NExt ApplicationS of Quantum Computing

D3.5: The NEASQC benchmark suite (TNBS)

Document Properties

Contract Number	951821
Contractual Deadline	31/10/2023
Dissemination Level	Public
Nature	Report
Editors	Diego Andrade, CITIC-UDC Andrés Gómez, CESGA
Authors	Diego Andrade, CITIC-UDC Gonzalo Ferro, CESGA Andrés Gómez, CESGA
Reviewers	Cyril Allouche, EVIDEN Arnaud Gazda, EVIDEN
Date	27/10/2023
Keywords	Benchmark, Quantum computers, kernels
Status	Submitted
Release	1.0

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, Organisation	Description of Changes
0.1	01/02/2023	Gonzalo Ferro, CESGA; Andrés Gómez, CESGA; Diego Andrade, CITIC-UDC	First version
0.2	08/10/2023	Andrés Gómez, CESGA; Diego Andrade, CITIC-UDC	Including web site and formatting
0.3	25/10/2023	Gonzalo Ferro, CESGA	Fixing the naming according to the Glossary of deliverable 3.5
1.0	26/10/2023	Gonzalo Ferro, CESGA; Diego Andrade, CITIC-UDC	Fixing comments of the reviewers

Table of Contents

1.Executive Summary

This document describes The NEASQC Benchmark Suite (TNBS). The objective of the document is to define the benchmarks that compose it, and the methodology for executing them and reporting their results. It includes also a short description of the TNBS website and the associated repository of submitted benchmark results.

TNBS has been designed to take into account four main objectives:

- Objective 1: the test cases which compose the suite must help computer architects, programmers and researchers to design future quantum computers, taking into account the variability in the performance introduced by the different components of the stack.
- Objective 2: the test cases must help to understand the evolution of quantum computers (more qubits, better topologies) or to improve the current one (reduction of noise, better compilers, etc.).
- Objective 3: the test cases must allow to compare the performance of different platforms. Currently, there are many different proposals (as transmons, ions, neutral atoms, etc.) which use different one- and two-qubit gates. The benchmarks must allow users and researchers to compare the performance of different platforms, and find their bottlenecks. For users, they should allow them to find the best platforms for their application.
- Objective 4: the test cases should consider other metrics which may be important to understand the quantum computing advantage (such as energy consumption, throughput or better scalability).

To achieve these objectives, TNBS has found representative kernels among the NEASQC uses cases to define a set of well-defined tests. Currently, TNBS is composed of four cases, each of them with:

1. A full and detailed documentation of the suite and of each microbenchmark. This documentation will include the definition of the microbenchmarks, the rules for executing them, the methods for reporting the results, and the methods for evaluating the final results.
2. An Eviden myQLM [1] reference implementation, that allows the analysis of their complete results.

This document includes sections for describing the general rules for executing the benchmarks, the methods for generating the results, the process to submit them, the list of benchmarks that compose the Suite, and a brief summary of the capabilities of the repository. As appendices, the JSON schema for reporting the results, the template for defining new cases, and the four documents that describe the current benchmarks are included. All the cases included a reference version based on Eviden myQLM framework.

In the future, the TNBS will increase the number of cases, from NEASQC use cases or from external proposals. So, this document will be a live document, with continuous improvements.

[1] See myQLM quantum software stack

2.Introduction

The NEASQC Benchmark Suite (TNBS) is composed of several microbenchmarks extracted from the use cases of the NEASQC project, covering several domains where Quantum Computing (QC) has been proven to be useful. The general objectives of the suite are:

- Objective 1: the suite must help computer architects, programmers and researchers to design future quantum computers, taking into account the variability in their performance introduced by the different components of the stack.
- Objective 2: the suite must help to guide the evolution of quantum computers (more qubits, better topologies) or to improve the current platforms (reduction of noise, better compilers, etc.).
- Objective 3: the suite must allow a comparison of the performance of different platforms. Currently, there are many different proposals for a physical implementation (as transmons, ions, neutral atoms, etc.) that use different one- and two-qubit gates. The suite must allow users and researchers to compare the performance of different platforms, and find their bottlenecks. For users, it should allow them to find the best platforms for their applications.
- Objective 4: The suite must consider metrics important to understand the quantum computing advantage (such as energy consumption, throughput or scalability).

The NEASQC benchmark suite does not address the topic of “quantum supremacy”, i.e., to find an algorithm that can be solved on a quantum computer but it is difficult or almost impossible to solve in a classical computer in a reasonable time.

The microbenchmarks that compose the suite meet the following characteristics:

- They are based on a quantum routine or kernel that is common to several use cases and representative of the needs of other algorithms of the same family.
- They scale with the number of qubits up to a reasonable number. Certainly, many of the benchmarks are limited by the classical capacity for pre-processing and post-processing information.
- They are defined at high level, i.e., they must be agnostic of the quantum computer architecture, programming language, etc.

2.1. Structure of the TNBS

As stated before, TNBS consists of a set of microbenchmarks (similar to the concept of synthetic benchmarks/kernels in classical computer benchmarking). It is organized into two categories: general benchmarks (such as Quantum Fourier Transform) which can be common to several fields and specific ones (such as the execution of one step of a Variational algorithm).

Taking into account this high-level structure, TNBS is composed of:

- A full and detailed general documentation of the suite (this document) and a separate document for each microbenchmark. This documentation includes the definition of the microbenchmarks, the rules for executing them, the methods for reporting the results and the methods for evaluating the final results.
- A compatible Eviden myQLM reference implementation.
- A web platform where every user of the suite can submit their own results to cross-compare the results on different platforms.

2.2. TNBS Glossary

The next terminology has to be used for defining TNBS:

- **Benchmark**: it is each one of the use cases in which the TNBS is divided. Each benchmark is composed of a description of the **Kernel** and its corresponding **Benchmark Test Case**.

- **Kernel**: it is a core quantum subroutine or step which may be implemented using different procedures or algorithmic approaches. Because of this, a **Kernel** is described using a high-level mathematical or procedural definition. Examples are the *Quantum Fourier Transformation*, the loading of an initial quantum state in a quantum circuit, etc.
- **Benchmark Test Case** (**BTC**): it is a particular problem that involves the execution of the **Kernel**. The output of this **BTC** must be verifiable analytically or through a classical simulation. The **BTC** is used for evaluating the performance of a Quantum platform that executes the **Kernel**. For example, the loading of a specific statistical distribution into a quantum circuit (**BTC**) is used to evaluate the performance of a platform for loading an initial quantum state in a quantum circuit (**Kernel**).

3.General execution rules

The rules for executing the cases include a set of good practices, such as:

- It is forbidden to make any specific improvement in the compilers, schedulers or other modules of the stack defined ad-hoc for the **Benchmark Test Cases**. Ad-hoc improvements that operate at the **Kernel** level are perfectly fine if they do not affect the generality of the **Kernel** but, addressing the **BTC** using any kind of shortcut or ad-hoc solution would be considered cheating.
- The precision of the Floating Point Numbers in the classical part is double-precision (64 bits)

In addition, the protocol to execute the benchmarks follows a set of rules and specifications:

- The minimum and maximum number of qubits supported by the benchmark.
- The number of shots to be used. These numbers can depend on the number of qubits.
- The number of repetitions of the case or a rule to calculate them. Because there is some uncertainty in the metrics to use, the results must be reported with errors. This means that a certain number of repetitions could be needed.
- If the test is executed in a loop (as could be the number of qubits or number of steps or number of layers in a Quantum Machine Learning algorithm), the point where the benchmark must be early stopped. For example, for failing the verification of results.

4.Benchmark results generation

Each **Benchmark** must have a well-documented execution procedure. This procedure must specify the set of number of qubits values to be tested. For evaluating a platform using a determined **Benchmark**, for a fixed number of qubits, two types of metrics will be used:

- A common metric across all the **Benchmarks** of the Suite: the **elapsed time**, which is the time that the platform needs to execute the **Benchmark**.
- One or several **Benchmark**-dependent metrics to verify the output. These verification metrics must be defined for each **Benchmark** inside its corresponding documentation.

To have a statistical significance, for each fixed number of qubits, the **Benchmark** should be executed a fixed number of repetitions, M, that is computed as follows:

1. The **Benchmark** should be executed 10 times.
2. The mean **elapsed time** (μ_T) and its corresponding standard deviation (σ_T) must be computed. Then the number of repetitions for having a relative error in the **elapsed time** of 5 %, $r_T = 0.05$, with a confidence level of 95 %, $\alpha = 0.95$, (M_T) must be computed following equation (4.1), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for the desired α

$$M_T = \left(\frac{\sigma_T Z_{1-\frac{1}{2}}}{r \mu_T} \right)^2 \tag{4.1}$$

3. For any verification metric, m, its standard deviation (σ_m) should be computed. Then the number of repetitions for having a desired absolute error (ϵ_m), which will depend on the **Benchmark**, with a confidence level of 95%, $\alpha = 0.95$, (M_m) must be computed following equation (4.2), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for the desired α. Depending on the **Benchmark** the desired error can be a relative one, so in this case the mean of the metric (μ_m) should be computed too. In any case, this should be explicitly described in the corresponding document.

$$M_m = \left(\frac{\sigma_T Z_{1-\frac{1}{2}}}{\epsilon_m} \right)^2 \tag{4.2}$$

4. The number of repetitions, then, will be equal to $M = \max{(M_T, M_m)}$

With this procedure, it can be guaranteed that the returned **elapsed time** will have a relative error lower than 5% with a confidence level of 95%. Additionally, any particular metric will have an absolute error ϵ_m with a confidence level of 95%.

The **elapsed time** can be measured separately for the classical and quantum parts of the **Benchmark**.

The procedure to execute the **Benchmark** will be as follows:

1. The algorithm is executed M times, measuring the total **elapsed time** for each execution.
    - The time measurement can be performed separately for the classical and the quantum parts.
2. The average and the standard deviation of the **elapsed time** should be calculated.
3. The average and the standard deviation of the different verification metrics should be calculated too.

5.Benchmark results submission

The results of each **Benchmark** must be reported in a separate JSON format file. The scheme of this JSON file is included as Appendix to this document. This file has two parts: one devoted to the description of the target platform, and another to the description of the specific results of the cases. The information to include in the file is:

- Name or id of the **Benchmark**.
- Start timestamp of the execution of this **Benchmark** following RFC 3339.
- End timestamp of the execution of this **Benchmark** following the same RFC.
- The programming language used for its implementation.
- Name of the provider of the programming language.
- List of APIs used by the **Benchmark**, including their name, version, and supplier.
- Ordered list of steps for the transpilation of QPU code.
- Ordered list of the steps for classical compilation.
- Time routine used for measuring execution time.
- Detailed report of the results. It must include, for each number of qubits tested:
  - The used placement, list of QPUs used (the benchmark allows to use of more than one QPU for executing it).
  - Total elapsed time in seconds and its sigma.
  - Total time for executing the quantum algorithm in seconds and its sigma.
  - Total time for executing the classical part of the algorithm in seconds and its sigma.
  - Information about additional metrics for the case.

During the execution of the case, other outputs can be included in external files or standard output. However, it is recommended that this information is placed out of the measuring loop, so it will not interfere with the measurement of the elapsed times. Any case that does not pass the verification should not be included in the output. The main idea of this JSON file is to simplify the reporting of the results, both to upload the data to a common data repository and to generate a printable report.

6.List of benchmarks

The current list of **Benchmark** that compose the Suite are

- T01: Benchmark for Probability Loading Algorithms (Annex C). This **Benchmark** addresses one of the current main problems of Quantum Computing: how to load classical data into the amplitudes. Extracted from the WP5, it is important for many different quantum algorithms.
- T02: Benchmark for Amplitude Estimation Algorithms (Annex D). Amplitude Estimation algorithms are used as a technique to accelerate some results. It has been defined from the needs of WP5.
- T03: Benchmark for Phase Estimation Algorithms (Annex E). Phase Estimation is a key algorithm in Quantum Computing that is used widely. Concretely, it is used in some possible solutions of the Use Case 5 from WP5.
- T04: Benchmark for Parent Hamiltonian (Annex F). Variational algorithms, especially Variation Quantum Eigensolver, are proposed tools to get good results from the current noisy quantum computers. However, their results depend on the selected ansatz and used optimizer. This **Benchmark** tests the quality of the quantum computers and libraries to execute typical ansatzes used in WP4, without depending on the optimizer.

More actualized information about the benchmarks can be obtained from:

- Online documentation: https://neasqc.github.io/TNBS/tnbs.html
- Reference Software: https://github.com/NEASQC/WP3_Benchmark

7.The NEASQC benchmark suite repository.

The Quantum Computing Benchmark Repository (QCBencRepo) gives support to the collection and centralized storage of the results generated by the TNBS suite. It stores the results reported for different Quantum Computing (QC) platforms (Quantum Processing Units, QPUs) by different organizations. We are commenting separately on the main principles applied to its design, at the architectural level, the design of the web interface of the repository, the results report visualization, and a discussion on the initial release.

7.1. Design of the architecture and main components

The repository will be composed of a backend exposed through a REST API that serves as an interface between web and Python clients, and the databases where the information is stored. As the benchmark suite generates the reports as JSON files, the database system will be a relational (SQL) database where the JSON files will be preserved as they were reported, and linked to the appropriate records of the database. Figure 1 shows the general architecture of the system. The web front end is implemented using a Javascript-responsive framework (NextJS, https://nextjs.org), complemented with a visual library of components (NextUI, https://nextui.org). The D3.js (https://d3js.org) library is used for the creation of data visualizations. All of them are well-established open-source projects with strong supporting communities. The backend is implemented using the Headless CMS framework (Strapi, https://strapi.io), which provides: the database system, the REST API, and an admin panel that can be used by the repository administrators and the reporting organizations.

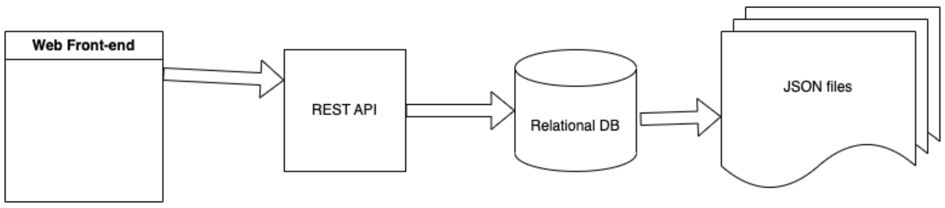


Figure 1: General architecture of the repository system

The reporting of the benchmark results will be done securely through the Strapi admin panel or directly through the REST API. The REST API report uses an auth key which is associated with each reporting organization and that ensures that the report is submitted by that organization. The REST API has also several public endpoints that can be accessed by third-party websites or tools to access benchmarking reports that have been published by different organizations. The user auth system is required to allow reporting organizations to securely access our repository and report verified results. Figure 2 shows the SQL database diagram and the integration of the JSON files reported.

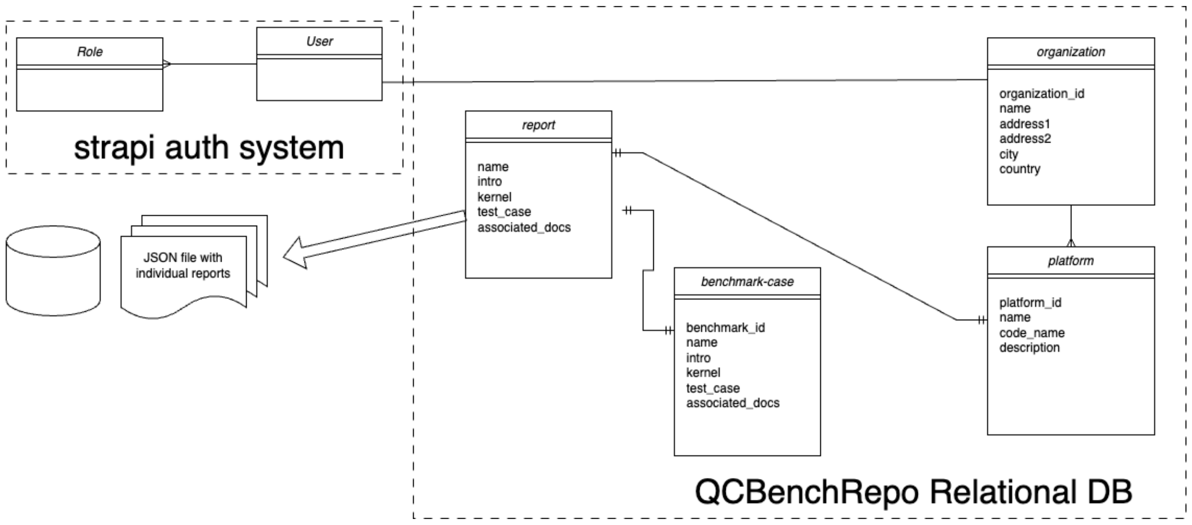
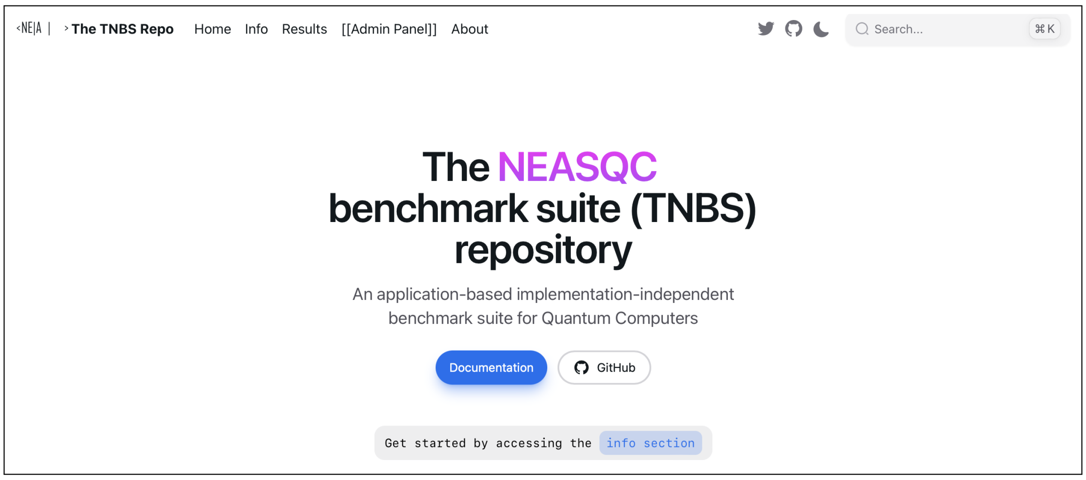


Figure 2: Database schema

7.2. Web interface organization

The web interface is divided into two parts: the repository front end and the admin panel. The admin panel can be used by organizations to create new Platforms and new Reports (reports) on these Platforms for the different Benchmark-Cases. The reporting form allows to add manually the JSON file with the output of the benchmark execution.

The design of the web interface is discussed here. The homepage of the repository contains a link to a page with the basic information of the benchmark suite: the philosophy behind it, the benchmarks that compose it, the procedure to execute it in a platform, and the method to report the results. In this initial release, the results section contains only a sample of the results. In the future, it will allow us to consult and compare the results provided by different reporting organizations that have been made public. A user will be able to visualize the existing benchmark reports submitted by the community. It will also enable the comparison of the reports from different platforms.



The NEASQC benchmark suite (TNBS) repository

An application-based implementation-independent benchmark suite for Quantum Computers

Get started by accessing the info section

Figure 3: Home page of the web front-end

7.2.1. Report visualization and comparison

Now, we must discuss in detail the best way to inspect visually individual reports and comparisons between two ones. For this, we must understand first which are the most important qualitative and quantitative results are contained in a report.

- Execution time of the benchmark: it is the wall time to execute each benchmark, and is normally decomposed in classical computing time and quantum computing time. As we will perform several executions per benchmark, we must report the average time, the maximum, the minimum, and the standard deviation. Figure 4 represents a possible illustration of this kind of result using a box plot.
- For the platform, the number of Qubits available, and the types of gates available. The gates are classified as single, entanglement, and global entanglement gates.
- Each benchmark can define several additional ad-hoc metrics to cover benchmark-dependent metrics, such as those related to measuring the accuracy of the output of the computation. The comparison of several of these ad-hoc metrics, and maybe also of the execution times, can be done using a radial chart. Figure 4 shows an example of this.

The boxplots for different metrics are already included in the sample results included in the initial release of the repository.

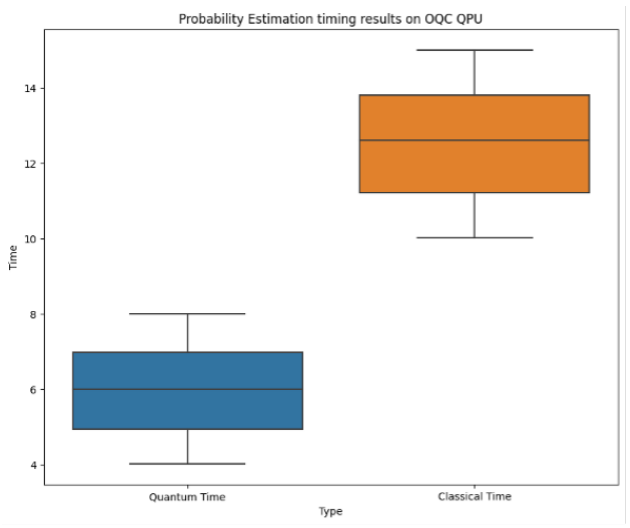


Figure 4: Sample boxplot to visualize timing results for a benchmark.

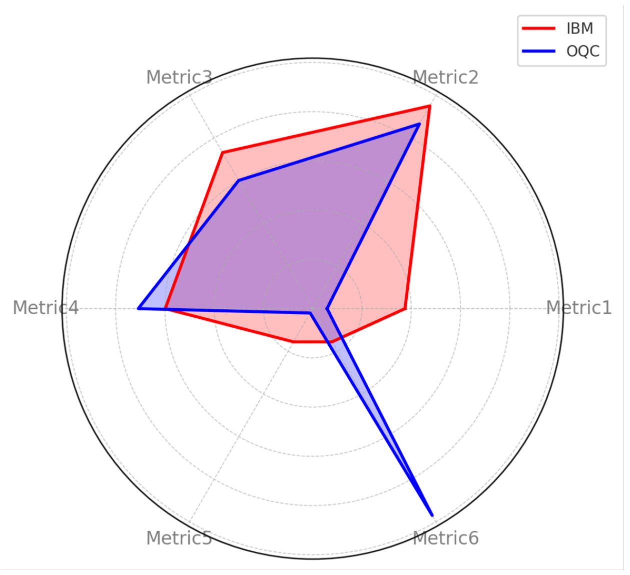


Figure 5: Radial chart as a condensed form of comparison between two platforms across several metrics

7.3. Initial implementation and first steps

The backend is based on Strapi v4 Headless CMS which enables the definition and access through a REST API to the backend database. Strapi is an open-source project with a great supporting community and it enables a secure and high-quality implementation of many of the features required for the backend of the repository. It is also flexible, adaptable, and extendable enough to not constrain future extensions and refinements of the backend.

The database of the backend is composed of several data entities that are defined formally by the previously discussed SQL database schema:

- Benchmark-Cases
- Users
  - Roles. The expected roles are:
    - Repository-Administrator: It has full access to everything. ©item Organization-Administrator: It has full access to the elements of an organization (Platforms, Reports, and Users).
- Organization (linked to a User with the role Organization-Administrator)

- Platforms (linked to Organizations)
- Reports (linked to both an Organization and a Platform)

The Strapi admin panel of the backend allows access to CRUD (Create Read Update Delete) operations to users assigned to different roles. For the users with the Organization-Administrator role, this backend access is restricted just to some data entities (Report, Platform, Organization), and only to the individual records created by that organization. Figure 6 contains a screenshot of the backend login page. Figure 7 contains a screenshot of the report creation form.

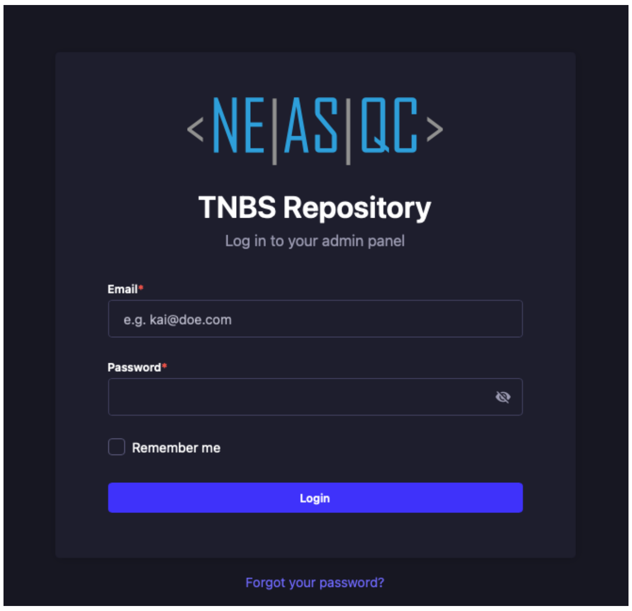


Figure 6: TNBS Repository backend login page

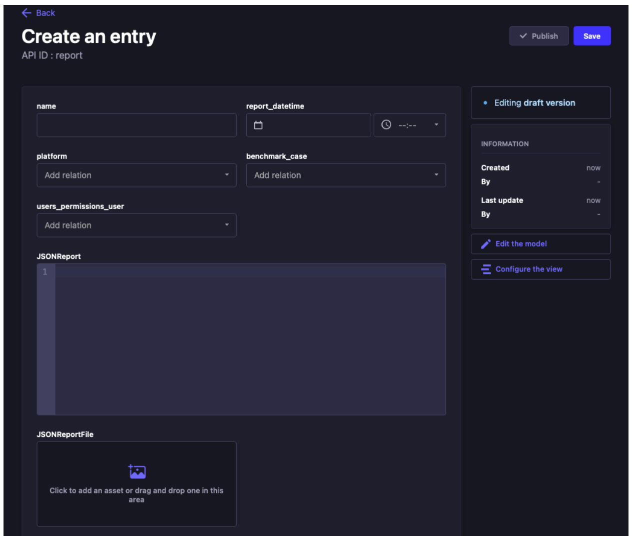
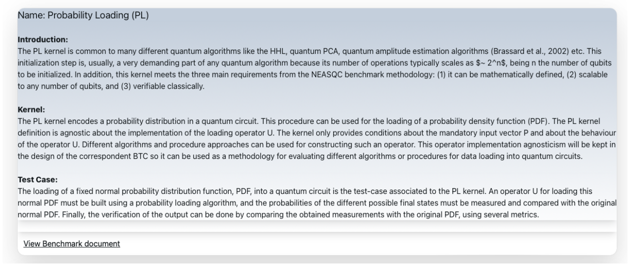


Figure 7: Report creation form

In the next months, the access to the backend to report results by organizations will be done through a closed beta. There will be a form on the front end for organizations to express interest in participating in the closed beta. The reporting will be done initially by uploading the generated JSON to the report form. In the future, the reporting will be able to be done directly through the Strapi API REST using authenticated requests.

The public front end of the repository is based on the Nextjs Javascript Framework and the NextUI visual components library. The use of these tools seeks to ensure a good visual quality that does not undermine the confidence in the quality of the suite. The initial release of the front end will contain information to disseminate the suite among the Quantum Computing Community, specifically:



Name: Probability Loading (PL)

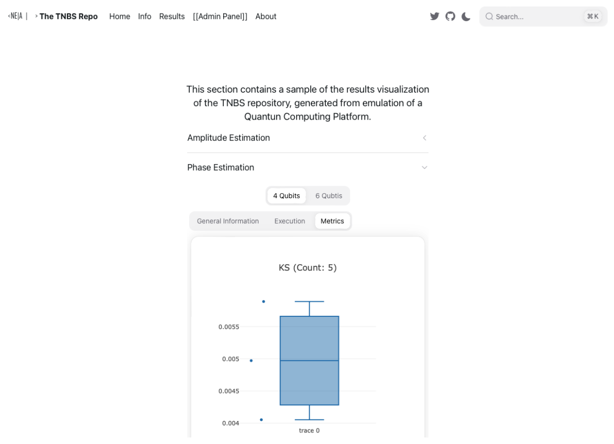
Introduction:
The PL kernel is common to many different quantum algorithms like the HHL, quantum PCA, quantum amplitude estimation algorithms (Brassard et al., 2002) etc. This initialization step is, usually, a very demanding part of any quantum algorithm because its number of operations typically scales as $~ 2^n$, being n the number of qubits to be initialized. In addition, this kernel meets the three main requirements from the NEASQC benchmark methodology: (1) it can be mathematically defined, (2) scalable to any number of qubits, and (3) verifiable classically.

Kernel:
The PL kernel encodes a probability distribution in a quantum circuit. This procedure can be used for the loading of a probability density function (PDF). The PL kernel definition is agnostic about the implementation of the loading operator U. The kernel only provides conditions about the mandatory input vector P and about the behaviour of the operator U. Different algorithms and procedure approaches can be used for constructing such an operator. This operator implementation agnosticism will be kept in the design of the correspondent BTC so it can be used as a methodology for evaluating different algorithms or procedures for data loading into quantum circuits.

Test Case:
The loading of a fixed normal probability distribution function, PDF, into a quantum circuit is the test-case associated to the PL kernel. An operator U for loading this normal PDF must be built using a probability loading algorithm, and the probabilities of the different possible final states must be measured and compared with the original normal PDF. Finally, the verification of the output can be done by comparing the obtained measurements with the original PDF, using several metrics.

View Benchmark document

***Figure 8**: Sample with the information of one of the benchmarks of the suite*



This section contains a sample of the results visualization of the TNBS repository, generated from emulation of a Quantun Computing Platform.

***Figure 9**: Sample result*

- A description of the philosophy behind the benchmark suite.
- A description of the benchmarks defined up to this day. Check Figure 8 to see a sample of the way the information of one of the benchmarks is presented in the current website, including a link to the pdf with the official documentation.
- The official general documentation of the suite (in pdf), and the official individual documents for each benchmark case defined up to this day.
- A link to the GitHub repository containing the reference implementations of the benchmarks using Eviden myQLM library.
- Several samples of the visualization of the output reports generated by our reference implementation in an emulator. Check one sample of this in the screenshot of Figure 9.

The main reason to start the repository with a closed beta model is that, so far, all our results with the benchmarks are based on emulation not in real hardware, and given that the benchmark definition is mathematical or procedural, and the reference implementation is only provided in one QC framework (Eviden myQLM library), it is very unlikely that we receive reports from many organizations in the first year of the benchmark suite. A completely open reporting model could populate the repository with bogus or fake results and irremediably undermine the reputation of the suite.

Regarding, showing a sample of results but not a real results section. We do have a single report on real hardware yet, so there is no point in presenting an empty results section in the initial release of the repository website, as this could also undermine its reputation. The presentation of sample results based on emulation of QC hardware is a good way to present to the community what to expect about the benchmark results presentation.

Our efforts in the next months must be focused on:

- Using the front end to disseminate the benchmark suite among the QC community.
- Getting our first reporters for the closed beta.
- Refine the way results are presented and compared.

- Refine to improve different aspects of the repository and the benchmark suite identified from our relationship with the beta reporters.

8.Conclusions

The TNBS is a benchmark suite that targets quantum platforms. The **Benchmarks** included in this suite belong to common processes performed in Quantum platforms by the NEASQC use cases. Currently, four **Benchmarks** are included, but they will be expanded soon. The **Benchmarks** should scale with the number of Qubits which makes them resilient to the upcoming evolution of this technology. Also, they are self-verifiable which allows us to check the correctness of the output in a reasonable time. They contemplate the existence of hybrid algorithms that combine classical and quantum computations and future modular and parallel architectures. Overall, this suite can help programmers to choose the most appropriate platforms, and architects to create better systems.

List of Acronyms

Term	Definition
API	Application Interface
CMS	Content Management System
JSON	JavaScript Object Notation
NEASQC	NExt ApplicationS of Quantum Computing project
QC	Quantum Computer/ing
QCBencRep	Quantum Computing Benchmark Repository
QLM	Quantum Learning Machine
QPU	Quantum Processing Unit
REST	Representational state transfer
SQL	Structured Query Language
TNBS	The NEASQC Benchmark Suite

Table 1: Acronyms and Abbreviations

List of Figures

List of Tables

List of Listings

Bibliography

Gómez, A., Mouriño Gallego, J. C., Andrade Canosa, D., Simon, M., & Musso, D. (2021). D3.2: Design document benchmark methodology.

A.TNBS JSON schema

This is the JSON schema which should be used to report results. This file can be downloaded from **TNBS repository**.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
    "$id" : "https://neasqc.eu/benchmark.V1.schema.json",
    "title" : "NEASQC JSON reporting schema",
    "description":"JSON Schema to help in the generation of reporting JSON of NEASQC benchmark
    suite",
    "type": "object",
    "properties": {
        "ReportOrganization": {
            "type": "string",
            "description": "Name of the organisation which reports the results"
        },
        "MachineName": {
            "type": "string"
        },
        "QPUModel": {
            "type": "string",
            "description": "Identification or model of the QPU"
        },
        "QPUDescription": {
            "type": "array",
            "description": "Description of capabilities of the Quantum Computer",
            "items": {
                "type": "object",
                "properties": {
                    "NumberOfQPUs": {
                        "type": "integer",
                        "description": "Number of QPUs of this type",
                        "minimum": 0,
                        "exclusiveMinimum": true
                    },
                    "QPUs": {
                        "type": "array",
                        "description": "Description of each QPU",
                        "items": {
                            "type": "object",
                            "properties": {
                                "BasicGates": {
                                    "type": "array",
                                    "uniqueItems": true,
                                    "minItems": 2,
                                    "items": {
                                        "type": "string"
                                    }
                                },
                                "NumberOfQubits": {
                                    "type": "integer",
                                    "description": "Number of Qubits of the QPU"
                                },
                                "Qubits": {
                                    "type": "array",
                                    "description": "List with the properties of each qubit",
                                    "items": {
                                        "type": "object",
                                        "properties": {
                                            "QubitNumber": {
                                                "type": "integer",
                                                "description": "number Assigned to the qubit"
                                            },
                                            "T1": {
                                                "type": "number",
                                                "description": "T1 in ns",
                                                "minimum": 0,
                                                "exclusiveMinimum": true
                                            },
                                            "T2": {
                                                "type": "number",
```

```json
                                            "description": "T2 in ns",
                                            "minimum": 0,
                                            "exclusiveMinimum": true
                                        }
                                    },
                                    "required": [
                                        "QubitNumber",
                                        "T1",
                                        "T2"
                                    ]
                                }
                            },
                            "Gates": {
                                "type": "array",
                                "description": "List of the basic gates for each qubit",
                                "items": {
                                    "type": "object",
                                    "properties": {
                                        "Gate": {
                                            "type": "string",
                                            "description": "Name of the gate"
                                        },
                                        "Type": {
                                            "enum": [

          "Single",
          "Entanglement",
          "GlobalEntanglement"

