# Course 3- ROS Essentials

## Introduction to ROS

### Welcome to ROS Essentials

Video 1.1

### Build Robots with ROS

Building a robot used to be a long and cumbersome process. Essentially all of the components had to be built from scratch.

In this lesson, we’ll introduce the Robot Operating System, or “ROS”, which is a software framework that greatly simplifies robot development. There are many advantages to developing robots with ROS. Let’s start by illustrating some of its components and features.

* Components and Features

ROS is an open-source software framework for robotics development. It is not an operating system in the typical sense. But like an OS, it provides a means of communicating with hardware. It also provides a way for different processes to communicate with one another via message passing. Lastly, ROS features a slick build and package management system called catkin, allowing you to develop and deploy software with ease. ROS also has tools for visualization, simulation, and analysis, as well as extensive community support and interfaces to numerous powerful software libraries.

**Summary**

Summary of ROS components and features:

* Open-source!
* Hardware abstraction of device drivers
* Communication via message passing
* Slick build and package management
* Tools for visualization, simulation, analysis
* Powerful software libraries
* Short Documentary

Check out this awesome [**short documentary on ROS**](https://www.bloomberg.com/news/videos/2017-05-17/the-silicon-valley-startup-creating-robot-dna-video) that Bloomberg published recently.

**History**

Before diving deeper, let’s take a brief tour of the history of ROS.

### Brief History of ROS

Video 1.3

### Nodes and Topics

Video 1.4

A screenshot of a questionnaire

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

### Message Passing

Video 1.5

A screenshot of a question

AI-generated content may be incorrect.

### Services

Video 1.6

A screenshot of a question

AI-generated content may be incorrect.

### Compute Graph

Video 1.7

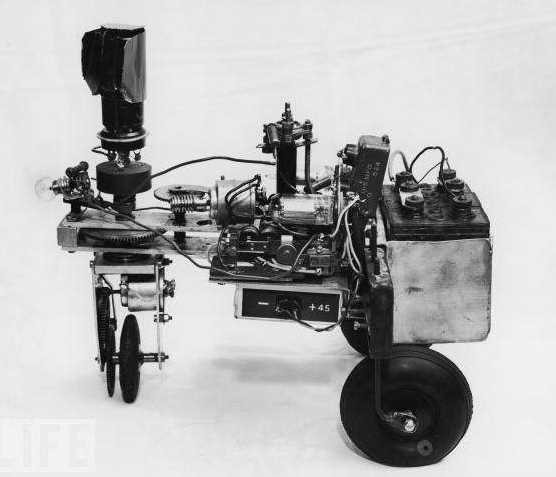
### Turtlesim Overview

Video 1.8

**More on the history of Turtles in Robotics**

As we mentioned in the video, William Grey Walter's influence is still felt today. He referred to his robots as 'turtles' and, as you will see, the moniker stuck.

The image below is William Walter's Elsie (the robot mentioned in the video) without her protective covering.



Elsie without her protective cover

Long after William Walter’s work with Elmer and Elsie, Dr. Seymour Papert, a professor at MIT, began to use turtle robots for education. One of the characteristics of Papert's robots was their ability to draw on paper.

In addition to being involved with the creation and development of MIT’s turtle robots, Dr. Papert is also known as the creator and evangelist for the educational programming language LOGO.

Despite being a general-purpose language, LOGO is known for its use of “[**turtle graphics**](https://en.wikipedia.org/wiki/Turtle_graphics)”, a system that allows users to draw by sending simple commands to a robotic turtle. The robotic turtle mentioned here could be either a real turtle robot, or a virtual on-screen cursor within the LOGO programming environment.

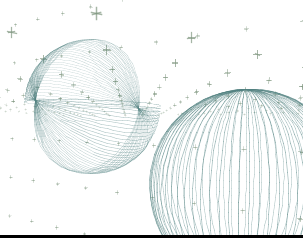
The image below shows an example of Valiant Technology’s Turtle robot drawing on a sheet of paper.



Robot using turtle graphics

While turtle graphics seem simple, people have used them to create a wide variety of interesting drawings and art.

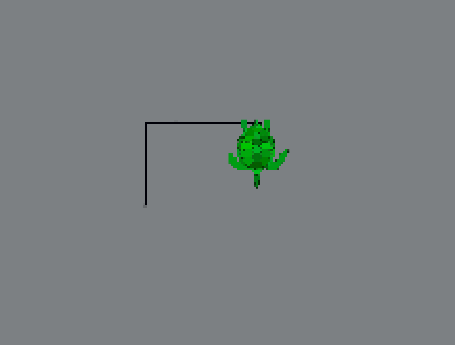
The image below depicts some drawings of three dimensional spheres creating using turtle graphics. The website [**turtleart.org**](http://turtleart.org/) hosts a gallery containing even more outstanding examples!



Three dimensional Turtle Graphic

If you'd like to have some fun with turtle graphics, feel free to try out the [**Turtle Graphics 1.01**](https://scratch.mit.edu/projects/1250518/) project, which allows you to experiment with turtle graphics through MIT’s graphical programming environment scratch.

Below is the result of a program written using Turtle Graphics. Looking at this image, answer the quiz question about it.



A screenshot of Turtle Graphics 1.01

A screenshot of a question

AI-generated content may be incorrect.



This tradition of turtles in robotics is alive and well today. In fact, each recent version of ROS has been named after some sort of turtle. In addition to this turtle-centric naming convention, the Open-Source Robotics Foundation also adds a new turtle to turtle\_sim with each release.

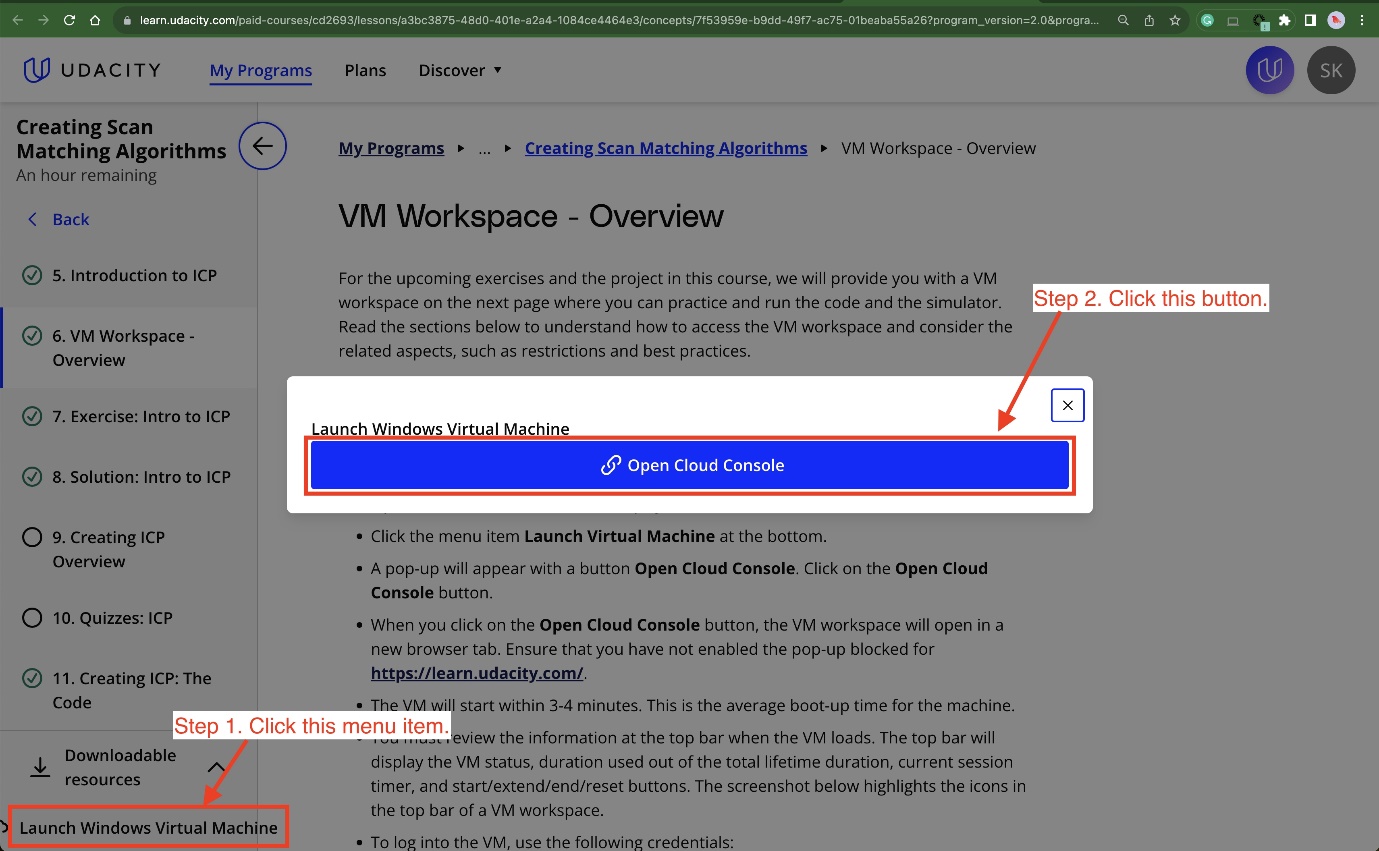
### Ubuntu VM Workspace-Overview

For the upcoming exercises/project in this course, we will provide you with a VM workspace. Read the sections below to understand how to access the VM workspace and consider the related aspects, such as restrictions and best practices.

**VM Workspace - Access**

Follow the instructions below to access your VM workspace.

* Open the left menu on the current page and scroll to the bottom of the menu.
* Click the menu item **Launch Virtual Machine** at the bottom.
* A pop-up will appear with a button **Open Cloud Console**. Click on the **Open Cloud Console** button.



A screenshot highlighting the steps to open a VM workspace.

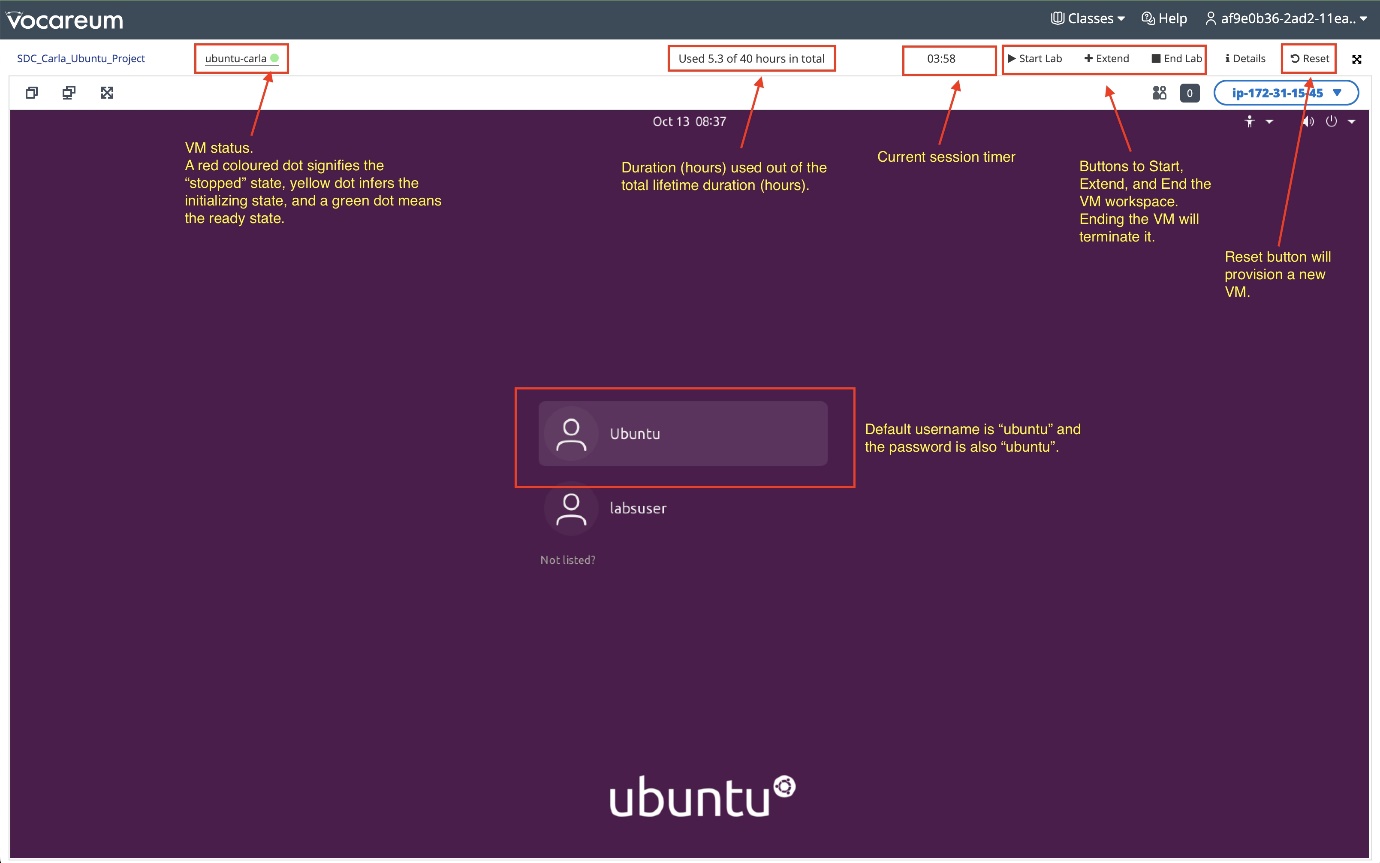
* When you click on the **Open Cloud Console** button, the VM workspace will open in a new browser tab. Ensure that you have not enabled the pop-up blocked for [**https://learn.udacity.com/**](https://learn.udacity.com/).
* The VM will start within 3-4 minutes. This is the average boot-up time for the machine.
* You must review the information at the top bar when the VM loads. The top bar will display the VM status, duration used out of the total lifetime duration, current session timer, and start/extend/end/reset buttons. The screenshot below highlights the icons in the top bar of a VM workspace.

To log into the VM, use the following credentials:

* Username: ubuntu
* Password: ubuntu

Also, an additional pair of credentials are:

* Username: labsuser
* Password: vocareum



A screenshot of a VM workspace

**VM Workspace - Overview**

The VM workspace is an Ubuntu 20.04 machine with NVIDIA Tesla T4 GPU support. It has all the necessary tools installed that you will need to run the exercises and the project in this course. For example, it has the following major tools and corresponding configurations:

* [**CARLA simulator 0.9.9.4**](https://github.com/carla-simulator/carla/releases/tag/0.9.9)
* [**NICE DCV Server**](https://docs.aws.amazon.com/dcv/latest/adminguide/setting-up-installing-linux-prereq.html). This includes the Nvidia drivers along with CUDA libraries for the underlying Tesla T4 GPU
* C++
* Git
* [**OpenCV**](https://docs.opencv.org/4.x/d7/d9f/tutorial_linux_install.html)
* [**CMake**](https://askubuntu.com/questions/161104/how-do-i-install-make) and Make
* [**VSCode**](https://code.visualstudio.com/download)
* [**Eigen Library for C++**](https://eigen.tuxfamily.org/index.php?title=Main_Page)
* [**Point Cloud Library**](https://pointclouds.org/downloads/)
* Python3 and Pip
* ROS

**However, you can install more tools as necessary for the classroom exercises or the project.**

An important aspect is that the VM workspace has several restrictions, as explained below. You must read the restrictions and best practices below before you plan your work in the VM workspace.

* VM Workspace - Restrictions and Best Practices

1. The VM workspace is a **Sandbox environment**. When you access the VM workspace, a session initiates for a limited duration.
   * After the session expires, the VM workspace will terminate.
   * You must reload the classroom page to access the VM workspace and thus start a new session.
   * Udacity will not preserve your work or any custom installations for your next session. You will get a fresh new VM workspace in each new attempt.
   * **Best practice**: You must save your edits to the starter/solution code and the Readme file in a Github repo so you can pull it again in your next session.
2. The VM workspace has a limited session duration and a fixed total lifetime duration. For example, each session can be four hours, and the total lifetime duration could be 40 hours.
   * You will find the session timer and the total lifetime duration at the top bar of your VM workspace.
   * When the VM workspace session is about to end, you will get a prompt to extend the session duration by another hour. You can make such extensions for up to four more hours. This means you will get the initial fixed session duration and the extended session duration in one session.

For example, if the initial fixed session duration is four hours, and you have made four extensions, you will get eight hours in a single session.

* + **Best practice**: You must save your work intermittently to avoid potential data loss if the VM workspace session ends abruptly.

