Institute for Cognitive Systems
Technische Universität München
Dr.-Ing. J. Rogelio Guadarrama Olvera

Modeling and control of legged robots
Tutorial 0: Ament, Cmake, ROS2 and Python

This tutorial, and the lecture in general requires basic knowledge of ROS 2 and python. It is recommended to do the basic and intermediate ROS 2 tutorials before.

`https://docs.ros.org/en/rolling/Tutorials.html`

1 Introduction

CMake is a cross-platform free and open-source software tool designed for build automation, testing, packaging, and installation of software using a compiler-independent method. It's not a build system itself but generates files for various build systems. The name "CMake" does not stand for anything specific. it's simply the name of the tool. The development of CMake began in response to the need for a cross-platform build environment. You can access CMake website by clicking the logo below.

2 CMakeLists.txt

The `"CMakeLists.txt"` is a configuration file used with CMake. As the above explanation, CMake is a tool that helps manage the build process of software projects. The `"CMakeLists.txt"` file contains instructions and configuration settings for CMake to generate build files (such as Makefiles or project files for IDEs) based on the specifications provided.

In a CMake project, developers define the project structure, dependencies, compiler options, and other build-related information in the `"CMakeLists.txt"` file. This allows for a consistent and platform-independent way to describe how the project should be built. CMake can then generate the necessary build files for various operating systems and build systems, simplifying the process of building and compiling the software on different platforms.

Typically, a `"CMakeLists.txt"` file includes directives to specify source files, target executables or libraries, compiler flags, dependencies, and other build-related configurations. The content of the file is written in CMake scripting language.

3 ROS2

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications. From drivers and state-of-the-art algorithms to powerful developer tools, ROS has the open source tools you need for your next robotics project. The original ROS project will no longer be supported from May 31st 2025. It is replaced by the ROS2 project which improves the performance and functionality of ROS1. However, the compilation and package managing is different.

A ROS2 workspace is a folder that contains at least one ROS2 package that implement the functionality to control a robot. ROS2 packages can be implemented in C++ or Python, requiring different creation processes. The tool used to create the packages is called Ament which provides the `ament_cmake` and `ament_python` tools.

4 `ament_cmake`

`ament_cmake` is the build system for CMake based packages in ROS 2 (in particular, it will be used for most if not all C/C++ projects). It is a set of scripts enhancing CMake and adding convenience functionality for package authors.

To create an ament package for ROS2, you can use the following command.

```
$ ros2 pkg create --build-type ament_cmake my_package --dependencies
    other_package_1 other_package_2
```

It will create a folder with a structure and configuration files including the `"CMakeLists.txt"`. The `"CMakeLists.txt"` plays a crucial role in the Robot Operating System 2 (ROS2) development environment. While CMake is a general-purpose build system, ROS uses a custom build system called `"colcon"`, which is built on top of CMake. This means most ROS2 packages have a `"CMakeLists.txt"` file to configure their build process.

a Details of `"CMakeLists.txt"`

The most important parts of the `"CMakeLists.txt"` file for a package that includes C++ and Python nodes are summarized below.

```
cmake_minimum_required(VERSION 3.8)
project(my_package)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(other_package_1 REQUIRED)
find_package(other_package_2 REQUIRED)

# add executable nodes
add_executable(node src/node.cpp)
ament_target_dependencies(node other_package_1_lib other_package_2_lib)

install(TARGETS node
DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
find_package(ament_lint_auto REQUIRED)
# the following line skips the linter which checks for copyrights
# comment the line when a copyright and license is added to all source files
set(ament_cmake_copyright_FOUND TRUE)
# the following line skips cpplint (only works in a git repo)
# comment the line when this package is in a git repo and when
# a copyright and license is added to all source files
set(ament_cmake_cpplint_FOUND TRUE)
ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Let us break apart the main parts of the `"CMakeLists.txt"` file.

1. `cmake_minimum_required(VERSION 3.8)`

   Minimum version for CMake to use in ROS2 Humble

2. `project(my_package)`

   Current project's name (my_package)

3.
```
find_package(ament_cmake REQUIRED)
find_package(other_package_1 REQUIRED)
find_package(other_package_2 REQUIRED)
```

Find a package from the environment. You can put "ROS2 packages" as components above. You must list here all the required dependencies to compile and run your package.

4.
```
add_executable(node src/node.cpp)
ament_target_dependencies(node other_package_1_lib other_package_2_lib)
```

Create an executable node from a cpp file. These command will tell ament the name of the target (executable app) and the libraries it depends from the dependency packages.

5.
```
install(TARGETS node
DESTINATION lib/${PROJECT_NAME})
```

This part will tell the compiler where to deploy the executable files once they are created, i.e. where the libraries and applications will be installed.

6.
```
if(BUILD_TESTING)
find_package(ament_lint_auto REQUIRED)
# the following line skips the linter which checks for copyrights
# comment the line when a copyright and license is added to all source
    files
set(ament_cmake_copyright_FOUND TRUE)
# the following line skips cpplint (only works in a git repo)
# comment the line when this package is in a git repo and when
# a copyright and license is added to all source files
set(ament_cmake_cpplint_FOUND TRUE)
ament_lint_auto_find_test_dependencies()
endif()
```

This part will tell colcon wether to build the unit tests of the contents of the package.

7. To build a ROS2 workspace use the following command.

