Simulating kinematic models in ROS

This lab introduces you to GitHub and the ROS BARC simulator. You will also practice downloading ROS packages and integrating them with your own code. In future labs it will be very important that you understand how to test your own controllers in simulation before you load them onto the actual BARC vehicle, so make sure you understand each step in this lab.

***PLEASE READ:** This lab contains series of instructions designed to help you familiarize yourself with the BARC simulator, and it is **critical** that you follow each of them in the order stated. Check that you have completed all previous steps before moving on to the next task. ***

Task 1 Cloning the BARC GitHub Repository

1. In this task you are going to familiarize yourself with the BARC GitHub repository. If you are unfamiliar with GitHub, you may want to review the basics at https://guides.github.com/activities/hello-world/. For purposes of this class, you should know at least how to fork and clone repositories to your account so you can use existing scripts for your work. You can follow the steps in this lab as a guide.

   (a) In order to use the existing BARC scripts, you need to fork the BARC GitHub repository to your GitHub account. This means you are essentially creating a new copy of the repository on your own account, so you can make changes to the code without overwriting the original version.

      i. Create a Github account for yourself if you don't already have one.

      ii. Log into your Github account, and access the BARC's Github page : **https://github.com/MPC-Berkeley/barc**

      iii. Use the `Fork` button on the top right of the webpage to fork the BARC repository to your GitHub account.

      iv. Log into your virtual Ubuntu machine. Open a new terminal, and clone the folder you forked to your account onto your virtual machine:
      `$ git clone your-forked-http-address`
      This makes the files available to you to open and use, by creating copies of your forked repository on your virtual machine. You should now have a folder called `barc` located in your home directory; this folder contains the repository files.

   (b) Next, you will finish setting up the BARC working environment on your virtual machine.

      i. Open the command window and navigate to the `barc` workspace folder,
      `/home/me131/barc/workspace`

      ii. Compile the BARC project contained in the cloned folder:
      `$ catkin_make`
      The `catkin_make` command compiles all the functions contained in the `src` folder, so they can be executed on your machine. Make sure to always run the `catkin_make` command from the same directory as the corresponding `src` folder.

iii. Now you need to source the ROS commands to use them from your command window. Navigate to your home directory (`$ cd`), and open the .bashrc file in Sublime:
`$ subl ~/.bashrc`
The .bashrc file contains a series of prompts that the system runs automatically whenever a new terminal window is opened. We want to use the file to set up our workspace, by using the command `source` to load the enviroment setup file contained in the BARC package. At the end of the .bashrc file, add:
`source /home/me131/barc/workspace/devel/setup.bash`

The `source file` command causes the program to execute the content of the indicated file. Thus this additional line means that each time a new terminal is opened, the `setup.bash` file in your `barc` folder will run.
Make sure to save the file with your edits before closing Sublime.

iv. Add the original BARC repository as your upstream repository. Make sure you are within the barc folder when you any git commands. This following command allows your local repository to track changes (e.g. adding files, modifying code, etc) we make on the master repository
`git remote add upstream https://github.com/MPC-Berkeley/barc`

v. Get the latest code changes form the master repository with the following command
`git pull upstream master`

vi. Close the terminal window you have been working in, and open a new window. This is required for the `catkin_make` changes to take effect.

Task 2 Designing an Adaptive Cruise Controller in ROS Simulation

1. You have now set up the BARC environment on your machine. The BARC repository contains a vehicle simulation environment – this is incredibly useful for testing the safety and effectiveness of controllers before uploading them to the actual BARC car. The simulator files are contained in the `barc/workspace/src/labs/src/lab2` folder. For today's assignment, you will primarily work with the `lab2.launch` file, found in the `barc/workspace/src/labs/launch` subdirectory.

   (a) Navigate to the launch subdirectory `barc/workspace/src/labs/launch`, and open the `lab2.launch` file in Sublime. Notice that the file defines several parameters and functions for modeling and simulating the BARC vehicle.

   - The `controller` node calculates actuation inputs for the BARC vehicle based on the algorithm contained in the `controller.py` file. Later in the exercise, you will add your code for a cruise controller into `controller.py`, and the `controller` node will ensure that the BARC vehicle receives the resulting actuation commands.
   - The `simulator` node simulates the BARC vehicle moving in response to the `controller` node's actuation inputs based on the vehicle model contained in the `vehicle_simulator.py` file. Later in the exercise, you will add your code for a discretized kinematic bicycle model into `vehicle_simulator.py`.
   - The `rosbag_record` node records information about our vehicle during the simulation. Rosbag is a standard ROS method for collecting node-to-node message data in a file format called

bags during ROS operation, so we do not need to link it to our own python function files. You can learn more about the rosbag feature **here**. In our launch file, we specify the path of recording the bag to be `/home/me131/`. So that after simulation, you can find your bag file there.

