



2021

**INFO 802**

**Master Advanced Mechatronics**

Luc Marechal



**ROS**

**ROS Command Tools, ROS  
message - Lecture 3**

# Objectives

- Know what a ROS message is made up of.
- Find which library a ROS message comes from.
- Create a custom ROS message file.
- Use messages with an Object in Python code.

# ROS Command Tools

# Turtlesim

## Turtle\_teleop\_key node



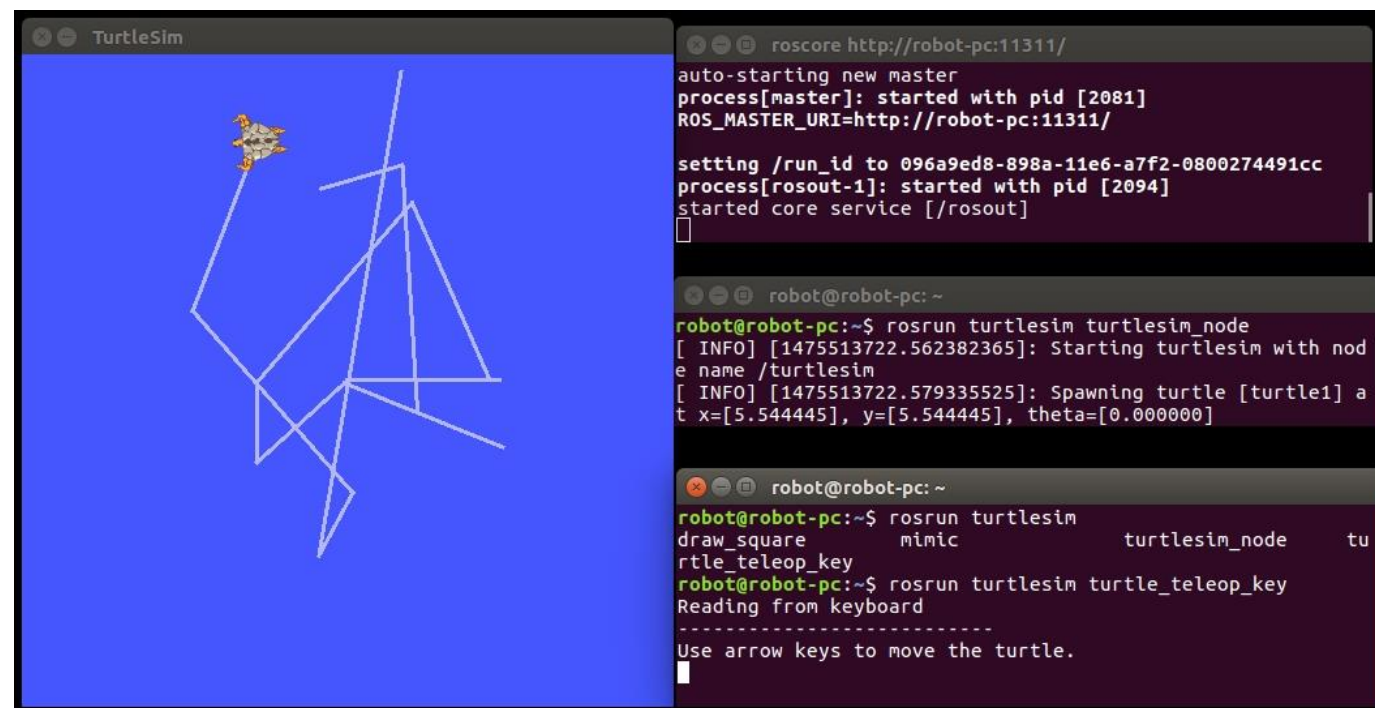
Test moving the turtle  
(with the *turtle\_teleop\_key* node)

Recall: Open a terminal for each command

```
> roscore
```

```
> rosrunc turtlesim turtlesim_node
```

```
> rosrunc turtlesim turtle_teleop_key
```



The terminal which *turtle\_teleop\_key* is running on MUST be selected.  
Change the turtle's position by pressing arrow keys on the keyboard.

# ROS Command Tools

List all active topics on ROS:

```
> rostopic list
```

See which message is used on a topic:

```
> rostopic type [topic_name]
```

Get more information on a topic:

```
> rostopic info [topic_name]
```

```
luc@USMB: ~  
luc@USMB:~$ rostopic list  
/rosout  
/rosout_agg  
/turtle1/cmd_vel  
/turtle1/color_sensor  
/turtle1/pose  
luc@USMB:~$
```

```
luc@USMB:~$ rostopic type /turtle1/pose  
turtlesim/Pose
```

```
luc@USMB:~$ rostopic type /turtle1/pose  
Type: turtlesim/Pose  
  
Publishers:  
* /turtlesim (http://localhost:40351/)  
  
Subscribers: None
```