1. **Best practice**: As Udacity pays for every minute of runtime, you must click the **End lab** button at the top to terminate the VM workspace when not in use. This step will ensure you have sufficient time remaining for the project.
2. Misusing the VM workspace can lead to **revoking access** or deactivating your Udacity account. Please follow the instructions described in the exercise/project instructions. Udacity monitors VM workspace usage to ensure the ethical use of resources for educational purposes only.

**VM Workspace - Getting Help**

If the VM workspace does not load for you, file a support ticket with Udacity. You will find the link for filing a support ticket at the bottom-right of your classroom page.

### Source the ROS Environment

Video 1.10

**Environment Setup**

* Open **Udacity's VM workspace** (Ubuntu 20.04 LTS) or the **local VM image** (Ubuntu 18.04 LTS) running on your VMWare/VirtualBox to practice the current exercise. If you haven't already, review the overview and restrictions outlined in the **Ubuntu VM Workspace - Overview** page.
* Once you log into the VM, open a Terminal window.
* Before we begin using ROS in a terminal, we must first ensure that all of the environment variables are present. To do this, we must source the setup script provided by ROS

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

* You must source this script in every bash terminal you use ROS in. However, it can be convenient to automatically source this script every time a new shell is launched. These commands will do that for you.

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

*source ~/.bashrc*

**Important Note**

The instructions in the video used ROS **Kinetic**, so this previous command should work well on the VM image, however for the VM Workspace, the instructions uses ROS **Noetic**, so you will just need to adjust the path as follows:

*echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc*

*source ~/.bashrc*

* Caveat

Make sure you use the bash command source rather than ./. There’s a subtle distinction between the two commands, in that source executes the script in the current session, while ./ will start a new session, containing a copy of the current environment. When a script executed via ./ is exited, all environment variables set by it will be lost. We don’t want this. For more information on environment variables and terminal sessions, please see [**here**](https://help.ubuntu.com/community/EnvironmentVariables).

A screenshot of a test

AI-generated content may be incorrect.

### Run Turtlesim

Video 1.11

Now that you’ve added ROS’ environment variables to your terminal session, you can run the turtlesim package!

**Starting the Master process**

Before you can run any ROS nodes, you must start the Master process.

The Master process is responsible for the following (and more):

* Providing naming and registration services to other running nodes
* Tracking all publishers and subscribers
* Aggregating log messages generated by the nodes
* Facilitating connections between nodes

To run the master process, execute the command

*roscore*

If all goes well (and it should), you will see output similar to this:

A screenshot of a computer

AI-generated content may be incorrect.

This indicates that ROS Master is running! To terminate the ROS Master process, you can simply send the SIGINT signal to the process by pressing ctrl-c in the roscore terminal window.

We are now ready to begin launching nodes!

**Running Turtlesim Nodes**

Now that the ROS Master is running, we can run our first two ROS nodes.

To do so, we will execute the rosrun command in a new terminal window, passing as parameters the name of the package we wish to run, and the name of the actual node.

**Note**

You can open a second instance of you current terminal using CTRL + SHIFT + T.

Tab completion is your friend. Each ROS distribution comes with a staggering number of packages, and an even more staggering number of nodes. In the bash shell, a single-tap of the tab key will autocomplete the command, if there is a single match. A double-tap of the tab key will result in a list of all possible matches, in the case that a single match cannot be found.

First we will start the turtlesim\_node, in the turtlesim package using the following command in a second terminal instance.

*$ rosrun turtlesim turtlesim\_node*

Next, we will start the turtle\_teleop\_key node, also from the turtlesim package in a third terminal instance.

*$ rosrun turtlesim turtle\_teleop\_key*

By using the arrow keys with the turtle\_teleop\_key node’s console selected, we are able to move the turtle in turtlesim!

A screenshot of a computer

AI-generated content may be incorrect.

### Turtlesim Comms: List Nodes

**Turtlesim Comms: List All Active Nodes**

In the following sections, we will investigate Turtlesim communication commands, and will cover the following steps:

* Listing all active nodes
* Listing all topics
* Getting information about topics
* Showing message information
* Echoing messages in real-time

Now that we’ve launched turtlesim\_node and played around with sending commands via the turtle\_teleop\_key node, let’s dig deeper, to see what’s actually happening underneath the surface...

Video 1.12

**Listing all Active Nodes**

To get a list of all nodes that are active and have been registered with the ROS Master, we can use the command rosnode list. Let’s do so now:

A screenshot of a computer

AI-generated content may be incorrect.

We can see that there are three active nodes that have been registered with the ROS Master: /rosout, /teleop\_turtle, and /turtlesim.

* /rosout: This node is launched by roscore. It subscribes to the standard /rosout topic, the topic to which all nodes send log messages.
* /teleop\_turtle: This is our keyboard teleop node. Notice that its not named turtle\_teleop\_key. There’s no requirement that a node’s broadcasted name is the same as the name of it’s associated executable.
* /turtlesim: The node name associated with the turtlebot\_sim node.

### Turtlesim Comms: List Topics

Video 1.13

**Listing All Topics**

In a similar fashion, we are able to query the ROS Master for a list of all topics. To do so, we use the command rostopic list.

A screenshot of a computer

AI-generated content may be incorrect.

* /rosout\_agg: Aggregated feed of messages published to /rosout.
* /turtle1/cmd\_vel: Topic on which velocity commands are sent/received. Publishing a velocity message to this topic will command turtle1 to move.
* /turtle1/color\_sensor: Each turtle in turtlesim is equipped with a color sensor, and readings from the sensor are published to this topic.
* /turtle1/pose: The position and orientation of turtle1 are published to this topic.

### Turtlesim Comms: Get Topic Info

Video 1.14

**Get Information About a Specific Topic**

If we wish to get information about a specific topic, who is publishing to it, subscribed to it, or the type of message associated with it, we can use the command rostopic info . Let’s check into the /turtle1/cmd\_vel topic:

A screenshot of a computer

AI-generated content may be incorrect.

As you might expect, there are two nodes registered on this topic. One publisher, the teleop\_turtle node, and one subscriber, the turtlesim node. Additionally, we see that the type of message used on this topic is geometry\_msgs/Twist.

### Turtlesim Comms: Message Information

Video 1.15

**Turtlesim Comms: Show Message Information**

Let’s get some more information about the geometry\_msgs/Twist message on the /turtle1/cmd\_vel topic, to do so, we will use the rosmsg info command.

A screenshot of a computer

AI-generated content may be incorrect.

We can see that a Twist message consists nothing more than two Vector3 messages. One for linear velocity, and another for angular velocity, with each velocity component (x,y,z) represented by a float64.

**Note**

Sometimes the message definition doesn’t provide an ample amount of detail about a message type. For instance, in the example above, how can we be sure that linear and angular vectors above refer to velocities, and not positions? One way to get more detail would be to look at the comments in the message’s definition file. To do so, we can issue the following command: rosed geometry\_msgs Twist.msg.

More information about rosed, including how to select which editor is used by default, can be found [**here**](http://wiki.ros.org/ROS/Tutorials/UsingRosEd).

### Turtlesim Comms: Echo a Topic

Video 1.16

**Turtlesim Comms: Echo Messages on a Topic**

Sometimes it may be useful to look at a topic’s published messages in real time. To do so, we can use the command rostopic echo. Let’s take a look at the /turtle1/cmd\_vel topic.

$ rostopic echo /turtle1/cmd\_vel

If we then command the turtle to move from the turtle\_teleop\_key window, we will be able to see the output message in real-time!

A computer screen with a black background

AI-generated content may be incorrect.

### Recap

Video 1.17

## Packages & Catkin Workspaces

### Overview of Catkin Workspaces and Packages

Video 2.1

**Catkin packages**

ROS software is organized and distributed into packages, which are directories that might contain source code for ROS nodes, libraries, datasets, and more. Each package also contains a file with build instructions - the CMakeLists.txt file - and a package.xml file with information about the package. Packages enable ROS users to organize useful functionality in a convenient and reusable format.

**Catkin workspaces**

A catkin workspace is a top-level directory where you build, install, and modify catkin packages. The workspace contains all of the packages for your project, along with several other directories for the catkin system to use when building executables and other targets from your source code.

**Lesson Roadmap**

* Create a catkin workspace
* Add packages
* Manage inter-package dependencies
* Compile packages

### Create a Catkin Workspace

**Getting Started**

Open **Udacity's VM workspace** (Ubuntu 20.04 LTS) or the **local VM image** (Ubuntu 18.04 LTS) running on your VMWare/VirtualBox to practice the current exercise.

Once you log into the VM, open a Terminal window.

**Step 1: Create a catkin workspace and a sub directory**

All of the ROS-related code you develop throughout this course will reside in your catkin workspace. You only need to create and initialize the workspace once.

First, create the top level catkin workspace directory and a sub-directory named src (pronounced source). The top level directory name is arbitrary, but is often called catkin\_ws (an abbreviation of catkin\_workspace), so we will follow this convention. You can create these two directories in /home with a single command:

$ mkdir -p ~/catkin\_ws/src

**Step 2: Navigate to the source directory**

Next, navigate to the src directory with the cd command:

$ cd ~/catkin\_ws/src

**Step 3: Initialize the catkin workspace**

Now you can initialize the catkin workspace which will create a CMakeLists.txt file:

$ catkin\_init\_workspace

A screenshot of a computer

AI-generated content may be incorrect.

Let’s list the contents of the current directory to see what changed.

$ ls -l

Notice that a symbolic link (CMakeLists.txt) has been created to /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake

**Step 4: Return to top level directory**

Return to the catkin\_ws,

$ cd ~/catkin\_ws

**Step 5: Build the Workspace**

$ catkin\_make

**Note**: you must issue this command from within the top level directory (i.e., within catkin\_ws NOT catkin\_ws/src)

While it is not essential that you have a deep understanding of the catkin build system, particularly if you are doing most of your development work in Python, it is helpful to learn about it. We encourage you to read the [**ROS wiki**](http://wiki.ros.org/catkin/conceptual_overview).

After the command is executed you will notice the output of the build processes being echoed to your display. When it has finished you should see the following lines at the end of the output:

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

You now have two new directories: build and devel. The aptly named build directory is the build space for C++ packages and, for the most part, you will not interact with it. The devel directory does contain something of interest, a file named setup.bash. This setup.bash script must be sourced before using the catkin workspace with source devel/setup.bash

**Step 6: Commentary**

Congratulations! You just created your first catkin workspace.

**Optional**

Before you begin to work with and develop your own ROS package, you can take a moment to get acquainted with catkin workspace conventional directory structure as described in the ROS Enhancement Proposal (REP) 128 by clicking [**here**](http://www.ros.org/reps/rep-0128.html).

A screenshot of a computer

AI-generated content may be incorrect.

### Add a Package

Video 2.3

**Cloning the simple\_arm Package**

One of the biggest benefits of using ROS is that it has a really large community of users and developers who have already created a lot of code that you can reuse.

Let’s clone an existing package and add it to our newly created workspace.

Start by navigating to the src directory and cloning the simple\_arm package for this lesson from its GitHub repo.

*$ cd ~/catkin\_ws/src/*

*$ git clone -b first\_interaction https://github.com/udacity/RoboND-simple\_arm/ simple\_arm*

**Building the simple\_arm package**

After the repo has finished cloning, you can change directory to the top-level of the ROS workspace and build the new package.

*$ cd ~/catkin\_ws*

*$ catkin\_make*

**Getting Errors?**

Sometimes you might get a missing package error while building the catkin workspace. For example: "Could not find a package configuration file provided by controller\_manager"

I happen to know that controller\_manager refers to a ROS package from ROS Control. We can fix this by installing the associated Linux package, which itself contains and will install the necessary ROS package. But, if I didn't already know this, I would probably have to rely on a Google search to figure out the exact name of the package required.

For your convenience, this package is already installed in the workspace and you shouldn’t receive the error. But, you can always re-install the package with this command:

*$ sudo apt-get install ros-noetic-controller-manager*

The previous command should work well in the Udacity Workspace, however in case you use the VM image, please use **kinetic** rather than **noetic:**

*$ sudo apt-get install ros-kinetic-controller-manager*

### Roslaunch

Video 2.4

roslaunch allows you to do the following

* Launch the ROS Master and multiple nodes with one simple command
* Set default parameters on the parameter server
* Automatically re-spawn processes that have died

To use roslaunch, you must first make sure that your workspace has been built and sourced:

*$ cd ~/catkin\_ws*

*$ catkin\_make*

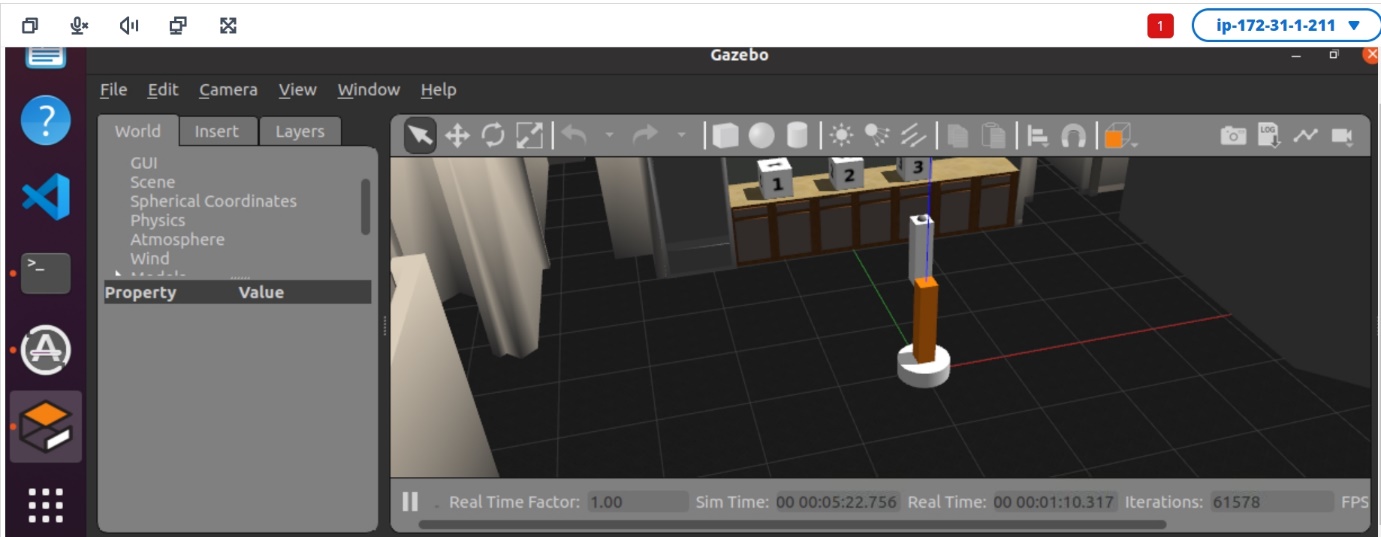
Once the workspace has been built, you can source it’s setup script:

*$ source devel/setup.bash*

With your workspace sourced you can now launch simple\_arm:

*$ roslaunch simple\_arm robot\_spawn.launch*

And there you have it! Your very own two-degrees-of-freedom arm in simulation!



Simple Arm

### Rosdep

**ROS Package Dependencies**

ROS packages have two types of dependencies: build dependencies and run dependencies.

The rosdep tool will check for a package's missing dependencies, download them, and install them.

To check for missing dependencies in a ROS package:

*$ rosdep check <package name>*

**Note**: In order for the command to work, the workspace must be first sourced with source devel/setup.bash.

This gives you a list of the system dependencies that are missing, and tells you where to get them.

To have rosdep install packages, invoke the following command from the root of the catkin workspace

*$ rosdep install -i <package name>*

**simple\_arm Package Dependencies**

Fortunately, the build and run dependencies for simple\_arm package have been already installed in the workspace. You can double-check using rosdep check:

*$ cd ~/catkin\_ws*

*$ source devel/setup.bash*

*$ rosdep check simple\_arm*

**Output:** All system dependencies have been satisfied.

**What’s next?**

With all of the packages properly installed, you will now learn more about ROS packages by creating yours from scratch, in preparation for writing your own nodes!

### Dive Deeper into Packages

You'll begin your dive into ROS packages by creating one of your own. All ROS packages should reside under the src directory.

Assuming you have already sourced your ROS environment and your catkin workspace, navigate to the src directory:

*$ cd ~/catkin\_ws/src*

The syntax for creating a catkin package is:

*$ catkin\_create\_pkg <your\_package\_name> [dependency1 dependency2 …]*

The name of your package is arbitrary but you will run into trouble if you have multiple packages with the same name in your catkin workspace. Try to make it descriptive and unique without being excessively long. Let’s name ours “first\_package” and we won’t specify any dependencies. By convention, package names are lowercase.

