Week 2 Lab – Your First ROS Node

This second lab is designed to help you get started with ROS. This lab will help you create your first ROS package, set up your development environment and explore some of the basic concepts you have seen in lectures.

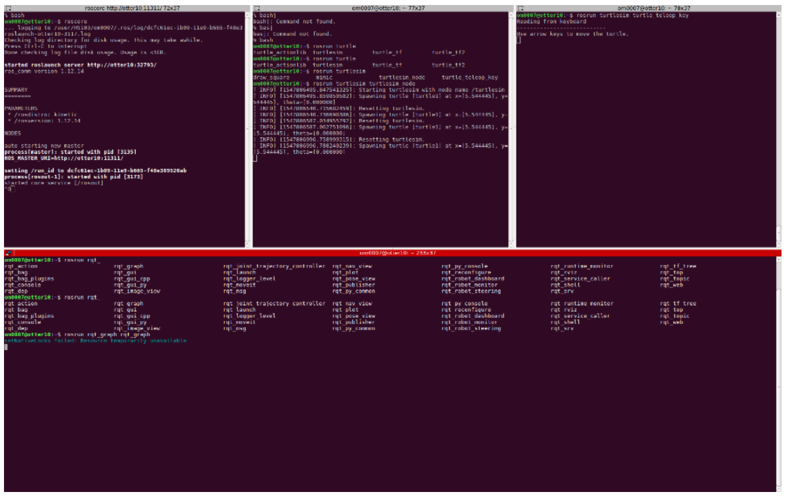
Exercise 1 – Turtlesim Parameters and Services

We will use the turtlesim simulator to explore the command-line use of parameters and services which we saw in the previous lecture.

1. **Start Turtlesim:** Start Terminator, and in the same tab start a roscore and use rosrun to start turtlesim and turtle_teleop_key. Remember the rosrun syntax and the use of **<TAB>** to autocomplete.

   ```
   $ rosrun <package_name> <node_name>
   ```

   Also remember how to split a space in terminator (*Right Click* or *Ctrl+Shif+O* and *Ctrl+Shift+E*). Your window should look something like this (don't worry about space placement):

   ```
   roscore http://otter10:11311/ 72x37
   % bash
   om0007@otter10:~$ roscore
   ... logging to /user/HS103/om0007/.ros/log/dcfc61ec-1b09-11e9-b665-f48e3
   roslaunch-otter10-3117.log
   Checking log directory for disk usage. This may take awhile.
   Press Ctrl-C to interrupt
   Done checking log file disk usage. Usage is <1GB.

   started roslaunch server http://otter10:32793/
   ros_comm version 1.12.14


   SUMMARY
   ========

   PARAMETERS
    * /rosdistro: kinetic
    * /rosversion: 1.12.14

   NODES

   auto starting new master
   process[master]: started with pid [3135]
   ROS_MASTER_URI=http://otter10:11311/

   setting /run_id to dcfc61ec-1b09-11e9-b665-f48e389520ab
   process[rosout-1]: started with pid [3173]
   started core service [/rosout]
   ^Q
   ```

   ```
   om0007@otter10: ~ 77x37
   % bashj
   bashj: Command not found.
   % basj
   basj: Command not found.
   % bash
   om0007@otter10:~$ rosrun turtle
   turtle_actionlib  turtlesim         turtle_tf         turtle_tf2
   om0007@otter10:~$ rosrun turtle
   turtle_actionlib  turtlesim         turtle_tf         turtle_tf2
   om0007@otter10:~$ rosrun turtlesim
   draw_square         mimic              turtlesim_node     turtle_teleop_key
   om0007@otter10:~$ rosrun turtlesim turtlesim_node
   [ INFO] [1547806495.847541325]: Starting turtlesim with node name /turtlesim
   [ INFO] [1547806495.859859582]: Spawning turtle [turtle1] at x=[5.544445], y=
   544445], theta=[0.000000]
   [ INFO] [1547806540.715602459]: Resetting turtlesim.
   [ INFO] [1547806540.736698306]: Spawning turtle [turtle1] at x=[5.544445], y=
   [ INFO] [1547806587.034955792]: Resetting turtlesim.
   [ INFO] [1547806587.062751096]: Spawning turtle [turtle1] at x=[5.544445], y=
   [5.544445], theta=[0.000000]
   [ INFO] [1547806996.758999315]: Resetting turtlesim.
   [ INFO] [1547806996.788240239]: Spawning turtle [turtle1] at x=[5.544445], y=
   [5.544445], theta=[0.000000]
   ```

