The Construct
Learn and develop for robots

- *"A fun and easy way to use ROS."*
  James Gray
  Ph.D. Student of A.I.

- *"The faster on-ramp to this next generation robotics toolkit."*
  Jim Keravala
  Chief Executive Officer at OffWorld Consortium

- *"An excellent tool for learning ROS and robotics in general."*
  Jorge Gomes
  Ph.D. Student

- *"I knew a little bit of ROS from fidding around with it before this course, but now I feel like I could do anything with it."*
  Daneil Ingram
  Lecturer | Researcher at Appalachian State University

- *"The ROS IN 5 DAYS is incredibly informative. I would highly recommend any future books by The Construct team!"*
  Miguel Duarte
  Researcher of Robotics and A.I.

The Construct

ROS BASICS IN 5 DAYS

Téllez | Ezquerro | Rodríguez

The Construct

Ricardo Téllez, Ph.D.
Alberto Ezquerro
Miguel Ángel Rodríguez

2nd Version

ROS BASICS IN 5 DAYS

www.theconstructsim.com

ENTIRELY PRACTICAL ROBOT OPERATING SYSTEM TRAINING

ROS BASICS IN 5 DAYS

Ricardo Téllez

Alberto Ezquerro

Miguel Angel Rodríguez

The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,

+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Written by Miguel Angel Rodríguez, Alberto Ezquerro and Ricardo Téllez
Edited by Yuhong Lin and Ricardo Téllez
Cover design by Lorena Guevara
Learning platform implementation by Ruben Alves

Version 3.0

The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,
+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Index

Prologue

Prologue

ROS, the most popular open-source, meta-operating system for robots, provides you with tools and libraries that allow breaking up code into modular and reusable packages. In this form, you can benefit from third-party code that your community has developed and share your own creations easily. However, this rather large system assumes good working knowledge of networking paradigms, so the ROS learning curve is a little steep and to become proficient is pretty hard for a complete beginner.

ROS in 5 days is the book associated to a course giving you the basic tools and knowledge to be able to understand and create any basic ROS related project. You will be able to move robots, read their sensor data, do parallel tasks, see visual representations of complex data... and much more. This course is divided in two main parts. In the first one, about learning ROS with exercises, you will learn executing code and using different simulated robots for it. The second one is about developing a project where you will apply what you have learned in each unit in a full project controlling a simple robot.

Cecilio Angulo, Barcelona 2016
Associated professor, research leader and former coordinator of the Master's degree in Automatic Control and Robotics at Technical University of Catalonia

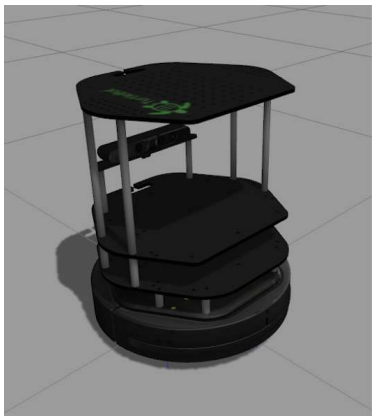
Preface

This book relies on a series of Gazebo simulations in order to teach you ROS. You need to have the simulations running while following the course, if you really want to learn ROS in 5 days, since we heavily rely on them to explain you the complex concepts of ROS.

The simulations

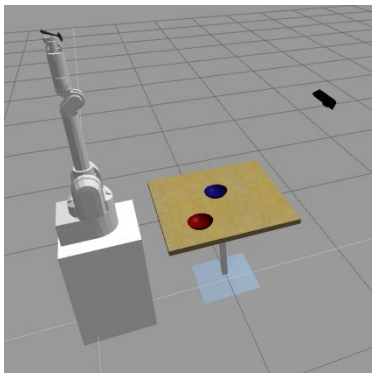
The list of simulations that you need to execute are the following:

Kobuki



Kobuki

Repository Link: https://bitbucket.org/theconstructcore/turtlebot/src/master/
WAM



Wam Arm

Repository Link: https://bitbucket.org/theconstructcore/iri_wam/src/master/

BB-8

BB8

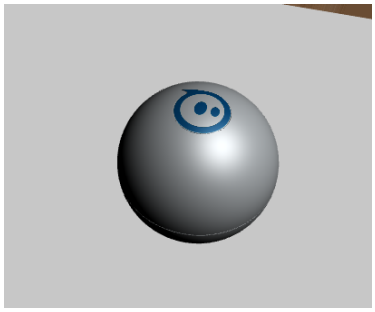
Repository Link: https://bitbucket.org/theconstructcore/bb8/src/master/

Parrot Drone

Parrot Drone

Repository Link: https://bitbucket.org/theconstructcore/parrot_ardrone/src/master/

Sphero



Sphero

Repository Link: https://bitbucket.org/theconstructcore/sphero/src/master/

How to launch the simulations

In order to launch the simulations you have two options:

1- **Launch the simulations in our ROS Development Studio (ROSDS) using the ROSjects we provide**. ROSDS is the The Construct web based tool to program ROS robots online. It is recommended that you use this option since it is very simple and requires no installation in your computer. Hence, you can use any type of computer to follow this book (Windows, Linux or Mac). Additionally, free accounts are available. You just need to paste the ROSject link we will provide to you in each Unit to a browser's URL, and you will automatically have the simulation prepared in your ROSDS workspace. **Create a free ROSDS account here**: http://rosds.online
2- **Launch the simulations in your own computer**. This option requires that you have a running Linux computer with ROS and Gazebo simulator installed in your computer. We do not cover those steps (but more information below)

1. Launch the simulation using ROSjects

You will find a complete guide about **ROSjects** and how to open them in the next Chapter of the Book.

2. Launch the simulation in your own computer

We do not recommend this option because it requires you to know about ROS concepts that you still don't have (because you are going to learn them in this book). However, we provide this option for more advanced users who more or less know what they are doing. If that is not your case, use the option of launching the simulations on the ROSDS, which requires no previous knowledge of ROS.

For this option you will need to:

1- Download the simulations from our Public repository, following the links provided above.

2- Install and set up a ROS + Gazebo environment in your PC. The instructions to do that are explained in the official page of ROS (http://www.ros.org). Bear in mind that the simulations you downloaded are supported for ROS Kinetic with Gazebo 7. Any other ROS version might not compile or work correctly.

3- You will also find some extra information on how to install ROS and setup a ROS workspace in your local computer on the **Appendix 1** and **Appendix 2** of the book.

Consultations

If you have any doubts while doing the Course, or you get really sucked in some of the exercises, you can ask for support in our forum:

http://forum.theconstructsim.com

ROSjects

ROSjects

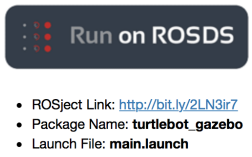
Throughout the whole book, you're going to **find a ROSject at the beginning of each Chapter**. With these ROSjects, you are going to be able to easily have access to all the material you'll need for each Chapter.

What is a ROSject?

A ROSject is, basically, a ROS project in the ROS Development Studio (ROSDS). ROSjects can easily be shared using a link. By clicking on the link, or copying it to the URL of your web browser, you will have a copy of the specific ROSject in your ROSDS workspace. This means you will have instant access to the ROSject. Also, you will be able to modify it as you wish.

How to open a ROSject?

At the beginning of each Chapter, you will see a section like this one:



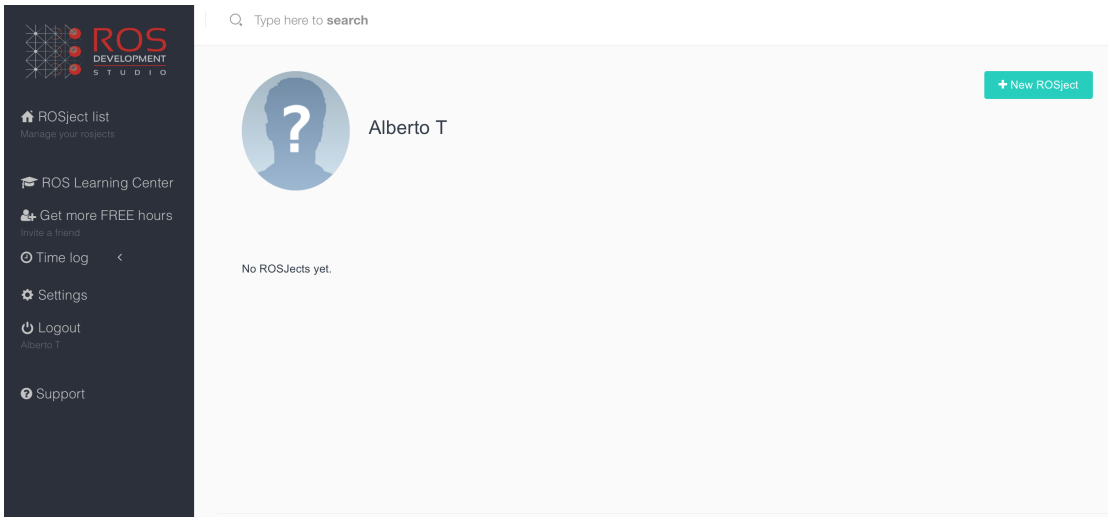
ROSject section

As you can see, it contains 3 things:

- **ROSject Link:** Link to get the ROSject
- **Package Name:** Name of the ROS package that contains the launch file to start the Gazebo simulation.
- **Launch File:** Name of the launch file that will start the Gazebo simulation related to the ROSject.

Step 1

Log into the ROSDS platform at http://rosds.online . If you don't have an account, you can create one for free. Once you log in, you will see an screen like the below one.

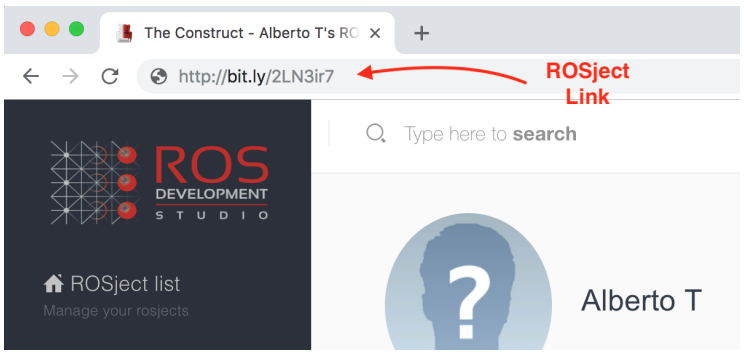


ROSjects Empty List

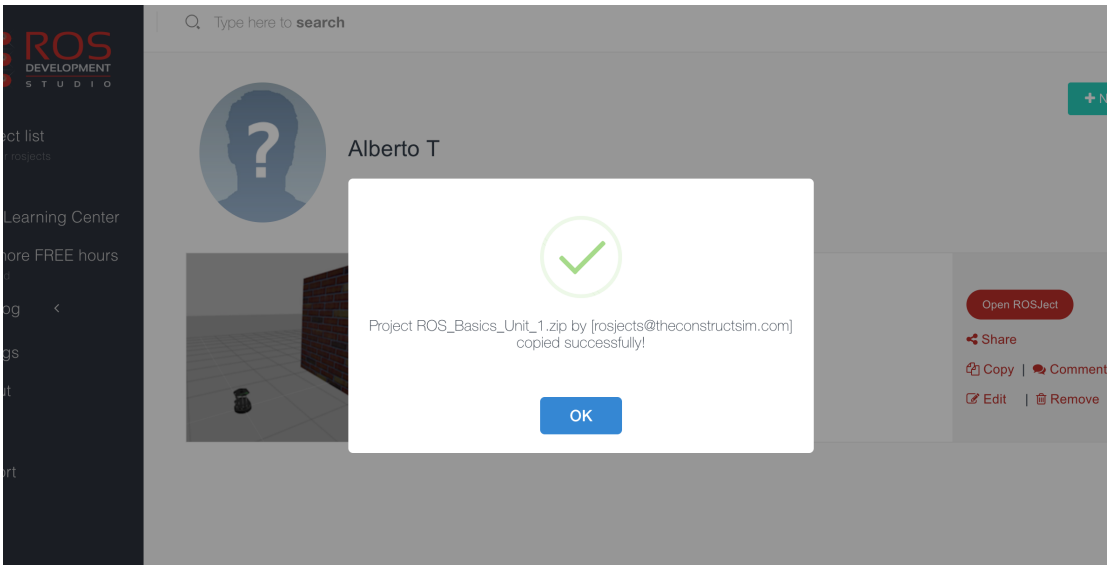
At this point you don't have any ROSject yet. So let's get one!

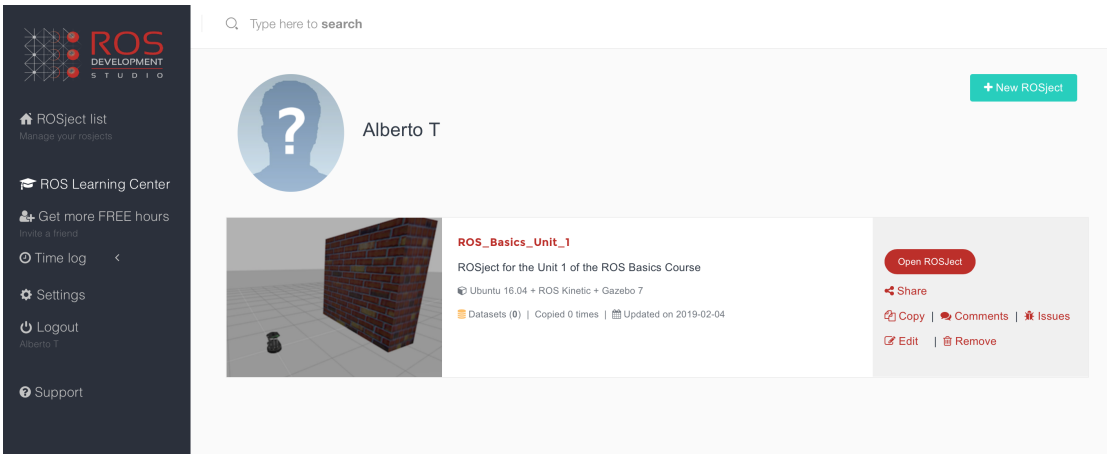
Step 2

Copy the ROSject Link to your web browser. Once you have copied the URL to your web browser, you will automatically have that ROSject available in your workspace.



ROSject Link

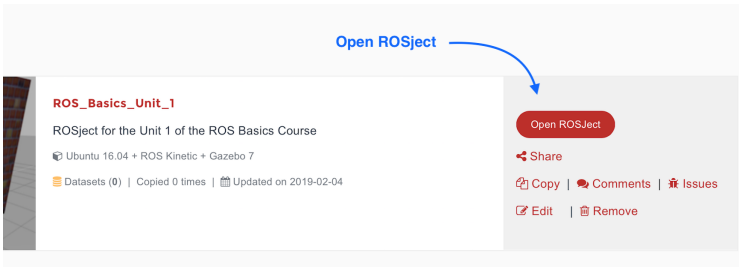




ROSject in ROSDS workspace

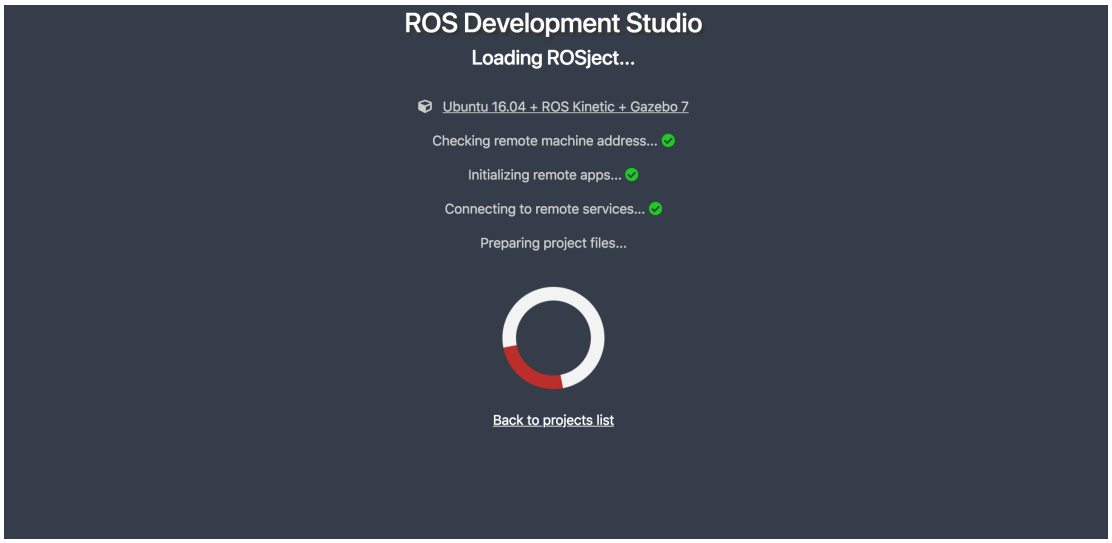
Step 3

Open the ROSject. You can open the ROSject by clicking on the **Open ROSject** button.

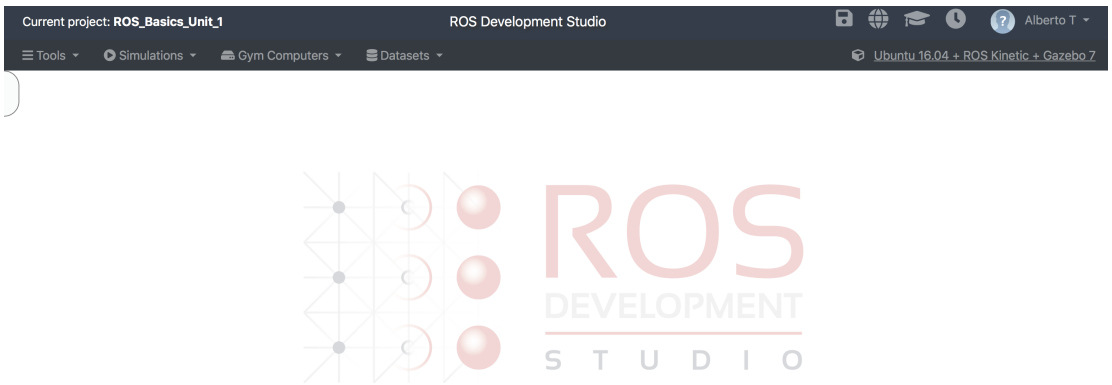


Open ROSject

You will then go to a loading screen like the below one.



After a few seconds, you will get an environment like the below one.



ROSDS Environment

Contents of the ROSjects

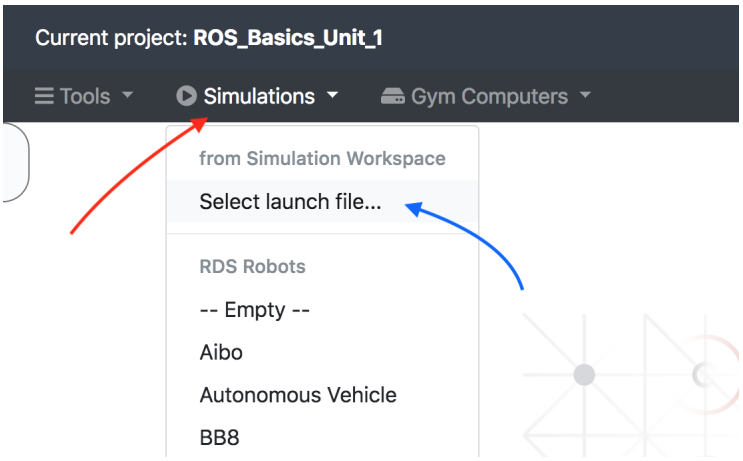
The ROSjects that are available for each chapter of the book will basically contain 2 things:

- The Gazebo simulation used for the Chapter
- All the scripts and files used for the Chapter

In order to open the simulation, follow the next steps:

Step 1

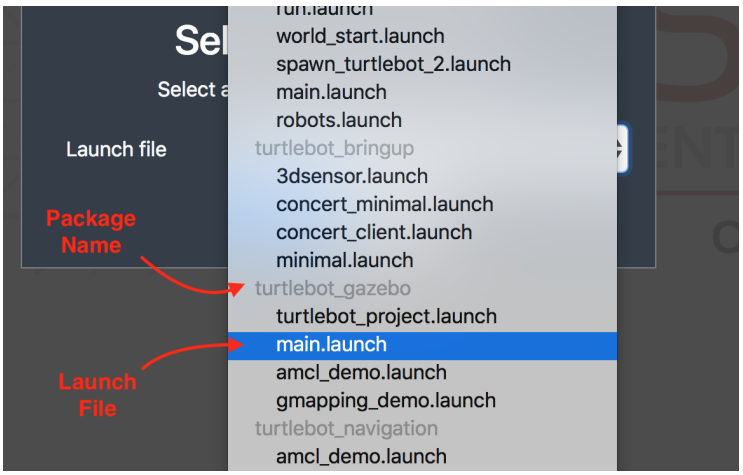
In the **Simulations** menu, click on the **Select launch file...** option. A list with all the packages and launch files available will appear.



Simulations menu

Step 2

Within the list, select the **package name** and **launch file** specified at the ROSject section of the chapter.

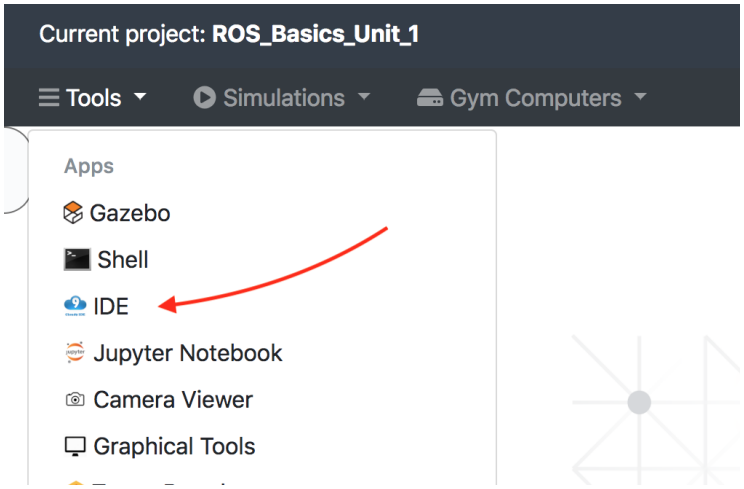


Launch Files list

In order to see all the files related to the chapter, follow the next steps:

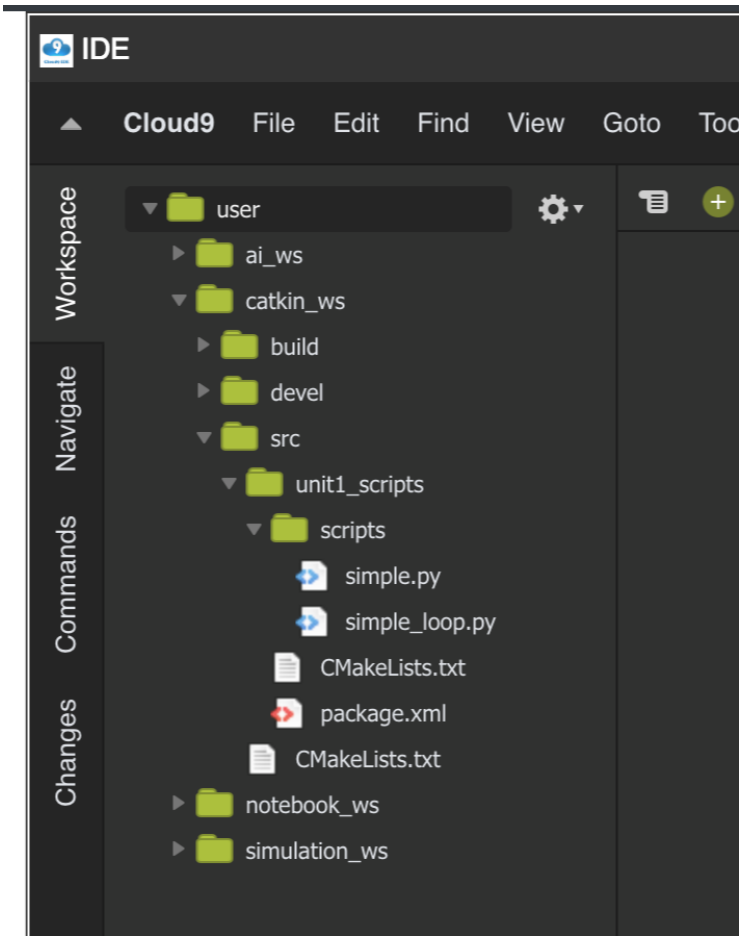
Step 1

In the **Tools** menu, click on the **IDE** option. An IDE will appear in your workspace.



Tools menu

2- You will find all the files inside the **catkin_ws** workspace. There, you have a ROS package containing all the files related to the chapter. This package will always follow the following name convention: **<unitX>_scripts**



IDE workspace tree

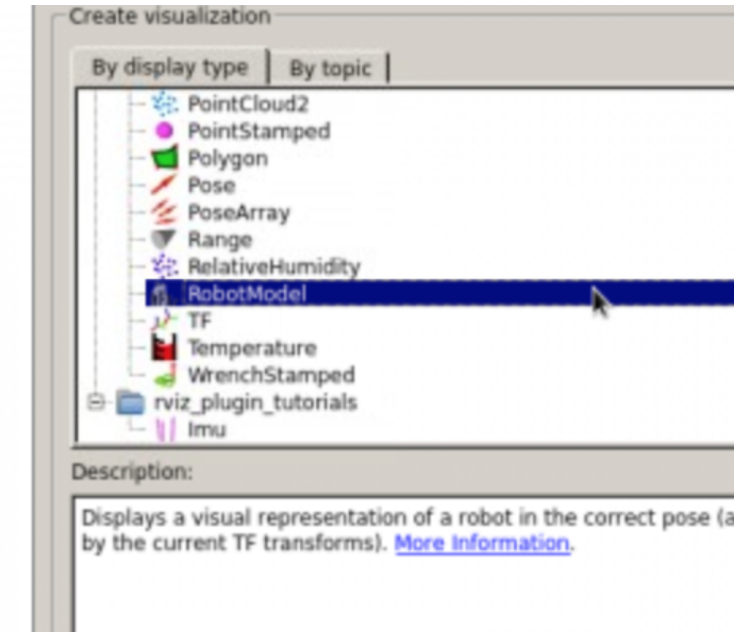
And that's it! Now just enjoy ROSDS and push your ROS learning.

Also, you can find more information and videotutorials about ROSDS here: https://www.youtube.com/watch?v=ELfRmuqgxns&list=PLK0b4e05LnzYGvX6EJN1gOQEl6aa3uyKS If you have any doubt regarding ROSDS, don't hesitate in contacting us to: **feedback@theconstructsim.com**

Chapter 0. A guide for ROS in 5 days

ROS IN 5 DAYS

Chapter 0: A Guide for ROS in 5 Days



RViz Robot Model plus TF

ROS in 5 days, I cannot believe it!

You have probably heard of ROS and how long it takes to learn to all students. It is very likely that you are thinking that learning ROS in 5 days is not possible. . . so let me tell you the news:

It is possible to learn ROS fast if you have the proper method!

Learn ROS fast if you have the proper method

How is it possible?

We have created a learning method that allows you to get most of ROS in the minimum amount of time. Our method has 4 parts:

1. **DECONSTRUCTION**: we have identified the important parts of ROS that you must master in order to understand 80% of ROS programs. You will concentrate on learning these parts very deep.
2. **REMOVING**: we have removed many things that are not needed and just add noise to your learning.
3. **LEARNING**: we guide you step by step in a progressive manner through all those important parts, starting always from a robot that does things.
4. **PRACTICING**: we make you practice a lot on every step, always on a robot using our simulated robots.

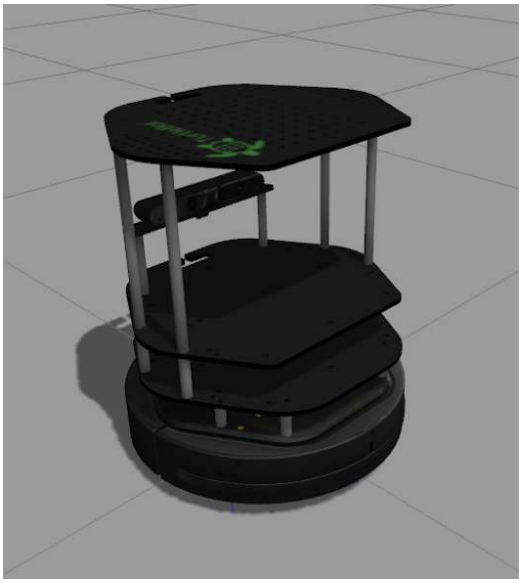
- Want to learn more about this method? Read this book: The First 20 Hours: How to Learn Anything . . . Fast!
- Want to have scientific evidence of this method? Read this paper: Cognitive Skill Acquisition.

Which robots are you going to use along the course?

Along this course you are going to program several robots of several types. Hence, you will be applying the same concepts over and over again, with different configurations and styles of robots. Applying the same concepts on different robots will make the concepts stick into your head.

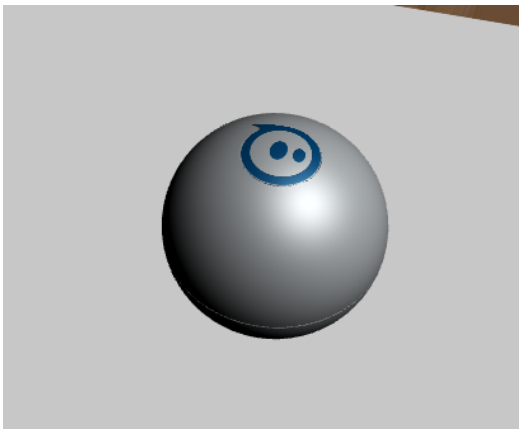
The following robots will be used along the course:

Parrot Drone



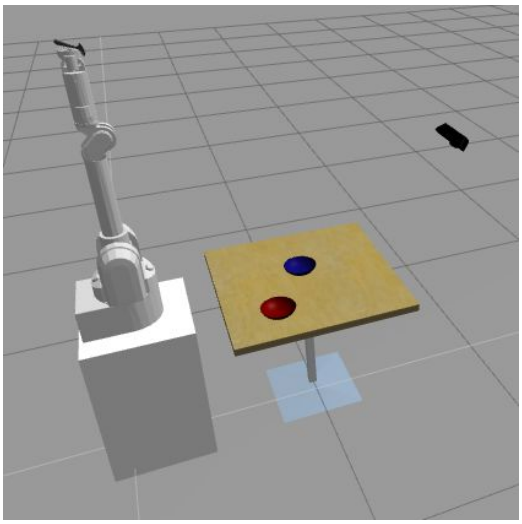
Kobuki

Husky



Sphero

BB-8



Wam Arm

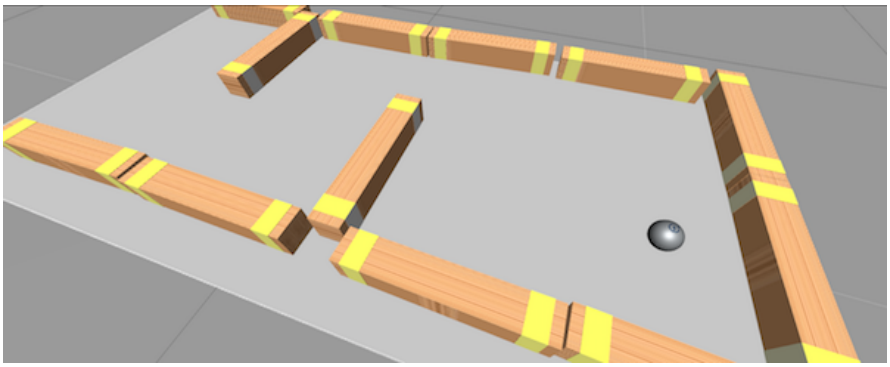
Main Objective of this course

1. The objective of this course is to give you the basic tools and knowledge to be able to understand and create any basic ROS related project. You will be able to **move robots, read their sensor data, make the robots perform intelligent tasks, see visual representations of complex data such as pointclouds and debug errors in the programs.**
2. The course will allow you to **understand packages that others have created**. So you can take ROS code made by others and understand what is happening and how to modify it for your own purposes.
3. This course can serve as an introduction to be able to understand the ROS documentation of complex ROS packages for object recognition, text to speech, navigation and all the other areas where has ROS developed code.

Learning ROS: attack in two ways

The course provides teaching lessons in two different ways:

1. Learn ROS programming several robots: This part consists of units each of which teaches you some topic of ROS. But you will learn creating and executing code while using different robots for it (the robots above).Theory through hands on experience.
2. Apply what you learnt to a Robot Project: Here you will apply what you have learnt in the previous units by attacking a full project controlling a Sphero robot. The objective is to make the Sphero get out of a maze of blocks with only imu and odometry data.



Sphero in the Maze

Learning ROS by programming several robots:

We will teach you the main ROS concepts that are the core of ROS. These are the most important concepts that you have to master. Once you master them, the rest of ROS can follow easily.

Along the units of this course, you will learn:

- Unit 1: How **ROS Basic Structure** works.
- Units 2 and 3: What are **ROS Topics** and how to use them.
- Units 4 and 5: What are **ROS Services** and how to use them.
- Units 7 and 8: What are **ROS Actions** and how to use them.
- Unit 9: How to use **ROS Debugging Tools** for finding errors in your programs (especially Rviz).

Each Unit is meant to be done in around one session of about 4-5 hours. We will use Python language to teach you and to program the robots.

Apply what you learnt to a Robot Project:

As you master the different concepts of ROS, you will have to apply your learning in a complete robot project. We suggest that you combine the learning of the lessons while progressing on the project. That is why, we have divided the project into 4 sections:

- Project Section 1: Create topic publishers and subscribers for moving the sphero and reading the imu data and the odometry data.
- Project Section 2: Create a service that, when used, it reads the imu and tells you which way to go.
- Project Section 3: Create an action that, when used, it records the odometry and tells you what distance has the Sphero moved.
- Project Section 4: Create a global program that reads/write into topics, uses the service and action that you created to make the robot get out of the maze.

How to proceed with the whole course?

The course has been designed to allow you to complete the whole thing in only 5 days, dedicating 6 hours of work per day.

However, YOU ARE NOT REQUIRED TO DO THE COURSE IN 5 DAYS! You can do it in as many days as you want, because this is a self paced course and you wont be restricted in the time of usage.

In any case, if you decide to take our proposal of 5 days we recommend you to follow this order:

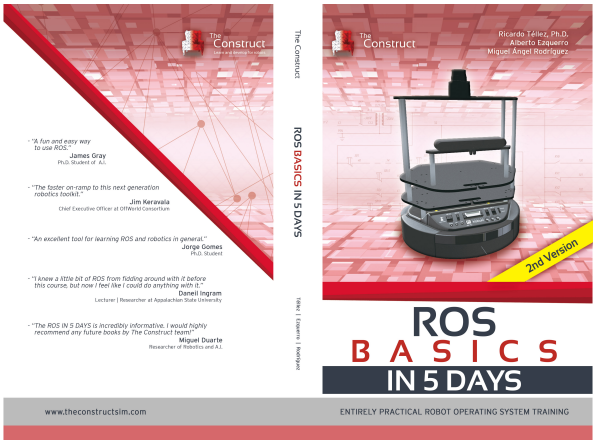
- **DAY 1**
- Unit 1: How ROS Basic Structure works, Time: 1'5h
- Start Unit 2: ROS Topics,Time: 2'5h
- Work on Project Section 1, Time: 2h
- **DAY 2**
- Finish Unit 2: ROS Topics, Time: 2'5h
- Unit 5: ROS Debugging Tools, Time: 1'5h
- Finish Project Section 1, Time: 2h
- **DAY 3**
- Start Unit 3: ROS Services, Time: 4h
- Finish Project Section 2, Time: 2h
- **DAY 4**
- Finish Unit 3: ROS Services, Time: 1'5h
- Start Unit 4: ROS Actions, Time: 2'5h
- Finish Project Section 3, Time: 2h
- **DAY 5**
- Finish Unit 4: ROS Actions, Time: 3h
- Finish Project Section 4, Time: 3h

The most important thing that must take into account is:

DO NOT SKIP EXERCISES.
Exercises are the core of this teaching method (remember, practice, practice, practice). If you avoid them, you will be missing the whole thing.

Other Options

If you're excited about the idea of Learning ROS in a very easy and fast way, but you still feel like you would prefer to do it more at your own pace... don't worry! We still have an option for you. The Construct has released a book version of this Course, so you can still follow the Course as you prefer. You can order the book in this link: http://www.theconstructsim.com/ros-in-5-days-book-page.



ROS in 5 Days Book

Where to go when finished

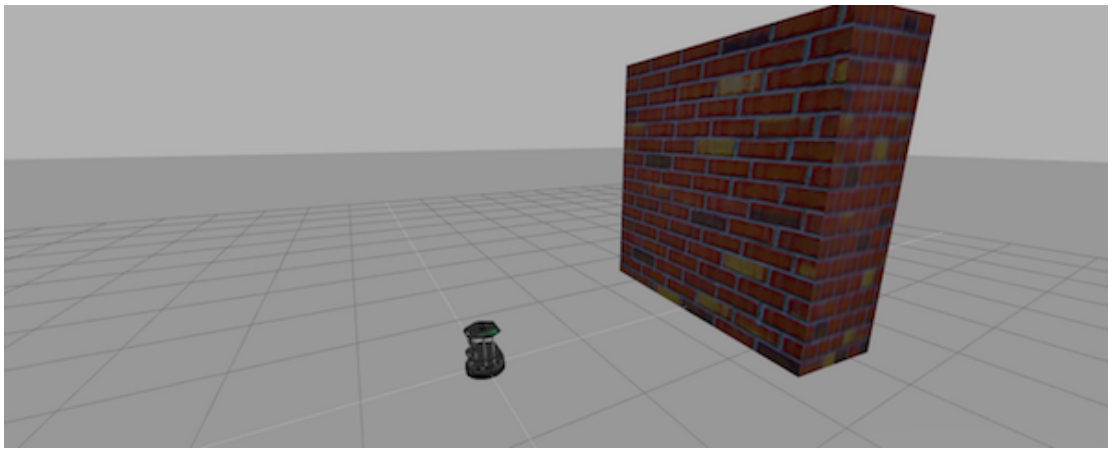
Once you have finished the course, you can still learn a lot of interesting subjects of ROS.