*$ catkin\_create\_pkg first\_package*

Voilà. You just created your first catkin package! Navigating inside our newly created package reveals that it contains just two files: CMakeLists.txt and package.xml. This is a minimum working catkin package. It is not very interesting because it doesn't do anything, but it meets all the requirements for a catkin package. One of the main functions of these two files is to describe dependencies and how catkin should interact with them. We won’t pay much attention to them right now.

I mentioned earlier that ROS packages have a conventional directory structure. Let’s take a look at a more typical package.

* scripts (python executables)
* src (C++ source files)
* msg (for custom message definitions)
* srv (for service message definitions)
* include -> headers/libraries that are needed as dependencies
* config -> configuration files
* launch -> provide a more automated way of starting nodes

Other folders may include

* urdf (Universal Robot Description Files)
* meshes (CAD files in .dae (Collada) or .stl (STereoLithography) format)
* worlds (XML like files that are used for Gazebo simulation environments)

A white and black text on a white background

AI-generated content may be incorrect.

You can install many ROS packages. To see a list of available packages for the Noetic distribution, take some time to explore [**the list of ROS package online**](https://index.ros.org/packages/#noetic).

### Recap

Video 2.7

**Congratulations!**

You now have some experience

* Creating workspaces
* Adding packages
* Managing Dependencies
* Troubleshooting build errors

**Note**

Before you begin the next lesson, it might be valuable to take a peek at the official [**ROS wiki**](http://wiki.ros.org/). There, you will be able to find in-depth documentation about some of the topics that have been covered up to this point.

## Write ROS Nodes

### Overview

Video 3.1

In this lesson, you’ll be learning how to write nodes in C++.

The first node that you will write is called simple\_mover. The simple\_mover node does nothing more than publish joint angle commands to simple\_arm.

Once you understand the general structure of a ROS node, you will write another node called arm\_mover. The arm\_mover node provides a service called safe\_move, which allows the arm to be moved to any position within its workspace that has been deemed safe. The safe zone is bounded by minimum and maximum joint angles, and is configurable via the ROS parameter server.

The last node you’ll write in this lesson is the look\_away node. This node subscribes to the arm joint positions and a topic where camera data is being published. When the camera detects an image with uniform color, meaning that it’s looking at the sky, and the arm is not moving, the node will call the safe\_move service via a client to move the arm to a new position.

### ROS in the VM Workspace

**ROS in the Workspace**

To follow along with this lesson, you will need to run ROS.

Open Udacity's VM workspace (Ubuntu 20.04 LTS) or the local VM image (Ubuntu 18.04 LTS) running on your VMWare/VirtualBox to practice the current exercise.

Once you log into the VM, open a Terminal window.

You're now ready to follow along in your development environment with this lesson!

### ROS Publishers

Before you see the code for simple\_mover, it may be helpful to see how ROS Publishers work in C++.

Publishers allow a node to send messages to a topic, so that data from the node can be used in other parts of ROS. In C++, ROS publishers typically have the following definition format, although other parameters and arguments are possible:

ros::Publisher pub1 = n.advertise<message\_type>("/topic\_name", queue\_size);

The pub1 object is a publisher object instantiated from the ros::Publisher class. This object allows you to publish messages by calling the publish() function.

To communicate with ROS master in C++, you need a **NodeHandle**. The node handle n will fully initialize the node.

The advertise() function is used to communicate with ROS and inform that you want to publish a message on a given topic name. The "/topic\_name" indicates which topic the publisher will be publishing to.

The message\_type is the type of message being published on "/topic\_name". For example, the string message data type in ROS is std\_msgs::String.

The queue\_size indicates the number of messages that can be stored in a queue. A publisher can store messages in a queue until the messages can be sent. If the number of messages stored exceeds the size of the queue, the oldest messages are dropped.

Once the publisher object pub1 has been created, as above, a message with the specified data type can be published as follows:

pub1.publish(msg);

For more information about C++ ROS publishers, see [**the documentation here**](http://docs.ros.org/jade/api/roscpp/html/classros_1_1Publisher.html).

A screenshot of a question

AI-generated content may be incorrect.

### Simple Mover

You will now go through the process of implementing your first ROS node in C++. This node is called simple\_mover. As its name implies, this node only has one responsibility, and that is to command joint movements for simple\_arm.

**Goal**

The goal of the simple\_mover node is to command each joint in the simple arm and make it swing between -pi/2 to pi/2 over time. Here’s a demonstration of this node in action:

Video 3.4

A screenshot of a computer

AI-generated content may be incorrect.

$ mkdir -p ~/catkin\_ws/src/

$ cd ~/catkin\_ws/src/

$ git clone -b first\_interaction https://github.com/udacity/RoboND-simple\_arm/ simple\_arm

A screenshot of a computer program

AI-generated content may be incorrect.

### Simple Mover: The Code

Below is the complete code for the simple\_mover C++ node, with line-by-line comments embedded. You can copy and paste this code into the simple\_mover script you created in ~/catkin\_ws/src/simple\_arm/src/ directory like this:

First, open a new terminal. Then:

$ cd ~/catkin\_ws/src/simple\_arm/src/

$ gedit simple\_mover.cpp

You have opened the C++ simple\_mover script with the **gedit** editor, now copy and paste the code below into the script and save the script. I encourage you to write this code instead of copying it so that you get more familiar with the syntax.

Video 3.5

#### simple\_mover.cpp

// simple\_mover.cpp

#include "ros/ros.h"

#include "std\_msgs/Float64.h"

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

{

    // Initialize the arm\_mover node

    ros::init(argc, argv, "arm\_mover");

    // Create a handle to the arm\_mover node

    ros::NodeHandle n;

    // Create a publisher that can publish a std\_msgs::Float64 message on the /simple\_arm/joint\_1\_position\_controller/command topic

    ros::Publisher joint1\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_1\_position\_controller/command", 10);

    // Create a publisher that can publish a std\_msgs::Float64 message on the /simple\_arm/joint\_2\_position\_controller/command topic

    ros::Publisher joint2\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_2\_position\_controller/command", 10);

    // Set loop frequency of 10Hz

    ros::Rate loop\_rate(10);

    int start\_time, elapsed;

    // Get ROS start time

    while (not start\_time) {

        start\_time = ros::Time::now().toSec();

    }

    while (ros::ok()) {

        // Get ROS elapsed time

        elapsed = ros::Time::now().toSec() - start\_time;

        // Set the arm joint angles

        std\_msgs::Float64 joint1\_angle, joint2\_angle;

        joint1\_angle.data = sin(2  \*M\_PI\*  0.1  \*elapsed)\*  (M\_PI / 2);

        joint2\_angle.data = sin(2  \*M\_PI\*  0.1  \*elapsed)\*  (M\_PI / 2);

        // Publish the arm joint angles

        joint1\_pub.publish(joint1\_angle);

        joint2\_pub.publish(joint2\_angle);

        // Sleep for the time remaining until 10 Hz is reached

        loop\_rate.sleep();

    }

    return 0;

}

**The code: Explained**

#include "ros/ros.h"

ros is the official client library for ROS. It provides most of the fundamental functionality required for interfacing with ROS via C++. It has tools for creating Nodes and interfacing with Topics, Services, and Parameters.

#include "std\_msgs/Float64.h"

From the std\_msgs package, the Float64 header file is imported. The **[std\_msgs](http://wiki.ros.org/std_msgs" \t "_blank)** package also contains the primitive message types in ROS. Later, you will be publish Float64 messages to the position command topics for each joint.

ros::init(argc, argv, "arm\_mover");

A ROS node is initialized with the init() function and registered with the ROS Master. Here arm\_mover is the name of the node. Notice that the main function takes both argc and argv arguments and passes them to the init() function.

ros::NodeHandle n;

A node handle object n is instantiated from the NodeHandle class. This node handle object will fully initialize the node and permits it to communicate with the ROS Master.

ros::Publisher joint1\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_1\_position\_controller/command", 10);

ros::Publisher joint2\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_2\_position\_controller/command", 10);

Two publishers are declared, one for joint 1 commands, and one for joint 2 commands. The node handle will tell the ROS master that a Float64 message will be published on the joint topic. The node handle also sets the queue size to 10 in the second argument of the advertise function.

ros::Rate loop\_rate(10);

A frequency of 10HZ is set using the loop\_rate object. Rates are used in ROS to limit the frequency at which certain loops cycle. Choosing a rate that is too high may result in unnecessary CPU usage, while choosing a value too low could result in high latency. Choosing sensible values for all of the nodes in a ROS system is a bit of a fine art.

start\_time = ros::Time::now().toSec();

We set start\_time to the current time. In a moment we will use this to determine how much time has elapsed. When using ROS with simulated time (as we are doing here), ros-Time-now will initially return 0, until the first message has been received on the /clock topic. This is why start\_time is set and polled continuously until a nonzero value is returned.

elapsed = ros::Time::now().toSec() - start\_time;

In the main loop, the elapsed time is evaluated by measuring the current time and subtracting the start time.

std\_msgs::Float64 joint1\_angle, joint2\_angle;

joint1\_angle.data = sin(2 \*M\_PI\* 0.1 \*elapsed)\* (M\_PI / 2);

joint2\_angle.data = sin(2 \*M\_PI\* 0.1 \*elapsed)\* (M\_PI / 2);

The joint angles are sampled from a sine wave with a period of 10 seconds, and in magnitude from [-pi/2, +pi/2].

joint1\_pub.publish(joint1\_angle);

joint2\_pub.publish(joint2\_angle);

Each trip through the body of the loop will result in two joint command messages being published.

loop\_rate.sleep();

Due to the call to loop\_rate.sleep(), the loop is traversed at approximately 10 Hertz. When the node receives the signal to shut down (either from the ROS Master, or via a signal from a console window), the loop will exit.

### Simple Mover: Build and Run

Before you can run the simple\_mover node, you have to compile the C++ script.

**Modifying CMakeLists.txt**

In order for catkin to generate the C++ libraries, you must first modify simple\_arm’s CMakeLists.txt.

CMake is the build tool underlying catkin, and CMakeLists.txt is a CMake script used by catkin. If you’re familiar with the concept of makefiles, this is similar.

Navigate to the package CMakeLists.txt file and open it:

*$ cd ~/catkin\_ws/src/simple\_arm/*

*$ gedit CMakeLists.txt*

First, ensure that the find\_package() macro lists std\_msgs, message\_generation, and controller\_manager as required packages. The find\_package() macro should look as follows:

*A close-up of a computer screen

AI-generated content may be incorrect.*

As the names might imply, the std\_msgs package contains all of the basic message types, and message\_generation is required to generate message libraries for all the supported languages (cpp, lisp, python, javascript). The contoller\_manager is another package responsible for controlling the arm.

Now, add the following block of code at the bottom of the file:

*A white background with black text

AI-generated content may be incorrect.*

These instructions ask the compiler to include the directories, executable file, link libraries, and dependencies for your C++ code:

*add\_executable(node\_name sourcecode\_directory)*

Creates the executable simple\_mover file.

*target\_link\_libraries(node\_name ${catkin\_LIBRARIES})*

This will add all the linked libraries to the compiler.

*add\_dependencies(node\_name package\_name\_generate\_messages\_cpp)*

Generates message headers for this package before you can use them.

Keep in mind that you should always include these instructions whenever you want to write a C++ ROS node. For more information about CMakeLists.txt check out [**the CMakeLists.txt page**](http://wiki.ros.org/catkin/CMakeLists.txt) on the ROS wiki.

**Building the Package**

Now that you have included specific instructions for your compiler, let’s build the package:

*$ cd ~/catkin\_ws/*

*$ catkin\_make*

**Running simple\_mover**

Assuming that your workspace has recently been built, you can launch simple\_arm as follows:

*$ cd ~/catkin\_ws/*

*$ source devel/setup.bash*

*$ roslaunch simple\_arm robot\_spawn.launch*

Once the ROS Master, Gazebo, and all of our relevant nodes are up and running, we can finally launch simple\_mover. To do so, open a new terminal and type the following commands:

*$ cd ~/catkin\_ws/*

*$ source devel/setup.bash*

*$ rosrun simple\_arm simple\_mover*

**simple\_mover GitHub branch**

You can always download a copy of this branch [**here**](https://github.com/udacity/RoboND-simple_arm/tree/simple_mover).

Congratulations! You’ve now written your first ROS node in C++!

### ROS Services

Now that you've written your first ROS node, you've seen how publishing to a topic works, and you were able to control the robotic arm by publishing to the /simple\_arm/joint\_1\_position\_controller/command topic and /simple\_arm/joint\_2\_position\_controller/command topic. Next, we'll see another node called arm\_mover, which implements the safe\_move service to allow service calls to control the arm.

**Defining services**

A ROS service allows request/response communication to exist between nodes. Within the node providing the service, request messages are handled by functions or methods. Once the requests have been handled successfully, the node providing the service sends a message back to the requester node. In C++, a ROS service server can be created using the following definition format:

*ros::ServiceServer service = n.advertiseService(`service\_name`, handler);*

In ROS, the service class name ServiceServer comes from the file name where the service definition exists. Each service provides a definition in a .srv file; this is a text file that provides the proper message type for both requests and responses.

The advertiseService() allows you to communicate with ROS through the node handle n and inform ROS that you want to create a service.

The service\_name is the name given to the service. Other nodes will use this name to specify the service to which they are sending requests.

The handler is the name of the function or method that handles the incoming service message. This function is called each time the service is called, and the message from the service call is passed to the handler function as an argument. The handler should return an appropriate service response message.

**Using Services**

**Command Line**

Services can be called directly from the command line, with:

*$ rosservice call service\_name “request”*

After calling the service, you will wait for an answer.

**ROS Service Client**

Another approach is to use a ROS service programmatically, from within a node. You will define a ROS client, which provides the interface for sending messages to the service:

*ros::ServiceClient client = n.serviceClient<package\_name::service\_file\_name>("service\_name");*

One way the ROS Client can then be used is to send requests is as follows:

*client.call(srv); // request a service*

For now, we’ll focus on how to create the ROS **service server**. Later, in the look\_away node, you will practice calling the service from a **service client** node.

See the ROS documentation [**on services**](http://wiki.ros.org/roscpp/Overview/Services) for detailed instructions on how to create and call ROS services.

A diagram of a diagram

AI-generated content may be incorrect.

A screenshot of a quiz

AI-generated content may be incorrect.

### Arm Mover

few things to get to this point.

But before we rush off, we have more ground to cover:

* Custom message generation
* Services
* Parameters
* Launch Files

In order to gain an understanding of the above, you will write another node called arm\_mover.

**Description of Arm Mover**

In many respects, arm\_mover is quite similar to simple\_mover. Like simple\_mover, it is responsible for commanding the arm to move. However, instead of simply commanding the arm to follow a predetermined trajectory, the arm\_mover node provides the service safe\_move, which allows other nodes in the system to send movement\_commands.

In addition to allowing movements via a service interface, arm\_mover also allows for configurable minimum and maximum joint angles, by using parameters.

**Creating a new service definition**

An interaction with a service consists of two messages. A node passes a request message to the service, and the service returns a response message to the node. The definitions of the request and response message types are contained within .srv files living in the srv directory under the package’s root.

Let’s define a new service for simple\_arm. We shall call it GoToPosition.

A white rectangular object with black text

AI-generated content may be incorrect.You should now edit GoToPosition.srv with gedit, so it contains the following:

A white rectangular object with a black border

AI-generated content may be incorrect.Service definitions always contain two sections, separated by a ‘---’ line. The first section is the definition of the request message. Here, a request consists of two float64 fields, one for each of simple\_arm’s joints. The second section contains the service response. The response contains only a single field, msg\_feedback. The msg\_feedback field is of type string, and is responsible for indicating that the arm has moved to a new position.

Note: Defining a custom message type is very similar. The only differences is that message definitions live within the msg directory of the package root, have a .msg extension, and do not contain the --- section divider. You can find more detailed information on creating [**messages**](http://wiki.ros.org/msg) and [**services**](http://wiki.ros.org/srv) on the ROS wiki.

**Modifying CMakeLists.txt**

As a reminder, in order for catkin to generate the C++ libraries which allow you to utilize messages in your code you must modify simple\_arm’s CMakeLists.txt file. You can find this file in ~/catkin\_ws/src/simple\_arm/.

First, uncomment the add\_service\_files() macro so it looks like this:

A screenshot of a computer program

AI-generated content may be incorrect.

**Modifying package.xml**

Now that you have updated the CMakeLists.txt file, there’s one more file which needs to be modified: package.xml.

package.xml is responsible for defining many of the package’s properties, such as the name of the package, version numbers, authors, maintainers, and dependencies.