                                            ],
                                            "description": "single qubit or entanglement
qubit (more than 1 qubit)"
                                        },
                                        "Qubits": {
                                            "type": "array",
                                            "description": "Ordered list of qubits where is
applied",
                                            "uniqueItems": true,
                                            "minItems": 1,
                                            "items": {
                                                "type": "integer"
                                            }
                                        },
                                        "MaxTime": {
                                            "type": "number",
                                            "description": "Maximum time for executing this
gate (in ns)",
                                            "minimum": 0,
                                            "exclusiveMinimum": true
                                        },
                                        "Symmetric": {
                                            "type": "boolean",
                                            "description": "If the gate can be applied in any
 order of qubits with the same timing"
                                        }
                                    },
                                    "required": [
                                        "Gate",
                                        "Type",
                                        "Qubits",
                                        "MaxTime"
                                    ]
                                }
                            },
                            "Technology": {
                                "enum": ["charge qubit","flux qubit","phase qubit","photon","
ion","neutral atom","diamond","quantum dot","other"]
                            },
                            "Other": {
                                "type": "string",
                                "description": "If other, name or description of the
technology"
                            }
                            },
                            "required": [
                                "BasicGates",
                                "Qubits",
                                "Gates",
                                "Technology"
                            ]
                        }
                    }
                },
                "required": [
                    "NumberOfQPUs",
                    "QPUs"
                ]
            }
        },
        "CPUModel": {
            "type": "string",
            "description": "model of the classical CPU"
        },
        "Frequency": {
            "type": "number",
            "description": "Frequency in GHz of the classical CPU"
        },
        "Network": {
            "type": "object",
            "properties": {
                "Model": {
                    "type": "string",
                    "description": "Model of the interconnection between CPUs"
                },
                "Version": {
                    "type": "string",
                    "description": "Version"
                },
                "Topology": {
                    "type": "string",
                    "description": "Type of topology of the network"
                }
            },
            "required": [
                "Model",
                "Version",
                "Topology"
            ]
        },
        "QPUCPUConnection": {
            "type": "object",
            "description": "Information about how the QPU is connected to the classical QPU",
            "properties": {
                "Type": {
                    "type": "string",
                    "description": "Type of connection as PCI"
                },
                "Version": {
                    "type": "string",
                    "description": "Version of the connection"
                }
            },
            "required": [
                "Type",
                "Version"
            ]
        },
        "Benchmarks": {
            "type": "array",
            "description": "Results of the different benchmarks",
            "items": {
                "type": "object",
                "properties": {
                    "BenchmarkKernel": {
                        "type": "string",
                        "description": "Name or id of the benchmark"
                    },
                    "StartTime": {
                        "type": "string",
                        "description": "Start time for this benchmark following RFC 3339, section 5.6.",
                        "format": "date-time"
                    },
                    "EndTime": {
                        "type": "string",
                        "description": "End time for this benchmark following RFC 3339, section 5.6.",
                        "format": "date-time"
                    },
                    "ProgramLanguage": {
                        "type": "string",
                        "description": "Programming language"
                    },
                    "ProgramLanguageVersion": {
                        "type": "string",
                        "description": "Programming language version"
                    },
                    "ProgramLanguageVendor": {
                        "type": "string",
                        "description": "Name of the provider of the programming language"
                    },
                    "API": {
                        "type": "array",
                        "description": "List of APIs used by the benchmark",
                        "items": {
                            "type": "object",
                            "properties": {
                                "Name": {
                                    "type": "string",
                                    "description": "Name of the API"
                                },
                                "Version": {
                                    "type": "string",
                                    "description": "Version of the API"
                                }
                            },
                            "required": [
                                "Name",
                                "Version"
                            ]
                        }
                    },
                    "QuantumCompililation": {
                        "type": "array",
                        "description": "List of steps for the transpilation of QPU code",
                        "items": {
                            "type": "object",
                            "properties": {
                                "Step": {
                                    "type": "string",
                                    "description": "Name of the step module"
                                },
                                "Version": {
                                    "type": "string",
                                    "description": "Version of this module"
                                },
                                "Flags": {
                                    "type": "string",
                                    "description": "Flag used for this step"
                                }
                            },
                            "required": [
                                "Step",
                                "Version",
                                "Flags"
                            ]
                        }
                    },
                    "ClassicalCompiler": {
                        "type": "array",
                        "description": "Ordered list of the steps for classical compilation",
                        "items": {
                            "type": "object",
                            "properties": {
                                "Step": {
                                    "type": "string",
                                    "description": "Step for classical compilation"
                                },
                                "Version": {
                                    "type": "string",
                                    "description": "version of this step"
                                },
                                "Flags": {
                                    "type": "string",
                                    "description": "Flags used for this step"
                                }
                            },
                            "required": [
                                "Step",
                                "Version",
                                "Flags"
                            ]
                        }
                    },
                    "TimeMethod": {
                        "type": "string",
                        "description": "Time routine for measuring execution time"
                    },
                    "Results": {
                        "type": "array",
                        "description": "Detailed report of the results",
                        "items": {
                            "type": "object",
                            "properties": {
                                "NumberOfQubits": {
                                    "type": "integer",
                                    "description": "Number of qubits used for this result",
                                    "minimum": 0,
                                    "exclusiveMinimum": true
                                },
                                "QubitPlacement": {
                                    "type": "array",
                                    "description": "Ordered list of qubits used for executing this case",
                                    "items": {
                                        "type": "integer"
                                    }
                                },
                                "QPUs": {
                                    "type": "array",
                                    "description": "List of QPUs used for the benchmark",
                                    "items": {
                                        "type": "integer"
                                    }
                                },
                                "CPUs": {
                                    "type": "array",
                                    "description": "List of CPUs used in the benchmark",
                                    "items": {
                                        "type": "integer"
                                    }
                                },
                                "TotalTime": {
                                    "type": "number",
                                    "description": "Total elapsed time in seconds",
                                    "minimum": 0,
                                    "exclusiveMinimum": true
                                },
                                "SigmaTotalTime": {
                                    "type": "number",
                                    "description": "Sigma of total execution time"
                                },
                                "QuantumTime": {
                                    "type": "number",
                                    "description": "Total time for executing the quantum
    algorithm in seconds",
                                    "minimum": 0,
                                    "exclusiveMinimum": true
                                },
                                "SigmaQuantumTime": {
                                    "type": "number",
                                    "description": "Sigma of quantum time"
                                },
                                "ClassicalTime": {
                                    "type": "number",
                                    "description": "Total time for executing the classical part
    of the algorithm in seconds",
                                    "minimum": 0,
                                    "exclusiveMinimum": true
                                },
                                "SigmaClassicalTime": {
                                    "type": "number"
                                },
                                "Metrics": {
                                    "type": "array",
                                    "description": "Additional defined metrics for this benchmark
    ",
                                    "items": {
                                        "type": "object",
                                        "properties": {
                                            "Metric": {
                                                "type": "string",
                                                "description": "Name of the metric"
                                            },
                                            "Value": {
                                                "type": "number",
                                                "description": "Value of the metric"
                                            },
           "STD": {
                                                "type": "number",
                                                "description": "Standard deviation"
                                            }
                                        },
                                        "required": [
                                            "Metric",
                                            "Value".
           "STD"
                                        ]
                                    }
                                }
                            },
                            "required": [
                                "NumberOfQubits",
                                "QubitPlacement",
                                "QPUs",
                                "CPUs",
                                "TotalTime",
                                "SigmaTotalTime",
                                "QuantumTime",
                                "SigmaQuantumTime",
                                "ClassicalTime",
                                "SigmaClassicalTime"
                            ]
                        }
                    }
                },
                "required": [
                    "BenchmarkKernel",
                    "StartTime",
                    "EndTime",
                    "ProgramLanguage",
                    "ProgramLanguageVersion",
                    "ProgramLanguageVendor",
                    "API",
                    "TimeMethod",
                    "Results"
                ]
            }
        }
    },
    "required": [
        "ReportOrganization",
        "MachineName",
        "QPUModel",
        "QPUDescription",
        "CPUModel",
        "Frequency",
        "Network",
        "QPUCPUConnection",
        "Benchmarks"
    ]
}
```

Listing A.1: JSON Schema for reporting results

B.Templates for NEASQC benchmark

As explained in the **NEASQC** deliverable (Gómez et al., 2021) and in the present document, each proposed **Kernel** and its correspondent **Benchmark Test Case**, **BTC**, must have a reference software implementation using Eviden myQLM environment[1]. In order to gather all these myQLM implementations for the NEASQC benchmark suite (**TNBS**), a repository named **WP3_Benchmark**, was created in the **NEASQC** GitHub. The different **Benchmark** implementations will be stored under the **tnbs** folder of this repository.

Additionally, **NEASQC** D3.2 deliverable sets up that each **Benchmark** execution should report their results in a separate JSON file, that must follow **NEASQC** json schema *NEASQC.Benchmark.V2.Schema.json* (included in Appendix A of present document). This json schema can be found, too, into the folder tnbs/templates of the **WP3_Benchmark** repository. So, when a **Benchmark** is implemented, the code needed for creating the **Benchmark** report must be included.

For simplifying this whole implementation process, a **Benchmark** developer, can use the script templates found in the folder tnbs/templates of the **WP3_Benchmark** repository:

- **my_benchmark_execution.py**
- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

The **my_benchmark_execution.py** script was designed to execute a complete **Benchmark** workflow. Section B.1 explains this script in great detail and provides different tips for helping the **Benchmark** developer.

The other four scripts were designed for gathering the **Benchmark** results, and the software and hardware info used for **Benchmark** execution, for creating the properly configured json report of the results. They are described in section B.2.

B.1. my_benchmark_execution.py

The main idea of this script is to implement a complete **Benchmark** workflow easily. The main parts of the script are:

- **KERNEL_BENCHMARK** python class
- **build_iterator** function
- **run_code** function.
- **compute_samples** function.
- **summarize_results** function

The **KERNEL_BENCHMARK** Python class defines the complete **Benchmark** workflow and its *exe* method will execute it. The other Python functions are called by the **KERNEL_BENCHMARK** class and contain the specific functionalities of a particular **Benchmark**. It is expected that the **Benchmark** developer modifies the Python functions but leaves the Python class unmodified.

B.1.1. KERNEL_BENCHMARK class

The **KERNEL_BENCHMARK** class defines a generic workflow for an execution of a given **Benchmark**. The workflow, in a high-level description, of the class is composed of the following steps:

1. *Pre-benchmark or warm up step*. The main idea is executing the **Benchmark** a fixed number of times for computing the **number of repetitions** the main part of **Benchmark** procedure should be executed.
2. *Compute sample step*. The obtained results from the *Pre-benchmark step* should be used for computing **number of repetitions** for assuring a desired statistical significance.

[1] See myQLM webpage

3. *Benchmark execution step*. The **Benchmark** should be executed a **number of repetitions** times and the **Benchmark** metrics should be collected for each repetition.
4. *Summarize results step*. The collected results from the *Benchmark execution step* are post-processed, following the indications of the corresponding **Benchmark** documentation, and stored in an output file.

Once the **KERNEL_BENCHMARK** class is properly configured and instantiated then its **exe** method will be called and the before workflow executed.

For executing a **Benchmark**, the class will execute the **run_code** function that must be modified by the **Benchmark** developer. It is expected that the **run_code** function returns a pandas DataFrame with the **Benchmark** metrics obtained from the execution (see section B.1.3 for more information about this function).

The output of the *Pre-benchmark step* execution will be a pandas DataFrame where each column will be the benchmark metrics and each row will have the results of one **Benchmark** execution.

The *Compute sample step* will execute the **compute_samples** function. The benchmark developer must modify this function for properly processing the results of the *Pre-benchmark step* and computing the **number of repetitions** (see section B.1.4 for more information).

In the *Benchmark execution step* the **run_code** function will be executed a determined **number of repetitions** (that can be computed using *Compute sample step* but can be provided as input, see below). The output of this part will be a pandas DataFrame where the columns are the desired metrics and each row stores the result of each repetition.

Finally in the *Summarize results step* the **summarize_results** function will be executed. The **Benchmark** developer must modify this function for processing the results from the different **Benchmark** executions according to its requirements (see section B.1.5 for more information).

In general, it is expected that for a particular **Benchmark** the complete workflow has to be executed for different configurations (for example different number of qubits). For dealing with the complete workflow, the **KERNEL_BENCHMARK** class executes a loop over an iterator that can be built into the *build_iterator* function, see B.1.2. The initial and the final time (for executing the complete workflow of the iterator) will be recorded and stored in a csv file.

The **Benchmark** workflow execution can be configured when the class is instantiated by giving it the typical Python *kwargs*. Following *keyword arguments* can be provided:

1. *pre_benchmark*: Boolean for executing or not the pre-benchmark step.
2. *pre_samples*: list of ints: Number of repetitions of the pre-benchmark step.
3. *pre_save*: Boolean for saving or not the metrics obtained during the pre-benchmark step.
4. *saving_folder*: string with the path of the folder where the different generated files will be stored.
5. *benchmark_times*: name for the file where the initial and final times of the benchmark will be stored (as a *csv* file).
6. *csv_results*: name for the file where the benchmark step results will be stored (as a *csv* file).
7. *summary_results*: name for the file where the summary benchmark results will be stored (as a *csv* file).
8. *alpha*: desired confidence interval for the benchmark results.
9. *min_mean*: number of minimum repetitions for benchmark step.
10. *max_mean*: number of maximum repetitions for benchmark step.
11. *list_of_qbits*: list of ints. Each element will be the number of used qubits for the benchmark.

In general, the following CSV files will be created during the complete benchmark workflow execution:

- Pre-benchmark files: during the pre-benchmark execution step each time the **run_code** function is called, the resulting DataFrame can be (or not) stored in the function of the input argument *pre_save*. The name of these files will be: *pre_benchmark_step_i.csv* where *i* will be the number of qubits.
- Benchmark csv file: during the benchmark step each time the **run_code** function is called the resulting DataFrame will be appended to a csv file, given by the input keyword argument *csv_results*.

- Summary csv file: when all the benchmark steps were done the results will be post-processed and a summary result DataFrame will be stored in a csv file given by the input keyword argument *summary_results*.
- Times csv files, with the initial and the final time of the complete benchmark execution. The file name will be given by the *benchmark_times* input keyword argument.

B.1.2. build_iterator

The **Benchmark** developer should modify this function with the correspondent software implementation of its **BTC** of the **Kernel**.

The main idea of this function is that the **Benchmark** developer creates the list with all the steps, that a complete benchmarking procedure should follow. It is expected that each element of the list be a Python tuple where all the mandatory arguments for executing a **Benchmark** step are provided.

B.1.3. run_code

The **Benchmark** developer should modify this function with the correspondent software implementation of its **BTC** of the **Kernel**.

The arguments of this function will be:

- *iterator_step*: tuple with all the mandatory inputs for executing a **Benchmark** step. The tuple will be an element of the list returned by the *build_iterator* function (B.1.2)
- *repetitions*: number of times the **Benchmark** will be executed.
- *stage_bench*: stage of the benchmark: can be *pre-benchmark* or *benchamrk*.
- *kwargs*: Python keyword arguments.

So the function should properly configure a **Benchmark** step (for the *pre benchmark* or the *benchmark* stage), execute it and post-process the results for getting the mandatory metrics. It is expected that the return of the function will be:

- pandas DataFrame with the **Benchmark** results. Each column of the DataFrame corresponds to a metric and each row will correspond to a repetition.
- a file name for storing results.

If the **Benchmark** developer needs more arguments, to configure or execute its **Benchmark**, can use the *kwargs* for passing to the function any desired argument.

B.1.4. compute_samples

The **Benchmark** developer should modify this function with the corresponding software implementation for the proposed particular **Benchmark**.

This function computes the **number of repetitions** for the *Benchmark execution step*. If the **Benchmark** developer wants the results of a complete **Benchmark** to have a desired statistical significance, it can use this function to codify the mandatory code for computing the **number of repetitions**.

The input of the **compute_samples** function is typical Python keyword arguments (*kwargs*). So the **Benchmark** developer can pass to the function the arguments needed for implementing the code. By default three following arguments are implemented in the skeleton of the function:

- *alpha*: for setting the desired confidence interval.
- *min_meas*: minimum number of repetitions.
- *max_meas*: maximum number of repetitions.

In general, it is expected that the computed number of repetitions is between *min_meas* and *max_meas*. Users can set these values to **None** to allow any value.

The **Benchmark** developer can use these arguments for their original purpose, change their functionality or even not use them. Even the **Benchmark** developer can define its own arguments and process them properly in the **compute_samples** function.

The output will be an integer number that will be the number of repetitions for a benchmark stage of a **Benchmark** step.

If the *pre-benchmark step* is enabled in the **KERNEL_BENCHMARK** class then, the corresponding metric results of this step are passed to the *compute_samples* function using the *keyword argument pre_metrics*

B.1.5. summarize_results

The **Benchmark** developer should modify this function with the corresponding software implementation for the proposed particular **Benchmark**.

In this function, the post-processing of the complete **Benchmark** results should be coded. The recommended way is to load the file where these results are stored and execute the corresponding **Benchmark** procedure post-processing for getting the final **Benchmark** results.

The name of the file with the results can be passed using the desired keyword arguments(*kwargs*)

B.2. Generating the benchmark report

When the complete **Benchmark** is executed and the results files are generated, a properly configured JSON report (following the **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json*) should be generated. This report contains information about the hardware, the software used in the **Benchmark** and the obtained results.

The following scripts from the **tnbs/templates** folder of the **WP3_Benchmark** repository can be used, by the **Benchmark** developer, for obtaining and properly formatting the mandatory info needed by the **Benchmark** report:

- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

B.2.1. my_environment_info.py

This script can be used as a template for gathering the info related to the hardware system used in the **Benchmark**. Table 2 shows the correspondence between the hardware information fields of the **NEASQC Benchmark** report and the function of the **my_environment_info.py** script.

field	function
ReportOrganization	my_organisation
MachineName	my_machine_name
QPUModel	my_qpu_model
QPUDescription	my_qpu
CPUModel	my_cpu_model
Frequency	my_frecuency
Network	my_network
QPUCPUConnection	my_QPUCPUConnection
Benchmarks	see Subsection B.2.2

*Table 2: Correspondence between the fields of the NEASQC benchmark report and the functions for implementing the information in the **my_environment_info.py** script.*

Benchmark developers can modify the different functions of Table 2 to adapt to their needs. Using as a template the Table 2 functions ensures that the collected info satisfies the **NEASQC** JSON schema.

B.2.2. my_benchmark_info.py

The functions implemented in the **my_benchmark_info.py** script can be used as templates for gathering the different info needed by the *Benchmarks* field of Table 2. This field has several sub-fields that gather information about the software and the compilers used in the **Benchmark** as well as the **Benchmark** results.

In Table 3 the correspondence between the different sub-fields of the *Benchmarks* field and the different functions implemented in the **my_benchmark_info.py** script is presented.

field	function
BenchmarkKernel	my_benchmark_kernel
StartTime	my_starttime
EndTime	my_endtime
ProgramLanguage	my_programlanguage
ProgramLanguageVersion	my_programlanguage_version
ProgramLanguageVendor	my_programlanguage_vendor
API	my_api
QuantumCompililation	my_quantum_compilation
ClassicalCompiler	my_classical_compilation
TimeMethod	my_timemethod
MetaData	my_metadata_info
Results	see subsection B.2.3

*Table 3: Correspondence between the fields of the **NEASQC** benchmark report and the functions for implementing the information in the **my_benchmark_info.py** script.*

Using as a template the Table 3 functions ensures that the collected info satisfies the **NEASQC** JSON schema.

The *MetaData* field is not a mandatory field in the **NEASQC** JSON schema. The main idea of this field is to gather additional information that is particular to the **Benchmark** and helps to keep traceability of the generated results.

B.2.3. my_benchmark_summary.py

The sub-field *Results* in Table 3 summarizes the obtained results of a **Benchmark** execution. These can be the different elapsed times and the verification metrics of a particular **Benchmark**. The function *summarize_results* from **my_benchmark_summary.py** script can be used by a **Benchmark** developer for gathering this information.

Table 4 shows the different sub-fields, and the information that they gather, from the *Results* sub-field.

sub-field	information
NumberOfQubits	number of qubits, n
TotalTime	mean of **elapsed time**
SigmaTotalTime	standard deviation of **elapsed time**
QuantumTime	mean of the **quantum time**
SigmaQuantumTime	standard deviation of **quantum time**
ClassicalTime	mean of the **classical time**
SigmaClassicalTime	standard deviation of **classical time**
Metrics	Several Info about verification metrics

*Table 4: Sub-fields of the Results fields of the **TNBS** benchmark report.*

The sub-field *Metrics*, last line of Table 4, gathers information about the different validation metrics used by the particular **Benchmark** case. Table 5 shows the different sub-fields of the *Metrics* one and the information that summarizes.

B.2.4. neasqc_benchmark.py

Finally, the *neasqc_benchmark.py* script implements the Python class **BENCHMARK**. The *exe* method of this class will create the final **NEASQC** benchmark report following the mandatory JSON scheme.

sub-field	information
metric	Name of different validation metrics
Value	mean value of the metric
STD	standard deviation of the metric
Count	number of samples for computing the statistics of the metric

*Table 5: Sub-fields of the Metrics field. For each validation metric of the **Benchmark**, all the information presented in the Table should be provided.*

The *exe* method needs as input a complete Python dictionary where each key corresponds to a main field of the report and the value will be the corresponding information in a suitable format. The different functions from *my_environment_info* and from *my_benchmark_info* can be used for creating this dictionary.

Listing B.1 shows the final part of the *neasqc_benchmark.py* script. The complete Python dictionary needed by the *exe* method is shown. As can be seen, each key of this dictionary invokes the function that provides the mandatory field information.

```python

    if __name__ == "__main__":

        import my_environment_info
        import my_benchmark_info

        ################## Configuration ##########################

        kwargs = {"None": None}

        benchmark_conf = {
            "ReportOrganization": my_environment_info.my_organisation(
                **kwargs),
            "MachineName": my_environment_info.my_machine_name(**kwargs),
            "QPUModel": my_environment_info.my_qpu_model(**kwargs),
            "QPUDescription": my_environment_info.my_qpu(**kwargs),
            "CPUModel": my_environment_info.my_cpu_model(**kwargs),
            "Frequency": my_environment_info.my_frecuency(**kwargs),
            "Network": my_environment_info.my_network(**kwargs),
            "QPUCPUConnection":my_environment_info.my_QPUCPUConnection(
                **kwargs),
            "Benchmarks": my_benchmark_info.my_benchmark_info(**kwargs),
            "json_file_name": "./benchmark_report.json"
        }

        benchmark = BENCHMARK()
        benchmark.exe(benchmark_conf)
```

*Listing B.1: Final part of the **neasqc_benchmark.py***

C.T01: Benchmark for Probability Loading Algorithms document

NExt ApplicationS of Quantum Computing
Benchmark Suite

T01: Benchmark for Probability Loading Algorithms

Document Properties

Contract Number	951821
Contractual Deadline	31/10/2023
Dissemination Level	Public
Nature	Test Case Definition
Editors	Gonzalo Ferro, CESGA
Authors	Gonzalo Ferro, CESGA Andrés Gómez, CESGA Diego Andrade, CITIC-UDC
Reviewers	Cyril Allouche, EVIDEN Arnaud Gazda, EVIDEN
Date	27/10/2023
Category	Generic
Keywords	
Status	Submitted
Release	1.0

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, Organisation	Description of Changes
0.1	04/01/2023	Gonzalo Ferro, CESGA; Andrés Gómez, CESGA	First version
0.2	26/01/2023	Gonzalo Ferro, CESGA Andrés Gómez, CESGA Diego Andrade, CITIC-UDC	Reordering sections and rewording
0.3	07/02/2023	Gonzalo Ferro, CESGA	Complete Test Case Documentation
0.4	18/08/2023	Gonzalo Ferro, CESGA	Corrections for the Test Case Documentation
0.5	08/10/2023	Andrés Gómez, CESGA	Formatting
1.0	24/10/2023	Gonzalo Ferro, CESGA	Fixing the naming according to the Glossary of deliverable 3.5

Table of Contents

1.Introduction

This document describes the T1: Probability Loading benchmark of The NEASQC Benchmarking Suite (**TNBS**). This document must be read accompanied by the document that describes the TNBS: D3.5: The NEASQC Benchmark Suite.

Section 2 describes the **Probability Loading Kernel**, referred to as **PL Kernel** along the document. With each **TNBS Kernel**, a **Benchmark Test Case** (**BTC**) must be designed and documented. This is done in Section 3. Finally, the benchmarking methodology provides a reference implementation of the **Benchmark** using the Eviden myQLM library. A complete documentation of this implementation is provided in Annex A.

2.Description of the kernel: Probability Loading

The **PL Kernel** encodes a probability distribution in a quantum circuit. Section 2.1 justifies the **Kernel** selection, and Section 2.2 describes the **Kernel** following the form indicated by the suite.

2.1. Kernel selection justification

The **PL Kernel** is common to many different quantum algorithms like the **HHL** (Harrow et al., 2009), quantum **PCA** (Lloyd et al., 2014), quantum amplitude estimation algorithms (Brassard et al., 2002) etc. This initialization step is, usually, a very demanding part of any quantum algorithm because its number of operations typically scales as $\sim 2^n$, being n the number of qubits to be initialized. In addition, this **Kernel** meets the three main requirements from the **NEASQC** benchmark methodology:

1. The **Kernel** can be described mathematically or procedurally. Using this description, a standalone circuit can be generated (see section 2.2).
2. The **Kernel** can be defined for different numbers of qubits.
3. The output can be verified with a classical computation (in the proposed **BTC**, see section 3.2, the result is known *a priori*)

For all these reasons, the **PL Kernel** is a good candidate for the **TNBS**.

2.2. Kernel Description

The **PL Kernel** can be defined, mathematically as follows:

Let $\mathbf{V}$ be a normalised vector of complex values:

$$\mathbf{V} = \{v_0, v_1, \cdot, v_{2^n-1}\}, v_i \in \mathbb{C} \tag{T01.2.1}$$

such that

$$\sum_{i=0}^{2^n-1} |v_i|^2 = 1 \tag{T01.2.2}$$

The main task of the **PL Kernel** is the creation of an operator $\mathbf{U}$, from the normalised vector $\mathbf{V}$, which satisfies equation (T01.2.3):

$$\mathbf{U}|0\rangle_n = \sum_{i=0}^{2^n-1} v_i|i\rangle_n \tag{T01.2.3}$$

This procedure can be used for the loading of a probability density function (**PDF**). In this case, equation (T01.2.1) can be reformulated as (T01.2.4)

$$\mathbf{P} = \{p_0, p_1, \cdot, p_{2^n-1}\}, p_i \in [0, 1] \tag{T01.2.4}$$

Equation (T01.2.2) must be transformed into equation (T01.2.5)

$$\sum_{i=0}^{2^n-1} |p_i|^2 = 1 \tag{T01.2.5}$$

And (T01.2.3) can be written as (T01.2.6):

$$\mathbf{U}|0\rangle_n = \sum_{i=0}^{2^n-1} \sqrt{p_i}|i\rangle_n \tag{T01.2.6}$$

The **BTC** for the **PL Kernel** developed in this document is based on this particular case.

Note: The **PL Kernel** definition is agnostic about the implementation of the loading operator **U**. The **Kernel** only provides conditions about the mandatory input vector **P** and about the behaviour of the operator **U**. Different algorithms and procedure approaches can be used for constructing such an operator. This operator implementation agnosticism will be kept in the design of the correspondent **BTC** so it can be used as a methodology for evaluating different algorithms or procedures for data loading into quantum circuits.

3.Description of the benchmark test case

This section presents the complete description of the **BTC** for the **PL Kernel**. Section 3.1 describes the problem addressed by the test case. Section 3.2 provides a high-level description of the case. Section 3.3 provides the execution workflow. Finally, section 3.4 documents how the results of such executions must be reported.

3.1. Description of the problem

The loading of a fixed normal probability distribution function, **PDF**, $N_{\mu,\sigma}(x)$, into a quantum circuit is the **BTC** associated to the **PL Kernel**. An operator $\mathbf{U}$ for loading this normal **PDF** must be built using a *probability loading* algorithm, and the probabilities of the different possible final states must be measured and compared with the original normal **PDF**.

Finally, the verification of the output can be done by comparing the obtained measurements with the original **PDF**, using several metrics.

3.2. Benchmark test case description

This section introduces a detailed step-by-step workflow of the **BTC**. Given a number of qubits, n, and using a specific input *probability loading* algorithm, the test case must follow the following steps:

1. Take a random uniform distribution with a particular mean, $\tilde{\mu}$ and standard deviation, $\tilde{\sigma}$, selected within the following ranges:
    - $\tilde{\mu} \in [-2, 2]$
    - $\tilde{\sigma} \in [0.1, 2]$
2. So the normal **PDF** is: $N_{\tilde{\mu},\tilde{\sigma}}(x)$
3. Create an array of 2^n values: $\mathbf{x} = \{x_0, x_1, x_2, \cdots, x_{2^n-1}\}$ where:
    - x_0 such that
    $$\int_{-\infty}^{x_0} N_{\tilde{\mu},\tilde{\sigma}}(x)dx = 0.05$$
    - x_{2^n-1} such that
    $$\int_{-\infty}^{x_{2^n-1}} N_{\tilde{\mu},\tilde{\sigma}}(x)dx = 0.95$$
    - $x_{i+1} = x_i + \Delta x$
    - $\Delta x = \frac{x_{2^n-1}-x_0}{2^n}$
4. Create a 2^n values array, $\mathbf{P}$ from $\mathbf{x}$ by:
$$\mathbf{P}(\mathbf{x}) = \{P(x_0), P(x_1), \cdots, P(x_{2^n-1})\} = \{N_{\tilde{\mu},\tilde{\sigma}}(x_0), N_{\tilde{\mu},\tilde{\sigma}}(x_1), \cdots, N_{\tilde{\mu},\tilde{\sigma}}(x_{2^n-1})\}$$
5. Normalize the $\mathbf{P}$ array:
$$\mathbf{P_{norm}}(\mathbf{x}) = \{P_{norm}(x_0), P_{norm}(x_1), \cdots, P_{norm}(x_{2^n-1})\}$$
where
$$P_{norm}(x_i) = \frac{P(x_i)}{\sum_{j=0}^{2^n-1} P(x_j)}$$
6. Compute the number of shots n_{shots} as:
$$n_{shots} = \min(10^6, \frac{100}{\min(\mathbf{P_{norm}}(\mathbf{x}))})$$

7. Use the $\mathbf{P_{norm}}(\mathbf{x})$ array as input of the particular *probability loading* algorithm for creating the $\mathbf{U}$ operator such that :
$$\mathbf{U}|0\rangle_n = \sum_{i=0}^{2^n-1} \sqrt{P_{norm}(x_i)}|i\rangle_n \tag{T01.2.1}$$
8. Execute the quantum program $\mathbf{U}|0\rangle_n$ and measure all the n qubits a number of times n_{shots}. Store the number of times each state $|i\rangle_n$ is obtained, m_i, and compute the probability of obtaining it as
$$Q_i = \frac{m_i}{n_{shots}} \forall i = \{0, 1, \cdots, 2^n - 1\}$$
9. With the measured array $\mathbf{Q} = \{Q_i\} \forall i = \{0, 1, \cdots, 2^n - 1\}$ and the initial normalised array $\mathbf{P_{norm}}$ compute following metrics:
   - The Kolmogorov-Smirnov (*KS*) between $\mathbf{Q}$ and $\mathbf{P_{norm}}$. This is the maximum of the absolute difference between the cumulative distribution functions of $\mathbf{P_{norm}}$ and $\mathbf{Q}$:
$$KS = \max\left(\left| \sum_{j=0}^{i} P_{norm}(x_j) - \sum_{j=0}^{i} Q_j \right|, \forall i = 0, 1, \cdots, 2^n - 1 \right)$$
   - The Kullback-Leibler divergence (*KS*) is defined as:
$$KL(\mathbf{Q}/\mathbf{P_{norm}}) = P_{norm}(x_j) \ln \frac{P_{norm}(x_j)}{\max(\epsilon, Q_k)}$$
   where $\epsilon = \min(P_{norm}(x_j)) * 10^{-5}$ which guarantees the logarithm exists when $Q_k = 0$
10. Execute a χ^2 test using $n_{shots}\mathbf{Q}$ and $n_{shots}\mathbf{P_{norm}}$ and get its p-value (using as null hypothesis that both sets are equal). If the p-value is lower than 0.05 then the obtained result should be considered invalid.

Additionally, the time from steps 1 to 10 is measured as the **elapsed time**. If possible, the time of the quantum part, step 8, should be measured separately as the **quantum time**.

3.3. Complete benchmark procedure

To execute a complete **PL Benchmark** the next procedure must be followed:

1. We must select in advance the set of the number of qubits to be tested (for example from n=4 to n=8).
2. For each number of qubits the following steps must be performed:
   a) Execute a warm-up step consisting of:
      i. Execute 10 iterations of the **BTC**, section 3.2, and compute the mean and the standard deviation for the **elapsed time**, $(\mu_T, \ \sigma_T)$, and for the standard deviation for the *KS* and *KL* metrics, σ_m with $m = \{KS, KL\}$
      ii. Compute the number of repetitions mandatory, M_T, for having a relative error for the **elapsed time** of 5%, $r_T = 0.05$, with a confidence level of 95%, $\alpha = 0.05$, following (T01.3.2), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for α :
$$M_T = (\frac{\sigma_T Z_{1-\frac{\alpha}{2}}}{r \mu_T})^2 \tag{T01.3.2}$$
      iii. Compute the number of repetitions mandatory, M_m with $m = \{KS, KL\}$, for having an absolute error of $\epsilon_m = 10^{-4}$, for KS and KL metrics with a confidence level of 95%, $\alpha = 0.05$, following (T01.3.3)
$$M_m = (\frac{\sigma_m Z_{1-\frac{\alpha}{2}}}{\epsilon_m})^2 \tag{T01.3.3}$$
   b) Execute the complete **BTC**, section 3.2, $M = \max(M_T, M_{KS}, M_{KL})$ times. M must be greater than 5.

c) Compute the mean and the standard deviation for the **elapsed time**, **quantum time**, if possible, and for the mentioned metrics in steps 9 and 10 of section 3.2: χ^2, *KS* and *KL*.

3. If the verification χ^2 test fails (the p-value is lower than 0.05), the process must be stopped.

The method used to calculate the number of repetitions M in the previous procedure guarantees that the **elapsed time** will have a relative error lower than 5% and the KS and the KL metrics will have an absolute error lower than 10^{-4} with a confidence level of 95%.

3.4. Benchmark report

Finally, the results of the complete benchmark execution must be reported for each of the tested numbers of qubits into a valid JSON file following the JSON schema *NEASQC.Benchmark.V2.Schema.json* provided in the document D3.5: The NEASQC Benchmark Suite of the NEASQC project. The mean elapsed time must be reported in the *TotalTime* field of the JSON and its standard deviation in the *SigmaTotalTime* field.

The verification metrics of the **PL Benchmark** should be stored under the field *Benchmarks* into the sub-field *Results* and inside the sub-field *Metrics* of the JSON **Benchmark** report. The **Kolmogorov-Smirnov** metric is stored under the name *KS*, the **Kullback-Leibler** divergence is stored under the name *KL*, the χ^2 under the name *chi2* and the **p-value** under the name *p-value*.

List of Acronyms

Term	Definition
BTC	Benchmark Test Case
NEASQC	NExt ApplicationS of Quantum Computing
PDF	Probability Density Function
PL	Probability Loading
QPU	Quantum Process Unit
TNBS	The **NEASQC** Benchmark Suite

Table T01.1: *Acronyms and Abbreviations*

List of Figures

List of Tables

List of Listings

Bibliography

Brassard, G., Høyer, P., Mosca, M., & Tapp, A. (2002). Quantum amplitude amplification and estimation. https://doi.org/10.1090/conm/305/05215

Grover, L., & Rudolph, T. (2002). Creating superpositions that correspond to efficiently integrable probability distributions. arXiv e-prints. https://doi.org/10.48550/arXiv.quant-ph/0208112

Harrow, A. W., Hassidim, A., & Lloyd, S. (2009). Quantum algorithm for linear systems of equations. Physical Review Letters, 103(15). https://doi.org/10.1103/physrevlett.103.150502

Lloyd, S., Mohseni, M., & Rebentrost, P. (2014). Quantum principal component analysis. Nature Physics, 10(9), 631–633. https://doi.org/10.1038/nphys3029

Shende, V., Bullock, S., & Markov, I. (2006). Synthesis of quantum-logic circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25(6), 1000–1010. https://doi.org/10.1109/tcad.2005.855930

A.NEASQC test case reference

As pointed out in deliverable **D3.5: The NEASQC Benchmark Suite** each proposed **Benchmark** for **TNBS**, must have a complete Eviden myQLM-compatible software implementation. For the **Benchmark** of the **PL Kernel**, this implementation can be found in the **tnbs/BTC_01_PL** folder of the **WP3_Benchmark** **NEASQC** GitHub repository. Additionally, the execution of a **Benchmark** must generate a complete result report into a separate JSON file, that must follow **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable.

Inside **tnbs/BTC_01_PL** following folders and files are presented:

- **PL** folder: contains the probability loading Python library, **PL** library from now, with all the mandatory code for executing a complete workflow of the **PL Kernel BTC** as explained in section 3.2
- **my_benchmark_execution.py**
- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

The modules and libraries inside **PL** folder in addition to the **my_benchmark_execution.py** deal with the **PL Benchmark** execution. Section A.1 documents these files. The other script files are related to **Benchmark** report generation and are properly explained in section A.2.

A.1. NEASQC implementation of benchmark test case.

This section presents a complete description of the **PL Benchmark** of the **TNBS**. Meanwhile the subsection A.1.1 documents Python implementation of a **BTC** step, see section 3.2, the subsection A.1.2 explains how to execute a complete **Benchmark** procedure, as explained in section 3.3, using script **my_benchmark_execution.py**.

A.1.1. The PL library

The **PL** library inside the **tnbs/BTC_01_PL/PL** allows executing the **BTC**, as explained in section 3.2. The following folder and files can be found inside it:

- *load_probabilities.py* package.
- *data_loading.py* package
- *utils* module

load_probabilities.py package

In this script the **BTC**, as explained in the section 3.2, is implemented as a Python class called *LoadProbabilityDensity*. Listing T01.1 shows the complete class implementation.

```python
class LoadProbabilityDensity:
    """
    Probability Loading
    """


    def __init__(self, **kwargs):
        """

        Method for initializing the class

        """

        self.n_qbits = kwargs.get("number_of_qbits", None)
        if self.n_qbits is None:
            error_text = "The number_of_qbits argument CAN NOT BE NONE."
```

```
                raise ValueError(error_text)
            self.load_method = kwargs.get("load_method", None)
            if self.load_method is None:
                error_text = "The load_method argument CAN NOT BE NONE."\
                    "Select between: multiplexor, brute_force or KPTree"
                raise ValueError(error_text)
            # Set the QPU to use
            self.qpu = kwargs.get("qpu", None)
            if self.qpu is None:
                error_text = "Please provide a QPU."
                raise ValueError(error_text)

            self.data = None
            self.p_gate = None
            self.result = None
            self.circuit = None
            self.quantum_time = None
            self.elapsed_time = None
            #Distribution related attributes
            self.x_ = None
            self.data = None
            self.mean = None
            self.sigma = None
            self.step = None
            self.shots = None
            self.dist = None
            #Metric stuff
            self.ks = None
            self.kl = None
            self.chi2 = None
            self.fidelity = None
            self.pvalue = None
            self.pdf = None
            self.observed_frecuency = None
            self.expected_frecuency = None

        def get_quantum_pdf(self):
            """
            Computing quantum probability density function
            """
            self.result, self.circuit, self.quantum_time = get_qlm_probability(
                self.data, self.load_method, self.shots, self.qpu)

        def get_theoric_pdf(self):
            """
            Computing theoretical probability densitiy function
            """
            self.x_, self.data, self.mean, self.sigma, \
                self.step, self.shots, self.dist = get_theoric_probability(self.n_qbits)

        def get_metrics(self):
            """
            Computing Metrics
            """
            #Kolmogorov-Smirnov
            self.ks = np.abs(
                self.result["Probability"].cumsum() - self.data.cumsum()
            ).max()
            #Kullback-Leibler divergence
            epsilon = self.data.min() * 1.0e-5
            self.kl = entropy(
                self.data,
                np.maximum(epsilon, self.result["Probability"])
            )
            #Fidelity
            self.fidelity = self.result["Probability"] @ self.data / \
                (np.linalg.norm(self.result["Probability"]) * \
                np.linalg.norm(self.data))

            #Chi square
            self.observed_frecuency = np.round(
                self.result["Probability"] * self.shots, decimals=0)
            self.expected_frecuency = np.round(
                self.data * self.shots, decimals=0)
            try:
                self.chi2, self.pvalue = chisquare(
                    f_obs=self.observed_frecuency,
                    f_exp=self.expected_frecuency
                )
            except ValueError:
                self.chi2 = np.sum(
                    (self.observed_frecuency - self.expected_frecuency) **2 / \
                        self.expected_frecuency
                )
                count = len(self.observed_frecuency)
                self.pvalue = chi2.sf(self.chi2, count -1)

        def exe(self):
            """
            Execution of workflow
            """
            #Create the distribution for loading
            tick = time.time()
            self.get_theoric_pdf()
            #Execute the quantum program
            self.get_quantum_pdf()
            self.get_metrics()
            tack = time.time()
            self.elapsed_time = tack - tick
            self.summary()

        def summary(self):
            """
            Pandas summary
            """
            self.pdf = pd.DataFrame()
            self.pdf["n_qbits"] = [self.n_qbits]
            self.pdf["load_method"] = [self.load_method]
            self.pdf["qpu"] = [self.qpu]
            self.pdf["mean"] = [self.mean]
            self.pdf["sigma"] = [self.sigma]
            self.pdf["step"] = [self.step]
            self.pdf["shots"] = [self.shots]
            self.pdf["KS"] = [self.ks]
            self.pdf["KL"] = [self.kl]
            self.pdf["fidelity"] = [self.fidelity]
            self.pdf["chi2"] = [self.chi2]
            self.pdf["p_value"] = [self.pvalue]
            self.pdf["elapsed_time"] = [self.elapsed_time]
            self.pdf["quantum_time"] = [self.quantum_time]
```

*Listing T01.1: LoadProbabilityDensity class from **load_probabilities.py** script*

For instantiating this *LoadProbabilityDensity* a typical python *kwargs* argument should be provided. Two following keyword arguments are mandatory:

- **n_qbits**: integer with the number of qubits used for loading probability distribution.
- **method**: a string for selecting different **Probability Loading** algorithms. Valid inputs are:
  - *brute_force*: for using an Atos *myqlm* implementation of the original probability loading algorithm (Grover & Rudolph, 2002). The controlled rotations needed by the algorithm were implemented straightforwardly. Figure T01.1 shows circuit implementation of a 3-qubit probability distribution using this option.
  - *mutiplexor*: The mandatory controlled rotations of the original loading algorithm were implemented using *quantum multiplexors* (Shende et al., 2006). This implementation is more efficient and compact than *brute_force* as can be shown in Figure T01.2 where the circuit implementation using this method, for the same probability distribution of the Figure T01.1, is presented.
  - *KPTree*: QLM implementation of a probability loading [1]

The *exe* method executes a complete **BTC**, as explained in the section 3.2, and fills different attributes of the class. Most important is the *pdf* one which is a pandas DataFrame where all the configuration and the correspondent metrics

[1] See myQLM KPTree class

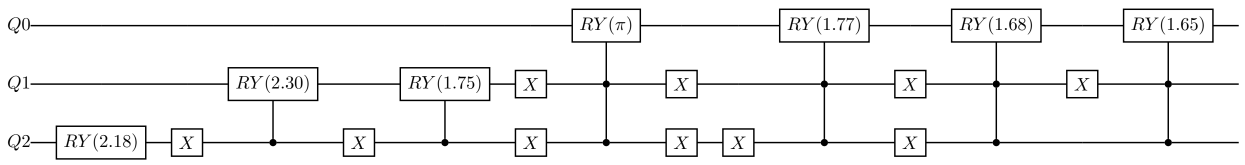


Figure T01.1: Circuit implementation for brute_force method.

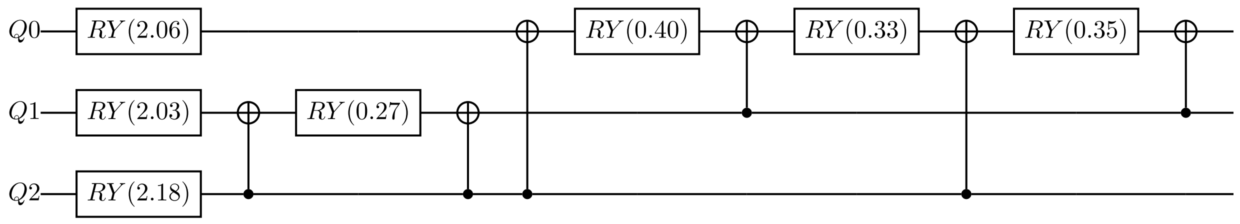


Figure T01.2: Circuit implementation for multiplexor. The same probability density function that used for Figure T01.1

results are stored. In table T01.2 an example of this attribute is shown. The **mean** and the **sigma** columns are the mean and the standard deviation of normal distribution used, step 1 of the section 3.2. The **step** column corresponds to the Δx as presented in step 3 of section 3.2.

	n_qbits	load_method	qpu	mean	sigma	step	shots	KS	KL	chi2	p_value	elapsed_time	quantum_time
0	4	KPTree	CLinalg	0.751392	1.247861	0.428570	20020	0.006791	0.000692	28.046995	0.021277	0.215811	0.210783

Table T01.2: Example of the pdf attribute from the LoadProbabilityDensity

The **load_probabilities.py** script can be executed from the command line. Different arguments can be provided to properly configure the **PL** algorithm and the **BTC**. For a usage guide, *-h* parameter can be provided. Listing T01.2 shows an example of how to use **load_probabilities.py** from the command line. In this case, the number of qubits for loading the **PDF** will be 6 and the *multiplexors* method will be used for building the correspondent operator.

```
    python load_probabilities.py -n_qbits 6 -method multiplexor
```

*Listing T01.2: Example of use the **load_probabilities.py** script from command line.*

data_loading.py package

The *data_loading.py* package contains all the auxiliary functions needed by the *LoadProbabilityDensity* of the *data_loading.py* package. All the functions of this module were obtained from **QQuantLib** library from **NEASQC Financial Applications** software package.

Main functions of this package, shown in Listing T01.3, are *get_theoric_probability*, that builds the theoretical probability distribution $\mathbf{P_{norm}}(\mathbf{x})$, and *get_qlm_probability* that deals with the building and execution of the mandatory quantum circuit (using Atos *myqlm*) for getting the measured probability distribution **Q**.

```
def get_theoric_probability(n_qbits: int) -> (np.ndarray, np.ndarray, float, float, float, int):
    """
    Get the discretization of the PDF for N qubits
    """
    mean = random.uniform(-2., 2.)
    sigma = random.uniform(0.1, 2.)

    intervals = 2 ** n_qbits

    ppf_min = 0.005
    ppf_max = 0.995
    norma = norm(loc=mean, scale=sigma)
    x_ = np.linspace(norma.ppf(ppf_min), norma.ppf(ppf_max), num=intervals)
    step = x_[1] - x_[0]

    data = norma.pdf(x_)
    data = data/np.sum(data)
    mindata = np.min(data)
    shots = min(1000000, max(10000, round(100/mindata)))
    #data = np.sqrt(data)
    return x_, data, mean, sigma, float(step), shots, norma

def get_qlm_probability(data, load_method, shots, qpu):
    """
    executing quantum stuff
    """
    if load_method == "multiplexor":
        p_gate = load_probability(data, method="multiplexor")
    elif load_method == "brute_force":
        p_gate = load_probability(data, method="brute_force")
    elif load_method == "KPTree":
        p_gate = KPTree(np.sqrt(data)).get_routine()
    else:
        error_text = "Not valid load_method argument."\
            "Select between: multiplexor, brute_force or KPTree"
        raise ValueError(error_text)
    tick = time.time()
    result, circuit, _, _ = get_results(
        p_gate,
        linalg_qpu=qpu,
        shots=shots
    )
    tack = time.time()
    quantum_time = tack - tick

    if load_method == "KPTree":
        #Use different order convention
        result.sort_values(by="Int", inplace=True)
    return result, circuit, quantum_time
```

Listing T01.3: Functions get_theoric_probability and get_qlm_probability from data_loading.py package. script

utils module

The *utils* packages contain all mandatory auxiliary functions needed by the *data_loading.py* package. All the functions of this module were obtained from **QQuantLib** library from **NEASQC Financial Applications** software package.

A.1.2. my_benchmark_execution.py

This script is a modification of the correspondent template script located in tnbs/templates folder of the **WP3_Benchmark** repository. Following the recommendations of Annex B of the deliverable **D3.5: The NEASQC Benchmark Suite** the **run_code**, **compute_samples**, **summarize_results** and the **build_iterator** functions were modified. Meanwhile, the **KERNEL_BENCHMARK** class was not modified. In the following sections, the software adaptations for the **PL Benchmark** are presented.

run_code

Listing T01.4 shows the modifications performed into the **run_code** function for the **PL Benchmark**. The main functionality is executing a **BTC** (section 3.2) for a fixed number of qubits (*n_qbits*), that is provided as the first element of the Python tuple *iterator_step*, an input number of times (*repetitions*), and gathering all the mandatory metrics obtained. The *stage_bench* is a boolean variable that indicates if the step is executed in the pre-benchmark (step *2.a* in section 3.3) or in the benchmark stage (step *2.b* in section 3.3). As can be seen, the *LoadProbabilityDensity* class, listing T01.1, and its *exe* method is used for doing the different executions of the **BTC**.

```
    def run_code(iterator_step, repetitions, stage_bench, **kwargs):
        """
        For configuration and execution of the benchmark kernel.

