Centro de Investigación en Tecnologías de la Información y Comunicaciones
Universidade da Coruña (UDC)

Implementation of the Benchmark Cases of the Suite

Lista de autores

Oluwatosin Esther Odubanjo
Diego Andrade Canosa
Juan Touriño Domínguez
María José Martín Santamaría
Basilio Fraguela Rodríguez

01/09/2023

Unha maneira De facer Europa

Fondos Europeos

Despregamento dunha infraestructura baseada en tecnoloxías cuánticas da información que permita impulsar a I+D+i en Galicia.

Apoiar a transición cara a una economía dixital.

Operación financiada pola Unión Europea, a través do FONDO EUROPEO DE DESENVOLVEMENTO REXIONAL (FEDER) como parte da resposta da Unión á pandemia da COVID-19

CESGA
GALICIA SUPERCOMPUTING CENTER

Baixo a licenca [CC-BY-SA]

DATA	AUTHOR	CHANGES	VERSION
18 / 07 / 2023	Diego Andrade Canosa	Initial version and structure	0.1
24 / 07 / 2023	Oluwatosin Esther Odubanjo	First complete draft	0.5
25 / 07 / 2023	Oluwatosin Esther Odubanjo	Final version	1.0
26 / 07 / 2023	Oluwatosin Esther Odubanjo	Final version (CESGA specification format)	1.1

CESGA GALICIA SUPERCOMPUTING CENTER

Table of contents

1 Introduction **7**

2 Benchmark Test Case Implementation for Probability Loading (PL) Algorithms **8**

2.1 Implementation of the PL Benchmark Test Case 8

2.1.1 The QQuantLib Library for Probability Loading 8

2.1.2 load_probabilities.py 9

2.2 Execution of the Complete PL Benchmark Procedure 13

2.3 Generation of the PL Benchmark Report 18

2.3.1 my_environment_info.py 18

2.3.2 my_benchmark_info.py 21

2.3.3 my_benchmark_summary.py 25

2.3.4 benchmark.py 27

3 Benchmark Test Case Implementation for Amplitude Estimation (AE) Algorithm **28**

3.1 Implementation of the AE Benchmark Test Case 28

3.1.1 The QQuantLib Library for Computing Integrals 29

3.1.2 ae_sine_integral.py 36

3.2 Execution of the Complete AE Benchmark Procedure 39

3.3 Generation of the AE Benchmark Report 45

3.3.1 my_environment_info.py 45

3.3.2 my_benchmark_info.py 48

3.3.3 my_benchmark_summary.py 52

3.3.4 benchmark.py 54

4 Benchmark Test Case Implementation for Quantum Phase Estimation (QPE) Algorithms **55**

4.1 Implementation of the QPE Benchmark Test Case 56

4.1.1 The rz_library for Computing Eigenvalues 56

4.1.2 qpe_rz.py: the *QPE_RZ* class 56

4.2 Execution of the Complete QPE Benchmark Procedure 76

4.3 Generation of the QPE Benchmark Report 83

4.3.1 my_environment_info.py 84

4.3.2 my_benchmark_info.py 87

4.3.3 my_benchmark_summary.py 90

4.3.4 benchmark.py 92

CESGA
GALICIA SUPERCOMPUTING CENTER

List of figures

1 Circuit implementation for brute_force method of the load_probability function. Line 7 of listing 1 . . . 9
2 Circuit implementation for for multiplexor method of the load probability function. Line 11 of listing 1 . . . 9
3 Example of the pdf attribute from the *LoadProbabilityDensity* . . . 13
4 Circuit implementation for the *class_encoding.oracle* from listing 11 . . . 30
5 Canonical Amplitude Estimation using Quantum Phase Estimation . . . 32
6 Circuit implementation of the $R_z^n(\vec{\theta})$ operator . . . 65
7 Canonical **QPE** circuit. . . . 66

List of tables

CESGA GALICIA SUPERCOMPUTING CENTER

List of acronyms

PL	*Probability Loading*
QQuantLib	*Quantum Quantitative Finance Library*
PDF	*Probability Density Function*
AE	*Amplitude Estimation*
MLAE	*Maximum Likelihood Amplitude Estimation*
IQAE	*Iterative Quantum Phase Estimation Amplitude Estimation*
RQAE	*Real Quantum Amplitude Estimation*
CQPEAE	*Classical Quantum Phase Estimation Amplitude Estimation*
IQPAE	*Amplitude Estimation*
MCAE	*Monte Carlo Amplitude Estimation*
QLM	*Quantum Learning Machine*
NumPy	*Numerical Python*
JSON	*JavaScript Object Notation*
CSV	*Comma-Separated Values*
QPE	*Quantum Phase Estimation*
KS	*Kolmogorov-Smirnov*

1 Introduction

The Quantum Computing (QC) Benchmark Suite defined within the framework of this contract comprises a set of benchmark cases selected from several domains where Quantum Computing (QC) is being explored for its exponential computing advantage.

Each proposed **Benchmark Case** has four components: the selection criteria, the kernel definition, the test case, and the benchmark execution procedure. And must be complemented by a complete QLM-compatible software implementation and a complete result report into a separate JSON file. This document describes the QLM-reference implementation, the execution procedure as well as the benchmark report format of the following **Benchmark Test Cases:** Probability Loading, PL, Amplitude Estimation, AE, and Quantum Phase Estimation, QPE.

The QLM-reference implementation of the Probability Loading and Amplitude Estimation **Benchmark Test Cases** are based on the Quantum Quantitative Finance Library (QQuantLib), which allows for the implementation of QLM-compatible Probability Loading algorithms and several Amplitude Estimation algorithms used in quantum finances ([3], [6], [9]) as well as the computation of integrals and the expectation value of functions, while that of the Quantum Phase Estimation **Benchmark Test Case** is based on the **rz_library** which allows for the computation of the eigenvalue of a n qubits Z-axis rotation unitary operator.

Attached to this documentation is a file - *sourceCodeCases.zip*, which contains all necessary scripts and templates for the **Implementation of the Benchmark Cases of the Suite**. Subsequently, sections 2, 3, and 4, discuss the implementation, execution and report generation of the PL, AE, and QPE **Benchmark Test Cases**.

2 Benchmark Test Case Implementation for Probability Loading (PL) Algorithms

The reference implementation for the **Benchmark Test Case** of the **Probability Loading kernel** can be found in the **sourceCodeCases/01-Probability-Loading** folder of the attached file.

This folder contains the following folder and files for implementing and executing the **PL kernel Benchmark Test Case** as well as for generating the benchmark report:

- **QQuantLib** folder: with a complete QQuantLib library.
- **load_probabilities.py**
- **my_benchmark_execution.py**
- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **benchmark.py**

The folder, **QQuantLib**, and the file, **load_probabilities.py** handle the **PL Benchmark Test Case** implementation, and the file, **my_benchmark_execution.py** addresses the **PL Benchmark Test Case** execution. While the files, **my_environment_info.py**, **my_benchmark_info.py**, **my_benchmark_summary.py** and **benchmark.py** are involved with the benchmark report generation.

In the following sections 2.1, 2.2 and 2.3, we give a detailed description of the actions of the files involved with the implementation and execution of the **Benchmark Test Case** as well as the generation of the benchmark report respectively.

2.1 Implementation of the PL Benchmark Test Case

The problem for the **Benchmark Test Case** of the **PL kernel** is the loading of a fixed normal distribution, PDF, $N_{\mu,\sigma}(x)$ into a quantum circuit. Subsequently, subsection 2.1.1 explains how to use the *probability loading* algorithms from **QQuantLib** library while subsection 2.1.2 explains the load_probabilities.py script where the Benchmark Test Case workflow can be configured and executed.

2.1.1 The QQuantLib Library for Probability Loading

The *load_probability* function from the **DL.data_loading** package of the **QQuantLib** allows to create, from a normalised input numpy array, a **QLM AbstractGate** for encoding it into a quantum circuit. The implementation is based on the algorithm from [5].

Two inputs should be provided to the load probability function:

- *probability_array*: a numpy array with the desired probability.
- *method*: a string for indicating the loading method to use:
  - *brute-force*: in this case the controlled rotation by state needed by the original algorithm ([5]) were implemented in an straightforward way. This implementation results in a long circuit having a lot of redundant operations.

- *multiplexors*: the mandatory controlled rotation by state were implemented using quantum multiplexors ([10]). This implementation is more efficient and compact.

Listing 1 shows an example for loading an input normalised array into a quantum circuit using the load probability function. In the example, the aforementioned loading methods are used. The circuit implementations for a loading probability operator, using these methods, are presented in figures 1 and 2, for the *brute force* and *multiplexors* methods respectively. As can be seen last method creates more compact circuits.

```
from QQuantLib.DL.data_loading import load_probability
f_normalised = np.array(
[0., 0.14285714, 0.28571429, 0.42857143, 0.57142857, 0.71428571,
   0.85714286, 1.])
#Method: brute_force
routine = load_probability(probability, method="brute_force")
#Display circuit representation for brute_force
%qatdisplay routine --depth --svg
#Method: multiplexors
routine = load_probability(probability, method="multiplexor")
#Display circuit representation for multiplexors
%qatdisplay routine --depth --svg
```

Listing 1: Loading a probability distribution using the **load_probability** function from **QQuantLib.** Both loading methods-*brute_force* and *multiplexors*, were used for the operator creation.

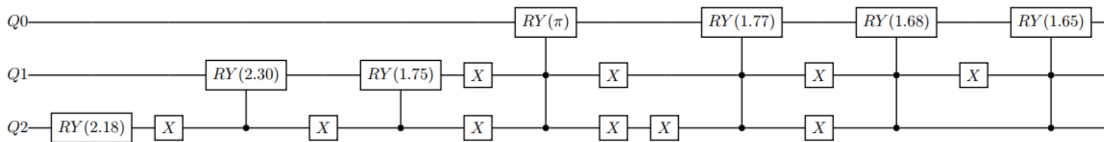

Figure 1: Circuit implementation for brute_force method of the load_probability function. Line 7 of listing 1

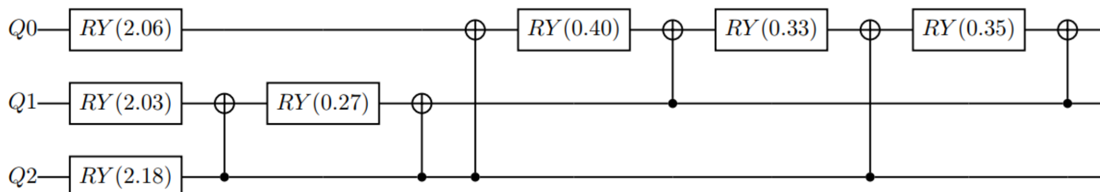

Figure 2: Circuit implementation for for multiplexor method of the load probability function. Line 11 of listing 1

2.1.2 load_probabilities.py

In this script the **Benchmark Test Case** of the **PL kernel**, is implemented as a python class called *LoadProbabilityDensity*. Listing 2 shows the complete class implementation.

```
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:
            print("Not QPU was provide. Default QPU will be used")
            self.qpu = "default"
        #get qpu object
        self.linalg_qpu = get_qpu(self.qpu)
        print(self.linalg_qpu)

        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
```

```
49         self.chi2 = None
50         self.pvalue = None
51         self.pdf = None
52         self.observed_frecuency = None
53         self.expeted_frecuency = None
54 
55     def loading_probability(self):
56         """
57         executing quantum stuff
58         """
59         self.result, self.circuit, self.quantum_time =
   loading_probability(
60             self.data, self.load_method, self.shots, self.linalg_qpu)
61 
62     def get_probabilities(self):
63         """
64         Computing probability densitiy array
65         """
66         self.x_, self.data, self.mean, self.sigma, \
67             self.step, self.shots, self.dist = get_probabilities(self.
   n_qbits)
68 
69     def get_metrics(self):
70         """
71         Computing Metrics
72         """
73         #Kolmogorov-Smirnov
74         self.ks = np.abs(
75             self.result["Probability"].cumsum() - self.data.cumsum()
76         ).max()
77         #Kullback-Leibler divergence
78         epsilon = self.data.min() * 1.0e-5
79         self.kl = entropy(
80             self.data,
81             np.maximum(epsilon, self.result["Probability"])
82         )
83 
84         #Chi square
85         self.observed_frecuency = np.round(
86             self.result["Probability"] * self.shots, decimals=0)
87         self.expeted_frecuency = np.round(
88             self.data * self.shots, decimals=0)
89         try:
90             self.chi2, self.pvalue = chisquare(
91                 f_obs=self.observed_frecuency,
92                 f_exp=self.expeted_frecuency
93             )
94         except ValueError:
95             self.chi2 = np.sum(
```

```
96                    (self.observed_frecuency - self.expeted_frecuency) **2
   / \
97                        self.expeted_frecuency
98                )
99                count = len(self.observed_frecuency)
100               self.pvalue = chi2.sf(self.chi2, count -1)
101
102       def exe(self):
103           """
104           Execution of workflow
105           """
106           #Create the distribution for loading
107           tick = time.time()
108           self.get_probabilities()
109           #Execute the quantum program
110           self.loading_probability()
111           self.get_metrics()
112           tack = time.time()
113           self.elapsed_time = tack - tick
114           self.summary()
115
116       def summary(self):
117           """
118           Pandas summary
119           """
120           self.pdf = pd.DataFrame()
121           self.pdf["n_qbits"] = [self.n_qbits]
122           self.pdf["load_method"] = [self.load_method]
123           self.pdf["qpu"] = [self.linalg_qpu]
124           self.pdf["mean"] = [self.mean]
125           self.pdf["sigma"] = [self.sigma]
126           self.pdf["step"] = [self.step]
127           self.pdf["shots"] = [self.shots]
128           self.pdf["KS"] = [self.ks]
129           self.pdf["KL"] = [self.kl]
130           self.pdf["chi2"] = [self.chi2]
131           self.pdf["p_value"] = [self.pvalue]
132           self.pdf["elapsed_time"] = [self.elapsed_time]
133           self.pdf["quantum_time"] = [self.quantum_time]
```

Listing 2: *LoadProbabilityDensity* class from **load_probabilities.py** script.

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

- **n-qubits**: 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 load probability function from the **DL.data_loading** with **direct** implementation see sub section 2.1.1.

- *multiplexor*: for using load_probability function from the **DL.data_loading** with **multiplexors** implementation, see sub section 2.1.1.
- *KPTree*: QLM implementation of a probability loading [1]

The *exe* method executes a complete **Benchmark Test Case** and fills different attributes of the class. Most important is the *pdf* attribute that is a pandas DataFrame where all the configuration and the correspondent metrics results are stored. In the table in figure 3, an example of this attribute is shown. The mean and the sigma columns are the mean and the standard deviation of normal distribution used while the step column corresponds to Δx, which represents the array of values used.

	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

Figure 3: 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 in order to properly configure the **PL** algorithm and the **Benchmark Test Case**.

Listing 3 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** (Probability Density Function) 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 3: Example of use the **load_probabilities.py** script from command line

2.2 Execution of the Complete PL Benchmark Procedure

This section explains how to execute a complete benchmark procedure, using the script **my_benchmark _execution.py**.

The **my_benchmark_execution.py** script is a modification of the corresponding template script, *my_benchmark_execution.py*, located in sourceCodeCases/templates folder of the attached file. The functions, **run_code**, **compute_samples** and **summarize_results** were modified; however, the
KERNEL_BENCHMARK class was not modified. The software adaptions for the **PL kernel** is presented in the subsequent subsections.

run_code

Listing 4 shows the modifications of the **run_code** function for the **Benchmark Test Case of the PL kernel**. Its main functionality is executing the **Benchmark Test Case** for a fixed number of qubits (*n_qbits*), an input number of times (*repetitions*), and gathering all the mandatory metrics obtained. As can be seen, the *LoadProbabilityDensity* class, listing 2, and its *exe* method is used for doing the different executions of the **Benchmark Test Case**.

