

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,
Balttycka 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]

12 computers can be accessed through Qiskit [11] or IBM Quantum Experience
13 web interface [12]. Some cloud services, like Amazon Braket [13], offer ac-
14 cess to several quantum devices under a unified API. On top of that, several
15 libraries and frameworks can integrate with multiple hardware vendors. Ex-
16 amples of such frameworks include IBM Q’s Qiskit or Zapata Computing’s
17 Orquestra [14].

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

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

33 2. Existing benchmarking methodologies and software

34 Unsurprisingly, PyQBench is not the only software package for bench-
35 marking gate-based devices. While we believe that our approach has signif-
36 icant benefits over other benchmarking techniques, for completeness, in this
37 section we discuss some of the currently available similar software.

38 Probably the simplest benchmarking method one could devise is simply
39 running known algorithms and comparing outputs with the expected ones.
40 Analyzing the frequency of the correct outputs, or the deviation between
41 actual and expected outputs distribution provides then a metric of the per-
42 formance of a given device. Libraries such as Munich Quantum Toolkit
43 (MQT) [24, 25] or SupermarQ [26, 27] contain benchmarks leveraging mul-
44 tiple algorithms, such as Shor’s algorithm or Grover’s algorithm. Despite
45 being intuitive and easily interpretable, such benchmarks may have some
46 problems. Most importantly, they assess the usefulness of a quantum device
47 only for a very particular algorithm, and it might be hard to extrapolate
48 their results to other algorithms and applications. For instance, the inability
49 of a device to consistently find factorizations using Shor’s algorithms does
50 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 $\{|u_0\rangle\langle u_0|, \dots, |u_{d-1}\rangle\langle u_{d-1}|\}$, 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.

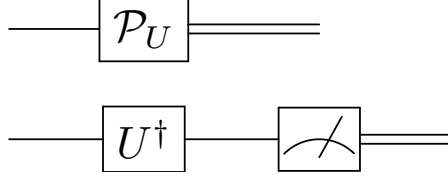


Figure 1: Implementation of a von Neumann measurement using measurement in computational basis. The upper circuit shows a symbolic representation of a von Neumann measurement \mathcal{P}_U . The bottom, equivalent circuit depicts its decomposition into a change of basis followed by measurement in the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

86 3.2. Discrimination scheme

87 Benchmarks in PyQBench work by experimentally determining the prob-
88 ability of correct discrimination between two von Neumann measurements
89 by the device under test and comparing the result with the ideal, theoretical
90 predictions.

91 Without loss of generality¹, we consider discrimination task between sin-
92 gle qubit measurements \mathcal{P}_1 , performed in the computational Z -basis, and an
93 alternative measurement \mathcal{P}_U performed in the basis U . The discrimination
94 scheme presented in Fig. 2 requires an auxiliary qubit. First, the joint system
95 is prepared in some state $|\psi_0\rangle$. Then, one of the measurements, either \mathcal{P}_U
96 or \mathcal{P}_1 , is performed on the first part of the system. Based on its outcome
97 i , we choose another binary measurement \mathcal{P}_{V_i} and perform it on the second
98 qubit, obtaining the outcome j . Finally, if $j = 0$, we say that the performed
99 measurement is \mathcal{P}_U , otherwise we say that it was \mathcal{P}_1 .

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

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

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

¹Explaining why we can consider only discrimination scheme between \mathcal{P}_1 and \mathcal{P}_U is beyond the scope of this paper. See [31] for a in depth explanation.

113 $|\psi_0\rangle$ and final measurements \mathcal{P}_{V_i} . However, we argue that it is best to choose
 114 those components so that they maximize the probability of correct discrim-
 115 ination. To see that, suppose that some choice of $|\psi_0\rangle, \mathcal{P}_{V_0}, \mathcal{P}_{V_1}$ allows for
 116 correctly discriminating between two measurements with probability equal
 117 to one, i.e. on a perfect quantum computer you will always make a cor-
 118 rect guess. Then, on real hardware, we might obtain any empirical value in
 119 range $[\frac{1}{2}, 1]$. On the other hand, if we choose the components of our scheme
 120 such that the successful discrimination probability is $\frac{3}{5}$, the possible range
 121 of empirically obtainable probabilities is only $[\frac{1}{2}, \frac{3}{5}]$. Hence, in the second
 122 case, the discrepancy between theoretical and empirical results will be less
 123 pronounced.