        Parameters
        ----------

        iterator_step : tuple
            tuple with elements from iterator built from build_iterator.
        repetitions : list
            number of repetitions for each execution
        stage_bench : str
            benchmark stage. Only: benchmark, pre-benchamrk
        kwargs : keyword arguments
            for configuration of the benchmark kernel

        Returns
        _______

        metrics : pandas DataFrame
            DataFrame with the desired metrics obtained for the
            integral computation
        save_name : string
            Desired name for saving the results of the execution

        """
        from PL.load_probabilities import LoadProbabilityDensity, get_qpu
        #if n_qbits is None:
        #    raise ValueError("n_qbits CAN NOT BE None")

        if stage_bench not in ['benchmark', 'pre-benchmark']:
            raise ValueError(
                "Valid values for stage_bench: benchmark or pre-benchmark'")

        if repetitions is None:
            raise ValueError("repetitions CAN NOT BE None")

        #Here the code for configuring and execute the benchmark kernel
        kernel_configuration = deepcopy(kwargs.get("kernel_configuration", None))
        if kernel_configuration is None:
            raise ValueError("kernel_configuration can not be None")

        # Here we built the dictionary for the LoadProbabilityDensity class
        n_qbits = iterator_step[0]
        list_of_metrics = []
        kernel_configuration.update({"number_of_qbits": n_qbits})
        kernel_configuration.update({"qpu": get_qpu(kernel_configuration['qpu'])})
        print(kernel_configuration)
        for i in range(repetitions):
            prob_dens = LoadProbabilityDensity(**kernel_configuration)
            prob_dens.exe()
            list_of_metrics.append(prob_dens.pdf)
        metrics = pd.concat(list_of_metrics)
        metrics.reset_index(drop=True, inplace=True)

        if stage_bench == 'pre-benchmark':
            # Name for storing Pre-Benchmark results
            save_name = "pre_benchmark_step_{}.csv".format(n_qbits)
        if stage_bench == 'benchmark':
            # Name for storing Benchmark results
            save_name = kwargs.get('csv_results')
            #save_name = "pre_benchmark_step_{}.csv".format(n_qbits)
        return metrics, save_name
```

*Listing T01.4: run_code function for **BTC** of the **PL Kernel***

compute_samples

Listing T01.5 shows the implementation of the **compute_samples** function for the **PL Benchmark**. The main objective is to codify a strategy for computing the number of times, the **BTC** should be executed, to get some desired statistical significance for the different metrics (see *2.a.i* and *2.a.ii* of section 3.3). This function would implement equations (T01.3.2) and (T01.3.3) and compute the corresponding maximum as explained in *2.b* of section 3.3.

```
def compute_samples(**kwargs):
    """
    This functions computes the number of executions of the benchmark
    for assure an error r with a confidence of alpha

    Parameters
    ----------

    kwargs : keyword arguments
        For configuring the sampling computation

    Returns
    _______

    samples : pandas DataFrame
        DataFrame with the number of executions for each integration interval

    """
    from scipy.stats import norm

    #Configuration for sampling computations

    #Desired Confidence level
    alpha = kwargs.get("alpha", None)
    if alpha is None:
        alpha = 0.05
    zalpha = norm.ppf(1-(alpha/2)) # 95% of confidence level

    #geting the metrics from pre-benchmark step
    metrics = kwargs.get("pre_metrics", None)
    bench_conf = kwargs.get('kernel_configuration')

    #Code for computing the number of samples for getting the desired
    #statististical significance. Depends on benchmark kernel

    #Desired Relative Error for the elapsed Time
    relative_error = bench_conf.get("relative_error", None)
    if relative_error is None:
        relative_error = 0.05
    # Compute samples for Elapsed Time
    samples_t = (zalpha * metrics[['elapsed_time']].std() / \
        (relative_error * metrics[['elapsed_time']].mean()))**2

    #Desired Absolute Error for KS and KL metrics
    absolute_error = bench_conf.get("absolute_error", None)
    if absolute_error is None:
        absolute_error = 1e-4
    std_metrics = metrics[['KS', 'KL']].std()
    samples_m = (zalpha * std_metrics / absolute_error) ** 2

    #Maximum number of sampls will be used
    samples_ = pd.Series(pd.concat([samples_t, samples_m]).max())

    #Apply lower and higher limits to samples
    #Minimum and Maximum number of samples
    min_meas = kwargs.get("min_meas", None)
    if min_meas is None:
        min_meas = 5
    max_meas = kwargs.get("max_meas", None)
    samples_.clip(upper=max_meas, lower=min_meas, inplace=True)
    samples_ = samples_.max().astype(int)

    #If user wants limit the number of samples
    return samples_
```

*Listing T01.5: compute_samples function for codifying the strategy for computing the number of repetitions for the **PL Benchmark**.*

summarize_results

Listing T01.6 shows the implementation of the **summarize_results** function for the **PL Benchmark**. The main objective is post-processing the results of the complete **Benchmark** execution, as described in step *2.c* of section 3.3.

This function expects that the results of the complete benchmark execution have been stored in a *csv* file. The function loads this file into a pandas DataFrame that is post-processed properly.

```python

def summarize_results(**kwargs):
    """
    Create summary with statistics
    """

    folder = kwargs.get("saving_folder")
    csv_results = folder + kwargs.get("csv_results")
    #Code for summarize the benchamark results. Depending of the
    #kernel of the benchmark

    pdf = pd.read_csv(csv_results, index_col=0, sep=";")
    pdf["classic_time"] = pdf["elapsed_time"] - pdf["quantum_time"]
    pdf = pdf[
        ["n_qbits", "load_method", "KS", "KL", "chi2",
        "p_value", "elapsed_time", "quantum_time"]
    ]
    results = pdf.groupby(["load_method", "n_qbits"]).agg(
        ["mean", "std", "count"])

    return results
```

Listing T01.6: summarize_results function for summarizing the results from ***PL Benchmark***

build_iterator

Listing T01.7 shows the implementation of the **build_iterator** function for the **PL Benchmark**. The main objective is to create a Python iterator for executing the desired complete **BTC**. In this case, the iterator creates a list with all the desired number of qubits, n, that want to be benchmarked.

```python

def build_iterator(**kwargs):
    """
    For building the iterator of the benchmark
    """

    iterator = [tuple([i]) for i in kwargs['list_of_qbits']]
    return iterator
```

Listing T01.7: build_iterator function for creating the iterator of the complete execution of the ***PL Benchmark***

KERNEL_BENCHMARK class

No modifications were made to the **KERNEL_BENCHMARK** class. This Python class defines the complete benchmark workflow, section 3.3, and its *exe* method executes it properly by calling the correspondent functions (*run_code*, *compute_samples*, *summarize_results* and *build_iterator*). Each time the **Benchmark** is executed, as defined in section 3.3, the result is stored in a given *CSV* file.

The only mandatory modification is configuring properly the input keyword arguments, at the end of the **my_benchmark_execution.py** script. These parameters will configure the **PL** algorithm, the complete benchmark workflow and additional options (as the name of the *CSV* files).

Listing T01.8 shows an example for configuring an execution of a **Benchmark**. In this case, the *multiplexor* method will be used for creating the loading operator. Additionally, the number of qubits to benchmark will be 4, 6, 8 and 10.

```python
if __name__ == "__main__":

    kernel_configuration = {
        "load_method" : "multiplexor",
        "qpu" : "c", #python, qlmass, default
        "relative_error": None,
        "absolute_error": None
    }
    name = "PL_{}".format(kernel_configuration["load_method"])

    benchmark_arguments = {
        #Pre benchmark configuration
        "pre_benchmark": True,
        "pre_samples": None,
        "pre_save": True,
        #Saving configuration
        "save_append" : True,
        "saving_folder": "./Results/",
        "benchmark_times": "{}_times_benchmark.csv".format(name),
        "csv_results": "{}_benchmark.csv".format(name),
        "summary_results": "{}_SummaryResults.csv".format(name),
        #Computing Repetitions configuration
        "alpha": None,
        "min_meas": None,
        "max_meas": None,
        #List number of qubits tested
        "list_of_qbits": [4, 6, 8, 10],
    }

    #Configuration for the benchmark kernel
    benchmark_arguments.update({"kernel_configuration": kernel_configuration})
    ae_bench = KERNEL_BENCHMARK(**benchmark_arguments)
    ae_bench.exe()
```

Listing T01.8: Example of configuration of a complete ***Benchmark*** *execution. This part of the code should be located at the end of the* ***my_benchmark_execution.py*** *script*

As can be seen in Listing T01.8, the input dictionary that **KERNEL_BENCHMARK** class needs, *benchmark_arguments*, have several keys that allow to modify the benchmark workflow, like:

- *pre_benchmark*: For executing or not the *pre-benchmark* step.
- *pre_samples*: number of repetitions of the benchmark step.
- *pre_save*: For saving or not the results from the *pre-benchmark* step.
- *saving_folder*: Path for storing all the files generated by the execution of the **KERNEL_BENCHMARK** class.
- *benchmark_times*: name for the *csv* file where the initial and the final times for the complete benchmark execution will be stored.
- *csv_results*: name for the *csv* file where the obtained metrics for the different repetitions of the benchmark step will be stored (so the different metrics obtained during step 2 from section 3.3 will be stored in this file)
- *summary_results*: name for the *csv* file where the post-processed results (using the *summarize_results*) will be stored (so the statistics over the metrics obtained during step 3 of section 3.3 will be stored in this file)
- *list_of_qbits*: list with the different number of qubits for executing the complete **Benchmark**.
- *alpha*: for configuring the desired confidence level α
- *min_meas*: For low limiting the number of executions a benchmark step should be executed during the benchmark stage.
- *max_meas*: For high limiting the number of executions a benchmark step should be executed during the benchmark stage.

Additionally, the *kernel_configuration* key is used for configuring the probability loading algorithm and execution. The following keys can be provided for configuring it:

- *load_method*: a string for selecting the probability loading algorithm.
- *qpu*: a string for selecting the quantum process unit (**QPU**).
- *relative_error*: for changing the desired relative error of the **elapsed_time** metric.
- *absolute_error*: for changing the desired absolute error for the KS and KL metrics.

In general, most of the keys should be fixed to *None* for executing the **Benchmark** according to the guidelines of the **PL Benchmark**.

For executing the **Benchmark** following command should be used:

python my_benchmark_execution.py

A.2. Generation of the benchmark report

Following deliverable **D3.5: The NEASQC Benchmark Suite** the results of a complete **Benchmark** must be reported in a separate JSON file that must satisfy the **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable. For automating this process the following files should be modified, as explained in Annex B of the deliverable **D3.5: The NEASQC Benchmark Suite**:

- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

my_environment_info.py

This script has the functions for gathering information about the hardware where the **Benchmark** is executed.

Listing T01.9 shows an example of the **my_environment_info.py** script. Here the compiled information corresponds to a classic computer because the case was simulated instead of executed in a quantum computer.

```
import platform
import psutil
from collections import OrderedDict

def my_organisation(**kwargs):
    """
    Given information about the organisation how uploads the benchmark
    """
    #name = "None"
    name = "CESGA"
    return name

def my_machine_name(**kwargs):
    """
    Name of the machine where the benchmark was performed
    """
    #machine_name = "None"
    machine_name = platform.node()
    return machine_name

def my_qpu_model(**kwargs):
    """
    Name of the model of the QPU
    """
    #qpu_model = "None"
    qpu_model = "QLM"
    return qpu_model

def my_qpu(**kwargs):
    """
    Complete info about the used QPU
    """
    #Basic schema
    #QPUDescription = {
    #    "NumberOfQPUs": 1,
    #    "QPUs": [
    #        {
    #            "BasicGates": ["none", "none1"],
    #            "Qubits": [
    #                {
    #                    "QubitNumber": 0,
    #                    "T1": 1.0,
    #                    "T2": 1.00
    #                }
    #            ],
    #            "Gates": [
    #                {
    #                    "Gate": "none",
    #                    "Type": "Single",
    #                     "Symmetric": False,
    #                     "Qubits": [0],
    #                     "MaxTime": 1.0
    #                 }
    #             ],
    #             "Technology": "other"
    #         },
    #     ]
    #}

    #Defining the Qubits of the QPU
    qubits = OrderedDict()
    qubits["QubitNumber"] = 0
    qubits["T1"] = 1.0
    qubits["T2"] = 1.0

    #Defining the Gates of the QPU
    gates = OrderedDict()
    gates["Gate"] = "none"
    gates["Type"] = "Single"
    gates["Symmetric"] = False
    gates["Qubits"] = [0]
    gates["MaxTime"] = 1.0


    #Defining the Basic Gates of the QPU
    qpus = OrderedDict()
    qpus["BasicGates"] = ["none", "none1"]
    qpus["Qubits"] = [qubits]
    qpus["Gates"] = [gates]
    qpus["Technology"] = "other"

    qpu_description = OrderedDict()
    qpu_description['NumberOfQPUs'] = 1
    qpu_description['QPUs'] = [qpus]

    return qpu_description

def my_cpu_model(**kwargs):
    """
    model of the cpu used in the benchmark
    """
    #cpu_model = "None"
    cpu_model = platform.processor()
    return cpu_model

def my_frecuency(**kwargs):
    """
    Frcuency of the used CPU
    """
    #Use the nominal frequency. Here, it collects the maximum frequency
    #print(psutil.cpu_freq())
    #cpu_frec = 0
    cpu_frec = psutil.cpu_freq().max/1000
    return cpu_frec

def my_network(**kwargs):
    """
    Network connections if several QPUs are used
    """
    network = OrderedDict()
    network["Model"] = "None"
    network["Version"] = "None"
    network["Topology"] = "None"
    return network

def my_QPUCPUConnection(**kwargs):
    """
    Connection between the QPU and the CPU used in the benchmark
    """
    #
    # Provide the information about how the QPU is connected to the CPU
    #
    qpuccpu_conn = OrderedDict()
    qpuccpu_conn["Type"] = "memory"
    qpuccpu_conn["Version"] = "None"
    return qpuccpu_conn
```

Listing T01.9: Example of configuration of the **my_environment_info.py** script

In general, it is expected that for each computer used (quantum or classic), the **Benchmark** developer should change this script to properly get the hardware info.

A.2.1. my_benchmark_info.py

This script gathers the information under the field *Benchmarks* of the benchmark report. Information about the software, the compilers and the results obtained from an execution of the **Benchmark** is stored in this field.

Listing T01.10 shows an example of the configuration of the **my_benchmark_info.py** script for gathering the aforementioned information.

```
    import sys
    import platform
    import psutil
    import pandas as pd
    from collections import OrderedDict
    from my_benchmark_summary import summarize_results


    def my_benchmark_kernel(**kwargs):
        """
        Name for the benchmark Kernel
        """
        return "ProbabilityLoading"

    def my_starttime(**kwargs):
        """
        Providing the start time of the benchmark
        """
        #start_time = "2022-12-12T16:46:57.268509+01:00"
        times_filename = kwargs.get("times_filename", None)
        pdf = pd.read_csv(times_filename, index_col=0)
        start_time = pdf["StartTime"][0]
        return start_time

    def my_endtime(**kwargs):
        """
        Providing the end time of the benchmark
        """
        #end_time = "2022-12-12T16:46:57.268509+01:00"
        times_filename = kwargs.get("times_filename", None)
        pdf = pd.read_csv(times_filename, index_col=0)
        end_time = pdf["EndTime"][0]
        return end_time

    def my_timemethod(**kwargs):
        """
        Providing the method for getting the times
        """
        time_method = "time.time"
        return time_method

    def my_programlanguage(**kwargs):
        """
        Getting the programing language used for benchmark
        """
        program_language = platform.python_implementation()
        return program_language

    def my_programlanguage_version(**kwargs):
        """
        Getting the version of the programing language used for benchmark
        """
        language_version = platform.python_version()
        return language_version

    def my_programlanguage_vendor(**kwargs):
        """
        Getting the version of the programing language used for benchmark
        """
        language_vendor = "Unknow"
        return language_vendor

    def my_api(**kwargs):
        """
        Collect the information about the used APIs
        """
        #api = OrderedDict()
        #api["Name"] = "None"
        #api["Version"] = "None"
        #list_of_apis = [api]
        modules = []
        list_of_apis = []
        for module in list(sys.modules):
            api = OrderedDict()
            module = module.split('.')[0]
            if module not in modules:
                modules.append(module)
                api["Name"] = module
                try:
                    version = sys.modules[module].__version__
                except AttributeError:
                    #print("NO VERSION: "+str(sys.modules[module]))
                    try:
                        if  isinstance(sys.modules[module].version, str):
                            version = sys.modules[module].version
                            #print("\t Attribute Version"+version)
                        else:
                            version = sys.modules[module].version()
                            #print("\t Methdod Version"+version)
                    except (AttributeError, TypeError) as error:
                        #print('\t NO VERSION: '+str(sys.modules[module]))
                        try:
                            version = sys.modules[module].VERSION
                        except AttributeError:
                            #print('\t\t NO VERSION: '+str(sys.modules[module]))
                            version = "Unknown"
                api["Version"] = str(version)
                list_of_apis.append(api)
        return list_of_apis

    def my_quantum_compilation(**kwargs):
        """
        Information about the quantum compilation part of the benchmark
        """
        q_compilation = OrderedDict()
        q_compilation["Step"] = "None"
        q_compilation["Version"] = "None"
        q_compilation["Flags"] = "None"
        return [q_compilation]

    def my_classical_compilation(**kwargs):
        """
        Information about the classical compilation part of the benchmark
        """
        c_compilation = OrderedDict()
        c_compilation["Step"] = "None"
        c_compilation["Version"] = "None"
        c_compilation["Flags"] = "None"
        return [c_compilation]

    def my_metadata_info(**kwargs):
        """
        Other important info user want to store in the final json.
        """

        metadata = OrderedDict()
        #metadata["None"] = None
        import pandas as pd
        benchmark_file = kwargs.get("benchmark_file", None)
        pdf = pd.read_csv(benchmark_file, header=[0, 1], index_col=[0, 1])
        pdf.reset_index(inplace=True)
        load_methods = list(set(pdf["load_method"]))
        metadata["load_method"] = load_methods[0]

        return metadata


    def my_benchmark_info(**kwargs):
        """
        Complete WorkFlow for getting all the benchmar informated related info
        """
        benchmark = OrderedDict()
        benchmark["BenchmarkKernel"] = my_benchmark_kernel(**kwargs)
        benchmark["StartTime"] = my_starttime(**kwargs)
        benchmark["EndTime"] = my_endtime(**kwargs)
        benchmark["ProgramLanguage"] = my_programlanguage(**kwargs)
        benchmark["ProgramLanguageVersion"] = my_programlanguage_version(**kwargs)
        benchmark["ProgramLanguageVendor"] = my_programlanguage_vendor(**kwargs)
        benchmark["API"] = my_api(**kwargs)
        benchmark["QuantumCompililation"] = my_quantum_compilation(**kwargs)
        benchmark["ClassicalCompiler"] = my_classical_compilation(**kwargs)
        benchmark["TimeMethod"] = my_timemethod(**kwargs)
        benchmark["Results"] = summarize_results(**kwargs)
        benchmark["MetaData"] = my_metadata_info(**kwargs)
        return benchmark
```

Listing T01.10: Example of configuration of the ***my_benchmark_info.py*** *script*

The *my_benchmark_info* function gathers all the mandatory information needed by the *Benchmarks* main field of the report (by calling the different functions listed in listing T01.10). To properly fill this field some mandatory information must be provided as the typical *python kwargs*:

- *times_filename*: This is the complete path to the file where the starting and ending time of the benchmark was stored. This file must be a *csv* one and it is generated when the **KERNEL_BENCHMARK** class is executed. This information is used by the *my_starttime* and *my_endtime* functions.
- *benchmark_file*: complete path where the file with the summary results of the benchmark are stored. This information is used by the *summarize_results* function from *my_benchmark_summary.py* script (see section A.2.2) and for the *my_metadata_info* function for filling the *MetaData* sub-field of *Benchmarks* main field of the report. This *MetaData* sub-field reports the method used for creating the **PL** operator. This field is not mandatory, following the JSON schema **NEASQC**, but it is important to get good traceability of the **Benchmark** results.

A.2.2. my_benchmark_summary.py

In this script, the *summarize_results* function is implemented. This function formats the results of a complete execution of a **PL Benchmark** with a suitable **NEASQC** benchmark report format. It can be used for generating the information under the sub-field *Results* of the main field *Benchmarks* in the report.

Listing T01.11 shows an example of implementation of *summarize_results* function for the **PL Benchmark**.

```

    from collections import OrderedDict
    import psutil

    def summarize_results(**kwargs):
        """
        Mandatory code for properly present the benchmark results following
        the NEASQC jsonschema
        """

        #n_qbits = [4]
        #Info with the benchmark results like a csv or a DataFrame
        #pdf = None
        #Metrics needed for reporting. Depend on the benchmark kernel
        #list_of_metrics = ["MRSE"]
        import pandas as pd
        benchmark_file = kwargs.get("benchmark_file", None)
        pdf = pd.read_csv(benchmark_file, header=[0, 1], index_col=[0, 1])
        pdf.reset_index(inplace=True)
        n_qbits = list(set(pdf["n_qbits"]))
        load_methods = list(set(pdf["load_method"]))
        list_of_metrics = [
            "KS", "KL",
            "chi2", "p_value"
        ]

        results = []
        #In the Probability Loading benchmark several qubits can be tested
        for n_ in n_qbits:
            #For selecting the different loading method using in the benchmark
            for method in load_methods:
                #Fields for benchmark test of a fixed number of qubits
                result = OrderedDict()
                result["NumberOfQubits"] = n_
                result["QubitPlacement"] = list(range(n_))
                result["QPUs"] = [1]
                result["CPUs"] = psutil.Process().cpu_affinity()

                #Select the proper data
                step_pdf = pdf[(pdf["load_method"] == method) & (pdf["n_qbits"] == n_)]

                #result["TotalTime"] = 10.0
                #result["SigmaTotalTime"] = 1.0
                #result["QuantumTime"] = 9.0
                #result["SigmaQuantumTime"] = 0.5
                #result["ClassicalTime"] = 1.0
                #result["SigmaClassicalTime"] = 0.1

                result["TotalTime"] = step_pdf["elapsed_time"]["mean"].iloc[0]
                result["SigmaTotalTime"] = step_pdf["elapsed_time"]["std"].iloc[0]
                result["QuantumTime"] = step_pdf["quantum_time"]["mean"].iloc[0]
                result["SigmaQuantumTime"] = step_pdf["quantum_time"]["std"].iloc[0]
                result["ClassicalTime"] = step_pdf["classic_time"]["mean"].iloc[0]
                result["SigmaClassicalTime"] = step_pdf["classic_time"]["std"].iloc[0]
                #For identify the loading method used. Not mandaatory but
                #useful for identify results
                result["load_method"] = method

