PyQBench: a Python library for benchmarking gate-based quantum computers

Konrad Jałowiecki*, Paulina Lewandowska, Łukasz Pawela

Institute of Theoretical and Applied Informatics, Polish Academy of Sciences, Bałtycka 5, 44-100 Gliwice, Poland

Abstract

We introduce PyQBench, an innovative open-source framework for benchmarking gate-based quantum computers. PyQBench can benchmark NISQ devices by verifying their capability of discriminating between two von Neumann measurements. PyQBench offers a simplified, ready-to-use, command line interface (CLI) for running benchmarks using a predefined parametrized Fourier family of measurements. For more advanced scenarios, PyQBench offers a way of employing user-defined measurements instead of predefined ones.

Keywords: Quantum computing, Benchmarking quantum computers, Discrimination of quantum measurements, Discrimination of von Neumann measurements, Open-source, Python programming
PACS: 03.67.-a, 03.67.Lx
2000 MSC: 81P68

*Corresponding author
Email address: `dexter2206@gmail.com` (Konrad Jałowiecki)

Current code version

C1	Current code version	0.1.1
C2	Permanent link to code/repository used for this code version	`https://github.com/iitis/PyQBench`
C3	Code Ocean compute capsule	`https://codeocean.com/capsule/89088992-9a27-4712-8525-d92a9b23060f/tree`
C4	Legal Code License	Apache License 2.0
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Python, Qiskit, AWS Braket

C7	Compilation requirements, operating environments & dependencies	`Python >= 3.8`
		`numpy ~= 1.22.0`
		`scipy ~= 1.7.0`
		`pandas ~= 1.5.0`
		`amazon-braket-sdk >= 1.11.1`
		`pydantic ~= 1.9.1`
		`qiskit ~= 0.37.2`
		`mthree ~= 1.1.0`
		`tqdm ~= 4.64.1`
		`pyyaml ~= 6.0`
		`qiskit-braket-provider ~= 0.0.3`

C8	If available Link to developer documentation/manual	`https://pyqbench.readthedocs.io/en/latest/`
C9	Support email for questions	`dexter2206@gmail.com`

Table 1: Code metadata

1. Motivation and significance

Noisy Intermediate-Scale Quantum (NISQ) [1] devices are storming the market, with a wide selection of devices based on different architectures and accompanying software solutions. Among hardware providers offering public access to their gate–based devices, one could mention Rigetti [2], IBM [3], Oxford Quantum Group [4], IonQ [5] or Xanadu [6]. Other vendors offer devices operating in different paradigms. Notably, one could mention D-Wave [7] and their quantum annealers, or QuEra devices [8] based on neural atoms. Most vendors provide their own software stack and application programming interface for accessing their devices. To name a few, Rigetti's computers are available through their Forest SDK [9] and PyQuil library [10] and IBM Q [3] computers can be accessed through Qiskit [11] or IBM Quantum Experience web interface [12]. Some cloud services, like Amazon Braket [13], offer access to several quantum devices under a unified API. On top of that, several libraries and frameworks can integrate with multiple hardware vendors. Examples of such frameworks include IBM Q's Qiskit or Zapata Computing's Orquestra [14].

It is well known that NISQ devices have their limitations [15]. The question is to what extent those devices can perform meaningful computations? To answer this question, one has to devise a methodology for benchmarking them. For gate–based computers, on which this paper focuses, there already exist several approaches. One could mention randomized benchmarking [16, 17, 18, 19, 20], benchmarks based on the quantum volume [21, 22, 23].

In this paper, we introduce a different approach to benchmarking gate–based devices with a simple operational interpretation. In our method, we test how well the given device is at guessing which of the two known von Neumann measurements were performed during the experiment. We implemented our approach in an open-source Python library called PyQBench. The library supports any device available through the Qiskit library, and thus can be used with providers such as IBM Q or Amazon Braket. Along with the library, the PyQBench package contains a command line tool for running most common benchmarking scenarios.

2. Existing benchmarking methodologies and software

Unsurprisingly, PyQBench is not the only software package for benchmarking gate–based devices. While we believe that our approach has significant benefits over other benchmarking techniques, for completeness, in this section we discuss some of the currently available similar software.

Probably the simplest benchmarking method one could devise is simply running known algorithms and comparing outputs with the expected ones. Analyzing the frequency of the correct outputs, or the deviation between actual and expected outputs distribution provides then a metric of the performance of a given device. Libraries such as Munich Quantum Toolkit (MQT) [24, 25] or SupermarQ [26, 27] contain benchmarks leveraging multiple algorithms, such as Shor's algorithm or Grover's algorithm. Despite being intuitive and easily interpretable, such benchmarks may have some problems. Most importantly, they assess the usefulness of a quantum device only for a very particular algorithm, and it might be hard to extrapolate their results to other algorithms and applications. For instance, the inability of a device to consistently find factorizations using Shor's algorithms does not tell anything about its usefulness in Variational Quantum Algorithm's.

Another possible approach to benchmarking quantum computers is randomized benchmarking. In this approach, one samples circuits to be run from some predefined set of gates (e.g. from the Clifford group) and tests how much the output distribution obtained from the device running these circuits differs from the ideal one. It is also common to concatenate randomly chosen circuits with their inverses (which should yield the identity circuit) and run those concatenated circuits on the device. Libraries implementing this approach include Qiskit [28] or PyQuil [29].

Another quantity used for benchmarking NISQ devices is quantum volume. The quantum volume characterizes capacity of a device for solving computational problems. It takes into account multiple factors like number of qubits, connectivity and measurement errors. The Qiskit library allows one to measure quantum volume of a device by using its `qiskit.ignis.verification.quantum_volume`. Other implementations of Quantum Volume can be found as well, see e.g. [30].

3. Preliminaries and discrimination scheme approach

In this section we describe how the benchmarking process in PyQBench works. To do so, we first discuss necessary mathematical preliminaries. Then, we present the general form of the discrimination scheme used in PyQBench and practical considerations on how to implement it taking into account limitations of the current NISQ devices.

3.1. Mathematical preliminaries

Let us first recall the definition of a von Neumann measurement, which is the only type of measurement used in PyQBench. A von Neumann measurement $\mathcal{P}$ is a collection of rank–one projectors $\{|u_0\rangle\langle u_0|, \ldots, |u_{d-1}\rangle\langle u_{d-1}|\}$, called effects, that sum up to identity, i.e. $\sum_{i=0}^{d-1} |u_i\rangle\langle u_i| = \mathbb{1}$. If U is a unitary matrix of size d, one can construct a von Neumann measurement $\mathcal{P}_U$ by taking projectors onto its columns. In this case we say that $\mathcal{P}_U$ is described by the matrix U.

Typically, NISQ devices can only perform measurements in computational Z-basis, i.e. $U = \mathbb{1}$. To implement an arbitrary von Neumann measurement $\mathcal{P}_U$, one has to first apply $U^\dagger$ to the measured system and then follow with Z-basis measurement. This process, depicted in Fig. 1, can be viewed as performing a change of basis in which measurement is performed prior to measurement in the computational basis.

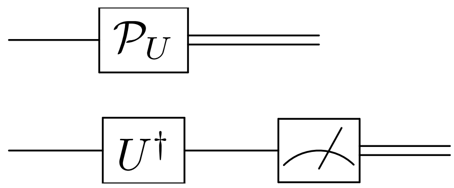

Figure 1: Implementation of a von Neumann measurement using measurement in computational basis. The upper circuit shows a symbolic representation of a von Neumann measurement $\mathcal{P}_U$. The bottom, equivalent circuit depicts its decomposition into a change of basis followed by measurement in the Z basis.

3.2. Discrimination scheme

Benchmarks in PyQBench work by experimentally determining the probability of correct discrimination between two von Neumann measurements by the device under test and comparing the result with the ideal, theoretical predictions.

Without loss of generality[1], we consider discrimination task between single qubit measurements $\mathcal{P}_{\mathbf{1}}$, performed in the computational Z-basis, and an alternative measurement $\mathcal{P}_U$ performed in the basis U. Note, however, that the discrimination scheme described below can work regardless of dimensionality of the system, see [31] for details.

In general, the discrimination scheme presented in Fig. 2, requires an auxiliary qubit. First, the joint system is prepared in some state $|\psi_0\rangle$. Then, one of the measurements, either $\mathcal{P}_U$ or $\mathcal{P}_{\mathbf{1}}$, is performed on the first part of the system. Based on its outcome i, we choose another POVM $\mathcal{P}_{V_i}$ and perform it on the second qubit, obtaining the output in j. Finally, if $j = 0$, we say that the performed measurement is $\mathcal{P}_U$, otherwise we say that it was $\mathcal{P}_{\mathbf{1}}$. Naturally, we need to repeat the same procedure multiple times for both measurements to obtain a reliable estimate of the underlying probability distribution. In PyQBench, we assume that the experiment is repeated the same number of times for both $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$.

Unsurprisingly, both the $|\psi_0\rangle$ and the final measurements $\mathcal{P}_{V_i}$ have to be chosen specifically for given U to maximize the probability of a correct guess. The detailed description how these choices are made in [32], and for now we will focus only how this scheme can be implemented on the actual devices, assuming that all the components are known.

[1] Explaining why we can consider only discrimination scheme between $\mathcal{P}_{\mathbf{1}}$ and $\mathcal{P}_U$ is beyond the scope of this paper. See [31] for a in depth explanation.

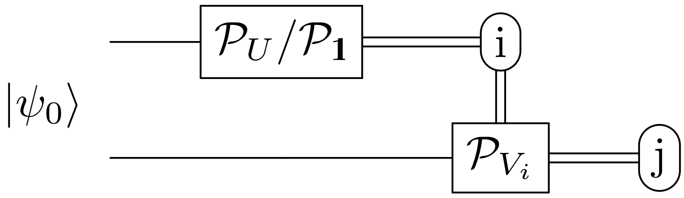

Figure 2: Theoretical scheme of discrimination between von Neumann measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$.

3.2.1. Implementation of discrimination scheme on actual NISQ devices

Current NISQ devices are unable to perform conditional measurements, which is the biggest obstacle to implementing our scheme on real hardware. However, we circumvent this problem by slightly adjusting our scheme so that it only uses components available on current devices. For this purpose, we use two possible options: using a postselection or a direct sum $V_0^\dagger \oplus V_1^\dagger$.

Scheme 1. (Postselection)

The first idea uses a postselection scheme. In the original scheme, we measure the first qubit and only then determine which measurement should be performed on the second one. Instead of doing this choice, we can run two circuits, one with $\mathcal{P}_{V_0}$ and one with $\mathcal{P}_{V_1}$ and measure both qubits. We then discard the results of the circuit for which label i does not match measurement label k. Hence, the circuit for postselection looks as depicted in Fig. 3.

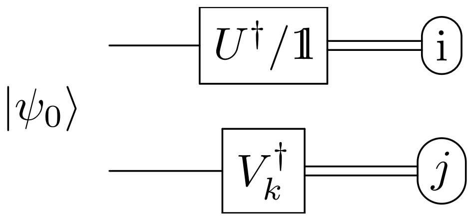

Figure 3: A schematic representation of the setup for distinguishing measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ using postselection approach. In postselection scheme, one runs such circuits for both $k = 0, 1$ and discards results for cases when there is a mismatch between k and i.

