QX Laboratory - Documentation

T. Rybotycki

VII 2019

Contents

1 Change Log

1. XII 2019: New functionalities update
2. VII 2019: Initial Version

2 Introduction

QX Laboratory (or QX Lab) is a project that started as Python software framework [5] / adapter [4] 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 loop and wrapping QISKit methods to endure QISKits frequent updates.

Experiments loop was necessary for proper statistics to be gathered as IBM limits number of experiments that can be performed at the time (as the access to the devices is opened for anyone willing to use them, for most of the time they are busy) and thus being able to place new experiment in the queue as soon as possible increased quantity of data gathered significantly. Moreover it automated the whole process as the user was no longer required to send new jobs to IBM manually.

QISKit is a framework that's still being rapidly developed and as such it usually experiences major changes from version to version. Although nowadays (from authors experience 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 even most basic methods provided by QISKit (such as job executing method) were changed which essentially required updating every script in order for it to work. The idea to solve this problem was wrapping these common function 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 are meant to help during experiments. It now offers variety of additional functionalities – from testing experiment scripts locally (with or without noise) through gathering jobs (possibly with error mitigation application) all the way 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 selected tool requires one additional line of code or calling a script. One should also point out that new functionalities are being added when a need arises.

Rest of the documentation is organized as follows. First a sample experiment's steps, utilizing framework's functionalities are proposed. In following section configuration files for experiments are described. Next tools provided by QX Lab are presented in more detail. Then, a brief description of wrapped / additional methods is provided. The last list all the experiments scripts available and wrote using framework methods are listed and briefly described. One should note that the project is filled with readme files which one should read in order to get more detailed information.

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

3 Sample Experiment's Steps

To conduct 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 IBM QX Lab functionalities in step 1 are only reduced to provide change-proof API, in next steps several provided utilities can be used and this will be the topic of this chapter. Analysis part is experiment-specific and is out of 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 obviously requires circuits to be tested, which should be provided by experiments scripts prepared earlier. They are based on `qasm_simulator` [6] provided by IBM.

First method has two variants. First describes ideal situation wherein all qubits are interconnected and thus can be used as control for each other qubit in multi-qubit gates. 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 will be described later) remote backend, one may say it creates a simulator of this chip and performs experiment on it. When testing one's script before sending it's good to use second variant of this method in order to determine whether connection scheme within the script is violated or not.

The other method works similarly to the first one, however it also adds noise to the results. It uses QISKit's `basic_device_noise_model` [2] to create a simulator with noise that'd resemble behavior of specified (again in config.py file) remote backend. This test should be used to determine expected output of a script on a real device for e.g. comparison of the simulation with experiments on real chip.

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

3.2 Running the script on remote backend

Normally QISKit provides some kind of *execute* method in order to run one's scripts on remote backend. This method places one job with prepared circuit in specified backend's queue and that's all. Note, than one may have multiple jobs in queue, but that'd require to use execute method more than once and thus a loop should be used for optimal job execution.

A simple loop, however, may prove not to be enough and thus, two methods were provided – one being extension of the other – `run_main_loop(circuits)` and `run_main_loop_with_chsh_test(circuits)`, wherein the difference between the first and the second method is that the second adds additional CHSH test circuits to given circuits list, which can later be used to determine whether the chip was working properly during the execution of given job (and thus enhance the results of the analysis). One should note, however, that in order to add CHSH test circuits to the experiment, circuits number in the experiments should be lower than maximum number of circuits (specified in [2]) by number of CHSH circuits (4).

QX Lab's main loop purpose is to execute 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 in 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 in not an issue anymore.

Next, main loop monitors status of the job it tries to send to remote backend. If the job is sent successfully, then another one is being processed. If it's not then it waits 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 new job to the backend due to external problems. Note that the main loop can receive multiple sets (lists) of circuits. In this scenario a job will

be created from each set and they will be send one after another, repeating from first job after last job was sent, until desired number of sent jobs is achieved.
Main loop gathers list of operational backends and checks if 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 can be used. This is a time consuming process, requiring connection to IBM, and thus no wait time was implemented here. Note, that main loop used to and still is able to serve multiple desired backends, however, 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 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 is repeated until desired number of jobs was sent.

3.3 Waiting

There's not much any framework can do in this matter. QX Lab, however, provides a tool, that shows whether last job send for execution on specified backend has been already done, which can be a good way of determining if jobs can be downloaded or not without accessing IBM QX web page (or using other means of checking). The author usually waits an arbitrary amount of time instead of using this tool, however, recognizes it's 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 data gathering methods will be summarized briefly, as their detailed description is provided within readme file, which also consists of information which method is preferred at the time (one should be aware, that this may change over time). More detailed description of reports will, however, be provided.

