**INFO 802**

**Master Advanced Mechatronics**

Luc Marechal

UNIVERSITÉ SAVOIE MONT BLANC
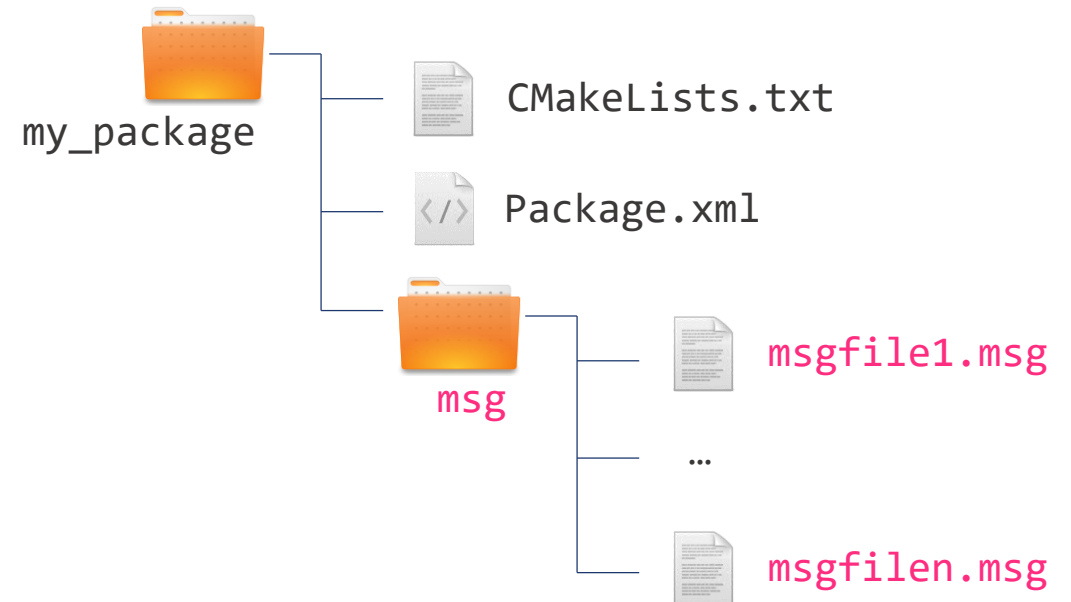
POLYTECH® ANNECY-CHAMBÉRY

2019

**⠿ROS**

**Introduction Course 3**

# ROS Messages

- They are files where we put a specification about the type of data to be transmitted and the values of this data.

- Defined in *.msg files stored in the msg subdirectory of a package



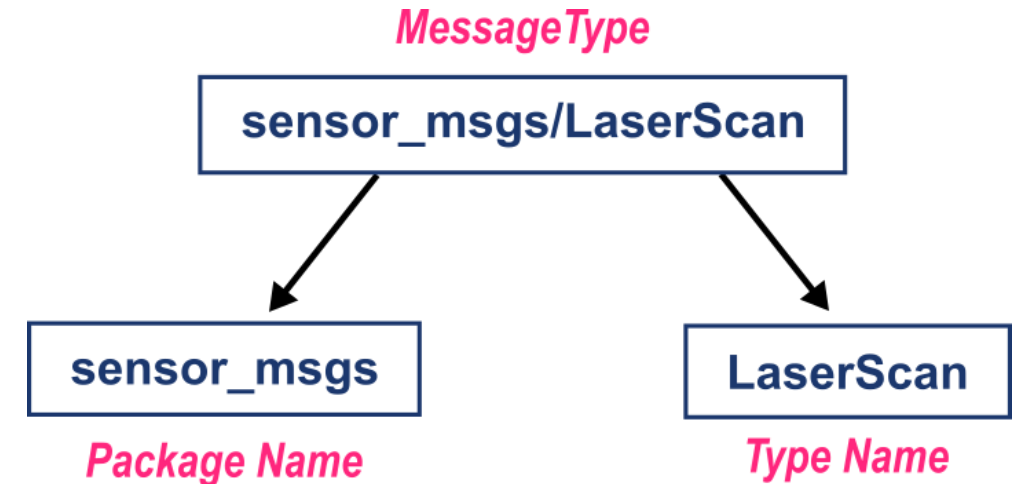See message definition information with

```
> rosmsg show [message_type]
```