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

[1] See myQLM KPTree class

```

    Parameters
    ----------

    n_qbits : int
        number of qubits used for domain discretization
    repetitions : list
        number of repetitions for the integral
    kwargs : keyword arguments
        for configuration of the benchmark kernel

    Returns
    -------

    metrics : pandas DataFrame
        DataFrame with the desired metrics obtained for the integral
   computation

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

    #Here the code for configuring and execute the benchmark kernel

    from load_probabilities import LoadProbabilityDensity
    kernel_configuration = deepcopy(kwargs.get("kernel_configuration",
   None))
    if kernel_configuration is None:
        raise ValueError("kernel_configuration can not be None")

    list_of_metrics = []
    kernel_configuration.update({"number_of_qbits": n_qbits})
    print(kernel_configuration)
    for i in range(repetitions[0]):
        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)

    return metrics
```

Listing 4: **run_code** function for **benchmark test case** of the **PL kernel**.

compute_samples

Listing 5 shows the implementation of the **compute_samples** function for the Benchmark Test Case of the **PL kernel**. The main objective of the function is to codify a strategy for computing the

number of times the **Benchmark Test Case** should be executed for getting some desired statistical significance.

```
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

    """

    #Configuration for sampling computations

    #Desired Error in the benchmark metrics
    relative_error = kwargs.get("relative_error", None)
    if relative_error is None:
        relative_error = 0.1
    #Desired Confidence level
    alpha = kwargs.get("alpha", None)
    if alpha is None:
        alpha = 0.05
    #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)

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

    from scipy.stats import norm
    #geting the metrics from pre-benchmark step
    metrics = kwargs.get("pre_metrics", None)

    #Compute mean and sd
    std_ = metrics.groupby("load_method").std()
    std_.reset_index(inplace=True)
```

```
46     mean_ = metrics.groupby("load_method").mean()
47     mean_.reset_index(inplace=True)
48     #Metrics
49     zalpha = norm.ppf(1-(alpha/2)) # 95% of confidence level
50     #columns = ["KS", "KL", "elapsed_time"]
51     columns = ["elapsed_time"]
52 
53     samples_ = (zalpha * std_[columns] / (relative_error * mean_[
       columns]))**2
54     samples_ = samples_.max(axis=1).astype(int)
55     samples_.name = "samples"
56 
57     #If user wants limit the number of samples
58     samples_.clip(upper=max_meas, lower=min_meas, inplace=True)
59     return list(samples_)
```

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

summarize_results

Listing 6 shows the implementation of the **summarize_results** function for the **Benchmark Test Case** of the **PL kernel**. The main objective of the function is post-processing the results of the complete **Benchmark Test Case** execution.

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.

```
1 def summarize_results(**kwargs):
2     """
3     Create summary with statistics
4     """
5 
6     folder = kwargs.get("saving_folder")
7     csv_results = folder + kwargs.get("csv_results")
8     #Code for summarize the benchamark results. Depending of the
9     #kernel of the benchmark
10 
11     pdf = pd.read_csv(csv_results, index_col=0, sep=";")
12     pdf["classic_time"] = pdf["elapsed_time"] - pdf["quantum_time"]
13     pdf = pdf[
14         ["n_qbits", "load_method", "KS", "KL", "chi2",
15         "p_value", "elapsed_time", "quantum_time", "classic_time"]
16     ]
17     results = pdf.groupby(["load_method", "n_qbits"]).agg(
18         ["mean", "std", "count"])
19 
20     return results
```

Listing 6: **summarize_results** function for summarizing the results from the **Benchmark Test Case** execution of the **PL kernel**.

KERNEL_BENCHMARK class

This python class defines the complete benchmark workflow and its *exe* method executes it properly by calling the correspondent functions (**run_code, compute_samples, summarize_results**). Each time the **Benchmark Test Case** is executed, the result is stored in a given CSV file.

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

Listing 7 shows an example for configuring an execution of a Benchmark Test Case: in this case the *multiplexor* method will be used for creating the loading operator.

```
1  if __name__ == "__main__":
2
3      kernel_configuration = {
4          "load_method" : "multiplexor",
5          "qpu" : "c", #python, qlmass, default
6      }
7      name = "PL_{}".format(kernel_configuration["load_method"])
8
9      benchmark_arguments = {
10         #Pre benchmark configuration
11         "pre_benchmark": True,
12         "pre_samples": [10],
13         "pre_save": True,
14         #Saving configuration
15         "saving_folder": "./Results/",
16         "benchmark_times": "{}_times_benchmark.csv".format(name),
17         "csv_results": "{}_benchmark.csv".format(name),
18         "summary_results": "{}_SummaryResults.csv".format(name),
19         #Computing Repetitions configuration
20         "relative_error": None,
21         "alpha": None,
22         "min_meas": None,
23         "max_meas": None,
24         #List number of qubits tested
25         "list_of_qbits": [4, 6, 8],
26     }
27
28     #Configuration for the benchmark kernel
29     benchmark_arguments.update({"kernel_configuration":
       kernel_configuration})
30     ae_bench = KERNEL_BENCHMARK(**benchmark_arguments)
31     ae_bench.exe()
```

Listing 7: Example of configuration of a complete **Benchmark Test Case** execution. This part of the code should be located at the end of the **my_benchmark_execution.py** script.

Possible *kwargs* for the **KERNEL_BENCHMARK** class, *benchmark_arguments* dictionary in listing 7 are:

- *pre_benchmark*: For executing or not executing 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.
- *summary_results*: name for the *csv* file where the post-processed results (using the **summarize_results** function) will be stored.
- *list_of_qbits*: list with the different number of qubits for executing the complete **Benchmark Test Case**.

Other parameters like: *relative_error*, *alpha*, *min_meas* or *max_meas* that are used for the **compute_samples** can be provided too, but for a complete **Benchmark Test Case** execution must be fixed to *None*. As can be seen in listing 7, the configuration of the implementation of **PL** algorithm is passed in the *benchmark_arguments* arguments under the key *kernel_configuration*.

For executing the **Benchmark Test Case**, the following command should be used:

python my_benchmark_execution.py

2.3 Generation of the PL Benchmark Report

The results of a complete **PL Benchmark Test Case** should be reported in a separate JSON file-**Benchmark.V2.Schema_modified.json** provided in the sourceCodeCases/templates folder. For automating this process the following files should be modified:

- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **benchmark.py**

2.3.1 my_environment_info.py

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

Listing 8 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.

```
1 import platform
2 import psutil
3 from collections import OrderedDict
4
5 def my_organisation(**kwargs):
6     """
```

```
7       Given information about the organisation how uploads the benchmark
8       """
9       #name = "None"
10      name = "CESGA"
11      return name
12
13  def my_machine_name(**kwargs):
14      """
15      Name of the machine where the benchmark was performed
16      """
17      #machine_name = "None"
18      machine_name = platform.node()
19      return machine_name
20
21  def my_qpu_model(**kwargs):
22      """
23      Name of the model of the QPU
24      """
25      #qpu_model = "None"
26      qpu_model = "QLM"
27      return qpu_model
28
29  def my_qpu(**kwargs):
30      """
31      Complete info about the used QPU
32      """
33      #Basic schema
34      #QPUDescription = {
35      #     "NumberOfQPUs": 1,
36      #     "QPUs": [
37      #         {
38      #             "BasicGates": ["none", "none1"],
39      #             "Qubits": [
40      #                 {
41      #                     "QubitNumber": 0,
42      #                     "T1": 1.0,
43      #                     "T2": 1.00
44      #                 }
45      #             ],
46      #             "Gates": [
47      #                 {
48      #                     "Gate": "none",
49      #                     "Type": "Single",
50      #                     "Symmetric": False,
51      #                     "Qubits": [0],
52      #                     "MaxTime": 1.0
53      #                 }
54      #             ],
55      #             "Technology": "other"
```

```
56      #         },
57      #     ]
58      #}
59 
60      #Defining the Qubits of the QPU
61      qubits = OrderedDict()
62      qubits["QubitNumber"] = 0
63      qubits["T1"] = 1.0
64      qubits["T2"] = 1.0
65 
66      #Defining the Gates of the QPU
67      gates = OrderedDict()
68      gates["Gate"] = "none"
69      gates["Type"] = "Single"
70      gates["Symmetric"] = False
71      gates["Qubits"] = [0]
72      gates["MaxTime"] = 1.0
73 
74 
75      #Defining the Basic Gates of the QPU
76      qpus = OrderedDict()
77      qpus["BasicGates"] = ["none", "none1"]
78      qpus["Qubits"] = [qubits]
79      qpus["Gates"] = [gates]
80      qpus["Technology"] = "other"
81 
82      qpu_description = OrderedDict()
83      qpu_description['NumberOfQPUs'] = 1
84      qpu_description['QPUs'] = [qpus]
85 
86      return qpu_description
87 
88  def my_cpu_model(**kwargs):
89      """
90      model of the cpu used in the benchmark
91      """
92      #cpu_model = "None"
93      cpu_model = platform.processor()
94      return cpu_model
95 
96  def my_frecuency(**kwargs):
97      """
98      Frcuency of the used CPU
99      """
100     #Use the nominal frequency. Here, it collects the maximum frequency
101     #print(psutil.cpu_freq())
102     #cpu_frec = 0
103     cpu_frec = psutil.cpu_freq().max/1000
104     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 8: 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.

2.3.2 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 Test Case** is stored in this field.

Listing 9 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"
```

```
14
15 def my_starttime(**kwargs):
16     """
17     Providing the start time of the benchmark
18     """
19     #start_time = "2022-12-12T16:46:57.268509+01:00"
20     times_filename = kwargs.get("times_filename", None)
21     pdf = pd.read_csv(times_filename, index_col=0)
22     start_time = pdf["StartTime"][0]
23     return start_time
24
25 def my_endtime(**kwargs):
26     """
27     Providing the end time of the benchmark
28     """
29     #end_time = "2022-12-12T16:46:57.268509+01:00"
30     times_filename = kwargs.get("times_filename", None)
31     pdf = pd.read_csv(times_filename, index_col=0)
32     end_time = pdf["EndTime"][0]
33     return end_time
34
35 def my_timemethod(**kwargs):
36     """
37     Providing the method for getting the times
38     """
39     time_method = "time.time"
40     return time_method
41
42 def my_programlanguage(**kwargs):
43     """
44     Getting the programing language used for benchmark
45     """
46     program_language = platform.python_implementation()
47     return program_language
48
49 def my_programlanguage_version(**kwargs):
50     """
51     Getting the version of the programing language used for benchmark
52     """
53     language_version = platform.python_version()
54     return language_version
55
56 def my_programlanguage_vendor(**kwargs):
57     """
58     Getting the version of the programing language used for benchmark
59     """
60     language_vendor = "Unknow"
```

```python

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)
```

```
154     return benchmark
```

Listing 9: 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 9). In order 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 2.3.3) 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 used, but it is important to get good traceability of the benchmark results.

2.3.3 my_benchmark_summary.py

In this script, the *summarize_results* function is implemented. This function formats the results of a complete execution of a **Benchmark Test Case** of the **PL kernel** with the provided 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 10 shows an example of implementation of *summarize_results* function for the **PL** benchmark procedure.

```
1  from collections import OrderedDict
2  import psutil
3
4  def summarize_results(**kwargs):
5      """
6      Mandatory code for properly present the benchmark results following
7      the jsonschema
8      """
9
10     #n_qbits = [4]
11     #Info with the benchmark results like a csv or a DataFrame
12     #pdf = None
13     #Metrics needed for reporting. Depend on the benchmark kernel
14     #list_of_metrics = ["MRSE"]
15
16     import pandas as pd
17     benchmark_file = kwargs.get("benchmark_file", None)
18     pdf = pd.read_csv(benchmark_file, header=[0, 1], index_col=[0, 1])
19     pdf.reset_index(inplace=True)
20     n_qbits = list(set(pdf["n_qbits"]))
21     load_methods = list(set(pdf["load_method"]))
```

```
22     list_of_metrics = [
23         "KS", "KL",
24         "chi2", "p_value"
25     ]
26 
27     results = []
28     #In the Probability Loading benchmark several qubits can be tested
29     for n_ in n_qbits:
30         #For selecting the different loading method using in the
   benchmark
31         for method in load_methods:
32             #Fields for benchmark test of a fixed number of qubits
33             result = OrderedDict()
34             result["NumberOfQubits"] = n_
35             result["QubitPlacement"] = list(range(n_))
36             result["QPUs"] = [1]
37             result["CPUs"] = psutil.Process().cpu_affinity()
38 
39             #Select the proper data
40             step_pdf = pdf[(pdf["load_method"] == method) & (pdf["
   n_qbits"] == n_)]
41 
42             #result["TotalTime"] = 10.0
43             #result["SigmaTotalTime"] = 1.0
44             #result["QuantumTime"] = 9.0
45             #result["SigmaQuantumTime"] = 0.5
46             #result["ClassicalTime"] = 1.0
47             #result["SigmaClassicalTime"] = 0.1
48 
49             result["TotalTime"] = step_pdf["elapsed_time"]["mean"].iloc
   [0]
50             result["SigmaTotalTime"] = step_pdf["elapsed_time"]["std"].
   iloc[0]
51             result["QuantumTime"] = step_pdf["quantum_time"]["mean"].
   iloc[0]
52             result["SigmaQuantumTime"] = step_pdf["quantum_time"]["std"
   ].iloc[0]
53             result["ClassicalTime"] = step_pdf["classic_time"]["mean"].
   iloc[0]
54             result["SigmaClassicalTime"] = step_pdf["classic_time"]["
   std"].iloc[0]
55             #For identify the loading method used. Not mandaatory but
56             #useful for identify results
57             result["load_method"] = method
58 
59             metrics = []
60             #For each fixed number of qbits several metrics can be
   reported
61             for metric_name in list_of_metrics:
```

```
62                  metric = OrderedDict()
63                  #MANDATORY
64                  metric["Metric"] = metric_name
65                  #metric["Value"] = 0.1
66                  #metric["STD"] = 0.001
67                  metric["Value"] = step_pdf[metric_name]["mean"].iloc[0]
68                  metric["STD"] = step_pdf[metric_name]["std"].iloc[0]
69                  #Depending on the benchmark kernel
70                  metric["COUNT"] = int(step_pdf[metric_name]["count"].
    iloc[0])
71                  metrics.append(metric)
72              result["Metrics"] = metrics
73              results.append(result)
74      return results
```

Listing 10: 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 Test Case** execution were stored.

2.3.4 benchmark.py

The *benchmark.py* script can be used directly for gathering all the **Benchmark Test Case** execution information and results, for creating the final mandatory benchmark report.

It is not necessary to 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 Test Case** execution was stored.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark Test Case** execution was stored.

3 Benchmark Test Case Implementation for Amplitude Estimation (AE) Algorithm

The reference implementation for the **Benchmark Test Case** of the **Amplitude Estimation kernel** can be found in the **sourceCodeCases/02-Amplitude-Estimation** folder of the attached file.

This folder contains the following folder and files for implementing and executing the **AE kernel Benchmark Test Case** as well as for generating the benchmark report:

- **QQuantLib** folder: with a complete copy of the QQuantLib library.
- **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**
- **benchmark.py**

The folders, **QQuantLib** and **jsons** and the file, **ae_sine_integral.py** handles the **AE Benchmark Test Case** implementation. On the other hand, the file **my_benchmark_execution.py** addresses the **AE Benchmark Test Case** execution, while the files, **my_environment_info.py**, **my_benchmark_info.py**, **my_benchmark_summary.py** and **benchmark.py** are involved with the benchmark report generation.

Furthermore, in the directory, **sourceCodeCases/02-Amplitude-Estimation**, of the attached file is a notebook- **_integral_computation_ae.ipynb**, that demonstrates the implementation of the **AE Benchmark Test Case**.

In the following sections 3.1, 3.2 and 3.3, we give a detailed description of the actions of the files involved with the implementation and execution of the Benchmark Test Case as well as the generation of the benchmark report respectively.

3.1 Implementation of the AE Benchmark Test Case

The mathematical problem for the **Benchmark Test Case** of the **AE kernel** is the computation of an integral in a very well-defined interval by using an AE algorithm whose input is an operator **A**, given as:

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

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

This section presents a complete description of the implementation of **AE kernel Benchmark Test Case**. Subsection 3.1.1 explains how to use the **QQuantLib** library, for computing the different mandatory operators and integrals discussed in the Benchmark test case description; subsection 3.1.2 explains how the **ae_sine_integral.py** script is used for implementing the **Benchmark Test Case**.

3.1.1 The QQuantLib Library for Computing Integrals

This subsection explains the software implementation of the different parts of the **Benchmark Test Case**, using the **QQuantLib** library.

3.1.1.1 Operator A Software Implementation

Given a properly normalised input array $f_norm_{x_i}$, the first step is the construction of the operator, **A**:

$$\mathbf{A}(f_{x_i}) = (\mathbb{I} \otimes H^{\otimes n})\mathbf{U}_f(\mathbb{I} \otimes H^{\otimes n}) \tag{2}$$

Where: the operator $\boldsymbol{U}_f$ can be constructed as:

$$\mathbf{U}_f(|0\rangle \otimes |i\rangle_n) = (\mathbf{R}_y(2 * \phi_{x_i}) |0\rangle) \otimes |i\rangle n \tag{3}$$

Where $\phi_{xi} = \arccos(f_norm_{x_i})$.

