A guided journey to the use of ROS

16-311 Qatar - Gianni A. Di Caro*

Abstract

This document is a step-by-step guide to learn how to use and program robots in ROS. It is mainly intended as a companion text for the homework assignments in the course. It assumes knowledge about the general design and architectural concepts behind ROS, as well as about the role of the main ROS components (nodes, topics, publish/subscribe, services, packages) and their relations. The document "develops" over time: when needed, new sections will be incrementally added to support the work required in new homework assignments.

Contents

*The content of this documents is based on a number of Internet sources, including ROS wiki pages.

1 General info on ROS packages and Catkin workspaces

- Packages are the most atomic unit of build and the unit of release.
- A package contains the *source* files for one node or more, and *configuration* files.
- A *node* is the ROS term for an executable that is connected to the ROS network.
- A ROS package is a *folder structure* inside a *catkin workspace* that has a `package.xml` file in it.
- A *catkin workspace* is a set of directories in which a set of related ROS code/packages live (`catkin` ∼ ROS build system: `CMake + Python scripts`).
- It's possible to have multiple workspaces, but work can be performed on only one-at-a-time.

Refer to **lecture slides for an overview of ROS**.

2 Create a catkin workspace

Let's start by creating a catkin workspace, named `catkin_ws` in the root of the home directory. Also the `src` sub-folder, where nodes' code will be, is created:

```
$ mkdir -p ∼/catkin_ws/src
```

Initialize the workspace (from the `src` sub-folder). This creates a `CMakeLists.txt` file in the `src` directory. This file is the input to the `CMake` build system used for building the software packages inside this workspace:

```
$ cd ∼/catkin_ws/src
$ catkin_init_workspace
```

The following `catkin_make`, executed from the workspace root, *builds* the packages in the workspace. In this case, since we haven't written yet any code for the package, it will only create the sub-folders `devel` and `install`, that contain development and execution level files.

```
$ cd ~/catkin_ws
$ catkin_make
```

`catkin_make` can be used to compile the code for all packages in the workspace. To compile a specific package, including its dependencies, the following command can be used:

```
$ catkin_make --only-pkg-with-deps <package_name>
```

It is necessary to *register the workspace in ROS*, using the command:

```
$ . ~/catkin_ws/devel/setup.bash
```

You can verify that your workspace has been correctly registered by issuing:

```
$ roscd
```

that should take you to the directory set by the environment variable `ROS_WORKSPACE`, that should precisely redirect to the ROS workspace.

In the next section, a first package will be added to the just created workspace. If multiple packages have been created, the layout of the catkin workspace would look like in Figure 1.

```
workspace_folder/         -- WORKSPACE
  src/                    -- SOURCE SPACE
    CMakeLists.txt        -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CATKIN_IGNORE       -- Optional empty file to exclude package_n from being processed
      CMakeLists.txt
      package.xml
      ...
  build/                  -- BUILD SPACE
    CATKIN_IGNORE         -- Keeps catkin from walking this directory
  devel/                  -- DEVELOPMENT SPACE (set by CATKIN_DEVEL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
  install/                -- INSTALL SPACE (set by CMAKE_INSTALL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
```

Figure 1: Layout of a catkin workspace with n packages.

The content of the different folder structures inside a workspace is summarized in the table of Figure 2.

Source space	Contains the source code of catkin packages. Each folder within the source space contains one or more catkin packages.
Build Space	is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here.
Development (Devel) Space	is where built targets are placed prior to being installed
Install Space	Once targets are built, they can be installed into the install space by invoking the install target.

Figure 2: Contents of the top folders of a catkin workspace.

3 Create the first package

Let's create a new package, named `random_control`. The package will have dependencies on rospy, the Python's API to interact with and use ROS entities, and on `std_msgs`, the standard messages, that define the data types that will be used for publishing topic messages. The package will consist of three nodes that will exchange information messages for the purpose of moving and monitoring a mobile robot. The purpose of the package is precisely to show the basic use of *topics* in the publish/subscribe scheme and of the basic elements of a ROS node. Therefore, the "control" will be quite dumb, indeed it will be making the robot moving quite at random!

The command for start creating the package has to be executed within the `src` folder of the workspace:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg random_control std_msgs rospy
```

☛ To *remove* a package (e.g., created in the wrong directory), just remove the folder and rebuild the workspace with `catkin_make`.

The package creation command will create the following files in `/catkin_ws/src`:

```
random_control/CMakeLists.txt
random_control/package.xml
random_control/src
```

The table in Figure 3 shows the general folder structure of a package directory in a catkin workspace. The source files in `src` implement ROS *nodes*. These can be written in multiple languages. In our case, we will implement all nodes in Python, which is the reason why we included `rospy` in the list of dependencies.

Nodes can be *launched* individually or in groups, using the launch files in the `launch` folder, which we will add later on.

Folders `msg` and `rsv` contain, respectively, the definition of *custom message types* and of *services/actions* that the package make available to other nodes/packages. These folders are optional: a package is not required to define or provide custom messages and services. In this first example, no custom messages or services will be defined.

Directory	Explanation
include/	C++ include headers
src/	Source files
msg/	Folder containing Message (msg) types
srv/	Folder containing Service (srv) types
launch/	Folder containing launch files
package.xml	The package manifest
CMakeLists.txt	CMake build file

Figure 3: Directory structure of a package folder in a catkin workspace.

3.1 Commands to retrieve information about existing packages

At this stage, the package `random_control` has been created and registered in the ROS file system, but it is still empty. The following command:

```
$ rospack list
```

lists all the packages present in your ROS system, which should now include `random_control` (e.g., use `$ rospack list | grep random_control`). You should see th following entry (in a list of more than 360 entries):

```
/home/name_of_root_directory/catkin_ws/src/random_control
```

The commans `rospack` can provide a number of different information about the packages including their *dependency trees and location in the filesystem*. For instance, executing

```
$ rospack depends random_control
```

a relatively long list of dependencies will appear, that include standard dependencies plus the dependencies related to the dependencies explicitly defined at package creation.

To find where a package is in the *filesystem*:

```
$ rospack find package_name
```

☛ Since ROS knows where packages are, the following command will take you to the *root directory of the workspace of the named package*:

```
$ roscd package_name
```

which makes it quite handy to navigate throughout the ROS filesystem!

☛ **Documentation about all ROS commands** (as well as other components) is readily accessible through

```
http://wiki.ros.org/name_of_the_command
```

(`http://wiki.ros.org/rospack` for `rospack`). The `Overview` page of a command includes at the end of the page a link to a page with the complete documentation.

☛ Check the **ROS cheatseet** provided on the course website: it contains all the rosbash commands (including `rospack`) and their options! Call a command with the `help` argument to get a description of the available options:

```
$ <rosbash_command> help
```

☛ **TAB completion** works to complete commands and list/complete admissible arguments. If TAB completions seems not to work, it means that something wrong is in there! In some cases, executing the following command it can help, since it reconstructs the entire tree of dependencies:

```
$ rospack profile
```

3.2 The *task* tackled by our package: random robot controller

This first package will make the robot moving in *open-loop* at random. The mobile robot that we will consider is the *TurtleBot 2*. We can say that we will be building a *random open-loop robot controller*. At random intervals, the controller, changes the *linear and angular velocity* of a TurtleBot robot. Also velocity changes are random, overall resulting into a random motion. Whenever a velocity change happens, the controller notifies the external world of the new velocity and of the new estimated position in the world coordinate system. The controller operates in open-loop, meaning that no input data from robot sensors are used to possibly take intelligent / safe decisions regarding the motion of the robot (i.e., the robot is "blind").

In principle, we could implement this simple behavior in one single "monolithic" controller function. However, it's convenient to split the scenario in multiple, communicating modules. In turn, each module will be a *node* in the ROS system. This way of proceeding will it allow, later on, to flexibly change the behavior inside the modules, without the need to change the organization of the controller. In particular, we will use the following modules/nodes, each tackling on a different aspect of the scenario:

- `random_values`: random process for changing velocities;
- `move_robot`: use the velocity values defined by `random_values` to set robot's velocity and move it;
- `pose_monitor`: notify each change in velocity and update robot's position in the world coordinate frame.

In practice, one module deals with generating velocity values according to "some" random process, one module gets these values and uses them to move the robot, one module gets notified of the changes and keeps an eye on what's going on about the (estimated) robot position.

Therefore, we need to create three nodes, and use (at least) three topics for letting them communicating, that we define as follows:

- `/new_vel`: for communicating new velocity values from node `random_values` to node `move_robot`;

- `/cmd_vel` for communicating velocity values from node `move_robot` to the on-board PID controller managing wheels actuation;
- `/change`: for communicating a change in the actual robot velocity from `move_robot` to `pose_monitor`.

For testing, for the time being we will not move an actual robot, but rather we will use the simulated model of the TurtleBot 2 in Gazebo.

The node organization of the package/controller is shown in Figure 4, together the with the named topics that will be used.

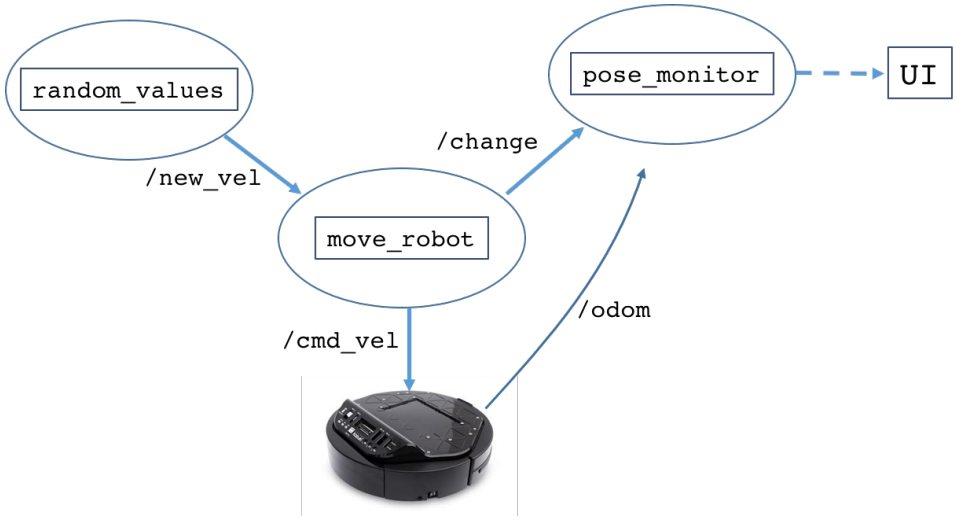

Figure 4: Node structure and topics for the `random-control` package.

3.3 Create the node files of the package

Let's start by writing the contents of the nodes. First, in `src` create the python files for the nodes and make them executable:

```
$ touch random_values.py move_robot.py pose_monitor.py
$ chmod +x *.py
```

The content of each node file is described in the sections that follow.

☛ Once created, a node file in a package can be *executed* with:

```
$ rosrun name_of_package name_of_node
```

☛ Before running any node, the **ROS Master** node needs to be activated, since it's the ROS Master that deals with all node/topic/service registrations and manage the entire ROS network. To run the ROS Master:

```
$ roscore
```

If a ROS master is already active, the above command will return with an error. Otherwise, a ROS master will start running, and the command will report a number of parameters regarding the ROS distribution, the address of the machine when it runs, the connection port, and so on.

An alternative (and more powerful and flexible) method for executing one or more nodes in a package is using a **launch** file. Launch files are located in the `launch` folder, and also allow for specifying node arguments. Later on we will see how to write a launch file for our package.

In the following, for each node, the full pyhton code is first reported, and then the code is explained piece by piece, using the code as a mean to introduce and describe basic ROS concepts and components.

3.4 Node `random_values`

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

# Import the Python library for ROS
import rospy

# Import the library for generating random numbers
import random

# Import the Twist message from the geometry_msgs package
# Twist data structure is used to represent velocity components
from geometry_msgs.msg import Twist

class RndVelocityGen():

    def __init__(self):
        # Initiate a node named 'random_velocity'
        rospy.init_node('random_velocity')

        # Create a Publisher object, that will publish on /new_vel topic
        # messages of type Twist
        self.vel_pub = rospy.Publisher('/new_vel', Twist, queue_size=1)

        # Creates var of type Twist
        self.vel = Twist()

        # Assign initial velocities
        self.vel.linear.x = 0.5     # m/s
        self.vel.angular.z = 0.5    # rad/s

        # Set max values for velocities, min is set to 0
        self.linear_vel_x_max = 1.2
        self.angular_vel_z_max = 0.5

        # Set max value for max time interval between velocity changes,
        #  min is set to 1
        self.max_interval = 10

    def generate_rnd_values(self):

        # Loop until someone stops the program execution
        while not rospy.is_shutdown():

            # positive x-vel move the robot forward, negative backward
            x_forward = random.choice((-1,1))

            # positive z-vel rotate robot counterclockwise, negative clockw
            z_counterclok = random.choice((-1,1))

            self.vel.linear.x  = (x_forward *
                                  random.uniform(0, self.linear_vel_x_max))
            self.vel.angular.z = (z_counterclok *
```

```
                                random.uniform(0, self.angular_vel_z_max))

            self.vel_pub.publish(self.vel)

            now = rospy.get_rostime()
            print "Time now: ", now.secs

            next = (random.randint(1, self.max_interval))

            rospy.loginfo("Twist: [%5.3f, %5.3f], next change in %i secs - ",
                          self.vel.linear.x, self.vel.angular.z, next)

            rospy.sleep(next)           # Sleeps for the selected seconds

if __name__ == '__main__':
    try:
        generator = RndVelocityGen()

        generator.generate_rnd_values()

    except rospy.ROSInterruptException:
        pass
```

3.4.1 Import statements and Topic messages

The first lines define the libraries to import and the message types that are used in the code:

```
# Import the Python library for ROS
import rospy

# Import the library for generating random numbers
import random

# Import the Twist message from the geometry_msgs package
# Twist data structure is used to represent velocity components
from geometry_msgs.msg import Twist
```

The first line makes sure that the node is executed as a Python script (given that it is given executable permissions). Every Python ROS node must have this declaration at the top since `rospy` needs always to be imported if writing a ROS node in Python. `random` is imported because we will use *random number generators*.

☛ The `geometry_msgs.msgs` import is so that message types from `geometry_msgs` types can be used for publishing. In our case, the subset `Twist` data type is imported. `Twist` is the common data format used to represent velocities (for the TurtleBot and other mobile robot models):

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

A `Twist` object is composed of two `Vector3` arrays, each with three `float64` elements, expressing respectively the *linear* and the *angular* velocity. In turn, the `Vector3` type is defined in `geometry_msgs`, while the `float64` type is defined in `std_msgs/Float64`.

☛ In general, `std_msgs` provides basic messages types, that are used in the ROS packages to build more specialized data message types. The list of message types included in `std_msgs` is the following, most of them have obvious meaning:

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

☛ With the use of the command `rosmsg`, the list of message types available in the ROS system, or in a particular package, can be inspected. The following arguments can be passed:

```
$ rosmsg show      Show message description (format, where)
$ rosmsg info      Alias for rosmsg show
$ rosmsg list      List all messages in the ROS filesystem
$ rosmsg md5       Display message md5sum
$ rosmsg package   List messages in a specific package
$ rosmsg packages  List packages that contain messages
```

3.4.2 Initialization statements, publish/subscribe declarations

For the sake of clarity, modularity, and reusability, the code of the node is organized in a class, named `RndVelocityGen`. However, this is only a specific (yet always suggested) design choice for writing node code. When we write a node we are writing an *executable (Python) code*, that can be structured according to the personal preferences.

The initialization actions are included in the standard `__init__` class function, which is executed when an instance of a class object is declared.

```
    def __init__(self):
        # Initiate a node named 'random_velocity'
        rospy.init_node('random_velocity')

        # Create a Publisher object, that will publish on /new_vel topic
        # messages of type Twist
        self.vel_pub = rospy.Publisher('/new_vel', Twist, queue_size=1)

        # Creates var of type Twist
        self.vel = Twist()

        # Assign initial velocities
        self.vel.linear.x = 0.5     # m/s
        self.vel.angular.z = 0.5    # rad/s

        # Set max values for velocities, min is set to 0
        self.linear_vel_x_max = 1.2
        self.angular_vel_z_max = 0.5
```

```
        # Set max value for max time interval between velocity changes,
        # min is set to 1
        self.max_interval = 10
```

☛ The `rospy.init_node()` function call needs to be in every node definition: it tells `rospy` the name of the node, which is a necessary piece of information to let the `rospy` API communicate with the ROS Master. In our case, the name given to the node is `random_velocity` (but any other usitable name could have been chosen). The name must be a *base name*, meaning that it cannot contain any slashes "/".

☛ More in general, `rospy.init_node()` is defined with the following syntax and default arguments:

```
rospy.init_node(name, anonymous=False, log_level=rospy.INFO, disable_signals=False)
```

Names in ROS must be *unique*. In cases where having unique names for a particular node is not important, the node can be initialized with an anonymous name (setting `anonymous=True`). In this case, a random number is added to the end of the given node's name, to make it unique automatically. Unique names are important for nodes like drivers, where it is an error if more than one is running. If two nodes with the same name are detected on a ROS graph, the older node is shutdown.

The `log_level = rospy.INFO` argument sets the default log level for publishing log messages to the `rosout` system's topic. See Logging `http://wiki.ros.org/rospy/Overview/Logging` for the description of the many modalities that ROS offers for data logging.

The argument `disable_signals` allows to enable/disable standard signal handlers. If enabled, system's signal handlers are registered in `rospy`, such that, for instance, it is possible to exit on `Ctrl-C`.

