Introduction to ROS
(Robot Operating System)

Julien MARZAT

ECE PARIS
ÉCOLE D'INGÉNIEURS

ONERA
THE FRENCH AEROSPACE LAB

Sources and Resources

- http://doc.ubuntu-fr.org/tutoriel/console_commandes_de_base
- http://wiki.ros.org/tutorials
- https://s3.amazonaws.com/CPR_PUBLIC/LEARN_ROS/ROS_Edu.zip
- http://www.clearpathrobotics.com/assets/guides/ros/index.html
- http://www.rsl.ethz.ch/education-students/lectures/ros.html
- http://mediawiki.isr.ist.utl.pt/wiki/Introduction_to_ROS

What is ROS?

- ROS = Robot Operating System

- Framework for robot software development providing operating system-like functionality
- Originated at Stanford Artificial Intelligence Lab, then further developed at Willow Garage

- Supports all major host operating systems

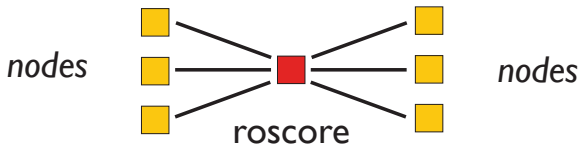
(Ubuntu recommended) Linux Mac OS X Windows Raspbian QNX

- Large user base; getting widespread use
- ROS users forum: http://answers.ros.org

ROS ANSWERS

Basic concept #1: Node

- Modularization in ROS is achieved by separated operating system processes
- *Node* = a process that uses ROS framework
- Nodes may reside in different machines transparently
- Nodes get to know one another via roscore



- roscore acts primarily as a name server
- Nodes use the roscore running in localhost by default overriden by the env. var. ROS_MASTER_URI

Command line tools

```
$ rosnode
```

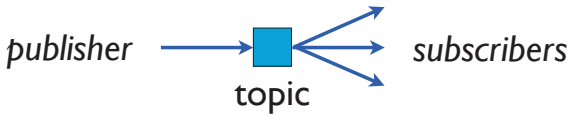
```
rosnode is a command-line tool for printing information about ROS Nodes.

Commands:
          rosnode ping     test connectivity to node
          rosnode list     list active nodes
          rosnode info     print information about node
          rosnode machine  list nodes running on a particular machine
                           or list machines
          rosnode kill     kill a running node
          rosnode cleanup  purge registration information of
                            unreachable nodes

Type rosnode <command> -h for more detailed usage, e.g. 'rosnode ping -h'
```

Basic concept #2: Topic

- *Topic* is a mechanism to send messages from a node to one or more nodes
- Follows a publisher-subscriber design pattern



- *Publish* = to send a message to a topic

  Subscribe = get called whenever a message is published
- Published messages are broadcast to all Subscribers
- Example: LIDAR publishing scan data

Command line tools

```
$ rostopic
```

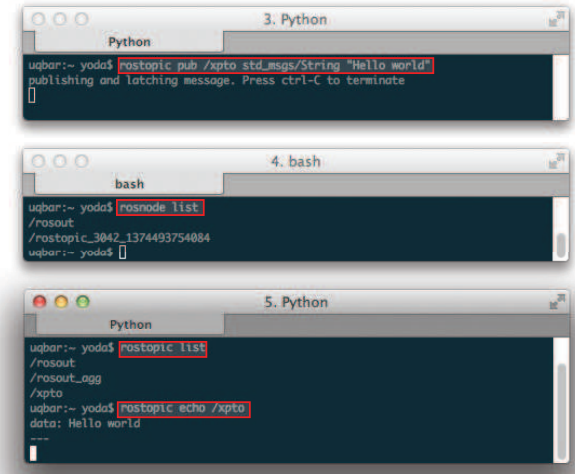
```
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
        rostopic bw     display bandwidth used by topic
        rostopic echo   print messages to screen
        rostopic find   find topics by type
        rostopic hz     display publishing rate of topic
        rostopic info   print information about active topic
        rostopic list   list active topics
        rostopic pub    publish data to topic
        rostopic type   print topic type

Type rostopic <command> -h for more detailed usage, e.g. 'rostopic echo -h'
```

