Welcome to the "zeroth" lab of ECE 346! This is a preliminary Robotics Assignment (or just Lab for short) in which we will learn the key concepts of ROS (the Robot Operating System) to control the Mini Trucks that will accompany us throughout the class.

In Lab 0, you will:

- Get familiar with the visualization and simulation tools for this class.
- Get familiar with the mini-truck platform.
- Learn how to interface with ROS subscribers, publishers, and parameter servers.
- Learn how to run your own software on the Mini Truck.
- Develop and test a goal-reaching controller for your robot.

Prerequisites (from Pre-Lab 0). Before you get started with Lab 0, make sure you have:

- Successfully installed ROS on your computer.
- Created a fork of the ECE346 GitHub repository.
- Become familiar with the basic ROS concepts.

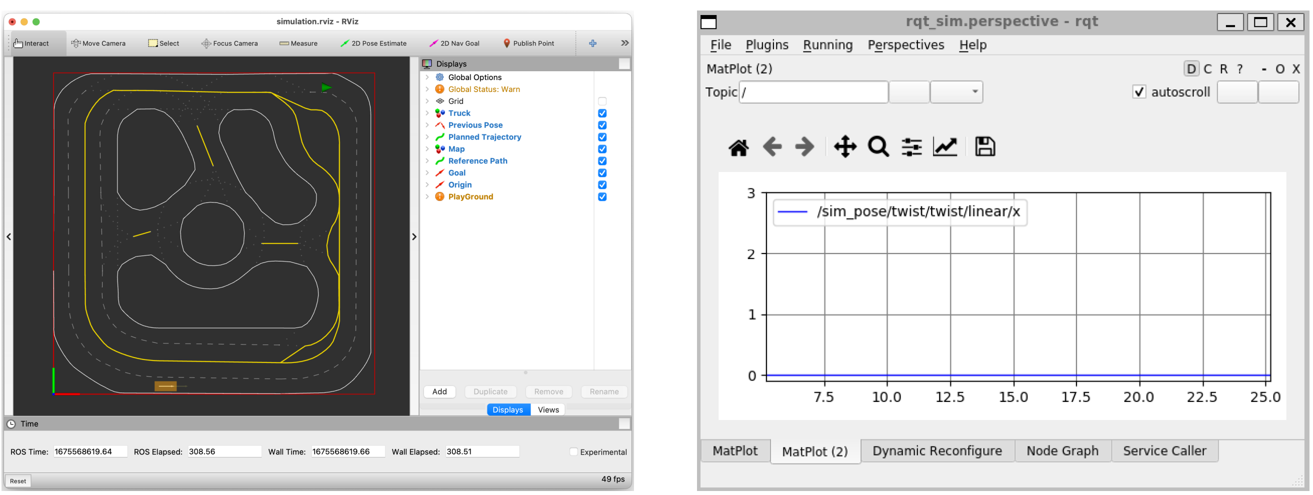
Contents

1 Getting Started

In this Lab, you will implement a simple goal-reaching controller and apply your knowledge of ROS to make it run on both the simulation environment and your Mini Truck robot. We will use a proportional controller for the throttle, and a pure pursuit controller for steering. You need to finish **6** tasks by filling in missing codes of file `pure_pursuit.py` (file path: `<Path to your repo>ECE346/ROS_Core/src/Labs/lab0/scripts/controller/pure_pursuit.py`).

Before we dive into technical details, let's take a look at what is provided for this Lab. First, please make sure you have the latest version of the ECE346 code in your own private fork. If you need a refresher on how to do this, you can check here. Under our catkin workspace (`ROS_Core`), we can build all packages, set up the environment, and launch our nodes, all from our terminal, as follows:

```
# Activate ROS environment
conda activate ros_base
# Navigate to the workspace
cd <Path to your repo>/ECE346/ROS_Core
# Build ROS packages
catkin_make
# Setup environment
source devel/setup.bash # .zsh for Mac users
# Launch Nodes
roslaunch lab0 lab0_simulation.launch
```



(a) Rviz visualization tool. (b) RQT GUI.