# ROS Messages

*Example:*

- Every message type belongs to a specific package

  Message type names always contain a slash, and the part before the slash is the name of the containing package:

  > package_name/type_name

**MessageType**

sensor_msgs/LaserScan

sensor_msgs

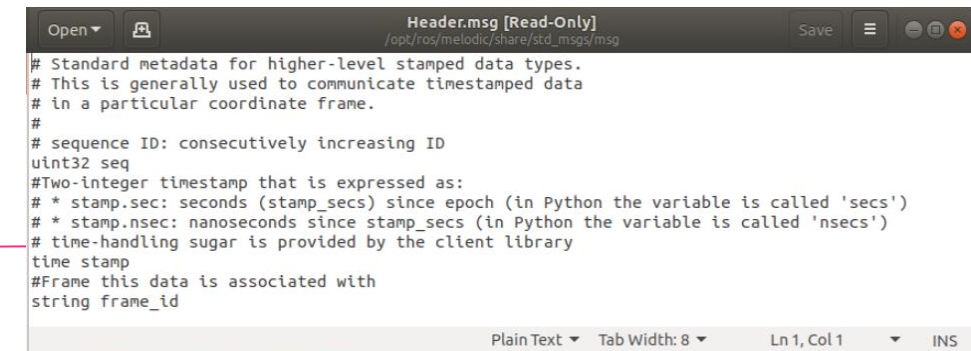**Package Name**

LaserScan

**Type Name**

# ROS Messages

▪ msgs are just simple text files with a field type and field name per line. The field types you can use are:

- •int8, int16, int32, int64 (plus uint*)
- •float32, float64
- •string
- •time, duration
- •other msg files
- •variable-length array[] and fixed-length array[C]

▪ Header: special type in ROS

The header contains a timestamp and coordinate frame information that are commonly used in ROS to communicate timestamped data in a particular coordinate frame.

uint32 seq

time stamp

string frame_id

```
Header.msg [Read-Only]
/opt/ros/melodic/share/std_msgs/msg
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.
#
# sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable is called 'secs')
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is called 'nsecs')
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

# ROS Messages

▪ Standard type to use in message
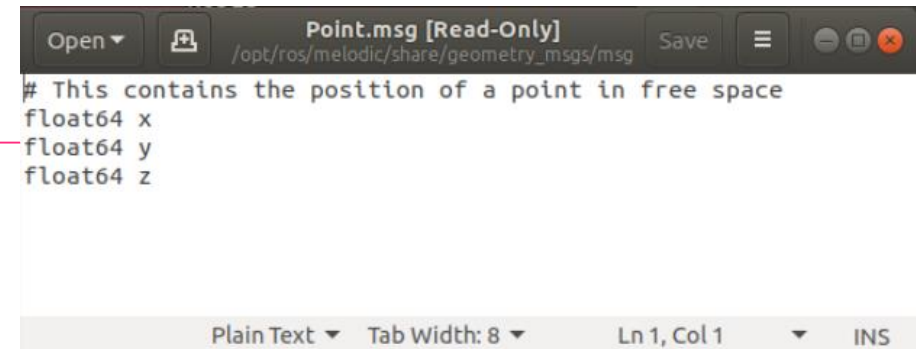
| Primitive type | Serialization | C++ | Python |
|---|---|---|---|
| bool | Unsigned 8-bit int | uint8_t | bool |
| int8 | Signed 8-bit int | int8_t | int |
| uint8 | Unsigned 8-bit int | uint8_t | int |
| int16 | Signed 16-bit int | int16_t | int |
| uint16 | Unsigned 16-bit int | uint16_t | int |
| int32 | Signed 32-bit int | int32_t | int |
| uint32 | Unsigned 32-bit int | uint32_t | int |
| int64 | Signed 64-bit int | int64_t | long |
| uint64 | Unsigned 64-bit int | uint64_t | long |
| float32 | 32-bit IEEE float | float | float |
| float64 | 64-bit IEEE float | double | float |
| string | ASCII string (4-bit) | std::string | string |
| time | Secs/nsecs signed 32-bit ints | ros::Time | rospy.Time |
| duration | Secs/nsecs signed 32-bit ints | ros::Duration | rospy.Duration |

# ROS Messages

- Examples

geometry_msgs/Points.msg

```
float64 x
float64 y
float64 z
```

geometry_msgs/Quaternions.msg

```
float64 x
float64 y
float64 z
float64 w
```

# ROS Messages

- Example

geometry_msgs/Points.msg