(b) Navigate to the directory `barc/workspace/src/labs/src/lab2`. Run the command `$ ls -l` to see all the files and their permissions (i.e. read-r, write-w, execute-x) within the directory. You can see that the file `controller.py` has permission -rw-rw-r–, which means the user does not have permission to execute the file, also indicated by the white color coding. Type `$ chmod u+x controller.py`. And then list all the files again `$ ls -l`. You can see that both the `controller.py` and `vehicle_simulator.py` files are now executable, also indicated here by the color green.

(c) ***__Deliverable__**: We will model the BARC vehicle using the one-step Forward Euler discretized kinematic bicycle model you covered in lecture. The simplified continuous-time kinematic bicycle model is governed by the following differential equations:

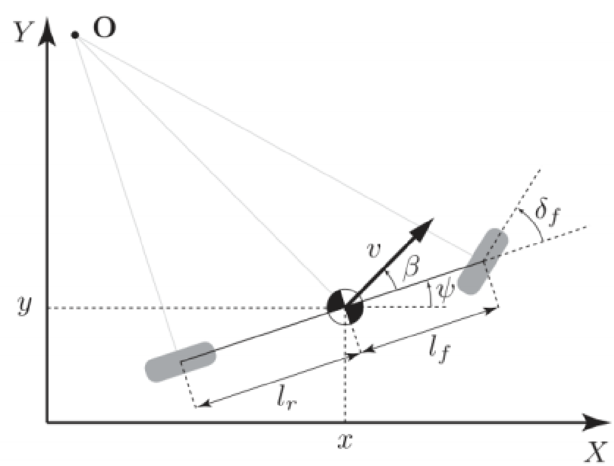

Figure 1: The parameters of the kinematic bicycle model.

$$
\begin{aligned}
\dot{x} &= v\cos(\psi+\beta) \\
\dot{y} &= v\sin(\psi+\beta) \\
\dot{v} &= a + F_{ext}/m \\
\dot{\psi} &= \frac{v}{l_r}\sin(\beta) \\
\beta &= \tan^{-1}\left(\frac{l_r}{l_f+l_r}\tan(\delta_f)\right)
\end{aligned}
\tag{1}
$$

where:
x = global x coordinate, mass center location
y = global y coordinate, mass center location
v = speed of the vehicle
ψ = global heading angle
a = tangential acceleration of the center of mass (along direction of velocity)
δ_f = steering angle of the front wheels with respect to the longitudinal axis of the car

F_{ext} = sum of gravity, drag and ground friction forces acting on the vehicle
l_r = distance from the center of mass of the vehicle to the rear axle
l_f = distance from the center of mass of the vehicle to the front axle
β = angle of the current velocity with respect to the longitudinal axis of the car
m = mass of the vehicle

Recall that the first-order, forward Euler solution to an ODE $\dot{x} = f(t, x(t), u(t))$ is

$$x((k+1)T_s) = x(kT_s) + T_s f(kT_s, x(kT_s), u(kT_s)) \tag{2}$$

for $k = 0, 1, \ldots$ where x and u are vector containing your system state variables and actuation inputs, respectively.
Write a python function of the kinematic bicycle model and add it to the `bike_model.py` file. The `vehicle_simulator` node will automatically import the vehicle model defined by the `bikeFE` function at run-time.
Note: Your simulation environment will contain both flat and inclined roads of various grades. The simulation environment will provide the incline angle parameter θ (in degrees) at each time instance, as well as the ground friction force F_f and air drag coefficient a_0, which we will define such that the drag force $F_d = a_0 \cdot v^2$. Use $l_f = l_r = 1.5[m]$, $m = 2000[kg]$ and $g = 9.8[\frac{m}{s^2}]$. Notice that these parameters are already defined for you in the `lab2.launch` and `bike_model.py` files. Your model should follow this general form:

```
def bikeFE(x, y, psi, v, a, d_f, a0, m, Ff, theta, ts):

    # compute slip angle
    beta        = TODO

    # compute next state
    x_next      = TODO
    y_next      = TODO
    psi_next    = TODO
    v_next      = TODO

    return array([x_next, y_next, psi_next, v_next])
```

Submit your `bike_model.py` file.

(d) *****Deliverable**: Write a python function for an adaptive cruise controller that tracks a reference speed of $8\frac{m}{s}$, and add it to the template in the `controller.py` file. The node corresponding to the `controller.py` file reads the topics `z_vhcl` (vehicle state) and `ez_vhcl` (vehicle state measurement error), and publishes the calculated actuation inputs to the topic `ECU`. Look through the file and make sure you understand the workflow and how the different nodes are connected.
Recall that PID controllers are of the form: $acc = K_p\Delta v + K_i \int \Delta v dt + K_d \dot{\Delta v}$, where $\Delta v = v_{ref} - v_{actual}$.
Write your PID controller function using this template (also reproduced in `controller.py`):

```
class PID():
    def __init__(self, kp=1, ki=1, kd=1):
        self.kp = kp
        self.ki = ki
        self.kd = kd
```

```
    def acc_calculate(self, speed_reference, speed_current):

        acc = TODO

        return acc
```

By the end of this exercise, you should be able to test and tune your controller gains using the simulator. When you have finished tuning, submit your final `controller.py` file, which will contain your chosen tuning gains in line 65 of the original file.

