Foundations of Robotics, Fall 2017

Coding Homework 1: Introduction to ROS
(CS 4750: 100 points, CS 5750: 120 points)

Due at the start of class, Wed, 6 Sep

Objective:

Become familiar with ROS and the Python interface to ROS. Demonstrate the ability to use ROS topics, publishers, subscribers, and services effectively.

Instructions:

This assignment can be discussed as a group of arbitrary size. You may work with up to one partner on this assignment, but each student or pair of students is responsible for writing and submitting their own solutions to the problems below. Include in a comment at the top of each file your name and the names of all classmates you discussed any part of the homework with.

Getting Started:

Officially, this course only supports Ubuntu 16.04, ROS Lunar, and Python 3.6. Unofficially, you may use whatever OS you like, provided that you accept that you alone will be responsible for getting the necessary software running on whatever system you choose, and the course staff bears no obligation to assist with debugging issues encountered on systems other than the course-provided Ubuntu 16.04 VMs[1].

We have provisioned a VM running the course software for each of you. Your submissions are expected to run on this VM image for the purpose of grading. You should already have been given instructions on how to access and use your personal VM. If you cannot access your VM or do not know how to use it, please contact the course staff. If you're new to Ubuntu, we recommend that you take a look at this introduction to Linux package management — specifically, the Ubuntu sections of the tables.

You will need to write Python code using rospy to solve the problems in this and future assignments. If you need a refresher for Python, we suggest this quick reference. If you feel the need for something more in-depth, please contact the course staff.

If you would like to use another language (e.g. MATLAB, C++, etc.) to interface with ROS, you are welcome to do so. However, (a) **the course staff do not provide support for any language other than Python** and (b) **your grade will be determined entirely by how your submitted code performs**. (b) means that if we cannot run your code, or if your code does not function correctly, you cannot get any partial credit.

To make it easier to work with ROS, we have created a simple command-line tool for you to use to fetch and build framework code and run your code for each problem. You can download this tool from the course website. It requires Python 3.6 or greater to run.

[1]Wil can also assist with getting things running on Arch Linux, however.

If you wish to run the course code on your personal computer, you will need to install the following dependencies:

- CMake: Necessary for the compilation of our framework code.
- Python 3: Necessary for the use of our provided simulator wrapper tool.
- GNU make (or ninja-build): The build system used as a backend by CMake.
- `wget`: To download provided source code files.
- Pip for Python 3: To install necessary Python modules.
- ROS Lunar, including rospy: To communicate with the simulator.

If you experience difficulty installing these dependencies, please contact the course staff for assistance.

The simulator tool provides several commands:

- `get <assignment name>`: Downloads the current version of the provided code for the specified assignment. For example, to get the code for this assignment, you would run `./simulator-tool get hw1`.
- `build <assignment name>`: Compiles the provided framework code for the specified assignment. For example, to build this assignment, you would run `./simulator-tool build hw1`.
- `run <assignment name> <problem>`: Runs the simulator and framework code for the specified assignment and problem. For example, to run the code for problem 1 on this assignment, you would run `./simulator-tool run hw1 p1`. **Make sure you have manually run `source devel/setup.bash` in your shell before using this command**.

Note that these commands will ensure that their dependencies are in place automatically, e.g. executing `./simulator-tool run hw1 p1` before you have downloaded the code for Homework 1 will automatically download and build the necessary code before running the code for problem 1.

If you get a "permission denied" error when trying to run `./simulator-tool`, please ensure that the file is marked as executable by running `chmod u+x simulator-tool`. **Please also be sure to run the simulator tool from the root of your `catkin` workspace; it will not run correctly otherwise.**

If you get errors about the launch file for a certain problem not existing, it may be that the framework code does not provide any programs for that problem. Please look in the `launch` subdirectory of the framework code (e.g. `src/foundations_hw1/launch`) to see which problems have provided code.

If you encounter difficulty using the simulator tool, please contact the course staff. Note that you can also run `./simulator-tool --help` to get a listing of commands and `./simulator-tool <command> --help` to get more information on the usage of a specific command.

Assignment:

Implement code to solve the following problems, using the provided simulator framework. Start by making a `catkin` workspace for your code. Create a new ROS package for this project in the workspace (call it "hw1"); write your code for each problem in a corresponding file in the package (named according to the convention "pX.py", where X is the problem number). Submit the complete ROS package on CMS in a single .tar.gz archive, named "solutions.tar.gz".

If you need a reference for ROS, we suggest the ROS "Getting Started" guide, the rospy documentation, and Jason O'Kane's "A Gentle Introduction to ROS"[2]

1 Topics

This problem is designed to assess your understanding of and ability to use the ROS command-line tools for inspecting topics, nodes, and services. You do not need to turn in any code for this problem; you must turn in a text file with output and explanations.