# ROS Command Tools

Show all messages available in ROS:

```
> rosmmsg list
```

```
ros@masterpc:~$ rosmmsg list
actionlib/TestAction
actionlib/TestActionFeedback
actionlib/TestActionGoal
actionlib/TestActionResult
actionlib/TestFeedback
actionlib/TestGoal
actionlib/TestRequestAction
actionlib/TestRequestActionFeedback
actionlib/TestRequestActionGoal
actionlib/TestRequestActionResult
actionlib/TestRequestFeedback
actionlib/TestRequestGoal
actionlib/TestRequestResult
actionlib/TestResult
actionlib/TwoIntsAction
actionlib/TwoIntsActionFeedback
actionlib/TwoIntsActionGoal
actionlib/TwoIntsActionResult
actionlib/TwoIntsFeedback
actionlib/TwoIntsGoal
actionlib/TwoIntsResult
actionlib_msgs/GoalID
actionlib_msgs/GoalStatus
actionlib_msgs/GoalStatusArray
actionlib_tutorials/AveragingAction
```

Show the content of a message type:

```
> rosmmsg show [message_type]
```

```
> rosmmsg show turtlesim/Pose
```

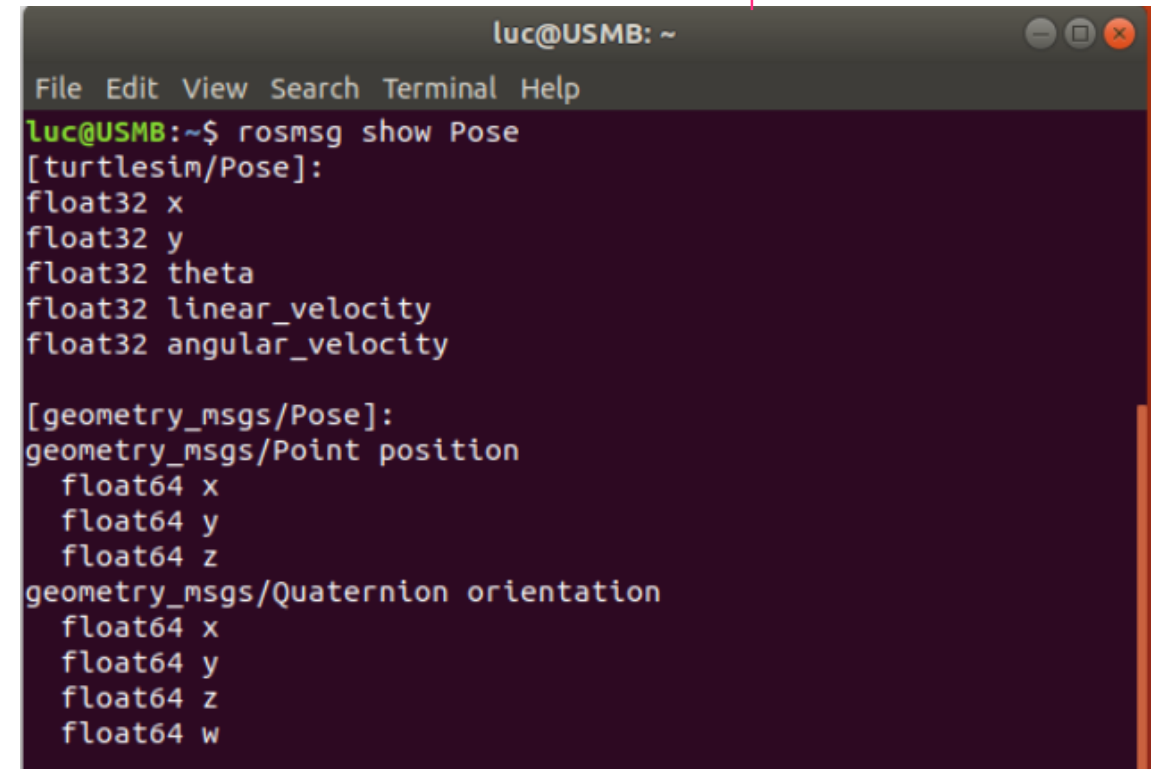
```
luc@USMB:~$ rosmmsg show turtlesim/Pose
[turtlesim/Pose]:
float64 x
float64 y
float64 theta
float64 linear_velocity
float64 angular_velocity
```

# ROS Command Tools

See message definition information:

```
> rosmmsg show [message_type]
```

```
> rosmmsg show Pose
```



```
luc@USMB: ~  
File Edit View Search Terminal Help  
luc@USMB:~$ rosmmsg 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 same !

# ROS Command Tools

## System File

Get information on packages

```
> rospack find [package_name]
```

Change directory (cd) directly to a package or a stack

```
> roscd [location_name[/subdir]]
```

/s directly in a package by name rather than by absolute path