- Take more advanced courses we offer at Ignite Academy, like Navigation, Perception or Manipulation.
- You can apply all you have learned in the free simulation environment of the The Construct, ROS Development Studio, and create your own programs for ROS based robots.
- Or you can go to the ROS Wiki and check the official documentation of ROS, now with different eyes.

Unit 1. Basic Concepts

ROS IN 5 DAYS

Unit 1: Basic Concepts

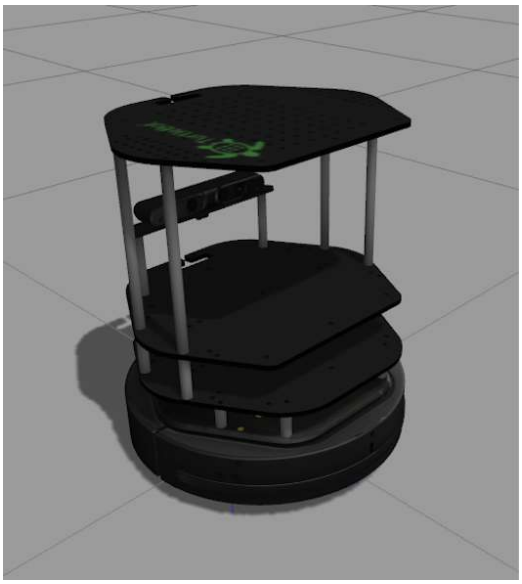


Kobuki

- ROSject Link: http://bit.ly/2LN3ir7
- Package Name: **turtlebot_gazebo**
- Launch File: **main.launch**

Estimated time to completion: 1'5 hours
Simulated robot: Turtlebot
What will you learn with this unit?

- How to structure and launch ROS programs (packages and launch files)
- How to create basic ROS programs (Python based)
- Basic ROS concepts: Nodes, Parameter Server, Environment Variables, Roscore



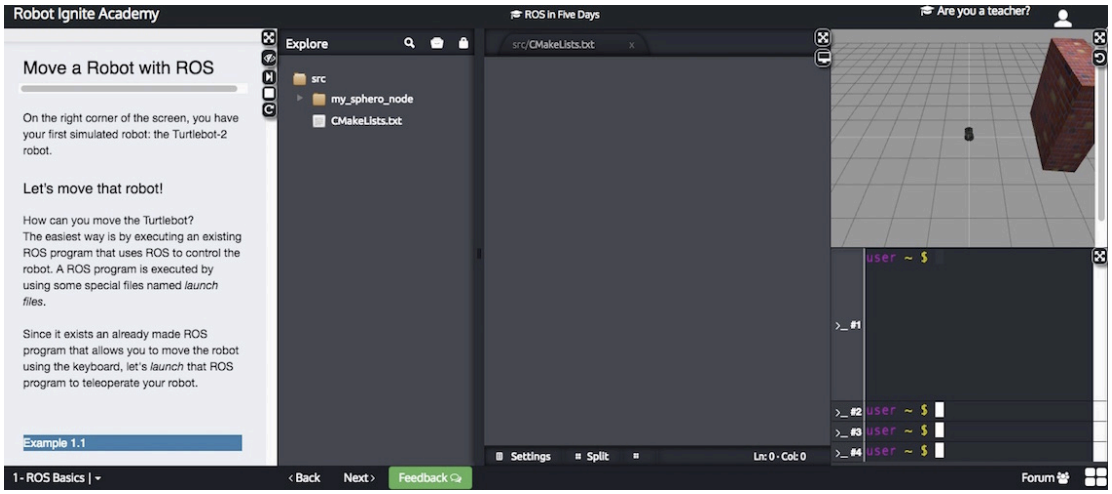
Kobuki Robot

What is ROS?

This is probably the question that has brought you all here. Well, let me tell you that you are still not prepared to understand the answer to this question, so... let's get some work done first.

Move a Robot with ROS

On the right corner of the screen, you have your first simulated robot: the Turtlebot 2 robot against a large wall.



Robot Ignite Environment

Let's move that robot!
How can you move the Turtlebot? The easiest way is by executing an existing ROS program to control the robot. A ROS program is executed by using some special files called launch files. Since a previously-made ROS program already exists that allows you to move the robot using the keyboard, let's launch that ROS program to teleoperate the robot.

Example 1.1
Execute the following command in WebShell number #1
Execute in WebShell #1

```
[ ]: roslaunch turtlebot_teleop keyboard_teleop.launch
```

WebShell #1 Output

```
[ ]: Control Your Turtlebot!
     ---------------------------
     Moving around:
        u    i    o
        j    k    l
        m    ,    .

     q/z : increase/decrease max speeds by 10%
     w/x : increase/decrease only linear speed by 10%
     e/c : increase/decrease only angular speed by 10%
     space key, k : force stop
     anything else : stop smoothly

     CTRL-C to quit
```

Now, you can use the keys indicated in the WebShell Output in order to move the robot around. The basic keys are the following:

i	**Move forward**
,	**Move backward**
j	**Turn left**
l	**Turn right**
k	**Stop**
q z	**Increase / Decrease Speed**

Keys to control Kobuki Robot

Try it!! When you're done, you can press **Ctrl+C** to stop the execution of the program.

roslaunch is the command used to launch a ROS program. Its structure goes as follows:

```
[ ]: roslaunch <package_name> <launch_file>
```

As you can see, that command has two parameters: the first one is the name of the package that contains the launch file, and the second one is the name of the launch file itself (which is stored inside the package).

Now... what's a package?

ROS uses packages to organize its programs. You can think of a package as all the files that a specific ROS program contains; all its cpp files, python files, configuration files, compilation files, launch files, and parameters files.

All those files in the package are organized with the following structure:

- **launch** folder: Contains launch files
- **src** folder: Source files (cpp, python)
- **CMakeLists.txt**: List of cmake rules for compilation
- **package.xml**: Package information and dependencies

To go to any ROS package, ROS gives you a command named roscd. When typing:

```
[ ]:  roscd <package_name>
```

It will take you to the path where the package package_name is located.

Example 1.2
Go to WebShell #1, navigate to the turtlebot_teleop package, and check that it has that structure.

Execute in WebShell #1

```
[ ]: roscd turtlebot_teleop
     ls
```

Every ROS program that you want to execute is organized in a package. Every ROS program that you create will have to be organized in a package. Packages are the main organization system of ROS programs.

And... what's a launch file?

We've seen that ROS uses launch files in order to execute programs. But... how do they work? Let's have a look.

Example 1.3
Open the **launch** folder inside the **turtlebot_teleop** package and check the **keyboard_teleop.launch file**.

Execute in WebShell #1

```
[ ]: roscd turtlebot_teleop
     cd launch
     cat keyboard_teleop.launch
```

WebShell #1 Output

```
[ ]: <launch>
       <!-- turtlebot_teleop_key already has its own built in velocity
     smoother -->
       <node pkg="turtlebot_teleop" type="turtlebot_teleop_key.py"
     name="turtlebot_teleop_keyboard"  output="screen">
           <param name="scale_linear" value="0.5" type="double"/>
           <param name="scale_angular" value="1.5" type="double"/>
           <remap from="turtlebot_teleop_keyboard/cmd_vel" to="/cmd_vel"/>
     <!-- cmd_vel_mux/input/teleop"/-->
       </node>
     </launch>
```

In the launch file, you have some extra tags for setting parameters and remaps. For now, don't worry about those tags and focus on the node tag.

All launch files are contained within a **<launch>** tag. Inside that tag, you can see a **<node>** tag, where we specify the following parameters:

1. **pkg="package_name"** # Name of the package that contains the code of the ROS program to execute
2. **type="python_file_name.py"** # Name of the program file that we want to execute
3. **name="node_name"** # Name of the ROS node that will launch our Python file
4. **output="type_of_output"** # Through which channel you will print the output of the Python file

Create a package

Until now we've been checking the structure of an already-built package... but now, let's create one ourselves.

When we want to create packages, we need to work in a very specific ROS workspace, which is known as **the catkin workspace**. The catkin workspace is the directory in your hard disk where your own **ROS packages must reside** in order to be usable by ROS. Usually, the **catkin workspace** directory is called **catkin_ws**.

Example 1.4
Go to the catkin_ws in your WebShell.

In order to do this, type **roscd** in the shell. You'll see that you are thrown to a **catkin_ws/devel** directory. Since you want to go to the workspace, just type **cd ..** to move up 1 directory. You must end up here in the /**home/user/catkin_ws**.

Execute in WebShell #1

```
[ ]: roscd
     cd ..
     pwd
```

WebShell #1 Output

```
[ ]: user ~ $ pwd
     /home/user/catkin_ws
```

Inside this workspace, there is a directory called **src**. This folder will contain all the packages created. Every time you want to create a new package, you have to be in this directory (**catkin_ws/src**). Type in your WebShell **cd src** in order to move to the source directory.

Execute in WebShell #1

```
[ ]: cd src
```

Now we are ready to create our first package! In order to create a package, type in your WebShell:

Execute in WebShell #1

```
[ ]: catkin_create_pkg my_package rospy
```

This will create inside our **src** directory a new package with some files in it. We'll check this later. Now, let's see how this command is built:

```
[ ]: catkin_create_pkg <package_name> <package_dependecies>
```

The **package_name** is the name of the package you want to create, and the **package_dependencies** are the names of other ROS packages that your package depends on.

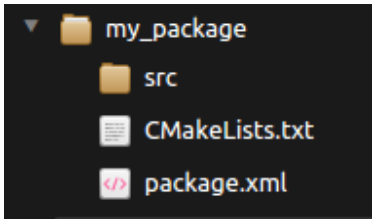
Example 1.5

In order to check that our package has been created successfully, we can use some ROS commands related to packages. For example, let's type:

Execute in WebShell #1

```
[ ]: rospack list
     rospack list | grep my_package
     roscd my_package
```

rospack list: Gives you a list with all of the packages in your ROS system. rospack list | grep my_package: Filters, from all of the packages located in the ROS system, the package named my_package. roscd my_package: Takes you to the location in the Hard Drive of the package, named my_package. You can also see the package created and its contents by just opening it through the IDE (similar to {Figure 1.1})
Fig.1.1 - IDE created package my_package

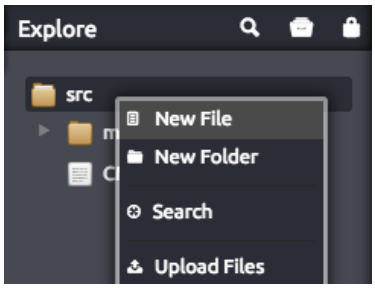


IDE created package

My first ROS program

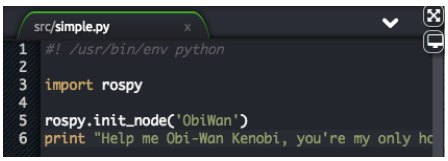
At this point, you should have your first package created... but now you need to do something with it! Let's do our first ROS program!

Example 1.6
1- Create in the **src** directory in my_package a Python file that will be executed. For this exercise, just copy this simple python code simple.py. You can create it directly by **RIGHT clicking** on the IDE on the src directory of your package, selecting New File, and writing the name of the file on the box that will appear.

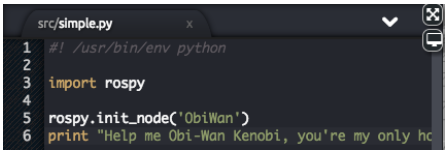


New File Creation

A new Tab should have appeared on the IDE with empty content. Then, copy the content of simple.py into the new file. Finally, press **Ctrl-S** to save your file with the changes. The Tab in the IDE will go from Green to no color (see pictures below).



Unsaved



Saved

Python Program {1.1a-py}: simple.py

```
[ ]: #! /usr/bin/env python

     import rospy

     rospy.init_node('ObiWan')
     print "Help me Obi-Wan Kenobi, you're my only hope"
```

NOTE: If you create your Python file from the shell, it may happen that it's created without execution permissions. If this happens, ROS won't be able to find it. If this is your case, you can give execution permissions to the file by typing the next command: **chmod +x name_of_the_file.py**
2- Create a launch directory inside the package named my_package {Example 1.4}.

Execute in WebShell #1

```
[ ]: roscd my_package
     mkdir launch
```

You can also create it through the IDE.

3- Create a new launch file inside the launch directory.

Execute in WebShell #1

```
[ ]: touch launch/my_package_launch_file.launch
```

You can also create it through the IDE.

4- Fill this launch file as we've previously seen in this course {Example 1.3}.

HINT: You can copy from the turtlebot_teleop package, the keyboard_teleop.launch file and modify it. If you do so, remove the param and remap tags and leave only the node tag, because you don't need those parameters.
The final launch file should be something similar to this my_package_launch_file.launch.

Launch File {1.1-l}: my_package_launch_file.launch
You should have something similar to this in your my_package_launch_file.launch:

NOTE: Keep in mind that in the example below, the Python file in the attribute type is named simple.py. So, if you have named your Python file with a different name, this will be different.

```
[ ]: <launch>
        <!-- My Package launch file -->
        <node pkg="my_package" type="simple.py" name="ObiWan"
    output="screen">
        </node>
    </launch>
```

5- Finally, execute the roslaunch command in the WebShell in order to launch your program.

Execute in WebShell #1

```
[ ]: roslaunch my_package my_package_launch_file.launch
```

Expected Result for Example 1.6
You should see Leia's quote among the output of the roslaunch command.

WebShell #1 Output

```
[ ]: user catkin_ws $ roslaunch my_package my_package_launch_file.launch
    ... logging to /home/user/.ros/log/d29014ac-911c-
    11e6-b306-02f9ff83faab/roslaunch-ip-172-31-30-5-28204.log
    Checking log directory for disk usage. This may take awhile.
    Press Ctrl-C to interrupt
    Done checking log file disk usage. Usage is <1GB.

    started roslaunch server http://ip-172-31-30-5:40504/

    SUMMARY
    ========

    PARAMETERS
     * /rosdistro: kinetic
     * /rosversion: 1.11.20

    NODES
```

```
  /
    ObiWan (my_package/simple.py)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[ObiWan-1]: started with pid [28228]
Help me Obi-Wan Kenobi, you're my only hope
[ObiWan-1] process has finished cleanly
log file: /home/user/.ros/log/d29014ac-911c-
11e6-b306-02f9ff83faab/ObiWan-1*.log
all processes on machine have died, roslaunch will exit
shutting down processing monitor...
... shutting down processing monitor complete
done
```

Sometimes ROS won't detect a new package when you have just created it, so you won't be able to do a roslaunch. In this case, you can force ROS to do a refresh of its package list with the command:

Execute in WebShell #1

```
[ ]: rospack profile
```

Code Explanation {1.1a-py}: simple.py
Although it is a very simple Python script, let's exploan it line by line, to avoid any confusion:

```
[ ]: #! /usr/bin/env python
     # This line will ensure the interpreter used is the first one on your
     environment's $PATH. Every Python file needs
     # to start with this line at the top.

     import rospy # Import the rospy, which is a Python library for ROS.

     rospy.init_node('ObiWan') # Initiate a node called ObiWan

     print "Help me Obi-Wan Kenobi, you're my only hope" # A simple Python
     print
```

Common Issues

From our experience, we've seen that it is a common issue when working with Python scripts in this Course, that users get an error similar to this one:

```
core service [/rosout] found
ERROR: cannot launch node of type [test_pkg/test.py]: can't locate node [test.py] in package [test_pkg]
No processes to monitor
shutting down processing monitor
```

Common error

This error usually appears to users when they create a Python script from the WebShell. It happens because when created from the shell, the Python scripts don't have execution permissions. You can check the permissions of a file using the following command, inside the directory where the file is located at:

Execute in WebShell #1

```
[ ]: ls -la
```

```
user:~/catkin_ws/src/test_pkg/src$ ls -la
total 12
drwxrwxr-x 2 user user 4096 Jan 15 20:13 .
drwxrwxr-x 4 user user 4096 Jan 15 20:15 ..
-rw-rw-r-- 1 user user   82 Jan 15 20:14 test.py
```

Not executable

The first row in the left indicates the permissions of this file. In this case, we have **-rw-rw-r-**. So, you only have **read(r)** and **write(w)** permissions on this file, but not execution permissions (which are represented with an **x**).

To add execution permissions to a file, you can use the following command:

Execute in WebShell #1

```
[ ]: chmod +x name_of_file.py
```

Using this command, you will see that execution permissions are added to the file. Also, the file will appear now in green color.

```
user:~/catkin_ws/src/test_pkg/src$ ls -la
total 12
drwxrwxr-x 2 user user 4096 Jan 15 20:13 .
drwxrwxr-x 4 user user 4096 Jan 15 20:15 ..
-rwxrwxr-x 1 user user   82 Jan 15 20:14 test.py
```

Executable

Doing this, the error shown above will disappear.

ROS Nodes

You've initiated a node in the previous code but... what's a node? ROS nodes are basically programs made in ROS. The ROS command to see what nodes are actually running in a computer

is:

```
[ ]: rosnode list
```

Example 1.7
Type this command in a new shell and look for the node you've just initiated (ObiWan).

Execute in WebShell #1

```
[ ]: rosnode list
```

You can't find it? I know you can't. That's because the node is killed when the Python program ends. Let's change that.

Update your Python file simple.py with the following code:

Python Program {1.1b-py}: simple_loop.py

```
[ ]: #! /usr/bin/env python

     import rospy

     rospy.init_node("ObiWan")
     rate = rospy.Rate(2)                # We create a Rate object of 2Hz
     while not rospy.is_shutdown():      # Endless loop until Ctrl + C
        print "Help me Obi-Wan Kenobi, you're my only hope"
        rate.sleep()                     # We sleep the needed time to
     maintain the Rate fixed above

     # This program creates an endless loop that repeats itself 2 times per
     second (2Hz) until somebody presses Ctrl + C
     # in the Shell
```

Launch your program again using the roslaunch command.

Execute in WebShell #1

```
[ ]: roslaunch my_package my_package_launch_file.launch
```

Now try again in another Web Shell:

Execute in WebShell #2

```
[ ]: rosnode list
```

Can you now see your node?

WebShell #2 Output

```
[ ]: user ~ $ rosnode list
     /ObiWan
     /cmd_vel_mux
     /gazebo
     /mobile_base_nodelet_manager
     /robot_state_publisher
     /rosout
```

In order to see information about our node, we can use the next command:

```
[ ]: rosnode info /ObiWan
```

This command will show us information about all the connections that our Node has.

Execute in WebShell #2

```
[ ]: rosnode info /ObiWan
```

WebShell #2 Output

```
[ ]: user ~ $ rosnode info /ObiWan
     --------------------------------------------------------------------------------
     Node [/ObiWan]
     Publications:
      * /rosout [rosgraph_msgs/Log]

     Subscriptions:
      * /clock [rosgraph_msgs/Clock]

     Services:
      * /ObiWan/set_logger_level
      * /ObiWan/get_loggers


     contacting node http://ip-172-31-30-5:58680/ ...
     Pid: 1215
     Connections:
      * topic: /rosout
         * to: /rosout
         * direction: outbound
         * transport: TCPROS
      * topic: /clock
         * to: /gazebo (http://ip-172-31-30-5:46415/)
         * direction: inbound
         * transport: TCPROS
```

For now, don't worry about the output of the command. You will understand more while going through the next chapters.

Compile a package

When you create a package, you will usually need to compile it in order to make it work. The command used by ROS to compile is the next one:

```
[ ]: catkin_make
```

This command will compile your whole src directory, and it needs to be issued in your catkin_ws directory in order to work. This is MANDATORY. If you try to compile from another directory, it won't work.

Example 1.8
Go to your catkin_ws directory and compile your source folder. You can do this by typing:

Execute in WebShell #1

```
[ ]: roscd; cd ..
     catkin_make
```

Sometimes (for example, in large projects) you will not want to compile all of your packages, but just the one(s) where you've made changes. You can do this with the following command:

```
[ ]: catkin_make --only-pkg-with-deps <package_name>
```

This command will only compile the packages specified and its dependencies.

Try to compile your package named my_package with this command.

Execute in WebShell #1

```
[ ]: catkin_make --only-pkg-with-deps my_package
```

Parameter Server

A Parameter Server is a **dictionary** that ROS uses to store parameters. These parameters can be used by nodes at runtime and are normally used for static data, such as configuration parameters.

To get a list of these parameters, you can type:

```
[ ]: rosparam list
```

To get a value of a particular parameter, you can type:

```
[ ]: rosparam get <parameter_name>
```

And to set a value to a parameter, you can type:

```
[ ]: rosparam set <parameter_name> <value>
```

Example 1.9
To get the value of the '/camera/imager_rate' parameter, and change it to '4.0,' you will have to do the following:

Execute in WebShell #1

```
[ ]: rosparam get /camera/imager_rate
     rosparam set /camera/imager_rate 4.0
     rosparam get /camera/imager_rate
```

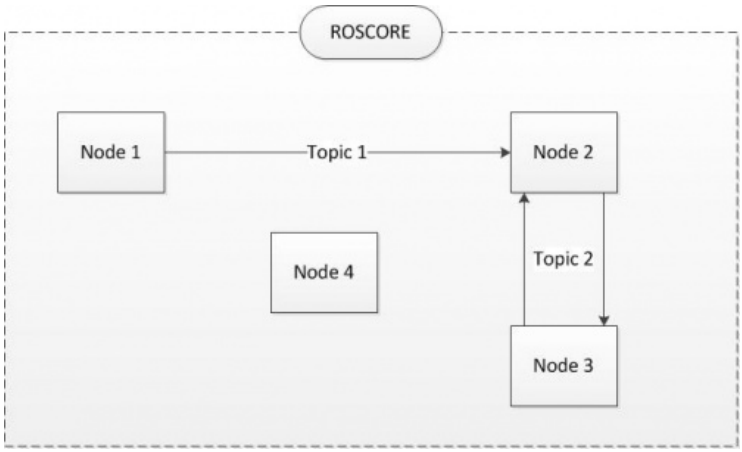
You can create and delete new parameters for your own use, but do not worry about this right now. You will learn more about this in more advanced tutorials

Roscore

In order to have all of this working, we need to have a roscore running. The roscore is the **main process** that manages all of the ROS system. You always need to have a roscore running in order to work with ROS. The command that launches a roscore is:

```
[ ]: roscore
```

Fig.1.2 - ROS Core Diagram



ROS Core Diagram

NOTE: At the platform you are using for this course, when you enter a course it automatically launches a roscore for you, so you don't need to launch one.

Environment Variables

ROS uses a set of Linux system environment variables in order to work properly. You can check these variables by typing:

```
[ ]: export | grep ROS
```

NOTE 1: Depending on your computer, it could happen that you can't type the | symbol directly in your WebShell. If that's the case, just **copy**/**paste** the command by **RIGHT-CLICKING** on the WebShell and select **Paste from Browser**. This feature will allow you to write anything on your WebShell, no matter what your computer configuration is.

```
[ ]: user ~ $ export | grep ROS
     declare -x
     ROSLISP_PACKAGE_DIRECTORIES="/home/user/catkin_ws/devel/share/common-
     lisp"
     declare -x ROS_DISTRO="kinetic"
     declare -x ROS_ETC_DIR="/opt/ros/kinetic/etc/ros"
     declare -x ROS_MASTER_URI="http://localhost:11311"
     declare -x ROS_PACKAGE_PATH="/home/user/catkin_ws/src:/opt/ros/kinetic
     /share:/opt/ros/kinetic/stacks"
     declare -x ROS_ROOT="/opt/ros/kinetic/share/ros"
```

The most important variables are the **ROS_MASTER_URI** and the **ROS_PACKAGE_PATH**.

```
[ ]: ROS_MASTER_URI -> Contains the url where the ROS Core is being
     executed. Usually, your own computer (localhost).
     ROS_PACKAGE_PATH -> Contains the paths in your Hard Drive where ROS
     has packages in it.
```

NOTE 2: At the platform you are using for this course, we have created an alias to display the environment variables of ROS. This alias is **rosenv**. By typing this on your shell, you'll get a list of ROS environment variables. It is important that you know that this is **not an official ROS command**, so you can only use it while working on this platform.

So now... what is ROS?

ROS is basically the framework that allows us to do all that we showed along this chapter. It provides the background to manage all these processes and communications between them... and much, much more!! In this tutorial you've just scratched the surface of ROS, the basic concepts. ROS is an extremely powerful tool. If you dive into our courses you'll learn much more about ROS and you'll find yourself able to do almost anything with your robots!

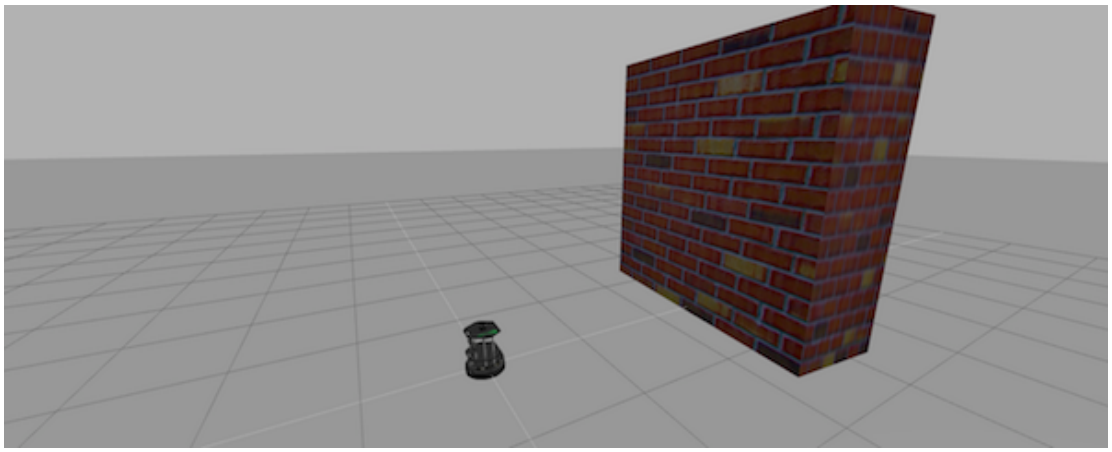
Additional material to learn more:

ROS Packages: http://wiki.ros.org/Packages
Ros Nodes: http://wiki.ros.org/Nodes
Parameter Server: http://wiki.ros.org/Parameter%20Server
Roscore: http://wiki.ros.org/roscore
ROS Environment Variables: http://wiki.ros.org/ROS/EnvironmentVariables

Unit 2. Topics Part1

ROS IN 5 DAYS

Unit 2: Topics

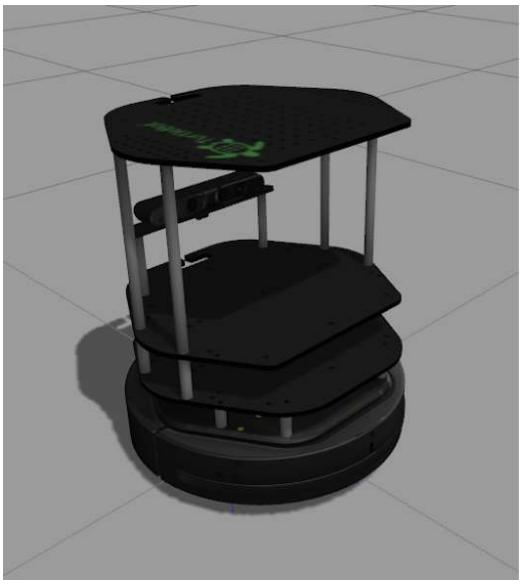


Kobuki

- ROSject Link: http://bit.ly/2LM5WgR
- Package Name: **turtlebot_gazebo**
- Launch File: **main.launch**

Estimated time to completion: 2.5 hours Simulation: Turtlebot What will you learn with this unit?

- What are ROS topics and how to manage them
- What is a publisher and how to create one
- What are topic messages and how they work



Kobuki Robot

Part 1: Publisher

Example 2.1
Execute the following Python code simple_topic_publisher.py by clicking on it and then clicking on the play button on the top right-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it.

Python Program {2.1}: simple_topic_publisher.py

```
[ ]: #! /usr/bin/env python

     import rospy
     from std_msgs.msg import Int32

     rospy.init_node('topic_publisher')
     pub = rospy.Publisher('/counter', Int32, queue_size=1)
     rate = rospy.Rate(2)
     count = Int32()
     count.data = 0

     while not rospy.is_shutdown():
       pub.publish(count)
       count.data += 1
       rate.sleep()
```

Nothing happens? Well... that's not actually true! You have just created a topic named /**counter**, and published through it as an integer that increases indefinitely. Let's check some things.

A topic is like a pipe. **Nodes use topics to publish information for other nodes** so that they can communicate. You can find out, at any time, the number of topics in the system by doing a **rostopic list**. You can also check for a specific topic.

On your webshell, type **rostopic list** and check for a topic named '/counter'.

Execute in WebShell #1

```
[ ]: rostopic list | grep  '/counter'
```

WebShell #1 Output

```
[ ]: user ~ $ rostopic list | grep '/counter'
     /counter
```

Here, you have just listed all of the topics running right now and filtered with the grep command the ones that contain the word /counter. If it appears, then the topic is running as it should.

You can request information about a topic by doing **rostopic info <name_of_topic>**.

Now, type **rostopic info /counter**.

Execute in WebShell #1

```
[ ]: rostopic info /counter
```

WebShell #1 Output

```
[ ]: user ~ $ rostopic info /counter
     Type: std_msgs/Int32

     Publishers:
      * /topic_publisher (http://ip-172-31-16-133:47971/)

     Subscribers: None
```

The output indicates the type of information published (**std_msgs/Int32**), the node that is publishing this information (/**topic_publisher**), and if there is a node listening to that info (None in this case).

Now, type ***rostopic echo /counter*** and check the output of the topic in realtime.

Execute in WebShell #1

```
[ ]: rostopic echo /counter
```

You should see a succession of consecutive numbers, similar to the following:

WebShell #1 Output

```
[ ]: rostopic echo /counter
     data:
     985
     ---
     data:
     986
     ---
     data:
     987
     ---
     data:
     988
     ---
```

Ok, so... what has just happened? Let's explain it in more detail. First, let's crumble the code we've executed. You can check the comments in the code below explaining what each line of the code does:

```
[ ]: #! /usr/bin/env python

     import rospy                                    # Import the Python library
     for ROS
     from std_msgs.msg import Int32                  # Import the Int32 message
     from the std_msgs package

     rospy.init_node('topic_publisher')              # Initiate a Node named
     'topic_publisher'
     pub = rospy.Publisher('counter', Int32)         # Create a Publisher
     object, that will publish on the /counter topic
                                                     #  messages of type Int32

     rate = rospy.Rate(2)                            # Set a publish rate of 2
     Hz
     count = Int32()                                 # Create a var of type
     Int32
     count.data = 0                                  # Initialize 'count'
     variable

     while not rospy.is_shutdown():                  # Create a loop that will
     go until someone stops the program execution
       pub.publish(count)                            # Publish the message
     within the 'count' variable
       count.data += 1                               # Increment 'count'
     variable
       rate.sleep()                                  # Make sure the publish
     rate maintains at 2 Hz
```

So basically, what this code does is to **initiate a node and create a publisher that keeps publishing into the '/counter' topic a sequence of consecutive integers**. Summarizing:

A publisher is a node that keeps publishing a message into a topic. So now... what's a topic?

A topic is a channel that acts as a pipe, where other ROS nodes can either publish or read information. Let's now see some commands related to topics (some of them you've already used).

To get a list of available topics in a ROS system, you have to use the next command:

```
[ ]: rostopic list
```

To read the information that is being published in a topic, use the next command:

```
[ ]: rostopic echo <topic_name>
```

This command will start printing all of the information that is being published into the topic, which sometimes (ie: when there's a massive amount of information, or when messages have a very large structure) can be annoying. In this case, you can read just the last message published into a topic with the next command:

```
[ ]: rostopic echo <topic_name> -n1
```

To get information about a certain topic, use the next command:

```
[ ]: rostopic info <topic_name>
```

Finally, you can check the different options that rostopic command has by using the next command:

```
[ ]: rostopic -h
```

IMPORTANT NOTE
When you have finished with this section of the Notebook, make sure to STOP the previously executed code by selecting the cell with the code and clicking on the Interrupt kernel button at the top right corner of the Notebook. This is very important for doing the Next Unit properly.

IMPORTANT NOTE

Messages

As you may have noticed, topics handle information through messages. There are many different types of messages.

In the case of the code you executed before, the message type was an **std_msgs/Int32**, but ROS provides a lot of different messages. You can even create your own messages, but it is recommended to use ROS default messages when its possible.

Messages are defined in **.msg** files, which are located inside a **msg** directory of a package.

To get information about a message, you use the next command:

```
[ ]: rosmsg show <message>
```

Example 2.2
For example, let's try to get information about the std_msgs/Int32 message. Type the following command and check the output.

Execute in WebShell #1

```
[ ]: rosmsg show std_msgs/Int32
```

WebShell #1 Output

```
[ ]: user ~ $ rosmsg show std_msgs/Int32
     [std_msgs/Int32]:
     int32 data
```

In this case, the **Int32** message has only one variable named **data** of type **int32**. This Int32 message comes from the package **std_msgs**, and you can find it in its **msg** directory. If you want, you can have a look at the Int32.msg file by executing the following command:

```
[ ]: roscd std_msgs/msg/
```

Now you're ready to create your own publisher and make the robot move, so let's go for it!

Exercise 2.1
Create a package with a launch file that launches the code simple_topic_publisher.py.

Modify the code you used previously to publish data to the /cmd_vel topic.

Launch the program and check that the robot moves.

Data for Excercice 2.1

1.- The /cmd_vel topic is the topic used to move the robot. Do a **rostopic info /cmd_vel** in order to get information about this topic, and identify the message it uses. You have to modify the code to use that message.

2.- In order to fill the Twist message, you need to create an instance of the message. In Python, this is done like this: **var = Twist()**
3.- In order to know the structure of the Twist messages, you need to use the ***rosmsg show*** command, with the type of the message used by the topic /**cmd_vel**.

4.- In this case, the robot uses a differential drive plugin to move. That is, the robot can only move linearly in the **x** axis, or rotationaly in the angular **z** axis. This means that the only values that you need to fill in the Twist message are the linear **x** and the angular **z**.

5.- The magnitudes of the Twist message are in m/s, so it is recommended to use values between 0 and 1. For example, 0'5 m/s.

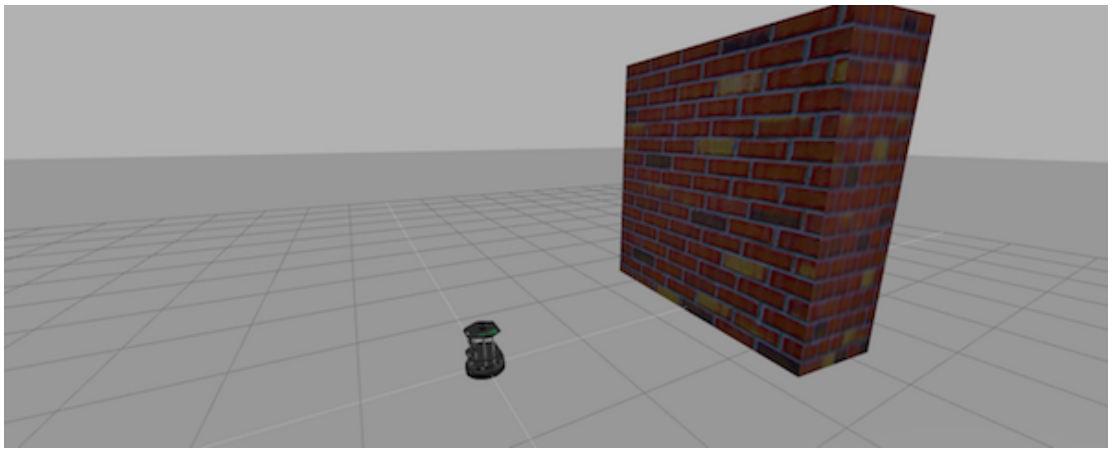
Solution Exercise 2.1
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit3 Services Part1:Topics Part1 Solutions

Unit 2. Topics Part 2

ROS IN 5 DAYS

Unit 2: Topics

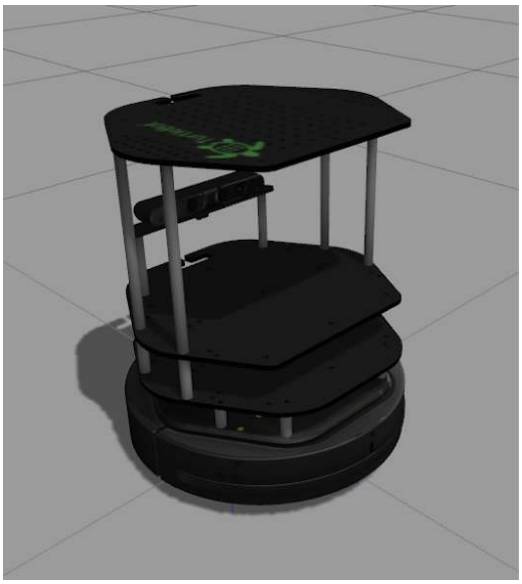


Kobuki

- ROSject Link: http://bit.ly/2LM5WgR
- Package Name: **turtlebot_gazebo**
- Launch File: **main.launch**

Estimated time to completion: 2.5 hours What will you learn with this unit?