Figure 1: Diagrams of the remote controller

Two windows should pop up when you run the above roslaunch command. The first one (Figure 1a), is managed by an **Rviz** node. In the RViz window, you should see an orange rectangle which represents your robot. Rviz will serve as the main visualization tool in our class. It is highly configurable, and we will introduce more functionalities (such as visualizing the map and planned routes) in future Labs.

The second window (Figure 1b), is the **RQT** GUI. It is a versatile tool that allows you to inspect your ongoing ROS processes, send ROS messages and call ROS services, visualize data, etc. RQT is highly configurable. You can adjust the layout and panels and even create your own plugins.

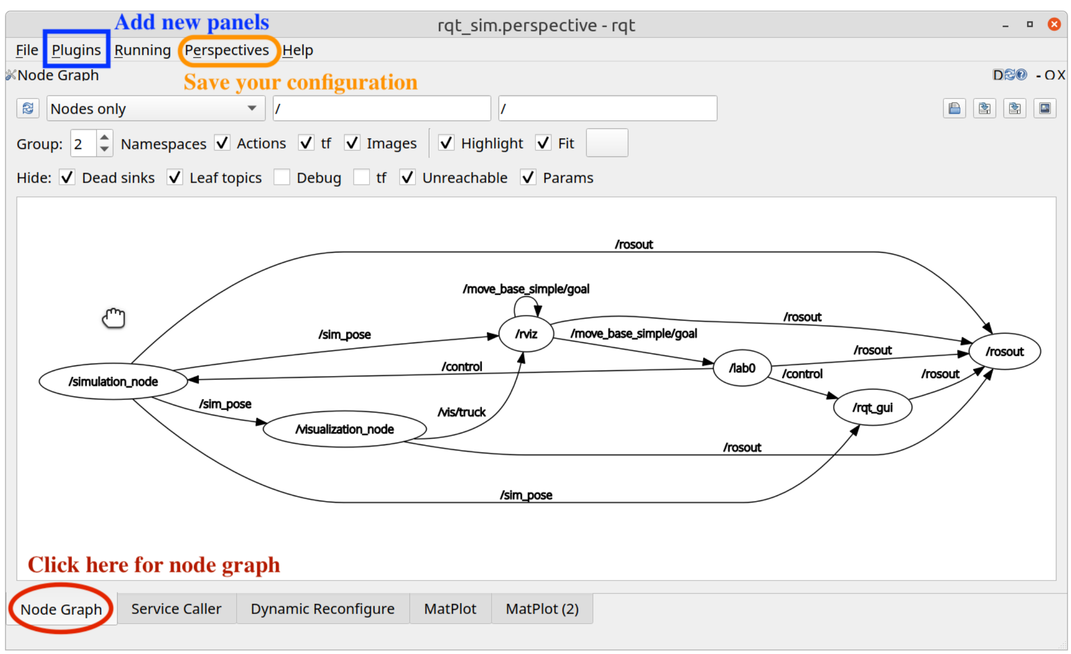


Figure 2: Node graph of Lab 0 from RQT GUI

From the RQT GUI, let's first take a look at the node graph page. If the node graph is not shown on your GUI, you can add one from **Plugins** menu on the top of the panel. Figure 2 shows a node graph of Lab 0 of 6 nodes. The `/rosout` node starts automatically with ROS Master, and it logs messages to your console. The `/rviz`, `/visualization_node` and `/rqt_gui` nodes handle visualization and process monitoring. The `/simulation_node` simulates the dynamics of our robot after executing control commands from the `/lab0` node. All those nodes are started with a single `roslaunch` command. In the next section, we will take a look at the basic functionality of roslaunch.

2 `roslaunch` Basics

`roslaunch` is a tool for easily launching multiple ROS nodes, as well as setting parameters on the Parameter Server. `roslaunch` takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch. In Lab 0, you just need to know how to interpret a launch file and pass arguments during `roslaunch`.

2.1 Reading a Launch File

let's first **take a look** at the launch file you just used. You can click the link or find it locally from `<Path to your repo>ECE346//ROS_Core/src/Labs/lab0/launch/lab0_simulation.launch`.

The first section, with syntax `<arg name="AA" default="BB" doc="CC"/>`, defines a list of arguments that you can pass into this launch file.

