**Install ROS Kinetic on Ubuntu 16.04**

The current system is setup to run with ROS Kinetic however ROS repos can work are developed in a way to allow for it to work with ROS Melodic as well.

Reason for choosing Kinetic because the System was built with Ubuntu 16.04 (the stable version at the time) and Melodic was in “beta”.

To install ROS kinetic on your system, follow the instruction on the ROS website: <http://wiki.ros.org/kinetic/Installation/Ubuntu>

Install the Desktop-Full to get all the ROS libraries and stimulation tools Follow along the reset of the instructions on that page to setup the ROS environment.

**NOTE:** **“sudo apt-get update”** and **“sudo apt-get upgrade”** are two commands that will update/upgrade all the ROS repos and libraries to the latest one. It is important to run these occasionally, to keep your ROS environment update.

**ROS Tutorials**

Get the ROS tutorials by typing

**“sudo apt-get install ros-kinetic-ros-tutorials”**

or by going to <http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem> and follow along.

**ROS Workspace**

<http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment>

Make a directory called “catkin\_ws/src” using the command

**“mkdir -p ~/catkin\_ws/sr”**

This is where you will store all your ROS files. You can compile all the files within that directory by doing

**“ cd ~/catkin\_ws/ “**

**“ catkin\_make “**

Once you have run the above commands you will see that there are now two folders in catkin\_ws called “devel” and “build”. The devel folder has all the setup files for the nodes you have/will write in future. To ensure your bash files are up-to-date with “new” setup files after a catkin\_make you need to source your devel folder. Run the following command in the current directory to source your devel directory to bash.

“ source devel/setup.bash “

**Creating a Package**

<http://wiki.ros.org/ROS/Tutorials/CreatingPackage>

A Catkin Package must contain an XML with the metadata about the package that is the dependencies or the message types, complier, libraries etc. that the package depends on. Package must also have a CMakeList.txt file. Both will be auto generated for you when you run the following command.

**“ catkin\_create\_pkg my\_package std\_msgs rospy roscpp “**

The above code describes the package name (my\_package), the dependencies for the package (std\_msgs, rospy, roscpp).

Looking up Dependencies

Once created a package try to look up the dependencies of the package by using

**“ rospack depends1 <package\_name> “**

This will list out all the depends defined in the “catkin\_create\_pkg” command. Check to see if they match.

Certain dependencies depend on other things for example try

**“ rospack depends1 rospy “**

This will list out all the ROS arguments (message type, complier, libraries etc) “rospy” depends on in the package.

There are additional ways to customize the Package by adding descriptions, licensing tag, dependencies tags, etc. all of which can be added to the package.xml to make your file system more approachable.

**Building a ROS Package**

<http://wiki.ros.org/ROS/Tutorials/BuildingPackages>

Now that you have created your package and defined its dependencies its time to Build the Package.

Building the Package essentially looks for all the dependencies you requested, checks if they are installed and associates them to the Package.

NOTE: To the Build package return to the root directory of the workspace (that is catkin\_ws)

**“ cd ~/catkin\_ws/ “** // returns to catkin\_ws directory

In this directory you should see a “src” folder which is where all your packages should be created or moved.

**“ catkin\_make “**

This will build all the packages in the src folder with appropriate dependencies and modified package.xml and Cmakelist.txt files.

Now in catkin\_ws directory perform a “ ls “ command, which will list out all the files present in the current directory.

**“ ls “**

You should see 3 folders – build, devel, and src. We discuss what is in src folder,

1. **“build”** is your build space where you will store your and call your **“cmake and/or make”** commands to build your modified packages.
2. “**devel**” is where you will store all your executables and libraries.  
   NOTE: you need to move your executables and libraries before installing the package, so they can be sourced properly.

**Running ROS**

<http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes>

Let’s install some simple ROS programs:

**“ sudo apt-get install ros-kinetic-ros-tutorials “**

This will install a simple turtle-sim program that we will use to for the next few labs.

Lets assume you have created a new package with appropriate dependencies, executables, libraries, Makefiles, etc. and want to test it the first thing you want to do is compile your workspace or in other works more relevant to ROS you need to “make” your workspace so that all the packages are build properly.

