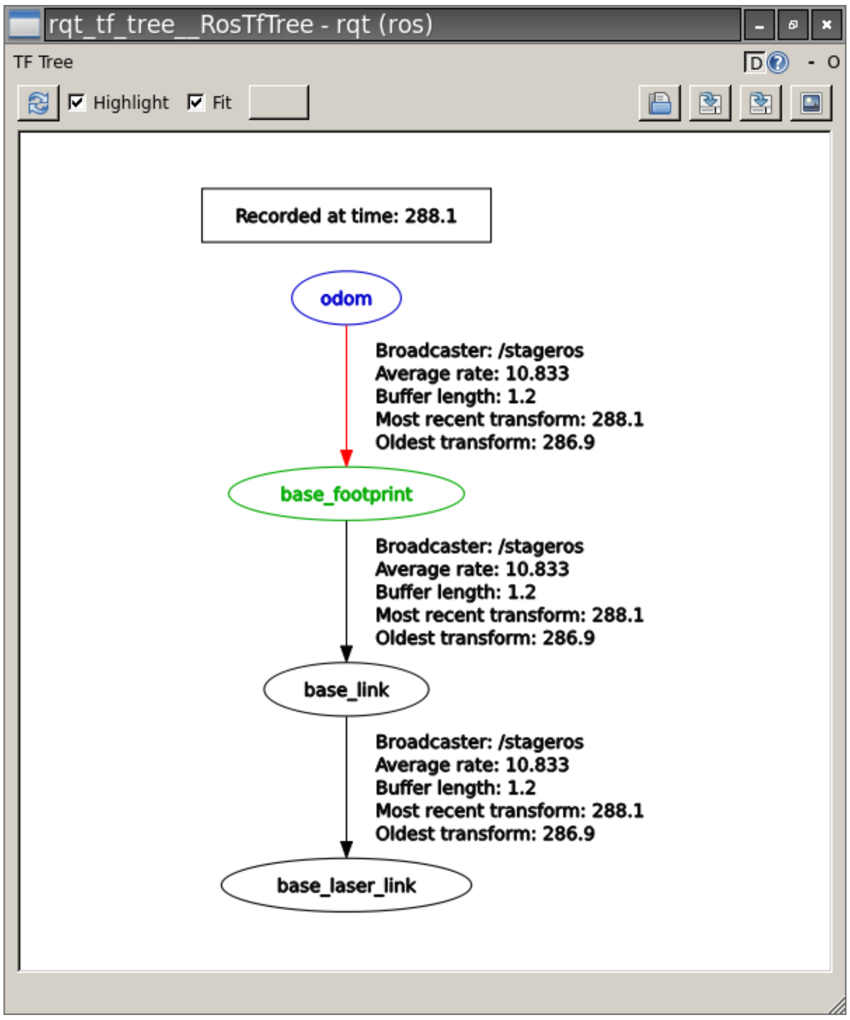
Below is an example transform tree when running in the stage simulator: Here are just some examples of transform issues that you might run into (in rviz).

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>
```



Recorded at time: 288.1

odom

Broadcaster: /stageros
Average rate: 10.833
Buffer length: 1.2
Most recent transform: 288.1
Oldest transform: 286.9

base_footprint

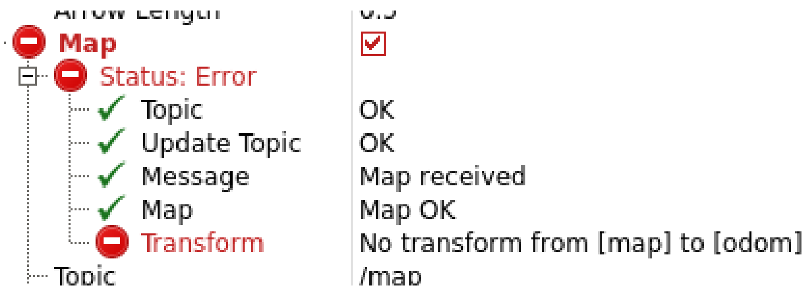
Broadcaster: /stageros
Average rate: 10.833
Buffer length: 1.2
Most recent transform: 288.1
Oldest transform: 286.9

base_link

Broadcaster: /stageros
Average rate: 10.833
Buffer length: 1.2
Most recent transform: 288.1
Oldest transform: 286.9

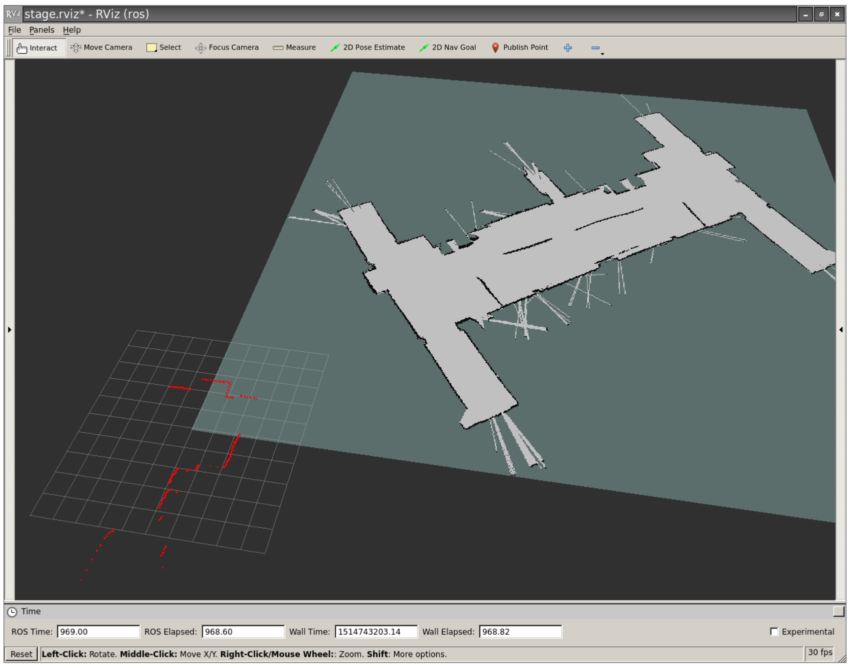
base_laser_link

Problem 2
Rviz is currently configured to base the global view around the /*map* frame, however it doesn't



Arrow Length	0.3
Map	✓
Status: Error	
Topic	OK
Update Topic	OK
Message	Map received
Map	Map OK
Transform	No transform from [map] to [odom]
Topic	/map

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



ROS Time: 969.00 ROS Elapsed: 968.60 Wall Time: 1514743203.14 Wall Elapsed: 968.82

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

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.