☛ Since the aim of our node is to generate velocities according to some random scheme, and publish them for letting `move_robot` use the information, it is necessary to define out the node interfaces to the rest of the ROS system. This is realized with the call `self.vel_pub = rospy.Publisher('/new_vel', Twist, queue_size=1)`, whose general form is:

```
publisher_var = rospy.Publisher('topic_name', message_type, queue_size)
```

The function call tells that the node is *publishing* to the `new_vel` topic using messages of type `Twist`. The `queue_size` parameter limits the amount of queued messages, which can be increased in the case any subscriber is not receiving the messages fast enough. The class variable `vel_pub` holds the reference to the topic where to publish.

The other initialization commands are quite self-explanatory. In particular, `self.linear_vel_x_max = 1.2` and `self.angular_vel_z_max = 0.5` set the maximum value that can be assigned respectively to the linear and angular velocities. Since velocity values are sampled irregularly every t seconds, where t is a random value, `self.max_interval = 10` sets the maximum value for t.

3.4.3 Node execution loop

The function `generate_random_values()` does the job of generating random velocity values and publish them on the `new_vel` topic.

```

```

```
def generate_rnd_values(self):

    # Loop until someone stops the program execution
    while not rospy.is_shutdown():

    # positive x-vel move the robot forward, negative backward
    x_forward = random.choice((-1,1))

    # positive z-vel rotate the robot counterclockwise, negative clockwise
    z_counterclok = random.choice((-1,1))

    self.vel.linear.x  = (x_forward *
                          random.uniform(0, self.linear_vel_x_max))
    self.vel.angular.z = (z_counterclok *
                          random.uniform(0, self.angular_vel_z_max))

    self.vel_pub.publish(self.vel)

    now = rospy.get_rostime()
    print "Time now: ", now.secs

    next = (random.randint(1, self.max_interval))

    rospy.loginfo("Twist: [%5.3f, %5.3f], next change in %i secs - ",
                  self.vel.linear.x, self.vel.angular.z, next)

    rospy.sleep(next)           # Sleeps for the selected seconds
```

The action loop is a fairly standard `rospy` construct: checking the `rospy.is_shutdown()` flag and then doing work. If `is_shutdown()` is true (e.g. there is a Ctrl-C or otherwise), the program should exit.

The first four lines in the body of the while loop, make use of Python's random generator library to assign (at random) *linear and angular velocities* within the ranges defined in the init function. A linear velocity can be *forward* (moving in the positive direction of the x axis), or *backward* (negative values) with respect to its defined heading. Similarly, an angular velocity can make the robot moving *counterclockwise* (positive values) or *clockwise* (negative values). The `Twist` object variable `vel` holds the defined velocities, and is published on the topic `new_vel` using the reference variable `vel_pub` with `self.vel_pub.publish(self.vel)`.

☛ In the last lines, a new time interval is generated and stored in the variable `next`, which is then used in `rospy.sleep(next)` to put the node to sleep for the related number of seconds. In practice, this defines the *rate* of activation of the node. It is important to remark that ROS is *not a real-time system*, meaning that there's no guarantee that the node will sleep the "precise" number of seconds `next`. However, from ROS 2.0, real-time performance will be guaranteed.

The call to `get_rostime()` is intended to show how to access (and then display) ROS time.

3.4.4 Main function

```
if __name__ == '__main__':
    try:
        generator = RndVelocityGen()

        generator.generate_rnd_values()

    except rospy.ROSInterruptException:
        pass
```

This part of the code is fairly standard: a class object of type `RndVelocityGen` is created (that determines the execution of the `__init__` function) and then the class function `generate_rnd_values()` is executed that in principle runs forever, generating new random velocities according to an irregular, random frequency.

☛ In addition to standard Python `__main__` check, the code catches a `rospy.ROSInterruptException` exception, which can be thrown by `rospy.sleep()` (and `rospy.Rate.sleep()`) methods when `Ctrl-C` is pressed on the terminal window when the node is otherwise shutdown. The reason this exception is raised is that the code doesn't continue executing when waking up after the sleep().

3.5 Node `move_robot`

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

# Import the Python library for ROS
import rospy
import time

# Import the Twist message
from geometry_msgs.msg import Twist

class MoveRobot():

    def __init__(self):
        # Initiate a named node
        rospy.init_node('MoveRobot', anonymous=False)

        # tell user how to stop TurtleBot
        rospy.loginfo("CTRL + C to stop the turtlebot")

        # What function to call when ctrl + c is issued
        rospy.on_shutdown(self.shutdown)

        # subscribe to the topic published by node random_values
        self.new_velocity_sub = rospy.Subscriber('/new_vel', Twist,
                                                 self.callback_new_velocity)

        # Create a Publisher object, will publish on cmd_vel_mux/input/teleop topic
        # to which the robot (real or simulated) is a subscriber
        self.vel_pub = rospy.Publisher('/cmd_vel_mux/input/teleop', Twist,
                                       queue_size=5)

        # Creates a var of msg type Twist for velocity
        self.vel = Twist()

        # publish a topic to notify node pose_monitor of the changed velocity
        self.new_velocity_sub = rospy.Publisher('/change', Twist, queue_size=1)

        # Set a publish velocity rate of in Hz
        self.rate = rospy.Rate(5)

    def callback_new_velocity(self, msg):

        #print "Received Twist msg: ", msg
        rospy.loginfo("Received velocity [linear x]%5.2f  [angular z]%5.2f",
                      msg.linear.x, msg.angular.z)

        # set velocities as received from the /new_vel topic
        self.vel.linear.x  =  msg.linear.x
```

```
        self.vel.angular.z =  msg.angular.z

        # publish the new velocities on the /cmd_vel_mux/input/teleop topic
        self.new_velocity_sub.publish(self.vel)

    def send_velocity_cmd(self):
        self.vel_pub.publish(self.vel)

    def shutdown(self):
        print "Shutdown!"
        # stop TurtleBot
        rospy.loginfo("Stop TurtleBot")

        self.vel.linear.x = 0.0
        self.vel.angular.z = 0.0

        self.vel_pub.publish(self.vel)

        # makes sure robot receives the stop command prior to shutting down
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        controller = MoveRobot()

        # keeping doing until ctrl+c
        while not rospy.is_shutdown():

            # send velocity commands to the robots
            controller.send_velocity_cmd()

            # wait for the selected mseconds and publish velocity again
            controller.rate.sleep()

    except:
        rospy.loginfo("move_robot node terminated")
```

Most of the statements in the code of this node are equivalent to those just discussed for node `random_values`, therefore, in the following these parts won't be (re)discussed.

3.5.1 Initialization statements, publish/subscribe declarations, callback

```
def __init__(self):
     # Initiate a named node
     rospy.init_node('MoveRobot', anonymous=False)

     # tell user how to stop TurtleBot
     rospy.loginfo("CTRL + C to stop the TurtleBot")

     # What function to call when ctrl + c is issued
     rospy.on_shutdown(self.shutdown)

     # subscribe to the topic published by node random_values
     self.new_velocity_sub = rospy.Subscriber('/new_vel', Twist,
                                              self.callback_new_velocity)

     # Create a Publisher object, will publish on cmd_vel_mux/input/teleop topic
     # to which the robot (real or simulated) is a subscriber
     self.vel_pub = rospy.Publisher('/cmd_vel_mux/input/teleop', Twist,
                                    queue_size=5)
```

```
    # Creates a var of msg type Twist for velocity
    self.vel = Twist()

    # publish a topic to notify node pose_monitor of the changed velocity
    self.new_velocity_sub = rospy.Publisher('/change', Twist, queue_size=1)

    # Set a publish velocity rate of in Hz
    self.rate = rospy.Rate(5)
```

☛ The call to the function `rospy.on_shutdown(self.shutdown)` tells what function to call on shutdown (e.g., when a `CRTL + C` is issued). In this case, the call is to the class function `shutdown()`, that sets both the linear and angular velocities to zero, publish them to the robot, and makes one second sleep before quitting, to ensure that the robot receives the commands.

This node is designed to receive velocity values from another node, publish them to the robot, and notify another node about the velocity change. Therefore, the node needs to subscribe to one topic and publish two topics.

☛ The call `rospy.Subscriber('/new_vel', Twist, self.callback_new_velocity)` subscribes to the topic `/new_vel` published by node `random_values`, the variable with the reference to the topic is `self.new_velocity_sub`. Topic messages are of type `Twist` (i.e., linear and angular velocities). The general form for defining topic subscribing is:

```
subscriber_var = rospy.Subscriber('topic_name', message_type, callback_function)
```

☛ The class function `callback_new_velocity()` is set as **callback function**. The mechanism for topic subscribing is in fact as follows: data are published by the topic publisher (`random_values` in our case) according to its internally specified *rate* (random intervals uniformly distributed between 1 and 10 seconds in our case). The `queue_size` parameter of the publisher specifies the amount of message buffering for the topic (which is in practice realized through a network transport protocol). From the subscriber side, consuming/reading topic data is performed in an *asynchronous, signal-based way*: whenever new data is available to the topic, a subscriber node gets signaled and data can be read and processed through a *callback function*. This function is precisely defined in the `Subscriber()` function. The callback function gets as input the new message data, of the type specified in `Subscriber()` (`Twist` in our case).

☛ From the publishing side, with the call `self.vel_pub = rospy.Publisher('/cmd_vel_mux/input/teleop', Twist, queue_size=5)` the node creates a Publisher object, that will publish on the topic

```
  cmd_vel_mux_input_teleop
```

As we will see later on, our **TurtleBot robot** (real or simulated, in Gazebo) is a subscriber to this topic. When it reads data from it, it passes the data to *wheels' controller*, that translates the input velocities into actual wheel velocities (taking care of the feasibility of the command and letting the change from the current to the new desired velocities happening in a possibly smooth way). As the name of the topics suggests, this topic can be used for **teleoperation** of the robot, meaning, in a broad sense, that the controls for the robot come from an external input source. For instance, we will see that we can launch a ROS package that precisely allows to control the robot from the keyboard (or a joypad) using data sent to topic `/cmd_vel_mux/input/teleop`.

When it happens, the node also notifies the change in velocity by publishing it on the `change` topic. Therefore, the node sets a publisher variable with the call `rospy.Publisher('/change', Twist, queue_size=1)`. This will be read by our next node, `pose_monitor`.

☛ The final step in the `__init__` function consists in setting the rate for node activation, that coincides with the publishing rate, and in turn with the rate for sending velocity commands to the robot.

```
self.rate = rospy.Rate(times_per_second)
```

This sets the duration of the sleep in the main `while` loop of the node to `1/times_per_second`. In our case, this means that the node will "spin" with a frequency (in Hz) of 5 cycles/second.

3.5.2 Callback function

```
def callback_new_velocity(self, msg):

    #print "Received Twist msg: ", msg
    rospy.loginfo("Received velocity [linear x]%5.2f  [angular z]%5.2f",
                  msg.linear.x, msg.angular.z)

    # set velocities as received from the /new_Vela topic
    self.vel.linear.x  =  msg.linear.x
    self.vel.angular.z =  msg.angular.z

    # publish the new velocities on the /cmd_vel_mux/input/teleop topic
    self.new_velocity_sub.publish(self.vel)
```

When called (asynchronously), the callback function gets the `Twist msg` data available in the `/new_vel` topic as argument, and it uses it to set the class velocity variable accordingly. Then, it immediately publishes (echoes) it on the `/cmd_vel_mux/input/teleop` topic, which will be read by another node directly connected to the robot motors (and will eventually make the robot move!).

3.5.3 Main function, making the robot move

```
if __name__ == '__main__':
    try:
        controller = MoveRobot()

        # keeping doing until ctrl+c
        while not rospy.is_shutdown():

            # send velocity commands to the robot
            controller.send_velocity_cmd()

            # wait for the selected mseconds and publish velocity again
            controller.rate.sleep()

    except:
        rospy.loginfo("move_robot node terminated")
```

The main function starts with creating an object of class `MoveRobot`, which calls the `init()` function. A potentially infinite activation loop is then performed. At each iteration, the function call `controller.send_velocity_cmd()` sends the velocity values currently stored in the class variable `vel` to the `/cmd_vel_mux/input/teleop` topic.

☛ The call to `controller.rate.sleep()` ensures that the loop is executed with the frequency specified in the `init()` function. In turn, this is the frequency at which velocity commands are

sent to the robot to move it. The higher the frequency, the more continuous the robot motion is. Whenever a velocity change happens, the callback function is invoked and the new velocity values get assigned to `vel` and published on `/cmd_vel_mux/input/teleop` until a new change happens.

3.6 Node `pose_monitor`

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

import rospy

# Import the Odometry message
from nav_msgs.msg import Odometry

# Import the Twist message
from geometry_msgs.msg import Twist

# TF allows to perform transformations between different coordinate frames
import tf

# For getting robot's ground truth from Gazebo
from gazebo_msgs.srv import GetModelState

class PoseMonitor():

    def __init__(self):
        # Initiate a named node
        rospy.init_node('pose_monitor', anonymous=True)

        # Subscribe to topic /odom published by the robot base
        self.odom_sub = rospy.Subscriber('/odom', Odometry, self.callback_odometry)

        # Subscribe to topic /change published by move_robot
        self.vel_change_sub = rospy.Subscriber('/change', Twist,
                                               self.callback_velocity_change)

        self.report_pose = False

        # subscribe to a service server, provided by the gazebo package to get
        # information about the state of the models present in the simulation

        print("Wait for service ....")
        rospy.wait_for_service("gazebo/get_model_state")

        print(" ... Got it!")

        self.get_ground_truth = rospy.ServiceProxy("gazebo/get_model_state",
                                                   GetModelState)

    def callback_velocity_change(self, msg):
        rospy.loginfo("Velocity has changed, now: %5.3f, %5.3f",
                      msg.linear.x, msg.angular.z)
        rospy.sleep(0.75) # to let the velocity being actuated and odometry updated
        self.report_pose = True

    def callback_odometry(self, msg):
        if self.report_pose == True:
            print "Position: (%5.2f, %5.2f, %5.2f)" % (msg.pose.pose.position.x,
                   msg.pose.pose.position.y, msg.pose.pose.position.z)
            self.quaternion_to_euler(msg)
            print "Linear twist: (%5.2f, %5.2f, %5.2f)" % (msg.twist.twist.linear.x,
```

```
                    msg.twist.twist.linear.y, msg.twist.twist.linear.z)
            print "Angular twist: (%5.2f, %5.2f, %5.2f)" % (msg.twist.twist.angular.x,
                    msg.twist.twist.angular.y, msg.twist.twist.angular.z)

            print "Ground Truth: ", self.get_ground_truth("mobile_base", "world")

            self.report_pose = False

    def quaternion_to_euler(self, msg):
        quaternion = (msg.pose.pose.orientation.x, msg.pose.pose.orientation.y,
                    msg.pose.pose.orientation.z, msg.pose.pose.orientation.w)
        (roll, pitch, yaw) = tf.transformations.euler_from_quaternion(quaternion)
        print "Roll: %5.2f  Pitch: %5.2f  Yaw: %5.2f" % (roll, pitch, yaw)


if __name__ == '__main__':
    PoseMonitor()
    rospy.spin()
```

The actions performed by this node are relatively simple and are considered here mainly for the purpose of illustrating a few additional useful ROS concepts and message types. In practice, the node gets notified by `move_robot` (on topic `/change`) of a change in velocity.

☛ When this happens, the node displays the new velocity together with the **pose and velocity** of the robot as estimated from robot's **odometry**, and the robot **ground truth (position and velocity)**, as returned from the Gazebo simulator. Note that odometry returns an estimate (possibly noisy) of the robot's state as computed b ythe robot based on proprioceptive sensing (in this case, based on wheel encoders and IMU data). The *exact* state (i.e., ground truth) can only be returned by a global, precise "sensor". In the case of a simulated robot, this information is actually available through the simulator. Also note that pose and velocity always need to be referred to a coordinate frame, which in this case is the fixed world one (assigned from Gazebo). For the case of a real robot, some global positioning system would be needed to return ground truth information (which would be anyway uncertain, to some degree).

To get access to odometry and ground truth information, the node needs to subscribe respectively to a *topic*, `/odom` (published by the robot base), and to a *service*, `gazebo/get_model_state` (provided by the Gazebo ROS package).