```
> rosls [location_name[/subdir]]
```

## ROS CHEAT SHEET MELODIC

### WORKSPACES

#### Create Workspace

```
mkdir catkin_ws && cd catkin_ws
wstool init src
catkin_make
source devel/setup.bash
```

#### Add Repo to Workspace

```
roscd; cd ../src
wstool set repo_name \
--git http://github.com/org/repo_name.git \
--version-melodic-devel
wstool up
```

#### Resolve Dependencies in Workspace

```
sudo rosdep init # only once
rosdep update
rosdep install --from-paths src --ignore-src \
--rosdistro=$(ROS_DISTRO) -y
```

### PACKAGES

#### Create a Package

```
catkin_create_pkg package_name [dependencies ...]
```

#### Package Folders

include/package_name	C++ header files
src	Source files. Python libraries in subdirectories
scripts	Python nodes and scripts
msg, srv, action	Message, Service, and Action definitions

#### Release Repo Packages

```
catkin_generate_changelog
# review & commit changelogs
catkin_prepare_release
bloom-release --track melodic --ros-distro melodic repo_name
```

#### Reminders

- Testable logic
- Publish diagnostics
- Desktop dependencies in a separate package

### CMakeLists.txt

#### Skeleton

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED)
catkin_package()
```

#### Package Dependencies

To use headers or libraries in a package, or to use a package's exported CMake macros, express a build-time dependency:

```
find_package(catkin REQUIRED COMPONENTS roscpp)
```

Tell dependent packages what headers or libraries to pull in when your package is declared as a catkin component:

```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ${PROJECT_NAME}
  CATKIN_DEPENDS roscpp)
```

Note that any packages listed as CATKIN\_DEPENDS dependencies must also be declared as a <run\_depend> in package.xml.

#### Messages, Services

These go after find\_package(), but before catkin\_package().

Example:

```
find_package(catkin REQUIRED COMPONENTS message_generation
std_msgs)
add_message_files(FILES MyMessage.msg)
add_service_files(FILES MyService.msg)
generate_messages(DEPENDENCIES std_msgs)
catkin_package(CATKIN_DEPENDS message_runtime std_msgs)w
```

#### Build Libraries, Executables

Goes after the catkin\_package() call.

```
add_library(${PROJECT_NAME} src/main)
add_executable(${PROJECT_NAME}_node src/main)
target_link_libraries(
  ${PROJECT_NAME}_node ${catkin_LIBRARIES})
```

#### Installation

```
install(TARGETS ${PROJECT_NAME}
DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION})
install(TARGETS ${PROJECT_NAME}_node
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
install(PROGRAMS scripts/myScript
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
install(DIRECTORY launch
DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION})
```

### RUNNING SYSTEM

Run ROS using plain:  
roscore

Alternatively, roslaunch will run its own roscore automatically if it can't find one:

```
roslaunch my_package package_launchfile.launch
```

Suppress this behaviour with the --wait flag.

#### Nodes, Topics, Messages

```
roscd
rostopic list
rostopic echo cmd_vel
rostopic hz cmd_vel
rostopic info cmd_vel
rostopic show geometry_msgs/Twist
```

#### Remote Connection

Master's ROS environment:

- ROS\_IP or ROS\_HOSTNAME set to this machine's network address.
- ROS\_MASTER\_URI set to URI containing that IP or hostname.

Your environment:

- ROS\_IP or ROS\_HOSTNAME set to your machine's network address.
- ROS\_MASTER\_URI set to the URI from the master.

To debug, check ping from each side to the other, run roswtf on each side.

#### ROS Console

Adjust using rqt\_logger\_level and monitor via rqt\_console. To enable debug output across sessions, edit the \$HOME/.ros/config/rosconsole.config and add a line for your package:

```
log4j.logger.${ros.package_name}=DEBUG
```

And then add the following to your session:

```
export ROSCONSOLE_CONFIG_FILE=$HOME/.ros/config/rosconsole.config
```

Use the roslaunch --screen flag to force all node output to the screen, as if each declared <node> had the output="screen" attribute.



www.clearpathrobotics.com/ros-cheat-sheet  
© 2019 Clearpath Robotics, Inc. All Rights Reserved.

