Introduction to ROS

Prof. V. Muthukumar

ROS IN MOTION

Install Ubuntu 20.04

- This course currently uses ROS 1: Noetic that runs on Ubuntu 20.04.

- It’s recommended that you have Ubuntu 20.04 installed in your computer in dual boot mode. Virtual machines are not recommended.

- Follow instructions here to prepare and install Ubuntu on existing Windows 10/11 system
  - https://www.xda-developers.com/dual-boot-windows-11-linux/

- We’ll refer to this system as the “Host System”

Install ROS

- Refer to this website for instruction to install ROS1:Noetic on Ubuntu.
  - http://wiki.ros.org/noetic/Installation/Ubuntu

- Setup your sources.list

-
  ```
  > sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
  ```

- Set up your keys

  -
    ```
    > sudo apt install curl # if you haven't already installed curl
    ```

  -
    ```
    > curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
    ```

- Update Repositiories

  -
    ```
    > sudo apt update
    ```

- Install ROS Desktop-Full: (Recommended) : Everything in Desktop plus 2D/3D simulators and 2D/3D perception packages

  -
    ```
    > sudo apt install ros-noetic-desktop-full
    ```

Getting info on your Ubuntu

- lsb_release
  - lsb_release command provides certain LSB (Linux Standard Base) and distribution- specific information. If no options are given, the -v option is assumed.

- uname -a
  - prints the name, version and other details about the current machine and the operating system running on it. Also, provides the kernel your are running.

- ipconfig –a
  - ifconfig (interface configuration) is a network management tool. It is used to configure and view the status of the network interfaces in Linux operating systems. Use to get your ip adds.

- lsusb
  - The lsusb command in Linux is used to display the information about USB buses and the devices connected to them. The properties displayed are speed, BUS, class, type details, etc.

- dmesg | grep -i usb
  - dmesg (diagnostic messages) is a command on most Unix-like operating systems that prints the message buffer of the kernel. Here we get the boot up message that contains the word “USB”.

ROS IN MOTION

Initial ROS Environment Setup

- Check if ROS is properly installed
  - > ls /opt/ros/noetic
  - > dpkg -l | grep ros-
- Source the default ROS packages
  - Bash shell
  - > echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
  - > source ~/.bashrc

  - Zsh Shell
  - > echo "source /opt/ros/noetic/setup.zsh" >> ~/.zshrc
  - > source ~/.zshrc
- Anytime you open a terminal ~/.bashrc or ~/.zshrc is executed
- Install ROS dependencies
  - > sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential
  - > sudo rosdep init
  - > rosdep update

Making shell commands easy

- I prefer zsh and have installed a zsh addon called “oh-my-zsh’
- Follow instructions here at @ https://ohmyz.sh/
- Customize the pulgins @
  - https://github.com/ohmyzsh/ohmyzsh/wiki/Plugins
- Must have plugins
  - autosuggestions suggests commands as you type based on history and completions.
  - syntax-highlighting Fish shell-like syntax highlighting for ZSH
- Follow some examples @ t.ly/c8J9t

Creating your ROS workspace

- You can have multiple workspaces and work with them together as far as their no duplicates or conflicts between the workspaces.

- ROS packages are managed and compiled using catkin
  - catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow.

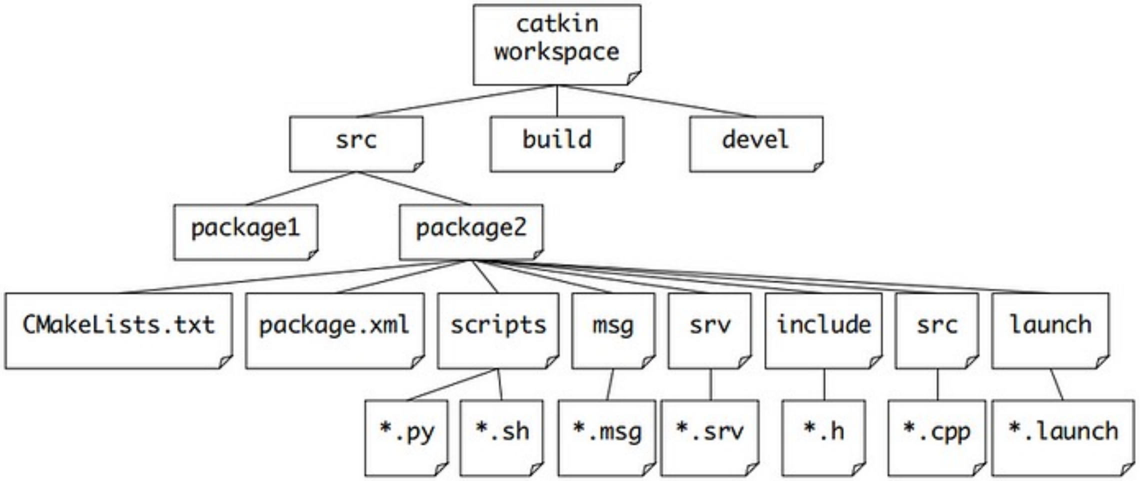
- # MAKE THE CATKIN WORKSPACE DIRECTORY
  - $ mkdir -p ~/catkin_ws/src
  - $ cd ~/catkin_ws/
  - $ catkin_make