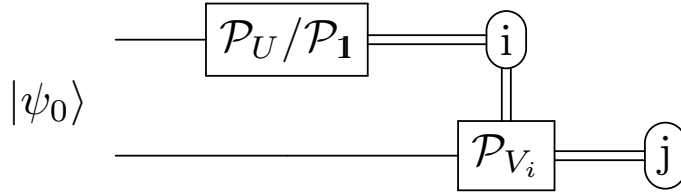


Figure 2: Theoretical scheme of discrimination between von Neumann measurements \mathcal{P}_U and \mathcal{P}_1 .

124 3.2.1. Implementation of discrimination scheme on actual NISQ devices

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

130 **Scheme 1.** (Postselection)

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

137 To perform the benchmark, one needs to run multiple copies of the post-
 138 selection circuit, with both \mathcal{P}_U and \mathcal{P}_1 . Each circuit has to be run in both
 139 variants, one with final measurement \mathcal{P}_{V_0} and the second with the final mea-
 140 surement \mathcal{P}_{V_1} . The experiments can thus be grouped into classes identified by
 141 tuples of the form (\mathcal{Q}, k, i, j) , where $\mathcal{Q} \in \{\mathcal{P}_U, \mathcal{P}_1\}$ denotes the chosen mea-
 142 surement, $k \in \{0, 1\}$ designates the final measurement used, and $i \in \{0, 1\}$

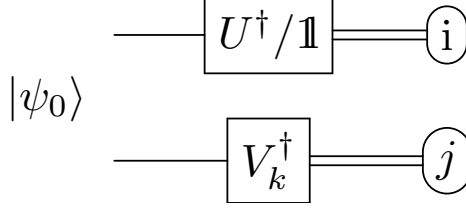


Figure 3: A schematic representation of the setup for distinguishing measurements \mathcal{P}_U and \mathcal{P}_1 using postselection approach. In postselection scheme, one runs such circuits for both $k = 0, 1$ and discards results for cases when there is a mismatch between k and i .

143 and $j \in \{0, 1\}$ being the labels of outcomes as presented in Fig. 3. We
 144 then discard all the experiments for which $i \neq k$. The total number of valid
 145 experiments is thus:

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

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

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

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

149 then the empirical success probability can be computed as

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

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

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

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

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

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

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



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.

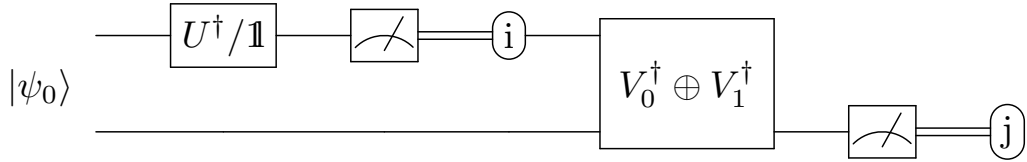


Figure 5: Rewritten representation of the setup for distinguishing measurements \mathcal{P}_U and \mathcal{P}_1 using the $V_0^\dagger \oplus V_1^\dagger$ direct sum.

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

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

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

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

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

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

168 Compared to these approaches, our approach allows for very simple operational
 169 interpretation of the scheme and its results. This is especially useful for
 170 newcomers to the field, who may be put off by more complicated approaches.
 171 Another benefit is, especially for advanced users, the ability to control resources
 172 utilized during the benchmarking. We can consider here resources such as
 173 entanglement or coherence. Finally, the figure of merit we wish calculate
 174 and to which we compare are results is fairly simple to obtain. The main
 175 downside of our approach is the exponential number of circuits we need to
 176 consider.

177 4. Software description

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

181 4.1. Software Functionalities

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

- 188 1. Assembling circuits for both postselection and direct sum-based dis-
189 crimination schemes.
- 190 2. Executing the whole benchmarking scenario on specified backend (ei-
191 ther real hardware or software simulator).
- 192 3. Interpreting the obtained outputs in terms of discrimination probab-
193 ities.

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

202 The PyQBench library also contains a readily available implementation
203 of all necessary components needed to run discrimination experiments for
204 parametrized Fourier family of measurements (see Section 3 in supplemental
205 material). However, if one only wishes to use this particular family of mea-
206 surements in their benchmarks, then using PyQBench as a command line
207 tool might be more straightforward. PyQBench’s command line interface
208 allows running the benchmarking process without writing Python code. The
209 configuration of CLI is done by YAML [32] files describing the benchmark
210 to be performed and the description of the backend on which the benchmark
211 should be run. The same benchmark can be used with different backends
212 and vice versa.