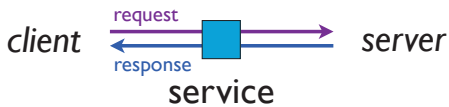
Basic concept #2: Topic

- Demo: publishing an "Hello world" String to topic /xpto



Basic concept #3: Service

- *Service* is a mechanism for a node to send a request to another node and receive a response in return
- Follows a request-response design pattern



- A service is called with a request structure, and in return, a response structure is returned
- Similar to a Remote Procedure Call (RPC)
- Example: set location to a localization node

Command line tools

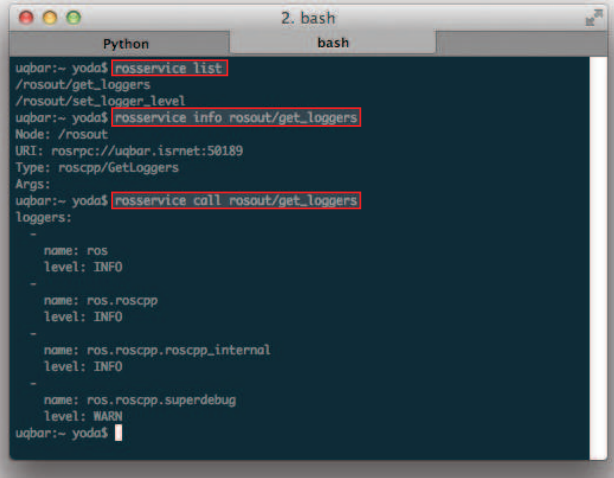
```
$ rosservice
```

```
Commands:
        rosservice args print service arguments
        rosservice call call the service with the provided args
        rosservice find find services by service type
        rosservice info print information about service
        rosservice list list active services
        rosservice type print service type
        rosservice uri  print service ROSRPC uri

Type rosservice <command> -h for more detailed usage, e.g. 'rosservice call -h'
```

Basic concept #3: Service

- Demo: querying and calling a service



Message types

- All messages (including service requests/responses) are defined in text files
- Example: *built-in laser scan data message*

```
--- sensor_msgs/msg/LaserScan.msg ---

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Command line tools

```
$ rosbag
```

```
Usage: rosbag <subcommand> [options] [args]

Available subcommands:
   check
   compress
   decompress
   filter
   fix
   help
   info
   play
   record
   reindex

For additional information, see http://code.ros.org/wiki/rosbag/
```

Development

- Two major languages are supported:
  - C++
  - Python
- ROS provides a portable build system
- *Package* = self-contained directory containing sources, makefiles, builds, etc.

- The code reuse units in ROS are packages
- A large variety of packages can be found on the web

  examples: sensor drivers, simulators, SLAM, image processing, etc.

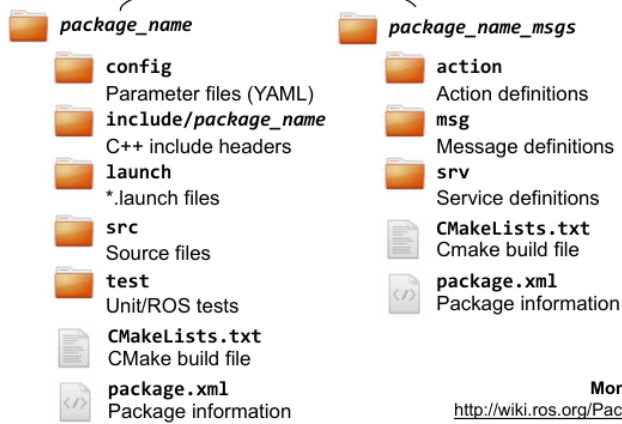
ROS Packages

- ROS software is organized into *packages*, which can contain source code, launch files, configuration files, message definitions, data, and documentation
- A package that builds up on/requires other packages (e.g. message definitions), declares these as *dependencies*

To create a new package, use