- What is a Subscriber and how to create one
- How to create your own message



Kobuki Robot

Part 2: Subscriber

You've learned that a topic is a channel where nodes can either write or read information. You've also seen that you can write into a topic using a publisher, so you may be thinking that there should also be some kind of similar tool to read information from a topic. And you're right! That's called a subscriber. A subscriber is a node that reads information from a topic. Let's execute the next code:

Example 2.2
Execute the following Python code simple_topic_subscriber.py by clicking on it and then clicking on the play button on the top righ-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it.

Python Program {2.2}: simple_topic_subscriber.py

```
[ ]: #! /usr/bin/env python

     import rospy
     from std_msgs.msg import Int32

     def callback(msg):
       print msg.data

     rospy.init_node('topic_subscriber')
     sub = rospy.Subscriber('counter', Int32, callback)
     rospy.spin()
```

What's up? Nothing happened again? Well, that's not actually true... Let's do some checks.

Go to your webshell and type the following:

Execute in WebShell #1

```
[ ]: rostopic echo /counter
```

You should see an output like this:

WebShell #1 Output

```
[ ]: user ~ $ rostopic echo /counter
     WARNING: no messages received and simulated time is active.
     Is /clock being published?
```

And what does this mean? This means that **nobody is publishing into the /counter topic**, so there's no information to be read. Let's then publish something into the topic and see what happens. For that, let's introduce a new command:

```
[ ]: rostopic pub <topic_name> <message_type> <value>
```

This command will publish the message you specify with the value you specify, in the topic you specify.

Open another webshell (leave the one with the ***rostopic echo*** opened) and type the next command:

Execute in WebShell #2

```
[ ]: rostopic pub /counter std_msgs/Int32 5
```

Now check the output of the console where you did the ***rostopic echo*** again. You should see something like this: .

WebShell #1 Output

```
[ ]: user ~ $ rostopic echo /counter
     WARNING: no messages received and simulated time is active.
     Is /clock being published?
     data:
     5
     ---
```

This means that the value you published has been received by your subscriber program (which prints the value on the screen).

Now check the output of the code you've just executed in the IPython notebook (that's the part right down the code). You should see something like this:

```
rospy.init_node('topic_subscr
sub = rospy.Subscriber('counte
rospy.spin()
```

```
5
```

Notebook Output

Before explaining everything with more detail, let's explain the code you executed.

```
[ ]: #! /usr/bin/env python

     import rospy
     from std_msgs.msg import Int32

     def callback(msg):                                    # Define a
     function called 'callback' that receives a parameter
                                                           # named 'msg'

         print msg.data                                    # Print the
     value 'data' inside the 'msg' parameter

     rospy.init_node('topic_subscriber')                   # Initiate a
     Node called 'topic_subscriber'

     sub = rospy.Subscriber('/counter', Int32, callback)   # Create a
     Subscriber object that will listen to the /counter
                                                           # topic and will
     cal the 'callback' function each time it reads
                                                           # something from
     the topic
     rospy.spin()                                          # Create a loop
     that will keep the program in execution
```

So now, let's explain what has just happened. You've basically created a subscriber node that listens to the /counter topic, and each time it reads something, it calls a function that does a print of the msg. Initially, nothing happened since nobody was publishing into the /counter topic, but when you executed the rostopic pub command, you published a message into the /counter topic, so the function has printed the number in the IPython's output and you could also see that message in the rostopic echo output. Now everything makes sense, right?

Now let's do some exercises to put into practice what you've just learned!

Exercise 2.2
Create a package that launches the code. Modify the code in order to print the odometry of the robot.

Data for Exercice 2.2

The odometry of the robot is published by the robot into the /odom topic.

You will need to figure out what message uses the /odom topic, and how the structure of this message is.

Solution Exercise 2.2
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit2 Topics Part2:Topics Part2 Solutions
Exercise 2.3

1. Add to {Exercice 2.2}, a Python file that creates a publisher that indicates the age of the robot.
2. For that, you'll need to create a new message called **Age.msg**. To see how you can do that, have a look at the detailed description How to prepare CMakeLists.txt and package.xml for custom topic message compilation.

Solution Exercise 2.3
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit2 Topics Part2:Topics Part2 Solutions

How to Prepare CMakeLists.txt and package.xml for Custom Topic Message Compilation

Now you may be wondering... in case I need to publish some data that is not an Int32, which type of message should I use? You can use all ROS defined (**rosmsg list**) messages. But, in case none fit your needs, you can create a new one.

In order to create a new message, you will need to do the following steps:

1. Create a directory named 'msg' inside your package
2. Inside this directory, create a file named Name_of_your_message.msg (more information down)
3. Modify CMakeLists.txt file (more information down)
4. Modify package.xml file (more information down)
5. Compile
6. Use in code

For example, let's create a message that indicates age, with years, months, and days.

1) Create a directory msg in your package.

```
[ ]: roscd <package_name>
     mkdir msg
```

2) The Age.msg file must contain this:

```
[ ]: float32 years
     float32 months
     float32 days
```

3) In CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_message_files()
- generate_messages()
- catkin_package()

I. find_package()

This is where all the packages required to COMPILE the messages of the topics, services, and actions go. In package.xml, you have to state them as build_depend.

HINT 1: If you open the CMakeLists.txt file in your IDE, you'll see that almost all of the file is commented. This includes some of the lines you will have to modify. Instead of copying and pasting the lines below, find the equivalents in the file and uncomment them, and then add the parts that are missing.

```
[ ]: find_package(catkin REQUIRED COMPONENTS
            rospy
            std_msgs
            message_generation    # Add message_generation here, after the
     other packages
     )
```

II. add_message_files()

This function includes all of the messages of this package (in the msg folder) to be compiled. The file should look like this.

```
[ ]: add_message_files(
           FILES
           Age.msg
         ) # Dont Forget to UNCOMENT the parenthesis and add_message_files
     TOO
```

III. generate_messages()

Here is where the packages needed for the messages compilation are imported.

```
[ ]: generate_messages(
           std_msgs
     ) # Dont Forget to uncoment here TOO
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the package.xml as exec_depend.

```
[ ]: catkin_package(
     only thing here
     )
```

Summarizing, this is the minimum expression of what is needed for the **CMakelists.txt** file to work:

Note: Keep in mind that the name of the package in the following example is topic_ex, so in your case, the name of the package may be different.

```
[ ]: cmake_minimum_required(VERSION 2.8.3)
     project(topic_ex)


     find_package(catkin REQUIRED COMPONENTS
       std_msgs
       message_generation
     )

     add_message_files(
       FILES
       Age.msg
     )

     generate_messages(
       std_msgs
     )


     catkin_package(
     )
```

```
include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

4) Modify package.xml

Just add these 2 lines to the package.xml file.

```
[ ]: <build_depend>message_generation</build_depend>

     <build_export_depend>message_runtime</build_export_depend>
     <exec_depend>message_runtime</exec_depend>
```

This is the minimum expression of the package.xml
Note: Keep in mind that the name of the package in the following example is topic_ex, so in your case, the name of the package may be different.

```
[ ]: <?xml version="1.0"?>
     <package format="2">
       <name>topic_ex</name>
       <version>0.0.0</version>
       <description>The topic_ex package</description>


       <maintainer email="user@todo.todo">user</maintainer>

       <license>TODO</license>

       <buildtool_depend>catkin</buildtool_depend>
       <build_depend>rospy</build_depend>
       <build_depend>std_msgs</build_depend>
       <build_depend>message_generation</build_depend>
       <build_export_depend>rospy</build_export_depend>
       <exec_depend>rospy</exec_depend>
       <build_export_depend>std_msgs</build_export_depend>
       <exec_depend>std_msgs</exec_depend>
       <build_export_depend>message_runtime</build_export_depend>
       <exec_depend>message_runtime</exec_depend>
       <export>

       </export>
     </package>
```

5) Now you have to compile the msgs. To do this, you have to type in a WebShell:

Execute in WebShell #1

```
[ ]: roscd; cd ..
     catkin_make
     source devel/setup.bash
```

VERY IMPORTANT: When you compile new messages, there is still an extra step before you can use the messages. You have to type in the Webshell, in the **catkin_ws** directory, the following command: **source devel/setup.bash**. This executes this bash file that sets, among other things, the newly generated messages created through the catkin_make. If you don't do this, it might give you a python import error, saying it doesn't find the message generated.
HINT 2: To verify that your message has been created successfully, type in your webshell rosmsg show Age. If the structure of the Age message appears, it will mean that your message has been created successfully and it's ready to be used in your ROS programs.

Execute in WebShell #1

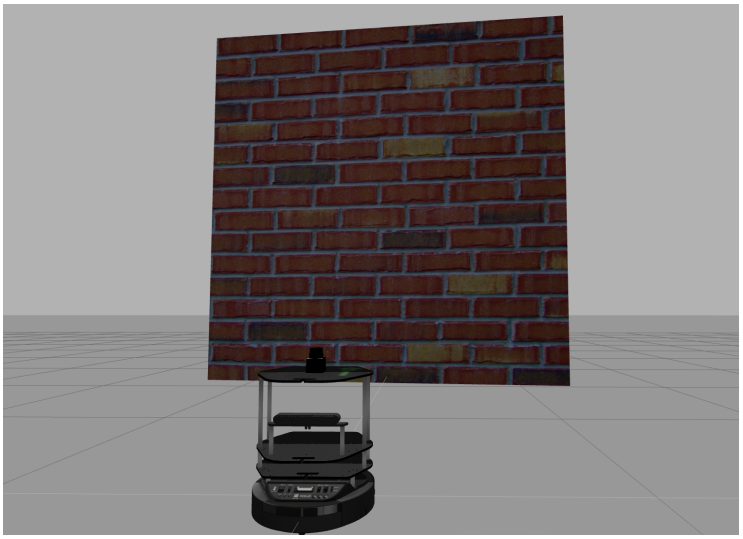
```
[ ]: rosmsg show Age
```

WebShell #1 Output

```
[ ]: user ~ $ rosmsg show Age
     [topic_ex/Age]:
     float32 years
     float32 months
     float32 days
```

WARNING
There is an issue in ROS that could give you problems when importing msgs from the msg directory. If your package has the same name as the Python file that does the import of the msg, this will give an error saying that it doesn't find the msg element. This is due to the way Python works. Therefore, you have to be careful to not name the Python file exactly the same as its parent package. Example: Package name = "my_package" Python file name = "my_package.py" This will give an import error because it will try to import the messagefrom the my_package.py file, from a directory .msg that doesn't exists.

Topics Quiz



Tultlebot2

With all you've learned during this course, you're now able to do a small Quiz to put everything together. Subscribers, Publisher, Messages... you will need to use all of this concepts in order to succeed!

For evaluating this Quiz, we will ask you to perform different tasks. For each task, very **specific instructions** will be provided: name of the package, names of the launch files and Python scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions, since they will allow our automated correction system to properly test your Quiz, and assign a score to it. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

In this Quiz, you will create a code to make the robot avoid the wall that is in front of him. To help you achieve this, let's divide the project down into smaller units:

1. Create a Publisher that writes into the /**cmd_vel** topic in order to move the robot.
2. Create a Subscriber that reads from the /**kobuki/laser/scan topic**. This is the topic where the laser publishes its data.
3. Depending on the readings you receive from the laser's topic, you'll have to change the data you're sending to the /cmd_vel topic, in order to avoid the wall. This means, use the values of the laser to decide.

HINT 1: The data that is published into the /kobuki/laser/scan topic has a large structure. For this project, you just have to pay attention to the 'ranges' array.

Execute in WebShell #1

```
[ ]: rosmsg show sensor_msgs/LaserScan
```

WebShell #1 Output

```
[ ]: user ~ $ rosmsg show sensor_msgs/LaserScan
     std_msgs/Header header
       uint32 seq
       time stamp
       string frame_id
     float32 angle_min
     float32 angle_max
     float32 angle_increment
     float32 time_increment
     float32 scan_time
     float32 range_min
     float32 range_max
     float32[] ranges <-- Use only this one
     float32[] intensities
```

HINT 2: The 'ranges' array has a lot of values. The ones that are in the middle of the array represent the distances that the laser is detecting right in front of him. This means that the values in the middle of the array will be the ones that detect the wall. So in order to avoid the wall, you just have to read these values.

HINT 3: The laser has a range of 30m. When you get readings of values around 30, it means that the laser isn't detecting anything. If you get a value that is under 30, this will mean that the laser is detecting some kind of obstacle in that direction (the wall).

HINT 4: The scope of the laser is about 180 degrees from right to left. This means that the values at the beginning and at the end of the 'ranges' array will be the ones related to the readings on the sides of the laser (left and right), while the values in the middle of the array will be the ones related to the front of the laser.

Specifications

- The name of the package where you'll place all the code related to the Quiz will be **topics_quiz**.
- The name of the launch file that will start your program will be **topics_quiz.launch**.
- The name of the ROS node that will be launched by your program will be **topics_quiz_node**.

Quiz Correction
When you have finished the Quiz, you can correct it in order to get a Mark. For that, just click on the following button at the top of this Notebook.
IMPORTANT
Quizes can only be done once. This means that, once you correct your Quiz, and get a score for it, you won't be able to do it again and improve your score. So, be sure enough when you decide to correct your Quiz!

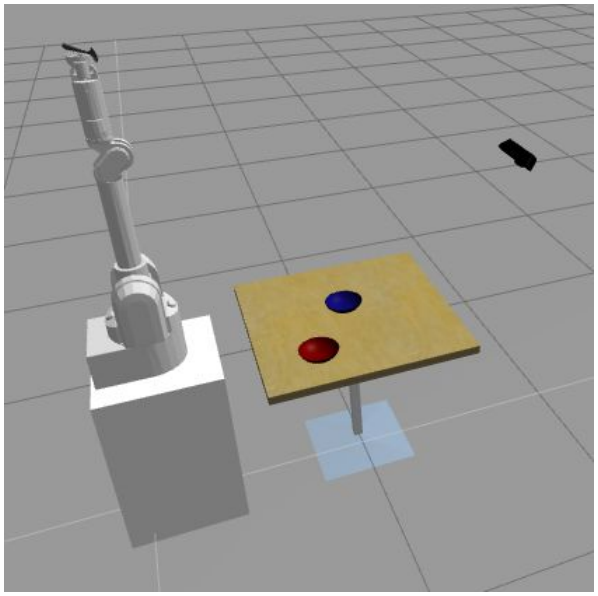
Additional material to learn more

ROS Topics: http://wiki.ros.org/Topics
ROS Messages: http://wiki.ros.org/Messages
msg Files: http://wiki.ros.org/msg
Publisher and Subscriber 1: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29
Publisher and Subscriber 2: http://wiki.ros.org/ROS/Tutorials/ExaminingPublisherSubscriber

Unit 3. Services in ROS Part 1

ROS IN 5 DAYS

Unit 3: Services in ROS



Iri Wam Robot

- ROSject Link: http://bit.ly/2LMWiKN
- Package Name: **iri_wam_gazebo**
- Launch File: **main.launch**

Estimated time to completion: 2.5 hours What will you learn with this unit?

- What a service is
- How to manage services of a robot
- How to call a service

Part 1

Congratulations! You now know **75%** of ROS-basics!

The reason is that, with topics, you can do more or less whatever you want and need for your astromech droid. Many ROS packages only use topics and have the work perfectly done.

Then why do you need to learn about **services**?

Well, that's because for some cases, topics are insufficient or just too cumbersome to use. Of course, you can destroy the Death Star with a stick, but you will just spend ages doing it. Better tell Luke SkyWalker to do it for you, right? Well, it's the same with services. They just make life easier.

Topics - Services - Actions

To understand what services are and when to use them, you have to compare them with topics and actions.

Imagine you have your own personal BB-8 robot. It has a laser sensor, a face-recognition system, and a navigation system. The laser will use a **Topic** to publish all of the laser readings at 20hz. We use a topic because we need to have that information available all the time for other ROS systems, such as the navigation system.

The Face-recognition system will provide a **Service**. Your ROS program will call that service and **WAIT** until it gives you the name of the person BB-8 has in front of it.

The navigation system will provide an **Action**. Your ROS program will call the action to move the robot somewhere, and **WHILE** it's performing that task, your program will perform other tasks, such as complain about how tiring C-3PO is. And that action will give you **Feedback** (for example: distance left to the desired coordinates) along the process of moving to the coordinates.

So... What's the difference between a **Service** and an **Action**?

Services are Synchronous. When your ROS program calls a service, your program can't continue until it receives a result from the service. Actions are Asynchronous. It's like launching a new thread. When your ROS program calls an action, your program can perform other tasks while the action is being performed in another thread.

Conclusion: Use services when your program can't continue until it receives the result from the service.

Services Introduction

Enough talk for now, let's go play with a robot and launch a prepared demo!

Example 3.1
Go to the WebShell and do the following:

Execute in WebShell #1

```
[ ]: roslaunch iri_wam_aff_demo start_demo.launch
```

This will make the Wam robot-arm of the simulation move. You should get something similar to this:

What did you do just now?

The launch file has launched two nodes (Yes! You can launch more than one node with a single launch file):

- /iri_wam_reproduce_trajectory
- /iri_wam_aff_demo

The first node provides the /**execute_trajectory** service. This is the node that contains the **service**. The second node, performs calls to that service. When the service is called, the robot will execute a given trajectory.

Let's learn more about services.

Example 3.2
Let's see a list of the available services in the Wam robot. For that, open another shell. **You have to leave the start_demo.launch running, otherwise the services won't be there to see them.** Execute the following command in a different shell from the one that has the roslaunch **start_demo.launch** running:

Execute in WebShell #2

```
[ ]: rosservice list
```

You should see something like the following image, listing all the services available:

WebShell #2 Output

```
[ ]: user ~ $ rosservice list
     /camera/rgb/image_raw/compressed/set_parameters
     /camera/rgb/image_raw/compressedDepth/set_parameters
     /camera/rgb/image_raw/theora/set_parameters
     /camera/set_camera_info
```

```
/camera/set_parameters
/execute_trajectory
/gazebo/apply_body_wrench
...
```

WARNING: If the /execute_trajectory server is not listed, maybe that's because you stopped the start_demo.launch. If that is the case, launch it again and check for the service.

There are a few services, aren't there? Some refer to the simulator system (/gazebo/...), and others refer to the Kinect Camera (/camera/...) or are given by the robot himself (/iri_wam/...). You can see how the service /execute_trajectory is listed there.

You can get more information about any service by issuing the following command:

```
[ ]: rosservice info /name_of_your_service
```

Execute the following command to know more about the service /**execute_trajectory**.

Execute in WebShell #2

```
[ ]: rosservice info /execute_trajectory
```

WebShell #2 Output

```
[ ]: user ~ $ rosservice info /execute_trajectory
     Node: /iri_wam_reproduce_trajectory
     URI: rosrpc://ip-172-31-17-169:35175
     Type: iri_wam_reproduce_trajectory/ExecTraj
     Args: file
```

Here you have two relevant parts of data.

- **Node**: It states the node that provides (has created) that service.
- **Type**: It refers to the kind of message used by this service. It has the same structure as topics do. It's always made of **package_where_the_service_message_is_defined** / **Name_of_the_File_where_Service_message_is_defined**. In this case, the package is **iri_wam_reproduce_trajectory**, and the file where the Service Message is defined is called **ExecTraj**.
- **Args**: Here you can find the arguments that this service takes when called. In this case, it only takes a **trajectory file path** stored in the variable called **file**.

Want to know how this /execute_trajectory service is started?

Here you have an example on how to check the **start_demo.launch** file through WebShell.

Example 3.3

Do you remember how to go directly to a package and where to find the launch files?

Execute in WebShell #2

```
[ ]: roscd iri_wam_aff_demo
     cd launch/
     cat start_demo.launch
```

You should get something like this:

```
[ ]: <launch>

       <include file="$(find
     iri_wam_reproduce_trajectory)/launch/start_service.launch"/>

       <node pkg ="iri_wam_aff_demo"
             type="iri_wam_aff_demo_node"
             name="iri_wam_aff_demo"
             output="screen">
       </node>

     </launch>
```

Some interesting things here: 1) The first part of the launch file calls another launch file called **start_service.launch**. That launch file starts the node that provides the /execute_trajectory service. Note that it's using a special ROS launch file function to find the path of the package given.

```
[ ]: <include file="$(find
     package_where_launch_is)/launch/my_launch_file.launch"/>
```

2) The second part launches a node just as you learned in the ROS Basics Unit. That node is the one that will call the /execute_trajectory service in order to make the robot move.

3) The second node is not a Python node, but a cpp compiled (binary) one. You can build ROS programs either in Cpp or Python. This course focuses on Python.

```
[ ]: <node pkg ="package_where_cpp_is"
           type="name_of_binary_after_compiling_cpp"
           name="name_of_the_node_initialised_in_cpp"
           output="screen">
     </node>
```

How to call a service

You can call a service manually from the console. This is very useful for testing and having a basic idea of how the service works.

```
[ ]: rosservice call /the_service_name TAB-TAB
```

Info: TAB-TAB means that you have to quickly press the TAB key twice. This will autocomplete the structure of the Service message to be sent for you. Then, you only have to fill in the values.

Example 3.4
Let's call the service with the name /**gazebo**/**delete_model** by issuing the following command.

Execute in WebShell #2

```
[ ]: rosservice call /gazebo/delete_model [TAB]+[TAB]
```

When you [TAB]+[TAB], an extra element appears. ROS autocompletes with the structure needed to input/request the service. In this case, it gives the following structure: "model_name: 'the_name_of_the_object_you_want_to_delete' "
The /**gazebo**/**delete_model** Service is a service provided by the simulator that allows you **to delete any object in the scene**.

Use that service to remove the table. The model_name of the table in the simulation is **cafe_table**.

Execute in WebShell #2

```
[ ]: rosservice call /gazebo/delete_model "model_name: 'cafe_table'"
```

Did it dissapear? You should have seen something similar to this:

Now, if you want to do this with any object of the simulation, you will have to learn how to get the list of objects in the scene. Do the following:

Execute in WebShell #2

```
[ ]: rostopic echo /gazebo/model_states -n1
```

You should get a list similar to this one:

```
[ ]: user:~$ rostopic echo /gazebo/model_states -n1
     name: ['ground_plane', 'kinect', 'bowl_2', 'bowl_1', 'unit_box_1',
     'iri_wam']
     pose:
          ...(the poses of each one of the models in the list in order)
```

But don't get too excited deleting objects or you'll end up without a robot.

But how do you interact with a service programatically?

Example 3.5
Execute the following Python code simple_service_client.py by clicking on it and then clicking on the play button on the top right-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it.

What do you think it will do?

Python Program {3.5}: simple_service_client.py

```
[ ]: #! /usr/bin/env python

     import rospy
     # Import the service message used by the service /gazebo/delete_model
     from gazebo_msgs.srv import DeleteModel, DeleteModelRequest
     import sys

     # Initialise a ROS node with the name service_client
     rospy.init_node('service_client')
     # Wait for the service client /gazebo/delete_model to be running
     rospy.wait_for_service('/gazebo/delete_model')
     # Create the connection to the service
     delete_model_service = rospy.ServiceProxy('/gazebo/delete_model',
     DeleteModel)
     # Create an object of type DeleteModelRequest
     delete_model_object = DeleteModelRequest()
     # Fill the variable model_name of this object with the desired value
     delete_model_object.model_name = "bowl_1"
     # Send through the connection the name of the object to be deleted by
     the service
     result = delete_model_service(delete_model_object)
     # Print the result given by the service called
     print result
```

You should have seen the bowl_1 disappear after executing this snippet of code.

How to know the structure of the service message used by the service?

Example 3.6
You can do a **rosservice info** to know the type of service message that it uses.

```
[ ]: rosservice info /name_of_the_service
```

This will return you the **name_of_the_package/Name_of_Service_message**
Then, you can explore the structure of that service message with the following command:

```
[ ]: rossrv show name_of_the_package/Name_of_Service_message
```

Execute the following command to see what is the service message used by the /**gazebo**/**delete_model** service:

Execute in WebShell #2

```
[ ]: rosservice info /gazebo/delete_model
```

You should see something like this:

WebShell #2 Output

```
[ ]: Node: /gazebo
     URI: rosrpc://ip-172-31-35-239:50272
     Type: gazebo_msgs/DeleteModel
     Args: model_name
```

Now, execute the following command to see the structure of the message **DeleteModel**:

Execute in WebShell #2

```
[ ]: rossrv show gazebo_msgs/DeleteModel
```

You should see something like this:

WebShell #2 Output

```
[ ]: user catkin_ws $ rossrv show gazebo_msgs/DeleteModel
     string model_name
     ---
     bool success
     string status_message
```

Does it seem familiar? It should, because it's the same structure as the Topics messages, with some addons.

1- Service message files have the extension .srv. Remember that Topic message files have the extension .msg

2- Service message files are defined inside a srv directory. Remember that Topic message files are defined inside a msg directory. You can type the following command to check it.

Execute in WebShell #2

```
[ ]: roscd gazebo_msgs; ls srv
```

3- Service messages have TWO parts: REQUEST
—
RESPONSE
In the case of the DeleteModel service, **REQUEST** contains a string called **model_name** and **RESPONSE** is composed of a boolean named **success**, and a string named **status_message**. The Number of elements on each part of the service message can vary depending on the service needs. You can even put none if you find that it is irrelevant for your service. The important part of the message is the three dashes —, because they define the file as a Service Message.

Summarizing:

The **REQUEST** is the part of the service message that defines **HOW you will do a call** to your service. This means, what variables you will have to pass to the Service Server so that it is able to complete its task.

The **RESPONSE** is the part of the service message that defines **HOW your service will respond** after completing its functionality. If, for instance, it will return an string with a certaing message saying that everything went well, or if it will return nothing, etc...
Exercise 3.1

- First of all, create a package to place all the future code. For better future reference, you can call it **unit_3_services**, with dependencies **rospy** and **iri_wam_reproduce_trajectory**.
- Create a launch called **my_robot_arm_demo.launch**, that starts the /**execute_trajectory** service. As explained in the Example 3.3, this service is launched by the launch file **start_service.launch**, which is in the package **iri_wam_reproduce_trajectory**.
- Get information of what type of service message does this /**execute_trajectory** service uses, as explained in Example 3.6.
- Make the robotic arm move following a trajectory, which is specified in a file. Modify the previous code of Example 3.5, which called the /**gazebo/delete_model** service, to call now the /**execute_trajectory** service instead. The new Python file could be called **exercise_3_1.py**, for future reference.
- Here you have the code necessary to get the path to the trajectory files based on the package where they are. Here, the trajectory file **get_food.txt** is selected, but you can use any of the available in the **config** folder of the **iri_wam_reproduce_trajectory** package.

```
[ ]: import rospkg
     rospack = rospkg.RosPack()
     # This rospack.get_path() works in the same way as $(find
     name_of_package) in the launch files.
     traj = rospack.get_path('iri_wam_reproduce_trajectory') +
     "/config/get_food.txt"
```

- Modify the main launch file **my_robot_arm_demo.launch**, so that now it also launches the Python code you have just created in **exercise_3_1.py**.
- Finally, execute the **my_robot_arm_demo.launch** file and see how the robot performs the trajectory.

Solution Exercise 3.1
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit3 Services Part1:solutions_unit3_services_part1

Summary

Services provide functionality to other nodes. If a node knows how to delete an object on the simulation, it can provide that functionality to other nodes through a service call, so they can call the service when they need to delete something. Services allow the specialization of nodes (each node specializes in one thing).

Unit 3. Services in ROS Part 2

ROS IN 5 DAYS

Unit 3: Services in ROS

BB8 Robot

- ROSject Link: http://bit.ly/2LPIBeb
- Package Name: **bb_8_gazebo**
- Launch File: **main.launch**

Estimated time to completion: 3 hours What will you learn with this unit?

- How to give a service
- Create your own service server message

Part 2: How to give a Service
Until now, you have just been calling services that other nodes provided. But now, you are going to create your own service!

Example 3.7
Execute the following Python code simple_service_server.py by clicking on it and then clicking on the play button on the top right-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it.

Python Program {3.7}: simple_service_server.py

```
[ ]: #! /usr/bin/env python

     import rospy
     from std_srvs.srv import Empty, EmptyResponse # you import the service
     message python classes generated from Empty.srv.


     def my_callback(request):
         print "My_callback has been called"
         return EmptyResponse() # the service Response class, in this case
     EmptyResponse
         #return MyServiceResponse(len(request.words.split()))

     rospy.init_node('service_server')
     my_service = rospy.Service('/my_service', Empty , my_callback) #
     create the Service called my_service with the defined callback
     rospy.spin() # maintain the service open.
```

Did something happen?

Of course not! At the moment, you have just created and started the Service Server. So basically, you have made this service available for anyone to call it.

This means that if you do a **rosservice list**, you will be able to visualize this service on the list of available services.

Execute in WebShell #1

```
[ ]: rosservice list
```

On the list of all available services, you should see the /**my_service** service.

```
[ ]: /base_controller/command_select
     /bb8/camera1/image_raw/compressed/set_parameters
     /bb8/camera1/image_raw/compressedDepth/set_parameters
     /bb8/camera1/image_raw/theora/set_parameters
     ...
     /my_service
     ...
```

Now, you have to actually **CALL** it. So, call the /**my_service** service manually. Remember the calling structure discussed in the previous chapter and don't forget to TAB-TAB to autocomplete the structure of the Service message.

Execute in WebShell #1

```
[ ]: rosservice call /my_service [TAB]+[TAB]
```

Did it work? You should've seen the message '**My callback has been called**' printed at the output of the cell with the Python code. Great!

```
rospy.init_node('service_server')
my_service = rospy.Service('/my_service',
rospy.spin() # mantain the service open.
```

```
My_callback has been called
```

INFO: Note that, in the example, there is a commented line in the **my_callback** function. This gives you an example of how you would access the request given by the caller of your service. It's always **request.variables_in_the_request_part_of_srv_message**.

So, for instance, let's do a flashback to the previous chapter. Do you remember Example 3.5, where you had to perform calls to a service in order to delete an object in the simulation? Well, for that case, you were passing the name of the object to delete to the Service Server in a variable called **model_name**. So, if you wanted to access the value of that **model_name** variable in the Service Server, you would have to do it like this:

```
[ ]: request.model_name
```

Quite simple, right?

That commented line also shows you what type of object you should return. Normally, the **Response** Python class is used. It always has the structure **name_of_the_messageResponse()**. That's why for the example code shown above, since it uses the **Empty** service message, the type of object that returns is **EmptyResponse()**. But, if your service uses another type of message, let's say one that is called **MyServiceMessage**, then the type of object that you would return would be **MyServiceMessageResponse()**.

Exercise 3.2

- The objective of Exercise 3.2 is to create a service that, when called, will make BB8 robot move in a circle-like trajectory.
- You can work on a new package or use the one you created for Exercise 3.1, called **unit_3_services**.
- Create a Service Server that accepts an Empty service message and activates the circle movement. This service could be called /**move_bb8_in_circle**. You will place the necessary code into a new Python file named **bb8_move_in_circle_service_server.py**. You can use the Python file simple_service_server.py as an example.
- Create a launch file called **start_bb8_move_in_circle_service_server.launch**. Inside it, you have to start a node that launches the **bb8_move_in_circle_service_server.py** file.
- Launch **start_bb8_move_in_circle_service_server.launch** and check that, when called through the WebShell, BB-8 moves in a circle.
- Now, create a new Python file called **bb8_move_in_circle_service_client.py** that calls the service /**move_bb8_in_circle**. Remember how it was done in the previous chapter: **Services Part 1**. Then, generate a new launch file called **call_bb8_move_in_circle_service_server.launch** that executes the code in the **bb8_move_in_circle_service_client.py** file.
- Finally, when you launch this **call_bb8_move_in_circle_service_server.launch** file, BB-8 should move in a circle.

Solution Exercise 3.2
Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit3 Services Part2: solutions_unit3_services_part2
How to create your own service message
So, what if none of the service messages that are available in ROS fit your needs? Then, you create your own message, as you did with the Topic messages. In order to create a service message, you will have to follow the next steps:

Example 3.8

1) Create a package like this:

Execute in WebShell #1

```
[ ]: roscd;cd ..;cd src
     catkin_create_pkg my_custom_srv_msg_pkg rospy
```

2) Create your own Service message with the following structure. You can put as many variables as you need, of any type supported by ROS: ROS Message Types. Create a **srv** folder inside your package , as you did with the topics **msg** folder. Then, inside this **srv** folder, create a file called **MyCustomServiceMessage.srv**. You can create with the IDE or the WebShell, as you wish.

Execute in WebShell #1

```
[ ]: roscd my_custom_srv_msg_pkg/
     mkdir srv
     vim srv/MyCustomServiceMessage.srv
```

You can also create the **MyCustomServiceMessage.srv** through the IDE, if you don't feel confortable with vim.

The MyCustomServiceMessage.srv could be something like this:

```
[ ]: int32 duration    # The number of times BB-8 has to execute the square
     movement when the service is called
     ---
     bool success          # Did it achieve it?
```

How to Prepare CMakeLists.txt and package.xml for Custom Service Compilation

You have to edit two files in the package similarly to how we explained for Topics:

- CMakeLists.txt
- package.xml

Modification of CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_service_files()
- generate_messages()
- catkin_package()

I. find_package()

All the packages needed to COMPILE the messages of topics, services, and actions go here. It's only getting its paths, and not really importing them to be used in the compilation. The same packages you write here will go in package.xml, stating them as build_depend.

```
[ ]: find_package(catkin REQUIRED COMPONENTS
       std_msgs
       message_generation
     )
```

II. add_service_files()

This function contains a list of all of the service messages defined in this package (defined in the srv folder). For our example:

```
[ ]: add_service_files(
       FILES
       MyCustomServiceMessage.srv
     )
```

III. generate_messages()

Here is where the packages needed for the service messages compilation are imported.

```
[ ]: generate_messages(
       std_msgs
     )
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the **package.xml** file as **<exec_depend>**.

```
[ ]: catkin_package(
           rospy
     )
```

Once you're done, you should have something similar to this:

```
[ ]: cmake_minimum_required(VERSION 2.8.3)
     project(my_custom_srv_msg_pkg)


     ## Here is where all the packages needed to COMPILE the messages of
     topics, services and actions go.
     ## It's only getting its paths, and not really importing them to be
     used in the compilation.
     ## It's only for further functions in CMakeLists.txt to be able to
     find those packages.
     ## In package.xml you have to state them as build
     find_package(catkin REQUIRED COMPONENTS
       std_msgs
```

```
  message_generation
)

## Generate services in the 'srv' folder
## In this function will be all the action messages of this package (
in the action folder ) to be compiled.
## You can state that it gets all the actions inside the action
directory: DIRECTORY action
## Or just the action messages stated explicitly: FILES
my_custom_action.action
## In your case you only need to do one of two things, as you wish.
add_service_files(
  FILES
  MyCustomServiceMessage.srv
)

## Here is where the packages needed for the action messages
compilation are imported.
generate_messages(
  std_msgs
)

## State here all the packages that will be needed by someone that
executes something from your package.
## All the packages stated here must be in the package.xml as
exec_depend
catkin_package(
)


include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

Modification of package.xml:

1. Add all of the packages needed to compile the messages. In this case, you only need to add the **message_generation**. You will have to import those packages as **<build_depend>**.
2. On the other hand, if you need a package for the execution of the programs inside your package, you will have to import those packages as **<exec_depend>**.

In this case, you will only need to add these 2 lines to your **package.xml** file:

```
[ ]: <build_depend>message_generation</build_depend>

     <build_export_depend>message_runtime</build_export_depend>
     <exec_depend>message_runtime</exec_depend>
```

So, at the end, you should have something similar to this:

```
[ ]: <?xml version="1.0"?>
     <package format="2">
       <name>my_custom_srv_msg_pkg</name>
       <version>0.0.0</version>
       <description>The my_custom_srv_msg_pkg package</description>

       <maintainer email="user@todo.todo">user</maintainer>

       <license>TODO</license>

       <buildtool_depend>catkin</buildtool_depend>
       <build_depend>rospy</build_depend>
       <build_depend>std_msgs</build_depend>
       <build_depend>message_generation</build_depend>
       <build_export_depend>rospy</build_export_depend>
       <exec_depend>rospy</exec_depend>
       <build_export_depend>std_msgs</build_export_depend>
       <exec_depend>std_msgs</exec_depend>
       <build_export_depend>message_runtime</build_export_depend>
       <exec_depend>message_runtime</exec_depend>

       <export>
       </export>
     </package>
```

Once you're done, compile your package and source the newly generated messages:

```
[ ]: roscd;cd ..
     catkin_make
     source devel/setup.bash
```

Important!! When you compile new messages through catkin_make, there is an extra step that needs to be done. You have to type in the WebShell, in the **catkin_ws** directory, the following command: **source devel/setup.bash**. This command executes the bash file that sets, among other things, the newly generated messages created with **catkin_make**. If you don't do this, it might give you a Python import error, saying that it doesn't find the message generated.
You should see among all the messages something similar to:

Generating Python code from SRV my_custom_srv_msg_pkg/MyCustomServiceMessage
To check that you have the new service message in your system, and ready to be used, type the following:

Execute in WebShell #1

```
[ ]: rossrv list | grep MyCustomServiceMessage
```

It should output something like:

WebShell #1 Output

```
[ ]: user ~ $ rossrv list | grep MyCustomServiceMessage
     my_custom_srv_msg_pkg/MyCustomServiceMessage
```

That's it! You have created your own Service Message. Now, create a Service Server that uses this type of message.