                metrics = []
                #For each fixed number of qbits several metrics can be reported
                for metric_name in list_of_metrics:
                    metric = OrderedDict()
                    #MANDATORY
                    metric["Metric"] = metric_name
                    #metric["Value"] = 0.1
                    #metric["STD"] = 0.001
                    metric["Value"] = step_pdf[metric_name]["mean"].iloc[0]
                    metric["STD"] = step_pdf[metric_name]["std"].iloc[0]
                    #Depending on the benchmark kernel
                    metric["COUNT"] = int(step_pdf[metric_name]["count"].iloc[0])
                    metrics.append(metric)
                result["Metrics"] = metrics
                results.append(result)
        return results
```

Listing T01.11: Example of configuration of the summarize_results function for ***PL benchmark***

As usual, the *kwargs* strategy is used for passing the arguments that the function can use. In this case, the only mandatory argument is *benchmark_file* with the path to the file where the summary results of the **Benchmark** execution were stored.

Table T01.3 shows the sub-fields and the information stored, under the *Results* field. To have proper traceability of the executions the sub-field *load_method* was created explicitly for the **PL Benchmark**.

The sub-field *Metrics* gathers information about the obtained metrics of the benchmark. Table T01.4 shows its different sub-fields and the information stored.

sub-field	information
NumberOfQubits	number of qubits, n
TotalTime	mean of **elapsed time**
SigmaTotalTime	standard deviation of **elapsed time**
QuantumTime	mean of the **quantum time**
SigmaQuantumTime	standard deviation of **quantum time**
ClassicalTime	mean of the **classical time**
SigmaClassicalTime	standard deviation of **classical time**
load_method	method used for loading probability (*brute_force*, *multiplexor* or *KPTree*)
Metrics	summarize verification metrics. See Table T01.4

Table T01.3: Sub-fields of the Results fields of the TNBS benchmark report.

sub-field	information
metric	*KS*, *KL*, *chi2* (χ^2) or *p_value*
Value	mean value of the metric
STD	standard deviation of the metric
Count	number of samples for computing the statistics of the metric

Table T01.4: Sub-fields of the Metrics field.

A.2.3. neasqc_benchmark.py

The *neasqc_benchmark.py* script can be used straightforwardly for gathering all the **Benchmark** execution information and results, for creating the final mandatory **NEASQC** benchmark report.

It does not necessarily change anything about the class implementation. It is enough to update the information of the *kwargs* arguments for providing the mandatory files for gathering all the information.

In this case, the following information should be provided as arguments for the *exe* method of the **BENCHMARK** class:

- *times_filename*: complete path where the file with the times of the **Benchmark** execution was stored.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark** execution was stored.

D.T02: Benchmark for Amplitude Estimation Algorithms

NExt ApplicationS of Quantum Computing
Benchmark Suite

T02: Benchmark for Amplitude Estimation Algorithms

Document Properties

Contract Number	951821
Contractual Deadline	31/10/2023
Dissemination Level	Public
Nature	Test Case Definition
Editors	Gonzalo Ferro (neasqc@cesga.es), CESGA
Authors	Gonzalo Ferro, CESGA Andrés Gómez, CESGA Diego Andrade, CITIC-UDC
Reviewers	Cyril Allouche, EVIDEN Arnaud Gazda, EVIDEN
Date	27/10/2023
Category	Generic
Keywords	
Status	Submitted
Release	1.0

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, Organisation	Description of Changes
0.1	04/01/2023	Gonzalo Ferro, CESGA; Andrés Gómez, CESGA	First version
0.2	26/01/2023	Gonzalo Ferro, CESGA Andrés Gómez, CESGA Diego Andrade, CITIC-UDC	Reordering sections and rewording
0.3	07/02/2023	Gonzalo Ferro, CESGA	Minor corrections and some documentation improvements
0.4	25/08/2023	Gonzalo Ferro, CESGA	Minor corrections and some documentation improvements
0.5	08/10/2023	Andrés Gómez, CESGA	Formatting
1.0	24/10/2023	Gonzalo Ferro, CESGA	Fixing the naming according to the Glossary of deliverable 3.5

Table of Contents

1.Introduction

This document describes the T2:Amplitude Estimation benchmark of The NEASQC Benchmarking Suite (**TNBS**). This document must be read in companion with the document that describes the TNBS: D3.5: The NEASQC Benchmark Suite.

Section 2 describes properly the selected **Amplitude Estimation Kernel**, **AE Kernel** from now. With each **TNBS Kernel**, a **Benchmark Test Case** (**BTC**) must be designed and documented. This is done in Section 3. Finally, the benchmarking methodology aims to develop a complete software implementation of the **Benchmark** using the Eviden myQLM library. A complete documentation of this implementation is provided in Annex A.

2.Description of the kernel: Amplitude Estimation

The present section describes the **AE Kernel** for the **TNBS**. Section 2.1 justifies kernel selection according to the **TNBS** benchmarking methodology meanwhile section 2.2 presents a complete description of this **AE Kernel**.

2.1. Kernel selection justification

The **AE Kernel** is a core step in quantum computation for various applications like finance (Gómez et al., 2022; Rebentrost et al., 2018; Woerner & Egger, 2019), chemistry (Aspuru-Guzik et al., 2005; Knill et al., 2007), machine learning (Wiebe et al., 2015, 2016) and, even, can be used for generic tasks such as numeric integration (Montanaro, 2015). For executing **AE Kernel**, different algorithm approaches, **AE** algorithms from now, were proposed recently (Brassard et al., 2000; Grinko et al., 2021; Lu & Lin, 2023; Manzano et al., 2023; Suzuki et al., 2020; Uno et al., 2021; Zhao et al., 2022). So the **AE Kernel** can be considered as an interesting candidate for **TNBS Kernel**. Additionally, it satisfies the three main requirements from the **NEASQC** benchmark methodology:

1. A mathematical definition of the **Kernel** can be given with enough accuracy to allow the construction of a standalone circuit (see sections 2.2 and 3.2).
2. The **Kernel** can be defined using a smaller or larger number of qubits.
3. The output can be verified with a classical computation (in the proposed **BTC**, see section 3, the result is known *a priori*)

2.2. Kernel Description

The **AE Kernel**, also know as the **Amplitude Estimation** problem, can be defined in the following way:

Let an unitary operator **A** that acts upon an initial n-qubits state $|0\rangle_n = |0\rangle^{\otimes n}$ as shown in equation (T02.2.1):

$$|\Psi\rangle = \mathbf{A}|0\rangle_n = \sum_{i=0}^{2^n-1} a_i |i\rangle_n \tag{T02.2.1}$$

Now we are interested in the sub-state composed by some basis states $J = \{j_0, j_1, \cdots, j_l\}$, so we can write down (T02.2.2):

$$|\Psi\rangle = \mathbf{A}|0\rangle_n = \sum_{j\in J} a_j |j\rangle_n + \sum_{i\notin J} a_i |i\rangle_n \tag{T02.2.2}$$

If we define the sub-states $|\Psi_0\rangle$ and $|\Psi_1\rangle$ using (T02.2.3):

$$|\Psi_0\rangle = \frac{1}{\sqrt{a}} \sum_{j\in J} a_j |j\rangle_n \ \ and \ \ |\Psi_1\rangle = \frac{1}{\sqrt{1-a}} \sum_{i\notin J}^{2^n-1} a_i |i\rangle_n \tag{T02.2.3}$$

The final $|\Psi\rangle$ can be expressed as (T02.2.4):

$$|\Psi\rangle = \mathbf{A}|0\rangle_n = \sqrt{a}|\Psi_0\rangle + \sqrt{1-a}|\Psi_1\rangle \tag{T02.2.4}$$

The **AE Kernel** consists in getting an estimation of the amplitude of $|\Psi_0\rangle$: a.

The following subsections present different approaches for solving the **AE Kernel**.

2.2.1. Monte Carlo Solution

One naive procedure for solving **AE Kernel**, *Monte Carlo* solution from now, is measuring all the qubits N times and getting the probability of obtaining the desired state $|\Psi_0\rangle$. In this case the estimator of a, $\tilde{a}$, will be given by equation (T02.2.5):

$$\tilde{a} = P_{|\Psi_0\rangle} = \frac{Number\ of\ times\ |\Psi_0\rangle\ was\ measured}{N} \tag{T02.2.5}$$

The error ϵ_a of this $\tilde{a}$ estimator can be obtained using the *Chernoff-Hoeffding* (Hoeffding's inequality, 2004) bound (T02.2.6):

$$P[\tilde{a} \in |a_j - \epsilon_a, a_j + \epsilon_a|] \geq 2e^{-2N\epsilon_a^2} \tag{T02.2.6}$$

So if we want $P[\tilde{a} \in |a_j - \epsilon_a, a_j + \epsilon_a|] \geq \alpha$ ($\alpha \in [0, 1]$) then the error is given by (T02.2.7):

$$\epsilon_a^2 \leq \frac{1}{2N} Ln[\frac{2}{\alpha}] \tag{T02.2.7}$$

So the error for the estimator $\tilde{a}$ has the following behaviour with the number of measurements N:

$$\epsilon_a \sim \frac{1}{\sqrt{N}} \tag{T02.2.8}$$

Usually, for the **AE Kernel**, instead of the number of measurements, the number of calls to the oracle (this is the operator **A**), N_{oracle}, is used. In the *Monte Carlo* solution: $N = N_{oracle}$, so equation (T02.2.8) can be rewritten as equation (T02.2.9)

$$\epsilon_a \sim \frac{1}{\sqrt{N_{oracle}}} \tag{T02.2.9}$$

2.2.2. Canonical AE solution with Quantum Phase Estimation

In equation (T02.2.4) the following substitution: $\sqrt{a} = \sin(\theta)$ can be performed and equation (T02.2.10) can be obtained:

$$|\Psi\rangle = \mathbf{A}|0\rangle_n = \sqrt{a}|\Psi_0\rangle + \sqrt{1-a}|\Psi_1\rangle = \sin(\theta)|\Psi_0\rangle + \cos(\theta)|\Psi_1\rangle \tag{T02.2.10}$$

Now a Grover-like operator (Brassard et al., 2000) based on **A**, can be built following equation (T02.2.11):

$$\mathbf{G}(\mathbf{A}) = \mathbf{A}\left(\hat{I} - 2|0\rangle\langle 0|\right)\mathbf{A}^\dagger\left(\hat{I} - 2|\Psi_0\rangle\langle\Psi_0|\right) \tag{T02.2.11}$$

This Grover-like operator acts as shown in equation (T02.2.12):

$$\mathbf{G}^k(\mathbf{A})|\Psi\rangle = \mathbf{G}^k(\mathbf{A})\mathbf{A}|0\rangle_n = \sin\left((2k+1)\theta\right)|\Psi_0\rangle + \cos\left((2k+1)\theta\right)|\Psi_1\rangle \tag{T02.2.12}$$

being k the number of times that operator **G** is applied.

The *canonical Quantum Amplitude Estimation* solution for **AE** problem, uses the *Quantum Phase Estimation* algorithm, **QPE**, (Brassard et al., 2000) over the operator $\mathbf{G}(\mathbf{A})$ for computing $\tilde{a}$. This algorithm allocates m auxiliary qubits and applies, over $|\Psi\rangle$, geometrically increasing controlled, by the different auxiliary qubits, powers of **G** as shown in the Figure T02.1

Finally, the complex conjugate of the *Quantum Fourier Transformation* ($QFT^\dagger$ in Figure T02.1) is applied over the auxiliary qubits, that will be measured generating an integer $y \in \{0, 1, ...M-1\}$, where $M = 2^m$. This integer can be mapped to an angle using:

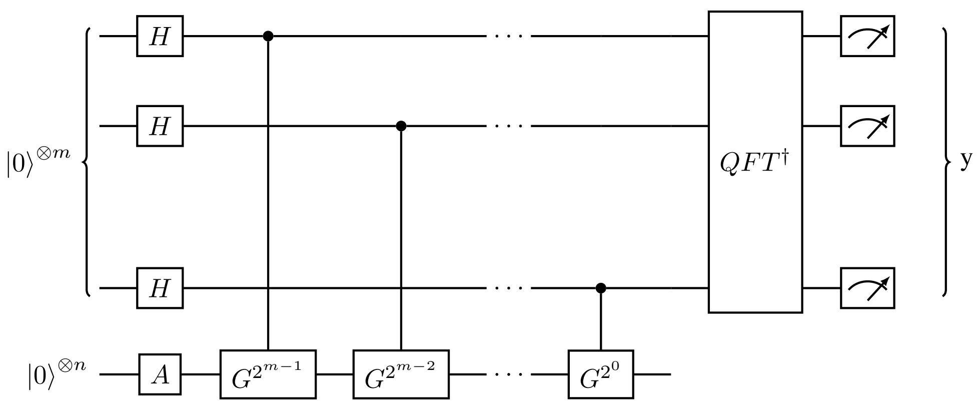


Figure T02.1: Canonical Amplitude Estimation using Quantum Phase Estimation.

$$\tilde{\theta} = \frac{y\pi}{2^m} \tag{T02.2.13}$$

In this case the estimation will be $\tilde{a} = \sin^2(\tilde{\theta})$. With a probability of at least $\frac{8}{\pi^2} \sim 81\%$ the error of the estimator will be given by (T02.2.14) (Brassard et al., 2000):

$$\epsilon = |\tilde{a} - a| \leq \frac{2\pi\sqrt{a(1-a)}}{M} + \frac{\pi^2}{M^2} \tag{T02.2.14}$$

So in this case the error for the estimator $\tilde{a}$ scales with:

$$\epsilon_a \sim \frac{1}{M} \tag{T02.2.15}$$

The number of auxiliary qbits, m, is related to the number of oracle calls by equation (T02.2.16):

$$M = 2^m = \frac{N_{oracle} + 1}{2} \tag{T02.2.16}$$

By plugging (T02.2.16) into (T02.2.15), the error for the *canonical Quantum Amplitude Estimation* algorithm can be obtained as a function of the number of oracle calls (T02.2.17):

$$\epsilon_a \sim \frac{2}{N_{oracle} + 1} \sim \frac{1}{N_{oracle}} \tag{T02.2.17}$$

This approach yields a quadratic speed up over the *Monte Carlo* method (T02.2.9).

2.2.3. Amplitude Estimation without Phase Estimation

Canonical Quantum Amplitude Estimation is computationally expensive and presents some caveats to be implemented in current quantum computers. However, there are several algorithms that can solve the **AE** problem, without the use of **QPE**, where the error of the $\tilde{a}$ estimation, ϵ_a, scales between *Monte Carlo* and *Canonical Quantum Amplitude Estimation* one, this is:

$$\frac{1}{N_{oracle}} < \epsilon_a < \frac{1}{\sqrt{N_{oracle}}} \tag{T02.2.18}$$

The main idea of these algorithms is to take advantage of the fact:

$$\mathbf{G}^k|\Psi\rangle = \mathbf{G}^k\mathbf{A}|0\rangle_n = \sin\big((2k+1)\theta\big)|\Psi_0\rangle + \cos\big((2k+1)\theta\big)|\Psi_1\rangle \tag{T02.2.19}$$

And in the use of very smart strategies for selecting k to maximize the probability of measuring the $|\Psi_0\rangle$:

$$P[|\Psi_0\rangle] = \sin^2\big((2k+1)\theta\big) \tag{T02.2.20}$$

The **TNBS AE Kernel** is agnostic to the algorithm used for solving it, so it can be used for testing not only quantum computers devices, it can be used for testing different **AE** algorithms.

3.Description of the benchmark test case

This section presents the complete description of the **BTC** for the **AE Kernel**. The main idea is the computation of the integral of a particular function, in a well-defined interval, using some particular implementation, usually an **AE** algorithm, of the **AE Kernel**.

Section 3.1 presents the base problem, integral computation, of the **BTC** in a formal way. Section 3.2 describes, exhaustively, how the **BTC** should be implemented from a formal perspective. Section 3.3 provides the workflow for complete execution of **AE Benchmark**. Finally, section 3.4 documents how the results of an execution of the **Benchmark** must be reported.

3.1. Description of the problem

The computation of the integral of a function, $f(x)$, in a closed interval $[a, b] \subset \mathbb{R}$, is the proposed **BTC** for **AE Kernel**.

An operator **A** must be constructed in such a way that the desired integral: **F**,

$$\mathbf{F} = \int_a^b f(x)dx \tag{T02.1.1}$$

must be encoded into the amplitude of a very well-defined state. This is, the operator **A** must act as showed in equation (T02.1.2)

$$|\Psi\rangle = \mathbf{A}|0\rangle_n = \sqrt{a}|\Psi_0\rangle + \sqrt{1-a}|\Psi_1\rangle \tag{T02.1.2}$$

where $\sqrt{a} = \mathbf{F}$

This **A** operator must be given as input to an **AE** algorithm, that must return the estimation of the $\tilde{\mathbf{F}}$. To evaluate the performance of the operator the estimator should be compared to the actual integral value **F**

The proposed function for the **BTC** is $f(x) = \sin x$, whose integral can be calculated easily as (T02.1.3):

$$\mathbf{F} = \int_a^b \sin(x)dx = -\cos x|_a^b = \cos(a) - \cos(b) \tag{T02.1.3}$$

and the two following integration intervals will be used:

- $[0, \frac{3\pi}{8}]$: $\mathbf{F}^0 = \int_0^{\frac{3\pi}{8}} \sin(x)dx = 0.6173165676349102$. This computation will be mandatory .
- $[\pi, \frac{5\pi}{4}]$: $\mathbf{F}^1 = \int_\pi^{\frac{5\pi}{4}} \sin(x)dx = -0.2928932188134523$. This computation will be optional.

In summary, for the **BTC**, an operator $\mathbf{A}^0$ must be built and a particular **AE** algorithm must compute and report the integral $\mathbf{F}^0$. Additionally, a second $\mathbf{A}^1$ operator can be built and the corresponding integral $\mathbf{F}^1$ can be reported.

3.2. Benchmark test case description

This section presents a complete mathematical description of the **BTC** for the **AE Kernel**.

The **BTC** proposed requires a set of steps that are explained in detail in Sections 3.2.1-3.2.4, namely: the discretization of the domain and the function, the normalization of the array, and the encoding of the function as a quantum circuit. Section 3.2.6 describes the metrics used to verify the output of the circuit, and Section 3.2.7 describes the general workflow of the **BTC** including the steps described before.

3.2.1. Domain Discretization

The first step is the discretization of each domain in 2^n intervals, with $n \in \mathbb{N}$ as shown in (T02.2.4):

$$\{[x_0, x_1], [x_1, x_2], ..., [x_{2^n-1}, x_{2^n}]\} \tag{T02.2.4}$$

Where

- $x_{i+1} > x_i$
- $a = x_0$
- $b = x_{2^n}$

3.2.2. Function discretization

For each domain, the following array with the discretization of the desired function, $f(x) = \sin(x)$, must be computed:

$$f_{x_i} = \frac{f(x_{i+1}) + f(x_i)}{2}$$

The desired integral, for each interval, can be approximated as Riemann sum (T02.2.5):

$$S_{[a,b]} = \sum_{i=0}^{2^n-1} f_{x_i} \cdot (x_{i+1} - x_i) \tag{T02.2.5}$$

Using $x_{i+1} - x_i = \frac{b-a}{2^n}$ then we can write down (T02.2.5) as:

$$S_{[a,b]} = \sum_{i=0}^{2^n-1} f_{x_i} \frac{b-a}{2^n} = \frac{b-a}{2^n} \sum_{i=0}^{2^n-1} f_{x_i} \tag{T02.2.6}$$

When $(x_{i+1} - x_i) \to 0$ (i.e., $n \to \infty$), $S_{[a,b]} \to \mathbf{F} = \int_a^b \sin(x)dx$

3.2.3. Array Normalisation

A normalization step, over the f_{x_i} array, must be performed before creating the operator $\mathbf{A}$ that will encode the function into a quantum circuit. This should be done using (T02.2.7):

$$f_{x_i}^{norm} = \frac{f_{x_i}}{\max(|f_{x_i}|)} \tag{T02.2.7}$$

Now the computed integral will be

$$S_{[a,b]} = \frac{b-a}{2^n} \sum_{i=0}^{2^n-1} f_{x_i} = \frac{b-a}{2^n} \sum_{i=0}^{2^n-1} \max(|f_{x_i}|) f_{x_i}^{norm} = \frac{\max(|f_{x_i}|)(b-a)}{2^n} \sum_{i=0}^{2^n-1} f_{x_i}^{norm} \tag{T02.2.8}$$

3.2.4. Encoding function into a quantum circuit

The next step is to codify $f_{x_i}^{norm}$ array in a quantum circuit. The following procedure must be used:

1. Initialize a quantum register with at least $n+1$ qubits[1], where n must be equal to the n used to define the 2^n discretization intervals (see section 3.2.1):

$$|0\rangle \otimes |0\rangle_n \tag{T02.2.9}$$

[1]Additional auxiliary qubits may be used

2. Apply the uniform distribution over the first n qubits as shown in (T02.2.10):

$$\left(\mathbb{I} \otimes H^{\otimes n}\right)\left(|0\rangle \otimes |0\rangle_n\right) = |0\rangle \otimes H^{\otimes n}|0\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |0\rangle \otimes |i\rangle_n \tag{T02.2.10}$$

3. Create an operator $\mathbf{U}_f$ for encoding the $f^{norm}_{x_i}$. This operator must act as shown in (T02.2.11):

$$\mathbf{U}_f\left(|0\rangle \otimes |i\rangle_n\right) = \left(f^{norm}_{x_i}|0\rangle + \beta_i|1\rangle\right) \otimes |i\rangle_n \tag{T02.2.11}$$

4. Apply the $\mathbf{U}_f$ operator over the $n+1$ qubits:

$$\mathbf{U}_f\left(\mathbb{I} \otimes H^{\otimes n}\right)|0\rangle \otimes |0\rangle_n \tag{T02.2.12}$$

5. Applying equation (T02.2.10) and (T02.2.11) into (T02.2.12) equation (T02.2.13) is obtained:

$$\begin{aligned}\mathbf{U}_f\left(\mathbb{I} \otimes H^{\otimes n}\right)|0\rangle \otimes |0\rangle_n = \mathbf{U}_f\left(\frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} |0\rangle \otimes |i\rangle_n\right) = \\ = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} \mathbf{U}_f\left(|0\rangle \otimes |i\rangle_n\right) = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} \left(f^{norm}_{x_i}|0\rangle + \beta_i|1\rangle\right) \otimes |i\rangle_n\end{aligned} \tag{T02.2.13}$$

6. In equation (T02.2.13) the amplitude β_i is not important.
7. Finally the uniform distribution is applied over the first n qubits again as shown in (T02.2.14):

$$|\Psi\rangle = \left(\mathbb{I} \otimes H^{\otimes n}\right)\mathbf{U}_f\left(\mathbb{I} \otimes H^{\otimes n}\right)|0\rangle \otimes |0\rangle_n \tag{T02.2.14}$$

8. So applying (T02.2.13) into (T02.2.14) the equation (T02.2.15) can be obtained:

$$|\Psi\rangle = \left(\mathbb{I} \otimes H^{\otimes n}\right)\mathbf{U}_f\left(\mathbb{I} \otimes H^{\otimes n}\right)|0\rangle \otimes |0\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} \left(f^{norm}_{x_i}|0\rangle + \beta_i|1\rangle\right) \otimes H^{\otimes n}|i\rangle_n \tag{T02.2.15}$$

9. Taking into account only the $|0\rangle \otimes |i\rangle_n$ terms, equation (T02.2.15) can be expressed as (T02.2.16):

$$|\Psi\rangle = \left(\mathbb{I} \otimes H^{\otimes n}\right)\mathbf{U}_f\left(\mathbb{I} \otimes H^{\otimes n}\right)|0\rangle \otimes |0\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{2^n-1} f^{norm}_{x_i}|0\rangle \otimes H^{\otimes n}|i\rangle_n + \cdots \tag{T02.2.16}$$

10. It is known that:

$$H^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n} \sum_{k=0}^{2^n} (-1)^{jk}|j\rangle_{nn}\langle k| \tag{T02.2.17}$$

11. So $H^{\otimes n}|i\rangle_n$ can be expressed using equation (T02.2.18):

$$H^{\otimes n}|i\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n} \sum_{k=0}^{2^n} (-1)^{jk}|j\rangle_{nn}\langle k|i\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n} (-1)^{ji}|j\rangle_n = \frac{1}{\sqrt{2^n}}|0\rangle_n + \frac{1}{\sqrt{2^n}} \sum_{j=1}^{2^n} (-1)^{ji}|j\rangle_n \tag{T02.2.18}$$

12. Finally applying (T02.2.18) into (T02.2.16) and taking only into account the $|0\rangle \otimes |0\rangle_n$ term, equation (T02.2.19) can be obtained:

$$|\Psi\rangle = \left(\mathbb{I} \otimes H^{\otimes n}\right) \mathbf{U}_f \left(\mathbb{I} \otimes H^{\otimes n}\right) |0\rangle \otimes |0\rangle_n = \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_{x_i}^{norm} |0\rangle \otimes |0\rangle_n + \cdots \tag{T02.2.19}$$

Using the previous steps, two different operators $\mathbf{A}^I(f_{x_i})$ must be created following equation (T02.2.20):

$$\mathbf{A}^I(f_{x_i}) = \left(\mathbb{I} \otimes H^{\otimes n}\right) \mathbf{U}_f^I \left(\mathbb{I} \otimes H^{\otimes n}\right) \tag{T02.2.20}$$

where the superscript I can take 0 or 1 depending on the domain integration interval

Using (T02.2.19) the behaviour of such operators will be:

$$|\Psi\rangle = \mathbf{A}^I(f_{x_i}) |0\rangle \otimes |0\rangle_n = \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_{x_i}^{norm} |0\rangle \otimes |0\rangle_n + \cdots \tag{T02.2.21}$$

Equation (T02.2.21) can be compared with the equation (T02.2.10):

$$|\Psi\rangle = \mathbf{A}|0\rangle_n = \sqrt{a}|\Psi_0\rangle + \sqrt{1-a}|\Psi_1\rangle$$

where

$$|\Psi_0\rangle = |0\rangle \otimes |0\rangle_n$$

and

$$\sqrt{a} = \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_{x_i}^{norm}$$

Now the Riemann sum approximation of the desired integral can be computed by measuring the probability of obtaining the state $|\Psi_0\rangle = |0\rangle \otimes |0\rangle_n$ as shown in (T02.2.22)

$$\mathbf{P}[|\Psi_0\rangle] = |\ \langle \Psi_0\ |\Psi\rangle\ |^2 = \left| \langle \Psi_0\ |\frac{1}{2^n} \sum_{i=0}^{2^n-1} f_{x_i}^{norm} |\Psi_0\rangle \right|^2 = \left| \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_{x_i}^{norm} \right|^2 = \tilde{a} \tag{T02.2.22}$$

The $\sim$ in $\tilde{a}$ indicates that the amplitude was obtained using a quantum measurement.

Now, plugging (T02.2.22) into the Riemann sum (T02.2.8) the desired integral can be computed as (T02.2.23)

$$\tilde{S}_{[a,b]} = \frac{\max(f_{x_i})\,(b-a)}{2^n} \left(2^n \sqrt{\mathbf{P}[|\Psi_0\rangle]}\right) \tag{T02.2.23}$$

In (T02.2.23) the $\sim$ in $\tilde{S}_{[a,b]}$ indicates that the integral was obtained using a measurement meanwhile the $S_{[a,b]}$ is for pure Riemann sum calculation as shown in (T02.2.6).

The 2^n terms can be removed from the equation but they will be kept for the moment.

3.2.4.1. Operator $\mathbf{U}_f$

This subsection describes the steps for building the $\mathbf{U}_f$ operator, equation (T02.2.11):

- The following array must be computed: $\phi_{x_i} = \arccos(f_{x_i}^{norm})$, using the values of array $f_{x_i}^{norm}$.

- For a given state $|i\rangle_n \otimes |0\rangle$, it must be implemented a rotation around the *y-axis* over the last qubit, $|0\rangle$, controlled by the state $|i\rangle_n$ of $2 * \phi_{x_i}$. So the following operation must be built:

$$|0\rangle \otimes |i\rangle_n \rightarrow \mathbf{R}_y(2 * \phi_{x_i})|0\rangle \otimes |i\rangle_n = (\cos(\phi_{x_i})|0\rangle + \sin(\phi_{x_i})|1\rangle) \otimes |i\rangle_n \quad \text{(T02.2.24)}$$

- Now undoing the ϕ_{x_i} and doing $\beta_i = \sin(\phi_{x_i})$ the desired operator $\mathbf{U}_f$ can be obtained by:

$$\left(f^{norm}_{x_i}|0\rangle + \beta_i|1\rangle\right) \otimes |i\rangle_n = \mathbf{U}_f\left(|0\rangle \otimes |i\rangle_n\right) \quad \text{(T02.2.25)}$$

So the operator $\mathbf{U}_f$ can be constructed following equations (T02.2.24) and (T02.2.25) that can be summarized into (T02.2.26):

$$\mathbf{U}_f\left(|0\rangle \otimes |i\rangle_n\right) = \left(\mathbf{R}_y(2 * \phi_{x_i})|0\rangle\right) \otimes |i\rangle_n \quad \text{(T02.2.26)}$$

The $\mathbf{U}_f$ is a controlled rotation by state. The recommended way for implementing it, is using **quantum multiplexors** (Shende et al., 2006). A direct implementation of this operator can be used but, in general, deeper circuits with redundant operations are obtained concerning the **quantum multiplexors** implementation.

3.2.5. Amplitude Estimation Algorithm

As shown in section 3.2.4 the two $\mathbf{A}^I$ operators allow to encode each correspondent integral in the amplitude of the state $|0\rangle \otimes |0\rangle_n$.

For a given **AE** algorithm:

- Operator $\mathbf{A}^0$ must be provided as input of the **AE** algorithm and the obtained $\tilde{S}^0_{[a,b]}$ integral must be reported as output.
- Additionally, operator $\mathbf{A}^1$ can be provided as input of the **AE** algorithm and the obtained $\tilde{S}^1_{[a,b]}$ integral can be reported as output.

Note: in general most **AE** algorithms use the Grover-like operator of **A**, equation (T02.2.11), for solving the **AE** problem. The **AE Kernel** and the correspondent **BTC** presented in this document are agnostic to Grover operators. The only mandatory input is the operator **A**.

3.2.6. Getting the metrics

The quality of the **AE** estimation of the integrals obtained in the previous steps, $\tilde{S}^I_{[a,b]}$ (where $I = \{0, 1\}$ stands for each of the intervals where the integral can be computed) must be evaluated using the following metrics:

- **Integral Absolute Error** metric: absolute difference between the **AE** estimator and the Riemann sum (see equation (T02.2.6)): IAE $= |\tilde{S}^I_{[a,b]} - S_{[a^I,b^I]}|$
- **Oracle calls**: total number of calls of the operator $\mathbf{A}^I$. **BE AWARE:** The number of shots should be taken into account in this calculation.

3.2.7. Summary of the BTC for AE Kernel

A step-by-step workflow of the **BTC** for the **AE Kernel**, with references to the before-explained components, is presented here.

For a desired number of qubits and one of the integration intervals, described in Section 3.1, the following steps must be executed:

1. Create the domain discretization as explained in Section 3.2.1
2. Create the array with the correspondent *sine* function discretization as explained in Section 3.2.2
3. Compute normalization of the array as explained in Section 3.2.3
4. Create the $\mathbf{A}^I$ oracle operator for encoding the array as explained in Section 3.2.4

5. Using the $\mathbf{A}^I$ oracle operator as input of the *AE* algorithm for computing the estimation of $\tilde{a}$, see equation (T02.2.22).
6. Post-process the result of the *AE* algorithm for getting the estimation of the integral as explained in Section 3.2.6 and in equation (T02.2.23)
7. Compute the desired metrics as explained in Section 3.2.6

Additionally, the following times should be computed:

- **elapsed time**: the complete **BTC** step time, from step 1 to step 7.
- **run time**: the execution time of the *Amplitude Estimation* algorithm, time of step 5.
- **quantum time**: If possible, the time of the pure quantum part of the algorithm should be registered.

BE AWARE. A few words should be mentioned about the *number of shots* that should be used in the **AE Kernel BTC**. The *number of shots* in general will depend on the **AE** algorithm used so it won't be requested as a fixed input of the **BTC** but for the computation of the *Oracle Calls* **must** be done taking it into account.

3.3. Complete benchmark procedure

To execute a complete **AE Benchmark** following steps should be done:

1. A range of a number of qubits should be selected (for example from n=4 to n=8).
2. Depending on the **AE** algorithm it should be selected if only the first integral interval, or both, will be tested.
3. For each selected number of qubits and the desired integration interval, the following steps should be executed:
   a) Execute a warm-up step consisting of:
      i. Executing 10 iterations of the **BTC**, as explained in subsection 3.2.7, compute the mean, μ, and the standard deviation, σ, for the number of **Oracle calls**, $(\mu_{calls}, \sigma_{calls})$ and for the **elapsed_time**, (μ_t, σ_t). Additionally, compute the standard deviation for **Integral Absolute Error**, σ_{IAE}.
      ii. Compute the number of mandatory repetitions, M_m with $m = \{calls, t\}$, for having a relative error of 5 %, $r = 0.05$, for the **Oracle calls** and for the **elapsed time**, with a confidence level of 95 %, $\alpha = 0.05$, following (T02.3.27), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for α.

$$M_m = (\frac{\sigma_m Z_{1-\frac{\alpha}{2}}}{r\mu_m})^2 \tag{T02.3.27}$$

      iii. Compute the number of mandatory repetitions, M_{IAE} for having an absolute error of $\epsilon = 10^{-4}$, for the **Integral Absolute Error** metric, with a confidence level of 95 %, $\alpha = 0.05$, following (T02.3.28), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for α.

$$M_{\text{IAE}} = (\frac{\sigma_{\text{IAE}} Z_{1-\frac{\alpha}{2}}}{\epsilon})^2 \tag{T02.3.28}$$

   b) Execute the complete **BTC** step, section 3.2.7, $M = \max(M_{calls}, M_t, M_{\text{IAE}})$ times. M must be greater than 5.
   c) Compute the mean and the standard deviation for each of the metrics presented in section 3.2.6 (**Integral Absolute Error**, **Oracle calls**) and for all the measured times explained in section 3.2.7 (*elapsed time*, *run time* and *quantum time*)

If the before workflow is followed it can be said that, for each number of qubits and integration interval executed, the reported **Oracle calls** and **elapsed time** will have a relative error lower than 5% and the reported **Integral Absolute Error** will have an absolute error lower than 10^{-4} with a confidence level of 95%.

3.4. Benchmark report

Finally, the results of the **BTC** execution, step 3 of section 3.3, should be reported, for each of the tested number of qubits and each integral interval, into a valid JSON file following the JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the deliverable 3.2 of the NEASQC project. The mean of the *elapsed time* must be reported into the *TotalTime* field and its standard deviation into the *SigmaTotalTime*.

The verification metrics of the **AE Benchmark** should be stored under the field *Benchmarks* into the sub-field *Results* and inside the sub-field *Metrics* of the JSON **Benchmark** report. The **Integral Absolute Error** is stored under the name *IntegralAbsoluteError* and the **number of oracle calls** under the name *oracle_calls*.

If the *AE* algorithm tested under this benchmark procedure has some important internal parameter configuration, the settings of such parameters must be included in the JSON final report. These parameters can be included under an additional field called *MetaData* into the *Benchmarks* main field of the report. This field is not mandatory following the **NEASQC** JSON schema document but it is important to get a good traceability of the benchmark results.

List of Acronyms

Term	Definition
AE	Amplitude Estimation
BTC	Benchmark Test Case
CQPEAE	Classical Quantum Phase Estimation Amplitude Estimation
IQAE	Iterative Quantum Amplitude Estimation
IQPEAE	Iterative Quantum Phase Estimation Amplitude Estimation
MLAE	Maximum Likelihood Amplitude Estimation
NEASQC	NExt ApplicationS of Quantum Computing
QPE	Quantum Phase Estimation
QFT	Quantum Fourier Transformation
QPU	Quantum Process Unit
RQAE	Real Quantum Amplitude Estimation
TNBS	The **NEASQC** Benchmark Suite
WP	Work Package

Table T02.1: Acronyms and Abbreviations

List of Figures

List of Tables

List of Listings

Bibliography

Aspuru-Guzik, A., Dutoi, A. D., Love, P. J., & Head-Gordon, M. (2005). Simulated quantum computation of molecular energies. Science, 309(5741), 1704–1707. https://doi.org/10.1126/science.1113479

Brassard, G., Hoyer, P., Mosca, M., & Tapp, A. (2000). Quantum amplitude amplification and estimation. AMS Contemporary Mathematics Series, 305. https://doi.org/10.1090/conm/305/05215

Dobsicek, M., Johansson, G., Shumeiko, V., & Wendin, G. (2007). Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: A two-qubit benchmark. Physical Review A, 76(3). https://doi.org/10.1103/physreva.76.030306

Ferro, G., Manzano, A., Gómez, A., Leitao, A., R. Nogueiras, M., & Vázque, C. (2022). D5.4: Evaluation of quantum algorithms for pricing and computation of var.

Gómez, A., Leitao Rodriguez, A., Manzano, A., Nogueiras, M., Ordóñez, G., & Vázquez, C. (2022). A survey on quantum computational finance for derivatives pricing and var. Archives of Computational Methods in Engineering. https://doi.org/10.1007/s11831-022-09732-9

Grinko, D., Gacon, J., Zoufal, C., & Woerner, S. (2021). Iterative quantum amplitude estimation. npj Quantum Information, 7(1). https://doi.org/10.1038/s41534-021-00379-1

Hoeffding's inequality. (2004). Hoeffding's inequality Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Hoeffding%5C%27s_inequality

Kitaev, A. Y. (1995). Quantum measurements and the abelian stabilizer problem. Electron. Colloquium Comput. Complex., TR96.

Knill, E., Ortiz, G., & Somma, R. D. (2007). Optimal quantum measurements of expectation values of observables. Physical Review A, 75, 012328. https://doi.org/10.1103/PhysRevA.75.012328

Lu, X., & Lin, H. (2023). Random-depth quantum amplitude estimation. https://doi.org/10.48550/ARXIV.2301.00528

Manzano, A., Musso, D., & Leitao, Á. (2023). Real quantum amplitude estimation. EPJ Quantum Technology, 10. https://doi.org/10.1140/epjqt/s40507-023-00159-0

Montanaro, A. (2015). Quantum speedup of monte carlo methods. Proceedings of the Royal Society A: Mathematical, Physical and En
471(2181). https://doi.org/10.1098/rspa.2015.0301

Nogueiras, M., Ordónez Sanz, G., Vázquez Cendón, C., Leitao Rodríguez, A., Manzano Herrero, A., Musso, D., & Gómez, A. (2021). D5.1: Review of state-of-the-art forpricing and computation of var.

Rebentrost, P., Gupt, B., & Bromley, T. R. (2018). Quantum computational finance: Monte Carlo pricing of financial derivatives. Physical Review A, 98(2).

Shende, V., Bullock, S., & Markov, I. (2006). Synthesis of quantum-logic circuits. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25(6), 1000–1010. https://doi.org/10.1109/tcad.2005.855930

Suzuki, Y., Uno, S., Raymond, R., Tanaka, T., Onodera, T., & Yamamoto, N. (2020). Amplitude estimation without phase estimation. Quantum Information Processing, 19(2). https://doi.org/10.1007/s11128-019-2565-2

Uno, S., Suzuki, Y., Hisanaga, K., Raymond, R., Tanaka, T., Onodera, T., & Yamamoto, N. (2021). Modified grover operator for quantum amplitude estimation. New Journal of Physics, 23(8), 083031. https://doi.org/10.1088/1367-2630/ac19da

Wiebe, N., Kapoor, A., & Svore, K. M. (2015). Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning. Quantum Info. Comput., 15(3–4), 316–356.

Wiebe, N., Kapoor, A., & Svore, K. M. (2016). Quantum deep learning. Quantum Info. Comput., 16(7–8), 541–587.

Woerner, S., & Egger, D. J. (2019). Quantum risk analysis. npj Quantum Information, 5(1). https://doi.org/10.1038/s41534-019-0130-6

Zhao, Y., Wang, H., Xu, K., Wang, Y., Zhu, J., & Wang, F. (2022). Adaptive algorithm for quantum amplitude estimation. https://doi.org/10.48550/ARXIV.2206.08449

A.NEASQC test case reference

As pointed out in deliverable **D3.5: The NEASQC Benchmark Suite** each proposed **Benchmark** for **TNBS**, must have a complete Eviden myQLM-compatible software implementation. For the **AE Benchmark**, this implementation can be found in the **tnbs/BTC_02_AE** folder of the **WP3_Benchmark** **NEASQC** GitHub repository. Additionally, the execution of a **Benchmark** must generate a complete result report into a separate JSON file, that must follow **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable.

The **tnbs/BTC_02_AE** locations contains the following folders and files:

- **QQuantLib** folder: with a complete copy of the **QQuantLib** library from the **NEASQC Financial Applications** GitHub's repository.
- jsons folder: contains several JSON files for configuring the different **AE** algorithms implemented in this library:
  - integral_mcae_configuration.json: for a pure **MonteCarlo** solution.
  - integral_mlae_configuration.json: for a **MLAE** algorithm.
  - integral_iqae_configuration.json: for a **IQAE** algorithm.
  - integral_rqae_configuration.json: for a **RQAE** algorithm
  - integral_cqpeae_configuration.json: for a **CQPEAE** algorithm.
  - integral_iqpeae_configuration.json: for the **IQPAE** algorithm.
- **ae_sine_integral.py**
- **my_benchmark_execution.py**
- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

The **ae_sine_integral.py**, **my_benchmark_execution.py** scripts in addition to the two folders (**QQuantLib** and **jsons**) deal with the execution of the **AE Benchmark**. Section A.1 documents, exhaustively, these files. The other script files are related to benchmark report generation and are properly explained in section A.2.

A.1. NEASQC implementation of BTC.

This section presents a complete description of the implementation of the **BTC** for **AE Kernel**. Subsection A.1.1 explains how to use the **QQuantLib** library, for computing the different mandatory operators and integrals shown in section 3.2. Subsection A.1.2 explains how the **ae_sine_integral.py** script is used for implementing the **BTC** as explained in section 3.2.7. Finally, subsection A.1.3 explains how to execute a complete **Benchmark**, as explained in section 3.3, using script **my_benchmark_execution.py**.

A.1.1. Computing integrals using the QQuantLib from NEASQC Financial Applications

As explained in section 3, the mathematical problem for the **BTC** of the **AE Kernel** is the computation of an integral, in a very well-defined interval (subsection 3.1), by using **AE** algorithms whose input is an operator **A**, given by equation (T02.2.20), which acts in the following form:

$$\mathbf{A}|0\rangle_n = \sqrt{a}|\Psi_0\rangle + \sqrt{1-a}|\Psi_1\rangle \tag{T02.1.1}$$

where the desired integral is encoded, as a Riemann sum, into the amplitude of the state $|\Psi_0\rangle$, a.

The **QQuantLib** library, from **NEASQC Financial Applications** software package, implements several Eviden myQLM-compatible **AE** algorithms that are used in quantum finances (Ferro et al., 2022; Gómez et al., 2022;

Nogueiras et al., 2021). Additionally, several functions of this library allow computing integrals, and the expected value of functions, using these **AE** algorithms. The implementation of the **BTC** for **AE Kernel** in the **tnbs/BTC_02_AE** folder of the **WP3_Benchmark** **NEASQC** GitHub repository is based on this **QQuantLib** library.

The following subsections explain the software implementation of the different parts of the **BTC**, using the **QQuantLib** library.

Operator $\mathbf{A}$ software implementation

As explained in section 3.2.4, given a properly normalised input array $f_{x_i}^{norm}$, the first step is the construction of the operator $\mathbf{A}$ using Eq.(T02.2.20) (the superscript I will be removed here for simplicity):

$$\mathbf{A}(f_{x_i}) = \left(\mathbb{I} \otimes H^{\otimes n}\right) \mathbf{U}_f \left(\mathbb{I} \otimes H^{\otimes n}\right)$$

where the operator $\mathbf{U}_f$ can be constructed following equation (T02.2.26):

$$\mathbf{U}_f \left(|0\rangle \otimes |i\rangle_n\right) = \left(\mathbf{R}_y(2 * \phi_{x_i})|0\rangle\right) \otimes |i\rangle_n$$

where $\phi_{x_i} = \arccos(f_{x_i}^{norm})$.

The $\mathbf{A}(f_{x_i})$ operator can be implemented using the Python class *Encoding* from *QQuantLib.DL.encoding_protocols* belonging to the **QQuantLib** library. This class implements up to 3 different encoding procedures (labelled as 0, 1, and 2) for loading a probability distribution and function as *Numpy* arrays into a quantum circuit. More information about this class can be found in the notebook 09_DataEncodingClass.ipynb of the **NEASQC Financial Applications** software package.

To reproduce the data encoding presented in section 3.2.4, the following arguments must be passed when the *Encoding* class is instantiated:

- *array_function*: NumPy array with the desired function to encode (this is $f_{x_i}^{norm}$).
- *array_probability*: NumPy array with the desired probability function to encode. In the benchmark case, this is the uniform distribution probability. This is the class's default distribution, so a *None* must be provided.
- *encoding*: For the benchmark case a 2 must be provided.

Once the class is instantiated correctly, the execution of its *run* method creates the myQLM implementation (a **QLM** *QRoutine*) of the operator $\mathbf{A}(f_{x_i})$, that is stored in the *oracle* property of the class. Listing T02.1 shows the use of the *Encoding* class. Figure T02.2 shows the circuit implementation of the *class_encoding.oracle* (the figure is the output of line 10 of the listing T02.1).

```
1   from QQuantLib.DL.encoding_protocols import Encoding
2   norm_f_x = np.array([0.17106865, 0.49847362, 0.78295039, 0.99999999])
3   class_encoding = Encoding(
4       array_function=norm_f_x,
5       array_probability = None,
6       encoding=2)
7   class_encoding.run()
8   #QLM circuit oracle implementation
9   oracle_circuit = class_encoding.oracle
10  %qatdisplay oracle_circuit --depth 1
```

*Listing T02.1: Creation of the $\mathbf{A}^I(f_{x_i})$ operator using the Encoding class from **QQuantLib** given an input Numpy array $f_{x_i}^{norm}$*

Following the guidelines introduced in section 3.2.5, the created $\mathbf{A}(f_{x_i})$ operator should be provided as an input of the **AE** algorithm for computing an estimation of the amplitude of the state $|\Psi_0\rangle$, $\tilde{a}$, as shown in equation (T02.2.22).

Amplitude Estimation Algorithms in QQuantLib

In general, most **AE** algorithms are based on a Grover-like operator created from the operator $\mathbf{A}(f_{x_i})$ using equation (T02.1.2)

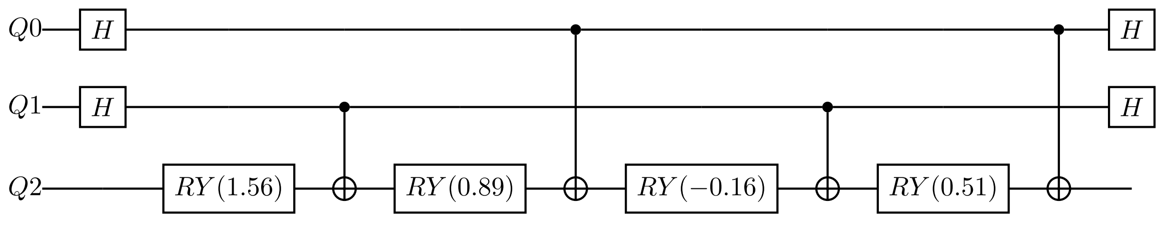


Figure T02.2: Circuit implementation for the class_encoding.oracle from listing T02.1.

$$\mathbf{G}(\mathbf{A}(\mathbf{f_{x_i}})) = \mathbf{A}(\mathbf{f_{x_i}}) \left(\mathbb{I} - 2|0\rangle\langle 0|\right) \mathbf{A}(\mathbf{f_{x_i}})^{\dagger} \left(\mathbb{I} - 2|\Psi_0\rangle\langle\Psi_0|\right) \tag{T02.1.2}$$

This operator acts as shown in equation (T02.1.3)

$$\mathbf{G}^k(\mathbf{A})\mathbf{A}|0\rangle_n = \sin\big((2k+1)\theta\big)|\Psi_0\rangle + \cos\big((2k+1)\theta\big)|\Psi_1\rangle \tag{T02.1.3}$$

The *Grover* function from the *QQuantLib.AA.amplitude_amplification* module belonging to the **QQuantLib** library allows to compute, given an input operator $\mathbf{A}(\mathbf{f_{x_i}})$, the corresponding Grover-like one in a straightforward way as shown in listing T02.2, where a Grover-like operator is created from the *Encoding* object created in the listing T02.1

```python
from QQuantLib.AA.amplitude_amplification import grover
grover_oracle = grover(
    oracle=class_encoding2.oracle,
    target=class_encoding2.target,
    index=class_encoding2.index
)
```

Listing T02.2: Creation of the correspondent Grover-like operator from an created operator $\mathbf{A}(f_{x_i})$.

Notebook 02_Amplitude_Amplification_Operators.ipynb of the **NEASQC Financial Applications** software package provides more information about this *Grover* function.

Note: Most **AE** algorithms rely on the Grover-like operator, but there are some algorithms where other operators are used. The **AE Benchmark** described in this document is agnostic about these operators. The only mandatory input is the **A** operator.

The *QQuantLib.AE* package provides five different myQLM implementations of **AE** algorithms which are provided in their corresponding modules:

- CQPEAE (*ae_classical_qpe* module) uses a classical phase estimation algorithm with QFT, see Figure T02.1, (Brassard et al., 2000). See notebook 04_Classical_Phase_Estimation_Class.ipynb for more information.
- IQPEAE (*ae_iterative_quantum_pe* module) uses an iterative implementation of QFT, using only one additional qubit, for classical phase estimation (Dobsicek et al., 2007; Kitaev, 1995). See notebook 05_Iterative_Quantum_Phase_Estimation_Class.ipynb for more information.
- MLAE (*maximum_likelihood_ae* module) uses a Maximum Likelihood algorithm (Suzuki et al., 2020). See notebook 03_Maximum_Likelihood_Amplitude_Estimation_Class.ipynb for more information.
- IQAE (*iterative_quantum_ae* module) uses an algorithm based on iterative applications of the Grover-like operator $\mathbf{G}(\mathbf{A}(\mathbf{f_{x_i}}))$ (Grinko et al., 2021). See notebook 06_Iterative_Quantum_Amplitude_Estimation_class.ipynb for more information.
- RQAE (*real_quantum_ae* module) uses an algorithm based on iterative applications of the Grover-like operator, $\mathbf{G}(\mathbf{A}(\mathbf{f_{x_i}}))$ (Manzano et al., 2023). See notebook 07_Real_Quantum_Amplitude_Estimation_class.ipynb for more information.
- MCAE (*montecarlo_ae* module) uses the *Monte Carlo* algorithm as presented in section 2.2.1. See notebook 08_AmplitudeEstimation_Class.ipynb for more information.

All these algorithms use the *Grover* function explained above to create the Grover-like operator. **QQuantLib** implements all the **AE** algorithms as Python classes that can be used similarly: each class should be instantiated passing the following parameters:

- oracle: A QLM AbstractGate or QRoutine with the implementation of the Oracle (the *oracle* property from a *Encoding* class can be used).
- target: This is the marked state in a binary representation as a Python list (the *target* property from a *Encoding* class can be used).
- index: a list of the qubits affected by the Oracle operator (the *index* property from a *Encoding* class can be used).
- kwargs: a typical Python *keyword arguments* where the different keywords can be used to configure different parameters of the **AE** algorithm. Configuration examples for the different **AE** algorithms can be found in the JSON files inside the jsons folder of the repository.

The *run* method of the class is then executed and the estimator $\tilde{a}$ is computed and returned using the properly configured **AE** algorithm (additionally it is stored in the *ae* property of the method).

Additionally, a class called *AE*, implemented in the module *ae_class* into the *QQuantLib.AE* package, can be used for selecting one of the available **AE** algorithms. In this case, another argument *ae_type* can be provided for selecting the algorithm, This argument is a Python string that can take the following values: [MLAE, CQPEAE, IQPEAE, IQAE, RQAE, MCAE] to select the appropriate **AE** algorithm. In this class, the desired estimator is stored as a Pandas DataFrame into the *ae_pdf* property of the class. Listing T02.3 shows how to use this class. The *Encoding* object from listing T02.1, is used for providing the mandatory $\mathbf{A}(\mathbf{f_{x_i}})$ operator to the class.

```

ae_dict = {
    #QPU
    'qpu': linalg_qpu,
    #Multi controlled decomposition
    'mcz_qlm': False,

    #shots
    'shots': 100,

    #MLAE
    'schedule': [None],
    'delta' : None,
    'ns' : None,

    #CQPEAE
    'auxiliar_qbits_number': 10,
    #IQPEAE
    'cbits_number': None,
    #IQAE & RQAQE
    'epsilon': None,
    #IQAE
    'alpha': None,
    #RQAE
    'gamma': None,
    'q': None
}
ae_object = AE(
    oracle=class_encoding.oracle,
    target=class_encoding.target,
    index=class_encoding.index,
    ae_type='CQPEAE',
    **ae_dict)
ae_object.run()
print(ae_object.ae_pdf)

#Result
ae   ae_l   ae_u
0 0.375536    NaN    NaN
```

Listing T02.3: Example of how to use the general AE class from QQuantLib.AE package. The Encoding class can be used for providing the mandatory $\mathbf{A}(\mathbf{f_{x_i}})$ operator in a transparent way.

The notebook 08_AmplitudeEstimation_Class.ipynb provides more information about this class.

Amplitude vs Probability estimation

In general, the **AE** algorithms compute the probability of the state $|\Psi_0\rangle$, instead of its amplitude. This is true for all the **AE** algorithms implemented in the **QQuantLib** library except for the **RQAE** one where the amplitude of such state is returned (thus it can be negative). Taking this into account, equation (T02.1.1) can be rewritten in the following form:

$$|\Psi\rangle = \mathbf{A}|0\rangle \otimes |0\rangle_n = \begin{cases} \sqrt{a}|\Psi_0\rangle + \sqrt{1-a}|\Psi_1\rangle & NO\ RQAE \\ a|\Psi_0\rangle + \beta|\Psi_1\rangle & RQAE \end{cases} \tag{T02.1.4}$$

As explained in section 3.2.4, only the $|\Psi_0\rangle = |0\rangle \otimes |0\rangle_n$ term must be taken into account, so using (T02.1.4) the equation (T02.2.22) now is transformed into (T02.1.5):

$$\tilde{a} = \begin{cases} \mathbf{P}_{|\Psi_0\rangle} = \left| \frac{1}{2^n} \sum_{i=0}^{2^n-1} f^{norm}_{x_i} \right|^2 & NO\ RQAE \\ \mathbf{Amplitude}_{|\Psi_0\rangle} = \frac{1}{2^n} \sum_{i=0}^{2^n-1} f^{norm}_{x_i} & RQAE \end{cases} \tag{T02.1.5}$$

So the desired sum in the **AE** estimation is:

$$\sum_{i=0}^{2^n-1} f^{norm}_{x_i} = \begin{cases} 2^n\sqrt{a} & NO\ RQAE \\ 2^n a & RQAE \end{cases} \tag{T02.1.6}$$

To use the **QQuantLib** implemented algorithms for the **BTC** of the **AE Kernel** (section 3.2.7), equation (T02.1.6) must be used for recovering the desired integral instead of (T02.2.22).

Fortunately the function *q_solve_integral* from *QQuantLib.finance.quantum_integration* module allows the computation of the integrals of input arrays by using *AE* techniques in a transparent way (so using this function the steps given in Sections 3.2.4 and 3.2.5 can be done easily). The *q_solve_integral* function computes the integral using (T02.1.6), taking into account the **AE** algorithm used, returning directly the integral of the input array function: $2^n\sqrt{a}$ (or the $2^n a$ for the **RQAE** algorithm).

The input of this function will be a Python *kwargs* where the different keys allow the complete configuration of the data encoding, the **AE** algorithm configuration etc... The most important keywords are:

- array_function: NumPy array with the desired array function for Riemann sum
- array_probability: Numpy array with a probability distribution for the computation of the expected values. In the benchmark case, this will be None (so a uniform distribution probability will be used),
- encoding: int for selecting the encoding. In the **BTC**, its value will be 2.
- ae_type: string for providing the **AE** algorithm for solving the desired integral.

The outputs of the *q_solve_integral* function are:

- ae_estimation: pandas DataFrame with the desired integral computation and the upper and lower limits if applied (depending on the **AE** algorithm).
- solver_ae: object based on the *AE* class

Listing T02.4 shows how to use *q_solve_integral* for computing the integral of an input numpy array by using 2 different **AE** algorithms straightforwardly.

```

from QQuantLib.finance.quantum_integration import q_solve_integral
norm_f_x = np.array([0.17106865, 0.49847362, 0.78295039, 0.99999999])

#Configuration of AE algorithm
ae_dict = {
    #QPU
    'qpu': linalg_qpu,
    #Multi controlled decomposition
    'mcz_qlm': False,

    #shots
```

```
        'shots': 100,

        #MLAE
        'schedule': [None],
        'delta' : None,
        'ns' : None,

        #CQPEAE
        'auxiliar_qbits_number': 10,
        #IQPEAE
        'cbits_number': None,
        #IQAE & RQAQE
        'epsilon': 0.001,
        #IQAE
        'alpha': 0.05,
        #RQAE
        'gamma': 0.05,
        'q': None,
    }

    #Important keywords configuration
    ae_dict.update(
        {"encoding" : 2,
         "ae_type" : "RQAE",
         "array_function":norm_f_x,
         "array_probability": None,
        })

    iqae_solution, iqae_object = q_solve_integral(**ae_dict)
    #second configure an RQAE algorithm

    ae_dict.update({"ae_type" : "RQAE",})

    rqae_solution, rqae_object = q_solve_integral(**ae_dict)

    print("Desired Riemann integral: {}".format(np.sum(norm_f_x)))
    print("IQAE integral computation: {}".format(iqae_solution["ae"].iloc[0]))
    print("RQAE integral computation: {}".format(rqae_solution["ae"].iloc[0]))

    #Results
    Desired Riemann integral: 2.45249265
    IQAE integral computation: 2.4531461375134835
    RQAE integral computation: 2.4534891011970776
```

Listing T02.4: Using q_solve_integral for solving the integral of an input array using the encoding procedure 2 and two different ***AE*** *algorithms.*

Finally, equation (T02.1.6) must be plugging into the Riemann sum (T02.2.8) for obtaining the desired integral as showed in equation (T02.1.7):

$$\tilde{S}_{[a,b]} = \begin{cases} \frac{\max(f_{x_i})(b-a)}{2^n} (2^n \sqrt{a}) & NO\ RQAE \\ \frac{\max(f_{x_i})(b-a)}{2^n} (2^n a) & RQAE \end{cases} \tag{T02.1.7}$$

A.1.2. ae_sine_integral.py

In this script the **BTC** for **AE Kernel**, as explained in the section 3.2.7, is implemented in the function *sine_integral*. This function needs as inputs:

- n_qbits: number of quits used for integral domain discretization.
- interval: for selecting with integration interval will be computed:
  - 0: $[0, \frac{3\pi}{8}]$
  - 1: $[\pi, \frac{5\pi}{4}]$
- ae_dictionary: python dictionary with the complete **AE** algorithm configuration.

The return of the function will be a pandas DataFrame with the complete information of the executed **BTC** (this includes the configuration of the **AE** algorithm, the used interval, the used **QPU** and the obtained results). The complete code is shown in listing T02.5.

```python

def sine_integral(n_qbits, interval, ae_dictionary):
    """
    Function for solving the sine integral between two input values:

    n_qbits : int
        for discretization of the input domain in 2ˆn intervals
    interval: int
        Interval for integration: Only can be:
            0 : [0,3pi/8]
            1 : [pi, 5pi/4]
    ae_dictionary : dict
        dictionary with the complete amplitude estimation
        algorithm configuration

    Return
    ----------

    pdf : pandas DataFrame
        DataFrame with the complete information of the benchmark
    """

    #Local copy for AE configuration dictionary
    ae_dictionary_ = deepcopy(ae_dictionary)

    start_time = time.time()

    #Section 2.1: Function for integration
    function = np.sin

    #Section 2.1: Integration Intervals
    start = [0.0, np.pi]
    end = [3.0*np.pi/8.0, 5.0*np.pi/4.0]
    if interval not in [0, 1]:
        raise ValueError("interval MUST BE 0 or 1")
    a_ = start[interval]
    b_ = end[interval]
    #Section 2.1: Computing exact integral
    exact_integral = np.cos(a_) - np.cos(b_)

    #Section 2.2: Domain discretization
    domain_x = np.linspace(a_, b_, 2 ** n_qbits + 1)

    #Section 2.3: Function discretization
    f_x = []
    for i in range(1, len(domain_x)):
        step_f = (function(domain_x[i]) + function(domain_x[i-1]))/2.0
        f_x.append(step_f)
        #x_.append((domain_x[i] + domain_x[i-1])/2.0)
    f_x = np.array(f_x)

    #Section 2.4: Array Normalisation
    normalization = np.max(np.abs(f_x)) + 1e-8
    f_norm_x = f_x/normalization

    #Sections 2.5 and 2.6: Integral computation using AE techniques

    #Section 3.2.3: configuring input dictionary for q_solve_integral
    q_solve_configuration = {
        "array_function" : f_norm_x,
        "array_probability" : None,
        "encoding" : 2
    }
    #Now added the AE configuration.
    #The ae_dictionary_ has a local copy of the AE configuration.
    q_solve_configuration.update(ae_dictionary_)
    #The q_solve_integral needs a QPU object.
    q_solve_configuration["qpu"] = get_qpu(q_solve_configuration["qpu"])
    #Compute the integral using AE algorithms!!
```

```
        solution, solver_object = q_solve_integral(**q_solve_configuration)

        #Section 3.2.3: eq (3.7). It is an adapatation of eq (2.22)
        estimator_s = normalization * (b_ - a_) * solution / (2 ** n_qbits)

        #Section 2.7: Getting the metrics
        absolute_error = np.abs(estimator_s["ae"] - exact_integral)
        relative_error = absolute_error / exact_integral
        oracle_calls = solver_object.oracle_calls

        end_time = time.time()
        elapsed_time = end_time - start_time

        #ae_dictionary_.pop('array_function')
        #ae_dictionary_.pop('array_probability')

        #Section 4.2: Creating the output pandas DataFrame for using
        #properly the KERNEL_BENCHMARK class

        #Adding the complete AE configuration
        pdf = pd.DataFrame([ae_dictionary_])

        #Adding information about the computed integral
        pdf["interval"] = interval
        pdf["n_qbits"] = n_qbits
        pdf["a_"] = a_
        pdf["b_"] = b_

        #Adding the output from q_solve_integral
        pdf = pd.concat([pdf, solution], axis=1)

        #Adding the AE computation of the integral
        integral_columns = ["integral_" + col for col in solution.columns]
        pdf[integral_columns] = estimator_s

        #Adding information about the integral that must be computed
        pdf["exact_integral"] = exact_integral
        pdf["riemann_sum"] = (b_ - a_) * np.sum(f_x) / (2 ** n_qbits)

        #Adding the normalization constant
        pdf["normalization"] = normalization

        #Error vs exact integral
        pdf["absolute_error_exact"] = absolute_error
        pdf["relative_error_exact"] = relative_error

        #Error vs Riemann Sum
        pdf["IntegralAbsoluteError"] = np.abs(pdf["integral_ae"] - pdf["riemann_sum"])

        #Error by Riemann approximation to Integral
        pdf["absolute_riemann_error"] = np.abs(
            pdf["riemann_sum"] - pdf["exact_integral"])
        pdf["oracle_calls"] = oracle_calls
        pdf["elapsed_time"] = elapsed_time
        pdf["run_time"] = solver_object.run_time
        pdf["quantum_time"] = solver_object.quantum_time

        #pdf will have a complete output for trazability.
        #Columns for the metric according to 2.7 and 2.8 will be:
        #[absolute_error_sum, oracle_calls,
        #elapsed_time, run_time, quantum_time]
        return pdf
```

*Listing T02.5: sine_integral function from **ae_sine_integral.py** script*

To configure properly the **AE** algorithm, the JSON files in the **tnbs/BTC_02_AE/jsons** can be edited and loaded as a dictionary that can be provided to the *sine_integral* function.

The **ae_sine_integral.py** script can be executed from the command line. Different arguments can be provided to properly configure the integral computation.

The usage guide of the *ae_sine_integral.py* script is obtained using the $-h$ parameter.

Listing T02.6 shows an example of how to use **ae_sine_integral.py** from the command line. In this case, the domain interval will be the 0, 4 qubits will be used for domain discretization, and the *Maximum Likelihood Amplitude Estimation* (**MLAE**) algorithm will be used for computing the integral. In this case, the algorithm is configured using the *json/integral_mlae_configuration.json* file.

