Assignment 0: Introduction to ROS

The goal of this assignment is to help you become acquainted with the ins and outs of the RACECAR platform. In doing so, you will need to employ basic concepts from Python, Numpy and ROS. Before starting this assignment, please review these three topics as necessary.
It is strongly suggested that everyone on the team is present for the completion of this assignment.

1 Controlling the RACECAR in Simulation

Take a look at section 5 to see the deliverables you will turn in. It's always a good idea to keep ALL your data from all your runs in an organized way.

In this section you will implement the necessary machinery to take commands from one ROS node, and combine them into meaningful commands to send the to the RACECAR's controlling mechanisms.

1.1 Specification

Your task is to receive messages from other nodes in the system and parse and forward them to the correct topics. The goal is to get you acquainted with the ROS environment. By the end you should have an understanding of how ROS nodes communicate over topics, how to create your own ROS node, and how to launch your ROS node and set parameters at runtime.

While you may (and should) look at the publisher code for guidance, you shouldn't modify it. We will use a clean version when testing your code.

In `src/subscriber.py` implement a ROS node that does the following:

Listens on topics:

1. Initial Pose (`/lab0/initialpose`): A string message of the form: `X,Y,`θ to be forwarded to the `/initialpose` topic.
2. Velocity (`/lab0/velocity`): A float velocity to be used as the current velocity.
3. Steering Angle (`/lab0/steering_angle`): A float steering angle to be used as current steering angle.

Implements a service (Reference):

1. Reset (`/subscriber/reset`): This service will take an empty message as an argument and return an empty message, clearing the current velocity and steering angle (setting them to zero). Reset functions are particularly useful for debugging your system without restarting your whole stack. From the command line you can reset the publisher by typing
   `$ rosservice call /subscriber/reset`

1.2 Running the Publisher

This section will walk you through starting the publisher code from end to end.

In separate windows (or tmux session) run the following commands:

1. `$ roscore`
2. `$ roslaunch mushr_sim teleop.launch`

3. Once you are done with your subscriber: `$ roslaunch lab0 subscriber.launch`
4. `$ roslaunch lab0 publisher.launch`

You should note the publisher launch file also starts the map server. The map server is integral to relating the car on a map. In future labs you may be responsible for running the map server; look at the publisher launch file for more information.

1.3 Example Publishing Trace

This is functionally what is happening in the publisher. The subscriber will be specifically interested in any of the commands starting with `Send`.

```
Send starting position: x=0, y=0, car_angle=0.0
Send velocity: 2.0 (m/s)
Send heading: 0.09 (rads)
Wait: 10 (seconds)
Send heading: -0.09 (rads)
Wait: 10 (seconds)
Send velocity: 0.0 (m/s)
```

2 Visualizing the RACECAR's path in Simulation

In this section you will use ROS markers to visualize the path the car takes in simulation.

2.1 Specification

In `src/pose_markers.py` implement a ROS node that does the following:

At a frequency of 2 Hz, publish a marker of the car's current pose as blue arrows as shown in Figure 1. Refer to the Marker Specification to see how to publish markers. The current position of the car can be found on the `/sim_car_pose/pose` topic. Publish your markers to `/pose_markers/markers`

Notes and help:

1. The header `frame_id` should be set to `"map"`. In `rviz` there are multiple "views" of the robot. The map frame gives an overhead view of the world.
2. The scale values used in Figure 1 for `scale.x,y,z` are 1, 0.05, 0.05 respectively (in the docs you should see how changing these values alters the markers).
3. If you don't see your markers in `rviz`, be sure you add it to your Displays. Under the Displays pane there is an "Add" button. Click this and chose the tab "By topic", then find the topic your markers are published on and add it.

3 Controlling the RACECAR

In this section, you will use ROS utilities to record control inputs as you drive the car around. You will then write a program to playback these inputs so that the robot can follow a pre-recorded path.

3.1 In Simulation

Use the `rosbag` (Reference) command to record data published to the `/vesc/low_level/ackermann_cmd_mux/input/teleop` topic as you teleoperate the robot to trace a Figure-8 path. The output path of `rosbag` can be specified with the `-o` argument. Once you have completed the recording, complete the following tasks:

1. Implement a Python script in `bag_follower.py` that:
   - Extracts the data from the recorded bag file into an ordered list. (Reference)

