

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

Konrad Jałowiecki*, Paulina Lewandowska, Łukasz Paweł

*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	<code>Python >= 3.8</code> <code>numpy ~= 1.22.0</code> <code>scipy ~= 1.7.0</code> <code>pandas ~= 1.5.0</code> <code>amazon-braket-sdk >= 1.11.1</code> <code>pydantic ~= 1.9.1</code> <code>qiskit ~= 0.37.2</code> <code>mthree ~= 1.1.0</code> <code>tqdm ~= 4.64.1</code> <code>pyyaml ~= 6.0</code> <code>qiskit-braket-provider ~= 0.0.3</code>
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 neutral 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, Zapata Computing’s Orchestra [14], XACC [15] and NVIDIA’s CUDA Quantum [16].

It is well known that NISQ devices have their limitations [17]. 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 [18, 19, 20, 21, 22], benchmarks based on the quantum volume [23, 24, 25].

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) [26, 27] or SupermarQ [28, 29] 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 [30] or PyQuil [31]. Another, equally popular, benchmarking method is quantum tomography [32]. Implementation of quantum tomography for benchmarking NISQ devices can be found in [33].

In [34], the authors evaluated several IBM-Q machines using seven benchmarks taking into account the errors and execution time.

QASMBench [35] is one of the first benchmark suites aiming at evaluating NISQ devices using quantum applications from a broad range of domains, mainly using an approach based on fidelity estimation. The benchmark presented in the paper compares the fidelity of execution among the IBM-Q machines, the IonQ QPU and the Rigetti Aspen M-1 system.

Another quantity used for benchmarking NISQ devices is quantum volume. The quantum volume characterizes the capacity of a device for solving computational problems. It takes into account multiple factors like the number of qubits, connectivity and measurement errors. The Qiskit library allows one to measure the 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. [36].

Finally, we should mention cross-entropy benchmarking [37], which was utilized in validation of the Sycamore-53 QPU supremacy experiments [38]. In this approach, the quality of an algorithm implemented on the QPU is measured by calculating the cross entropy of bit-strings actually sampled from the QPU, compared to ideal bitstrings.

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 $\{|u_0\rangle\langle u_0|, \dots, |u_{d-1}\rangle\langle u_{d-1}|\}$, 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.

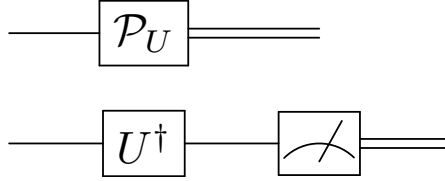


Figure 1: Implementation of a von Neumann measurement using measurement in computational basis. The upper circuit shows a symbolic representation of a von Neumann measurement \mathcal{P}_U . The bottom, equivalent circuit depicts its decomposition into a change of basis followed by measurement in the 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 generality¹, we consider discrimination task between single qubit measurements \mathcal{P}_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}_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}_1 .

¹Explaining why we can consider only discrimination scheme between \mathcal{P}_1 and \mathcal{P}_U is beyond the scope of this paper. See [39] for a in depth explanation.

114 Note, however, that the discrimination scheme described above can work
 115 regardless of dimensionality of the unitary U . The main difference is the
 116 dimension of the auxiliary system, which in general can be larger than two.
 117 This dimension depends on the Schmidt rank of the optimal input state.
 118 However, for most discrimination schemes, the Schmidt rank equals at most
 119 two, and hence the auxiliary system is also a qubit, see [39] for details.
 120 Note that the final measurement \mathcal{P}_{V_i} is always binary, independently of the
 121 dimension of auxiliary system.

122 Naturally, we need to repeat the same procedure multiple times for both
 123 measurements to obtain a reliable estimate of the underlying probability
 124 distribution. In PyQBench, we assume that the experiment is repeated the
 125 same number of times for both \mathcal{P}_U and \mathcal{P}_1 .