To perform the benchmark, one needs to run multiple copies of the postselection circuit, with both $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$. Each circuit has to be run in both variants, one with final measurement $\mathcal{P}_{V_0}$ and the second with the final measurement $\mathcal{P}_{V_1}$. The experiments can thus be grouped into classes identified by tuples of the form $(\mathcal{Q}, k, i, j)$, where $\mathcal{Q} \in \{\mathcal{P}_U, \mathcal{P}_{\mathbf{1}}\}$ denotes the chosen measurement, $k \in \{0, 1\}$ designates the final measurement used, and $i \in \{0, 1\}$ and $j \in \{0, 1\}$ being the labels of outcomes as presented in Fig. 3. We then discard all the experiments for which $i \neq k$. The total number of valid experiments is thus:

$$N_{\text{total}} = \#\{(\mathcal{Q}, k, i, j) : k = i\}. \tag{1}$$

Finally, we count the valid experiments resulting in successful discrimination. If we have chosen $\mathcal{P}_U$, then we guess correctly iff $j = 0$. Similarly, for $P_{\mathbf{1}}$, we guess correctly iff $j = 1$. If we define

$$\begin{aligned} N_{\mathcal{P}_U} &= \#\{(\mathcal{Q}, k, i, j) : \mathcal{Q} = \mathcal{P}_U, k = i, j = 0\}, & (2)\\ N_{\mathcal{P}_{\mathbf{1}}} &= \#\{(\mathcal{Q}, k, i, j) : \mathcal{Q} = \mathcal{P}_{\mathbf{1}}, k = i, j = 1\}, & (3) \end{aligned}$$

then the empirical success probability can be computed as

$$p_{\text{succ}}(\mathcal{P}_U, \mathcal{P}_{\mathbf{1}}) = \frac{N_{\mathcal{P}_U} + N_{\mathcal{P}_{\mathbf{1}}}}{N_{\text{total}}}. \tag{4}$$

The p_{succ} is the quantity reported to the user as the result of the benchmark.

Scheme 2. (Direct sum)

The second idea uses the direct sum $V_0^\dagger \oplus V_1^\dagger$ implementation. Here, instead of performing a conditional measurement $\mathcal{P}_{V_k}$, where $k \in \{0, 1\}$, we run circuits presented in Fig. 4.

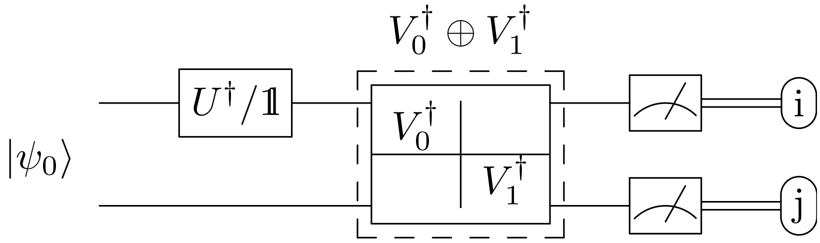

Figure 4: A schematic representation of the setup for distinguishing measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ using the $V_0^\dagger \oplus V_1^\dagger$ direct sum.

One can see why such a circuit is equivalent to the original discrimination scheme. If we rewrite the block-diagonal matrix $V_0^\dagger \oplus V_1^\dagger$ as follows:

$$V_0^\dagger \oplus V_1^\dagger = |0\rangle\langle 0| \otimes V_0^\dagger + |1\rangle\langle 1| \otimes V_1^\dagger, \tag{5}$$

we can see that the direct sum in Eq. (5) commutes with the measurement on the first qubit. Thanks to this, we can switch the order of operations to obtain the circuit from Fig. 5. Now, depending on the outcome i, one of the

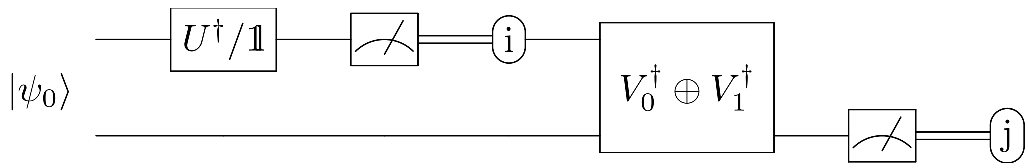

Figure 5: Rewritten representation of the setup for distinguishing measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ using the $V_0^\dagger \oplus V_1^\dagger$ direct sum.

summands in Eq. (5) vanishes, and we end up performing exactly the same operations as in the original scheme.

In this scheme, the experiment can be characterized by a pair $(\mathcal{Q}, i, j)$, where $\mathcal{Q} = \{\mathcal{P}_U, \mathcal{P}_{\mathbf{1}}\}$ and $i, j \in \{0, 1\}$ are the output labels. The number of successful trials for U and $\mathbb{1}$, respectively, can be written as

$$N_{\mathcal{P}_U} = \#\{(\mathcal{Q}, i, j) : \mathcal{Q} = \mathcal{P}_U, j = 0\}, \tag{6}$$
$$N_{\mathcal{P}_{\mathbf{1}}} = \#\{(\mathcal{Q}, i, j) : \mathcal{Q} = \mathcal{P}_{\mathbf{1}}, j = 1\}. \tag{7}$$

Then, the probability of correct discrimination between $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ is given by

$$p_{\text{succ}} = \frac{N_{\mathcal{P}_U} + N_{\mathcal{P}_{\mathbf{1}}}}{N_{\text{total}}}, \tag{8}$$

where N_{total} is the number of trials.

3.2.2. Importance of choosing the optimal discrimination scheme

In principle, the schemes described in the previous section could be used with any choice of $|\psi_0\rangle$ and final measurements $\mathcal{P}_{V_i}$. However, we argue that it is best to choose those components in such a way that they maximize the probability of correct discrimination. To see that, suppose that some choice of $|\psi_0\rangle, \mathcal{P}_{V_0}, \mathcal{P}_{V_1}$ yields the theoretical upper bound of discriminating between two measurements of one, i.e. on a perfect quantum computer you will always make a correct guess. Then, on real hardware, we might obtain any empirical value in range $\left[\frac{1}{2}, 1\right]$. On the other hand, if we choose the components of our scheme such that the successful discrimination probability is only $\frac{3}{5}$, the possible range of empirically obtainable probabilities is only $\left[\frac{1}{2}, \frac{3}{5}\right]$. Hence, in the second case, the discrepancy between theoretical and empirical results will be less pronounced.

3.2.3. Constructing optimal discrimination scheme

To construct the optimal discrimination scheme, one starts by calculating the probability of correct discrimination. Using the celebrated result by

Helstrom [33], one finds that the optimal probability of correct discrimination between two quantum measurements, $\mathcal{P}$ and $\mathcal{Q}$, is

$$p_{\text{succ}}(\mathcal{P}, \mathcal{Q}) = \frac{1}{2} + \frac{1}{4}\|\mathcal{P} - \mathcal{Q}\|_\diamond, \tag{9}$$

where

$$\|\mathcal{P} - \mathcal{Q}\|_\diamond = \max_{\||\psi\rangle\|_1=1} \|\left((\mathcal{P} - \mathcal{Q}) \otimes \mathbb{1}\right)(|\psi\rangle\langle\psi|)\|_1. \tag{10}$$

The quantum state $|\psi_0\rangle$ maximizing the diamond norm above is called the *discriminator*, and can be computed e.g. using semidefinite programming (SDP) [32, 34]. Furthermore, using the proof of the Holevo-Helstrom theorem, it is possible to construct corresponding unitaries V_0, V_1 to create the optimal discrimination strategy. For brevity, we do not describe this procedure here. Instead, we refer the interested reader to [32].

4. Discrimination scheme for parameterized Fourier family and implementation

So far, we only discussed how the discrimination is performed assuming that all needed components $|\psi_0\rangle$, V_0, and V_1 are known. In this section, we provide a concrete example using parametrized Fourier family of measurements.

The parametrized Fourier family of measurements is defined as a set of the measurements $\{\mathcal{P}_{U_\phi} : \phi \in [0, 2\pi]\}$, where

$$U_\phi = H \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix} H^\dagger, \tag{11}$$

and H is the Hadamard matrix of dimension two. For each element of this set, the discriminator is a Bell state:

$$|\psi_0\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right). \tag{12}$$

Observe that $|\psi_0\rangle$ does not depend on the angle ϕ. However, the unitaries V_0, V_1 depend on ϕ and take the following form:

$$V_0 = \begin{pmatrix} i\sin\left(\frac{\pi-\phi}{4}\right) & -i\cos\left(\frac{\pi-\phi}{4}\right) \\ \cos\left(\frac{\pi-\phi}{4}\right) & \sin\left(\frac{\pi-\phi}{4}\right) \end{pmatrix}, \tag{13}$$

$$V_1 = \begin{pmatrix} -i\cos\left(\frac{\pi-\phi}{4}\right) & i\sin\left(\frac{\pi-\phi}{4}\right) \\ \sin\left(\frac{\pi-\phi}{4}\right) & \cos\left(\frac{\pi-\phi}{4}\right) \end{pmatrix}. \tag{14}$$

Finally, the theoretical probability of correct discrimination between von Neumann measurements $\mathcal{P}_{U_\phi}$ and $\mathcal{P}_{\mathbf{1}}$ is given by

$$p_{\text{succ}}(\mathcal{P}_{U_\phi}, \mathcal{P}_{\mathbf{1}}) = \frac{1}{2} + \frac{|1 - e^{i\phi}|}{4}. \tag{15}$$

We explore the construction of $|\psi_0\rangle$, V_0 and V_1 for parametrized Fourier family of measurements in Appendix C.

5. Software description

This section is divided into two parts. In Section 5.1 we describe functionalities of PyQBench package. Next, in Section 5.2, we give a general overview of the software architecture.

5.1. Software Functionalities

The PyQBench can be used in two modes: as a Python library and as a CLI script. When used as a library, PyQBench allows the customization of discrimination scheme. The user provides a unitary matrix U defining the measurement to be discriminated, the discriminator $|\psi_0\rangle$, and unitaries V_0 and V_1 describing the final measurement. The PyQBench library provides then the following functionalities.

1. Assembling circuits for both postselection and direct sum–based discrimination schemes.
2. Executing the whole benchmarking scenario on specified backend (either real hardware or software simulator).
3. Interpreting the obtained outputs in terms of discrimination probabilities.

Note that the execution of circuits by PyQBench is optional. Instead, the user might want to opt in for fine-grained control over the execution of the circuits. For instance, suppose the user wants to simulate the discrimination experiment on a noisy simulator. In such a case, they can define the necessary components and assemble the circuits using PyQBench. The circuits can then be altered, e.g. to add noise to particular gates, and then run using any Qiskit backend by the user. Finally, PyQBench can be used to interpret the measurements to obtain discrimination probability.

The PyQBench library also contains a readily available implementation of all necessary components needed to run discrimination experiments for parametrized Fourier family of measurements, defined previously in Section 4. However, if one only wishes to use this particular family of measurements in their benchmarks, then using PyQBench as a command line tool might be more straightforward. PyQBench's command line interface allows running the benchmarking process without writing Python code. The configuration of CLI is done by YAML [35] files describing the benchmark to be performed and the description of the backend on which the benchmark should be run. Notably, the YAML configuration files are reusable. The same benchmark can be used with different backends and vice versa.

The following section describes important architectural decisions taken when creating PyQBench, and how they affect the end-user experience.

5.2. Software Architecture

5.2.1. Overview of the software structure

As already described, PyQBench can be used both as a library and a CLI. Both functionalities are implemented as a part of `qbench` Python package. The exposed CLI tool is also named `qbench`. For brevity, we do not discuss the exact structure of the package here, and instead refer an interested reader to the source code available at GitHub [36] or at the reference manual [37].

PyQBench can be installed from official Python Package Index (PyPI) by running `pip install pyqbench`. In a properly configured Python environment the installation process should also make the `qbench` command available to the user without a need for further configuration.