The second section `<rosparam command="load" file="$(find lab0)/configs/config.yaml"/>` loads a list of parameters defined in a YAML file to the ROS parameter server. `$(find lab0)` will ask ROS to find the path to the `lab0` package so that you do not need to type the absolute path.

The third section, with blocks enclosed by `<include> xxxx </include>`, specifies other launch files to include during this launch, effectively allowing nesting of launch files. In addition, the syntax `<arg name="XX" value="$(arg AA)"/>` passes argument AA in this launch file into argument XX of the included launch file.

The last section, with blocks encolosed by `<node> xxxx </node>`, starts nodes defined in `lab0_node.py`. Similar to the above section, the syntax `<param name="YY" value="$(arg AA)"/>` loads argument *AA* from the first section into the ROS parameter server with name *YY*.

We will not be writing our own launch file for this Lab. That said, you can find a complete guide to launch file syntax here if you are interested in learning more.

2.2 Passing Arguments During `roslaunch`

To pass an argument, we can simply append `Argument_Name:=Argument_Value` to your `roslaunch` command. For example, let's close our previously launched ROS nodes and try

```
roslaunch lab0 lab0_simulation.launch init_x:=1 init_y:=1
```

You will see the car is now starting at a different location.

2.3 Getting ROS Parameters

The argument you set in during `roslaunch` are read by your ROS nodes through ROS Parameter Server API. In ECE346, we provided a wrapper around this API to search and load a parameter. This can be achieved by using the function

```
get_ros_param(param_name, default_value)
```

You can find this function in here, and it will be included for all Labs.

3 ROS Messages, Topics, Publishers and Subscribers

One primary function of ROS is the communication between nodes using messages. The publisher sends out the message to ROS topics, and the subscribers receive the messages. This section will use two examples borrowed from the ROS official tutorial to understand how publishers and subscribers work in ROS. For visualization, the node graph in Figure 2 indicates which nodes are publishers or subscribers to a certain topic. For example, the `/lab0` node publishes to the `/Control` topic, and the `/simulation_node` is subscribed to the `/Control` topic.

3.1 ROS Publisher

First, let us look at the simple ROS publisher code below. In this Python code, we have a node named `talker` that sends messages to a topic named `chatter` at a rate of 10 HZ (every 0.1 seconds).

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher("chatter", String, queue_size=10)
    rospy.init_node("talker", anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world {}".format(rospy.get_time())
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == "__main__":
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Listing 1: Code for publisher

Now, let's break down the code.

```python
#!/usr/bin/env python
```

Every Python ROS Node will have this declaration at the top of the Python file. This line ensures your script is executed as a Python script.

```python
import rospy
from std_msgs.msg import String
```

You need to import `rospy` if you are writing a ROS node in Python. We also need to import our desired data type for ROS messages. Here, we will import `String` from the standard `std_msgs.msg` data type package.

```python
def talker():
```

Over the next few lines of code, we will define a `talker` node, and it's publishing capabilities.

```python
pub = rospy.Publisher("chatter", String, queue_size=10)
```

Here, we define a publisher node to publish messages to a desired topic. It declares that we want to publish to the `chatter` topic, using a `String` message, with a `queue_size` argument of 10. The `queue_size` limits the amount of queued messages for the case where a subscriber is not receiving the messages fast enough. Here we are queuing 10 messages.

```
rospy.init_node("talker", anonymous=True)
```

Now, we will initialize a ROS node in Python. The function `init_node` tells the ROS master the name of your node. The node will be Labeled as `talker`. We will also set `anonymous = true`. This allows the compiler to append numbers to the end of the node's name to ensure a unique node name (this is useful for more complex simulations).

```
rate = rospy.Rate(10) # 10hz
```

The ROS rate function, `rate()`, defines the speed at which we want to perform some task. Here we define a `rate` handle to maintain a desired speed of 10 Hz (0.1 seconds). Later, we will see that `rate` is used to define the amount of time that the system should sleep in the `while` loop.

```
while not rospy.is_shutdown():
    hello_str = "hello world {}".format(rospy.get_time())
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
```

This `while` loop structure is fairly standard in `rospy`. The `while` loop will begin to iterate after checking the `is_shutdown` flag. Generally, when we launch nodes, the `is_shutdown` flag is not activated, but when we terminate a node using `Ctrl-C` the `is_shutdown` flag is activated and the loop terminates.

