Homework 1

ME 416 - Prof. Tron

Monday 18th February, 2019

The goal of this homework is to get some practice with writing and running ROS nodes in Python, and to make the robot move. Please read the instructions carefully.

General instructions

The assignment is intended to be completed in pairs using *pair programming*. See the syllabus for details, and see Blackboard for the group assignments.

Homework report (required). Solve each problem, and then prepare a small PDF report containing comments on what you did for each problem, any particular difficulty you might have encountered, and whether you tackled the **optional** questions. You do not need to include your code here (unless you think it is necessary to make your discussion clear).

Demo video (required). The results shown in the video will be the first and foremost part of your submission that I will use for grading. You need to prepare a one to three minutes video with the following elements:

- A foreground framing of yourself (both members of the group) saying your names and the name(s) of the robot(s).
- For each question marked as **video**, a short segment showing the laptop and robot demonstrating that you completed the requested work. If you could not complete a question, you must explain what were the problems that you encountered.
- A final conclusion where you explain what you learned with the activity.

How to submit this homework. To facilitate grading, the submission of the homework will be split across Blackboard and Gradescope:

- **Gradescope**: Report. It is fine if you have multiple questions on each page, but please mark them correctly during submission.
- **Blackboard**: Video (through My Media on Blackboard, or a Dropbox/Google Drive link), Code (in a ZIP file), PDF copy of the report (this copy is only for records).

Grading. Each question is worth 1 point unless otherwise noted. Questions marked as **optional** are provided just to further your understanding of the subject, and not for credit. Questions marked as **provided** are already solved and included with the assignment. If you submit an answer I will correct it, but it will not count toward your grade. See the Syllabus on Blackboard for more detailed grading criteria.

Maximum possible score. The total points available for this assignment are 14.0 (13.5 from questions, plus 0.5 that you can only obtain with beauty points). Points for the video questions are assigned separately on Blackboard.

Hints Some hints are available for some questions, and can be found at the end of the assignment (you are encouraged to try to solve the questions without looking at the hints first). If you use these hints, please state so in your report (your grading will not change based on this information, but it is a useful feedback for me).

Use of external libraries and toolboxes All the problems can be solved using the basic ROS Python facilities. You are **not allowed** to use functions or scripts from external libraries or packages unless explicitly allowed.

Where to store your files You should store all the files that you will generate under the directory `~/ros_ws/src/me416_lab/nodes` on your robot. Use an SFTP client (e.g., Cyberduck), to copy the files, and compress them in a ZIP file before submitting your work.

Problem 1: Listener-talker node

As a starting point, this question will ask you to make a node that subscribes and publishes on a topic.

Question 1.1. Implement a node that combines the templates provided in the `talker.py` and `listener.py` files.

> File name: `talker_listener.py`
> Subscribes to topics: `chatter` (type `std_msgs/String`)
> Publishes to topics: `chatter` (type `std_msgs/String`)
> Description: Combine the scripts in the files `talker.py` and `listener.py`. The result should be a script for a single node that publishes regularly (1 Hz) a message on the topic `chatter` and prints on screen what it hears on the same topic.

You can solve Question 1.3 below in lieu of this one. Note: for this question, the node can "work alone", i.e., it will produce a continuous output even if only one instance is launched. However, you should be able to launch multiple instances at the same time; in this case, each instance would "hear" messages from every other instance.

Question 1.2. Modify your node so that it publishes something more interesting than the current time or a count (e.g., the verses of a poem, lines of a song, or the table of elements). Upon reaching the end, the messages should be repeated from the beginning.

Question 1.3 (optional). Same as Question 1.1, but instead of using the files `talker.py` and `listener.py`, use the files `talkerClass.py` and `listenerClass.py`. That is, the node from your new script should be wrapped in a Python class.

Question 1.4 (optional). Transform the `talker_listener.py` node into a `talker_listener_chatbot.py` node that *1)* publishes on the topic `/chatter` when it is started, prefixing the messages with the name of the node, *2)* replies every time something is heard on the topic `/chatter`, but *3)* does not reply to its own messages. You should get a "continous" stream of messages on the topic only if you launch two instances of the node.

Question 1.5 (video). Show the results after launching two nodes, both in the nodes' terminal windows, and also the output of the command `rostopic echo /chatter`. Ideally, you should use a screen recorder for this.

Problem 2: Motor command

In this question you will make

1) A function that transforms high-level desired robot velocities to actual motor speeds (file `motor_command_model.py`);
2) A node that uses this function to actually move the robot given ROS messages, and that publishes the speeds used (file `motor_command.py`).

Question 2.1. Create the following function

> File name: `motor_command_model.py`
> Function name: `twist_to_speeds`
> Description: Given the desired linear and angular velocity of the robot, returns normalized speeds for the left and right motors. Speeds are normalized to be between -1.0 (backward at maximum speed) and 1.0 (forward at maximum speed).
> Input arguments
>
> - `speed_linear` (type `float`) Linear speed of the robot (i.e., along the x-axis of the robot).
> - `speed_angular` (type `float`) Angular speed of the robot (i.e., around the z axis of the robot).
>
> Returns arguments
>
> - `left` (type `float`) The speed for the left motor.
> - `right` (type `float`) The speed for the right motor.