5.2.2. Integration with hardware providers and software simulators

PyQBench is built around the Qiskit [11] ecosystem. Hence, both the CLI tool and the `qbench` library can use any Qiskit–compatible backend. This includes, IBM Q backends (available by default in Qiskit) and Amazon Braket devices and simulators (available through `qiskit-braket-provider` package [38, 39]).

When using PyQBench as library, instances of Qiskit backends can be passed to functions that expect them as parameters. However, in CLI mode, the user has to provide a YAML file describing the backend. An example of such file can be found in Section 6, and the detailed description of the expected format can be found at PyQBench's documentation.

5.2.3. Command Line Interface

The Command Line Interface (CLI) of PyQBench has nested structure. The general form of the CLI invocation is shown in listing 1.

Listing 1: Invocation of `qbench` script

```
qbench <benchmark-type> <command> <parameters>
```

Currently, PyQBench's CLI supports only one type of benchmark (discrimination of parametrized Fourier family of measurements), but we decided on structuring the CLI in a hierarchical fashion to allow for future extensions. Thus, the only accepted value of `<benchmark-type>` is `disc-fourier`. The `qbench disc-fourier` command has four subcommands:

- `benchmark`: run benchmarks. This creates either a result YAML file containing the measurements or an intermediate YAML file for asynchronous experiments.
- `status`: query status of experiments submitted for given benchmark. This command is only valid for asynchronous experiments.
- `resolve`: query the results of asynchronously submitted experiments and write the result YAML file. The output of this command is almost identical to the result obtained from synchronous experiments.
- `tabulate`: interpret the results of a benchmark and summarize them in the CSV file.

We present usage of each of the above commands later in section 6.

5.2.4. Asynchronous vs. synchronous execution

PyQBench's CLI can be used in synchronous and asynchronous modes. The mode of execution is defined in the YAML file describing the backend (see Section 6 for an example of this configuration). We decided to couple the mode of execution to the backend description because some backends cannot work in asynchronous mode.

When running `qbench disc-fourier benchmark` in asynchronous mode, the PyQBench submits all the circuits needed to perform a benchmark and then writes an intermediate YAML file containing metadata of submitted experiments. In particular, this metadata contains information on correlating submitted job identifiers with particular circuits. The intermediate file can be used to query the status of the submitted jobs or to resolve them, i.e. to wait for their completion and get the measurement outcomes.

In synchronous mode, PyQBench first submits all jobs required to run the benchmark and then immediately waits for their completion. The advantage of this approach is that no separate invocation of `qbench` command is needed to actually download the measurement outcomes. The downside, however, is that if the script is interrupted while the command is running, the intermediate results will be lost. Therefore, we recommend using asynchronous mode whenever possible.

6. Illustrative examples

In this section, we present two examples demonstrating the usage of PyQBench. In the first example, we show how to implement a discrimination scheme for a user–defined measurement and possible ways of using this scheme with `qbench` library. The second example demonstrates the usage of the CLI. We show how to prepare the input files for the benchmark and how to run it using the `qbench` tool.

6.1. Using user-defined measurement with `qbench` package

In this example, we will demonstrate how `qbench` package can be used with user–defined measurement. For this purpose, we will use $U = H$ (the Hadamard gate). The detailed calculations that lead to the particular form of the discriminator and final measurements can be found in Appendix B. The explicit formula for discriminator in this example reads:

$$|\psi_0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \tag{16}$$

with final measurements being equal to

$$V_0 = \begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix}, \tag{17}$$

and

$$V_1 = \begin{pmatrix} -\beta & \alpha \\ \alpha & \beta \end{pmatrix}, \tag{18}$$

where

$$\alpha = \frac{\sqrt{2-\sqrt{2}}}{2} = \cos\left(\frac{3}{8}\pi\right), \tag{19}$$

$$\beta = \frac{\sqrt{2+\sqrt{2}}}{2} = \sin\left(\frac{3}{8}\pi\right). \tag{20}$$

To use the above benchmarking scheme in PyQBench, we first need to construct circuits that can be executed by actual hardware. To this end, we need to represent each of the unitaries as a sequence of standard gates, keeping in mind that quantum circuits start execution from the $|00\rangle$ state. The circuit taking $|00\rangle$ to the Bell state $|\psi_0\rangle$ comprises the Hadamard gate followed by CNOT gate on both qubits (see Fig. 6). For V_0 and V_1 observe that $V_0 = \mathrm{RY}\left(\frac{3}{4}\pi\right)$, where RY is rotation gate around the Y axis defined by

$$\mathrm{RY}(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}. \tag{21}$$

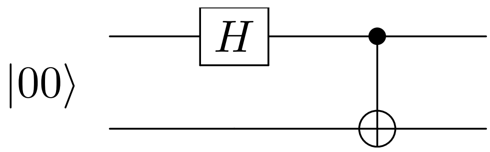

Figure 6: Decomposition of the Bell state $|\psi_0\rangle$.

To obtain V_1 we need only to swap the columns, i.e.

$$V_1 = \mathrm{RY}\left(\frac{3}{4}\pi\right)\mathrm{X}\,. \tag{22}$$

Finally, the optimal probability of correct discrimination is equal to

$$p_{\mathrm{succ}}(\mathcal{P}_U, \mathcal{P}_{\mathbf{1}}) = \frac{1}{2} + \frac{\sqrt{2}}{4}. \tag{23}$$

We will now demonstrate how to implement this theoretical scheme in PyQBench. For this example we will use the Qiskit Aer simulator [40]. First, we import the necessary functions and classes from PyQBench and Qiskit. We also import `numpy` for the definition of `np.pi` constant and the exponential function. The exact purpose of the imported functions will be described at the point of their usage.

Listing 2: Imports needed for running benchmarking example

```
import numpy as np
from qiskit import QuantumCircuit, Aer
from qbench.schemes.postselection import
    benchmark_using_postselection
from qbench.schemes.direct_sum import benchmark_using_direct_sum
```

To implement the discrimination scheme in PyQBench, we need to define all the necessary components as Qiskit instructions. We can do so by constructing a circuit object acting on qubits 0 and 1 and then converting them using `to_instruction()` method.

Listing 3: Defining components for Hadamard experiment

```
def state_prep():
    circuit = QuantumCircuit(2)
    circuit.h(0)
    circuit.cnot(0, 1)
    return circuit.to_instruction()

def u_dag():
```

```
    circuit = QuantumCircuit(1)
    circuit.h(0)
    return circuit.to_instruction()

def v0_dag():
    circuit = QuantumCircuit(1)
    circuit.ry(-np.pi * 3 / 4, 0)
    return circuit.to_instruction()

def v1_dag():
    circuit = QuantumCircuit(1)
    circuit.ry(-np.pi * 3 / 4, 0)
    circuit.x(0)
    return circuit.to_instruction()

def v0_v1_direct_sum_dag():
    circuit = QuantumCircuit(2)
    circuit.ry(-np.pi * 3 / 4, 0)
    circuit.cnot(0, 1)
    return circuit.to_instruction()
```

We now construct a backend object, which in this case is an instance of Aer simulator.

Listing 4: Defining a backend

```
simulator = Aer.get_backend("aer_simulator")
```

In the simplest scenario, when one does not want to tweak execution details and simply wishes to run the experiment on a given backend, everything that is required is now to run `benchmark_using_postselection` or `benchmark_using_direct_sum` function, depending on the user preference.

Listing 5: Simulation benchmark by using postselection

```
postselection_result = benchmark_using_postselection(
    backend=simulator,
    target=0,
    ancilla=1,
    state_preparation=state_prep(),
    u_dag=u_dag(),
    v0_dag=v0_dag(),
    v1_dag=v1_dag(),
    num_shots_per_measurement=10000,
)
```

Listing 6: Simulation benchmark by using direct sum

```
direct_sum_result = benchmark_using_direct_sum(
   backend=simulator,
   target=1,
   ancilla=2,
   state_preparation=state_prep(),
   u_dag=u_dag(),
   v0_v1_direct_sum_dag=v0_v1_direct_sum_dag(),
   num_shots_per_measurement=10000,
)
```

The postselection_result and direct_sum_result variables contain now the empirical probabilities of correct discrimination. We can compare them to the theoretical value and compute the absolute error.

Listing 7: Examining the benchmark results

```
p_succ = (2 + np.sqrt(2)) / 4
print(f"Analytical p_succ = {p_succ}")
print(
f"Postselection: p_succ = {postselection_result}, abs. error =
    {p_succ - postselection_result}"
)
print(f"Direct sum: p_succ = {direct_sum_result}, abs. error =
    {p_succ - direct_sum_result}")
```

```
Analytical p_succ = 0.8535533905932737
Postselection: p_succ = 0.8559797193791593, abs. error =
    -0.0024263287858855564
Direct sum: p_succ = 0.85605, abs. error = -0.0024966094067262468
```

In the example presented above we used functions that automate the whole process – from the circuit assembly, through running the simulations to interpreting the results. But what if we want more control over some parts of this process? One possibility would be to add some additional parameters to benchmark_using_xyz functions, but this approach is not scalable. Moreover, anticipating all possible uses cases isimpossible. Therefore, we decided on another approach. PyQBench provides functions performing:

1. Assembly of circuits needed for experiment, provided the components discussed above.

2. Interpretation of the obtained measurements.

The difference between the two approaches is illustrated on the diagrams in Fig. 7.

For the rest of this example we focus only on the postselection case, as the direct sum case is analogous. We continue by importing two more functions from PyQBench.

Listing 8: Assembling circuits

```
from qbench.schemes.postselection import (
    assemble_postselection_circuits,
    compute_probabilities_from_postselection_measurements,
)

circuits = assemble_postselection_circuits(
   target=0,
   ancilla=1,
   state_preparation=state_prep(),
   u_dag=u_dag(),
   v0_dag=v0_dag(),
   v1_dag=v1_dag(),
)
```

Recall that for a postselection scheme we have two possible choices of the "unknown" measurement and two possible choices of a final measurement, which gives a total of four circuits needed to run the benchmark. The function `assemble_postselection_circuits` creates all four circuits and places them in a dictionary with keys `"id_v0"`, `"id_v1"`, `"u_v0"`, `"u_v1"`.

We will now run our circuits using noisy and noiseless simulation. We start by creating a noise model using Qiskit.