It could be something similar to this:

Python Program {3.3}: custom_service_server.py

```
[ ]: #! /usr/bin/env python

     import rospy
     from my_custom_srv_msg_pkg.srv import MyCustomServiceMessage,
     MyCustomServiceMessageResponse # you import the service message python
     classes
                        # generated from MyCustomServiceMessage.srv.


     def my_callback(request):

         print "Request Data==> radius="+str(request.radius)+", repetitions
     = "+str(request.repetitions)
         my_response = MyCustomServiceMessageResponse()
         if request.radius > 5.0:
             my_response.success = True
         else:
             my_response.success = False
         return  my_response # the service Response class, in this case
     MyCustomServiceMessageResponse

     rospy.init_node('service_client')
     my_service = rospy.Service('/my_service', MyCustomServiceMessage ,
     my_callback) # create the Service called my_service with the defined
     callback
     rospy.spin() # maintain the service open.
```

Exercise 3.3

- Create a new Python file called **bb8_move_custom_service_server.py**. Inside this file, modify the code you used in **Exercise 3.2**, which contained a Service Server that accepted an Empty Service message to activate the circle movement. This new service will be called **/move_bb8_in_circle_custom**. This new service will have to be called through a custom service message. The structure of this custom message is presented below:

```
[ ]: int32 duration     # The time (in seconds) during which BB-8 will keep
     moving in circles
     ---
     bool success       # Did it achieve it?
```

- Use the data passed to this new /**move_bb8_in_circle_custom** to change the BB-8 behavior. During the specified **duration** time, BB-8 will keep moving in circles. Once this time has ended, BB-8 will then stop its movement and the Service Server will return a **True** value (in the **success** variable). Keep in mind that even after BB-8 stops moving, there might still be some rotation on the robot, due to inertia.
- Create a new launch file called **start_bb8_move_custom_service_server.launch** that launches the new **bb8_move_custom_service_server.py** file.
- Test that when calling this new /**move_bb8_in_circle_custom** service, BB-8 moves accordingly.
- Create a new python code called **call_bb8_move_custom_service_server.py** that calls the service /**move_bb8_in_circle_custom**. Remember how it was done in **Unit 3 Services Part 1**. Then, generate a new launch file called **call_bb8_move_custom_service_server.launch** that executes the **call_bb8_move_custom_service_server.py** through a node.

Solution Exercise 3.3
Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit3 Services Part2: solutions_unit3_services_part2

Summary

Let's do a quick summary of the most important parts of **ROS Services**, just to try to put everything in place.

A **ROS Service** provides a certain functionality of your robot. A ROS Service is composed of 2 parts:

- **Service Server**: This is what **PROVIDES** the functionality. Whatever you want your Service to do, you have to place it in the Service Server.
- **Service Client**: This is what **CALLS** the functionality provided by the Service Server. That is, it CALLS the Service Server.

ROS Services use a special service message, which is composed of 2 parts:

- **Request**: The request is the part of the message that is used to CALL the Service. Therefore, it is sent by the Service Client to the Service Server.
- **Response**: The response is the part of the message that is returned by the Service Server to the Service Client, once the Service has finished.

ROS Services are synchronous. This means that whenever you CALL a Service Server, you have to wait until the Service has finished (and returns a response) before you can do other stuff with your robot.

Services Quiz

For evaluation, this quiz will ask you to perform different tasks. For each task, very **specific instructions** will be provided: name of the package, names of the launch files and Python scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions, since they will allow our automated correction system to properly grade your quiz and assign a score to it. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

- Upgrade the Python file called **bb8_move_custom_service_server.py**. Modify the code you used in **Exercise 3.3**, which contained a Service Server that accepted a custom Service message to activate the circle movement (with a defined duration). This new service will be called /**move_bb8_in_square_custom**. This new service will have to use service messages of the **BB8CustomServiceMessage** type, which is defined here:

The BB8CustomServiceMessage.srv will be something like this:

```
[ ]: float64 side          # The distance of each side of the square
     int32 repetitions     # The number of times BB-8 has to execute the
     square movement when the service is called
     ---
     bool success          # Did it achieve it?
```

NOTE: The **side** variable doesn't represent the real distance along each size of the square. It's just a variable that will be used to change the size of the square. The bigger the **size** variable is, the bigger the square performed by the BB-8 robot will be.

- In the previous exercises, you were triggering a circle movement when calling to your service. In this new service, the movement triggered will have to be a square, like in the image below:

Fig.3.1 - BB8 Square Movement Diagram

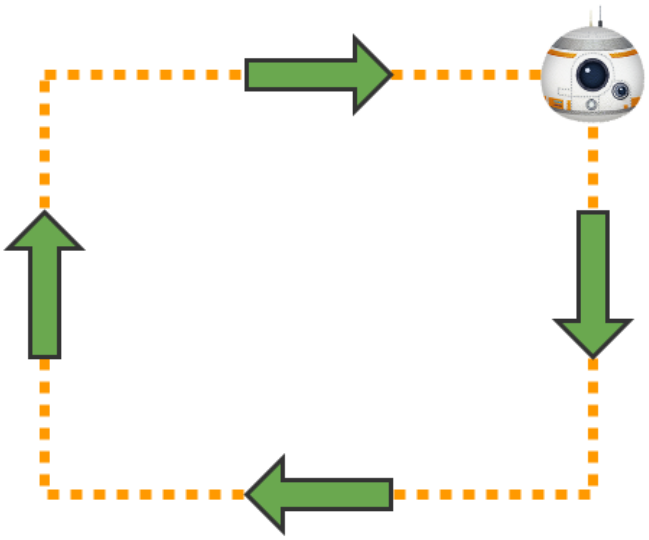


Fig.3.1 - BB8 Square Movement Diagram

- Use the data passed to this new /**move_bb8_in_square_custom** to change the way BB-8 moves. Depending on the **side** value, the service must move the BB-8 robot in a square movement based on the **side** given. Also, the BB-8 must repeat the shape as many times as indicated in the **repetitions** variable of the message. Finally, it must return **True** if everything went OK in the **success** variable.

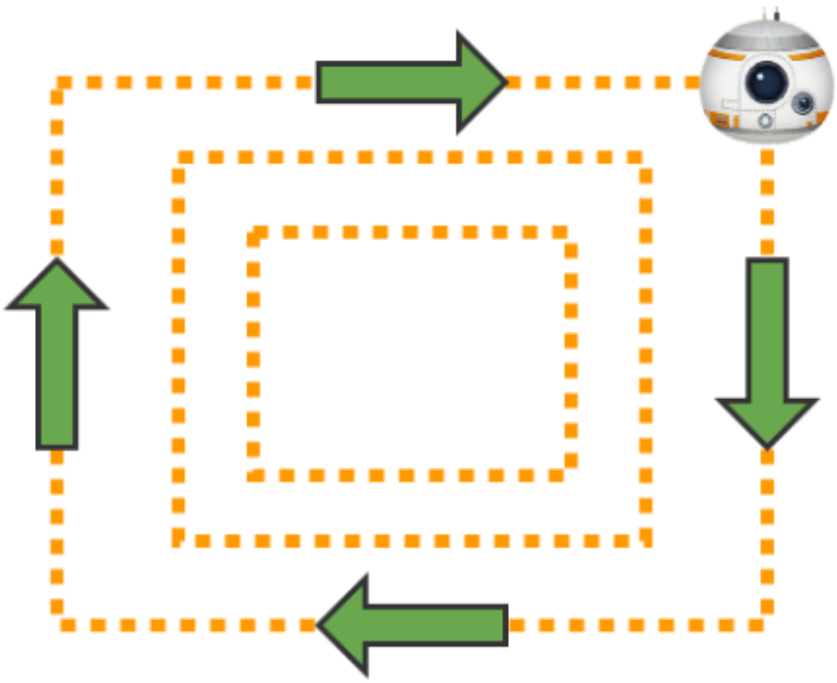
NOTE: The **side** variable doesn't represent the real distance along each size of the square. It's just a variable that will be used to change the size of the square. The bigger the **side** variable is, the bigger the square performed by the BB-8 robot will be.

- Create a new launch file, called **start_bb8_move_custom_service_server.launch**, that launches the new **bb8_move_custom_service_server.py** file.

- Test that when calling this new /**move_bb8_in_square_custom** service, BB-8 moves accordingly. This means, the square is performed taking into account the **side** and **repetitions** variables.

- Create a new service client that calls the service /**move_bb8_in_square_custom**, and makes BB-8 move in a small square **twice** and in a bigger square **once**. It will be called **bb8_move_custom_service_client.py**. The launch that starts it will be called **call_bb8_move_in_square_custom_service_server.launch**.

Fig.3.2 - BB8 Dynamic Square Diagram



BB8 Dynamic Square/Circle Movement Diagram

- The **small square** has to be of, at least, **1 sqm**. The **big square** has to be of, at least, **2 sqm**.

Specifications

- The name of the package where you'll place all the code related to the quiz will be **services_quiz**.
- The name of the launch file that will start your Service Server will be **start_bb8_move_custom_service_server.launch**.
- The name of the service will be /**move_bb8_in_square_custom**.
- The name of your Service message file will be **BB8CustomServiceMessage.srv**.
- The name of the launch file that will call your Services Server will be **call_bb8_move_in_square_custom_service_server.launch**.
- The **small square** has to be of, at least, **1 sqm**. The **big square** has to be of, at least, **2 sqm**.

Quiz Correction
When you have finished the quiz, you can correct it in order to get a score. For that, just click on the following button at the top of this Notebook.
IMPORTANT
Quizes can only be done once. This means that once you correct your quiz and get a score for it, you won't be able to do it again and improve your score. So, be sure when you decide to correct

your quiz!

Congratulations! You are now ready to add all of the services that you want to your own personal astromech droid!

Additional Materials to Learn More

ROS Services: http://wiki.ros.org/Services
Simple Service and Client (Python): http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28python%29
srv Files: http://wiki.ros.org/srv

Python Classes

ROS IN 5 DAYS

Using Python Classes in ROS

BB8

- ROSject Link: https://bit.ly/2RYG5Fv
- Package Name: **bb_8_gazebo**
- Launch File: **main.launch**

Estimated time to completion: 3 hours What will you learn with this unit?

- Introduction to OOP and Python classes
- How to use Python classes in ROS

Note for experienced programmers

If you already know about Object-Oriented Programming, and Python Classes, you can skip the introductory explanations and go straight for the exercises.

Object-Oriented Programming

In all the programs we've wrote until now, we have designed our programs around functions, which manipulate data. This is called the procedure-oriented way of programming. But let me tell you that there is another way of programming! This one is based on combining data and functionality, and wrapping them inside "things" known as objects. This way of programming is known as object-oriented programming, or OOP. You may be asking... why are you telling me about this? And why now? And those are great questions!

Most of the time, you will still be able to use procedural programming, but when writing larger and more complex programs, this method can become a pain. In these cases, it is much better to use OOP, since your code will be better organized and it will be much easier to understand (and debug).

Anyway, let me remind you that the goal of this unit is not to learn what OOP is, or what a Python class is, but how to apply these concepts to your ROS code. This a ROS course, isn't it? So, we'll try to summarize these concepts in the easiest and fastest way possible, just to put everything in context.

So, as I've already said before, OOP is based on the concept of "objects." These objects are usually defined by two things:

- Attributes: This is the data associated with the object, defined in fields.
- Methods: These are the procedures or functions associated with the object.

The most important feature of objects is that the methods associated with them can access and modify the data fields of the object with which they are associated.

What are Python classes?

So, everything sounds super cool and interesting, but I haven't read a word yet about Python classes. What the heck are Python classes? Let's see!

A Python classes is, basically, a code template for creating an object. An object is created using the constructor of the class. This object will then be called the instance of the class. Well, what about seeing an example of a Python class, so that we can understand everything better? Let's see then!

Python File: jedi_class.py

```
[ ]: class Jedi:
         def __init__(self, name):
             self.name = name

         def say_hi(self):
             print('Hello, my name is', self.name)

     j = Jedi('ObiWan')
     j.say_hi()
```

Let's quickly analyze the above class.

```
[ ]: class Jedi:
```

Here, we are creating a new class named Jedi.

```
[ ]: def __init__(self, name):
             self.name = name
```

This is the **__init__** method of the class. It is also known as the constructor because it will be called as soon as an object of a class is created. It is usually used for the initialization of the attributes. In this case, we are initializing an attribute of the class, which is the name of the Jedi.

```
[ ]: def say_hi(self):
             print('Hello, my name is', self.name)
```

This is another method of the class. In this case, it just contains a simple print inside. So, when this method is called, the print will be executed.

```
[ ]: j = Jedi('ObiWan')
```

Here we are creating an object of the Jedi class, which will be stored on the variable **j**. Remember that when this is executed, the **__init__** method of the class will also be automatically executed. So, in this case, the string '**ObiWan**' will be stored in the **self.name** attribute of the class.

```
[ ]: j.say_hi()
```

Finally, we are calling the **say_hi()** method from the class. So, this will result in the following string being printed: **Hello, my name is Obiwan**.

Note that we are using the **self** keyword. This keyword refers to the object itself. Each time we define a new method of the class, we will need to pass this keyword as the first argument. Also, when accessing the attributes of the class, we will use this keyword, as you can see in:

```
[ ]: self.name
```

And that's it! As I've already mentioned before, this unit is not meant to exhaustively teach you how to work with Python classes, but to introduce you to them so that you can use them on your ROS projects. If you are interested in learning more about them, you can refer to the many online tutorials you will find on the web.

You can test the class above by creating the Python file, and executing using the **python** keyword.

Execute in WebShell #1

```
[ ]: python jedi_class.py
```

If everything goes fine, you should get the following output:

```
[ ]: ('Hello, my name is', 'ObiWan')
```

Using Python classes in ROS

Great! So now that you already know what a Python Class is, and how it works, let's try to apply this to a ROS code. For instance, we can start by creating a simple class that will move the BB-8 robot in a circular movement, just as you did in the previous unit.

Example P1
Below you can have a look at a class that will control the movement of the BB-8 robot.

Python File: bb8_move_circle_class.py

```
[ ]: #!/usr/bin/env python

     import rospy
     from geometry_msgs.msg import Twist

     class MoveBB8():

         def __init__(self):
             self.bb8_vel_publisher = rospy.Publisher('/cmd_vel', Twist,
```

```
queue_size=1)
        self.cmd = Twist()
        self.ctrl_c = False
        self.rate = rospy.Rate(10) # 10hz
        rospy.on_shutdown(self.shutdownhook)

    def publish_once_in_cmd_vel(self):
        """
        This is because publishing in topics sometimes fails the first
time you publish.
        In continuous publishing systems, this is no big deal, but in
systems that publish only
        once, it IS very important.
        """
        while not self.ctrl_c:
            connections = self.bb8_vel_publisher.get_num_connections()
            if connections > 0:
                self.bb8_vel_publisher.publish(self.cmd)
                rospy.loginfo("Cmd Published")
                break
            else:
                self.rate.sleep()

    def shutdownhook(self):
        # works better than the rospy.is_shutdown()
        self.ctrl_c = True

    def move_bb8(self, linear_speed=0.2, angular_speed=0.2):

        self.cmd.linear.x = linear_speed
        self.cmd.angular.z = angular_speed

        rospy.loginfo("Moving BB8!")
        self.publish_once_in_cmd_vel()

if __name__ == '__main__':
    rospy.init_node('move_bb8_test', anonymous=True)
    movebb8_object = MoveBB8()
    try:
        movebb8_object.move_x_time(5)
    except rospy.ROSInterruptException:
        pass
```

Let's analyze some of the most important parts of the above class.

```
[ ]: def __init__(self):
        self.bb8_vel_publisher = rospy.Publisher('/cmd_vel', Twist,
```

```
queue_size=1)
        self.cmd = Twist()
        self.ctrl_c = False
        self.rate = rospy.Rate(10) # 10hz
        rospy.on_shutdown(self.shutdownhook)
```

This is the constructor of the class. Here, we are defining several attributes:

- **bb8_vel_publisher**: Creates a Publisher for the /**cmd_vel** topic
- **cmd**: Contains a Twist message
- **ctrl_c**: Contains a boolean indicating if we have pressed Ctrl+C in order to stop the program
- **rate**: Contains the rate frequency

Also, as you can see in the last line of the **__init__** method, we are setting that when our ROS program is closed (**rospy.on_shutdown**), the class method **self.shutdownhook** will be triggered.

```
[ ]: def publish_once_in_cmd_vel(self):
        """
        This is because publishing in topics sometimes fails the first
    time you publish.
        In continuous publishing systems, this is no big deal, but in
    systems that publish only
        once, it IS very important.
        """
        while not self.ctrl_c:
            connections = self.bb8_vel_publisher.get_num_connections()
            if connections > 0:
                self.bb8_vel_publisher.publish(self.cmd)
                rospy.loginfo("Cmd Published")
                break
            else:
                self.rate.sleep()
```

This class method is used in order to make sure that the first message we publish into a topic is successfully received, as you can read on the commented code. Don't pay too much attention to the code inside it, since it's not important right now. Just keep in mind that you can use this class method for situations where you will publish one single command into a topic.

```
[ ]: def shutdownhook(self):

        self.ctrl_c = True
```

This method will be called when our ROS program is closed, as you saw on the constructor of the class. Therefore, we are setting the **ctrl_c** attribute to True.

```
[ ]: def move_bb8(self, linear_speed=0.2, angular_speed=0.2):

            self.cmd.linear.x = linear_speed
            self.cmd.angular.z = angular_speed

            rospy.loginfo("Moving BB8!")
            self.publish_once_in_cmd_vel()
```

This method is used to move the robot. As you can see, we just fill in the **cmd** attribute with the values we want, and we then call the **publish_once_in_cmd_vel()** method.

```
[ ]: if __name__ == '__main__':
        rospy.init_node('move_bb8_test', anonymous=True)
        movebb8_object = MoveBB8()
        try:
            movebb8_object.move_bb8()
        except rospy.ROSInterruptException:
            pass
```

Finally, in the main function, we are doing three things:

- We create a ROS node named **move_bb8_test**
- We create an object of the MoveBB8 class, which is stored into a variable named **movebb8_object**.
- We call the **move_bb8()** method of the class in order to start moving the BB-8 robot.

Note that when calling the **move_bb8()** method, we are using a **try/except** structure. This is because if the call to the method fails, the error would be caught by the **ROSInterruptException**, avoiding error messages.

Great! So now, we already have a class that moves our BB-8 robot in a circle movement. Let's test it. In this case, since we are using a ROS node, we are going to create a ROS package in which to place the class.

Execute in WebShell #1

```
[ ]: catkin_create_pkg my_python_class rospy
```

Now, add the bb8_move_circle_class.py file to your package and execute it by running the following command:

Execute in WebShell #1

```
[ ]: rosrun my_python_class bb8_move_circle_class.py
```

If everything is OK, you should see your BB-8 robot start moving in circles.

Excellent!! But now you may be asking yourself... what happens with the ROS Service? I don't see any ROS Service in the Python class above. It's just a Python script that moves the robot, right? Well yes, you're totally right!

What happens with the ROS Service is... nothing. Or everything. It's up to you. What I mean is that now you have a class that can be used in order to move the BB-8 robot. And it can be used by anybody, included a ROS Service, of course. In fact, now it's extremely easy to add this Python class to a Service Server. Let me show you an example below:

Python File: bb8_move_circle_service_server.py

```
[ ]: #! /usr/bin/env python

     import rospy
     from std_srvs.srv import Empty, EmptyResponse
     from bb8_move_circle_class import MoveBB8

     def my_callback(request):
         rospy.loginfo("The Service move_bb8_in_circle has been called")
         movebb8_object = MoveBB8()
         movebb8_object.move_bb8()
         rospy.loginfo("Finished service move_bb8_in_circle")
         return EmptyResponse()

     rospy.init_node('service_move_bb8_in_circle_server')
     my_service = rospy.Service('/move_bb8_in_circle', Empty , my_callback)
     rospy.loginfo("Service /move_bb8_in_circle Ready")
     rospy.spin() # mantain the service open.
```

So basically, we just have the following lines:

```
[ ]: from move_bb8_circle_class import MoveBB8
```

This line of code imports the **MoveBB8** class we've created from the Python file **move_bb8_circle_class.py**.

```
[ ]: movebb8_object = MoveBB8()
     movebb8_object.move_bb8()
```

Then here, we are creating an object of the **MoveBB8**, and we are calling the **move_bb8()** method in order to start moving the robot. Quite simple, right?

So, let's test all this together. First, add the bb8_move_circle_service_server.py file to your package. Add also a launch file that starts the Service Server, like the one below:

Launch File: bb8_move_circle_service_server.launch

```
[ ]: <launch>
        <!-- Start Service Server for move_bb8_in_circle service -->
        <node pkg="my_python_class"
     type="bb8_move_circle_service_server.py"
     name="service_move_bb8_in_circle_server"  output="screen">
        </node>
     </launch>
```

Finally, execute the launch file and check that everything works as expected. That is, the BB-8 robot starts moving in circles.

Exercise P1
Modify the Python scripts you created in **Example P1** so that now they include the custom Service Message you used for **Exercise 3.3**, in the previous unit.

- You will need to modify the class so that now, the BB-8 robot stops moving after the specified time in the message has passed.
- You will also need to modify the Service Server code so that you pass the specified duration to the class.

Solution Exercise P1
Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Python Classes: solutions_python_classes

Additional Materials to Learn More

Python Classes: https://docs.python.org/2/tutorial/classes.html

Unit 4. ROS Actions Part1

ROS IN 5 DAYS

Unit 4: ROS Actions

Drone

- ROSject Link: http://bit.ly/2naOJid
- Package Name: **drone_construct**
- Launch File: **main.launch**

Estimated time to completion: 3 hours What will you learn with this unit?

- What is a ROS action
- How to manage the actions of a robot
- How to call an action server

Part 1

1) Did you understand the previous sections about topics and services?
2) Are they clear to you?
3) Did you have a good breakfast today?

If your answers to all of those questions were yes, then you are ready to learn about ROS actions. Otherwise, go back and do not come back until all of those answers are a big YES. You are going to need it...

Quadrotor simulation Before starting with ROS actions learning, let's have some fun with the quadrotor simulation. Make the quadrotor take off and control it with the keyboard. How would you do that?

By issuing the following commands:

First, you need to take off.

Execute in WebShell #1

```
[ ]: rostopic pub /drone/takeoff std_msgs/Empty "{}"
```

Hit **CTRL+C** to stop it and to be able to type more commands. In this case, the commands to move the drone with the keyboard.

Execute in WebShell #1

```
[ ]: rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

To land the drone, just publish into the /**drone/land** topic:

Execute in WebShell #1

```
[ ]: rostopic pub /drone/land std_msgs/Empty "{}"
```

Code Explanation #1
"rosrun": ROS command that allows you to run a ROS program without having to create a launch file to launch it (it is a different way from what we've been doing here).

"teleop_twist_keyboard": Name of the package where the ROS program is. In this case, where the python executable is.

"teleop_twist_keyboard.py": Python executable that will be run. In this case, it's an executable that allows you to input movement commands through the keyboard. When executed, it displays the instructions to move the robot.

End Code Explanation #1
Fig.4.1 - Ardrone moved with teleop_twist_keyboard.py.

Ardrone moved with teleop twist keyboard

Exercise 4.1
Try to takeoff, move, and land the drone using the keyboard, as shown in the gif in {Fig:4.1}

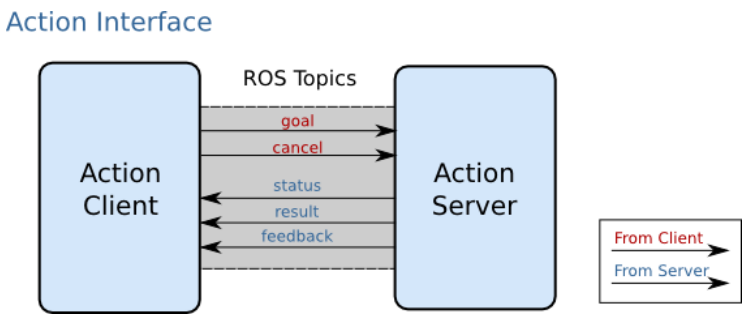
What are actions?

Actions are like asynchronous calls to services Actions are very similar to services. When you call an action, you are calling a functionality that another node is providing. Just the same as with services. The difference is that when your node calls a service, it must wait until the service finishes. When your node calls an action, it doesn't necessarily have to wait for the action to complete.

Hence, an action is an asynchronous call to another node's functionality.

- The node that provides the functionality has to contain an action server. The action server allows other nodes to call that action functionality.
- The node that calls to the functionality has to contain an action client. The action client allows a node to connect to the action server of another node.

Fig.4.2 - Action Interface Diagram showing the under-the-hood communication system



Action Interface Diagram showing the under-the-hood communication system

Now let's see an action in action (I'm so funny!)
Exercise 4.2
Go to a shell and launch the ardrone action server with the following command: **Important!!** Keep this program running for the rest of the tutorial, since it is the one that provides the action server you are going to use.
Execute in WebShell #1, Leave It Running

```
[ ]: roslaunch ardrone_as action_server.launch
```

Questions:

- How do you know the whole list of topics available?

Execute in WebShell #2

```
[ ]: rostopic list
```

- How do you know the whole list of services available?

Execute in WebShell #2

```
[ ]: rosservice list
```

- How do you know the whole list of action servers available?

Execute in WebShell #2

```
[ ]: rosaction list
```

WRONG GUESS!!!

Fig.4.3 - Didn't Say The Magic Word

Did not Say The Magic Word

In order to find which actions are available on a robot, you must do a **rostopic list.**
Execute in WebShell #2

```
[ ]: rostopic list
```

WebShell #2 Output

```
[ ]: user ~ $ rostopic list
     ...
     ...
     /ardrone_action_server/cancel
     /ardrone_action_server/feedback
     /ardrone_action_server/goal
     /ardrone_action_server/result
     /ardrone_action_server/status
     ...
     ...
```

When a robot provides an action, you will see that in the topics list. There are 5 topics with the same base name, and with the subtopics **cancel**, **feedback**, **goal**, **result**, and **status**.

For example, in the previous rostopic list, the following topics were listed:

```
[ ]: /ardrone_action_server/cancel
     /ardrone_action_server/feedback
     /ardrone_action_server/goal
     /ardrone_action_server/result
     /ardrone_action_server/status
```

This is because you previously launched the ardrone_action_server with the command roslaunch ardrone_as action_server.launch (Excercice 4.2).

Every action server creates those 5 topics, so you can always tell that an action server is there because you identified those 5 topics.

Therefore, in the example above:

- ardrone_action_server: Is the name of the Action Server.
- cancel, feedback, goal, result and status: Are the messages used to comunicate with the Action Server.

Calling an action server

The **ardrone_action_server** action server is an action that you can call. If you call it, it will start taking pictures with the front camera, one picture every second, for the amount of seconds specified in the calling message (it is a parameter that you specify in the call).

Calling an action server means sending a message to it. In the same way as with topics and services, it all works by passing messages around.

- The message of a topic is composed of a single part: the information the topic provides.
- The message of a service has two parts: the goal and the response.
- The message of an action server is divided into three parts: the goal, the result, and the feedback.

All of the action messages used are defined in the **action** directory of their package. You can go to the **ardrone_as** package and see that it contains a directory called **action**. Inside that action directory, there is a file called **Ardrone.action**. That is the file that specifies the type of the message that the action uses.

Type in a shell the following commands to see the message structure:

Exercise 4.3
Type in a shell the following commands to see the message structure:

Execute in WebShell #2

```
[ ]: roscd ardrone_as/action; cat Ardrone.action
```

WebShell #2 Output

```
[ ]: user ~ $ roscd ardrone_as/action; cat Ardrone.action
     #goal for the drone
     int32 nseconds  # the number of seconds the drone will be taking
     pictures
     ---
     #result
     sensor_msgs/CompressedImage[] allPictures # an array containing all
     the pictures taken along the nseconds
     ---
     #feedback
     sensor_msgs/CompressedImage lastImage  # the last image taken
```

You can see in the previous step how the message is composed of three parts: goal: Consists of a variable called **nseconds** of type **Int32**. This Int32 type is a standard ROS message, therefore, it can be found in the std_msgs package. Because it's a standard package of ROS, it's not needed to indicate the package where the Int32 can be found. result: Consists of a variable called **allPictures**, which is an array of type **CompressedImage[]**, found in the sensor_msgs package. feedback: Consists of a variable called **lastImage** of type **CompressedImage[]**, found in the sensor_msgs package.

You will learn in the second part of this chapter about how to create your own action messages. For now, you must only understand that every time you call an action, the message implied

contains three parts, and that each part can contain more than one variable.

Actions provide feedback

Due to the fact that calling an action server does not interrupt your thread, action servers provide a message called the feedback. The feedback is a message that the action server generates every once in a while to indicate how the action is going (informing the caller of the status of the requested action). It is generated while the action is in progress.

How to call an action server

The way you call an action server is by implementing an action client.

The following is a self-explanatory example of how to implement an action client that calls the ardrone_action_server and makes it take pictures for 10 seconds.

Exercise 4.4
Execute the following Python code ardone_action_client.py by clicking on it and then clicking on the play button on the top righ-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it. Observe how the feedback messages are received (printed below the code).

Important!! Remember that you have to have the **roslaunch ardrone_as action_server.launch** running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

Exercise 4.4
Python Program {4.4a}: ardrone_action_client.py

```
[ ]: #! /usr/bin/env python
     import rospy
     import time
     import actionlib
     from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
     ArdroneFeedback

     nImage = 1

     # definition of the feedback callback. This will be called when
     feedback
     # is received from the action server
     # it just prints a message indicating a new message has been received
     def feedback_callback(feedback):
```

```
        global nImage
        print('[Feedback] image n.%d received'%nImage)
        nImage += 1

    # initializes the action client node
    rospy.init_node('drone_action_client')

    # create the connection to the action server
    client = actionlib.SimpleActionClient('/ardrone_action_server',
    ArdroneAction)
    # waits until the action server is up and running
    client.wait_for_server()

    # creates a goal to send to the action server
    goal = ArdroneGoal()
    goal.nseconds = 10 # indicates, take pictures along 10 seconds

    # sends the goal to the action server, specifying which feedback
    function
    # to call when feedback received
    client.send_goal(goal, feedback_cb=feedback_callback)

    # Uncomment these lines to test goal preemption:
    #time.sleep(3.0)
    #client.cancel_goal()  # would cancel the goal 3 seconds after
    starting

    # wait until the result is obtained
    # you can do other stuff here instead of waiting
    # and check for status from time to time
    # status = client.get_state()
    # check the client API link below for more info

    client.wait_for_result()

    print('[Result] State: %d'%(client.get_state()))
```

Code Explanation Python Program: {4.4a}
The code to call an action server is very simple:

- First, you create a client connected to the action server you want:

  client = actionlib.SimpleActionClient('/ardrone_action_server', ArdroneAction)
  client = actionlib.SimpleActionClient('/the_action_server_name', the_action_server_message_python_object)
- First parameter is the name of the action server you want to connecto to.

- Second parameter is the type of action message that it uses. The convention goes as follows: If your action message file was called **Ardrone.action**, then the type of action message you must specify is **ArdroneAction**. The same rule applies to any other type (**R2Action**, for an **R2.action** file or **LukeAction** for a **Luke.action** file). In our exercise it is:

  client = actionlib.SimpleAction('/ardrone_action_server', ArdroneAction)
- Then you create a goal:

  goal = ArdroneGoal()
  Again, the convention goes as follows: If your action message file was called **Ardrone.action**, then the type of goal message you must specify is **ArdroneGoal()**. The same rule applies to any other type (**R2Goal()** for an **R2.action** file or **LukeGoal()** for a **Luke.action** file). Because the goal message requires to provide the number of seconds taking pictures (in the **nseconds variable**), you must set that parameter in the goal class instance:

  goal.nseconds = 10
- Next, you send the goal to the action server:

  client.send_goal(goal, feedback_cb=feedback_callback)
  That sentence calls the action. In order to call it, you must specify 2 things:

1. The goal parameters
2. A feedback function to be called from time to time to know the status of the action.

At this point, the action server has received the goal and started to execute it (taking pictures for 10 seconds). Also, feedback messages are being received. Every time a feedback message is received, the **feedback_callback** function is executed. * Finally, you wait for the result:

client.wait_for_result()
End Code Explanation {4.4a}

How to perform other tasks while the Action is in progress

You know how to call an action and wait for the result but... That's exactly what a service does! Then why are you learning actions? Good point! So, the **SimpleActionClient** objects have two functions that can be used for knowing if the action that is being performed has finished, and how:

1) wait_for_result(): This function is very simple. When called, it will wait until the action has finished and returns a true value. As you can see, it's useless if you want to perform other tasks in parallel because the program will stop there until the action is finished.
2) get_state(): This function is much more interesting. When called, it returns an integer that indicates in which state is the action that the SimpleActionClient object is connected to.

DONE 3 ==> WARN 4 ==> ERROR
This allows you to create a while loop that checks if the value returned by get_state() is 2 or higher. If it is not, it means that the action is still in progress, so you can keep doing other things.

Exercise 4.5
Execute the following Python codes {4.5a: wait_for_result_test.py} and {4.5b: no_wait_for_result_test.py} by clicking on them, one at a time, and then clicking on the play button on the top righ-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it.

When the program has finished, don't forget to restart the Kernel. This will clean up all the nodes generated by ROS through the python program. This is necessary because python programs can only launch one rospy node. Because this notebook is just a divided python script, it will give rospy Exception if you try to execute two snipets consecutively without restarting the kernel. You can do it by pressing the icon.

Observe the difference between them (the code is printed below) and think about why this is.

Important!! Remember that you need to have the roslaunch ardrone_as action_server.launch** running (probably in WebShell #1), otherwise this won't work because there will be NO action-Server to be connected to.

Python Program {4.5a}: wait_for_result_test.py

```
[ ]: #! /usr/bin/env python

     import rospy
     import time
     import actionlib
     from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
     ArdroneFeedback

     nImage = 1

     # definition of the feedback callback. This will be called when
     feedback
     # is received from the action server
     # it just prints a message indicating a new message has been received
     def feedback_callback(feedback):
         global nImage
         print('[Feedback] image n.%d received'%nImage)
         nImage += 1


     # initializes the action client node
     rospy.init_node('example_with_waitforresult_action_client_node')
```

```
action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name,
ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)


# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)
rate = rospy.Rate(1)

rospy.loginfo("Lets Start The Wait for the Action To finish Loop...")
while not client.wait_for_result():
    rospy.loginfo("Doing Stuff while waiting for the Server to give a
result....")
    rate.sleep()

rospy.loginfo("Example with WaitForResult Finished.")
```

Python Program {4.5b}: no_wait_for_result_test.py

```
[ ]: #! /usr/bin/env python

import rospy
import time
import actionlib
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
ArdroneFeedback

"""
class SimpleGoalState:
    ACTIVE = 1
    DONE = 2
    WARN = 3
    ERROR = 4

"""
# We create some constants with the corresponing vaules from the
SimpleGoalState class
ACTIVE = 1
DONE = 2
WARN = 3
```

```
ERROR = 4

nImage = 1

# definition of the feedback callback. This will be called when
feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    """
    Error that might jump

    self._feedback.lastImage =
AttributeError: 'ArdroneAS' obj

    """
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('example_no_waitforresult_action_client_node')

action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name,
ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)


# You can access the SimpleAction Variable "simple_state", that will
be 1 if active, and 2 when finished.
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2
if NO ERROR, 3 If Any Warning, and 3 if ERROR
state_result = client.get_state()

rate = rospy.Rate(1)
```

```
rospy.loginfo("state_result: "+str(state_result))

while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a
result....")
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result))

rospy.loginfo("[Result] State: "+str(state_result))
if state_result == ERROR:
    rospy.logerr("Something went wrong in the Server Side")
if state_result == WARN:
    rospy.logwarn("There is a warning in the Server Side")

#rospy.loginfo("[Result] State: "+str(client.get_result()))
```

Code Explanation Python Programs: {4.5a} and {4.5b}
Essentially, the difference is that in the first program (4.5a), the log message rospy.loginfo("Doing Stuff while waiting for the Server to give a result. . . .") will never be printed, while in the 4.5b it will.

This is because in 4.5a, the program starts the while loop to check if the value returned by client.wait_for_result() is True or False, but it will wait for a value that will only be returned when the Action has Finished. Therefore, it will never get inside the while loop because it will always return the value True.

On the other hand, in the 4.5b program, it checks if state_result < DONE. And because the function **get_state()** will return the current state of the action immediately, it allows other tasks to perform in parallel. In this case, printing the log message WHILE printing also the feedback of the the Action.

End Code Explanation Python Programs: {4.5a} and {4.5b}
You can get more information about ROS action clients in python in this client API
Exercise 4.6
Create a package that contains and launches the action client from Excercice {4.4a}: ardrone_action_client.py, from a launch file.

Add some code that makes the quadcopter move around while the action server has been called (in order to take pictures while the robot is moving).

Stop the movement of the quadcopter when the last picture has been taken (action server has finished).

Data for Exercice 4.6
1) You can send Twist commands to the quadcopter in order to move it. These commands have to be published in /**cmd_vel** topic. Remember the **TopicsUnit**. 2) You must send movement commands while waiting until the result is received, creating a loop that sends commands at the

same time that check for completion. In order to be able to send commands while the action is in progress, you need to use the SimpleActionClient function **get_state()**.

Solution Exercise 4.6
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit3 Services Part1:Actions Part1 Solutions

Preempting a goal

It happens that you can cancel a goal previously sent to an action server prior to its completion. Cancelling a goal while it is being executed is called preempting a goal You may need to preempt a goal for many reasons, like, for example, the robot went mad about your goal and it is safer to stop it prior to the robot doing some harm. In order to preempt a goal, you send the cancel_goal to the server through the client connection.