More info

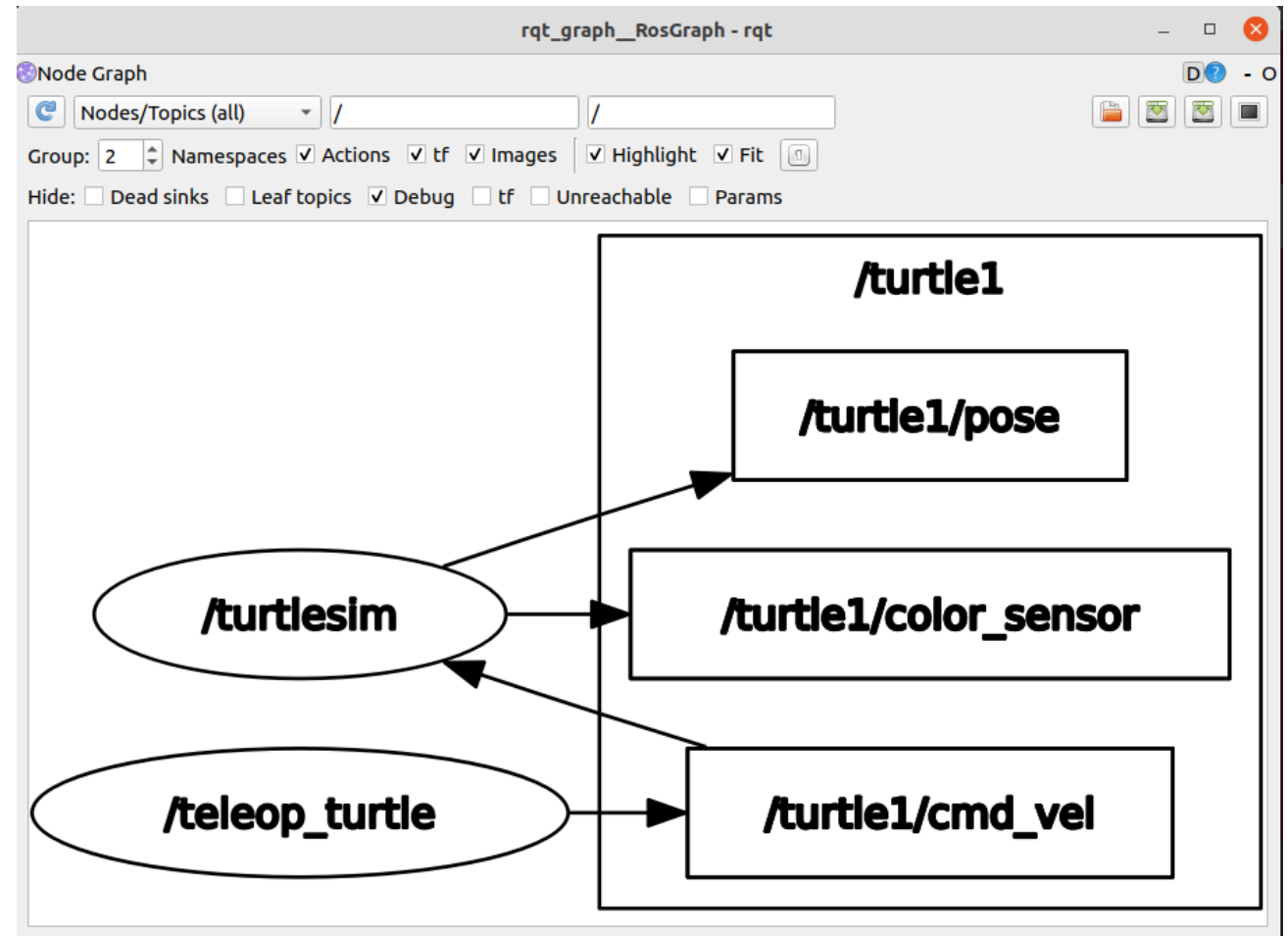
<http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem>