Right now, we’ll focus on the dependencies. You already learned about build-time dependencies and run-time package dependencies. When rosdep is searching for these dependencies, it’s the package.xml file that is being parsed. So make sure that the message\_generation build dependency and the message\_runtime run dependency exist in package.xml.

A screenshot of a computer program

AI-generated content may be incorrect.

For more information about package.xml, check out the [**ROS Wiki**](http://wiki.ros.org/catkin/package.xml).

Checking Service with ROS

Now that you’ve created your GoToPosition service file, let's make sure that ROS can see it using the rossrv show command:

A screenshot of a computer program

AI-generated content may be incorrect.

This indicates that ROS can see your service.

Great job, you accomplished so much in this lesson! First you created the GoToPosition.srv file. Then, you’ve added its dependencies in CMakeLists.txt. In addition, you checked for the build and run dependencies in package.xml. Lastly, you checked if ROS can see your service file. Now, let’s move onto the code for arm\_mover.

### Arm Mover: The Code

**Creating the empty arm\_mover node script**

The steps that you should take to create the arm\_mover node are exactly the same as the steps you took to create the simple\_mover node, except the actual name of the node itself.

Open a new terminal, and type the following:

$ cd ~/catkin\_ws/src/simple\_arm/src/

$ gedit arm\_mover.cpp

You have created and opened the C++ arm\_mover source code with the **gedit** editor. Now copy and paste the code below into the source code and save the file.

Video 3.9

// arm\_mover.cpp

#include "ros/ros.h"

#include "simple\_arm/GoToPosition.h"

#include <std\_msgs/Float64.h>

// Global joint publisher variables

ros::Publisher joint1\_pub, joint2\_pub;

// This function checks and clamps the joint angles to a safe zone

std::vector<float> clamp\_at\_boundaries(float requested\_j1, float requested\_j2)

{

    // Define clamped joint angles and assign them to the requested ones

    float clamped\_j1 = requested\_j1;

    float clamped\_j2 = requested\_j2;

    // Get min and max joint parameters, and assigning them to their respective variables

    float min\_j1, max\_j1, min\_j2, max\_j2;

    // Assign a new node handle since we have no access to the main one

    ros::NodeHandle n2;

    // Get node name

    std::string node\_name = ros::this\_node::getName();

    // Get joints min and max parameters

    n2.getParam(node\_name + "/min\_joint\_1\_angle", min\_j1);

    n2.getParam(node\_name + "/max\_joint\_1\_angle", max\_j1);

    n2.getParam(node\_name + "/min\_joint\_2\_angle", min\_j2);

    n2.getParam(node\_name + "/max\_joint\_2\_angle", max\_j2);

    // Check if joint 1 falls in the safe zone, otherwise clamp it

    if (requested\_j1 < min\_j1 || requested\_j1 > max\_j1) {

        clamped\_j1 = std::min(std::max(requested\_j1, min\_j1), max\_j1);

        ROS\_WARN("j1 is out of bounds, valid range (%1.2f,%1.2f), clamping to: %1.2f", min\_j1, max\_j1, clamped\_j1);

    }

    // Check if joint 2 falls in the safe zone, otherwise clamp it

    if (requested\_j2 < min\_j2 || requested\_j2 > max\_j2) {

        clamped\_j2 = std::min(std::max(requested\_j2, min\_j2), max\_j2);

        ROS\_WARN("j2 is out of bounds, valid range (%1.2f,%1.2f), clamping to: %1.2f", min\_j2, max\_j2, clamped\_j2);

    }

    // Store clamped joint angles in a clamped\_data vector

    std::vector<float> clamped\_data = { clamped\_j1, clamped\_j2 };

    return clamped\_data;

}

// This callback function executes whenever a safe\_move service is requested

bool handle\_safe\_move\_request(simple\_arm::GoToPosition::Request& req,

    simple\_arm::GoToPosition::Response& res)

{

    ROS\_INFO("GoToPositionRequest received - j1:%1.2f, j2:%1.2f", (float)req.joint\_1, (float)req.joint\_2);

    // Check if requested joint angles are in the safe zone, otherwise clamp them

    std::vector<float> joints\_angles = clamp\_at\_boundaries(req.joint\_1, req.joint\_2);

    // Publish clamped joint angles to the arm

    std\_msgs::Float64 joint1\_angle, joint2\_angle;

    joint1\_angle.data = joints\_angles[0];

    joint2\_angle.data = joints\_angles[1];

    joint1\_pub.publish(joint1\_angle);

    joint2\_pub.publish(joint2\_angle);

    // Wait 3 seconds for arm to settle

    ros::Duration(3).sleep();

    // Return a response message

    res.msg\_feedback = "Joint angles set - j1: " + std::to\_string(joints\_angles[0]) + " , j2: " + std::to\_string(joints\_angles[1]);

    ROS\_INFO\_STREAM(res.msg\_feedback);

    return true;

}

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

{

    // Initialize the arm\_mover node and create a handle to it

    ros::init(argc, argv, "arm\_mover");

    ros::NodeHandle n;

    // Define two publishers to publish std\_msgs::Float64 messages on joints respective topics

    joint1\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_1\_position\_controller/command", 10);

    joint2\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_2\_position\_controller/command", 10);

    // Define a safe\_move service with a handle\_safe\_move\_request callback function

    ros::ServiceServer service = n.advertiseService("/arm\_mover/safe\_move", handle\_safe\_move\_request);

    ROS\_INFO("Ready to send joint commands");

    // Handle ROS communication events

    ros::spin();

    return 0;

}

**The code: Explained**

#include "ros/ros.h"

#include "simple\_arm/GoToPosition.h"

#include <std\_msgs/Float64.h>

The included modules for arm\_mover are the same as simple\_arm, with the exception of one new file. Namely, the GoToPosition.h header file, which is the header file generated from the GoToPosition.srv file we created earlier.

ros::init(argc, argv, "arm\_mover");

ros::NodeHandle n;

Inside the C++ main function, the arm\_mover node is initialized and a ROS NodeHandle object n is instantiated to communicate with ROS.

joint1\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_1\_position\_controller/command", 10);

joint2\_pub = n.advertise<std\_msgs::Float64>("/simple\_arm/joint\_2\_position\_controller/command", 10);

As we did earlier in the simple\_arm node, two publisher objects are created to publish joint angles to the arm. These objects are defined globally so as to be easily accessible from all the other functions.

ros::ServiceServer service = n.advertiseService("/arm\_mover/safe\_move", handle\_safe\_move\_request);

Next, the GoToPosition service is created with the node name followed by safe\_move. Generally, you want to name your services with the node name first to easily find them in large projects. This service is defined with a handle\_safe\_move\_request callback function. The callback function runs when a service request is received.

ros::spin();

The ros::spin() function simply blocks until a shutdown request is received by the node.

bool handle\_safe\_move\_request(simple\_arm::GoToPosition::Request& req, simple\_arm::GoToPosition::Response& res)

When a client sends a GoToPosition request to the safe\_move service, either from the terminal or from a separate node the handle\_safe\_move\_request function is called. The function parameter req is of type GoToPosition::Request. And the service response parameter res is of type GoToPosition::Response.

std::vector<float> joints\_angles = clamp\_at\_boundaries(req.joint\_1, req.joint\_2);

This function passes the requested angles to the clamp\_at\_boundaries() function.

std::vector<float> clamp\_at\_boundaries(float requested\_j1, float requested\_j2)

{

    // Define clamped joint angles and assign them to the requested ones

    float clamped\_j1 = requested\_j1;

    float clamped\_j2 = requested\_j2;

    // Get min and max joint parameters, and assign them to their respective variables

    float min\_j1, max\_j1, min\_j2, max\_j2;

    // Assign a new node handle since we have no access to the main one

    ros::NodeHandle n2;

    // Get node name

    std::string node\_name = ros::this\_node::getName();

    // Get joints min and max parameters

    n2.getParam(node\_name + "/min\_joint\_1\_angle", min\_j1);

    n2.getParam(node\_name + "/max\_joint\_1\_angle", max\_j1);

    n2.getParam(node\_name + "/min\_joint\_2\_angle", min\_j2);

    n2.getParam(node\_name + "/max\_joint\_2\_angle", max\_j2);

    // Check if joint 1 falls in the safe zone, otherwise clamp it

    if (requested\_j1 < min\_j1 || requested\_j1 > max\_j1) {

        clamped\_j1 = std::min(std::max(requested\_j1, min\_j1), max\_j1);

        ROS\_WARN("j1 is out of bounds, valid range (%1.2f,%1.2f), clamping to: %1.2f", min\_j1, max\_j1, clamped\_j1);

    }

    // Check if joint 2 falls in the safe zone, otherwise clamp it

    if (requested\_j2 < min\_j2 || requested\_j2 > max\_j2) {

        clamped\_j2 = std::min(std::max(requested\_j2, min\_j2), max\_j2);

        ROS\_WARN("j2 is out of bounds, valid range (%1.2f,%1.2f), clamping to: %1.2f", min\_j2, max\_j2, clamped\_j2);

    }

    // Store clamped joint angles in a clamped\_data vector

    std::vector<float> clamped\_data = { clamped\_j1, clamped\_j2 };

    return clamped\_data;

}

The clamp\_at\_boundaries() function is responsible for enforcing the minimum and maximum joint angles for each joint. If the joint angles passed in are outside of the operable range, they will be “clamped” to the nearest allowable value. The minimum and maximum joint angles are retrieved from the parameter server each time clamp\_at\_boundaries is called. The rest of this function simply clamps the joint angle if necessary. Warning messages are logged if the requested joint angles are out of bounds.

std\_msgs::Float64 joint1\_angle, joint2\_angle;

joint1\_angle.data = joints\_angles[0];

joint2\_angle.data = joints\_angles[1];

joint1\_pub.publish(joint1\_angle);

joint2\_pub.publish(joint2\_angle);

Then, the handle\_safe\_move\_request() function publishes the clamped joint angles to the arm.

ros::Duration(3).sleep();

The safe\_move service will be blocked for 3 seconds so the arm has enough time to move to the requested position.

res.msg\_feedback = "Joint angles set - j1: " + std::to\_string(joints\_angles[0]) + " , j2: " + std::to\_string(joints\_angles[1]);

ROS\_INFO\_STREAM(res.msg\_feedback);

Finally, the safe\_move service returns back a message indicating that the arm has moved to its new position and displays the clamped joint angles.

**Next steps**

Now that you've written the arm\_mover node, the next step is to **build** it, **launch** it, and **test** it out via the command line!

### Arm Mover: Build, Launch and Interact

**Modifying CMakeLists.txt**

Before compiling the arm\_mover.cpp code, you have to include instructions for the compiler. To do so, open the simple\_arm package CMakeLists.txt file located in ~/catkin\_ws/src/simple\_arm/, and add the following instructions at the bottom of the file:

add\_executable(arm\_mover src/arm\_mover.cpp)

target\_link\_libraries(arm\_mover ${catkin\_LIBRARIES})

add\_dependencies(arm\_mover simple\_arm\_generate\_messages\_cpp)

**Building the package**

Now that you’ve written the arm\_mover C++ script, and included specific instructions for your compiler, let’s build the package:

$ cd ~/catkin\_ws/

$ catkin\_make

**Launching the project with the new service**

To get the arm\_mover node, and accompanying safe\_move service, to launch along with all of the other nodes, modify robot\_spawn.launch.

Launch files, when they exist, are located within the launch directory in the root of a catkin package. Inside a launch file, you can instruct ROS Master which nodes to run. Also you can specify certain parameters and arguments for each of your nodes. Thus, a launch file is necessary inside a ROS package containing more than one node or a node with multiple parameters and arguments. This launch file can run all the nodes within a single command: roslaunch package\_name launch\_file.launch. simple\_arm’s launch file is located in ~/catkin\_ws/src/simple\_arm/launch

To get the arm\_mover node to launch, add the following:

  <!-- The arm mover node -->

  <node name="arm\_mover" type="arm\_mover" pkg="simple\_arm" output="screen">

    <rosparam>

      min\_joint\_1\_angle: 0

      max\_joint\_1\_angle: 1.57

      min\_joint\_2\_angle: 0

      max\_joint\_2\_angle: 1.0

    </rosparam>

  </node>

Inside the launch file, the node tag specifies the name, type, package name and output channel. The ROS parameters specify the min and max joint angles. More information on the format of the launch file can be found on the [**XML page of the ROS wiki**](http://wiki.ros.org/roslaunch/XML).

**Testing the new service**

Now that you've built your code and modified the launch file, you are ready to test it all out.

Launch the simple\_arm, verify that the arm\_mover node is running and that the safe\_move service is listed:

**Note:** You will need to make sure that you've exited your previous roslaunch session before re-launching.

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ roslaunch simple\_arm robot\_spawn.launch

Then, in a new terminal, verify that the node and service have indeed launched.

$ rosnode list

$ rosservice list

Check that both the service (/arm\_mover/safe\_move) and the node (/arm\_mover) show up as expected. If they do not appear, check the logs in the roscore console. You can now interact with the service using rosservice.

To view the camera image stream, you can use the command rqt\_image\_view (you can learn more about rqt and the associated tools on the [**RQT page of the ROS wiki**](http://wiki.ros.org/rqt)):

$ rqt\_image\_view /rgb\_camera/image\_raw

A screenshot of a computer

AI-generated content may be incorrect.

**Adjusting the view**

The camera is displaying a gray image. This is to be expected, given that it is pointing straight up, towards the gray sky of our Gazebo world.

To point the camera towards the numbered blocks on the countertop, we need to rotate both joint 1 and joint 2 by approximately pi/2 radians. Let’s give that a try:

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ rosservice call /arm\_mover/safe\_move "joint\_1: 1.57

joint\_2: 1.57"

Note: rosservice call can tab-complete the request message, so that you don’t have to worry about writing it out by hand. Also, be sure to include a line break between the two joint parameters.

Upon entering the command, you should see the arm move and eventually stop, reporting the new joint clamped angles to the console. This is as expected.

What was not expected was the resulting position of the arm. Looking at the roscore console, we can see the problem. The requested angle for joint 2 was out of the safe bounds, so it was clamped. We requested 1.57 radians, but the maximum joint angle was set to 1.0 radians.

By setting the max\_joint\_2\_angle on the parameter server, we should be able to increase joint 2’s maximum angle and bring the blocks into view the next time we request a service. To update that parameter, use the command rosparam

$ rosparam set /arm\_mover/max\_joint\_2\_angle 1.57

Now we should be able to move the arm such that all of the blocks are within the field of view of the camera:

$ rosservice call /arm\_mover/safe\_move "joint\_1: 1.57

joint\_2: 1.57"

A screenshot of a computer

AI-generated content may be incorrect.

And there you have it. All of the blocks are within the field of view!

**arm\_mover GitHub branch**

You can always download a copy of this branch [**from the GitHub repo**](https://github.com/udacity/RoboND-simple_arm/tree/arm_mover).

### ROS Clients and Subscribers

**Writing the arm\_mover node**

You have practiced generating custom messages, publishing to a topic, building ROS services servers, setting parameters, and creating launch files. You almost have a complete overview of ROS, but you still have to learn ROS **clients** to request services from client nodes, as well as ROS **subscribers**.

**ROS Clients**

A service client defined inside a service client node can request services from a service server node. In C++, ROS clients frequently have the following format, although other parameters and arguments are possible:

ros::ServiceClient client = n.serviceClient<package\_name::service\_file\_name>("service\_name");

The client object is instantiated from the ros::ServiceClient class. This object allows you to request services by calling the client.call() function.

To communicate with the ROS Master in C++, you need a **NodeHandle**. The node handle n will initialize the node.

The package\_name::service\_file\_name indicates the name of the service file located in the srv directory of the package.

The service\_name argument indicates the name of the service which is defined in the service server node.

**ROS Subscribers**

A subscriber enables your node to read messages from a topic, allowing useful data to be streamed to the node. In C++, ROS subscribers frequently have the following format, although other parameters and arguments are possible:

ros::Subscriber sub1 = n.subscribe("/topic\_name", queue\_size, callback\_function);

The sub1 object is a subscriber object instantiated from the ros::Subscriber class. This object allows you to subscribe to messages by calling the subscribe() function.

To communicate with the ROS Master in C++, you need a **NodeHandle**. The node handle n will initialize the node.

The "/topic\_name" indicates the topic to which the Subscriber should listen.

The queue\_size determines the number of messages that can be stored in a queue. If the number of messages published exceeds the size of the queue, the oldest messages are dropped. For example, if the queue\_size is set to 100 and the number of messages stored in it is equal to 100, we will have to start deleting old messages to make room in the queue for new messages. This means that we are unable to process messages fast enough, and we probably need to increase the queue\_size.