```
[ ]: client.cancel_goal()
```

Exercise 4.7
Execute the following Python code {4.6a: cancel_goal_test.py} by clicking on it and then clicking on the play button on the top right-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it.

When the program has finished, don't forget to restart the Kernel. This will clean up all of the nodes generated by ROS through the python program. This is necessary because python programs can only launch one rospy node. Because this notebook is just a divided python script, it will give rospy Exception if you try to execute two snipets consecutively without restarting the kernel. You can do it by pressing the icon.

See how the goal gets cancelled.

Important!! Remember that you have to have the **roslaunch ardrone_as action_server.launch** running (probably in WebShell #1), otherwise this won't work because there will be NO action server to be connected to.

Python Program {4.6a}: cancel_goal_test.py

```
[ ]: #! /usr/bin/env python

     import rospy
     import time
     import actionlib
```

```
from ardrone_as.msg import ArdroneAction, ArdroneGoal, ArdroneResult,
ArdroneFeedback

# We create some constants with the corresponing vaules from the
SimpleGoalState class
ACTIVE = 1
DONE = 2
WARN = 3
ERROR = 4

nImage = 1

# definition of the feedback callback. This will be called when
feedback
# is received from the action server
# it just prints a message indicating a new message has been received
def feedback_callback(feedback):
    """
    Error that might jump

    self._feedback.lastImage =
AttributeError: 'ArdroneAS' obj

    """
    global nImage
    print('[Feedback] image n.%d received'%nImage)
    nImage += 1

# initializes the action client node
rospy.init_node('example_no_waitforresult_action_client_node')

action_server_name = '/ardrone_action_server'
client = actionlib.SimpleActionClient(action_server_name,
ArdroneAction)

# waits until the action server is up and running
rospy.loginfo('Waiting for action Server '+action_server_name)
client.wait_for_server()
rospy.loginfo('Action Server Found...'+action_server_name)

# creates a goal to send to the action server
goal = ArdroneGoal()
goal.nseconds = 10 # indicates, take pictures along 10 seconds

client.send_goal(goal, feedback_cb=feedback_callback)


# You can access the SimpleAction Variable "simple_state", that will
```

```
be 1 if active, and 2 when finished.
#Its a variable, better use a function like get_state.
#state = client.simple_state
# state_result will give the FINAL STATE. Will be 1 when Active, and 2
if NO ERROR, 3 If Any Warning, and 3 if ERROR
state_result = client.get_state()

rate = rospy.Rate(1)

rospy.loginfo("state_result: "+str(state_result))
counter = 0
while state_result < DONE:
    rospy.loginfo("Doing Stuff while waiting for the Server to give a
result....")
    counter += 1
    rate.sleep()
    state_result = client.get_state()
    rospy.loginfo("state_result: "+str(state_result)+", counter
="+str(counter))
    if counter == 2:
        rospy.logwarn("Canceling Goal...")
        client.cancel_goal()
        rospy.logwarn("Goal Canceled")
        state_result = client.get_state()
        rospy.loginfo("Update state_result after Cancel :
"+str(state_result)+", counter ="+str(counter))
```

Code Explanation Python Program: {4.6a}
It's exactly the same code as the {4.5b}, except for the use of the **cancel_goal()** function. This program counts to 2, and then it cancels the goal. This triggers the server to finish the goal and, therefore, the function **get_state()** returns the value DONE (2).

End Code Explanation Python Program: {4.6a}
There is a known ROS issue with Actions. It issues a warning when the connection is severed. It normally happens when you cancel a goal or you just terminate a program with a client object in it. The warning is given in the Server Side. [WARN] Inbound TCP/IP connection failed: connection from sender terminated before handshake header received. 0 bytes were received. Please check sender for additional details. Just don't panic, it has no effect on your program.

How does all that work?

You need to understand how the communication inside the actions works. It is not that you are going to use it for programming. As you have seen, programming an action client is very simple. However, it will happen that your code will have bugs and you will have to debug it. In order to do proper debugging, you need to understand how the communication between action servers and action clients works.

As you already know, an *action message* has three parts:

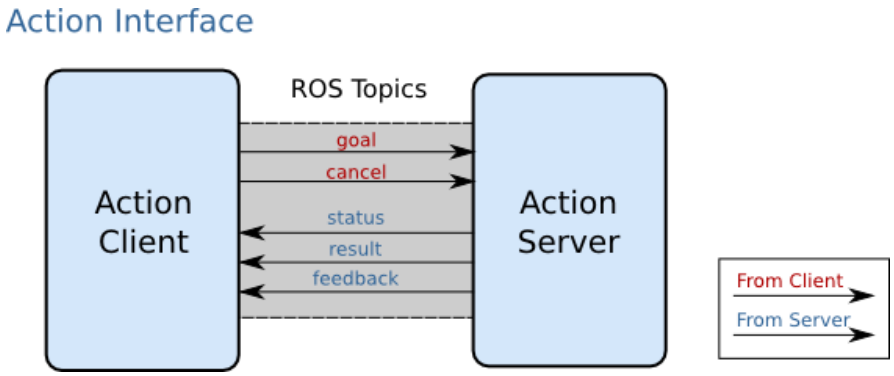
- the goal
- the result
- the feedback

Each one corresponds to a topic and to a type of message. For example, in the case of the ardrone_action_server, the topics involved are the following:

- the goal topic: /ardrone_action_server/goal
- the result topic: /ardrone_action_server/result
- the feedback topic: /ardrone_action_server/feedback

Look again at the ActionClient+ActionServer communication diagram.

Fig.4.4 - Action Interface Diagram Copy



Action Interface Diagram

So, whenever an action server is called, the sequence of steps are as follows:

1. When an **action client** calls an **action server** from a node, what actually happens is that the **action client** sends to the **action server** the goal requested through the /ardrone_action_server/goal topic.
2. When the **action server** starts to execute the goal, it sends to the **action client** the feedback through the /ardrone_action_server/feedback topic.
3. Finally, when the **action server** has finished the goal, it sends to the **action client** the result through the /ardrone_action_server/result topic.

Now, let's do the following exercise in order to see how all this ballet happens underneath your programs.

Exercise 4.8
Read the whole exercise prior to executing it, because you have to get properly set up prior to the start of executing anything.

Execute in WebShell #1: Demonize action_server and echo the goal topic
(press [CTRL]+[C] to kill the ardrone_as action_server.launch if you had it still running)

```
[ ]: roslaunch ardrone_as action_server.launch &
```

(to kill it when finished, use the command: **rosnode kill /ardrone_as**)

```
[ ]: rostopic echo /ardrone_action_server/goal
```

Execute in WebShell #2: echo the feedback topic

```
[ ]: rostopic echo /ardrone_action_server/feedback
```

Execute in WebShell #3: echo the result topic

```
[ ]: rostopic echo /ardrone_action_server/result
```

Execute in WebShell #4: Launch the action server client
(Execute the launch you created in Exercice 4.6, so that the drone starts to take pictures and move around)
Now do the following:

- Quickly visit the terminal that contained the goal echo (WebShell #1). A message should have appeared indicating the goal you sent (with 10 seconds of taking pictures).
- Quickly visit the feedback terminal containing the feedback echo (WebShell #2). A new message should be appearing every second. This is the feedback sent by the **action server** to the **action client**.
- Quickly visit the result terminal (WebShell #3). If 10 seconds have not yet passed since you launched the **action client**, then there should be no message. If you wait there until the 10 seconds pass, you will see the result message sent by the **action server** to the **action client** appear.

Data for Excercice 4.8
Each one of those three topics have their own type of message. The type of the message is built automatically by ROS from the **.action** file.

For example, in the case of the **ardrone_action_server** the **action file** is called **Ardrone.action**.

When you compile the package (with **catkin_make**), ROS will generate the following types of messages from the **Ardrone.action** file:

- ArdroneActionGoal
- ArdroneActionFeedback
- ArdroneActionResult

Each topic of the action server uses its associated type of message accordingly.

Exercise 4.9
Do a rostopic info of any of the action topics and check that the types really correspond with the ones indicated above.

Exercise 4.10
Due to the way actions work, you can actually call the ardrone_action_server action server publishing in the topics directly (emulating by hence, what the Python code action client is doing). It is important that you understand this because you will need this knowledge to debug your programs.

Execute in WebShell #1: Run the action server
(press [CTRL]+[C] to kill the ardrone_as action_server.launch if you had it still running. If you had it demonized, use the command: rosnode kill /ardrone_as) roslaunch ardrone_as action_server.launch

```
[ ]: roslaunch ardrone_as action_server.launch
```

Let's activate the ardrone_action_server action server through topics with the following exercise. Use the webshell to send a goal to the /ardrone_action_server action server, and to observe what happens in the result and feedback topics.

Execute in WebShell #2: Send goal to the action server

```
[ ]: rostopic pub /[name_of_action_server]/goal
     /[type_of_the_message_used_by_the_topic] [TAB][TAB]
```

Expected Result for Exercice 4.10
You should see the same result as in exercice 4.8, with the difference that the goal has been sent by hand, publishing directly into the goal topic, instead of publishing through a Python program.

Data for Excercice 4.10

- You don't have to type the message by hand. Remember to use TAB-TAB to make ROS autocomplete your commands (or give you options).
- Once you achieve that, and ROS autocompletes the message you must send, you will have to modify the parameter **nseconds**, because the default value is zero (remember that parameter indicates the number of seconds taking pictures). Move to the correct place of the message using the keyboard.

The axclient

Until now, you've learnt to send goals to an Action Server using these 2 methods:

- Publishing directly into the /goal topic of the Action Server
- Publishing the goal using Python code

But, let me tell you that there's still one more method you can use in order to send goals to an Action Server, which is much easier and faster than the 2 methods you've learnt: using the axclient.

The axclient is, basically, a GUI tool provided by the actionlib package, that allows you to interact with an Action Server in a very easy and visual way. The command used to launch the axclient is the following:

```
[ ]: rosrun actionlib axclient.py /<name_of_action_server>
```

Want do you think? Do you want to try it? Let's go then!

Exercise 4.11
Before starting with this exercise, make sure that you have your action server running. If it is not running, the axclient won't work.

Execute in WebShell #1: Run the action server
(press [CTRL]+[C] to kill the ardrone_as action_server.launch if you had it still running. If you had it demonized, use the command: rosnode kill /ardrone_as)

```
[ ]: roslaunch ardrone_as action_server.launch
```

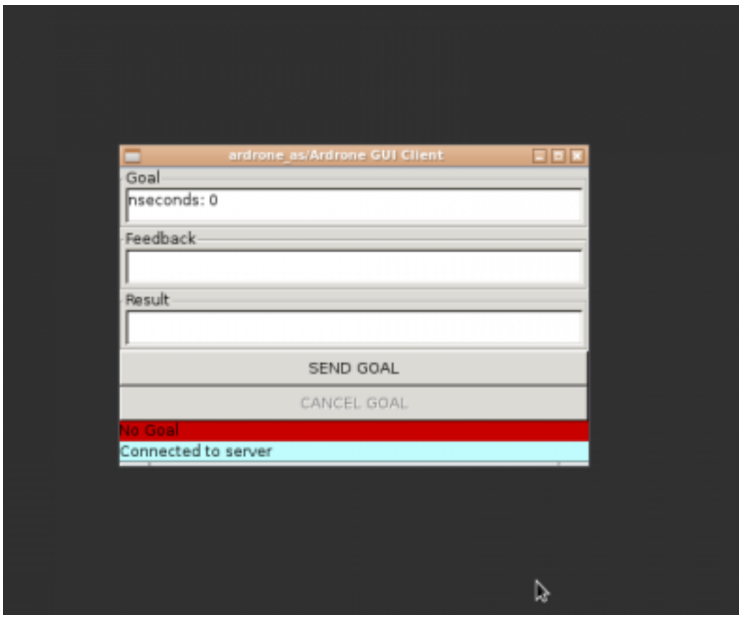
Now, let's launch the axclient in order to send goals to the Action Server.

Execute in WebShell #2: Launch axclient

```
[ ]: rosrun actionlib axclient.py /ardrone_action_server
```

In order to be able to visualize axclient, you will need to open the graphical interface window. To open this graphical interface window, just hit the icon with a screen in the IDE and a new tab will open.

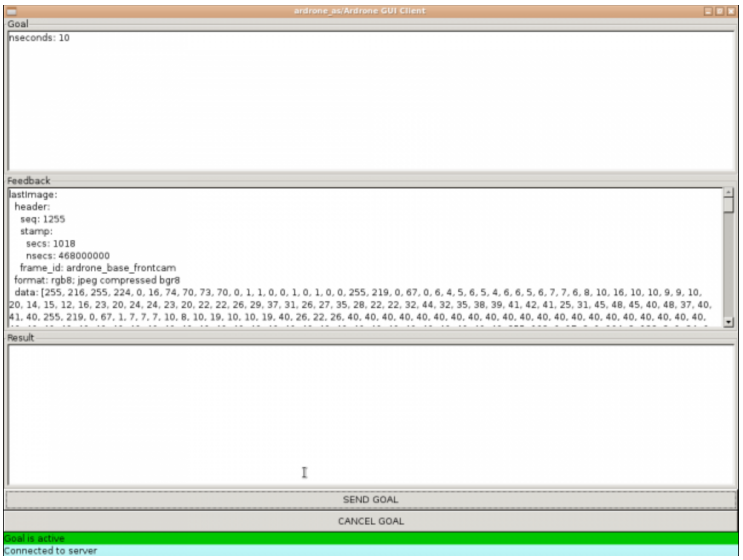
Now, you should see in your new tab something like this:



Axclient GUI

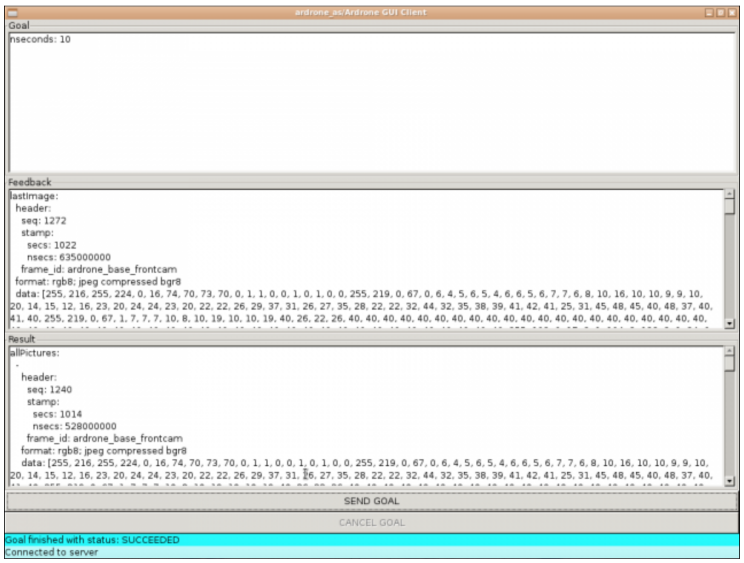
Now everything is settled, and you can start playing with axclient! For instance, you could send goals to the Action Server and visualize the different topics that take part in an Action, which you have learnt in this Chapter.

Expected Result for Exercice 4.11
Action in process:



Action in progress

Action succeeded:



Action Succeeded

Data for Exercice 4.11

- when you try to change the value of the goal you want to send, you can't interact with the axclient screen. If that's the case, just go to another tab and return again to the tab with axclient, and everything will work fine.

Exercise 4.11
Maybe you're a little bit angry at us now, because we didn't show you this tool before? Don't be!! The very simple reason why we didn't talk about this tool earlier is because we want you to learn how Actions really work inside. Once you have the knowledge, you are then ready to use the shortcuts.

Unit 4. ROS Actions Part 2

ROS IN 5 DAYS

Unit 4: ROS Actions

Drone

- ROSject Link: http://bit.ly/2naOJid
- Package Name: **drone_construct**
- Launch File: **main.launch**

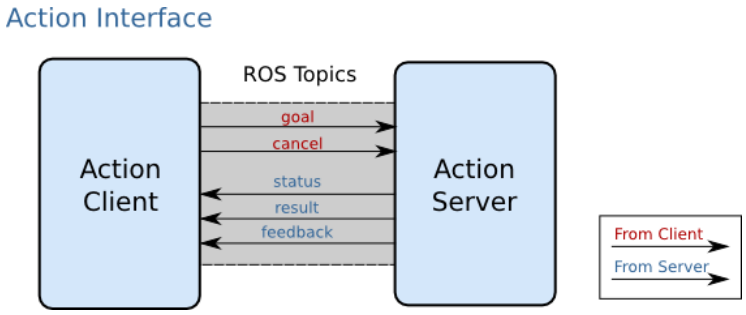
Estimated time to completion: 2.5 hours What will you learn with this unit?

- How to create an action server
- How to build your own action message

Part 2

In the previous lesson, you learned how to CALL an action server creating an action client. In this lesson, you are going to learn how to CREATE your own action server.

Fig.4.5 - Action Interface Diagram Copy 2



Action Interface Diagram Copy 2

Writing an action server

Exercise 4.11: Test Fibonacci Action Server through Notebook
Execute the following Python code by clicking on it and then clicking on the play button on the top righ-hand corner of the IPython notebook.

You can also press **[CTRL]+[Enter]** to execute it.

When the program has finished, don't forget to restart the Kernel. This will clean up all the nodes generated by ROS through the python program. This is necessary because python programs can only launch one rospy node. Because this notebook is just a divided python script, it will give rospy Exception if you try to execute two snipets consecutively without restarting the kernel. You can do it by pressing the icon.

What follows is the code of an example of a ROS action server. When called, the action server will generate a Fibonacci sequence of a given order. The action server goal message must indicate the order of the sequence to be calculated, the feedback of the sequence as it is being computed, and the result of the final Fibonacci sequence.

Python Program {4.11a}: fibonacci_action_server.py

```
[1]: #! /usr/bin/env python
     import rospy

     import actionlib

     from actionlib_tutorials.msg import FibonacciFeedback,
     FibonacciResult, FibonacciAction
```

```
class FibonacciClass(object):

  # create messages that are used to publish feedback/result
  _feedback = FibonacciFeedback()
  _result   = FibonacciResult()

  def __init__(self):
    # creates the action server
    self._as = actionlib.SimpleActionServer("fibonacci_as",
FibonacciAction, self.goal_callback, False)
    self._as.start()

  def goal_callback(self, goal):
    # this callback is called when the action server is called.
    # this is the function that computes the Fibonacci sequence
    # and returns the sequence to the node that called the action
server

    # helper variables
    r = rospy.Rate(1)
    success = True

    # append the seeds for the fibonacci sequence
    self._feedback.sequence = []
    self._feedback.sequence.append(0)
    self._feedback.sequence.append(1)

    # publish info to the console for the user
    rospy.loginfo('"fibonacci_as": Executing, creating fibonacci
sequence of order %i with seeds %i, %i' % ( goal.order,
self._feedback.sequence[0], self._feedback.sequence[1]))

    # starts calculating the Fibonacci sequence
    fibonacciOrder = goal.order
    for i in xrange(1, fibonacciOrder):

      # check that preempt (cancelation) has not been requested by the
action client
      if self._as.is_preempt_requested():
        rospy.loginfo('The goal has been cancelled/preempted')
        # the following line, sets the client in preempted state (goal
cancelled)
        self._as.set_preempted()
        success = False
        # we end the calculation of the Fibonacci sequence
        break

      # builds the next feedback msg to be sent
```

```
        self._feedback.sequence.append(self._feedback.sequence[i] +
    self._feedback.sequence[i-1])
        # publish the feedback
        self._as.publish_feedback(self._feedback)
        # the sequence is computed at 1 Hz frequency
        r.sleep()

      # at this point, either the goal has been achieved (success==true)
      # or the client preempted the goal (success==false)
      # If success, then we publish the final result
      # If not success, we do not publish anything in the result
      if success:
        self._result.sequence = self._feedback.sequence
        rospy.loginfo('Succeeded calculating the Fibonacci of order %i'
    % fibonacciOrder )
        self._as.set_succeeded(self._result)

    if __name__ == '__main__':
      rospy.init_node('fibonacci')
      FibonacciClass()
      rospy.spin()
```

Code Explanation Python Program: {4.11a}
In this case, the action server is using an action message definition called **Fibonacci.action**. That message has been created by ROS into its **actionlib_tutorials** package.

```
[ ]: from actionlib_tutorials.msg import FibonacciFeedback,
     FibonacciResult, FibonacciAction
```

Here we are importing the message objects generates by this **Fibonacci.action** file.

```
[ ]: _feedback = FibonacciFeedback()
     _result   = FibonacciResult()
```

Here, we are creating the message objects that will be used for publishing the **feedback** and the **result** of the action.

```
[ ]: def __init__(self):
         # creates the action server
         self._as = actionlib.SimpleActionServer("fibonacci_as",
     FibonacciAction, self.goal_callback, False)
         self._as.start()
```

This is the constructor of the class. Inside this constructor, we are creating an Action Server that will be called **"fibonacci_as"**, that will use the Action message **FibonacciAction**, and that will

have a callback function called **goal_callback**, that will be activated each time a new goal is sent to the Action Server.

```
[ ]: def goal_callback(self, goal):

        r = rospy.Rate(1)
        success = True
```

Here me define the **goal callback** function. Each time a new goal is sento to the Action Server, this function will be called.

```
[ ]: self._feedback.sequence = []
     self._feedback.sequence.append(0)
     self._feedback.sequence.append(1)

     rospy.loginfo('"fibonacci_as": Executing, creating fibonacci sequence
     of order %i with seeds %i, %i' % ( goal.order,
     self._feedback.sequence[0], self._feedback.sequence[1]))
```

Here we are **initializing** the Fibonacci sequence, and setting up the first values (seeds) of it. Also, we print data for the user related to the Fibonacci sequence the Action Server is going to calculate.

```
[ ]: fibonacciOrder = goal.order
     for i in xrange(1, fibonacciOrder):
```

Here, we start a loop that while go until the **goal.order** value is reached. This value is, obviously, the order of the Fibonacci sequence that the user has sent from the Action Client.

```
[ ]: if self._as.is_preempt_requested():
         rospy.loginfo('The goal has been cancelled/preempted')
         # the following line, sets the client in preempted state (goal
     cancelled)
         self._as.set_preempted()
         success = False
         # we end the calculation of the Fibonacci sequence
         break
```

We check if the goal has been cancelled (preempted). Remember you saw how to to preempt a goal in the previous Chapter.

```
[ ]: self._feedback.sequence.append(self._feedback.sequence[i] +
     self._feedback.sequence[i-1])
     self._as.publish_feedback(self._feedback)
     r.sleep()
```

Here, we are actually calculating the values of the Fibonacci sequence. You can check how a Fibonacci sequence is calculated here: Fibonacci sequence. Also, we keep publishing **feedback** each time a new value of the sequence is calculated.

```
[ ]: if success:
       self._result.sequence = self._feedback.sequence
       rospy.loginfo('Succeeded calculating the Fibonacci of order %i' %
     fibonacciOrder )
       self._as.set_succeeded(self._result)
```

If everything went OK, we publish the **result**, which is the whole Fibonacci sequence, and we set the Action as succedeed using the **set_succeeded()** function.

End Code Explanation Python Program: {4.11a}
Exercise 4.12a: Check Fibonacci action msg structure
Check the structure of the Fibonacci.action message definition by visiting the **action** directory of the **actionlib_tutorials** package.

Exercise 4.12b: Watch feedback and result topic messages output from the action server
Launch again the python code above {4.11a} to have the Fibonacci server running. Then, execute the following commands in their corresponding WebShells.

Execute in WebShell #1: Echo the result topic

```
[ ]: rostopic echo /fibonacci_as/result
```

Execute in WebShell #2: Echo the feedback topic

```
[ ]: rostopic echo /fibonacci_as/feedback
```

Execute in WebShell #3: Manually send the goal to your Fibonacci server, publishing directly to the topic (as you learned in the previous chapter)

```
[ ]: rostopic pub /fibonacci_as/goal
     actionlib_tutorials/FibonacciActionGoal [TAB][TAB]
```

Expected Result for Exercise 4.12b
After having called the action, the feedback topic should be publishing the feedback, and the result once the calculations are finished.

Data for Exercise 4.12b

You must be aware that the name of the messages (the class) used in the Python code are called **FibonacciGoal**, **FibonacciResult**, and **FibonacciFeedback**, while the name of the messages used in the topics are called **FibonacciActionGoal**, **FibonacciActionResult**, and **FibonacciActionFeedback**. Do not worry about that, just bear it in mind and use it accordingly.

Exercice 4.13: Create Package with Action Server that moves the Ardrone in the air, making a square
a) Create a package with an action server that makes the drone move in a square when called. b) Call the action server through the topics and observe the result and feedback. c) Base your code in the previous Fibonacci example {4.11a} and the client you did in Exercice 4.6 that moved the ardrone while taking pictures.

Expected Result for Exercice 4.13
The result must show the ardrone doing a square in the air when the action server is called, as shown in the animation beneath {Fig:4.6}
Data for Exercice 4.13

- The size of the side of the square should be specified in the goal message as an integer.
- The feedback should publish the current side (as a number) the robot is at while doing the square.
- The result should publish the total number of seconds it took the drone to do the square
- Use the Test.action message for that action server. Use the shell command find /opt/ros/kinetic/ -name Test.action to find where that message is defined. Then, analyze the different fields of the msg in order to learn how to use it in your action server. As you can see its in the package **actionlib**.

Solution Exercise 4.13
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions notebook for Unit3 Services Part1:Actions Part2 Solutions
Fig.4.6 - Ardrone moved through commands issed by an custom action server Ex 4.13

Ardrone moved through commands issed by an custom action server Ex 4.13

How to create your own action server message

It is always recommended that you use the action messages already provided by ROS. These can be found in the following ROS packages:

- actionlib
- Test.action
- TestRequest.action
- TwoInts.action
- actionlib_tutorials
- Fibonacci.action
- Averaging.action

However, it may happen that you need to create your own type. Let's learn how to do it.

To create your own custom action message you have to:

1.- Create an **action** directory within your package.

2.- Create your **Name.action** action message file.

- The Name of the action message file will determine later the name of the classes to be used in the **action server** and/or **action client**. ROS convention indicates that the name has to be camel-case.
- Remember the Name.action file has to contain three parts, each part separated by three hyphens.

```
[ ]: #goal
     package_where_message_is/message_type goal_var_name
     ---
     #result
     package_where_message_is/message_type result_var_name
     ---
     #feedback
     package_where_message_is/message_type feedback_var_name
```

- If you do not need one part of the message (for example, you don't need to provide feedback), then you can leave that part empty. But you must always specify the hyphen separtors.

3.- Modify the file CMakeLists.txt and the package.xml to include action message compilation. Read the detailed description below.

How to prepare CMakeLists.txt and package.xml files for custom action messages compilation

You have to edit two files in the package, in the same way that we explained for topics and services:

- CMakeLists.txt
- package.xml

Modification of CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- find_package()
- add_action_files()
- generate_messages()
- catkin_package()

I. find_package()

All of the packages needed to COMPILE the messages of topic, services, and actions go here. In package.xml, you have to state them as built.

```
[ ]: find_package(catkin REQUIRED COMPONENTS
            # your packages are listed here
            actionlib_msgs
     )
```

II. add_action_files()

This function will contain all of the action messages from this package (which are stored in the action folder) that need to be compiled. Place them beneath the FILES tag.

```
[ ]: add_action_files(
            FILES
            Name.action
     )
```

III. generate_messages()

The packages needed for the action messages compilation are imported here. Write the same here as you wrote in the find_package.

```
[ ]: generate_messages(
            actionlib_msgs
            # Your packages go here
     )
```

IV. catkin_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the **package.xml** file as **<exec_depend>**.

```
[ ]: catkin_package(
            rospy
            # Your package dependencies go here
     )
```

Summarizing, You should end with a **CMakeLists.txt** file similar to this:

```
[ ]: cmake_minimum_required(VERSION 2.8.3)
     project(my_custom_action_msg_pkg)

     ## Find catkin macros and libraries
     ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS
     xyz)
     ## is used, also find other catkin packages
     find_package(catkin REQUIRED COMPONENTS
       std_msgs
       actionlib_msgs
     )

     ## Generate actions in the 'action' folder
     add_action_files(
        FILES
        Name.action
      )

     ## Generate added messages and services with any dependencies listed
     here
     generate_messages(
        std_msgs actionlib_msgs
      )

     catkin_package(
     )

     ## Specify additional locations of header files
     ## Your package locations should be listed before other locations
     # include_directories(include)
     include_directories(
       ${catkin_INCLUDE_DIRS}
     )
```

Modification of package.xml:

1.- Add all of the packages needed to compile the messages.

If, for example, one of your variables in the **.action** file uses a message defined outside the std_msgs package, let's say **nav_msgs/Odometry**, you will need to import it. To do so, you would have to add as **<build_depend>** the **nav_msgs** package, adding the following line:

```
[ ]: <build_depend>nav_msgs<build_depend>
```

2.- On the other hand, if you need a package for the execution of the programs inside your package, you will have to import those packages as **<exec_depend>**, adding the following line:

```
[ ]: <build_export_depend>nav_msgs<build_export_depend>
     <exec_depend>nav_msgs<exec_depend>
```

When you compile custom action messages, it's **mandatory** to add the **actionlib_msgs** as build_dependency.

```
[ ]: <build_depend>actionlib_msgs</build_depend>
```

When you use Python, it's **mandatory** to add the **rospy** as **run_dependency**.

```
[ ]: <build_export_depend>rospy<build_export_depend>
     <exec_depend>rospy<exec_depend>
```

This is due to the fact that the rospy python module is needed in order to run all of your python ROS code.

Summarizing, you should end with a package.xml file similar to this:

```
[ ]: <?xml version="1.0"?>
     <package format="2">
       <name>my_custom_action_msg_pkg</name>
       <version>0.0.0</version>
       <description>The my_custom_action_msg_pkg package</description>
       <maintainer email="user@todo.todo">user</maintainer>
       <license>TODO</license>

       <buildtool_depend>catkin</buildtool_depend>
       <build_depend>actionlib</build_depend>
       <build_depend>actionlib_msgs</build_depend>
       <build_depend>rospy</build_depend>
       <build_depend>std_msgs</build_depend>
       <build_export_depend>actionlib</build_export_depend>
       <build_export_depend>actionlib_msgs</build_export_depend>
       <build_export_depend>rospy</build_export_depend>
       <exec_depend>actionlib</exec_depend>
       <exec_depend>actionlib_msgs</exec_depend>
       <exec_depend>rospy</exec_depend>

       <export>
       </export>
     </package>
```

Finally, when everything is correctly set up, you just have to compile:

```
[ ]: roscd; cd ..
     catkin_make
     source devel/setup.bash
     rosmsg list | grep Name
```

You will get an output to the last command, similar to this:

```
[ ]: my_custom_action_msg_pkg/NameAction
     my_custom_action_msg_pkg/NameActionFeedback
     my_custom_action_msg_pkg/NameActionGoal
     my_custom_action_msg_pkg/NameActionResult
     my_custom_action_msg_pkg/NameFeedback
     my_custom_action_msg_pkg/NameGoal
     my_custom_action_msg_pkg/NameResult
```

Note
Note that you haven't imported the std_msgs package anywhere. But you can use the messages declared there in your custom .actions. That's because this package forms part of the roscore file systems, so therefore, it's embedded in the compilation protocols, and no declaration of use is needed.

Actions Quiz

For evaluating this Quiz, we will ask you to perform different tasks. For each task, very **specific instructions** will be provided: name of the package, names of the launch files and Python scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions, since they will allow our automated correction system to properly test your Quiz, and assign a score to it. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

Create a Package with an action server with custom action message to move ardone

- The new action server will receive two words as a goal: UP or DOWN.
- When the action server receives the UP word, it will move the drone 1 meter up.
- When the action server receives the DOWN word, it will move the drone 1 meter down.
- As a feedback, it publishes once a second what action is taking place (going up or going down).

- When the action finishes, the result will return nothing.

Useful Data for the Quiz:

- You need to create a new action message with the following structure:

```
[ ]: string goal
     ---
     ---
     string feedback
```

- Since we are talking about a drone, you can specify Twist velocities in the three axes. You will need to do that in order to move the robot up and down.

Specifications

- The name of the package where you'll place all the code related to the Quiz will be **actions_quiz**.
- The name of the launch file that will start your Action Server will be **action_cutom_msg.launch**.
- The name of the action will be /**action_custom_msg_as**.
- The name of your Action message file will be **CustomActionMsg.action**.

Quiz Correction
When you have finished the Quiz, you can correct it in order to get a Mark. For that, just click on the following button at the top of this Notebook.
IMPORTANT
Quizes can only be done once. This means that, once you correct your Quiz, and get a score for it, you won't be able to do it again and improve your score. So, be sure enough when you decide to correct your Quiz!

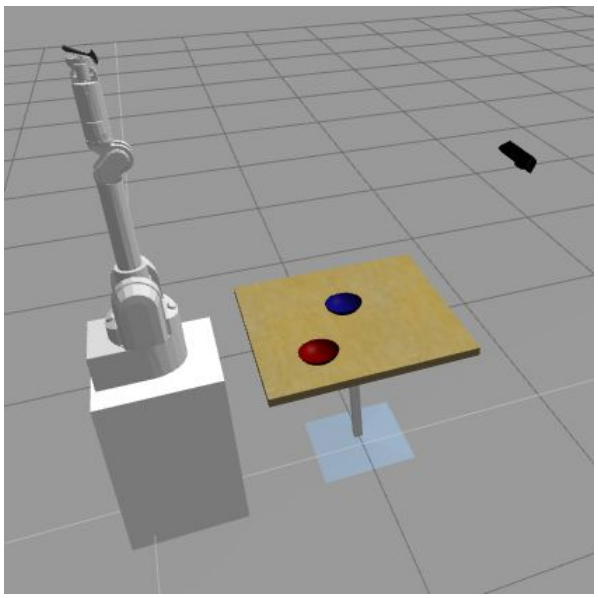
Additional information to learn more

ROS Actions: http://wiki.ros.org/actionlib
How actions work: http://wiki.ros.org/actionlib/DetailedDescription

Unit 5. Debugging Tools

ROS IN 5 DAYS

Unit 5: Debugging Tools



Iri Wam Robot

- ROSject Link: http://bit.ly/2n8FZcL
- Package Name: **iri_wam_gazebo**
- Launch File: **main.launch**

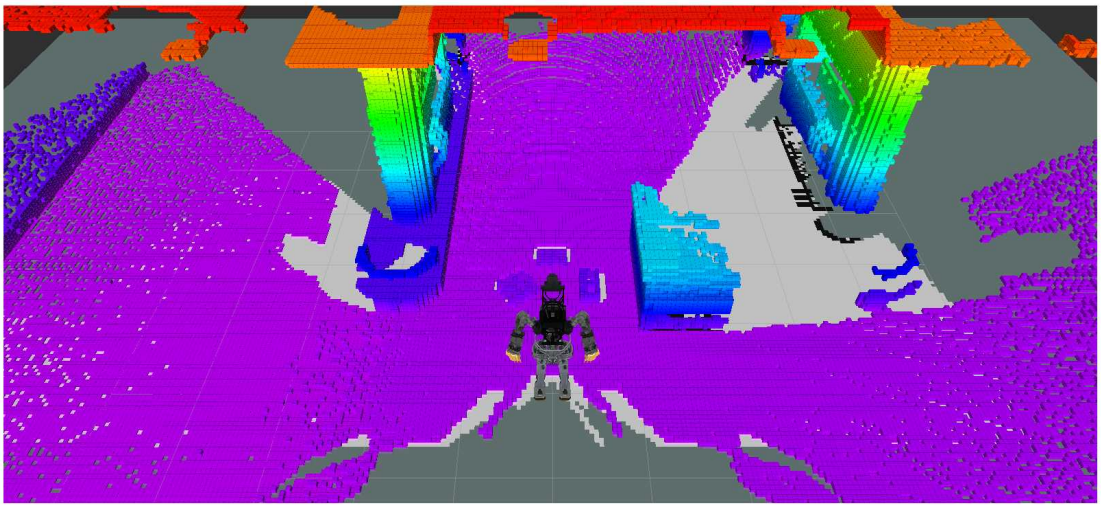
Estimated time of completion: 1.5 hours What will you learn with this unit?

- How can ROS What the F*ck help you debug
- Add Debugging ROS logs

- Filter ROS logs
- Record and replay sensory data
- Plot Topic Data
- Draw connections between different nodes of your system
- Basic use of RViz debugging tool

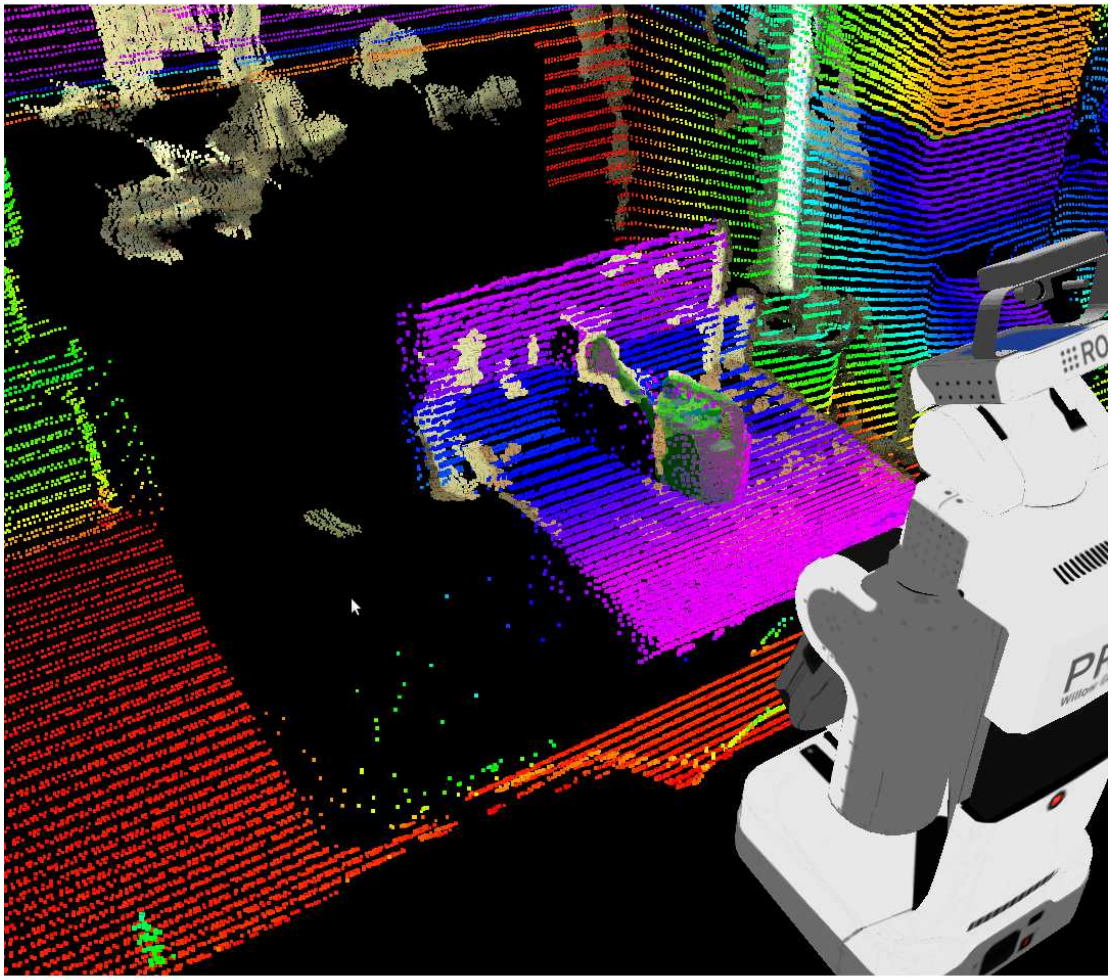
One of the most difficult, but important, parts of robotics is: **knowing how to turn your ideas and knowledge into real projects**. There is a constant in robotics projects: **nothing works as in theory**. Reality is much more complex and, therefore, you need tools to discover what is going on and find where the problem is. That's why debugging and visualization tools are essential in robotics, especially when working with complex data formats such as **images**, **laser-scans**, **pointclouds** or **kinematic data**. Examples are shown in {Fig-5.i} and {Fig-5.ii}.

Fig.5.i - Atlas Laser



Rviz Example 1

Fig.5.ii - PR2 Laser and PointCloud



Rviz Example 2

So here you will be presented with the most important tools for debugging your code and visualizing what is really happening in your robot system.

ROS What The F*ck!

It seems like a joke, but it isn't! Roswtf is a great tool to shed some light when you really don't know where to start solving a problem.

Go to the WebShell and type the following command:

Execute in WebShell #1

```
[ ]: roswtf
```

WebShell #1 Output

```
[ ]: user ~ $ roswtf
     the rosdep view is empty: call 'sudo rosdep init' and 'rosdep update'
     No package or stack in context
     ================================================================================
     ==========
     Static checks summary:

     Found 1 error(s).