```
> catkin_create_pkg package_name
    {dependencies}
```

Separate message definition packages from other packages!



More info
http://wiki.ros.org/Packages

ROS Packages
package.xml

- The `package.xml` file defines the properties of the package
  - Package name
  - Version number
  - Authors
  - **Dependencies on other packages**
  - …

package.xml

```
<?xml version="1.0"?>
<package format="2">
  <name>ros_package_template</name>
  <version>0.1.0</version>
  <description>A template for ROS packages.</description>
  <maintainer email="pfankhauser@e…">Peter Fankhauser</maintainer>
  <license>BSD</license>
  <url type="website">https://github.com/ethz-asl/ros_best_pr…</url>
  <author email="pfankhauser@ethz.ch">Peter Fankhauser</author>

  <buildtool_depend>catkin</buildtool_depend>

  <depend>roscpp</depend>
  <depend>sensor_msgs</depend>
</package>
```

More info
http://wiki.ros.org/catkin/package.xml

ROS Packages
CMakeLists.xml

The `CMakeLists.txt` is the input to the CMakebuild system

1. Required CMake Version (`cmake_minimum_required`)
2. Package Name (`project()`)
3. Find other CMake/Catkin packages needed for build (`find_package()`)
4. Message/Service/Action Generators (`add_message_files()`, `add_service_files()`, `add_action_files()`)
5. Invoke message/service/action generation (`generate_messages()`)
6. Specify package build info export (`catkin_package()`)
7. Libraries/Executables to build (`add_library()`/`add_executable()`/`target_link_libraries()`)
8. Tests to build (`catkin_add_gtest()`)
9. Install rules (`install()`)

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_package_template)

## Use C++11
add_definitions(--std=c++11)

## Find catkin macros and libraries
find_package(catkin REQUIRED
  COMPONENTS
    roscpp
    sensor_msgs
)

…
```

More info
http://wiki.ros.org/catkin/CMakeLists.txt

rqt User Interface

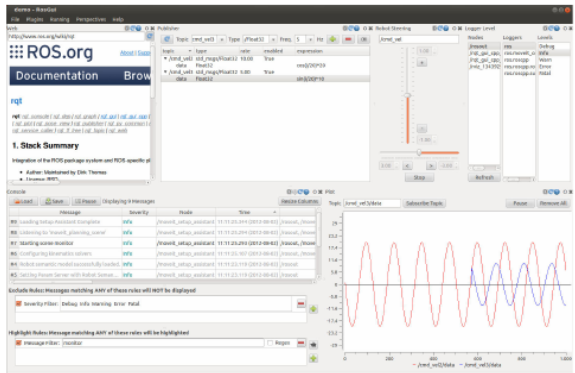
- User interface base on Qt
- Custom interfaces can be setup
- Lots of existing plugins exist
- Simple to write own plugins