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 (labeled 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** located in the folder- **sourceCodeCases/misc/notebooks** of the attached file.

In order to encode the function in a quantum circuit, the following arguments must be passed when the *Encoding* class is instantiated:

- *array_function*: NumPy array with the desired function to encode (this is $f_norm_{x_i}$).
- *array_probability*: NumPy array with the desired probability function to encode. In the benchmark case, this is the uniform distribution probability - the default distribution of the class, 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 QLM implementation (a **QLM** *QRoutine*) of the operator $\mathbf{A}(f_{x_i})$, that is stored in the *oracle* property of the class. Listing 11 shows the use of the *Encoding* class. Figure 4 shows the circuit implementation of the *class_encoding.oracle* (the figure is the output of line 10 of the listing 11).

```
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(
```

```
    array_function=norm_f_x,
    array_probability = None,
    encoding=2)
class_encoding.run()
#QLM circuit oracle implementation
oracle_circuit = class_encoding.oracle
%qatdisplay oracle_circuit --depth 1
```

Listing 11: Creation of the $\mathbf{A}^I(f_{x_i})$ operator, using the *Encoding* class from **QQuantLib**, given an input Numpy array *norm_f_x*

Following the guidelines of the **AE** algorithm (see subsection 3.2.5 of the Benchmark for Amplitude Estimation Algorithms documentation), the created $\mathbf{A}(f_{x_i})$ operator should 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:

$$\mathbf{P}[|\Psi_0\rangle] = | \ \langle\Psi_0|\Psi\rangle \ |^2 = \left| \langle\Psi_0| \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_norm_{x_i} |\Psi_0\rangle \right| = \left| \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_norm_{x_i} \right|^2 = \tilde{a} \tag{4}$$

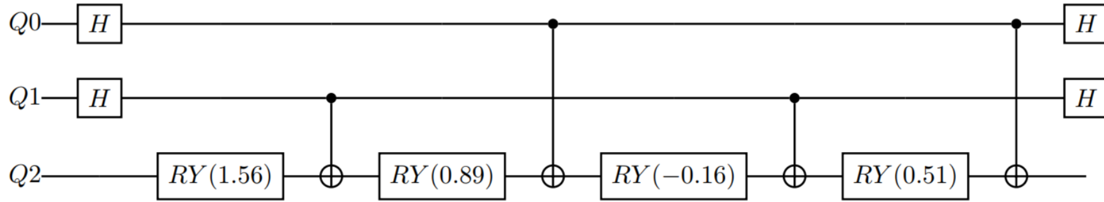

Figure 4: Circuit implementation for the *class_encoding.oracle* from listing 11

3.1.1.2 Amplitude Estimation Algorithm 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 5

$$\mathbf{G}(\mathbf{A}(f_{x_i})) = \mathbf{A}(f_{x_i})(\mathbb{I} - 2|0\rangle\langle 0|)\ \mathbf{A}(f_{x_i})^\dagger(\mathbb{I} - 2|\Psi_0\rangle\langle\Psi_0|) \tag{5}$$

This operator acts as shown in the equation 6:

$$\mathbf{G}^k(\mathbf{A})\mathbf{A}|0\rangle_n = \sin\left((2k+1)\theta\right)|\Psi_0\rangle + \cos\left((2k+1)\theta\right)|\Psi_1\rangle \tag{6}$$

The Grover function from the *QQuantLib.AA.amplitude_amplification* module belonging to the QQuantLib library allows to compute, given an input operator $\mathbf{A}(f_{x_i})$, the corresponding Grover-like one in a straightforward way as shown in listing 12, where a Grover-like operator is created from the *Encoding* object created in the listing 11.

```
from QQuantLib.AA.amplitude_amplification import grover
grover_oracle = grover(
    oracle=class_encoding2.oracle,
```

```
    target=class_encoding2.target,
    index=class_encoding2.index
)
```

Listing 12: Creation of the correspondent Grover-like operator from an created operator, $\mathbf{A}^I(f_{x_i})$ operator.

The notebook **02-Amplitude-Amplification-Operators.ipynb**, in the folder sourceCodeCases/misc/notebooks, 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 **Benchmark Test Case** for **AE kernel** described in this document is agnostic about these operators. The only mandatory input is the **A** operator.

- CQPEAE (*ae_classical_qpe module*) uses a classical phase estimation algorithm with QFT, see Figure 5, [1]. See notebook **sourceCodeCases/misc/notebooks/03-Maximum-Likelihood-Amplitude-Estimation-Class**
  for more information.
- IQPEAE (ae_iterative_quantum_pe module) uses an iterative implementation of QFT, using only one additional qubit, for classical phase estimation ([2], [7]). See notebook **sourceCodeCases /misc/notebooks/05-Iterative-Quantum-Phase-Estimation-Class.ipynb** for more information.
- MLAE (maximum_likelihood_ae module) uses a Maximum Likelihood algorithm [11]. See notebook **sourceCodeCases/misc/notebooks/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 $\mathbf{G}(\mathbf{A}(f_{x_i}))$ [4]. See notebook **sourceCodeCases/misc/notebooks/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, $\mathbf{G}(\mathbf{A}(f_{x_i}))$ [8]. See notebook **sourceCodeCases/misc/notebooks/07-Real-Quantum-Amplitude-Estimation-class.ipynb** for more information.
- MCAE (montecarlo_ae_module) uses the Monte Carlo algorithm. See notebook **sourceCodeCas es/misc/notebooks/08-AmplitudeEstimation-Class.ipynb** for more information.

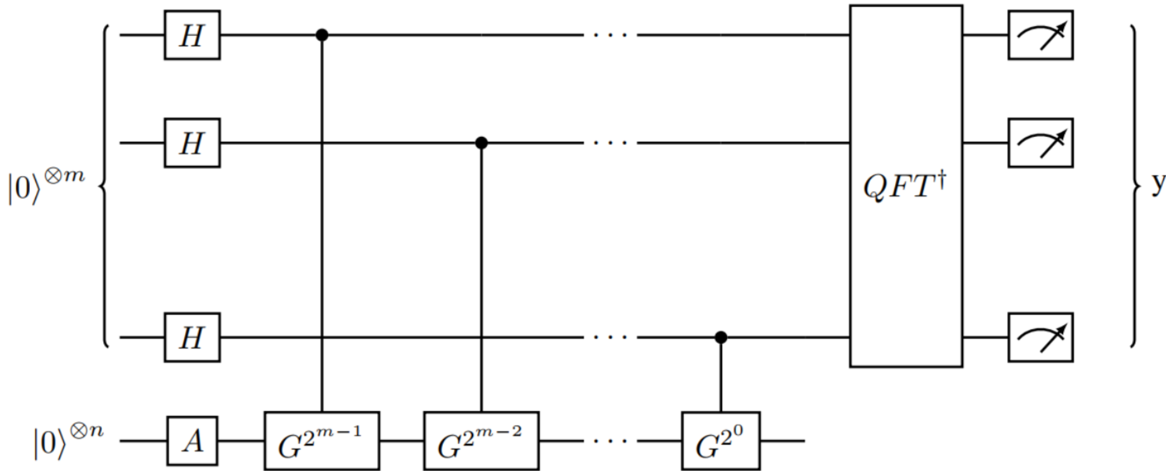

Figure 5: Canonical Amplitude Estimation using Quantum Phase Estimation

All of the algorithms in the bulleted list above use the *Grover* function explained in listing 12 to create the Grover-like operator. **QQuantLib** implements all the **AE** algorithms as Python classes that can be used in a similar way: 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 **sourceCodeCases/02-Amplitude-Estimation/jsons** folder of the attached file.

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 13 shows how to use this class. The Encoding object from listing 11, is used for providing the mandatory $\mathbf{A}(f_{x_i})$ operator to the class.

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

```
5       'mcz_qlm': False,
6   
7       #shots
8       'shots': 100,
9   
10      #MLAE
11      'schedule': [None],
12      'delta' : None,
13      'ns' : None,
14  
15      #CQPEAE
16      'auxiliar_qbits_number': 10,
17      #IQPEAE
18      'cbits_number': None,
19      #IQAE & RQAQE
20      'epsilon': None,
21      #IQAE
22      'alpha': None,
23      #RQAE
24      'gamma': None,
25      'q': None
26  }
27  ae_object = AE(
28      oracle=class_encoding.oracle,
29      target=class_encoding.target,
30      index=class_encoding.index,
31      ae_type='CQPEAE',
32      **ae_dict)
33  ae_object.run()
34  print(ae_object.ae_pdf)
35  
36  #Result
37  ae ae_l ae_u
38  0 0.375536 NaN NaN
```

Listing 13: Example of how to use the general *AE* class from *QQuantLib.AE* package. The Encoding class can ben used for providing the mandatory $\mathbf{A}(f_{x_i})$ operator in a transparent way

The notebook **sourceCodeCases/misc/notebooks/08-Amplitude Estimation-Class.ipynb** provides more information about this class.

3.1.1.3 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 1 can be rewritten as:

$$|\Psi\rangle = \mathbf{A}|0\rangle \otimes |0\rangle_n = \begin{cases} \sqrt{a}|\Psi_0\rangle + \sqrt{1-a}|\Psi_1\rangle & \text{NO RQAE} \\ a|\Psi_0\rangle + \beta|\Psi_1\rangle & \text{RQAE} \end{cases} \tag{7}$$

Only the $|\Psi\rangle = |0\rangle \otimes |0\rangle_n$ term must be taken into account, so using equation 7, the equation in 4, is now transformed into:

$$\tilde{a} = \begin{cases} \mathbf{P}_{\Psi_0} = \left| \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_norm_{x_i} \right|^2 & \text{NO RQAE} \\ \mathbf{Amplitude}_{\Psi_0} = \frac{1}{2^n} \sum_{i=0}^{2^n-1} f_norm_{x_i} & \text{RQAE} \end{cases} \tag{8}$$

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

$$\sum_{i=0}^{2^n-1} f_norm_{x_i} = \begin{cases} 2^n \sqrt{\tilde{a}} & \text{NO RQAE} \\ 2^n \tilde{a} & \text{RQAE} \end{cases} \tag{9}$$

In order to use the **QQuantLib** implemented algorithms for the **Benchmark Test Case** of the **AE kernel** equation 9 must be used for recovering the desired integral instead of equation of equation 4.

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. This function, computes the integral using equation 9, taking into account the **AE** algorithm used, and returning directly the integral of the input array function: $2^n\sqrt{\tilde{a}}$(or the $2^n\tilde{a}$ 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 **the Benchmark Test Case**, 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 one the **AE** algorithm).
- solver_ae: object based on the *AE* class.

Listing 14 shows how to use *q_solve_integral* for computing the integral of an input numpy array by using 2 different **AE** algorithms in a straightforward way.

```
1 from QQuantLib.finance.quantum_integration import q_solve_integral
2 norm_f_x = np.array([0.17106865, 0.49847362, 0.78295039, 0.99999999])
3 
4 #Configuration of AE algorithm
5 ae_dict = {
6     #QPU
7     'qpu': linalg_qpu,
8     #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 14: Using **q_solve_integral** for solving the integral of an input array using the encoding procedure 2 and two different **AE** algorithms.

Finally, equation 9 must be plugged into the Riemann sum:

$$S_{[a,b]} = \frac{b-a}{2^n}\sum_{i=0}^{2^n-1} f_{x_i} = \frac{b-a}{2^n}\sum_{i=0}^{2^n-1} \max(|f_{x_i}|) f_norm_{x_i} = \frac{\max(|f_{x_i}|)(b-a)}{2^n}\sum_{i=0}^{2^n-1} f_norm_{x_i} \tag{10}$$

for obtaining the desired integral as shown in equation 11

$$\tilde{S}_{[a,b]} = \begin{cases} \frac{\max(f_{x_i})(b-a)}{2^n}(2^n\sqrt{a}) & \text{NO RQAE} \\ \frac{\max(f_{x_i})(b-a)}{2^n}(2^n a) & \text{RQAE} \end{cases} \tag{11}$$

3.1.2 ae_sine_integral.py

In this script, the **Benchmark Test Case** for the **AE** kernel 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, computed as:
  - 0: $[0, \frac{3\pi}{8}]$
  - 1: $[\pi, \frac{5\pi}{4}]$
- ae_dictionary: python dictionary with the complete **AE** algorithm. configuration.

The function, *sine_integral*, returns a pandas DataFrame with the complete information of the executed **Benchmark Test Case** (this includes the configuration of the **AE** algorithm, the interval used, the qpu used and the obtained results). The complete code is shown in listing 15.

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

    Parameters
    ----------

    n_qbits : int
        for discretization of the input domain in 2^n 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
```

```
17
18    Return
19    ----------
20
21    pdf : pandas DataFrame
22        DataFrame with the complete information of the benchmark
23    """
24
25    #Local copy for AE configuration dictionary
26    ae_dictionary_ = deepcopy(ae_dictionary)
27
28    start_time = time.time()
29
30    #Section 2.1: Function for integration
31    function = np.sin
32
33    #Section 2.1: Integration Intervals
34    start = [0.0, np.pi]
35    end = [3.0*np.pi/8.0, 5.0*np.pi/4.0]
36    if interval not in [0, 1]:
37        raise ValueError("interval MUST BE 0 or 1")
38    a_ = start[interval]
39    b_ = end[interval]
40    #Section 2.1: Computing exact integral
41    exact_integral = np.cos(a_) - np.cos(b_)
42
43    #Section 2.2: Domain discretization
44    domain_x = np.linspace(a_, b_, 2 ** n_qbits + 1)
45
46    #Section 2.3: Function discretization
47    f_x = []
48    for i in range(1, len(domain_x)):
49        step_f = (function(domain_x[i]) + function(domain_x[i-1]))/2.0
50        f_x.append(step_f)
51        #x_.append((domain_x[i] + domain_x[i-1])/2.0)
52    f_x = np.array(f_x)
53
54    #Section 2.4: Array Normalisation
55    normalization = np.max(np.abs(f_x)) + 1e-8
56    f_norm_x = f_x/normalization
57
58    #Sections 2.5 and 2.6: Integral computation using AE techniques
59
60    #Section 3.2.3: configuring input dictionary for q_solve_integral
61    q_solve_configuration = {
62        "array_function" : f_norm_x,
63        "array_probability" : None,
64        "encoding" : 2
65    }
```

```
66     #Now added the AE configuration.
67     #The ae_dictionary_ has a local copy of the AE configuration.
68     q_solve_configuration.update(ae_dictionary_)
69     #The q_solve_integral needs a QPU object.
70     q_solve_configuration["qpu"] = get_qpu(q_solve_configuration["qpu"
    ])
71     #Compute the integral using AE algorithms!!
72     solution, solver_object = q_solve_integral(**q_solve_configuration)
73 
74     #Section 3.2.3: eq (3.7). It is an adapatation of eq (2.22)
75     estimator_s = normalization * (b_ - a_) * solution / (2 ** n_qbits)
76 
77     #Section 2.7: Getting the metrics
78     absolute_error = np.abs(estimator_s["ae"] - exact_integral)
79     relative_error = absolute_error / exact_integral
80     oracle_calls = solver_object.oracle_calls
81 
82     end_time = time.time()
83     elapsed_time = end_time - start_time
84 
85     #ae_dictionary_.pop('array_function')
86     #ae_dictionary_.pop('array_probability')
87 
88     #Section 4.2: Creating the output pandas DataFrame for using
89     #properly the KERNEL_BENCHMARK class
90 
91     #Adding the complete AE configuration
92     pdf = pd.DataFrame([ae_dictionary_])
93 
94     #Adding information about the computed integral
95     pdf["interval"] = interval
96     pdf["n_qbits"] = n_qbits
97     pdf["a_"] = a_
98     pdf["b_"] = b_
99 
100    #Adding the output from q_solve_integral
101    pdf = pd.concat([pdf, solution], axis=1)
102 
103    #Adding the AE computation of the integral
104    integral_columns = ["integral_" + col for col in solution.columns]
105    pdf[integral_columns] = estimator_s
106 
107    #Adding information about the integral that must be computed
108    pdf["exact_integral"] = exact_integral
109    pdf["riemann_sum"] = (b_ - a_) * np.sum(f_x) / (2 ** n_qbits)
110 
111    #Adding the normalization constant
112    pdf["normalization"] = normalization
```

```
113
114     #Error vs exact integral
115     pdf["absolute_error_exact"] = absolute_error
116     pdf["relative_error_exact"] = relative_error
117
118     #Error vs Riemann Sum
119     pdf["absolute_error_sum"] = np.abs(pdf["integral_ae"] - pdf["
    riemann_sum"])
120
121     #Error by Riemann aproximation to Integral
122     pdf["absolute_riemann_error"] = np.abs(
123         pdf["riemann_sum"] - pdf["exact_integral"])
124     pdf["oracle_calls"] = oracle_calls
125     pdf["elapsed_time"] = elapsed_time
126     pdf["run_time"] = solver_object.run_time
127     pdf["quantum_time"] = solver_object.quantum_time
128
129     #pdf will have a complete output for trazability.
130     #Columns for the metric according to 2.7 and 2.8 will be:
131     #[absolute_error_sum, oracle_calls,
132     #elapsed_time, run_time, quantum_time]
133     return pdf
```

Listing 15: *sine_integral* function from **ae_sine_integral.py** script.