# ROS computation graph *rqt*

Visualize running topics and nodes

```
> rosrun rqt_graph rqt_graph
```

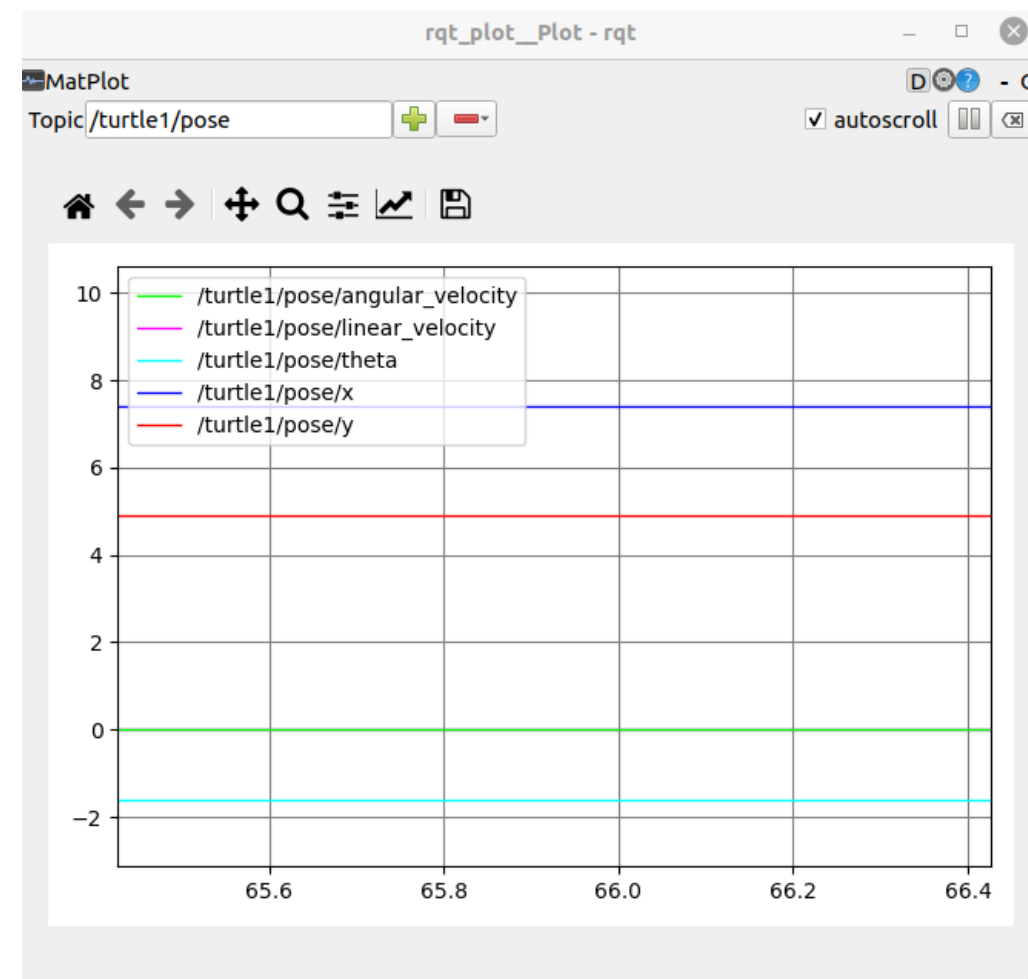
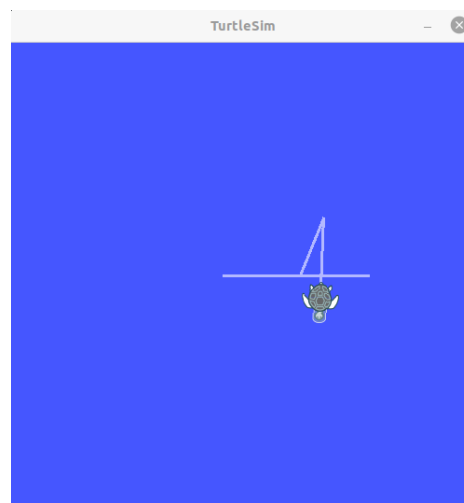


# ROS computation graph *rqt*

Visualize running topics and nodes

```
> rosrun rqt_plot rqt_plot
```

It shows the values published on a topic



## 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.
- Prerequisite: Install rqt package

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

Launch *rqt\_console*

```
> rosrun rqt_console rqt_console
```

Launch *roslaunch rqt\_logger\_level rqt\_logger\_level* (in an other terminal)

```
> roslaunch rqt_logger_level rqt_logger_level
```