Run RQT with

```
> rosrun rqt_gui rqt_gui
```

or

```
> rqt
```



More info

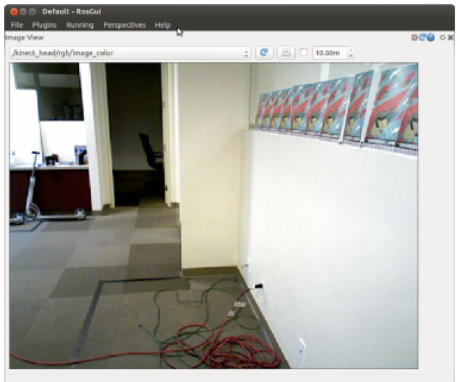
http://wiki.ros.org/rqt/Plugins

rqt User Interface
rqt_image_view

- Visualizing images

Run *rqt_graph* with

```
> rosrun rqt_image_view rqt_image_view
```



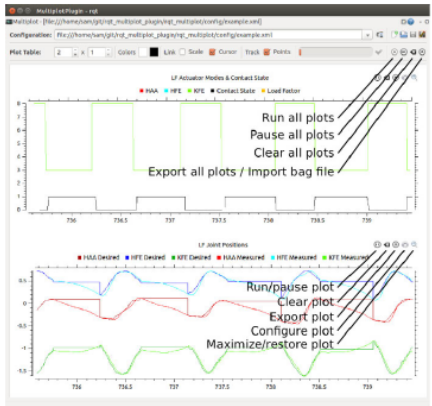
More info
http://wiki.ros.org/rqt_image_view

rqt User Interface
rqt_multiplot

- Visualizing numeric values in 2D plots

Run *rqt_multiplot* with

```
> rosrun rqt_multiplot rqt_multiplot
```



More info
http://wiki.ros.org/rqt_multiplot

rqt User Interface
rqt_graph

- Visualizing the ROS computation graph

Run *rqt_graph* with

```
> rosrun rqt_graph rqt_graph
```

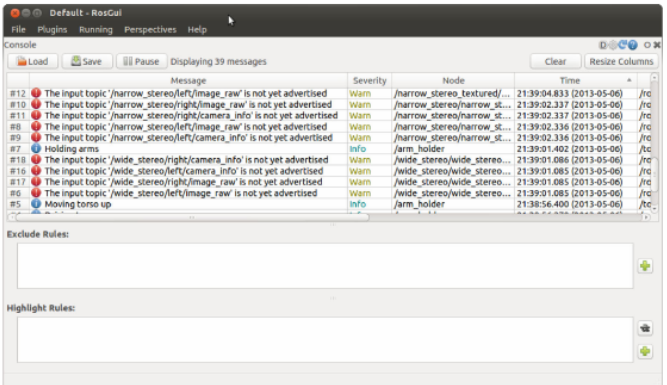
More info
http://wiki.ros.org/rqt_graph

rqt User Interface
rqt_console

- Displaying and filtering ROS messages

Run *rqt_console* with

```
> rosrun rqt_console rqt_console
```



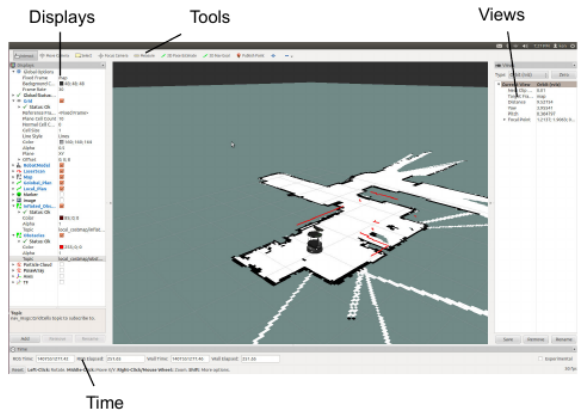
More info
http://wiki.ros.org/rqt_console

RViz

- 3D visualization tool for ROS
- Subscribes to topics and visualizes the message contents
- Different camera views (orthographic, top-down, etc.)
- Interactive tools to publish user information
- Save and load setup as RViz configuration
- Extensible with plugins

