WSL (Windows Subsystem for Linux) / ROS (Robot Operating System) Tutorial

ROS in WSL

Zhen Liu

Fall 2018

Lawrence Technological University
Computer Science

Contents

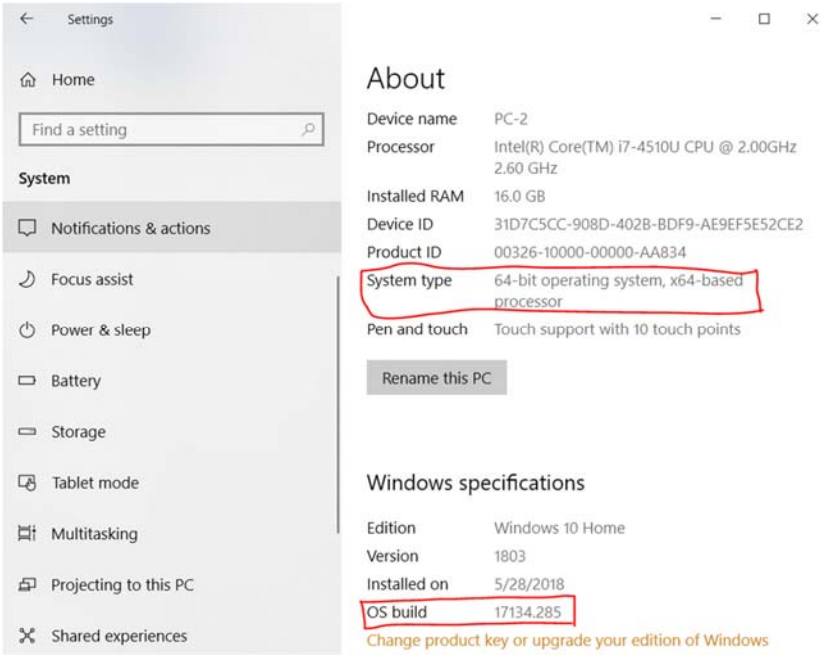
Install Ubuntu for Windows

The wonderful Ubuntu terminal is freely available for Windows 10. The Ubuntu terminal for Windows has many of the same features you'll find using the terminal on Ubuntu:

- Unrivalled breadth of packages, updates and security features
- Bash shell environments without virtual machines or dual-booting
- Run native tools such as SSH, git, apt and dpkg directly from your Windows computer
- Invoke Windows applications using a Unix-like command-line shell
- A huge community of friendly, approachable users

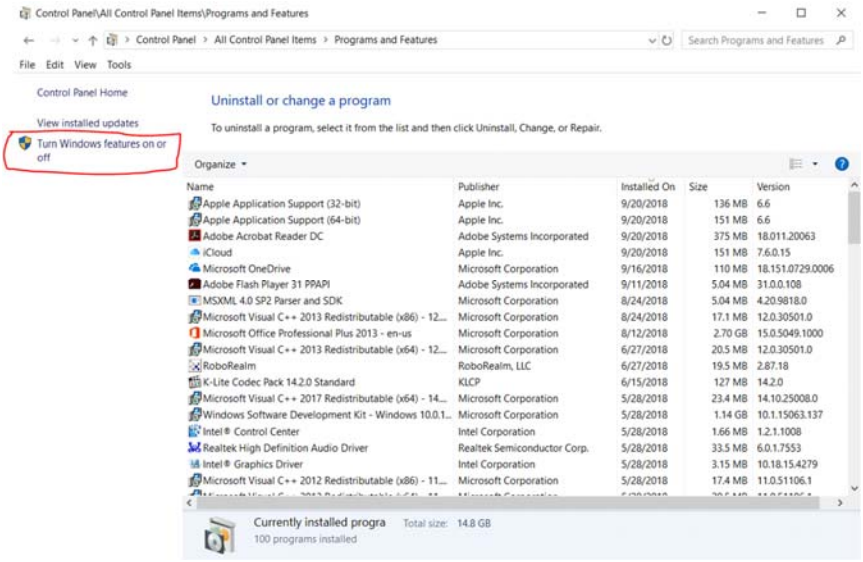
Before installing Ubuntu, you need to update Windows 10 to include the Windows 10 Fall Creator update, released October 2017 (Windows build 16215 or later). This update includes the Windows Subsystem for Linux (WSL) which is needed to run the Ubuntu terminal. To check your build, please follow these steps:

1) Open **Setting** > **System** > **About**
2) Look for the OS Build and System Type fields.

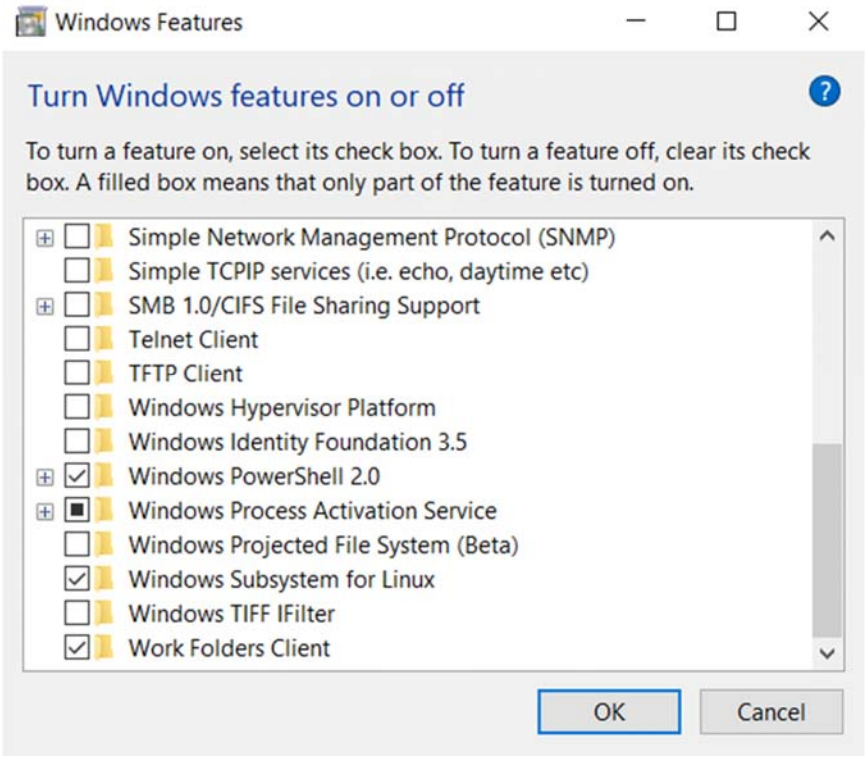


Also, you must make sure the Windows Subsystem for Linux optional feature is enabled.

1) Open **Control Panel**.
2) Click **Programs and Features** link.
3) Click **Turn Windows features on or off** link.



4) On “Windows Features”, check **Windows Subsystem for Linux** option.



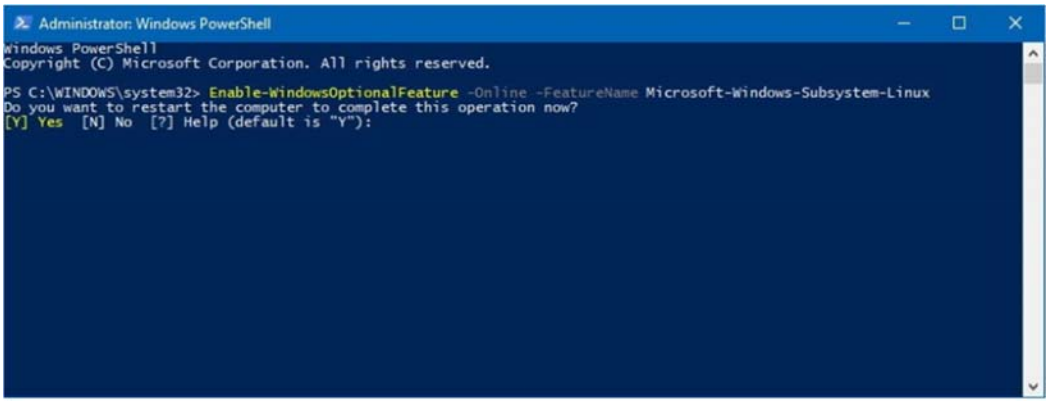
5) Click **OK**.
6) Click **Restart now**.

If you prefer using the command line method, you can also install Windows Subsystem for Linux using PowerShell.

1) Search for **PowerShell**, right-click the result, and click **Run as administrator**.
2) Type the following command to add the required module and press **Enter**:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

3) Type Y to complete the installation and restart your computer.



After adding the Windows Subsystem for Linux module to your computer, you can install Ubuntu from the Windows Store.

1) Search for **Microsoft Store** application from start menu and launch it.
2) Search for **Ubuntu 16.04**, published by Canonical Group Limited. (Make sure that you install 16.04 version instead of 18.04; Otherwise, when you install ROS, you will encounter "keyserver receive failed: No dirmngr" error.
3) Click on the **Install** button.

Ubuntu will be downloaded and installed automatically. Progress will be reported within the Microsoft Store application.

Once the installation completes, launch the app for the first time, Ubuntu will inform you that it's installing and you will need to wait for a few minutes. When complete, you will be asked for a username and password specific to your Ubuntu installation. These doesn't need to be the same as your Windows 10 credentials. With this step complete, you will find yourself at the Ubuntu bash command line.

Ubuntu install of ROS Kinetic

There is more than one ROS distribution you can choose. Some are older release with long term support, making them more stable, while others are newer with shorter support times, but with options for more recent platforms and more recent versions of ROS packages. I recommend the ROS Kinetic Kame version, since ROS Kinetic supports Xenial (Ubuntu 16.04).

Install ROS

You can follow the official [ROS installation guide for Ubuntu] by the word.

1) Set up computer to accept package form ros.org.

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

2) Set up keys

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 42
1C365BD9FF1F717815A3895523BAEEB01FA116
```

3) Installation

```
sudo apt-get update
```

There are many different libraries and tools in ROS. You can install them individually, but we recommend you to install the default configurations- the full installation.

Desktop-Full Install: (Recommended) : ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception

```
sudo apt-get install ros-kinetic-desktop-full
```

Depending on the speed of your system, this step could take a couple of hours, so you might want to go for a walk once you kick-off the install process.

4) Initialize rosdep

```
sudo rosdep init
rosdep update
```

5) Environment setup

If you are at this step, you have an installed version of ROS on your system. If you want to source ROS kinetic automatically for your bash session every time when a new shell is launched, you can do this with the following command:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

6) Get rosinstall

Now, let's install a command tool that will help us install other packages with a single command. To install this tool on Ubuntu, run the following command:

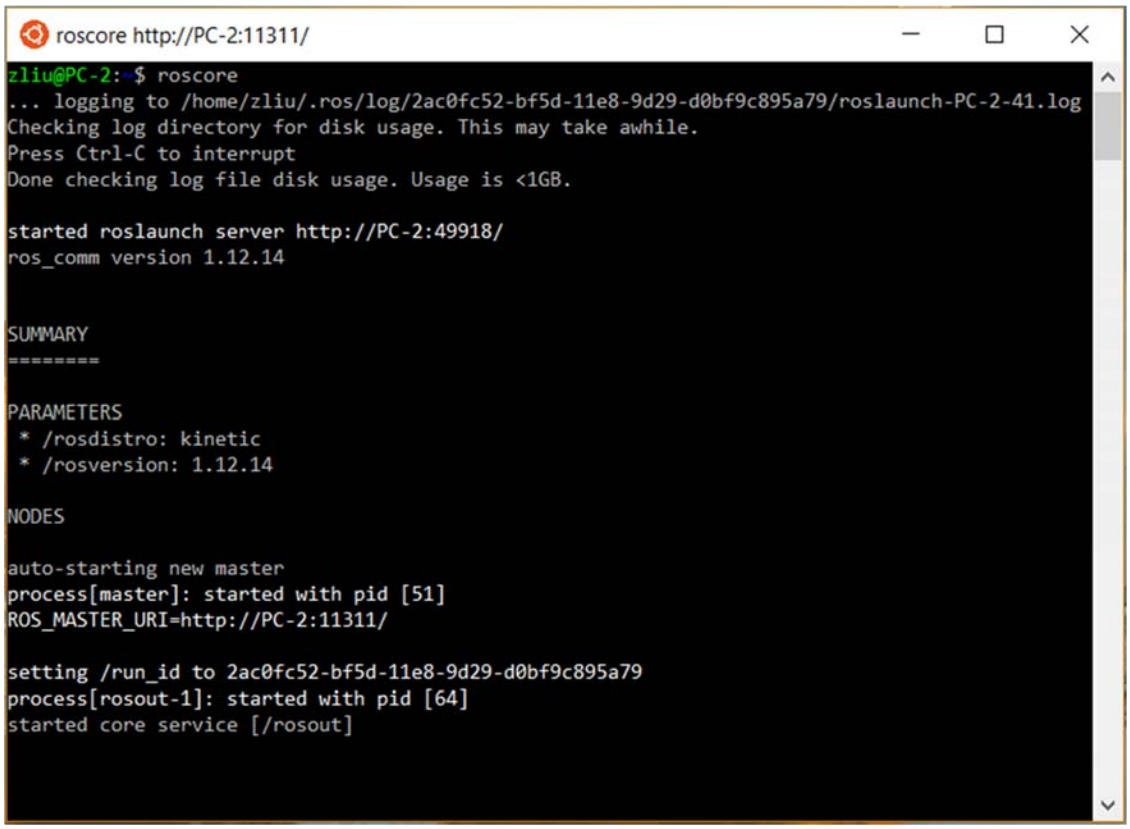
```
sudo apt-get install python-rosinstall
```

Or you can install this tool and other dependencies for building ROS packages, run

```
sudo apt-get install python-rosinstall python-rosinstall-generator python-wsto
ol build-essential
```

Run Simple Test

Up till now we have installed what's needed to run the core ROS packages. To check whether you have ROS installed correctly, you can type `roscore` in the shell:



Or if you want to check the ROS version installed, you can run the following two commands:

```
rosversion -d

echo $ROS_DISTRO
```

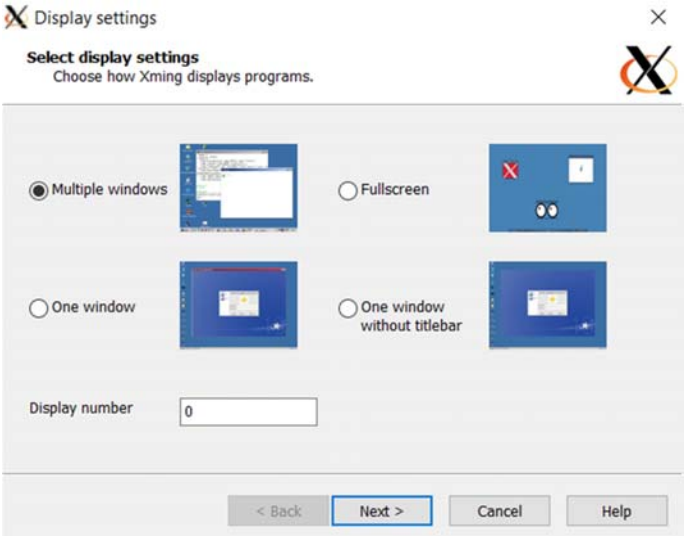
Now let's run popular turtle_sim to test whether ROS has been installed successfully.

To run applications with graphical output, we need to install an X server on Windows. In this aspect, Xming did a great job. Go to SOURCEFORGE, click Xming link, then click 6.9.0.31 folder link, download the setup.exe file and install it.

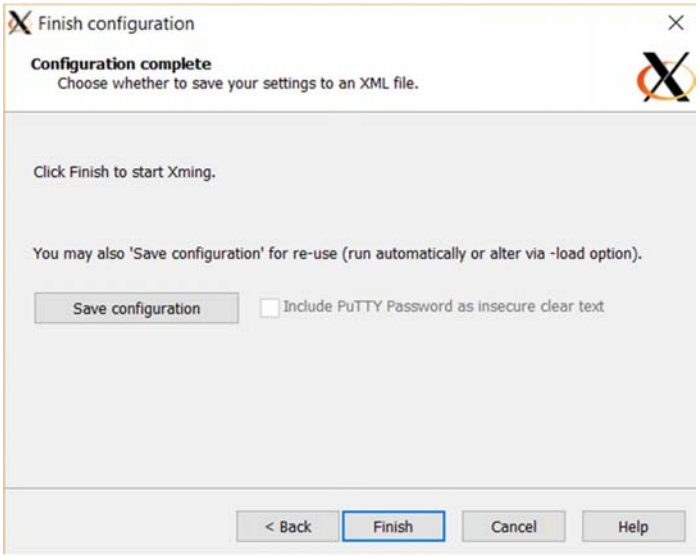
After installing Xming, we also need to configure WSL to use it. Run the following commands to modify .bashrc:

```
echo ''export DISPLAY=:0'' >> ~/.bashrc
source ~/.bashrc
```

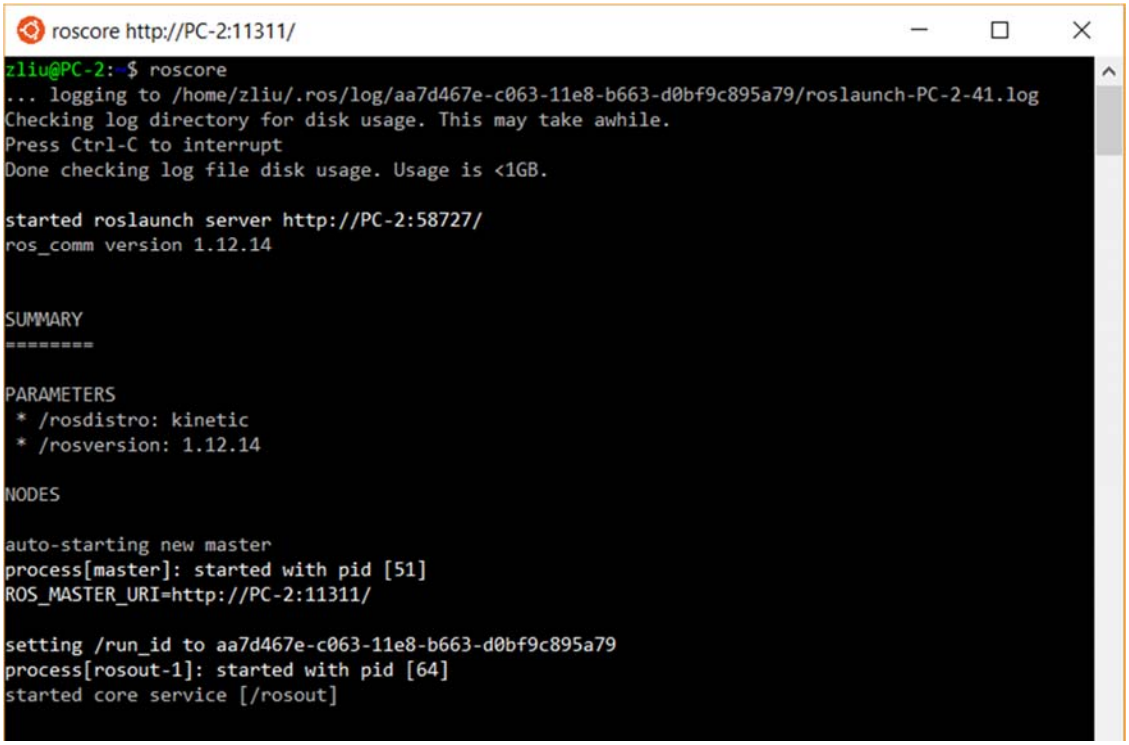
Then launch the Xming application from the start menu, and follow instructions to configure Xming.



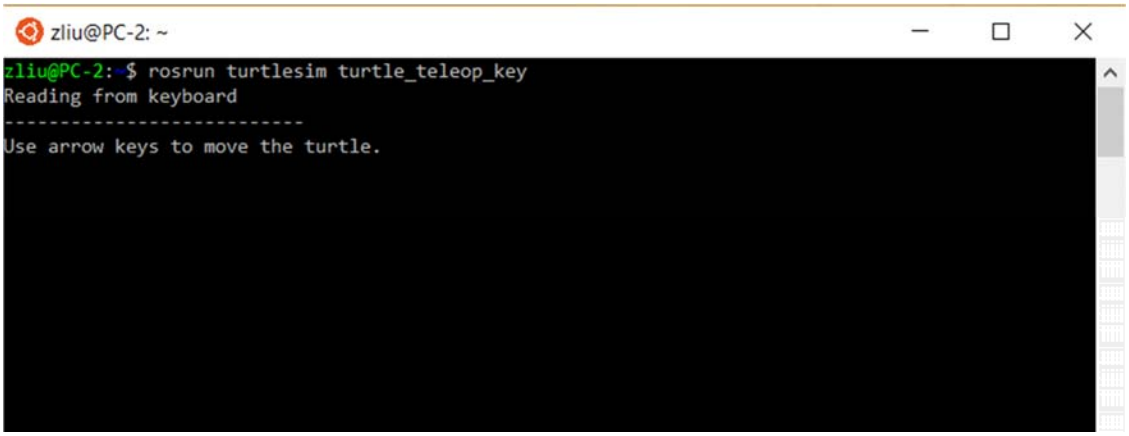
Clink finish to complete configuration.



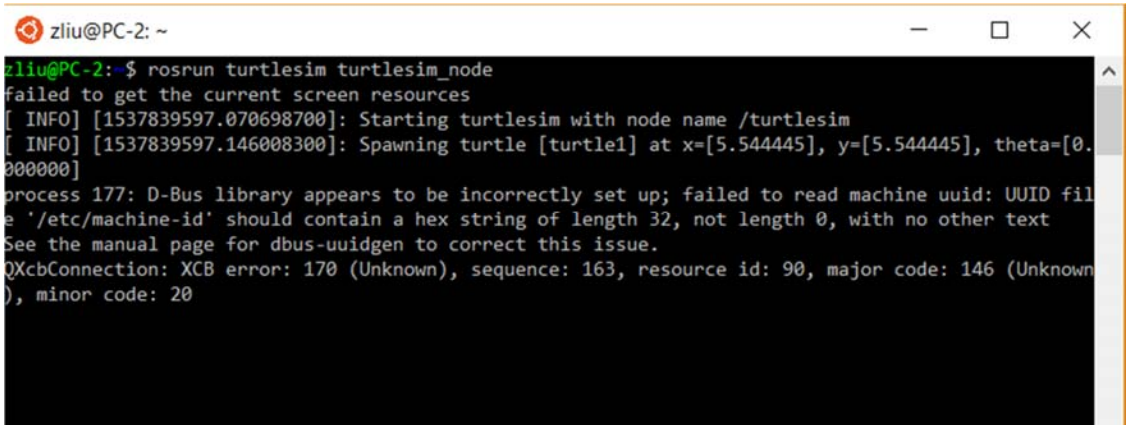
After installing, configuring and running X Server, start a new bash prompt and run `roscore`.



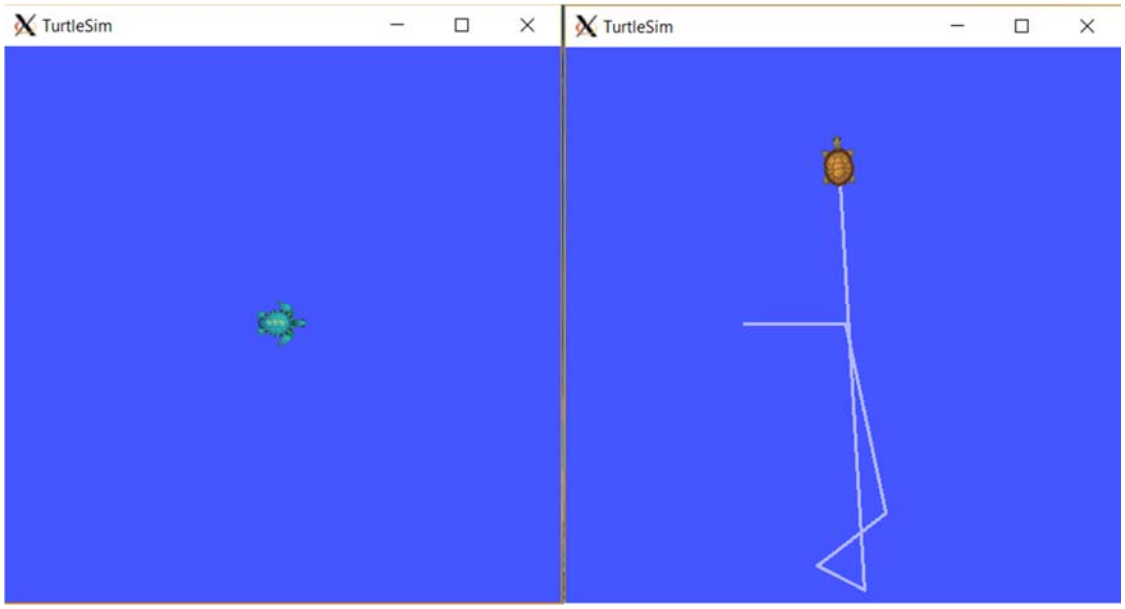
Start a second bash prompt and run `rosrun turtlesim turtle_teleop_key`.



Start a third bash prompt and run `rosrun turtlesim turtlesim_node`.



You can see a new window pop up with a little turtle in the middle, as shown in the following screenshot. You can control the turtle to move around by using the arrow keys by going back to the second prompt.



Now, you have everything installed to play around with ROS and practice examples in this manual.

ROS Environment

During installation of ROS, environment setup files are generated for you, but can come from different places, for example, ROS packages installed with package managers, rosbuild workspace, or a by-product of building or installing catkin packages provide setup.*sh files. A good way to check whether you have your environment properly setup is to ensure that environment variables, such as ROS_ROOT or ROS_PACKAGE_PATH are set by using following command:

```
$ printenv | grep ROS
```

```
zliu@PC-2: $ printenv | grep ROS
ROS_ROOT=/opt/ros/kinetic/share/ros
ROS_PACKAGE_PATH=/opt/ros/kinetic/share
ROS_MASTER_URI=http://localhost:11311
ROS_VERSION=1
ROSLISP_PACKAGE_DIRECTORIES=
ROS_DISTRO=kinetic
ROS_ETC_DIR=/opt/ros/kinetic/etc/ros
```

If you are not, you might need to source some setup.*sh files. If you installed ROS Kinetic from apt on Ubuntu, you will have setup.*sh file in '/opt/ros/kinetic/', and source them like this:

```
$ source /opt/ros/kinetic/setup.bash
```

You will need to run this command on every new shell you open to have access to ROS commands, unless you add this line to your .bashrc. Run following command to open .bashrc file.

```
$ nano ~/.bashrc
```

Add this line at the end of the .bashrc file.

```
source /opt/ros/kinetic/setup.bash
```

ROS Workspace

The workspace is a folder where you have packages, edit the source files or compile packages. A typical workspace is shown in the following screenshot. Each folder is a different space with different roles.

```
catkin_ws/
├── build
│   ├── catkin
│   ├── catkin_generated
│   ├── Makefile
│   └── ...
├── devel
│   ├── bin
│   ├── setup.zsh
│   └── ...
└── src
    ├── CMakeLists.txt -> /opt/ros/hydro/share/catkin/cmake/toplevel.cmake
    ├── ros_tutorials-hydro-devel
    └── ...
```

- The Build space: in this folder, CMake and catkin keep the cache information, configuration, and other intermediate files for packages and projects.
- The Development (devel) space: this folder is used to keep the complied programs. This is used to test the programs without the installation step.
- The Source (src) space: this folder contains packages, projects, clone packages, and so on. One of the most import files in this space is CMakeLists.txt, invoked by CMake when you configure the packages in the workspace.

Let's create a catkin workspace for L2Bot.

```
$ mkdir -p ~/l2bot_ws/src
```

```
$ cd ~/l2bot_ws/
$ catkin_make
```

The catkin_make command is a convenient tool for working with catkin workspace. Running it the first time, it will create a CMakeLists.txt link in your 'src' folder. If you look into your current directory, you should see 'build' and 'devel' folder. In Windows subsystem, the directory is C:\Users\(username)\AppData\Local\Packages\CanonicalGroupLimited.Ubuntu16.04onWindows_79rhkp1fndgsc\LocalState\rootfs\home\zliu\l2bot_ws. Or you can use ls command in the directory:

To finish the configuration, use the following command to reload the setup.bash file :

```
$ source devel/setup.bash
```

Now our environment is setup.

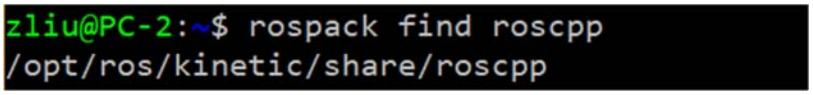
Navigate ROS Filesystem

Packages are referred to a typical structure of files and folders. Each package can contain libraries, executables, or scripts. Code is spread across many ROS packages. To create, modify, or work with package, ROS gives us command-line tools for assistance:

rospack is used to get information or find packages in the system.

Usage:

```
$ rospack find [package_name]
```



roscd is used to change directory, similar to cd command in Linux.

Usage:

```
$ roscd [locationname[/subdir]]
```

rosls is used to list the files from a package, similar to ls command in Linux

Usage:

```
$ rosls [locationname[/subdir]]
```

```
zliu@PC-2:~$ rosls turtlesim
cmake  images  msg  package.xml  srv
```

roscp is used to copy a file from a package to target location

Usage:

```
$ roscp [package][filename][target]
```

Packages in a Catkin Workspace

The recommended method of working with catkin packages is using a catkin workspace. A trivial workspace might look like this:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

Create a Catkin Package

We can use catkin_create_pkg to create a new catkin package. Let's change to the source space directory of the catkin package we have created before.

```
$ cd ~/l2bot_ws/src
```

Now use the catkin_create_pkg to create a new package called 'l2bot' which depends on roscpp, std_msgs, geometry_msgs:

```
$ catkin_create_pkg l2bot1 std_msgs roscpp geometry_msgs
```

This will create a l2bot folder which contains a package.xml and a CMakeLists.txt.

We can use rospack, roscd, and rosls commands to retrieve information about new package. For example, use rospack to view these dependencies.