213 4.2. Software Architecture

214 4.2.1. Overview of the software structure

215 As already described, PyQBench can be used both as a library and a CLI.
216 Both functionalities are implemented as a part of `qbench` Python package.
217 The exposed CLI tool is also named `qbench`. For brevity, we do not discuss
218 the exact structure of the package here, and instead refer an interested reader
219 to the source code available at GitHub [33] or at the reference manual [34].

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

224 4.2.2. Integration with hardware providers and software simulators

225 PyQBench is built around the Qiskit [11] ecosystem. Hence, both the
226 CLI tool and the `qbench` library can use any Qiskit-compatible backend.
227 This includes, IBM Q backends (available by default in Qiskit) and Amazon
228 Braket devices and simulators (available through `qiskit-braket-provider`
229 package [35, 36]).

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

235 4.2.3. Command Line Interface

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

Listing 1: Invocation of `qbench` script

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

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

- 246 • **benchmark**: run benchmarks. This creates either a result YAML file
247 containing the measurements or an intermediate YAML file for asyn-
248 chronous experiments.

- 249 • **status**: query status of experiments submitted for given benchmark.
250 This command is only valid for asynchronous experiments.
- 251 • **resolve**: query the results of asynchronously submitted experiments
252 and write the result YAML file. The output of this command is almost
253 identical to the result obtained from synchronous experiments.
- 254 • **tabulate**: interpret the results of a benchmark and summarize them
255 in the CSV file.

256 We present usage of each of the above commands later in section 5.

257 4.2.4. *Asynchronous vs. synchronous execution*

258 PyQBench’s CLI can be used in synchronous and asynchronous modes.
259 The mode of execution is defined in the YAML file describing the backend
260 (see Section 5 for an example of this configuration). We decided to couple
261 the mode of execution to the backend description because some backends
262 cannot work in asynchronous mode.

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

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

277 5. Illustrative examples

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

285 5.1. Using *qbench* CLI

286 PyQBench offers a simplified way of conducting benchmarks using a Com-
287 mand Line Interface (CLI). The workflow with PyQBench’s CLI can be sum-
288 marized as the following list of steps:

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

297 5.1.1. Preparing configuration files

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

Listing 2: Defining the experiment

```
304 

---


305 type: discrimination-fourier
306 qubits:
307   - target: 0
308     ancilla: 1
309   - target: 1
310     ancilla: 2
311 angles:
312   start: 0
313   stop: 2 * pi
314   num_steps: 3
315 gateset: ibmq
316 method: direct_sum
317 num_shots: 100
318 

---


```

319 The second configuration file describes the backend. We decided to de-
320 couple the experiment and the backend files because it facilitates their reuse.
321 For instance, the same experiment file can be used to run benchmarks on
322 multiple backends, and the same backend description file can be used with
323 multiple experiments.

324 Different Qiskit backends typically require different data for their initial-
 325 ization. Hence, there are multiple possible formats of the backend config-
 326 uration files understood by PyQBench. We refer the interested reader to
 327 the PyQBench’s documentation. Below we describe an example YAML file
 328 describing IBM Q backend named Quito.

Listing 3: IBMQ backend

```
329
330 name: ibmq_quito
331 asynchronous: false
332 provider:
333   hub: ibm-q
334   group: open
335   project: main
```

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

340 5.1.2. Running the experiment and collecting measurements data

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

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

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

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

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

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

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

```
361 data:
```

```

363 - target: 0
364   ancilla: 1
365   phi: 0.0
366   results_per_circuit:
367     - name: id
368     histogram: {'00': 28, '01': 26, '10': 21, '11': 25}
369     mitigation_info:
370       target: {prob_meas0_prep1: 0.052200000000000024,
371               prob_meas1_prep0: 0.0172}
372       ancilla: {prob_meas0_prep1: 0.059000000000000005,
373                prob_meas1_prep0: 0.0202}
374     mitigated_histogram: {'00': 0.2637212373658018, '01':
375                           0.25865061319892463, '10': 0.2067279352110304, '11':
376                           0.2709002142242433}
377

```

378 5.1.3. (Optional) Getting status of asynchronous jobs

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

```

381
382 qbench disc-fourier status async_results.yml
383

```

384 and it will display dictionary with histogram of statuses.