```
$ colcon build --symlink-install
```

8. To load the contents of the workspace in the current environment, use the following command.

```
$ source install/setup.bash
```

9. After building this package, you can find your cpp node below.

```
$ ros2 run my_package node
```

5 ament_python

Similarly to the C++ Cmake implementations, Python scripts can be implemented in exported modules (libraries) and executable scripts (ROS2 nodes).

Use the following command to create a ROS2 python-only package.

```
$ ros2 pkg create --build-type ament_python my_package --dependencies
    other_package_1 other_package_2
```

It will create a folder with a structure and configuration files including the "setup.py" file. The "setup.py" plays a crucial role in the Robot Operating System 2 (ROS2) python development environment. It contains instructions for the colcon system to find and deploy python modules and executable scripts in the ament package. An example of the "setup.py" is:

```
from setuptools import find_packages, setup

package_name = 'my_package'

setup(
name=package_name,
version='0.0.0',
packages=find_packages(exclude=['test']),

data_files=[
('share/ament_index/resource_index/packages',
['resource/' + package_name]),
('share/' + package_name, ['package.xml']),
],
install_requires=['setuptools'],

zip_safe=True,
maintainer='yourname',
maintainer_email='your.email@tum.de',
description='TODO: Package description',
license='TODO: License declaration',

tests_require=['pytest'],
entry_points={
  'console_scripts': [
```

```
  'node = my_package.node:main',
  ],
},
)
```

Let us break down the main parts again.

1. ```
   from setuptools import find_packages, setup

   package_name = 'my_package'
   ```

   Declaration of package name (as variable) and import of the setup modules.

2. ```
   setup(
   name=package_name,
   version='0.0.0',
   packages=find_packages(exclude=['test']),
   ```

   Name for the packaging system and version.

3. ```
   data_files=[
   ('share/ament_index/resource_index/packages',
   ['resource/' + package_name]),
   ('share/' + package_name, ['package.xml']),
   ],
   install_requires=['setuptools'],
   ```

   All the files that compose the package. The package.xml file is also automatically generated. It contains the information that the debian system needs to trace the dependencies of the package.

4. ```
   zip_safe=True,
   maintainer='yourname',
   maintainer_email='your.email@tum.de',
   description='TODO: Package description',
   license='TODO: License declaration',
   ```

   Your developer/maintainer info.

5. ```
   tests_require=['pytest'],
   entry_points={
     'console_scripts': [
     'node = my_package.node:main',
     ],
   ```
   ```

```
},
)
```

Declaration of testing scripts and the executable node scripts.

6. After building this package, you can run your Python nodes as shown below.

```
$ ros2 run my_package node
```

You can note here that Ament python packages do not contain a `CMakeLists.txt` file. However, the `package.xml` is still generated and its structure is the same as for the `C++` packages.

6 Coding style

The ROS community has defined coding standards for developing packages in rospy. The main guidelines can be consulted at `https://docs.ros.org/en/rolling/The-ROS2-Project/Contributing/Code-Style-Language-Versions.html`

a Naming conventions

Python code should follow PEP 8 style guide.

```
package_name

ClassName

method_name

field_name

_private_something

self.__really_private_field

_member_variable

4 space indentation
```

b File structure of a rospy package

Standard ROS practice is to place Python modules under the `src/your_package` subdirectory, making the top-level module name the same as your package. Python requires that directory to have an `__init__.py` file, too.

The files of a rospy package must be arranged as follows.

