Due in part by 11:59PM on Thursday, November 2 and in part at the beginning of class on Friday, November 3. (Please see the notes below!)

- The code for this assignment should be submitted electronically using the `submit395` script (see below) by 11:59pm on Thursday, Nov. 2, and the required printouts should be turned in at the beginning of class on Friday, Nov. 3.

- Collaboration is restricted for this assignment. You should use only the ROS wiki pages and the texts *A Gentle Introduction To ROS* (by O’Kane) and *Programming Robots with ROS* (by Quigley, Gerkey, and Smart) as outside resources relating to ROS; you are not permitted to share code with classmates or look at the code of a classmate. (You may also use any Python resources for help with coding, but nothing else that is ROS-related.)

**Programming Assignment: Topics and Nodes!**

For the below, please recall that to run the turtlebot teleoperation, the command is

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

and to run the turtlesim, the command is

```
rosrun turtlesim turtlesim_node
```

The purpose of this assignment is to give you practice working with ROS topics and creating your own ROS nodes. Most of it will focus on you writing “translator” nodes that will translate messages published by the turtlebot keyboard teleop nodes so that they can be understood by the turtlesim node. (Please keep in mind that `turtlebot` refers to the actual robot—or the 3D simulation of a robot in Gazebo—and `turtlesim` refers to the simple 2D simulation of turtles moving in a window on your screen.)

1. Finish working through the ROS tutorials listed for the October 25 class meeting on the Lectures page from our course website.

2. The October 25 entry in our course Lectures page also contains a link to a TwistToTurtle.py function (with thanks to colleagues from Union College!) that is intended to translate messages from the teleoperation of a turtlebot (robot) to messages for a turtlesim (simulated turtle). For the purposes of our CS395 class, it is good code to learn from, but it does not work for ROS Kinetic.

   For this exercise, you’ll modify that code so that it works for ROS Kinetic. You should:

   (a) Make a new catkin workspace for this HW assignment. In principle, it is possible to re-use the one you made for the tutorials, but as part of the assignment, please make a new one.

   (b) Make a new package containing your translator node.
(c) Modify the provided code so that it does the intended translation, and build the package containing your code so that your translator node can be run.

(d) Test your translator node: run it, a turtlebot teleoperation node, and a turtlesim node together to see the translation work!

As part of this exercise, you will need to find the relevant topics and message types, subscribe and publish to the appropriate topics, and send the appropriate messages to do the translation.

3. Next, create a new package with a bad translator node! For this node, whenever the keyboard teleop is telling the robot to move straight ahead without turning, the translation node should make the turtlesim turtle veer off at a random angle instead. Although there are several ways you might accomplish this, for this assignment, you should:

(a) Create a new package in your catkin workspace for this HW. In principle, it is possible to re-use the one you made for the tutorials, but as part of the assignment, please make a new one.

(b) Write code that will do the following:
   • Take in a message from the turtlebot teleop node
   • Check if the message is a “move straight ahead” message
   • If the received message is not a “move straight ahead” message, publish a correctly translated message to the turtlesim (i.e., moving and steering the turtlesim exactly as it would steer the turtlebot)
   • If the received message is a “move straight ahead” message, use a random number generator in Python to create an amount by which the turtlesim should be steered away from the “straight ahead” direction, and use that to make a message to publish to the turtlesim. The new / modified message should move the turtlesim forward as much as in the “move straight ahead” message, but it should also steer it left or right by the randomly generated amount.

   The use of a random number generator means that the steering can be differently affected every time a “move straight ahead” message is received from the turtlebot teleop. Be sure that these “mistranslated” messages could randomly steer the turtlesim in either direction, left or right, off its “straight ahead” bearing!

(c) Build the package containing your code and test it, as above.

Therefore, every time a message is received from the turtlebot teleop, a message should be published to the turtlesim, as in the previous exercise; for this exercise, however, if the received message is a “move straight ahead” message, the published message should contain modified steering information to make the turtlesim move forward-but-not-quite-straight-ahead.

Something like this could be done in a simpler way, but the point of this exercise is to practice analyzing a received message and publishing a message with new, modified information.
4. Next, create a new package with a node that uses bad translation to control **two different turtlesims** (in two different windows) with the same keystroke from a turtlebot teleop node! Your single ROS node should act somewhat like the bad translator node written for exercise 3, above, with some modifications. It should:

- Take in a message from the turtlebot teleop node (as in exercise 3)
- Check if the message is a “move straight ahead” message (as in exercise 3)
- If the received message is **not** a “move straight ahead” message, create **two** new messages—one for each turtlesim—and send a message to each turtlesim, moving and steering it exactly as it would move the turtlebot (this is just as in exercise 3, except for publishing to two different turtlesims)
- If the received message is a “move straight ahead” message, create **two** new messages, one for each of the turtlesims: one turtlesim should get the same kind of “bad translator” message as in exercise 3, telling that turtlesim to move forward-but-not-quite-straight-ahead, with the direction determined by a random number generator in Python; the other turtlesim should get a message identical to that of the first turtlesim except steering it in the “mirror opposite” direction. That is, if one turtlesim turns left at a certain angle, the other should turn right by exactly that angle. (More abstractly, if the first turtlesim gets a message steering it an angle $\theta$ off of the “straight ahead” direction of the turtle, then the second turtlesim should get a message steering it $-\theta$ off of the “straight ahead” direction of the turtle.)

(You also need to build the package and test it, as above.) Something like this could be done with multiple ROS nodes, but the point of this exercise is to have one node control both turtlesims to do non-identical things.

5. Finally, submit a printout containing some documentation accompanying your work for exercises 2-4 above. The purpose of this documentation is to briefly explain important features of your work and how to run your code in context, as well as to include your code itself. Therefore, the printout should include, for each of exercises 2-4:

- A brief explanation of the approach you took to solve the problems in the exercise. *Note:* If you already provide a thorough explanation as part of comments in your code, you do not need to include a separate one for this documentation; including your code in the printout will suffice.
- Documentation of how to run and test your code. That is, submit a list of all the command-line ROS commands one would need to run in order to demonstrate how your code works. For example, for exercise 2, `roslaunch turtlebot_teleop keyboard_teleop.launch` and `rosrun turtlesim turtlesim_node` are two of the commands needed to test your code; your list should include those and any others needed to demonstrate your work (e.g., `rosrun exercise2 translator_node.py`).
- Your Python code for that exercise. You need not print out any other elements of the ROS packages, just the Python file you wrote for each exercise.
Notes and Submission Instructions

Please submit the catkin workspace for these exercises using the `submit395` command:

```
submit395 hw2 <your-hw2-dir>
```

Common guidelines for good style apply to your code and your submission, including the following:

- Make sure all code is properly commented, with a *contract* for each function.
- For readability, all lines of code should be at most 80 characters. Also, please use whitespace appropriately.
- The submitted directory can contain all files created by catkin building the packages for this HW—and it may also contain a text or PDF file of your documentation—but it should not contain extraneous or irrelevant files.

Code that is not easy to read and test as submitted may not merit full credit. If there are questions about code style or readability, please ask your Prof.!