In order to configure properly the **AE** algorithm, the JSON files in the **sourceCodeCases/02-Amplitude-Estimation/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 in order to properly configure the integral computation.

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

Listing 16 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 **sourceCodeCases/jsons/integral_mlae_configuration.json** file.

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

Listing 16: Example of the usage of the **ae_sine_integral.py** script from the command line.

3.2 Execution of the Complete AE Benchmark Procedure

This section explains how to execute a complete benchmark procedure, using the script **my_benchmark_execution.py**.

The **my_benchmark_execution.py** script is a modification of the corresponding template script located in sourceCodeCases/templates folder of the attached file. The functions, **run_code**, **compu-**

te_samples and **summarize_results** were modified; however, **KERNEL_BENCHMARK class** was not modified. The software adaptions for the **AE kernel** as presented in the subsequent sections.

run_code

Listing 17 shows the modifications of the **run_code** function for the **Benchmark Test Case** of the **AE kernel**. The main functionality of the function is executing the **Benchmark Test Case** for a fixed number of qubits (*n_qbits*), an input number of times (*repetitions*), and gathering all the mandatory metrics obtained- this step is done by the *sine_integral* function (see listing 15).

```
1  def run_code(n_qbits, repetitions, **kwargs):
2      """
3      For configuration and execution of the benchmark kernel.
4  
5      Parameters
6      ----------
7  
8      n_qbits : int
9          number of qubits used for domain discretization
10     repetitions : list
11         number of repetitions for the integral
12     kwargs : keyword arguments
13         for configuration of the benchmark kernel
14 
15     Returns
16     -------
17 
18     metrics : pandas DataFrame
19         DataFrame with the desired metrics obtained for the integral
   computation
20 
21     """
22     if n_qbits is None:
23         raise ValueError("n_qbits CAN NOT BE None")
24     if repetitions is None:
25         raise ValueError("samples CAN NOT BE None")
26 
27     from ae_sine_integral import sine_integral
28     #Here the code for configuring and execute the benchmark kernel
29     ae_configuration = kwargs.get("ae_configuration")
30     print(ae_configuration)
31     ae_configuration.update({"qpu": kwargs['qpu']})
32 
33     columns = [
34         "interval", "n_qbits", "absolute_error_sum", "oracle_calls",
35         "elapsed_time", "run_time", "quantum_time"
36     ]
37 
38     list_of_metrics = []
39     for j, interval in enumerate([0, 1]):
```

```
40         for i in range(repetitions[j]):
41             metrics = sine_integral(n_qbits, interval, ae_configuration
   )
42             list_of_metrics.append(metrics)
43     metrics = pd.concat(list_of_metrics)
44     metrics.reset_index(drop=True, inplace=True)
45     return metrics[columns]
```

Listing 17: **run_code** function for the **Benchmark Test Case** of the **AE kernel**.

compute_samples

Listing 18 shows the implementation of the **compute_samples** function for the **Benchmark Test Case** of the **AE kernel**. The main objective of the function, is to codify a strategy for computing the number of times, the **Benchmark Test Case** should be executed for getting some desired statistical significance.

```
1 def compute_samples(**kwargs):
2     """
3     This functions computes the number of executions of the benchmark
4     for assure an error r with a confidence of alpha
5 
6     Parameters
7     ----------
8 
9     kwargs : keyword arguments
10        For configuring the sampling computation
11
12    Returns
13    -------
14
15    samples : pandas DataFrame
16        DataFrame with the number of executions for each integration
   interval
17
18    """
19
20    #Configuration for sampling computations
21
22    #Desired Error in the benchmark metrics
23    relative_error = kwargs.get("relative_error", None)
24    if relative_error is None:
25        relative_error = 0.1
26    #Desired Confidence level
27    alpha = kwargs.get("alpha", None)
28    if alpha is None:
29        alpha = 0.05
30    #Minimum and Maximum number of samples
31    min_meas = kwargs.get("min_meas", None)
32    if min_meas is None:
```

```
        min_meas = 5
    max_meas = kwargs.get("max_meas", None)
    #if max_meas is None:
    #    max_meas = 100

    #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"

    #Compute mean and sd by integration interval
    metrics = kwargs.get("pre_metrics")
    std_ = metrics.groupby("interval").std()
    std_.reset_index(inplace=True)
    mean_ = metrics.groupby("interval").mean()
    mean_.reset_index(inplace=True)

    columns = [
        "absolute_error_sum", "oracle_calls",
        "elapsed_time", "run_time", "quantum_time"
    ]

    #Metrics
    from scipy.stats import norm
    zalpha = norm.ppf(1-(alpha/2)) # 95% of confidence level
    samples_ = (zalpha * std_[columns] / (relative_error * mean_[
   columns]))**2
    samples_ = samples_.max(axis=1).astype(int)
    samples_.name = "samples"

    #If user wants limit the number of samples
    samples_.clip(upper=max_meas, lower=min_meas, inplace=True)
    return list(samples_)
```

Listing 18: **compute_samples** function for codifying the strategy for computing the number of repetitions for the **Benchmark Test Case** of the **AE kernel**.

summarize_results

Listing 19 shows the implementation of the **summarize_results** function for the **Benchmark Test Case** of the **AE kernel**. The main objective of the function is post-processing the results of the complete **Benchmark Test Case** execution.

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
    """
```

```
5
6     #Code for summarize the benchamark results. Depending of the
7     #kernel of the benchmark
8     folder = kwargs.get("saving_folder")
9     csv_results = folder + kwargs.get("csv_results")
10
11    #results = pd.DataFrame()
12    pdf = pd.read_csv(csv_results, index_col=0, sep=";")
13    pdf["classic_time"] = pdf["elapsed_time"] - pdf["quantum_time"]
14    results = pdf.groupby(["interval", "n_qbits"]).describe()
15
16    return results
```

Listing 19: **summarize_results** function for summarizing the results from the **Benchmark Test Case** execution of the **AE kernel**.

KERNEL_BENCHMARK class

This python class defines the complete benchmark workflow and its *exe* method executes it properly by calling the correspondent functions (**run_code, compute_samples, summarize_results**). Each time the **Benchmark Test Case** is executed, the result is stored in a given CSV file.

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

Listing 20 shows an example for configuring an execution of an **IQAE** algorithm. The **source-Cod eCases/02-Amplitude-Estimation/jsons/integral_iqae_configurations.json** file is used for the **AE** algorithm configuration.

```
1  if __name__ == "__main__":
2      from ae_sine_integral import select_ae
3
4      AE = "IQAE"
5      benchmark_arguments = {
6          #Pre benchmark configuration
7          "pre_benchmark": True,
8          "pre_samples": [10, 10],
9          "pre_save": True,
10         #Saving configuration
11         "saving_folder": "./IQAE_Results/",
12         "benchmark_times": "{}_times_benchmark.csv".format(AE),
13         "csv_results": "{}_benchmark.csv".format(AE),
14         "summary_results": "{}_SummaryResults.csv".format(AE),
15         #Computing Repetitions configuration
16         "relative_error": None,
17         "alpha": None,
18         "min_meas": None,
19         "max_meas": None,
20         #List number of qubits tested
21         "list_of_qbits": [4, 6, 8, 10],
```

```
22          "qpu": "c"
23      }
24      #Setting the AE algorithm configuration
25      ae_problem = select_ae(AE)
26      #Added QPU to ae_problem
27      #ae_problem.update({"qpu":benchmark_arguments["qpu"]})
28
29      json_object = json.dumps(ae_problem)
30      #Writing the AE algorithm configuration
31      conf_file = benchmark_arguments["saving_folder"] + \
32          "benchmark_ae_conf.json"
33      with open(conf_file, "w") as outfile:
34          outfile.write(json_object)
35      #Added ae configuration
36      benchmark_arguments.update({
37          "ae_configuration": ae_problem,
38      })
39      ae_bench = KERNEL_BENCHMARK(**benchmark_arguments)
40      ae_bench.exe()
```

Listing 20: Example of configuration of a complete **Benchmark Test Case** execution. This part of the code should be located at the end of the **my_benchmark_execution.py** script.

Possible *kwargs* for the **KERNEL_BENCHMARK** class, *benchmark_arguments* dictionary in listing 20 are:

- *pre_benchmark*: For executing or not executing the *pre-benchmark* step.
- *pre_samples*: number of repetitions of the benchmark step. The first element is for the integration interval **0** and the second for the interval **1**.
- *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.
- *summary_results*: name for the *csv* file where the post-processed results (using the **summarize_results** function) will be stored.
- *list_of_qbits*: list with the different number of qubits for executing the complete **Benchmark Test Case**.

Other parameters like: *relative_error*, *alpha*, *min_meas* or *max_meas* that are used for the **compute_samples** function can be provided too, but for a complete **Benchmark Test Case** execution must be fixed to *None*.

As can be seen in the listing 20 the configuration of the implementation of **AE** algorithm 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 20 is located at the end of the **my_benchmark_execution.py** script. The different parts of the **Benchmark Test Case** complete execution and the **AE** algorithm used can be easily changed (the **AE** algorithm configuration can be changed by editing the corresponding JSON). For executing the **Benchmark Test Case**, the following command should be used:

python my_benchmark_execution.py

Finally, it is worth to comment lines 30-35 of the listing 20: here the configuration of the **AE** algorithm will be saved to JSON format file for traceability purposes.

3.3 Generation of the AE Benchmark Report

The results of a complete **AE Benchmark Test Case** must be reported in a separate JSON file- **Benchmark.V2.Schema_modified.json** provided in the sourceCodeCases/templates folder. For automating this process the following files should be modified:

- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **benchmark.py**

3.3.1 my_environment_info.py

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

Listing 21 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.

```
1  import platform
2  import psutil
3  from collections import OrderedDict
4
5  def my_organisation(**kwargs):
6      """
7      Given information about the organisation how uploads the benchmark
8      """
9      name = "CESGA"
10     return name
11
12 def my_machine_name(**kwargs):
13     """
14     Name of the machine where the benchmark was performed
15     """
16     machine_name = platform.node()
17     return machine_name
18
19 def my_qpu_model(**kwargs):
20     """
```

```
21      Name of the model of the QPU
22      """
23      qpu_model = "QLM"
24      return qpu_model
25
26  def my_qpu(**kwargs):
27      """
28      Complete info about the used QPU
29      """
30      #Basic schema
31      #QPUDescription = {
32      #    "NumberOfQPUs": 1,
33      #    "QPUs": [
34      #        {
35      #            "BasicGates": ["none", "none1"],
36      #            "Qubits": [
37      #                {
38      #                    "QubitNumber": 0,
39      #                    "T1": 1.0,
40      #                    "T2": 1.00
41      #                }
42      #            ],
43      #            "Gates": [
44      #                {
45      #                    "Gate": "none",
46      #                    "Type": "Single",
47      #                    "Symmetric": False,
48      #                    "Qubits": [0],
49      #                    "MaxTime": 1.0
50      #                }
51      #            ],
52      #            "Technology": "other"
53      #        },
54      #    ]
55      #}
56
57      #Defining the Qubits of the QPU
58      qubits = OrderedDict()
59      qubits["QubitNumber"] = 0
60      qubits["T1"] = 1.0
61      qubits["T2"] = 1.0
62
63      #Defining the Gates of the QPU
64      gates = OrderedDict()
65      gates["Gate"] = "none"
66      gates["Type"] = "Single"
67      gates["Symmetric"] = False
68      gates["Qubits"] = [0]
69      gates["MaxTime"] = 1.0
```

```
70
71      #Defining the Basic Gates of the QPU
72      qpus = OrderedDict()
73      qpus["BasicGates"] = ["none", "none1"]
74      qpus["Qubits"] = [qubits]
75      qpus["Gates"] = [gates]
76      qpus["Technology"] = "other"
77
78      qpu_description = OrderedDict()
79      qpu_description['NumberOfQPUs'] = 1
80      qpu_description['QPUs'] = [qpus]
81
82      return qpu_description
83
84  def my_cpu_model(**kwargs):
85      """
86      model of the cpu used in the benchmark
87      """
88      cpu_model = platform.processor()
89      return cpu_model
90
91  def my_frecuency(**kwargs):
92      """
93      Frcuency of the used CPU
94      """
95      #Use the nominal frequency. Here, it collects the maximum frequency
96      #print(psutil.cpu_freq())
97      cpu_frec = psutil.cpu_freq().max/1000
98      return cpu_frec
99
100 def my_network(**kwargs):
101     """
102     Network connections if several QPUs are used
103     """
104     network = OrderedDict()
105     network["Model"] = "None"
106     network["Version"] = "None"
107     network["Topology"] = "None"
108     return network
109
110 def my_QPUCPUConnection(**kwargs):
111     """
112     Connection between the QPU and the CPU used in the benchmark
113     """
114     #
115     # Provide the information about how the QPU is connected to the CPU
116     #
117     qpuccpu_conn = OrderedDict()
118     qpuccpu_conn["Type"] = "memory"
```

```
119     qpuccpu_conn["Version"] = "None"
120     return qpuccpu_conn
```

Listing 21: 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.

3.3.2 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 Test Case** is stored in this field.

Listing 22 shows an example of the configuration of the **my_benchmark_info.py** script for gathering the aforementioned information.

```
1  import platform
2  import psutil
3  import sys
4  import json
5  import jsonschema
6  import pandas as pd
7  from collections import OrderedDict
8  from my_benchmark_summary import summarize_results
9
10
11 def my_benchmark_kernel(**kwargs):
12     """
13     Name for the benchmark Kernel
14     """
15     return "AmplitudeEstimation"
16
17 def my_starttime(**kwargs):
18     """
19     Providing the start time of the benchmark
20     """
21     #start_time = "2022-12-12T16:46:57.268509+01:00"
22     times_filename = kwargs.get("times_filename", None)
23     pdf = pd.read_csv(times_filename, index_col=0)
24     start_time = pdf["StartTime"][0]
25     return start_time
26
27 def my_endtime(**kwargs):
28     """
29     Providing the end time of the benchmark
30     """
31     #end_time = "2022-12-12T16:46:57.268509+01:00"
32     times_filename = kwargs.get("times_filename", None)
33     pdf = pd.read_csv(times_filename, index_col=0)
34     end_time = pdf["EndTime"][0]
```

```
35     return end_time
36
37 def my_timemethod(**kwargs):
38     """
39     Providing the method for getting the times
40     """
41     #time_method = "None"
42     time_method = "time.time"
43     return time_method
44
45 def my_programlanguage(**kwargs):
46     """
47     Getting the programing language used for benchmark
48     """
49     #program_language = "None"
50     program_language = platform.python_implementation()
51     return program_language
52
53 def my_programlanguage_version(**kwargs):
54     """
55     Getting the version of the programing language used for benchmark
56     """
57     #language_version = "None"
58     language_version = platform.python_version()
59     return language_version
60
61 def my_programlanguage_vendor(**kwargs):
62     """
63     Getting the version of the programing language used for benchmark
64     """
65     #language_vendor = "None"
66     language_vendor = "Unknow"
67     return language_vendor
68
69 def my_api(**kwargs):
70     """
71     Collect the information about the used APIs
72     """
73     #api = OrderedDict()
74     #api["Name"] = "None"
75     #api["Version"] = "None"
76     #list_of_apis = [api]
77     modules = []
78     list_of_apis = []
79     for module in list(sys.modules):
80         api = OrderedDict()
81         module = module.split('.')[0]
82         if module not in modules:
83             modules.append(module)
```