See Figure 1 for an illustration of the axes and robot velocities. In general, the inputs could specify a mix of linear and angular velocities; i.e., the desired trajectory of the robot could be an arc, not simply going forward/backward or rotating in place.

Note: For this question, think about how different combinations of motor speeds can affect the overall linear and angular velocity of the robot, but do not try to be too formal. For this assignment, just write something sensible; you will revise this function and make the model more formal in the next homework after discussing the robot kinematics in class.

Preparation . The next questions refer to the following node specification.

> File name: `motor_command.py`
> Subscribes to topics: `robot_twist` (type `geometry_msgs/Twist`)
> Publishes to topics: `motor_speeds` (type `me416_lab/MotorSpeedsStamped`)
> Description: See questions below for the expected behavior of the node.

Preparation (optional). Look at the script `scripts/zero_motors.py` as an example on how to set the motor speed.

Question 2.2. As initialization, the node should perform three steps:

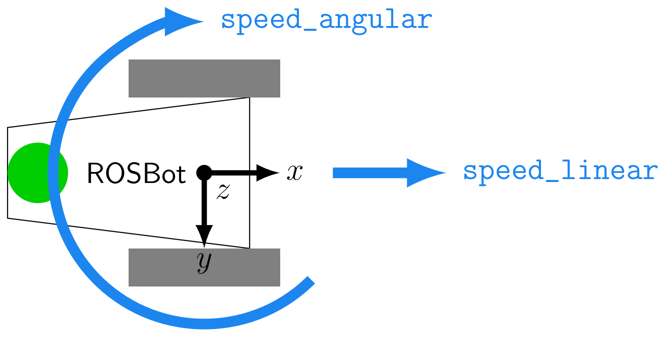

Figure 1: Axes on ROSBot to define linear and angular velocities.

1) Import the modules `motor_command_model` (from Question 2.1) and `me416_utilities` (provided in the repository).

2) Create two objects `motor_left` and `motor_right` from the classes `MotorSpeedLeft` and `MotorSpeedRight`, which are provided under the `me416_utilities` module.

3) Create a subscriber and a publisher for, respectively, the topics `robot_twist` and `motor_speeds`.

Note that the constructors of `MotorSpeedLeft` and `MotorSpeedRight` accept an optional argument `speed_offset` (which is intended to be between 0.0 and 1.0) that acts as a multiplier for the set velocity; for instance, if the set speed for a motor is 1.0, but `speed_offset` for that motor is 0.9, then the effective velocity of that motor will be 0.9 of its max speed. In Question 3.4 you might have to set this to make the robot go straight.

Question 2.3. In the callback function for the topic `robot_twist`, the node should transform the content of the `Twist` message into the speeds for the left and right motors using the function `twist_to_speed` from the module `motor_command_model`, and then set those speeds on the motors using the methods `.set_speed(speed)` of the objects `motor_left` and `motor_right`.

Question 2.4. In the same callback for the topic `robot_twist`, create a message `MotorSpeedsStamped`, fill in the time in the `header.stamp` attribute using `rospy.Time.now()` (see the file `talker_motor_speeds.py` from the in-class activity).

Question 2.5 (video). Run the `motor_command` node together with the node `zigzag_op` (provided in the repository). This should make the robot follow forward-moving arches.

Question 2.6 (video). Same as the previous question, but this time run also the command `rostopic echo /motor_speeds`. Make sure that the output of this command is visible in the video. For best results, use a screen recorder.

Problem 3: Keyboard teleoperation

In this question you will make a generic keyboard teleoperation node that translates key presses into messages that define the commanded linear and angular speeds for the robot. The velocities published by the node `key_op` in this question will be used as a reference to command the motors by the node in the next question.

Preparation (optional). Look at the scripts `key_terminate.py` to learn how to read key presses, and `zigzag_op.py` for how to fill `Twist` messages.

Preparation . The questions below will guide you through the implementation of a node with the following specification.

> File name: `key_op.py`
> Publishes to topics: `robot_twist` (type `geometry_msgs/Twist`)
> Description: The node should maintain internally two variables, `speed_linear` and `speed_angular`, that store the desired linear and angular velocity of the robot. The user should be able to manipulate these variables using key presses. See questions below for details.

Question 3.1. As initialization the node should:

1) Set a global constant `SPEED_DELTA=0.2`.

2) Initialize both variables `speed_linear` and `speed_angular` to 0.0.

3) Create a publisher to the topic `robot_twist`.

4) Print a brief description of the available keyboard commands (see next question), together with the current values of `speed_linear` and `speed_angular`.