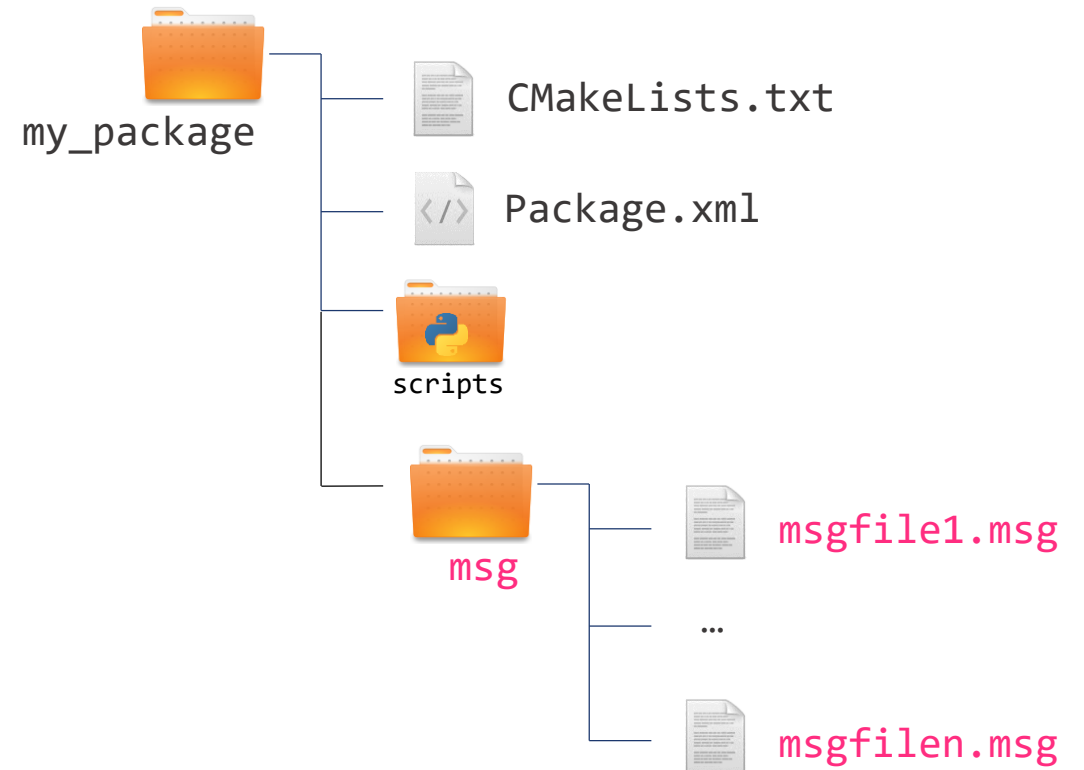
# ROS Message

# 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

```
> rosmmsg show [message_type]
```



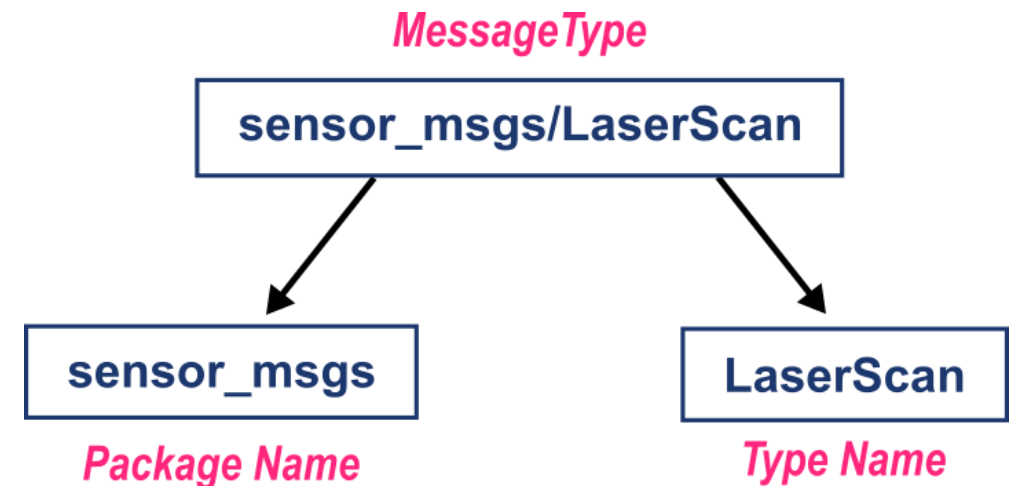
# ROS Messages

- 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
```

*Example:*



# ROS Messages

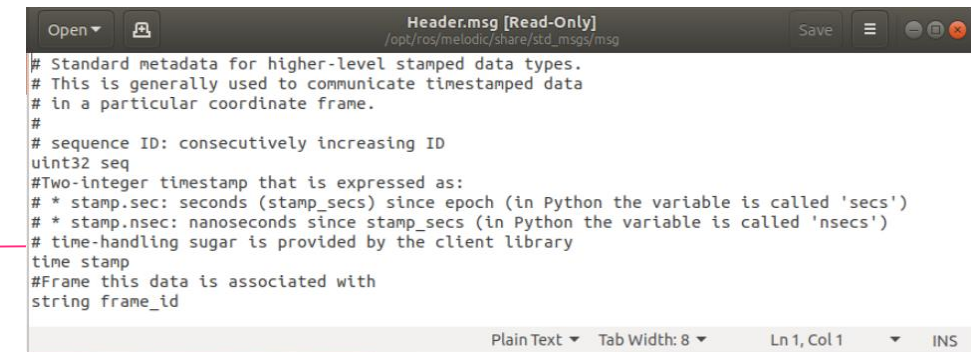
- msgs are just simple text files with a field type and field name per line. The field types you can use are:

- 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
```



The screenshot shows a text editor window titled "Header.msg [Read-Only]" with the file path "/opt/ros/melodic/share/std\_msgs/msg". The content of the file is as follows:

```
# 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
```

The editor interface includes a menu bar with "Open", "Save", and other icons, and a status bar at the bottom showing "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