```
84             api["Name"] = module
85             try:
86                 version = sys.modules[module].__version__
87             except AttributeError:
88                 #print("NO VERSION: "+str(sys.modules[module]))
89                 try:
90                     if  isinstance(sys.modules[module].version, str):
91                         version = sys.modules[module].version
92                         #print("\t Attribute Version"+version)
93                     else:
94                         version = sys.modules[module].version()
95                         #print("\t Methdod Version"+version)
96                 except (AttributeError, TypeError) as error:
97                     #print('\t NO VERSION: '+str(sys.modules[module]))
98                     try:
99                         version = sys.modules[module].VERSION
100                    except AttributeError:
101                        #print('\t\t NO VERSION: '+str(sys.modules[
   module]))
102                        version = "Unknown"
103            api["Version"] = str(version)
104            list_of_apis.append(api)
105    return list_of_apis
106
107 def my_quantum_compilation(**kwargs):
108     """
109     Information about the quantum compilation part of the benchmark
110     """
111     q_compilation = OrderedDict()
112     q_compilation["Step"] = "None"
113     q_compilation["Version"] = "None"
114     q_compilation["Flags"] = "None"
115     return [q_compilation]
116
117 def my_classical_compilation(**kwargs):
118     """
119     Information about the classical compilation part of the benchmark
120     """
121     c_compilation = OrderedDict()
122     c_compilation["Step"] = "None"
123     c_compilation["Version"] = "None"
124     c_compilation["Flags"] = "None"
125     return [c_compilation]
126
127
128 def my_metadata_info(**kwargs):
129     """
130     Other important info user want to store in the final json.
131     """
```

```
132
133     metadata = OrderedDict()
134     #metadata["None"] = None
135
136     json_file = kwargs.get("ae_config")
137     with open(json_file, 'r') as openfile:
138         #Reading from json file
139         json_object = json.load(openfile)
140
141     for key, value in json_object.items():
142         metadata[key] = value
143     return metadata
144
145
146 def my_benchmark_info(**kwargs):
147     """
148     Complete WorkFlow for getting all the benchmar informated related
    info
149     """
150     benchmark = OrderedDict()
151     benchmark["BenchmarkKernel"] = my_benchmark_kernel(**kwargs)
152     benchmark["StartTime"] = my_starttime(**kwargs)
153     benchmark["EndTime"] = my_endtime(**kwargs)
154     benchmark["ProgramLanguage"] = my_programlanguage(**kwargs)
155     benchmark["ProgramLanguageVersion"] = my_programlanguage_version(**
    kwargs)
156     benchmark["ProgramLanguageVendor"] = my_programlanguage_vendor(**
    kwargs)
157     benchmark["API"] = my_api(**kwargs)
158     benchmark["QuantumCompililation"] = my_quantum_compilation(**kwargs
    )
159     benchmark["ClassicalCompiler"] = my_classical_compilation(**kwargs)
160     benchmark["TimeMethod"] = my_timemethod(**kwargs)
161     benchmark["Results"] = summarize_results(**kwargs)
162     benchmark["MetaData"] = my_metadata_info(**kwargs)
163     return benchmark
```

Listing 22: 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 22). In order 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 subsection 3.2). 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 used, 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 3.3.3).

3.3.3 my_benchmark_summary.py

In this script, the *summarize_results* function is implemented. This function formats the results of a complete execution of a **Benchmark Test Case** of the **AE kernel** with the provided 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 23 shows an example of implementation of *summarize_results* function for the **AE** benchmark procedure.

```
from collections import OrderedDict
import psutil

def summarize_results(**kwargs):
    """
    Mandatory code for properly present the benchmark results following
    the 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",
        "absolute_error_sum", "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_
```

```
35              result["QubitPlacement"] = list(range(n_))
36              result["QPUs"] = [1]
37              result["CPUs"] = psutil.Process().cpu_affinity()
38              #Getting the data for n_qbiotd and interval
39              step_pdf = \
40                  pdf[(pdf["interval"] == interval) & (pdf["n_qbits"] ==
    n_)]
41              #result["TotalTime"] = 10.0
42              #result["SigmaTotalTime"] = 1.0
43              #result["QuantumTime"] = 9.0
44              #result["SigmaQuantumTime"] = 0.5
45              #result["ClassicalTime"] = 1.0
46              #result["SigmaClassicalTime"] = 0.1
47              result["TotalTime"] = step_pdf["elapsed_time"]["mean"].iloc
    [0]
48              result["SigmaTotalTime"] = step_pdf["elapsed_time"]["std"].
    iloc[0]
49              result["QuantumTime"] = step_pdf["quantum_time"]["mean"].
    iloc[0]
50              result["SigmaQuantumTime"] = step_pdf["quantum_time"]["std"
    ].iloc[0]
51              result["ClassicalTime"] = step_pdf["classic_time"]["mean"].
    iloc[0]
52              result["SigmaClassicalTime"] = step_pdf["classic_time"]["
    std"].iloc[0]
53              #For identify integration interval info. Not mandaatory but
54              #useful for indentify results
55              result["Interval"] = interval
56              metrics = []
57              #For each fixed number of qbits several metrics can be
    reported
58              for metric_name in list_of_metrics:
59                  metric = OrderedDict()
60                  #MANDATORY
61                  metric["Metric"] = metric_name
62                  #metric["Value"] = 0.1
63                  #metric["STD"] = 0.001
64                  metric["Value"] = step_pdf[metric_name]["mean"].iloc[0]
65                  metric["STD"] = step_pdf[metric_name]["std"].iloc[0]
66                  #Depending on the benchmark kernel
67                  metric["MIN"] = step_pdf[metric_name]["min"].iloc[0]
68                  metric["MAX"] = step_pdf[metric_name]["max"].iloc[0]
69                  metric["COUNT"] = step_pdf[metric_name]["count"].iloc
    [0]
70                  metrics.append(metric)
71              result["Metrics"] = metrics
72              results.append(result)
```

```
73     return results
```

Listing 23: 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 Test Case** execution were stored.

3.3.4 benchmark.py

The *benchmark.py* script can be used directly for gathering all the **Benchmark Test Case** execution information and results, for creating the final mandatory benchmark report.

It is not necessary to 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 Test Case** execution was stored.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark Test Case** execution was stored.
- *ae_config*: complete path where the configuration of the **AE** algorithm used in the benchmark (in JSON format) was stored.

4 Benchmark Test Case Implementation for Quantum Phase Estimation (QPE) Algorithms

The reference implementation for the **Benchmark Test Case** of the **Quantum Phase Estimation kernel** can be found in the **sourceCodeCases/03-Phase-Estimation** folder of the attached file.

This folder contains the following folder and files for implementing and executing the **QPE kernel Benchmark Test Case** as well as for generating the benchmark report:

- **QPE** folder: with the following files-
  - data_extracting.py: a module for executing QLM programs based on QLM Operators and for post-processing results from QLM QPU executions.
  - qpe.py: Script for executing classical Phase Estimation.
  - qpe_rz.py: Python class for executing the n qubits unitary operator, $R_z^{\otimes n}(\vec{\theta})$, for the **Benchmark Test Case**.
  - rz_lib.py: Script for executing theoretical and QLM simulation for computing the eigenvalues of the **Benchmark Test Case** unitary operator.
  - rz_qlm.py: Script for executing the QLM simulation for computing the eigenvalues of the unitary operator of the **Benchmark Test Case**.
- benchmark_exe.sh
- my_benchmark_execution.py
- my_benchmark_info.py
- my_benchmark_summary.py
- my_environment_info.py
- benchmark.py

The folder, **QPE**, handles the **QPE Benchmark Test Case** implementation and the files **my_benchmark_execution.py** and **benchmark_exe.sh.py** addresses the **PE kernel Benchmark Test Case** execution. While the files, **my_benchmark_info.py**, **my_benchmark_summary.py**, **my_environment_info.py** and **benchmark.py** are involved with the benchmark report generation.

Other files in the directory of the QPE of the attached file are python notebooks which demonstrates the **QPE kernel Benchmark Test Case** implementation and execution. They are:

1. 01_BTC_03_QPE_for_rzn_rz_library.ipynb
2. 02_BTC_03_QPE_for_rzn_Procedure.ipynb
3. 03_BTC_03_QPE_for_rzn_my_benchmark_execution.ipynb

In the following sections 4.1, 4.2 and 4.3, we give a detailed description of the actions of the files involved with the implementation and execution of the **Benchmark Test Case** as well as the generation of the benchmark report respectively.

4.1 Implementation of the QPE Benchmark Test Case

The problem for the **Benchmark Test Case** of the **QPE kernel** is the computation of the eigenvalues of a n qubits unitary operator given as:

$$R_z^{\otimes n}(\vec{\theta}) = \otimes_{i=1}^{n} R_z(\theta_i) \tag{12}$$

for a vector of n angles $\vec{\theta} = \{\theta_i\}; i = 0, 1, n-1$

Where: the $R_z(\theta)$ operator is a Z-axis rotation gate given by equation (13).

$$R_z(\theta) = (e^{-i\frac{\theta}{2}} |0\rangle \langle 0| + e^{i\frac{\theta}{2}} |1\rangle \langle 1|) \tag{13}$$

The QLM-reference implementation of the **QPE Benchmark Test Case**, makes use of the **rz_library**, a python package with all necessary functions for computing the eigenvalues of the n qubits $R_z^{\otimes n}(\vec{\theta})$ operator.

This section presents a complete description of the implementation of **QPE kernel Benchmark Test Case**. Subsections 4.1.1 and 4.1.2 discusses the library and class used in the implementation of the **QPE Benchmark Test Case**.

4.1.1 The rz_library for Computing Eigenvalues

The **rz_library** has been developed with all important functions for executing the theoretical and QLM simulation of the **QPE Benchmark Test Case**- the computation of eigenvalues of a n qubits $R_z^{\otimes n}(\vec{\theta})$ unitary operator. It is located in sourceCodeCases/03-Phase-Estimation/QPE/rz_lib.py of the attached file and the **qpe_class** in subsection 4.1.2 is defined within the framework of the **rz_library**.

4.1.2 qpe_rz.py: the *QPE_RZ* class

The **QPE_RZ** class located in **sourceCodeCases/03-Phase-Estimation/QPE/qpe_rz.py** has been developed for executing the complete benchmark step procedure for the **QPE kernel**. The description of how to use this class is given in the notebook located in **sourceCodeCases/03-Phase-Estimation/02_BTC_03_QPE_for_rzn_Procedure**
.ipynb. The functions implemented *QPE_RZ* class are already defined with the framework of the **rz_library** and the mandatory arguments that must be passed when this class is instantiated are:

- **number_of_qbits**: the number of qubits for apply the Kronecker products of R_z.
- **auxiliar_qbits_number**: auxiliary number of qubits for executing the **QPE Benchmark Test Case**.
- **angles**: the angles for the $R_z^{\otimes n}(\vec{\theta})$ operator. It can be:
  - float number: the same angle to all R_z.
  - list with angles: Each angle will be applied to a R_z. The Number of elements MUST be equal to the number of qubits.
  - string: Two different strings can be provided:
    - random: random angles will be provided to each R_z.

* exact: In this case, random angles will be provided to each R_z, but the final result will have a precision related to the number of auxiliary qubits for the **QPE Test Case.**

Listing 24 demonstrates the initializing of the *QPE_RZ* class:

```
import numpy as np
import pandas as pd

# myQLM qpus
from qat.qpus import PyLinalg, CLinalg
qpu_c = CLinalg()
qpu_p = PyLinalg()

from QPE.qpe_rz import QPE_RZ

n_qbits = 5
# Fix the precision of the QPE
aux_qbits = 6
# angles
angles = [np.pi / 2.0 for i in range(n_qbits)]
# Dictionary for configuring the class
qpe_dict = {
    'number_of_qbits' : n_qbits,
    'auxiliar_qbits_number' : aux_qbits,
    'angles' : angles,
    #'angles' : 'random',
    'qpu' : qpu_c,
    'shots' : 0
}
```

Listing 24: Initializing the *QPE_RZ* class.

Now, we discuss the Implementation of the main steps of the Benchmark Test Case with respect to the **rz_library** and the **QPE_RZ class** under the following headings:

1. Generation of the reference / theoretical probability distribution of the eigenvalues of $R_z^{\otimes n}(\vec{\theta})$ using the **rz_library**
2. Implementation of the Reference / Theoretical Probability Distribution of the eigenvalues of $R_z^{\otimes n}(\vec{\theta})$ using the **QPE_RZ class**
3. Computation of the Number of Shots for Simulating the PE Circuit on the Quantum Device
4. Generation of the QPE Probability Distribution of the Eigenvalues of $R_z^{\otimes n}(\vec{\theta})$ using the **rz_library**
5. Implementation of the QPE probability distribution of the eigenvalues of $R_z^{\otimes n}(\vec{\theta})$ using the **QPE_RZ** class.
6. Computing the metrics
7. Implementing the complete **QPE Benchmark Test Case** (subsection

The steps 1-7 above are explained as follows-

1. **Generation of the reference / theoretical probability distribution of the eigenvalues of R_z using the rz_library**

In this section, using a bulleted list, we discuss the steps and functions for generating a reference probability distribution, $P^{th}_{\lambda,m}(\frac{k}{2^m})$; k = 1, 2, ..., 2^m of the eigenvalues using the **rz_library** located in **sourceCodeCases/03-Phase-Estimation/QPE/rz_lib.py**.

(a) **Computation of the Eigenvalues of States, 2^n**

Given a system of n-qubits with the pure state $|i_0, i_1, i_2, .., i_n\rangle\,; i_j = \{0, 1\}$, the eigenvalues of an arbitrary $R_z^{\otimes n}(\vec{\theta})$ operator can be computed as:

$$2\pi\lambda_{|i_0,i_1,i_2,..,i_n\rangle} = -\frac{\sum_k^n (-1)^{i_k}\theta_k}{2} \tag{14}$$

For a set of the eigenvalues, $\lambda^{[2^n]}$, is given as:

$$\lambda_{|i_0,i_1,i_2,..,i_n\rangle} = -\frac{\sum_k^n (-1)^{i_k}\theta_k}{4\pi} \equiv \lambda_j \tag{15}$$

For an element in the set of eigenvalues, λ_j, is given as:

$$\lambda_i = -\frac{(-1)^i\theta}{4\pi}; \tag{16}$$

The computation of the eigenvalues of the states, 2^n, is implemented with the bulleted list below.

- **Computing the States, 2^n, as a Binary String**

  The first step is the computation of each state, 2^n as a binary string. This is implemented with the **bit_field** function, which computes the binary string of a given integer.

```
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)
```

```
19     left = np.zeros(max(size - right.size, 0))
20     full = np.concatenate((left, right))
21     return full.astype(int)
```

Listing 25: Computing 2^n as a binary string

- **Computing the corresponding eigenvalue, λ_j, of each state, $|j\rangle$**

  The second step, is the computation of the corresponding eigenvalue, λ_j, of each state, $|j\rangle$. This is implemented with the **rz_eigv_from_state** function, which takes as an argument, the output from the **bit_field** function and the n angles, $\vec{\theta}$, and returns the eigenvalue of the respective state:

```
1 def rz_eigenv_from_state(state, angles):
2     """
3     For a fixed input state and the angles of the R_z^n
      operator compute
4     the correspondent eigenvalue.
5
6     Parameters
7     ----------
8
9     state : np.array
10        Array with the binnary representation of the input
      state
11    angles: np.array
12        Array with the angles for the R_z^n operator.
13
14    Returns
15    -------
16
17    lambda_ : float
18        The eigenvalue for the input state of the R_z^n
      operator with
19        the input angles
20
21    """
22    new_state = np.where(state == 1, -1, 1)
23    # Computing the eigenvalue correspondent to the input
      state
24    thetas = - 0.5 * np.dot(new_state, angles)
25    # We want the angle between [0, 2pi]
26    thetas_2pi = np.mod(thetas, 2 * np.pi)
27    # Normalization of the angle between [0,1]
28    lambda_ = thetas_2pi / (2.0 * np.pi)
29    return lambda_
```

Listing 26: The **rz_eigv_from_state** function for computing the respective eigenvalues for each state, 2^n.

(b) **Obtaining the Histogram plot for the Theoretical Probability Distribution**

A histogram plot that corresponds to the complete list of the eigenvalues, $\lambda^{\otimes[2^n]}$, must be obtained for all possible states, 2^n. The bulleted list below explains the implementation steps for this.

- **Computing the complete list of the eigenvalues, $\lambda^{\otimes[2^n]}$:** The function, **rz_eigv**, takes as an argument the list of n angles and returns a pandas Dataframe object of all the eigenvalues of the arbitrary $R_z(\theta)$, unitary operator.