Listing 9: Adding noise model

```
from qiskit.providers.aer import noise

error = noise.ReadoutError([[0.75, 0.25], [0.8, 0.2]])

noise_model = noise.NoiseModel()
noise_model.add_readout_error(error, [0])
noise_model.add_readout_error(error, [1])
```

Once we have our noise model ready, we can execute the circuits with and without noise. To this end, we will use Qiskit's `execute` function. One caveat is that we have to keep track which measurements correspond to

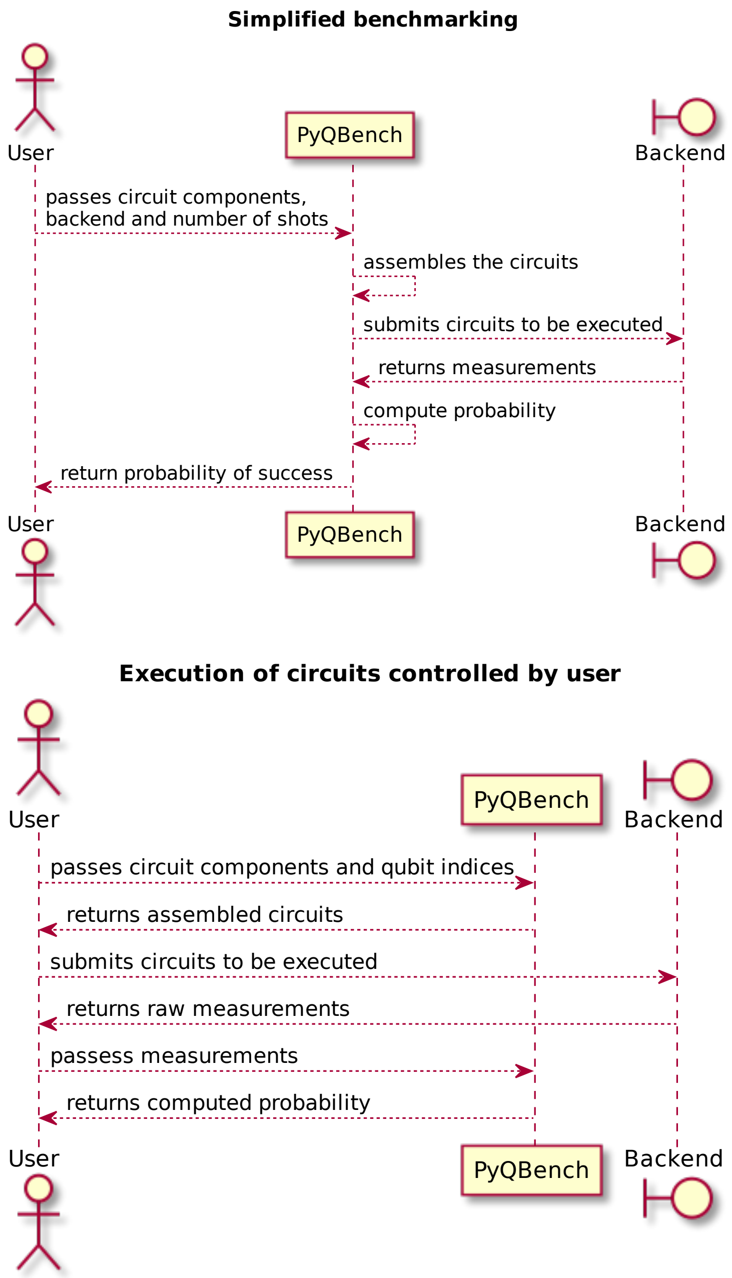

Figure 7: Differences between simplified (top) and user–controlled (bottom) execution of benchmarks in PyQBench. Compared to simplified benchmarking, in user-controlled benchmarks the user has direct access to the circuits being run, and hence can alter them (e.g. by adding noise) and/or choose the parameters used for executing them on the backend.

which circuit. We do so by fixing an ordering on the keys in the `circuits` dictionary.

Listing 10: Running circuits

```
from qiskit import execute

keys_ordering = ["id_v0", "id_v1", "u_v0", "u_v1"]
all_circuits = [circuits[key] for key in keys_ordering]

counts_noisy = execute(
   all_circuits,
   backend=simulator,
   noise_model=noise_model,
   shots=10000
).result().get_counts()

counts_noiseless = execute(
   all_circuits,
   backend=simulator,
   shots=10000
).result().get_counts()
```

Finally, we use the measurement counts to compute discrimination probabilities using `compute_probabilities_from_postselection_measurements` function.

Listing 11: Computation probabilities

```
prob_succ_noiseless =
    compute_probabilities_from_postselection_measurements(
   id_v0_counts=counts_noiseless[0],
   id_v1_counts=counts_noiseless[1],
   u_v0_counts=counts_noiseless[2],
   u_v1_counts=counts_noiseless[3],
)


prob_succ_noisy =
    compute_probabilities_from_postselection_measurements(
   id_v0_counts=counts_noisy[0],
   id_v1_counts=counts_noisy[1],
   u_v0_counts=counts_noisy[2],
   u_v1_counts=counts_noisy[3],
)
```

We can now examine the results. As an example, in one of our runs, we obtained `prob_succ_noiseless = 0.8524401115559386` and `prob_succ_noisy = 0.5017958400693446`. As expected, for noisy simulations, the result lies further away from the target value of `0.8535533905932737`.

This concludes our example. In the next section, we will show how to use PyQBench's CLI.

6.2. Using `qbench` CLI

Using PyQBench as a library allows one to conduct a two–qubits benchmark with arbitrary von Neumann measurement. However, as discussed in the previous guide, it requires writing some amount of code. For a Fourier parametrized family of measurements, PyQBench offers a simplified way of conducting benchmarks using a Command Line Interface (CLI). The workflow with PyQBench's CLI can be summarized as the following list of steps::

1. Preparing configuration files describing the backend and the experiment scenario.
2. Submitting/running experiments. Depending on the experiment scenario, execution can be synchronous, or asynchronous.
3. (optional) Checking the status of the submitted jobs if the execution is asynchronous.
4. Resolving asynchronous jobs into the actual measurement outcomes.
5. Converting obtained measurement outcomes into tabulated form.

6.2.1. Preparing configuration files

The configuration of PyQBench CLI is driven by YAML files. The first configuration file describes the experiment scenario to be executed. The second file describes the backend. Typically, this backend will correspond to the physical device to be benchmarked, but for testing purposes one might as well use any other Qiskit–compatible backend including simulators. Let us first describe the experiment configuration file, which might look as follow.

Listing 12: Defining the experiment

```
type: discrimination-fourier
qubits:
   - target: 0
     ancilla: 1
   - target: 1
      ancilla: 2
angles:
```

```
    start: 0
    stop: 2 * pi
    num_steps: 3
gateset: ibmq
method: direct_sum
num_shots: 100
```

The experiment file contains the following fields:

- `type`: a string describing the type of the experiment. Currently, the only option of `type` is `discrimination-fourier`.
- `qubits`: a list enumerating pairs of qubits on which the experiment should be run. For configuration in listing 12, the benchmark will run on two pairs of qubits. The first pair is 0 and 1, and the second one is 1 and 2. We decided to describe a pair by using `target` and `ancilla` keys rather than using a plain list to emphasize that the role of qubits in the experiment is not symmetric.
- `angles`: an object describing the range of angles for Fourier parameterized family. The described range is always uniform, starts at the `start`, ends at `stop` and contains `num_steps` points, including both `start` and `stop`. The `start` and `stop` can be arithmetic expressions using `pi` literal. For instance, the range defined in listing 12 contains three points: 0, π and 2π.
- `gateset`: a string describing the set of gates used in the decomposition of circuits in the experiment. The PyQBench contains explicit implementations of circuits The possible options are `[ibmq, lucy, rigetti]`, corresponding to decompositions compatible with IBM Q devices, OQC Lucy device, and Rigetti devices. Alternatively, one might wish to turn off the decomposition by using a special value `generic`. However, for this to work a backend used for the experiment must natively implement all the gates needed for the experiment, as described in 4.
- `method`: a string, either `postselection` or `direct_sum` determining which implementation of the conditional measurement is used.
- `num_shots`: an integer defines how many shots are performed in the experiment for a particular angle, qubit pair and circuit. Note that if one wishes to compute the total number of shots in the experiment, it is necessary to take into account that the `postselection` method uses twice as many circuits as the `direct_sum` method.

The second configuration file describes the backend. We decided to decouple the experiment and the backend files because it facilitates their reuse. For instance, the same experiment file can be used to run benchmarks on multiple backends, and the same backend description file can be used with multiple experiments.

Different Qiskit backends typically require different data for their initialization. Hence, there are multiple possible formats of the backend configuration files understood by PyQBench. We refer the interested reader to the PyQBench's documentation. Below we describe an example YAML file describing IBM Q backend named Quito.

Listing 13: IBMQ backend

```yaml
name: ibmq_quito
asynchronous: false
provider:
  hub: ibm-q
  group: open
  project: main
```

IBMQ backends typically require an access token to IBM Quantum Experience. Since it would be unsafe to store it in plain text, the token has to be configured separately in `IBMQ_TOKEN` environmental variable.

6.2.2. Remarks on using the asynchronous flag

For backends supporting asynchronous execution, the `asynchronous` setting can be configured to toggle it. For asynchronous execution to work, the following conditions have to be met:

- Jobs returned by the backend have unique `job_id`.
- Jobs are retrievable from the backend using the `backend.retrieve_job` method, even from another process (e.g. if the original process running the experiment has finished).

Since PyQBench cannot determine if the job retrieval works for a given backend, it is the user's responsibility to ensure that this is the case before setting `asynchronous` to `true`.

6.2.3. Running the experiment and collecting measurements data

After preparing YAML files defining experiment and backend, running the benchmark can be launched by using the following command line invocation:

```
qbench disc-fourier benchmark experiment_file.yml backend_file.yml
```

The output file will be printed to stdout. Optionally, the `- -output OUTPUT` parameter might be provided to write the output to the `OUTPUT` file instead.

```
qbench disc-fourier benchmark experiment_file.yml backend_file.yml
    --output async_results.yml
```

The result of running the above command can be twofold:

- If backend is asynchronous, the output will contain intermediate data containing, amongst others, `job_ids` correlated with the circuit they correspond to.
- If the backend is synchronous, the output will contain measurement outcomes (bitstrings) for each of the circuits run.

For synchronous experiment, the part of output looks similar to the one below. The whole YAML file can be seen in Appendix E.


```
data:
- target: 0
  ancilla: 1
  phi: 0.0
  results_per_circuit:
  - name: id
  histogram: {'00': 28, '01': 26, '10': 21, '11': 25}
  mitigation_info:
   target: {prob_meas0_prep1: 0.052200000000000024,
       prob_meas1_prep0: 0.0172}
   ancilla: {prob_meas0_prep1: 0.05900000000000005,
       prob_meas1_prep0: 0.0202}
  mitigated_histogram: {'00': 0.2637212373658018, '01':
      0.25865061319892463, '10': 0.2067279352110304, '11':
      0.2709002142242433}
```


The data includes `target, ancilla, phi`, and `results_per_circuit`. The first three pieces of information have already been described. The last data `results_per_circuit` gives us the following additional information:

- `name`: the information which measurement is used during experiment, either string `"u"` for $\mathcal{P}_U$ or string `"id"` for $\mathcal{P}_{\mathbf{1}}$. In this example we consider $\mathcal{P}_{\mathbf{1}}$.
- `histogram`: the dictionary with measurements' outcomes. The keys represent possible bitstrings, whereas the values are the number of occurrences.

- `mitigation_info`: for some backends (notably for backends corresponding to IBM Q devices), `backends.properties().qubits` contains information that might be used for error mitigation using the MThree method [41, 42]. If this info is available, it will be stored in the `mitigation_info` field, otherwise this field will be absent.

- `mitigated_histogram`: the histogram with measurements' outcomes after the error mitigation.

6.2.4. (Optional) Getting status of asynchronous jobs

PyQBench provides also a helper command that will fetch the statuses of asynchronous jobs. The command is:

```
qbench disc-fourier status async_results.yml
```

and it will display dictionary with histogram of statuses.

6.2.5. Resolving asynchronous jobs

For asynchronous experiments, the stored intermediate data has to be resolved in actual measurements' outcomes. The following command will wait until all jobs are completed and then write a result file.

```
qbench disc-fourier resolve async-results.yml resolved.yml
```

The resolved results, stored in `resolved.yml`, would look just like if the experiment was run synchronously. Therefore, the final results will look the same no matter in which mode the benchmark was run, and hence in both cases the final output file is suitable for being an input for the command computing the discrimination probabilities.