```
$ rospack depends1 l2bot1
```

```
zliu@PC-2:~/l2bot_ws$ rospack depends1 l2bot1
geometry_msgs
roscpp
std_msgs
```

These dependencies for a package are stored in the package.xml file:

```
$ roscd l2bot1
$ cat package.xml
```

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>geometry_msgs</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<build_export_depend>geometry_msgs</build_export_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>geometry_msgs</exec_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>std_msgs</exec_depend>
    <!-- The export tag contains other, unspecified, tags -->
- <export>
        <!-- Other tools can request additional information be placed here -->
  </export>
</package>
```

The dependencies are split into buildtool_depend, build_depend, exec_depend. So all of our specified dependencies are available at build and run time. You can tailor the package.xml as you need.

Build an ROS Package

Now we need to build the packages in the workspace folder. When we build the package, what happens really is that the code is compiled.

```
$ cd ~/l2bot_ws
~/l2bot_ws $ catkin_make
```

Then we need to source the generated setup file in order to add the workspace to ROS environment:

```
~/l2bot_ws $ source devel/setup.bash
```

Play with ROS Nodes

Nodes are executable programs, and these executables are in the devel space.

Install l2bot Repository

Now you have known how to create a catkin package. You can also download the whole l2bot package into the src directory by using git clone:

```
$ cd ~/l2bot_ws/src
```

```
~/l2bot_ws/src $ git clone https://github.com/LTU-AutoEV/l2bot.git
```

Then run catkin_make to build the package:

```
~/l2bot_ws/src $ git clone https://github.com/LTU-AutoEV/l2bot.git
~/l2bot_ws $ catkin_make
~/l2bot_ws $ source devel/setup.bash
```

Then install dependencies:

```
~/l2bot_ws $ rosdep install --from-paths src --ignore-src -r -y
```

Topics

ROS system consist of a number of independent nodes that comprise a graph. Nodes need to communicate with each other and exchange information as well as data. The most common way to do so is through topics. A topic is a name for a stream of messages with a defined type. For example, the data from a camera might be sent over a topic called image, with a message type of Image.

Topics implement a *publish/subscribe* communication mechanism, the most common way to exchange data in a distributed system. Before nodes start to transmit data over topics, they must first announce, or *advertise*, both the topic name and the types of messages that are going to be sent. Then they start to send, or *publish*, the actual data on the topic. Nodes that want to receive message on a topic can *subscribe* to that topic by making a request to *roscore*. After subscribing, all messages on the topic are delivered to the node that made the request. One of the main advantages of using ROS is that all the messy details of setting up the necessary connections when nodes advertise or subscribe to topics is handled by the underlying communication mechanism so that you don't have to worry about it yourself.

We will start off by looking at how a node advertise a topic and publishes data on it.

Publishing to a Topic

The following is a minimalist ROS node that advertises the message topic and publishes hello world message on it.

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char** argv)
{
    // Initialize ROS
    ros::init(argc, argv, "hello_world_pub");

    // Print messages to ROS
    ROS_INFO_STREAM("hello_world_pub is running!");

    // Every node must a NodeHandle
    // It is the main communication point
    ros::NodeHandle nh{"~"};

    // The publisher object
    // Try changing the published topic to "hello_world" (no forward slash)
    ros::Publisher pub = nh.advertise<std_msgs::String>("/hello_world", 10);

    // How often to publish?
    ros::Rate loop_rate(10);

    // The publish loop
    while (ros::ok())
    {
        // Create the message object
        std_msgs::String msg;

        // Set the data
        msg.data = "Hello world!";

        // Publish the message
        pub.publish(msg);

        // wait for loop_rate
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

```cpp
#include <ros/ros.h>
```

```
#include <std_msgs/String.h>
```

The headers included are ros/ros.h and std_msgs/String.h. ros/ros.h includes all the files necessary to use the node with ROS, and std_msgs/String.h denotes the type of message we are going to use.

```
ros::init(argc, argv, "hello_world_pub");
```

Initiate the node and set the name.

```
ROS_INFO_STREAM("hello_world_pub is running!");
```

Print the information to the console.

```
ros::NodeHandle nh{"~"};
```

Set the handler of the process.

```
ros::Publisher pub = nh.advertise<std_msgs::String>("/hello_world", 10);
```

Set a publisher and tell the master the name of the topic and the type. In this case, the name is message hello world, and the second parameter is the buffer size.

```
ros::Rate loop_rate(10);
```

Set the frequency to send the data, which in this case is 10 Hz.

```
while (ros::ok())
{
   std_msgs::String msg;
   msg.data = "Hello world!";
```

Create a variable for the message with the correct type to send the data, and set the message data.

```
   pub.publish(msg);
```

Publish the message.

```
   ros::spinOnce();
```

Here there is a subscriber, where ROS updates and reads all the topics.

```
   loop_rate.sleep();
```

Sleep for the necessary time to make sure that we run the body of the while loop at 10 Hz frequency.

```
}
```

Subscribing to a Topic

The following shows a minimalist node that subscribes to the message topic and prints out the message in the message as they arrive.

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>

class HelloWorldSub
{
public:
    HelloWorldSub();
    void helloWorldCB(const std_msgs::String& msg);

private:
    // Node handle and subscriber are private class member
    ros::NodeHandle nh_;
    ros::Subscriber sub_;
};

HelloWorldSub::HelloWorldSub()
    :nh_{"~"}
{
    sub_ = nh_.subscribe("/hello_world", 10, &HelloWorldSub::helloWorldCB, this);
}

void HelloWorldSub::helloWorldCB(const std_msgs::String& msg)
{
    ROS_INFO_STREAM(msg.data);
}

int main(int argc, char** argv)
{

    // Init ros
    ros::init(argc, argv, "hello_world_sub");

    // All we have to do is create an instance of the object
    HelloWorldSub hw{};

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

```cpp
class HelloWorldSub
{
public:
    HelloWorldSub();
    void helloWorldCB(const std_msgs::String& msg);
```

```
private:
    ros::NodeHandle nh_;
    ros::Subscriber sub_;
}
```

In this class, initiate the node handler and subscriber as well as the subscriber and callback functions.

```
HelloWorldSub::HelloWorldSub():nh_{"~"}
{
    sub_ = nh_.subscribe("/hello_world", 10, &HelloWorldSub::helloWorldCB, this);
}
```

In this function, a subscriber is created and starts to listen to the topic with the name hello world. The buffer will be of 10, and the function to handle the message will be helloWorldCB.

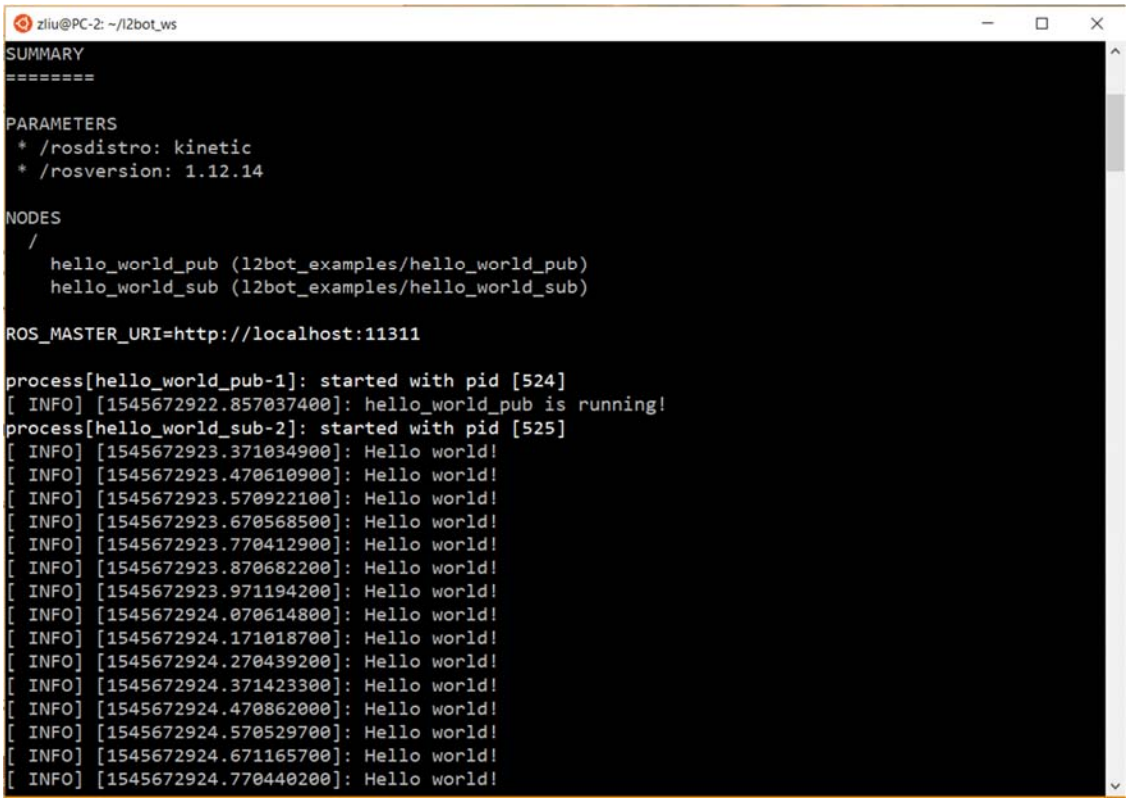
```
void HelloWorldSub::helloWorldCB(const std_msgs::String& msg)
{
    ROS_INFO_STREAM(msg.data);
}
```

This is the callback function that will be called when a new message has arrived on the hello world topic. This is where we do something with the data; in this case, we print it in the console.

```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "hello_world_sub");
    HelloWorldSub hw{};
    ros::spin();
}
```

Finally in the main function, initiate the node and create an instance of the object for HelloWorldSub. Once the subscription is made, we give control over to ROS by calling ros::spin(). The function will only return when the node is ready to shut down.

Now let's run the nodes in the terminal, and you will see the node is running.



Arduino Setup

L2bot is equipped with Arduino Uno, an open-source microcontroller board. The Arduino Uno is able to read inputs - light on a sensor or a finger on a button - and turn it into an output – activating a motor or turning on an LED. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so, you can use the Arduino Software (IDE). The Arduino IDE allows you to write programs and upload them to your board.

Download and Install Arduino IDE

You can choose between Windows Installer (.exe) and ZIP packages. It's advised to use the first one that installs directly everything needed to use the Arduino Software (IDE), including drivers, while you need to install drivers manually with the ZIP package.

When the download finishes, proceed with the installation, follow the instructions and also allow the driver installation process when getting a warning from the Windows. When installation finishes, run the Arduino IDE the first time and then close it once it has finished launching. This will automatically create a directory for all your sketchbooks.

Add serial write permissions

Open Ubuntu terminal, run the following command:

```
$ sudo usermod -a -G dialout <UNAME>
```

<UNAME> is your username.

```
zliu@PC-2:~$ sudo usermod -a -G dialout zliu
zliu@PC-2:~$
```

Restart your computer.

Arduino IDE Setup

ROS bindings are implemented as an Arduino library. Using the rosserial_arduino package, you can use ROS directly with the Arduino IDE.

Install Related Libraries

Install rosserial for Arduino by running the following command:

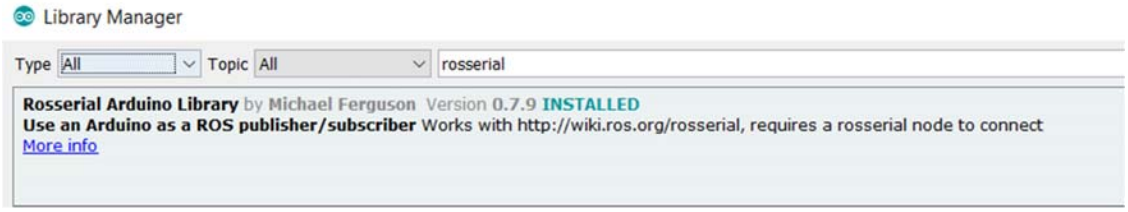
```
sudo apt-get install ros-kinetic-rosserial-arduino
sudo apt-get install ros-kinetic-rosserial
```

The preceding installation creates ros_lib. ros_lib works by putting its library implementation into the libraries folder of your sketchbook, so it has to be copied into the Arduino build environment to enable Arduino program to interact with ROS.

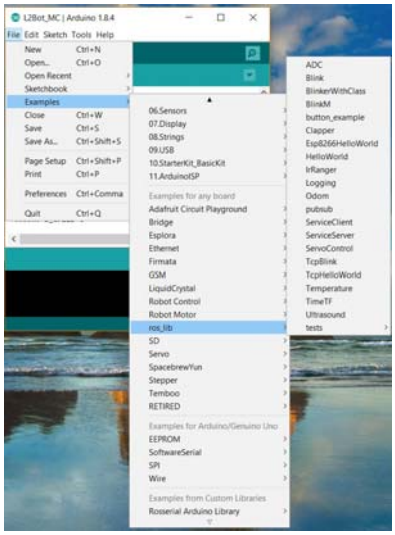
Go to the Arduino sketchbook/libraries directory, and run the following command:

```
cd <sketchbook>/libraries
rosrun rosserial_arduino make_libraries.py .
```

Or you can install the libraries directly in the Arduino IDE. Open the Library Manager from the IDE menu in Sketch -> Include Library -> Manage Library, search for “rosserial”, and then clink the Install button on the right side.



After reopening the IDE, you can see ros_lib listed under examples:



Load Interface into Arduino IDE

Open Arduino IDE, go to File -> Open -> Open file L2Bot_MC.ino, which is located at C:\Users\(username)\AppData\Local\Packages\CanonicalGroupLimited.Ubuntu16.04onWindows_79rhkp1fndgsc\LocalState\rootfs\home\zliu\l2bot_ws\src\l2bot\arduino\L2Bot_MC

Plug in the Arduino and turn on the L2Bot.

Go to Tools -> Port -> select your Arduino device (to view your Arduino device, you can search for Device Manager -> Port

Upload to Arduino

Identify Your Arduino Board

Plug in your Arduino and turn on L2Bot, and execute the detect_arduino.py script.

Gazebo Simulation

Although Windows Subsystem provides a convenient way for us to use bash, it has its limitations. It doesn't support camera at the moment. So instead of running code on L2bot, I will run ROS node in simulation.

