Pre-Lab Tutorial: Intro to ROS

Welcome to the Robotics Assignment ("Lab") component of Intelligent Robotic Systems!

Over the semester, we will implement various methods for robot decision-making, both in simulated environments and on physical robotic hardware. In the upcoming "zeroth" lab, we will learn the key concepts of ROS (the Robot Operating System) to control the Mini Trucks that will accompany us throughout the class.

This is a Pre-Lab Tutorial to help you get ready for our labs. Although this material is not graded, we strongly encourage you to walk through it and try out all commands on your own computer before your first lab section. It will make things much smoother! The following are the objectives of this Pre-Lab Tutorial:

- Install ROS on your own computer.
- Get familiar with basic ROS concepts.
- Be able to build and run a provided ROS package.

Contents

1 Setting Up ROS

Before we get started, you need to set up the Git Repository and configure your computer for the lab. Please read through the detailed instructions here.

2 Intro to ROS

Here, we will go over some basic concepts of ROS (which stands for "Robot Operating System"). ROS is an open-sourced framework for controlling robotic components from a computer. You can generally think of ROS as a graph or network of independent **ROS nodes**. Each ROS node communicates via **ROS messages** by **publishing** and **subscribing** to **ROS topics**. Published messages will be received by any node in the graph subscribed to the corresponding **ROS topic**.

Most roboticists today create ROS software using either C++ (roscpp) or Python (rospy), as both languages are well-supported in the ROS community. In this class, our default working language is Python. However, you will find that it is simple to adapt ROS in the other language once you master one of them.

2.1 Key Concepts in ROS

- **ROS Master**
  The ROS Master serves as the central coordinator of the ROS system. It provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics and services.

- **ROS Nodes**
  A node is a process that performs some computation. For example, a robot with a control system will usually be comprised of many nodes. A robot can have many nodes, each meant to operate at a fine-grained scale. For example, a node for processing camera images, a node that controls the robot's motors, a node that performs localization, a node that performs path planning, etc.

- **ROS Message**
  Communications between ROS nodes are through ROS messages. A ROS message is a simple data structure, comprising integers, floating-point numbers, booleans, as well as arrays of those data types defined by a `.msg` file. Messages can include arbitrarily nested structures and arrays as defined by the user.

- **ROS Topics**
  ROS nodes communicate with one another by publishing and subscribing to topics that contain messages. The ROS topics provide ID to the correct channel of communication.

- **ROS Services**
  A ROS service is a type of message that allows two-way communication. This may be necessary

for robotics applications where you need to change the robot's mode and receive acknowledgment of receiving the request. This process can be accomplished by using ROS services which depends on `.srv` files.

- **Catkin Build System**
  A build system compiles source code and creates executable programs. ROS has a custom build system called catkin. Catkin was created to alleviate the complexity of using existing build tools.

- **ROS Catkin Workspace**
  A catkin workspace is a directory used to modify, build, and install multiple catkin packages. Anything we do with ROS will be inside of a catkin workspace.

- **ROS Catkin Package**
  A catkin package is a directory that contains source code for your ROS nodes, descriptions for your custom ROS messages and services, or other libraries used in ROS.

- **Parameter Server**
  A parameter server is a shared, multi-variate dictionary, which is accessible to all nodes to store and retrieve data by keys from at runtime.

