Foundations of Robotics, Fall 2018

Coding Project 1: Introduction to ROS
(CS 4750: 95 points, CS 5750: 100 points)

Due Friday, Sept. 7 at 3:00 PM

Objective:

Become familiar with ROS, the Python interface to ROS, and numpy. Demonstrate the ability to use ROS topics, publishers, subscribers, and services effectively, and to perform simple linear algebra tasks in numpy.

Documentation and References:

The primary purpose of this coding project is to help you learn to use several libraries that you will use in future assignments. 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[1]. If you feel the need for something more in-depth, please contact the course staff. You should also use the numpy library for matrix calculations.

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].

Collaboration Policy:

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 is responsible for writing and submitting a separate solution, and **no student should see another student's solution**. Include in a comment at the top of each file your name and the names of all classmates you discussed any part of the project with.

Getting Started:

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, email admin TA Julia Proft. If you are unfamiliar with the Linux command-line interface (terminal), we recommend that you take a look at this command-line primer.

Making a Catkin Workspace:

Before starting the assignment, you need to set up a catkin workspace. To do this, run the following commands in a terminal window. (The terminal application is pinned to the application dock.)

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws
catkin build
```

[1] For compatibility purposes, we use Python 2 rather than Python 3 in this class.

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

These commands (1) make a directory in your home folder called `catkin_ws` that contains an empty directory named `src`, (2) change the current directory to the new `catkin_ws` directory, and (3) initialize your catkin workspace.

Getting the Simulator Tool:

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. To get the simulator tool, run the following commands in your terminal:

