QX Laboratory - Documentation

T. Rybotycki

VII 2019

Contents

1 Change Log

1. XII 2020: Update to Qiskit 0.23 (noisy simulation API changed)
2. XI 2020: Language correction
3. XII 2019: New functionalities update
4. VII 2019: Initial Version

2 Introduction

QX Laboratory (or QX Lab) is a project that started as Python software framework [4] / adapter [1] for IBM's Qiskit [3], which again is a quantum computing framework. Initially, QX Lab was meant to provide two main functionalities – performing experiments in the loop and wrapping Qiskit methods to endure Qiskits' frequent updates.

IBM limits the number of experiments that we can perform at once. The devices are usually busy, as the access to them is opened for anyone willing to use them. To gather a large number of samples, we needed the experiments loop. It allowed us to enqueue a new experiment as soon as possible, thus significantly increasing the quantity of data we obtain. Moreover, it automated the whole process, so we didn't have to send new jobs to IBM manually.

Qiskit is a framework that's still being rapidly developed, and as such, it usually experiences various changes from version to version. Although nowadays (from the author's observations, it'd be post version 0.7), the updates don't change Qiskit API (application programming interface), it wasn't always the case. There was a time when the updates changed even the most basic methods provided by Qiskit (such as the job executing method). It essentially required updating every script that used Qiskit. The idea to solve this problem was wrapping these common functions with one's own, and therefore in case of an API-changing update, only one script (containing wrapped methods) would have to be fixed. This approach proved to work very well in practice.

Currently, QX Lab is not only a methods wrapper with a loop. It includes several utilities that we meant to help during experiments. It now offers a variety of additional functionalities – from testing experiment scripts locally (with or without noise) through gathering jobs (possibly with error mitigation application) up to monitoring backends and jobs in the queue. It's also continuously updated, as it's used for research by the creators and thus will be up to date with new Qiskit releases almost all of the time (hopefully). Usually, using a selected tool requires one additional line of code or calling a script. One should also point out that we add new functionalities when a need arises.

We organize the rest of the documentation as follows. First, we propose the exemplary experiment's steps, utilizing the framework's functionalities. In the following section, we describe the configuration files for experiments. We present the tools provided by the QX lab in more detail. Then, we give a brief description of wrapped / additional methods. Finally, a list of all the experiments scripts available and wrote using framework methods are listed and briefly described. One should note that we've filled the project readme files, which one should read to get more detailed information.

The QX Lab project is available on Github (`https://github.com/Tomev/IBMQE`).

3 Sample Experiment's Steps

To run a generic experiment one should follow these steps

1. Write experiment script.
2. (Optional) Test the script.
3. Run script on remote IBM backend.

4. Wait for jobs to finish.
5. Download jobs data.
6. Results analysis.

Although QX Lab functionalities in step 1 can only provide change-proof API, in the next steps QX Lab provides several utilities that we can use to ease the process. That will be the topic of this chapter. The analysis part is experiment-specific and is out of the scope of this document.

3.1 Script testing

Script testing can be done with simply using one of two methods `test_locally(circuits, use_mapping=false)` or `test_locally_with_noise(circuits)`. Both methods require circuits to be tested – these should be inside experiment scripts that one wants to test. Test methods are based on `qasm_simulator` [5] provided by IBM.

The first method has two variants. First describes an ideal situation wherein all qubits are interconnected so one can use it as a control for each other qubit in multi-qubit gates. The second variant, accessed by setting `use_mapping` to `true`, is more specific and uses qubit connection scheme (or `coupling_map` if one will) for specified (in consts.py file, which we will describe later) remote backend, one may say it creates a simulator of this chip and performs an experiment on it. When testing one's script before running it on a real backend, it's good to use the second variant of this method to determine whether the connection scheme within the script is violated or not.

The other method works similarly to the first one, but it also adds noise to the results. It uses Qiskit's `noise_model` [2] to create a simulator with noise that'd resemble the behavior of specified (again in config.py file) remote backend. One should use this test to determine the expected output of a script on a real device, for example, a comparison of the simulation with experiments on the backend.

Both testing methods should provide results within less than a minute.

3.2 Running the script on remote backend

Qiskit provides the *execute* method to run one's scripts on a remote backend. This method places one job with a prepared circuit in a specified backend's queue, and that's all. Note that one may have multiple jobs in the queue, but that'd require to use the method more than once. One should use a loop for optimal job execution.

A simple loop may prove not to be enough, so the QX Lab provides a method for that – `run_main_loop(circuits)`. QX Lab's main loop purpose is to execute the desired number of jobs (given by a variable in consts.py) in the most elegant and automated manner. First it creates a `current_iterations_holder.txt` file, if one isn't already there. This file is necessary for situations wherein lack of power or the internet would break the loop and without the file terminate index of currently performed execution. In such cases, one would need to gather the jobs from the start without knowing the number of jobs already sent. With `current_iterations_holder.txt` loop restarting process is not an issue anymore.