```
python ae_sine_integral.py -n_qbits 4 -interval 0 -ae_type MLAE -qpu python --save --folder
 Results --exe
```

*Listing T02.6: Example of use the **ae_sine_integral.py** script from command line.*

A.1.3. my_benchmark_execution.py

This script is a modification of the correspondent template script located in tnbs/templates folder of the **WP3_Benchmark** repository. Following the recommendations of Annex 2 of the deliverable **D3.5: The NEASQC Benchmark Suite** the **run_code**, **compute_samples**, **summarize_results** and the **build_iterator** functions were modified. Meanwhile, the **KERNEL_BENCHMARK** class was not modified. In the following sections, the software adaptations for the **AE** benchmark case are presented.

run_code

Listing T02.7 shows the modifications performed into the **run_code** function for the **BTC** of the **AE Kernel**. The main functionality is executing a **BTC**, as explained in subsection 3.2.7, for a fixed number of qubits (*n_qbits*), provided by the first element of the input Python tuple *iterator_step*, and a fixed integration interval, given by the second element of *iterator_step*, a *repetitions* number of times, and gathering all the mandatory metrics obtained. As can be seen, this function calls to the *sine_integral*, listing T02.5, one.

```

def run_code(iterator_step, repetitions, stage_bench, **kwargs):
    """
    For configuration and execution of the benchmark kernel.

    Parameters
    ----------

    iterator_step : tuple
        tuple with elements from iterator built from build_iterator.
    repetitions : list
        number of repetitions for each execution
    stage_bench : str
        benchmark stage. Only: benchmark, pre-benchamrk
    kwargs : keyword arguments
        for configuration of the benchmark kernel

    Returns
    _______

    metrics : pandas DataFrame
        DataFrame with the desired metrics obtained for the integral computation
    save_name : string
        Desired name for saving the results of the execution

    """
    #if n_qbits is None:
    #    raise ValueError("n_qbits CAN NOT BE None")
    if stage_bench not in ['benchmark', 'pre-benchmark']:
        raise ValueError(
            "Valid values for stage_bench: benchmark or pre-benchmark'")
    if repetitions is None:
        raise ValueError("samples CAN NOT BE None")

    n_qbits = iterator_step[0]
    interval = iterator_step[1]

    from ae_sine_integral import sine_integral
    #Here the code for configuring and execute the benchmark kernel
    ae_configuration = kwargs.get("ae_configuration")
    print(ae_configuration)
    ae_configuration.update({"qpu": kwargs['qpu']})
```

```

        columns = [
            "interval", "n_qbits", "IntegralAbsoluteError", "oracle_calls",
            "elapsed_time", "run_time", "quantum_time"
        ]

        list_of_metrics = []
        for i in range(repetitions):
            metrics = sine_integral(n_qbits, interval, ae_configuration)
            list_of_metrics.append(metrics)
        metrics = pd.concat(list_of_metrics)
        metrics.reset_index(drop=True, inplace=True)

        if stage_bench == 'pre-benchmark':
            # Name for storing Pre-Benchmark results
            save_name = "pre_benchmark_nqubits_{}_integral_{}.csv".format(
                n_qbits, interval)
        if stage_bench == 'benchmark':
            # Name for storing Benchmark results
            save_name = kwargs.get('csv_results')
            #save_name = "pre_benchmark_step_{}.csv".format(n_qbits)
        return metrics[columns], save_name
```

Listing T02.7: run_code function for ***AE Benchmark***

compute_samples

Listing T02.8 shows the implementation of the **compute_samples** function for the **AE Benchmark**. The main objective is to codify a strategy for computing the number of times the **BTC**, subsection 3.2.7, should be executed, to get some desired statistical significance for the different metrics (see *3.a.i* and *3.a.ii* of section 3.3). This function would implement the equations (T02.3.27) and (T02.3.28) and computes the corresponding maximum as explained in *3.b* of section 3.3.

```
    def compute_samples(**kwargs):
        """
        This functions computes the number of executions of the benchmark
        for assure an error r with a confidence of alpha

        Parameters
        ----------

        kwargs : keyword arguments
            For configuring the sampling computation

        Returns
        _______

        samples : int
            Computed number of executions for desired significance

        """

        from scipy.stats import norm

        #Configuration for sampling computations

        #Desired Confidence level
        alpha = kwargs.get("alpha", None)
        if alpha is None:
            alpha = 0.05
        zalpha = norm.ppf(1-(alpha/2)) # 95% of confidence level

        #geting the metrics from pre-benchmark step
        metrics = kwargs.get("pre_metrics", None)
        #getting the configuration of the algorithm and kernel
        bench_conf = kwargs.get('kernel_configuration')

        #Code for computing the number of samples for getting the desired
        #statististical significance. Depends on benchmark kernel

```

```python
        #Desired Relative Error for the elapsed Time
        relative_error = bench_conf.get("relative_error", None)
        if relative_error is None:
            relative_error = 0.05
        # Compute samples for realtive error metrics:
        # Elapsed Time and Oracle Calls
        samples_re = (zalpha * metrics[['elapsed_time', 'oracle_calls']].std() / \
            (relative_error * metrics[['elapsed_time', 'oracle_calls']].mean()))**2

        #Desired Absolute Error.
        absolute_error = bench_conf.get("absolute_error", None)
        if absolute_error is None:
            absolute_error = 1e-4
        samples_ae = (zalpha * metrics[['IntegralAbsoluteError']].std() \
            / absolute_error) ** 2

        #Maximum number of sampls will be used
        samples_ = pd.Series(pd.concat([samples_re, samples_ae]).max())

        #Apply lower and higher limits to samples
        #Minimum and Maximum number of samples
        min_meas = kwargs.get("min_meas", None)
        if min_meas is None:
            min_meas = 5
        max_meas = kwargs.get("max_meas", None)

        samples_.clip(upper=max_meas, lower=min_meas, inplace=True)
        samples_ = samples_.max().astype(int)

        return samples_
```

Listing T02.8: compute_samples function for codifying the strategy for computing the number of repetitions for ***AE Benchmark****.*

summarize_results

Listing T02.9 shows the implementation of the **summarize_results** function for the **AE Benchmark**. The main objective is post-processing the results of the complete **Benchmark** execution, as described in step *3* of section 3.3.

This function expects that the results of the complete benchmark execution have been stored in a *csv* file. The function loads this file into a pandas DataFrame that is post-processed properly.

```python
    def summarize_results(**kwargs):
        """
        Create a summary with statistics
        """

        #Code to summarize the benchamark results. Depending of the
        #kernel of the benchmark
        folder = kwargs.get("saving_folder")
        csv_results = folder + kwargs.get("csv_results")

        #results = pd.DataFrame()
        pdf = pd.read_csv(csv_results, index_col=0, sep=";")
        pdf["classic_time"] = pdf["elapsed_time"] - pdf["quantum_time"]
        results = pdf.groupby(["interval", "n_qbits"]).describe()
        return results
```

Listing T02.9: summarize_results function for summarizing the results from a ***BTC*** *execution of the* ***AE Kernel***

build_iterator

Listing T02.10 shows the implementation of the **build_iterator** function for **AE Benchmark**. The main objective is to create a Python iterator for executing the desired complete **BTC**. In this case, the iterator creates a list with all the possible combinations of the desired number of qubits, n and the two integration intervals.

```python

    def build_iterator(**kwargs):
        """
        For building the iterator of the benchmark
        """
        import itertools as it

        list4int = [
            kwargs['list_of_qbits'],
            [0, 1],
        ]

        iterator = it.product(*list4int)
        return iterator
```

*Listing T02.10: build_iterator function for creating the iterator of the complete **BTC** execution of the **AE Kernel***

KERNEL_BENCHMARK class

No modifications were made to the **KERNEL_BENCHMARK** class. This Python class defines the complete **Benchmark** workflow, section 3.3, and its *exe* method executes it properly by calling the correspondent functions (*run_code*, *compute_samples*, *summarize_results*). Each time the **Benchmark** is executed, as defined in section 3.3, the result is stored in a given *CSV* file.

The only mandatory modification is configuring properly the input keyword arguments. These parameters will configure the **AE** algorithm, the complete benchmark workflow, and additional options (as the name of the *CSV* files).

Listing T02.11 shows an example for configuring the benchmark arguments for an **IQAE** algorithm. The json/integral_iqae_configuration.json file is used for **AE** algorithm configuration.

```
    if __name__ == "__main__":
        from ae_sine_integral import select_ae

        AE = "IQAE"
        #Setting the AE algorithm configuration
        ae_problem = select_ae(AE)

        ae_problem.update({
            "qpu": "c",
            "relative_error": None,
            "absolute_error": None
        })

        benchmark_arguments = {
            #Pre benchmark sttuff
            "pre_benchmark": True,
            "pre_samples": None,
            "pre_save": True,
            #Saving stuff
            "save_append" : True,
            "saving_folder": "./{}_Results/".format(AE),
            "benchmark_times": "{}_times_benchmark.csv".format(AE),
            "csv_results": "{}_benchmark.csv".format(AE),
            "summary_results": "{}_SummaryResults.csv".format(AE),
            #Computing Repetitions stuff
            "alpha": None,
            "min_meas": None,
            "max_meas": None,
            #List number of qubits tested
            "list_of_qbits": [4, 5],
        }


        json_object = json.dumps(ae_problem)
        #Writing the AE algorithm configuration
        conf_file = benchmark_arguments["saving_folder"] + \
            "benchmark_ae_conf.json"
        with open(conf_file, "w") as outfile:
            outfile.write(json_object)
        #Added ae configuration
        benchmark_arguments.update({
            "kernel_configuration": ae_problem,
        })
        ae_bench = KERNEL_BENCHMARK(**benchmark_arguments)
        ae_bench.exe()
```

*Listing T02.11: Example of configuration of a complete **Benchmark** execution. This part of the code should be located at the end of the **my_benchmark_execution.py** script*

As can be seen in Listing T02.11, the input dictionary that **KERNEL_BENCHMARK** class needs, *benchmark_arguments*, have several keys that allow to modify the benchmark workflow, like:

- *pre_benchmark*: For executing or not the *pre-benchmark* step.
- *pre_samples*: number of repetitions of the benchmark step.
- *pre_save*: For saving or not the results from the *pre-benchmark* step.
- *saving_folder*: Path for storing all the files generated by the execution of the **KERNEL_BENCHMARK** class.
- *benchmark_times*: name for the *csv* file where the initial and the final times for the complete benchmark execution will be stored.
- *csv_results*: name for the *csv* file where the obtained metrics for the different repetitions of the benchmark step will be stored (so the different metrics obtained during step 3 from section 3.3 will be stored in this file)
- *summary_results*: name for the *csv* file where the post-processed results (using the *summarize_results*) will be stored (so the statistics over the metrics obtained during step 3 of section 3.3 will be stored in this file)
- *alpha*: for configuring the desired confidence level α
- *min_meas*: For low limiting the number of executions a **BTC** should be executed during the benchmark stage.
- *max_meas*: For high limiting the number of executions a **BTC** should be executed during the benchmark stage.
- *list_of_qbits*: list with the different number of qubits for executing the complete **Benchmark**.

Additionally, the *kernel_configuration* key is used for configuring the **AE** algorithm and its execution. The function *select_ae* from **ae_sine_integral** script is used for loading the **JSON** files with the complete **AE** algorithm configuration. These **JSON** files can be found inside the jsons folder of the repository. Additionally following keys can be modified for configuring the **AE** algorithm execution:

- *qpu*: a string for selecting the quantum process unit (**QPU**).
- *relative_error*: for changing the desired relative error of the **Oracle calls** and **time elapsed** metrics.
- *absolute_error*: for changing the desired absolute error for the **Integral Absolute Error** metric.

In general, most of the keys should be fixed to *None* for executing the **Benchmark** according to the guidelines of the **AE Benchmark**.

As can be seen in the listing T02.11 the **AE** configuration is passed in the *benchmark_arguments* arguments under the key *ae_configuration*. The methods of the class send it as *kwargs* to the different functions of the scripts in a transparent way.

Listing T02.11 is located at the end of the **my_benchmark_execution.py** script. The different parts of the **Benchmark** complete execution and the **AE** algorithm used can be easily changed (the **AE** algorithm configuration can be changed by editing the correspondent JSON).

For executing the **Benchmark** following command should be used:

python my_benchmark_execution.py

Finally, it is worth commenting on lines 34-39 of the listing T02.11: here the configuration of the **AE** algorithm will be saved to a JSON format file for traceability purposes.

A.2. Generation of the benchmark report

Following deliverable **D3.5: The NEASQC Benchmark Suite** the results of a complete **Benchmark** must be reported in a separate JSON file that must satisfy the **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable. For automating this process the following files should be modified, as explained in Annex 2 of the deliverable **D3.5: The NEASQC Benchmark Suite**:

- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

my_environment_info.py

This script has the functions for gathering information about the hardware where the **Benchmark** is executed.

Listing T02.12 shows an example of the **my_environment_info.py** script. Here the compiled information corresponds to a classic computer because the case was simulated instead of executed in a quantum computer.

```
import platform
import psutil
from collections import OrderedDict

def my_organisation(**kwargs):
    """
    Given information about the organisation how uploads the benchmark
    """
    name = "CESGA"
    return name

def my_machine_name(**kwargs):
    """
    Name of the machine where the benchmark was performed
    """
    machine_name = platform.node()
    return machine_name

def my_qpu_model(**kwargs):
    """
    Name of the model of the QPU
    """
    qpu_model = "QLM"
    return qpu_model

def my_qpu(**kwargs):
    """
    Complete info about the used QPU
    """
    #Basic schema
    #QPUDescription = {
    #    "NumberOfQPUs": 1,
    #    "QPUs": [
    #        {
    #            "BasicGates": ["none", "none1"],
    #            "Qubits": [
    #                {
    #                    "QubitNumber": 0,
    #                    "T1": 1.0,
    #                    "T2": 1.00
    #                }
    #            ],
    #            "Gates": [
    #                {
    #                    "Gate": "none",
    #                    "Type": "Single",
    #                    "Symmetric": False,
    #                    "Qubits": [0],
    #                    "MaxTime": 1.0
    #                }
    #            ],
    #            "Technology": "other"
    #        },
    #    ]
    #}

    #Defining the Qubits of the QPU
```

```python
        qubits = OrderedDict()
        qubits["QubitNumber"] = 0
        qubits["T1"] = 1.0
        qubits["T2"] = 1.0

        #Defining the Gates of the QPU
        gates = OrderedDict()
        gates["Gate"] = "none"
        gates["Type"] = "Single"
        gates["Symmetric"] = False
        gates["Qubits"] = [0]
        gates["MaxTime"] = 1.0


        #Defining the Basic Gates of the QPU
        qpus = OrderedDict()
        qpus["BasicGates"] = ["none", "none1"]
        qpus["Qubits"] = [qubits]
        qpus["Gates"] = [gates]
        qpus["Technology"] = "other"

        qpu_description = OrderedDict()
        qpu_description['NumberOfQPUs'] = 1
        qpu_description['QPUs'] = [qpus]

        return qpu_description

    def my_cpu_model(**kwargs):
        """
        model of the cpu used in the benchmark
        """
        cpu_model = platform.processor()
        return cpu_model

    def my_frecuency(**kwargs):
        """
        Frcuency of the used CPU
        """
        #Use the nominal frequency. Here, it collects the maximum frequency
        #print(psutil.cpu_freq())
        cpu_frec = psutil.cpu_freq().max/1000
        return cpu_frec

    def my_network(**kwargs):
        """
        Network connections if several QPUs are used
        """
        network = OrderedDict()
        network["Model"] = "None"
        network["Version"] = "None"
        network["Topology"] = "None"
        return network

    def my_QPUCPUConnection(**kwargs):
        """
        Connection between the QPU and the CPU used in the benchmark
        """
        #
        # Provide the information about how the QPU is connected to the CPU
        #
        qpuccpu_conn = OrderedDict()
        qpuccpu_conn["Type"] = "memory"
        qpuccpu_conn["Version"] = "None"
        return qpuccpu_conn
```

*Listing T02.12: Example of configuration of the **my_environment_info.py** script*

In general, it is expected that for each computer used (quantum or classic), the **Benchmark** developer should change this script to properly get the hardware info.

A.2.1. my_benchmark_info.py

This script gathers the information under the field *Benchmarks* of the **Benchmark** report. Information about the software, the compilers and the results obtained from an execution of the **Benchmark** is stored in this field.

Listing T02.13 shows an example of the configuration of the **my_benchmark_info.py** script for gathering the aforementioned information.

```python
import platform
import psutil
import sys
import json
import jsonschema
import pandas as pd
from collections import OrderedDict
from my_benchmark_summary import summarize_results

def my_benchmark_kernel(**kwargs):
    """
    Name for the benchmark Kernel
    """
    return "AmplitudeEstimation"

def my_starttime(**kwargs):
    """
    Providing the start time of the benchmark
    """
    #start_time = "2022-12-12T16:46:57.268509+01:00"
    times_filename = kwargs.get("times_filename", None)
    pdf = pd.read_csv(times_filename, index_col=0)
    start_time = pdf["StartTime"][0]
    return start_time

def my_endtime(**kwargs):
    """
    Providing the end time of the benchmark
    """
    #end_time = "2022-12-12T16:46:57.268509+01:00"
    times_filename = kwargs.get("times_filename", None)
    pdf = pd.read_csv(times_filename, index_col=0)
    end_time = pdf["EndTime"][0]
    return end_time

def my_timemethod(**kwargs):
    """
    Providing the method for getting the times
    """
    #time_method = "None"
    time_method = "time.time"
    return time_method

def my_programlanguage(**kwargs):
    """
    Getting the programing language used for benchmark
    """
    #program_language = "None"
    program_language = platform.python_implementation()
    return program_language

def my_programlanguage_version(**kwargs):
    """
    Getting the version of the programing language used for benchmark
    """
    #language_version = "None"
    language_version = platform.python_version()
    return language_version

def my_programlanguage_vendor(**kwargs):
    """
    Getting the version of the programing language used for benchmark
    """
    #language_vendor = "None"
    language_vendor = "Unknow"
```

```
        return language_vendor

    def my_api(**kwargs):
        """
        Collect the information about the used APIs
        """
        #api = OrderedDict()
        #api["Name"] = "None"
        #api["Version"] = "None"
        #list_of_apis = [api]
        modules = []
        list_of_apis = []
        for module in list(sys.modules):
            api = OrderedDict()
            module = module.split('.')[0]
            if module not in modules:
                modules.append(module)
                api["Name"] = module
                try:
                    version = sys.modules[module].__version__
                except AttributeError:
                    #print("NO VERSION: "+str(sys.modules[module]))
                    try:
                        if  isinstance(sys.modules[module].version, str):
                            version = sys.modules[module].version
                            #print("\t Attribute Version"+version)
                        else:
                            version = sys.modules[module].version()
                            #print("\t Methdod Version"+version)
                    except (AttributeError, TypeError) as error:
                        #print('\t NO VERSION: '+str(sys.modules[module]))
                        try:
                            version = sys.modules[module].VERSION
                        except AttributeError:
                            #print('\t\t NO VERSION: '+str(sys.modules[module]))
                            version = "Unknown"
                api["Version"] = str(version)
                list_of_apis.append(api)
        return list_of_apis

    def my_quantum_compilation(**kwargs):
        """
        Information about the quantum compilation part of the benchmark
        """
        q_compilation = OrderedDict()
        q_compilation["Step"] = "None"
        q_compilation["Version"] = "None"
        q_compilation["Flags"] = "None"
        return [q_compilation]

    def my_classical_compilation(**kwargs):
        """
        Information about the classical compilation part of the benchmark
        """
        c_compilation = OrderedDict()
        c_compilation["Step"] = "None"
        c_compilation["Version"] = "None"
        c_compilation["Flags"] = "None"
        return [c_compilation]

    def my_metadata_info(**kwargs):
        """
        Other important info user want to store in the final json.
        """

        metadata = OrderedDict()
        #metadata["None"] = None

        json_file = kwargs.get("ae_config")
        with open(json_file, 'r') as openfile:
            #Reading from json file
            json_object = json.load(openfile)

        for key, value in json_object.items():
            metadata[key] = value
        return metadata

    def my_benchmark_info(**kwargs):
        """
        Complete WorkFlow for getting all the benchmar informated related info
        """
        benchmark = OrderedDict()
        benchmark["BenchmarkKernel"] = my_benchmark_kernel(**kwargs)
        benchmark["StartTime"] = my_starttime(**kwargs)
        benchmark["EndTime"] = my_endtime(**kwargs)
        benchmark["ProgramLanguage"] = my_programlanguage(**kwargs)
        benchmark["ProgramLanguageVersion"] = my_programlanguage_version(**kwargs)
        benchmark["ProgramLanguageVendor"] = my_programlanguage_vendor(**kwargs)
        benchmark["API"] = my_api(**kwargs)
        benchmark["QuantumCompililation"] = my_quantum_compilation(**kwargs)
        benchmark["ClassicalCompiler"] = my_classical_compilation(**kwargs)
        benchmark["TimeMethod"] = my_timemethod(**kwargs)
        benchmark["Results"] = summarize_results(**kwargs)
        benchmark["MetaData"] = my_metadata_info(**kwargs)
        return benchmark
```

Listing T02.13: Example of configuration of the ***my_benchmark_info.py*** *script*

The *my_benchmark_info* function gathers all the mandatory information needed by the *Benchmarks* main field of the report (by calling the different functions listed in listing T02.13). To properly fill this field some mandatory information must be provided as the typical *python kwargs*:

- *times_filename*: This is the complete path to the file where the starting and ending time of the **Benchmark** was stored. This file must be a *csv* one and it is generated when the **KERNEL_BENCHMARK** class is executed. This information is used by the *my_starttime* and *my_endtime* functions.
- *ae_config*: complete path where the configuration of the **AE** algorithm used in the **Benchmark** (in JSON format) is stored (see last paragraph of section A.1.3). This information is used by the *my_metadata_info* function for filling the *MetaData* sub-field of *Benchmarks* main field of the report. This field is not mandatory, following the JSON schema **NEASQC**, but it is important to get good traceability of the **Benchmark** results.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark** are stored. This information is used by the *summarize_results* function from *my_benchmark_summary.py* script (see section A.2.2).

A.2.2. my_benchmark_summary.py

In this script, the *summarize_results* function is implemented. This function formats the results of a complete execution of a **AE Benchmark** with a suitable **NEASQC** benchmark report format. It can be used for generating the information under the sub-field *Results* of the main field *Benchmarks* in the report.

Listing T02.14 shows an example of implementation of *summarize_results* function for the **AE Benchmark** procedure.

```

    def summarize_results(**kwargs):
        """
        Mandatory code for properly present the benchmark results following
        the NEASQC jsonschema
        """

        #Info with the benchmark results like a csv or a DataFrame
        import pandas as pd
        #pdf = None
        benchmark_file = kwargs.get("benchmark_file")
        pdf = pd.read_csv(benchmark_file, header=[0, 1], index_col=[0, 1])
        pdf.reset_index(inplace=True)
        #n_qbits = [4]
        n_qbits = list(set(pdf["n_qbits"]))
        intervals = list(set(pdf["interval"]))

        #Metrics needed for reporting. Depend on the benchmark kernel
        #list_of_metrics = ["MRSE"]
        list_of_metrics = [
        #    "absolute_error_exact", "relative_error_exact",
            "IntegralAbsoluteError", "oracle_calls"
        ]

        results = []
        #If several qbits are tested
        for n_ in n_qbits:
            #Fields for benchmark test of a fixed number of qubits
            #For each qubit 2 different integration interval is tested
            for interval in intervals:
                result = OrderedDict()
                result["NumberOfQubits"] = n_
                result["QubitPlacement"] = list(range(n_))
                result["QPUs"] = [1]
                result["CPUs"] = psutil.Process().cpu_affinity()
                #Getting the data for n_qbiotd and interval
                step_pdf = \
                    pdf[(pdf["interval"] == interval) & (pdf["n_qbits"] == n_)]
                #result["TotalTime"] = 10.0
                #result["SigmaTotalTime"] = 1.0
                #result["QuantumTime"] = 9.0
                #result["SigmaQuantumTime"] = 0.5
                #result["ClassicalTime"] = 1.0
                #result["SigmaClassicalTime"] = 0.1
                result["TotalTime"] = step_pdf["elapsed_time"]["mean"].iloc[0]
                result["SigmaTotalTime"] = step_pdf["elapsed_time"]["std"].iloc[0]
                result["QuantumTime"] = step_pdf["quantum_time"]["mean"].iloc[0]
                result["SigmaQuantumTime"] = step_pdf["quantum_time"]["std"].iloc[0]
                result["ClassicalTime"] = step_pdf["classic_time"]["mean"].iloc[0]
                result["SigmaClassicalTime"] = step_pdf["classic_time"]["std"].iloc[0]
                #For identify integration interval info. Not mandaatory but
                #useful for indentify results
                result["Interval"] = interval
                metrics = []
                #For each fixed number of qbits several metrics can be reported
                for metric_name in list_of_metrics:
                    metric = OrderedDict()
                    #MANDATORY
                    metric["Metric"] = metric_name
                    #metric["Value"] = 0.1
                    #metric["STD"] = 0.001
                    metric["Value"] = step_pdf[metric_name]["mean"].iloc[0]
                    metric["STD"] = step_pdf[metric_name]["std"].iloc[0]
                    #Depending on the benchmark kernel
                    metric["MIN"] = step_pdf[metric_name]["min"].iloc[0]
                    metric["MAX"] = step_pdf[metric_name]["max"].iloc[0]
                    metric["COUNT"] = step_pdf[metric_name]["count"].iloc[0]
                    metrics.append(metric)
                result["Metrics"] = metrics
                results.append(result)
        return results
```

*Listing T02.14: Example of configuration of the summarize_results function for **AE** benchmark*

As usual, the *kwargs* strategy is used for passing the arguments that the function can use. In this case, the only mandatory argument is *benchmark_file* with the path to the file where the summary results of the **Benchmark** execution were stored.

Table T02.2 shows the sub-fields and the information stored, under the *Results* field. To have proper traceability of the executions the sub-field *interval* was created explicitly for the **AE Benchmark**.

The sub-field *Metrics* gathers information about the obtained metrics of the benchmark. Table T02.3 shows its different sub-fields and the information stored. The **Integral Absolute Error** is stored under the name *IntegralAbsoluteError*, and the number of **oracle calls** is under the name *oracle_calls*.

A.2.3. neasqc_benchmark.py

The *neasqc_benchmark.py* script can be used straightforwardly for gathering all the **Benchmark** execution information and results, for creating the final mandatory **NEASQC** benchmark report.

sub-field	information
NumberOfQubits	number of qubits, n
TotalTime	mean of **elapsed time**
SigmaTotalTime	standard deviation of **elapsed time**
QuantumTime	mean of the **quantum time**
SigmaQuantumTime	standard deviation of **quantum time**
ClassicalTime	mean of the **classical time**
SigmaClassicalTime	standard deviation of **classical time**
interval	integration interval (can be 0 or 1)
Metrics	summarize verification metrics. See Table T02.3

*Table T02.2: Sub-fields of the Results fields of the **TNBS** benchmark report.*

sub-field	information
metric	*IntegralAbsoluteError*, *oracle_calls*
Value	mean value of the metric
STD	standard deviation of the metric
Count	number of samples for computing the statistics of the metric

Table T02.3: Sub-fields of the Metrics field.

It does not necessarily change anything about the class implementation. It is enough to update the information of the *kwargs* arguments for providing the mandatory files for gathering all the information.

In this case, the following information should be provided as arguments for the *exe* method of the **BENCHMARK** class:

- *times_filename*: complete path where the file with the times of the **Benchmark** execution was stored.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark** execution was stored.
- *ae_config*: complete path where the configuration of the **AE** algorithm used in the benchmark (in JSON format) was stored.

E.T03: Benchmark for Phase Estimation Algorithms

NExt ApplicationS of Quantum Computing
Benchmark Suite

T03: Benchmark for Phase Estimation Algorithms

Document Properties

Contract Number	951821
Contractual Deadline	31/10/2023
Dissemination Level	Public
Nature	Test Case Definition
Editors	Gonzalo Ferro, CESGA
Authors	Gonzalo Ferro, CESGA Oluwatosin Esther, CITIC-UDC Andrés Gómez, CESGA
Reviewers	Cyril Allouche, EVIDEN Arnaud Gazda, EVIDEN
Date	27/10/2023
Category	[Generic]
Keywords	
Status	Submitted
Release	1.0

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, Organisation	Description of Changes
0.1	24/07/2023	Gonzalo Ferro, CESGA Oluwatosin Esther, CITIC-UDC	Quantum Phase Estimation Benchmark
0.2	25/08/2023	Gonzalo Ferro, CESGA	Quantum Phase Estimation Benchmark
0.3	08/10/2023	Andrés Gómez, CESGA	Formatting
0.4	25/10/2023	Gonzalo Ferro, CESGA	Fixing the naming according to the Glossary of deliverable 3.5

Table of Contents

1.Introduction

This section describes the T3: Phase Estimation benchmark of The NEASQC Benchmarking Suite (**TNBS**). This document must be read accompanied by the document that describes the TNBS: D3.5: The NEASQC Benchmark Suite.

Section 2 describes the **Quantum Phase Estimation kernel**, referred as **QPE Kernel** along the document. With each **TNBS** a **Benchmark Test Case** (**BTC**) must be designed and documented. This is done in Section 3. Finally, the benchmarking methodology provides a reference implementation of the **BTC** using the Eviden myQLM library. This implementation is described in Annex A.

2.Description of the Kernel: Quantum Phase Estimation

The present section describes the **QPE Kernel** for the **TNBS**. Section 2.1 justifies **Kernel** selection according to the **TNBS** benchmarking methodology meanwhile section 2.2 presents a complete description of this **QPE Kernel**.

2.1. Kernel selection justification

The **QPE Kernel** (Jiang et al., 2021) is utilized to estimate the eigenvalues (or phases) of a unitary operator. The **QPE Kernel** implements a measurement for any Hermitian operator and serves as a key component in numerous quantum algorithms, such as Shor's algorithm, the quantum algorithm for solving linear systems of equations, or the measurement of the energy of a Hamiltonian in Variational Quantum Eigensolver (VQE) algorithms (Cao et al., 2019). So the **QPE Kernel** can be considered as an interesting candidate for **TNBS**. Additionally, it satisfies the three main requirements from the **NEASQC** benchmark methodology:

1. A mathematical definition of the **Kernel** can be provided with sufficient accuracy to enable the construction of a standalone circuit (refer to sections 2.2 and 3).
2. The **Kernel** can be defined for a configurable number of qubits.
3. The output can be verified through a classical computation (see section 3.2).

2.2. Kernel Description

The **QPE Kernel** can be defined in the following way:

Let U be an n-qubit unitary operator. The eigenvalues of this operator are phases that can be represented as $e^{2i\pi\lambda_j}$ for $j = 0, 1, 2, \cdots 2^n - 1$. For a particular *eigenstate* $|\psi_j\rangle$:

$$U|\psi_j\rangle = e^{2i\pi\lambda_j}|\psi_j\rangle \tag{T03.2.1}$$

where $0 \leq \lambda_j < 1$. The objective of the **QPE Kernel** is to obtain an estimation, up to a finite level, of the different eigenvalues λ_j.

The canonical solution, from a quantum computing perspective, is to use the **Quantum Phase Estimation** algorithm (Kitaev, 1995) whose circuit implementation is shown in Figure T03.1.

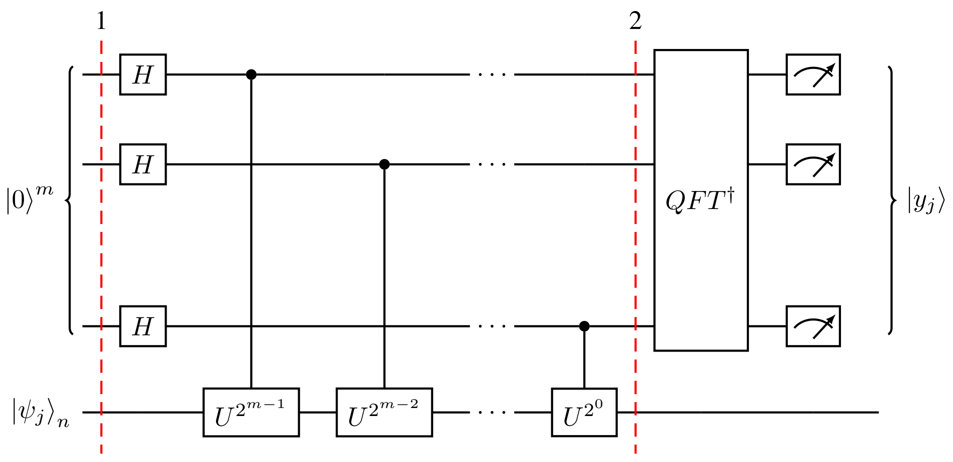


Figure T03.1: Canonical QPE circuit.

This algorithm operates on two different registers: a n qubits one, initialized to the eigenstate $|\psi_j\rangle$, and a m qubits one to estimate the phase. Thus, the total number of qubits will be $n + m$.

The initial state is set to (dashed line 1 in Figure T03.1):

$$|\psi_j\rangle \otimes |\mathbf{0}\rangle \tag{T03.2.2}$$

where $|\mathbf{0}\rangle = |0\rangle^{\otimes m}$.

As shown in Figure T03.1, a sequence of controlled-U^{2j} operations, where the j-th control qubit is the $m-j$-th counting qubit, is applied over the state $|\psi_j\rangle$ (i.e. over the n qubits register)

This operation maps the initial state to the state (dashed line 2 in Figure T03.1):

$$\frac{1}{\sqrt{2^m}}\left(|0\rangle + e^{2\pi i 2^{m-1}\lambda_j}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i 2^{-1}\lambda_j}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 2^{0}\lambda_j}|1\rangle\right) \otimes |\psi_j\rangle \tag{T03.2.3}$$

where λ_j is the unknown eigenvalue correspondent to the eigenstate $|\psi_j\rangle$. After applying the complex conjugate of the **Quantum Fourier Transformation** (operator $QFT^\dagger$ in Figure T03.1) on the m qubits register, they are measured and the resulting binary string $|y_j\rangle$ is obtained. Then it can be transformed to an estimator $\tilde{\lambda}_j$ of the eigenvalue λ_j by (T03.2.4).

$$\tilde{\lambda}_j = \frac{y_j}{2^m} \tag{T03.2.4}$$

3.Description of the benchmark test case

This section presents the complete description of the **BTC** for the **QPE Kernel**. Section 3.1 describes the problem addressed by the test case. Section 3.2 provides a high-level description of the **BTC**. Section 3.3 provides the **Benchmark** execution workflow. Finally, section 3.4 documents how the results of such executions must be reported.

3.1. Description of the problem

The computation of the eigenvalues of a n qubits unitary operator $R_z^n(\vec{\theta}) = \otimes_{i=1}^n R_z(\theta_i)$, for a vector of n angles $\vec{\theta} = \{\theta_i\}\ i = 0, 1, \cdots, n-1$ is the proposed **BTC** for the **QPE Kernel**.

The $R_z(\theta)$ operator is a Z-axis rotation gate given by (T03.1.1).

$$R_z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix} = e^{-i\frac{\theta}{2}} |0\rangle \langle 0| + e^{i\frac{\theta}{2}} |1\rangle \langle 1| \quad \text{(T03.1.1)}$$

The circuit implementation of the proposed problem is shown in Figure T03.2

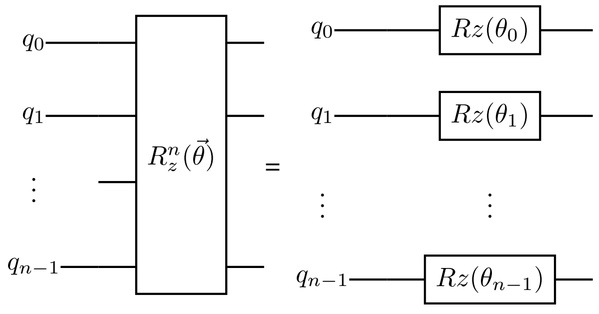


Figure T03.2: Circuit implementation of the $R_z^n(\vec{\theta})$ operator

Two different set of $\vec{\theta}$ angles will be used:

1. **Random**: in this case the n angles will be selected randomly between the interval $[0, \pi]$
2. **Exact**: in this case, the angle selection will depend on the number of auxiliary qubits for the **QPE** m and is given by (T03.1.2) where $\delta\theta = \frac{4\pi}{2^m}$ and a will be a binary random variable that can be $\{-1, 1\}$

$$\theta_{i+1} = \theta_i + a * \delta\theta \quad \text{(T03.1.2)}$$

A fixed QPE implementation must be used to compute the probabilities of the different eigenvalues of the $R_z^n(\vec{\theta})^n$ operator, for the two aforementioned cases, and these probabilities must be compared with the theoretical probability distribution using various metrics.

3.2. Benchmark test case description

This section provides a step-by-step workflow of the **BTC** for the **QPE Kernel**. The inputs are the number of qubits n, the number of auxiliary qubits (or discretization parameter[1]) m, see Figure T03.1, and a set of angles built following the **random** or the **exact** method (provided at the end of Section 3.1).

The following 7 steps should be followed:

[1]In the canonical solution, m sets the precision of the destination ($\frac{1}{2^m}$). To avoid linking the benchmark to a fixed implementation this m will be used for the benchmark as a discretization parameter for comparing histograms.

1. **Generation of the reference probability distribution of the eigenvalues.** The theoretical probabilities must be computed $P^{th}_{\lambda,m}(\frac{k}{2^m})$ where $k = 0, 1, \cdots 2^m - 1$ following the next steps:

   a) Compute each state of the 2^n possible states as a binary string: $|i_0 i_1 \cdots i_{n-1}\rangle$ with $i_j = \{0, 1\}$

   b) For each state compute the associated eigenvalue λ_j using (T03.2.3):

$$\lambda_j = \frac{\sum_k^n (-1)^{i_k} \theta_k}{4\pi} \tag{T03.2.3}$$

   c) So for each possible state $|j\rangle$ an eigenvalue λ_j should be computed. For the angles from the **exact** case some eigenvalues can have degeneracy (i.e. can occur several times). In this case, the less frequent eigenvalue (λ_{lf}) and its frequency ($f_{\lambda_{lf}}$) should be stored. This is more unlikely for the **random** case (in this case, with very high probability $f_{\lambda_{lf}} = 1$).

   d) Compute a histogram for the complete list of eigenvalues λ^i according to the following guidelines:

      - The number of histogram bins will be 2^m.
      - The range of the histogram will be $[0, 1]$
      - The frequency of eigenvalues in each bin k should be computed: f_{λ_k}
      - Each bin must be labelled as $\frac{k}{2^m}$ where k is the number of the bin ($k = 0, 1, 2, \cdots 2^m - 1$)

   e) This histogram must be used to build a theoretical discrete probability distribution of the eigenvalues: $P^{th}_{\lambda,m}(\frac{k}{2^m}) = f_{\lambda_k}$. Figure T03.3 shows an example of the theoretical probability distribution of the eigenvalues, for a $n = 7$ qubits $R^n_z(\theta)$ operator with a discretization parameter of $m = 7$ for the **exact** case.

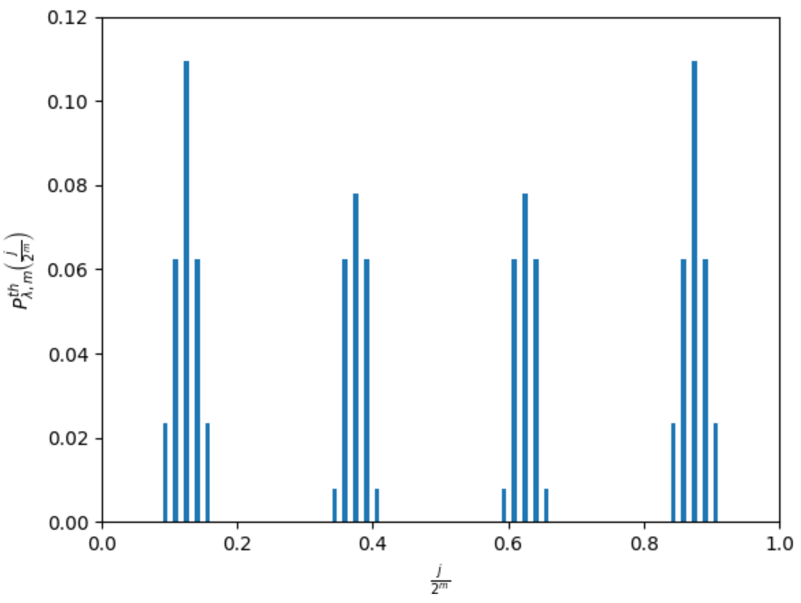


*Figure T03.3: Example of a histogram showing the theoretical probability distribution of the eigenvalues for a $n = 7$ qubits operator $R_z(\theta)^n$ with a parameter discretization $m = 7$ for the **exact** case.*

2. Compute the number of shots, n_{shots} that will be used for executing the **QPE** routine in the quantum device. For this computation, we want a number of shots that ensures that the less frequent eigenvalue should be measured at least 1000 times. This condition is satisfied by fixing the number of shots to (T03.2.4) where $f_{\lambda_{lf}}$ is the frequency of the less frequent eigenvalue.

$$n_{shots} = \frac{1000}{0.81 * f_{\lambda_{lf}}} \tag{T03.2.4}$$

3. Create the operator $R_z^n(\vec{\theta})$ with the input angles.
4. Create an initial state for the QPE routine that is an equiprobable combination of all the 2^n possible states:

$$|\psi_0\rangle = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{n-1} |i\rangle$$

5. Provide the $R_z^n(\vec{\theta})$ operator and the initial state $|\psi_0\rangle$ to the **QPE** routine, execute it and measure the eigenvalue. This should be done n_{shots} times. So a complete list of n_{shots} **QPE** eigenvalues is generated: λj^{QPE} where $j = 1, 2, \cdots n_{shots}$
6. The list of eigenvalues generated by the **QPE** (λj^{QPE}) must be used to draw a second histogram using the same procedure as in step **1.d**. This generates the measured probability distribution of the values generated by the **QPE**: $P^{QPE}_{\lambda,m}(\frac{k}{2^m})$ with $k = 0, 1, \cdots 2^m - 1$.
7. The two discrete probability distributions, $P^{th}_{\lambda,m}$ and $P^{QPE}_{\lambda,m}$, must be compared using the following metrics:
   - The **Kolmogorov-Smirnov** (**KS**) between $P^{th}_{\lambda,m}$ and $P^{QPE}_{\lambda,m}$. This is the maximum of the absolute difference between the cumulative distribution functions of $P^{th}_{\lambda,m}$ and $P^{QPE}_{\lambda,m}$ computed following (T03.2.5), This will be the comparative metric for the angles from **random** case.

$$KS = \max\left(\left| \sum_{k=0}^{i} P^{th}_{\lambda,m}(\frac{k}{2^m}) - \sum_{k=0}^{i} P^{QPE}_{\lambda,m}(\frac{k}{2^m}) \right|, \forall k = 0, 1, \cdots, 2^m - 1 \right) \qquad \text{(T03.2.5)}$$

   - The **fidelity**. In this case the 2 distributions ($P^{th}_{\lambda,m}$ and $P^{QPE}_{\lambda,m}$) will be considered as vectors and the **fidelity** will be the cosine of the angle between them that will be computed following (T03.2.6). $|P^{th}_{\lambda}|$ and $|P^{QPE}_{\lambda}|$ will be the norm of the vectors corresponding to the *theoretical* and *QPE* distributions respectively. This will be the comparative metric for the angles from **exact** case.

$$fidelity = \frac{\sum_{k=0}^{n-1} P^{th}_{\lambda,m}(\frac{k}{2^m}) * P^{QPE}_{\lambda,m}(\frac{k}{2^m})}{|P^{th}_{\lambda}| * |P^{QPE}_{\lambda}|} \qquad \text{(T03.2.6)}$$

The time from steps 2 to 7 should also be measured and labelled as the **elapsed time**. If possible, the time of the pure quantum part, step 5, should be calculated separately as the **quantum time**.

3.3. Complete benchmark procedure

To execute a complete **QPE Benchmark** the next procedure must be followed:

1. We must fix in advance the different numbers of qubits to be tested (for example from n=4 to n=8).
2. For each number of qubits n, several numbers of auxiliary qubits (or discretization parameter) m should be tested, in general it is recommended that $m \geq n$. For example, m can range between 4 and 12 in steps of 2.
3. For each possible combination of number of qubits n, discretization parameter m and the two different selection methods for the angles, **random** and **exact**, following steps must be performed:
   a) Execute a warm-up step consisting of:
      i. Execute 10 iterations of the **BTC**, as explained in section 3.2, and compute the standard deviation (σ_{metric}) for the used metric of the angle selection method ($metric =$ **KS** for **random** and $metric =$ **fidelity** for the **exact** case). Additionally, compute the mean and the standard deviation for the **elapsed time**, μ_t, σ_t
      ii. Compute the number of repetitions, M_{metric}, using equation (T03.3.7). Where r_{metric} will be the desired absolute error for the metric of the angle selection method: $r_{KS} = 0.05$ for **exact** case and

$r_{fidelity} = 0.001$ for the **random** case. $Z_{1-\frac{\alpha}{2}}$ is the percentile for $\alpha = 0.95$

$$M = (\frac{\sigma_{metric} Z_{1-\frac{\alpha}{2}}}{r_{metric}})^2 \tag{T03.3.7}$$

iii. Compute the number of repetitions mandatory, M_t, for having a relative error for the **elapsed time** of 5%, $r_t = 0.05$, with a confidence level of 95%, $\alpha = 0.05$, following (T03.3.8), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for α :

$$M_T = (\frac{\sigma_t Z_{1-\frac{\alpha}{2}}}{r\mu_t})^2 \tag{T03.3.8}$$

b) Execute the complete **BTC**, section 3.2, $M = \max(M_t, M_{metric})$ times. M must be greater than 5.

c) Compute the mean, the standard deviation (σ) and the standard deviation of the mean ($\frac{\sigma}{\sqrt{M}}$) for the **elapsed time**, **quantum time**, if possible, and for the mentioned metrics in step 7 of section 3.2: **KS** and **fidelity**.

The method used to calculate the number of repetitions M in the previous procedure guarantees that the desired metric (**KS** for **random** and **fidelity** for **exact**) will have a relative error lower than its correspondent r_{metric} and the **elapsed time** one will have a relative error lower than 5% with a confidence level of 95%, in both cases.

3.4. Benchmark report

Finally, the results of the complete benchmark execution must be reported into a valid JSON file following the JSON schema *NEASQC.Benchmark.V2.Schema.json* provided in the document D3.5: The NEASQC Benchmark Suite of the NEASQC project.

The results of the **Benchmark** should be stored in the field *Benchmarks*, under the sub-field *Results*. This sub-field of the JSON has associated a list of elements, where each element is a dictionary with the complete result information for a **Benchmark** execution: this is, for a fixed number of qubits n (sub-field *NumberOfQubits* of the dictionary), fixed discretization parameter m (a new sub-field called *AuxiliarNumberOfQubits* was created for storing this information) and angle selection method (a new sub-field called *MethodForSettingAngles* was created for storing this information). Each one of these dictionaries stores, under the sub-field *Metrics*, the mandatory verification metrics: the **Kolmogorov-Smirnov** one is stored under the name *KS* and the **fidelity** is stored under the name *fidelity*. Additionally, the mean elapsed time must be reported in the *TotalTime* sub-field and its standard deviation in the *SigmaTotalTime* one.

List of Acronyms

Term	Definition
BTC	Benchmark Test Case
NEASQC	NExt ApplicationS of Quantum Computing
QPE	Quantum Phase Estimation
QPU	Quantum Process Unit
TNBS	The **NEASQC** Benchmark Suite

***Table T03.1**: Acronyms and Abbreviations*

List of Figures

List of Tables

List of Listings

Bibliography

Cao, Y., Romero, J., Olson, J. P., Degroote, M., Johnson, P. D., Kieferová, M., Kivlichan, I. D., Menke, T., Peropadre, B., Sawaya, N. P. D., Sim, S., Veis, L., & Aspuru-Guzik, A. (2019). Quantum chemistry in the age of quantum computing [PMID: 31469277]. Chemical Reviews, 119(19), 10856–10915. https://doi.org/10.1021/acs.chemrev.8b00803

Jiang, D., Liu, X., Song, H., & Xie, H. (2021). An survey: Quantum phase estimation algorithms. 2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC), 5, 884–888. https://doi.org/10.1109/ITNEC52019.2021.9587010

Kitaev, A. Y. (1995). Quantum measurements and the abelian stabilizer problem. Electron. Colloquium Comput. Complex., TR96.

A.NEASQC test case reference

As pointed out in deliverable **D3.5: The NEASQC Benchmark Suite** each proposed **Benchmark** for **TNBS**, must have a complete Eviden myQLM-compatible software implementation. For the **QPE kernel**, this implementation can be found in the **tnbs/BTC_03_QPE** folder of the **WP3_Benchmark** **NEASQC** GitHub repository. Additionally, the execution of a **Benchmark** must generate a complete result report into a separate JSON file, that must follow **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable.

The **tnbs/BTC_03_QPE** locations contains the following folders and files:

- **QPE** folder: with all the Python modules mandatory for executing the complete workflow of the **QPE** as explained in section 3.2
- **my_benchmark_execution.py**
- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

The modules inside **QPE** folder in addition to **my_benchmark_execution.py** deal with the **QPE Benchmark** execution. Section A.1 documents, exhaustively, these files. The other script files are related to **Benchmark** report generation and are properly explained in section A.2.

A.1. NEASQC implementation of benchmark test case.

This section presents a complete description of the **QPE Benchmark** implementation. In the subsection A.1.1 the Python implementation of a **BTC** step, as explained in 3.2, is presented. Subsection A.1.2 documents the modifications done in the script **my_benchmark_summary.py** that is used for executing a complete **Benchmark** procedure, as explained in 3.3.

A.1.1. Implementation of QPE for R_z^n operator

All the Python mandatory modules for executing the complete workflow of the presented **BTC**, as explained in 3.2, are stored inside the **tnbs/BTC_03_QPE/QPE** folder. Inside it following folders and files can be found:

- library **qpe_rz.py**
- library **rz_lib.py**
- folder **utils**
- folder **notebooks**

qpe_rz.py library

The **qpe_rz.py** library contains the Python class *QPE_RZ* where the basis workflow of the **BTC** (section 3.2) for a fixed number of qubits n, a discretization parameter m and an angle selection method is implemented. Listing T03.1 shows the complete class implementation.