126 In principle, our discrimination scheme could be used with any choice of
 127 $|\psi_0\rangle$ and final measurements \mathcal{P}_{V_i} . However, we argue that it is best to choose
 128 those components so that they maximize the probability of correct discrim-
 129 ination. To see that, suppose that some choice of $|\psi_0\rangle, \mathcal{P}_{V_0}, \mathcal{P}_{V_1}$ allows for
 130 correctly discriminating between two measurements with probability equal
 131 to one, i.e. on a perfect quantum computer you will always make a cor-
 132 rect guess. Then, on real hardware, we might obtain any empirical value in
 133 range $[\frac{1}{2}, 1]$. On the other hand, if we choose the components of our scheme
 134 such that the successful discrimination probability is $\frac{3}{5}$, the possible range
 135 of empirically obtainable probabilities is only $[\frac{1}{2}, \frac{3}{5}]$. Hence, in the second
 136 case, the discrepancy between theoretical and empirical results will be less
 137 pronounced.

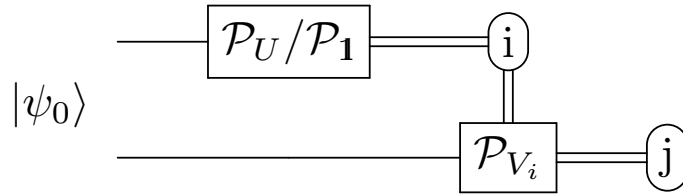


Figure 2: Theoretical scheme of discrimination between von Neumann measurements \mathcal{P}_U and \mathcal{P}_1 .

138 3.2.1. Implementation of discrimination scheme on actual NISQ devices

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

144 **Scheme 1.** (Postselection)

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

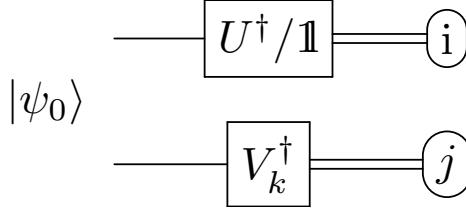


Figure 3: A schematic representation of the setup for distinguishing measurements \mathcal{P}_U and \mathcal{P}_1 using postselection approach. In postselection scheme, one runs such circuits for both $k = 0, 1$ and discards results for cases when there is a mismatch between k and i .

151 To perform the benchmark, one needs to run multiple copies of the post-
 152 selection circuit, with both \mathcal{P}_U and \mathcal{P}_1 . Each circuit has to be run in both
 153 variants, one with final measurement \mathcal{P}_{V_0} and the second with the final mea-
 154 surement \mathcal{P}_{V_1} . The experiments can thus be grouped into classes identified by
 155 tuples of the form (\mathcal{Q}, k, i, j) , where $\mathcal{Q} \in \{\mathcal{P}_U, \mathcal{P}_1\}$ denotes the chosen mea-
 156 surement, $k \in \{0, 1\}$ designates the final measurement used, and $i \in \{0, 1\}$
 157 and $j \in \{0, 1\}$ being the labels of outcomes as presented in Fig. 3. We
 158 then discard all the experiments for which $i \neq k$. The total number of valid
 159 experiments is thus:

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

160 Finally, we count the valid experiments resulting in successful discrimi-
 161 nation. If we have chosen \mathcal{P}_U , then we guess correctly iff $j = 0$. Similarly,
 162 for \mathcal{P}_1 , we guess correctly iff $j = 1$. If we define

$$N_{\mathcal{P}_U} = \#\{(\mathcal{Q}, k, i, j) : \mathcal{Q} = \mathcal{P}_U, k = i, j = 0\}, \quad (2)$$

$$N_{\mathcal{P}_1} = \#\{(\mathcal{Q}, k, i, j) : \mathcal{Q} = \mathcal{P}_1, k = i, j = 1\}, \quad (3)$$

163 then the empirical success probability can be computed as

$$p_{\text{succ}}(\mathcal{P}_U, \mathcal{P}_1) = \frac{N_{\mathcal{P}_U} + N_{\mathcal{P}_1}}{N_{\text{total}}}. \quad (4)$$

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

165 **Scheme 2.** (Direct sum)

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

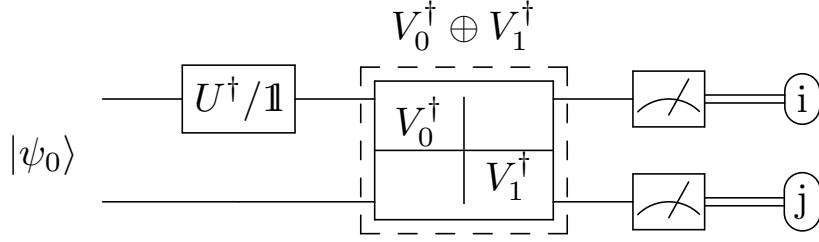


Figure 4: A schematic representation of the setup for distinguishing measurements \mathcal{P}_U and \mathcal{P}_1 using the $V_0^\dagger \oplus V_1^\dagger$ direct sum.

