NExt ApplicationS of Quantum Computing

Initial release of the open-source libraries

Document Properties

Contract Number	951821
Contractual Deadline	M25
Dissemination Level	Public
Nature	Report
Edited by :	Simon Martiel (Atos)
Authors	Simon Martiel (Atos)
Reviewers	Alfons Laarman (ULEI), Mohamed Hibti (EDF)
Date	30/09/2022
Keywords	Open-source, Applications
Status	Final
Release	1.2

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 951821

History of Changes

Release	Date	Author, Organization	Description of Changes
1.0	07/09/22	Simon Martiel (ATOS)	Initial release for review
1.2	26/09/22	Simon Martiel (ATOS)	Implementation of the reviews

Table of Contents

1. Executive Summary

This report summarizes deliverable 3.3: initial release of open-source libraries. Its goal is to provide an insight on the rationale behind the release of the open source libraries developed in the project and provide details on the various steps taken to help and coordinate these releases.

Before diving into the technical details, let us introduce the methodology as specified in the initial proposal.

1. The libraries will focus on circuit-based programming
2. The programming language will be myQLM python (pyAQASM)
3. Bull will act as the integrator of the libraries, as in Bull will notify developers when new bugs are introduced due to changes in the core quantum programming library
4. Following the best practice in software engineering, continuous integration (CI) will be used. The CI platform is in place and operated by Bull.
5. Every 4months, the WP leaders and Bull will synchronize on the UC developments and will identify candidates for additional libraries, in full agreement with the partners who developed the code.
6. During the project, the libraries (source codes and compiled) will be made public outside only once they have reached a correct level of integration quality. By default, no external contribution will be possible until the end of the project, in order to avoid disorganizing in the use case developments. External change requests may be allowed, but without any warranty
7. The library source code is required to be documented and accompanied by application examples, accordingly to the standard coding best practices

Points 6 and 7 are here to ensure best practice and user experience. Points 1 to 5 were decided during proposal redaction and ensure a uniformity of the released libraries: they are all supposed to rely on the same core package, thus allowing for interoperability between them.

2. Context

2.1. Project

One of the core objectives of the NEASQC project, is the development of applicative, quantum based, open-source, software libraries. Each of these libraries develops one range of applications of quantum computing to industrial use cases.
One of the roles of WP3 is to guide the development and release of these libraries. It gives basic coding guidelines and support tools to the different partners to achieve a relative homogeneity in the code and quality of the various libraries. This includes: providing a space to publish the code of the library (here, a project on github), providing coding guidelines and tools to assess their implementation, providing automated testing support for regression testing, etc.
This report details the tools put in place to implement all of software engineering aspects of the work package.

2.2. Work package

As previously introduced WP3 activities serve the project as technical enabling activities that need to happen across the project's use cases. In addition to the points mentioned before and going in more detail, Tasks 3.2 and 3.3 deal with the provision of application centric benchmarks and the software bridge between applications and hardware platforms. On the other side, Task 3.3, is responsible for the management and standardization of the applicative libraries of the project.

3. Proposed library structure

Let us start by describing the proposed structure (or template) for the open source libraries.
All the libraries are hosted on github under the project named NEASQC. Bull added a template library to this project, which consortium partners can fork it in order to start the development of their libraries.

The template assumed that the libraries would be developed in Python (as implicitely implied by the first methodology points). We thus adopted a very standard packaging tool for python called setuptools, that has the merits of being compatible with pip and thus with the package repository pypi.

3.1. Library template

The template assumed that the libraries would be developed in Python (as implied by the first methodology points). We thus adopted a standard packaging tool for python called **setuptools**, that has the merits of being compatible with pip and thus with the package repository **pypi**.

The template further stipulates the following directory structure:

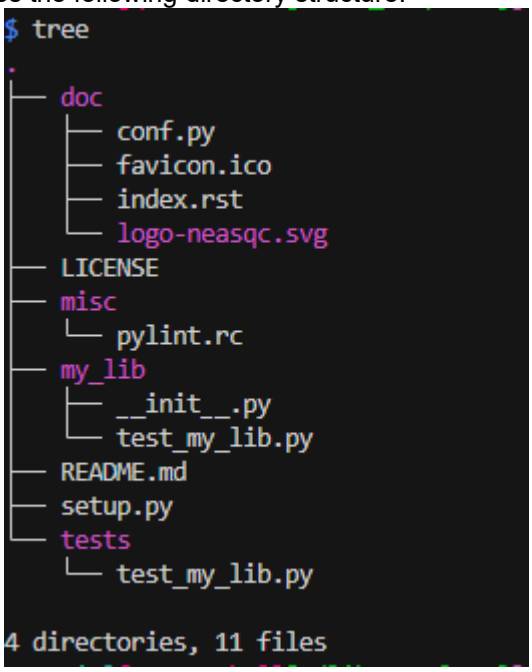

Figure 1: directory and file structure of the library template

The **doc** directory contains the basic files in order to generate a standardized documentation (using the standard tool sphinx) for the library.

The **misc** directory is dedicated to files that are not *per se* part of the library. By default it contains a single file containing a configuration for the linting tool **pylint** (see below).

The **my_lib** directory is a place-holder for the main directory of the library that will later contain the source code. It is initialized with a basic library template (the **__init__.py** and **test_my_lib.py** files).

The **tests** directory contains the unit tests of the library.