     ERROR ROS Dep database not initialized: Please initialize rosdep
     database with sudo rosdep init.
     ================================================================================
     ==========
     Beginning tests of your ROS graph. These may take awhile...
     analyzing graph...
     ... done analyzing graph
     running graph rules...
     ... done running graph rules

     Online checks summary:

     Found 2 warning(s).
     Warnings are things that may be just fine, but are sometimes at fault

     WARNING The following node subscriptions are unconnected:
      * /gazebo:
        * /gazebo/set_model_state
        * /gazebo/set_link_state
        * /iri_wam/iri_wam_controller/follow_joint_trajectory/cancel
        * /iri_wam/iri_wam_controller/follow_joint_trajectory/goal
        * /iri_wam/e_stop
        * /iri_wam/iri_wam_controller/command

     WARNING These nodes have died:
      * urdf_spawner-4
```

In this particular case, it tells you {webshell-out-5.1} that the package rosdep hasn't been initialised, so you may have issues installing new ROS packages from the Internet. In this case, there is no problem because the system you are using (Robot Ignite Academy system) is not meant for installing anything. And this takes us to the question: What does roswtf do?

By default, it checks two ROS fields:

- File-system issues: It checks enviromental variables, packages, and launch files, among other things. It looks for any inconsistencies that might be errors. You can use the command

roswtf alone to get the system global status. But you can also use it to check particular launch files before using them.

Go to the WebShell and type the following command:

Execute in WebShell #1

```
[ ]: roslaunch iri_wam_aff_demo false_start_demo.launch
```

WebShell #1 Output

```
[ ]: user ~ $ roslaunch iri_wam_aff_demo false_start_demo.launch
     ... logging to
     /home/user/.ros/log/fd58c97c-9068-11e6-9889-02c6d37ebbf9/roslaunch-
     ip-172-31-20-234-12087.log
     Checking log directory for disk usage. This may take awhile.
     Press Ctrl-C to interrupt
     Done checking log file disk usage. Usage is <1GB.

     started roslaunch server http://ip-172-31-20-234:38217/

     SUMMARY
     ========

     PARAMETERS
      * /rosdistro: kinetic
      * /rosversion: 1.11.20

     NODES
       /
         iri_wam_aff_demo
     (iri_wam_reproduce_trajectory/iri_wam_aff_demo_node)
         iri_wam_reproduce_trajectory
     (iri_wam_reproduce_trajectory/iri_wam_reproduce_trajectory_node)

     ROS_MASTER_URI=http://localhost:11311

     core service [/rosout] found


     process[iri_wam_reproduce_trajectory-1]: started with pid [12111]
     ERROR: cannot launch node of type
     [iri_wam_reproduce_trajectory/iri_wam_aff_demo_node]: can't locate
     node[iri_wam_aff_demo_node] in package [iri_wam_reproduce_trajectory]
```

Any clue? It just tells you that it can't locate your iri_wam_aff_demo_node. Now try roswtf on that launch file to get a bit more information on what might be the problem:

Go to the WebShell and type the following commands:

Execute in WebShell #1

```
[ ]: roscd iri_wam_aff_demo/launch
     roswtf false_start_demo.launch
```

WebShell #1 Output

```
[ ]: user launch $ roswtf false_start_demo.launch
     the rosdep view is empty: call 'sudo rosdep init' and 'rosdep update'
     [rospack] Error: the rosdep view is empty: call 'sudo rosdep init' and
     'rosdep update'
     ================================================================================
     ==========
     Static checks summary:

     Found 2 error(s).

     ERROR ROS Dep database not initialized: Please initialize rosdep
     database with sudo rosdep init.
     ERROR Several nodes in your launch file could not be located. These
     are either typed incorrectly or need to be built:
      * node [iri_wam_aff_demo_node] in package
     [iri_wam_reproduce_trajectory]
```

To make **roswtf yourlaunchfile.launch** work, you need to go to the path where the file is. That's why you had to use the **roscd** command. The error shown above it is telling you that **roswtf** can't find the **iri_wam_aff_demo_node**. It states that because it's a binary, it might be that you haven't compiled it yet or that you spelt something wrong. Essentially, it's telling you that there is no node **iri_wam_aff_demo_node** in the package **iri_wam_reproduce_trajectory**.

- **Online/graph issues**: **roswtf** also checks for any inconsistencies in the connections between nodes, topics, actions, and so on. It warns you if something is not connected or it's connected where it shouldn't be. **These warnings aren't necessarily errors**. They are simply things that ROS finds odd. It's up to you to know if it's an error or if it's just the way your project is wired.

Go to the WebShell and type the following command:

Execute in WebShell #1

```
[ ]: roswtf
```

WebShell #1 Output

```
[ ]: user ~ $ roswtf
     the rosdep view is empty: call 'sudo rosdep init' and 'rosdep update'
     No package or stack in context
```

```
================================================================================
==========
Static checks summary:

Found 1 error(s).

ERROR ROS Dep database not initialized: Please initialize rosdep
database with sudo rosdep init.
================================================================================
==========
Beginning tests of your ROS graph. These may take awhile...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules

Online checks summary:

Found 2 warning(s).
Warnings are things that may be just fine, but are sometimes at fault

WARNING The following node subscriptions are unconnected:
 * /gazebo:
   * /gazebo/set_model_state
   * /gazebo/set_link_state
   * /iri_wam/iri_wam_controller/follow_joint_trajectory/cancel
   * /iri_wam/iri_wam_controller/follow_joint_trajectory/goal
   * /iri_wam/e_stop
   * /iri_wam/iri_wam_controller/command

WARNING These nodes have died:
 * urdf_spawner-4
```

You executed this command at the start, but you didn't pay attention to the lower part warnings. These warnings are Graph issues.

- It states that some subscribers are not connected to the topics that they are meant to be connected to: This is quite normal as they might be nodes that only connect at the start or in certain moments. No error here.
- The second warning states that a node has died: This also is quite normal as nodes, like in this case, that only run when they spawn objects, die after being used. But ROS is so kind that it lets you know, just in case it shouldn't work that way.

ROS Debugging Messages and Rqt-Console

You have used print() during this course to print information about how your programs are doing. Prints are the Dark Side of the Force, so from now on, you will use a more Jedi way of doing

things. LOGS are the way. Logs allow you to print them on the screen, but also to store them in the ROS framework, so you can classify, sort, filter, or else.

In logging systems, there are always levels of logging, as shown in {Fig-5.1}. In ROS logs case, there are five levels. Each level includes deeper levels. So, for example, if you use Error level, all the messages for Error and Fatal will be shown. If your level is Warning, then all the messages for levels Warning, Error and Fatal will be shown.

Fig.5.1 - LOG Levels

Use the Python module *rospy* to access the logging functionality in Python.
DEBUG ==> rospy.logdebug(msg, *args)*
INFO ==> rospy.loginfo(msg, args)
WARNING ==> rospy.logwarn(msg, *args)*
ERROR ==> rospy.logerr(msg, args)
FATAL ==> rospy.logfatal(msg, *args)

LOG Levels

Run the following Python code:

Example 5.1
Execute the following Python code logger_example.py by clicking on it and then clicking on the play button on the top right-hand corner of the IPython notebook.

You can also press [CTRL]+[Enter] to execute it.

Python Program {5.1}: logger_example.py

```
[ ]: #! /usr/bin/env python

     import rospy
     import random
     import time

     # Options: DEBUG, INFO, WARN, ERROR, FATAL
     rospy.init_node('log_demo', log_level=rospy.DEBUG)
     rate = rospy.Rate(0.5)

     #rospy.loginfo_throttle(120, "DeathStars Minute info:
     "+str(time.time()))

     while not rospy.is_shutdown():
         rospy.logdebug("There is a missing droid")
         rospy.loginfo("The Emperors Capuchino is done")
```

```
        rospy.logwarn("The Revels are coming time "+str(time.time()))
        exhaust_number = random.randint(1,100)
        port_number = random.randint(1,100)
        rospy.logerr(" The thermal exhaust port %s, right below the main
    port %s", exhaust_number, port_number)
        rospy.logfatal("The DeathStar Is EXPLODING")
        rate.sleep()
```

The best place to read all of the logs issued by all of the ROS Systems is: /rosout
Go to the WebShell and type the following command:

Execute in WebShell #1

```
[ ]: rostopic echo /rosout
```

You should see all of the ROS logs in the current nodes, running in the system.

Exercise 5.1
1- Create a package that has a launch that launches a python file with the previous code {logger_example.py}.

2- Change the LOG level in the rospy.init_node and see how the different messages are printed or not in /rosout topic, depending on the level selected.

As you can see, with only one node publishing every 2 seconds, the amount of data is big. Now imagine ten nodes, publishing image data, laser data, using the actions, services, and publishing debug data of your DeepLearning node. It's really difficult to get the logging data that you want. That's where rqt_console comes to the rescue.

Type in the WebShell #1 the roslaunch command for launching the Exercice 5.1 launch.
Go to the graphical interface window (hit the icon with a screen in the IDE)
And type in the WebShell #2 : rqt_console, to activate the GUI log printer.

Execute in WebShell #1

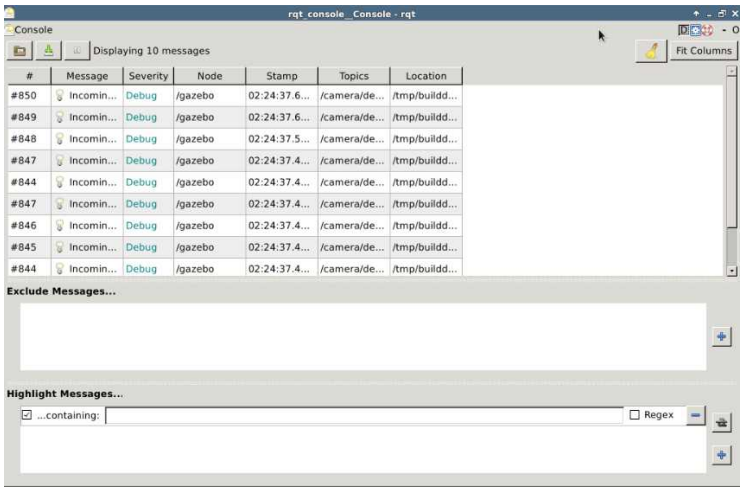
```
[ ]: roslaunch package_exercise_5p1 exercise_5p1.launch
```

Execute in WebShell #2

```
[ ]: rqt_console
```

You will get a window similar to {Fig-5.2} in the browser tab that opened when clicking on the icon:

Fig.5.2 - Rqt Console



Rqt Console

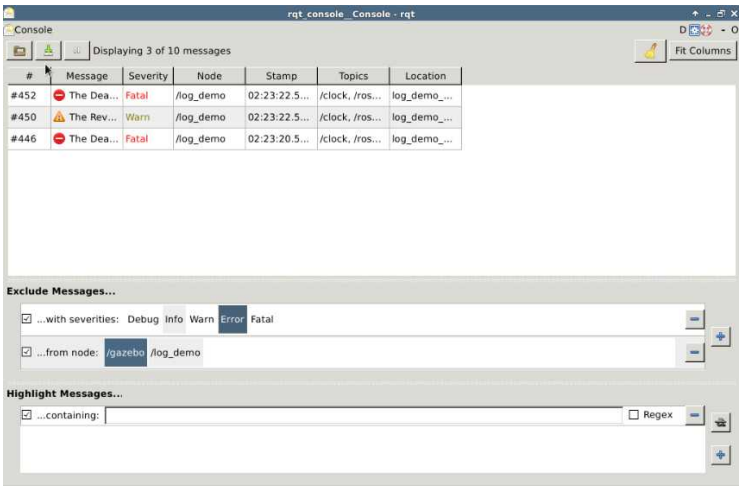
The rqt_console window is divided into three subpanels.

- The first panel outputs the logs. It has data about the message, severity/level, the node generating that message, and other data. Is here where you will extract all your logs data.
- The second one allows you to filter the messages issued on the first panel, excluding them based on criteria such as: node, severity level, or that it contains a certain word. To add a filter, just press the plus sign and select the desired one.
- The third panel allows you to highlight certain messages, while showing the other ones.

You have to also know that clicking on the tiny white gear on the right top corner, you can change the number of messages shown. Try to keep it as low as possible to avoid performance impact in your system.

Filter the logs so that you only see the Warning and Fatal messages of the node from exercice 5.1
You should see something like {Fig-5.3}:

Fig.5.3 - Rqt Console Filter



Rqt Console Filter

Plot topic data and Rqt Plot

This is a very common need in any scientific discipline, but especially important in robotics. You need to know if your inclination is correct, your speed is the right one, the torque readings in an arm joint is above normal, or the laser is having anomalous readings. For all these types of data, you need a graphic tool that makes some sense of all the data you are receiving in a fast and real-time way. Here is where rqt_plot comes in handy.

To Continue you should have stopped the Exercice 5.1 node launched in the WebShell #1

- **Go to the WebShell and type the following command to start moving the robot arm:**

Execute in WebShell #1

```
[ ]: roslaunch iri_wam_aff_demo start_demo.launch
```

- **Go to another Webshell and type the following command to see the positions and the effort made by each joint of the robot arm:**

Execute in WebShell #2

```
[ ]: rostopic echo /joint_states -n1
```

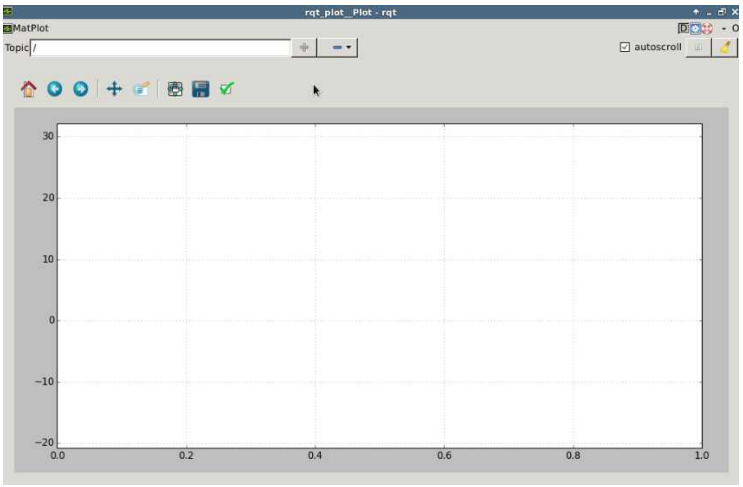
As you can probably see, knowing what's happening in the robots joints with only arrays of numbers is quite daunting. So let's use the rqt_plot command to plot the robot joints array of data.

Go to the graphical interface and type in a terminal the following command to open the rqt_plot GUI:
Remember to hit [CTRL]+[C] to stop the rostopic echo
Execute in WebShell #2

```
[ ]: rqt_plot
```

You will get a window similar to {Fig-5.4}:

Fig.5.4 - Rqt Plot



Rqt Plot

In the topic input located in the top-left corner of the window, you have to write the topic structure that leads to the data that you want to plot. Bear in mind that in order to be plotted, the topic has to publish a number. Once written, you press the PLUS SIGN to start plotting the Topic. In the case that we want to plot the robot joints, we need to plot the topic /joint_states, which has the following structure (that you can already get by extracting the topic message type with rostopic info , and afterwards using rosmsg show command from Unit 2):

```
[ ]: std_msgs/Header header
     string[] name
     float64[] position
     float64[] velocity
     float64[] effort
```

Then, to plot the velocity of the first joint of the robot, we would have to type /joint_states/velocity[0]. You can add as many plots as you want by pressing the “plus” button.

Exercise 5.2

- Plot in rqt_plot the effort made by the four first joints of the robot while it moves.

Node Connections and Rqt graph

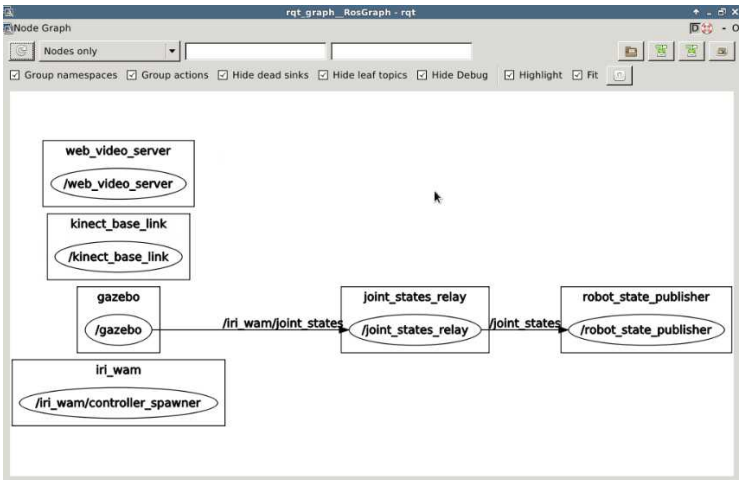
Is your node connected to the right place? Why are you not receiving data from a topic? These questions are quite normal as you might have experienced already with ROS systems. Rqt_graph can help you figure that out in an easier way. It displays a visual graph of the nodes running in ROS and their topic connections. It’s important to point out that it seems to have problems with connections that aren’t topics.

Go to the graphical interface and type in a terminal the following command to open the rqt_graph GUI:
Remember to have in the WebShell #1 the roslaunch iri_wam_aff_demo start_demo.launch
Execute in WebShell #2

```
[ ]: rqt_graph
```

You will get something like the following image:

Fig.5.5 - Rqt-Graph Result



Rqt Graph Result

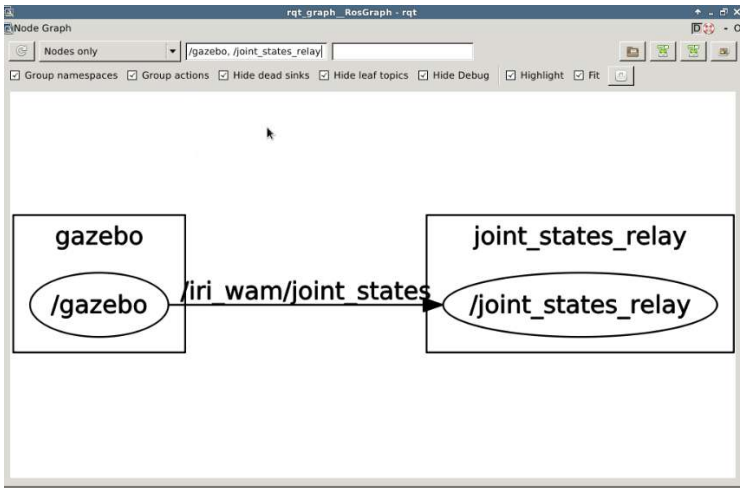
In the diagram {Fig-5.5}, you will be presented with all of the nodes currently running, connected by the topics they use to communicate with each other. There are two main elements that you need to know how to use:

- The refresh button: Which you have to press any time you have changed the nodes that are running:

- The filtering options: These are the three boxes just beside the refresh button. The first element lets you select between only nodes or topics. The second box allows you to filter by names of nodes.

Here is an example where you filter to just show the /gazebo and the /joint_states_relay {Fig-5.6}:

Fig-5.6 - Rqt-Graph Result Filtered /gazebo and /joint_states_relay



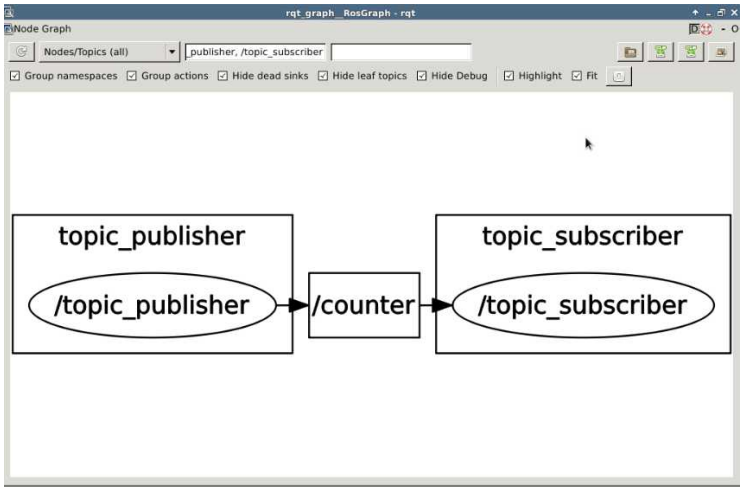
Rqt-Graph Result Filtered

Exercise 5.3

- Create a package that launches a simple Topic publisher and a subscriber, and filter the ros_graph output to show you only the two nodes and the topic you are interested in.

You should get something like this {Fig-5.7}:

Fig-5.7 - Rqt-Graph from Exercise 5.7



Rqt-Graph from Exercise 5.7

Record Experimental Data and Rosbags

One very common scenario in robotics is the following: You have a very expensive real robot, let's say R2-D2, and you take it to a very difficult place to get, let's say The Death Star. You execute

your mission there and you go back to the base. Now, you want to reproduce the same conditions to improve R2-D2s algorithms to open doors. But you don't have the DeathStar nor R2-D2. How can you get the same exact sensory readings to make your test? Well, you record them, of course! And this is what rosbag does with all the ROS topics generated. It records all of the data passed through the ROS topics system and allows you to replay it any time through a simple file.

The commands for playing with rosbag are:

- To Record data from the topics you want: rosbag record -O name_bag_file.bag name_topic_to_record1 name_topic_to_record2 ... name_topic_to_recordN
- To Extract general information about the recorded data: rosbag info name_bag_file.bag
- To Replay the recorded data: rosbag play name_bag_file.bag

  Replaying the data will make the rosbag publish the same topics with the same data, at the same time when the data was recorded.

Example 5.2
1- Go to the WebShell and type the following command to make the robot start moving if you don't haven't already:

Execute in WebShell #1

```
[ ]: roslaunch iri_wam_aff_demo start_demo.launch
```

2- Go to another WebShell and go to the src directory. Type the following command to record the data from the /laser_scan:

Execute in WebShell #2

```
[ ]: roscd; cd ../src
     rosbag record -O laser.bag laser_scan
```

This last command will start the recording of data.

3- After 30 seconds or so, a lot of data have been recorded, press [CTRL]+[C] in the rosbag recording WebShell #2 to stop recording. Check that there has been a laser.bag file generated and it has the relevant information by typing:

Execute in WebShell #2

```
[ ]: rosbag info laser.bag
```

4- Once checked, CTRL+C on the start_demo.launch WebShell #1 to stop the robot.

5- Once the robot has stopped, type the following command to replay the laser.bag (the -l option is to loop the rosbag infinitely until you CTRL+C):

Execute in WebShell #2

```
[ ]: rosbag play -l laser.bag
```

6- Go to WebShell #3 and type the following command to read the ranges[100] of Topic /laser_scan :

Execute in WebShell #3

```
[ ]: rostopic echo /laser_scan/ranges[100]
```

7- Type the following command in another WebShell , like WebShell #4. Then, go to the graphical interface and see how it plots the given data with rqt_plot:

Execute in WebShell #4

```
[ ]: rqt_plot /laser_scan/ranges[100]
```

Is it working? Did you find something odd? There are various things that are wrong, but it's important that you memorize this.

1) The first thing you notice is that when you echo the topic /**laser_scan** topic, you get sudden changes in values. Try getting some more information on who is publishing in the /**laser_scan** topic by writing in the WebShell:

Remember to hit [CTRL]+[C] if there was something running there first.
Execute in WebShell #3

```
[ ]: rostopic info /laser_scan
```

You will get something similar to this:

WebShell #3 Output

```
[ ]: user ~ $ rostopic info /laser_scan
     Type: sensor_msgs/LaserScan

     Publishers:
      * /gazebo (http://ip-172-31-27-126:59384/)
      * /play_1476284237447256367 (http://ip-172-31-27-126:41011/)

     Subscribers: None
```

As you can see, **TWO** nodes are publishing in the /**laser_scan** topic: **gazebo** (the simulation) and **play_x** (the rosbag play).

This means that not only is rosbag publishing data, but also the simulated robot.

So the first thing to do is to **PAUSE** the simulation, so that gazebo stops publishing laser readings.

For that, you have to execute the following command:

Execute in WebShell #3

```
[ ]: rosservice call /gazebo/pause_physics "{}"
```

And to **UnPAUSE** it again, just for you to know, just:

Execute in WebShell #3

```
[ ]: rosservice call /gazebo/unpause_physics "{}"
```

With this, you should have stopped all publishing from the simulated robot part and only left the rosbag to publish.

Now you should be able to see a proper /**laser_scan** plot in the rqt_plot.

Still nothing?

2) Check the time you have in the rqt_plot. Do you see that, at a certain point, the time doesn't keep on going?

That's because you have stopped the simulation so that the time is not running anymore, apart from the tiny time frame in the rosbag that you are playing now.

Once rqt_plot reaches the maximum time, it stops. It doesn't return to the start, and therefore, if the values change outside the last time period shown, you won't see anything.

Take a look also at the rosbag play information of your currently running rosbag player
WebShell #2 Output

```
[ ]: user ~ $ rosbag play -l laser.bag
     [ INFO ] [1471001445.5575545086]: Opening laser.bag

     Waiting 0.2 seconds after advertising topics... done.

     Hit space to toggle paused, or 's' to stop.
     [RUNNING] Bag Time: 61.676140 Duration: 41.099140 / 41.452000
```

In this example, you can see that the current time is 41.099 of 41.452 seconds recorded. And the rosbag time, therefore, will reach a maximum of around 62 seconds. 62 seconds, in this case, is the maximum time the rqt_plot will show, and will start around 20 seconds. Therefore, you have to always CLEAR the plot area with the clear button in rqt_plot. By doing a CLEAR, you should get something similar to {Fig-5.8}:

Fig-5.8 - RosBag Rqt-Plot

Rosbag in Rqt Plot

To summarize: To use rosbag files, you have to make sure that the original data generator (real robot or simulation) is NOT publishing. Otherwise, you will get really weird data (the collision between the original and the recorded data). You have to also keep in mind that if you are reading from a rosbag, time is finite and cyclical, and therefore, you have to clear the plot area to view all of the time period.

rosbag is especially useful when you don't have anything in your system (neither real nor simulated robot), and you run a bare roscore. In that situation, you would record all of the topics of the system when you do have either a simulated or real robot with the following command:

```
[ ]: rosbag record -a
```

This command will record ALL the topics that the robot is publishing. Then, you can replay it in a bare roscore system and you will get all of the topics as if you had the robot.

Before continuing any further, please check that you've done the following:

- You have stopped the rosbag play by going to the WebShell #2 where it's executing and [CTRL]+[C]
- You have unpaused the simulation to have it working as normal:

Execute in WebShell #3

```
[ ]: rosservice call /gazebo/unpause_physics "{}"
```

Visualize Complex data and Rviz

And here you have it. The HollyMolly! The Milenium Falcon! The most important tool for ROS debugging....RVIZ. RVIZ is a tool that allows you to visualize Images, PointClouds, Lasers, Kinematic Transformations, RobotModels... The list is endless. You even can define your own markers. It's one of the reasons why ROS got such a great acceptance. Before RVIZ, it was really difficult to know what the Robot was perceiving. And that's the main concept: RVIZ is NOT a simulation. I repeat: It's NOT a simulation. RVIZ is a representation of what is being published in the topics, by the simulation or the real robot.

RVIZ is a really complex tool and it would take you a whole course just to master it. Here, you will get a glimpse of what it can give you.

Remember that you should have unpaused the simulations and stopped the rosbag as described in the rosbag section.
1- Type in WebShell #2 the following command:

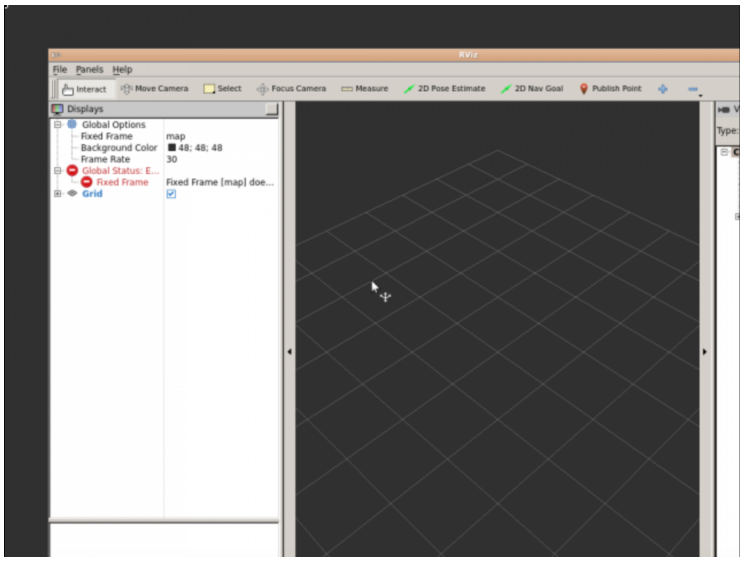
Execute in WebShell #2

```
[ ]: rosrun rviz rviz
```

2- Then go to the graphical interface to see the RVIZ GUI:

You will be greeted by a window like {Fig-5.9}:

Fig-5.9 - RVIZ Starting Window

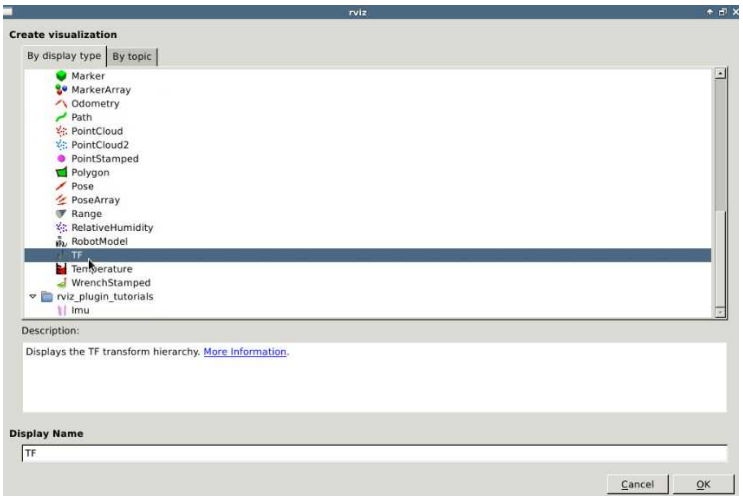


RVIZ Starting Window

NOTE: In case you don't see the lower part of Rviz (the Add button, etc.), double-click at the top of the window to maximize it. Then you'll see it properly.
You need only to be concerned about a few elements to start enjoying RVIZ.

- Central Panel: Here is where all the magic happens. Here is where the data will be shown. It's a 3D space that you can rotate (LEFT-CLICK PRESSED), translate (CENTER MOUSE BUTTON PRESSED) and zoom in/out (LEFT-CLICK PRESSED).
- Left Displays Panel: Here is where you manage/configure all the elements that you wish to visualize in the central panel. You only need to use two elements:
- In **Global Options**, you have to select the Fixed Frame that suits you for the visualization of the data. It is the reference frame from which all the data will be referred to.
- The **Add button**. Clicking here you get all of the types of elements that can be represented in RVIZ.

Go to RVIZ in the graphical interface and add a TF element. For that, click "Add" and select the element TF in the list of elements provided, as shown in {Fig-5.10}.
Fig-5.10 - RVIZ Add element



RVIZ Add element

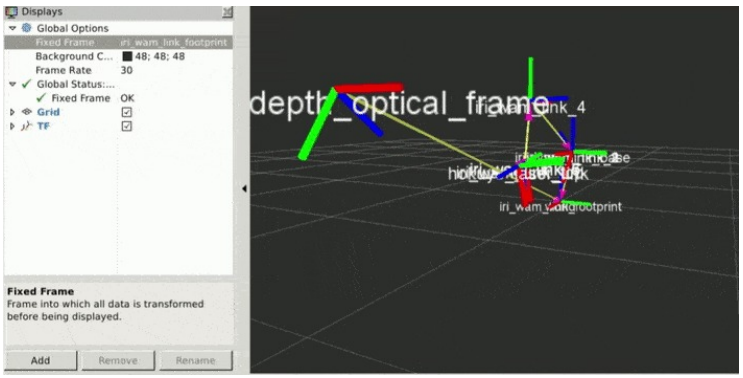
- Go to the RVIZ Left panel, select as Fixed Frame the **iri_wam_link_footprint** and make sure that the **TF** element checkbox is checked. In a few moments, you should see all of the Robots Elements Axis represented in the CENTRAL Panel.
- Now, go to a WebShell #1 and enter the command to move the robot:

Execute in WebShell #1

