

Programming ROS with Python

Inteligencia Artificial en los Sistemas de Control Autónomo

Objectives

- Introduce the `catkin` build system
- Implement ROS nodes with Python

Bibliography

Rospy package tutorials ([Link](#))

Table of Contents

- Overview
- Catkin workspaces
 - Spaces
 - Workspace creation
- Catkin packages
 - Package creation
 - Package build
 - Exercise
- Nodes programming with Python
 - Topics
 - Messages
 - Services

Overview

Two native languages: C++ and Python

- C++ good for high performance
- Python good for prototyping

ROS has its own build system, named **catkin**

- Similar to **make**
- Built on CMake
- Make uses Makefile, catkin uses CMakeLists.txt
(Note: **roscpp** is deprecated)

As almost everything in ROS, catkin is a package

- Documented in <http://wiki.ros.org/catkin>
- (Suggested read)

Catkin workspaces

Spaces (I)

Catkin requires a **workspace**

- Folder where install, modify and build packages
- Located in `~/catkin_ws` in our VM

Three folders (**spaces**, in ROS' terminology) in a workspace

`src` Packages source code

`build` Intermediate build files

`devel` Intermediate installation files. Environment scripts

Catkin workspaces

Spaces (II)

Typical catkin workspace

```
workspace_folder/      -- WORKSPACE
  src/                 -- SOURCE SPACE
    CMakeLists.txt     -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      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
    ...
```

Catkin workspaces

Workspace creation

Creation of a new catkin workspace

1. Create folder: `mkdir -p ~/catkin_ws/src`
2. Change working directory: `cd ~/catkin_ws/src`
3. Initialize WS: `catkin_init_workspace`

(Already done in the VM)

Catkin packages

Package creation

Simplest package

```
myPackage/  
  CMakeLists.txt  
  package.xml
```

Creation of a new package (from the workspace)

1. `catkin_create_pkg <package_name> [depend1] [depend2] [depend3]`

Example: `catkin_create_pkg myPackage std_msgs rospy`

2. Customize `package.xml`

Warning: Package still not available

Catkin packages

Package build

To build the workspace: `catkin_make` targets

- From the folder `src`
- Uses a `CMakeLists.txt` file
- By default builds all the packages

To execute the node

1. Make it accesible: `source devel/setup.bash`
2. Execute it: `roslaunch package node`

Warning: package is given in `packages.xml`

To install a package: `catkin_make install`

Catkin packages

Exercise (I)

Implement a “Hello, world” node

- Create a `exercises` package dependent on `rospy`
- Customize `package.xml`
- Create a folder named `scripts`
- Create a file named `hello.py`
- Give execution permissions to `hello.py`
- Edit the file (next slide)
- Build the project
- Run `source devel/setup.bash`
- Execute `roscore`
- In other tab, execute the node (`hello.py`)

Catkin packages

Exercise (II)

scripts/hello.py

```
#!/usr/bin/python

import rospy

rospy.init_node("hello")

while not rospy.is_shutdown():
    print "Hello, world"
```

Extra points: Run `hello.py` with a launch file

Nodes programming with Python

Topics (I)

- Code stored in folder `scripts`
- Scripts must have execution permissions
- Must import `rospy` Python module
- Must init the node: `rospy.init_node('name')`
- Convenient assets
 - `Rate` class and `sleep()` method
 - `rospy.is_shutdown()`

Nodes programming with Python

Topics (II)

scripts/talker.py

```
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world"
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Nodes programming with Python

Topics (III)

scripts/listener.py

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() keeps python from exiting
    rospy.spin()

if __name__ == '__main__':
    listener()
```

std_msgs/msg/String.msg

string data

Nodes programming with Python

Topics (IV): Exercise

Run the example

1. Make them accesible: `source devel/setup.bash`
2. Initialize ROS: `roscore`
3. Execute the nodes:
 - `roslaunch exercises talker.py`
 - `roslaunch exercises listener.py`

Nodes programming with Python

Topics (V)