   ```
   om0007@otter10: ~ 78x37
   om0007@otter10:~$ rosrun turtlesim turtle_teleop_key
   Reading from keyboard
   ---------------------------
   Use arrow keys to move the turtle.
   ```

   ```
   om0007@otter10: ~ 255x37
   om0007@otter10:~$ rosrun rqt_
   rqt_action        rqt_graph         rqt_joint_trajectory_controller  rqt_nav_view      rqt_py_console       rqt_runtime_monitor  rqt_tf_tree
   rqt_bag           rqt_gui           rqt_launch                       rqt_plot          rqt_reconfigure      rqt_rviz             rqt_top
   rqt_bag_plugins   rqt_gui_cpp       rqt_logger_level                 rqt_pose_view     rqt_robot_dashboard  rqt_service_caller   rqt_topic
   rqt_console       rqt_gui_py        rqt_moveit                       rqt_publisher     rqt_robot_monitor    rqt_shell            rqt_web
   rqt_dep           rqt_image_view    rqt_msg                          rqt_py_common     rqt_robot_steering   rqt_srv
   om0007@otter10:~$ rosrun rqt_
   rqt_action        rqt_graph         rqt_joint_trajectory_controller  rqt_nav_view      rqt_py_console       rqt_runtime_monitor  rqt_tf_tree
   rqt_bag           rqt_gui           rqt_launch                       rqt_plot          rqt_reconfigure      rqt_rviz             rqt_top
   rqt_bag_plugins   rqt_gui_cpp       rqt_logger_level                 rqt_pose_view     rqt_robot_dashboard  rqt_service_caller   rqt_topic
   rqt_console       rqt_gui_py        rqt_moveit                       rqt_publisher     rqt_robot_monitor    rqt_shell            rqt_web
   rqt_dep           rqt_image_view    rqt_msg                          rqt_py_common     rqt_robot_steering   rqt_srv
   om0007@otter10:~$ rosrun rqt_graph rqt_graph
   setNativeLocks failed: Resource temporarily unavailable
   ```

   drive the turtle around using the keyboard and familiarise yourself with the interface again. Once you are comfortable these nodes are running correctly move on to the next task.

2. **Finding Services:** Services are a crucial part of the ROS ecosystem. They allow you to perform simple tasks on command. Services are defined by two messages, a request and a response. The simplest services use the *std_srvs/Empty* type. These services simply send an empty request and receive an empty reply. Turtlesim contains two of these services, to figure out what they are run the command

   ```
   $ rosservice list
   ```

   you should see, among others, two services called "/clear" and "/reset".

3. **Calling Services:** Run the command

   ```
   rosservice call /clear "{}"
   ```

   you should see the track that the turtle leaves dissapear. Now try to call the "/reset" service. You should see the turtle reset to its original position.

4. **Teleport Service:** Getting the turtle to an exact place is practically impossible using the keyboard teleoperation node. Instead, we can use the teleport services to move the turtlebot. Using autocomplete, run the command

```
$ rosservice call /turtle1/teleport_absolute <TAB><TAB>
```

ros should automatically give you the service syntax. Teleport the turtle to the coordinates (x,y,theta) coordinates (5,5,2).

Try calling the teleport_relative service. What is the difference?

Can you figure out the set_pen service?

5. **Parameters:** Parameters are another important part of ROS. To find out which are available to us, run the command

```
$ rosparam list
```

you should see a list of parameters. The "/background_*" parameters belong to the turtlesim. Lets figure out what values these parameters currently hold, using the command (note the <colour> placeholder)

```
$ rosparam get /background_<colour>
```

what values did you get? These numbers represent the RGB colour value of the background. Lets try changing it using the command (note the range [0-255] means any number between those values)

```
$ rosparam set /background_<colour> [0-255]
```

what happened? Now try calling the "/reset" or "/clear" services.

Play with these parameters while subscribed to the color_sensor topic. Also try disabling the pen using the set_pen service. What happens?

Exercise 2 – Your First ROS Package

We have reached the limits of the what can be done using the command-line interface on the turtlesim. To do more complex operations, we will need to develop our own packages and nodes to interface with the simulator.