The callback\_function is the name of the function that will be run for each incoming message. Each time a message arrives, it is passed as an argument to callback\_function. Typically, this function performs a useful action with the incoming data. Note that unlike service handler functions, the callback\_function is not required to return anything.

A screenshot of a question

AI-generated content may be incorrect.

For more information about subscribers, see [**the Subscriber documentation on the ROS wiki**](http://docs.ros.org/jade/api/roscpp/html/classros_1_1Subscriber.html). Let's move on to the look\_away node so you can see **subscribers** and **clients** in action!

### Look Away

**Description of Look Away**

To see a ROS **subscriber** and **client** in action, you'll write a node called look\_away. The look\_away node will subscribe to the /rgb\_camera/image\_raw topic, which has image data from the camera mounted on the end of the robotic arm. Whenever the camera is pointed towards an uninteresting image - in this case, an image with uniform color - the callback function will request a safe\_move service to safely move the arm to something more interesting. There are a few extra pieces in the code to ensure that this procedure is executed smoothly, but we’ll focus on those later.

**Updating the launch file**

Just as you did with the arm\_mover node, to get look\_away to launch with the rest of the nodes, you will need to modify robot\_spawn.launch, which can be found in ~/catkin\_ws/src/simple\_arm/launch. You can add the following code there:

  <!-- The look away node -->

  <node name="look\_away" type="look\_away" pkg="simple\_arm"/>

Remember that a half turn of a joint requires pi/2 radians of revolution. Numerically, pi/2 is approximately 1.57. Since we want to be able to revolve a joint halfway around with one request, it will be helpful to set max\_joint\_2\_angle: 1.57 in arm\_mover:

  <!-- The arm mover node -->

  <node name="arm\_mover" type="arm\_mover" pkg="simple\_arm">

    <rosparam>

      min\_joint\_1\_angle: 0

      max\_joint\_1\_angle: 1.57

      min\_joint\_2\_angle: 0

      max\_joint\_2\_angle: 1.57

    </rosparam>

  </node>

### Look Away: The Code

**Creating the empty look\_away node script**

The steps that you should take to create the look\_away node are exactly the same as the steps you took to create the simple\_mover and arm\_mover scripts, but of course change the actual name of the file itself.

Open a new terminal, and type the following:

$ cd ~/catkin\_ws/src/simple\_arm/src/

$ gedit look\_away.cpp

You have created and opened the C++ look\_away file with the **gedit** editor. Now copy and paste the code below and save the file.

Video 3.13

// look\_away.cpp

#include "ros/ros.h"

#include "simple\_arm/GoToPosition.h"

#include <sensor\_msgs/JointState.h>

#include <sensor\_msgs/Image.h>

// Define global vector of joints last position, moving state of the arm, and the client that can request services

std::vector<double> joints\_last\_position{ 0, 0 };

bool moving\_state = false;

ros::ServiceClient client;

// This function calls the safe\_move service to safely move the arm to the center position

void move\_arm\_center()

{

    ROS\_INFO\_STREAM("Moving the arm to the center");

    // Request centered joint angles [1.57, 1.57]

    simple\_arm::GoToPosition srv;

    srv.request.joint\_1 = 1.57;

    srv.request.joint\_2 = 1.57;

    // Call the safe\_move service and pass the requested joint angles

    if (!client.call(srv))

        ROS\_ERROR("Failed to call service safe\_move");

}

// This callback function continuously executes and reads the arm joint angles position

void joint\_states\_callback(const sensor\_msgs::JointState js)

{

    // Get joints current position

    std::vector<double> joints\_current\_position = js.position;

    // Define a tolerance threshold to compare double values

    double tolerance = 0.0005;

    // Check if the arm is moving by comparing its current joints position to its latest

    if (fabs(joints\_current\_position[0] - joints\_last\_position[0]) < tolerance && fabs(joints\_current\_position[1] - joints\_last\_position[1]) < tolerance)

        moving\_state = false;

    else {

        moving\_state = true;

        joints\_last\_position = joints\_current\_position;

    }

}

// This callback function continuously executes and reads the image data

void look\_away\_callback(const sensor\_msgs::Image img)

{

    bool uniform\_image = true;

    // Loop through each pixel in the image and check if its equal to the first one

    for (int i = 0; i < img.height \* img.step; i++) {

        if (img.data[i] - img.data[0] != 0) {

            uniform\_image = false;

            break;

        }

    }

    // If the image is uniform and the arm is not moving, move the arm to the center

    if (uniform\_image == true && moving\_state == false)

        move\_arm\_center();

}

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

{

    // Initialize the look\_away node and create a handle to it

    ros::init(argc, argv, "look\_away");

    ros::NodeHandle n;

    // Define a client service capable of requesting services from safe\_move

    client = n.serviceClient<simple\_arm::GoToPosition>("/arm\_mover/safe\_move");

    // Subscribe to /simple\_arm/joint\_states topic to read the arm joints position inside the joint\_states\_callback function

    ros::Subscriber sub1 = n.subscribe("/simple\_arm/joint\_states", 10, joint\_states\_callback);

    // Subscribe to rgb\_camera/image\_raw topic to read the image data inside the look\_away\_callback function

    ros::Subscriber sub2 = n.subscribe("rgb\_camera/image\_raw", 10, look\_away\_callback);

    // Handle ROS communication events

    ros::spin();

    return 0;

}

**The code: Explained**

#include "ros/ros.h"

#include "simple\_arm/GoToPosition.h"

#include <sensor\_msgs/JointState.h>

#include <sensor\_msgs/Image.h>

The header files are similar to those in arm\_mover, except this time we included the JointState.h header file so that we can read the arm joints’ positions. We also include the Image.h header file so that we can use the camera data.

ros::init(argc, argv, "look\_away");

ros::NodeHandle n;

Inside the C++ main function, the look\_away node is initialized and a ROS NodeHandle object n is instantiated to communicate with ROS.

client = n.serviceClient<simple\_arm::GoToPosition>("/arm\_mover/safe\_move");

A client object is created here. This object can request GoToPosition services from the /arm\_mover/safe\_move service created earlier in the arm\_mover node. This client object is defined globally in the code, so we can request services within any function. In particular, this happens in the move\_arm\_center() function.

ros::Subscriber sub1 = n.subscribe("/simple\_arm/joint\_states", 10, joint\_states\_callback);

The first subscriber object sub1, subscribes to the /simple\_arm/joint\_states topic. By subscribing to this topic, we can track the arm position by reading the angle of each joint. The queue\_size is set to 10, meaning that a maximum of 10 messages can be stored in the queue. The data from each new incoming message is passed to the joint\_states\_callback function.

ros::Subscriber sub2 = n.subscribe("rgb\_camera/image\_raw", 10, look\_away\_callback);

The second subscriber object sub2, subscribes to the /rgb\_camera/image\_raw topic. The queue\_size is also set to 10. And the look\_away\_callback function is called each time a new message arrives.

ros::spin();

The ros::spin() function simply blocks until a shutdown request is received by the node.

void joint\_states\_callback(const sensor\_msgs::JointState js)

{

    // Get joints current position

    std::vector<double> joints\_current\_position = js.position;

    // Define a tolerance threshold to compare double values

    double tolerance = 0.0005;

    // Check if the arm is moving by comparing its current joints position to its latest

    if (fabs(joints\_current\_position[0] - joints\_last\_position[0]) < tolerance && fabs(joints\_current\_position[1] - joints\_last\_position[1]) < tolerance)

        moving\_state = false;

    else {

        moving\_state = true;

        joints\_last\_position = joints\_current\_position;

    }

}

When sub1 receives a message on the/simple\_arm/joint\_states topic, the message is passed to the joint\_states\_callback in the variable js. The joint\_states\_callback() function checks if the current joint states provided in js are the same as the previous joint states, which are stored in the global joints\_last\_position variable. If the current and previous joint states are the same (up to the specified error tolerance), then the arm has stopped moving, and the moving\_state flag is set to False. This flag is defined globally so as to be shared with other functions in the code. On the other hand, if the current and previous joint states are different, then the arm is still moving. In this case, the function sets moving\_state to true and updates the joints\_Last\_position variable with current position data stored in joints\_current\_position.

void look\_away\_callback(const sensor\_msgs::Image img)

{

    bool uniform\_image = true;

    // Loop through each pixel in the image and check if its equal to the first one

    for (int i = 0; i < img.height \* img.step; i++) {

        if (img.data[i] - img.data[0] != 0) {

            uniform\_image = false;

            break;

        }

    }

    // If the image is uniform and the arm is not moving, move the arm to the center

    if (uniform\_image == true && moving\_state == false)

        move\_arm\_center();

}

The look\_away\_callback() function receives [**image data**](http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Image.html) from the /rgb\_camera/image\_raw topic. The callback function first checks if all color values in the image are the same as the color value of the first pixel. Then, if the image is uniform and the arm is not moving, the move\_arm\_center() function is called.

void move\_arm\_center()

{

    ROS\_INFO\_STREAM("Moving the arm to the center");

    // Request centered joint angles [1.57, 1.57]

    simple\_arm::GoToPosition srv;

    srv.request.joint\_1 = 1.57;

    srv.request.joint\_2 = 1.57;

    // Call the safe\_move service and pass the requested joint angles

    if (!client.call(srv))

        ROS\_ERROR("Failed to call service safe\_move");

}

Inside the move\_arm\_center function, a GoToPosition request message is created and sent using the arm\_mover/safe\_move service, moving both joint angles to 1.57 radians.

### Look Away: Build, Launch and Interact

**Modifying CMakeLists.txt**

Before compiling the look\_away.cpp code, you have to include instructions for the compiler. As a reminder, for every C++ ROS node you write, you have to add its dependencies in CMakeLists.txt file. Open the simple\_arm package’s CMakeLists.txt file, located in ~/catkin\_ws/src/simple\_arm/, and add the following instructions at the bottom of the file:

add\_executable(look\_away src/look\_away.cpp)

target\_link\_libraries(look\_away ${catkin\_LIBRARIES})

add\_dependencies(look\_away simple\_arm\_generate\_messages\_cpp)

**Building the package**

Now that you’ve written the look\_away C++ script, and included specific instructions for your compiler, let’s build the package:

$ cd ~/catkin\_ws/

$ catkin\_make

**Launching the nodes**

You can now launch and interact with simple\_arm just as before:

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ roslaunch simple\_arm robot\_spawn.launch

**Interacting with the arm**

After launching, the arm should move away from the grey sky and look towards the blocks. To view the camera image stream, you can use the same command as before:

$ rqt\_image\_view /rgb\_camera/image\_raw

To check that everything is working as expected, open a new terminal and send a service call to point the arm directly up towards the sky (note that the line break in the message is necessary):

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ rosservice call /arm\_mover/safe\_move "joint\_1: 0

joint\_2: 0"

**look\_away GitHub branch**

You can always download a copy of this lab that includes all three nodes by visiting [**the GitHub repo**](https://github.com/udacity/RoboND-simple_arm/).

What happens?

A screenshot of a question

AI-generated content may be incorrect.

### Pub-Sub Class

Inside the publisher and subscriber nodes of this lesson, global variables and objects were defined to be used anywhere in the code. We did this to simplify the code, but it is not a good practice. You should always write a pub-sub class to easily share variables and objects with any callback function in your code. Here’s a [**ROS pub-sub template class**](https://answers.ros.org/question/59725/publishing-to-a-topic-via-subscriber-callback-function/) that you can use:

We challenge you to use this template class to implement the nodes in this lesson.

**ROS Class C++ Code**

// ROS Class C++ Code

#include <ros/ros.h>

class SubscribeAndPublish

{

public:

  SubscribeAndPublish()

  {

    //Topic you want to publish

    pub\_ = n\_.advertise<PUBLISHED\_MESSAGE\_TYPE>("/published\_topic", 1);

    //Topic you want to subscribe

    sub\_ = n\_.subscribe("/subscribed\_topic", 1, &SubscribeAndPublish::callback, this);

  }

  void callback(const SUBSCRIBED\_MESSAGE\_TYPE& input)

  {

    PUBLISHED\_MESSAGE\_TYPE output;

    //.... do something with the input and generate the output...

    pub\_.publish(output);

  }

private:

  ros::NodeHandle n\_;

  ros::Publisher pub\_;

  ros::Subscriber sub\_;

};//End of class SubscribeAndPublish

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

{

  //Initiate ROS

  ros::init(argc, argv, "subscribe\_and\_publish");

  //Create an object of class SubscribeAndPublish that will take care of everything

  SubscribeAndPublish SAPObject;

  ros::spin();

  return 0;

}

### Recap

Video 3.16

**Primary Resources**

* [**ROS Wiki**](http://wiki.ros.org/)
* [**ROS Answers**](http://answers.ros.org/)

**Additional Resources**

* [**ROS Cheat Sheet**](https://github.com/ros/cheatsheet/releases/download/0.0.1/ROScheatsheet_catkin.pdf): This is the official ROS cheat sheet. Even though the title indicates that it is for the indigo distribution, almost all of the commands still work in Kinetic, the distribution that we are using for this program. Download this Cheat Sheet and keep it next to you while working with ROS so that you don’t have to remember all the commands.
* [**A gentle Introduction to ROS**](https://cse.sc.edu/~jokane/agitr/): This is a great book, and is available for free as a PDF. A print version is also for sale.

## Project: Go Chase It!

### Introduction

Video 4.1

**Project Preview**

Here’s a preview of the final outcome of this project. Note that your world and the robot will look different than mine as you will be implementing you own world and robot.

Video 4.1.1

### ROS in the VM Workspace

**ROS in the Workspace**

* To follow along with the project instructions, you will need to run ROS.
* Open **Udacity's VM workspace** (Ubuntu 20.04 LTS) or the **local VM image** (Ubuntu 18.04 LTS) running on your VMWare/VirtualBox to practice the current exercise.
* Once you log into the VM, open a Terminal window.
* You're now ready to follow along in your development environment with this project!

### Setting up my\_robot

The first task in this project is to create the my\_robot ROS package. Inside my\_robot, you will store and launch an empty Gazebo world file. As you proceed with the project, you will model and store a robot, as well as replace the empty world file with the world you created in the **Build My World** project. For now, follow these steps to set up my\_robot.

*Note: Do not have more than one my\_robot instance in the Gazebo world otherwise it would not be able to launch.*

**Create the my\_robot Package**

A screenshot of a chat

AI-generated content may be incorrect.

**Create and Store an Empty Gazebo World File**

Inside the worlds directory, create and store an empty Gazebo world file. As a reminder, in Gazebo a **world** is a collection of models, such as your robot, together with a specific environment. You can also define several other physical properties specific to this world.

**1- Create an empty Gazebo world**

An empty world in Gazebo is a simple world, with no objects or models.

$ cd ~/catkin\_ws/src/my\_robot/worlds/

$ touch empty.world

**2- Add the following to** **empty.world**

<?xml version="1.0" ?>

<sdf version="1.4">

  <world name="default">

    <include>

      <uri>model://ground\_plane</uri>

    </include>

    <!-- Light source -->

    <include>

      <uri>model://sun</uri>

    </include>

    <!-- World camera -->

    <gui fullscreen='0'>

      <camera name='world\_camera'>

        <pose>4.927360 -4.376610 3.740080 0.000000 0.275643 2.356190</pose>

        <view\_controller>orbit</view\_controller>

      </camera>

    </gui>

  </world>

</sdf>

The .world file uses the XML file format to describe all the elements with respect to the Gazebo environment. The simple world that you are creating here has the following elements:

* <sdf>: The base element which encapsulates the entire file structure and content.
* <world>: The world element defines the world description and several properties pertaining to that world. In this example, you are adding a ground plane, a light source, and a camera to your world. Each model or property can have further elements that add detail. For example, the camera has a pose element which defines its position and orientation.
* <include>: The include element, along with the <uri> element, provide a path to a particular model. In Gazebo there are several models that are available by default.
* Create a Launch File

Launch files in ROS allow us to execute more than one node simultaneously, which helps avoid the potentially tedious task of defining and launching several nodes in separate shells or terminals.

**1- Create the** **world.launch** **file**

$ **cd** ~/catkin\_ws/src/my\_robot/launch/

$ **touch** world.launch

**2- Add the following to** **world.launch**

<?xml version="1.0" encoding="UTF-8"?>

<launch>

  <!-- World File -->

  <arg name="world\_file" default="$(find my\_robot)/worlds/empty.world"/>

  <!-- Launch Gazebo World -->

  <include file="$(find gazebo\_ros)/launch/empty\_world.launch">

    <arg name="use\_sim\_time" value="true"/>

    <arg name="debug" value="false"/>

    <arg name="gui" value="true" />

    <arg name="world\_name" value="$(arg world\_file)"/>

  </include>

</launch>

As in the case of the .world file, the .launch files are also based on XML. The structure for the launch files has two parts -

* First, define arguments using the <arg> element. Each such element will have a name attribute and a default value.
* Second, include the world.launch file from the gazebo\_ros package. The **[empty\_world](https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_ros/launch/empty_world.launch" \t "_blank)** file includes a set of important definitions that are inherited by the world that we create. Using the world\_name argument and the path to your .world file passed as the value to that argument, you will be able to launch your world in Gazebo.

