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 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. We start by discussing 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 the limitations of the current NISQ devices. We encourage the readers interested in a more in–depth discussion of the mathematical foundations behind our discrimination scheme to read Section 1 in the supplemental materials.

3.1. Von Neumann Measurements

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 the identity operator, 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.

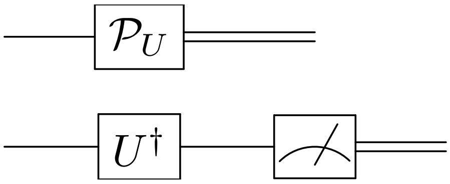

Figure 1: Implementation of a von Neumann measurement using measurement in computational basis. The upper circuit shows a symbolic representation of a von Neumann measurement $\mathcal{P}_U$. The bottom, equivalent circuit depicts its decomposition into a change of basis followed by measurement in the 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. 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 binary measurement $\mathcal{P}_{V_i}$ and perform it on the second qubit, obtaining the outcome j. Finally, if $j = 0$, we say that the performed measurement is $\mathcal{P}_U$, otherwise we say that it was $\mathcal{P}_{\mathbf{1}}$.

Note, however, that the discrimination scheme described above can work regardless of dimensionality of the unitary U. The main difference is the dimension of the auxiliary system, which in general can be larger than two. This dimension depends on the Schmidt rank of the optimal input state. However, for most discrimination schemes, the Schmidt rank equals at most two, and hence the auxiliary system is also a qubit, see [31] for details. Note that the final measurement $\mathcal{P}_{V_i}$ is always binary, independently of the dimension of auxiliary system.

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 $\mathcal{P}_{\mathbf{1}}$.

In principle, our discrimination scheme could be used with any choice of

[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.

$|\psi_0\rangle$ and final measurements $\mathcal{P}_{V_i}$. However, we argue that it is best to choose those components so that they maximize the probability of correct discrimination. To see that, suppose that some choice of $|\psi_0\rangle, \mathcal{P}_{V_0}, \mathcal{P}_{V_1}$ allows for correctly discriminating between two measurements with probability equal to 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 $\frac{3}{5}$, 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.

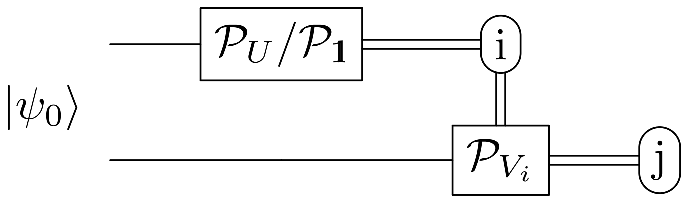

Figure 2: Theoretical scheme of discrimination between von Neumann measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$.

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.

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 $i \in \{0, 1\}$

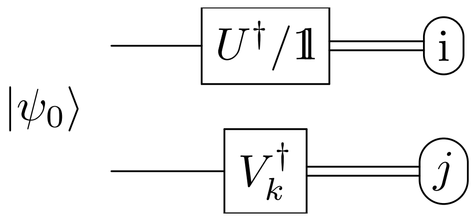

Figure 3: A schematic representation of the setup for distinguishing measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ using postselection approach. In postselection scheme, one runs such circuits for both $k = 0, 1$ and discards results for cases when there is a mismatch between k and i.

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

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

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.

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

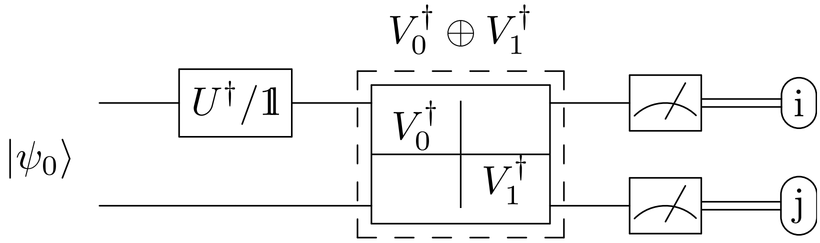

Figure 4: A schematic representation of the setup for distinguishing measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ using the $V_0^\dagger \oplus V_1^\dagger$ direct sum.

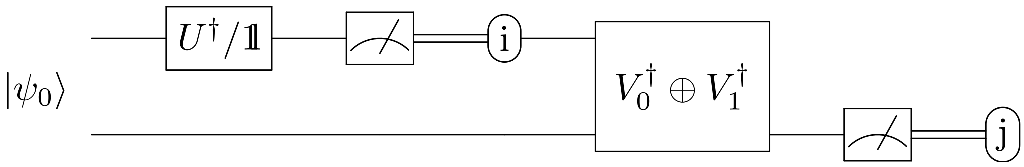

Figure 5: Rewritten representation of the setup for distinguishing measurements $\mathcal{P}_U$ and $\mathcal{P}_{\mathbf{1}}$ using the $V_0^\dagger \oplus V_1^\dagger$ direct sum.

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

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

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.

4. Software description

This section is divided into two parts. In Section 4.1 we describe functionalities of PyQBench package. Next, in Section 4.2, we give a general overview of the software architecture.

4.1. 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 (see Section 3 in supplemental material). 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 [32] files describing the benchmark to be performed and the description of the backend on which the benchmark should be run. The same benchmark can be used with different backends and vice versa.

4.2. Software Architecture

4.2.1. 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 [33] or at the reference manual [34].

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.

4.2.2. 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 [35, 36]).

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 5, and the detailed description of the expected format can be found at PyQBench's documentation.

4.2.3. 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 hierarchically structuring the CL 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 5.

4.2.4. *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 5 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.

5. Illustrative examples

In this section, we demonstrate the usage of PyQBench. For brevity, we decided to present only the usage of the CLI tool, as it is likely to be the most popular use case. We refer readers interested in implementing their discrimination schemes using custom measurements to PyQBench's documentation [34], where we describe the whole process, and to the Section 2 in the supplemental material, where we discuss the relevant mathematical details.

5.1. *Using `qbench` CLI*

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.

5.1.1. 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 2: 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 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 3: IBMQ backend

```
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.

5.1.2. 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 the 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 the synchronous experiment, the part of the output looks similar to the one below. The whole YAML file can be seen in Section ?? in the supplemental material.


```
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}
```

5.1.3. (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.

5.1.4. 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.

5.1.5. 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 in Table 2.

6. 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 access to different figures of merit for their architectures. Our package allows the user to test various architectures, available through `qiskit` and Amazon

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 [37, 38], if it was applied.

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 [33] under an open source license which will allow users to utilize and extend our package in their specific applications.

7. Conclusions

In this paper, we presented 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.

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

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

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

[35] 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.

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

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

[38] 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.