(e) In a new command window, launch your simulation:
`$ roslaunch labs lab2.launch`
This will launch the nodes `vehicle_simulator.py`, `controller.py` as well as the rosbag recorder node.

(f) In a new command window, explore the z_vhcl topic by typing `$ rostopic echo /z_vhcl`. Recall that this topic contains the state messages sent between the vehicle model and the controller. When the vehicle has traversed approximately 180 meters, stop the simulation by typing `CTRL+C` in your terminal window running the simulation.

(g) The recorded rosbag is automatically saved in your home directory, i.e. `/home/me131/`. Navigate to this directory, and check the name of your recorded bag. Notice that the bag names are the date and time of your experiment.

(h) Open the `~/barc/workspace/src/labs/src/lab2/acc_simulation.py` file, and modify line 10 to be:
`bag = rosbag.Bag(os.path.expanduser("~/name-of-your-bag.bag"))`
The `acc_simulation.py` file will allow you to visualize your vehicle move along the terrain under your control law, by replaying the data saved in the rosbag file. Save and close the file.

(i) Run the plot_longitudinal.py file with
`$ python acc_simulation.py`

(j) *****Deliverable**: For a different visualization, use the `plot.py` file, making sure to include the appropriate bag file name in line 7. This function tracks your vehicle's velocity against the reference speed of $8\frac{m}{s}$ as the vehicle moves along the path. After choosing the best tuning parameters, submit your plot.

Task 3 Teleoperation

1. In Lab 1, you controlled the movement of a simulated turtle using your keyboard. Recall the Turtlesim teleoperation workflow (you may want to review your rqt_graph from Lab 1): The `teleop_turtle` node, which we downloaded as part of the Turtlesim teleoperation package, listens for keyboard commands. When a keyboard key is pressed, the node turns these key presses into a message of type `geometry msgs/Twist` corresponding to some internal logic contained in the node. These messages are then published to the `turtle1/cmd_vel` topic. The `turtlesim` node, which controls the movement of the turtle in your simulation window, subscribes to the `turtle1/cmd_vel` topic and thus receives the `geometry msgs/Twist` messages. It adjusts the turtle's movement based on the velocity commands

received in the message and the turtle's simple kinematic model, and reflects these movements in your simulation window.

Your specific task for this assignment is to use your computer keyboard to provide actuation inputs to the simulated BARC vehicle. Certain key presses (e.g. up/down arrows) should correspond to longitudinal acceleration/deceleration, while others (e.g. left/right arrows) should control your change in steering angle inputs to your vehicle. For simplicity, you can reuse the same `vehicle_simulator` node as in the previous task, which models your vehicle using the discretized kinematic bicycle model you wrote. This task is meant to challenge you, so we are providing less guidance than in previous examples. If you get stuck, refer back to some of the ROS debugging tools you learned in Lab 1.
There are several ways of organizing the work flow for this task, but the simplest is to follow the outline shown below:

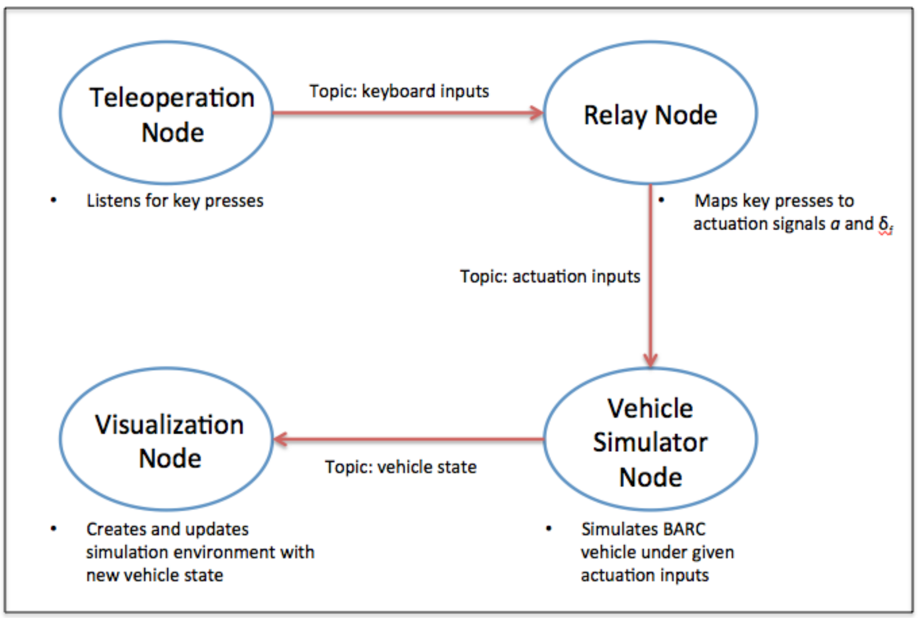

Figure 2: Suggested workflow for the Teleoperation task.