# 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

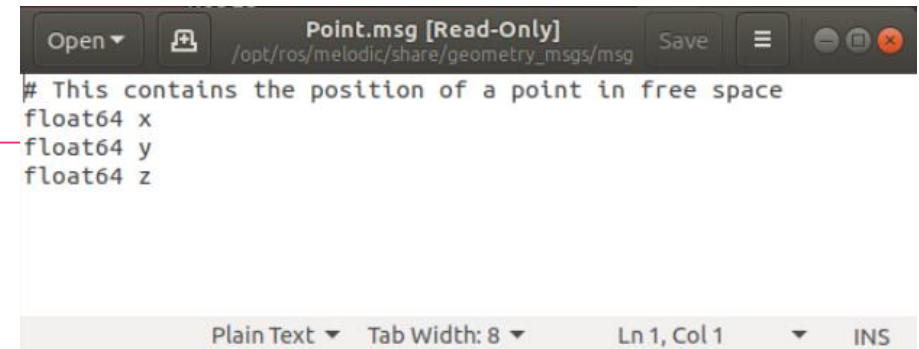


# ROS Messages

- Examples

geometry\_msgs/Points.msg

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



The screenshot shows a text editor window titled "Point.msg [Read-Only]" with the file path "/opt/ros/melodic/share/geometry\_msgs/msg". The editor contains the following text: "# This contains the position of a point in free space", "float64 x", "float64 y", and "float64 z". The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

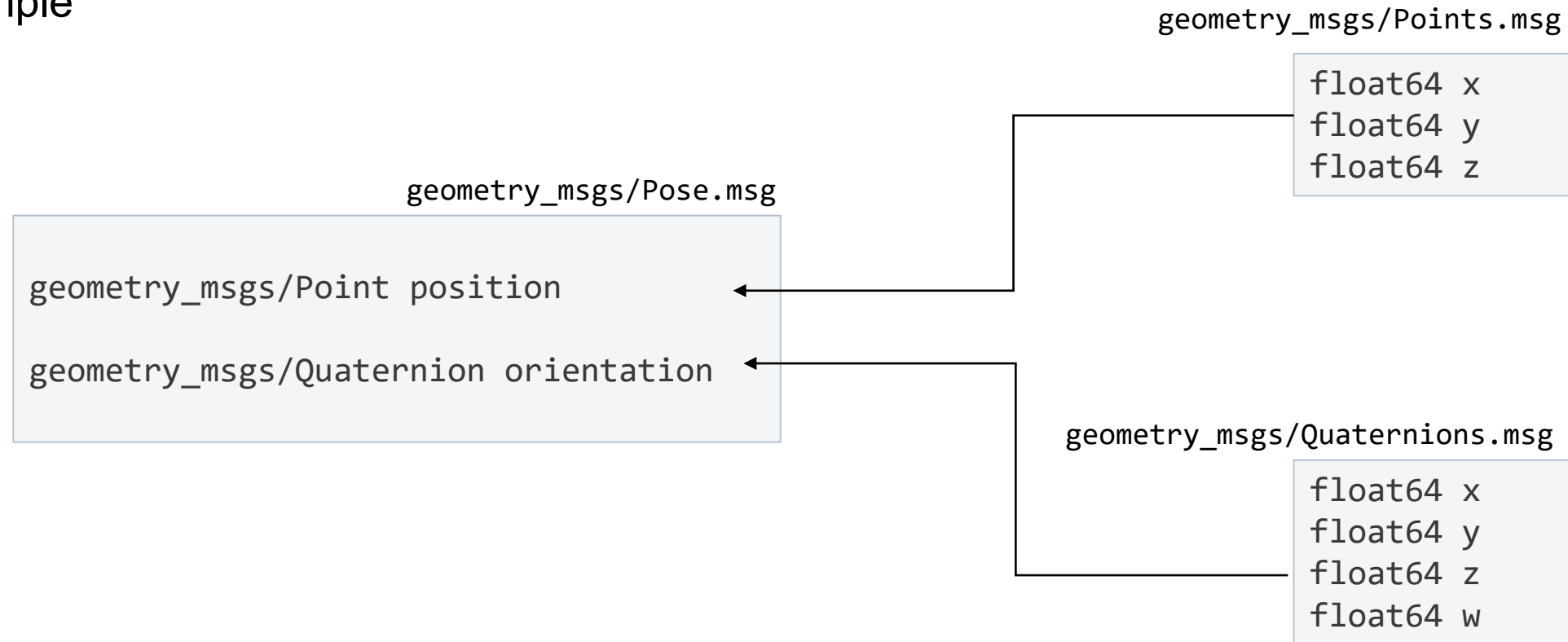
```
# This contains the position of a point in free space  
float64 x  
float64 y  
float64 z
```

geometry\_msgs/Quaternions.msg

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

# ROS Messages

- Example



You can use message type from already existing message

# 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

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

from geometry_msgs.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 *rosmmsg* to see if there is already something there that you can use instead).