Next, the main loop monitors status of the job it tries to send to the remote backend. If the job is sent successfully, then another one is being processed. If it's not, then it waits a specified amount of time (by default, it's 5 minutes) in order not to stress the computer by pointlessly repeating the loop that's unable to send a new job to the backend due to external problems. Note that the main loop can receive multiple sets (lists) of circuits as input. In this scenario, the loop will create a job from each set and then sent one job after another. The loop will keep repeating until it sends the desired number of jobs.
The main loop gathers a list of operational backends and checks if a specified backend is present in it. If so, it outputs the information that no desired backend can be used at the time and enters the loop that continuously asks IBM whether the backend is available. That is a time-consuming process, requiring connection to IBM, and thus QX Lab doesn't implement wait time. Note that the main loop used to and still can serve multiple desired

backends, but this functionality hasn't been used for long and may not work as expected.

Lastly, if everything is going well, the job is being sent to a specified chip (or is being executed, if one will), and the `current_iterations_holder.txt`, as well as loops iteration number, is increased. The loop repeats until it sends the desired number of jobs.

3.3 Waiting

There's not much any framework can do in this matter. QX Lab, however, provides a tool that shows whether the last job sent for execution on the specified backend has already been executed. That can be an alternative way of determining if one can download the jobs or not without accessing the IBM QX web page (or using other means of checking). The author usually waits an arbitrary amount of time instead of using this tool but recognizes its potential during presentations.

3.4 Data gathering

The last step of experiments that can be aided by QX Lab is data gathering. The framework contains two methods of data gathering – *JobDataGatherer.py* and *singularJobDataGatherer.py* – and two ways of reporting them – default report and *aReport*.

In this subsection, we will summarize the data gathering methods, as we provide their detailed description within their respective readme files. They also hold information on which method one should prefer at the time (one should be aware that this may change over time). We, however, provide a more detailed description of the reports.

The main difference between both data gathering techniques is that one of them is safe but slow, and the second is greedy but fast. *JobDataGatherer.py*, was implemented first, and it's a greedy one. It downloads many jobs at once in several intervals until all the jobs (one specifies their number in consts.py) are gathered and then saves them. We set the default number of jobs that the gatherer downloads in one interval to ten, which is the current maximum value. One should be aware that previously there were some issues with connecting to IBM, and thus, during job gathering via this method, the connection tended to break, and thus the gatherer didn't download any jobs at all. Therefore, we've developed a safer method – *singularJobDataGatherer.py* – which work in the following way. It downloads a single job, writes it to the file, and continues the process until it has gathered all jobs. In the case of connection breakdown (or other issues that'd cause the script to abort), it will read the number of jobs written already in the report file and continue the loop after the restart and thus is resistant to problems of JobDataGatherer. Downloading jobs one at a time, however, requires the script to renew the connection with IBM each time it gathers a new job. That is a time-consuming process. With default settings, this method requires roughly ten times more connection requests than JobDataGatherer and thus is significantly slower than the latter, which is its major drawback. Currently, IBM has stable connections, and so JobDataGatherer is a better choice.

When the gatherer has downloaded the jobs, it generates a report. Usually, it creates two kinds of reports – standard and aReport – which consists of the same data, but their structure is different and therefore can be analyzed differently. The first kind consists of rows that contain information about circuits containing all measurements of all states for that circuit. The aReport differs from the standard in such a way that each row consists of data with only one state per row. These reports have a lot more redundant data but may prove to be easier to analyze. To generate aReport it's usually necessary to change the report file name in the *aReportConverter.py* script. Along these two additional reports are also being generated – mitigation report. It contains job results after application of error mitigation to it (which we obtain using the data from the chip at the time of job execution).

One should note that all scripts described in this subsection have their specific variables within (not only in consts.py file), so one should edit them properly before execution to achieve the desired results.

4 Config files

The project contains two config files – Qconfig.py and consts.py – which proper setting is required to perform experiments using the QX Lab. This chapter contains description of both of the files.

4.1 Qconfig.py

Qconfig.py is a legacy file. Keeping it in the project causes issues with current version (0.23) of Qiskit, therefore it should be deleted and is no logner part of this project.

4.2 Consts.py

Consts.py, as the name suggests, contains configuration constants that are used by the QX Lab methods. In this subsection these constants shall be listed and described briefly.