Installation of Gazebo

Type the following commands in the terminal:

1. Set up computer to accept packages from osrfoundation.org

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_rel
ease -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
```

2. Update debian database

```
sudo apt-get update
```

3. Install gazebo7

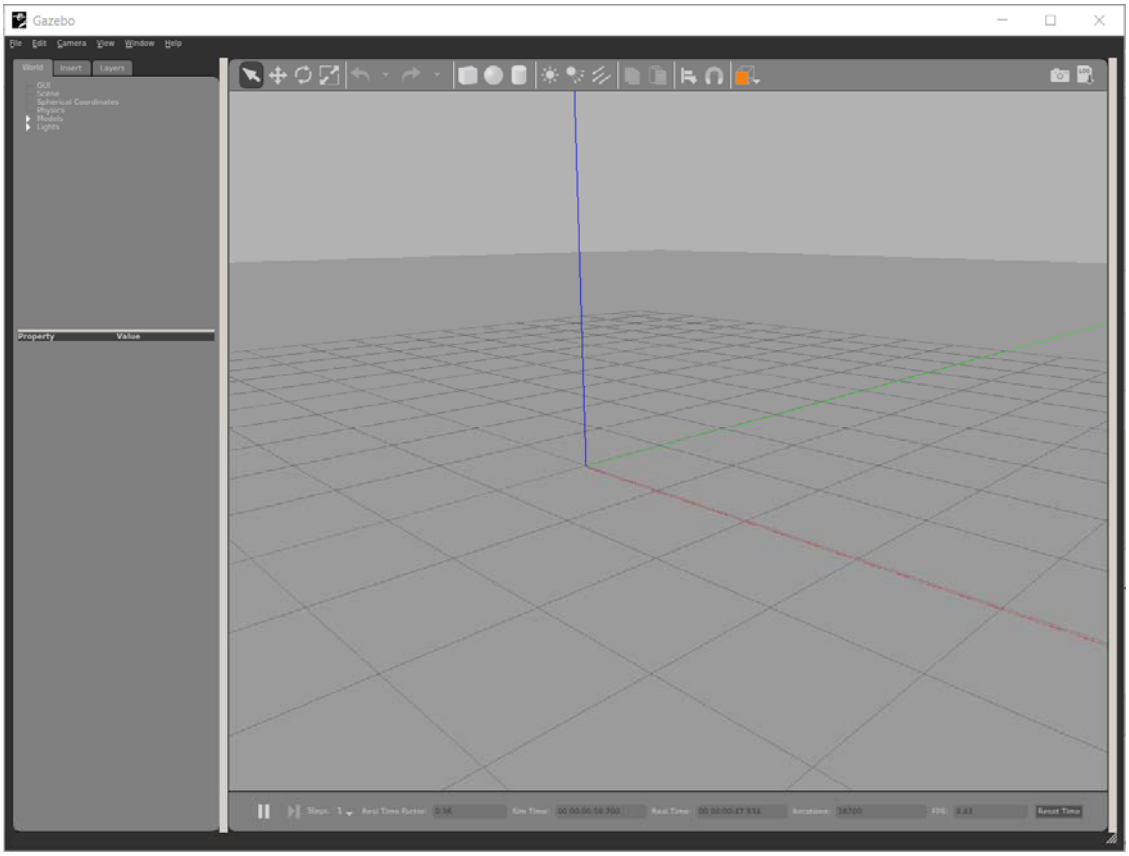
Gazebo is an independent project. Usually the latest version of gazebo available at the beginning of every ROS release cycle is selected as the official version to be integrated and will be kept for the whole life of the ROS distribution. So be sure to select the correct gazebo version based on your ROS. Our ROS kinetic goes with Gazebo7. so download gazebo7.

```
sudo apt-get install gazebo9
```

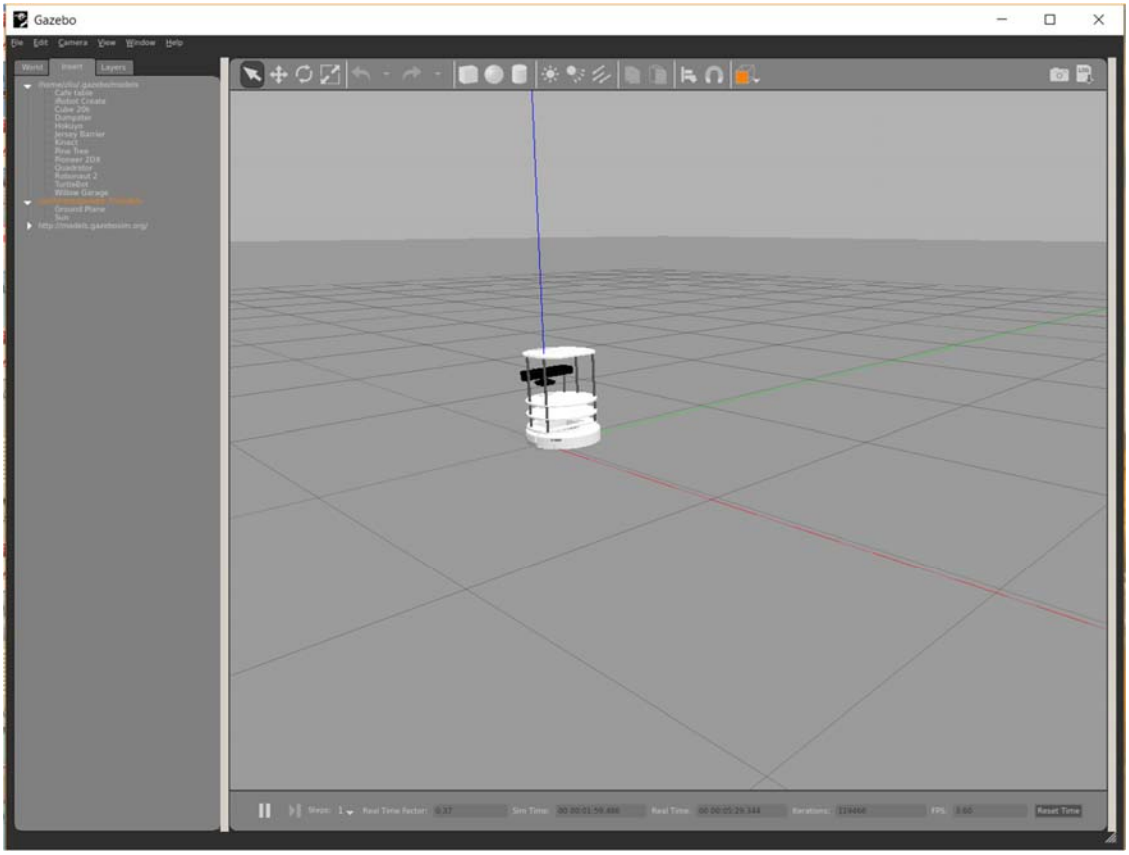
4. Launch gazebo

```
gazebo
```

After Gazebo is launched, you can see the GUI with an empty world.



You can download models from http://models.gazebosim.org/ and drag them into the world. Let's drag a TurtleBot in the world.



Now let's connect Gazebo to ROS.

The ROS packages to interface with Gazebo is contained in gazebo_ros_pkgs. Let's install gazebo_ros_pkgs.

Before installing the package, make sure the stand-alone Gazebo works and test whether you have the right version of Gazebo installed.

```
which gzserver
which gzclient
```

Now run the following command to install the package.

```
sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-ros-control
```

Now test Gazebo with ROS integration. Be sure to source ROS setup file first:

```
source /opt/ros/kinetic/setup.bash
```

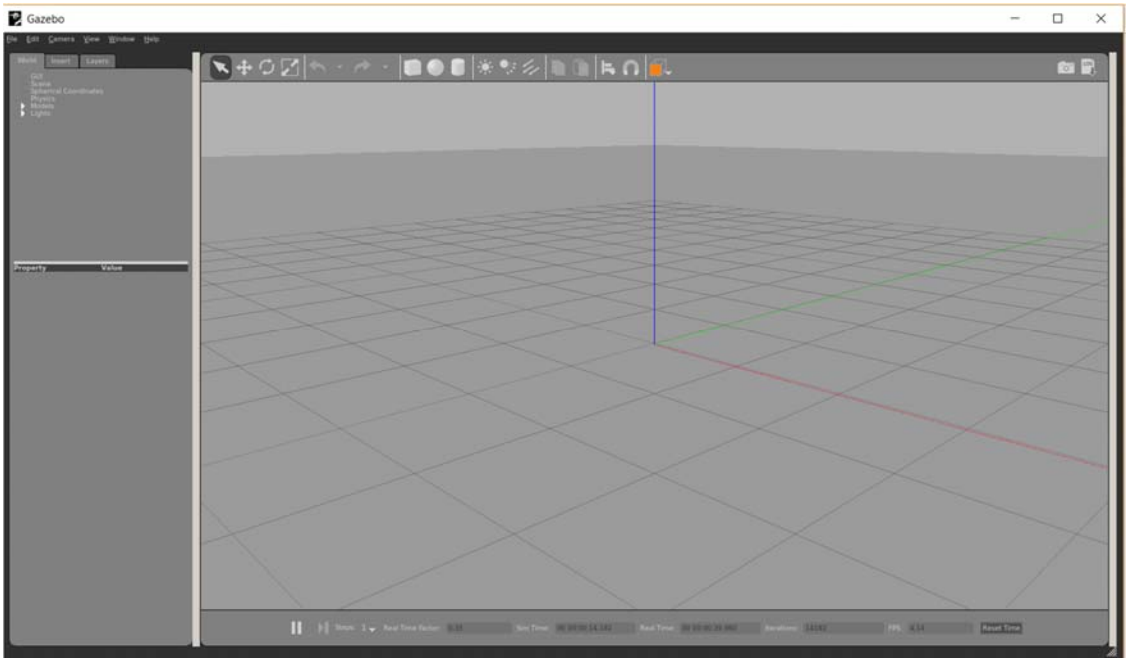
Or

```
source ~/catkin_ws/devel/setup.bash
```

Now let's run Gazebo.

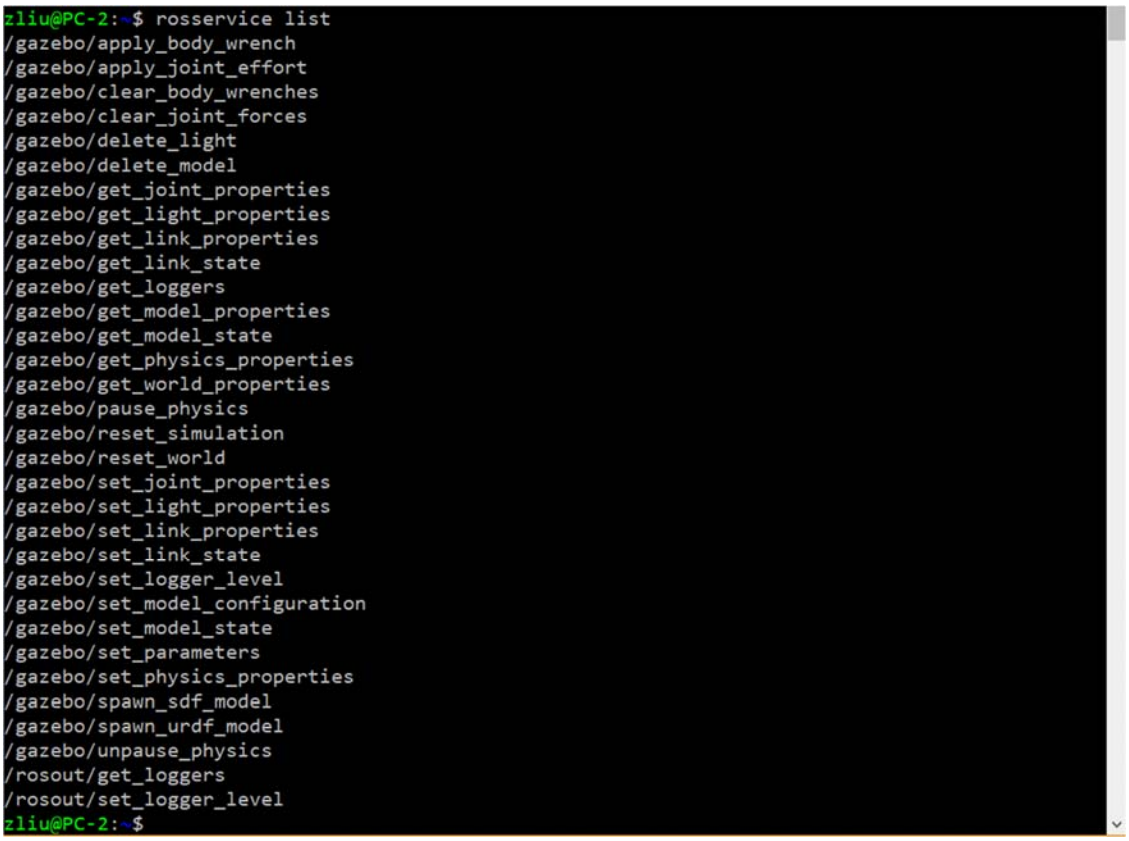
```
roscore &
rosrun gazebo_ros gazebo
```

Gazebo GUI will appear with an empty world.



To verify ROS is connected, we can view the available ROS topics.

We can also verify Gazebo service exist.



Nodes Example

Forward:

This example sends commands to L2bot and make L2bot drive forward for 8 seconds.

```cpp
// ROS and messages
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>

// L2Bot Controller Topic
#define TWIST_PUB "/rb_drive/rb_drive/twist_cmd"
// Time to drive forward (in seconds)
#define FWD_TIME 8.0

int main(int argc, char** argv)
{

    // Initialize ROS
    ros::init(argc, argv, "go_fwd");
    ros::NodeHandle nh;

    // Publish
    ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(TWIST_PUB, 10);

    // Publish rate
    ros::Rate loop_rate(10);

    // The twist message to publish
    geometry_msgs::Twist msg;
    msg.linear.x = 2.0f;
    msg.angular.z = 0.0f;

    ros::Time begin = ros::Time::now();

    ROS_INFO_STREAM("l2bot_example 'forward' publishing for " << FWD_TIME << " seconds!");

```

```
32      // Publish loop
33      while(ros::ok())
34      {
35          // Publish the message
36          pub.publish(msg);
37
38          // Wait for ROS (time based on loop_rate above)
39          ros::spinOnce();
40          loop_rate.sleep();
41
42          // Get runtime duration
43          ros::Time end = ros::Time::now();
44          ros::Duration dur = end - begin;
45
46          // 5 seconds
47          if (dur.toSec() > FWD_TIME) {
48              // Publish stop command
49              msg.linear.x = 0.0f;
50              pub.publish(msg);
51
52              // Exit loop
53              break;
54          }
55      }
56
57      ROS_INFO_STREAM("l2bot: forward complete!");
58      ros::spin();
59
60      return 0;
61  }
```

Here is a further explanation of the preceding code:

```
include <ros/ros.h>
```

```
include <geometry_msgs/Twist.h>
```

These lines includes the headers we are going to need. Here `ros/ros.h` includes all the files necessary to use the node with ROS, like creating a node and creating a publisher, and `geometry_msgs/Twist.h` includes the header that denotes the type of the message we are going to use, in this case, the Twist message. The Twist message is composed of 3 linear components and 3 angular components.

```
define TWIST_PUB "/rb_drive/rb_drive/twist_cmd"
```

Define the topic the node is publishing to.

```
define FWD_TIME 8.0
```

Define the moving forward time.

```
ros::init(argc, argv, "go_fwd");
```

```
ros::NodeHandle nh;
```

`ros::init` is used to initialize the ROS rode, and name it “go_fwd”, while `ros::NodeHandle` starts the node.

```
ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(TWIST_PUB, 10);
```

Publishing a message is done through `ros::Publisher pub = nh.advertise`, followed by the message type that we are going to send. In this case it is a `geometry_msgs::Twist`, and the topic that we are going to be sending too, which in this case is `TWIST_PUB`. The second parameter is the buffer size or message queue size. In this example, the buffer size is 10. If you are publishing message faster than what roscpp can send, a large buffer size will be used. the larger the buffer, the more delay in robot movement in case of buffering. Therefore, in real life example, you will want to have a smaller buffer in case of robot movement, where delay in movement commands are undesirable and even dangerous, but dropped message is acceptable.