3.6.1 Import statements: Odometry messages, Model states

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

import rospy

# Import the Odometry message
from nav_msgs.msg import Odometry

# Import the Twist message
from geometry_msgs.msg import Twist

# TF is to perform transformations between different coordinate frames
import tf

# For getting robot's ground truth from Gazebo
from gazebo_msgs.srv import GetModelState
```

There are a few new import statements compared to the previous nodes. The statement `from nav_msgs.msg import Odometry` imports `Odometry` messages, that are used to represent all information about odometry. The format of the messages is the following, as obtained from the

`rosmsg` command:

```
$ rosmsg info Odometry

[nav_msgs/Odometry]:
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
```

☛ It can be noted that the message consists of an *header*, that includes the sequence number and the timestamp of the message, plus information about the estimated pose and twist (velocity). For instance, `Odometry/pose/pose/x` gives access to the estimated value of the x coordinate in the robot position in the word reference frame. Both pose and twist data come with a *covariance matrix* specifying the uncertainty associated to the estimated values.

☛ The orientation of the robot is provided using a **quaternion**, a four-dimensional vector that can effectively encode the *roll, pitch*, and *yaw* (RPY) angles that can be used to describe the orientation of a vector in 3D. The `import tf` statement allows to use the `tf` coordinate transformation package. This is a very important package in ROS, but here we only use it to call the function `tf.transformations.euler_from_quaternion(quaternion)`, which is used in the class function `quaternion_to_euler()` precisely to transform the orientation expressed a a quaternion to an orientation expressed as a more intuitive RPY triple of angles.

☛ The last import statement is to use a *service* provide by `Gazebo` to read robot's ground truth: `from gazebo_msgs.srv import GetModelState`. The fact that it's a service message is indicated by the `.srv` extension. In practice, a Gazebo world scenario could be populated by a number of different entities, as in the example shown in Figure 5.

☛ Each world's entity has a *state*, expressed as a pose and a twist (other than a number of additional features). The service that we use will allow us to ask for the state of all or of a particular entity in the world. In our node, we will be (mostly) interested in the state of our TurtleBot.

Figure 5: A gazebo scenario can include a number of different entities, other than robots.

3.6.2 Init, Service server subscription

Most of the statements in the init function should now be self-explanatory. The only new part is related to *subscribing to the service server* for getting the ground truth:

```
        rospy.wait_for_service("gazebo/get_model_state")
```

which subscribes to the service passed as function argument. The service is provide by an entity that must be active in the ROS network at the moment of subscribing. If it is not, the call will be blocking and stay on hold.

Service definitions are a container for the request and response types. ROS Services are defined by `srv` files, which contain a request message and a response message. These are identical to the messages used with ROS topics. The following function call imports the service definition and passes it to a service initialization method, which subscribes to service (messages) `GetModelState` from the service provider `gazebo/get_model_state`:

```
rospy.ServiceProxy("gazebo/get_model_state", GetModelState)
```

3.6.3 Display velocity, odometry, ground truth with callbacks

```
def callback_velocity_change(self, msg):
    rospy.loginfo("Velocity has changed, now: %5.3f, %5.3f",
                  msg.linear.x, msg.angular.z)
    rospy.sleep(0.75) # to let velocity being actuated & odometry updated
    self.report_pose = True

def callback_odometry(self, msg):
    if self.report_pose == True:
        print "Position: (%5.2f, %5.2f, %5.2f)" % (msg.pose.pose.position.x,
               msg.pose.pose.position.y, msg.pose.pose.position.z)
        self.quaternion_to_euler(msg)
        print "Linear twist: (%5.2f, %5.2f, %5.2f)" %
                              (msg.twist.twist.linear.x,
```

```
                            msg.twist.twist.linear.y,
                            msg.twist.twist.linear.z)
        print "Angular twist: (%5.2f, %5.2f, %5.2f)" %
                            (msg.twist.twist.angular.x,
                             msg.twist.twist.angular.y,
                             msg.twist.twist.angular.z)

        print "Ground Truth: ", self.get_ground_truth("mobile_base","world")

        self.report_pose = False
```

The behavior of the node is fully based on the two callback functions, that notify a velocity change (`callback_velocity_change()` when a new message arrive on the `/change` topic, and that at the same time display robot odometry and ground truth.

☛ The only notable part of the code (apart from showing how to print out pose information), is how to get the ground truth from the Gazebo service server:

```
  print "Ground Truth: ", self.get_ground_truth("mobile_base", "world")
```

which shows how the server reference variable is used to access the state service for the entity `mobile_base` (the Kobuki part of the TurtleBot) in reference to the `world` coordinate frame.

3.6.4 Main function

```
if __name__ == '__main__':
    PoseMonitor()
    rospy.spin()
```

The main function is indeed very simple: it creates an object of the class `PoseMonitor()` and keeps it active forever (until `CRTL-C` or another process killing signal). The call to `rospy.spin()` makes the code to run as fast as possible according to the machine scheduler. In practice, being the code all made by callback functions, their activation will be based on the frequency of arrival of the messages on the topics.

3.7 Execute the nodes, start Gazebo + TurtleBot

Now it's finally time to execute the nodes (assuming no errors are there)! We ar not using a launch file, therefore, let's open five different terminal shells. each shell will be used to run a different node and perform further monitoring actions.

Let's first build our package, that now contains a number of source file:

```
  $ roscd; cd ..
  $ catkin_make
```

As pointed out before, be sure that the `roscore` command has already been issued. Each node can be executed with:

```
$ rosrun random_control node_name.py
```

where `random_control` is our package name.

3.7.1 Start Gazebo / TurtleBot, use `rostopic` and teleoperation

☛ Before start running the nodes, let's start **Gazebo**, that will bring up our TurtleBot:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

This command launch nodes in the `turtlebot_gazebo` package as specified by the `turtlebot_world.launch` file. In turn, the *launch* file specifies a `world` model for the scenario, which is the one that appears, containing a TurtleBot plus a few other physical elements (see Figure 6)

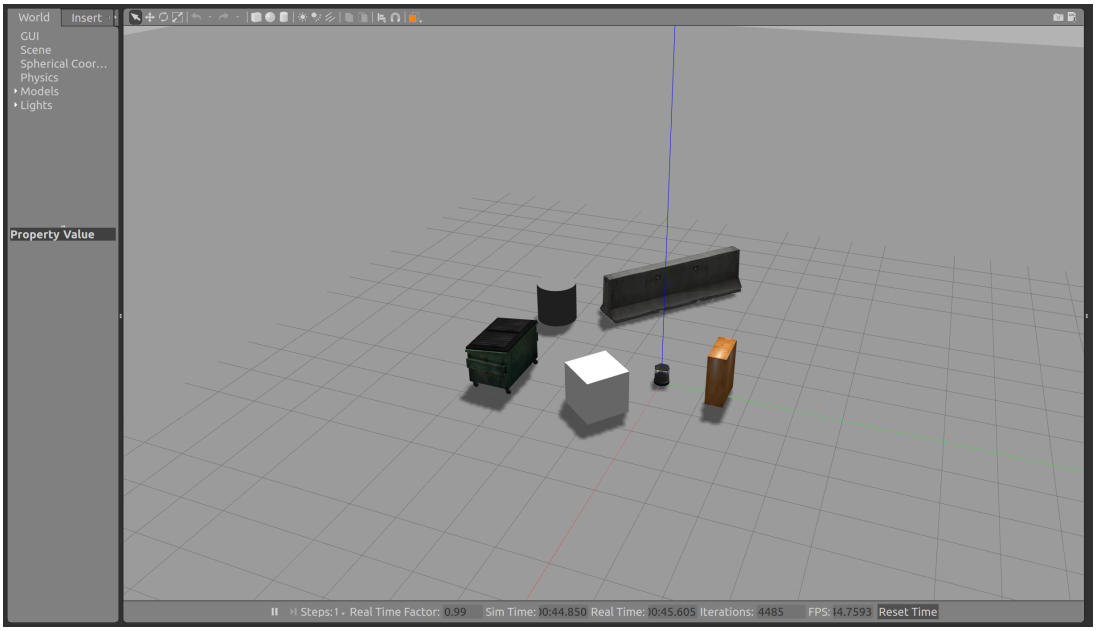

Figure 6: The *playground* world model for our TurtleBot. The robot coordinate frame is indicated by the three orthogonal axes, with the red one indicating the orientation of the robot.

You can click on the elements and get information about them (shown in the left panel). The buttons on the top-left part of the GUI allow to move around the elements. The single arrow button selects an element. The cross-shaped one moves it, the circle-shaped one rotates it around the three possible axis, and the cubic-shaped ones allows to change dimensions (not always possible, however).

☛ Now, let's inspect what the Gazebo package plus the set up world bring to the current ROS system network:

```
$ rostopic list
```

will show all topics being currently published and/or subscribed, that, mainly, are related to the sensors and components of the running TurtleBot. `rostopic` can be used to get a number of useful information about topics:

```
 $ rostopic <command> <topic>

rostopic bw display bandwidth used by topic
rostopic delay display delay of topic from timestamp in header
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 or field type
```

We are going to use the topic `/odom`, therefore we can get information about the topic (who is publishing it, etc.) by using `rostopic`:

```
$ rostopic info /odom
Type: nav_msgs/Odometry

Publishers:
 * /gazebo (http://ros1-Latitude-7480:44219/)

Subscribers: None
```

In turn, in order to know (and access) the format of the message, which is `nav_msgs/Odometry`, we can use the previously seen `rosmsg`:

```
$ rosmsg info Odometry
```

Since the TurtleBot is actually publishing `/odom`, we can read its odometry data from the *command line*:

```
$ rostopic echo /odom
```

and we can add the `-n 1` flag to read only the last one.

We can do the same for all topics being published!

☛ We can also *write* on a topic to which our TurtleBot is subscribing, such as the topic `/cmd_vel_mux/input/teleop` where our node `move_robot` writes its velocity commands:

```
 $ rostopic pub -r 10 /cmd_vel_mux/input/teleop geometry_msgs/Twist
 >     '{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0, y: 0, z: -0.5}}'
```

which for instance will move the robot on a small circle (velocity commands are published 10 times per second, the `-r 10` argument).

☛ We can also move the robot in a more handy way, by *keyboard teleoperation* (or a joypad, if you have it):

```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

that provides a way to control the robot using keyboard keys as follows:

```
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
```

Try it and you should see your TurtleBot moving around! At the same time, check the `/odom` topic, and see how it changes.