168 One can see why such a circuit is equivalent to the original discrimination
 169 scheme. If we rewrite the block-diagonal matrix $V_0^\dagger \oplus V_1^\dagger$ as follows:
 170

$$V_0^\dagger \oplus V_1^\dagger = |0\rangle\langle 0| \otimes V_0^\dagger + |1\rangle\langle 1| \otimes V_1^\dagger, \quad (5)$$

171 we can see that the direct sum in Eq. (5) commutes with the measurement
 172 on the first qubit. Thanks to this, we can switch the order of operations to
 173 obtain the circuit from Fig. 5. Now, depending on the outcome i , one of the
 174 summands in Eq. (5) vanishes, and we end up performing exactly the same
 175 operations as in the original scheme.

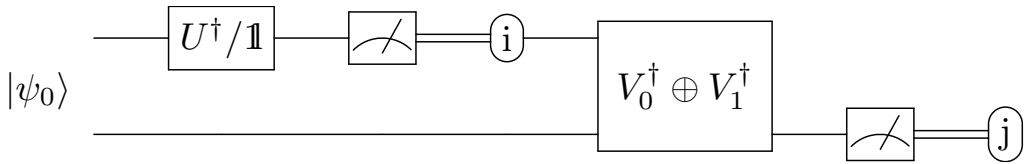


Figure 5: Rewritten representation of the setup for distinguishing measurements \mathcal{P}_U and \mathcal{P}_1 using the $V_0^\dagger \oplus V_1^\dagger$ direct sum.

176 In this scheme, the experiment can be characterized by a pair (\mathcal{Q}, i, j) ,
 177 where $\mathcal{Q} = \{\mathcal{P}_U, \mathcal{P}_1\}$ and $i, j \in \{0, 1\}$ are the output labels. The number of
 178 successful trials for U and $\mathbb{1}$, respectively, can be written as

$$N_{\mathcal{P}_U} = \#\{(\mathcal{Q}, i, j) : \mathcal{Q} = \mathcal{P}_U, j = 0\}, \quad (6)$$

$$N_{\mathcal{P}_1} = \#\{(\mathcal{Q}, i, j) : \mathcal{Q} = \mathcal{P}_1, j = 1\}. \quad (7)$$

179 Then, the probability of correct discrimination between \mathcal{P}_U and \mathcal{P}_1 is given
 180 by

$$p_{\text{succ}} = \frac{N_{\mathcal{P}_U} + N_{\mathcal{P}_1}}{N_{\text{total}}}, \quad (8)$$

181 where N_{total} is the number of trials.

182 Compared to these approaches, our approach allows for a very simple
 183 operational interpretation of the scheme and its results. This is especially
 184 useful for newcomers to the field, who may be put off by more complicated
 185 approaches. Another benefit is, especially for advanced users, the ability to
 186 control resources utilized during the benchmarking. We can consider here
 187 resources such as entanglement or coherence. Finally, the figure of merit we
 188 wish to calculate and to which we compare the results is fairly simple to
 189 obtain. The main downside of our approach is the exponential number of
 190 circuits we need to consider.

191 4. Software description

192 This section is divided into two parts. In Section 4.1 we describe func-
 193 tionalities of PyQBench package. Next, in Section 4.2, we give a general
 194 overview of the software architecture.

195 4.1. Software Functionalities

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

- 202 1. Assembling circuits for both postselection and direct sum-based dis-
 203 crimination schemes.
- 204 2. Executing the whole benchmarking scenario on specified backend (ei-
 205 ther real hardware or software simulator).
- 206 3. Interpreting the obtained outputs in terms of discrimination probab-
 207 ities.