**“ cd ~/catkin\_ws/ ”**

**“ catkin\_make ”**

From the previous lab we know what these two commands will do

Now open a separate terminal and start ROS

(in new terminal)

**“ roscore ”**

You will see a bunch of text fly by on your screen this is ROS setting up its environment and starting a “MASTER” that can be used to control different machines, servers, programs. “roscore” is like ROS’s own executable that will setup a service for all ROS nodes to communicate to.

**rosnode**

Open another terminal and see all the nodes that are currently communicating with the MASTER

**“ rosnode list ”**

You should only have one node working called “/rosout” since no other programs running. This is type of ros\_comm or a means for all the services to communicated with MASTER.

**“ rosnode info /rosout ”**

This will give a detailed information about the node – publication, subscription, and services.

A full list of all the ros\_comm can be found here <http://wiki.ros.org/ros_comm?distro=kinetic>

**rosrun**

Now let’s run a node from the tutorials we installed.

ROS allows us to run a specific node in a package directly from the command line using the “rosrun” command. In a new terminal try the following

**“ rosrun turtlesim turtlesim\_node ”**

This would open a new window the turtle stimulator.

Let’s dissect the rosrun command – the first argument “turtlesim” is the [package\_name] and the second argument “turtlesim\_node” is the [node\_name] that we want to run. Now if you type in “rosnode list” you should see two nodes running. What are they? Do they make sense?

You can rename the node so that its easier to read follow using

“ rosrun [package\_name] [node\_name]\_name := [name\_of\_node\_you\_like] ”

Try changing the name of the node from turtlesim\_node to “my\_turtle” and do a “rosnode list” to see if it worked.

**RQT\_Graph and ROS Topics**

<http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics>

With turtlesim running start another node called “**turtle\_teleop\_key**” you should know how to do this from the previous lab. This node allows you to control the turtle with your keyboard

**RQT\_GRAPH**

This is tool to graph all the interaction between different nodes and what type of messages they use. Lets first install RQT\_graph if its not already installed

**“ sudo apt-get install ros-kinetic-rqt ”**

**“sudo apt-get install ros-kinetic-rqt-common-plugins ”**

Now in new terminal run the following

**“ rosrun rqt\_graph rqt\_graph ”**

This will open a window with a flow diagram of the communication between different nodes. You should see the turtle\_teleop\_key and turtlesim nodes communicating with each other over “cmd\_velocity” which a message type used by the them to send and receive data to move the turtle.

**ROS TOPIC**

You can look at all the messages that are being sent to and from this node using the **“rostopic”** command.

**“ rostopic -h ”**

This will list out all the different types of arguments you can pass to rostopic. Some of the most useful once are echo, list, pub, find, info, and type. Let’s look at some of them in detail

1. ***ECHO:***  
   **“ rostopic echo [topic\_name] ”  
   “ rostopic echo /turtle1/cmd\_vel “**  
   Echo will publish the data being send back and forth on the terminal (works similar to have print statements inside the code). If you have rqt\_graph open you should also see a new entity being made with name “/rostopic\_####\_####... ” this indicates that the teleop\_turtle is publishing to two places (turtlesim and terminal)
2. ***LIST:*“ rostopic list -[argument] ”  
   Argument = -h – HELP, -v – VERBOSE (list of all the sub and pub messages)**Lists out all the publication and subscriptions for the current session
3. ***TYPE:***“ rostopic type [topic] ”  
   “ rostopic type /turtle1/cmd\_vel ”  
     
   This command used to return the type of message type “cmd\_vel” is using. In this case the return should be **geometry\_msgs/Twist.** You can see the details of the message type by:  
     
   **“ rosmsg show geometry\_msgs/Twist ”**This will show that this is 2 vectors – Linear and Angular each with x, y, z. A complete list of the common (std\_msgs) message type can be found here: <http://wiki.ros.org/std_msgs>
4. ***PUB:***  
   **“ rostopic pub [topic] [msg\_type] [args] ”  
   “ rostopic pub -1 /turtle1/cmd\_vel geometry\_msgs/Twist “**  
     
   The “-1” in the command above tells ROS to publish this command once and exit out of it and the control back to original publisher. Otherwise the program will wait until you a manual command is put in again. A better way to do this is type the command until the msg\_type (geometry\_msgs) and hit tab to fill the arguments and use arrow keys to navigate to argument you want to change and then hit enter.