6.2.6. Computing probabilities

As a last step in the processing workflow, the results file has to be passed to `tabulate` command:

```
qbench disc-fourier tabulate results.yml results.csv
```

A sample CSV file is provided below:

7. Impact

With the surge of availability of quantum computing architectures in recent years it becomes increasingly difficult to keep track of their relative performance. To make this case even more difficult, various providers give

target	ancilla	phi	ideal_prob	disc_prob	mit_disc_prob
0	1	0	0.5	0.46	0.45
0	1	3.14	1	0.95	0.98
0	1	6.28	0.5	0.57	0.58
1	2	0	0.5	0.57	0.57
1	2	3.14	1	0.88	0.94
1	2	6.28	0.5	0.55	0.56

Table 2: The resulting CSV file contains table with columns `target`, `ancilla`, `phi`, `ideal_prob`, `disc_prob` and, optionally, `mit_disc_prob`. Each row in the table describes results for a tuple of `(target, ancilla, phi)`. The reference optimal value of discrimination probability is present in `ideal_prob` column, whereas the obtained, empirical discrimination probability can be found in the `disc_prob` column. The `mit_disc_prob` column contains empirical discrimination probability after applying the `Mthree` error mitigation [41, 42], if it was applied.

access to different figures of merit for their architectures. Our package allows the user to test various architectures, available through `qiskit` and Amazon BraKet using problems with simple operational interpretation. We provide one example built-in in the package. Furthermore, we provide a powerful tool for the users to extend the range of available problems in a way that suits their needs.

Due to this possibility of extension, the users are able to test specific aspects of their architecture of interest. For example, if their problem is related to the amount of coherence (the sum of absolute value of off-diagonal elements) of the states present during computation, they are able to quickly prepare a custom experiment, launch it on desired architectures, gather the result, based on which they can decide which specific architecture they should use.

Finally, we provide the source code of PyQBench on GitHub [36] under an open source license which will allow users to utilize and extend our package in their specific applications.

8. Conclusions

In this study, we develop a Python library PyQBench, an innovative open-source framework for benchmarking gate-based quantum computers.

PyQBench can benchmark NISQ devices by verifying their capability of discriminating between two von Neumann measurements. PyQBench offers a simplified, ready-to-use, command line interface (CLI) for running benchmarks using a predefined parameterized Fourier family of measurements. For more advanced scenarios, PyQBench offers a way of employing user-defined measurements instead of predefined ones.

9. Conflict of Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its out- come.

Acknowledgements

This work is supported by the project “Near-term quantum computers Challenges, optimal implementations and applications” under Grant Number POIR.04.04.00-00-17C1/18-00, which is carried out within the Team-Net programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund. PL is also a holder of European Union scholarship through the European Social Fund, grant InterPOWER (POWR.03.05.00-00-Z305).

References

[1] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[2] Rigetti computing. `https://www.rigetti.com/`. Accessed on 2023-02-18.

[3] IBM Quantum. `https://www.ibm.com/quantum`. Accessed on 2023-02-18.

[4] Oxford quantum. `http://oxfordquantum.org/`. Accessed on 2023-02-18.

[5] IonQ. `https://ionq.com/`. Accessed on 2023-02-18.

[6] Xanadu. `https://www.xanadu.ai/`. Accessed on 2023-02-18.

[7] D-Wave Systems. `https://www.dwavesys.com/`. Accessed on 2023-02-18.

[8] QuEra. `https://www.quera.com/`. Accessed on 2023-02-18.

[9] QCS Documentation. `https://docs.rigetti.com/qcs/`. Accessed on 2023-02-18.

[10] PyQuil Documentation. `https://pyquil-docs.rigetti.com/en/stable/`. Accessed on 2023-02-18.

[11] Qiskit. `https://qiskit.org/`. Accessed on 2023-02-18.

[12] IBM Quantum Experience. `https://quantum-computing.ibm.com/`. Accessed on 2023-02-18.

[13] Amazon Braket. `https://aws.amazon.com/braket/`. Accessed on 2023-02-18.

[14] Zapata Orquestra Platform. `https://www.zapatacomputing.com/orquestra-platform/`. Accessed on 2023-02-18.

[15] J. Preskill. Quantum computing 40 years later. *arXiv preprint arXiv:2106.10522*, 2021.

[16] Y. Liu, M. Otten, R. Bassirianjahromi, L. Jiang, and B. Fefferman. Benchmarking near-term quantum computers via random circuit sampling. *arXiv preprint arXiv:2105.05232*, 2021.

[17] E. Knill, D. Leibfried, R. Reichle, J. Britton, R. B. Blakestad, J. D. Jost, C. Langer, R. Ozeri, S. Seidelin, and D. J. Wineland. Randomized benchmarking of quantum gates. *Physical Review A*, 77(1):012307, 2008.

[18] J. J. Wallman and S. T. Flammia. Randomized benchmarking with confidence. *New Journal of Physics*, 16(10):103032, 2014.

[19] J. Helsen, I. Roth, E. Onorati, A. H. Werner, and J. Eisert. General framework for randomized benchmarking. *PRX Quantum*, 3(2):020357, 2022.

[20] A. Cornelissen, J. Bausch, and A. Gilyén. Scalable benchmarks for gate-based quantum computers. *arXiv preprint arXiv:2104.10698*, 2021.

[21] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3):032328, 2019.

[22] N. Moll, P. Barkoutsos, L. S. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn, A. Kandala, A. Mezzacapo, P. Müller, W. Riess, G. Salis, J. Smolin, I. Tavernelli, and K. Temme. Quantum optimization using variational algorithms on near-term quantum devices. *Quantum Science and Technology*, 3(3):030503, 2018.

[23] E. Pelofske, A. Bärtschi, and S. Eidenbenz. Quantum volume in practice: What users can expect from nisq devices. *IEEE Transactions on Quantum Engineering*, 3:1–19, 2022.

[24] N. Quetschlich, L. Burgholzer, and R. Wille. Mqt bench: Benchmarking software and design automation tools for quantum computing. *arXiv preprint arXiv:2204.13719*, 2022.

[25] MQTBench. `https://github.com/cda-tum/MQTBench`. Accessed on 2023-02-18.

[26] T. Tomesh, P. Gokhale, V. Omole, G. S. Ravi, K. N. Smith, J. Viszlai, X. Wu, N. Hardavellas, M. R. Martonosi, and F. T. Chong. SupermarQ: A scalable quantum benchmark suite. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 587–603. IEEE, 2022.

[27] SupermarQ. `https://github.com/SupertechLabs/SupermarQ`. Accessed on 2023-02-18.

[28] Qiskit benchmarks. `https://github.com/qiskit-community/qiskit-benchmarks`. Accessed on 2023-02-18.

[29] Forest Benchmarking: QCVV using PyQuil. `https://github.com/rigetti/forest-benchmarking`. Accessed on 2023-02-18.

[30] Quantum volume in practice. `https://github.com/lanl/Quantum-Volume-in-Practice`. Accessed on 2023-02-18.

[31] Z. Puchała, Ł. Pawela, A. Krawiec, and R. Kukulski. Strategies for optimal single-shot discrimination of quantum measurements. *Physical Review A*, 98(4):042103, 2018.

[32] J. Watrous. *The Theory of Quantum Information*. Cambridge University Press, 2018.

[33] C. W. Helstrom. Quantum detection and estimation theory. *Journal of Statistical Physics*, 1(2):231–252, 1969.

[34] J. Watrous. Simpler semidefinite programs for completely bounded norms. *Chicago Journal of Theoretical Computer Science*, 8, 2013.

[35] YAML Ain't Markup Language (YAML) Version 1.2. `https://yaml.org/spec/1.2.2/`. Accessed on 2023-02-18.

[36] PyQBench GitHub Repository. https://github.com/iitis/PyQBench. Accessed on 2023-02-18.

[37] PyQBench Documentation. https://pyqbench.readthedocs.io/en/latest/. Accessed on 2023-02-18.

[38] Introducing the Qiskit Provider for Amazon Braket. https://aws.amazon.com/blogs/quantum-computing/introducing-the-qiskit-provider-for-amazon-braket/. Accessed on 2023-02-18.

[39] Qiskit Braket Provider GitHub Repository. https://github.com/qiskit-community/qiskit-braket-provider. Accessed on 2023-02-18.

[40] Qiskit Aer GitHub Repository. https://github.com/Qiskit/qiskit-aer. Accessed on 2023-02-18.

[41] mthree Documentation. https://qiskit.org/documentation/partners/mthree/stubs/mthree.M3Mitigation.html. Accessed on 2023-02-10.

[42] P. D. Nation, H. Kang, N. Sundaresan, and J. M. Gambetta. Scalable mitigation of measurement errors on quantum computers. *PRX Quantum*, 2(4):040326, 2021.

[43] F. D. Murnaghan. On the field of values of a square matrix. *Proceedings of the National Academy of Sciences*, 18(3):246–248, 1932.

[44] Numerical shadow. https://numericalshadow.org/. Accessed on 2022-10-02.

[45] P. Lewandowska, A. Krawiec, R. Kukulski, Ł. Pawela, and Z. Puchała. On the optimal certification of von Neumann measurements. *Scientific Reports*, 11(1):3623, 2021.

[46] F. Hausdorff. Der wertvorrat einer bilinearform. *Mathematische Zeitschrift*, 3(1):314–316, 1919.

[47] O. Toeplitz. Das algebraische analogon zu einem satze von fejér. *Mathematische Zeitschrift*, 2(1-2):187–197, 1918.

Appendix A. Mathematical preliminaries

Let $\mathcal{M}_{d_1,d_2}$ be the set of all matrices of dimension $d_1 \times d_2$ over the field $\mathbb{C}$. For simplicity, square matrices will be denoted by $\mathcal{M}_d$. By Ω_d, we will denote the set of quantum states, that is positive semidefinite operators having trace equal to one. The subset of $\mathcal{M}_d$ consisting of unitary matrices will be denoted by $\mathcal{U}_d$, while its subgroup of diagonal unitary operators will be denoted by $\mathcal{DU}_d$.

We will also need a linear mapping transforming $\mathcal{M}_{d_1}$ into $\mathcal{M}_{d_2}$, which will be denoted

$$\Phi : \mathcal{M}_{d_1} \to \mathcal{M}_{d_2}. \tag{A.1}$$

There exists a bijection between the set of linear mappings Φ and the set of matrices $\mathcal{M}_{d_1 d_2}$, known as the Choi-Jamiołkowski isomorphism. For a given linear mapping Φ the corresponding Choi operator $J(\Phi)$ is explicitly written as

$$J(\Phi) \coloneqq \sum_{i,j=0}^{d-1} \Phi(|i\rangle\langle j|) \otimes |i\rangle\langle j|. \tag{A.2}$$

We also introduce a special subset of all mappings Φ, called quantum channels, which are completely positive and trace preserving (CPTP). In this work we will consider a special class of quantum channels, called unitary channels. A quantum channel Φ_U is said to be a unitary channel if it has the following form $\Phi_U(\cdot) = U \cdot U^\dagger$ for any $U \in \mathcal{U}_d$.

Let us recall a general form of a quantum measurement, so called Positive Operator Valued Measure (POVM). A POVM $\mathcal{P}$ is a collection of positive semidefinite operators $\{E_1, \ldots, E_m\}$, called effects, that sum up to the identity operator, *i.e.* $\sum_{i=1}^{m} E_i = \mathbb{1}$. In PyQBench, we are interested only in von Neumann measurements, that is measurements for which all the effects are rank-one projectors. Every such measurement can be parameterized by a unitary matrix $U \in \mathcal{U}_d$ with effects $\{|u_0\rangle\langle u_0|, \ldots, |u_{d-1}\rangle\langle u_{d-1}|\}$, are created by taking $|u_i\rangle$ as $(i+1)$-th column of the unitary matrix U. We will denote von Neumann measurements described by the matrix U by $\mathcal{P}_U$. The action of von Neumann measurement $\mathcal{P}_U$ on some state $\rho \in \Omega_d$ can be seen as a measure-and-prepare quantum channel as follows

$$\mathcal{P}_U : \rho \to \sum_{i=0}^{d-1} \langle u_i|\rho|u_i\rangle |i\rangle\langle i|. \tag{A.3}$$

Moreover, observe that each von Neumann measurement $\mathcal{P}_U$ poses a composition of a unitary channel $\Phi_{U^\dagger}$ and the maximally dephasing channel Δ, that means $\mathcal{P}_U = \Delta \circ \Phi_{U^\dagger}$.