208 Note that the execution of circuits by PyQBench is optional. Instead, the
 209 user might want to opt in for fine-grained control over the execution of the
 210 circuits. For instance, suppose the user wants to simulate the discrimination
 211 experiment on a noisy simulator. In such a case, they can define the necessary
 212 components and assemble the circuits using PyQBench. The circuits can

213 then be altered, e.g. to add noise to particular gates, and then run using any
 214 Qiskit backend by the user. Finally, PyQBench can be used to interpret the
 215 measurements to obtain discrimination probability.

216 The PyQBench library also contains a readily available implementation
 217 of all necessary components needed to run discrimination experiments for
 218 parametrized Fourier family of measurements (see Section 3 in supplemental
 219 material). However, if one only wishes to use this particular family of mea-
 220 surements in their benchmarks, then using PyQBench as a command line
 221 tool might be more straightforward. PyQBench’s command line interface
 222 allows running the benchmarking process without writing Python code. The
 223 configuration of CLI is done by YAML [40] files describing the benchmark
 224 to be performed and the description of the backend on which the benchmark
 225 should be run. The same benchmark can be used with different backends
 226 and vice versa.

227 4.2. Software Architecture

228 4.2.1. Overview of the software structure

229 As already described, PyQBench can be used both as a library and a CLI.
 230 Both functionalities are implemented as a part of `qbench` Python package.
 231 The exposed CLI tool is also named `qbench`. For brevity, we do not discuss
 232 the exact structure of the package here, and instead limit ourselves to sum-
 233 marizing the architecture on the diagram in Fig. 6. For further details, we
 234 refer an interested reader to the source code available at GitHub [41] or at
 235 the reference manual [42].

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

240 4.2.2. Integration with hardware providers and software simulators

241 PyQBench is built around the Qiskit [11] ecosystem. Hence, both the
 242 CLI tool and the `qbench` library can use any Qiskit-compatible backend.
 243 This includes, IBM Q backends (available by default in Qiskit) and Amazon
 244 Braket devices and simulators (available through `qiskit-braket-provider`
 245 package [43, 44]).

246 When using PyQBench as library, instances of Qiskit backends can be
 247 passed to functions that expect them as parameters. However, in CLI mode,
 248 the user has to provide a YAML file describing the backend. An example
 249 of such file can be found in Section 5, and the detailed description of the
 250 expected format can be found at PyQBench’s documentation.

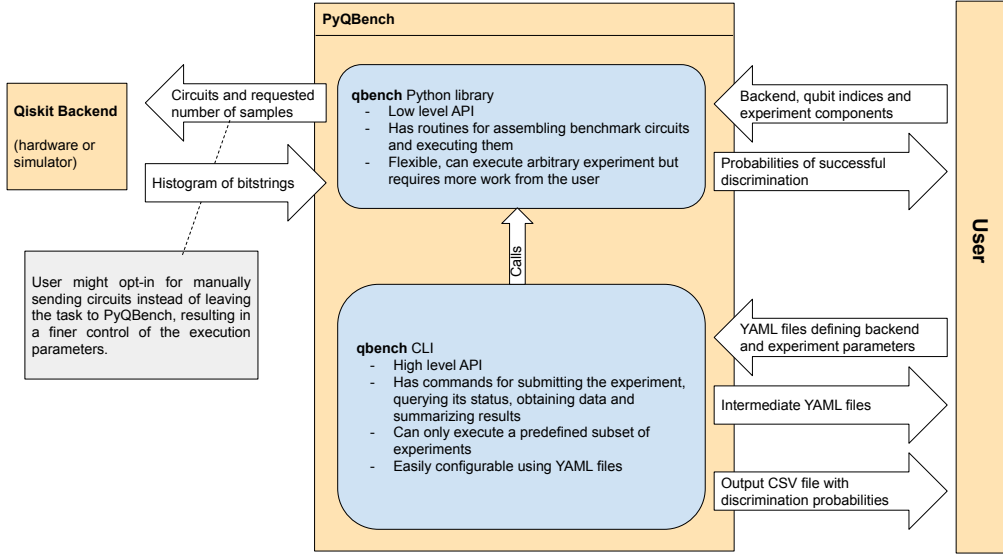


Figure 6: Overview of PyQBench's architecture.