1. **Create a Workspace:** The first step in writing ROS code is to create a catkin workspace. Catkin is the official build system for ROS, it is a complicated tool that adds functionality over standard CMake. Details of catkin are beyond the scope of this lab, but more information can be found here: http://wiki.ros.org/catkin/conceptual_overview

   To create a catkin workspace, run the commands

   ```
   $ mkdir -p ~/catkin_ws/src
   $ cd ~/catkin_ws/
   $ catkin_make
   ```

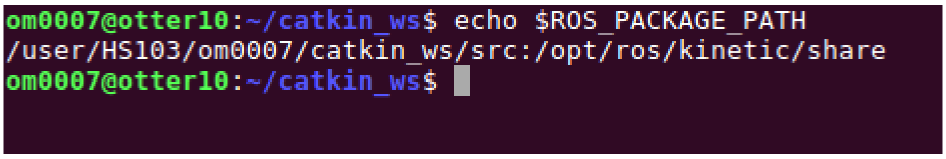
   "mkdir" and "cd" are standard unix commands, make sure you understand what they are doing. "catkin_make" is a the default method for compiling packages in ROS. It can also be used to initialise a workspace, like we have here. Inspect the files and folders that have been created, "catkin_make" will create a CMakeLists.txt file in your src folder, as well as a build and devel folder.
2. **Source your Workspace:** For ROS to find the packages/nodes in your workspace, we need to source it in the same way we sourced ROS in the last lab. To do this, run the command

   ```
   $ source ~/catkin_ws/devel/setup.bash
   ```

   you should also add this to your ".bashrc" file **after** the line that sources ROS. Once you have done this, open a new terminator tab/space and run the command

   ```
   $ echo $ROS_PACKAGE_PATH
   ```

   and ensure that your new workspace is included, it should look something like

   ```
   om0007@otter10:~/catkin_ws$ echo $ROS_PACKAGE_PATH
   /user/HS103/om0007/catkin_ws/src:/opt/ros/kinetic/share
   om0007@otter10:~/catkin_ws$
   ```

   NOTE: if you don't add this to your ~/.bashrc, your terminals won't know about your workspace! This will make the next parts in this lab more difficult.

3. **Create a Package:** Run the command

   ```
   $ cd ~/catkin_ws/src
   ```

   to go into your src folder. From here we can create a catkin package using the command

   ```
   $ catkin_create_pkg lab2 std_msgs rospy roscpp
   ```

   this will create a package called *lab2* with dependencies on *std_msgs, rospy* and *roscpp*.

The syntax for the command is:
catkin_create_pkg <package_name> [depend1] [depend2] [depend3].
go back to the root of your workspace by running

```
$ cd ~/catkin_ws
```

now compile your workspace using

```
$ catkin_make
```

note that "catkin_make" can only be called from the root of your workspace. Calling "catkin_make" from anywhere else will cause an error.

4. **Add C++ Code:** Go to your new package using

```
$ roscd lab2
```

inspect this path. The "catkin_create_pkg" script created several files. The "package.xml" file contains all the information ROS needs to know about your package. This includes name, version, maintainer, etc. More importantly, it contains a list of dependencies (both build and runtime). The "CMakelists.txt" file is essentially a standard CMake that contains all the information required by catkin to compile your package.

Take some time to read through these files, they contain important information.

Once you have familiarised yourself with the contents of this file, we can add a new node. First, cd into the "src" directory in your package. Once there, create a file called "pose_listener.cpp" by running the command

```
$ gedit pose_listener.cpp
```

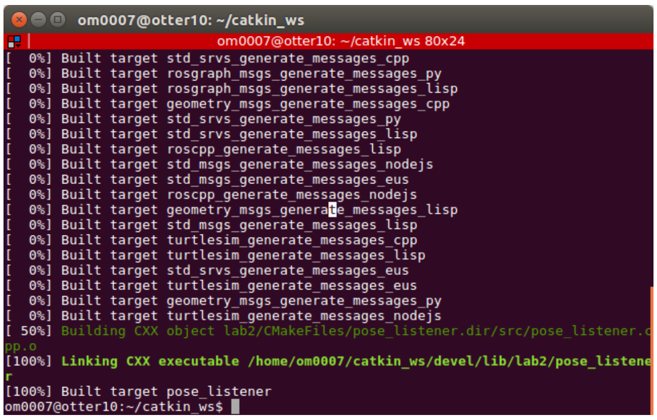
then paste the following code into it

```cpp
#include <ros/ros.h>
#include <turtlesim/Pose.h>

void turtle_pose_callback(const turtlesim::Pose& msg){
        ROS_INFO("X=%f",msg.x);
        ROS_INFO_STREAM("Y="<<msg.y);
        ROS_INFO_STREAM("T="<<msg.theta);
}

int main(int argc, char **argv){
  ros::init(argc, argv, "pose_subscriber");
  ros::NodeHandle nh;
  ros::Subscriber sub = nh.subscribe("turtle1/pose",10, &turtle_pose_callback);
  ros::spin();
}
```