```
[ ]: roslaunch iri_wam_aff_demo start_demo.launch
```

You should see something like this:

Fig-5.11 - RVIZ TF

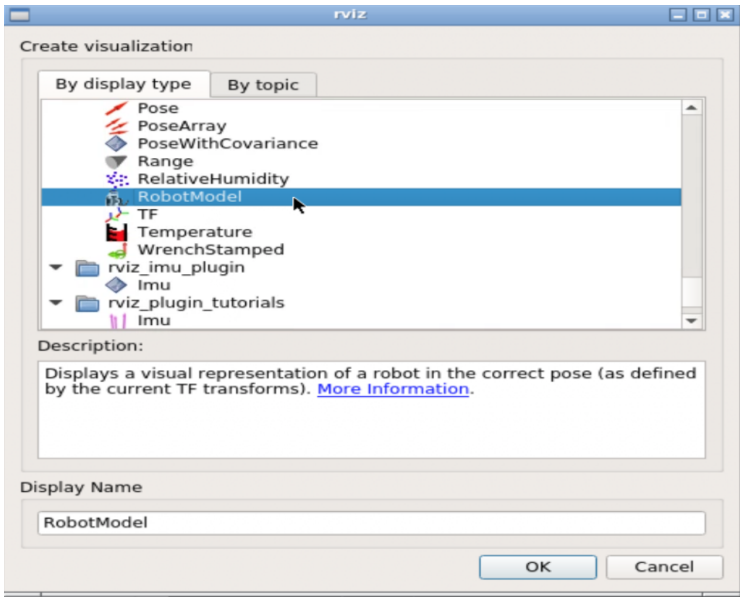


RVIZ TF

In {Fig-5.11}, you are seeing all of the transformations elements of the IRI Wam Simulation in real-time. This allows you to see exactly what joint transformations are sent to the robot arm to check if it's working properly.

- Now press "Add" and select RobotModel, as shown in {Fig-5.12}

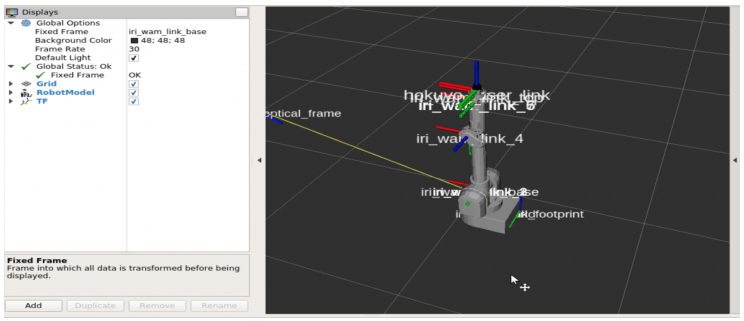
Fig-5.12 - RVIZ Add Robot Model



RVIZ Add Robot Model

You should see now the 3D model of the robot, as shown in {Fig-5.13}:

Fig-5.13 - RVIZ Robot Model + TF



RVIZ Robot Model + TF

Why can't you see the table? Or the bowl? Is there something wrong? Not at all! Remember: RVIZ is NOT a simulation, it represents what the TOPICS are publishing. In this case the models that are represented are the ones that the RobotStatePublisher node is publishing in some ROS topics. There is NO node publishing about the bowl or the table. Then how can you see the object around? Just like the robot does, through cameras, lasers, and other topic data. Remember: RVIZ shows what your robot is perceiving, nothing else.

Exercise 5.4
Add to RVIZ the visualization of the following elements:

- What the RGB camera from the Kinect is seeing. **TIP: The topic it has to read is /camera/rgb/image_raw. It might take a while to load the images, so just be patient.**

- What the Laser mounted at the end effector of the robot arm is registering. **TIP: You can adjust the appearance of the laser points through the element in the LEFT PANEL.**
- What the PointCloud Camera / Kinect mounted in front of the robot arm is registering. **TIP: You can adjust the appearance of the pointcloud points through the element in the LEFT PANEL. You should select points for better performance.**

TIP: You should have a similar result as the one depicted beneath: Notice that activating the pointcloud has a huge impact on the system performance. This is due to the huge quantity of data being represented. It's highly recommended to only use it with a high-end graphics card. Play around with the type of representations of the laser, size, and so on, as well as with the pointcloud configuration.

Fig-5.14 - RVIZ Robot with camera and laser
Fig-5.15 - RVIZ Robot with camera, laser, and point-cloud
Congratulations! Now you are ready to debug any AstroMech out there!

Additional information to learn more

roswtf: http://wiki.ros.org/roswtf
Ros Logging System: http://wiki.ros.org/rospy/Overview/Logging
rqt_console: http://wiki.ros.org/rqt_console
rqt_plot: http://wiki.ros.org/rqt_plot
rqt_graph: http://wiki.ros.org/rqt_graph
Rosbag: http://wiki.ros.org/rosbag
Rviz: http://wiki.ros.org/rviz

ROS IN 5 DAYS

Course Project

Sphero Robot

Run on ROSDS

- ROSject Link: http://bit.ly/2LO3XZf
- Package Name: **sphero_gazebo**
- Launch File: **main.launch**

Estimated time to completion: 10 hours What will you learn in this unit?

- Practice everything you learn through the course
- Put together everything you learn into a big project
- Create a main program

Win the Sphero Race!

In this project, you will have to make a Sphero robot move through a maze faster than the other students. The fastest one will win a prize.

For this goal, you will have to apply all the things that you have learned throughout the course. It is really important that you complete this race because all of the structures that you create for this project will be asked about in our official exam. Note: Our offical exam is only available for on-site and virtual classes at present.

Get the robot out of the maze as fast as possible and you will get the prize. The ideal would be for Spheros to come out cleanly, but it is possible that your robot may collide with the maze. You can use the collision detection to get data and help you in your strategy to get out.

Basically, in this project, you will have to:

1. Apply all the theory that is given in the course
2. Decide on a strategy to solve the problem
3. Implement this strategy in the simulation environment
4. Do as many tests as required in the simulation environment until it works

To achieve success in this project, please follow the 5 steps below that will provide clear instructions, and even solutions.

Also, remember to:

- Create your packages and code in the simulation environment as you have been doing throughout the course.
- Use the consoles to gather information about the status of the simulated robot.
- Use the IDE to create your programs and execute them through the consoles, observing the results on the simulation screen. You can use other consoles to watch calls to topics, services, or action servers.
- Everything that you create in this unit will be automatically saved in your space. You can come back to this unit at any time and continue with your work from where you left off.
- Every time you need to reset the position of the robot, just press the restart button in the simulation window.
- Use the debugging tools to try to find what is not working and why (for instance, the rviz tool is very useful for this purpose).

One final note: Because the program that you create should work with a real robot, if needed, you can't move the Sphero in a closed loop. This is because in reality, the circuit could be different, the robot could not be as responsive, there might be errors in the readings, and so on. So in this project, you should create a program that can cope with all of those uncertainties.

What does Sphero Provide to Program It?

So the main question is, what can you do with Sphero from a ROS programming point of view? Which sensors and actuators does Sphero provide that will allow you to do the maze test?

Good question! Sphero provides the following sensors and actuators:

Sensors

- **IMU sensor**: Sphero has an Inertial Measurement Unit (IMU), which provides information about acceleration and orientation. The value of the sensor is provided through the /sphero/imu/data3 topic.
- **Odometry**: The odometry of the robot can be accessed through the /odom topic.

Actuators

- **Speed**: You can send speed commands to move the robot through the /cmd_vel topic.

Now that you know the relevant topics of the robot, it is your job to figure out the types of messages, and how to use them, in order to make the robot do what you want it to do.

Ideas to Start Working On

This is a list of things that you can start with. You do not have to follow them. They are provided just in case you don't know where to start.

1. Start watching some of the messages the sensor topics are publishing. Try to get an idea about the information that they are providing. Move the robot in the simulation and see how those messages change their values. It is very important that you understand how changes in the robot produce changes in the topics.
2. Try to move the robot by sending messages to the /cmd_vel (either through the console or through python programs).
3. Observe how the topic messages change when the robot hits an obstacle.
4. Is the odometry trustworthy? Can you move the robot the exact amount, even when it collides with something?

Steps You Should Cover

These are the steps that you should follow throughout the duration of the project. These steps will ensure that you have practised and created all of the structures that will be asked about in the final exam of this course. If you perform all of the steps mentioned here, you will find completion of the exam to be achievable.

1. Step 1: Read and Write Topics (Dedicate 2 hours)
2. Step 2: Use topics through Services (Dedicate 3 hours)
3. Step 3: Use topics through Actions (Dedicate 4 hours)
4. Step 4: Create a main program to manage everything (Dedicate 1 hour)
5. EXTRA Step: How to use python modules from different packages (Not required, included here for information purposes only)

NOTE 1: We provide the solutions for each step. Do not watch unless you are really stuck.

NOTE 2: The fifth step may not be required, but we have found that some students organize their code in such a way that they need this step. We have provided it here just in case you need it, but you don't have to use it.

Step 1: Read and Write Topics

This step requires 3 actions:

1. Create a package called my_sphero_topics that will contain all of the programs related to the topics
2. Create a topic publisher that allows you to move the Sphero.
3. Create two topic subscribers that extract the data that you need from Odometry and the IMU.

So, let's get start on them!

1. Create package my_sphero_topics, with rospy as dependency.

2. Create the Topic Publisher to move Sphero.

To move Sphero, you need to publish in the topic /**cmd_vel**. It is important that you always encapsulate your topic subscribers and publishers inside classes. This will allow you to store values and manage the publishing callbacks easily.

First, you have to see if there is a topic like /cmd_vel running. Note: It will not always be this simple. With real robots, you will need to access the code to see what the name of the topic is that moves the robot, or even use rostopic info /name_of_topic to know which one it could be.

Execute in WebShell #1

```
[ ]: rostopic list
```

As you can see, there is a /cmd_vel topic. You, then, have to extract the type of message that /cmd_vel uses.

Execute in WebShell #1

```
[ ]: rostopic info /cmd_vel
```

Test it to see if it works py publishing different values:

Execute in WebShell #1

```
[ ]: rostopic pub /cmd_vel message_type_of_cmd_vel [TAB][TAB]
```

Once you have the information, you are ready to create the python class. Create a python file in the src folder of the package that you just created, “my_sphero_topics”. This file has to have not only the class, but also a way of testing that the class works.

Here is an example of a way in which this could be done (DO NOT watch unless you are stuck):

```
[ ]: #! /usr/bin/env python

    import rospy
    from geometry_msgs.msg import Twist


    class CmdVelPub(object):
        def __init__(self):
            self._cmd_vel_pub = rospy.Publisher('/cmd_vel', Twist,
    queue_size=1)
            self._twist_object = Twist()
            self.linearspeed = 0.2
            self.angularspeed = 0.5

        def move_robot(self, direction):
            if direction == "forwards":
                self._twist_object.linear.x = self.linearspeed
                self._twist_object.angular.z = 0.0
            elif direction == "right":
                self._twist_object.linear.x = 0.0
                self._twist_object.angular.z = self.angularspeed
            elif direction == "left":
                self._twist_object.linear.x = 0.0
                self._twist_object.angular.z = -self.angularspeed
            elif direction == "backwards":
                self._twist_object.linear.x = -self.linearspeed
                self._twist_object.angular.z = 0.0
            elif direction == "stop":
                self._twist_object.linear.x = 0.0
                self._twist_object.angular.z = 0.0
            else:
                pass
```

```
            self._cmd_vel_pub.publish(self._twist_object)


    if __name__ == "__main__":
        rospy.init_node('cmd_vel__publisher_node')
        cmd_publisher_object = CmdVelPub()

        rate = rospy.Rate(1)

        ctrl_c = False
        def shutdownhook():
            # works better than the rospy.is_shut_down()
            global ctrl_c
            global twist_object
            global pub

            rospy.loginfo("shutdown time!")

            ctrl_c = True
            cmd_publisher_object.move_robot(direction="stop")

        rospy.on_shutdown(shutdownhook)

        while not ctrl_c:
            cmd_publisher_object.move_robot(direction="forwards")
            rate.sleep()
```

3. Create Two Topic Subscribers that extract the data that you need from the Odometry and the IMU.

To get the Odometry data and the IMU data, you need to read the appropriate topics. Try to find them on your own by typing:

Execute in WebShell #1

```
[ ]: rostopic list
```

Have you found them? What type of message do they use? Are they the same as the ones listed here?

/sphero/imu/data3, type = sensor_msgs/Imu /odom, type = nav_msgs/Odometry
Are they publishing? What's the data like?

Execute in WebShell #1

```
[ ]: rostopic echo name_of_topic
```

Once you have the information, you are ready to create the python classes for each one. Create two different python files in the src folder of the package that you just created, “my_sphero_topics.” These files have to have not only the class, but also a way of testing the class objects.

Remember that you need to move the Sphero to see the changes in both the /odom and the /sphero/imu/data3 topics. Use the previously created program to move it.
Here is an example of how this could be done (DO NOT watch unless you are stuck):

```
[ ]: #! /usr/bin/env python

     import rospy
     from nav_msgs.msg import Odometry

     class OdomTopicReader(object):
         def __init__(self, topic_name = '/odom'):
             self._topic_name = topic_name
             self._sub = rospy.Subscriber(self._topic_name, Odometry,
     self.topic_callback)
             self._odomdata = Odometry()

         def topic_callback(self, msg):
             self._odomdata = msg
             rospy.logdebug(self._odomdata)

         def get_odomdata(self):
             """
             Returns the newest odom data

             std_msgs/Header header
               uint32 seq
               time stamp
               string frame_id
             string child_frame_id
             geometry_msgs/PoseWithCovariance pose
               geometry_msgs/Pose pose
                 geometry_msgs/Point position
                   float64 x
                   float64 y
                   float64 z
                 geometry_msgs/Quaternion orientation
                   float64 x
                   float64 y
                   float64 z
                   float64 w
               float64[36] covariance
             geometry_msgs/TwistWithCovariance twist
```

```
            geometry_msgs/Twist twist
              geometry_msgs/Vector3 linear
                float64 x
                float64 y
                float64 z
              geometry_msgs/Vector3 angular
                float64 x
                float64 y
                float64 z
            float64[36] covariance

        """
        return self._odomdata

if __name__ == "__main__":
    rospy.init_node('odom_topic_subscriber', log_level=rospy.INFO)
    odom_reader_object = OdomTopicReader()
    rospy.loginfo(odom_reader_object.get_odomdata())
    rate = rospy.Rate(0.5)

    ctrl_c = False
    def shutdownhook():
        # works better than the rospy.is_shut_down()
        global ctrl_c
        print "shutdown time!"
        ctrl_c = True

    rospy.on_shutdown(shutdownhook)

    while not ctrl_c:
        data = odom_reader_object.get_odomdata()
        rospy.loginfo(data)
        rate.sleep()
```

The /**sphero/imu/data3** topic will be done in exactly the same way. So try it yourself!

Step 2: Use Topics through Services

Now you need to take it a step further. Instead of having a topic subscriber/publisher on its own, you need to create a service that reads from topics. You have to do the following: Create a service that, when called, tells you if the robot has crashed or not, through the imu data. It also has to return some extra information, such as in what direction to move now that it has crashed.

We divided this into 3 tasks:

1. Determine the data
2. Modify the subscriber
3. Create the Server and Client

1. Determine data

1. Determine what data input you need (request)
2. Determine what data you want the service to return (response)

Then, you have to search for a service message that is already done in the system. You can find them in the std_srvs or rospy_tutorials package. You can also find other service messages created in non-standard packages. Bear in mind that using packages that might not be installed in ROS systems, or third party packages, is not recommened at this point. In this case, it is better to just generate your own service messages and use the std_msgs message types. It is always preferable to use messages that are already done, just because it's faster and you don't have to deal with compilation.

In this case, you need a service message with the following structure (DO NOT watch unless you are stuck):

```
[ ]: # request, Empty because no data is needed
     ---
     #response
     bool movement_successfull
     string extra_data
```

In this case, you have a service message that has this exact structure. It's in the std_srvs package and is called Trigger.srv. This is no coincidence. This service structure is very useful because normally you would ask a service to give you data, without providing any input.

So you just need to create a service that uses this Trigger.srv, that reads from the imu topic and tells you if you have crashed or not. It will also tell you based in the imu crash data, in which direction to move to now.

2. Modify the /sphero/imu/data3 topic susbcriber

Now you have to modify the /sphero/imu/data3 topic susbcriber for it to be able to tell you from which direction the crash occurred. Here is how it could be done (DO NOT watch unless you are stuck):

When you need to use an object from another python file, it has to be in the same package. Using python modules from other packages is not as easy as it may seem.

```
[ ]: #! /usr/bin/env python

     import rospy
```

```
from sensor_msgs.msg import Imu


class ImuTopicReader(object):
    def __init__(self, topic_name = '/sphero/imu/data3'):
        self._topic_name = topic_name
        self._sub = rospy.Subscriber(self._topic_name, Imu,
self.topic_callback)
        self._imudata = Imu()
        self._threshhold = 7.00
    def topic_callback(self, msg):
        self._imudata = msg
        rospy.logdebug(self._imudata)

    def get_imudata(self):
        """
        Returns the newest imu data

        std_msgs/Header header
          uint32 seq
          time stamp
          string frame_id
        geometry_msgs/Quaternion orientation
          float64 x
          float64 y
          float64 z
          float64 w
        float64[9] orientation_covariance
        geometry_msgs/Vector3 angular_velocity
          float64 x
          float64 y
          float64 z
        float64[9] angular_velocity_covariance
        geometry_msgs/Vector3 linear_acceleration
          float64 x
          float64 y
          float64 z
        float64[9] linear_acceleration_covariance

        """
        return self._imudata

    def four_sector_detection(self):
        """
        Detects in which four directions there is an obstacle that
made the robot crash
        Based on the imu data
        Axis:
```

```
            ^y
            |
           z0-->x

           """
           x_accel = self._imudata.linear_acceleration.x
           y_accel = self._imudata.linear_acceleration.y
           z_accel = self._imudata.linear_acceleration.z


           axis_list = [x_accel, y_accel, z_accel]

           max_axis_index = axis_list.index(max(axis_list))
           positive = axis_list[max_axis_index] >= 0
           significative_value = abs(axis_list[max_axis_index]) >
self._threshhold


           if significative_value:
               if max_axis_index == 0:
                   # Winner is in the x axis, therefore its a side crash
left/right
                   rospy.logwarn("[X="+str(x_accel))
                   rospy.loginfo("Y="+str(y_accel)+",
Z="+str(z_accel)+"]")
                   if positive:
                       message = "right"
                   else:
                       message = "left"

               elif max_axis_index == 1:
                   # Winner is the Y axis, therefore its a forn/back
crash
                   rospy.logwarn("[Y="+str(y_accel))
                   rospy.loginfo("X="+str(x_accel)+",
Z="+str(z_accel)+"]")
                   if positive:
                       message = "front"
                   else:
                       message = "back"
               elif max_axis_index == 2:
                   # Z Axixs is the winner, therefore its a crash that
made it jump
                   rospy.logwarn("[Z="+str(z_accel))
                   rospy.loginfo("X="+str(x_accel)+",
Y="+str(y_accel)+"]")

                   if positive:
```

```
                    message = "up"
                else:
                    message = "down"
            else:
                message = "unknown_direction"
        else:
            rospy.loginfo("X="+str(x_accel)+"Y="+str(y_accel)+",
Z="+str(z_accel)+"]")
            message = "nothing"

        return self.convert_to_dict(message)

    def convert_to_dict(self, message):
        """
        Converts the fiven message to a dictionary telling in which
direction there is a detection
        """
        detect_dict = {}
        # We consider that when there is a big Z axis component there
has been a very big front crash
        detection_dict = {"front":(message=="front" or message=="up"
or message=="down"),
                          "left":message=="left",
                          "right":message=="right",
                          "back":message=="back"}
        return detection_dict


if __name__ == "__main__":
    rospy.init_node('imu_topic_subscriber', log_level=rospy.INFO)
    imu_reader_object = ImuTopicReader()
    rospy.loginfo(imu_reader_object.get_imudata())
    rate = rospy.Rate(0.5)

    ctrl_c = False
    def shutdownhook():
        # works better than the rospy.is_shut_down()
        global ctrl_c
        print "shutdown time!"
        ctrl_c = True

    rospy.on_shutdown(shutdownhook)

    while not ctrl_c:
        data = imu_reader_object.get_imudata()
        rospy.loginfo(data)
        rate.sleep()
```

A way of evaluating the threshold that you consider relevant crash, or how the axis may be placed is by executing the code, and then controlling the Sphero with the:

```
[ ]: roslaunch sphero_gazebo keyboard_teleop.launch
```

3. Create the Server and client that tells you if there was a crash and what direction to move

Why do you need to create a client,too? Well, this is not needed for your core program to run, but its highly recommended because it allows you to test the server. So one way that it could be done is the following. Bear in mind that there is not only one way of doing things, and this is just a basic example (DO NOT watch unless you are stuck):

The Server:

```
[ ]: #! /usr/bin/env python

    import rospy
    from std_srvs.srv import Trigger, TriggerResponse
    from imu_topic_susbcriber import ImuTopicReader
    import time


    class CrashDirectionService(object):
        def __init__(self, srv_name='/crash_direction_service'):
            self._srv_name = srv_name
            self._imu_reader_object = ImuTopicReader()
            self.detection_dict = {"front":False, "left":False,
    "right":False, "back":False}
            self._my_service = rospy.Service(self._srv_name, Trigger ,
    self.srv_callback)

        def srv_callback(self, request):
            self.detection_dict =
    self._imu_reader_object.four_sector_detection()

            message = self.direction_to_move()

            rospy.logdebug("[LEFT="+str(self.detection_dict["left"])+",
    FRONT="+str(self.detection_dict["front"])+",
    RIGHT="+str(self.detection_dict["right"])+"]"+",
    BACK="+str(self.detection_dict["back"])+"]")
            rospy.logdebug("DIRECTION ==>"+message)

            response = TriggerResponse()
            """
```

```
            ---
            bool success   # indicate if crashed
            string message # Direction
            """
            response.success = self.has_crashed()
            response.message = message

            return response


        def has_crashed(self):
            for key, value in self.detection_dict.iteritems():
                if value:
                    return True

            return False

        def direction_to_move(self):

            if not self.detection_dict["front"]:
                message = "forwards"

            else:
                if not self.detection_dict["back"]:
                        message = "backwards"
                else:
                    if not self.detection_dict["left"]:
                        message = "left"
                    else:
                        if not self.detection_dict["right"]:
                            message = "right"
                        else:
                            message = "un_stuck"


            return message

    if __name__ == "__main__":
        rospy.init_node('crash_direction_service_server',
    log_level=rospy.INFO)
        dir_serv_object = CrashDirectionService()
        rospy.spin() # mantain the service open.
```

The Client:

```
[ ]: #! /usr/bin/env python
```

```
import rospy
from std_srvs.srv import Trigger, TriggerRequest
import sys

rospy.init_node('crash_direction_service_client') # initialise a ROS
node with the name service_client
service_name = "/crash_direction_service"
rospy.wait_for_service(service_name) # wait for the service client
/gazebo/delete_model to be running
direction_service = rospy.ServiceProxy(service_name, Trigger) # create
the connection to the service
request_object = TriggerRequest()

rate = rospy.Rate(5)

ctrl_c = False
def shutdownhook():
    # works better than the rospy.is_shut_down()
    global ctrl_c
    print "shutdown time!"
    ctrl_c = True

rospy.on_shutdown(shutdownhook)

while not ctrl_c:
    result = direction_service(request_object) # send through the
connection the name of the object to be deleted by the service
    """
    ---
    bool success   # indicate succes
    string message # informational,
    """
    if result.success:
        rospy.logwarn("Success =="+str(result.success)) # print the
result given by the service called
        rospy.logwarn("Direction To Go=="+str(result.message)) # print
the result given by the service called
    else:
        rospy.loginfo("Success =="+str(result.success)) # print the
result given by the service called
        rospy.loginfo("Direction To Go=="+str(result.message)) # print
the result given by the service called
    rate.sleep()
```

Step 3: Use Topics through Actions

Now you need to create an action that, when called, starts to save odometry data and checks if the robot has exited the maze. To accomplish that, you have to measure the distance from the

starting point to the current position. If the distance is bigger that the maze distance, you are out. A more elaborate one would be to consider the vector, and therefore know if you exited correctly or you just jumped over a wall. The Action should also stop if a certain amount of time has passed without the robot exiting the maze. The task is then:

Create an action server that finishes when it has detected that the robot has exited the maze, or has been working for a certain period of time. Use only the /odom topic subscriber.

We divided it into 3 subtasks:

1. Define the action message
2. Create the action server, action client, and algorithm to exit the maze
3. Test it

1. The first thing you have to think about is what kind of message you need for this action to work as intended.

You need to call this action, without any input. It doesn't need to give feedback because the only thing that matters is that it returns the needed data to evaluate the distance. It needs to return the data used to calculate the distance, for post-completion calculations. So the action message should look something like this (DO NOT watch this unless you are stuck):

```
[ ]: #goal, empty
     ---
     #result, Odometry array
     nav_msgs/Odometry[] result_odom_array
     ---
     #feedback, empty
```

This message is, as you can see, customized. Therefore, you will need to compile the package. The steps to do this are as follows:

Step 1: Create a new package called my_sphero_actions to store all of the action servers and the message. Step 2: Create an action directory, and within, an action message called record_odom.action. Step 3: Make all of the needed changes to the package.xml and CMake-Lists.txt files in order to correctly compile the action message. These are the two files as they should be, if the only external dependency of your my_sphero_actions package is:

CMakeLists.txt

```
[ ]: cmake_minimum_required(VERSION 2.8.3)
     project(my_sphero_actions)

     ## Find catkin macros and libraries
     ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS
```

```
xyz)
## is used, also find other catkin packages
## Here go the packages needed to COMPILE the messages of topic,
services and actions.
## in package.xml you have to state them as build
find_package(catkin REQUIRED COMPONENTS
  actionlib_msgs
  nav_msgs
)

## Generate actions in the 'action' folder
add_action_files(
   FILES
   record_odom.action
)

## Generate added messages and services with any dependencies listed
here
generate_messages(
   actionlib_msgs
   nav_msgs
 )

####################################
## catkin specific configuration ##
####################################
## Declare things to be passed to dependent projects
## State here all the packages that will be needed by someone that
executes something from your package
## All the packages stated here must be in the package.xml as
exec_depend
catkin_package(
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

package.xml

```
[ ]: <?xml version="1.0"?>
     <package format="2">
```

```
  <name>my_sphero_actions</name>
  <version>0.0.0</version>
  <description>The my_sphero_actions package</description>

  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>actionlib_msgs</build_depend>
  <build_depend>nav_msgs</build_depend>

  <build_export_depend>rospy</build_export_depend>
  <exec_depend>rospy</exec_depend>
  <build_export_depend>nav_msgs</build_export_depend>
  <exec_depend>nav_msgs</exec_depend>

  <export>
  </export>
</package>
```

Once you think you have it, execute the following commands in the WebShell:

Execute in WebShell #1

```
[ ]: roscd;cd ..
     catkin_make
     source deve/setup.bash
     rosmsg list | grep record_odom
```

Output in WebShell #1

```
[ ]: my_sphero_actions/record_odomAction
     my_sphero_actions/record_odomActionFeedback
     my_sphero_actions/record_odomActionGoal
     my_sphero_actions/record_odomActionResult
     my_sphero_actions/record_odomFeedback
     my_sphero_actions/record_odomGoal
     my_sphero_actions/record_odomResult
```

rosmsg list | grep record_odom: This command is listing all of the rosmsgs defined in the system and in your devel folder, and only filtering with grep command the ones with the name record_odom. All of the messages compiled from your packages are stored in the devel folder. This is one of the best ways to know that your action message has been correctly compiled and is accesible for all of your ROS system.

2. Create the action server, the client, and the program to compute out of the maze.

This action server has to start recording the /odom topic and stop when a certain time period has passed or the distance moved reaches a certain value.

When you need to use an object from another python file, it has to be in the same package. Using python modules from other packages is not as easy as it may seem.

Move your /odom topic susbcriber to your my_sphero_actions package. This way, your server can use it easily. Now create the action server and action client. It is the same here as it is in services: you don't need the client, but it's very useful to test the server and it also gives you a template for how to use it later in the core program.

Note: It may happen that when you test it, you get the following error:

ImportError: No module named my_sphero_actions.msg
This error is quite common when you generate your own messages. It doesn't find the my_sphero_actions.msg. But you have compiled it and doing the rosmsg list returns the correct output. Then, why? Because in order to allow your program to find the messages, you have to compile them and execute the source devel/setup.bash. This script sets not only the ROS environment, but also other systems related to message generation. So, in order to always make your messages work, do the following:

Execute in WebShell #1

```
[ ]: catkin_make
     source devel/setup.bash
```

Now you are ready to work on your action server. This is an example of how it could be done (DO NOT watch unless you are stuck):

```
[ ]: #! /usr/bin/env python

     import rospy
     import actionlib
     from my_sphero_actions.msg import record_odomFeedback,
     record_odomResult, record_odomAction
     from nav_msgs.msg import Odometry
     from odom_topic_subscriber import OdomTopicReader
     from odometry_analysis import check_if_out_maze


     class RecordOdomClass(object):

         def __init__(self, goal_distance):
             """
             It starts an action Server. To test it was created correctly,
     just rostopic the list and search for /rec_odom_as/...
             When launching, bear in mind that you should have:
```

```
        $catkin_make
        $source devel/setup.bash
        """
        # creates the action server
        self._as = actionlib.SimpleActionServer("/rec_odom_as",
record_odomAction, self.goal_callback, False)
        self._as.start()

        # Create an object that reads from the topic Odom
        self._odom_reader_object = OdomTopicReader()

        # create messages that are used to publish result
        self._result   = record_odomResult()

        self._seconds_recording = 120
        self._goal_distance = goal_distance

    def goal_callback(self, goal):

        success = True
        rate = rospy.Rate(1)

        for i in range(self._seconds_recording):
            rospy.loginfo("Recording Odom index="+str(i))
            # check that the preempt (cancelation) has not been
requested by the action client
            if self._as.is_preempt_requested():
                rospy.logdebug('The goal has been
cancelled/preempted')
                # the following line sets the client in a preempted
state (goal cancelled)
                self._as.set_preempted()
                success = False
                # we end the action loop
                break

            else:# builds the next feedback msg to be sent
                if not self.reached_distance_goal():
                    rospy.logdebug('Reading Odometry...')
                    self._result.result_odom_array.append(self._odom_r
eader_object.get_odomdata())
                else:
                    rospy.logwarn('Reached distance Goal')
                    # we end the action loop
                    break
            rate.sleep()

        # at this point, either the goal has been achieved
```

```
(success==true)
        # or the client preempted the goal (success==false)
        # If successful, then we publish the final result
        # If not successful, we do not publish anything in the result
        if success:
            self._as.set_succeeded(self._result)
            # Clean the Result Variable

        self.clean_variables()

    def clean_variables(self):
        """
        Cleans variables for the next call
        """
        self._result    = record_odomResult()

    def reached_distance_goal(self):
        """
        Returns True if the distance moved from the first instance of
recording until now has reached the self._goal_distance
        """
        return check_if_out_maze(self._goal_distance,
self._result.result_odom_array)



if __name__ == '__main__':
  rospy.init_node('record_odom_action_server_node')
  RecordOdomClass(goal_distance=2.0)
  rospy.spin()
```

Here is an example of how the odometry could be processed to know if the Sphero has exited the maze:

```
[ ]: #! /usr/bin/env python

import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Vector3
import math


class OdometryAnalysis(object):
    def __init__(self):
        pass

    def get_distance_moved(self, odmetry_data_list):
```

```
        distance = None

        if len(odmetry_data_list) >= 2 :
            start_odom = odmetry_data_list[0]
            end_odom = odmetry_data_list[len(odmetry_data_list)-1]

            start_position = start_odom.pose.pose.position
            end_position = end_odom.pose.pose.position

            rospy.loginfo("start_position ==>"+str(start_position))
            rospy.loginfo("end_position ==>"+str(end_position))


            distance_vector = self.create_vector(start_position,
end_position)
            rospy.loginfo("Distance Vector ==>"+str(distance_vector))

            distance = self.calculate_legth_vector(distance_vector)
            rospy.loginfo("Distance ==>"+str(distance))

        else:
            rospy.logerr("Odom array doesnt have the minimum number of
elements = "+str(len(odmetry_data_list)))

        return distance

    def create_vector(self, p1, p2):

        distance_vector = Vector3()
        distance_vector.x = p2.x - p1.x
        distance_vector.y = p2.y - p1.y
        distance_vector.z = p2.z - p1.z

        return distance_vector

    def calculate_legth_vector(self,vector):

        length = math.sqrt(math.pow(vector.x,2)+math.pow(vector.y,2)+m
ath.pow(vector.z,2))
        return length


def check_if_out_maze(goal_distance, odom_result_array):
    odom_analysis_object = OdometryAnalysis()
    distance =
odom_analysis_object.get_distance_moved(odom_result_array)
    rospy.loginfo("Distance Moved="+str(distance))
```

```
        # To exit we consider that each square in the floor is around
    0.5m, there fore to exit correctly
        # distance has to be sqrt (6*5 + 5*4) = 7.8
        return distance > goal_distance
```

And here is the client example:

```
[ ]: #! /usr/bin/env python

    import rospy
    import time
    import actionlib
    from my_sphero_actions.msg import record_odomGoal,
    record_odomFeedback, record_odomResult, record_odomAction
    from nav_msgs.msg import Odometry


    # definition of the feedback callback. This will be called when
    feedback
    # is received from the action server
    # it just prints a message indicating a new message has been received
    def feedback_callback(feedback):
        rospy.loginfo("Rec Odom Feedback feedback ==>"+str(feedback))


    def count_seconds(seconds):
        for i in range(seconds):
            rospy.loginfo("Seconds Passed =>"+str(i))
            time.sleep(1)

    # initializes the action client node
    rospy.init_node('record_odom_action_client_node')

    # create the connection to the action server
    client = actionlib.SimpleActionClient('/rec_odom_as',
    record_odomAction)

    rate = rospy.Rate(1)

    # waits until the action server is up and running
    rospy.loginfo('Waiting for action Server')
    client.wait_for_server()
    rospy.loginfo('Action Server Found...')

    # creates a goal to send to the action server
    goal = record_odomGoal()
```

```
    # sends the goal to the action server, specifying which feedback
    function
    # to call when feedback received
    client.send_goal(goal, feedback_cb=feedback_callback)



    # simple_state will be 1 if active, and 2 when finished. It's a
    variable, better use a function like get_state.
    #state = client.simple_state
    # state_result will give the FINAL STATE. Will be 1 when Active, and 2
    if NO ERROR, 3 If Any Warning, and 3 if ERROR
    state_result = client.get_state()

    """
    class SimpleGoalState:
        ACTIVE = 1
        DONE = 2
        WARN = 3
        ERROR = 4

    """

    rospy.loginfo("state_result: "+str(state_result))

    while state_result < 2:
        rospy.loginfo("Waiting to finish: ")
        rate.sleep()
        state_result = client.get_state()
        rospy.loginfo("state_result: "+str(state_result))


    state_result = client.get_state()
    rospy.loginfo("[Result] State: "+str(state_result))
    if state_result == 4:
        rospy.logerr("Something went wrong in the Server Side")
    if state_result == 3:
        rospy.logwarn("There is a warning in the Server Side")

    rospy.loginfo("[Result] State: "+str(client.get_result()))
```

3. Test it.

Launch it:

```
[ ]: rosrun my_sphero_actions rec_odom_action_server.py
```

Check that it's working:

```
[ ]: rosnode list | grep record_odom_action_server_node

[ ]: rostopic list | grep rec_odom_as
```

Launch the client to test if it really works.

1. Leave it running until the time runs out. It should return to the client side all of the /odom recordings until then.
2. Force that the distance goal gets reached. Use the **roslaunch sphero_gazebo keyboard_teleop.launch** to move the robot. It should also return the /odom topics recorded until then.

Step 4: Create a Main Program to Manage Everything

So, you finally have all the tools needed to create a main program that does the following:

1. Calls a service that tells you if it has crashed and in what direction you should move.
2. Moves the sphero based on the service response.
3. Checks if it has exited the maze or the time given has run out. If so, the program ends.

We have divided this into 3 sub-steps:

1. Create a package called my_sphero_main.

This package has to contain the main program and the python files that it needs. You might need to copy some files from other packages.

2. Create a launch file that launches the Action Server, the Service Server, and the main program.

Try it first with the node tags, and if it works, then generate a launch for the action_server and the service_server and use the include tag. Here is an example of how it could be done (DO NOT watch unless you are stuck):

```
[ ]: <launch>

       <node pkg ="my_sphero_actions"
             type="rec_odom_action_server.py"
```

```
        name="record_odom_action_server_node"
        output="screen">
    </node>

    <node pkg ="my_sphero_services"
        type="direction_service_server.py"
        name="crash_direction_service_server"
        output="screen">
    </node>

    <node pkg ="my_sphero_main"
        type="sphero_main.py"
        name="sphero_main_node"
        output="screen">
    </node>


</launch>
```

3. Create the main program.

Use all of the data and knowledge that you have extracted from the clients of the server and action to reuse as much code as possible. This is an example of how it could be done: (DO NOT watch unless you are stuck)

```
[ ]: #! /usr/bin/env python

    import rospy
    import actionlib
    from std_srvs.srv import Trigger, TriggerRequest
    from exam_action_rec_odom.msg import record_odomGoal,
    record_odomFeedback, record_odomResult, record_odomAction
    from cmd_vel_publisher import CmdVelPub
    from odometry_analysis import OdometryAnalysis
    from odometry_analysis import check_if_out_maze

    class ControlSphero(object):
        def __init__(self, goal_distance):
            self._goal_distance = goal_distance
            self.init_direction_service_client()
            self.init_rec_odom_action_client()
            self.init_move_sphero_publisher()

        def init_direction_service_client(self, service_name =
    "/crash_direction_service"):
            rospy.loginfo('Waiting for Service Server')
            rospy.wait_for_service(service_name) # wait for the service
    client /gazebo/delete_model to be running
```

```
        rospy.loginfo('Service Server Found...')
        self._direction_service = rospy.ServiceProxy(service_name,
Trigger) # create the connection to the service
        self._request_object = TriggerRequest()

    def make_direction_request(self):

        result = self._direction_service(self._request_object) # send
the name of the object to be deleted by the service through the
connection
        return result.message

    def init_rec_odom_action_client(self):
        self._rec_odom_action_client =
actionlib.SimpleActionClient('/rec_odom_as', record_odomAction)
        # waits until the action server is up and running
        rospy.loginfo('Waiting for action Server')
        self._rec_odom_action_client.wait_for_server()
        rospy.loginfo('Action Server Found...')
        self._rec_odom_action_goal = record_odomGoal()

    def send_goal_to_rec_odom_action_server(self):
self._rec_odom_action_client.send_goal(self._rec_odom_action_goal,
feedback_cb=self.rec_odom_feedback_callback)

    def rec_odom_feedback_callback(self,feedback):
        rospy.loginfo("Rec Odom Feedback feedback ==>"+str(feedback))

    def rec_odom_finished(self):

        has_finished = ( self._rec_odom_action_client.get_state() >= 2
)

        return has_finished

    def get_result_rec_odom(self):
        return self._rec_odom_action_client.get_result()

    def init_move_sphero_publisher(self):
        self._cmdvelpub_object = CmdVelPub()

    def move_sphero(self, direction):
        self._cmdvelpub_object.move_robot(direction)

    def got_out_maze(self, odom_result_array):
        return check_if_out_maze(self._goal_distance,
odom_result_array)
```

```
rospy.init_node("sphero_main_node", log_level=rospy.INFO)
controlsphero_object = ControlSphero(goal_distance=2.0)
rate = rospy.Rate(10)

controlsphero_object.send_goal_to_rec_odom_action_server()

while not controlsphero_object.rec_odom_finished():
    direction_to_go = controlsphero_object.make_direction_request()
    rospy.loginfo(direction_to_go)
    controlsphero_object.move_sphero(direction_to_go)
    rate.sleep()


odom_result = controlsphero_object.get_result_rec_odom()
odom_result_array = odom_result.result_odom_array

if controlsphero_object.got_out_maze(odom_result_array):
    rospy.loginfo("Out of Maze")
else:
    rospy.loginfo("In Maze")

rospy.loginfo("Sphero Maze test Finished")
```

Step 5: How to Use Python Modules from Different Packages

As you might have noticed, if you wanted to use something declared in a python module that was from another package, you had to copy it into your code. This is because in ROS, python module importing from other packages is not as easy as it might seem.