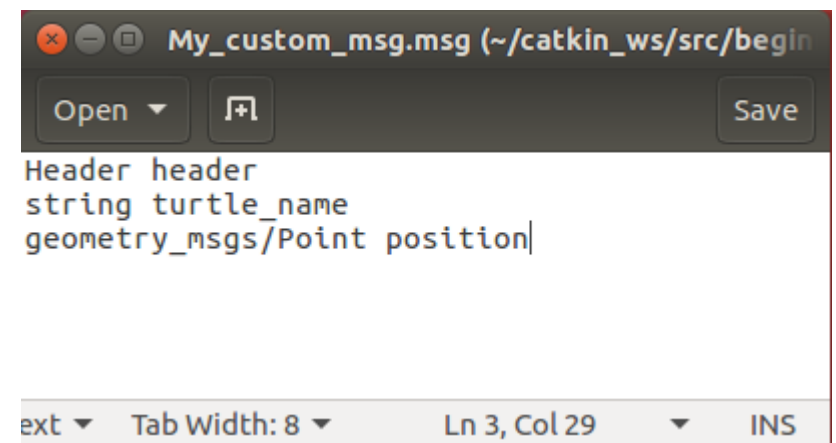
However, there are times when the built-in message types are not enough, and we have to define our own messages

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

```
> subl msg/my_custom_msg.msg
```



# Creating a custom ROS msg

## Modify 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

## Modify 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

## Modify 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

## Modify 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  
)
```

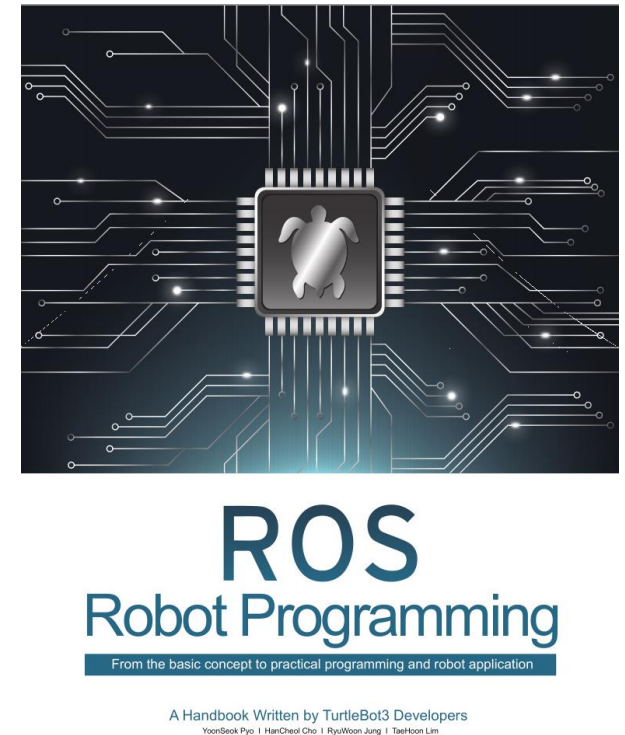
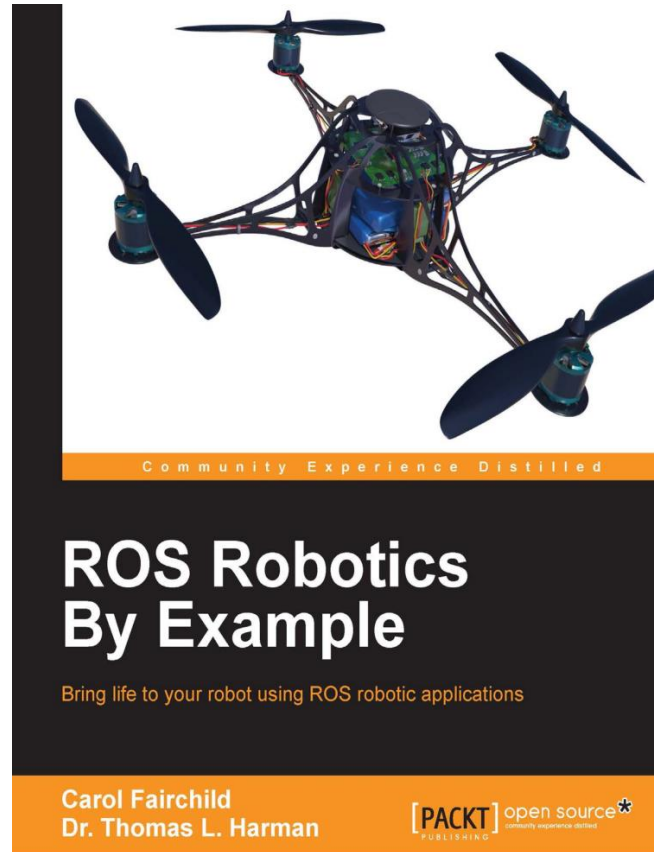
cmake will then know that the project needs to be reconfigured with the addition of msg files



## 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/](https://kapeli.com/cheat_sheets/ROS.docset/)
- **ROS Best Practices**
  - [https://github.com/leggedrobotics/ros\\_best\\_practices/wiki](https://github.com/leggedrobotics/ros_best_practices/wiki)
- **ROS Package Template**
  - [https://github.com/leggedrobotics/ros\\_best\\_practices/tree/master/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 Chambéry  
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)



SYMME