save the file. Now open the CMakeLists.txt file and replace the contents with

```
cmake_minimum_required(VERSION 2.8.3)
project(lab2)

add_compile_options(-std=c++11)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
)

catkin_package()

include_directories(
  ${catkin_INCLUDE_DIRS}
)

add_executable(pose_listener src/pose_listener.cpp)
add_dependencies(pose_listener
 ${${PROJECT_NAME}_EXPORTED_TARGETS}
 ${catkin_EXPORTED_TARGETS})
target_link_libraries(pose_listener
  ${catkin_LIBRARIES}
)
```

save the file, go back to the root of your workspace and compile it (using catkin_make). If your node compiled successfully, you should see output like this



```
om0007@otter10: ~/catkin_ws
om0007@otter10: ~/catkin_ws 80x24
[  0%] Built target std_srvs_generate_messages_cpp
[  0%] Built target rosgraph_msgs_generate_messages_py
[  0%] Built target rosgraph_msgs_generate_messages_lisp
[  0%] Built target geometry_msgs_generate_messages_cpp
[  0%] Built target std_srvs_generate_messages_py
[  0%] Built target std_srvs_generate_messages_lisp
[  0%] Built target roscpp_generate_messages_lisp
[  0%] Built target std_msgs_generate_messages_nodejs
[  0%] Built target std_msgs_generate_messages_eus
[  0%] Built target roscpp_generate_messages_nodejs
[  0%] Built target geometry_msgs_generate_messages_lisp
[  0%] Built target std_msgs_generate_messages_lisp
[  0%] Built target turtlesim_generate_messages_cpp
[  0%] Built target turtlesim_generate_messages_lisp
[  0%] Built target std_srvs_generate_messages_eus
[  0%] Built target turtlesim_generate_messages_eus
[  0%] Built target geometry_msgs_generate_messages_py
[  0%] Built target turtlesim_generate_messages_nodejs
[ 50%] Building CXX object lab2/CMakeFiles/pose_listener.dir/src/pose_listener.c
pp.o
[100%] Linking CXX executable /home/om0007/catkin_ws/devel/lib/lab2/pose_listene
r
[100%] Built target pose_listener
om0007@otter10:~/catkin_ws$ 
```

once you are satisfied that the code is compiled, we can try to run it.

5. **Run your Code**: Use “rosrun” to run your code. Use autocomplete to help.
6. **Add new Node:** Use the above example, as well as the lecture material, to add the following code as a new node

```
#include <ros/ros.h>
#include <turtlesim/Pose.h>
#include <geometry_msgs/PoseStamped.h>
#include <turtlesim/Pose.h>
#include <tf/transform_datatypes.h>

int main(int argc, char **argv) {
      ros::init(argc, argv, "pose_publisher");
      ros::NodeHandle nh;

      ros::Publisher pose_pub =
nh.advertise<geometry_msgs::PoseStamped>("pose_output", 1000);
      ros::Rate loop_rate(10);

      int count = 0;
      while (ros::ok()) {
            geometry_msgs::PoseStamped msg;

            msg.header.frame_id="map";
            msg.header.stamp=ros::Time::now();
            msg.header.seq=count++;

            msg.pose.orientation = tf::createQuaternionMsgFromYaw(0);
            msg.pose.position.x = 1;
            msg.pose.position.y = 1;

            pose_pub.publish(msg);
            ros::spinOnce();

            loop_rate.sleep();
      }
      return 0;

}
```

you should add "turtlesim" as a dependency. This will include in your CMakeLists.txt and your package.xml files. Make sure the new code compiles and runs.