Inside the loop, we first define a string message called `hello_str` that will contain the text "hello world". (Note: `%s` concatenates string messages, and `% rospy.get_time` prints the current time). Next, using the `loginfo()` function, the string message (`hello_str`) will be printed in the terminal, written to the node's log file, and written to rosout (used for debugging). Using `pub.publish()`, the string message is published to the `chatter` topic. Lastly, `rate.sleep()` is used to maintain a desired loop rate (we previously defined the loop rate as 10 Hz).

```
if __name__ == "__main__":
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

This snippet of code is where we make everything happen. It is the main block of the code which calls upon our previously set up definitions. The main block includes the `talker()` definition and also includes a `ROSInterruptException` to prevent the code from continuing to execute during `sleep()`.

Task 1: Set up a publisher for the ServoMsg message

Now you know how to publish a ROS message. Let's write our first ROS code! Open your `pure_pursuit.py` file in the text editor of your choice (file path: <Path of your repo>ECE346/ROS Core/src/Labs/Lab0/scripts/controller/pure_pursuit.py) Your first task is to set up a missing publisher in the function `setup_publisher` following instructions under **TODO**. Make sure you read through the code to get an understanding of variable names (e.g topic name). Once you are finished, show your code to the TA and proceed.

3.2 ROS Subscriber

The code for the subscriber is very similar to the publisher and can be seen below. Now, instead of publishing to the `chatter` topic, we are subscribing to it.

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo("I heard %s", data.data)

def listener():
    # Define listener node
    rospy.init_node("listener", anonymous=True)
    # Subscribe the listener node to chatter topic
    rospy.Subscriber("chatter", String, callback)
    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == "__main__":
    listener()
```

Listing 2: Code for subscriber

Let us now break down the subscriber code.

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
```

Exactly like the publisher code, we need to setup our Python script and import necessary packages.

```python
def callback(data):
    rospy.loginfo("I heard %s", data.data)
```

Now, we will define a `callback(data)` function. This function is used to process the received message data. Here, `loginfo()` is used to print to the terminal. The printed messages will begin with the text `"I heard"` and will then print out the message data received from the chatter topic (`data.data`). This function should make more sense when we discuss the subscriber.

```python
def listener():
    rospy.init_node("listener", anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    rospy.spin()
```

Here, we define the `listener` node, also known as the node that is subscribed to the `chatter` topic. First, the node is initialized so that the master knows the name our new node.

Next, the `listener` is defined as a subscriber. There are three important categories we need to specify when using the `Subscriber()` function. First, we need to declare the topic that we want to subscribe to. Here we are subscribing to the `chatter` topic. Second, we need to identify the data type of the ROS message. The data type for the message of the publisher and subscriber needs to be the same. Therefore the data type of the message will be a `String`. Third, we need to identify the name of the function where the message data will be sent. In our case, we are sending the message data to the `callback()` function.

The `spin()` function keeps the node active until it is manually shut down (`Ctrl-C`).