We need to also briefly discuss about the distance between quantum operations. From [31, Theorem 1], the distance between measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ can be expressed in the notion of diamond norm, that is

$$\|\mathcal{P}_U - \mathcal{P}_{\mathbf{1}}\|_\diamond = \min_{E \in \mathcal{DU}_d} \|\Phi_{UE} - \Phi_{\mathbf{1}}\|_\diamond. \tag{A.4}$$

To express the distance between unitary channels, we need to introduce the definition of numerical range [43]. The set

$$W(A) = \{\langle x|A|x\rangle : |x\rangle \in \mathbb{C}^d, \ \ \langle x|x\rangle = 1\} \tag{A.5}$$

is called the numerical range of a given matrix $A \in \mathcal{M}_d$. The detailed properties of the numerical range and its generalizations we can read on the website [44].

Due to the definition of $W(A)$, the distance between two unitary channels Φ_U and $\Phi_{\mathbf{1}}$ can be written as

$$\|\Phi_U - \Phi_{\mathbf{1}}\|_\diamond = 2\sqrt{1-\nu^2}, \tag{A.6}$$

where $\nu = \min_{x \in W(U^\dagger)} |x|$.

Appendix B. Discrimination task for Hadamard gate

For the discrimination task between von Neumann measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$, where $U = H$ (the Hadamard gate), the key is to calculate the diamond norm $\|\mathcal{P}_H - \mathcal{P}_{\mathbf{1}}\|_\diamond$ and determine the discriminator $|\psi_0\rangle$. Using semidefinite programming [34], we obtain

$$\|\mathcal{P}_H - \mathcal{P}_{\mathbf{1}}\|_\diamond = \sqrt{2}. \tag{B.1}$$

From [45] we have

$$\|\mathcal{P}_H - \mathcal{P}_{\mathbf{1}}\|_\diamond = \|\Phi_{HE_0} - \Phi_{\mathbf{1}}\|_\diamond, \tag{B.2}$$

where Φ_U is a unitary channel and E_0 is of the form

$$E_0 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1+i & 0 \\ 0 & -1-i \end{pmatrix}. \tag{B.3}$$

Next, in order to construct the discriminator $|\psi_0\rangle$ we use Lemma 5 and the proof of Theorem 1 in [31]. We show that there exist states ρ_1 and ρ_2 of the form $\rho_1 = \frac{1}{2}\begin{pmatrix} 1 & i \\ -i & 1 \end{pmatrix}$ and $\rho_2 = \frac{1}{2}\begin{pmatrix} 1 & -i \\ i & 1 \end{pmatrix}$, respectively. Thus, we construct the quantum state ρ_0 as follows:

$$\rho_0 = \frac{1}{2}\rho_1 + \frac{1}{2}\rho_2 = \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{B.4}$$

According to the Lemma 5 and the proof of Theorem 1 in [31] we assume that

$$|\psi_0\rangle = \left|\sqrt{\rho_0^\top}\right\rangle\!\!\Big\rangle. \tag{B.5}$$

It directly implies that

$$|\psi_0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{B.6}$$

Next, from Holevo-Helstrom theorem [32], we determine the final measurement $\mathcal{P}_{V_i}$. Let us consider

$$X = (\mathcal{P}_H \otimes \mathbb{1})\left(|\psi_0\rangle\langle\psi_0|\right) - (\mathcal{P}_{\mathbf{1}} \otimes \mathbb{1})\left(|\psi_0\rangle\langle\psi_0|\right). \tag{B.7}$$

From the Hahn-Jordan decomposition [32], let us note

$$X = P - Q, \tag{B.8}$$

where $P, Q \geq 0$. Let us define projectors Π_P and Π_Q onto $\mathrm{im}(P)$ and $\mathrm{im}(Q)$, respectively. Observe, that P and Q are block-diagonal. Then, Π_P and Π_Q have the following forms

$$\Pi_P = \begin{pmatrix} |x_p\rangle\langle x_p| & 0 \\ 0 & |y_p\rangle\langle y_p| \end{pmatrix}, \tag{B.9}$$

and

$$\Pi_Q = \begin{pmatrix} |x_q\rangle\langle x_q| & 0 \\ 0 & |y_q\rangle\langle y_q| \end{pmatrix}. \tag{B.10}$$

Hence, we define V_0 as

$$\begin{cases} |x_p\rangle = V_0|0\rangle \\ |x_q\rangle = V_0|1\rangle \end{cases} \tag{B.11}$$

and V_1 as

$$\begin{cases} |y_p\rangle = V_1|0\rangle \\ |y_q\rangle = V_1|1\rangle \end{cases}. \tag{B.12}$$

For the discrimination task between $\mathcal{P}_H$ and $\mathcal{P}_{\mathbf{1}}$ the explicit form of V_0 and V_1 is given as follows (see also `mathematics/optimal_final_measurement_discrimination.nb` in the source code repository):

$$V_0 = \begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix}, \tag{B.13}$$

and

$$V_1 = \begin{pmatrix} -\beta & \alpha \\ \alpha & \beta \end{pmatrix}, \tag{B.14}$$

where

$$\alpha = \frac{\sqrt{2-\sqrt{2}}}{2} = \cos\left(\frac{3}{8}\pi\right), \tag{B.15}$$

and

$$\beta = \frac{\sqrt{2+\sqrt{2}}}{2} = \sin\left(\frac{3}{8}\pi\right). \tag{B.16}$$

Appendix C. Optimal probability for parameterized Fourier family

Let us focus on single-qubit von Neumann measurements $\mathcal{P}_{\mathbf{1}}$ and $\mathcal{P}_U$. Assume that the unitary matrix U is of the form

$$U = H \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix} H^\dagger \tag{C.1}$$

where H is the Hadamard matrix of dimension two and $\phi \in [0, 2\pi)$. In this section we present theoretical probability of correct discrimination between these measurements. To do that, we will present an auxiliary lemma.

Lemma 1. *Let $U = H\mathrm{diag}(1, e^{i\phi})H^\dagger$, $\phi \in [0, 2\pi)$ and let Φ_U and $\Phi_{\mathbf{1}}$ be two unitary channels. Then, the following equation holds*

$$\min_{E\in\mathcal{DU}_2} \|\Phi_{UE} - \Phi_{\mathbf{1}}\|_\diamond = \|\Phi_U - \Phi_{\mathbf{1}}\|_\diamond, \tag{C.2}$$

Proof. Recall that the distance between two unitary channels is given by $\|\Phi_U - \Phi_{\mathbf{1}}\|_\diamond = 2\sqrt{1-\nu^2}$, where $\nu = \min_{x\in W(U^\dagger)} |x|$ for any $U \in \mathcal{U}_d$. For $U = H \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix} H^\dagger$ the readers briefly observe that $\nu^2 = 1 - \frac{|1-e^{-i\phi}|^2}{4} = 1 - \frac{|1-e^{i\phi}|^2}{4}$. So,

$$\|\Phi_U - \Phi_{\mathbf{1}}\|_\diamond = |1 - e^{i\phi}|. \tag{C.3}$$

It implies that it is enough to prove

$$\min_{E\in\mathcal{DU}_2} \|\Phi_{UE} - \Phi_{\mathbf{1}}\|_\diamond = \ |1 - e^{i\phi}|. \tag{C.4}$$

This condition is equivalent to show that for every $E \in \mathcal{DU}_2$

$$\nu_E \leq \frac{|1+e^{i\phi}|}{2}, \tag{C.5}$$

where $\nu_E = \min_{x \in W(U^\dagger E)} |x|$.

The celebrated Hausdorf-Töplitz theorem [46, 47] states that $W(A)$ of any matrix $A \in \mathcal{M}_d$ is a convex set, and therefore we have

$$W(A) = \{\mathrm{tr}(A\rho) : \rho \in \Omega_d\}. \tag{C.6}$$

So, we can assume that

$$\min_{|x\rangle \in \mathbb{C}^2 : |x\rangle\langle x| = 1} |\langle x|U^\dagger|x\rangle| = \min_{\rho \in \Omega_2} |\mathrm{tr}(U^\dagger \rho)|. \tag{C.7}$$

Then, we have

$$\nu_E = \min_{\rho \in \Omega_2} \left|\mathrm{tr}\left(\rho U E\right)\right|. \tag{C.8}$$

For that, our task is reduced to show that for every $E \in \mathcal{DU}_2$ there exists $\rho \in \Omega_2$ such that

$$\left|\mathrm{tr}\left(\rho U E\right)\right| \leq \frac{|1 + e^{i\phi}|}{2}. \tag{C.9}$$

Let us define $E = \begin{pmatrix} E_0 & 0 \\ 0 & E_1 \end{pmatrix}$ and take $\rho = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$. From spectral theorem, let us decompose U as

$$U = \lambda_0 |x_0\rangle\langle x_0| + \lambda_1 |x_1\rangle\langle x_1|, \tag{C.10}$$

where for eigenvalue $\lambda_0 = 1$, the corresponding eigenvector is of the form $|x_0\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$, whereas for $\lambda_1 = e^{i\phi}$ we have $|x_1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$. Then, for every $E \in \mathcal{DU}_2$ we have

$$\begin{aligned} |\mathrm{tr}(\rho U E)| &= \frac{1}{2}\left|\mathrm{tr}\left(H \mathrm{diag}(1, e^{i\phi}) H^\dagger E\right)\right| = \frac{1}{2}\left|\mathrm{tr}\left((|x_0\rangle\langle x_0| + e^{i\phi}|x_1\rangle\langle x_1|)E\right)\right| \\ &= \frac{1}{2}\left|\langle x_0|E|x_0\rangle + e^{i\phi}\langle x_1|E|x_1\rangle\right| = \frac{1}{2}\left|\frac{E_0 + E_1}{2} + e^{i\phi}\frac{E_0 + E_1}{2}\right| \\ &= \frac{\left|1 + e^{i\phi}\right|}{2}\left|\frac{E_0 + E_1}{2}\right| \leq \frac{|1 + e^{i\phi}|}{2}, \end{aligned} \tag{C.11}$$

which completes the proof. □

Theorem 1. *The optimal probability of correct discrimination between von Neumann measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ for $U = H\mathrm{diag}(1, e^{i\phi})H^\dagger$, where $\phi \in [0, 2\pi)$ is given by*

$$p_{succ}(\mathcal{P}_U, \mathcal{P}_{\mathbf{1}}) = \frac{1}{2} + \frac{|1 - e^{i\phi}|}{4}. \tag{C.12}$$

Proof. From Holevo-Helstrom theorem, we obtain

$$p_{\text{succ}}(\mathcal{P}_U, \mathcal{P}_{\mathbf{1}}) = \frac{1}{2} + \frac{1}{4}\|\mathcal{P}_U - \mathcal{P}_{\mathbf{1}}\|_\diamond. \tag{C.13}$$

From [31, Theorem 1], we have

$$\|\mathcal{P}_U - \mathcal{P}_{\mathbf{1}}\|_\diamond = \min_{E \in \mathcal{DU}_d} \|\Phi_{UE} - \Phi_{\mathbf{1}}\|_\diamond. \tag{C.14}$$

From Lemma 1, we know that for $U = H\text{diag}(1, e^{i\phi})H^\dagger$, it also holds that

$$\min_{E \in \mathcal{DU}_2} \|\Phi_{UE} - \Phi_{\mathbf{1}}\|_\diamond = \|\Phi_U - \Phi_{\mathbf{1}}\|_\diamond, \tag{C.15}$$

which is exactly equal to

$$\|\Phi_U - \Phi_{\mathbf{1}}\|_\diamond = 2\sqrt{1-\nu^2} = |1 - e^{i\phi}|. \tag{C.16}$$

It implies that

$$p_{\text{succ}}(\mathcal{P}_U, \mathcal{P}_{\mathbf{1}}) = \frac{1}{2} + \frac{|1 - e^{i\phi}|}{4}, \tag{C.17}$$

which completes the proof. □

Appendix D. Optimal discrimination strategy for parameterized Fourier family

In this Appendix we create the optimal theoretical strategy of discrimination between $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$. To indicate the optimal strategy, we will present two propositions. The first one is concentrated around the discriminator as the optimal input state of discrimination strategy, whereas the second one describes the optimal final measurement.