251 4.2.3. Command Line Interface

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

Listing 1: Invocation of `qbench` script

```

254
255 qbench <benchmark-type> <command> <parameters>
256

```

257 Currently, PyQBench's CLI supports only one type of benchmark (discrimi-
 258 nation of parametrized Fourier family of measurements), but we decided on
 259 hierarchically structuring the CL to allow for future extensions. Thus, the
 260 only accepted value of `<benchmark-type>` is `disc-fourier`. The `qbench`
 261 `disc-fourier` command has four subcommands:

- 262 • **benchmark**: run benchmarks. This creates either a result YAML file
 263 containing the measurements or an intermediate YAML file for asyn-
 264 chronous experiments.
- 265 • **status**: query status of experiments submitted for given benchmark.
 266 This command is only valid for asynchronous experiments.
- 267 • **resolve**: query the results of asynchronously submitted experiments
 268 and write the result YAML file. The output of this command is almost
 269 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 [42], 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

```

345
346 name: ibmq_quito
347 asynchronous: false
348 provider:
349   hub: ibm-q
350   group: open
351   project: main
352

```

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

356 5.1.2. Running the experiment and collecting measurements data

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

```

359 qbench disc-fourier benchmark experiment_file.yml backend_file.yml
360
361

```

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

```

364 qbench disc-fourier benchmark experiment_file.yml backend_file.yml
365 --output async_results.yml
366
367

```

368 The result of running the above command can be twofold:

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

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

```

377 data:
378 - target: 0
379   ancilla: 1
380   phi: 0.0
381   results_per_circuit:
382   - name: id
383     histogram: {'00': 28, '01': 26, '10': 21, '11': 25}
384

```

```

385 mitigation_info:
386   target: {prob_meas0_prep1: 0.052200000000000024,
387           prob_meas1_prep0: 0.0172}
388   ancilla: {prob_meas0_prep1: 0.059000000000000005,
389            prob_meas1_prep0: 0.0202}
390   mitigated_histogram: {'00': 0.2637212373658018, '01':
391                        0.25865061319892463, '10': 0.2067279352110304, '11':
392                        0.2709002142242433}
393

```

394 5.1.3. (Optional) Getting status of asynchronous jobs

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

```

397
398 qbench disc-fourier status async_results.yml
399

```

400 and it will display dictionary with histogram of statuses.

401 5.1.4. Resolving asynchronous jobs

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

```

405
406 qbench disc-fourier resolve async_results.yml resolved.yml
407

```

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

413 5.1.5. Computing probabilities

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

```

416
417 qbench disc-fourier tabulate results.yml results.csv
418

```

419 A sample CSV file is provided in Table 2.

420 6. Impact

421 With the surge of availability of quantum computing architectures in
422 recent years it becomes increasingly difficult to keep track of their relative
423 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 [45, 46], if it was applied.

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

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

437 Finally, we provide the source code of PyQBench on GitHub [41] under an
438 open source license which will allow users to utilize and extend our package
439 in their specific applications.

440 7. Conclusions

441 In this paper, we presented a Python library PyQBench, an innovative
442 open-source framework for benchmarking gate-based quantum computers.
443 PyQBench can benchmark NISQ devices by verifying their capability of dis-
444 criminating between two von Neumann measurements. PyQBench offers a
445 simplified, ready-to-use, command line interface (CLI) for running bench-
446 marks using a predefined parameterized Fourier family of measurements. For

447 more advanced scenarios, PyQBench offers a way of employing user-defined
448 measurements instead of predefined ones.