```

class QPE_RZ:
    """
    Probability Loading
    """


    def __init__(self, **kwargs):
        """

        Method for initializing the class

        """

        self.n_qbits = kwargs.get("number_of_qbits", None)
        if self.n_qbits is None:
            error_text = "The number_of_qbits argument CAN NOT BE NONE."
            raise ValueError(error_text)
        self.auxiliar_qbits_number = kwargs.get("auxiliar_qbits_number", None)
        if self.auxiliar_qbits_number is None:
            error_text = "Provide the number of auxiliary qubits for QPE"
            raise ValueError(error_text)

        # Minimum angle measured by the QPE
        self.delta_theta = 4 * np.pi / 2 ** self.auxiliar_qbits_number

        angles = kwargs.get("angles", None)
        if type(angles) not in [str, float, list]:
            error_text = "Be aware! angles keyword can only be str" + \
                ", float or list "
            raise ValueError(error_text)

        self.angle_name = ''

        if isinstance(angles, str):
            if angles == 'random':
                self.angle_name = 'random'
                self.angles = [np.pi * np.random.random() \
                    for i in range(self.n_qbits)]
            elif angles == 'exact':
                # Here we compute the angles of the R_z^n operator for
                # obtaining exact eigenvalues in QPE. We begin in 0.5pi
                # and sum or rest randomly the minimum QPE measured
                # angle
                self.angle_name = 'exact'
                self.angles = []
                angle_0 = np.pi / 2.0
                for i_ in range(self.n_qbits):
                    angle_0 = angle_0 + (-1) ** np.random.randint(2) *\
                        self.delta_theta
                    self.angles.append(angle_0)
            else:
                error_text = "Be aware! If angles is str then only" + \
                    "can be random"
                raise ValueError(error_text)

        if isinstance(angles, float):
            self.angles = [angles for i in range(self.n_qbits)]

        if isinstance(angles, list):
            self.angles = angles
            if len(self.angles) != self.n_qbits:
                error_text = "Be aware! The number of elements in angles" + \
                    "MUST BE equal to the n_qbits"
                raise ValueError(error_text)

        # Set the QPU to use
        self.qpu = kwargs.get("qpu", None)
        if self.qpu is None:
            error_text = "Please provide a QPU."
            raise ValueError(error_text)

        # Shots for measuring the QPE circuit
        self.shots = kwargs.get("shots", None)
        if self.shots is not None:
            text = "BE AWARE! The keyword shots should be None because" +\
                "shots should be computed in function of the theoretical" +\
                "eigenvalues. You can only provide 0 for doing some testing" +\
                "in the class. 0 will imply complete simulation of QPE circuit"
            print(text)
            if self.shots != 0:
                error_text = "BE AWARE! The keyword shots must be None or 0"
                raise ValueError(error_text)


        # For storing classical eigenvalue distribution
        self.theorical_eigv = None
        self.theorical_eigv_dist = None
        # For storing quantum eigenvalue distribution
        self.quantum_eigv_dist = None
        # For storing attributes from CQPE class
        self.circuit = None
        self.quantum_time = None

        # Computing complete time of the process
        self.elapsed_time = None

        # Metric attributes
        self.ks = None
        self.fidelity = None

        # Pandas DataFrame for summary
        self.pdf = None

    def theoretical_distribution(self):
        """
        Computes the theoretical distribution of Rz eigenvalues
        """
        # Compute the complete eigenvalues
        self.theorical_eigv = rz_lib.rz_eigv(self.angles)
        # Compute the eigenvalue distribution using auxiliar_qbits_number
        self.theorical_eigv_dist = rz_lib.make_histogram(
            self.theorical_eigv['Eigenvalues'], self.auxiliar_qbits_number)
        if self.shots is None:
            # Compute the number of shots for QPE circuit
            self.shots = rz_lib.computing_shots(self.theorical_eigv)
        else:
            if self.shots != 0:
                self.shots = rz_lib.computing_shots(self.theorical_eigv)
            else:
                pass

    def quantum_distribution(self):
        """
        Computes the quantum distribution of Rz eigenvalues
        """
        self.quantum_eigv_dist, qpe_object = rz_lib.qpe_rz_qlm(
            self.angles,
            auxiliar_qbits_number=self.auxiliar_qbits_number,
            shots=self.shots,
            qpu=self.qpu

        )
        self.circuit = qpe_object.circuit
        self.quantum_time = qpe_object.quantum_times

    def get_metrics(self):
        """
        Computing Metrics
        """
        # Kolmogorov-Smirnov
        self.ks = np.abs(
            self.theorical_eigv_dist['Probability'].cumsum() \
                - self.quantum_eigv_dist['Probability'].cumsum()
        ).max()
        # Fidelity
        qv = self.quantum_eigv_dist['Probability']
        tv = self.theorical_eigv_dist['Probability']
        self.fidelity = qv @ tv / (np.linalg.norm(qv) * np.linalg.norm(tv))


    def exe(self):
        """
        Execution of workflow
        """
        tick = time.time()
        # Compute theoretical eigenvalues
```

```python
        self.theoretical_distribution()
        # Computing eigenvalues using QPE
        self.quantum_distribution()
        # Compute the metrics
        self.get_metrics()
        tack = time.time()
        self.elapsed_time = tack - tick
        self.summary()

    def summary(self):
        """
        Pandas summary
        """
        self.pdf = pd.DataFrame()
        self.pdf["n_qbits"] = [self.n_qbits]
        self.pdf["aux_qbits"] = [self.auxiliar_qbits_number]
        self.pdf["delta_theta"] = self.delta_theta
        self.pdf["angle_method"] = [self.angle_name]
        self.pdf["angles"] = [self.angles]
        self.pdf["qpu"] = [self.qpu]
        self.pdf["shots"] = [self.shots]
        self.pdf["KS"] = [self.ks]
        self.pdf["fidelity"] = [self.fidelity]
        self.pdf["elapsed_time"] = [self.elapsed_time]
        self.pdf["quantum_time"] = [self.quantum_time[0]]
```

Listing T03.1: *QPE_RZ class from* ***qpe_rz.py*** *library .*

For configuring the *QPE_RZ* class a Python dictionary should be provided, see Listing T03.2 for an example.

```python

    qpe_rz_dict = {
        'number_of_qbits' : 7,
        'auxiliar_qbits_number' : 7,
        'angles' : 'exact',
        'qpu' : qat.qpus.CLinalg(),
    }

    qpe_rz_b = QPE_RZ(**qpe_rz_dict)
    qpe_rz_b.exe()
```

Listing T03.2: Example for QPE_RZ class configuration using a Python dictionary and how to use it

For executing the workflow the *exe* method of *QPE_RZ* should be used. The following attributes are populated when this *exe* method is invoking:

- *KS*: Kolmogorov-Smirnov distance, see equation (T03.2.5)
- *fidelity*: fidelity metric, see equation (T03.2.6)
- *theorical_eigv_dist*: pandas DataFrame with the theoretical probability distribution of the eigenvalues: $P^{th}_{\lambda,m}(\frac{k}{2^m})$
- *quantum_eigv_dist*: pandas DataFrame with the probability distribution of the eigenvalues from **QPE**: $P^{QPE}_{\lambda,m}(\frac{k}{2^m})$

The **qpe_rz.py** library can be executed from the command line. Different arguments can be provided to properly configure the *QPE_RZ* class. For a usage guide -h parameter can be provided. Listing T03.3 shows an example of the command line usage. In this case, the R_z^n operator will be of $n = 7$ qubit, the parameter discretization $m = 7$ and the method for loading angles will be the **exact** method (the 0 argument for -angles). Additionally, **CLinAlg** solver, from Eviden myQLM library, will be used for simulating the canonical **QPE**.

```
    python qpe_rz.py -n_qbits 7 -aux_qbits 7 -qpu c -angles 0
```

Listing T03.3: Command line example for configuring the QPÊ_RZ class and executing the benchmark workflow

rz_lib.py library

The *rz_lib* library, Listing T03.4 contains all the auxiliary functions needed by the *QPE_RZ* class from **qpe_rz** library.

```python

import numpy as np
import pandas as pd
import qat.lang.AQASM as qlm
from QPE.utils.qpe import CQPE


def get_qpu(qpu=None):
    """
    Function for selecting solver.

    Parameters
    ----------

    qpu : str
        * qlmass: for trying to use QLM as a Service connection to CESGA QLM
        * python: for using PyLinalg simulator.
        * c: for using CLinalg simulator

    Returns
    ----------

    linal_qpu : solver for quantum jobs
    """

    if qpu is None:
        raise ValueError(
            "qpu CAN NOT BE NONE. Please select one of the three" +
            " following options: qlmass, python, c")
    if qpu == "qlmass":
        try:
            from qlmaas.qpus import LinAlg
            linalg_qpu = LinAlg()
        except (ImportError, OSError) as exception:
            raise ImportError(
                "Problem Using QLMaaS. Please create config file" +
                "or use mylm solver") from exception
    elif qpu == "python":
        from qat.qpus import PyLinalg
        linalg_qpu = PyLinalg()
    elif qpu == "c":
        from qat.qpus import CLinalg
        linalg_qpu = CLinalg()
    elif qpu == "default":
        from qat.qpus import get_default_qpu
        linalg_qpu = get_default_qpu()
    else:
        raise ValueError(
            "Invalid value for qpu. Please select one of the three "+
            "following options: qlmass, python, c")
    #print("Following qpu will be used: {}".format(linalg_qpu))
    return linalg_qpu

# Functions for generating theoretical eigenvalues of R_z^n
def bitfield(n_int: int, size: int):
    """Transforms an int n_int to the corresponding bitfield of size size

    Parameters
    ----------
    n_int : int
        integer from which we want to obtain the bitfield
    size : int
        size of the bitfield

    Returns
    ----------
    full : list of ints
        bitfield representation of n_int with size size

    """
    aux = [1 if digit == "1" else 0 for digit in bin(n_int)[2:]]
    right = np.array(aux)
    left = np.zeros(max(size - right.size, 0))
    full = np.concatenate((left, right))
    return full.astype(int)

def rz_eigenv_from_state(state, angles):
    """
    For a fixed input state and the angles of the R_z^n operator compute
    the correspondent eigenvalue.

    Parameters
    __________

    state : np.array
        Array with the binary representation of the input state
    angles: np.array
        Array with the angles for the R_z^n operator.

    Returns
    _______

    lambda_ : float
        The eigenvalue for the input state of the R_z^n operator with
        the input angles

    """
    new_state = np.where(state == 1, -1, 1)
    # Computing the eigenvalue correspondent to the input state
    thetas = - 0.5 * np.dot(new_state, angles)
    # We want the angle between [0, 2pi]
    thetas_2pi = np.mod(thetas, 2 * np.pi)
    # Normalization of the angle between [0,1]
    lambda_ = thetas_2pi / (2.0 * np.pi)
    return lambda_

def rz_eigv(angles):
    """
    Computes the complete list of eigenvalues for a R_z^n operator
    for an input list of angles
    Provides the histogram between [0,1] with a bin of 2**discretization
    for the distribution of eigenvalues of a R_z^n operator for a given
    list of angles.

    Parameters
    __________

    angles: np.array
        Array with the angles for the R_z^n operator.

    Returns
    _______

    pdf : pandas DataFrame
        DataFrame with all the eigenvalues of the R_z^n operator for
        the input list angles. Columns
            * States : Eigenstates of the Rz^n operator (least
                significant bit is leftmost)
            * Int_lsb_left : Integer conversion of the state
                (leftmost lsb)
            * Int_lsb_rightt : Integer conversion of the state
                (rightmost lsb)
            * Eigenvalues : correspondent eigenvalue

    """

    n_qubits = len(angles)
    # Compute eigenvalues of all possible eigenstates
    eigv = [rz_eigenv_from_state(bitfield(i, n_qubits), angles)\
        for i in range(2**n_qubits)]
    pdf = pd.DataFrame(
        [eigv],
        index=['Eigenvalues']
    ).T
    pdf['Int_lsb_left'] = pdf.index
    state = pdf['Int_lsb_left'].apply(
        lambda x: bin(x)[2:].zfill(n_qubits)
    )
    pdf['States'] = state.apply(lambda x: '|' + x[::-1] + '>')
    pdf['Int_lsb_right'] = state.apply(lambda x: int('0b'+x[::-1], base=0))
    pdf = pdf[['States', 'Int_lsb_left', 'Int_lsb_right', 'Eigenvalues']]
    return pdf

def make_histogram(eigenvalues, discretization):
    """
    Given an input list of eigenvalues compute the correspondent
    histogram using a bins = 2^discretization

    Parameters
    __________

    eigenvalues : list
        List with the eigenvalues
    discretization: int
        Histogram discretization parameter: The number of bins for the
        histogram will be: 2^discretization

    Returns
    _______

    pdf : pandas DataFrame
        Pandas Dataframe with the 2^m bin frequency histogram for the
        input list of eigenvalues. Columns
            * lambda : bin discretization for eigenvalues based on the
              discretization input
            * Probability: probability of finding any eigenvalue inside
              of the correspoondent lambda bin
    """

    # When the histogram is computed can be some problems with numeric
    # approaches. So we compute the maximum number of decimals for
    # a bare discretization of the bins and use it for rounding properly
    # the eigenvalues
    lambda_strings = [len(str(i / 2 ** discretization).split('.')[1]) \
        for i in range(2 ** discretization)]
    decimal_truncation = max(lambda_strings)
    trunc_eigenv = [round(i_, decimal_truncation) for i_ in list(eigenvalues)]
    pdf = pd.DataFrame(
        np.histogram(
            trunc_eigenv,
            bins=2 ** discretization,
            range=(0, 1.0)
        ),
        index=["Counts", "lambda"]
    ).T[:2 ** discretization]
    pdf["Probability"] = pdf["Counts"] / sum(pdf["Counts"])
    pdf.drop(columns=['Counts'], inplace=True)

    return pdf

# Below are functions for Atos myqlm simulation of R_z^n
def qpe_rz_qlm(angles, auxiliar_qbits_number, shots=0, qpu=None):
    """
    Computes the Quantum Phase Estimation for a Rz Kronecker product

    Parameters
    __________

    angles : list
        list with the angles that are applied to each qubit of the circuit
    auxiliar_qbits_number : int
        number of auxiliary qubits for doing QPE
    shots : int
        number of shots for getting the results. 0 for exact solution
    qpu : Atos QLM QPU object
        QLM QPU for solving the circuit

    Returns
    _______

    results : pandas DataFrame
        pandas DataFrame with the distribution of the eigenvalues with
        a bin discretization of 2^auxiliar_qbits_number
        * lambda : bin discretization for eigenvalues based on the
            discretization input (auxiliar_qbits_number input)
        * Probability: probability of finding any eigenvalue inside
            of the correspoondent lambda bin

    qft_pe : CQPE object

    """
    n_qbits = len(angles)
    #print('n_qubits: {}'.format(n_qbits))
    initial_state = qlm.QRoutine()
    q_bits = initial_state.new_wires(n_qbits)

    # Creating the superposition initial state
    for i in range(n_qbits):
        #print(i)
        initial_state.apply(qlm.H, q_bits[i])

    # Creating the Operator Rz_n
    rzn_gate = rz_angles(angles)
    #We create a python dictionary for configuration of class
    qft_pe_dict = {
        'initial_state': initial_state,
        'unitary_operator': rzn_gate,
        'qpu' : qpu,
        'auxiliar_qbits_number' : auxiliar_qbits_number,
        'complete': True,
        'shots' : shots
    }
    qft_pe = CQPE(**qft_pe_dict)
    qft_pe.run()
    qft_pe_results = qft_pe.result
    qft_pe_results.sort_values('lambda', inplace=True)
    results = qft_pe_results[['lambda', 'Probability']]
    results.reset_index(drop=True, inplace=True)
    return results, qft_pe

def rz_angles(thetas):
    """
    Creates a QLM abstract Gate with a R_z^n operator of an input array of angles

    Parameters
    __________

    thetas : array
        Array with the angles of the R_z^n operator

    Returns
    _______

    r_z_n : QLM AbstractGate
        AbstractGate with the implementation of R_z_^n of the input angles

    """
    n_qbits = len(thetas)

    @qlm.build_gate("Rz_{}".format(n_qbits), [], arity=n_qbits)
    def rz_routine():
        routine = qlm.QRoutine()
        q_bits = routine.new_wires(n_qbits)
        for i in range(n_qbits):
            routine.apply(qlm.RZ(thetas[i]), q_bits[i])
        return routine
    r_z_n = rz_routine()
    return r_z_n

def computing_shots(pdf):
    """
    Compute the number of shots. The main idea is that the samples for
    the lowest degeneracy eigenvalues will be enough. In this case
    enough is that that we measured an eigenvalue that will have an
    error from respect to the theoretical one lower than the
    discretization precision at least 100 times

    Parameters
    __________

    pdf : pandas DataFrame
        DataFrame with the theoretical eigenvalues

    Returns
    _______

    shots : int
        number of shots for QPE algorithm

    """
    # prob of less frequent eigenvalue
    lfe = min(pdf.value_counts('Eigenvalues')) / len(pdf)
    shots = int((1000 / (lfe * 0.81))) + 1
    return shots
```

Listing T03.4: rz_lib.py library

Main important functions from *rz_lib* library are:

- *rz_eigv*: computes the theoretical *eigenvalues* for a R_z^n operator providing it the input angles
- *qpe_rz_qlm*: computes the *eigenvalues* of a R_z^n operator using Eviden myQLM implementation of the **QPE** algorithm.
- *make_histogram*: computes the histogram of a list of *eigenvalues*

utils folder

The **utils** folder contains all the mandatory functions related to the Eviden **myQLM** implementation of the canonical **QPE** algorithm as explained in section 2.2 and Figure T03.1. This implementation was obtained from the **QQuantLib** library of the **NEASQC Financial Applications** GitHub repository.

notebooks folder

In the notebook folder, the following jupyter notebooks are stored:

- *01_BTC_03_QPE_for_rzn_rz_library.ipynb*: this notebooks is a tutorial of how to use the *rz_lib* library. Detailed documentation about the canonical **QPE** and the R_z^n operator is provided here.
- *02_BTC_03_QPE_for_rzn_Procedure.ipynb*: this notebook is a tutorial of how to use the *QPE_RZ* class from **qpe_rz** library.

A.1.2. my_benchmark_execution.py

This script is a modification of the correspondent template script located in tnbs/templates folder of the **WP3_Benchmark** repository. Following the recommendations of Annex B of the deliverable **D3.5: The NEASQC Benchmark Suite** the **run_code**, **compute_samples**, **summarize_results** and the **build_iterator** functions were modified. Meanwhile, the **KERNEL_BENCHMARK** class was not modified. In the following sections, the software adaptations for the **QPE Benchmark** are presented.

run_code

Listing T03.5 shows the modifications performed into the **run_code** function for the **QPE Benchmark**. The main functionality is executing the **BTC** (section 3.2) for a fixed number of qubits, n, discretization parameter, m, and an angle selection method. These parameters should be provided as a 3-element Python tuple (*iterator_step*). The execution will be done *repetitions* number of times and all the mandatory metrics will be gathered as pandas DataFrame (*metrics*). The *stage_bench* is a boolean variable that indicates if the step is executed in the pre-benchmark (step *2.a* in section 3.3) or in the benchmark stage (step *2.b* in section 3.3). As can be seen the **QPE_RZ** from **qpe_rz.py** library and its *exe* method is used for doing the different executions.

```python

def run_code(iterator_step, repetitions, stage_bench, **kwargs):
    """
    For configuration and execution of the benchmark kernel.

    Parameters
    ----------

    iterator_step : tuple
        tuple with elements from iterator built from build_iterator.
    repetitions : list
        number of repetitions for each execution
    stage_bench : str
        benchmark stage. Only: benchmark, pre-benchamrk
    kwargs : keyword arguments
        for configuration of the benchmark kernel

    Returns
    _______

    metrics : pandas DataFrame
        DataFrame with the desired metrics obtained for the integral computation
    save_name : string
        Desired name for saving the results of the execution

    """
    # if n_qbits is None:
    #     raise ValueError("n_qbits CAN NOT BE None")

    if stage_bench not in ['benchmark', 'pre-benchmark']:
        raise ValueError(
            "Valid values for stage_bench: benchmark or pre-benchmark'")

    if repetitions is None:
        raise ValueError("samples CAN NOT BE None")

    #Here the code for configuring and execute the benchmark kernel
    kernel_configuration_ = deepcopy(kwargs.get("kernel_configuration", None))
    if kernel_configuration_ is None:
        raise ValueError("kernel_configuration can not be None")
    # Here we built the dictionary for the QPE_RZ class
    n_qbits = iterator_step[0]
    aux_qbits = iterator_step[1]
    angles = iterator_step[2]
    # print('n_qbits :{}. aux_qbits: {}. angles: {}'.format(
    #     n_qbits, aux_qbits, angles))
    qpu = get_qpu(kernel_configuration_['qpu'])
    qpe_rz_dict = {
        'number_of_qbits' : n_qbits,
        'auxiliar_qbits_number' : aux_qbits,
        'angles' : angles,
        'qpu' : qpu,
    }

    list_of_metrics = []
    #print(qpe_rz_dict)
    for i in range(repetitions[0]):
        rz_qpe = QPE_RZ(**qpe_rz_dict)
        rz_qpe.exe()
        list_of_metrics.append(rz_qpe.pdf)

    metrics = pd.concat(list_of_metrics)
    metrics.reset_index(drop=True, inplace=True)

    if stage_bench == 'pre-benchmark':
        # Name for storing Pre-Benchmark results
        save_name = "pre_benchmark_nq_{}_auxq_{}_angles_{}.csv".format(
            n_qbits, aux_qbits, angles)
    if stage_bench == 'benchmark':
        # Name for storing Benchmark results
        save_name = kwargs.get('csv_results')
        #save_name = "pre_benchmark_step_{}.csv".format(n_qbits)
    return metrics, save_name
```

*Listing T03.5: run_code function for executing the **BTC** of the **QPE kernel**, as explained in section 3.2*

compute_samples

Listing T03.6 shows the implementation of the **compute_samples** function for the **QPE Benchmark**. The main objective is to codify a strategy for computing the number of times a step of the **BTC** should be executed, for getting some desired statistical significance (see *3.a.ii* and *3.a.iii* of section 3.3). This function would implement equations (T03.3.7) and (T03.3.8) and compute the corresponding maximum as explained in *3.b* of section 3.3. As can be seen depending on the selection angle method the **fidelity** or the **KS** metric is used for computing it.

```
    def compute_samples(**kwargs):
        """
        This function computes the number of executions of the benchmark
        for ensuring an error r with a confidence level of alpha

        Parameters
        ----------

        kwargs : keyword arguments
            For configuring the sampling computation

        Returns
        _______

        samples : pandas DataFrame
            DataFrame with the number of executions for each integration interval

        """

        #Configuration for sampling computations

        #Desired Confidence level
        alpha = kwargs.get("alpha", 0.05)
        if alpha is None:
            alpha = 0.05
        metrics = kwargs.get('pre_metrics')
        bench_conf = kwargs.get('kernel_configuration')

        #Code for computing the number of samples for getting the desired
        #statististical significance. Depends on benchmark kernel
        #samples_ = pd.Series([100, 100])
        #samples_.name = "samples"

        method = metrics['angle_method'].unique()
        if len(method) != 1:
            raise ValueError('Only can provide one angle method!')

        from scipy.stats import norm
        zalpha = norm.ppf(1-(alpha/2)) # 95% of confidence level

        method = method[0]

        if method == 'exact':

            # Error expected for the means fidelity
            error_fid = bench_conf.get("fidelity_error", 0.001)
            if error_fid is None:
                error_fid = 0.001
            metric_fidelity = ['fidelity']
            std_ = metrics[metric_fidelity].std()
            samples_metric = (zalpha * std_ / error_fid) ** 2
        elif method == 'random':
            # Error expected for the means KS
            error_ks = bench_conf.get("ks_error", 0.05)
            if error_ks is None:
                error_ks = 0.05
            metric_ks = ['KS']
            std_ = metrics[metric_ks].std()
            samples_metric = (zalpha * std_ / error_ks) ** 2
        else:
            raise ValueError('Angle method can be only: exact or random')

        time_error = bench_conf.get("time_error", 0.05)
        if time_error is None:
            time_error = 0.05
        mean_time = metrics[["elapsed_time"]].mean()
        std_time = metrics[["elapsed_time"]].std()
        samples_time = (zalpha * std_time / (time_error * mean_time)) ** 2

        #Maximum number of sampls will be used
        samples_ = pd.Series(pd.concat([samples_time, samples_metric]).max())

        #Apply lower and higher limits to samples
        #Minimum and Maximum number of samples
        min_meas = kwargs.get("min_meas", None)
        if min_meas is None:
            min_meas = 5
        max_meas = kwargs.get("max_meas", None)
        samples_.clip(upper=max_meas, lower=min_meas, inplace=True)
        samples_ = samples_.max().astype(int)
        return samples_
```

*Listing T03.6: compute_samples function for codifying the strategy for computing the number of repetitions for the **QPE Benchmark**.*

summarize_results

Listing T03.7 shows the implementation of the **summarize_results** function for the **QPE Benchmark**. The main objective is post-processing the results of a complete **Benchmark** execution, as described in step *3-c* of section 3.3.

This function expects that the results of the complete benchmark execution have been stored in a *csv* file. The function loads this file into a pandas DataFrame that is post-processed properly.

```

    def summarize_results(**kwargs):
        """
        Create summary with statistics
        """

        folder = kwargs.get("saving_folder")
        csv_results = folder + kwargs.get("csv_results")

        #Code for summarize the benchamark results. Depending of the
        #kernel of the benchmark
        pdf = pd.read_csv(csv_results, index_col=0, sep=";")
        pdf["classic_time"] = pdf["elapsed_time"] - pdf["quantum_time"]
        # The angles are randomly selected. Not interesting for aggregation
        pdf.drop(columns=['angles'], inplace=True)
        results = pdf.groupby(["n_qbits", "aux_qbits", "angle_method"]).agg(
            ["mean", "std", "count"] + \
                [('std_mean', lambda x: np.std(x)/np.sqrt(len(x)))])
        results.drop(columns=[
            ('delta_theta', 'std'),
            ('delta_theta', 'count'),
            ('delta_theta', 'std_mean'),
            ('shots', 'std'),
            ('shots', 'count'),
            ('shots', 'std_mean')],
            inplace=True
        )

        results['qpu'] = [''.join(list(b_['qpu'].unique())) for a_, b_ \
            in pdf.groupby(['n_qbits', 'aux_qbits', 'angle_method'])]
        #results = pd.DataFrame()
        return results
```

*Listing T03.7: summarize_results function for summarizing the results from **BTC** execution of the **QPE kernel***

build_iterator

Listing T03.8 shows the implementation of the **build_iterator** function for **QPE Benchmark**. The main objective is to create a Python iterator for executing the desired complete **BTC**. In this case, the iterator creates a list with all the possible combinations of the desired number of qubits, n, parameter discretization, m and angle selection methods that want to be benchmarked.

```
def build_iterator(**kwargs):
    """
    For building the iterator of the benchmark
    """

    list4int = [
        kwargs['list_of_qbits'],
        kwargs['kernel_configuration']['auxiliar_qbits_number'],
        kwargs['kernel_configuration']['angles'],
    ]

    iterator = it.product(*list4int)
    return iterator
```

*Listing T03.8: build_iterator function for creating the iterator of the complete execution of the **QPE Benchmark***

KERNEL_BENCHMARK class

No modifications were made to the **KERNEL_BENCHMARK** class. This Python class defines the complete benchmark workflow, section 3.3, and its *exe* method executes it properly by calling the correspondent functions (*run_code*, *compute_samples*, *summarize_results*, *build_iterator*). Each time a **Benchmark** step is executed, as defined in section 3.3, the result is stored in a given *CSV* file.

The only mandatory modification is configuring properly the input keyword arguments, at the end of the **my_benchmark_execution.py** script. These parameters will configure the complete **Benchmark** workflow, and additional options (as the name of the *CSV* files). Listing T03.9 shows an example for configuring an execution of a **Benchmark**.

```

if __name__ == "__main__":

    import os
    import shutil

    kernel_configuration = {
        "angles" : ["random", 'exact'],
        "auxiliar_qbits_number" : [4, 6, 8, 10],
        "qpu" : "c", #python, qlmass, default
        "fidelity_error" : None,
        "ks_error" : None,
        "time_error": None
    }

    benchmark_arguments = {
        #Pre benchmark sttuff
        "pre_benchmark": True,
        "pre_samples": None,
        "pre_save": True,
        #Saving stuff
        "save_append": True,
        "saving_folder": "./Results/",
        "benchmark_times": "kernel_times_benchmark.csv",
        "csv_results": "kernel_benchmark.csv",
        "summary_results": "kernel_SummaryResults.csv",
        #Computing Repetitions stuff
        "alpha": None,
        "min_meas": None,
        "max_meas": None,
        #List number of qubits tested
        "list_of_qbits": [4, 6, 8, 10, 12],
    }

```

```
        #Configuration for the benchmark kernel
        benchmark_arguments.update({"kernel_configuration": kernel_configuration})
        kernel_bench = KERNEL_BENCHMARK(**benchmark_arguments)
        kernel_bench.exe()
```

*Listing T03.9: Example of configuration of a complete **Benchmark** execution. This part of the code should be located at the end of the **my_benchmark_execution.py** script*

As can be seen in Listing T03.9, the input dictionary that **KERNEL_BENCHMARK** class needs, *benchmark_arguments*, have several keys that allow to modify the benchmark workflow, like:

- *pre_benchmark*: For executing or not the *pre-benchmark* step.
- *pre_samples*: number of repetitions of the benchmark step.
- *pre_save*: For saving or not the results from the *pre-benchmark* step.
- *saving_folder*: Path for storing all the files generated by the execution of the **KERNEL_BENCHMARK** class.
- *benchmark_times*: name for the *csv* file where the initial and the final times for the complete benchmark execution will be stored.
- *csv_results*: name for the *csv* file where the obtained metrics for the different repetitions of the benchmark step will be stored (so the different metrics obtained during step 2 from section 3.3 will be stored in this file)
- *summary_results*: name for the *csv* file where the post-processed results (using the *summarize_results*) will be stored (so the statistics over the metrics obtained during step 3 of section 3.3 will be stored in this file)
- *list_of_qbits*: list with the different number of qubits for executing the complete **Benchmark**.
- *alpha*: for configuring the desired confidence level α
- *min_meas*: for low limiting the number of executions a benchmark step should be executed during the benchmark stage.
- *max_meas*: for high limiting the number of executions a benchmark step should be executed during the benchmark stage.

Additionally, the *kernel_configuration* key is used for configuring the kernel execution. The following keys can be provided for configuring it:

- *angles*: for configuring what angle loading methods will be used (in the methodology it is expected that the 2 methods must be tested).
- *auxiliar_qbits_number*: For configuring the discretization parameter that will be tested.
- *qpu*: a string for selecting the quantum process unit (**QPU**)
- *fidelity_error*: for changing the desired absolute error for the **fidelity** metric.
- *ks_error*: for changing the desired absolute error for the **KS** metric.
- *time_error*: for changing the desired relative error for the **elapsed time**

In general, most of the keys should be fixed to *None* for executing the **Benchmark** according to the guidelines of the **QPE Benchmark**

For executing the **Benchmark** following command should be used:

python my_benchmark_execution.py

A.2. Generation of the benchmark report

Following deliverable **D3.5: The NEASQC Benchmark Suite** the results of a complete **Benchmark** must be reported in a separate JSON file that must satisfy the **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable. For automating this process the following files should be modified, as explained in Annex B of the deliverable **D3.5: The NEASQC Benchmark Suite**:

- **my_environment_info.py**

- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

my_environment_info.py

This script has the functions for gathering information about the hardware where the **Benchmark** is executed.

Listing T03.10 shows an example of the **my_environment_info.py** script. Here the compiled information corresponds to a classic computer because the case was simulated instead of executed in a quantum computer.

```python

import platform
import psutil
from collections import OrderedDict

def my_organisation(**kwargs):
    """
    Given information about the organisation how uploads the benchmark
    """
    name = "CESGA"
    return name

def my_machine_name(**kwargs):
    """
    Name of the machine where the benchmark was performed
    """
    #machine_name = "None"
    machine_name = platform.node()
    return machine_name

def my_qpu_model(**kwargs):
    """
    Name of the model of the QPU
    """
    qpu_model = "CLinalg"
    return qpu_model

def my_qpu(**kwargs):
    """
    Complete info about the used QPU
    """
    #Basic schema
    #QPUDescription = {
    #    "NumberOfQPUs": 1,
    #    "QPUs": [
    #        {
    #            "BasicGates": ["none", "none1"],
    #            "Qubits": [
    #                {
    #                    "QubitNumber": 0,
    #                    "T1": 1.0,
    #                    "T2": 1.00
    #                }
    #            ],
    #            "Gates": [
    #                {
    #                    "Gate": "none",
    #                    "Type": "Single",
    #                    "Symmetric": False,
    #                    "Qubits": [0],
    #                    "MaxTime": 1.0
    #                }
    #            ],
    #            "Technology": "other"
    #        },
    #    ]
    #}

    #Defining the Qubits of the QPU
```

```
        qubits = OrderedDict()
        qubits["QubitNumber"] = 0
        qubits["T1"] = 1.0
        qubits["T2"] = 1.0

        #Defining the Gates of the QPU
        gates = OrderedDict()
        gates["Gate"] = "none"
        gates["Type"] = "Single"
        gates["Symmetric"] = False
        gates["Qubits"] = [0]
        gates["MaxTime"] = 1.0


        #Defining the Basic Gates of the QPU
        qpus = OrderedDict()
        qpus["BasicGates"] = ["none", "none1"]
        qpus["Qubits"] = [qubits]
        qpus["Gates"] = [gates]
        qpus["Technology"] = "other"

        qpu_description = OrderedDict()
        qpu_description['NumberOfQPUs'] = 1
        qpu_description['QPUs'] = [qpus]

        return qpu_description

    def my_cpu_model(**kwargs):
        """
        model of the cpu used in the benchmark
        """
        cpu_model = platform.processor()
        return cpu_model

    def my_frecuency(**kwargs):
        """
        Frcuency of the used CPU
        """
        #Use the nominal frequency. Here, it collects the maximum frequency
        #print(psutil.cpu_freq())
        cpu_frec = psutil.cpu_freq().max/1000
        return cpu_frec

    def my_network(**kwargs):
        """
        Network connections if several QPUs are used
        """
        network = OrderedDict()
        network["Model"] = "None"
        network["Version"] = "None"
        network["Topology"] = "None"
        return network

    def my_QPUCPUConnection(**kwargs):
        """
        Connection between the QPU and the CPU used in the benchmark
        """
        #
        # Provide the information about how the QPU is connected to the CPU
        #
        qpuccpu_conn = OrderedDict()
        qpuccpu_conn["Type"] = "memory"
        qpuccpu_conn["Version"] = "None"
        return qpuccpu_conn
```

Listing T03.10: Example of configuration of the ***my_environment_info.py*** *script*

In general, it is expected that for each computer used (quantum or classic), the **Benchmark** developer should change this script to properly get the hardware info.

A.2.1. my_benchmark_info.py

This script gathers the information under the field *Benchmarks* of the **Benchmark** report. Information about the software, the compilers and the results obtained from an execution of the **Benchmark** is stored in this field.

Listing T03.11 shows an example of the configuration of the **my_benchmark_info.py** script for gathering the aforementioned information.