```
float64 x
float64 y
float64 z
```

geometry_msgs/Pose.msg

```
geometry_msgs/Point position

geometry_msgs/Quaternion orientation
```

geometry_msgs/Quaternions.msg

```
float64 x
float64 y
float64 z
float64 w
```
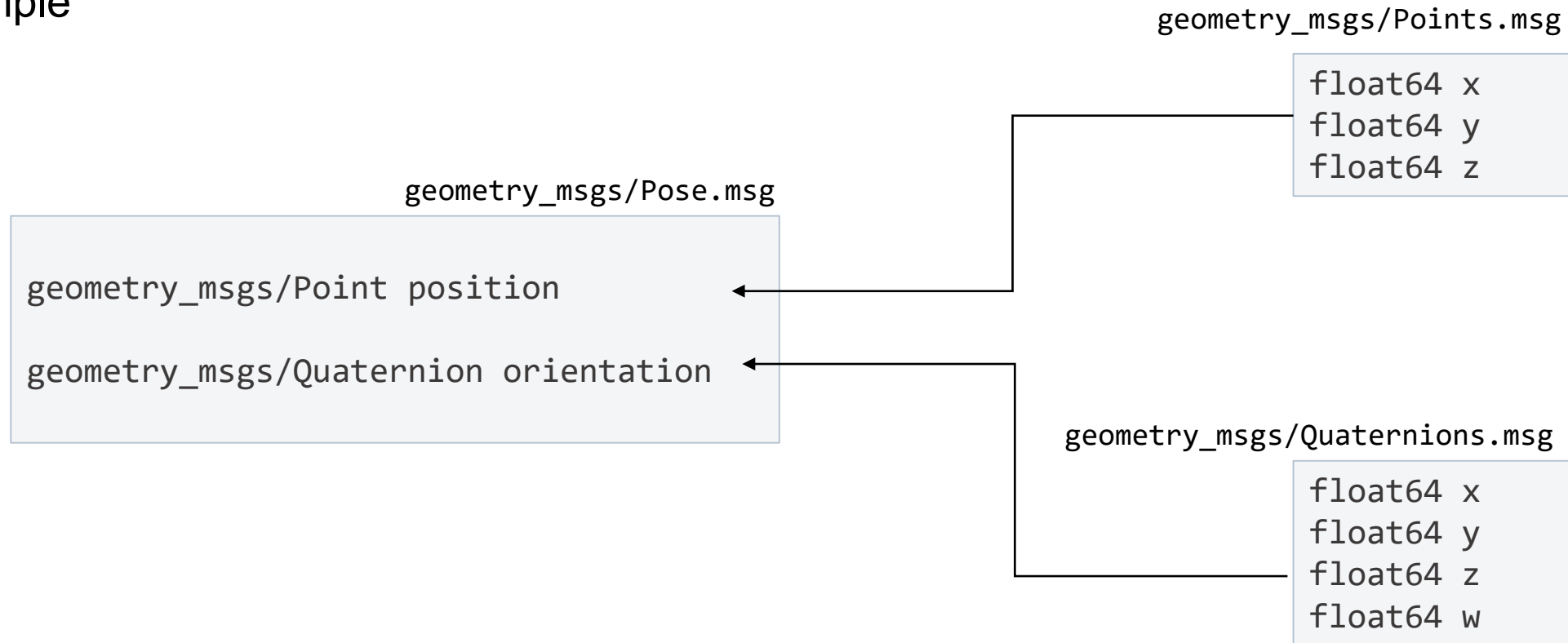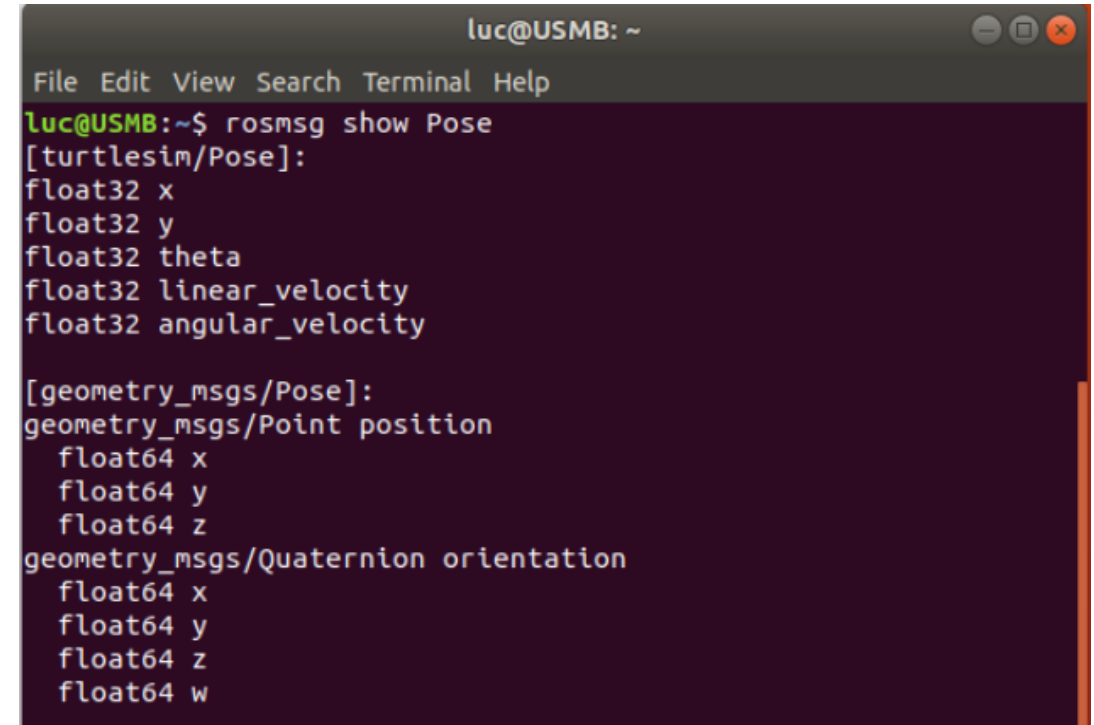
You can use message type from already existing message

# ROS Command Tools

See message definition information:

```
> rosmsg show [message_type]
```

```
luc@USMB: ~
File  Edit  View  Search  Terminal  Help
luc@USMB:~$ rosmsg show Pose
[turtlesim/Pose]:
float32 x
float32 y
float32 theta
float32 linear_velocity
float32 angular_velocity

[geometry_msgs/Pose]:
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

The message of type *Pose* is defined in the package *turtlesim* but also in the package *geometry_msgs* but they are not the sames !

# ROS Command Tools

See message definition information:

```
> rosmsg show [message_type]
```

See active topics:

```
> rostopic list
```

See node information:

```
> rosnode info [message_type]
```

# How to use ROS Messages in code?

Import the message type from the msg library

Use the message directly with an oriented object way

Use the message with an Object

```python
#!/usr/bin/env python

from geometry_msg.msg import Pose
from beginner_tutorials.msg import My_Custom_Message

# without creating an object
Pose.position.x = 1.0

# by creating an object
My_Object = Pose()
My_Object.position.x = 1.0

My_Object.orientation.y = My_Object.position.x  + 43.2
```

```
[geometry_msgs/Pose]:
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

# Creating a custom ROS msg

Edit *.msg file

- When Should You Make a New Message Type?

Only when you absolutely have to (check before with *rosmsg* to see if there is already something there that you can use instead).
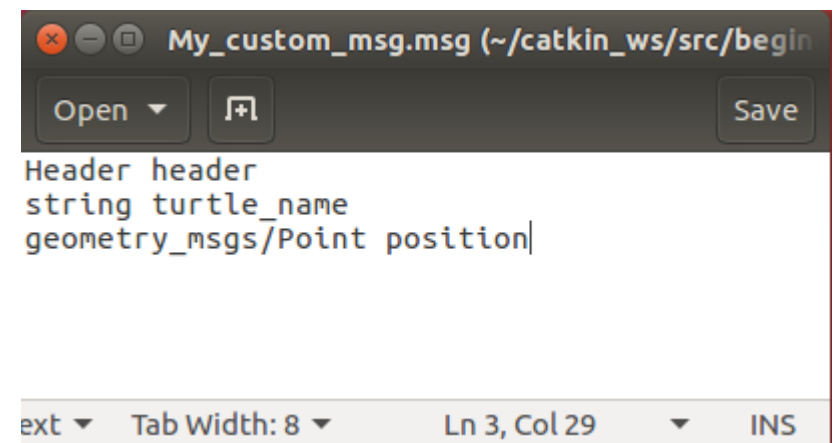
However, there are times when the built-in message types are not enough, and we have to define our own messages

Create a subfolder named **msg** in your package folder