More information about the necessary nodes is provided below:

- **Teleoperation Node**: You will need a ROS node that listens for and interprets keyboard presses. Many such nodes have already been written by other ROS users, and they are available for you to download as packages from Github and include in your code (this is one of the main benefits of using ROS). You can use any package you like, but you may want to check out the ROS teleop-twist package as a starting point, cited **here**. Whichever package you choose, fork it to your Github account and then clone your forked version to your machine using the `git clone` command you used in the previous task. Note that depending on which package you choose, you may also have to download some dependencies. Remember to compile your new package before testing and editing it!
- **Relay Node**: You will also need a node that translates your key presses into actuation signals for the simulated BARC vehicle. The relay node will receive messages from the teleop node, and send messages to the vehicle simulator node. The vehicle simulator nodes expects messages of type

ECU. Check what format this is using the `$ rosmsg info ECU` command. This is the format your relay node output needs to be in.

You are free to write your own relay node, or adapt existing code (either from previous tasks or existing Github repositories). Alternatively, it may be easier for you to combine the teleoperation and relay nodes into one single node – this is perfectly fine, so long as you keep your workflow clearly structured.

Creating a new node essentially consists of writing a new function. ROS allows you to use both python and C++ code, and you may use whichever you prefer. An example template for a publishing and subscribing node (in Python) is given here:

```
class RateRelay(object):

    def __init__(self, dt=0.05, extra_parameters):
        self.sub=rospy.Subscriber('keyboard/keyup', Key, self._up_cb)
        self.pub = rospy.Publisher('ecu', ECU, queue_size=10)
        self.rate = rospy.Rate(1.0/dt)

    def _publish_ecu(self):
        #add code to assemble the message to publish
        self.pub.publish(ecu)

    def _keypress(self,msg):
        #add code to interpret different keypresses

    def run(self):
        while not rospy.is_shutdown():
            self._publish_ecu()
            self.rate.sleep()

def main():
    relay = RateRelay()
    relay.run()
    rospy.spin()

if __name__=='__main__':
    rospy.init_node(node name)
    main()
```

The linked **ROS tutorial** provides more information about how to write nodes.

Hint: Look at the code structure of the node defined in the `controller.py` file. Recall that the `controller.py` node is written so as to interpret messages from the `Z_DynBkMdl` topic, and publish `ECU` messages to the vehicle simulator. In this task, you are essentially reproducing this workflow, but replacing the `controller.py` node with a new teleoperation node that serves a very similar purpose.

- **Vehicle Simulator Node**: The vehicle simulator node is the same node you used in the previous task. It receives ECU messages, and uses your discretized kinematic bicycle model to update the state of the simulated BARC vehicle. Use the `vehicle_simulator` node we have provided this node for you.

  Note: Since we want the vehicle to run on flat ground for this simulation, assign the value of `theta` to be `theta = 0` in line 70 of the `vehicle_simulator.py` code.

- **Visualization Node**: Use the `view_car_trajectory.py` node we have provided for you. This node will provide a basic visualization of the simulated vehicle moving according to the control inputs it receives.

- **Launch File**: Use the `lab2.launch` file as a template for a new launch file which organizes your workflow. This new launch file should be used to start your four nodes:
  - Replace the `controller.py` node with your teleoperation nodes
  - Reuse the `vehicle_simulator.py` node, but be sure to change the theta value to 0
  - Include a visualization node that calls `view_car_trajectory.py`

*****Deliverables:**

(a) Take a video of your screen showing the simulated BARC vehicle controlled from your keyboard.

(b) Submit an rqt_graph visualizing your workflow during vehicle teleoperation. Include a short description of your nodes, their source, and their operation.

(c) Submit your teleoperation launch file.

Resources:

- Sourcing ROS packages from Github:
  https://wiki.nps.edu/display/RC/Setting+up+a+ROS+package+from+Git
- Creating a catkin workspace: http://wiki.ros.org/catkin/Tutorials/create_a_workspace
- ROS packages overview: http://wiki.ros.org/Packages
- Google! This assignment is meant to be challenging, and encourage you to familiarize yourself with the Github and ROS workflows. If you find yourself stuck, don't hesitate to ask your classmates or instructors, but note that these two systems are heavily documented and you are most likely not the first to have your question!