- # SOURCE THE devel/setup.bash or devel/setup.zsh FILE FROM THE CATKIN WORKSPACE
  - > source devel/setup.bash (or)
  - > source devel/setup.zsh (or)

- Add the packages in your catkin workspace to the default ROS packages
  - > echo "source /home/user/catkin_ws/devel/setup.bash" >> ~/.bashrc
  - echo "source /home/user/catkin_ws/devel/setup.zsh" >> ~/.zshrc

- # CREATE THE ROS PACKAGE
  - > cd src/
  - > catkin_create_pkg **package2** std_msgs rospy roscpp

- This will create multiple folders under the folder **package2**



- # Compile the empty package2
  - > catkin_make

- # update the package repository using
  - > source ~/.bashrc or
  - > source ~/.zshrc

Let's create or first real empty package

- Always get to this directory
  - > cd /home/user/catkin_ws/src
- Create a package called empty node with dependencies (cpp and python)
  - > catkin_create_pkg **empty_pkg** std_msgs rospy roscpp
- # Create a cpp file under the folder “src” (using any editor)
  - > vim empty_node.cpp
  - See the code explanation in the next slide
- # Create a python file under the folder “script” (using any editor)
  - > vim empty_node.py
  - See the code explanation in the next slide

```
#include "ros/ros.h"
#include <ros/package.h>

int main(int argc, char* argv[])
{
    // Initialise the node
    ros::init(argc, argv, “empty_cpp_node");
    // Start the node by initialising a node handle
    ros::NodeHandle nh("~");
    // Display the namespace of the node handle
    ROS_INFO_STREAM(“EMPTY CPP NODE] namespace of nh =
                                \ " << nh.getNamespace());
    // Spin as a single-threaded node
    ros::spin();
    // Main has ended, return 0
    return 0;
}
```

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy

if __name__ == '__main__':
    # Initialise the node
    rospy.init_node(“empty_py_node")
    # Display the namespace of the node handle
    rospy.loginfo(“EMPTY PY NODE] namespace of node =
                                                        \" + rospy.get_namespace());
    # Spin as a single-threaded node
    rospy.spin()
```

Fixed the CmakeLists.txt file

- Edit the CmakeLists.txt file under the current package (**empty_pkg**)
  - > vim CmakeLists.txt

- # add the following lines
  - add_executable(empty_cpp_node src/empty_node.cpp)
  - add_dependencies(empty_cpp_node ${catkin_EXPORTED_TARGETS})
  - target_link_libraries(empty_cpp_node ${catkin_LIBRARIES})

- # Make the python files executable
  - > chmod a+x empty_node.py

- # Compile, update environment
  - > catkin_make
  - > source ~/.zshrc (alias zsrc)

Run, launch, and interrogate nodes

- We use terminator (support multiple windows. Can be installed by
  - > sudo apt install terminator

- In each sub-terminal input the following commands
  - > roscore
  - > rosrun empty_pkg empty_cpp_node
  - > rosrun empty_pkg empty_py_node
  - > rosnode list

- # Explain the outputs of the terminal #4.

Let’s also create a launch file

- # Create a launch file
  - > cd catkin_ws/src/empty_pkg
  - > mkdir launch
  - > cd launch
  - > vim empty.launch
  - > chmod a+x empty.launch

- # type the following in the file

```
<launch>
  <node name="empty_cpp_node" pkg="empty_pkg"
            type="empty_cpp_node" output="screen"/>
  <node name="empty_py_node" pkg="empty_pkg" type="empty_py_node.py"
output="screen"/>
</launch>
```

- # Compile, update environment

- # Execute the launch file (make sure the pervious windows are closed, no need to run roscore)
- > roslaunch empty_pkg empty.launch

Periodic execution using loop_rate

- # Users can use loop_rate in the main code of cpp