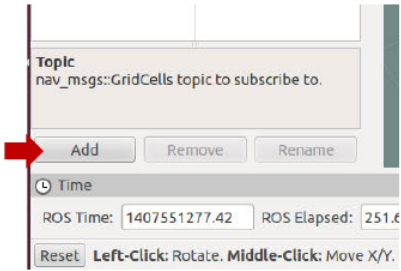
Run RViz with

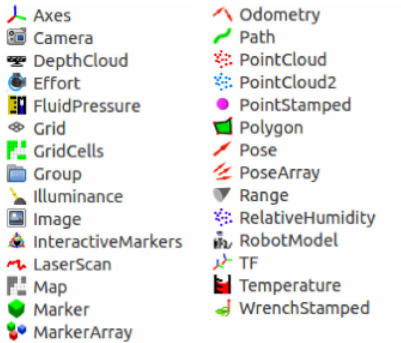
```
> rosrun rviz rviz
```

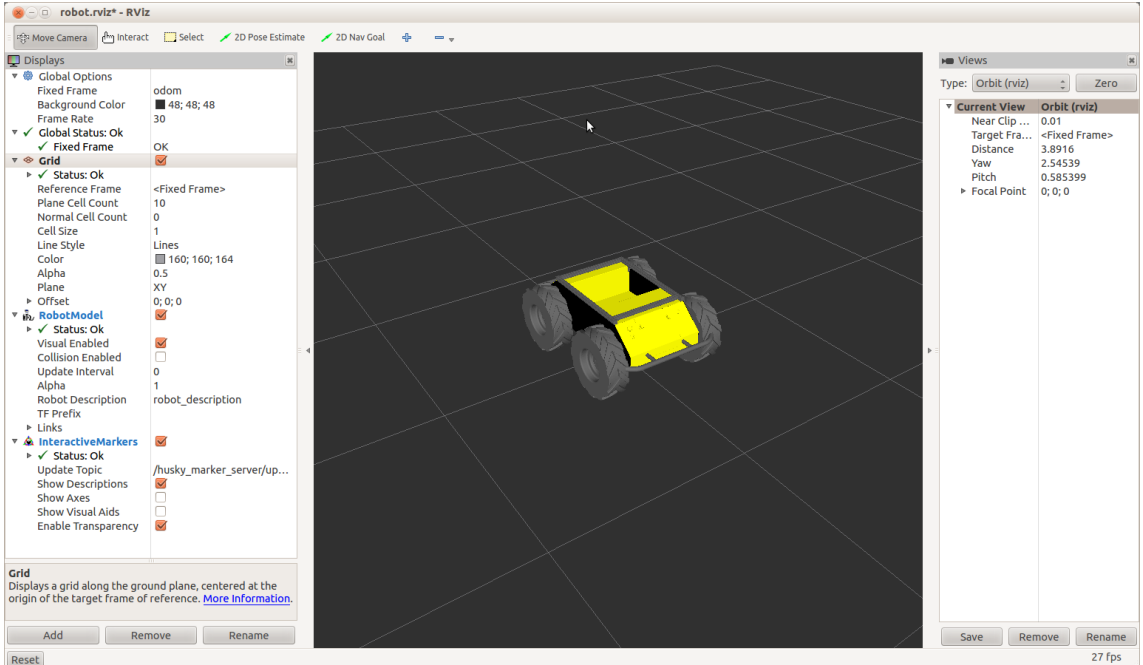


More info
http://wiki.ros.org/rviz

RViz
Display Plugins






robot.rviz* - RViz
Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Displays
Global Options
Fixed Frame
odom
Background Color
48; 48; 48
Frame Rate
30
Global Status: Ok
Fixed Frame
OK
Grid
Status: Ok
Reference Frame
<Fixed Frame>
Plane Cell Count
10
Normal Cell Count
0
Cell Size
1
Line Style
Lines
Color
160; 160; 164
Alpha
0.5
Plane
XY
Offset
0; 0; 0
RobotModel
Status: Ok
Visual Enabled
Collision Enabled
Update Interval
0
Alpha
1
Robot Description
robot_description
TF Prefix
Links
InteractiveMarkers
Status: Ok
Update Topic
/husky_marker_server/up…
Show Descriptions
Show Axes
Show Visual Aids
Enable Transparency
Grid
Displays a grid along the ground plane, centered at the origin of the target frame of reference. More Information.
Add
Remove
Rename
Reset
Views
Type:
Orbit (rviz)
Zero
Current View
Orbit (rviz)
Near Clip …
0.01
Target Fra…
<Fixed Frame>
Distance
3.8916
Yaw
2.54539
Pitch
0.585399
Focal Point
0; 0; 0
Save
Remove
Rename
27 fps

roslaunch tool

- Inpractical to launch manually many ROS nodes

- roslaunch allows automatic launch of nodes from a single shell command
- roslaunch is configured using XML files
- Launch files are typically stored in the launch/ directory of a package