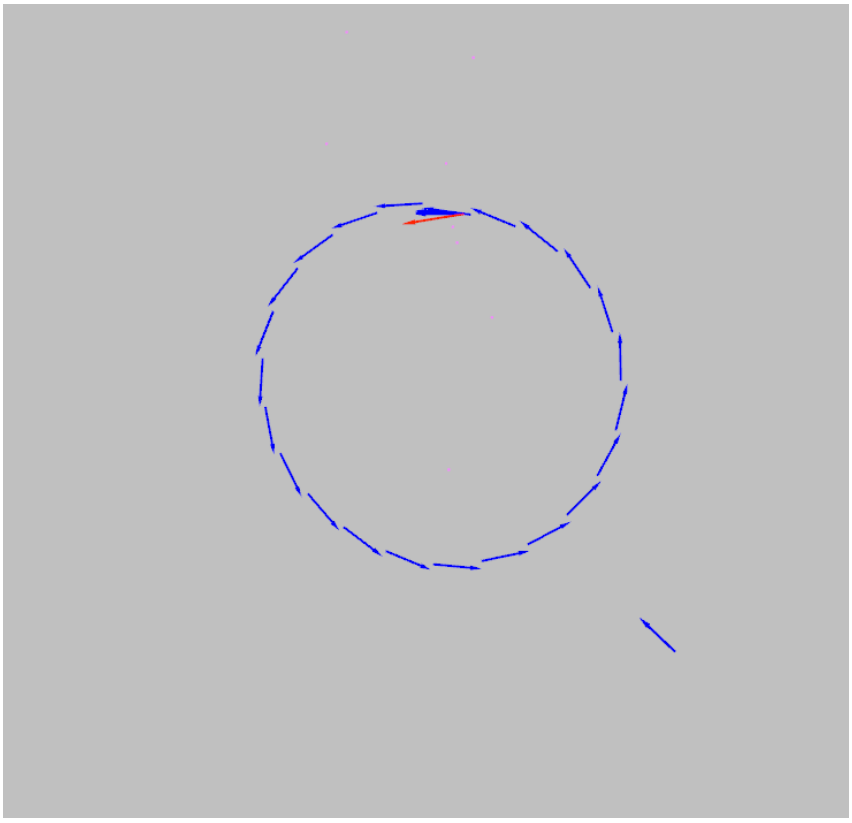

Figure 1: A blue marker is places every 0.5 seconds at the car's position, create a trace of the path taken.

- Re-publishes the extracted data at a rate (specified in the launch file, with default being 20Hz) such that the robot follows a Figure-8 path without being tele-operated. Publish the data to the `/vesc/high_level/ackermann_cmd_mux/input/nav_0` topic.
- *In our solution we republish at 20Hz. Think about what will happen when you lower the rate. You can verify your hypothesis by testing changing the rate in both simulation and on the real car.*

2. Implement a launch file in `bag_follower.launch` that:
   - Passes the bag file's path to your script
   - Launches your script
3. Test your script using the bag file in simulation. The car should move in a Figure-8 pattern without being teleoperated. *Make sure to set the initial pose of the robot.*
4. Add functionality to your script that allows the robot to follow the recorded path backwards (i.e. the robot drives in reverse). Add a parameter to your launch file that specifies whether the robot follows the path forwards or backwards. Test that the robot correctly does a Figure-8 backwards.

3.2 On the Car

1. Collect a bag of you teleoperating the real robot performing a Figure-8.
2. Play the bag back so that the robot autonomously does a Figure-8. Assuming that nothing is currently running on the robot, the following commands should cause your robot to follow a Figure-8 path:
   (a) `$ roslaunch racecar teleop.launch`
   (b) Wait for robot to finish launching
   (c) Hold down the RIGHT bumper on the Logitech controller (it is labeled 'RB')

(d) `$ roslaunch lab0 bag_follower.launch`

3. Again play this bag back so that the robot autonomously executes the commands recorded in the bag. Make sure to test both forwards and backwards figure 8s.

4 Extra Credit

These extra credit problems are designed to get more exposure to the cars. They will have less structured guidance in their descriptions. Be creative and principled with your solutions.

4.1 Analytical Figure 8

Create a ROS node `src/fig8.py` that, given a steering angle and velocity (from the parameter server), drives the car in a figure 8. The figure 8 will not be perfect, but it should fairly closely resemble a figure 8. You can use your marker code to track the path the car takes.

4.2 Safety Controller

Create a ROS node that listens for laser scans on the `/scan` topic. While the car is within some (parameterized) distance from an obstacle, send a 0 velocity.

To override teleoperator controls, you'll need to change the priority of the controller. In the package `racecar` (`$ roscd racecar`), edit `config/common/low_level_mux.yaml` changing the priorities of the teleop and the safety node.

```
// racecar/config/common/low_level_mux.yaml
subscribers:
    - name: "Teleoperation"
      ...
      priority: 5
      ...
    - name: "Safety"
      ...
      priority: 10
      ...
```

Now when you send controls to the topic `/vesc/low_level/ackermann_cmd_mux/input/safety` they will overwrite the teleoperator.

5 Deliverables

Put the associated work in the directories and `tar` your repository.

`$ tar czf lab0.tar.gz lab0`

1. `lab0/videos/`: Videos of . . .
   (a) Your `subscriber.py` driving the car with messages from `publisher.py` (You can use Kazam to take a video of your screen.) `subscriber.mp4`
   (b) Your RACECAR performing a figure 8 in simulation (forward and backwards). `sim-figure8-forward.mp4, sim-figure8-backward.mp4`
   (c) Your RACECAR performing a figure 8 in real (forwards and backwards). A cell phone camera of the car will suffice. `real-figure8-forward.mp4, real-figure8-backward.mp4`
2. `lab0/bags/`: Bags:
   (a) A bag of your figure 8 in sim: `sim-figure8.bag`
   (b) A bag of your figure 8 in real: `real-figure8.bag`
3. `lab0/src/`: Code:

   (a) Your `subscriber.py`
   (b) Your `pose_markers.py`
   (c) Your `bag_follower.py`
   (d) Any extra credit source files.
4. `lab0/launch/`: Launch files:
   (a) Your `subscriber.launch`
   (b) Your `pose_markers.launch`
   (c) Your `bag_follower.launch`
   (d) Any extra credit launch files.
5. `lab0/writeup.txt`: Writeup (answer the following):
   (a) You should have collected two bags of the robot doing a figure-8, one in simulation, and one on the real robot. Play back both of these bags on the real robot. Explain any significant differences between the two performances.
   (b) Is the robot successfully able to navigate through Sieg Hall when playing back the corresponding bag? Note any deficiencies in performance. Explain why these deficiencies might occur.
   (c) If you did any of the extra credit, describe which ones you did and shortly describe what you did.