**Launch empty.world**

A white rectangular object with black text

AI-generated content may be incorrect.

Empty Gazebo world with a sun shining from the top!
Empty Gazebo world with a sun shining from the top!

It does look a bit bland, but don't worry, there will soon be a different world for your robot to explore!

A screenshot of a computer program

AI-generated content may be incorrect.

### Understanding URDF

**Understanding Unified Robot Description Format (URDF)**

In the Build My World project, you used the **Model Editor** tool in Gazebo to model a robot with the Simulation Description Format, or **SDF**. Now that you are working with ROS, you have to model a robot with the Unified Robot Description Format, or **URDF**. Both of these formats use [**XML**](https://www.w3schools.com/xml/xml_whatis.asp) markup language. We can use a URDF file to define a robot model, its kinodynamic properties, visual elements and even model sensors for the robot. URDF can only describe a robot with rigid links connected by joints in a chain or tree structure. It cannot describe a robot with flexible or parallel links.

A simple robot with two links and a joint can be described using URDF as follows:

<?xml version="1.0"?>

<robot name="two\_link\_robot">

  <!--Links-->

  <link name="link\_1">

    <visual>

      <geometry>

        <cylinder length="0.5" radius="0.2"/>

      </geometry>

    </visual>

  </link>

  <link name="link\_2">

    <visual>

      <geometry>

        <box size="0.6 0.1 0.2"/>

      </geometry>

    </visual>

  </link>

  <!--Joints-->

  <joint name="joint\_1" type="continuous">

    <parent link="link\_1"/>

    <child link="link\_2"/>

  </joint>

</robot>

Since we use URDF files to describe several robot and environmental properties, the files tend to be long and tedious. This is why we use Xacro (XML Macros) to divide our single URDF file into multiple Xacro files. While the syntax remains the same, we can now divide our robot description into smaller subsystems.

Since URDF (and Xacro) files are basically XML, they use tags to define robot geometry and properties. The most important and commonly used tags with their elements are described below:

<robot> </robot>

This is a top level tag that contains all the other tags related to a given robot.

<link> </link>

Each rigid link in a robot must have this tag associated with it.

**Attributes**

**name**: Requires a unique link name attribute.

A screenshot of a computer

AI-generated content may be incorrect.

  <link name="link\_1">

    <inertial>

      <origin xyz="0 0 0.4" rpy="0 0 0"/>

      <mass value="${mass1}"/>

      <inertia ixx="30" ixy="0" ixz="0" iyy="50" iyz="0" izz="50"/>

    </inertial>

    <visual>

      <origin xyz="0 0 0" rpy="0 0 0"/>

      <geometry>

        <mesh filename="package://kuka\_arm/meshes/kr210l150/visual/link\_1.dae"/>

      </geometry>

      <material name="">

        <color rgba="0.75294 0.75294 0.75294 1"/>

      </material>

    </visual>

    <collision>

      <origin xyz="0 0 0" rpy="0 0 0"/>

      <geometry>

        <mesh filename="package://kuka\_arm/meshes/kr210l150/collision/link\_1.stl"/>

      </geometry>

    </collision>

  </link>

The <link> tag has many more optional elements that can be used to define other properties like color, material, texture, etc. Refer [**this link**](http://wiki.ros.org/urdf/XML/link) for details on those tags.

<joint> </joint>

This tag typically defines a single joint between two links in a robot. The type of joints you can define using this tag include:

A screenshot of a web page

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Other optional elements under the <joint> tag can be [**found here**](http://wiki.ros.org/urdf/XML/joint).

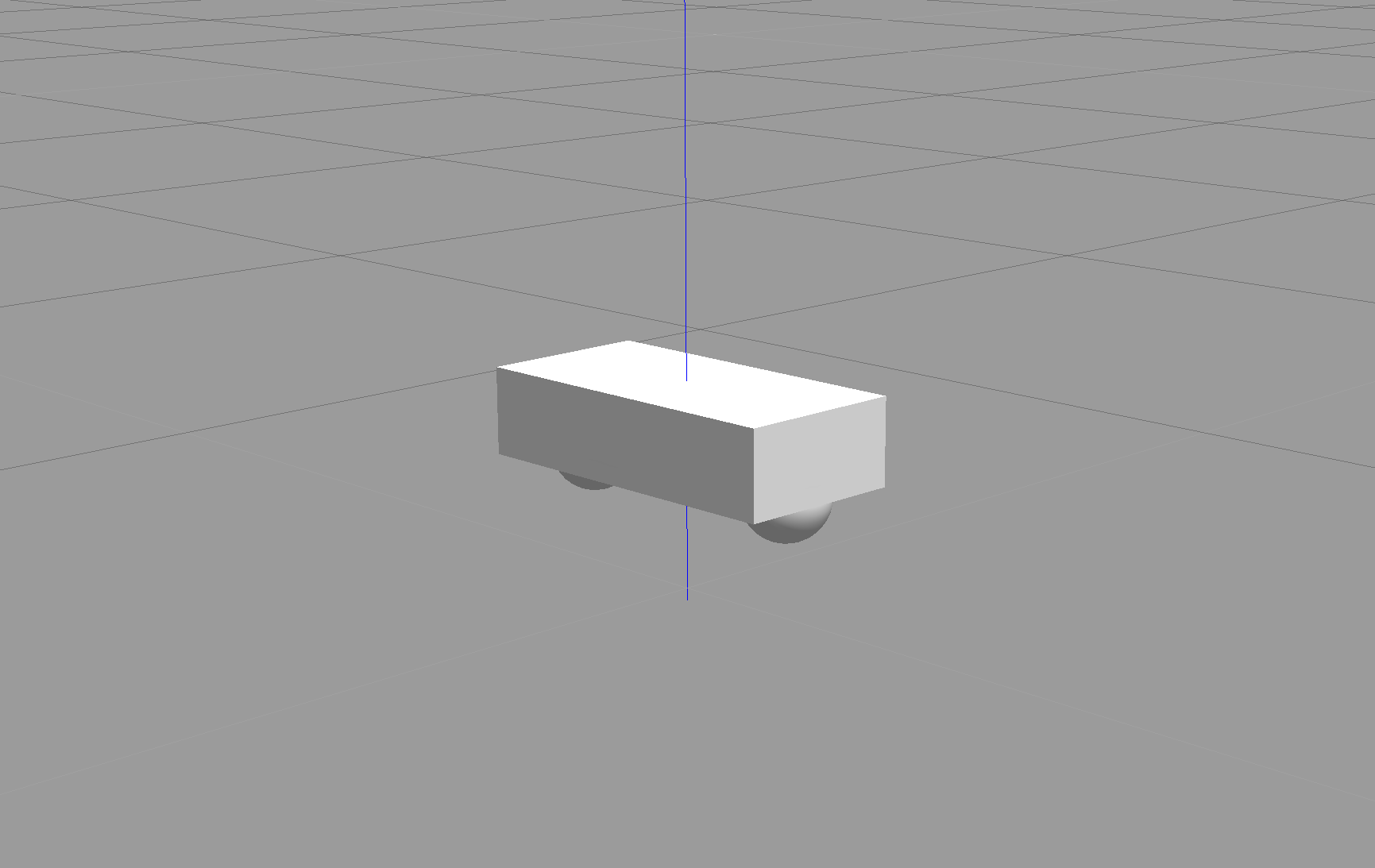
There are many more optional tags and attributes that help to define various dynamic and kinematic properties of a robot, along with sensors and actuators. For a full list, refer to the [**ROS documentation on URDF**](http://wiki.ros.org/urdf).

### Robot Basic Setup

Let’s build a basic mobile robot model by creating a URDF file and launch it inside an empty Gazebo world.

We can break the effort down into smaller components - a robot base, wheels, and sensors.

For this model, we will create a cuboidal base with two caster wheels. The caster wheels will help stabilize this model. They aren't always required, but they can help with weight distribution, preventing the robot from tilting along the z-axis.



Robot base with two castor wheels

A screenshot of a computer

AI-generated content may be incorrect.

<?xml version='1.0'?>

<robot name="my\_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

                                                  <link name="robot\_footprint"></link>

                                                  <joint name="robot\_footprint\_joint" type="fixed">

                                                    <origin xyz="0 0 0" rpy="0 0 0" />

                                                    <parent link="robot\_footprint"/>

                                                    <child link="chassis" />

                                                  </joint>

                                                  <link name='chassis'>

                                                    <pose>0 0 0.1 0 0 0</pose>

                                                    <inertial>

                                                      <mass value="15.0"/>

                                                      <origin xyz="0.0 0 0" rpy=" 0 0 0"/>

                                                      <inertia

                                                          ixx="0.1" ixy="0" ixz="0"

                                                          iyy="0.1" iyz="0"

                                                          izz="0.1"

                                                      />

                                                    </inertial>

                                                    <collision name='collision'>

                                                      <origin xyz="0 0 0" rpy=" 0 0 0"/>

                                                      <geometry>

                                                        <box size=".4 .2 .1"/>

                                                      </geometry>

                                                    </collision>

                                                    <visual name='chassis\_visual'>

                                                      <origin xyz="0 0 0" rpy=" 0 0 0"/>

                                                      <geometry>

                                                        <box size=".4 .2 .1"/>

                                                      </geometry>

                                                    </visual>

                                                    <collision name='back\_caster\_collision'>

                                                      <origin xyz="-0.15 0 -0.05" rpy=" 0 0 0"/>

                                                      <geometry>

                                                        <sphere radius="0.0499"/>

                                                      </geometry>

                                                    </collision>

                                                    <visual name='back\_caster\_visual'>

                                                      <origin xyz="-0.15 0 -0.05" rpy=" 0 0 0"/>

                                                      <geometry>

                                                        <sphere radius="0.05"/>

                                                      </geometry>

                                                    </visual>

                                                    <collision name='front\_caster\_collision'>

                                                      <origin xyz="0.15 0 -0.05" rpy=" 0 0 0"/>

                                                      <geometry>

                                                        <sphere radius="0.0499"/>

                                                      </geometry>

                                                    </collision>

                                                    <visual name='front\_caster\_visual'>

                                                      <origin xyz="0.15 0 -0.05" rpy=" 0 0 0"/>

                                                      <geometry>

                                                        <sphere radius="0.05"/>

                                                      </geometry>

                                                    </visual>

                                                  </link>

</robot>

We have a single link, with the name defined as "chassis", encompassing the base as well as the caster wheels. Every link has specific elements, such as the inertial or the collision elements. You can quickly review the details of these elements covered in the previous section. The chassis is a cube, whereas the casters are spherical, as denoted by their <geometry> tags. Each link (or joint) has an origin (or pose), as well. Every element of that link or joint will have its own origin, which will be relative to the link's frame of reference.

For this base, the casters are included as part of the link for stability. There is no need for any additional links to define the casters, and therefore no joints to connect them. The casters do, however, have friction coefficients defined for them. These friction coefficients are set to 0, to allow for free motion while moving.

**Launch the Robot**

Now that you’ve built the basic robot model, let’s create a launch file to load it inside an empty Gazebo world.

**1- Create a new launch file to load the** **URDF** **model file**

$ cd ~/catkin\_ws/src/my\_robot/launch/

$ touch robot\_description.launch

**2- Copy the following code into** **robot\_description.launch** **file**

<?xml version="1.0"?>

<launch>

  <!-- send urdf to param server -->

  <param name="robot\_description" command="$(find xacro)/xacro --inorder '$(find my\_robot)/urdf/my\_robot.xacro'" />

</launch>

To generate the URDF file from the Xacro file, you must first define a parameter, robot\_description. This parameter will set a single command to use the **[xacro package](http://wiki.ros.org/urdf/Tutorials/Using%20Xacro%20to%20Clean%20Up%20a%20URDF%20File" \t "_blank)** to generate the URDF from the xacro file.

**3- Update the world.launch file created earlier so that Gazebo can load the robot URDF model**

Add the following to the launch file (after <launch>):

                 <!-- Robot pose -->

                 <arg name="x" default="0"/>

                 <arg name="y" default="0"/>

                 <arg name="z" default="0"/>

                 <arg name="roll" default="0"/>

                 <arg name="pitch" default="0"/>

                 <arg name="yaw" default="0"/>

                 <!-- Launch other relevant files-->

                 <include file="$(find my\_robot)/launch/robot\_description.launch"/>                                                                          <!-- Robot pose -->

**Note: If you have copied your gazebo world from Project 1 then you could skip this step, since you already have** **my\_robot** **in your Gazebo world.**

Add the following to the launch file (before </launch>):

    <!-- Find my robot Description-->

             <param name="robot\_description" command="$(find xacro)/xacro --inorder '$(find my\_robot)/urdf/my\_robot.xacro'"/>

             <!-- Spawn My Robot -->

             <node name="urdf\_spawner" pkg="gazebo\_ros" type="spawn\_model" respawn="false" output="screen"

                   args="-urdf -param robot\_description -model my\_robot

                         -x $(arg x) -y $(arg y) -z $(arg z)

                         -R $(arg roll) -P $(arg pitch) -Y $(arg yaw)"/>

The **[gazebo\_ros package](http://wiki.ros.org/gazebo_ros" \t "_blank)** spawns the model from the URDF that robot\_description helps generate.

A screenshot of a computer program

AI-generated content may be incorrect.

### Robot Enhancements

**Robot Enhancement**

Now that you’ve built a basic model of your robot, enhance it and add wheels. Each wheel is represented as a link and is connected to the base link (the chassis) with a joint.

A white object with wheels

AI-generated content may be incorrect.

**Create Wheel Links**

You will first create the links for each wheel using the specifications given below and add that to your Xacro file. For each wheel, you will have a collision, inertial, and visual element, along with the following properties:

* link name - "SIDE\_wheel", where the SIDE is either left or right.
* geometry - "cylinder" with radius 0.1 and length 0.05.
* origin for each element - [0, 0, 0, 0, 1.5707, 1.5707]
* mass of each wheel - "5".
* You can use the same inertia values as the ones for the chassis for simplicity: ixx="0.1" ixy="0" ixz="0" iyy="0.1" iyz="0" izz="0.1"
* Create Joints for the two wheels

Once define the links, you need to create the corresponding joints. The following elements will create a joint between your left wheel (the child link) and the robot chassis (the parent link):

  <joint type="continuous" name="left\_wheel\_hinge">

    <origin xyz="0 0.15 0" rpy="0 0 0"/>

    <child link="left\_wheel"/>

    <parent link="chassis"/>

    <axis xyz="0 1 0" rpy="0 0 0"/>

    <limit effort="10000" velocity="1000"/>

    <dynamics damping="1.0" friction="1.0"/>

  </joint>

The joint type is set to "continuous" and is similar to a revolute joint but has no limits on its rotation. This means that the joint can rotate continuously. The joint will have its own axis of rotation. Also, the joint will have certain limits to enforce the maximum "effort" and "velocity" for that joint. The limits are useful constraints in for a real robot and can help in simulation as well. ROS has [**good documentation on safety limits**](http://wiki.ros.org/pr2_controller_manager/safety_limits). In addition, the joint will have specific joint dynamics that correspond to the physical properties of the joint like "damping" and “friction”.

Add the left wheel joint to your Xacro file. Then use it as a template to create the joint between the right wheel and the chassis.

A screenshot of a question

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### Robot Sensors

Earlier, you built a very basic robot model by creating its own URDF file. Then, you enhanced the model and added a wheel and a joint on each side of the chassis. Now, it’s time to add sensors to our robot so it can perceive its surroundings. You’ll add two sensors - a **camera** and a **lidar**.

**Sensors**

**Camera:** Cameras are one of the most common sensors in Robotics. They capture information that is easily interpreted by humans at a high resolution compared to other sensors. Every image captures millions of pixels. To extract depth information from camera images, people have started using stereo cameras. These work like your eyes do and are able to estimate distances to objects.

**Lidar:** Lidar stands for Light Detection and Ranging. It uses arrays of lasers to sense “point cloud” models of the environment. By measuring thousands of millions of times per second, lidar builds an accurate model of the world. However, the resolution is not nearly as high as that of a camera.

A white object with a black hole

AI-generated content may be incorrect.