Why do you need the turtlesim dependency?

7. **Add Python Node:** Make a scripts folder in your lab2 package. In that folder, make a text file called "pose_listener.py". Open the file and insert the following code

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

import rospy
from turtlesim.msg import Pose

def callback(data):
    print(data)

if __name__ == '__main__':
    rospy.init_node('pose_listener_python')

    rospy.Subscriber('/turtle1/pose', Pose, callback, queue_size=1)
    rospy.spin()
```

make the file executable by running

```
$ chmod +x pose_listener.py
```

go back to the root of your workspace and run catkin_make. You should now have a python node that listens to a turtlesim pose.

Exercise 3 – Turtlesim in RVIZ

In this exercise, you will take some of the basic output from the Turtlesim and convert it into topics that are visualisable in RVIZ. We will then set up RVIZ to visualise these topics.

1. **Write a new Node:** Using the code snippets above, write a node called *pose_rviz* that subscribes to "/turtle1/pose" and converts it to a geometry_msgs/PoseStamped Message. Make sure this is written in a new file called pose_rviz.cpp for c++ or pose_rviz.py for python. Use the callbacks below as a starting point:

   C++:

```cpp
#include "geometry_msgs/PoseStamped.h"
#include <tf/tf.h>
void turtle_pose_callback(const turtlesim::Pose& msg){
      geometry_msgs::PoseStamped pose_rviz;
      pose_rviz.header.frame_id="map";
      pose_rviz.header.stamp=ros::Time::now();
      pose_rviz.header.seq=0;
      pose_rviz.pose.orientation =
tf::createQuaternionMsgFromYaw(msg.theta);
      pose_rviz.pose.position.x = msg.x;
      pose_rviz.pose.position.y = msg.y;
}
```

   Python:

```python
from geometry_msgs.msg import PoseStamped, Quaternion
msg = PoseStamped()
def callback(data):
    msg.header.frame_id = "map"
    msg.header.seq = 0
    msg.header.stamp = rospy.Time.now()
    q = tf.transformations.quaternion_from_euler(0, 0, data.theta)
    msg.pose.orientation = Quaternion(q[0],q[1],q[2],q[3])
    msg.pose.position.x = data.x
    msg.pose.position.y = data.y
```

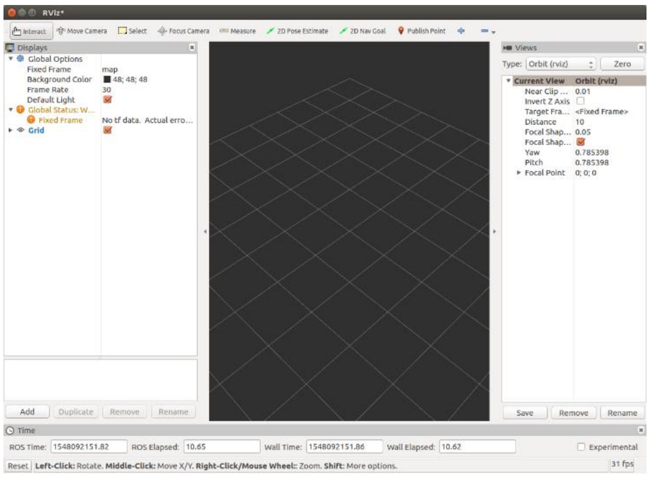
   You will need to add a new dependency on "tf".

2. **Publish the Pose:** Edit your node to also publish the pose after it has been converted (HINT: Either use a global publisher, or wrap your code in a class).

3. **Display in RVIZ:** With your node and turtlesim running, open RVIZ using the command

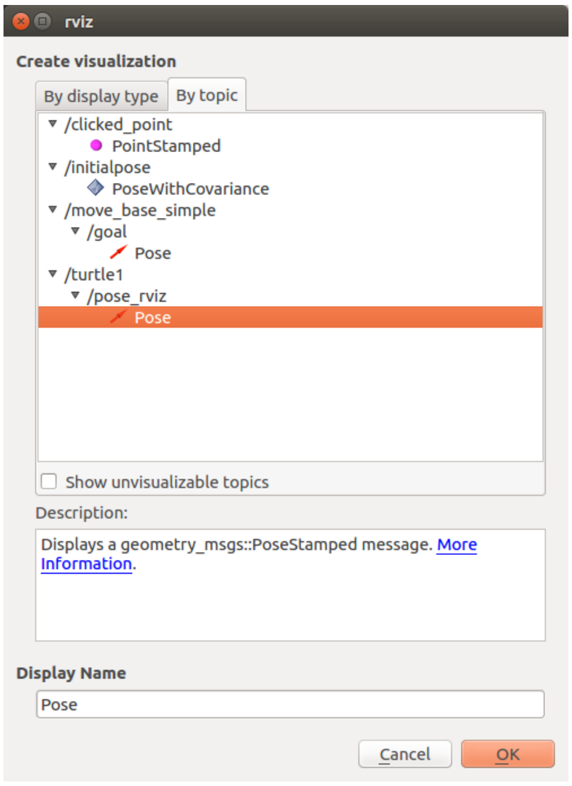
```
$ rosrun rviz rviz
```

you should get an RVIZ window like this