ROS only provides a callback to read topics

- Reading just the last message is not out-of-the-box

A common practice is to have a listener in background

- Updates a global variable with the message

Example

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

from nav_msgs.msg import Odometry

def callbackOdometry(msg):
    print msg.pose.pose

if __name__ == "__main__":
    rospy.init_node('oodometry', anonymous=True)
    rospy.Subscriber('odom', Odometry, callbackOdometry)
    rospy.spin()
```


Nodes programming with Python

Topics (VI): Exercise

Excercise:

1. Modify listener in slide 14 to store the last message
2. Show the message five times per second
 - Hint: Use `rospy.Rate()` and `rospy.spin()`

Nodes programming with Python

Messages (I)

Same structure in Python than in the msg file

geometry_msgs/Twist.msg

```
Vector3  linear
Vector3  angular
```

geometry_msgs/Vector3.msg

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

Message usage

```
from geometry_msgs.msg import Twist
from geometry_msgs.msg import
    Vector3

vector = Vector3()
vector.x = 0
vector.y = 0
vector.z = 1
twist = Twist()
twist.linear = vector
twist.angular.x = 0
twist.angular.y = 0
twist.angular.z = 0
otroVector = Vector3(1, 0, 0)
```

Nodes programming with Python

Messages (II)

TYPE	KEYWORD
Integer	int8, int16, int32, int64 (plus uint*)
Float	float32, float64
String	string
Time	time, duration
Variable-length array	array[] (example: float32[])
Fixed-length array	array[C] (example: float32[5])
Struct	other msg files

Custom messages need wrappers classes

- Automatically generated by catkin
- Requires configure dependencies (i.e. set up `packages.xml` and `CMakeLists.txt`)

Nodes programming with Python

Messages (III)

Implement the following tasks:

1. Execute `roscore`
2. Execute `roslaunch turtlesim turtlesim_node`
3. Implement a node that moves the turtle forward with constant velocity
4. Implement a node that shows the turtle pose

Execute `roslaunch stdr_launchers`

`server_with_map_and_gui_plus_robot.launch` and write a node that

1. Shows the odometry as it appears
2. Shows sonar measures as they appear
3. Stores the last odometry and sensor measures

Nodes programming with Python

Services: Setting up the build-system (I)

Automatic generation of proxies (proxy = interface)

- Python and C++
- Similar messages and services
- Stored in \$(WS)/devel/lib/python2.7/dist-packages

Modify `packages.xml` and `CMakeLists.txt` to inform catkin

- (More info) (More)

Service creation process:

1. Create the `srv` file in folder `srv`
2. Enable code generation by editing `packages.xml`

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

Nodes programming with Python

Services: Setting up the build-system (II)

3. Add dependency to `CMakeLists.txt`, uncommenting

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

4. Add service file

```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

Nodes programming with Python

Services: Service provider (I)

Two components

- Provider and consumer (or server and client)
- Both uses proxies (like local function calls)
- Both implemented in nodes

Service provider

- Method `rospy.Service()`
- Request as `fooRequest`
- Response as `fooResponse`

AddTwoInts.srv

```
int64 a
int64 b
---
int64 sum
```

Nodes programming with Python

Services: Service provider (II)

scripts/add_two_ints_server.py

```
from beginner_tutorials.srv import *
import rospy

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

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

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


Nodes programming with Python. Services: Service consumer (I)

Two methods

- Wait until service available: `rospy.wait_for_service()`
- Get proxy: `rospy.ServiceProxy()`

Exception: `rospy.ServiceException`

Nodes programming with Python. Services: Service consumer (II)

scripts/add_two_ints_client.py.py

```
import sys
import rospy
from beginner_tutorials.srv import *

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

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

Nodes programming with Python

Services: Exercises

Run the previous examples

1. Make them accesible: `source dev/setup.bash`
2. Initialize ROS: `roscore`
3. Execute the nodes:
 - `roslaunch myPackage add_two_ints_client.py`
 - `roslaunch myPackage add_two_ints_client.py 5 6`

Execute STDR with a robot

1. Invoke programmatically a service to move the robot to coordinates (15, 15)

Exercises

Implement the following tasks in ROS

1. Launch STDR with any robot
2. Move the robot four distance units to the east

Hint: Use odometry

3. Move the robot four distance units to the east and then one to the north
4. Move the robot to the opposite side of the map