```
def rz_eigv(angles):
    """
    Computes the complete list of eigenvalues for a R_z^n
    operator
    for ainput 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
                significative 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 posible eigenstates
    eigv = [rz_eigenv_from_state(bitfield(i, n_qubits),
    angles)\
        for i in range(2**n_qubits)]
```

```
35      pdf = pd.DataFrame(
36          [eigv],
37          index=['Eigenvalues']
38      ).T
39      pdf['Int_lsb_left'] = pdf.index
40      state = pdf['Int_lsb_left'].apply(
41          lambda x: bin(x)[2:].zfill(n_qubits)
42      )
43      pdf['States'] = state.apply(lambda x: '|' + x[::-1] + '>
    ')
44      pdf['Int_lsb_right'] = state.apply(lambda x: int('0b'+x
    [::-1], base=0))
45      pdf = pdf[['States', 'Int_lsb_left', 'Int_lsb_right', '
    Eigenvalues']]
46      return pdf
```

Listing 27: The **rz_eigv** function for computing a complete list of the eigenvalues, $\lambda^{\otimes[2^n]}$.

It is important to note that, for the *option of exact choice of angles* (see subsection 4.1.2), depending on the angles, $\vec{\theta}$, for the operator, $R_z^{\otimes n}(\vec{\theta})$, some eigenvalues can have degeneracy (occurring several times). In this case, the less frequent eigenvalue and its frequency should be stored.

- **Obtaining the corresponding Histogram of the Eigenvalues, given in,** $\lambda^{\otimes[2^n]}$:

  For the complete list of eigenvalues obtained from the **rz_eigv** function, we need to generate the corresponding Histogram in the range $[0, 1]$ with 2^m, k-bins. In each bin, the histogram contains information about the frequency of the eigenvalues, $f_{\lambda k}$. The generation of the histogram is done with the **make_histogram** function. And the frequency, $f_{\lambda k}$, corresponds to the discrete Theoretical Probability Distribution of the eigenvalues, $P^{th}_{\lambda,m}(\frac{k}{2^m})$, where m is the discretization parameter, fixed for a number of auxiliary qubits.

```
1  def make_histogram(eigenvalues, discretization):
2      """
3      Given an input list of eigenvalues compute the
   correspondent
4      histogram using a bins = 2^discretization
5
6      Parameters
7      ----------
8
9      eigenvalues : list
10         List with the eigenvalues
11     discretization: int
12         Histogram discretization parameter: The number fo
   bins for the
13         histogram will be: 2^discretization
14
15     Returns
16     -------
```

```
17
18     pdf : pandas DataFrame
19         Pandas Dataframe with the 2^m bin frequency
    histogram for the
20         input list of eigenvalues. Columns
21             * lambda : bin discretization for eigenvalues
    based on the
22                 discretization input
23             * Probability: probability of finding any
    eigenvalue inside
24                 of the correspoondent lambda bin
25     """
26
27     # When the histogram is computed can be some problems
    with numeric
28     # approaches. So we compute the maximum number of
    decimals for
29     # a bare discretization of the bins and use it for
    rounding properly
30     # the eigenvalues
31     lambda_strings = [len(str(i / 2 ** discretization).split
    ('.')[1]) \
32         for i in range(2 ** discretization)]
33     decimal_truncation = max(lambda_strings)
34     trunc_eigenv = [round(i_, decimal_truncation) for i_ in
    list(eigenvalues)]
35     pdf = pd.DataFrame(
36         np.histogram(
37             trunc_eigenv,
38             bins=2 ** discretization,
39             range=(0, 1.0)
40         ),
41         index=["Counts", "lambda"]
42     ).T[:2 ** discretization]
43     pdf["Probability"] = pdf["Counts"] / sum(pdf["Counts"])
44     pdf.drop(columns=['Counts'], inplace=True)
45
46     return pdf
```

Listing 28: The **make_histogram** function for generating the histogram corresponding to the complete list of eigenvalues, $\lambda^{\otimes[2^n]}$.

2. **Implementation of the Reference / Theoretical Probability Distribution of the eigenvalues of R_z using the QPE_RZ class**

   Using the **QPE_RZ** class (see subsection 4.1.2), the generation of the reference probability can be obtained with the **theoretical_distribution** method. This method calls the functions, **rz_eigv** and **make_histogram** from the **rz_library** for computing the eigenvalues, $\lambda^{\otimes[2^n]}$ and generating a histogram for building the theoretical probability distribution, $P^{th}_{\lambda,m}(\frac{k}{2^m})$.

This function also calls the **computing_shots** function for computing the number of shots for simulating the Quantum Circuit discussed in step 4.

```
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
```

Listing 29: The **theoretical_distribution** method of the **QPE_RZ** class for generating the Theoretical Probability Distribution of the R_z eigenvalues.

Now, we discuss the implementation of the steps for executing the **QPE kernel** on the quantum device.

3. Computation of the Number of Shots for Simulating the QPE Circuit on the Quantum Device

The function **computing_shots** in the **rz_library** computes the number of shots:

$$n_{shots} = \frac{1000}{0.81 * f_{\lambda_{lf}}}; \; f_{\lambda_{lf}} \text{ is the frequency of the less frequent eigenvalue.} \tag{17}$$

for executing the **QPE Quantum Circuit** in subsection 4a. This function is implemented such that, the least occurring eigenvalues are measured at least 1000 times; it takes as an argument, the *Pandas DataFrame* containing the theoretical probability distribution and returns the number of shots for the **QPE** algorithm.

```
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 theorical one lower than the
```

```
7       discretization precision at least 1000 times
8
9       Parameters
10      ----------
11
12      pdf : pandas DataFrame
13          DataFrame with the theoretical eigenvalues
14
15      Returns
16      -------
17
18      shots : int
19          number of shots for QPE algorithm
20
21      """
22      # prob of less frequent eigenvalue
23      lfe = min(pdf.value_counts('Eigenvalues')) / len(pdf)
24      shots = int((1000 / (lfe * 0.81))) + 1
25      return shots
```

Listing 30: The **computing_shots** function, for executing and measuring the QPE circuit on the quantum device n-times

4. **Generation of the Quantum QPE Probability Distribution of the Eigenvalues of R_z using the rz_library**

   In this section, we discuss the steps and functions for generating a QPE probability distribution, $P^{QPE}_{\lambda,m}(\frac{k}{2^m})$; k = 1, 2, ..., 2^m of the eigenvalues. The implemented functions for this step are located in the files **rz_lib.py** and **rz_qlm.py** in the **sourceCodeCases/03-Phase-Estimation/QPE** folder of the attached file. The bulleted list below explains the implementation steps for this.

   (a) **Building the Quantum Circuit**

   Here we discuss the implementation of the quantum circuit for obtaining the Quantum probability distribution.

   i. **Building the $R_z^{\otimes n}(\vec{\theta})$ operator**:

   The **rz_angles** function from the **rz_library**, located in **sourceCodeCases/03-Phase-Estimation/QPE/rz_lib.py** allows for the building of the $R_z^{\otimes n}(\vec{\theta})$ operator shown in figure 6, with the input n angles. It creates a QLM abstract gate with the $R_z^{\otimes n}(\vec{\theta})$ for an input array of angles.

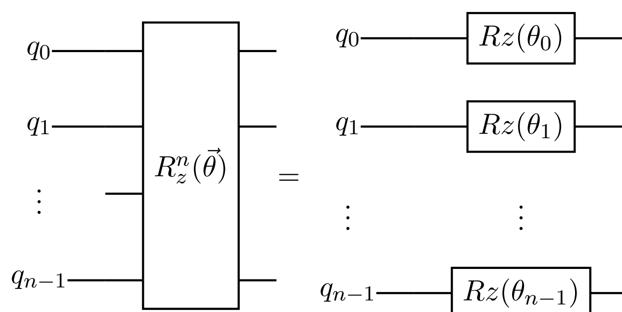

Figure 6: Circuit implementation of the $R^n_z(\vec{\theta})$ operator

```python
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

```

Listing 31: The function, **rz_angles** for building the $R_z^{\otimes n}(\vec{\theta})$ operator.

ii. **Computing the Initial State, Ψ_0 of the $R_z^{\otimes n}(\vec{\theta})$ operator**: This is implemented with the *Hadamard*, *H*-operator, which puts the, 2^n state in a linear combination of state, as shown in lines 243 - 246, in the file **rz_lib.py** and lines 45 - 48, in the file **rz_qlm.py**.

```
    # Creating the superposition initial state
    for i in range(n_qbits):
        #print(i)
        initial_state.apply(qlm.H, q_bits[i])

```

Listing 32: Initial state of the $R_z^{\otimes n}(\vec{\theta})$ operator.

iii. **Building the QPE circuit**: For a particular configuration of the **PE** (that is, exact or random configuration) kernel for the $R_z^{\otimes n}(\vec{\theta})$ operator, the **CQPE** class which implements the Classical Quantum Phase Estimation using the inverse of the Quantum Fourier Transform, builds the **QPE** circuit shown in figure 7. The **CQPE** is shown in listing 33 and is located in **sourceCodeCases/03-Phase-Estimation/QPE/qpe.py**.

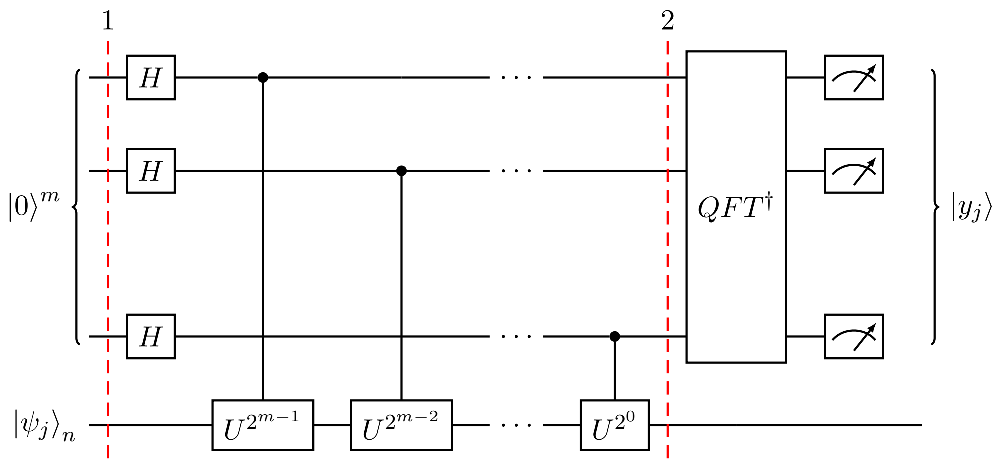

Figure 7: Canonical **QPE** circuit.

```
class CQPE:
    """
    Class for using classical Quantum Phase Estimation, with
    inverse of
    Quantum Fourier Transformation.

    Parameters
    ----------

    kwars : dictionary
        dictionary that allows the configuration of the CQPE
    algorithm: \\
        Implemented keys:

        initial_state : QLM Program
            QLM Program with the initial Psi state over the
            Grover-like operator will be applied
```

```
16              Only used if oracle is None
17          unitary_operator : QLM gate or routine
18              Grover-like operator which autovalues want to be
     calculated
19              Only used if oracle is None
20          cbits_number : int
21              number of classical bits for phase estimation
22          qpu : QLM solver
23              solver for simulating the resulting circuits
24          shots : int
25              number of shots for quantum job. If 0 exact
    probabilities
26              will be computed.
27      """
28
29      def __init__(self, **kwargs):
30          """
31          Method for initializing the class
32          """
33
34          # Setting attributes
35          # In this case we load directly the initial state
36          # and the grover operator
37          self.initial_state = kwargs.get("initial_state",
    None)
38          self.q_gate = kwargs.get("unitary_operator", None)
39          if (self.initial_state is None) or (self.q_gate is
    None):
40              text = "initial_state and grover keys should be
    provided"
41              raise KeyError(text)
42
43          # Number Of classical bits for estimating phase
44          self.auxiliar_qbits_number = kwargs.get("
    auxiliar_qbits_number", 8)
45
46          # Set the QPU to use
47          self.linalg_qpu = kwargs.get("qpu", None)
48          if self.linalg_qpu is None:
49              print("Not QPU was provide. PyLinalg will be
    used")
50              from qat.qpus import get_default_qpu
51              self.linalg_qpu = get_default_qpu()
52
53          self.shots = kwargs.get("shots", 10)
54          self.complete = kwargs.get("complete", False)
55
56          #Quantum Routine for QPE
57          #Auxiliar qbits
```

```
58         self.q_aux = None
59         #Qubits rtegisters
60         self.registers = None
61         #List of ints with the position of the qbits for
    measuring
62         self.meas_qbits = None
63         #For storing results
64         self.result = None
65         #For storing qunatum times
66         self.quantum_times = []
67         #For storing the QPE routine
68         self.circuit = None
69 
70     def run(self):
71         """
72         Creates the quantum phase estimation routine
73         """
74         qpe_routine = qlm.QRoutine()
75         #Creates the qbits foe applying the operations
76         self.registers = qpe_routine.new_wires(self.
    initial_state.arity)
77         #Initializate the registers
78         qpe_routine.apply(self.initial_state, self.registers
    )
79         #Creates the auxiliary qbits for phase estimation
80         self.q_aux = qpe_routine.new_wires(self.
    auxiliar_qbits_number)
81         #Apply controlled Operator an increasing number of
    times
82         for i, aux in enumerate(self.q_aux):
83             #Apply Haddamard to all auxiliary qbits
84             qpe_routine.apply(qlm.H, aux)
85             #Power of the unitary operator depending of the
    position
86             #of the auxiliary qbit.
87             step_q_gate = load_qn_gate(self.q_gate, 2**i)
88             #Controlled application of power of unitary
    operator
89             qpe_routine.apply(step_q_gate.ctrl(), aux, self.
    registers)
90         #Apply the QFT
91         qpe_routine.apply(qlm.qftarith.QFT(len(self.q_aux)).
    dag(), self.q_aux)
92         self.circuit = qpe_routine
93 
94         start = time.time()
95         #Getting the result
96         self.meas_qbits = [
```

```
97             len(self.registers) + i for i, aux in enumerate(
    self.q_aux)]
98         self.result, _, _, _ = get_results(
99             self.circuit,
100            linalg_qpu=self.linalg_qpu,
101            shots=self.shots,
102            qubits=self.meas_qbits,
103            complete=self.complete
104        )
105        end = time.time()
106        self.quantum_times.append(end-start)
107        del self.result["Amplitude"]
108        self.result["lambda"] = self.result["Int"] / (2**len
    (self.q_aux))
109
```

Listing 33: The **CQPE** class for Building and Simulating the QPE Quantum Circuit.

(b) **Obtaining the Histogram Plot for the QPE Probability Distribution**

As in the case of the theoretical probability distribution, $P^{th}_{\lambda,m}(\frac{k}{2^m})$, a histogram plot that corresponds to the complete list of the eigenvalues, $\lambda^{[2^n]}$, must be obtained for all possible states, 2^n for building the QPE probability distribution, $P^{QPE}_{\lambda,m}(\frac{k}{2^m})$. The bulleted list below explains the implementation steps for this.

i. **Computing the complete list of the eigenvalues,** $\lambda^{\otimes[2^n]}$:

- **Simulating the Quantum Circuit**:

  The, $R_z^{\otimes n}(\vec{\theta})$ operator and its initial state, Ψ_0 in subsection 4a.1, is provided to the **qpe_rz_qlm** function of the **rz_library** (see listing 34), and is simulated by calling the *run* method of the **CQPE** class, shown in listing 33.

- **The complete list of the eigenvalues**: This is obtained by the **qpe_rz_qlm** function of the **rz_library**.

```
1     qft_pe = CQPE(**qft_pe_dict)
2     qft_pe.run()
3     qft_pe_results = qft_pe.result
4     qft_pe_results.sort_values('lambda', inplace=True)
5     results = qft_pe_results[['lambda', 'Probability']]
6     results.reset_index(drop=True, inplace=True)
7     return results, qft_pe
8
```

Listing 34: Code section from the **qpe_rz_qlm** function from the **rz_library** for computing the complete list of the eigenvalues, $\lambda^{\otimes[2^n]}$, for the Quantum Probability Distribution.

ii. **Obtaining the corresponding histogram of the eigenvalues, given in,** $\lambda^{\otimes[2^n]}$:

For the complete list of eigenvalues obtained from the **qpe_rz_qlm** function, we need to generate the corresponding Histogram as in the case of the Reference/Theoretical Probability Distribution. This histogram is in the range $[0, 1]$ with 2^m, k-bins. In each bin, the histogram contains information about the frequency of the eigenvalues,

$f_{\lambda k}$. The generation of the histogram is done with the **make_histogram** function (listing 28) in the **rz_library**. And the frequency, $f_{\lambda k}$, corresponds to the discrete QPE probability distribution of the eigenvalues, $P^{QPE}_{\lambda,m}(\frac{k}{2^m})$, where m is the discretization parameter, fixed for a number of auxiliary qubits.

5. **Implementation of the Quantum probability distribution of the eigenvalues of R_z using the QPE_RZ class**

The computation of the QPE Probability Distribution, is implemented with the **quantum_distribution** method of the **QPE_RZ** class.