```
my_python_libs
├── my_python_libs
│   ├── __init__.py
│   ├── python_lib_1.py
│   └── python_lib_2.py
├── resource
│   └── my_python_libs
├── package.xml
├── setup.cfg
├── setup.py
└── test
    ├── test_copyright.py
    ├── test_node_1.py
    └── test_node_2.py
```

Nevertheless, it is recommended that the Python modules and the executable scripts be separated into different packages. This way, the executable nodes can be stored separately with their configuration parameters and launch files. It also prevents unnecessary dependencies from being added to the Python module's package when the scripts do additional functionality.

Finally, it is recommended (but not mandatory) that only one module be implemented per package to prevent namespace clashes.

7 Homework

Let us practice how to create ament python packages and use their contents in another ament python package.

1. Create a catkin workspace and populate it as detailed at the end of this document. To create the rospy packages you can use the following command:

   ```
   $ ros2 pkg create --build-type ament_python my_package --dependencies
       other_package_1 other_package_2
   ```

   This will create the internal structure for the package and give you a setup.py and package.xml template pre-filled with the dependencies. For this exercise, `my_python_libs` depends on `rclpy`. `my_other_python_libs` depends on `rclpy`. And, `my_robot_tutorials` depends on `my_python_libs`, `my_other_python_libs` and `rclpy`.

2. Copy the following code to the corresponding files:

python_lib_1.py

```
 #!/usr/bin/env py -3

def say_it_works():
print("Imported from Library 1 !!")
```

python_lib_2.py

```
 #!/usr/bin/env py -3

class libpy2():
'''
Test library class
'''
def __init__(self, a=1, b=2):
self._a = a
self._b = b

def say_it_too(self, node):
node.get_logger().info("Imported from Library 2 !!")

def calculate(self, node):
c = self._a + self._b
node.get_logger().info("a + b = %f" % (c,) )
```

other_python_lib_1.py

```
#!/usr/bin/env py -3

def say_it_again():
print("Imported from the other Library 1 !!")
```

other_python_lib_2.py

```
#!/usr/bin/env py -3

import rclpy

class libpy3():
'''
Another test library class
'''
def __init__(self, a=1, b=2):
self._a = a
```

```
self._b = b

def say_it_too(self, node):
node.get_logger().info("Imported from Library 3 !!")

def calculate(self, node):
c = self._a - self._b
node.get_logger().info("a - b = %f" % (c,) )
```

3. Once the library packages are implemented, let's prepare the scripts in `my_robot_tutorials` to import and use our libraries. Copy the most basic python script toto use our library into `test_import_1.py`

test_import_1.py

```
#!/usr/bin/env py -3

from my_python_libs.python_lib_1 import say_it_works

say_it_works()
```

4. Compile the workspace and execute the script. If everything works, you should see the printout in the terminal.

5. Implement `test_import_2.py` to import all our libraries and call all their functionality. Here you must consider that `libpy2` and `libpy3` are implemented as a class. Therefore, you must instantiate them before calling their functions. Then, compile and run the script. Also, `libpy2` and `libpy3` depend on the ROS 2 logging functionality. Therefore, you should implement `test_import_2.py` as a ROS 2 Node.

a Questionnaire

1. Did we follow the correct structure and naming convention for developing our rospy packages?

2. Do you need to compile the workspace every time you make a change in a file?

3. If your answer was no, then when is it needed to recompile the workspace?

b File structure of the workspace

```
tutorial_ws
├── README.md
└── src
    ├── my_python_libs
    │   ├── my_python_libs
    │   │   ├── __init__.py
    │   │   ├── python_lib_1.py
    │   │   └── python_lib_2.py
    │   ├── resource
    │   │   └── my_python_libs
    │   ├── package.xml
    │   ├── setup.cfg
    │   └── setup.py
    ├── my_other_python_libs
    │   ├── my_other_python_libs
    │   │   ├── __init__.py
    │   │   ├── other_python_lib_1.py
    │   │   └── other_python_lib_2.py
    │   ├── resource
    │   │   └── my_other_python_libs
    │   ├── package.xml
    │   ├── setup.cfg
    │   └── setup.py
    └── my_robot_tutorials
        ├── my_robot_tutorials
        │   ├── __init__.py
        │   ├── test_import_1.py
        │   └── test_import_2.py
        ├── resource
        │   └── my_robot_tutorials
        ├── package.xml
        ├── setup.cfg
        └── setup.py
```