```
// Initialise the ROS rate variable
   float loop_frequency_in_hz = 2.0;
   ros::Rate loop_rate(loop_frequency_in_hz);
   // Intialise a counter
   uint counter = 0;
   // Enter a while loop that spins while ROS is ok
   while (ros::ok)
   {
      counter++;
      // Display the current counter value to the console
      ROS_INFO_STREAM("[EMPTY CPP NODE] counter = \
                                                                                                   " << counter);

      // Spin once to service anything that need servicing
      ros::spinOnce();
      // Sleep at the loop rate
      loop_rate.sleep();
   }
```

Periodic execution using loop_rate

- # Users can use loop_rate in the main code of py file

```
# Initialise the ROS rate variable
 loop_frequency_in_hz = 2.0;
 loop_rate = rospy.Rate(loop_frequency_in_hz);
 # Intialise a counter
 counter = 0;
 # Enter a while loop that spins while ROS is ok
 while not rospy.is_shutdown():
   counter += 1
   # Display the current counter value to the console
   rospy.loginfo("[EMPTY PY NODE] counter = \
                                              " + str(counter))
   loop_rate.sleep()
```

Periodic execution thro’ callback of the cpp nodes

- # In the empty_node cpp file add the following before ros::spin();

```
 // Initialise a timer
   float timer_delta_t_in_seconds = 0.5;
   m_timer_for_counting = nh.createTimer(ros::Duration \                    (timer_delta_t_in_seconds),
timerCallback, false);
```

- # Create a timer and callback function

```
// Declare "member" variables
ros::Timer m_timer_for_counting;


// Declare the function prototypes
void timerCallback(const ros::TimerEvent&);


// Implement the timer callback function
void timerCallback(const ros::TimerEvent&)
{
   static uint counter = 0;
   counter++;
   // Display the current counter value to the console
   ROS_INFO_STREAM("[EMPTY CPP NODE] counter = " << counter);
}
```

Periodic execution thro’ callback of the py nodes

- # In the empty_node python files add the following before rospy.spin();

```
        # Start an instance of the class
          empty_py_node = EmptyPyNode()
```

- # Create a timer and callback function

```
class EmptyPyNode:

  def __init__(self):
    # Initialise a counter
    self.counter = 0
    # Initialise a timer
    timer_delta_t_in_seconds = 0.5;
    rospy.Timer(rospy.Duration(timer_delta_t_in_seconds),\            self.timerCallback)

  # Respond to timer callback
  def timerCallback(self, event):
    self.counter += 1
    # Display the current counter value to the console
    rospy.loginfo("[EMPTY PY NODE] counter = " + str(self.counter))
```

Minimalistic Publisher

```
#include <ros/ros.h>
#include <std_msgs/Float64.h>

int main(int argc, char **argv) {
   ros::init(argc, argv, "minimal_publisher”);
   ros::NodeHandle n
   ros::Publisher my_publisher_object =                                    n.advertise<std_msgs::Float64>("topic1", 1);
   //"topic1" is the name of the topic to which we will publish
   // the "1" argument says to use a buffer size of 1; could make larger, if expect network backups

   std_msgs::Float64 input_float; //create a variable of type "Float64",

   input_float.data = 0.0;

while (ros::ok())
   {
      input_float.data = input_float.data + 0.001;
      //increment by 0.001 each iteration
       my_publisher_object.publish(input_float);
      // publish the value--of type Float64--
       //to the topic "topic1"
   }
}
```

Minimalistic Subscriber

```
#include<ros/ros.h>
#include<std_msgs/Float64.h>
void myCallback(const std_msgs::Float64& message_holder)
{
  ROS_INFO("received value is: %f",message_holder.data);
}

int main(int argc, char **argv)
{
  ros::init(argc,argv,"minimal_subscriber");
  ros::NodeHandle n;
  ros::Subscriber my_subscriber_object=   n.subscribe("topic1",1,myCallback);

  ros::spin();
  return 0;
}
```

Running Minimalistic Publisher & Subscriber

Minimalistic Periodic Publisher

```
#include <ros/ros.h>
#include <std_msgs/Float64.h>

int main(int argc, char **argv) {
    ros::init(argc, argv, "minimal_publisher2");
    ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
    ros::Publisher my_publisher_object = \
                          n.advertise<std_msgs::Float64>("topic1", 1);

    std_msgs::Float64 input_float; //create a variable of type "Float64",

   ros::Rate naptime(1.0); //create a ros object from the ros “Rate” class;

    input_float.data = 0.0;

while (ros::ok())
    {
        input_float.data = input_float.data + 0.001; //increment by 0.001 each iteration
        my_publisher_object.publish(input_float); // publish the value--of type Float64
        naptime.sleep();
    }
}
```

Running Periodic Minimalistic Publisher & Subscriber