```python
import sys
import platform
from collections import OrderedDict
from my_benchmark_summary import summarize_results
import pandas as pd


def my_benchmark_kernel(**kwargs):
    """
    Name for the benchmark Kernel
    """
    return "QuantumPhaseEstimation"

def my_starttime(**kwargs):
    """
    Providing the start time of the benchmark
    """
    times_filename = kwargs.get("times_filename", None)
    pdf = pd.read_csv(times_filename, index_col=0)
    start_time = pdf["StartTime"][0]
    return start_time

def my_endtime(**kwargs):
    """
    Providing the end time of the benchmark
    """
    times_filename = kwargs.get("times_filename", None)
    pdf = pd.read_csv(times_filename, index_col=0)
    end_time = pdf["EndTime"][0]
    return end_time

def my_timemethod(**kwargs):
    """
    Providing the method for getting the times
    """
    time_method = "time.time"
    return time_method

def my_programlanguage(**kwargs):
    """
    Getting the programing language used for benchmark
    """
    program_language = platform.python_implementation()
    return program_language

def my_programlanguage_version(**kwargs):
    """
    Getting the version of the programing language used for benchmark
    """
    language_version = platform.python_version()
    return language_version

def my_programlanguage_vendor(**kwargs):
    """
    Getting the version of the programing language used for benchmark
    """
    language_vendor = "None"
    return language_vendor

def my_api(**kwargs):
    """
    Collect the information about the used APIs
    """
    # api = OrderedDict()
    # api["Name"] = "None"
```

```
        # api["Version"] = "None"
        # list_of_apis = [api]
        modules = []
        list_of_apis = []
        for module in list(sys.modules):
            api = OrderedDict()
            module = module.split('.')[0]
            if module not in modules:
                modules.append(module)
                api["Name"] = module
                try:
                    version = sys.modules[module].__version__
                except AttributeError:
                    #print("NO VERSION: "+str(sys.modules[module]))
                    try:
                        if  isinstance(sys.modules[module].version, str):
                            version = sys.modules[module].version
                            #print("\t Attribute Version"+version)
                        else:
                            version = sys.modules[module].version()
                            #print("\t Methdod Version"+version)
                    except (AttributeError, TypeError) as error:
                        #print('\t NO VERSION: '+str(sys.modules[module]))
                        try:
                            version = sys.modules[module].VERSION
                        except AttributeError:
                            #print('\t\t NO VERSION: '+str(sys.modules[module]))
                            version = "Unknown"
                api["Version"] = str(version)
                list_of_apis.append(api)
        return list_of_apis

    def my_quantum_compilation(**kwargs):
        """
        Information about the quantum compilation part of the benchmark
        """
        q_compilation = OrderedDict()
        q_compilation["Step"] = "None"
        q_compilation["Version"] = "None"
        q_compilation["Flags"] = "None"
        return [q_compilation]

    def my_classical_compilation(**kwargs):
        """
        Information about the classical compilation part of the benchmark
        """
        c_compilation = OrderedDict()
        c_compilation["Step"] = "None"
        c_compilation["Version"] = "None"
        c_compilation["Flags"] = "None"
        return [c_compilation]

    def my_metadata_info(**kwargs):
        """
        Other important info user want to store in the final json.
        """

        metadata = OrderedDict()
        #metadata["None"] = None

        return metadata


    def my_benchmark_info(**kwargs):
        """
        Complete WorkFlow for getting all the benchmar informated related info
        """
        benchmark = OrderedDict()
        benchmark["BenchmarkKernel"] = my_benchmark_kernel(**kwargs)
        benchmark["StartTime"] = my_starttime(**kwargs)
        benchmark["EndTime"] = my_endtime(**kwargs)
        benchmark["ProgramLanguage"] = my_programlanguage(**kwargs)
        benchmark["ProgramLanguageVersion"] = my_programlanguage_version(**kwargs)
        benchmark["ProgramLanguageVendor"] = my_programlanguage_vendor(**kwargs)
        benchmark["API"] = my_api(**kwargs)
        benchmark["QuantumCompililation"] = my_quantum_compilation(**kwargs)
        benchmark["ClassicalCompiler"] = my_classical_compilation(**kwargs)
        benchmark["TimeMethod"] = my_timemethod(**kwargs)
        benchmark["Results"] = summarize_results(**kwargs)
        benchmark["MetaData"] = my_metadata_info(**kwargs)
        return benchmark
```

Listing T03.11: Example of configuration of the ***my_benchmark_info.py*** *script*

The *my_benchmark_info* function gathers all the mandatory information needed by the *Benchmarks* main field of the report (by calling the different functions listed in listing T03.11). In order to properly fills this field some mandatory information must be provided as the typical *python kwargs*:

- *times_filename*: This is the complete path to the file where the starting and ending time of the benchmark was stored. This file must be a *csv* one and it is generated when the **KERNEL_BENCHMARK** class is executed. This information is used by the *my_starttime* and *my_endtime* functions.
- *benchmark_file*: complete path where the file with the summary results of the benchmark are stored. This information is used by the *summarize_results* function from *my_benchmark_summary.py* script (see section A.2.2).

A.2.2. my_benchmark_summary.py

In this script, the *summarize_results* function is implemented. This function formats the results of a complete execution of a **QPE Benchmark** with a suitable **NEASQC** benchmark report format. It can be used for generating the information under the sub-field *Results* of the main field *Benchmarks* in the report.

Listing T03.12 shows an example of implementation of *summarize_results* function for the **QPE** benchmark procedure.

```
    def summarize_results(**kwargs):
        """
        Mandatory code for properly present the benchmark results following
        the NEASQC jsonschema
        """

        # n_qbits = [4]
        # #Info with the benchmark results like a csv or a DataFrame
        # pdf = None
        # #Metrics needed for reporting. Depend on the benchmark kernel
        # list_of_metrics = ["MRSE"]

        import pandas as pd
        benchmark_file = kwargs.get("benchmark_file", None)
        index_columns = [0, 1, 2, 3, 4, 5]
        pdf = pd.read_csv(benchmark_file, header=[0, 1], index_col=index_columns)
        pdf.reset_index(inplace=True)
        n_qbits = list(set(pdf["n_qbits"]))
        angle_methods = list(set(pdf["angle_method"]))
        aux_qbits = list(set(pdf["aux_qbits"]))
        list_of_metrics = [
            "KS", "fidelity",
        ]

        results = []
        #If several qbits are tested
        # For ordering by n_qbits
        for n_ in n_qbits:
            # For ordering by auxiliar qbits
            for aux_ in aux_qbits:
                for angle_ in angle_methods:
                    result = OrderedDict()
                    result["NumberOfQubits"] = n_
                    result["QubitPlacement"] = list(range(n_))
                    result["QPUs"] = [1]
                    result["CPUs"] = psutil.Process().cpu_affinity()
                    #Select the proper data
                    indice = (pdf['n_qbits'] == n_) & (pdf['aux_qbits'] == aux_) \
                        & (pdf['angle_method'] == angle_)
                    step_pdf = pdf[indice]
                    result["TotalTime"] = step_pdf["elapsed_time"]["mean"].iloc[0]
                    result["SigmaTotalTime"] = step_pdf["elapsed_time"]["std"].iloc[0]
                    result["QuantumTime"] = step_pdf["quantum_time"]["mean"].iloc[0]
                    result["SigmaQuantumTime"] = step_pdf["quantum_time"]["std"].iloc[0]
                    result["ClassicalTime"] = step_pdf["classic_time"]["mean"].iloc[0]
                    result["SigmaClassicalTime"] = step_pdf["classic_time"]["std"].iloc[0]

                    # For identifying the test
                    result['AuxiliarNumberOfQubits'] = aux_
                    result['MethodForSettingAngles'] = angle_
                    result['QPEAnglePrecision'] = step_pdf['delta_theta'].iloc[0]
                    result['Shots'] = step_pdf['shots'].iloc[0]
                    metrics = []
                    #For each fixed number of qbits several metrics can be reported
                    for metric_name in list_of_metrics:
                        metric = OrderedDict()
                        #MANDATORY
                        metric["Metric"] = metric_name
                        metric["Value"] = step_pdf[metric_name]["mean"].iloc[0]
                        metric["STD"] = step_pdf[metric_name]["std"].iloc[0]
                        metric["COUNT"] = int(step_pdf[metric_name]["count"].iloc[0])
                        metrics.append(metric)
                    result["Metrics"] = metrics
            results.append(result)
        return results
```

Listing T03.12: Example of configuration of the summarize_results function for ***QPE benchmark***

As usual, the *kwargs* strategy is used for passing the arguments that the function can use. In this case, the only mandatory argument is *benchmark_file* with the path to the file where the summary results of the **Benchmark** execution were stored.

Table T03.2 shows the sub-fields and the information stored, under the *Results* field. To have proper traceability of the executions the sub-fields *AuxiliarNumberOfQubits*, *MethodForSettingAngles* and *QPEAnglePrecision* were created explicitly for the **QPE Benchmark**.

sub-field	information
NumberOfQubits	number of qubits, n
TotalTime	mean of **elapsed time**
SigmaTotalTime	standard deviation of **elapsed time**
QuantumTime	mean of the **quantum time**
SigmaQuantumTime	standard deviation of **quantum time**
ClassicalTime	mean of the **classical time**
SigmaClassicalTime	standard deviation of **classical time**
AuxiliarNumberOfQubits	discretization parameter, m
MethodForSettingAngles	angle selection method (**random** or **exact**)
QPEAnglePrecision	$\delta\theta = \frac{4\pi}{2^m}$
Shots	number of shots
Metrics	summarize verification metrics. See Table T03.3

Table T03.2*: Sub-fields of the Results fields of the* ***TNBS*** *benchmark report. The metrics related with the* ***quantum time*** *and* ***classical time*** *are not mandatory*

The sub-field *Metrics* gathers information about the obtained metrics of the benchmark. Table T03.3 shows its different sub-fields and the information stored.

sub-field	information
metric	*fidelity* or *KS*
Value	mean value of the metric
STD	standard deviation of the metric
Count	number of samples for computing the statistics of the metric

Table T03.3*: Sub-fields of the Metrics field.*

A.2.3. neasqc_benchmark.py

The *neasqc_benchmark.py* script can be used straightforwardly for gathering all the **Benchmark** execution information and results, for creating the final mandatory **NEASQC** benchmark report.

It does not necessarily change anything about the class implementation. It is enough to update the information of the *kwargs* arguments for providing the mandatory files for gathering all the information.

In this case, the following information should be provided as arguments for the *exe* method of the **BENCHMARK** class:

- *times_filename*: complete path where the file with the times of the **Benchmark** execution was stored.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark** execution was stored.

F.T04: Benchmark for Parent Hamiltonian

NExt ApplicationS of Quantum Computing
Benchmark Suite

T04: Benchmark for Parent Hamiltonian

Document Properties

Contract Number	951821
Contractual Deadline	31/10/2023
Dissemination Level	Public
Nature	Test Case Definition
Editors	Gonzalo Ferro, CESGA
Authors	Gonzalo Ferro, CESGA Diego Andrade, UDC Andrés Gómez, CESGA Jan Reiner, HQS Giorgio Silvi, HQS
Reviewers	Cyril Allouche, EVIDEN Arnaud Gazda, EVIDEN
Date	27/10/2023
Category	Variational
Keywords	
Status	Submitted
Release	1.0

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, Organisation	Description of Changes
0.1	04/10/2023	Gonzalo Ferro, CESGA	First version of test case
0.2	08/10/2023	Andrés Gómez, CESGA	Formatting
1.0	25/10/2023	Gonzalo Ferro, CESGA	Fixing the naming according to the Glossary of deliverable 3.5

Table of Contents

1.Introduction

This section describes the T4: Parent Hamiltonian benchmark of The NEASQC Benchmarking Suite (**TNBS**). This document must be read alongside the document that describes the TNBS: **D3.5: The NEASQC Benchmark Suite**.

Section 2 describes the Parent Hamiltonian kernel termed the **PH kernel** in the document. With each **TNBS** kernel, a **Benchmark Test Case** (**BTC**) must be designed and documented; this is done in Section 3. Finally, the benchmarking methodology aims to develop a complete software implementation of the **BTC** using the Eviden myQLM library. A complete documentation of this implementation is provided in Annex A.

2.Description of the Kernel: Parent Hamiltonian

The present section describes the **PH Kernel** for the **TNBS**. Section 2.1 justifies **Kernel** selection according to the **TNBS** benchmarking methodology, while section 2.2 presents a complete description of this **PH Kernel**.

2.1. Kernel selection justification

Variational Quantum Algorithms (VQAs) are a promising group of hybrid classical-quantum algorithms that can reach quantum advantage for solving many relevant problems in the Noise Intermediate Scale Quantum (**NISQ**) era of quantum computers (Cerezo et al., 2021). In a **VQA** algorithm there are two types of code: the first one is executed in a quantum computer and consists of a parameterized quantum circuit (usually known as ansatz); on the other hand, the second type is executed in a classical computer and consists of an optimization routine that tries to find the optimal parameters of the quantum circuit for solving a desired problem (usually codified as a Hamiltonian). The quantum parameterized circuits are typically shallow and are more suitable for the actual quantum computers.

One of the most common **VQA** algorithms is the Variational Quantum Eigensolver, **VQE**, which aims to find the quantum state that minimizes the energy of a given Hamiltonian (Peruzzo et al., 2014; Tilly et al., 2022). **VQE** can solve small molecules and lattice models, as well as simulate large chemical reactions, perform exact calculations on crystalline solids and uncover the physics behind complex systems such as the Hubbard model or exotic states of matter.

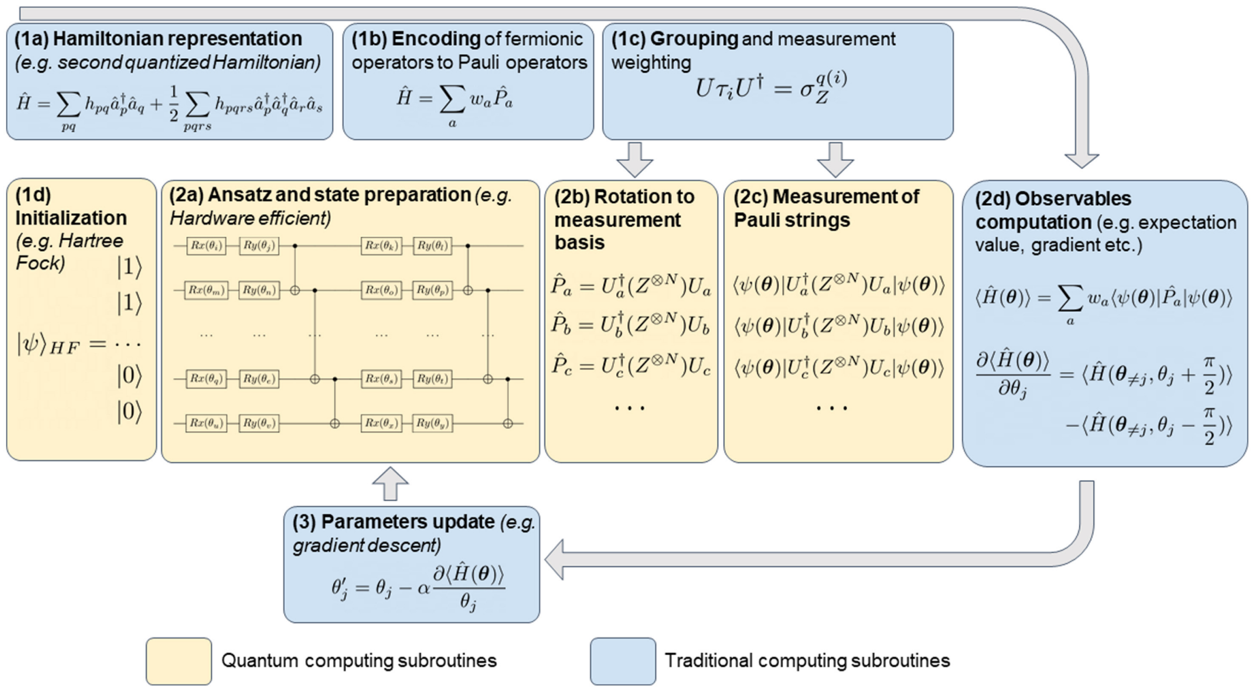

(1a) Hamiltonian representation (e.g. second quantized Hamiltonian)
$\hat{H} = \sum_{pq} h_{pq}\hat{a}_p^\dagger \hat{a}_q + \frac{1}{2}\sum_{pqrs} h_{pqrs}\hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_s$
(1b) Encoding of fermionic operators to Pauli operators
$\hat{H} = \sum_a w_a \hat{P}_a$
(1c) Grouping and measurement weighting
$U\tau_i U^\dagger = \sigma_Z^{q(i)}$
(1d) Initialization (e.g. Hartree Fock)
$|\psi\rangle_{HF} = |1\rangle |1\rangle \cdots |0\rangle |0\rangle$
(2a) Ansatz and state preparation (e.g. Hardware efficient)
(2b) Rotation to measurement basis
$\hat{P}_a = U_a^\dagger (Z^{\otimes N}) U_a$
$\hat{P}_b = U_b^\dagger (Z^{\otimes N}) U_b$
$\hat{P}_c = U_c^\dagger (Z^{\otimes N}) U_c$
(2c) Measurement of Pauli strings
$\langle \psi(\boldsymbol{\theta}) | U_a^\dagger (Z^{\otimes N}) U_a | \psi(\boldsymbol{\theta}) \rangle$
$\langle \psi(\boldsymbol{\theta}) | U_b^\dagger (Z^{\otimes N}) U_b | \psi(\boldsymbol{\theta}) \rangle$
$\langle \psi(\boldsymbol{\theta}) | U_c^\dagger (Z^{\otimes N}) U_c | \psi(\boldsymbol{\theta}) \rangle$
(2d) Observables computation (e.g. expectation value, gradient etc.)
$\langle \hat{H}(\boldsymbol{\theta}) \rangle = \sum_a w_a \langle \psi(\boldsymbol{\theta}) | \hat{P}_a | \psi(\boldsymbol{\theta}) \rangle$
$\frac{\partial \langle \hat{H}(\boldsymbol{\theta}) \rangle}{\partial \theta_j} = \langle \hat{H}(\boldsymbol{\theta}_{\neq j}, \theta_j + \frac{\pi}{2}) \rangle - \langle \hat{H}(\boldsymbol{\theta}_{\neq j}, \theta_j - \frac{\pi}{2}) \rangle$
(3) Parameters update (e.g. gradient descent)
$\theta_j' = \theta_j - \alpha \frac{\partial \langle \hat{H}(\boldsymbol{\theta}) \rangle}{\theta_j}$
Quantum computing subroutines
Traditional computing subroutines

***Figure T04.1**: The **VQE** pipeline extracted from (Tilly et al., 2022). The yellow blocks correspond to the code executed in a quantum computer, while the blue ones are related to subroutines executed in a classical computer. The block 2a is the parametric part of the circuit (the ansatz).*

Figure T04.1 shows a complete pipeline for a **VQE** algorithm, extracted from (Tilly et al., 2022). The figure shows the two typical subroutines of a **VQA**: the quantum ones (yellow blocks) and the classical subroutines (blue blocks). The ansatz of the **VQE** is represented by the *2a* block, while the blocks *2b* and *2c* depict the Hamiltonian to be solved. The optimization routines are represented by the blue blocks *2d* and *3*.

In order to achieve a successful solution, when using **VQE** algorithms, it is mandatory for the two types of code (quantum and classical code) to execute properly in their respective platforms. Therefore, it is necessary to develop benchmark proposals that assess the performance of both hardware devices involved in these kinds of algorithms.

Originally, the Parent Hamiltonian problem, **PH Kernel**, was proposed as a benchmark problem for **VQE** (Kobayashi et al., 2022) from the perspective of evaluating optimizers (blue blocks *2d* and *3* in Figure T04.1). In this document, we use the **PH Kernel** for assessing the quantum subroutine corresponding to the yellow blocks of Figure T04.1.

The **PH Kernel** satisfies the three main requirements of the **NEASQC** benchmark methodology:

1. A mathematical definition of the **Kernel** can be provided with sufficient accuracy to enable the construction of a standalone circuit (refer to sections 2.2).
2. The **Kernel** can be defined for a configurable number of qubits.
3. The output can be verified through a classical computation.

2.2. Kernel Description

The **PH Kernel** can be defined in the following way:

Let $|\Psi(\vec{\theta})\rangle$ be the state of a given *ansatz*. The **PH Kernel**, aims to find a particular Hamiltonian (the parent Hamiltonian), H^{PH}, such that the ansatz is its ground state with energy equal to 0:

$$H^{PH} |\Psi(\vec{\theta})\rangle = E_0 |\Psi(\vec{\theta})\rangle \text{ with } E_0 = 0. \tag{T04.2.1}$$

In the original approximation, (Kobayashi et al., 2022), given an ansatz $|\Psi(\vec{\theta})\rangle$, the main idea is setting the parameter vector to $\vec{\theta}^*$ (this would be the optimal parameter vector) and obtain its particular parent Hamiltonian. Then, the **VQE** algorithm would be used by initializing the parameter vector to $\vec{\theta}_0$ (which should be different from $\vec{\theta}^*$), and an optimiser, which is the element to be tested, is used to obtain the minimum of the energy. If the optimizer works properly then the obtained ground state energy, E_0, should be close to 0 and the final parameter vector $\vec{\theta}_f$ should be similar to the optimal ones ($\vec{\theta}_f \sim \vec{\theta}^*$).

In the framework of the **TNBS**, the **PH Kernel** is proposed to assess the performance of a quantum device for executing the quantum part of the **VQE** algorithm. The **PH Kernel**, then, can be summarized as follow:

Given:

- A fixed ansatz.
- A fixed parameter vector $\vec{\theta}^*$ for the ansatz.
- The corresponding parent Hamiltonian for the ansatz with the fixed $\vec{\theta}^*$

the **PH Kernel** consists on executing a complete quantum step of the **VQE** algorithm. This implies executing the block *2a* of Figure T04.1 (the ansatz with the fixed parameters) followed by the block *2b* (the parent Hamiltonian) of the same figure and, finally, the corresponding measurements of block *2c* for getting the ground state energy of the ansatz under the action of the parent Hamiltonian. Ideally, this energy should be close to 0.

One of the mandatory inputs of the **PH Kernel** is to build the corresponding parent Hamiltonian of the input ansatz. This will be explained in the following sub-sections.

2.2.1. Getting the parent Hamiltonian: naive method

Given a n-qubit input *ansatz*, a fixed parameter vector $\vec{\theta}^*$ and its associated state $|\Psi\rangle = |\Psi(\vec{\theta}^*)\rangle$ the following steps should be performed for computing its corresponding parent Hamiltonian:

1. Compute the associated $2^n \times 2^n$ density matrix, $\rho(\vec{\theta})$, of the state of the ansatz, see equation (T04.2.2).

$$\rho(\vec{\theta}) = |\Psi\rangle \langle\Psi| \tag{T04.2.2}$$

2. Compute the **null space**[1] of the density matrix. The **null space** of a matrix is the set of linearly independent vectors such that, the product of the matrix with these vectors is zero, as shown in equation (T04.2.3).

$$\text{Null Space}(\rho) = \{|v^i\rangle \;/\; \rho\,|v^i\rangle = 0,\ i = 0, 1, \cdots m-1 \text{ with } m \leq dim(\rho);\ \langle v^i|v^j\rangle = \delta_{ij}\} \tag{T04.2.3}$$

[1]Null space of a matrix is also known as the kernel of the matrix. To avoid confusion with **TNBS Kernel** definition we prefer the null space term.

3. Notice that the m vectors of the **null space** of ρ are orthogonal to the state of the ansatz. This condition can be obtained from the definition of the **null space**, equation (T04.2.3), as seen in (T04.2.4).

$$\rho \left|v^i\right\rangle = 0 = \left|\Psi\right\rangle \left\langle \Psi | v^i \right\rangle = 0 \rightarrow \left\langle \Psi | v^i \right\rangle = 0 \text{ for } i = 0, 1, \cdots, m-1 \tag{T04.2.4}$$

4. Create the correspondent projectors of each $|v^i\rangle$, for $i = 0, 1, \cdots, m-1$, as shown in equation (T04.2.5). The projectors will be matrices of $2^n \times 2^n$.

$$h^i = \left|v^i\right\rangle \left\langle v^i\right| \tag{T04.2.5}$$

5. Using condition (T04.2.4) it can be shown that the product of each projector with the state $|\Psi\rangle$ is 0, as seen in equation (T04.2.6).

$$h^i \left|\Psi\right\rangle = \left|v^i\right\rangle \left\langle v^i | \Psi \right\rangle = 0 \tag{T04.2.6}$$

6. Finally, compute the **PH** for the state $|\Psi\rangle$ given as (T04.2.7).

$$H^{PH} = \sum_{i=0}^{m-1} h^i \tag{T04.2.7}$$

By construction, the built **PH** from (T04.2.7) satisfies the equation (T04.2.1) as can be seen in the following reasoning where equations (T04.2.5) and (T04.2.6) were used.

$$H^{PH} \left|\Psi\right\rangle = \sum_{i=0}^{m-1} h^i \left|\Psi\right\rangle = \sum_{i=0}^{m-1} \left|v^i\right\rangle \left\langle v^i | \Psi \right\rangle = 0$$

H^{PH} will be a matrix of $2^n \times 2^n$.

In the **VQE**, given an input Hamiltonian matrix, the most common way to implement it, into a quantum circuit, is by computing its linear decomposition into the basis of n-generalized Pauli matrices.

A n-generalized Pauli matrix is a $2^n \times 2^n$ matrix resulting from a n-Kronecker product, as shown in equation (T04.2.8), of the typical 2×2 Pauli matrices, see (T04.2.9).

$$\sigma_I^n = \bigotimes_{j=0}^{n-1} \sigma_{i_j} = \sigma_{i_0} \otimes \sigma_{i_1} \cdots \otimes \sigma_{i_{n-1}}. \text{ with } i_j \in \{0, 1, 2, 3\} \tag{T04.2.8}$$

$$\sigma_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \; \sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \; \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \; \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{T04.2.9}$$

There are 4^n different n-generalized Pauli matrices that form a basis for the group of all the possible $2^n \times 2^n$ matrices. So, in **VQE**, the H^{PH} should be decomposed into this n-generalized Pauli matrices basis as show in equation (T04.2.10)

$$H^{PH} = \sum_{I=0}^{4^n-1} a_I \sigma_I^n = \sum_{i_0, i_1, \cdots, i_{n-1}=0}^{3} a_{i_0, i_1, \cdots, i_{n-1}} \sigma_{i_0} \otimes \sigma_{i_1} \cdots \otimes \sigma_{i_{n-1}} \tag{T04.2.10}$$

The coefficients a_I of the linear combination decomposition can be obtained by computing the Frobenius norm of the product of the H^{PH} with the corresponding σ_I^n as can be seen in equation (T04.2.11).

$$a_I = \frac{\text{Tr}(H^{PH} \sigma_I^n)}{2^n} \tag{T04.2.11}$$

The main limitation of this method is that the number of σ_I^n matrices in the linear decomposition scale with 4^n. This implies, for example, that for a $n = 12$ qubit Hamiltonian the number of mandatory σ_I^n matrices is: 16777216. So this naive approach of computing the **PH** is computationally expensive and unaffordable when the number of qubits increases.

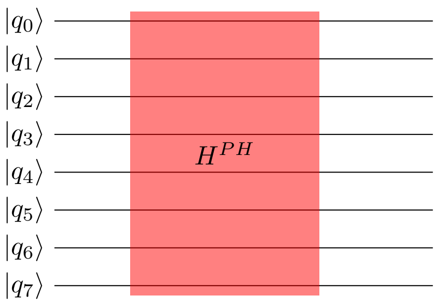


*Figure T04.2: Example of a circuit where naive **PH** method was used. In this case, all the particles (qubits) have an all-to-all interaction.*

The naive method for computing the **PH** assumes that the particles involved with the Hamiltonian have an all-to-all interaction. This behaviour is schematized in Figure T04.2 where the Hamiltonian affects simultaneously all the qubits of the circuit.

2.2.2. Getting the parent Hamiltonian: local Hamiltonian method

The local Hamiltonian method for computing the **PH**, (Kobayashi et al., 2022), is a more efficient method from a computing perspective. The main idea is computing a local parent Hamiltonian for each particle (qubit) where the interaction affects only near particles (qubits). Figure T04.3 shows a schematic example of this idea. As can be seen, each particle (qubits) has an associated local Hamiltonian that affects only adjacent particles. As schematized in the figure, the different local Hamiltonian can affect different numbers of particles (qubits) and each one is a parent Hamiltonian ($H_i |\Psi(\theta)\rangle = 0$).

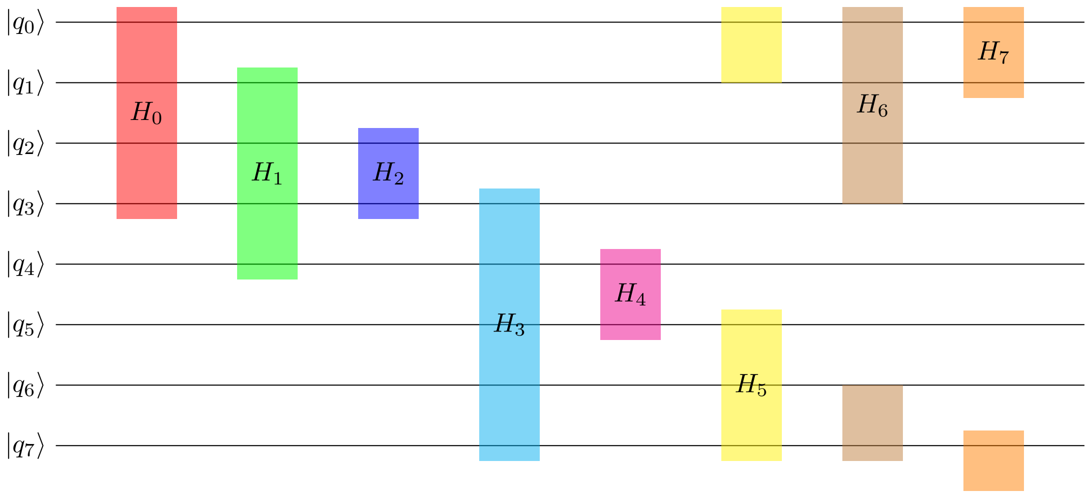


Figure T04.3: Example of a circuit where the local parent Hamiltonian method is used. As can be seen, each particle (qubit) has an associated local Hamiltonian, H_i, that affects only its neighbours

Before describing the local **PH** method, some concepts and notation will be addressed.

For a given n qubit ansatz, it is a mandatory step to compute its complete state by using any technique like **MPS** or **StateVector** simulation. So, if the state is expressed following the equation (T04.2.12) all the amplitudes b_k must be computed.

$$|\Psi\rangle = \sum_{k=0}^{2^n-1} b_k |k\rangle \tag{T04.2.12}$$

Additionally, it will be more useful to organize the different amplitudes as a n-rank tensor as shown in the equation (T04.2.13) where the Einstein summation convention (sum over repeated indices) is used. In this equation $b_k = \Psi_{i_0 i_1 \cdots i_{n-1}}$ where $k = \sum_{j=0}^{n-1} 2^{i_j}$ and $i_j = \{0, 1\}$ for $j = 0, 1, \cdots, n-1$.

$$|\Psi\rangle = \Psi_{i_0 i_1 \cdots i_{n-1}} |i_0 i_1 \cdots i_{n-1}\rangle \tag{T04.2.13}$$

In this framework, the density matrix in equation (T04.2.2), associated with the state $|\Psi\rangle$ can be organized as a $2n$-rank tensor as shown in equation (T04.2.14).

$$\rho^{j_0 j_1 \cdots j_{n-1}}_{i_0 i_1 \cdots i_{n-1}} = \Psi_{i_0 i_1 \cdots i_{n-1}} \Psi^{* j_0 j_1 \cdots j_{n-1}} \tag{T04.2.14}$$

Finally, the last ingredient is the reduced density matrix for m consecutive qubits from a fixed qubit i_k, ρ^m_k. For its computation all the qubits, except the consecutive set $\{i_k, i_{k+1}, \cdots, i_{k+m}\}$, should be traced out from ρ as seen in equation (T04.2.15), where the contracted indices are shown in red for visual purposes.

$$\begin{aligned}\rho^m_k &= \mathrm{Tr}_{i_l \notin \{i_k, i_{k+1}, \cdots, i_{k+m}\}}(\rho) = \rho^{j_k j_{k+1} \cdots j_{k+m}}_{i_k i_{k+1} \cdots i_{k+m}} = \\ &= \Psi_{\color{red}{i_0 i_1}\cdots i_k i_{k+1} \cdots i_{k+m} \color{red}{i_{k+m+1} \cdots i_{n-1}}} \Psi^{* \color{red}{i_0 i_1}\cdots j_k j_{k+1} \cdots j_{k+m} \color{red}{i_{k+m+1} \cdots i_{n-1}}}\end{aligned} \tag{T04.2.15}$$

For clarity, some examples of the used notation are provided. For a 5 qubit ansatz, the amplitude tensor will be: $\Psi_{i_0, i_1, i_2, i_3, i_4}$ and the associated density matrix will be $\rho^{j_0, j_1, j_2, j_3, j_4}_{i_0, i_1, i_2, i_3, i_4}$. The following computations could be done:

- Computation of the reduced density matrix from i_0 with $m = 1$: $\rho^{m=1}_0 = \rho^{j_0 j_1}_{i_0 i_1} = \Psi_{i_0 i_1 \color{red}{i_2 i_3, i_4}} \Psi^{*\ j_0 j_1 \color{red}{i_2 i_3 i_4}}$ (qubits i_2, i_3, i_4 are traced out).
- Computation of the reduced density matrix from i_1 with $m = 3$: $\rho^{m=3}_1 = \rho^{j_1 j_2 j_3 j_4}_{i_1 i_2 i_3 i_4} = \Psi_{\color{red}{i_0} i_1 i_2 i_3, i_4} \Psi^{*\ \color{red}{i_0} j_1 j_2 j_3 j_4}$ (qubit i_0 are traced out)
- Computation of the reduced density matrix from i_3 with $m = 2$: $\rho^{m=2}_3 = \rho^{j_3 j_4 j_0}_{i_3 i_4 i_0} = \Psi_{i_0 \color{red}{i_1 i_2} i_3, i_4} \Psi^{*\ j_0 \color{red}{i_1 i_2} j_3 j_4}$ (qubits i_1, i_2 are traced out)

With these definitions, the method for getting the local **PH** can be described as follows:

Given an input ansatz and its corresponding amplitude tensor, $\Psi_{i_0 i_1 \cdots i_{n-1}}$, for each qubit i_j, beginning with $j = 0$, the following steps should be carried out by starting with $m_j = 1$:

1. Compute the reduced density matrix for qubit i_j and m_j, $\rho^{m_j}_j$ using equation (T04.2.15).
2. Compute the rank of the reduced density matrix $\rho^{m_j}_j$: $rank(\rho^{m_j}_j)$.
3. Using the Rank-nullity theorem, determine if the **null space** of the reduced density matrix, null space($\rho^{m_j}_j$), can be computed:
   - If $dim(\rho^{m_j}_j) = rank(\rho^{m_j}_j)$, then, the **null space** cannot be computed. Go to step 1 with $m_j = m_j + 1$.
   - If $dim(\rho^{m_j}_j) > rank(\rho^{m_j}_j)$, then, the **null space** can be computed go to step 4.
4. Compute the **null space** of the reduced density matrix, null space($\rho^{m_j}_j$), see equation (T04.2.3).
5. Compute the corresponding projectors from the computed **null space** using equation (T04.2.5) and compute the local **PH**, $H^{m_j}_{i_j}$, using equation (T04.2.7). This Hamiltonian will be a $2^{m_j} \times 2^{m_j}$ matrix and only qubits from i_j to i_{j+m_j} will be affected by it. In addition, by construction, $H^{m_j}_{i_j}$ is a parent Hamiltonian over the affected qubits. Trivially this behaviour is extended to the rest of the circuit by doing nothing to the non-affected qubits.
6. Compute the linear combination decomposition of $H^{m_j}_{i_j}$ in the basis of m_j generalized Pauli matrices by using (T04.2.10) and (T04.2.11). So, for a $H^{m_j}_{i_j}$, a list of 4^{m_j} tuples $(\sigma^{i_j, m_j}_I, a^{i_j, m_j}_I)$ should be obtained, where σ^{i_j, m_j}_I are all the m_j generalized Pauli matrices and a^{i_j, m_j}_I, the corresponding decomposition coefficients ($I = 0, 1, \cdots 4^{m_j} - 1$).

7. It should be noted that the m_j-generalized Pauli matrices σ_I^{j,m_j} act over qubits from i_j to i_{j+m_j}. Trivially this m_j generalized Pauli matrices can be generalized to the overall circuit by doing the Kronecker's product with identity matrices in the non-affected qubits.
8. Repeat the complete process for qubit, i_{j+1} (setting the corresponding $m_{j+1} = 1$) until all qubits are processed.

At the end of the process, n local parent Hamiltonian, $H_{i_j}^{m_j}$, and their respective decomposition in generalized Pauli matrices, $(\sigma_I^{j,m_j}, a_I^{j,m_j})$, with $j = 0, 1, \cdots n-1$, and $I = 0, 1, \cdots 4^{m_j} - 1$, should be obtained. The final local **PH** is given by (T04.2.16).

$$H^{PH} = \sum_{j=0}^{n-1} H_{i_j}^{m_j} \tag{T04.2.16}$$

For the **PH Kernel** of the **TNBS** the mandatory parent Hamiltonian should be computed following this local parent Hamiltonian method.

3.Description of the benchmark test case

This section presents the complete description of the **BTC** for the **PH kernel**. Section 3.1 describes the problem addressed by the test case. Section 3.2 provides a high-level description of the case. Section 3.3 provides the execution workflow. Finally, section 3.4 documents how the results of such executions must be reported.

3.1. Description of the problem

The computation of the ground state energy of the ansatz presented in the original parent Hamiltonian paper (Kobayashi et al., 2022), depicted in Figure T04.4, under its corresponding local parent Hamiltonian is the proposed **BTC** of the **PH Kernel**.

As can be seen in Figure T04.4, the ansatz is built of several circuit layers, each one composed of parametrized R_x and R_z gates alternated by a ladder of controlled Z gates.

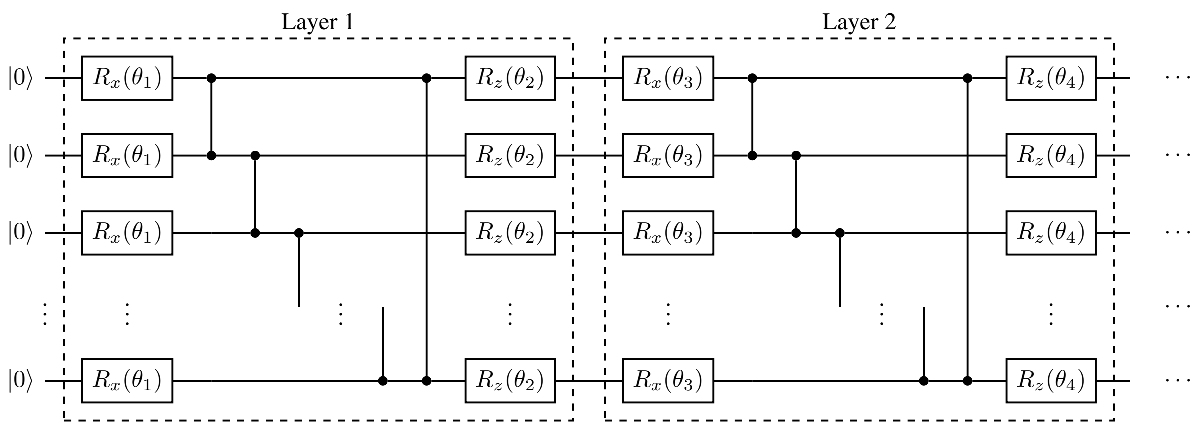


*Figure T04.4: Invariant translational ansatz proposed for **BTC**.*

For a fixed i layer the parameters θ_i (for R_x gate) and θ_{i+1} (for R_z gate) should be fixed as a function of the number of layers of the ansatz, n_l, following equation (T04.1.1).

$$\theta_i = (i+1)\delta\theta \; i = 0, 1, \cdots 2n_l - 1 \text{ where } \delta\theta = \frac{\pi}{4 * (n_l + 1)} \tag{T04.1.1}$$

The selected ansatz is a translational invariant one. For this type of ansatz the procedure provided in section 2.2.2, for computing the local parent Hamiltonian, can be executed only for the first qubit and the obtained Pauli decomposition should be replicated for the rest of the qubits of the circuit.

3.2. Benchmark test case description

This section provides a step-by-step workflow of the **BTC** for the **PH kernel**.

Given a fixed number of qubits, n, and a selected circuit depth, n_l, the quantum **VQE** step, complete yellow block code in Figure T04.1, should be executed for obtaining the corresponding energy of the ansatz under the local parent Hamiltonian. The inputs of this **VQE** step are:

1. The desired ansatz implementation following Figure T04.4.
2. The parameters of the ansatz following equation (T04.1.1)
3. The complete Pauli decomposition of the local parent Hamiltonian: this is the generalized Pauli matrices, σ_I^{j,m_j}, see equation (T04.2.8), and their corresponding Pauli coefficients a_I^{j,m_j}, see equation (T04.2.11)

The **VQE** step should be executed for $n_{shots} = 10000$ and **Ground State Energy** E_0 should be returned. The value of E_0 should be close to 0. The complete time for the **VQE** step should be measured and labelled as the **elapsed time**. If possible, the time of the pure quantum part should be measured separately as the **quantum time**.

The following remarks should be taken into account:

In a naive approach, the procedure would be to build the ansatz with the corresponding parameters, the I th Pauli matrix, σ_I, would be added to the circuit and finally, the state would be measured n_{shots} and the obtained mean would be multiplied by the corresponding Pauli coefficient a_I, and the energy term E_I would be obtained. This procedure would be performed for all the Pauli matrices and the ground state energy would be $E_0 = \sum_I E_I$.

There exist, however, different algorithms that allow grouping Pauli matrix to decrease the number of executions. Any algorithm for boosting and optimizing this **VQE** step can be used but the number of shots for each individual term should be $n_{shots} = 10000$.

As can be seen the Pauli decomposition of the local parent Hamiltonian is a mandatory input for the **BTC** of the **PH Kernel**. This computation should be done independently of the execution of the **Benchmark** and it should not be included in the **Benchmark** report nor included in the **elapsed time** computations.

In the folder configuration_files of the **WP3_Benchmark** **NEASQC** GitHub repository this Pauli decomposition, as *csv* files, can be found for several qubits (from 3 to 30) and for different number of layers (from 1 to 4). Additionally, the parameters used, following equation (T04.1.1), were stored too. See section A.1.2 of the Appendix for more information.

3.3. Complete benchmark procedure

To execute a complete **PH Benchmark**, subsequent steps should be followed:

1. Fix in advance the different numbers of qubits to be tested, for example from n=4 to n=8 (using the configuration files in the **WP3_Benchmark** **NEASQC** GitHub repository until 30 qubits can be tested, see section A.1.2)
2. For each possible number of qubits n, different number of layers n_l should be tested, for example from $n_l = 1$ to $n_l = 4$ (using the configuration files in the **WP3_Benchmark** **NEASQC** GitHub repository 1, 2, 3 and 4 number of layers can be tested, see section A.1.2).
3. For each possible combination of number of qubits n and number of layers n_l following steps must be performed
   a) Execute a warm-up step consisting of:
      i. Execute 10 iterations of the **BTC**, section 3.2, and compute the standard deviation of the ground state energy E_0, s_{E_0}, and the mean and the standard deviation for the **elapsed time**, μ_t, s_t.
      ii. Compute the number of repetitions, M_{E_0}, for having an absolute error for the ground state energy, E_0, of 0.01, $r_{E_0} = 0.01$, with a confidence level of 95%, $\alpha = 0.95$, following equation (T04.3.2), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for α.

$$M_{E_0} = \left(\frac{s_{E_0} Z_{1-\frac{\alpha}{2}}}{r_{E_0}}\right)^2 \tag{T04.3.2}$$

      iii. Compute the number of repetitions mandatory, M_t, for having a relative error for the **elapsed time** of 5%, $r_t = 0.05$, with a confidence level of 95%, $\alpha = 0.05$, following (T04.3.3), where $Z_{1-\frac{\alpha}{2}}$ is the percentile for α.

$$M_t = \left(\frac{s_t Z_{1-\frac{\alpha}{2}}}{r_t \mu_t}\right)^2 \tag{T04.3.3}$$

   b) Execute the complete **BTC**, section 3.2, $M = \max(M_t, M_{E_0})$ times. M must be greater than 5.
   c) Compute the mean and the standard deviation for E_0, the **elapsed time** and the **quantum time**, if possible.

The method used to calculate the number of repetitions, M, in the previous procedure guarantees that the **Ground State Energy**, E_0, will have an absolute error lower than $r_{E_0} = 0.01$, and the **elapsed time** will have a relative error lower than $r_t = 5\%$ with a confidence level of 95%.

3.4. Benchmark report

Finally, the results of the complete benchmark execution must be reported into a valid JSON file following the JSON schema *NEASQC.Benchmark.V2.Schema.json* provided in the document D3.5: The NEASQC Benchmark Suite of the NEASQC project.

The results of the **Benchmark** should be stored in the field *Benchmarks*, under the sub-field *Results*. This sub-field of the JSON has associated a list of elements, where each element is a dictionary with the complete result information for a **Benchmark** execution: this is, for a fixed number of qubits n (sub-field *NumberOfQubits* of the dictionary) and for a fixed number of layers n_l (a new sub-field called *AnsatzDepth* was created for storing this information). Each one of these dictionaries stores, under the sub-field *Metrics*, the mandatory verification metric: the **Ground State Wnergy** one is stored under the name *gse*. Additionally, the mean elapsed time must be reported in the *TotalTime* sub-field and its standard deviation in the *SigmaTotalTime* one.

List of Acronyms

Term	Definition
BTC	Benchmark Test Case
JSON	JavaScript Object Notation
NEASQC	NExt ApplicationS of Quantum Computing
NISQ	Noise Intermediate-Scale Quantum
PH	Parent Hamiltonian
TNBS	The **NEASQC** Benchmark Suite
VQA	Variational Quantum Algorithm
VQE	Variational Quantum Eigensolver

***Table T04.1**: Acronyms and Abbreviations*

List of Figures

List of Tables

List of Listings

Bibliography

Cerezo, M., Arrasmith, A., Babbush, R., Benjamin, S. C., Endo, S., Fujii, K., McClean, J. R., Mitarai, K., Yuan, X., Cincio, L., & Coles, P. J. (2021). Variational quantum algorithms. Nature Reviews Physics, 3(9), 625–644. https://doi.org/10.1038/s42254-021-00348-9

Kobayashi, F., Mitarai, K., & Fujii, K. (2022). Parent hamiltonian as a benchmark problem for variational quantum eigensolvers. Phys. Rev. A, 105, 052415. https://doi.org/10.1103/PhysRevA.105.052415

Peruzzo, A., McClean, J., Shadbolt, P., Yung, M.-H., Zhou, X.-Q., Love, P. J., Aspuru-Guzik, A., & O'Brien, J. L. (2014). A variational eigenvalue solver on a photonic quantum processor. Nature Communications, 5(1). https://doi.org/10.1038/ncomms5213

Tilly, J., Chen, H., Cao, S., Picozzi, D., Setia, K., Li, Y., Grant, E., Wossnig, L., Rungger, I., Booth, G. H., & Tennyson, J. (2022). The Variational Quantum Eigensolver: A review of methods and best practices. Physics Reports, 986, 1–128. https://doi.org/https://doi.org/10.1016/j.physrep.2022.08.003

A.NEASQC test case reference

As pointed out in deliverable **D3.5: The NEASQC Benchmark Suite** each proposed **Benchmark** for **TNBS**, must have a complete Eviden myQLM compatible software implementation. For the **PH Benchmark**, this implementation can be found in the **tnbs/BTC_04_PH** folder of the **WP3_Benchmark** **NEASQC** GitHub repository. Additionally, the execution of a **Benchmark** must generate a complete result report into a separate JSON file, that must follow **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable.

The **tnbs/BTC_04_PH** locations contains the following folders and files:

- **PH** folder: with all the Python modules mandatory for executing the complete workflow of the **PH Kernel BTC** as explained in section 3.2. Additionally, it contains several modules for building parent Hamiltonians of any input ansatz, as explained in sections 2.2.1 and 2.2.2 (for naive and local methods respectively).
- **configuration_files** folder: where Pauli decomposition of local **PH** for **BTC** ansatzes (see Figure T04.4) from 3 to 30 qubits and for a number of layers from 1 to 4 were stored as *csv* files. Additionally, *csv* files with the ansatz parameters, see equation (T04.1.1), were stored too.
- **my_benchmark_execution.py**
- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

The modules of **PH** folder, the files from the **configuration_files** and the **my_benchmark_execution.py** file allows to execute a complete **PH Benchmark** using the Eviden myQLM library. Section A.1 documents, these files. The other script files are related to benchmark report generation and are properly explained in section A.2.

A.1. NEASQC implementation of benchmark test case.

A.1.1. my_benchmark_execution.py

This script is a modification of the correspondent template script located in tnbs/templates folder of the **WP3_Benchmark** repository. Following the recommendations of Annex B of the deliverable **D3.5: The NEASQC Benchmark Suite** the **run_code**, **compute_samples**, **summarize_results** and the **build_iterator** functions were modified. Meanwhile, the **KERNEL_BENCHMARK** class was not modified. In the following sections, the software adaptations for the **PH Benchmark** are presented.

run_code

Listing T04.1 shows the modifications performed into the **run_code** function for the **PH Benchmark**. The main functionality is executing the **BTC**, as explained in section 3.2, for a fixed number of qubits, n, a number of layers n_l and its corresponding local parent Hamiltonian Pauli decomposition. The two first parameters should be passed as a 2-element Python tuple (*iterator_step*). The execution will be done *repetitions* number of times and all the mandatory metrics will be gathered as pandas DataFrame (*metrics*). The *stage_bench* is a boolean variable that indicates if the step is executed in the pre-benchmark (step *3.a* in section 3.3) or in the benchmark stage (step *3.b* in section 3.3).