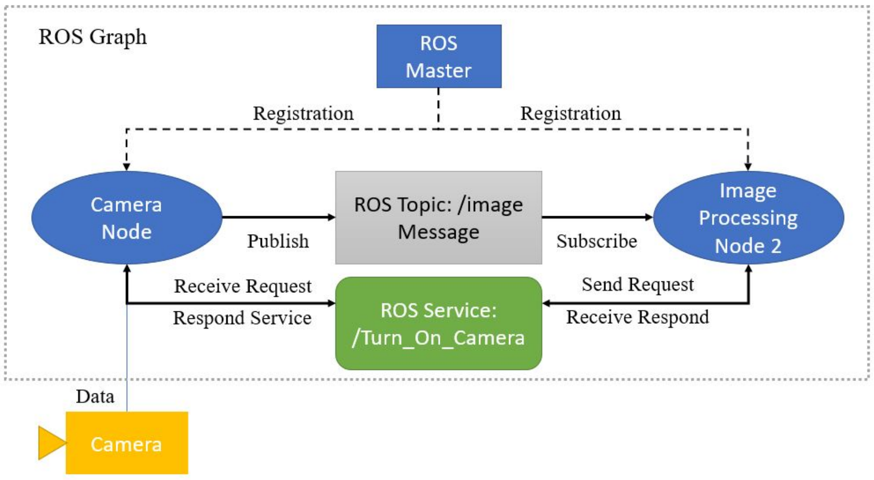


Figure 1: A ROS graph example containing two nodes

Putting those terms together, let us consider the example in Figure 1. We have a ROS master with two registered nodes: the camera node and the image processing node. The camera node talks to the camera through an API that is independent of ROS. It publishes an image message to the`/image` topic. The image processing node is subscribed to the `/image` topic and receives the image message for further processing. Additionally, let's assume the camera costs a lot of energy to run and should only be used when needed. We can use the ROS service `/Turn_On_Camera` to ask the camera node to turn on the camera. The image processing node can send this service request, and it begins computation once the camera sends back the response.

The main advantage of ROS is that it is a modular system. What if we want to have another process to do localization based on camera images? Using ROS, we can simply set up a new localization node that subscribes to the same `/image` topic, without the need to change code in the other two modules. Similarly, if we deploy the system to another robotic platform with a different camera, we just need to make sure the new camera node publishes the same type of image messages, and keeps using other downstream packages.

3 ROS Catkin Workspace

A catkin workspace is a directory (folder) that contains all catkin packages. You can think of this as the main folder that contains everything you need for a specific project or lab. A catkin workspace can store multiple catkin packages and allows us to build all of the catkin packages at the same time. We will learn more about the build process later in this reading, but we must build a workspace to convert source code into executable files. If you don't use a catkin workspace, you can build catkin packages independently, however, this can be very tedious when there are several packages that need to be built.

First, open a terminal and go into to your desired directory. Then, let's activate our ROS environment by

```
# Activate ros_base environment. Note: the name can vary
conda activate ros_base
```

3.1 Create a Catkin Workspace

Begin by creating a directory labeled `first_catkin_ws` with a source sub-directory labeled `src`:

```
# Create a workspace labeled catkin_ws with a src sub-directory
mkdir -p first_catkin_ws/src
```

While using the command `mkdir`, the `-p` flag allows us to create sub-directories simultaneously with the main directory.

Note: Alternatively, we can do the same thing in three separate steps:

```
# Create a directory labeled first_catkin_ws
mkdir first_catkin_ws
# Navigate indside the first_catkin_ws
cd first_catkin_ws
# Create a directory labeled src
mkdir src
```

The directory labeled `first_catkin_ws` is the catkin workspace. Within the catkin workspace, we have a directory labeled `src` which represents the source space and will be discussed later in the lab. The `src` directory is **necessary** to build code in the catkin workspace.

4 ROS Catkin Package

As previously mentioned, a catkin package refers to a directory that contains source code for ROS nodes, services, messages, etc. Catkin packages are located inside the catkin workspace. under the `src` sub-directory. A given project workspace will likely contain many packages inside `src`.

While ROS provides a tool to create a catkin package using the command `catkin_create_pkg`, we have provided a template package called `first_pkg` **in our Git Repo** under the catkin workspace within the `src` directory (`catkin_ws/src`) for this tutorial and also for your future labs.

Let's first inspect what's inside the `first_pkg` catkin package. A typical file tree of a catkin package can be seen in Figure 2.

```
first_pkg
 ├→CMakeLists.txt
 ├→Package.xml
 ├→src
 └→scripts
     └→first_node.py
   launch
     └→first_launch.launch
```

Figure 2: File tree for the `first_pkg` catkin package

The `launch` directory has launch files for the purpose of launching one or multiple nodes within the package.

The `scripts` directory holds executable python scripts that contain nodes and their dependency packages.

The `src` directory is for users who prefer to use C++. This is where C++ users will place their source code.

The `package.xml` is the manifest file containing metadata about a package. It includes the package's name, version, description, license information, dependencies, and other meta information like exported packages. The detailed requirements of a `package.xml` is listed in this documentation.

The `CMakeLists.txt` file is the input to the CMake build system for building software packages. In ROS, the `CMakeLists.txt` file tells ROS which files need to be built and what packages need to be linked. You are still required to have a `CMakeLists.txt` file, even if the entire ROS package is written in Python. Luckily, ROS has a very detailed template for the `CMakeLists.txt` file, and we will provide the proper files for you during this class. To learn more about this, please check out this documentation.