To learn how to do it, go through this example, divided in 4 sub-steps:

1. Create packages
2. Prepare common_pkg
3. Import in testing
4. Test everything

Let's say that you have a package named common_pkg and another package named testing.

1. Create those 2 packages.

Once done, edit both of them to be able to use a python class defined in common_pkg, in a python program located in the package testing.

2. Prepare common_pkg.

To prepare common_pkg so that anyone can use the python files in it, follow these steps:

- Execute the following commands on WebShell #1:

  Execute in WebShell #1

```
[ ]: roscd; cd ..
     cd src/common_pkg
     mkdir src/common_dir
     touch src/common_dir/_\_\_init\_\_.py
```

This will generate an extra directory named common_dir inside a special python file, which you are able to find through the python named **init**.py,.

- Once done, you create your python file inside the common_dir folder. Here is an example of what you could put in it:

Python Program {common_things.py}: common_things.py

```
[ ]: #! /usr/bin/env python

     import rospy
     from geometry_msgs.msg import Twist
     import time

     def cool(name):
         print('Cool ' + name)

     class CmdVelPub(object):
         def __init__(self):
             self._cmd_vel_pub = rospy.Publisher('/cmd_vel', Twist,
     queue_size=1)
             self._twist_object = Twist()
             self.linearspeed = 0.2
             self.angularspeed = 0.5

         def move_robot(self, direction):
             if direction == "forwards":
                 self._twist_object.linear.x = self.linearspeed
                 self._twist_object.angular.z = 0.0
             elif direction == "right":
                 self._twist_object.linear.x = 0.0
                 self._twist_object.angular.z = self.angularspeed
             elif direction == "left":
                 self._twist_object.linear.x = 0.0
```

```
            self._twist_object.angular.z = -self.angularspeed
        elif direction == "backwards":
            self._twist_object.linear.x = -self.linearspeed
            self._twist_object.angular.z = 0.0
        elif direction == "stop":
            self._twist_object.linear.x = 0.0
            self._twist_object.angular.z = 0.0
        else:
            pass

        self._cmd_vel_pub.publish(self._twist_object)
```

- Go to the root of the common_pkg

Execute in WebShell #1

```
[ ]: roscd common_pkg
```

- Create a file named setup.py, with the following content:

```
[ ]: ## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD

     from distutils.core import setup
     from catkin_pkg.python_setup import generate_distutils_setup

     # fetch values from package.xml
     setup_args = generate_distutils_setup(
         packages=['common_dir'],
         package_dir={'': 'src'},
     )

     setup(**setup_args)
```

- Now edit the CMakeLists.txt file, and uncomment the following line:

```
[ ]: ## Uncomment if the package has a setup.py
     catkin_python_setup()
```

- Compile and check that there is no error:

  Execute in WebShell #1

```
[ ]: roscd; cd ..
     catkin_make
     source devel/setup.bash
```

3. Import in testing.

Once you have set up the common_pkg, you just have to import the elements in the following way:

```
[ ]: from common_dir.common_things import cool, CmdVelPub
```

Note that you have imported with the name of the directory inside the src directory of your package. name of the package = common_pkg name of the directory and where you import = common_dir This is very important because it can lead to errors.

So here you have an example of the python file in the src directory of the testing package:

Python Program {test_import.py}: test_import.py

```
[ ]: #! /usr/bin/env python
     import rospy
     import time

     from common_dir.common_things import cool, CmdVelPub


     if __name__ == '__main__':
         cool('TheConstruct')

         stop_time = 1
         move_time = 3

         rospy.init_node('test_import', log_level=rospy.INFO)

         move_object = CmdVelPub()
         rospy.loginfo("Starting...")
         move_object.move_robot(direction="stop")
         time.sleep(stop_time)

         rospy.loginfo("Forwards...")
         move_object.move_robot(direction="forwards")
         time.sleep(move_time)

         rospy.loginfo("Stopping...")
         move_object.move_robot(direction="stop")
         time.sleep(stop_time)

         rospy.loginfo("Forwards...")
         move_object.move_robot(direction="backwards")
         time.sleep(move_time)

         rospy.loginfo("Stopping...")
         move_object.move_robot(direction="stop")
```

4. Test everything.

Now execute the **test_import.py** file:

Execute in WebShell #1

```
[ ]: rosrun testing test_import.py
```

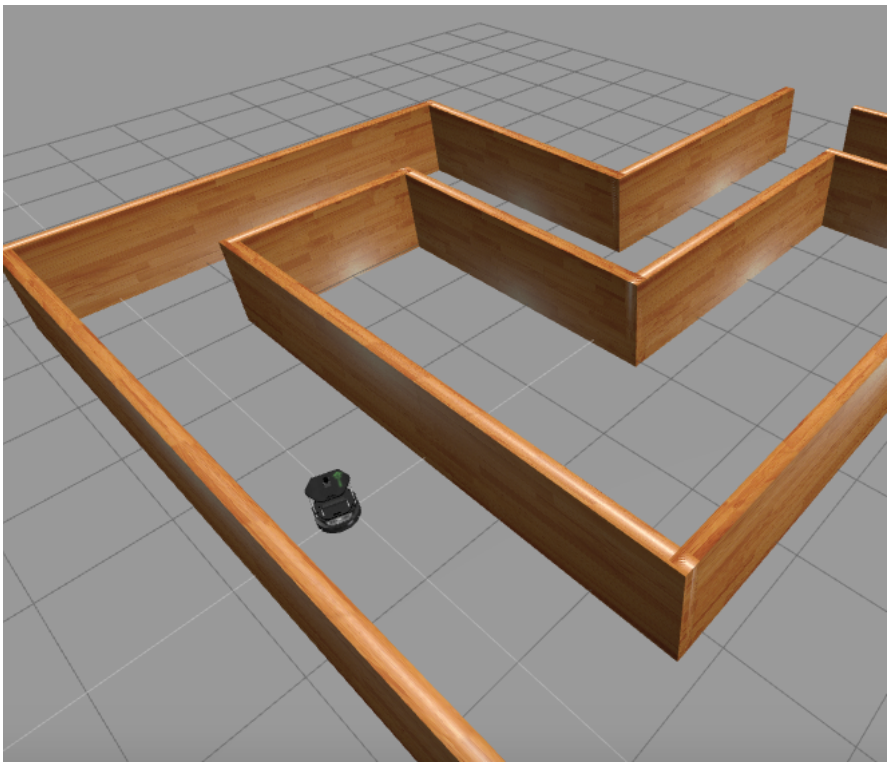
You should see the robot moving.

Conclusion

Now try to optimise your system. Play with the rates and the detection strategy. Play with the movement and the "AI" used to decide what to do. You especially need to know this project (except STEP 5) by heart because the exam will be very similar.

ROS IN 5 DAYS

Course Project



Turtlebot in Maze

- ROSject Link: http://bit.ly/2LQpSzi
- Package Name: **turtlebot_project**
- Launch File: **main.launch**

Estimated time to completion: 10 hours What you will learn with this unit?

- Practice everything you learn through the course
- Put together everything you learn into a big project
- Create a main program

Win the TurtleBot Race!

In this project, you will have to make a Turtlebot 2 Robot move along a maze faster than the other students. The fastest one will win a prize.

For this goal, you will have to apply all of the things that you are learning along the course. It's really important that you complete it because all of the structures that you create for this project will be asked about in our Official Exam. Get the robot out of the maze as fast as possible and you will get the prize. The ideal would be that Turtlebot goes out cleanly, but it may happen that you collide with the maze. You can use the collision detection to get data and help you in your strategy to get out.

Basically, in this project you will have to:

1. Apply all of the theory given in the course
2. Decide on a strategy to solve the problem
3. Implement this strategy in the simulation environment
4. Make as many tests as required in the simulation environment until it works

To achieve success in this project, we provide 5 steps below that you should follow with clear instructions and even solutions.

Also, remember to:

- Create your packages and code in the simulation environment, as you have been doing throughout the course.
- Use the consoles to gather information about the status of the simulated robot.
- Use the IDE to create your programs and execute them through the consoles, observing the results on the simulation screen. You can use other consoles to watch calls to topics, services, or action servers.
- Everything that you create in this unit will be automatically saved in your space. You can come back to this unit at any time and continue with your work from the point that you left it.
- Every time you need to reset the position of the robot just press the restart button in the simulation window.
- Use the debugging tools to try to find what is not working and why (for instance, the rviz tool is very useful for this purpose).

One final note: Because the program that you create should work with a real robot, if needed, you can't move the Turtlebot in a closed loop. This is because, in reality, the circuit could be different, the robot could not be as responsive, there might be errors in the readings, and so on. So in this project, you should create a program that can cope with all of these uncertainties.

What does Turtlebot Provide to Program It?

So the main question is, what can you do with Turtlebot from a ROS programming point of view? Which sensors and actuators does Turtlebot provide that will allow you to do the maze test?

Good question! Turtlebot provides the following sensors and actuators:

Sensors

- **Laser sensor**: Turtlebot has a 2D Laser which provides information about the environment you are in. The value of the sensor is provided through the topic /kobuki/laser/scan.
- **Odometry**: The odometry of the robot can be accessed through the /odom topic.

Actuators

- **Speed**: You can send speed commands to move the robot through the /cmd_vel topic.

Now that you know the relevant topics of the robot, it is your job to figure out the types of messages and how to use them in order to make the robot do the strategy you want it to do.

Ideas to Start Working On

Here is a list of things that you can start from. You do not have to follow them. They are provided just in case you don't know how to start.

1. Start watching some of the messages that the sensor topics are publishing. Try to get an idea about the information they are providing. Move the robot in the simulation and see how those messages change their values. It is very important that you understand how changes in the robot produce changes in the topics.
2. Try to move the robot sending messages to the /cmd_vel (either through the console or through python programs).
3. Observe how the messages of the topics change when the robot moves around the environment, or when it crashes into a wall.
4. Is the odometry trustworthy? Can you move the robot the exact amount even when it collides with something?

Steps You Should Cover

These are the steps that you should follow throughout the duration of the project. These steps will ensure that you have practised and created all of the structures asked for in the final exam of this course. If you perform all of the steps mentioned here, you will find the exam feasible.

1. Step 1: Read and Write Topics (Dedicate 2 hours)
2. Step 2: Use topics through Services (Dedicate 3 hours)
3. Step 3: Use topics through Actions (Dedicate 4 hours)
4. Step 4: Create a main program to manage everything (Dedicate 1 hour)
5. EXTRA Step: How to use python modules from different packages (Not required, included here for information purposes only)

NOTE: The 5th Step may not be required, but we have found some students that organize their code in such a way that they need this step. We have provided it here just in case you need it, but you don't have to use it.

Step 1: Read and Write Topics

This step has 3 actions for you to do:

1. Create a package called my_turtlebot_topics that will contain all of the programs related to topics
2. Create a topic publisher that allows you to move the Turtlebot.
3. Create two topic subscribers that extract the data that you need from the Odometry and the Laser.

So, let's get started.

1. Create package my_turtlebot_topics, with rospy as dependency.

2. Create the Topic Publisher to move Turtlebot.

To move Turtlebot, you need to publish in the topic /cmd_vel. It's important that you always encapsulate your topic subscribers and publishers inside classes. This will allow you to store values and manage the publishing callbacks easily.

First, you have to see if there is a topic like /cmd_vel running. Note: It will not always be this simple. In real robots, you will need to access the code to see what is the name of the topic that moves the robot, or even use rostopic info /name_of_topic to know which one it could be.

Execute in WebShell #1

```
[ ]: rostopic list
```

As you may see, there is a /cmd_vel. You, then, have to extract the type of message that /cmd_vel uses.

Execute in WebShell #1

```
[ ]: rostopic info /cmd_vel
```

Test that it works by publishing different values:

Execute in WebShell #1

```
[ ]: rostopic pub /cmd_vel message_type_of_cmd_vel [TAB][TAB]
```

Once you have the information, you are ready to create the python class. Create a python file in the src folder of the package you just created, “my_turtlebot_topics”. This file has to have not only the class, but also a way of testing that the class works.

3. Create Two Topic Subscribers that extract the data that you need from the Odometry and the Laser.

To get the Odometry data and the Laser data, you need to read the appropriate topics. Try finding them by yourself, first by typing:

Execute in WebShell #1

```
[ ]: rostopic list
```

Have you found them? What type of message do they use? Are they the same as the ones listed here? /kobuki/laser/scan, type = sensor_msgs/LaserScan /odom, type = nav_msgs/Odometry Are they publishing? What’s the data like?

Execute in WebShell #1

```
[ ]: rostopic echo name_of_topic
```

Once you have the information, you are ready to create the python classes for each one. Create two different python files in the src folder of the package that you just created, “my_turtlebot_topics”. These files have to have not only the class, but also a way of testing the class objects.

Remember that you need to move the Turtlebot to see the changes in both the /odom and the /kobuki/laser/scan topics. Use the previously created program to move it.

Step 2: Use Topics through Services

Now you need to take it a step further. Instead of having a topic subscriber/publisher on its own, you need to create a service that reads from the topics. You have to do the following: Create a service that, when called, it tells you if the robot is about to hit an obstacle, through the laser data. It also has to return some extra information, such as the direction that the robot should move in next.

We divided this into 3 tasks:

1. Determine data
2. Modify the subscriber
3. Create the Server and Client

1. Determine data

1. Determine what data input you need (request)
2. Determine what data you want the service to return (response)

Then, you have to search for a previously done service message in the system. You can find them in the std_srvs or rospy_tutorials package. You can also find other service messages created in non-standard packages. Bear in mind that using packages that might not be installed in ROS systems or from third party packages is not recommened at this point. In this case, it's better to just generate your own service messages and use the std_msgs message types. It's always preferable to use previously done messages, just because it's faster and you don't have to deal with compilation.

In this case, you need a service message of the following structure (DO NOT watch unless you are stuck):

```
[ ]: # request, Empty because no data is needed
     ---
     #response
     bool movement_successfull
     string extra_data
```

In this case, you have a service message that has this exact structure. It's in the std_srvs package and is called Trigger.srv. This is no coincidence. This service structure is very useful because, normally, you ask a service to give you data without providing any input.

So you just need to create a service that uses this Trigger.srv that reads from the laser topic and tells you if you are about to crash or not. It will also tell you, based on the laser data, in which direction to move in now.

2. Modify the /kobuki/laser/scan topic subscriber

Now you have to modify the /kobuki/laser/scan topic susbcriber to be able to tell you in what direction it is going to be the crash.

When you need to use an object from another python file, it has to be in the same package. Using python modules from other packages is not as easy as it may seem.

A way of evaluating the threshold that you consider it may be a potential crash, is by executing the code, and then controlling the Turtlebot with the: roslaunch turtlebot_teleop keyboard_teleop.launch

3. Create the Server and client that tell you if there is a potential crash, and in what direction to move

Why do you need to create a client, too? Well, this is not needed for your core program to run, but it's highly recommended because it allows you to test the server.

Step 3: Use Topics through Actions

Now you need to create an action that, when called, will start to save odometry data and check if the robot has exited the maze. To accomplish that, you have to measure the distance from the starting point to the current position. If the distance is larger than the maze distance, you are out. A more elaborate one would be to consider the vector, and therefore, know if you exited correctly or you just jumped over a wall. The Action should also stop in case a certain period of time has passed without the robot exiting the maze. The task is then:

Create an action server that finishes when it has detected that the robot has exited the maze, or has been working for a certain period of time. Use only the /odom topic subscriber.

We divided it into 3 subtasks:

- Define the action message
- Create the action server, action client, and algorithm to exit the maze
- Test it

1. The first thing you have to think about is what kind of message you need for this action to work as intended.

You need to call this action, without any input. It doesn't need to give feedback because the only thing that matters is that it returns the needed data to evaluate the distance. It needs to return the data used to calculate the distance for post-completion calculations. So the action message should look something like this (DO NOT watch this unless you are stuck):

```
[ ]: #goal, empty
     ---
     #result, Odometry array
     nav_msgs/Odometry[] result_odom_array
     ---
     #feedback, empty
```

This message is, as you can see, custom. Therefore, you will need to compile the package. The steps to do this are as follows:

Step 1: Create a new package called my_turtlebot_actions to store all the action servers and the message. Step 2: Create an action directory, and within, an action message called record_odom.action. Step 3: Make all of the needed changes to the package.xml and CMakeLists.txt files, in order to correctly compile the action message. These are the two files as they should be, if the only external dependency of your my_turtlebot_actions package is:

CMakeLists.txt

```
[ ]: cmake_minimum_required(VERSION 2.8.3)
     project(my_turtlebot_actions)

     ## Find catkin macros and libraries
     ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS
     xyz)
     ## is used, also find other catkin packages
     ## Here go the packages needed to COMPILE the messages of topic,
     services and actions.
     ## in package.xml you have to state them as build
     find_package(catkin REQUIRED COMPONENTS
       actionlib_msgs
       nav_msgs
     )

     ## Generate actions in the 'action' folder
     add_action_files(
        FILES
        record_odom.action
     )

     ## Generate added messages and services with any dependencies listed
     here
     generate_messages(
        actionlib_msgs
        nav_msgs
      )

     ###################################
     ## catkin specific configuration ##
     ###################################
```

```
## Declare things to be passed to dependent projects
## State here all the packages that will be needed by someone that
executes something from your package
## All the packages stated here must be in the package.xml as
exec_depend
catkin_package(
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

package.xml

```
[ ]: <?xml version="1.0"?>
     <package format="2">
       <name>my_turtlebot_actions</name>
       <version>0.0.0</version>
       <description>The my_turtlebot_actions package</description>

       <maintainer email="user@todo.todo">user</maintainer>
       <license>TODO</license>

       <buildtool_depend>catkin</buildtool_depend>
       <build_depend>actionlib_msgs</build_depend>
       <build_depend>nav_msgs</build_depend>

       <build_export_depend>rospy</build_export_depend>
       <build_export_depend>nav_msgs</build_export_depend>

       <exec_depend>rospy</exec_depend>
       <exec_depend>nav_msgs</exec_depend>

       <export>
       </export>
     </package>
```

Once you think you have it, execute the following commands in the WebShell:

Execute in WebShell #1

```
[ ]: roscd;cd ..
     catkin_make
```

```
source deve/setup.bash
rosmsg list | grep record_odom
```

Output in WebShell #1

```
[ ]: my_turtlebot_actions/record_odomAction
     my_turtlebot_actions/record_odomActionFeedback
     my_turtlebot_actions/record_odomActionGoal
     my_turtlebot_actions/record_odomActionResult
     my_turtlebot_actions/record_odomFeedback
     my_turtlebot_actions/record_odomGoal
     my_turtlebot_actions/record_odomResult
```

rosmsg list | grep record_odom: This command lists all of the rosmsgs defined in the system and in your devel folder, and only filters with the grep command the ones with the name record_odom. All of the messages compiled from your packages are stored in the devel folder. This is one of the best ways to know that your action message has been correctly compiled and is accessible for your entire ROS system.

2. Create the action server, the client, and the program to compute out of the maze.

This action server has to start recording the /odom topic and stop when a certain time period has passed, or the distance moved reaches a certain value.

When you need to use an object from another python file, it has to be in the same package. Using python modules from other packages is not as easy as it may seem.

Move your /odom topic subscriber to your my_turtlebot_actions package. This way, your server can use it easily. Now create the action server and action client. It is the same here as in services: you don't need the client but it's very useful to test the server, and it also gives you a template for how to use it later in the core program.

Note: It may happen that, when you test it, you get the following error:

ImportError: No module named my_turtlebot_actions.msg
This error is quite common when you generate your own messages. It doesn't find the my_turtlebot_actions.msg. But you have compiled it and doing the rosmsg list returns the correct output. Then, why? Because in order to allow your program to find the messages, you have to compile them and execute the source devel/setup.bash. This script sets not only the ROS environment, but also other systems related to message generation. So, in order to always make your messages work, do the following:

Execute in WebShell #1

```
[ ]: catkin_make
     source devel/setup.bash
```

Now you are ready to work on your action server.

3. Test it.

Launch it: rosrun my_turtlebot_actions rec_odom_action_server.py
Check that it's working: rosnode list | grep record_odom_action_server_node rostopic list | grep rec_odom_as
Launch the client to test that it really works.

1. Leave it running until the time runs out. It should return in the client side all of the /odom recordings up until then.
2. Force the distance goal to be reached. Use the roslaunch turtlebot_teleop keyboard_teleop.launch. It should return the /odom topics recorded up until then as well.

Step 4: Create a Main Program to Manage Everything

So, finally, you have all of the tools needed to create a main program that does the following:

1. Calls a service that tells you if it is going to crash and in what direction you should move.
2. Moves the turtlebot based on the service response.
3. Checks if it has exited the maze or the time given has run out. If so, the program ends.

We have divided this into 3 sub-steps:

1. Create a package called my_turtlebot_main.

This package has to contain the main program and the python files that it needs. You might need to copy some files of other packages.

2. Create a launch file that launches the Action Server, the Service Server, and the main program.

Try it first with the node tags and, if it works, then generate launched for the action_server and the service_server and use the include tag. Here is an example of how it could be done (DO NOT watch unless you are stuck):

```
[ ]: <launch>

       <node pkg ="my_turtlebot_actions"
             type="rec_odom_action_server.py"
```

```
        name="record_odom_action_server_node"
        output="screen">
    </node>

    <node pkg ="my_turtlebot_services"
        type="direction_service_server.py"
        name="crash_direction_service_server"
        output="screen">
    </node>

    <node pkg ="my_turtlebot_main"
        type="turtlebot_main.py"
        name="turtlebot_main_node"
        output="screen">
    </node>


</launch>
```

3. Create the main program.

Use all of the data and knowledge that you have extracted from the clients of the server and action, to reuse as much code as possible.

Step 5: How to Use Python Modules from Different Packages

As you might have noticed, if you wanted to use something declared in a python module that was from another package, you had to copy it into your code. This is because in ROS, python module importing from other packages is not as easy as it might seem.

To learn how to do it, you will go through this example, divided into 4 sub-steps:

1. Create the packages
2. Prepare common_pkg
3. Import in testing
4. Test everything

Let's say that you have a package named common_pkg and another package named testing.

1. Create those 2 packages.

Once done, edit both of them to be able to use a python class defined in common_pkg, in a python program located in the package testing.

2. Prepare common_pkg.

To prepare common_pkg so that anyone can use the python files in it, follow these steps:

- Execute the following commands on WebShell #1:

  Execute in WebShell #1

```
[ ]: roscd; cd ..
     cd src/common_pkg
     mkdir src/common_dir
     touch src/common_dir/__init__.py
```

This will generate an extra directory named common_dir, and inside a special python file, to be able to find it through python, named **init**.py,.

- Once done, you create your python file inside the common_dir folder. Here is an example of what you could put in it:

Python Program {common_things.py}: common_things.py

```
[ ]: #! /usr/bin/env python

     import rospy
     from geometry_msgs.msg import Twist
     import time

     def cool(name):
         print('Cool ' + name)

     class CmdVelPub(object):
         def __init__(self):
             self._cmd_vel_pub = rospy.Publisher('/cmd_vel', Twist,
     queue_size=1)
             self._twist_object = Twist()
             self.linearspeed = 0.2
             self.angularspeed = 0.5

         def move_robot(self, direction):
             if direction == "forwards":
                 self._twist_object.linear.x = self.linearspeed
                 self._twist_object.angular.z = 0.0
             elif direction == "right":
                 self._twist_object.linear.x = 0.0
                 self._twist_object.angular.z = self.angularspeed
             elif direction == "left":
```

```
            self._twist_object.linear.x = 0.0
            self._twist_object.angular.z = -self.angularspeed
        elif direction == "backwards":
            self._twist_object.linear.x = -self.linearspeed
            self._twist_object.angular.z = 0.0
        elif direction == "stop":
            self._twist_object.linear.x = 0.0
            self._twist_object.angular.z = 0.0
        else:
            pass

        self._cmd_vel_pub.publish(self._twist_object)
```

- Go to the root of the common_pkg

Execute in WebShell #1

```
[ ]: roscd common_pkg
```

- Create a file named setup.py, with the following content:

```
[ ]: ## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD

     from distutils.core import setup
     from catkin_pkg.python_setup import generate_distutils_setup

     # fetch values from package.xml
     setup_args = generate_distutils_setup(
         packages=['common_dir'],
         package_dir={'': 'src'},
     )

     setup(**setup_args)
```

- Now edit the CMakeLists.txt file, and uncomment the following line:

```
[ ]: ## Uncomment if the package has a setup.py
     catkin_python_setup()
```

- Compile and check that there is no error:

Execute in WebShell #1

```
[ ]: roscd; cd ..
     catkin_make
     source devel/setup.bash
```

3. Import in testing.

Once you have set up the common_pkg, you just have to import the elements in the following way:

```
[ ]: from common_dir.common_things import cool, CmdVelPub
```

Note that you import with the name of the directory inside the src directory of your package. name of the package = common_pkg name of the directory and where you import = common_dir This is very important because it can lead to errors.

So here you have an example of the python file in the src directory of the testing package:

Python Program {test_import.py}: test_import.py

```
[ ]: #! /usr/bin/env python
     import rospy
     import time

     from common_dir.common_things import cool, CmdVelPub


     if __name__ == '__main__':
         cool('TheConstruct')

         stop_time = 1
         move_time = 3

         rospy.init_node('test_import', log_level=rospy.INFO)

         move_object = CmdVelPub()
         rospy.loginfo("Starting...")
         move_object.move_robot(direction="stop")
         time.sleep(stop_time)

         rospy.loginfo("Forwards...")
         move_object.move_robot(direction="forwards")
         time.sleep(move_time)

         rospy.loginfo("Stopping...")
         move_object.move_robot(direction="stop")
         time.sleep(stop_time)

         rospy.loginfo("Forwards...")
         move_object.move_robot(direction="backwards")
         time.sleep(move_time)

         rospy.loginfo("Stopping...")
         move_object.move_robot(direction="stop")
```

4. Test everything.

Now execute the test_import.py:

Execute in WebShell #1

```
[ ]: rosrun testing <i>test_import.py
```

You should see the robot moving.

Conclusion

Now try to optimise your system. Play with the rates and the detection strategy. Play with the movement and the “AI” used to decide what to do. You especially have to know this project (except STEP 5) by heart because the exam will be very similar.

##
Solutions
Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.

Follow this link to open the solutions for the Turtlebot Project:Turtlebot Project Solutions

Appendix 1

ROS IN 5 DAYS

Appendix 1: Installing ROS

How to install ROS in my local computer?

During the whole Course, you have already been using ROS with its full capabilities. In fact, you are almost a master in ROS programming! And that is because in our Robot Ignite Academy, you already have everything installed and set up, so that you can go straight and focus in learning the really important things of ROS: how to apply it to interact with robots. But what if you want now to apply everything you have learned during the Course, in your local computer?

Well, the first step would be, of course, to install ROS in your local computer. And that is what you are going to do during this appendix. For this case, we are going to explain the steps to install and setup the same environment that you've been using during the whole Course. This is, a **ROS Kinetic** distribution installed in an **Ubuntu 16.04 Xenial** machine. If you want to install a different version, or you are using a different machine, please refer to the official documentation, here: http://wiki.ros.org/kinetic/Installation
With the proper introductions made, let's go with the steps needed in order to install ROS.

Setup your sources.list

First of all, you'll need to setup your computer in order to be able to download packages from **packages.ros.org**. For that, execute the below command in your local shell:

Execute in Local Shell #1

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

Setup your keys

Next, you will download the key from the keyserver using the following command:

Execute in Local Shell #1

```
[ ]: sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
     --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

If everything goes fine, you should see something like this:

Installation

Great! Now we are ready to actually install ROS. First of all, we'll make sure that our Debian packages index is up to date. For that, execute the following command:

Execute in Local Shell #1

```
[ ]: sudo apt-get update
```

Great! Now you are ready to start installing ROS packages into your system. In order to have all the basic packages for start working with ROS, we recommend you to install the **Desktop Full** installation. For doing so, you can execute the following command:

Execute in Local Shell #1

```
[ ]: sudo apt-get install ros-kinetic-desktop-full
```

NOTE: Since it will download and install several packages, this installation can take some minutes. So be patient.
At this point, you have installed some of the basic tools that ROS provides, like RViz, rqt, navigation libraries... With these tools, you will be ready to start working with ROS. Anyways, you will need to install some extra packages eventually. For installing an specific ROS package, you just need to use the following command structure:

```
[ ]: sudo apt-get install ros-kinetic-<PACKAGE_NAME>
```

For instance:

```
[ ]: sudo apt-get install ros-kinetic-slam-gmapping
```

The above command will install the slam gmapping package for the ROS Kinetic version.

Initialize rosdep

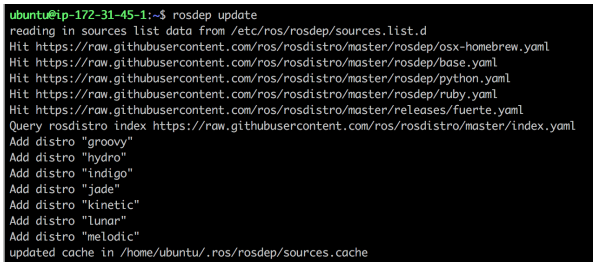
Before you can actually start using ROS, though, you will need to initialize **rosdep**. rosdep will allow you to easily install system dependencies, and it is also required to run some core components in ROS. To initialize rosdep, execute the following command:

Execute in Local Shell #1

```
[ ]: sudo rosdep init
```

```
[ ]: rosdep update
```

If everything goes fine, you should get something like this after executing the **rosdep update** command.



Environment setup

Finally, it is also recommended to automatically add the ROS Environment Variables (do you remember them from the 1st Chapter of the Course?) to your bash session every time a new shell is launched. For doing so, you can execute the following command:

Execute in Local Shell #1

```
[ ]: source /opt/ros/kinetic/setup.bash
```

You will need to run this command on every new shell you open to have access to the ROS commands, unless you add this line to your **.bashrc** file. So, unless you wish to run that command every time you open a new shell, you should add it to your **.bashrc**. For that, you can run the following command:

Execute in Local Shell #1

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

With the above command, you will add the line ***source /opt/ros/kinetic/setup.bash*** to your **.bashrc** file. This way, each time you open a new shell in your computer, all the ROS Environment variables will be automatically set up.

Also, this process allows you to install several ROS distributions (for instance, indigo and kinetic) on the same computer and switch between them. So, for instance, if you also had ROS Indigo installed in your local computer, you could switch between both distributions by using the bellow commands:

```
[ ]: source /opt/ros/indigo/setup.bash # To use Indigo
```

```
[ ]: source /opt/ros/kinetic/setup.bash # To use Kinetic
```

Dependencies for building packages

Great! So at this point, you have already installed and set up everything you need to run the core ROS packages. Anyways, there are various tools that you will also need in order to manage your ROS workspaces (remember your **catkin_ws**?). To install all this tools, you can run the following command:

Execute in Local Shell #1

```
[ ]: sudo apt-get install python-rosinstall python-rosinstall-generator
     python-wstool build-essential
```

Test your setup

And we are done! Now let's test that our setup actually works, and that we can run ROS on our local machine. For that, let's follow the next Example.

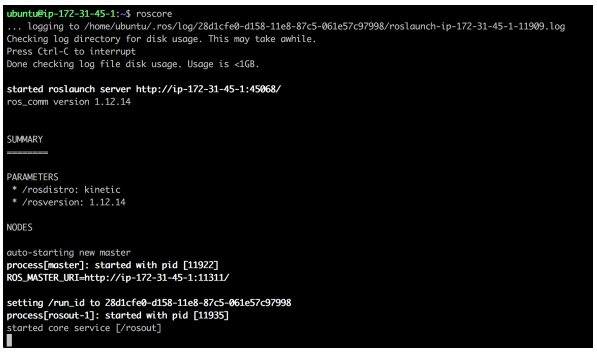
Example A.1

a) In the same shell where you have set up everything during this Notebook, run the following command:

Execute in Local Shell #1

```
[ ]: roscore
```

If everything goes fine, you should see something like this:



By the way... do you remember **roscore**? We introduced it to you back in the first chapter of this Course. **roscore** is the **main process** that manages all the ROS systems. So, if we want to do anything with ROS, we will always need to first start **roscore** in one shell.

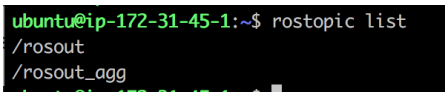
And you may ask yourself... and why I didn't use this command during the Course, if it so important? Well, it is because in the Ignite Academy, the roscore is automatically started for you whenever you enter a Course. But now, if you want to work in your local computer, you will need to manage it yourself!

b) With the roscore running in one shell, let's open a new shell. Within this new shell, type the following command:

Execute in Local Shell #2

```
[ ]: rostopic list
```

If everything goes fine, you should get the following topics:

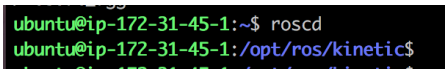


Also, you can execute the ***roscd*** command to make sure that your ROS system is properly set.

Execute in Local Shell #2

```
[ ]: roscd
```

If everything goes fine, you should go to the following path:



Excellent! So you have successfully installed ROS in your local computer.

End Example A.1

Excellent! So you have successfully installed ROS in your local computer. In the next chapter, **Appendix 2**, you will see how to create new workspaces and how to manage them, so that you can start developing your own ROS packages.

Appendix 2

ROS IN 5 DAYS

Appendix 2: Managing ROS workspaces

How to install ROS in my local computer?

During the whole Course, you have been working within a directory called **catkin_ws**. As you may already know, this directory is known as the **catkin workspace** or **ROS workspace**, and it is basically the place where you will have to place all the new ROS packages that you create. But... How do you create one of these workspaces? Is it possible to have more than one workspace?

During the Course, this workspace was already provided to you, so you just had to worry about putting all your packages inside there. But now, let's try to answer some of these interesting questions during the following Notebook.

Create a ROS workspace

For this example, you can use the same Robot Ignite Academy environment you've been working during the whole Course. You will just add a new workspace to it.

First of all, let's create the folder where we will create our ROS workspace.

Execute in Local Shell #1

```
[ ]: mkdir -p ~/mynew_ws/src
```

Now, you will need to compile it by running the **catkin_make** command.

Execute in Local Shell #1

```
[ ]: cd ~/mynew_ws/
```

```
[ ]: catkin_make
```

You already know about **catkin_make** command, don't you? You've used it along the course in order to compile your packages. The **catkin_make** command is a convenience tool for working with catkin workspaces. Running it the first time in your workspace, it will create the **CMakeLists.txt** file in your **src** folder. Also, if you look in your current directory you should now have a **build** and **devel** folder. Inside the **devel** folder you can see that there are now several _setup.*sh_ files. Sourcing any of these files will overlay this workspace on top of your environment.

Execute in Local Shell #1

```
[ ]: source devel/setup.bash
```

To make sure your workspace is properly overlayed by the setup script, you can check the **ROS_PACKAGE_PATH** environment variable with the following command.

Execute in Local Shell #1

```
[ ]: echo $ROS_PACKAGE_PATH
```

If everything goes fine, you should get the following:

```
user:~/mynew_ws$ echo $ROS_PACKAGE_PATH
/home/user/mynew_ws/src:/home/user/catkin_ws/src:/home/simulations/public_sim_ws/src:/opt/ros/kinetic/share
```

As you can see, your new workspace, named **mynew_ws**, is now on top of the **catkin_ws** you've been using during the whole course.

NOTE: Bear in mind that the IDE is configured to work only with 1 workspace, which is the **catkin_ws**, so you won't be able to visualize this new workspace within the IDE.
Exercise A.1

a) Now, repeat the whole process withint your local computer. Create a new workspace in your local ROS installation, which you completed in **Appendix 1**.

After you setup your workspace in your local ROS installation, you'll be able to start creating new ROS packages there, or even download and place there the ones you created during the Course!

End Exercise A.1
Congratulations! You are now able to download and use all the packages you created during the Course into your local machine. Enjoy it! And... keep pushing your ROS learning!

Final Recommendations

I'm finished, now what?

ROS Development Studio (ROSDS)

ROSDS logo

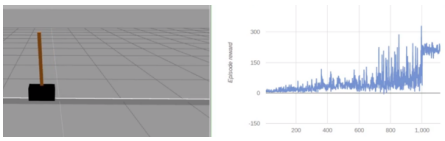
ROSDS is the The Construct web based tool to program ROS robots online. It requires no installation in your computer. Hence, you can use any type of computer to work on it (Windows, Linux or Mac). Additionally, free accounts are available. **Create a free ROSDS account here**: http://rosds.online
You can use any of the many ROSjects available in order to apply all the things you've learned during the Course. You just need to paste the ROSject link to your browser's URL, and you will automatically have the simulation prepared in your ROSDS workspace.

Down below you can check some examples of the Public Rosjects we provide:

ARIAC Competition

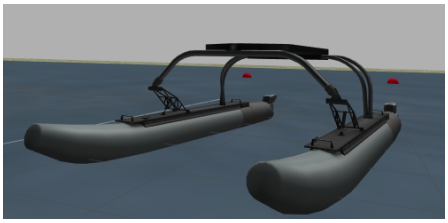
- ROSject Link: https://bit.ly/2t2px0t

Cartpole Reinforcement Learning



- ROSject Link: https://bit.ly/2t2uGWr

ARIAC Competition



- ROSject Link: https://bit.ly/2Tt4lw8

Want to learn more?

Once you have finished the course, you can still learn a lot of interesting ROS subjects.

- Take more advanced courses that we offer at the Robot Ignite Academy, like Perception or Navigation. Access the Academy here: http://www.robotigniteacademy.com
- Or, you can go to the ROS Wiki and check the official documentation of ROS, now with new eyes.

Thank You and hope to see you soon!