**Add a Camera**

First, add the camera link and a corresponding joint. Open the my\_robot.xacro file and add a camera sensor based on the following specifications:

* link name - "camera"
* link origin - "[0, 0, 0, 0, 0, 0]"
* geometry - box with size "[0.05, 0.05, 0.05]"
* mass - "0.1"
* inertia - ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6"
* joint name - "camera\_joint"
* joint type - "fixed"
* joint axis - "[0, 1, 0]"
* joint origin - "[0.2, 0, 0, 0, 0, 0]"
* joint parent link - "chassis"
* joint child link - "camera"

As we covered in the previous section, each link should have its own visual, collision and inertial elements.

**Add a Lidar**

Now, let's add the lidar sensor. ROS supports many different types of [**sensors**](http://wiki.ros.org/Sensors#A2D_range_finders). One of them, that you will use for this robot and for the project, is the Hokuyo rangefinder sensor.

A black square object with a round black cap

AI-generated content may be incorrect.

The Hokuyo sensor can be added to your robot model just like the camera sensor, except that you first need to add a mesh file to your robot model. Mesh files define the shape of the object or model you are working with. There are some basic shapes, like the box or cylinder, that do not require a mesh file. However, for more advanced designs, mesh files are necessary. The mesh file should be located in a directory called meshes that you can create in your package folder, my\_robot.

**1- Create** **meshes** **directory**

Let’s create a meshes directory in my\_robot to hold sensor mesh files:

$ cd ~/catkin\_ws/src/my\_robot/

$ mkdir meshes

**2- Now, download this** **[hokuyo.dae](https://s3-us-west-1.amazonaws.com/udacity-robotics/hokuyo.dae" \t "_blank)** **file and place it under the** **meshes** **directory you just created.**

Wondering where I got the mesh file for the Hokuyo sensor? Gazebo shares the mesh files for its [**entire library of models**](http://models.gazebosim.org/).

**3- Add the Hokuyo sensor to** **my\_robot.xacro**

Here are the Hokuyo lidar sensor specifications:

* link name - "hokuyo"
* link origin - "[0, 0, 0, 0, 0, 0]"
* geometry for <collision> - box with size "[0.1, 0.1, 0.1]"
* geometry for <visual> - filename = “package://my\_robot/meshes/hokuyo.dae”
* mass - "1e-5"
* inertia - ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6"
* joint name - "hokuyo\_joint"
* joint type - "fixed"
* joint axis - "[0 1 0]"
* joint origin - "[0.15, 0, .1, 0, 0, 0]"
* joint parent link - "chassis"
* joint child link - "hokuyo"

As we covered in the previous section, each link should have its own visual, collision and inertial elements.

Remember to use the <mesh> tag inside the <geometry> tag to define the mesh file you want to use as a visual for the sensor link. You can find an example of how to do it [**here**](https://classic.gazebosim.org/tutorials?tut=attach_meshes&cat=build_robot).

**Launch**

Excellent work! You created a robot model and added sensors to it. Now you can test your updated model in Gazebo:

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ roslaunch my\_robot world.launch

Now, let's see what your model looks like!

A blue and green object with green wheels

AI-generated content may be incorrect.

Wait, something's definitely wrong here.

A screenshot of a survey

AI-generated content may be incorrect.

### Gazebo Plugins

You added sensors to your robot, allowing it to visualize the world around it! But how exactly does the **camera** sensor takes those images in simulation? How does a **lidar** sensor take laser measurements in simulation? How exactly does your robot **move** in a simulated environment?

URDF in itself can't help with that. However, Gazebo allows for plugins that implement specific use-cases.

**Sensor and Actuators Plugins**

We will cover the use of three such plugins:

* A plugin for the **camera** sensor.
* A plugin for the **Hokuyo lidar** sensor.
* A plugin for the **wheel joints** actuator.

**Add Plugins**

Download the **[my\_robot.gazebo](https://s3-us-west-1.amazonaws.com/udacity-robotics/my_robot.gazebo" \t "_blank)** file, which includes the 3 plugins mentioned above, and place it inside the urdf directory of my\_robot.

**Gazebo Plugin Files**

Since we have a two-wheeled mobile robot, we will use a plugin that implements a differential drive controller. Let's take a look at how the plugin is defined in the my\_robot.gazebo file.

<gazebo>

  <plugin name="differential\_drive\_controller" filename="libgazebo\_ros\_diff\_drive.so">

    <legacyMode>false</legacyMode>

    <alwaysOn>true</alwaysOn>

    <updateRate>10</updateRate>

    <leftJoint>left\_wheel\_hinge</leftJoint>

    <rightJoint>right\_wheel\_hinge</rightJoint>

    <wheelSeparation>0.4</wheelSeparation>

    <wheelDiameter>0.2</wheelDiameter>

    <torque>10</torque>

    <commandTopic>cmd\_vel</commandTopic>

    <odometryTopic>odom</odometryTopic>

    <odometryFrame>odom</odometryFrame>

    <robotBaseFrame>robot\_footprint</robotBaseFrame>

    <publishWheelTF>false</publishWheelTF>

    <publishWheelJointState>false</publishWheelJointState>

    <rosDebugLevel>na</rosDebugLevel>

    <wheelAcceleration>0</wheelAcceleration>

    <wheelTorque>5</wheelTorque>

    <odometrySource>world</odometrySource>

    <publishTf>1</publishTf>

    <publishOdomTF>true</publishOdomTF>

  </plugin>

</gazebo>

libgazebo\_ros\_diff\_drive.so is the shared object file created from compiling the C++ source code. The plugin accepts information specific to your robot's model, such as wheel separation, joint names, and more. Then it calculates and publishes the robot's odometry information to the topics that you specified, like odom. In an upcoming section, you will send velocity commands to your robot to move it in a specific direction. This controller helps achieve that result.

If you'd like to understand how this plugin was created, you can refer to its C++ [**source code**](https://github.com/ros-simulation/gazebo_ros_pkgs/blob/noetic-devel/gazebo_plugins/src/gazebo_ros_diff_drive.cpp).

Gazebo already has several such plugins publicly available. We will utilize the preexisting plugins for the [**camera sensor**](https://classic.gazebosim.org/tutorials?tut=ros_gzplugins#Camera) and the preexisting plugins for the Hokuyo lidar sensor. Both of these are already included in the my\_robot.gazebo file linked previously.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

### RViz Basics

A robot standing next to a sign

AI-generated content may be incorrect.

**RViz**

RViz stands for ROS Visualization. RViz is our one-stop tool to visualize all three core aspects of a robot: perception, decision-making, and actuation.

While Gazebo is a physics simulator, RViz can visualize any type of sensor data being published over a ROS topic: camera images, point clouds, ultrasonic measurements, lidar data, inertial measurements, and more. This data can be a live stream directly from the sensor or pre-recorded data stored as a **bagfile**.

You can also visualize live joint angle values from a robot and hence construct a real-time 3D representation of any robot. Having said that, RViz is not a simulator and does not interface with a physics engine. So RViz models neither collisions nor dynamics. RViz is not an alternative to Gazebo, but rather a complementary tool to keep an eye on every single process under the hood of a robotic system.

**A screenshot of a computer program

AI-generated content may be incorrect.**A screenshot of a computer

AI-generated content may be incorrect.

The empty window in the center is called **3D view**. This is where you will spend most of your time observing the robot model, the sensors, and other meta-data.

The panel on the left is a list of loaded **displays**, while the one on the right shows different **views** available.

At the top we have a number of useful **tools**. The bottom bar displays information like time elapsed, frames per second, and some handy instructions for the selected tool.

**Displays**

For anything to appear in the **3D view**, you first need to load a proper display. A display could be as simple as a basic 3D shape or a complex robot model.

Displays can also be used to visualize sensor data streams like 3D point clouds, lidar scans, or depth images.

RViz by default starts with two fixed property fields that cannot be removed: **Global Options** and **Global Status**. One governs simple global settings, while the other detects and displays useful status notifications.

For more information on RViz, check out their official guide [**here**](http://wiki.ros.org/rviz/UserGuide).

### RViz Integration

In this section, you will display your model into RViz and visualize data from the **camera** and **lidar** sensors. You will also **actuate** your robot and drive it around!

**Modify robot\_description**

Start by modifying the robot\_description.launch file. Open it and add these lines after the first param definition.

  <!-- Send fake joint values-->

  <node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher">

    <param name="use\_gui" value="false"/>

  </node>

  <!-- Send robot states to tf -->

  <node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="robot\_state\_publisher" respawn="false" output="screen"/>

Those elements add two nodes - the **joint\_state\_publisher** and the **robot\_state\_publisher**.

* joint\_state\_publisher: Publishes joint state messages for the robot, such as the angles for the non-fixed joints.
* robot\_state\_publisher: Publishes the robot's state to tf (transform tree). Your robot model has several frames corresponding to each link/joint. The robot\_state\_publisher publishes the 3D poses of all of these links. This offers a convenient and efficient advantage, especially for more complicated robots.

**Modify world.launch**

Next, you need to launch RViz along with Gazebo. Open the world.launch file and add these elements after the urdf\_spawner node definition:

*<!--launch rviz-->*

<node name="rviz" pkg="rviz" type="rviz" respawn="false"/>

This will create a node that launches the package rviz. Let's launch it 🚀

**Launch!**

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ roslaunch my\_robot world.launch

This time both Gazebo and RViz should launch!

**RViz Setup**

Setup RViz to visualize the sensor readings. On the left side of RViz, under Displays:

* Select odom for **fixed frame**
* Click the **Add** button and
  + add RobotModel and your robot model should load up in RViz.
  + add Camera and select the **Image topic** that was defined in the camera Gazebo plugin
  + add LaserScan and select the **topic** that was defined in the Hokuyo Gazebo plugin

**Add Objects**

In Gazebo, add a box, sphere or cylinder object in front of your robot. Your robot’s sensors should be able to visualize it. You can check the Camera viewer on the bottom left side of RViz to see a picture of the new object. Also, you can see a red Laser scan inside the scene, reflecting from your object.

A screenshot of a computer

AI-generated content may be incorrect.

**Drive Around**

While everything above is still running, test the robot’s actuators and drive it around. Open a new terminal window and publish velocity commands to the robot’s wheel actuators:

A screenshot of a computer program

AI-generated content may be incorrect.

**Note:** Do not type the entire rostopic pub command. Type up to rostopic pub /cmd\_vel, then press the **TAB key twice** to autocomplete the message. This ensures that you are publishing the command in the correct format. Once the complete message appears, modify the values to set the desired velocity.

This command publishes messages to the cmd\_vel topic defined in the drive controller plugin. The values set for linear.x and angular.z will enable the robot to start moving in a circle.

A screenshot of a computer program

AI-generated content may be incorrect.

### House your Robot

So far, you created a robot model from scratch, added sensors to it to visualize its surroundings, and developed a package to launch the robot in a simulated environment. That's a real accomplishment!

But you haven’t yet placed the robot in an environment. Let’s house it inside the world you built in **Build My World** project.

A computer screen shot of a room

AI-generated content may be incorrect.

**Adding the World File**

Copy the <yourname>.world file from the world directory of the **Build My World** project and paste it in the worlds directory of my\_robot.

Inside your package’s worlds directory you should now see two files - the empty.world that we created earlier and the <yourname>.world file that you just added.

Feel free to delete the empty.world file. We won’t need it anymore.

**Launch the World**

Edit the world.launch file and add a reference to the <yourname>.world file that you just added. To do so, open the world.launch file and edit this line:

<arg name="world\_file" **default**="$(find my\_robot)/worlds/empty.world"/>

Replace it with this:

<arg name="world\_file" **default**="$(find my\_robot)/worlds/<yourname>.world"/>

**Launch!**

Now, that you’ve added your world file to the my\_robot package, let’s launch and visualize our robot inside our home.

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ roslaunch my\_robot world.launch

A computer screen shot of a room

AI-generated content may be incorrect.

**Initialize the Robot’s Position and Orientation**

As you can see, my robot’s initial position is outside of my world! You might face the same problem. I have to change my robot’s initial pose: its position and orientation. This can be done through editing the world.launch file:

<!-- Robot pose -->

  <arg name="x" default="0"/>

  <arg name="y" default="0"/>

  <arg name="z" default="0"/>

  <arg name="roll" default="0"/>

  <arg name="pitch" default="0"/>

  <arg name="yaw" default="0"/>

The best way to figure out these numbers is to change the robot’s position and orientation within Gazebo, record its pose, and then update the launch file.

A computer screen shot of a room

AI-generated content may be incorrect.

A screenshot of a chat box

AI-generated content may be incorrect.

### Setting up ball\_chaser

The second major task in this project is to create the ball\_chaser ROS package. Within this package, you'll analyze the image captured by the camera to determine the position of a white ball. Then you’ll drive the robot toward the ball. The nodes in ball\_chaser will communicate with the my\_robot package by subscribing to the robot camera sensor and publishing to the robot’s wheel joints.

**Package Nodes**

The ball\_chaser package will have two C++ nodes: the drive\_bot and process\_image

* drive\_bot: This server node will provide a ball\_chaser/command\_robot **service** to drive the robot around by controlling its linear x and angular z velocities. The service will **publish** a message containing the velocities to the wheel joints.
* process\_image: This client node will **subscribe** to the robot’s camera images and analyze each image to determine the position of the white ball. Once ball position is determined, the client node will request a service to drive the robot either left, right or forward.

Now, follow along with the steps to set up ball\_chaser.

**Create the ball\_chaser Package**

**1- Navigate to the** **src** **directory of your** **catkin\_ws** **and create the** **ball\_chaser** **package:**

We will be writing nodes in C++. Since we already know in advance that this package will contain C++ source code and messages, let’s create the package with those dependencies:

$ cd ~/catkin\_ws/src/

$ catkin\_create\_pkg ball\_chaser roscpp std\_msgs message\_generation

**2- Next, create an** **srv** **and a** **launch** **folder, which will further define the structure of your package:**

$ cd ~/catkin\_ws/src/ball\_chaser/

$ mkdir srv

$ mkdir launch

Remember, srv is the directory where you store **service** files and **launch** is the directory where you store launch files. The src directory where you will store C++ programs is created by default.

**Build the Package**

$ cd ~/catkin\_ws/

$ catkin\_make

Now you should be ready to write some code!

A screenshot of a computer

AI-generated content may be incorrect.

### ROS Node: drive\_bot

This server node provides a ball\_chaser/command\_robot **service** to drive the robot around by setting its linear x and angular z velocities. The service server **publishes** messages containing the velocities for the wheel joints.

After writing this node, you will be able to request the ball\_chaser/command\_robot service, either from the terminal or from a client node, to drive the robot by controlling its linear x and angular z velocities.

**Reference**

The drive\_bot.cpp node is similar to the arm\_mover.cpp node that you already wrote. Both nodes contain a ROS **publisher** and **service**. But this time, instead of publishing messages to the arm joint angles, you have to publish messages to the wheels joint angles. Please refer to the arm\_mover.cpp node before you begin coding the drive\_bot.cpp node.

**ROS Service File**

**1- Write the** **DriveToTarget.srv** **file**

Create a DriveToTarget.srv file under the srv directory of ball\_chaser. Then, define the DriveToTarget.srv file as follows:

**Request**:

* linear\_x type **float64**
* angular\_z type **float64**

**Response**:

* msg\_feedback type **string**

**A screenshot of a computer

AI-generated content may be incorrect.**

**drive\_bot.cpp Node**

Now it’s time to write the drive\_bot.cpp server node that will provide the ball\_chaser/command\_robot service. Create the script under the src directory of your ball\_chaser package. It might be a bit challenging to write this script from scratch, thus I am providing you with some hints.

Attached below is a program that will continuously publish to the robot /cmd\_vel topic. This code will drive your robot forward:

#include "ros/ros.h"

#include "geometry\_msgs/Twist.h"

//TODO: Include the ball\_chaser "DriveToTarget" header file

// ROS::Publisher motor commands;

ros::Publisher motor\_command\_publisher;

// TODO: Create a handle\_drive\_request callback function that executes whenever a drive\_bot service is requested

// This function should publish the requested linear x and angular velocities to the robot wheel joints

// After publishing the requested velocities, a message feedback should be returned with the requested wheel velocities

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

{

        // Initialize a ROS node

        ros::init(argc, argv, "drive\_bot");

        // Create a ROS NodeHandle object

        ros::NodeHandle n;

        // Inform ROS master that we will be publishing a message of type geometry\_msgs::Twist on the robot actuation topic with a publishing queue size of 10

        motor\_command\_publisher = n.advertise<geometry\_msgs::Twist>("/cmd\_vel", 10);

        // TODO: Define a drive /ball\_chaser/command\_robot service with a handle\_drive\_request callback function

        // TODO: Delete the loop, move the code to the inside of the callback function and make the necessary changes to publish the requested velocities instead of constant values

        while (ros::ok()) {

            // Create a motor\_command object of type geometry\_msgs::Twist

            geometry\_msgs::Twist motor\_command;

            // Set wheel velocities, forward [0.5, 0.0]

            motor\_command.linear.x = 0.5;

            motor\_command.angular.z = 0.0;

            // Publish angles to drive the robot

            motor\_command\_publisher.publish(motor\_command);

        }

        // TODO: Handle ROS communication events

        //ros::spin();

        return 0;

}

Take a look at this program and try to understand what is happening. Then, copy it to drive\_bot.cpp, and make the necessary changes to define a ball\_chaser/command\_robot service.