```
cd ~/catkin_ws
wget https://rpal.cs.cornell.edu/foundations/hw/simulator-tool
chmod u+x simulator-tool
```

These commands (1) change the current directory to your catkin workspace, (2) download the simulator tool from our website into the current directory, and (3) make the simulator tool executable.

Using the Simulator Tool:

The simulator tool provides several commands. Before running the commands given below, run `cd ~/catkin_ws` to change the current directory to your catkin workspace.

- `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 p1`.
- `build <assignment name>`: Compiles the provided framework code for the specified assignment. For example, to build this assignment, you would run `./simulator-tool build p1`.
- `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 p1 problem1`. **Make sure you have manually run `source devel/setup.bash` in each terminal window where you run this command**.
- `check <assignment name>`: Verifies that your submission has the necessary files to be graded correctly. If this command returns errors or warnings, you will lose points on the assignment. **You are strongly advised to run it before submission.**
- `package <assignment name>`: Compresses your `catkin_ws/src` directory for submission. **You must manually submit this package to CMS.**

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

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. When you're finished, compress your complete ROS package with the command `./simulator-tool package p1`. Then upload the compressed package (named `p1_solutions.tar.gz`) to CMS. Note that you can upload to CMS as many times as you'd like up until the deadline; the most recent submission will override all the previous ones. Keep this in mind and don't wait until the last minute to submit! We recommend uploading at least a half hour before the deadline to ensure that your solutions are submitted on time.

0 Written

(8 points) This problem is designed to assess your understanding of and ability to use the ROS command-line tools for inspecting topics, nodes, and services. In order to submit your answers to this part of the assignment please go to Coding Project 1 Quiz on CMS. You can complete this problem before you begin any of the coding in this assignment.

1 Topics

(20 points) This problem is designed to familiarize you with ROS and get you started with turtlesim. The goal of this problem is to have you learn how to move a turtle using Twist messages and calculate the distance between turtle and a point.

The work for this problem should be done in file `problem1.py`. Look for the comments throughout the code that correspond to the instructions in the writeup. Uncomment the given template code and replace `FILL IN` with the appropriate code.

(a) In `__init__`:

  i. First, have your turtle subscribe to `turtle1/pose` with the callback function `self.update_pose` in the indicated section of the code.
  ii. Have your turtle publish to `turtle1/cmd_vel`.
  iii. Specify a publishing rate of 10.

(b) In `move`:

  i. Create a Twist message with values that change both the angular and linear velocities such that the turtle moves both angularly and linearly.
  ii. Publish the Twist message to the velocity publisher.

(c) In `update_pose`:

  i. Set the instance variable `self.pose` to the information provided by the subscriber.

(d) In `calculate_distance`:

  i. Using numpy, set `vector` to be the difference between the coordinates of the turtle's current pose and the point (x, y). (Hint: Use numpy array for your vector)
  ii. Using numpy and `vector`, set `distance` to the distance between the turtle and the point (x, y).
  iii. Set the ROS parameter to `distance`. (Note: the value of distance must be cast to a float).

(e) In `problem1.launch`:

  i. Review the format of this file and understand what it does (see ROS launch file documentation for more information).

2 Single Flower

(25 points) This problem is designed to have you create a node and generate specific behavior for your turtle using numpy commands. Your turtle's objective is to create a single flower pattern. See Figure 1 for exactly what the flower should look like.

The work for this problem should be done in file `problem2.py`. Look for the comments throughout the code that correspond to the instructions in the writeup.

(a) In `__init__`:

  i. Start a new node with the name 'flower'.
  ii. Create a publisher that publishes to `/turtle1/cmd_vel` with type `Twist` and a queue size of 10.
  iii. Set your rate to 5.

(b) In `draw_flower`:

  i. To calculate constants, use numpy to perform the following operations on the following matrices:

$$M_1 = \begin{bmatrix} 1.5 & -1.5 & 0 \\ -1.5 & 0 & 2 \\ 2.5 & 0 & 0 \end{bmatrix} \; M_2 = \begin{bmatrix} 3 & 5 & 9 \\ -1 & 2 & 6 \\ -3 & 2 & 0 \end{bmatrix}$$

$$a = (M_1 * M_2)_{(0,1)} + (M_1 * M_2)_{(1,1)}$$

$$b = -(\langle M_1, M_2 \rangle_{(0,2)} + \langle M_1, M_2 \rangle_{(1,2)}) - 1$$

$$A = \langle M_1, M_2 \rangle_{(0,1)} - \langle M_1, M_2 \rangle_{(0,0)} + 0.5$$

$$B = -((M_2 * M_1)_{(1,2)} - (M_2 * M_1)_{(2,1)})$$

$$\texttt{max\_times} = (\texttt{det}(M_2)/-9) - 4$$

  Note: Angle brackets indicate taking an inner product, subscripts indicate the indices of the term that should be taken from a matrix

  ii. Each of the above computed constants should be set as rosparams `a`, `b`, `A`, `B`, and `max_times` with the type `float`.