```

def run_code(iterator_step, repetitions, stage_bench, **kwargs):
    """
    For configuration and execution of the benchmark kernel.

    Parameters
    ----------

    iterator_step : tuple
        tuple with elements from iterator built from build_iterator.
    repetitions : list
        number of repetitions for each execution
    stage_bench : str
```

```
        benchmark stage. Only: benchmark, pre-benchamrk
    kwargs : keyword arguments
        for configuration of the benchmark kernel

    Returns
    _______

    metrics : pandas DataFrame
        DataFrame with the desired metrics obtained for the integral computation
    save_name : string
        Desired name for saving the results of the execution

    """

    if stage_bench not in ["benchmark", "pre-benchmark"]:
        raise ValueError(
            "Valid values for stage_bench: benchmark or pre-benchmark")
    if repetitions is None:
        raise ValueError("samples CAN NOT BE None")
    #Here the code for configuring and execute the benchmark kernel
    kernel_configuration = deepcopy(kwargs.get("kernel_configuration", None))
    del kernel_configuration["gse_error"]
    del kernel_configuration["time_error"]
    del kernel_configuration["depth"]
    if kernel_configuration is None:
        raise ValueError("kernel_configuration can not be None")
    # Configuring kernel

    nqubits = str(iterator_step[0]).zfill(2)
    depth = str(iterator_step[1])
    logger.info("Creating ansatz circuit")
    ansatz_conf = {
        "nqubits" :int(nqubits),
        "depth" : int(depth),
    }
    circuit = ansatz_selector("simple01", **ansatz_conf)
    # Formating Parameters
    base_fn = "configuration_files/nqubits_{}_depth_{}".format(nqubits, depth)
    param_file = base_fn + "_parameters.csv"
    logger.info("Loading Parameters from: %s", param_file)

    parameters_pdf = pd.read_csv(param_file, sep=";", index_col=0)
    circuit, _ = angles_ansatz01(circuit, parameters_pdf)

    # Loading PH Pauli decomposition
    pauli_file = base_fn + "_pauli.csv"
    logger.info("Loading PH Pauli decomposition from: %s", pauli_file)
    # Loading Pauli
    pauli_pdf = pd.read_csv(pauli_file, sep=";", index_col=0)
    affected_qubits = [ast.literal_eval(i_) for i_ in list(pauli_pdf["Qbits"])]
    pauli_pdf["Qbits"] = affected_qubits

    # Executing VQE step
    logger.info("Executing VQE step")
    nb_shots = kernel_configuration.get("nb_shots", None)
    if nb_shots is None:
        nb_shots = 10000
    truncation = kernel_configuration.get("truncation", None)
    t_inv = kernel_configuration.get("t_inv", None)
    if t_inv is None:
        t_inv = True

    vqe_conf = {
        "qpu" : get_qpu(kernel_configuration["qpu_ph"]),
        "nb_shots": nb_shots,#kernel_configuration["nb_shots"],
        "truncation": truncation, #kernel_configuration["truncation"],
        "t_inv": t_inv,#kernel_configuration["t_inv"],
        "filename": None,
        "save": False,
    }
    list_of_metrics = []
    for i in range(repetitions):
        exe_ph = PH_EXE(circuit, pauli_pdf, int(nqubits), **vqe_conf)
```

```
87          exe_ph.run()
88          pdf_info = pd.DataFrame(
89              [int(nqubits), int(depth)], index=["nqubits", "depth"]).T
90          list_ = [
91              pdf_info,
92              pd.DataFrame(kernel_configuration, index=[0]),
93              exe_ph.pdf_result
94          ]
95          pdf_info = pd.concat(list_, axis=1)
96          list_of_metrics.append(pdf_info)
97      metrics = pd.concat(list_of_metrics)
98      metrics.reset_index(drop=True, inplace=True)
99      metrics["elapsed_time"] = metrics["observable_time"] + \
100         metrics["quantum_time"]
101     if stage_bench == "pre-benchmark":
102         # Name for storing Pre-Benchmark results
103         save_name = "pre_benchmark_nq_{}_depth_{}.csv".format(
104             iterator_step[0],
105             iterator_step[1]
106             )
107     if stage_bench == "benchmark":
108         # Name for storing Benchmark results
109         save_name = kwargs.get("csv_results")
110         #save_name = "pre_benchmark_step_{}.csv".format(n_qbits)
111     return metrics, save_name
```

*Listing T04.1: run_code function for executing a step of the **BTC** of the **PH kernel**, as explained in section 3.2*

The workflow of the *run_code* can be schematized in the following steps:

1. Built of the ansatz (Figure T04.4) for the requested n and n_l as a myQLM circuit. Line 49 of Listing T04.1. This is done using the function *ansatz_selector* from **PH.ansatzes** module.
2. Loading the correspondent parameters of the ansatz following equation (T04.1.1). This is done by loading the corresponding *csv* file, line 55, for n and n_l from the **configuration_files** folder (see Section A.1.2). These files are the ones that have a *_parameters* in their name. The parameters are then loaded into the myQLM ansatz circuit (line 56) using function *angles_ansatz01* from **PH.ansatzes** module.
3. The 2 first steps are equivalent to the block *2a* of Figure T04.1.
4. Loading the complete Pauli decomposition of the corresponding local parent Hamiltonian for n qubits and n_l layers from the proper *csv* file, line 62, from the **configuration_files** folder (see Section A.1.2). They are located in the files that have a *_pauli* in their names. The Pauli terms would correspond to the block *2b* of Figure T04.1.
5. Finally the complete VQE step execution (blocks *2a*, *2b* and *2c* of Figure T04.1 is executed in lines 86 and 87 by creating the **PH_EXE** class, from module **PH.execution_ph**, and invoking its *run* method.
6. The mandatory metrics results, **Ground State Energy** and **elapsed time** are gathered in the subsequent lines. A pandas DataFrame is built with this information. Additionally, the configuration parameters of the execution are added to the DataFrame for traceability purposes.

Steps 5 and 6 are done several times (*repetitions* number of times) for generating a final DataFrame where all results are gathered (this the *metrics* one).

The return of the *run_code* function are the *metrics* DataFrame and filename for storing results if desired.

compute_samples

Listing T04.2 shows the implementation of the **compute_samples** function for the **PH Benchmark**. The main objective is to codify a strategy for computing the number of times the **BTC**, as explained in section 3.2, should be executed, for getting some desired statistical significance (see *3.a.ii* and *3.a.iii* of section 3.3). This function would implement equations (T04.3.2) and (T04.3.3) and compute the corresponding maximum as explained in *3.b* of Section 3.3.

```

def compute_samples(**kwargs):
    """
    This function computes the number of executions of the benchmark
    for ensuring an error r with a confidence level of alpha

    Parameters
    ----------

    kwargs : keyword arguments
        For configuring the sampling computation

    Returns
    _______

    samples : pandas DataFrame
        DataFrame with the number of executions for each integration interval

    """

    #Configuration for sampling computations

    #Desired Confidence level
    alpha = kwargs.get("alpha", 0.05)
    if alpha is None:
        alpha = 0.05
    metrics = kwargs.get("pre_metrics")
    bench_conf = kwargs.get("kernel_configuration")

    #Code for computing the number of samples for getting the desired
    #statistical significance. Depends on benchmark kernel

    from scipy.stats import norm
    zalpha = norm.ppf(1-(alpha/2)) # 95% of confidence level

    # Error expected for the Groud State Energy
    error_gse = bench_conf.get("gse_error", 0.01)
    if error_gse is None:
        error_gse = 0.01
    std_ = metrics[["gse"]].std()
    samples_gse = (zalpha * std_ / error_gse) ** 2

    # Relative Error for elapsed time
    time_error = bench_conf.get("time_error", 0.05)
    if time_error is None:
        time_error = 0.05
    mean_time = metrics[["elapsed_time"]].mean()
    std_time = metrics[["elapsed_time"]].std()
    samples_time = (zalpha * std_time / (time_error * mean_time)) ** 2

    #Maximum number of sampls will be used
    samples_ = pd.Series(pd.concat([samples_time, samples_gse]).max())

    #Apply lower and higher limits to samples
    #Minimum and Maximum number of samples
    min_meas = kwargs.get("min_meas", None)
    if min_meas is None:
        min_meas = 5
    max_meas = kwargs.get("max_meas", None)
    samples_.clip(upper=max_meas, lower=min_meas, inplace=True)
    samples_ = samples_[0].astype(int)
    return samples_
```

Listing T04.2: compute_samples function for codifying the strategy for computing the number of repetitions for the ***BTC*** *of the* ***PH kernel****.*

summarize_results

Listing T04.3 shows the implementation of the **summarize_results** function for the **PH Benchmark**. The main objective is post-processing the results of the complete **Benchmark** execution, as described in step *3-c* of Section 3.3.

This function expects that the results of the complete benchmark execution have been stored in a *csv* file. The function loads this file into a pandas DataFrame that is post-processed properly.

```

def summarize_results(**kwargs):
    """
    Create summary with statistics
    """

    folder = kwargs.get("saving_folder")
    csv_results = folder + kwargs.get("csv_results")

    #Code for summarize the benchamark results. Depending of the
    #kernel of the benchmark
    pdf = pd.read_csv(csv_results, index_col=0, sep=";")
    pdf["classic_time"] = pdf["elapsed_time"] - pdf["quantum_time"]
    pdf.fillna("None", inplace=True)
    group_columns = [
        "nqubits", "depth", "t_inv", "qpu_ph",
        "nb_shots", "truncation"]
    metric_columns = ["gse", "elapsed_time", "quantum_time", "classic_time"]
    results = pdf.groupby(group_columns)[metric_columns].agg(
        ["mean", "std", "count"])
    results = results.replace('None', None)
    return results
```

Listing T04.3: summarize_results function for summarizing the results from an execution of the ***PH Benchmark***

build_iterator

Listing T04.4 shows the implementation of the **build_iterator** function for the **PH Benchmark**. The main objective is to create a Python iterator for executing the desired complete **BTC**. In this case, the iterator creates a list with all the possible combinations of the desired number of qubits, n and the number of layers of the ansatz, n_l.

```
def build_iterator(**kwargs):
    """
    For building the iterator of the benchmark
    """
    import itertools as it

    list4it = [
        kwargs["list_of_qbits"],
        kwargs["kernel_configuration"]["depth"]
    ]

    iterator = it.product(*list4it)

    return list(iterator)
```

Listing T04.4: build_iterator function for creating the iterator of the complete execution of the ***PH Benchmark***

KERNEL_BENCHMARK class

No modifications were made to the **KERNEL_BENCHMARK** class. This Python class defines the complete benchmark workflow, section 3.3, and its *exe* method executes it properly by calling the correspondent functions (*run_code*, *compute_samples*, *summarize_results*, *build_iterator*). Each time a **Benchmark** step is executed, as defined in section 3.3, the result is stored in a given *CSV* file.

The only mandatory modification is configuring properly the input keyword arguments, at the end of the **my_benchmark_execution.py** script. These parameters will configure the complete benchmark workflow, and additional options (as the name of the *CSV* files). Listing T04.5 shows an example for configuring an execution of a **PH Benchmark**.

```

if __name__ == "__main__":


    #Anstaz
    depth = [1, 2, 3, 4]
    qpu_ph = "c"

    kernel_configuration = {
        #Ansatz
        "depth": depth,
        "t_inv": True,
        # Ground State Energy
        "qpu_ph" : qpu_ph,
        "nb_shots" : 10000,
        "truncation": None,
        # Saving
        "save": True,
        "folder": None,
        # Errors for confidence level
        "gse_error" : None,
        "time_error": None,
    }

    list_of_qbits = [3, 4, 5, 6, 7]
    benchmark_arguments = {
        #Pre benchmark sttuff
        "pre_benchmark": True,
        "pre_samples":  None,
        "pre_save": True,
        #Saving stuff
        "save_append" : True,
        "saving_folder": "./Results/",
        "benchmark_times": "kernel_times_benchmark.csv",
        "csv_results": "kernel_benchmark.csv",
        "summary_results": "kernel_SummaryResults.csv",
        #Computing Repetitions stuff
        "alpha": None,
        "min_meas": None,
        "max_meas": None,
        #List number of qubits tested
        "list_of_qbits": list_of_qbits,
    }

    #Configuration for the benchmark kernel
    benchmark_arguments.update({"kernel_configuration": kernel_configuration})
    kernel_bench = KERNEL_BENCHMARK(**benchmark_arguments)
    kernel_bench.exe()
```

*Listing T04.5: Example of configuration of a complete **PH Benchmark** execution. This part of the code should be located at the end of the **my_benchmark_execution.py** script*

As can be seen in Listing T04.5, the input dictionary that **KERNEL_BENCHMARK** class needs, *benchmark_arguments*, have several keys that allow to modify the benchmark workflow, like:

- *pre_benchmark*: For executing or not the *pre-benchmark* step.
- *pre_samples*: number of repetitions of the benchmark step.
- *pre_save*: For saving or not the results from the *pre-benchmark* step.
- *saving_folder*: Path for storing all the files generated by the execution of the **KERNEL_BENCHMARK** class.
- *benchmark_times*: name for the *csv* file where the initial and the final times for the complete benchmark execution will be stored.
- *csv_results*: name for the *csv* file where the obtained metrics for the different repetitions of the benchmark step will be stored (so the different metrics obtained during step 2 from section 3.3 will be stored in this file)
- *summary_results*: name for the *csv* file where the post-processed results (using the *summarize_results*) will be stored (so the statistics over the metrics obtained during step 3 of section 3.3 will be stored in this file)
- *list_of_qbits*: list with the different number of qubits for executing the complete **Benchmark**.
- *alpha*: for configuring the desired confidence level α
- *min_meas*: for low limiting the number of executions a benchmark step should be executed during the benchmark stage.
- *max_meas*: for high limiting the number of executions a benchmark step should be executed during the benchmark stage.

Additionally, the *kernel_configuration* key is used for configuring the kernel execution. The following keys can be provided for configuring it:

- *depth*: for configuring the number of layers, n_l, of the ansatz.
- *t_inv*: for specifying if the ansatz is or not translational invariant. For **PH Benchmark** should be set to True.
- *qpu_ph*: a string for selecting the quantum process unit (**QPU**)
- *nb_shot*: number of shots for measuring the energy of the **VQE** step. For **PH Benchmark** should be set to 10000.
- *truncation*: for discarding Pauli coefficients lower than $10^{-truncation}$. For **PH Benchmark** should be set to None.
- *gse_error*: for changing the desired absolute error for the **Ground State Energy** metric.
- *time_error*: for changing the desired relative error for the **elapsed time**

In general, most of the keys should be fixed to *None* for executing the **Benchmark** according to the guidelines of the **PH Benchmark**

For executing the **Benchmark** following command should be used:

python my_benchmark_execution.py

A.1.2. configuration_files folder

In this folder, there are stored the *csv* files mandatory for executing the **BTC** of the **PH kernel** using the *run_code* function from **my_benchmark_execution** (see the corresponding paragraph of section A.1.1). There are two types of files:

- *_parameters.csv*: These files have the parameters of the **BTC** ansatz following equation (T04.1.1) for a fixed number of qubits n and a fixed number of layers n_l. Table T04.2 shows an example of the content of these type of files.

	key	value
0	\theta_0	0.19634954084936207
1	\theta_1	0.39269908169872414
2	\theta_2	0.5890486225480862
3	\theta_3	0.7853981633974483
4	\theta_4	0.9817477042468103
5	\theta_5	1.1780972450961724

Table T04.2*: Example of the content of a _parameters.csv file (nqubits_04_depth_3_parameters.csv)*

- *_pauli.csv*: These files have the Pauli decomposition of the corresponding parent Hamiltonian for the **BTC** ansatz for a fixed number of qubits n and a fixed number of layers n_l. Table T04.3 shows an example of the content of these type of files. In this case the files have 4 columns:
  - The first column is a dummy index.
  - The second column, **PauliCoefficients**, holds the Pauli coefficients a_I, see equation (T04.2.11).
  - The third column, **PauliStrings**, holds the m_j-generalized Pauli matrix σ_I, see equation (T04.2.8), provided as Pauli strings ($\sigma_0 : I, \sigma_1 = X, \sigma_2 = Y, \sigma_3 = Z$).
  - The fourth column, **Qbits**, is the affected qubits for each of the elements of the corresponding Pauli strings.

The names of the different files follow the pattern:

*nqubits_n_depth_*n_l

where **n** is the number of qubits and n_l the number of layers followed by *_parameters.csv* or *_pauli.csv*.

There are files with information for parameters and Pauli decompositions for ansatzes from 3 to 30 qubits and for each qubit for a number of layers from 1 to 4 (there are 112 files for parameters and for paulis).

	PauliCoefficients	PauliStrings	Qbits
0	0.7500000000000003	III	[0, 1, 2]
1	-0.00561596334353707	IIX	[0, 1, 2]
2	-0.04441440856772065	IIY	[0, 1, 2]
3	-0.04249275519392258	IIZ	[0, 1, 2]
4	0.007083873530394136	IXI	[0, 1, 2]
5	0.02485387932781642	IXX	[0, 1, 2]
6	0.029646274839496427	IXY	[0, 1, 2]
7	-0.0298965933873333	IXZ	[0, 1, 2]
8	-0.04728533359686948	IYI	[0, 1, 2]
9	0.00205881519227738	IYX	[0, 1, 2]
10	-6.01116541139278e-05	IYY	[0, 1, 2]
11	-0.06548679398263388	IYZ	[0, 1, 2]
12	-0.04106203612667689	IZI	[0, 1, 2]
13	0.012413011400994774	IZX	[0, 1, 2]
14	-0.06617634515720369	IZY	[0, 1, 2]
15	0.025838820409241972	IZZ	[0, 1, 2]
16	-0.005615963343537162	XII	[0, 1, 2]
17	-0.18582269635551796	XIX	[0, 1, 2]
18	-0.014995653778895798	XIY	[0, 1, 2]
19	-0.024342246603799002	XIZ	[0, 1, 2]
20	0.024853879327816464	XXI	[0, 1, 2]
21	-0.0003911670057600569	XXX	[0, 1, 2]
22	0.019376291486339785	XXY	[0, 1, 2]
23	0.039944057923917003	XXZ	[0, 1, 2]
24	0.002058815192277178	XYI	[0, 1, 2]
25	-0.049992530640230466	XYX	[0, 1, 2]
26	-0.022494667621329505	XYY	[0, 1, 2]
27	0.07065426613789572	XYZ	[0, 1, 2]
28	0.01241301140099471	XZI	[0, 1, 2]
29	-0.028120665143850098	XZX	[0, 1, 2]
30	-0.06372069429523346	XZY	[0, 1, 2]
31	0.022495267000582728	XZZ	[0, 1, 2]
32	-0.04441440856772076	YII	[0, 1, 2]
33	-0.014995653778895923	YIX	[0, 1, 2]
34	-0.032691597279734835	YIY	[0, 1, 2]
35	-0.01591480891702651	YIZ	[0, 1, 2]
36	0.029646274839496514	YXI	[0, 1, 2]
37	0.019376291486339778	YXX	[0, 1, 2]
38	0.20402965738563522	YXY	[0, 1, 2]
39	-0.04823614667152235	YXZ	[0, 1, 2]
40	-6.011165411385407e-05	YYI	[0, 1, 2]
41	-0.02249466762132951	YYX	[0, 1, 2]
42	0.014068547954461415	YYY	[0, 1, 2]
43	0.004672737470102694	YYZ	[0, 1, 2]
44	-0.06617634515720376	YZI	[0, 1, 2]
45	-0.0637206942952335	YZX	[0, 1, 2]
46	0.009606830354296339	YZY	[0, 1, 2]
47	-0.008961855599010384	YZZ	[0, 1, 2]
48	-0.04249275519392251	ZII	[0, 1, 2]
49	-0.024342246603799116	ZIX	[0, 1, 2]
50	-0.015914808917026586	ZIY	[0, 1, 2]
51	-0.03148570636474737	ZIZ	[0, 1, 2]
52	-0.02989659338733319	ZXI	[0, 1, 2]
53	0.039944057923917003	ZXX	[0, 1, 2]
54	-0.048236146671522415	ZXY	[0, 1, 2]
55	-0.19655461684948114	ZXZ	[0, 1, 2]
56	-0.06548679398263389	ZYI	[0, 1, 2]
57	0.07065426613789574	ZYX	[0, 1, 2]
58	0.004672737470102616	ZYY	[0, 1, 2]
59	-0.011361350911100621	ZYZ	[0, 1, 2]
60	0.025838820409241986	ZZI	[0, 1, 2]
61	0.02249526700058272	ZZX	[0, 1, 2]
62	-0.008961855599010395	ZZY	[0, 1, 2]
63	-0.022548201337122944	ZZZ	[0, 1, 2]

***Table T04.3**: Example of the content of a _pauli.csv file (nqubits_04_depth_3_pauli.csv)*

A.1.3. PH package

In the **PH** folder several modules that implement several functionalities, like computing parent Hamiltonian decomposition, implementing ansatzes as myQLM circuits or executing the **VQE** quantum step, are located.

Following is a quick summary of the most important Python modules:

- *ansatzes*: this module contains Eviden myQLM implementation of different ansatzes. Additionally, the functions (and classes) for simulating them using Eviden myQLM solvers are coded here.
- *parent_hamiltonian*: this module contains the **PH** class that allows computing a parent Hamiltonian using the naive of the local methods (see sections 2.2.1 and 2.2.2 respectively).
- *execution_ph*: this module contains the mandatory functions and classes for, given an ansatz and its local parent Hamiltonian Pauli decomposition, executing the **VQE** quantum step (blocks *2a*, *2b* and *2c* of Figure T04.1) and getting its corresponding ground state energy.
- *workflow*: this module uses the three previous ones for executing a complete workflow (this is the computation of the state of a desired ansatz, its local parent Hamiltonian and its ground state energy using quantum **VQE** step).
- *pauli*: this module has functions that deal with the decomposition of matrices in n-generalized Pauli matrices (see equations (T04.2.8) and (T04.2.11))
- *contractions*: this module contains several functions for computing contraction of indices and for computing

reduced density matrices mandatory for the parent Hamiltonian computations (see equations (T04.2.14) and (T04.2.15)).

- *utils*: this module contains several auxiliary functions needed for the rest of the modules of the package.

The other python files presented in the **PH** are for executing the different modules in an easy way (launch_ansatzes.py, launch_parent_hamiltonian.py, launch_execution_ph.py, launch_get_jobs.py and launch_workflow.py). And the *json* files are for configuring in an easy way these executions (ansatzes.json, parent_hamiltonian.json, execution_ph.json and workflow.json).

Finally, a folder called **notebook** is presented in the **PH** one. This folder contains several *jupyter notebooks* that explain how to use the different modules of the **PH** package.

A.2. Generation of the benchmark report

Following deliverable **D3.5: The NEASQC Benchmark Suite** the results of a complete **Benchmark** must be reported in a separate JSON file that must satisfy the **NEASQC** JSON schema *NEASQC.Benchmark.V2.Schema.json* provided into the aforementioned deliverable. For automating this process the following files should be modified, as explained in Annex B of the deliverable **D3.5: The NEASQC Benchmark Suite**:

- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **neasqc_benchmark.py**

my_environment_info.py

This script has the functions for gathering information about the hardware where the **Benchmark** is executed.

Listing T04.6 shows an example of the **my_environment_info.py** script. Here the compiled information corresponds to a classic computer because the case was simulated instead of executed in a quantum computer.

```

import platform
import psutil
from collections import OrderedDict

def my_organisation(**kwargs):
    """
    Given information about the organisation how uploads the benchmark
    """
    name = "CESGA"
    return name

def my_machine_name(**kwargs):
    """
    Name of the machine where the benchmark was performed
    """
    machine_name = platform.node()
    return machine_name

def my_qpu_model(**kwargs):
    """
    Name of the model of the QPU
    """
    qpu_model = "None"
    return qpu_model

def my_qpu(**kwargs):
    """
    Complete info about the used QPU
    """
    #Basic schema
    #QPUDescription = {
    #    "NumberOfQPUs": 1,
    #    "QPUs": [
    #        {
    #            "BasicGates": ["none", "none1"],
    #            "Qubits": [
    #                {
    #                    "QubitNumber": 0,
    #                    "T1": 1.0,
    #                    "T2": 1.00
    #                }
    #            ],
    #            "Gates": [
    #                {
    #                    "Gate": "none",
    #                    "Type": "Single",
    #                    "Symmetric": False,
    #                    "Qubits": [0],
    #                    "MaxTime": 1.0
    #                }
    #            ],
    #            "Technology": "other"
    #        },
    #    ]
    #}

    #Defining the Qubits of the QPU
    qubits = OrderedDict()
    qubits["QubitNumber"] = 0
    qubits["T1"] = 1.0
    qubits["T2"] = 1.0

    #Defining the Gates of the QPU
    gates = OrderedDict()
    gates["Gate"] = "none"
    gates["Type"] = "Single"
    gates["Symmetric"] = False
    gates["Qubits"] = [0]
    gates["MaxTime"] = 1.0


    #Defining the Basic Gates of the QPU
    qpus = OrderedDict()
    qpus["BasicGates"] = ["none", "none1"]
    qpus["Qubits"] = [qubits]
    qpus["Gates"] = [gates]
    qpus["Technology"] = "other"

    qpu_description = OrderedDict()
    qpu_description['NumberOfQPUs'] = 1
    qpu_description['QPUs'] = [qpus]

    return qpu_description

def my_cpu_model(**kwargs):
    """
    model of the cpu used in the benchmark
    """
    cpu_model = platform.processor()
    return cpu_model

def my_frecuency(**kwargs):
    """
    Frcuency of the used CPU
    """
    #Use the nominal frequency. Here, it collects the maximum frequency
    #print(psutil.cpu_freq())
    cpu_frec = psutil.cpu_freq().max/1000
    return cpu_frec

def my_network(**kwargs):
    """
    Network connections if several QPUs are used
    """
    network = OrderedDict()
    network["Model"] = "None"
    network["Version"] = "None"
    network["Topology"] = "None"
    return network

def my_QPUCPUConnection(**kwargs):
    """
    Connection between the QPU and the CPU used in the benchmark
    """
    #
    # Provide the information about how the QPU is connected to the CPU
    #
    qpuccpu_conn = OrderedDict()
    qpuccpu_conn["Type"] = "memory"
    qpuccpu_conn["Version"] = "None"
    return qpuccpu_conn
```

Listing T04.6: Example of configuration of the ***my_environment_info.py*** *script*

In general, it is expected that for each computer used (quantum or classic), the **Benchmark** developer should change this script to properly get the hardware info.

A.2.1. my_benchmark_info.py

This script gathers the information under the field *Benchmarks* of the benchmark report. Information about the software, the compilers and the results obtained from an execution of the **Benchmark** is stored in this field.

Listing T04.7 shows an example of the configuration of the **my_benchmark_info.py** script for gathering the aforementioned information.

```

import sys
import platform
from collections import OrderedDict
from my_benchmark_summary import summarize_results
import pandas as pd


def my_benchmark_kernel(**kwargs):
    """
    Name for the benchmark Kernel
    """
    return "ParentHamiltonian"

def my_starttime(**kwargs):
    """
    Providing the start time of the benchmark
    """
    times_filename = kwargs.get("times_filename", None)
    pdf = pd.read_csv(times_filename, index_col=0)
    start_time = pdf["StartTime"][0]
    return start_time

def my_endtime(**kwargs):
    """
    Providing the end time of the benchmark
    """
    times_filename = kwargs.get("times_filename", None)
    pdf = pd.read_csv(times_filename, index_col=0)
    end_time = pdf["EndTime"][0]
    return end_time

def my_timemethod(**kwargs):
    """
    Providing the method for getting the times
    """
    time_method = "time.time"
    return time_method

def my_programlanguage(**kwargs):
    """
    Getting the programing language used for benchmark
    """
    program_language = platform.python_implementation()
    return program_language

def my_programlanguage_version(**kwargs):
    """
    Getting the version of the programing language used for benchmark
    """
    language_version = platform.python_version()
    return language_version

def my_programlanguage_vendor(**kwargs):
    """
    Getting the version of the programing language used for benchmark
    """
    language_vendor = "None"
    return language_vendor

def my_api(**kwargs):
    """
    Collect the information about the used APIs
    """
    # api = OrderedDict()
    # api["Name"] = "None"
    # api["Version"] = "None"
    # list_of_apis = [api]
    modules = []
    list_of_apis = []
    for module in list(sys.modules):
        api = OrderedDict()
        module = module.split('.')[0]
        if module not in modules:
            modules.append(module)
            api["Name"] = module
            try:
                version = sys.modules[module].__version__
            except AttributeError:
                #print("NO VERSION: "+str(sys.modules[module]))
                try:
                    if  isinstance(sys.modules[module].version, str):
                        version = sys.modules[module].version
                        #print("\t Attribute Version"+version)
                    else:
                        version = sys.modules[module].version()
                        #print("\t Methdod Version"+version)
                except (AttributeError, TypeError) as error:
                    #print('\t NO VERSION: '+str(sys.modules[module]))
                    try:
                        version = sys.modules[module].VERSION
                    except AttributeError:
                        #print('\t\t NO VERSION: '+str(sys.modules[module]))
                        version = "Unknown"
            api["Version"] = str(version)
            list_of_apis.append(api)
    return list_of_apis

def my_quantum_compilation(**kwargs):
    """
    Information about the quantum compilation part of the benchmark
    """
    q_compilation = OrderedDict()
    q_compilation["Step"] = "None"
    q_compilation["Version"] = "None"
    q_compilation["Flags"] = "None"
    return [q_compilation]

def my_classical_compilation(**kwargs):
    """
    Information about the classical compilation part of the benchmark
    """
    c_compilation = OrderedDict()
    c_compilation["Step"] = "None"
```

```python
    c_compilation["Version"] = "None"
    c_compilation["Flags"] = "None"
    return [c_compilation]

def my_metadata_info(**kwargs):
    """
    Other important info user want to store in the final json.
    """
    metadata = OrderedDict()
    return metadata
```

Listing T04.7: Example of configuration of the **my_benchmark_info.py** script

The *my_benchmark_info* function gathers all the mandatory information needed by the *Benchmarks* main field of the report (by calling the different functions listed in listing T04.7). In order to properly fills this field some mandatory information must be provided as the typical *python kwargs*:

- *times_filename*: This is the complete path to the file where the starting and ending time of the benchmark was stored. This file must be a *csv* one and it is generated when the **KERNEL_BENCHMARK** class is executed. This information is used by the *my_starttime* and *my_endtime* functions.
- *benchmark_file*: complete path where the file with the summary results of the benchmark are stored. This information is used by the *summarize_results* function from *my_benchmark_summary.py* script (see section A.2.2).

A.2.2. my_benchmark_summary.py

In this script, the *summarize_results* function is implemented. This function formats the results of a complete execution of the **PH Benchmark** with a suitable **NEASQC** benchmark report format. It can be used for generating the information under the sub-field *Results* of the main field *Benchmarks* in the report.

Listing T04.8 shows an example of implementation of *summarize_results* function for the **PH Benchmark** procedure.

```python
def summarize_results(**kwargs):
    """
    Mandatory code for properly present the benchmark results following
    the NEASQC jsonschema
    """

    # n_qbits = [4]
    # #Info with the benchmark results like a csv or a DataFrame
    # pdf = None
    # #Metrics needed for reporting. Depend on the benchmark kernel
    # list_of_metrics = ["MRSE"]

    import pandas as pd
    benchmark_file = kwargs.get("benchmark_file", None)
    index_columns = [0, 1, 2, 3, 4, 5]
    pdf = pd.read_csv(benchmark_file, header=[0, 1], index_col=index_columns)
    pdf.reset_index(inplace=True)
    n_qbits = list(set(pdf["nqubits"]))
    depth = list(set(pdf["depth"]))
    list_of_metrics = ["gse"]

    results = []
    #If several qbits are tested
    # For ordering by n_qbits
    for n_ in n_qbits:
        for depth_ in depth:
            # For ordering by auxiliar qbits
            result = OrderedDict()
            result["NumberOfQubits"] = n_
            result["QubitPlacement"] = list(range(n_))
            result["QPUs"] = [2]
            result["CPUs"] = psutil.Process().cpu_affinity()
            #Select the proper data
            indice = (pdf["nqubits"] == n_) & (pdf["depth"] == depth_)
            step_pdf = pdf[indice]
            result["TotalTime"] = step_pdf["elapsed_time"]["mean"].iloc[0]
            result["SigmaTotalTime"] = step_pdf["elapsed_time"]["std"].iloc[0]
            result["QuantumTime"] = step_pdf["quantum_time"]["mean"].iloc[0]
            result["SigmaQuantumTime"] = step_pdf["quantum_time"]["std"].iloc[0]
            result["ClassicalTime"] = step_pdf["classic_time"]["mean"].iloc[0]
            result["SigmaClassicalTime"] = step_pdf["classic_time"]["std"].iloc[0]

            # For identifying the test
            result["AnsatzDepth"] = depth_
            result["Shots"] = int(step_pdf["nb_shots"].iloc[0])
            result["Truncation"] = int(step_pdf["nb_shots"].iloc[0])
            metrics = []
            #For each fixed number of qbits several metrics can be reported
            for metric_name in list_of_metrics:
                metric = OrderedDict()
                #MANDATORY
                metric["Metric"] = metric_name
                metric["Value"] = step_pdf[metric_name]["mean"].iloc[0]
                metric["STD"] = step_pdf[metric_name]["std"].iloc[0]
                metric["COUNT"] = int(step_pdf[metric_name]["count"].iloc[0])
                metrics.append(metric)
            result["Metrics"] = metrics
            results.append(result)
    return results
```

*Listing T04.8: Example of configuration of the summarize_results function for **PH** benchmark*

As usual, the *kwargs* strategy is used for passing the arguments that the function can use. In this case, the only mandatory argument is *benchmark_file* with the path to the file where the summary results of the **Benchmark** execution were stored.

Table T04.4 shows the sub-fields and the information stored, under the *Results* field. To have proper traceability of the executions the sub-fields *AnsatzDepth*, *Truncation* and *Shots* were created explicitly for the **PH Benchmark**.

sub-field	information
NumberOfQubits	number of qubits, n
TotalTime	mean of **elapsed time**
SigmaTotalTime	standard deviation of **elapsed time**
QuantumTime	mean of the **quantum time**
SigmaQuantumTime	standard deviation of **quantum time**
ClassicalTime	mean of the **classical time**
SigmaClassicalTime	standard deviation of **classical time**
Shots	number of shots
Metrics	sub-field with the obtained metrics
AnsatzDepth	number of layers of the ansatz
Truncation	for specifying the truncation of the Pauli coefficients (None)

***Table T04.4**: Sub-fields of the Results fields of the **TNBS** benchmark report. The metrics related with the **quantum time** and **classical time** are not mandatory*

The sub-field *Metrics* gathers information about the obtained metrics of the benchmark. Table T04.5 shows its different sub-fields and the information stored.

sub-field	information
metric	**Ground State Energy**
Value	mean value of the metric
STD	standard deviation of the metric
Count	number of samples for computing the statistics of the metric

***Table T04.5**: Sub-fields of the Metrics field.*

A.2.3. neasqc_benchmark.py

The *neasqc_benchmark.py* script can be used straightforwardly for gathering all the **Benchmark** execution information and results, for creating the final mandatory **NEASQC** benchmark report.

It does not necessarily change anything about the class implementation. It is enough to update the information of the *kwargs* arguments for providing the mandatory files for gathering all the information.

In this case, the following information should be provided as arguments for the *exe* method of the **BENCHMARK** class:

- *times_filename*: complete path where the file with the times of the **Benchmark** execution was stored.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark** execution was stored.