```
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
```

Listing 35: The **quantum_distribution** method of the **QPE_RZ** class for generating the QPE Probability Distribution of the R_z eigenvalues.

6. **Computing the metrics**

The Probability Distributions obtained for the theoretical implementation, $P^{th}_{\lambda,m}(\frac{k}{2^m})$, and the QPE implementation, $P^{QPE}_{\lambda,m}(\frac{k}{2^m})$, is compared with **The Kolmogorov-Smirnov(KS)** and **fidelity** metric, implemented in the **get_metrics** method of the **QPE_RZ** class, shown in listing 36.

It is important to note here that, the results of the **KS** and **fidelity** metric is relative to the configuration of the **QPE kernel**. While the **KS** metric is a good metric for both *random* and *exact* configurations of the **QPE kernel**, the **fidelity** metric is best for the *exact* configuration of the **QPE kernel**. The notebook in **sourceCodeCases/03-Phase-Estimation/02_BTC_03_QPE_for_rzn_Procedure.ipynb**, illustrates this relativity.

```
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
```

```
11        qv = self.quantum_eigv_dist['Probability']
12        tv = self.theorical_eigv_dist['Probability']
13        self.fidelity = qv @ tv / (np.linalg.norm(qv) * np.linalg.
    norm(tv))
```

Listing 36: **get_metrics** method for computing the metrics for $P^{th}_{\lambda,m}(\frac{k}{2^m})$ and $P^{QPE}_{\lambda,m}(\frac{k}{2^m})$.

7. **Implementing the complete QPE Benchmark Test Case**

The **QPE_RZ** class for implementing the complete **QPE** Benchmark Test Case is given in listing 37 and a demonstration of its usage is given in the notebook located in **sourceCodeCases/03-Phase-Estimation/QPE/qpe_rz.py**.

```
1  import time
2  import numpy as np
3  import pandas as pd
4  from scipy.stats import norm, entropy, chisquare, chi2
5  from QPE import rz_lib
6
7  class QPE_RZ:
8      """
9      Probability Loading
10     """
11
12
13     def __init__(self, **kwargs):
14         """
15
16         Method for initializing the class
17
18         """
19
20         self.n_qbits = kwargs.get("number_of_qbits", None)
21         if self.n_qbits is None:
22             error_text = "The number_of_qbits argument CAN NOT BE
    NONE."
23             raise ValueError(error_text)
24         self.auxiliar_qbits_number = kwargs.get("
    auxiliar_qbits_number", None)
25         if self.auxiliar_qbits_number is None:
26             error_text = "Provide the number of auxiliar qubits for
     QPE"
27             raise ValueError(error_text)
28
29         # Minimum angle measured by the QPE
30         self.delta_theta = 4 * np.pi / 2 ** self.
    auxiliar_qbits_number
31
32         angles = kwargs.get("angles", None)
33         if type(angles) not in [str, float, list]:
```

```
34            error_text = "Be aware! angles keyword can only be str"
 + \
35                ", float or list "
36            raise ValueError(error_text)
37
38        self.angle_name = ''
39
40        if isinstance(angles, str):
41            if angles == 'random':
42                self.angle_name = 'random'
43                self.angles = [np.pi * np.random.random() \
44                    for i in range(self.n_qbits)]
45            elif angles == 'exact':
46                # Here we compute the angles of the R_z^n operator
for
47                # obtaining exact eigenvalues in QPE. We begin in
0.5pi
48                # and sum or rest randomly the minimum QPE measured
49                # angle
50                self.angle_name = 'exact'
51                self.angles = []
52                angle_0 = np.pi / 2.0
53                for i_ in range(self.n_qbits):
54                    angle_0 = angle_0 + (-1) ** np.random.randint
(2) *\
55                        self.delta_theta
56                    self.angles.append(angle_0)
57            else:
58                error_text = "Be aware! If angles is str then only"
 + \
59                    "can be random"
60                raise ValueError(error_text)
61
62        if isinstance(angles, float):
63            self.angles = [angles for i in range(self.n_qbits)]
64
65        if isinstance(angles, list):
66            self.angles = angles
67            if len(self.angles) != self.n_qbits:
68                error_text = "Be aware! The number of elements in
angles" + \
69                    "MUST BE equal to the n_qbits"
70                raise ValueError(error_text)
71
72        # Set the QPU to use
73        self.qpu = kwargs.get("qpu", None)
74        if self.qpu is None:
75            error_text = "Please provide a QPU."
76            raise ValueError(error_text)
```

```
77
78         # Shots for measuring thw QPE circuit
79         self.shots = kwargs.get("shots", None)
80         if self.shots is not None:
81             text = "BE AWARE! The keyword shots should be None
    because" +\
82                 "shots should be computed in function of the
    theorical" +\
83                 "eigenvalues. You can only provide 0 for doing some
     testing" +\
84                 "in the class. 0 will imply complete simnulation of
     QPE circuit"
85             print(text)
86             if self.shots != 0:
87                 error_text = "BE AWARE! The keyword shots must be
    None or 0"
88                 raise ValueError(error_text)
89
90         # For storing classical eigenvalue distribution
91         self.theorical_eigv = None
92         self.theorical_eigv_dist = None
93         # For storing quantum eigenvalue distribution
94         self.quantum_eigv_dist = None
95         # For storing attributes from CQPE class
96         self.circuit = None
97         self.quantum_time = None
98
99         # Computing complete time of the procces
100        self.elapsed_time = None
101
102        # Metric attributes
103        self.ks = None
104        self.fidelity = None
105
106        # Pandas DataFrame for summary
107        self.pdf = None
108
109    def theoretical_distribution(self):
110        """
111        Computes the theoretical distribution of Rz eigenvalues
112        """
113        # Compute the complete eigenvalues
114        self.theorical_eigv = rz_lib.rz_eigv(self.angles)
115        # Compute the eigenvalue distribution using
    auxiliar_qbits_number
116        self.theorical_eigv_dist = rz_lib.make_histogram(
117            self.theorical_eigv['Eigenvalues'], self.
    auxiliar_qbits_number)
118        if self.shots is None:
```

```
119             # Compute the number of shots for QPE circuit
120             self.shots = rz_lib.computing_shots(self.theorical_eigv
    )
121         else:
122             if self.shots != 0:
123                 self.shots = rz_lib.computing_shots(self.
    theorical_eigv)
124             else:
125                 pass
126 
127     def quantum_distribution(self):
128         """
129         Computes the quantum distribution of Rz eigenvalues
130         """
131         self.quantum_eigv_dist, qpe_object = rz_lib.qpe_rz_qlm(
132             self.angles,
133             auxiliar_qbits_number=self.auxiliar_qbits_number,
134             shots=self.shots,
135             qpu=self.qpu
136 
137         )
138         self.circuit = qpe_object.circuit
139         self.quantum_time = qpe_object.quantum_times
140 
141     def get_metrics(self):
142         """
143         Computing Metrics
144         """
145         # Kolmogorov-Smirnov
146         self.ks = np.abs(
147             self.theorical_eigv_dist['Probability'].cumsum() \
148                 - self.quantum_eigv_dist['Probability'].cumsum()
149         ).max()
150         # Fidelity
151         qv = self.quantum_eigv_dist['Probability']
152         tv = self.theorical_eigv_dist['Probability']
153         self.fidelity = qv @ tv / (np.linalg.norm(qv) * np.linalg.
    norm(tv))
154 
155 
156     def exe(self):
157         """
158         Execution of workflow
159         """
160         tick = time.time()
161         # Compute theoretical eigenvalues
162         self.theoretical_distribution()
163         # Computing eigenvalues using QPE
164         self.quantum_distribution()
```

```
165         # Compute the metrics
166         self.get_metrics()
167         tack = time.time()
168         self.elapsed_time = tack - tick
169         self.summary()
170 
171     def summary(self):
172         """
173         Pandas summary
174         """
175         self.pdf = pd.DataFrame()
176         self.pdf["n_qbits"] = [self.n_qbits]
177         self.pdf["aux_qbits"] = [self.auxiliar_qbits_number]
178         self.pdf["delta_theta"] = self.delta_theta
179         self.pdf["angle_method"] = [self.angle_name]
180         self.pdf["angles"] = [self.angles]
181         self.pdf["qpu"] = [self.qpu]
182         self.pdf["shots"] = [self.shots]
183         self.pdf["KS"] = [self.ks]
184         self.pdf["fidelity"] = [self.fidelity]
185         self.pdf["elapsed_time"] = [self.elapsed_time]
186         self.pdf["quantum_time"] = [self.quantum_time[0]]
187 
188 if __name__ == "__main__":
189     import argparse
190 
191     parser = argparse.ArgumentParser()
192 
193     parser.add_argument(
194         "-n_qbits",
195         dest="n_qbits",
196         type=int,
197         help="Number of qbits for unitary operator.",
198         default=None,
199     )
200     parser.add_argument(
201         "-aux_qbits",
202         dest="aux_qbits",
203         type=int,
204         help="Number of auxiliar qbits for QPE",
205         default=None,
206     )
207     #QPU argument
208     parser.add_argument(
209         "-qpu",
210         dest="qpu",
211         type=str,
212         default="python",
213         help="QPU for simulation: [qlmass, python, c]",
```

```
214     )
215     parser.add_argument(
216         "-shots",
217         dest="shots",
218         type=int,
219         help="Number of shots: Only valid number is 0 (for exact
    simulation)",
220         default=None,
221     )
222     parser.add_argument(
223         "-angles",
224         dest="angles",
225         type=int,
226         help="Select the angle load method: 0->exact. 1->random",
227         default=None,
228     )
229 
230     args = parser.parse_args()
231     print(args)
232 
233     if args.angles == 0:
234         angles = 'exact'
235     elif args.angles == 1:
236         angles = 'random'
237     else:
238         raise ValueError("angles parameter can be only 0 or 1")
239 
240     configuration = {
241         "number_of_qbits" : args.n_qbits,
242         "auxiliar_qbits_number" : args.aux_qbits,
243         "qpu" : rz_lib.get_qpu(args.qpu),
244         "shots" : args.shots,
245         "angles" : angles
246     }
247     qpe_rz_b = QPE_RZ(**configuration)
248     qpe_rz_b.exe()
249     print(qpe_rz_b.pdf)
250     print('KS: {}'.format(list(qpe_rz_b.pdf['KS'])[0]))
251     print('fidelity: {}'.format(list(qpe_rz_b.pdf['fidelity'])[0]))
```

Listing 37: The **QPE_RZ** class for executing the complete **QPE** Benchmark Test Case.

4.2 Execution of the Complete QPE Benchmark Procedure

The action of the **my_benchmark_execution.py** script is discussed in this section. This script is a modification of the corresponding template script located in **sourceCodeCases/templates** folder of the attached file. The functions, **run_code**, **compute_samples** and **summarize_results** were modified; however, the **KERNEL_BENCHMARK class** was not modified. The software adaptions for the **PE kernel** are presented in the subsequent sections.

run_code

Listing 38 shows the modifications of the **run_code** function for the **Benchmark Test Case** of the **QPE kernel**. The main functionality of the function is executing the **Benchmark Test Case** for a fixed number of- qubits (*n_qbits*), auxiliary qubits, (*aux_qbits*), angles, repetitions for each execution of the benchmark, (*repetitions*), specifying a benchmark stage and gathering all the mandatory metrics obtained. Listing 38 shows the **qpe_rz** class and its *exe* method for the execution of the **PE Benchmark Test Case.**

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

    Parameters
    ----------

    n_qbits : int
        number of qubits used for domain discretization
    repetitions : list
        number of repetitions for the integral
    kwargs : keyword arguments
        for configuration of the benchmark kernel

    Returns
    -------

    metrics : pandas DataFrame
        DataFrame with the desired metrics obtained for the integral
   computation

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

    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", "absolute_error_sum", "oracle_calls",
        "elapsed_time", "run_time", "quantum_time"
    ]

    list_of_metrics = []
    for j, interval in enumerate([0, 1]):
        for i in range(repetitions[j]):
```

```
41            metrics = sine_integral(n_qbits, interval, ae_configuration
   )
42            list_of_metrics.append(metrics)
43     metrics = pd.concat(list_of_metrics)
44     metrics.reset_index(drop=True, inplace=True)
45     return metricsdef run_code(iterator_step, repetitions, stage_bench,
       **kwargs):
46     """
47     For configuration and execution of the benchmark kernel.
48 
49     Parameters
50     ----------
51 
52     iterator_step : tuple
53         tuple with elements from iterator built from build_iterator.
54     repetitions : list
55         number of repetitions for each execution
56     stage_bench : str
57         benchmark stage. Only: benchmark, pre-benchamrk
58     kwargs : keyword arguments
59         for configuration of the benchmark kernel
60 
61     Returns
62     -------
63 
64     metrics : pandas DataFrame
65         DataFrame with the desired metrics obtained for the integral
   computation
66     save_name : string
67         Desired name for saving the results of the execution
68 
69     """
70     # if n_qbits is None:
71     #     raise ValueError("n_qbits CAN NOT BE None")
72 
73     if stage_bench not in ['benchmark', 'pre-benchmark']:
74         raise ValueError(
75             "Valid values for stage_bench: benchmark or pre-benchmark'"
   )
76 
77     if repetitions is None:
78         raise ValueError("samples CAN NOT BE None")
79 
80     #Here the code for configuring and execute the benchmark kernel
81     kernel_configuration_ = deepcopy(kwargs.get("kernel_configuration",
       None))
82     if kernel_configuration_ is None:
83         raise ValueError("kernel_configuration can not be None")
84     # 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 38: **run_code** function for the **Benchmark Test Case** of the **QPE kernel**.

compute_samples

Listing 39 shows the implementation of the **compute_samples** function for the **Benchmark Test Case** of the **QPE kernel**. The main objective of the function, is to codify a strategy for computing the number of times, the **Benchmark Test Case** should be executed for getting some desired statistical significance.

```
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
    ----------
```

```
8
9       kwargs : keyword arguments
10          For configuring the sampling computation
11
12      Returns
13      -------
14
15      samples : pandas DataFrame
16          DataFrame with the number of executions for each integration
    interval
17
18      """
19
20      #Configuration for sampling computations
21
22      #Desired Confidence level
23      alpha = kwargs.get("alpha", 0.05)
24      metrics = kwargs.get('pre_metrics')
25      bench_conf = kwargs.get('kernel_configuration')
26
27      #Code for computing the number of samples for getting the desired
28      #statististical significance. Depends on benchmark kernel
29      #samples_ = pd.Series([100, 100])
30      #samples_.name = "samples"
31
32      method = metrics['angle_method'].unique()
33      if len(method) != 1:
34          raise ValueError('Only can provide one angle method!')
35
36      from scipy.stats import norm
37      zalpha = norm.ppf(1-(alpha/2)) # 95% of confidence level
38
39      method = method[0]
40
41      if method == 'exact':
42
43          # Error expected for the means fidelity
44          error_fid = bench_conf.get("fidelity_error", 0.05)
45          metric_fidelity = ['fidelity']
46          std_ = metrics[metric_fidelity].std()
47          mean_ = metrics[metric_fidelity].mean()
48          samples_ = (zalpha * std_ / error_fid)**2
49      elif method == 'random':
50          # Error expected for the means KS
51          error_ks = bench_conf.get("ks_error", 0.05)
52          metric_ks = ['KS']
53          std_ = metrics[metric_ks].std()
54          mean_ = metrics[metric_ks].mean()
55          samples_ = (zalpha * std_ / error_ks)**2
```

```
56     else:
57         raise ValueError('Angle method can be only: exact or random')
58     #samples_ = pd.concat([samples_fidelity, samples_ks], axis=0).T
59     #samples_ = pd.Series(samples_.max(axis=0).astype(int) + 1)
60     samples_ = samples_.astype(int) + 1
61     print(method, samples_)
62 
63     #If user wants limit the number of samples
64     #Minimum and Maximum number of samples
65     min_meas = kwargs.get("min_meas", 100)
66     max_meas = kwargs.get("max_meas", None)
67     samples_.clip(upper=max_meas, lower=min_meas, inplace=True)
68     return list(samples_)
```

Listing 39: **compute_samples** function for codifying the strategy for computing the number of repetitions for the **Benchmark Test Case** of the **QPE kernel**.

summarize_results

Listing 40 shows the implementation of the **summarize_results** function for the **Benchmark Test Case** of the **QPE kernel**. The main objective of the function is post-processing the results of the complete **Benchmark Test Case** execution.

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.