```
ros::Rate loop_rate(10);
```

ROS is able to control the loop frequency to send the data using `ros::Rate` to indicate how rapidly the loop will run in Hz. All Clearpath robots require a minimum loop rate of 10Hz.

```
geometry_msgs::Twist msg;
```

This line creates the message we are going to send, `msg` of the type `geometry_msgs::Twist`.

```
msg.linear.x = 2.0f;
```

```
msg.angular.z = 0.0f;
```

These two lines calculate the linear x and angular z values that will be sent to `TWIST_PUB`. Our l2bot robots come with very simple commands, for example, only rotate in its axis and move in one direction. The amount of rotational (angular) motion can be specified using the angular parameter and the amount of linear motion can be specified using the linear parameter. In our example, since the robot moves forward in a straight line, we just need linear x velocity as our moving straight forward speed, and angular velocity z is set to 0, so our robot will not turn (counter clock-wise). Once you set these parameter values into the variables in `geometry_msgs::Twist` and publish it, the robot will move accordingly.

```
ros::Time begin = ros::Time::now();
```

This line of code is to get the current time as a `ros::Time` instance.

```
ROS_INFO_STREAM("l2bot_example 'forward' publishing for " << FWD_TIME << " seconds!");
```

Output the logging message by using c++ STL streams function.

```
while(ros::ok())
```

`ros::ok()`function will return true if it receives a command to shut down, either by using the rosnode `kill` command, or by the user pressing `Ctrl+C` in the terminal.

```
pub.publish(msg);
```

Now we are finally ready to publish the message. The `pub.publish` adds msg to the publisher queue to be sent.

```
ros::spinOnce();
```

Usually `ros::spinOnce()` will call the callbacks waiting to be called at that point in time. In this case, we have a subscriber in this part, where ROS updates and reads all topics.

```
loop_rate.sleep();
```

the `loop_rate` instance will attempt to keep the loop at 10 Hz frequency by sleeping for the necessary time.

```
ros::Time end = ros::Time::now();

ros::Duration dur = end - begin;
```

Get the current time as a `ros::Time` instance, and perform arithmetic operations on Duration instance to get runtime duration.

```
if (dur.toSec() > FWD_TIME)

{

msg.linear.x = 0.0f;

pub.publish(msg);

}
```

If the duration time is larger than the five seconds, publish the stop message by using `msg.linear.x = 0.0f.`

```
ros::spin();
```

The `ros::spin()` line is a loop where the node starts to read the topic, and will not return until the node exits the loop and ends, either through users presses Ctrl + c or a call to `ros::shutdown()`.

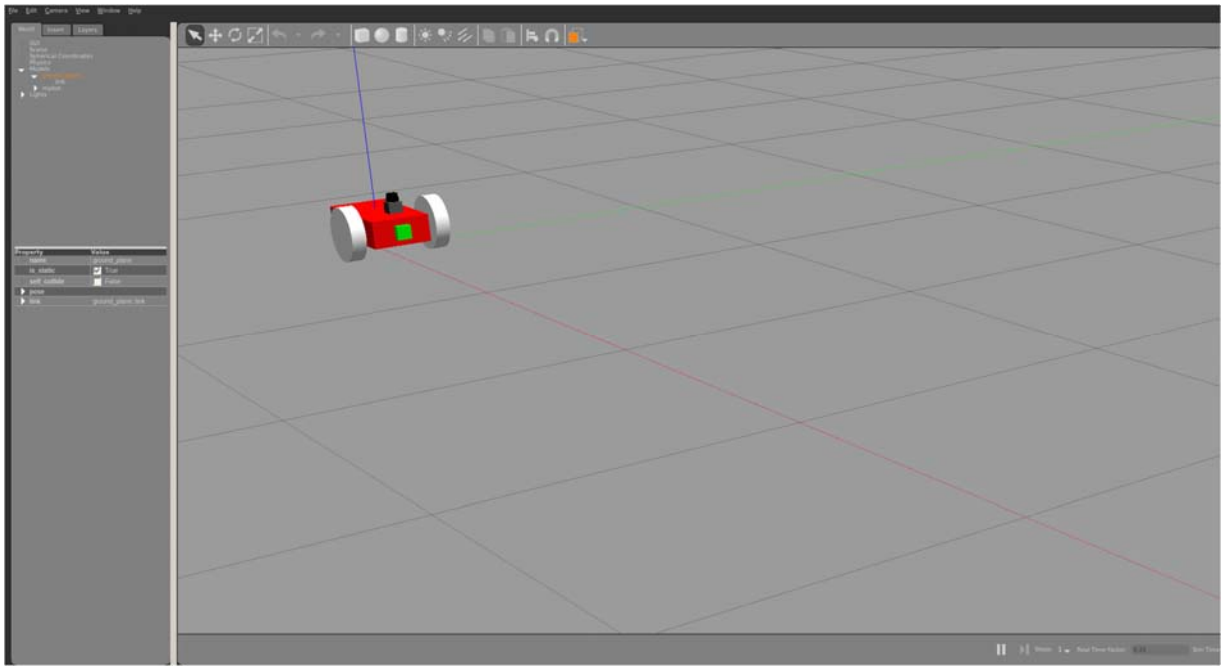
First run the code on our l2bot.

```
~/l2bot_ws$ source devel/setup.sh
~/l2bot_ws$ roslaunch l2bot_examples forward.launch
```

L2bot_Forward8s Video link

https://youtu.be/OYMLEpvoWcI

Then run in the Gazebo simulator:



Gazebo_Forward8s video link:

https://youtu.be/YcYFsVhMTN4

Using a Joystick

A joystick is nothing more than a series of buttons and potentiometers. With this device, you can perform or control a wide range of actions. In ROS, a joystick is used to telecontrol a robot to change its velocity or direction.

Joy_Nav:

This example shows how to convert hardware input messages into L2bot controls, and allows users to drive the L2bot with a joystick.

```
#include <ros/ros.h>
#include <sensor_msgs/Joy.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/UInt32.h>
#include <string>
#include <math.h>

#define JOY_SUB "/joy"
#define TWIST_PUB "/rb_drive/rb_drive/twist_cmd"
#define MULTIPLIER 3
```

```
/* Class JoyNav
 * Convert gamepad input to a geometry_msgs::Twist
 *
 * Publish:     Twist on TWIST_PUB
 * Subscribe:   Joy on "/joy"
 */
class JoyNav
{
public:
    JoyNav();

private:

    // Callback function
    void joyCallback(const sensor_msgs::Joy::ConstPtr& joy);

    // ROS node handle
    ros::NodeHandle nh_;

    // Publisher and subscriber
    ros::Publisher twist_pub_;
    ros::Publisher state_pub_;
    ros::Subscriber joy_sub_;

    int axes_linear_;
    int axes_angular_;
};
```

```
// Constructor
// Set up publisher and subscriber
JoyNav::JoyNav() : nh_{"~"}
{
    // Subscribe to the controller
    joy_sub_ = nh_.subscribe<sensor_msgs::Joy>(JOY_SUB, 10, &JoyNav::joyCallback, this);

    // Publish control vectors
    twist_pub_ = nh_.advertise<geometry_msgs::Twist>(TWIST_PUB, 100);

    // Load JS Mappings
    if (!nh_.getParam("/joy_mappings/axes_linear", axes_linear_))
    {
        ROS_ERROR_STREAM("joy_nav: Could not load joystick configuration.");
        ROS_ERROR_STREAM("joy_nav: Please run `roslaunch l2bot_examples joy_setup.launch`");
        axes_linear_ = -1;
        axes_angular_ = -1;
    }

    nh_.getParam("/joy_mappings/axes_angular", axes_angular_);
}
```

```
// Callback function
// Take input from contoller and create a vector from the input
void JoyNav::joyCallback(const sensor_msgs::Joy::ConstPtr& joy)
{
    // Get right-left and fwd-bkwd values
    // Values range from -1 to 1
    float rl = joy->axes[axes_angular_];
    float fb = joy->axes[axes_linear_]; //(1 - (joy->axes[axes_linear_]))/2.0f;

    // Create a vector
    geometry_msgs::Twist vec;
    vec.linear.x = MULTIPLIER * fb * -1;
    vec.angular.z = atan(rl);

    // Publish the vector
    twist_pub_.publish(vec);
}
```

```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "joy_nav");
    JoyNav joy_nav;

    ros::spin();
}
```

We're going to break the code to explain how it works.

```
#include <ros/ros.h>
#include <sensor_msgs/Joy.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/UInt32.h>
#include <string>
#include <math.h>
```

These lines includes the headers and type of messages we are going to need for the joystick topic. We've maintained `ros/ros.h,` which includes all the files necessary to use the node with ROS, and `geometry_msgs/Twist.h,` which includes the header that denotes the type of the message we are going to create. `sensor_msgs/Joy.h` includes the joystick message so that we can listen to the joy topic. It includes fields such as std_msgs/header, axes, and buttons you can use if you use a joystick.

```
define JOY_SUB "/joy
define TWIST_PUB "/rb_drive/rb_drive/twist_cmd"
define MULTIPLIER 3


int main(int argc, char** argv)
{
   ……
   JoyNav joy_nav;
   ……
```

In the main function, we create an instance of the JoyNav class. In the `JoyNav` class, we define the jobCallback function that will take a joy message, and we also create a node handler, publisher, and subscriber for use.

```
JoyNav::JoyNav() : nh_{"~"}
{
   joy_sub_ = nh_.subscribe<sensor_msgs::Joy>(JOY_SUB, 10, &JoyNav::joyCallback, this);
   twist_pub_ = nh_.advertise<geometry_msgs::Twist>(TWIST_PUB, 100);
   if (!nh_.getParam("/joy_mappings/axes_linear", axes_linear_))
   {
      ……
      axes_linear_ = -1;
      axes_angular_ = -1;
   }
```

```
    nh_.getParam("/joy_mappings/axes_angular", axes_angular_);
}
```

In the `JoyNav` constructor, four variables are initialized. The first two variables are the advertiser and subscriber. The advertiser will publish a topic with the `geometry_msgs::Twist` message type with the buffer size of 100. The subscriber will get data from the topic with the name `Joy`. The node that is handling the joystick sends this topic. The buffer will be of 1, meaning that if our node is slow in processing incoming messages on the joystick topic, 1 message will be buffered, and the function to handle the message will be `joyCallback`. The next two variables are filled using data from parameter server. These variables are joystick axes.

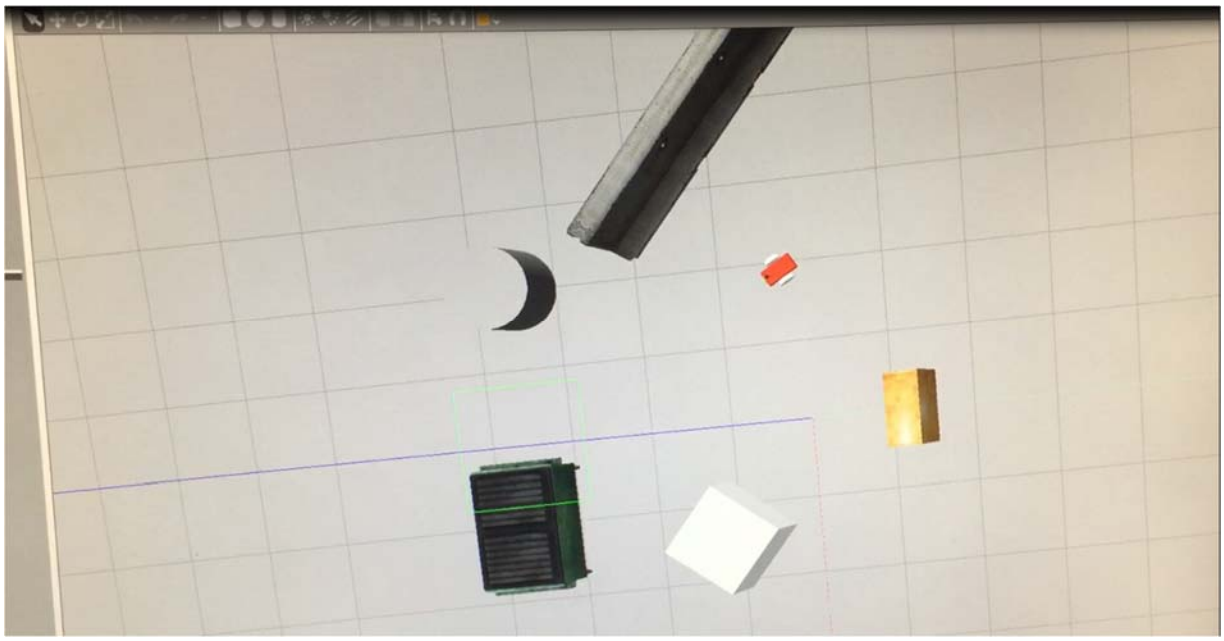
```
void JoyNav::joyCallback(const sensor_msgs::Joy::ConstPtr& joy)
{
    float rl = joy->axes[axes_angular_];
    float fb = joy->axes[axes_linear_];
    geometry_msgs::Twist vec;
    vec.linear.x = MULTIPLIER * fb * -1;
    vec.angular.z = atan(rl);
    twist_pub_.publish(vec);
}
```

Callback function is called every time that the node receives a message. This is where we do something with the data; in this case, we create a new variable with the name `vec`, which will be used to publish data. The values of the axes of the joystick are assigned to `vec` variable, and used to control the linear and angular velocities of the robot. Finally, we publish the prepared message using `pub.publish(vec)`.