For now, we do not need to worry about `package.xml` and `CMakeLists.txt` as they will be provided for most of the packages in this class. Later in the semester, we will dive into those concepts while building more open-ended software for our final projects.

4.1 Building a Catkin Package

From Section 1 (Setup ROS), you should have downloaded the class GitHub repository. Using the terminal you can to navigate to the ECE346 directory (`ECE346`) and then navigate to the catkin workspace (`catkin_ws`).

Now, let's build the code in the catkin workspace using the ROS command `catkin_make`. The command `catkin_make` should be executed at the top level of your catkin workspace.

```
# Navigate to the top level of your catkin workspace
cd ECE346/catkin_ws
# Build catkin workspace
catkin_make
```

After using `catkin_make`, a lot of messages are printed in your console. If everything went well, you should see something similar to:

```
        .
        .
        .
        .
Configuring done
-- Generating done
-- Build files have been written to: XXXXX/catkin_ws/build
####
#### Running command: "make -j12 -l12" in "XXXXX/catkin_ws/build"
####
```

After using `catkin_make`, you should notice that your catkin workspace contains two new subdirectories labeled `build` and `devel` and a `CMakeLists.txt` file in the `src` directory. Because `build` and `devel` are generated by `catkin_make`, and you do not typically have to worry about them.

A catkin workspace can be broken down into three separate parts: source (`src`) directory, build (`build`) directory, and development (`devel`) directory. A typical file tree for a catkin workspace with a catkin package can be seen in Figure 3. Let's break down the purpose of the source directory, build directory, and development directory.

- **Source (`src`) Directory**
  The `src` directory is the home for all catkin packages. It contains all source code, and it is where we will soon create nodes using Python.
- **Build (`build`) Directory**
  The `build` directory is the location where CMake builds all of the code from the `src` directory. It keeps catkin and CMake's cache information and other intermediate files there.

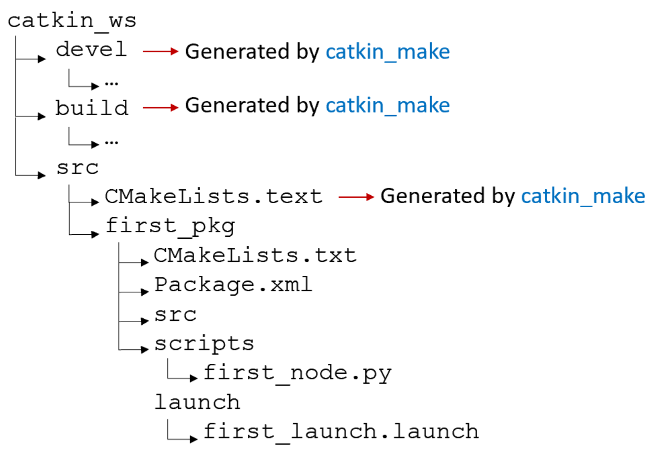


Figure 3: File tree for a catkin workspace with a catkin package after using `catkin_make`

- **Development (`devel`) Directory**
  The `devel` directory is the location for the built source code.

5 ROS Master and Node

Simply put, a ROS node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming messages through topics, sending requests through services, and setting values through Parameter Server. ROS nodes can be written in Python (rospy) or C++ (roscpp) using the ROS client libraries.

5.1 Making Node Executable

In the last section, we discussed how to build catkin packages using `catkin_make`. In order to add the workspace to the ROS environment you need to navigate to the top level of your catkin workspace `catkin_ws` and `source` the `setup.bash` file. For users of *zsh* (default for Mac OS), you should use file `devel/setup.zsh` instead.

```
# Navigate to the top level of your catkin workspace (eg: catkin_ws)
cd ECE346/catkin_ws
# Add workspace to ROS environment
source devel/setup.bash
```