Question 3.2. After initialization, the node should wait in a loop for a key press, and then map the following keys (**case insensitive**) to the following actions:

- `'W'`: increment `speed_linear`;
- `'S'`: decrement `speed_linear`;
- `'D'`: increment `speed_angular`;
- `'A'`: decrement `speed_angular`.
- `'Z'`: set `speed_linear` to zero.
- `'C'`: set `speed_angular` to zero.
- `'X'`: set both `speed_linear` and `speed_angular` to zero.
- `'Q'`: sends a ROS shutdown signal (same as in the `key_terminate` script).

Increments and decrements should be done by adding or removing `SPEED_DELTA` to the variable of interest, thresholding the result so that it always remains between -1.0 and 1.0.

Question 3.3 (2.5 points). Every time either `speed_linear` or `speed_angular` changes, the node should:

1) Print on screen a short description of the action (e.g., `'Increment linear speed'`), followed by updated values of `speed_linear` and `speed_angular`.

2) Prepare a `Twist` message where the `linear.x` field is set to `speed_linear` and the `angular.z` field is set to `speed_angular` (all other fields are left to zero).

3) Publish the `Twist` message on `robot_twist`.

Note: The message `Twist` is defined for 3-D velocities (`linear` and `angular`); however, since our robot is essentially 2-D, and can rotate, but not slide sideways, we use only two fields (`linear.x` and `angular.z`) out of the six available.

Question 3.4. Run the node `key_op` together with `motor_command` (from Problem 2). Adjust `speed_offset` in `motor_command.py` for one of the motors so that, on a level, non-slippery surface, the robot will actually go straight when commanded to do so. In the report, comment on the values you settled upon.

Question 3.5 (video). Run the node `key_op` alone (without the node `motor_command`), and record the results on the terminal while giving different sequences of commands, together with the results of the command `rostopic echo /robot_twist`. Make sure the results of multiple key presses, and the corresponding messages on `/robot_twist` are clearly visible in the video. For best results, use a screen recorder.

Question 3.6 (video). Run both nodes `key_op` and `motor_command` together. Record the results of a few maneuvers; try to avoid some obstacles (when everything works, it is pretty fun).

Problem 4: Scripted operation

This problem is similar to Problem 3, except that the values for desired robot speeds are read from a file, instead of a keyboard.

File format The file with the desired velocities will be a simple text file with two floating point numbers on each row, separated by a comma. The first number corresponds to the desired `speed_linear`, while the second to `speed_angular`. Each row represents a desired twist that should be maintained for 1 s (one second). An example of a file with this format is `test.csv`, which is provided in the directory `data` of the repository.

Preparation (provided). Create a function to read a file in the format specified above.

> File name: `me416utilities.py`
> Function name: `read_two_columns_csv`
> Description: Reads the contents of the first two columns of the specified file. Values in the file must be separated by a comma. If there is an error in opening the file (e.g., wrong path), the returned list will be empty. Rows with less than two numbers in them are ignored.
> Input arguments
>
> - `filename` (type `str`) Path of the file to open.
>
> Returns arguments
>
> - `pair_list` (type `list`) A list of lists, where each element is a list of two elements with the first two values of each row of the file specified by `filename`.

Question 4.1. Create a node that reads the desired velocities from a file and publishes them to `robot_twist` at the frequency of 1 Hz.

> File name: `scripted_op.py`
> Publishes to topics: `robot_twist` (type `geometry_msgs/Twist`)
> Description: The node should use the function `read_two_columns_csv` to read the contents of the file `/ros_ws/src/me416_lab/data/scripted_op.csv` into a list. Pairs of `speed_linear` and `speed_angular` from the list will published in `Twist` messages

(populated as in `key_op`) on the topic `robot_twist` at intervals of one second. When the end of the list is reached, the node should restart from the beginning.

Question 4.2 (video). Write commands in the file `scripted_op.csv` such that, when `scripted_op` and `motor_command` are run together, make the robot:

- Go straight for about 2 ft;
- Turn left by about 90 degrees.

Since `scripted_op` will repeat these commands, the robot should trace an approximately square trajectory. **Note:** Since the motors are controlled in voltage and not speed, the result might depend on the charge level of your 9 V battery.

Problem 5: Homework feedback

Question 5.1. Indicate an estimate of the number of hours you spent on this homework (possibly broken down by problem). Explain what you found hard/easy about this homework, and include any suggestion you might have for improving it.

Hint for question 1.4: The easiest way to satisfy the requirement of not replying to the node's own publishing, is to keep an internal variable `last_str` with the last message published. If the message received is equal to `last_str`, then the node should not publish a reply. Additionally, you will need to insert a pause to avoid replying too quickly to a message.

Hint for question 2.3: The message type for the publisher can be imported using the following:

```
from me416_lab.msg import MotorSpeedsStamped
```

Hint for question 3.1: Since this node needs to keep some internal state, it is the best to wrap your callbacks and main functions in an object.