```
int main(int argc, char** argv)
{
    ros::init(argc, argv, "joy_nav");
    ……
    ros::spin();
}
```

Finally, in the main function, we initialize ROS node, create an instance of the JoyNav class, and use `ros::spin()` line. The `ros::spin()` line is a loop where the node starts to read the topic and when a message arrives, joyCallback is called. When users presses Ctrl + c, the node exits the loop and ends.

We will use tele control to move robot in Gazebo simulation to avoid those obstacles.



Video link:

https://youtu.be/3sy5WML1J3Q

Computer Vision

ROS provides basic support for Computer Vision. Drivers are available for different cameras and protocols, and an image pipeline helps with the camera calibration process, distortion ratification, color decoding, etc. For more complex tasks, you can use `OpenCV`, and the `cv_bridge` and `image_transport` libraries to interface with it and subscribe and publish images on topics. The following example demonstrates how to publish and subscribe to images in ROS, and also how to use OpenCV in nodes.

Cam_Pub

First, we will create an image publish node which will continually publish an image.

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <cv_bridge/cv_bridge.h>
#include <ctime>
#include <cstdlib>
#include <string>
#include <vector>

#define CVWIN_OUT "cam_pub output"
```

```
12  /**
13   * Image publisher for cv images
14   * Publishes images at ~30fps
15   */
16  int main(int argc, char** argv)
17  {
18      //Initialize and set up ROS
19      ros::init(argc, argv, "cam_pub");
20
21      ros::NodeHandle nh("~");
22      image_transport::ImageTransport it(nh);
23      image_transport::Publisher pub = it.advertise("image_raw", 1);
24
25      std::string source;
26      cv::VideoCapture cap;
27
28      ros::Rate loop_rate(30);
29
30      int empty_frame_count = 0;
31      cv::Mat frame;
32      sensor_msgs::ImagePtr msg;

34      // Open the video source
35      if (nh.getParam("source", source))
36      {
37          cap.open(source);
38          ROS_INFO_STREAM("cam_pub: publishing using video source " << source << "...");
39      }
40      else
41      {
42          cap.open(0);
43          ROS_ERROR_STREAM("param '~source' not defined, using default camera 0");
44      }
45
46      // Check if video device can be opened with the given index
47      if(!cap.isOpened())
48      {
49          ROS_ERROR_STREAM("video device cannot be opened");
50          return 1;
51      }
```

```
54        ////////////////
55        // Parameters //
56        ////////////////
57
58        // Flip along horizontal axis?
59        bool hflip = false;
60        nh.getParam("hflip", hflip);
61        if (hflip) ROS_INFO_STREAM(source << ": hflip active!");
62
63        // Show output
64        bool show_output = false;
65        nh.getParam("show_output", show_output);
66        if (show_output) ROS_INFO_STREAM(source << ": show_output active!");
```

```
        while (nh.ok())
        {
            cap >> frame;

            // Check if grabbed frame is actually full with some content
            if(!frame.empty())
            {
                empty_frame_count = 0;

                // Flip the image upside down
                if (hflip) cv::flip(frame, frame, -1);

                if(show_output)
                {
                    cv::imshow(CVWIN_OUT, frame);
                }

                msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8", frame).toImageMsg();
                pub.publish(msg);
                cv::waitKey(1);
            }
            else
            {
                empty_frame_count++;
                if (empty_frame_count > 20)
                {
                    ROS_ERROR_STREAM("Could not read input, closing cam pub");
                    return 1;
                }
            }

            ros::spinOnce();
            loop_rate.sleep();

        }
    }
```

```
include <ros/ros.h>
include <image_transport/image_transport.h>
include <opencv2/highgui/highgui.hpp>
include <cv_bridge/cv_bridge.h>
include <ctime>
include <cstdlib>
include <string>
```

```
include <vector>
```

The image_transport API allows the publishing of images using several transport formats seamlessly, which can be compressed images, with different codecs, based on the plugins installed in the ROS system. The cv_bridge is used to load an image using OpenCV and convert it to ROS Image message format, for which we may need the image encoding of sensor_msgs, in the case of grayscale/color conversion. Finally, we need the highgui API of OpenCV 2 in order to use cv::VideoCapture.

```
int main(int argc, char** argv)
{
   ros::init(argc, argv, "cam_pub");
   ros::NodeHandle nh("~");
   image_transport::ImageTransport it(nh);
   image_transport::Publisher pub = it.advertise("image_raw", 1);
```

In main function, we first create a node handle and an ImageTransport instance used to send images in all available formats, initializing it with NodeHandle. We use methods of ImageTransport to create image publishers, advertising that we are going to be publishing images on the basic topic "image_raw". Additional topics may also be advertised depending on whether there are more plugins built. The second argument is the size of the publishing queue. advertise() returns an image_transport::Publisher Object, which serves as two purposes: first, the publish() method contained lets you to publish images onto the basic topic it's created with; second, when out of scope, it will automatically un-advertise.

```
  std::string source;
  cv::VideoCapture cap;
  ros::Rate loop_rate(30);
  int empty_frame_count = 0;
  cv::Mat frame;
  sensor_msgs::ImagePtr msg;
```

These are OpenCV stuff to capture images/frames, and image pointer of OpenCV images.

```
  if (nh.getParam("source", source))
  {
     cap.open(source);
     ……
  }
```

This block of code is used to open the video source.

```
if(!cap.isOpened())
{
    ……
    return 1;
```

```
}
```

This is to check whether video device can be opened with the given index.

```
bool hflip = false;
nh.getParam("hflip", hflip);
if (hflip) ROS_INFO_STREAM(source << ": hflip active!");
bool show_output = false;
nh.getParam("show_output", show_output);
if (show_output) ROS_INFO_STREAM(source << ": show_output active!");
```

These lines use `getParam()` to flip an image and show the output. Note that `getParam()` returns a bool, which provides the ability to check whether retrieving the parameter succeeded or not.

```
while (nh.ok())
{
   cap >> frame;
   if(!frame.empty())
   {
      empty_frame_count = 0;
      if (hflip) cv::flip(frame, frame, -1);
      cv::imshow(CVWIN_OUT, frame);
      msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8", frame).toImageMsg();
      pub.publish(msg);
      cv::waitKey(1);
   }
}
Else
{
   ……
   if (empty_frame_count > 20)
   {
      ……
      return 1
   }
}
```

In the publish loop, first retrieve frame from the video device. If the retrieved frame from the video device is not empty it will be flipped, showed on GUI window, and then converted to a ROS message, which will be published by the publisher. If the retrieved frame is empty, camera publish node will be closed.

Cam_Edge_Detect

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

#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/highgui/highgui.hpp>

// Change this value if you are subscribing to a different camera
#define CAM_TOPIC "/camera_1/cam_pub/image_raw"

// The name of the preview window
#define CVWIN_PREVIEW "cam_edge_detect preview"
```

```
/**
 * Simple Camera Subscriber
 * ========================
 *
 * In this example we use a class to modularize the functionality
 *   of this node. We can include several member functions and
 *   variables which hide the functionality from main().
 */
class SimpleCamSub
{
public:
    SimpleCamSub();
    ~SimpleCamSub();
    void imageCb(const sensor_msgs::ImageConstPtr& msg);

private:
    void getEdges(cv::Mat& src, cv::Mat& dst);

    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
};
```

```
39  /**
40   * Constructor
41   * ===========
42   *
43   * Do all initilization code here. This way, our main() function only
44   *   needs to instantiate the SimpleCamSub object once and do nothing
45   *   else (see main() below).
46   *
47   * In this case, we only need to set up the image subscriber
48   */
49  SimpleCamSub::SimpleCamSub()
50      :nh_{"~"}, it_{nh_}
51  {
52      image_sub_ = it_.subscribe(CAM_TOPIC, 1, &SimpleCamSub::imageCb, this);
53  }
57  /**
58   * Destructor
59   * ==========
60   *
61   * Destroy CV windows
62   */
63  SimpleCamSub::~SimpleCamSub()
64  {
65      cv::destroyWindow(CVWIN_PREVIEW);
66  }
```

```
70  /**
71   * Callback function
72   * =================
73   *
74   * Called once every time a image is published on the topic this
75   *   node is subscribed to. The image is passed to the function as
76   *   a ImageConstPtr. A few lines of code validate the image and
77   *   convert it into a cv::Mat
78   */
79  void SimpleCamSub::imageCb(const sensor_msgs::ImageConstPtr& msg)
80  {
81      //Convert to cv image
82      cv_bridge::CvImagePtr cv_ptr;
83      try
84      {
85          cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
86      }
87      catch (cv_bridge::Exception& e)
88      {
89          ROS_ERROR("cv_bridge exception: %s", e.what());
90          return;
91      }
92
93      // Run edge detection function
94      cv::Mat edges;
95      edges.create(cv_ptr->image.size(), cv_ptr->image.type());
96      getEdges(cv_ptr->image,  edges);
97
98      // Show preview window
99      cv::imshow(CVWIN_PREVIEW, edges);
100
101     // Update GUI Window
102     cv::waitKey(3);
103
104 }
```

```
108     * Edge Detection Example
109     * ======================
110     *
111     * An example function which detects the edges of the input image.
112     * The image is first converted to grayscale. Then, amedian blur
113     *   is applied to increase performance of the Canny edge detect
114     *   function. Finally, the edged are dialated using MORPH_RECT.
115     */
116    void SimpleCamSub::getEdges(cv::Mat& src, cv::Mat& dst)
117    {
118        // Convert the source to grayscale
119        cv::Mat src_gray;
120        cv::cvtColor(src, src_gray, CV_BGR2GRAY);
121
122        // Edge detection parameters
123        int lowThreshold = 65;
124        int rat = 3;
125        int kernel = 1;
126        int blur = 1;
127        int dilation_type = cv::MORPH_RECT;
128        int dilation_size = 5;
129
130        // These variables must be odd
131        int kernel_size = kernel*2+1;
132        int blur_size = blur*2+1;
133
134        // Apply median blur and edge detect
135        cv::medianBlur( src_gray, dst, blur_size );
136        cv::Canny( dst, dst, lowThreshold, lowThreshold*rat, kernel_size );
137
138        // DIaliate edges
139        cv::Mat dilate_element = cv::getStructuringElement( dilation_type,
140                cv::Size( 2*dilation_size + 1, 2*dilation_size+1 ),
141                cv::Point( dilation_size, dilation_size ) );
142        cv::dilate( dst, dst, dilate_element );
143    }
```

```
148  int main(int argc, char** argv)
149  {
150      ros::init(argc, argv, "cam_edge_detect");
151
152      // Create a SimpleCamSub object.
153      // Since initilization code is in the constructor, we do
154      //   not need to do anythong else with this object
155      SimpleCamSub sd{};
156
157      ROS_INFO_STREAM("cam_edge_detect running!");
158      ros::spin();
159      return 0;
160  }
```

```
include <ros/ros.h>
include <image_transport/image_transport.h>
include <cv_bridge/cv_bridge.h>
include <sensor_msgs/image_encodings.h>
include <opencv2/highgui/highgui.hpp>
```

We've talked about `image_transport, cv_bridge, highgui` in the last example: `image_transport` headers have functions to publish and subscribe to image messages; `cv_bridge` header has functions to convert between OpenCV ROS data types; `highgui` has GUI-related functions. `image_encodings` header has the image-encoding format used during ROS-OpenCV conversions.

```
define CAM_TOPIC "/camera_1/cam_pub/image_raw"
```

This line of code defines the camera you are subscribing to.

```
define CVWIN_PREVIEW "cam_edge_detect preview"
```

This line of code gives a name to the camera edge detect window.

```
int main(int argc, char** argv)
{
   ros::init(argc, argv, "cam_edge_detect");
   SimpleCamSub sd{};
   ……
}
```

In the main function, an instance of `SimpleCamSub object` is created. In the following `SimpleCamSub` class, we define the `imageCb` callback function that will take an image as a `ImageConstPtr`, and we also create a node handler, `image_transport` to help send ROS Image messages across the ROS computing graph, and subscriber for later use.

```
class SimpleCamSub {
public:
```

```
    SimpleCamSub();
   ~SimpleCamSub();
   void imageCb(const sensor_msgs::ImageConstPtr& msg);
private:
   void getEdges(cv::Mat& src, cv::Mat& dst);
   ros::NodeHandle nh_;
   image_transport::ImageTransport it_;
   image_transport::Subscriber image_sub_;
}
SimpleCamSub::SimpleCamSub():nh_{"~"}, it_{nh_}
{
   image_sub_ = it_.subscribe(CAM_TOPIC, 1, &SimpleCamSub::imageCb, this);
}
```

In the `SimpleCamSub` constructor, a subscriber variable is initialized for the input image topic. Whenever an image arrives on the input image topic, it will call a function named `imageCb`. The names of the topics are retrieved from ROS parameters.

```
SimpleCamSub::~SimpleCamSub()
{
   cv::destroyWindow(CVWIN_PREVIEW);
}
```

In the destructor, OpenCV highgui calls to destroy a display window on shutdown.

```
void SimpleCamSub::imageCb(const sensor_msgs::ImageConstPtr& msg)
{
   cv_bridge::CvImagePtr cv_ptr;
   try
   {
        cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
   }
```

The above code is the definition of `imageCb`, which is a callback for `image_raw`. What it basically does is that it converts the `sensor_msgs/Image` data into the `cv::Mat` OpenCV data type. The `cv_bridge::CvImagePtr cv_ptr` buffer is allocated for storing the OpenCV image after performing the ROS-OpenCV conversion using the `cv_bridge::toCvCopy` function.

```
   cv::Mat edges;
   edges.create(cv_ptr->image.size(), cv_ptr->image.type());
   getEdges(cv_ptr->image,  edges);
   cv::imshow(CVWIN_PREVIEW, edges);
```

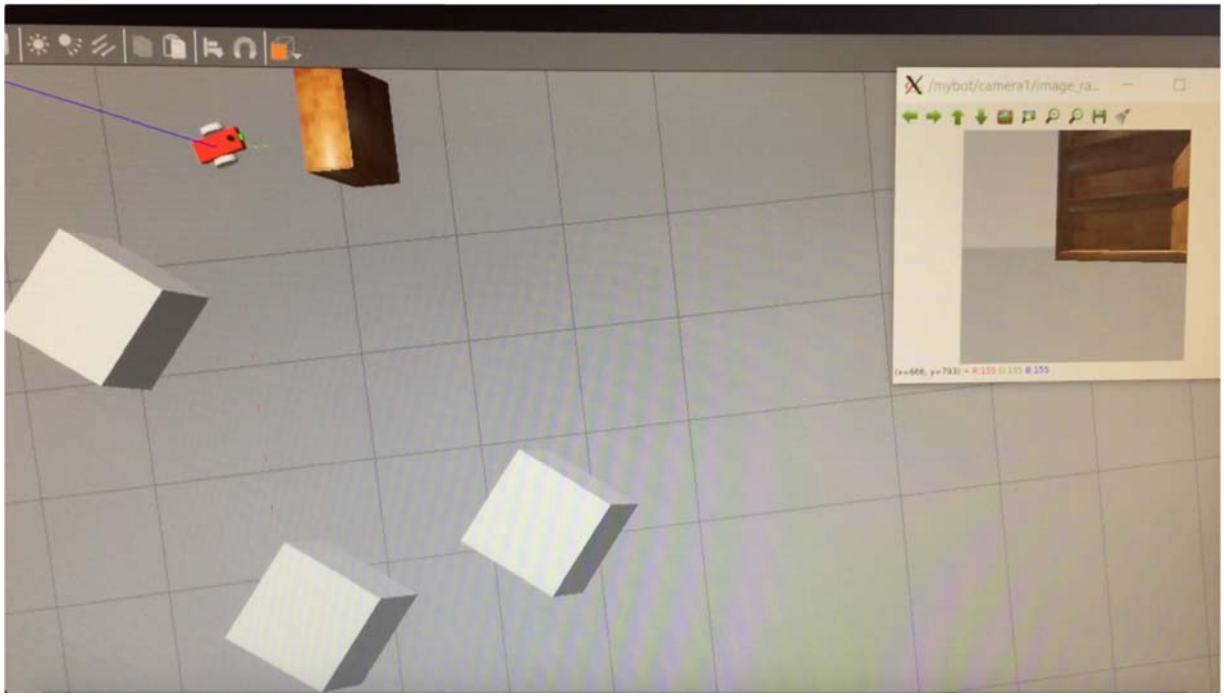
```
    cv::waitKey(3);
}
```

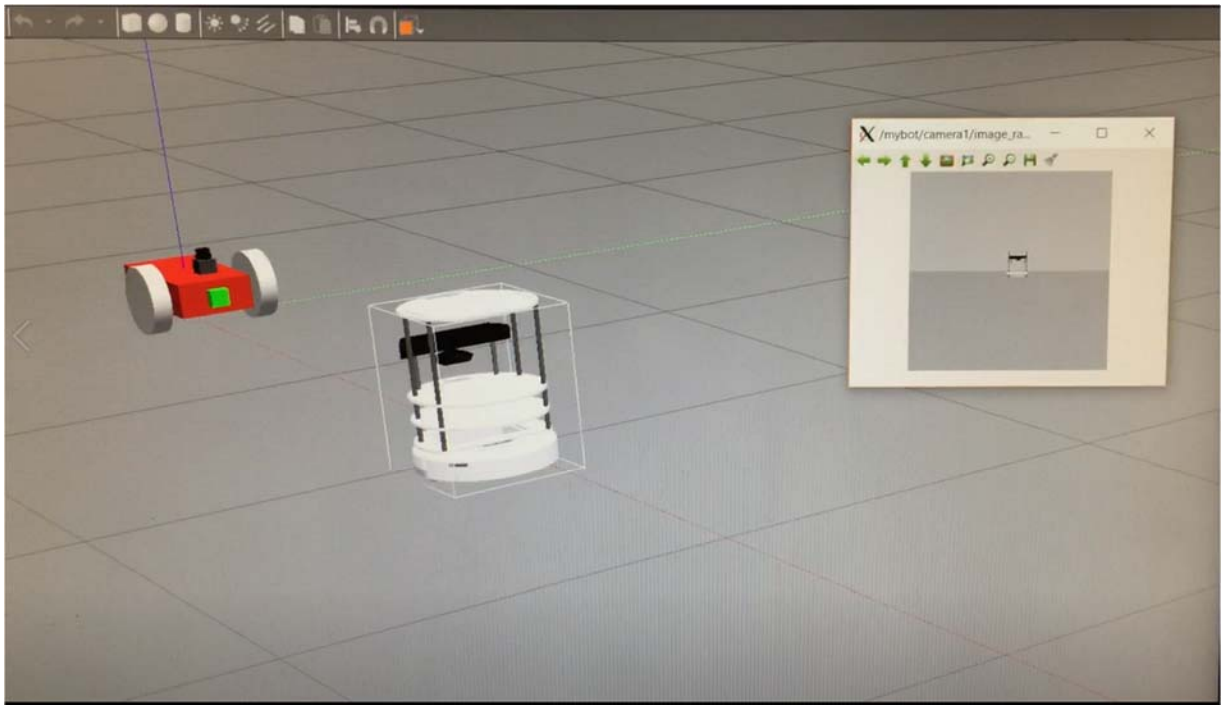
A `cv::Mat` data type is created for edges, and get the size and type of this OpenCV image data type. And then a functional call of getEdges(), which is performing image edges detection on the converted OpenCV image data type from the ROS image message. Using `cv_ptr->image`, we can retrieve the `cv::Mat` data type, and pass edges variables as second argument. In the last two lines, we update GUI window to show preview image.