```
Global Options
  Fixed Frame        map
  Background Color   48; 48; 48
  Frame Rate         30
  Default Light      ✓
Global Status: W...
  Fixed Frame        No tf data.  Actual erro...
Grid                 ✓

Type: Orbit (rviz)   Zero
Current View   Orbit (rviz)
  Near Clip ...  0.01
  Invert Z Axis  ☐
  Target Fra...  <Fixed Frame>
  Distance       10
  Focal Shap...  0.05
  Focal Shap...  ✓
  Yaw            0.785398
  Pitch          0.785398
  Focal Point    0; 0; 0

ROS Time: 1548092151.82  ROS Elapsed: 10.65  Wall Time: 1548092151.86  Wall Elapsed: 10.62  ☐ Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   31 fps
```

click on the bottom-left button "add", then go to by topic and select the message you've republished from the list, as shown below



```
▼ /clicked_point
      PointStamped
▼ /initialpose
      PoseWithCovariance
▼ /move_base_simple
   ▼ /goal
        Pose
▼ /turtle1
   ▼ /pose_rviz
        Pose
☐ Show unvisualizable topics
```

Description:

Displays a geometry_msgs::PoseStamped message. More Information.

Display Name

Pose

click ok. The pose of the turtle should now be visible as a red arrow. Try driving the turtle around and see the arrow change. **HINT: If you can't see the arrow, make sure that the "Fixed Frame" on the top-left of RVIZ matches the frame_id in the header file of your message.**

Exercise 4 – Putting it all Together (HOMEWORK)

In this exercise, you will take everything you have learned so far (in labs and lectures) and put it together to make the Turtlesim perform more complex behaviours. This is a more complex exercise, so don't worry if you don't finish it in the allocated time, but try to finish it before next week.

1. **Create a new Package:** Make sure it includes all the dependencies you require for interacting with the Turtlesim, the geometry_msgs package and RVIZ.
2. **Create a new Launch File:** Use the lecture content to create a new launch file that starts the turtlesim simulator and spawns an extra turtle by calling the "/spawn" service. Use the following example to see how this service is called from launch files

```
<node pkg="rosservice" type="rosservice" name="spawn"
args="call spawn --wait 5.5 5.5 0 'turtle2'"/>
```

   and look at this link http://docs.ros.org/melodic/api/turtlesim/html/srv/Spawn.html to understand the parameters being used.
3. **Create a new Node:** This node should subscribe to the "/turtle1/cmd_vel" topic, invert the commands (multiply by -1) and publish them to the second turtle ("/turtle2/cmd_vel"). You can write this node in either Python or C++.
4. **Add Node To Launch File:** Add this node to your launch file. Make sure you can start/stop the simulator and your node reliably.
5. **Create a new Node:** Use the code snippet below

```
from turtlesim.srv import TeleportAbsolute, Spawn

spawn_client = rospy.ServiceProxy('/spawn', Spawn)
spawn_client(x_pos, y_pos, theta, "turtle3")

teleport_client = rospy.ServiceProxy('/turtle3/teleport_absolute', TeleportAbsolute)
teleport_client(x_pos, y_pos, theta)
```

   to create a new node that spawns a new turlte (in code) and teleports it around randomly at 1 Hz. Add it to the launch file. **Note: we are using rospy's service interface.**
6. **Have fun!** See what you can make the turtles do in code! Experiment with teleporting to other turtle's positions, following with noise, changing pen colour, etc.