The main difference between both data gathering techniques is that one of them is safe, but slow, and the second is greedy, but faster. *JobDataGatherer.py*, was implemented first and it's a greedy one. In download many jobs at once in several intervals, until all the jobs (their number is specified in consts.py) are gathered and then saves them. Default number of jobs gathered in one interval is set to 10 which is 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 no jobs were downloaded at all. Therefore safer method was developed – *singularJobDataGatherer.py* – which work in following way. It downloads a single job, writes it to the file, and continues the process, until all jobs are gathered. In case of connection breakdown (or other issues that'd cause the script to abort) it will read the number of jobs already in the report file, and continue the loop after restart and thus is resistant to problems of JobDataGatherer. Downloading jobs one at the time, however, requires the script to renew connection with IBM each time new job is being gathered which is time consuming process. With default settings this method requires roughly 10 times more connections than JobDataGatherer and thus is significantly slower, that the latter, which is it's major drawback. Currently IBM has stable connections and so JobDataGatherer is better choice.

When jobs are gathered, a report is generated. Usually two kinds of reports are generated – standard report and aReport – which consists of same data, but their structure is different and therefore can be analysed differently. Standard report consists of rows, that contains information about circuits containing all measurements of all states for that circuit. The aReport – named after a person, that asked for such report structure – differs from the first on in such way, that each row consists of information with only one state per row. These reports have a lot more redundant data, but may prove to be easier to analyse. In order to generate aReport it's usually necessary to change report file name in the *aReportConverter.py* script. Along these two additional report is also being generated – mitigation report. It contains job results after application of error mitigation to it (which is obtained using data from chip at the time when the job was being executed).

One should note, that all scripts described in this subsection has their specific variables within (not only in consts.py file) and therefore should be edited properly before execution in order to achieve 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 QX Lab as one would want to. This chapter contains description of both of the files.

4.1 Qconfig.py

QConfig.py is a legacy file, however, it's still very important. It's not included in the QX Lab repository as it's used to store sensitive info. It's sample file, however, is provided. It contains following code.

```
# Input your API Token here and remove ".sample" from file's name.

APItoken = 'Your_API_Token'

if 'APItoken' not in locals():
        raise Exception('Please set up your access token. See Qconfig.py.')
```

For IBM QX Lab to properly function in the main project folder one should create Qconfig.py file (basing on the sample) with one's own API Token, which can be generated via [1].

4.2 Consts.py

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

- SHOTS – number of times (max 8192, default 1024) job will be performed on the backend.
- `CONSIDERED_REMOTE_BACKENDS` – remote backends 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` – number of jobs one would like to be enqueued.
- `CSV_SEPARATOR` – CSV separator in report CSV. Note, that comma can not be used, as it's used by qiskit in dictionaries with data.
- `JOBS_DOWNLOAD_LIMIT` – limit of jobs to download.
- `MAX_JOBS_SINGLE_DOWNLOAD_NUM` – number of jobs that can be download at once. It should be lowered in case of problems.
- `JOBS_REPORT_HEADER` – header of reports.
- `JOBS_FILE_NAME` – default name of report file.

5 Tools

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 tools used to monitor (get detailed information about) backends. It currently outputs information about ibmqx2 and ibmqx4, however, it can be easily modified.

2. `last_job_status_checker.py` – a tool uses to check last job's status. It needs name of backend to get job from as a 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 job counter. It 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 jobs. They were described in subsection 3.4 in more detail.
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 by `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 are not directly used in experiments scripts, but are used within more complex methods of methods.py file. One may think of them as kind of private / protected functions.) used by QX Lab. In this section methods and their respective outline is provided.

- `get_operational_remote_backends()` – a method which lists operational remote backends.
- `get_backends_names(backends)` – creates list of backend names from list of backends objects.
- `get_backend_from_name(name)` – get backend object from it's name.
- `get_sim_backend_from_name(name)` – get backend simulator object form it's name.
- `get_current_credits()` – get number of credits available for loaded account. It's used in main loop.
- `get_backend_name_from_number(number)` – legacy method used in main loop, when
- `execute_circuits(circuits, backend, use_mapping=False, noise_model=None)` – execute 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 previously described in section 3.
- `reset_jobs_counter()` – resets job counter same as corresponding tool. It's called at the end of main loop.
- `test_locally` –this method was previously described in section 3.
- `generate_gate_times()` – generates gates time using qiskit.aer. Required by noised simulation.
- `test_locally_with_noise` –this method was previously described in section 3.
- `get_jobs_from_backend` – 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=T`
  – report saving function. It contains several technical / configuration arguments and requires specifying
- `get_chsh_circuits()` – returns CHSH test circuits.

- `run_main_loop_with_chsh_test(circuits)` – this method was previously described in section 3.
- `create_circuit_from_qasm(qasm_file_path)` – creates circuit from specified qasm file.
- `get_mitigation_report_string(job)` – returns report string of a signle job with application of error mitigation basing on quantum device data at the time of job execution.
- `add_measure_in_base(QuantumCircuit, str)` – applies measurements given by *str* to each qubit of given quantum circuit.
- `custom_backend_monitor(backend)` – legacy method. Can be used to monitor backend, specified by name. Returns monitored data.
- `save_calibration_data(backend, data)` – legacy method. Can be used to save calibration data, specified as argument, of 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. Their respective description is provided within experiment's folder and thus it won't be included in this document. The list of experiments (up to date) is following:

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

References

[1] Ibm quantum experience page. https://www.research.ibm.com/ibm-q/. Access: 28 VII 2019.

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

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

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

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

[6] 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.