```
void SimpleCamSub::getEdges(cv::Mat& src, cv::Mat& dst)
{
    cv::Mat src_gray;
    cv::cvtColor(src, src_gray, CV_BGR2GRAY);
    ……
    cv::medianBlur( src_gray, dst, blur_size );
    cv::Canny( dst, dst, lowThreshold, lowThreshold*rat, kernel_size );
    cv::Mat dilate_element = cv::getStructuringElement( dilation_type, cv::Size(
2*dilation_size + 1, 2*dilation_size+1 ), cv::Point( dilation_size, dilation_size ) );
    cv::dilate( dst, dst, dilate_element );
}
```

The above is the core port of the program, which is the detection of edges of the input image. The image is first converted to grayscale. Then, median blur is applied to increase performance of the Canny edge detection function. Finally, the edges are dilated using MORPH_RECT.

In the Gazebo simulation, on the upper corner of the image view, you can see what can be seen from the robot's camera.





Gazebo_CamView video files link

https://youtu.be/jnGNG8uNntA

https://youtu.be/IFLmwaSIyUg

stop_on_white

\

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>

// Includes for dynamic reconfigure
#include <dynamic_reconfigure/server.h>
#include <l2bot_examples/StopOnWhiteConfig.h>

// Includes for working with images
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/highgui/highgui.hpp>

// Change this value if you are subscribing to a different camera
#define CAM_TOPIC "/camera_1/cam_pub/image_raw"
#define TWIST_PUB "/rb_drive/rb_drive/twist_cmd"

#define CVWIN_PREVIEW "threshold preview"
```

```
20  /**
21   * Stop on White
22   * =====================
23   *
24   * In this example we use a class to modularize the functionality
25   *   of this node. We can include several member functions and
26   *   variables which hide the functionality from main().
27   */
28  class StopOnWhite
29  {
30  public:
31      StopOnWhite();
32      ~StopOnWhite();
33      void imageCb(const sensor_msgs::ImageConstPtr& msg);
34      void configCallback(l2bot_examples::StopOnWhiteConfig &config, uint32_t level);
35
36  private:
37      float countWhite(const cv::Mat& src, cv::Mat& dst);
38
39      ros::NodeHandle nh_;
40      image_transport::ImageTransport it_;
41      image_transport::Subscriber image_sub_;
42      ros::Publisher pub_;
43
44      dynamic_reconfigure::Server<l2bot_examples::StopOnWhiteConfig> server_;
45
46      float white_all_ratio_;
47      int thresh_value_;
48      int max_BINARY_value_;
49      int thresh_type_;
50      bool use_median_blur_;
51      int blur_amount_;
```

```
/**
 * Constructor
 * ===========
 *
 * Do all initilization code here. This way, our main() function only
 *   needs to instantiate the StopOnWhite object once and do nothing
 *   else (see main() below).
 *
 * In this case, we only need to set up the image subscriber
 */
StopOnWhite::StopOnWhite()
    :nh_{"~"}, it_{nh_}
{
    // Subscribe to the camera publisher node
    image_sub_ = it_.subscribe(CAM_TOPIC, 1, &StopOnWhite::imageCb, this);

    // Publish on the l2bot twist command topic
    pub_ = nh_.advertise<geometry_msgs::Twist>(TWIST_PUB, 10);

    // Dynamic Reconfigure
    server_.setCallback(boost::bind(&StopOnWhite::configCallback, this, _1, _2));

    // Default values
    thresh_value_ = 180;
    white_all_ratio_ = 0.5;
    thresh_type_ = cv::THRESH_BINARY;
    max_BINARY_value_ = 255;
    use_median_blur_ = true;
    blur_amount_ = 3;
}
```

```
/**
 * Destructor
 * ==========
 *
 * Destroy CV windows
 */
StopOnWhite::~StopOnWhite()
{
    cv::destroyWindow(CVWIN_PREVIEW);
}
```

```
101  /**
102   * Dynamic Reconfigure Callback
103   * ============================
104   *
105   * This function is called every time the Dynamic Reconfigure UI
106   *   is updated by the user.
107   */
108  void StopOnWhite::configCallback(l2bot_examples::StopOnWhiteConfig &config, uint32_t level)
109  {
110      white_all_ratio_  = config.white_all_ratio;
111      thresh_value_     = config.thresh_value;
112      max_BINARY_value_ = config.max_value;
113      use_median_blur_  = config.use_median_blur;
114      blur_amount_      = config.blur_amount;
115
116      switch (config.thresh_type) {
117          case 0: thresh_type_ = cv::THRESH_BINARY; break;
118          case 1: thresh_type_ = cv::THRESH_BINARY_INV; break;
119          case 2: thresh_type_ = cv::THRESH_TRUNC; break;
120          case 3: thresh_type_ = cv::THRESH_TOZERO; break;
121          case 4: thresh_type_ = cv::THRESH_TOZERO_INV; break;
122          case 5: thresh_type_ = cv::THRESH_OTSU; break;
123      }
124  }
```

```
132     *
133     * Called once every time a image is published on the topic this
134     *   node is subscribed to. The image is passed to the function as
135     *   a ImageConstPtr
136     */
137    void StopOnWhite::imageCb(const sensor_msgs::ImageConstPtr& msg)
138    {
139        //Convert to cv image
140        cv_bridge::CvImagePtr cv_ptr;
141        try
142        {
143            cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
144        }
145        catch (cv_bridge::Exception& e)
146        {
147            ROS_ERROR("cv_bridge exception: %s", e.what());
148            return;
149        }
150
151        // Apply a threshold and count the number of white pixels
152        cv::Mat preview;
153        float white_visible = countWhite(cv_ptr->image, preview);
154
155        // Uncomment to print pixel ratio to terminal
156        //ROS_INFO_STREAM("white/all pixel ratio: " << white_visible);
157
158        // If the number of white pixels is above a certain percent, stop
159        geometry_msgs::Twist twist;
160        if (white_visible > white_all_ratio_)
161        {
162            twist.linear.x = 0.0;
163            ROS_INFO_STREAM("Stopping!!");
164        }
165        else
166        {
167            twist.linear.x = 2.0;
168        }
169        pub_.publish(twist);
170
171        // Show preview window
172        cv::imshow(CVWIN_PREVIEW, preview);
173
174        // Update GUI Window
175        cv::waitKey(3);
```

```
180  /**
181   * Count White Pixels Example
182   * ==========================
183   * This function uses dynamic thresholding to binarize the imput image.
184   * It then counts the number of white pixels and returns (#white)/(#total)
185   *
186   * https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html?highlight=threshold
187   */
188  float StopOnWhite::countWhite(const cv::Mat& src, cv::Mat& preview)
189  {
190      // Convert the source to grayscale
191      cv::Mat src_gray;
192      cv::cvtColor(src, src_gray, CV_BGR2GRAY);
193
194      // Blur the image to reduce noise (kernel must be odd)
195      if (use_median_blur_) cv::medianBlur(src_gray, src_gray, 2*blur_amount_ + 1);
196
197      // Threshold parameters
198      cv::threshold(src_gray, preview, thresh_value_, max_BINARY_value_, thresh_type_);
199
200      // The number of white pixels
201      int white_count = cv::countNonZero(preview);
202
203      // Function return ratio #white / #total
204      return (float)white_count / (float)(src.rows * src.cols);
205  }
```

```
210   int main(int argc, char** argv)
211   {
212       ros::init(argc, argv, "stop_on_white");
213
214       // Create a StopOnWhite object.
215       // Since initilization code is in the constructor, we do
216       //   not need to do anythong else with this object
217       StopOnWhite sd{};
218
219       ROS_INFO_STREAM("stop_on_white running!");
220       ros::spin();
221       return 0;
222   }
```

```
include <ros/ros.h>
include <geometry_msgs/Twist.h>
include <dynamic_reconfigure/server.h>
include <l2bot_examples/StopOnWhiteConfig.h>
include <image_transport/image_transport.h>
include <cv_bridge/cv_bridge.h>
include <sensor_msgs/image_encodings.h>
```

```
include <opencv2/highgui/highgui.hpp>
```

These lines include the header for ROS, a dynamic reconfigure parameter server, config file created earlier, and headers for working with images.

```
int main(int argc, char** argv)
{
   ros::init(argc, argv, "stop_on_white");
   StopOnWhite sd{};
   ……
}
```

In the main function, after initializing the ROS rode named “stop_on_white”, an instance of `SimpleCamSub` object is created. In the following `StopOnWhite` class, we define the `imageCb` callback function that will take an image as a `ImageConstPtr`, and `configCallback` callback function. we also create a node handler, `image_transport` to help send ROS Image messages across the ROS computing graph, and a publisher for L2bot twist message. Then a server is initialized through `dynamic_reconfigure::Server` where we pass the `StopOnWhiteConfig` configuration file. The remaining definitions are for handling parameter values.

```
class StopOnWhite
{
public:
   StopOnWhite();
   ~StopOnWhite();
   void imageCb(const sensor_msgs::ImageConstPtr& msg);
   void configCallback(l2bot_examples::StopOnWhiteConfig &config, uint32_t level);
private:
   float countWhite(const cv::Mat& src, cv::Mat& dst);
   ros::NodeHandle nh_;
   image_transport::ImageTransport it_;
   image_transport::Subscriber image_sub_;
   ros::Publisher pub_;
   dynamic_reconfigure::Server<l2bot_examples::StopOnWhiteConfig> server_;
   ……
}
StopOnWhite::StopOnWhite()
{
   image_sub_ = it_.subscribe(CAM_TOPIC, 1, &StopOnWhite::imageCb, this);
   pub_ = nh_.advertise<geometry_msgs::Twist>(TWIST_PUB, 10);
```

```
    server_.setCallback(boost::bind(&StopOnWhite::configCallback, this, _1, _2));
    ……
}
```

In the `StopOnWhite` constructor, a subscriber for the input image topic and publisher for the Twist message are created. Whenever an image arrives on the input image topic, it will call a function named `imageCb`. Every time the node receives a message, it takes the data from the message, creates a new `geometry_msgs::Twist` message with the movement commands, and publishes it. We also set a `callback` and send the `callback` function to the server. When the server gets a reconfiguration request, it will call the callback function. The remaining is for initialize default values for variables.

```
void StopOnWhite::configCallback(l2bot_examples::StopOnWhiteConfig &config, uint32_t level)
{
    white_all_ratio_  = config.white_all_ratio;
    ……
    switch (config.thresh_type) {
        case 0: thresh_type_ = cv::THRESH_BINARY; break;
        case 1: thresh_type_ = cv::THRESH_BINARY_INV; break;
        case 2: thresh_type_ = cv::THRESH_TRUNC; break;
        case 3: thresh_type_ = cv::THRESH_TOZERO; break;
        case 4: thresh_type_ = cv::THRESH_TOZERO_INV; break;
        case 5: thresh_type_ = cv::THRESH_OTSU; break;
}
```

In the `configCallback` function, first we set new values for the parameters. The way to access the parameters is, for example, `config.white_all_ratio`. The name of the parameter must be the same as the one that you configured in the .cfg file. Then we use switch cases to define different styles of thresholding. `THRESH_BINARY` gives the most common type of binary thresholding. `THRESH_BINARY_INV` is the opposite of binary thresholding. The destination pixel is set to zero if the corresponding source pixel is greater than the threshold, and to maxValue if the source pixel is less than the threshold. In the type of `THRESH_TRUNC`, the destination pixel is set to the threshold if the source pixel value is greater than the threshold; otherwise it is set to the source pixel value. maxValue is ignored. In the type of `THRESH_TOZERO`, the destination pixel is set to the corresponding source pixel value if the source pixel value is greater than the threshold; otherwise it is set to zero. maxValue is ignored. `THRESH_TOZERO_INV` is the opposite of `THRESH_TOZERO`. In this case, the destination pixel value is set to zero if the source pixel value greater than the threshold; Otherwise it is set to the source pixel value. `THRESH_OTSU` is the Otsu's Binarization. This algorithm finds the optimal threshold value from image histogram and returns you as the second output, retVal.

```
void StopOnWhite::imageCb(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    try
```

```
    {
        cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }
```