```
> cd ~/catkin_ws/src/beginner_tutorials
> mkdir msg
```

Create a new my_custom_msg.msg file and add the following lines

```
> gedit msg/my_custom_msg.msg
```



```
My_custom_msg.msg (~/catkin_ws/src/begin

Open

Header header
string turtle_name
geometry_msgs/Point position

Tab Width: 8    Ln 3, Col 29    INS
```

# Creating a custom ROS msg

## Modifify package.xml file

- We need to make sure that the msg files are turned into source code for C++, Python, and other languages

  uncomment those two lines in the package.xml file

  ```
  <build_depend>message_generation</build_depend>
  <run_depend>message_runtime</run_depend>
  ```

- Note that at build time, we need "message_generation", while at runtime, we need "message_runtime"

# Creating a custom ROS msg

## Modifify CMakefile.txt file

- In CMakeLists.txt add the message_generation dependency to the find package call so that you can generate messages:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
}
```

- Also make sure you export the message runtime dependency:

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES multi_sync
    CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
#  DEPENDS system_lib
)
```

# Creating a custom ROS msg

Modifify CMakefile.txt file

- Find the following block

```
## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

- Uncomment it by removing the # symbols and then replace the stand in Message*.msg files with your .msg file, such that it looks like this:

```
add_message_files(
    FILES
    my_custom_msg.msg
)
```

# Creating a custom ROS msg

Modifify CMakefile.txt file

- ensure the generate_messages() function is called: uncomment this lines

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

- So it looks like:

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

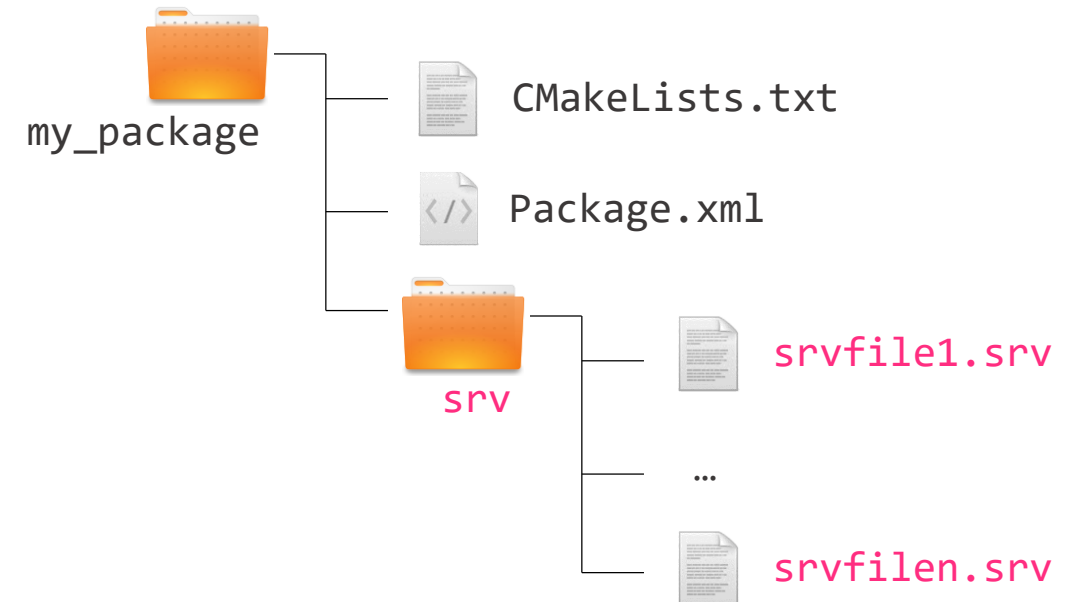cmake will then know that the project needs to be reconfigurated with the addition of msg files