```python
if __name__ == "__main__":
    listener()
```

This last snippet of code is where we actually run the listener code.

Task 2: Set up a subscriber for the Odometry message

Open your `pure_pursuit.py` file. Your second task is to set up a missing subscriber in the function `setup_subscriber` following instructions under **TODO**. Once you are finished, show your code to the TA and proceed.

3.3 Inspecting ROS Messages using `rostopic` and `rosmsg`

Now you are an expert in setting up ROS publisher and subscriber. However, you may be wondering how to decode those ROS messages or figure out what's inside of each datatype in order to write a callback function. The command line tool `rostopic` and `rosmsg` are designed for this usage.

Let's try this out! First, make sure your Rviz is still running. Now, open a new terminal, activate our ROS environment (`conda activate ros_base`), setup the ROS environment (`source devel/setup.bash`), and try:

- `rostopic list`
  This will print the names of active topics

- `rostopic info <Topic Name>`
  This will information about a desired topic, including its datatype, publisher, and active subscribers.

- `rostopic echo <Topic Name>`
  This will print out ROS messages from a desired topic in your terminal

- `rostopic type <Topic Name> | rosmsg show`
  This will first look up the datatype of the topic, then print out its data structure.

A full list of rostopic and rosmsg functionalities can be found in their documentations.

Task 3: Fill in the subscriber callback function

Open your `pure_pursuit.py` file. Your third task is to fill in the missing code of the function `goal_callback` following instructions under **TODO**.
Once you are finished, **restart** `lab0_simulation.launch`. From the RViz simulator, you can add a desired goal location by selecting **2D Nav Goal** from the top panel and then clicking a point on the map. You will see that the position of your clicked point is printed on your terminal.

Task 4: Construct and publish a ROS message

Open your `pure_pursuit.py` file. Your fourth task is to fill in the missing code of the function `publish_control` following instructions under **TODO**. Once you are finished, show your code to your TA.

4 Goal Reaching Controller

In this Lab, you will implement a simple goal-reaching controller. We will use a proportional controller for the throttle, and a pure pursuit controller for steering.

4.1 Throttle Control

Our robot can control its acceleration through the motor's throttle input. In this Lab, we will implement a proportional controller to track reference speed V_{ref}.

$$a = K_p(V_{ref} - V_{robot}) \tag{1}$$

4.2 Steering Control

The pure pursuit method is a geometry-based algorithm to determine desired steering angle for a car to follow a path. As shown in Figure 3, pure pursuit calculates the steering angle δ to ensure the vehicle reaches the target point (**TP**) according to the kinematic bicycle model. This tutorial provides an excellent interactive explanation of the pure pursuit algorithm.
In short, you can obtain the steering angle δ by Equation 2, where L is the wheelbase of the robot, α is the relative angle of the look-ahead point w.r.t the robot, and l_d is the distance between the robot and the look-ahead point.

$$\delta = \arctan\left(\frac{2L\sin(\alpha)}{l_d}\right) \tag{2}$$

In this Lab, we assume the reference path is the straight line connecting your robot and goal point. Therefore, the **TP** is a point on this line segment defined by user parameters.

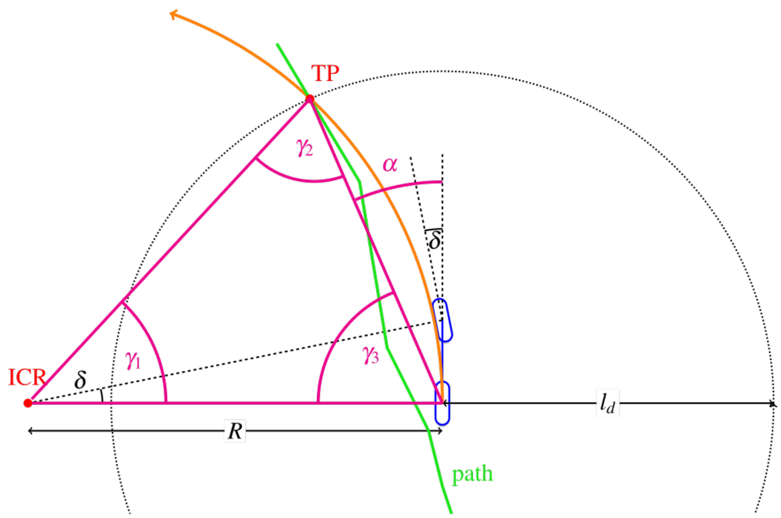


Figure 3: Geometric Interpretation of Pure-Pursuit Algorithm [source]

Task 5: Implement the goal reaching controller

Open your `pure_pursuit.py` file. You will finish the function `planning_thread` following the implementation details under the **TODO** block. This task concludes all coding parts of Lab 0. Re-launch the simulation, set **2D Nav Goal** as any points on Rviz, and drive your robot towards the goal point. The default parameter should work well in the simulation if your implementation is correct. **Show your simulation results to your TAs.**

5 Let's Get Real

The modularity of ROS allows us to quickly deploy our algorithms from the simulated environment into the real robot with minimal changes to your code.

Task 6: Try Out On Mini Truck

Follow the instructions in the *Intro to Mini Truck* tutorial and test your goal-reaching controller on the Mini Truck with the provided `lab0_truck.launch`. **Demo your robot to your TAs.**