**Edit CMakeLists.txt**

After you write the server node in C++, you’ll have to add the following dependencies:

* Add the add\_compile\_options for C++ 11 dependency, this step is optional and depends on your code
* Add the add\_service\_files dependency which defines the DriveToTarget.srv file
* Add the generate\_messages dependency
* Add include\_directories dependency
* Add the add\_executable, target\_link\_libraries, and add\_dependencies dependency for your drive\_bot.cppscript

**Note:** Use drive\_bot as the executable name for the add\_executable, target\_link\_libraries, and add\_dependencies functions to ensure that you can run with the upcoming rosrun command in the next sections.

A screenshot of a computer program

AI-generated content may be incorrect.

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ rosservice call /ball\_chaser/command\_robot "linear\_x: 0.5

angular\_z: 0.0"  # This request should drive your robot forward

$ rosservice call /ball\_chaser/command\_robot "linear\_x: 0.0

angular\_z: 0.5"  # This request should drive your robot left

$ rosservice call /ball\_chaser/command\_robot "linear\_x: 0.0

angular\_z: -0.5"  # This request should drive your robot right

$ rosservice call /ball\_chaser/command\_robot "linear\_x: 0.0

angular\_z: 0.0"  # This request should bring your robot to a complete stop

**Launch Files**

Let’s add the drive\_bot node to a launch file. Create a ball\_chaser.launch file under the launch directory of your ball\_chaser package and then copy this code to it:

<launch>

          <!-- The drive\_bot node -->

          <node name="drive\_bot" type="drive\_bot" pkg="ball\_chaser" output="screen">

          </node>

</launch>

This code will launch your drive\_bot node, which is contained in the ball\_chaser package. The server node outputs all the logs to the terminal window.

### Model a White Ball

Before you proceed to code the process\_image client node, you have to model a white ball and place it in your Gazebo World scene.

After modeling the white ball, you'll control its position in Gazebo by placing it at different positions in front of the robot’s camera. The process\_image client node will be responsible for analyzing the robot’s image and requesting services from the server drive\_bot node to drive the robot towards it.

Now, let’s go ahead and model the white ball using the Model Editor tool in Gazebo!

**Model Editor**

Here’s a reminder of how to open the model editor:

$ gazebo # then Edit-> Model Editor

**Insert Sphere**

Under the **simple shapes** menu of the Model Editor tool, click on a sphere and insert it anywhere in the scene.

**Edit Size**

Double click on the sphere, and change its radius to **0.1** both in Visual and Collision.

**Change Color**

To change the ball’s color to white, set its **Visual** Ambient, Diffuse, Specular, and Emissive RGBA values to 1.

A screenshot of a computer

AI-generated content may be incorrect.

**Save**

Save the white ball model as my\_ball under the /home directory. Then exit the Model Editor tool and go back to the Gazebo main world.

**Insert Ball**

Now that you are back in the Gazebo main world, you can click on “Insert” and drop the white ball anywhere in the scene.

A screenshot of a computer

AI-generated content may be incorrect.

**Inserting the ball into the world**

Now that you have modeled the white ball, open your world file using the command gazebo <path\_to\_your\_world\_file>. This command opens the world without spawning your robot. Then insert a my\_ball anywhere inside your world. The my\_ball model can be found under the Insert tab.

A computer screen shot of a room

AI-generated content may be incorrect.

**Save**

Place the white ball anywhere outside of your building structure, so that the robot would not see it. Then, save a copy of this new world under ~/catkin\_ws/src/my\_robot/worlds by replacing your old <yourname>.world file. Whenever you launch this newly saved world you should be able to see your building environment, in addition, the white ball.

A computer screen shot of a room

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

### ROS Node: process\_image

The second node that you’ll write in this project is the process\_image node. This client node will **subscribe** to the robot’s camera images and analyze them to determine the position of the white ball. Once the ball position is determined, the client node will request a service from the drive\_bot server node to drive the robot toward the ball. The robot can drive either left, right or forward, depending on the robot position inside the image.

After you write this node, place the white ball in front of the robot’s camera. If everything works, your node should analyze the image, detect the ball’s position, and then request a ball\_chaser/command\_robot service to drive the robot towards the ball!

**Reference**

The process\_image.cpp client node is similar to the look\_away.cpp client node that you wrote in this lesson. Both nodes contain a ROS **subscriber** and **client**. Please review the look\_away.cpp node before you start coding the process\_image.cpp node.

**Analyzing the Images**

To identify the ball’s **presence** and **position** inside the image, you will use a simple approach. First, search for white pixels inside the array image. Since the ball is the only object in the world that is white, white pixels indicate the ball’s presence. Then, once you find that the ball, identify its position with respect to the camera - either the left, middle, or right side of the image.

A diagram of a step forward

AI-generated content may be incorrect.

You’ll have to subscribe to the /camera/rgb/image\_raw topic to get instantaneous images from the robot’s camera. Inside the callback function, retrieve the image by reading the image data message. The image message contains many fields, such as the image height, data and more. Check out the complete [**ROS sensor\_msgs/Image documentation**](http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Image.html). Now that you have the image messages, you have to loop through the image data. For each pixel compare it to a value of 255 indicating a bright white pixel, if this pixel is found try to identify in which section of the image it fall either left, mid, or right. Then, request a service to drive toward that direction.

**Write process\_image.cpp**

Now it’s time to write the process\_image.cpp client node. This node will analyze the image and request services to drive the robot. Create the source code file within the src directory of your ball\_chaser package. It might be a bit challenging to write this program from scratch, thus I am providing you with some hints. Attached below is a piece of the complete code with multiple hints to help you finish the implementation.

#include "ros/ros.h"

#include "ball\_chaser/DriveToTarget.h"

#include <sensor\_msgs/Image.h>

// Define a global client that can request services

ros::ServiceClient client;

// This function calls the command\_robot service to drive the robot in the specified direction

void drive\_robot(float lin\_x, float ang\_z)

{

    // TODO: Request a service and pass the velocities to it to drive the robot

}

// This callback function continuously executes and reads the image data

void process\_image\_callback(const sensor\_msgs::Image img)

{

    int white\_pixel = 255;

    // TODO: Loop through each pixel in the image and check if there's a bright white one

    // Then, identify if this pixel falls in the left, mid, or right side of the image

    // Depending on the white ball position, call the drive\_bot function and pass velocities to it

    // Request a stop when there's no white ball seen by the camera

}

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

{

    // Initialize the process\_image node and create a handle to it

    ros::init(argc, argv, "process\_image");

    ros::NodeHandle n;

    // Define a client service capable of requesting services from command\_robot

    client = n.serviceClient<ball\_chaser::DriveToTarget>("/ball\_chaser/command\_robot");

    // Subscribe to /camera/rgb/image\_raw topic to read the image data inside the process\_image\_callback function

    ros::Subscriber sub1 = n.subscribe("/camera/rgb/image\_raw", 10, process\_image\_callback);

    // Handle ROS communication events

    ros::spin();

    return 0;

}

Copy this code to process\_image.cpp, and make the necessary changes.

**Edit CMakeLists.txt**

In addition to all the dependencies you added earlier for drive\_bot.cpp, these are the dependencies that you should add for process\_image.cpp :

* Add add\_executable
* Add target\_link\_libraries
* Add add\_dependencies

**Build Package**

Now that you’ve included specific instructions for your process\_image.cpp code in CMakeLists.txt, compile it with:

$ cd ~/catkin\_ws/

$ catkin\_make

* Launch File

Edit the ball\_chaser.launch file saved under ~/catkin\_ws/src/ball\_chaser/launch and add the process\_image node to it.

Now, launching this file should run the drive\_bot and process\_image!

**Test process\_image**

To test if the code you just wrote is working as expected, first launch the robot inside your world and then run both the drive\_bot and process\_image nodes.

**1- Launch the robot inside your world**

This can be done by launching the world.launch file:

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ roslaunch my\_robot world.launch

**2- Run** **drive\_bot** **and** **process\_image**

This can be done by executing ball\_chaser.launch:

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ roslaunch ball\_chaser ball\_chaser.launch

**3- Visualize**

To visualize the robot’s camera images, you can subscribe to camera RGB image topic from RViz. Or you can run the rqt\_image\_view node:

$ cd ~/catkin\_ws/

$ source devel/setup.bash

$ rosrun rqt\_image\_view rqt\_image\_view

Now place the white ball at different positions in front of the robot and see if the robot is capable of chasing the ball!

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

### Processing Image Messages

**Understanding and Processing sensor\_msgs::Image Messages in ROS**

This section will familiarize you with the Image messages and how to process them to find the white ball in your project. In ROS, camera data is represented using the sensor\_msgs::Image message type, which includes all necessary details about the image captured by the robot's camera. Minimal code is provided to help you overcome the challenging part of processing the image data.

Concepts you will learn:

* The structure of sensor\_msgs::Image messages.
* Learn how to process ROS image data.
* Understand how to detect white color pixels in an image.

**What is sensor\_msgs::Image?**

The sensor\_msgs::Image message is a standard ROS message type used to represent camera images.

**Key fields include:**

* height and width: The image dimensions in pixels.
* encoding: The format of the image (e.g., rgb8, mono8).
* step: The number of bytes per row (equals width × number of channels).
* data: A flattened array of pixel values, where pixel information is stored sequentially.

**How to Process an Image?**

**Set up a subscriber:**

* Subscribe to the camera topic (e.g. /camera/rgb/image\_raw) to receive the image messages. Change the topic name to your camera's topic.
* Use a callback function to process each incoming image.

**Iterate over pixel data in the callback function:**

* Extract pixel values (RGB components) from the data field of the message.
* Loop through the pixel data based on the image's encoding and step.
* Example code for reading image data:

for (size\_t i = 0; i < img->height \* img->step; i += 3) {

       int red = img->data[i];

       int green = img->data[i + 1];

       int blue = img->data[i + 2];

       // Next check if you found the white color ball

}

* **Why i += 3?** Each pixel is represented by three consecutive bytes for Red, Green, and Blue. Incrementing by 3 ensures you move to the next pixel correctly.

**How to Detect White Color Pixels?**

* To detect the white color, check pixel values to identify specific colors, like white (R=255, G=255, B=255) or red (R=255, G=0, B=0).
* Example code for color detection:

if (red == 255 && green == 255 && blue == 255) {

       // Found a white pixel, which means finding the ball

       // Next step would be decide how to move based on the

       // calculated pixel position or coordinates in the image

}

**Calculate the** **white pixel position in the image:**

* Notice that the image pixel data is stored in a flattened 1D array (img->data), even though the image itself is a 2D grid of pixels.
* Use the pixel index to compute its position in the image which helps you decide the direction towards which the robot should move if the ball is found.
  + **Horizontal position:** (i / 3) % img->width
  + **vertical position:** (i / 3) / img->width
* Note: the **horizontal position** (or x position) can help you decide where the ball with respect to the robot. For example, if you divide the image width into three regions from the left to the right, if the ball is detected in the first third, it would make it to the robot's left. If it is in the middle, it is in front of the robot, and if it is on the right third of the image, it is to the right.

This information will facilitate your selection of the robot motion direction to chase the ball based on the detected white pixels position in the image.

### Project Rubric

**Basic Requirements**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Does the submission include the **my\_robot** and the **ball\_chaser** ROS packages? | The student submitted all required files specified in the criteria. |
| Do these packages follow the directory structure detailed in the project description section? | The student followed the same directory structure detailed in the project description section. |

**Robot Design**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Does the submission include a design for a differential drive robot, using the Unified Robot Description Format? | Robot design requirements:   * Lidar and camera sensors. * Gazebo plugins for the robot’s differential drive, lidar, and camera. * Housed inside the world * Significant changes from the sample taught in the project lesson. Minimum mandatory changes are :   + Color,   + Wheel radius, and   + Chassis dimensions * Robot design is stable |

**Gazebo World**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Does the **my\_robot** ROS package contain the Gazebo world? | Gazebo world requirements:   * A new world that you design on the building editor for this project. * Includes a white-colored ball |

**Ball Chasing**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Does the **ball\_chaser** ROS package contain two C++ ROS nodes to interact with the robot and make it chase a white-colored ball? | drive\_bot requirements:   * A ball\_chaser/command\_robot service. * Service accepts linear x and angular z velocities. * Service publishes to the the wheel joints. * Service returns the requested velocities.   process\_image requirements:   * Subscribes to the robot’s camera image. * A function to analyze the image and determine the presence and position of a white ball. * Requests a service to drive the robot towards a white ball (when present). |

**Launch Files**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Does the submission include world.launch and ball\_chaser.launch files that launch all the nodes in this project? | world.launch requirements:   * Launch the world (which includes a white ball). * Launch the robot.   ball\_chaser.launch requirements:   * Run the drive\_bot C++ node. * Run the process\_image C++ node. |

**Suggestions to Make Your Project Stand Out**

Stand out submissions should have skid-steer robots designs or robots that can chase balls of any color.

### Submit Project

Project Description

**Summary of Tasks**

In this project, you should create two ROS packages inside your catkin\_ws/src: the drive\_bot and the ball\_chaser. Here are the steps to design the robot, house it inside your world, and program it to chase white-colored balls:

1. drive\_bot:
   * Create a my\_robot ROS package to hold your robot, the white ball, and the world.
   * Design a differential drive robot with the Unified Robot Description Format. Add two sensors to your robot: a lidar and a camera. Add Gazebo plugins for your robot’s differential drive, lidar, and camera. The robot you design should be significantly different from the one presented in the project lesson. Minimum required changes are adjusting the color, wheel radius, and chassis dimensions. You can also completely redesign the robot model! After all, you want to impress your future employers :-D
   * Create a new world, which is different from the world you built in the **Build My World** project and house your robot inside that world.
   * Add a white-colored ball to your Gazebo world and save a new copy of this world.
   * The world.launch file should launch your world with the white-colored ball and your robot.
2. ball\_chaser:
   * Create a ball\_chaser ROS package to hold your C++ nodes.
   * Write a drive\_botC++ node that will provide a ball\_chaser/command\_robot service to drive the robot by controlling its linear x and angular z velocities. The service should publish to the wheel joints and return back the requested velocities.
   * Write a process\_image C++ node that reads your robot’s camera image, analyzes it to determine the presence and position of a white ball. If a white ball exists in the image, your node should request a service via a client to drive the robot towards it.
   * The ball\_chaser.launch should run both the drive\_bot and the process\_image nodes.

The robot you design in this project will be used as a base model for all your upcoming projects in this Robotics Software Engineer Nanodegree Program.

**Evaluation**

Once you finish designing your robot and building the nodes, check the [**Project Rubric**](https://review.udacity.com/#!/rubrics/2397/view) to see if it meets the specifications. If you meet the specifications, then you are ready to submit!

If you do not meet specifications, keep working and discussing with your fellow students and mentors.

**Submission Folder**

Your submission should follow the directory structure and contain all the files listed here:

    .Project2                          # Go Chase It Project

    ├── my\_robot                       # my\_robot package

    │   ├── launch                     # launch folder for launch files

    │   │   ├── robot\_description.launch

    │   │   ├── world.launch

    │   ├── meshes                     # meshes folder for sensors

    │   │   ├── hokuyo.dae

    │   ├── urdf                       # urdf folder for xarco files

    │   │   ├── my\_robot.gazebo

    │   │   ├── my\_robot.xacro

    │   ├── world                      # world folder for world files

    │   │   ├── <yourworld>.world

    │   ├── CMakeLists.txt             # compiler instructions

    │   ├── package.xml                # package info

    ├── ball\_chaser                    # ball\_chaser package

    │   ├── launch                     # launch folder for launch files

    │   │   ├── ball\_chaser.launch

    │   ├── src                        # source folder for C++ scripts

    │   │   ├── drive\_bot.cpp

    │   │   ├── process\_images.cpp

    │   ├── srv                        # service folder for ROS services

    │   │   ├── DriveToTarget.srv

    │   ├── CMakeLists.txt             # compiler instructions

    │   ├── package.xml                # package info

    └──

A screenshot of a checklist

AI-generated content may be incorrect.