3.7.2 Start `random_value` node

```
$ rosrun random_control random_values.py
```

which should show the process of generating the random velocities. Now `$ rostopic list` will show also the topic `/new_vel`. Again, see the data published to the topic via the `rostopic` command.

3.7.3 Start `move_robot` node

```
$ rosrun random_control move_robot.py
```

You should see the robot start moving according to the velocity commands provided by the `random_value` node (obstacles permitting ...). It should look quite random :-)

☛ Now, since our robot controller is just a random one, maybe it's better to run an obstacle-free world:

```
$ export TURTLEBOT_GAZEBO_WORLD_FILE=/usr/share/gazebo-7/worlds/empty.world;
  >  roslaunch turtlebot_gazebo turtlebot_world.launch
```

which will use the `empty.world` from the folder where all Gazebo models are included (give them a look!). The `export` is needed since in the launch file the exported environment variable is looked for defining the world model.

3.7.4 Start `pose_monitor` node, use `rossrv`

```
$ rosrun random_control pose_monitor.py
```

Finally, the last node can be run. Of course, the order for running the nodes is *not* really important. This node will show how odometry and ground truth evolve over time.

☛ Since the node is using a service, we might ask what are the services actually available in our ROS network:

```
$ rossrv list
```

that will give a quite long list of service servers! How do we know the message type for service request and answer associated to a service? Again, using:

```
$  rosmsg info <srv message>
```

4 On-board sensors: depth camera, laser scan, bumpers, cliff sensors

In the previous package, a random robot controller has been developed. The controller did not use input data from sensors to assign velocity values. In a sense, the robot was operating

blindly, or, in other words, in open-loop modality. However, sensory inputs are fundamental to cope with the challenges of real-world scenarios. To support safe and effective navigation, the TurtleBot2 is equipped with an RGB-D *depth camera*, an *array of bumpers*, and an *array of cliff sensors*, which are described in the two sections that follow.

4.1 Depth camera (a first introduction)

The depth camera returns a *point cloud* that can be used to assert the presence or not of objects within the 3D field of view of the camera.

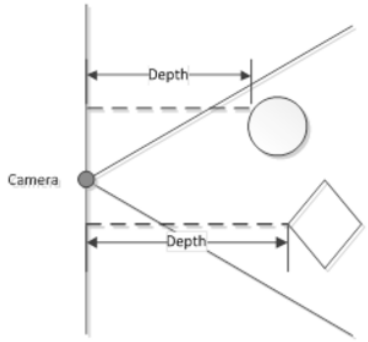

Figure 7: Depth of an object within the FOV of the camera. Each pixel gets a depth value.

In our real TurtleBot2, the RGB-D camera is an ORBBEC Astra, that has a declared FOV of 60° horizontal and 49.5° vertical, and outputs a depth image of 640 × 480 pixels updated at 30 Hz. The operational range is between 0.6 to 8 meters. The Microsoft Kinect, the sort of original RGB-D camera, has a FOV of 57° horizontal and 43° vertical, with an operational range of 0.9 to 3.5 meters (and same depth image size). The model which is include as a default RGB-D camera for the TurtleBot2 since ROS Indigo, is the *Asus Xtion Pro Live 3D Sensor*, shown in the figure. The working of sensor is based on an infrared emitter, an infrared camera, and an RGB camera. The FOV is 58° H, 45° V, 70° D (Horizontal, Vertical, Diagonal). The distance of use is between 0.8 and 3.5 meters.

Figure 8: The Asus Xtion Pro Live sensor which is the default mounted on the TurtleBot2 in ROS and Gazebo models.

For the time being, we will not use the depth camera, but rather we will use a 2D slice of its data, that would be equivalent of the output returned from a *planar laser scanner (range finder)*. A virtual sensor exists that precisely returns this information, which would tell about the presence of obstacles on a plane at the height of the depth camera sensor. This is performed by the `depthimage_to_laserscan` package.

☛ However, before moving to check what kind of messages the laser scan sensor provides, it is instructive to check what the camera "sees". At this aim, first starts Gazebo (if you don't have yet a real robot), for instance with the maze world from the homework:

```
$ export TURTLEBOT_GAZEBO_WORLD_FILE=~/catkin_ws/worlds/funky-maze.world;
> roslaunch turtlebot_gazebo turtlebot_world.launch
```

then, in two different shell, execute the following rosrun commands, that will run the `image_view` package, which is a viewer for ROS image topics (`http://wiki.ros.org/image_view`):

```
$ rosrun image_view image_view image:=/camera/rgb/image_raw
$ rosrun image_view image_view image:=/camera/depth/image_raw
```

where the first command shows what the robot sees with the RGB camera, while the second shows the depth images, where the grayscale encodes depth. In gazebo, you can manually move the robot and/or add objects to the scene, in front of the robots, and you will see how the images do change. If you have a real robot, you can keep track of what the robot is seeing in the real world. In the figure, the maze world is shown as from `Gazebo`, together with the RGB and depth images from `image_view`. We can also use `rviz` for an enhanced visualization:

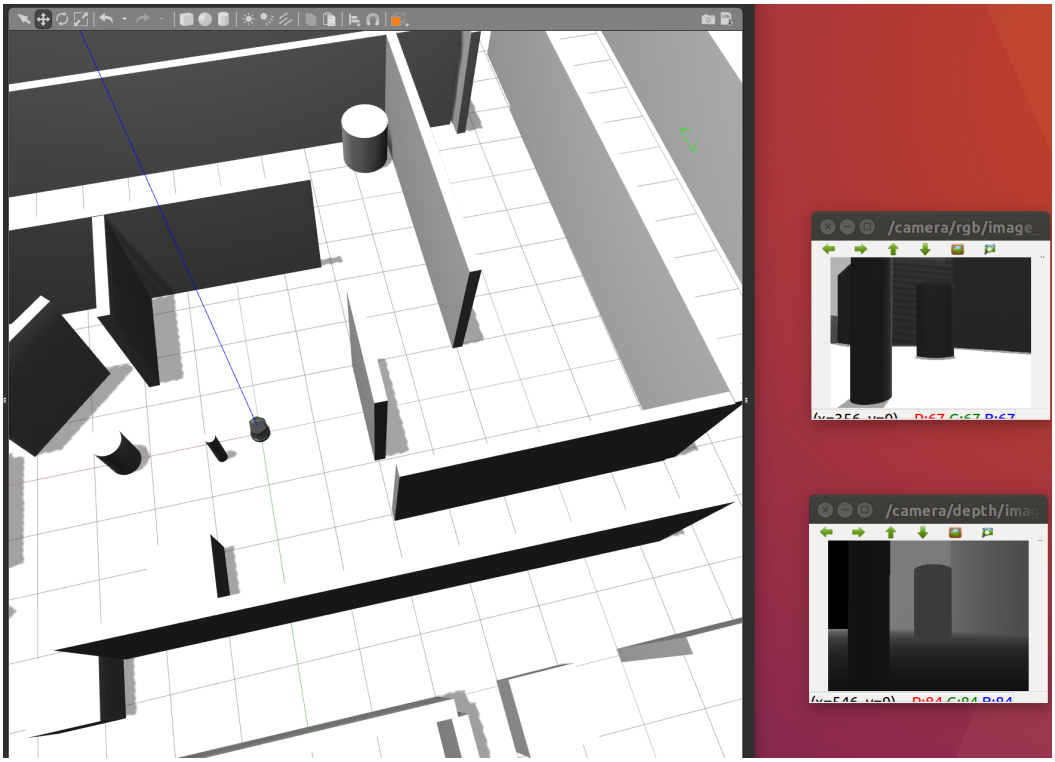

Figure 9: The maze world in Gazebo, and the RGB and depth images as seen from the robot camera.

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```

check the `PointCloud` button and you will see something similar to what shown in the figure below (Left). If you uncheck `PointCloud` and check `LaserScan`, you will visualize the output from the laser scan sensor, that will show the obstacle in range of the FOV of the robot, as shown in the figure (Right).