# Creating a ROS srv

- Defined in *.srv* files stored in the srv subdirectory of a package

- srv files are just like msg files, except they contain two parts: a request and a response. The two parts are separated by a '---' line.



```
int64 a
int64 b
---
int64 sum
```
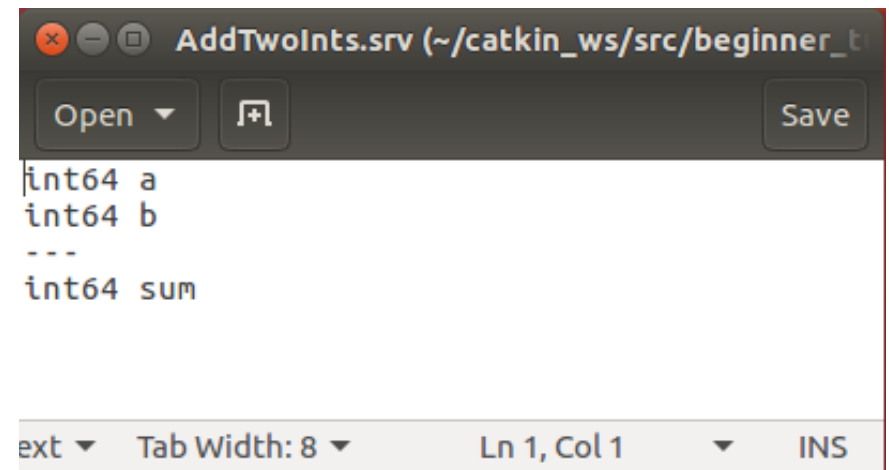
# Creating a ROS srv

Modifify package.xml file

Create a subfolder named **srv** in your package folder

```
> cd ~/catkin_ws/src/beginner_tutorials
> mkdir srv
```

Example: create `AddTwoInts.srv` file and add the following lines

```
> gedit srv/AddTwoInts.srv
```

# Creating a ROS srv

## Modifify package.xml file

- We need to make sure that the srv files are turned into source code for C++, Python, and other languages

  uncomment those two lines in the package.xml file

  ```
  <build_depend>message_generation</build_depend>
  <run_depend>message_runtime</run_depend>
  ```

- Note that at build time, we need "message_generation", while at runtime, we need "message_runtime"

# Creating a ROS srv

## Modifify CMakefile.txt file

- In CMakeLists.txt add the message_generation dependency to the find package call so that you can generate messages:

  (Despite its name, message_generation works for both msg and srv.)

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
}
```

# Creating a ROS srv

Modifify CMakefile.txt file

- Find the following block

```
## Generate messages in the 'msg' folder
# add_service_files(
#   FILES
#   Service1.msg
#   Service2.msg
# )
```

- Uncomment it by removing the # symbols and then replace the stand in Service*.msg files with your .srv file, such that it looks like this:
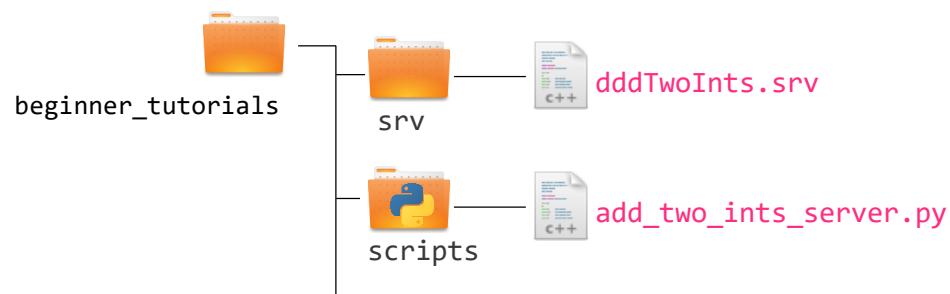
```
add_service_files(
    FILES
    AddTwoInts.srv
)
```

# Creating a Service and Client Node (Python)

## Writing the Service Node

Edit a py file in scripts folder

```
> cd ~/catkin_ws/beginner_tutorials/src/scripts
> sudo gedit Add_two_ints_server.py
```



beginner_tutorials
srv — dddTwoInts.srv
scripts — add_two_ints_server.py

… and make it executable

```
> chmod +x scripts/add_two_ints_server.py
```

```python
#!/usr/bin/env python

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts,
handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