(a) List the default topics and explain their purposes. (*3 points*)

(b) Run `rosrun turtlesim turtlesim_node`. What topics were added? What information do they contain? (*3 points*)

(c) Give and explain the message types of `/turtle1/cmd_vel` and `/turtle1/pose`. (*3 points*)

(d) How many subscribers and publishers are there for `/turtle1/cmd_vel`? How many for `/turtle1/pose`? (*3 points*)

(e) What services are available? What service would you use to move the first turtle instantly to a set of coordinates? (*3 points*)

(f) What command would you use to find all services which use a `geometry_msgs/Twist` type? (*5 points*)

2 Publishers and Subscribers

This problem is designed to assess your understanding of and ability to use publishers and subscribers in ROS.

(a) Write a node to make the turtle simulator move. Assuming that `rosrun turtlesim turtlesim_node` is running, use `/turtle1/cmd_vel` to make the turtle's final position different from its starting position. (*5 points*)

(b) Re-implement the basic functionality of the `rostopic echo` command. That is, write a node that takes a topic name and the topic's type as parameters and outputs the messages published to that topic. You may assume that the topic type will be given as a string, and will be restricted to one of `std_msgs/Float64`, `std_msgs/Int64MultiArray`, `std_msgs/String`, `geometry_msgs/Point`, or `geometry_msgs/Twist`. (*10 points*)

[2]Note that AGITR uses C++ instead of Python. The concepts will be the same, but the language used is different.

(c) You need to make the turtle track a target. Subscribe to the topic `/hw1/target_loc` and use the messages to keep the turtle always moving toward its target. To test your code, run `./simulator-tool run hw1 p2c` *after* your node has been started. (*10 points*)

(d) Use the available turtle topics to write a node to draw a heptagon with the turtle. (*5 points*)

3 Services

This problem is designed to assess your understanding of and ability to use services in ROS, particularly as they are distinguished from the publisher/subscriber model.

(a) Implement a node which spawns four turtles[3] and draws a separate square with each turtle. The squares should not intersect or touch, and should be regularly spaced (i.e. the amount of space between the perimeters of any two separate squares should be the same). (Hint: Look at the existing `turtlesim` services to figure out how to spawn a turtle) (*10 points*)

(b) We have a node which creates a random number of turtles which must navigate the environment without being "caught" by a special chasing turtle. The chasing turtle will always move toward the nearest turtle to it; the turtles being chased can move to a given point. Write a service called `/escape` which provides the turtles being chased with safe points to move to in order to escape the chasing turtle. Your service should use the `Escape.srv` service type defined in the provided code. (*10 points*)

4 Timing

This problem is designed to assess your understanding of and ability to use various rate control and timing options in ROS.

(a) Make a node that publishes random velocities for the turtle at a fixed rate, which will be passed in as a command line argument. You may not use `sleep` or any explicit loops (i.e. no `while`, `for`, etc.) (*8 points*)

(b) Write a node that subscribes to `/turtle1/image_sensor`. This topic publishes values of an "image sensor" every time the turtle moves. The image sensor returns images as 2D arrays of pixel intensity readings. Make a publisher which, every $\frac{1}{20}$ of a second, computes the maximum sub-array value for each sensor reading. Make an array of the maximum sub-array values, and find the indices of the maximum sub-array within this array. Publishes these indices as a `std_msgs/UInt32MultiArray` message. It is very important that your publisher keeps to the correct frequency. (*6 points*)

(c) Make a node which subscribes to `/pub_rate` and uses the values it receives to determine the publishing rate of a simple "ping" message publishing on `/ping`. (*6 points*)

[3] If you're not sure what to name the turtles, we suggest drawing inspiration from popular culture.

5 Multiple Nodes

This problem is designed to assess your ability to use multiple nodes to process information in a ROS system.

(a) Make a node to draw a square with a turtle, parameterized on the location of the top left corner. Use a launch file to draw a chain (overlapping squares with the top left corner of a square at the center of the square to its left) with 8 links. (*10 points*)

(b) Make one node which spawns a turtle and moves it randomly in the left side of the environment. Make a second node which mirrors the movements of the first on the right side of the environment. Use a launch file to start both nodes simultaneously. (*10 points*)

(c) Make a node which controls a turtle to search the space for the maximum value of a function. Use the service `/reward` (provided in the framework code) to check for the value at a point. Create a custom message type to publish information about the gradient at each point you explore. Use a launch file to start several simultaneous searcher nodes starting from different points in the environment. (*10 points*)