385 5.1.4. Resolving asynchronous jobs

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

```

389
390 qbench disc-fourier resolve async-results.yml resolved.yml
391

```

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

397 5.1.5. Computing probabilities

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

```

400
401 qbench disc-fourier tabulate results.yml results.csv
402

```

403 A sample CSV file is provided in Table 2.

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.

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 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.

424 7. Conclusions

425 In this paper, we presented a Python library PyQBench, an innovative
426 open-source framework for benchmarking gate-based quantum computers.
427 PyQBench can benchmark NISQ devices by verifying their capability of dis-
428 criminating between two von Neumann measurements. PyQBench offers a
429 simplified, ready-to-use, command line interface (CLI) for running bench-
430 marks using a predefined parameterized Fourier family of measurements. For
431 more advanced scenarios, PyQBench offers a way of employing user-defined
432 measurements instead of predefined ones.

433 8. Conflict of Interest

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

437 Acknowledgements

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

445 References

- 446 [1] John Preskill. Quantum computing in the nisq era and beyond. *Quan-*
447 *tum*, 2:79, 2018.
- 448 [2] Rigetti computing. <https://www.rigetti.com/>. Accessed on 2023-02-
449 18.
- 450 [3] IBM Quantum. <https://www.ibm.com/quantum>. Accessed on 2023-02-
451 18.
- 452 [4] Oxford quantum. <http://oxfordquantum.org/>. Accessed on 2023-02-
453 18.
- 454 [5] IonQ. <https://ionq.com/>. Accessed on 2023-02-18.

- 455 [6] Xanadu. <https://www.xanadu.ai/>. Accessed on 2023-02-18.
- 456 [7] D-Wave Systems. <https://www.dwavesys.com/>. Accessed on 2023-02-18.
- 457
- 458 [8] QuEra. <https://www.quera.com/>. Accessed on 2023-02-18.
- 459 [9] QCS Documentation. <https://docs.rigetti.com/qcs/>. Accessed on 2023-02-18.
- 460
- 461 [10] PyQuil Documentation. <https://pyquil-docs.rigetti.com/en/stable/>. Accessed on 2023-02-18.
- 462
- 463 [11] Qiskit. <https://qiskit.org/>. Accessed on 2023-02-18.
- 464 [12] IBM Quantum Experience. <https://quantum-computing.ibm.com/>. Accessed on 2023-02-18.
- 465
- 466 [13] Amazon Braket. <https://aws.amazon.com/braket/>. Accessed on 2023-02-18.
- 467
- 468 [14] Zapata Orchestra Platform. <https://www.zapatacomputing.com/orchestra-platform/>. Accessed on 2023-02-18.
- 469
- 470 [15] J. Preskill. Quantum computing 40 years later. *arXiv preprint arXiv:2106.10522*, 2021.
- 471
- 472 [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.
- 473
- 474
- 475 [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.
- 476
- 477
- 478 [18] J. J. Wallman and S. T. Flammia. Randomized benchmarking with confidence. *New Journal of Physics*, 16(10):103032, 2014.
- 479
- 480 [19] J. Helsen, I. Roth, E. Onorati, A. H. Werner, and J. Eisert. General framework for randomized benchmarking. *PRX Quantum*, 3(2):020357, 2022.
- 481
- 482
- 483 [20] A. Cornelissen, J. Bausch, and A. Gilyén. Scalable benchmarks for gate-based quantum computers. *arXiv preprint arXiv:2104.10698*, 2021.
- 484

- [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. Bäertschi, 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. Visslai, 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. Paweł, A. Krawiec, and R. Kukulski. Strategies for optimal single-shot discrimination of quantum measurements. *Physical Review A*, 98(4):042103, 2018.

- 518 [32] YAML Ain't Markup Language (YAML) Version 1.2. <https://yaml.org/spec/1.2.2/>. Accessed on 2023-02-18.
- 519
- 520 [33] PyQBench GitHub Repository. <https://github.com/iitis/PyQBench>. Accessed on 2023-02-18.
- 521
- 522 [34] PyQBench Documentation. <https://pyqbench.readthedocs.io/en/latest/>. Accessed on 2023-02-18.
- 523
- 524 [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.
- 525
- 526
- 527
- 528 [36] Qiskit Braket Provider GitHub Repository. <https://github.com/qiskit-community/qiskit-braket-provider>. Accessed on 2023-02-18.
- 529
- 530
- 531 [37] mthree Documentation. <https://qiskit.org/documentation/partners/mthree/stubs/mthree.M3Mitigation.html>. Accessed on 2023-02-10.
- 532
- 533
- 534 [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.
- 535
- 536