- SHOTS – a number of times (max 8192, default 1024) the job will be performed on the backend.
- `CONSIDERED_REMOTE_BACKENDS` – remote backends that are considered during experiments. Note that it's a list, as main loop can enqueue jobs into several back ends at once. Multi-backend approach is, however, still not compatible with all the scripts/methods and should be avoided.
- `ITERATIONS_NUMBER` – a number of the jobs one would like to have enqueued.
- `CSV_SEPARATOR` – the CSV separator in the CSV reports. Note that the comma can not be used, as it's used by qiskit in dictionaries with data.
- `JOBS_DOWNLOAD_LIMIT` – a limit of the jobs to download.
- `MAX_JOBS_SINGLE_DOWNLOAD_NUM` – number of the jobs that can be download at once. It should be lowered in case of problems (e.g. connection instability).
- `JOBS_REPORT_HEADER` – header of the reports.
- `JOBS_FILE_NAME` – default name of the report file.

5 Tools

The QX Lab provides several utility tools that are meant to help in conducting experiments and in working with the framework. The tools and their respective and brief descriptions are listed below.

1. `backend_monitor.py` – a tool used to monitor (get detailed information about) the backends. It currently outputs information about ibmqx2 and ibmqx4, however, it can be easily modified.
2. `last_job_status_checker.py` – a tool used to check last job's status. It needs name of backend, to get job from, as an argument.
3. `operational_backends_checker.py` – a tool used to determine which backends are currently operational.
4. `job_counter_reseter.py` – a tool used to reset the job counter (which can also be done manually). One should use it in case any experiment was aborted before end of main loop.
5. Job data gatherers – a set of tools (currently two) that can be used to gather the jobs. They were described in subsection 3.4.
6. `report_integrator` – integrates reports in the same folder for each job after given date. Works both with standard (mitigation) and a reports.

Using the tool requires calling it via python with simple command:

```
python script_name_py [args],
```

where args are optional arguments and are currently used only in `last_job_status_checker.py`.

6 Methods description

The heart of QX Lab lies definitely in *methods.py* file. It contains most of the methods and submethods (as the author will call functions that we don't use directly in the experiments scripts, but QX Lab uses within more complex methods of methods.py file. One may think of them as private/protected functions.) used by QX Lab. In this section, we provide methods and their respective outline.

- `get_operational_remote_backends()` – lists operational remote backends.
- `get_backends_names(backends)` – creates a list of backend names from a list of backends objects.
- `get_backend_from_name(name)` – get a backend object from its name.
- `get_sim_backend_from_name(name)` – get a backend simulator object form its name.
- `get_backend_name_from_number(number)` – legacy method. It was used in main the loop.
- `execute_circuits(circuits, backend, use_mapping=False, noise_model=None)` – executes method wrapper. It's used to execute jobs on simulators as well as real chips, thus different parameters.
- `run_main_loop(circuits)` –this method was already described in the section 3.
- `reset_jobs_counter()` – resets job counter, same as corresponding tool. It's called at the end of the main loop.
- `test_locally` – this method was already described in the section 3.
- `generate_gate_times()` – generates gates time using qiskit.aer. Required by noisy simulation.
- `test_locally_with_noise` – this method was already described in the section 3.
- `get_jobs_from_backend` – the main function of JobDataGatherer.
- `parse_job_to_report_string(job)` – default *report job object* converting function.
- `report_to_csv(csv_file, report_file=consts.JOBS_FILE_NAME, sep=consts.CSV_SEPARATOR, lowercase_header=True)` – reports saving function. It contains several technical / configuration arguments.
- `create_circuit_from_qasm(qasm_file_path)` – creates a circuit from the specified qasm file.
- `get_mitigation_report_string(job)` – returns a report string of a single job with application of error mitigation basing on quantum device data at the time of job execution.
- `add_measure_in_base(QuantumCircuit, str)` – applies the measurements given by a *str* to each qubit of a given quantum circuit.
- `custom_backend_monitor(backend)` – legacy method. Can be used to monitor a backend specified by a name. Returns the monitored data.
- `save_calibration_data(backend, data)` – legacy method. Can be used to save the calibration data specified as argument of a backend.

One should also note that *methods.py* file also provides logging into IBM QX at the end of the file (basing on current IBM requirements).

7 Experiments

Several experiments have already been written and conducted using QX Lab. We've provided their respective description within the experiment's folder, and we won't include them in this document. The list of experiments (up to date) is the following:

1. Bell,
2. CHSH,
3. Grover,
4. HelloQuantum,
5. Mermin,
6. Teleportation,
7. No signaling.

References

[1] Gamma E.; et. al. *Design Patterns, Elements of Reusable Object-Oriented Programming*. Addison-Wesley, 1994.

[2] *Qiskit documentation*. `https://qiskit.org/documentation`. Access: 28 VII 2019.

[3] *Qiskit page*. `https://qiskit.org/`. Access: 5 VI 2019.

[4] Techopedia. *Software Framework Definition*. `https://www.techopedia.com/definition/14384/software-framework`. Access: 5 VI 2019.

[5] Christopher J. Wood. *Introducing Qiskit Aer: A high performance simulator framework for quantum circuits*. `https://medium.com/qiskit/qiskit-aer-d09d0fac7759`. Access: 28 VII 2019.