```
1  def summarize_results(**kwargs):
2      """
3      Create summary with statistics
4      """
5  
6      folder = kwargs.get("saving_folder")
7      csv_results = folder + kwargs.get("csv_results")
8  
9      #Code for summarize the benchamark results. Depending of the
10     #kernel of the benchmark
11     pdf = pd.read_csv(csv_results, index_col=0, sep=";")
12     pdf["classic_time"] = pdf["elapsed_time"] - pdf["quantum_time"]
13     # The angles are randomly selected. Not interesting for aggregation
14     pdf.drop(columns=['angles'], inplace=True)
15     results = pdf.groupby(["n_qbits", "aux_qbits", "angle_method"]).agg
       (
16         ["mean", "std", "count"] + \
17             [('std_mean', lambda x: np.std(x)/np.sqrt(len(x)))])
18     results.drop(columns=[
19         ('delta_theta', 'std'),
20         ('delta_theta', 'count'),
21         ('delta_theta', 'std_mean'),
22         ('shots', 'std'),
23         ('shots', 'count'),
24         ('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 40: **summarize_results** function for summarizing the results from the **Benchmark Test Case** execution of the **QPE kernel**.

KERNEL_BENCHMARK class

This python class defines the complete benchmark workflow and its *exe* method executes it properly by calling the correspondent functions (**run_code, compute_samples, summarize_results**). Each time the **Benchmark Test Case** is executed, the result is stored in a given CSV file.

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

Listing 41 shows an example for configuring an execution of a Benchmark Test Case: in this case, the **PE kernel** for the $R_z^{\otimes n}(\vec{\theta})$ operator is configured with *random* angles and *exact* angles; the *random* configuration is evaluated based on the **Kolmogorov-Smirnov (KS)** metric, while the *exact* configuration is evaluated based on the *fidelity* metric.

```
if __name__ == "__main__":

    import os
    import shutil

    kernel_configuration = {
        "angles" : ["random", 'exact'],
        'auxiliar_qbits_number' : [4, 6, 8, 10],
        "qpu" : "default", #python, qlmass, default
        "fidelity_error" : 0.001,
        "ks_error" : 0.05
    }

    benchmark_arguments = {
        #Pre benchmark sttuff
        "pre_benchmark": True,
        "pre_samples": [15, 20],
        "pre_save": False,
        #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": 0.05,
```

```
27          "min_meas": 20,
28          "max_meas": None,
29          #List number of qubits tested
30          "list_of_qbits": [4, 6, 8, 10, 12],
31      }
32
33      #Configuration for the benchmark kernel
34      benchmark_arguments.update({"kernel_configuration":
      kernel_configuration})
35      kernel_bench = KERNEL_BENCHMARK(**benchmark_arguments)
36      kernel_bench.exe()
```

Listing 41: Example of configuration of a complete **Benchmark Test Case** execution. This part of the code should be located at the end of the **my_benchmark_execution.py** script.

Possible *kwargs* for the **KERNEL_BENCHMARK** class, *benchmark_arguments* dictionary in listing 20 are:

- *pre_benchmark*: For executing or not executing the *pre-benchmark* step.
- *pre_samples*: number of repetitions of the benchmark step. The first element is for the integration interval **0** and the second for the interval **1**.
- *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_BENCH MARK** 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.
- *summary_results*: name for the *csv* file where the post-processed results (using the **summarize_results** function) will be stored.
- *list_of_qbits*: list with the different number of qubits for executing the complete **Benchmark Test Case**.

Other parameters like: *alpha*, *min_meas* or *max_meas* that are used for the **compute_samples** function can be provided too, but for a complete **Benchmark Test Case** execution must be fixed to *0.05*, *20* and *None* respectively.

As shown in listing 41 the configuration of the implementation of **QPE** algorithm is passed in the benchmark arguments arguments under the key *kernel_configuration*.

For executing the **QPE Benchmark Test Case**, the following command should be used:

python my_benchmark_execution.py

4.3 Generation of the QPE Benchmark Report

The results of a complete **QPE Benchmark Test Case** must be reported in a separate JSON file-**Benchmark.V2.Schema_modified.json** provided in the sourceCodeCases/templates folder. For automating this process the following files should be modified:

- **my_environment_info.py**
- **my_benchmark_info.py**
- **my_benchmark_summary.py**
- **benchmark.py**

4.3.1 my_environment_info.py

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

Listing 42 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.

```
1
2   import platform
3   import psutil
4   from collections import OrderedDict
5
6   def my_organisation(**kwargs):
7       """
8       Given information about the organisation how uploads the
    benchmark
9       """
10      name = "CESGA"
11      return name
12
13  def my_machine_name(**kwargs):
14      """
15      Name of the machine where the benchmark was performed
16      """
17      #machine_name = "None"
18      machine_name = platform.node()
19      return machine_name
20
21  def my_qpu_model(**kwargs):
22      """
23      Name of the model of the QPU
24      """
25      qpu_model = "CLinalg"
26      return qpu_model
27
28  def my_qpu(**kwargs):
29      """
30      Complete info about the used QPU
31      """
32      #Basic schema
33      #QPUDescription = {
34      #    "NumberOfQPUs": 1,
```

```
35          #    "QPUs": [
36          #        {
37          #            "BasicGates": ["none", "none1"],
38          #            "Qubits": [
39          #                {
40          #                    "QubitNumber": 0,
41          #                    "T1": 1.0,
42          #                    "T2": 1.00
43          #                }
44          #            ],
45          #            "Gates": [
46          #                {
47          #                    "Gate": "none",
48          #                    "Type": "Single",
49          #                    "Symmetric": False,
50          #                    "Qubits": [0],
51          #                    "MaxTime": 1.0
52          #                }
53          #            ],
54          #            "Technology": "other"
55          #        },
56          #    ]
57          #}
58
59          #Defining the Qubits of the QPU
60          qubits = OrderedDict()
61          qubits["QubitNumber"] = 0
62          qubits["T1"] = 1.0
63          qubits["T2"] = 1.0
64
65          #Defining the Gates of the QPU
66          gates = OrderedDict()
67          gates["Gate"] = "none"
68          gates["Type"] = "Single"
69          gates["Symmetric"] = False
70          gates["Qubits"] = [0]
71          gates["MaxTime"] = 1.0
72
73
74          #Defining the Basic Gates of the QPU
75          qpus = OrderedDict()
76          qpus["BasicGates"] = ["none", "none1"]
77          qpus["Qubits"] = [qubits]
78          qpus["Gates"] = [gates]
79          qpus["Technology"] = "other"
80
81          qpu_description = OrderedDict()
82          qpu_description['NumberOfQPUs'] = 1
83          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 42: 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.

4.3.2 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 Test Case** is stored in this field.

Listing 43 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 "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
```

```
42          """
43          program_language = platform.python_implementation()
44          return program_language
45
46      def my_programlanguage_version(**kwargs):
47          """
48          Getting the version of the programing language used for
    benchmark
49          """
50          language_version = platform.python_version()
51          return language_version
52
53      def my_programlanguage_vendor(**kwargs):
54          """
55          Getting the version of the programing language used for
    benchmark
56          """
57          language_vendor = "None"
58          return language_vendor
59
60      def my_api(**kwargs):
61          """
62          Collect the information about the used APIs
63          """
64          # api = OrderedDict()
65          # api["Name"] = "None"
66          # api["Version"] = "None"
67          # list_of_apis = [api]
68          modules = []
69          list_of_apis = []
70          for module in list(sys.modules):
71              api = OrderedDict()
72              module = module.split('.')[0]
73              if module not in modules:
74                  modules.append(module)
75                  api["Name"] = module
76                  try:
77                      version = sys.modules[module].__version__
78                  except AttributeError:
79                      #print("NO VERSION: "+str(sys.modules[module]))
80                      try:
81                          if  isinstance(sys.modules[module].version, str
    ):
82                              version = sys.modules[module].version
83                              #print("\t Attribute Version"+version)
84                          else:
85                              version = sys.modules[module].version()
86                              #print("\t Methdod Version"+version)
87                      except (AttributeError, TypeError) as error:
```

```
88                          #print('\t NO VERSION: '+str(sys.modules[module
   ]))
89                          try:
90                              version = sys.modules[module].VERSION
91                          except AttributeError:
92                              #print('\t\t NO VERSION: '+str(sys.modules[
   module]))
93                              version = "Unknown"
94                  api["Version"] = str(version)
95                  list_of_apis.append(api)
96          return list_of_apis
97
98      def my_quantum_compilation(**kwargs):
99          """
100         Information about the quantum compilation part of the benchmark
101         """
102         q_compilation = OrderedDict()
103         q_compilation["Step"] = "None"
104         q_compilation["Version"] = "None"
105         q_compilation["Flags"] = "None"
106         return [q_compilation]
107
108     def my_classical_compilation(**kwargs):
109         """
110         Information about the classical compilation part of the
   benchmark
111         """
112         c_compilation = OrderedDict()
113         c_compilation["Step"] = "None"
114         c_compilation["Version"] = "None"
115         c_compilation["Flags"] = "None"
116         return [c_compilation]
117
118     def my_metadata_info(**kwargs):
119         """
120         Other important info user want to store in the final json.
121         """
122
123         metadata = OrderedDict()
124         #metadata["None"] = None
125
126         return metadata
127
128
129     def my_benchmark_info(**kwargs):
130         """
131         Complete WorkFlow for getting all the benchmar informated
   related info
132         """
```

```
133         benchmark = OrderedDict()
134         benchmark["BenchmarkKernel"] = my_benchmark_kernel(**kwargs)
135         benchmark["StartTime"] = my_starttime(**kwargs)
136         benchmark["EndTime"] = my_endtime(**kwargs)
137         benchmark["ProgramLanguage"] = my_programlanguage(**kwargs)
138         benchmark["ProgramLanguageVersion"] =
    my_programlanguage_version(**kwargs)
139         benchmark["ProgramLanguageVendor"] = my_programlanguage_vendor
    (**kwargs)
140         benchmark["API"] = my_api(**kwargs)
141         benchmark["QuantumCompililation"] = my_quantum_compilation(**
    kwargs)
142         benchmark["ClassicalCompiler"] = my_classical_compilation(**
    kwargs)
143         benchmark["TimeMethod"] = my_timemethod(**kwargs)
144         benchmark["Results"] = summarize_results(**kwargs)
145         benchmark["MetaData"] = my_metadata_info(**kwargs)
146         return benchmark
```

Listing 43: 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 43). In order 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 4.3.3).

4.3.3 my_benchmark_summary.py

In this script, the *summarize_results* function is implemented. This function formats the results of a complete execution of a **Benchmark Test Case** of the **QPE kernel** with the provided 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 44 shows an example of implementation of *summarize_results* function for the **QPE** benchmark procedure.

```
1
2     def summarize_results(**kwargs):
3         """
4         Mandatory code for properly present the benchmark results
    following
5         the jsonschema
6         """
7
```

```
8         # n_qbits = [4]
9         # #Info with the benchmark results like a csv or a DataFrame
10        # pdf = None
11        # #Metrics needed for reporting. Depend on the benchmark kernel
12        # list_of_metrics = ["MRSE"]
13
14        import pandas as pd
15        benchmark_file = kwargs.get("benchmark_file", None)
16        pdf = pd.read_csv(benchmark_file, header=[0, 1], index_col=[0,
    1, 2])
17        pdf.reset_index(inplace=True)
18        n_qbits = list(set(pdf["n_qbits"]))
19        angle_methods = list(set(pdf["angle_method"]))
20        aux_qbits = list(set(pdf["aux_qbits"]))
21        list_of_metrics = [
22            "KS", "fidelity",
23        ]
24
25        results = []
26        #If several qbits are tested
27        # For ordering by n_qbits
28        for n_ in n_qbits:
29            # For ordering by auxiliar qbits
30            for aux_ in aux_qbits:
31                for angle_ in angle_methods:
32                    result = OrderedDict()
33                    result["NumberOfQubits"] = n_
34                    result["QubitPlacement"] = list(range(n_))
35                    result["QPUs"] = [1]
36                    result["CPUs"] = psutil.Process().cpu_affinity()
37                    #Select the proper data
38                    indice = (pdf['n_qbits'] == n_) & (pdf['aux_qbits']
     == aux_) \
39                        & (pdf['angle_method'] == angle_)
40                    step_pdf = pdf[indice]
41                    result["TotalTime"] = step_pdf["elapsed_time"]["
    mean"].iloc[0]
42                    result["SigmaTotalTime"] = step_pdf["elapsed_time"
    ]["std"].iloc[0]
43                    result["QuantumTime"] = step_pdf["quantum_time"]["
    mean"].iloc[0]
44                    result["SigmaQuantumTime"] = step_pdf["quantum_time
    "]["std"].iloc[0]
45                    result["ClassicalTime"] = step_pdf["classic_time"][
    "mean"].iloc[0]
46                    result["SigmaClassicalTime"] = step_pdf["
    classic_time"]["std"].iloc[0]
47
48                    # For identifying the test
```

```
49                         result['AuxiliarNumberOfQubits'] = aux_
50                         result['MethodForSettingAngles'] = angle_
51                         result['QPEAnglePrecision'] = \
52                             step_pdf['delta_theta']['mean'].iloc[0]
53                         metrics = []
54                         #For each fixed number of qbits several metrics can
     be reported
55                         for metric_name in list_of_metrics:
56                             metric = OrderedDict()
57                             #MANDATORY
58                             metric["Metric"] = metric_name
59                             metric["Value"] = step_pdf[metric_name]["mean"
    ].iloc[0]
60                             metric["STD"] = step_pdf[metric_name]["std"].
    iloc[0]
61                             metric["COUNT"] = int(step_pdf[metric_name]["
    count"].iloc[0])
62                             metrics.append(metric)
63                         result["Metrics"] = metrics
64                 results.append(result)
65         return results
```

Listing 44: 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 Test Case** execution were stored.

4.3.4 benchmark.py

The *benchmark.py* script can be used directly for gathering all the **Benchmark Test Case** execution information and results, for creating the final mandatory benchmark report.

It is not necessary to 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 Test Case** execution was stored.
- *benchmark_file*: complete path where the file with the summary results of the **Benchmark Test Case** execution was stored.

CESGA
GALICIA SUPERCOMPUTING CENTER

Bibliography

[1] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. AMS Contemporary Mathematics Series, 305, 2000. doi: 10.1090/conm/305/05215.

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

[3] Gonzalo Ferro, Alberto Manzano, Andrés Gómez, Alvaro Leitao, María R. Nogueiras, and Carlos Vázque. D5.4: Evaluation of quantum algorithms for pricing and computation of var, 2022.

[4] Dmitry Grinko, Julien Gacon, Christa Zoufal, and Stefan Woerner. Iterative quantum amplitude estimation. npj Quantum Information, 7(1), 2021. doi: 10.1038/s41534-021-00379-1.

[5] Lov Grover and Terry Rudolph. Creating superpositions that correspond to efficiently integrable probability distributions. arXiv e-prints, 2002. doi: 10.48550/arXiv.quant-ph/0208112.

[6] Andrés Gómez, Alvaro Leitao Rodriguez, Alberto Manzano, Maria Nogueiras, Gustavo Ordóñez, and Carlos Vázquez. A survey on quantum computational finance for derivatives pricing and var. Archives of Computational Methods in Engineering, 03 2022. doi: 10.1007/s11831-022-09732-9.

[7] A.Y. Kitaev. Quantum measurements and the abelian stabilizer problem. Electron. Colloquium Comput. Complex., TR96, 1995.

[8] A. Manzano, D. Musso, and Á. Leitao. Iterative quantum amplitude estimation. EPJ Quantum Technology, 10, 2023. doi: https://doi.org/10.1140/epjqt/s40507-023-00159-0.

[9] María Nogueiras, Gustavo Ordónez Sanz, Carlos Vázquez Cendón, Alvaro Leitao Rodríguez, Alberto Manzano Herrero, Daniele Musso, and Andrés Gómez. D5.1: Review of state-of-the-art forpricing and computation of var, 2021.

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

[11] Yohichi Suzuki, Shumpei Uno, Rudy Raymond, Tomoki Tanaka, Tamiya Onodera, and Naoki Yamamoto. Amplitude estimation without phase estimation. Quantum Information Processing, 19(2), 2020. doi: 10.1007/s11128-019-2565-2.