449 8. Conflict of Interest

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

453 Acknowledgements

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

461 References

- 462 [1] John Preskill. Quantum computing in the nisc era and beyond. *Quan-*
463 *tum*, 2:79, 2018.
- 464 [2] Rigetti computing. <https://www.rigetti.com/>. Accessed on 2023-02-
465 18.
- 466 [3] IBM Quantum. <https://www.ibm.com/quantum>. Accessed on 2023-02-
467 18.
- 468 [4] Oxford quantum. <http://oxfordquantum.org/>. Accessed on 2023-02-
469 18.
- 470 [5] IonQ. <https://ionq.com/>. Accessed on 2023-02-18.
- 471 [6] Xanadu. <https://www.xanadu.ai/>. Accessed on 2023-02-18.
- 472 [7] D-Wave Systems. <https://www.dwavesys.com/>. Accessed on 2023-02-
473 18.
- 474 [8] QuEra. <https://www.quera.com/>. Accessed on 2023-02-18.
- 475 [9] QCS Documentation. <https://docs.rigetti.com/qcs/>. Accessed on
476 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] Alexander J McCaskey, Dmitry I Lyakh, Eugene F Dumitrescu, Sarah S Powers, and Travis S Humble. XACC: a system-level software infrastructure for heterogeneous quantum-classical computing. *Quantum Science and Technology*, 5(2):024002.
- [16] The CUDA Quantum development team. CUDA Quantum. <https://github.com/NVIDIA/cuda-quantum>. Accessed on 2023-08-27.
- [17] J. Preskill. Quantum computing 40 years later. *arXiv preprint arXiv:2106.10522*, 2021.
- [18] 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.
- [19] 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.
- [20] J. J. Wallman and S. T. Flammia. Randomized benchmarking with confidence. *New Journal of Physics*, 16(10):103032, 2014.
- [21] J. Helsen, I. Roth, E. Onorati, A. H. Werner, and J. Eisert. General framework for randomized benchmarking. *PRX Quantum*, 3(2):020357, 2022.
- [22] A. Cornelissen, J. Bausch, and A. Gilyén. Scalable benchmarks for gate-based quantum computers. *arXiv preprint arXiv:2104.10698*, 2021.
- [23] 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.

- [24] 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.
- [25] E. Pelofske, A. Bäertschi, and S. Eidenbenz. Quantum volume in practice: What users can expect from nisq devices. *IEEE Transactions on Quantum Engineering*, 3:1–19, 2022.
- [26] N. Quetschlich, L. Burgholzer, and R. Wille. Mqt bench: Benchmarking software and design automation tools for quantum computing. *arXiv preprint arXiv:2204.13719*, 2022.
- [27] MQTBench. <https://github.com/cda-tum/MQTBench>. Accessed on 2023-02-18.
- [28] 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.
- [29] SupermarQ. <https://github.com/SupertechLabs/SupermarQ>. Accessed on 2023-02-18.
- [30] Qiskit benchmarks. <https://github.com/qiskit-community/qiskit-benchmarks>. Accessed on 2023-02-18.
- [31] Forest Benchmarking: QCVV using PyQuil. <https://github.com/rigetti/forest-benchmarking>. Accessed on 2023-02-18.
- [32] A Yu Chernyavskiy and Yu I Bogdanov. Quantum tomography benchmarking. *Quantum Information Processing*, 20:1–20, 2021.
- [33] Quantum tomography benchmarking. <https://github.com/PQCLab/pyQTB>. Accessed on 2023-08-29.
- [34] Tirthak Patel, Abhay Potharaju, Baolin Li, Rohan Basu Roy, and Devesh Tiwari. Experimental evaluation of nisq quantum computers: Error measurement, characterization, and implications. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

- [35] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasm-bench: A low-level quantum benchmark suite for nisq evaluation and simulation. *ACM Transactions on Quantum Computing*, 4(2):1–26, 2023.
- [36] Quantum volume in practice. <https://github.com/lanl/Quantum-Volume-in-Practice>. Accessed on 2023-02-18.
- [37] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, 14(6):595–600, 2018.
- [38] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.
- [39] Z. Puchała, Ł. Paweł, A. Krawiec, and R. Kukulski. Strategies for optimal single-shot discrimination of quantum measurements. *Physical Review A*, 98(4):042103, 2018.
- [40] YAML Ain’t Markup Language (YAML) Version 1.2. <https://yaml.org/spec/1.2.2/>. Accessed on 2023-02-18.
- [41] PyQBench GitHub Repository. <https://github.com/iitis/PyQBench>. Accessed on 2023-02-18.
- [42] PyQBench Documentation. <https://pyqbench.readthedocs.io/en/latest/>. Accessed on 2023-02-18.
- [43] 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.
- [44] Qiskit Braket Provider GitHub Repository. <https://github.com/qiskit-community/qiskit-braket-provider>. Accessed on 2023-02-18.
- [45] mthree Documentation. <https://qiskit.org/documentation/partners/mthree/stubs/mthree.M3Mitigation.html>. Accessed on 2023-02-10.

578 [46] P. D. Nation, H. Kang, N. Sundaresan, and J. M. Gambetta. Scalable
579 mitigation of measurement errors on quantum computers. *PRX Quan-*
580 *tum*, 2(4):040326, 2021.