Week 1 Lab Exercise 1: The Robot Operating System: Keyboard Control

The F1/10 Team

Introduction

In this tutorial we are going to cover the details regarding the low-level motor control/interface setup. The following sections will detail the Lab, the exercise and the last section contains the transcript of the accompanying video tutorial. This tutorial details the python implementation of the keyboard control. This is the configuration of the keyboard control that you will need later on as well, however if you are interested in implementing this using C++, you will find the necessary tutorial **here**.

The Teensy, the Jetson and the ESC..

The RF reciever module on the traxxas sends PWM signals to the servo(controls the steer) and the ESC(controls the throttle). This PWM signal is a 100Hz wave with varying duty cycles. A duty cycle of 10% on the throttle and the steering makes the car reverse at full speed/orient wheels in one direction. The opposite actions are observed when the duty cycle was set to 20%. A signal of 15% duty cycle on both channels results in 0 throttle and 0 steer(wheels oriented straight ahead).
To bypass the RF reciever, we feed the required signals directly to the ESC and the servo. We generate these signals using the Teensy Microcontroller. The Teensy gets the commands for the desired duty cycle values from the Jetson via USB interface. The Teensy runs a ROS node that listens for this particular command.
There are two nodes that are relevant to the keyboard control on the Jetson. The **talker.py** node and the **keyboard.py**. The talker.py node takes in user input in the (-100 to 100) range for the steer and the throttle channel. This node listens to the published messages from **keyboard.py** and converts this to particular PWM values. The teensy node(the teensy board is flashed with the code already - this file can be found in the git repo under teensy) listens to the messages published by **talker.py** and generates the 100Hz square waves with the specified duty cycle.

The Task Description & the Code Layout

Your task in this exercise is to

- Implement the talker.py node
- Implement the keyboard.py node (OPTIONAL)

The code is available at Git Repo

If you want to change the way the mapping from stdin(keyboard) to duty cycle is performed, feel free to alter the teensy code. A couple of things that might help you if you alter the teensy code is mentioned in [1].

You will be provided with the skeleton package '**keyboard_control**'. You can choose to either incorporate this as a new package or you could just take the new files from this package and include them in your existing package.

The code uses custom messages - a pair of float values to define the desired throttle and accelaration (keyboard.py). Another custom message (an int pair) is used to transmit the commanded PWM to the teensy. You might want to go through the ROS tutorials regarding custom messages[2] for reference.

The **keyboard.py** node: In keyboard.py node, user input from keyboard is implemented via a non blocking call to stdin. As soon as the user pressess the arrow keys, the respective command is published via the **drive_parameters** topic. This node is responsible for setting the rate at which the throttle/steer command changes. If you decide to implement this node on your own, you must

- Include the custom command drive_param and assign float values to its respective fields
- Perform a non blocking call and wait on the arrow keys(or other keys you prefer) and assign a drive command accordingly
- Publish the custom message on the **drive_parameters** topic.

The **talker.py** node: You will have to code almost the entirety of this node - write callback functions[4] that listen to the topic published by the keyboard.py node. Write methods to convert these values to the required PWM duty cycle values. If you follow the provided keyboard.py node, you will need to map from values of **(-100, 100)** to **(6554,13108)**, where 6554 corresponds to a duty cycle of 10% and 13108 corresponds to a 20% duty cycle. These values are dependent on the registers used in the teensy to generate the PWM, so unless you change the Teensy code accordingly, stick to this range of values.

The PWM values must be stored in the custom messge("drive_values") and published via the "drive_pwm" topic. The teensy listens on this topic and generates the required waveforms.

This will be the first time you write a node from scratch and you should look at the related tutorials on the ROS website to ensure you are doing it right. The way ROS builds python and C++ nodes is different and if you are implementing nodes in C++ then you must follow the ROS C++ guide.

The Transcript: Video Tutorial 5

In this tutorial we are going to go over your keyboard control lab exercise.

Oh, and just as an aside, I want to mention this terminal multiplexer I use. It might be beneficial for those of you who haven't heard of it. On the screen right now, I have 4 terminal panes through which I have 'ssh'd' into the car(ubuntu@teg...). It gets rather cumbersome to operate a bunch of separate terminal windows. To avoid the clutter of so many different windows, I prefer using either tmux or terminator. The one you are seeing in this video is the terminator application. The terminals become a lot easier to manage and view.

Anyway, back to the tutorial - Im going to walk you through your Lab Exercise wherein you are asked to control the car with your keyboard. This video will make full sense to you only if you have read the accompanying description of the lab exercise. Ill talk you through the different nodes that you need to run and also the importance of each of them. Lets start by going into our workspace and sourcing the environment.
Once we got our environment setup, we begin by launching

```
$ roscore
```

We now run the **talker.py** node using

```
$ rosrun race talker.py
```

This node is waiting for desired throttle and steering from the user which will be sent via the **keyboard.py** node.

We next start the rosserial node - i.e, the Teensy. use the rosserial command to launch the node.

```
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

We run the **keyboard.py** node using

```
$ rosrun race keyboard.py
```

You will see the talker.py node print the velocities and steering angles its recieving. The 0 throttle and the 0 steer refer to the 15% duty cycle that was mentioned in the write up - where the car is oriented straight and is stationary. The range of user input in this implementation is limited to $(-100, +100)$ which corresponds to 10% and 20% duty cycle respectively. Again, refer to the lab handout if you aren't sure why these thresholds of the duty cycle are important.
We will now command the steering through the keyboard. The same type of control holds good for throttle.

You can see that I have chosen the incremental steps to be of the order of 0.1. This is something you can take a call upon in your implementation. You should be mindful of not exceeding the duty cycle thresholds and should implement a saturation condition in your code.

There are a lot of opportunities to be creative in this lab and we hope you guys enjoy.

References

[1] Rosserial setup and tutorials

[2] ROS Tutorials - Custom Messages

[3] Building a Package

[4] Simple Subscriber and Publisher ROS python