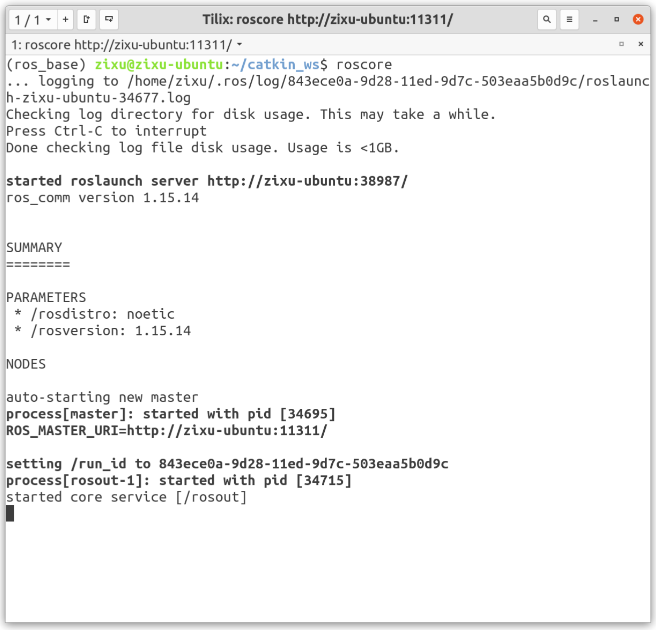
Important: You need to use the `catkin_make` command every time you add a new catkin package to your catkin workspace, define a new message type or build a new service for Python users. However, it is good practice to use `catkin_make` whenever you make any changes to elements in the catkin workspace. You will also need to do `source devel/setup.bash` every time you open a new terminal window to use your packages under the workspace.

5.2 Running ROS Nodes with `rosrun`

In `first_pkg`, the `scripts` directory has a Python script for the ROS node (`first_node.py`). To run this node, let us first open a ROS Master by

```
# Start a ROS Master
roscore
```

Figure 4 is an exemplary output you will see from your terminal.


1 / 1
Tilix: roscore http://zixu-ubuntu:11311/
1: roscore http://zixu-ubuntu:11311/

```
(ros_base) zixu@zixu-ubuntu:~/catkin_ws$ roscore
... logging to /home/zixu/.ros/log/843ece0a-9d28-11ed-9d7c-503eaa5b0d9c/roslaunc
h-zixu-ubuntu-34677.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://zixu-ubuntu:38987/
ros_comm version 1.15.14


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.15.14

NODES

auto-starting new master
process[master]: started with pid [34695]
ROS_MASTER_URI=http://zixu-ubuntu:11311/

setting /run_id to 843ece0a-9d28-11ed-9d7c-503eaa5b0d9c
process[rosout-1]: started with pid [34715]
started core service [/rosout]
```

Figure 4: Example of `roscore` output

Then, we open a new terminal, activate `ros_base` environment, navigate to your catkin workspace, and add environment variables.

```
# Activate ros_base
conda activate ros_base
# Navigate to the top level of your catkin workspace
cd catkin_ws
# Add ROS env variables
source devel/setup.bash
```

Now, in this terminal, we can run our nodes by

```
# run the node
# rosrun <pkg_name> <node_file>
rosrun first_pkg first_node.py
```

You should see 'Hello World' printed out continuously. To stop the ROS process, you need to first terminate your node (`ctrl-C`), then turn off your ROS Master (`ctrl-C`).

5.3 Running ROS Nodes with `roslaunch`

This is so cumbersome! However, recall that the `first_pkg` (Figure 2) also contains a `launch` directory with the file (`first_launch.launch`). The key idea here is to start the ROS Master, run one or multiple nodes, and handle parameters using one single command. We will learn how to create our own `.launch` files in future labs. Using `roslaunch` let's run our nodes without opening hundreds of terminals.

```
# Launch our first node
# rosrun <pkg_name> <launch_file>
roslaunch first_pkg first_launch.launch
```

You should see exact same results you saw with `rosrun`. When finished, press `ctrl-C`, and all processes will be terminated. A detailed guide of the launch file can be found here.

6 References and Additional Materials

Over the previous sections, we have covered a tiny portion of what ROS offers. Throughout the semester, we will learn more topics while implementing exciting algorithms on robots. We also encourage you to go over some of the excellent ROS tutorials and examples available online. You may find these materials very useful for gaining a deeper and more advanced understanding of ROS. Below are a few pointers to get you started on your ROS journey.

- Official ROS documentation
- ROS tutorial from Clearpath Robotics
- ROS lecture notes from ME495 at Northwestern University
- A Gentle Introduction to ROS by Jason M. O'Kane

If you encounter issues, bugs, or unsolvable puzzles, your best helper is always Google (much wiser than any one of us). If your questions are ROS-related or you are unsure how to achieve some advanced features, you can check ROS Answers, where you are most likely to find solutions to your problems.