Proposition 1. *Consider the problem of discrimination between von Neumann measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$, $U = H\text{diag}(1, e^{i\phi})H^\dagger$ and $\phi \in [0, 2\pi)$. The discriminator has the form*

$$|\psi_0\rangle = \frac{1}{\sqrt{2}}|\mathbb{1}_2\rangle\rangle. \tag{D.1}$$

Proof. Let $U = H\text{diag}(1, e^{i\phi})H^\dagger, \phi \in [0, 2\pi)$ be decomposed as

$$U = \lambda_0|x_0\rangle\langle x_0| + \lambda_1|x_1\rangle\langle x_1|, \tag{D.2}$$

where for eigenvalue $\lambda_0 = 1$, the corresponding eigenvector is of the form $|x_0\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$, whereas for $\lambda_1 = e^{i\phi}$ we have $|x_1\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$. For Hermitian-preserving maps [32] the diamond norm may be expressed as

$$\|\Phi\|_\diamond = \max_{\rho\in\Omega_d} \| \left(\mathbb{1}\otimes\sqrt{\rho}\right) J(\Phi) \left(\mathbb{1}\otimes\sqrt{\rho}\right) \|_1. \tag{D.3}$$

Hence, we obtain

$$\begin{aligned}
\|\mathcal{P}_U - \mathcal{P}_{\mathbf{1}}\|_\diamond &= \max_{\rho\in\Omega_2} \|(\mathbb{1}\otimes\sqrt{\rho})\, J(\mathcal{P}_U - \mathcal{P}_{\mathbf{1}})\, (\mathbb{1}\otimes\sqrt{\rho})\|_1 \\
&= \max_{\rho\in\Omega_2} \left\| (\mathbb{1}\otimes\sqrt{\rho}) \sum_{i=0}^{1} |i\rangle\langle i| \otimes \left(|u_i\rangle\langle u_i| - |i\rangle\langle i|\right)^\top (\mathbb{1}\otimes\sqrt{\rho}) \right\|_1 \\
&= \max_{\rho\in\Omega_2} \left\| \sum_{i=0}^{1} |i\rangle\langle i| \otimes \sqrt{\rho} \left(|u_i\rangle\langle u_i| - |i\rangle\langle i|\right)^\top \sqrt{\rho} \right\|_1 \\
&= \max_{\rho\in\Omega_2} \sum_{i=0}^{1} \left\| \sqrt{\rho} \left(|u_i\rangle\langle u_i| - |i\rangle\langle i|\right)^\top \sqrt{\rho} \right\|_1 .
\end{aligned} \tag{D.4}$$

One can prove that for all $\alpha, \beta \geq 0$, and unit vectors $|x\rangle, |y\rangle$ the following equation holds [32]

$$\|\alpha|x\rangle\langle x| - \beta|y\rangle\langle y|\|_1 = \sqrt{(\alpha+\beta)^2 - 4\alpha\beta|\langle x|y\rangle|^2}. \tag{D.5}$$

By taking $|x\rangle = \frac{\sqrt{\rho}|\bar{u}_i\rangle}{\|\sqrt{\rho}|\bar{u}_i\rangle\|}$ and $|y\rangle = \frac{\sqrt{\rho}|i\rangle}{\|\sqrt{\rho}|i\rangle\|}$ we have

$$\|\mathcal{P}_U - \mathcal{P}_{\mathbf{1}}\|_\diamond = \max_{\rho\in\Omega_2} \sum_{i=0}^{1} \sqrt{\left(\langle\bar{u}_i|\rho|\bar{u}_i\rangle + \langle i|\rho|i\rangle\right)^2 - 4|\langle\bar{u}_i|\rho|i\rangle|^2}. \tag{D.6}$$

Let us take $\rho_0 = \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, we obtain

$$\begin{aligned}
||\mathcal{P}_U - \mathcal{P}_{\mathbf{1}}||_\diamond &\geq \sum_{i=0}^{1} \sqrt{\left(\langle \bar{u}_i|\rho_0|\bar{u}_i\rangle + \langle i|\rho_0|i\rangle\right)^2 - 4|\langle i|\rho_0|\bar{u}_i\rangle|^2} \\
&= \sum_{i=0}^{1} \sqrt{1 - |\langle i|U|i\rangle|^2} \\
&= \sum_{i=0}^{1} \sqrt{1 - |1 \cdot \langle i|u_0\rangle\langle u_0|i\rangle + e^{i\phi} \cdot \langle i|u_1\rangle\langle u_1|i\rangle|^2} \\
&= \sum_{i=0}^{1} \sqrt{1 - \left|\frac{1+e^{i\phi}}{2}\right|^2} = 2\sqrt{1 - \left|\frac{1+e^{i\phi}}{2}\right|^2} \\
&= |1 - e^{i\phi}|.
\end{aligned} \tag{D.7}$$

Due to Theorem 1 and the following equality

$$\|\left(\mathbb{1}\otimes\sqrt{\rho}\right) J(\mathcal{P}_U - \mathcal{P}_{\mathbf{1}}) \left(\mathbb{1}\otimes\sqrt{\rho}\right)\|_1 = \left\|\left((\mathcal{P}_U - \mathcal{P}_{\mathbf{1}})\otimes\mathbb{1}\right)\left(|\sqrt{\rho}^\top\rangle\rangle\langle\langle\sqrt{\rho}^\top|\right)\right\|_1, \tag{D.8}$$

the discriminator $|\psi_0\rangle$ is equal to

$$|\psi_0\rangle = |\sqrt{\rho_0}^\top\rangle\rangle = \frac{1}{\sqrt{2}}|\mathbb{1}_2\rangle\rangle, \tag{D.9}$$

which completes the proof. □

Proposition 2. *Consider the problem of discrimination between von Neumann measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$, $U = H\mathrm{diag}(1, e^{i\phi})H^\dagger$ and $\phi \in [0, 2\pi)$. The controlled unitaries V_0 and V_1 have the form*

$$V_0 = \begin{pmatrix} i\sin\left(\frac{\pi-\phi}{4}\right) & -i\cos\left(\frac{\pi-\phi}{4}\right) \\ \cos\left(\frac{\pi-\phi}{4}\right) & \sin\left(\frac{\pi-\phi}{4}\right) \end{pmatrix}, \tag{D.10}$$

and

$$V_1 = \begin{pmatrix} -i\cos\left(\frac{\pi-\phi}{4}\right) & i\sin\left(\frac{\pi-\phi}{4}\right) \\ \sin\left(\frac{\pi-\phi}{4}\right) & \cos\left(\frac{\pi-\phi}{4}\right) \end{pmatrix}. \tag{D.11}$$

Proof. From Proposition 1 we obtain the exact form of discriminator given by

$$|\psi_0\rangle = \frac{1}{\sqrt{2}}|\mathbb{1}_2\rangle\rangle. \tag{D.12}$$

Repeating the procedure used to distinguish the von Neumann measurements in the Hadamard basis (see Appendix B), we use the Hahn-Jordan decomposition and then we create the projective operators into the positive and negative part of X matrix. Hence, the explicit form of V_0 and V_1 is given as follows:

$$V_0 = \begin{pmatrix} i\sin\left(\frac{\pi-\phi}{4}\right) & -i\cos\left(\frac{\pi-\phi}{4}\right) \\ \cos\left(\frac{\pi-\phi}{4}\right) & \sin\left(\frac{\pi-\phi}{4}\right) \end{pmatrix}, \tag{D.13}$$

and

$$V_1 = \begin{pmatrix} -i\cos\left(\frac{\pi-\phi}{4}\right) & i\sin\left(\frac{\pi-\phi}{4}\right) \\ \sin\left(\frac{\pi-\phi}{4}\right) & \cos\left(\frac{\pi-\phi}{4}\right) \end{pmatrix}, \tag{D.14}$$

where $\phi \in [0, 2\pi)$. □

Appendix E. Output YAML files

In this appendix we present examples of YAML's files obtained from synchronous ans asynchronous experiments. We will start at synchronous case.

Listing 14: Defining experiment file

```yaml
type: discrimination-fourier
qubits:
   - target: 0
     ancilla: 1
   - target: 1
     ancilla: 2
angles:
   start: 0
   stop: 2 * pi
   num_steps: 3
gateset: ibmq
method: direct_sum
num_shots: 100
```

Listing 15: Defining backend file

```yaml
name: ibmq_quito
asynchronous: false
provider:
   hub: ibm-q
   group: open
   project: main
```