4.2 Laser scan sensor

Going back to the 2D laser scan sensor, the output of the scan readings are published on the topic `/scan`:

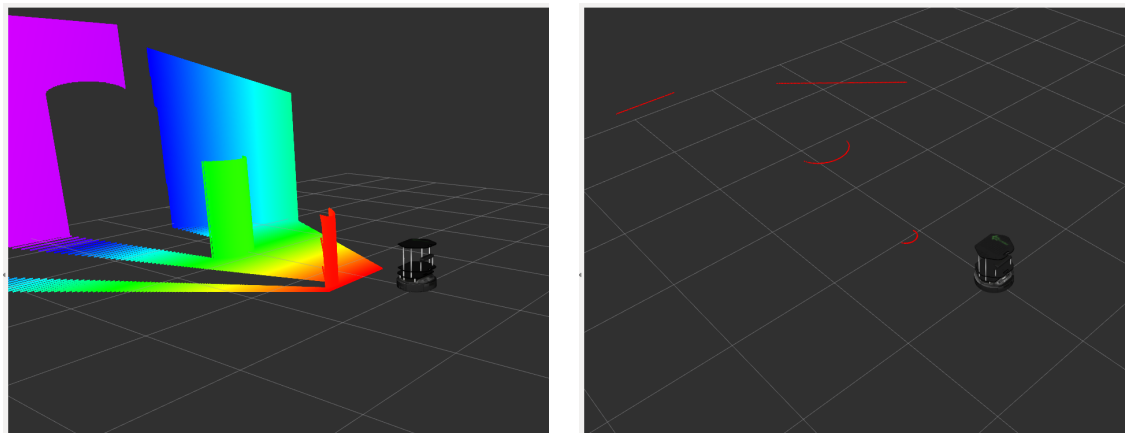

Figure 10: Visualization of the point cloud (Left) and of the laser scan (Right) in `rviz` for the same scenario of the previous figure.

```
$ rostopic info /scan
```

that makes use of `sensor_msgs/LaserScan` as type of topic messages:

```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

with the following meaning:

```
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]
                         # (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.
```

☛ In order to subscribe and read the messages from the `/scan` topic, the above message type must be imported in the python code, with a statement:

```
from sensor_msgs.msg import Laserscan
```

The run-time value of the fields of the range finder messages (e.g., the *ranges* of the objects in the FOV) can be read as usual from the command line:

```
$ rostopic echo /scan/ranges
```

The sensor has a number of parameters (`http://wiki.ros.org/pointcloud_to_laserscan`):

```
-min_height (double, default: 0.0)          The minimum height to sample in the point cloud in meters.
-max_height (double, default: 1.0)          The maximum height to sample in the point cloud in meters.
-angle_min (double, default: -90)           The minimum scan angle in radians.
-angle_max (double, default: 90)            The maximum scan angle in radians.
-angle_increment (double, default: pi/360)  Resolution of laser scan in radians per ray.
-scan_time (double, default: 1.0/30.0)      The scan rate in seconds.
-range_min (double, default: 0.45)          The minimum ranges to return in meters.
-range_max (double, default: 4.0)           The maximum ranges to return in meters.
-target_frame (str, default: none)          If provided, transform the pointcloud into this frame before
                                              converting to a laser scan. Otherwise, laser scan will be
                                              generated in the same frame as the input point cloud.
-concurrency_level (int, default: 1)        Number of threads to use for processing pointclouds. If 0,
                                              automatically detect number of cores and use the equivalent
                                              number of threads. Input queue size is tied to this parameter.
-use_inf   (boolean, default: true)         If disabled, report infinite range (no obstacle) as range_max -+ 1.
                                              Otherwise report infinite range  as +inf.
```

☛ Parameter values can be retrieved or can be set from command line or from inside a program using the `rosparam` service:

```
$ rosparam list
$ rosparam get /depthimage_to_laserscan/range_min
```

☛ In addition, the package `/depthimage_to_laserscan` publishes the topic `/depthimage_to_laserscan/parameter_descriptions` that precisely describes the parameters and their values in use:

```
$ rostopic echo /depthimage_to_laserscan/parameter_descriptions
```

Many complex packages include such a `parameter_descriptions` topic precisely to ease the access to relevant parameters. A `parameters_updates` topic is available for *dynamic updates*. As a general rule, it is necessary to check what are the specific parameters of a sensor before using it!

In particular, for the laser scan sensor, the `ranges[]` arrays returns a depth value for the presence (or not) of an object along n radial directions, where n depends on the max and min scan angles (`angle_min` and `angle_max`), and on the angle increment parameter, `angle_increment`. If an object is detected at a distance d_i along the i-th radial direction, the distance value is reported in `ranges[i]`. If nothing is detected (up to the maximum range), a `NaN` is reported (or any arbitrary value over the declared ranges). `ranges[1]` corresponds to the rightmost scan direction with respect to the heading of the robot, while `ranges[n]` corresponds to the leftmost one. `ranges[n/2]` corresponds to the measure along the heading of the robot. Since the size of the depth image is 640×480 pixels, the number n is equal to 640 (an horizontal slice). The difference between max and min ranges divided by n must be then equal to the angular resolution.

In general, the bearing of the particular range estimate in position `i` of `ranges[]` can be obtained as follows (assuming that the message retrieved from the topic channel is named `msg`):

```
bearing = msg.angle_min + i * msg.angle_max / len(msg.ranges)
```

while the distance of the closest obstacle detected by the laser scanner is:

```
range_closest = min(msg.ranges)
```

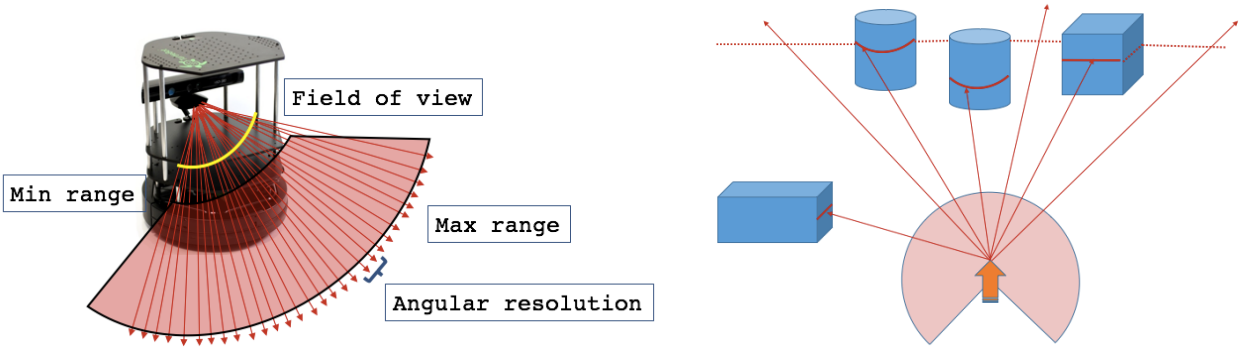

Figure 11: Illustration of laser scan operations.

4.3 Bumpers

In the TurtleBot2 there are *three bumper sensors* in the base of robot (which is the *Kobuki* robot):

- one in the middle-front;
- one in the left front;
- one the right front.

☛ The bumper sensor is a mechanical sensor that helps the robot *detecting collision with obstacles*. When the robot touches an external object, at least one bumper sensor (if in correct position) gets pressed and throws a *bumper event*. A bumper event is also generated when the bumper is released.

Bumper messages following a bumper event are published in the topic:

```
/mobile_base/events/bumper
```

As usual, if we check the info of the topic with:

```
$ rostopic info /mobile_base/events/bumper
```

we find that topic messages are of type:

```
kobuki_msmgs/BumperEvent
```

which means that a statement

```
from kobuki_msmgs.msg import BumperEvent
```

will need to be part of the initial part of a python code using bumpers.

To check what is the structure of the `BumperEvent` messages:

```
$ rosmsg info kobuki_msmgs/BumperEvent
```

which shows the following:

```
# bumper
uint8 LEFT   = 0
uint8 CENTER = 1
uint8 RIGHT  = 2

# state
uint8 RELEASED = 0
uint8 PRESSED  = 1

uint8 bumper
uint8 state
```

The `state` field says what actions has triggered the event, while the `bumper` field says to which bumper the event refers to.

The state of the bumpers can be controlled at any time from the command line by echoing the topic:

```
$ rostopic echo /mobile_base/events/bumper
```

For instance, if in Gazebo you manually move the robot in order to let it touch an obstacle, the above topic should report it.

4.4 Cliff sensors

Cliff sensors are responsible for *detecting cliffs and altitude changes* when the robot is moving, especially to prevent crashes when reaching stairs. They are located in the bottom of the Kobuki base. The behavior is very similar to the bumper sensor. Likely, we are not going to use cliff sensors, however, they are described below for completeness.

There are *three cliff sensors*:

- one in the center,
- one in the left side,
- one in right side.

Cliff events are published in the topic:

```
/mobile_base/events/cliff
```

Again, getting the info from `rostopic` and the type of message from `rosmsg` we get that type is:

```
kobuki_msmgs/CliffEvent
```

which means that the following statement needs to be part of the initial part of a python code using cliff sensors:

```
from kobuki_msmgs.msg import CliffEvent
```

To check the structure of the `CliffEvent` messages:

```
$ rosmsg info kobuki_msmgs/CliffEvent
```

which shows the following:

```
# cliff sensor
uint8 LEFT   = 0
uint8 CENTER = 1
uint8 RIGHT  = 2

# cliff sensor state
uint8 FLOOR = 0
uint8 CLIFF = 1

uint8 sensor
uint8 state

# distance to floor when cliff was detected
uint16 bottom
```

5 Setting the robot pose for simulation experiments

When running a simulation experiment, it might be appropriate to set the initial pose of the robot at a specific location of the environment and a specific orientation. This can be realized in different ways.

5.1 Using `world` files

First, the pose can be flexibly specified in the `world` file which is given as input to a `roslaunch` command. We will study later how to create or modify a world file, therefore, let's skip this for the time being.

5.2 Using environment variables

Another way is by changing a world file, more precisely by *exporting the environment variable holding the robot pose* when calling gazebo. For instance:

```
$ export ROBOT_INITIAL_POSE="-y 2 -x 3 -z 0 -R 0 -P 0 -Y 2";
> roslaunch turtlebot_gazebo turtlebot_world.launch
```

where the pose is set in terms of position variables (`x,y,z`) and orientation ones (`Roll Pitch, and Yaw`). Any subset of the pose variables can be set, while the remaining ones will be taking their default values. In a similar way, also the initial `Twist` (linear and angular velocity vectors) can be set.

In Gazebo, after selecting the robot, in the left panel it is possible to check in `[Property - Value]` and control the pose of the robot or of other objects at any time.

5.3 Manually

☛ Of course, an easy, yet not automatic way of changing the pose is *manually*, by moving the robot inside Gazebo (first select, then move). The robot can be manually moved at any time.

5.4 Using a `service` to set robot model state

Inside a simulation it is possible to set the robot pose by using a *service*, analogous to what was done for reading the ground truth. In this way the service it allows to set, rather than returning, a value. How to use a service for the purpose is shown in the code below.

One issue with this way of proceeding consists in the so-called problem of the *kidnapping robot*: in pratice the robot is being teleported, which means that its *state* knowledge is not valid anymore (e.g., its odometry information would be totally wrong after the teleportation act).

The code below is the same as in `pose-monitor.py`, except for a modification in the `__init__` function and in the `import` statements (lines 15–16, 39–54). The code for the functions other than the `__init__` one are not reported being the same. The node gets connected to the service for setting the *state of an entity* in the Gazebo world and then sets the pose of the `mobile_base` entity, that physically moves the TurtleBot2, by assigning `position.x=2` (all the other parameters stay from default). If Gazebo is launched first as usual, and then `rosrun` this node, the robot will be "teleported" in the new coordinate position (from the default one at `0,0,0`).

```
1  #!/usr/bin/env python
2  import rospy
3
4  # Import the Odometry message
5  from nav_msgs.msg import Odometry
6
7  # Import the Twist message
8  from geometry_msgs.msg import Twist
9
10 import tf
11
12 # for the ground truth
13 from gazebo_msgs.srv import GetModelState
14 from gazebo_msgs.srv import SetModelState
15 from gazebo_msgs.msg import ModelState
16
17 class PoseMonitor():
18
19     def __init__(self):
20         # Initiate a named node
21         rospy.init_node('pose_monitor', anonymous=True)
22
23         self.odom_sub = rospy.Subscriber('/odom', Odometry, self.callback_odometry)
24
25         self.vel_change_sub = rospy.Subscriber('/change', Twist,
26                                                self.callback_velocity_change)
27         self.rate = rospy.Rate(1)
28
29         self.report_pose = False
30
31         print("Wait for GET service ....")
32         rospy.wait_for_service("gazebo/get_model_state")
33
34         print(" ... Got it!")
35
36         self.get_ground_truth = rospy.ServiceProxy("gazebo/get_model_state",
37                                                    GetModelState)
38         print("Wait for SET service ....")
39         rospy.wait_for_service("gazebo/set_model_state")
40         #rospy.wait_for_service("gazebo/spawn_model")
41
42         print(" ... Got it!")
43
```

```
44          self.model_state = ModelState()
45
46          self.model_state.model_name = "mobile_base"
47          self.model_state.pose.position.x = 3
48
49          self.set_model_state = rospy.ServiceProxy("gazebo/set_model_state",
50                                                    SetModelState)
51          self.set_model_state(self.model_state)
52
53      .... (the same as in pose-monitor.py)
```

6 Dynamic creation/modification of a robot scenario from a ROS node

In the previous section, a `service` was used to set the state of the robot model. In simulation, this way of proceeding can be generalized to set the state of any entity, and in particular, to dynamically add entities to a scenario from a ROS node (as an alternative of statically create a scenario using a world file).

An example of how to proceed is shown in the following example, where a coke can is added to the scene. The coke can model is available from Gazebo's model library. Adding any other model would consist of the same steps, referencing to different model files. After `__init__()`, that performs steps similar to those described in the previous section, the function `spawn_marker_at_location(self, x,y,z)` can be called to add the coke can model at the specified location, if the coke is not already in the scene (in this case, it's only the position that gets reassigned).

```
1  from gazebo_msgs.srv import SpawnModel
2  from gazebo_msgs.srv import GetModelState
3  from gazebo_msgs.srv import SetModelState
4  from gazebo_msgs.msg import ModelState
5
6  class MyClass():
7
8      def __init__(self):
9
10         ...
11
12         # check service names with rosservice list
13         print("Wait for model spawn service ....")
14         rospy.wait_for_service("gazebo/spawn_sdf_model")
15         print(" ... SPAWN Service is up!")
16
17         self.spawn_marker = rospy.ServiceProxy("gazebo/spawn_sdf_model", SpawnModel)
18
19         print("Wait for get service (to check whether the marker model is  already there) ....")
20         rospy.wait_for_service("gazebo/get_model_state")
21         print(" ... GET Service is up!")
22
23         self.marker_model_name = "Coke_can"
24
25         self.get_marker_state = rospy.ServiceProxy("gazebo/get_model_state", GetModelState)
26         if((self.get_marker_state(self.marker_model_name, "world")).success == False):
27             print "Marker %s is not in the scene" % (self.marker_model_name)
28             self.marker_present = False
29         else:
30             self.marker_present = True
```

```
31              print "Marker %s is already in the scene .. let's move it!" % (self.marker_model_name)
32              print("Wait for set service to move the marker model ..")
33
34              rospy.wait_for_service("gazebo/set_model_state")
35              print(" ... SET Service is up!")
36
37              self.set_marker_state = rospy.ServiceProxy("gazebo/set_model_state",
38                                                         SetModelState)
39              self.marker_state = ModelState()
40
41              self.marker_state.model_name = self.marker_model_name
42
43
44      def spawn_marker_at_location(self, x,y,z):
45
46          if self.marker_present == True:
47              self.marker_state.pose.position.x = float(x)
48              self.marker_state.pose.position.y = float(y)
49              self.marker_state.pose.position.z = float(z)
50
51              self.marker_state.pose.orientation = Quaternion(0,0,0,1)
52
53              self.set_marker_state(self.marker_state)
54
55          else:
56              print "Pose: ", x, y, z
57              marker_pose   =   Pose(Point(float(x), float(y), float(z)), Quaternion(0,0,0,1))
58              model_file = open('/home/ros-1/.gazebo/models/coke_can/model.sdf','r')
59              model_sdf  = model_file.read()
60
61              self.spawn_marker("Coke_can", model_sdf, "robots_name_space", marker_pose, "world")
```

7 YAML markup language on the command line

Several ROS tools (`rostopic, rosservice`) use the *YAML markup language* on the command line (and could also be used to make file scripts). YAML was chosen as, in most cases, it offers a very simple, nearly markup-less solution to typing in typed parameters.

An important use of YAML is for passing values when publishing ROS Message using the command line. ROS Messages can be represented either as a YAML list or dictionary. If represented as a list, the arguments are filled in-order and must align 1-to-1 with the fields of the message. If represented as a dictionary, the dictionary keys are assumed to map onto fields of the same name in the message. Unassigned fields are given default values. Please refere to the wiki ROS pages (`http://wiki.ros.org/ROS/YAMLCommandLine` for an extensive description of the different options when using YAML in ROS.

☛ Be aware that the are a few different, yet equivalent ways to publish messages by using YAML in the command line. The following example illustrates the different possibilities.

Let's consider messages of type `geometry_msg/PointStamped`, that represent a point in the space and include also an header. The message data structure, as shown below, is composed of two parts:

```
Header header
  uint32 seq
  time stamp
  string frame_id
```

```
Point point
  float64 x
  float64 y
  float64 z
```

These are all equivalent ways of publishing the same data on the `pt` topic using YAML syntax:

```
$rostopic pub pt geometry_msgs/PointStamped '[0, now, base_link]' '[1.0, 2.0, 3.0]'
$rostopic pub pt geometry_msgs/PointStamped '{header: {stamp: now, frame_id: base_link}, point: [1.0, 2.0, 3.0]}'
$rostopic pub pt geometry_msgs/PointStamped '{header: {stamp: now, frame_id: base_link}, point: {x: 1.0, y: 2.0, z: 3.0}}'
```