# Creating a Service and Client Node (Python)

Examining the Service Node

The *service* file has been defined and is located in the *srv* folder

*init_node()*: declare the node

This declares a new service named *add_two_ints* with the *AddTwoInts service* type. All requests are passed to *handle_add_two_ints* function. *handle_add_two_ints* is called with instances of AddTwoIntsRequest and returns instances of AddTwoIntsResponse.

```python
#!/usr/bin/env python

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```
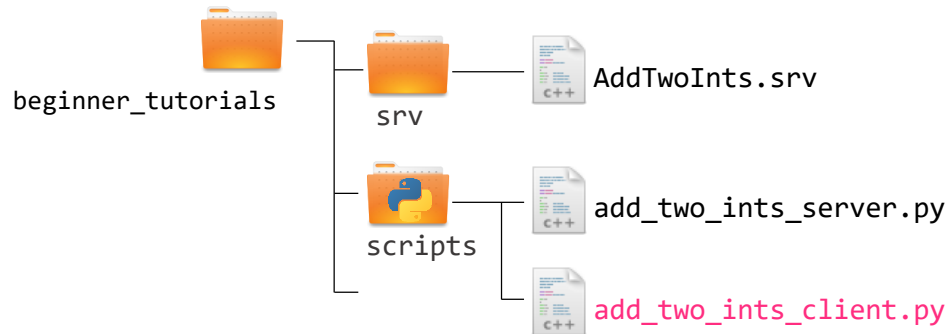
# Creating a Service and Client Node (Python)

Writing the Client Node

Edit a py file in scripts folder

```
> cd ~/catkin_ws/beginner_tutorials/src/scripts
> sudo gedit Add_two_ints_client.py
```

beginner_tutorials
srv — AddTwoInts.srv
scripts — add_two_ints_server.py
add_two_ints_client.py

… and make it executable

```
> chmod +x scripts/Add_two_ints_client.py
```

```python
#!/usr/bin/env python

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

# Creating a Service and Client Node (Python)

Examining the Client Node

Wait for the service named *add_two_ints* to be advertised by the server

Once the service is advertised, we can set up a local proxy for it

```python
#!/usr/bin/env python

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

# Creating a Publisher and a Subscriber Node (Python)

## Building the nodes

If not done yet: make the node executable (for Python only)

```
> chmod +x scripts/add_two_ints_server.py
> chmod +x scripts/add_two_ints_client.py
```

Build package
(we use Cmake as the build system even for Python nodes)

```
> cd ~/catkin_ws
> catkin_make beginner_tutorials
```

Make sure you have sourced your workspace's setup.bash file

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

# ROS Bags

- **rosbag:** set of tools for recording messages and playing back later to ROS topics offline.

- Can be used to mimic real sensor streams for offline debugging.

- Useful for debugging algorithm.

- The file a name is in the format: *file_name_YYYY-MM-DD-HH-mm-ss.bag*

Record topics with

```
> rosbag record [topic_1] [topic_2] -o [bag_name]
```

Playback messages with

```
> rosbag play [bag_name]
```

Examples

```
> rosbag record –a
```

```
> rosbag play --clock mybag.bag
```

```
luc@luc: ~/catkin_ws/bagfiles
luc@luc:~/catkin_ws/bagfiles$ rosbag info 2019-03-07-18-03-41.bag
path:        2019-03-07-18-03-41.bag
version:     2.0
duration:    32.5s
start:       Mar 07 2019 18:03:41.15 (1551978221.15)
end:         Mar 07 2019 18:04:13.63 (1551978253.63)
size:        294.7 KB
messages:    4111
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
             rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
             turtlesim/Color     [353891354491c51aabe32df673fb446]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /rosout               4 msgs    : rosgraph_msgs/Log
             /turtle1/cmd_vel     70 msgs    : geometry_msgs/Twist
             /turtle1/color_sensor 2022 msgs : turtlesim/Color
             /turtle1/pose        2015 msgs  : turtlesim/Pose
```

# Example logging *turtlesim*

# ROS computation graph  *rqt*

- *rqt_graph* creates a dynamic graph of what's going on in the system

- *rqt_console* attaches to ROS's logging framework to display output from nodes. rqt_logger_level allows us to change the verbosity level (DEBUG, WARN, INFO, and ERROR) of nodes as they run.


- Prerequisit: Install rqt package

```
> sudo apt-get install ros-melodic-rqt ros-melodic-rqt-common-plugins
```

Launch rqt_console

```
> rosrun rqt_console rqt_console
```

Launch rosrun rqt_logger_level rqt_logger_level (in an other terminal)

```
> rosrun rqt_logger_level rqt_logger_level
```

# ROS computation graph *rqt*

Visualize running topics and nodes