Listing 16: Results (synchronous)


```
metadata:
   experiments:
      type: discrimination-fourier
      qubits:
      - {target: 0, ancilla: 1}
      - {target: 1, ancilla: 2}
      angles: {start: 0.0, stop: 6.283185307179586, num_steps: 3}
      gateset: ibmq
      method: direct_sum
      num_shots: 100
   backend_description:
      name: ibmq_quito
      asynchronous: false
      provider: {group: open, hub: ibm-q, project: main}
data:
- target: 0
  ancilla: 1
  phi: 0.0
  results_per_circuit:
  - name: id
   histogram: {'00': 28, '01': 26, '10': 21, '11': 25}
   mitigation_info:
      target: {prob_meas0_prep1: 0.052200000000000024,
          prob_meas1_prep0: 0.0172}
      ancilla: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
   mitigated_histogram: {'00': 0.2637212373658018, '01':
       0.25865061319892463, '10': 0.2067279352110304, '11':
       0.2709002142242433}
  - name: u
   histogram: {'00': 30, '01': 16, '10': 28, '11': 26}
   mitigation_info:
      target: {prob_meas0_prep1: 0.052200000000000024,
          prob_meas1_prep0: 0.0172}
      ancilla: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
   mitigated_histogram: {'00': 0.2857378991684036, '01':
       0.14975297832942433, '10': 0.28142307224788693, '11':
       0.2830860502542851}
- target: 0
  ancilla: 1
  phi: 3.141592653589793
```

```
  results_per_circuit:
  - name: id
   histogram: {'00': 4, '01': 5, '10': 45, '11': 46}
   mitigation_info:
      target: {prob_meas0_prep1: 0.052200000000000024,
          prob_meas1_prep0: 0.0172}
      ancilla: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
   mitigated_histogram: {'00': 0.011053610583159325, '01':
       0.02261276648026373, '10': 0.4593955619936729, '11':
       0.5069380609429042}
  - name: u
   histogram: {'00': 56, '01': 43, '10': 1}
   mitigation_info:
      target: {prob_meas0_prep1: 0.052200000000000024,
          prob_meas1_prep0: 0.0172}
      ancilla: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
   mitigated_histogram: {'00': 0.5573987337172156, '01':
       0.44424718645642625, '10': -0.0016459201736417181}
- target: 0
  ancilla: 1
  phi: 6.283185307179586
  results_per_circuit:
  - name: id
   histogram: {'00': 36, '01': 18, '10': 25, '11': 21}
   mitigation_info:
      target: {prob_meas0_prep1: 0.052200000000000024,
          prob_meas1_prep0: 0.0172}
      ancilla: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
   mitigated_histogram: {'00': 0.3488190312089973, '01':
       0.17355281935572894, '10': 0.2505792064871127, '11':
       0.22704894294816103}
  - name: u
   histogram: {'00': 32, '01': 27, '10': 24, '11': 17}
   mitigation_info:
      target: {prob_meas0_prep1: 0.052200000000000024,
          prob_meas1_prep0: 0.0172}
      ancilla: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
   mitigated_histogram: {'00': 0.3025357275361897, '01':
       0.27413673119534815, '10': 0.24313373302688793, '11':
```

```
      0.18019380824157433}
- target: 1
  ancilla: 2
  phi: 0.0
  results_per_circuit:
  - name: id
   histogram: {'00': 27, '01': 20, '10': 24, '11': 29}
   mitigation_info:
      target: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
      ancilla: {prob_meas0_prep1: 0.07540000000000002,
          prob_meas1_prep0: 0.0528}
   mitigated_histogram: {'00': 0.2594378169217188, '01':
       0.19318893233269735, '10': 0.23035366874292057, '11':
       0.3170195820026633}
  - name: u
   histogram: {'00': 31, '01': 24, '10': 23, '11': 22}
   mitigation_info:
      target: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
      ancilla: {prob_meas0_prep1: 0.07540000000000002,
          prob_meas1_prep0: 0.0528}
   mitigated_histogram: {'00': 0.30056875246775644, '01':
       0.2438221628798003, '10': 0.22180309809696985, '11':
       0.23380598655547338}
- target: 1
  ancilla: 2
  phi: 3.141592653589793
  results_per_circuit:
  - name: id
   histogram: {'00': 5, '01': 4, '10': 50, '11': 41}
   mitigation_info:
      target: {prob_meas0_prep1: 0.05900000000000005,
          prob_meas1_prep0: 0.0202}
      ancilla: {prob_meas0_prep1: 0.07540000000000002,
          prob_meas1_prep0: 0.0528}
   mitigated_histogram: {'00': 0.009552870928837118, '01':
       0.007194089383161034, '10': 0.5236791012692514, '11':
       0.4595739384187503}
  - name: u
   histogram: {'00': 41, '01': 51, '10': 3, '11': 5}
   mitigation_info:
      target: {prob_meas0_prep1: 0.05900000000000005,
```

```
        prob_meas1_prep0: 0.0202}
      ancilla: {prob_meas0_prep1: 0.07540000000000002,
        prob_meas1_prep0: 0.0528}
    mitigated_histogram: {'00': 0.4073387714165384, '01':
        0.5614614121117936, '10': 0.006431862814564833, '11':
        0.024767953657102992}
- target: 1
  ancilla: 2
  phi: 6.283185307179586
  results_per_circuit:
  - name: id
    histogram: {'00': 30, '01': 28, '10': 23, '11': 19}
    mitigation_info:
      target: {prob_meas0_prep1: 0.05900000000000005,
        prob_meas1_prep0: 0.0202}
      ancilla: {prob_meas0_prep1: 0.07540000000000002,
        prob_meas1_prep0: 0.0528}
    mitigated_histogram: {'00': 0.2868459834940102, '01':
        0.2919564941384742, '10': 0.22466574543735374, '11':
        0.19653177693016174}
  - name: u
    histogram: {'00': 15, '01': 20, '10': 36, '11': 29}
    mitigation_info:
      target: {prob_meas0_prep1: 0.05900000000000005,
        prob_meas1_prep0: 0.0202}
      ancilla: {prob_meas0_prep1: 0.07540000000000002,
        prob_meas1_prep0: 0.0528}
    mitigated_histogram: {'00': 0.1187719606657805, '01':
        0.1962085394489247, '10': 0.3710195249988589, '11':
        0.31399997488643583}
```


For the same experiment file, we use the flag `asynchronous: true` to define asynchronous experiment.

Listing 17: Backend file

```
name: ibmq_quito
asynchronous: true
provider:
  hub: ibm-q
  group: open
  project: main
```

If the backend is asynchronous, the output will contain intermediate data containing, amongst others, job ids correlated with the circuit they correspond to.

Listing 18: Resolved results

```
metadata:
  experiments:
     type: discrimination-fourier
     qubits:
     - {target: 0, ancilla: 1}
     - {target: 1, ancilla: 2}
     angles: {start: 0.0, stop: 6.283185307179586, num_steps: 3}
     gateset: ibmq
     method: direct_sum
     num_shots: 100
  backend_description:
     name: ibmq_quito
     asynchronous: true
     provider: {group: open, hub: ibm-q, project: main}
data:
- job_id: 63e7f17a17b7ed49ca24e05b
  keys:
  - [0, 1, id, 0.0]
  - [0, 1, u, 0.0]
  - [0, 1, id, 3.141592653589793]
  - [0, 1, u, 3.141592653589793]
  - [0, 1, id, 6.283185307179586]
  - [0, 1, u, 6.283185307179586]
  - [1, 2, id, 0.0]
  - [1, 2, u, 0.0]
  - [1, 2, id, 3.141592653589793]
  - [1, 2, u, 3.141592653589793]
  - [1, 2, id, 6.283185307179586]
  - [1, 2, u, 6.283185307179586]
```

Finally, if the status of jobs is `DONE`, we resolve the measurements from the submitted jobs obtaining the following file.

Listing 19: Results (asynchronous)

```
metadata:
  experiments:
   type: discrimination-fourier
   qubits:
   - target: 0
```

```
     ancilla: 1
   - target: 1
     ancilla: 2
   angles:
     start: 0.0
     stop: 6.283185307179586
     num_steps: 3
   gateset: ibmq
   method: direct_sum
   num_shots: 100
  backend_description:
   name: ibmq_quito
   asynchronous: true
   provider:
      group: open
      hub: ibm-q
      project: main
data:
- target: 0
  ancilla: 1
  phi: 0.0
  results_per_circuit:
  - name: id
   histogram:
      '00': 27
      '01': 28
      '10': 18
      '11': 27
   mitigation_info:
     target:
      prob_meas0_prep1: 0.052200000000000024
      prob_meas1_prep0: 0.0172
     ancilla:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
   mitigated_histogram:
      '00': 0.254196166145997
      '01': 0.2790358060520916
      '10': 0.1732699847244092
      '11': 0.29349804307750227
  - name: u
   histogram:
      '00': 29
```

```
      '01': 17
      '10': 30
      '11': 24
   mitigation_info:
     target:
      prob_meas0_prep1: 0.052200000000000024
      prob_meas1_prep0: 0.0172
     ancilla:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
   mitigated_histogram:
      '00': 0.2733793468261183
      '01': 0.1621115306717096
      '10': 0.3045273800167787
      '11': 0.2599817424853933
- target: 0
  ancilla: 1
  phi: 3.141592653589793
  results_per_circuit:
  - name: id
   histogram:
      '00': 3
      '01': 5
      '10': 37
      '11': 55
   mitigation_info:
     target:
      prob_meas0_prep1: 0.052200000000000024
      prob_meas1_prep0: 0.0172
     ancilla:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
   mitigated_histogram:
      '00': 0.006189545789708441
      '01': 0.016616709640352317
      '10': 0.3675478279476653
      '11': 0.6096459166222741
  - name: u
   histogram:
      '00': 56
      '01': 42
      '10': 2
   mitigation_info:
```

```
     target:
      prob_meas0_prep1: 0.05220000000000024
      prob_meas1_prep0: 0.0172
     ancilla:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
   mitigated_histogram:
      '00': 0.55731929321128
      '01': 0.43367489257574243
      '10': 0.009005814212977551
- target: 0
  ancilla: 1
  phi: 6.283185307179586
  results_per_circuit:
  - name: id
   histogram:
      '00': 18
      '01': 28
      '10': 30
      '11': 24
   mitigation_info:
     target:
      prob_meas0_prep1: 0.05220000000000024
      prob_meas1_prep0: 0.0172
     ancilla:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
   mitigated_histogram:
      '00': 0.15258295844557557
      '01': 0.2829079190522524
      '10': 0.3071204587046501
      '11': 0.25738866379752195
  - name: u
   histogram:
      '00': 32
      '01': 28
      '10': 23
      '11': 17
   mitigation_info:
     target:
      prob_meas0_prep1: 0.05220000000000024
      prob_meas1_prep0: 0.0172
     ancilla:
```

```
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
   mitigated_histogram:
      '00': 0.3026150836796529
      '01': 0.28491749668524724
      '10': 0.23230862145681827
      '11': 0.18015879817828173
- target: 1
  ancilla: 2
  phi: 0.0
  results_per_circuit:
  - name: id
   histogram:
      '00': 27
      '01': 16
      '10': 30
      '11': 27
   mitigation_info:
     target:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
     ancilla:
      prob_meas0_prep1: 0.07540000000000002
      prob_meas1_prep0: 0.0528
   mitigated_histogram:
      '00': 0.256742095057232
      '01': 0.15000257115061383
      '10': 0.29821012040758116
      '11': 0.29504521338457296
  - name: u
   histogram:
      '00': 34
      '01': 22
      '10': 25
      '11': 19
   mitigation_info:
    target:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
     ancilla:
      prob_meas0_prep1: 0.07540000000000002
      prob_meas1_prep0: 0.0528
   mitigated_histogram:
```

```
      '00': 0.3325088211394024
      '01': 0.22335261496979697
      '10': 0.2441636375921354
      '11': 0.19997492629866526
- target: 1
  ancilla: 2
  phi: 3.141592653589793
  results_per_circuit:
  - name: id
   histogram:
      '00': 3
      '01': 9
      '10': 51
      '11': 37
   mitigation_info:
     target:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
     ancilla:
      prob_meas0_prep1: 0.07540000000000002
      prob_meas1_prep0: 0.0528
   mitigated_histogram:
      '00': -0.016627023111853642
      '01': 0.06778554570877951
      '10': 0.53899887367658
      '11': 0.40984260372649417
  - name: u
   histogram:
      '00': 43
      '01': 45
      '10': 7
      '11': 5
   mitigation_info:
     target:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
     ancilla:
      prob_meas0_prep1: 0.07540000000000002
      prob_meas1_prep0: 0.0528
   mitigated_histogram:
      '00': 0.42955729968594086
      '01': 0.49336080079582095
      '10': 0.04937406434533623
```

```
    '11': 0.02770783517290191
- target: 1
  ancilla: 2
  phi: 6.283185307179586
  results_per_circuit:
  - name: id
   histogram:
      '00': 22
      '01': 19
      '10': 35
      '11': 24
   mitigation_info:
     target:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
     ancilla:
      prob_meas0_prep1: 0.07540000000000002
      prob_meas1_prep0: 0.0528
   mitigated_histogram:
      '00': 0.19592641048040849
      '01': 0.18787721420415215
      '10': 0.3590258049844047
      '11': 0.25717057033103463
  - name: u
      histogram:
      '00': 27
      '01': 24
      '10': 25
      '11': 24
   mitigation_info:
     target:
      prob_meas0_prep1: 0.05900000000000005
      prob_meas1_prep0: 0.0202
     ancilla:
      prob_meas0_prep1: 0.07540000000000002
      prob_meas1_prep0: 0.0528
   mitigated_histogram:
      '00': 0.25555866817587225
      '01': 0.2429501641251142
      '10': 0.24509293912212946
      '11': 0.2563982285768841
```