The **README.md** file contains a description of the linting requirement and build procedure (see below).

The **setup.py** file contains the package declaration (i.e. name, version, authors, dependencies, build procedure, etc).

3.2. Packaging tool – setup.py

As mentioned above, we rely on a, now standard, tool called **setuptools**. This tools is a python library that contains all the necessary routines in order to build a python package from a collection of source files.
It can build different type of achives or packages, and even, if necessary, trigger compilation of C or C++ dependencies of the Python library. Moreover, the standard Python package management tool **pip** can use setuptools configurations to intall a library directly from the sources. As such, this is the obvious choice for Python library packaging.

Our **setup.py** file is minimalistic by covers installation and testing of the library. Figure 1 shows the default content of this file. The **setup** method contains all the meta data of the library (pretty explicitly), while the PyTest class is a helper class addition a new recipe to the tool in order to automatically run all tests present in the project. This allows the developer to either include the unit tests in the **tests** directory, or directely in the source code of the library (as advised in some modern approach of software engineering).

3.3. Linting and coding standards

In order to improve the quality of the code across the libraries, we also provided advice and support to the library developers on the coding standards to adopt via some regular meetings with the different partners. This was done mainly via the usage of a *linting* tool. This type of tool is used to specify to the user which portion of the code is seemingly of bad quality. The linting tool we advised is called **pylint.** It follows a certain number of community standards, detect transgression, and provides a detailed report and a global grade (between 0 and 10).
The typical rules enforce variable naming style (CaML case vs. snake case), the presence of documentation, the number of local variables used, the length of the declared functions, etc.

We recommended the developers to reach a grade of 9/10 which guarantees a decently good code (we usually require 9.9 or 10 for industrial grade code).
Of course, this grade is only indicative and does not necessarily correlate with direct code quality, however, for inexperienced developers, it constitutes a faithful indicator.

3.4. Unit testing

As mentioned above, we required the library to be package with unit tests. This ensures an efficient detection of regressions, especially for applicative libraries that usually rely on many different libraries. We implemented a basic testing class using the **setuptools** and **pytest** libraries. It can automatically detect tests present in the source files or in the **tests** directory (see Figure 1).

```python
import os, sys
from setuptools import setup, find_packages
from setuptools.command.test import test as TestCommand

class PyTest(TestCommand):
    """
    A test command to run pytest on a the full repository.
    This means that any function name test_XXXX
    or any class named TestXXXX will be found and run.
    """
    def initialize_options(self):
        TestCommand.initialize_options(self)
        self.pytest_args = []

    def finalize_options(self):
        TestCommand.finalize_options(self)
        self.test_args = []
        self.test_suite = True

    def run_tests(self):
        import pytest
        errno = pytest.main([".", "-vv"])
        sys.exit(errno)

setup(
    name="mylib",
    version="0.0.1",
    author="XXX",
    license="European Union Public License 1.2",

    packages=find_packages(),
    install_requires=["numpy"],
    # Don't change these two lines
    tests_require=["pytest"],
    cmdclass={'test': PyTest},
)
```

Figure 2: default setup.py content

4. Documentation

We created a repository, called **neasqc.github.io** in the NEASQC github project dedicated to generating a common documentation for the libraries. This repository hosts the generated html documentation for the libraries. These html files are updated by actions located in the libraries repositories (see next section).

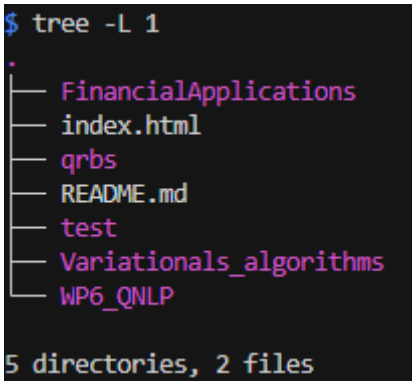

Figure 3: structure of the html documentation

The documentation contains a trivial index.html, and additional directories for each of the libraries that provides documentation.

5. Unit testing and continuous integration

To facilitate regressions and bugs detection, the template library was also equipped with a basic continuous integration action, directly handled by github.
This action is triggered every time a new version of the code is pushed. It consists in running two linting tools to provide a quality grade for the repo, and then running the unit tests of the repository via the “test” recipe implemented via the setuptools library.
Another action builds the documentation and updates the main documentation repository.

These actions are specified via .yaml configuration files (as required by github).

Having the unit testing embedded in an automated action allows to avoid regressions in the typical situation where a developer makes changes and run tests in its private environment, failing to detect bugs due to missing or outdated dependencies.

6. Status of the libraries

As of today, the NEASQC github project hosts 7 different repositories:

- The library template “lib_template”
- The documentation content “neasqc.github.io”
- 5 released libraries

The currently released libraries are the following:

- **NEASQC/FinancialApplications**
- **NEASQC/Variational_Algorithms**
- **NEASQC/ft-2-quantum-sat** (corresponding to D6.6: QPSA Divide and quantum open source software)
- **NEASQC/D4.2** (corresponding to D4.2: QCCC Alpha)
- **NEASQC/WP6_QNLP** (corresponding to D6.7: QNLP alpha prototype)

These mostly correspond to pre-releases (or alpha prototypes). Most of these repository meet all the requirements described in this document and constitute solid milestones to the final releases.

7. List of Figures