- roslaunch tool arguments:

  roslaunch [package] filename [arg_name:=value]*

Launch files

- Minimal launch file for the xpto package

```
# file l1.launch
<launch>
        <node pkg="xpto" name="node1" type="publisher.py"/>
</launch>
```

  - attributes:
    pkg="package_name"
    name="node_name"
    type="executable_filename"

Launch files

- Running...

```
$ roslaunch xpto l1.launch
[...]
core service [/rosout] found
process[node1-1]: started with pid [23774]
```

```
$ rosnode list
/node1
/rosout

$ rostopic list
/abc
/rosout
/rosout_agg
```

Launch files

- Several nodes

```
# file l2.launch
<launch>
        <node pkg="xpto" name="publisher" type="publisher.py"/>
        <node pkg="xpto" name="subscriber" type="subscriber.py"
         output="screen"/>
</launch>
```

```
$ roslaunch xpto l2.launch
[...]
process[publisher-1]: started with pid [23919]
process[subscriber-2]: started with pid [23920]
Received 'hello world #1'
Received 'hello world #2'
Received 'hello world #3'
```

Launch files

- Other <node> arguments:
  - launch node on a different machine
    machine="hostname"
    (use <machine> tags to declare machine names)
  - restart node whenever it quits
    respawn="true"
  - start node in a different namespace
    ns="namespace"
  - pass arguments to node
    args="arg1 arg2 arg3 ..."
  - ...

Launch files

- Tags allowed inside the <node> tag:
  - set environment variables
    <env name=''variable'' value=''value''/>
  - remap names (nodes, topics, parameters)
    <remap from="original" to="new"/>
  - handle ROS parameters
    <rosparam command="load|dump|delete" file=''...''/>
  - send parameters to parameters server
    <param name=''...'' type=''...'' value=''...''/>
- *These tags can also be used in other scopes, i.e., globally scoped within the launch file*

Launch files

- Other relevant tags:
  - include launch files
    <include file=''filename''/>
  - group tags within a scope
    <group name=''...''>
    ...
    </group>
  - declare machines
    <machine name="..." address="..." user="...'' ...>
    ...
    </machine>

Launch files

- Substitution arguments (i.e., macros)
  - package path name
    $(find package_name)
  - evaluates to value declared with tag <arg>
    $(arg argument_name)
  - evaluates to an environment variable that has to exist
    $(env variable_name)
  - same as $(env ...) but defaults to a given value if undefined
    $(optenv variable_name)
  - generate a unique (anonymous) name
    $(anon base_name)

Launch files

- Simple example:

```
# file l3.launch
<launch>
        <include ns="foo" file="$(find xpto)/launch/l2.launch"/>
        <include ns="bar" file="$(find xpto)/launch/l2.launch"/>
</launch>
```

```
$ rosnode list
/bar/publisher
/bar/subscriber
/foo/publisher
/foo/subscriber
/rosout

$ rostopic list
/bar/abc
/foo/abc
/rosout
/rosout_agg
```

Launch files

- Example top level organization

```
<launch>
  <group name="wg">
    <include file="$(find pr2_alpha)/$(env ROBOT).machine" />
    <include file="$(find 2dnav_pr2)/config/new_amcl_node.xml" />
    <include file="$(find 2dnav_pr2)/config/base_odom_teleop.xml" />
    <include file="$(find 2dnav_pr2)/config/lasers_and_filters.xml" />
    <include file="$(find 2dnav_pr2)/config/map_server.xml" />
    <include file="$(find 2dnav_pr2)/config/ground_plane.xml" />

    <!-- The navigation stack and associated parameters -->
    <include file="$(find 2dnav_pr2)/move_base/move_base.xml" />
  </group>
</launch>
```