In the subscriber imageCb callback, it converts the sensor_msgs/Image data into the cv::Mat OpenCV data type. The cv_bridge::CvImagePtr cv_ptr buffer is allocated for storing the OpenCV image after performing the ROS-OpenCV conversion using the cv_bridge::toCvCopy function. Since we will count white pixels on the image, we need a mutable copy of it, so we use `toCvCopy()`. `sensor_msgs::image_encodings::BGR8` is constant for "bgr8". Finally, we should always wrap calls to `toCvCopy()` to catch conversion errors as this function will not check for the validity of the data.

```
    cv::Mat preview;
    float white_visible = countWhite(cv_ptr->image, preview);
    geometry_msgs::Twist twist;
    if (white_visible > white_all_ratio_)
    {
        twist.linear.x = 0.0;
    }
    Else
    {
        twist.linear.x = 2.0;
    }
    pub_.publish(twist);
    cv::imshow(CVWIN_PREVIEW, preview);
    cv::waitKey(3);
}
```

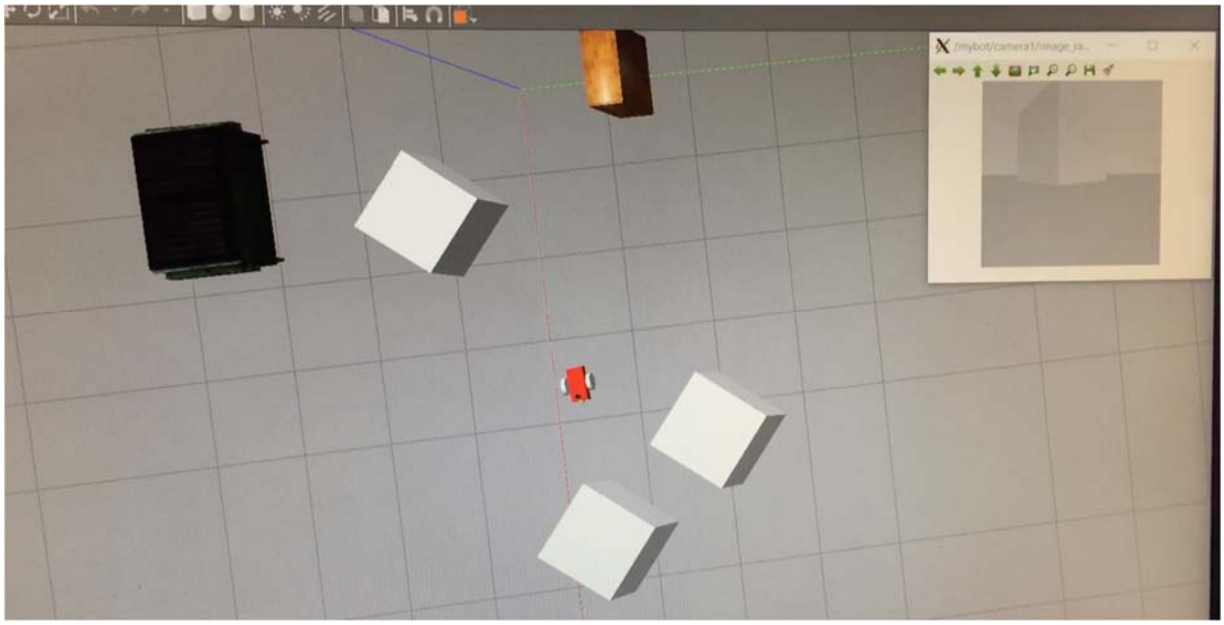
Then we call `countWhite()` function to count white pixels on the OpenCV image pointed by `cv_ptr`, and create the message we are going to send, `twist` of the type `geometry_msgs::Twist`. Then we compare the counted white pixel value with the parameter `white_all_ratio`: if the white pixel counted is larger than the parameter, the robot will stop; otherwise, it will continue moving forward. After comparing, we publish the message, and update the GUI to show image.

```
float StopOnWhite::countWhite(const cv::Mat& src, cv::Mat& preview)
{
    cv::Mat src_gray;
```

```
    cv::cvtColor(src, src_gray, CV_BGR2GRAY);
    if (use_median_blur_) cv::medianBlur(src_gray, src_gray, 2*blur_amount_ + 1);
    cv::threshold(src_gray, preview, thresh_value_, max_BINARY_value_, thresh_type_);
    int white_count = cv::countNonZero(preview);
    return (float)white_count / (float)(src.rows * src.cols);
}
```

In the `countWhite()` callback function, we first call `cvtColor` to convert the input image from BGR to grayscale, and then blur the grayscale image using the median filter. Next, we sue dynamic thresholding to binarize the input image. Then, it counts the number of white pixels and return the number of white pixels / the number of total pixels.

In Gazebo Simulation, the robot will stop when see a white object.



Gazebo_StopOnWhite video link

https://youtu.be/67sDI8lKVVo

Following line

```
#include "linedetect.hpp"
#include <cv_bridge/cv_bridge.h>
#include <cstdlib>
#include <string>
#include <opencv2/highgui/highgui.hpp>
#include "ros/ros.h"
#include "opencv2/opencv.hpp"
#include "ros/console.h"
#include "line_follower_turtlebot/pos.h"

void LineDetect::imageCallback(const sensor_msgs::ImageConstPtr& msg) {
  cv_bridge::CvImagePtr cv_ptr;
  try {
    cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    img = cv_ptr->image;
    cv::waitKey(30);
  }
  catch (cv_bridge::Exception& e) {
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());
  }
}

cv::Mat LineDetect::Gauss(cv::Mat input) {
  cv::Mat output;
// Applying Gaussian Filter
  cv::GaussianBlur(input, output, cv::Size(3, 3), 0.1, 0.1);
  return output;
}
```

```
int LineDetect::colorthresh(cv::Mat input) {
  // Initializaing variables
  cv::Size s = input.size();
  std::vector<std::vector<cv::Point> > v;
  auto w = s.width;
  auto h = s.height;
  auto c_x = 0.0;
  // Detect all objects within the HSV range
  cv::cvtColor(input, LineDetect::img_hsv, CV_BGR2HSV);
  LineDetect::LowerYellow = {20, 100, 100};
  LineDetect::UpperYellow = {30, 255, 255};
  cv::inRange(LineDetect::img_hsv, LowerYellow,
   UpperYellow, LineDetect::img_mask);
  img_mask(cv::Rect(0, 0, w, 0.8*h)) = 0;
  // Find contours for better visualization
  cv::findContours(LineDetect::img_mask, v, CV_RETR_LIST, CV_CHAIN_APPROX_NONE);
  // If contours exist add a bounding
  // Choosing contours with maximum area
  if (v.size() != 0) {
  auto area = 0;
  auto idx = 0;
  auto count = 0;
  while (count < v.size()) {
    if (area < v[count].size()) {
       idx = count;
       area = v[count].size();
    }
    count++;
  }
  cv::Rect rect = boundingRect(v[idx]);
  cv::Point pt1, pt2, pt3;
  pt1.x = rect.x;
  pt1.y = rect.y;
  pt2.x = rect.x + rect.width;
  pt2.y = rect.y + rect.height;
  pt3.x = pt1.x+5;
  pt3.y = pt1.y-5;
```

```
  // Drawing the rectangle using points obtained
  rectangle(input, pt1, pt2, CV_RGB(255, 0, 0), 2);
  // Inserting text box
  cv::putText(input, "Line Detected", pt3,
    CV_FONT_HERSHEY_COMPLEX, 1, CV_RGB(255, 0, 0));
  }
  // Mask image to limit the future turns affecting the output
  img_mask(cv::Rect(0.7*w, 0, 0.3*w, h)) = 0;
  img_mask(cv::Rect(0, 0, 0.3*w, h)) = 0;
  // Perform centroid detection of line
  cv::Moments M = cv::moments(LineDetect::img_mask);
  if (M.m00 > 0) {
    cv::Point p1(M.m10/M.m00, M.m01/M.m00);
    cv::circle(LineDetect::img_mask, p1, 5, cv::Scalar(155, 200, 0), -1);
  }
  c_x = M.m10/M.m00;
  // Tolerance to chooise directions
  auto tol = 15;
  auto count = cv::countNonZero(img_mask);
  // Turn left if centroid is to the left of the image center minus tolerance
  // Turn right if centroid is to the right of the image center plus tolerance
  // Go straight if centroid is near image center
  if (c_x < w/2-tol) {
    LineDetect::dir = 0;
  } else if (c_x > w/2+tol) {
    LineDetect::dir = 2;
  } else {
    LineDetect::dir = 1;
  }
  // Search if no line detected
  if (count == 0) {
    LineDetect::dir = 3;
  }
  // Output images viewed by the turtlebot
  cv::namedWindow("Turtlebot View");
  imshow("Turtlebot View", input);
  return LineDetect::dir;
}
```

```
int main(int argc, char **argv) {
    // Initializing node and object
    ros::init(argc, argv, "detection");
    ros::NodeHandle n;
    LineDetect det;
    // Creating Publisher and subscriber
    ros::Subscriber sub = n.subscribe("/camera/rgb/image_raw",
        1, &LineDetect::imageCallback, &det);

    ros::Publisher dirPub = n.advertise<
    line_follower_turtlebot::pos>("direction", 1);
        line_follower_turtlebot::pos msg;

    while (ros::ok()) {
        if (!det.img.empty()) {
            // Perform image processing
            det.img_filt = det.Gauss(det.img);
            msg.direction = det.colorthresh(det.img_filt);
            // Publish direction message
            dirPub.publish(msg);
            }
        ros::spinOnce();
    }
    // Closing image viewer
    cv::destroyWindow("Turtlebot View");
}
```

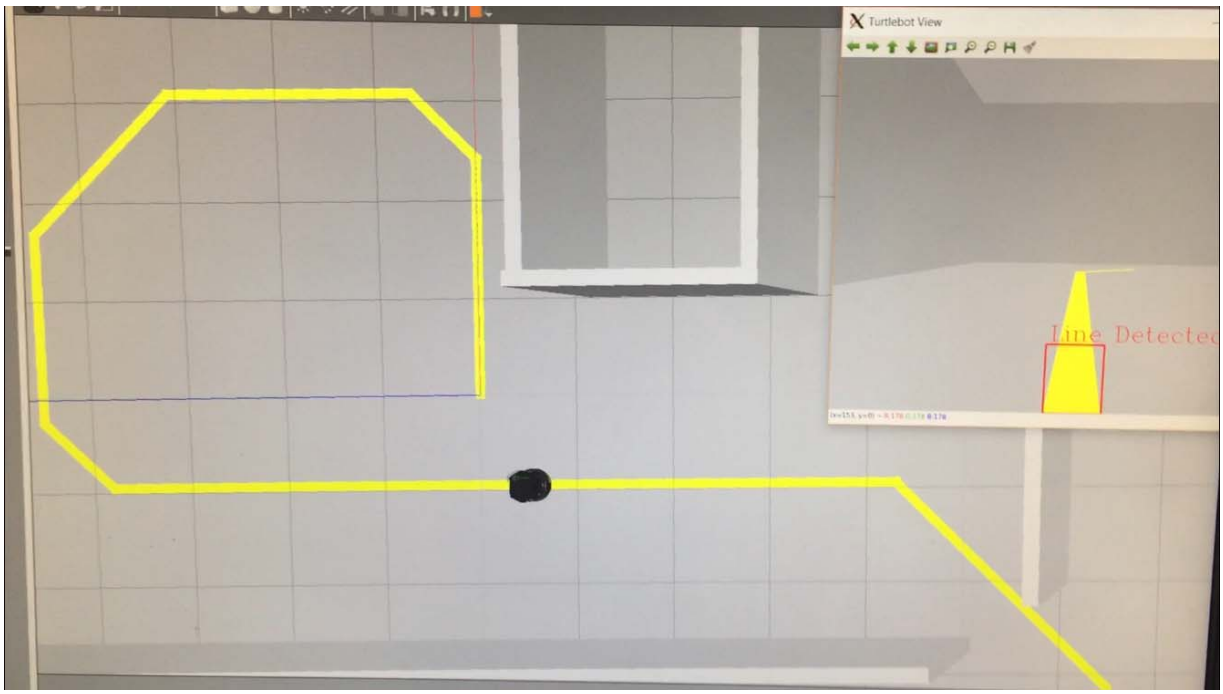
This part of code is for line detection. In the main, we initialize a node and a node handler, as well as an instance of LineDetect object. Then we create a subscriber for the input image and publisher for turtlebot movement direction. Whenever an image arrives on the input image topic, `imageCallback()` will be called. In the while loop, while image is not empty, we will do image processing and publish the turtlebot movement message.

In the LineDetect:: imageCallback function, first we covert the ROS image to CvImage suitable for working with OpenCV. Then we apply Gaussian Filter on the image. In the core part of line detection, we first detect the upper and lower part of the line within the HSV range, and find the contours of the line and draw bounding box. Next, we detect the centroid of the line, and perform left or right turn and straight forward based on the position of centroid on the image. Finally, we show the image viewed by turtlebot.

```
#include <geometry_msgs/Twist.h>
#include <vector>
#include "ros/ros.h"
#include "ros/console.h"
#include "turtlebot.hpp"
#include "line_follower_turtlebot/pos.h"
void turtlebot::dir_sub(line_follower_turtlebot::pos msg) {
    turtlebot::dir = msg.direction;
}
void turtlebot::vel_cmd(geometry_msgs::Twist &velocity,
 ros::Publisher &pub, ros::Rate &rate) {
    // If direction is left
    if (turtlebot::dir == 0) {
        velocity.linear.x = 0.1;
        velocity.angular.z = 0.15;
        pub.publish(velocity);
        rate.sleep();
        ROS_INFO_STREAM("Turning Left");
    }
    // If direction is straight
    if (turtlebot::dir == 1) {
        velocity.linear.x = 0.15;
        velocity.angular.z = 0;
        pub.publish(velocity);
        rate.sleep();
        ROS_INFO_STREAM("Straight");
    }
    // If direction is right
    if (turtlebot::dir == 2) {
        velocity.linear.x = 0.1;
        velocity.angular.z = -0.15;
        pub.publish(velocity);
        rate.sleep();
        ROS_INFO_STREAM("Turning Right");
    }
    // If robot has to search
    if (turtlebot::dir == 3) {
        velocity.linear.x = 0;
        velocity.angular.z = 0.25;
        pub.publish(velocity);
        rate.sleep();
        ROS_INFO_STREAM("Searching");
    }
}
```

```
int main(int argc, char **argv) {
    // Initializing node and object
    ros::init(argc, argv, "Velocity");
    ros::NodeHandle n;
    turtlebot bot;
    geometry_msgs::Twist velocity;
    // Creating subscriber and publisher
    ros::Subscriber sub = n.subscribe("/direction",
        1, &turtlebot::dir_sub, &bot);
    ros::Publisher pub = n.advertise<geometry_msgs::Twist>
        ("/cmd_vel_mux/input/teleop", 10);
    ros::Rate rate(10);
    while (ros::ok()) {
        ros::spinOnce();
        // Publish velocity commands to turtlebot
        bot.vel_cmd(velocity, pub, rate);
        rate.sleep();
    }
    return 0;
}
```

In this part of the code, we perform turtlebot movement. In the main, we initialize a node and a node handler, an instance for turtlebot object, and a geometry_msgs message object. Then we created a subscriber to subscribe to the direction topic and a publisher to publish a message type of geometry_msgs on the topic teleop. In the while loop, we publish the geometry_msgs message commands to control turtlebot movement.



Gazebo_LineFollow video link

https://youtu.be/PUEOByChnUs

https://youtu.be/1xAOLqfhr2I