```
> rosrun rqt_graph rqt_graph
```

Visualize running topics and nodes

```
> rosrun rqt_plot rqt_plot
```

# Further References

- ## ROS Wiki
  - http://wiki.ros.org/

- ## Installation
  - http://wiki.ros.org/ROS/Installation

- ## Tutorials
  - http://wiki.ros.org/ROS/Tutorials

- ## Available packages
  - http://www.ros.org/browse/

- ## ROS Cheat Sheet
  - https://www.clearpathrobotics.com/ros-robot-operating-system-cheat-sheet/
  - https://kapeli.com/cheat_sheets/ROS.docset/

- ## ROS Best Practices
  - https://github.com/leggedrobotics/ros_best_practices/wiki

- ## ROS Package Template
  - https://github.com/leggedrobotics/ros_best_practices/tree/master/ros_package_template

# Relevant books
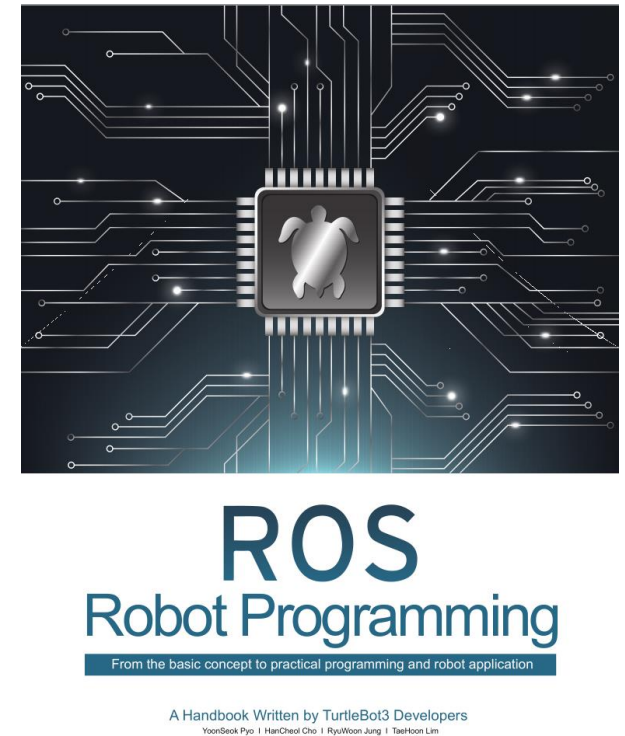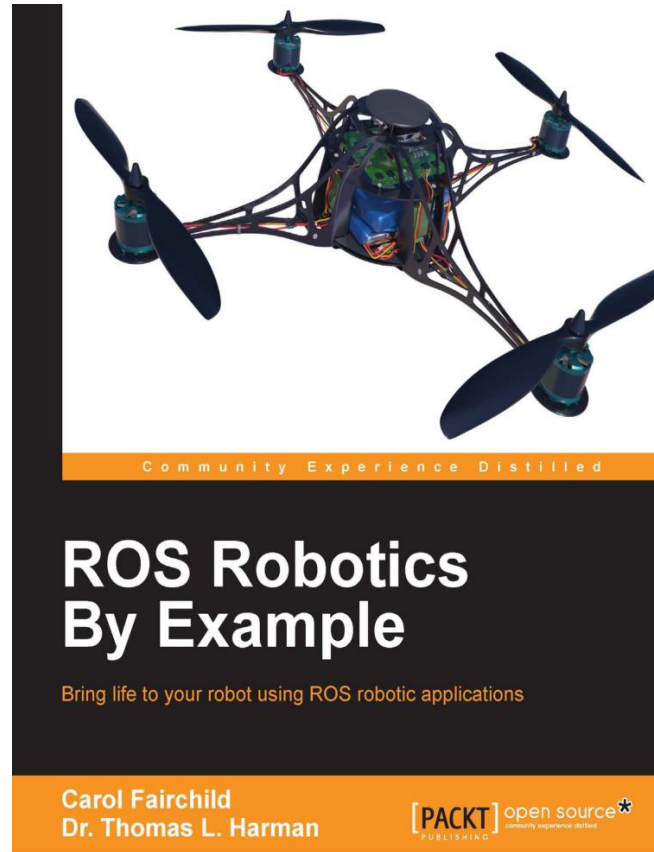
# Contact Information

## Université Savoie Mont Blanc

Polytech' Annecy Chambery
Chemin de Bellevue
74940 Annecy
France

https://www.polytech.univ-savoie.fr

## Lecturer

Luc Marechal (luc.marechal@univ-smb.fr)
SYMME Lab  (Systems and Materials for Mechatronics