Message type definition

- Message types are defined in simple text files in the msg/ directory
- Sintax:

  # this is a comment

  fieldtype1 fieldname1

  fieldtype2 fieldname2

  ...
- Example:

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

Message field types

- Built-in types:

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	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

  - use '[]' after type to denote an array
    example: float64[] *is a string of float64's*

Message field types

- **Examples from** geometry_msgs **package**
  - Point.msg

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

  - Quaternion.msg

```
# This represents an orientation in free space in quaternion form.
float64 x
float64 y
float64 z
float64 w
```

Message field types

- Message types are themselves field types that can be used in another message type definitions
  - example: Header type, defined in std_msgs/Header.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.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
# 0: no frame
# 1: global frame
string frame_id
```

  - this type is almost always used in other message types

Message field types

- **Example from** sensor_msgs **package:** LaserScan.msg

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Message field types

- **Examples from** geometry_msgs **package**
  - Pose.msg

```
# A representation of pose in free space, composed of postion and orientation.
Point position
Quaternion orientation
```

  - PoseWithCovariance.msg

```
# This represents a pose in free space with uncertainty.

Pose pose

# Row-major representation of the 6x6 covariance matrix
# The orientation parameters use a fixed-axis representation.
# In order, the parameters are:
# (x, y, z, rotation about X axis, rotation about Y axis, rotation about Z axis)
float64[36] covariance
```

Available message types

- **from** std_msgs **package:**

```
Bool.msg                Header.msg              String.msg
Byte.msg                Int16.msg               Time.msg
ByteMultiArray.msg      Int16MultiArray.msg     UInt16.msg
Char.msg                Int32.msg               UInt16MultiArray.msg
ColorRGBA.msg           Int32MultiArray.msg     UInt32.msg
Duration.msg            Int64.msg               UInt32MultiArray.msg
Empty.msg               Int64MultiArray.msg     UInt64.msg
Float32.msg             Int8.msg                UInt64MultiArray.msg
Float32MultiArray.msg   Int8MultiArray.msg      UInt8.msg
Float64.msg             MultiArrayDimension.msg UInt8MultiArray.msg
Float64MultiArray.msg   MultiArrayLayout.msg
```

Available message types

- **from** geometry_msgs **package:**

```
Point.msg                          QuaternionStamped.msg
Point32.msg                        Transform.msg
PointStamped.msg                   TransformStamped.msg
Polygon.msg                        Twist.msg
PolygonStamped.msg                 TwistStamped.msg
Pose.msg                           TwistWithCovariance.msg
Pose2D.msg                         TwistWithCovarianceStamped.msg
PoseArray.msg                      Vector3.msg
PoseStamped.msg                    Vector3Stamped.msg
PoseWithCovariance.msg             Wrench.msg
PoseWithCovarianceStamped.msg      WrenchStamped.msg
Quaternion.msg
```

Available message types

- **from** sensor_msgs **package:**

```
CameraInfo.msg          JoyFeedback.msg         PointCloud2.msg
ChannelFloat32.msg      JoyFeedbackArray.msg    PointField.msg
CompressedImage.msg     LaserEcho.msg           Range.msg
FluidPressure.msg       LaserScan.msg           RegionOfInterest.msg
Illuminance.msg         MagneticField.msg       RelativeHumidity.msg
Image.msg               MultiEchoLaserScan.msg  Temperature.msg
Imu.msg                 NavSatFix.msg           TimeReference.msg
JointState.msg          NavSatStatus.msg
Joy.msg                 PointCloud.msg
```

Available message types

- **Example from** sensor_msgs **package:** LaserScan.msg

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```