  iii. In `while not rospy.is_shutdown()` section: Use numpy to set angular z to $B * \cos(b * count)$ and linear x to $A * \sin(a * count)$. Be sure to publish your velocity message and sleep your turtle for your preset rate.
  iv. Also in `while not rospy.is_shutdown()` section: If value *count* is greater than $2 * (\texttt{times}) * \pi$, publish a velocity message with angular $z = 5$ and linear $x = 0$, then increment `times` by 1. If `times` is greater than `max_times` stop drawing the flower.

(c) In `if __name__ == '__main__'` section:

  Create your turtle and have the turtle use your written function to draw a flower. Your final product should look something like Figure 1.

(d) In `problem2.launch`:

  i. Launch the turtlesim node.
  ii. Launch `flower_turtle` node from `problem2.py` in package `p1`.

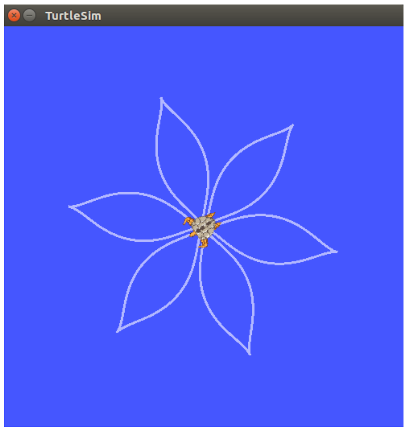

Figure 1: For Problem 2. Single Flower

3 Service Clients

(20 points) This problem is designed to introduce the client-service structure. For this question you will be using our implemented service and writing your own client to create a slightly different single flower then before. See Figure 2 for what the flower should look like.

The work for this problem should be done in file `problem3.py`. Look for the comments throughout the code that correspond to the instructions in the writeup.

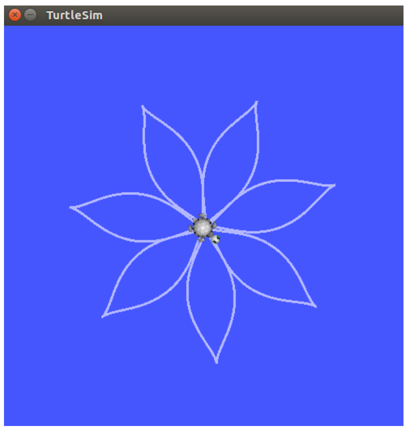

Figure 2: For Problem 3. Service Clients

(a) In `draw_flower`:

  i. Initialize your node with the name `drawing_turtle`.

  ii. Wait for the service `draw`.

(b) In `try` section: Fill in service handle for the draw service with `draw` and `Draw`. Check `srv/Draw.srv` and rospy documentation for more information.

(c) After the `try` section: Fill in with a publisher that publishes to `/turtle1/cmd_vel` with type `Twist` and queue size of 10 then set your rate to 5 (much like in 3a).

(d) In `while not rospy.is_shutdown()` section:

   i. Use the draw service to get the velocity message to publish. Look at `draw.srv` to determine the parameters and appropriate return name.

   ii. Sleep for the previously stated rate.

(e) In `if __name__ == '__main__'`: Call your function to draw the flower!

(f) In `problem3.launch`:

   i. Launch the turtlesim node.

   ii. Launch `draw_service` node from `draw` in package `p1`.

   iii. Launch `draw_flower` node from `problem3.py` in package `p1`.

4 Services and Spawning

The first part of this problem is meant to introduce you to writing a service for the client-service structure.

4.1 Part one

(15 points) The work for this problem should be done in the file `problem4a.py`.

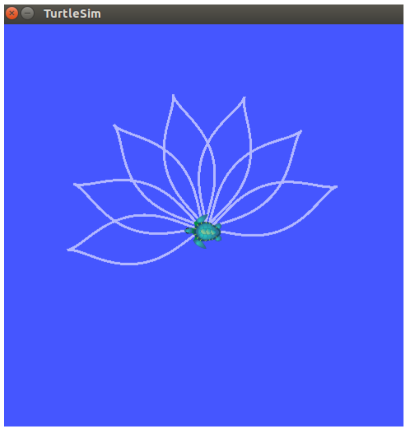

Figure 3: For Problem 4.1 Services and Spawning Part one

(a) In `draw`:

   i. Initialize `vel_msg`.

   ii. Set k to be the count parameter of `req`.

   iii. To calculate constants, use numpy to perform the following operations on the following matrices:

$$M_3 = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 5 & 8 \\ 13 & 21 & 34 \end{bmatrix} \quad M_4 = \begin{bmatrix} 1 & 8 & 7 \\ 2 & 5 & 3 \\ 9 & 2 & 6 \end{bmatrix}$$

$$a = det(M_3) - 1$$

$$b = -((M_3 * M_4)_{(2,0)} - (M_3 * M_4)_{(2,2)}) + 1$$

$$A = (\langle M_3, M_4\rangle^T_{(0,0)} - \langle M_3, M_4\rangle^T_{(1,0)})/10$$

$$B = -((M_3 * M_4)^T_{(2,1)} - \langle M_3, M_4\rangle^T_{(2,1)})/2.0$$

Note: Be sure not to perform integer division for `B`

iv. Each of the above computed constants should be set as rosparams `a`, `b`, `A`, and `B` with the type `float`.

(b) In `if req.rotate` section:

i. If `req.rotate`, set angular z to -3 and linear x to 0.

ii. Otherwise use numpy to set angular z to $B * cos(b * count)$ and linear x to $A * sin(a * count)$.

(c) Also be sure to return your velocity message before the end of the `draw` function.

(d) In `draw_server`: Create a service with the name `'draw'`, service_class `Draw` and handler `draw`. Please see rospy documentation for further details.

(e) In `__name__ == '__main__'`: Call your function!

(f) In `problem4a.launch`:

i. Launch the turtlesim node.

ii. Launch `draw_service` node from `problem4a.py` in package `p1`.

iii. Launch `draw_flower` node from `problem3.py` in package `p1`.

Your resulting flower should resemble Figure 3.

4.2 Part two

(12 points) The work for this problem should be done in file `problem4b.py`.

(a) In `draw_flower`:

i. Initialize your `'draw_turtle'` node .

ii. Block while waiting for the `'spawn'` service to become available. See rospy documentation for more information.

iii. In first `try`: Create a handle for calling the spawn service.

iv. In second `try`: Uncomment the provided lines of code

v. Block while waiting for `'/draw'` service, and then set up the service handle for the draw service (in the same way as problem 3).

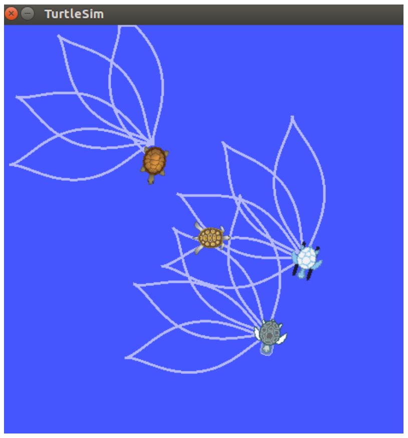

Figure 4: For Problem 4.1 Services and Spawning Part two

vi. Set up velocity publisher for `'/cmd_vel'.format(turtle_name)` and set the rate to 5.

(b) In `while not rospy.is_shutdown()` section of `draw_flower`: Publish the velocity messages returned from draw. Then sleep for predetermined rate.

(c) In `__name__ == '__main__'`: Call your function!

(d) In `problem4b.launch`:

i. Launch the turtlesim node.

ii. Launch `draw_service` node from `problem4a.py` in package `p1`.

iii. Launch `launcher` node from `launcher.py` in package `p1`.

Your resulting product should resemble Figure 4. There should be between 2 and 5 turtles spawned, in addition to the original turtle. The starting points of the turtles is random.