QX-simulator

QuTech, TU Delft

Sep 21, 2023

USER MANUAL

1 Installation **3**
1.1 Installation of the Python package . . . 3
1.2 Installation from source . . . 3
1.2.1 Building the Python package . . . 3
1.2.2 Building the C++ executable from source . . . 4

2 Usage **5**
2.1 Running the simulator from Python . . . 5
2.1.1 Python API . . . 5
2.1.2 Using a constant seed for random number generation . . . 6
2.1.3 JSON output . . . 6
2.2 Running the binary built from source . . . 7

3 Output **9**
3.1 Contents of the output . . . 9
3.2 Iterating a simulation . . . 10

4 Error models **11**
4.1 Depolarizing channel . . . 11

5 Internals of the simulator **13**
5.1 Sparse state vector . . . 13

QX-simulator is a simulator for cQasm quantum circuits.

It is developed at QuTech in Delft, The Netherlands.

It comes both as an executable binary built from C++ and a Python wrapper API called QXelarator.

For questions and comments, please contact the current maintainer of the package, Pablo Le Hénaff <`p.lehenaff@tudelft.nl`>.

CHAPTER

ONE

INSTALLATION

1.1 Installation of the Python package

To install QXelarator easily, you can use pip directly:

```
pip install qxelarator
```

The package is available on PyPI here . This will install the latest released version. For normal users, this is the only thing you have to do, so you can stop reading.

1.2 Installation from source

QX-simulator can be built from source, provided you have installed some dependencies:

- Bison (for building libqasm)
- Flex (for building libqasm)

On top of that, you will need a C++ compiler with support for C++20, `make` (for Linux) and `cmake`.

```
git clone https://github.com/QuTech-Delft/qx-simulator.git

cd qx-simulator
```

1.2.1 Building the Python package

To build QXelarator yourself from source and add it to your local Python packages, inside the repo:

```
NPROCS=16 python3 -m pip install -v -e .
```

You will need to have SWIG installed for the above to work.

1.2.2 Building the C++ executable from source

This is particularly useful for debugging purposes, as the executable can then be run under `gdb`, for instance. To do so, do not forget to build it in debug mode, otherwise compiler optimizations will make debugging more difficult.

```
mkdir build

cd build

cmake -S .. -B . -DCMAKE_BUILD_TYPE=Release

make -j16 qx-simulator
```

This will produced an optimized binary `qx-simulator` (`-DCMAKE_BUILD_TYPE=Debug` to disable optimizations).

To build the C++ tests, add the following option to the `cmake` call: `-DQX_BUILD_TESTS=ON`. To run them, call `ctest`.

CHAPTER

TWO

USAGE

2.1 Running the simulator from Python

2.1.1 Python API

The most straightforward way to execute a cQasm file is using the `qxelarator.execute_string` method:

```
>>> import qxelarator
>>> r = qxelarator.execute_string("version 1.0;qubits 2;h q[0]")
>>> r
Shots requested: 1
Shots done: 1
Results: {'00': 1}
State: {'00': (0.7071067811865475+0j), '01': (0.7071067811865475+0j)}
```

The return value is a `qxelarator.SimulationResult` object and offers access to aggregated measurement register results and final quantum state:

```
>>> isinstance(r, qxelarator.SimulationResult)
True
>>> r.results
{'00': 1}
>>> r.state["00"]
(0.7071067811865475+0j)
```

You can also execute a cQasm file:

```
>>> qxelarator.execute_file("bell_pair.qc")
Shots requested: 1
Shots done: 1
Results: {'11': 1}
State: {'11': (1+0j)}
```

In case the parsing/analysis of the cqasm string or file fails, you end up with a `qxelarator.SimulationError` object:

```
>>> r = qxelarator.execute_string("version 1.0;qubits 2;h q[0")
<unknown>:1:27: syntax error, unexpected end of file, expecting ',' or ']'
Failed to parse <unknown>
Cannot parse and analyze string version 1.0;qubits 2;h q[0
>>> r
Quantum simulation failed: Simulation failed!
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(r, qxelarator.SimulationError)
True
```

To simulate a quantum circuit multiple times, pass an integer number of iterations to e.g. `execute_file`:

```
>>> qxelarator.execute_file("bell_pair.qc", iterations = 10)
Shots requested: 10
Shots done: 10
Results: {'00': 3, '11': 7}
State: {'11': (1+0j)}
```

To make sense of the output of QX-simulator, please visit Output

Alternatively, you can use the "old" API by creating a `qxelarator.QX` instance:

```
import qxelarator

qx = qxelarator.QX()
```

Then, load and execute a number of times the cQasm file (e.g. `bell_pair.qc`):

```
qx.set('bell_pair.qc') # Alternatively: qx.set_string('version 1.0;qubits 1;x q[0]')
json_string = qx.execute(3)
```

2.1.2 Using a constant seed for random number generation

By default QX-simulator will generate different random numbers for different executions of a given circuit. This means that `measure`, `prep_z` and error models will make simulation results non-deterministic.

In some cases this is not desired. To make the output of the simulator deterministic over different runs, you can pass a constant `seed` parameter:

```
qxelarator.execute_string("version 1.0;qubits 2;h q[0];measure_all", iterations=1000,␣
↪seed=123)
```

2.1.3 JSON output

The "old" API provides a function to set a file to output JSON:

```
qx.set_json_output_path("simulation_result.json")
```

After another `execute(1000)` call, that JSON output will look like this:

```
> cat simulation_result.json
{
    "info": {
        "shots requested": 1000,
        "shots_done": 1000
    },
    "results": {
        "000": 516,
```

(continues on next page)

(continued from previous page)

```
        "001": 241,
        "011": 243
    },
    "state": {
        "001": {
            "real": 1.00000000,
            "imag": 0.00000000,
            "norm": 1.00000000
        }
    }
}
```

Note: The json string `json_string` obtained as output of `json_string = qx.execute(n)` is equal to the content of this file.

2.2 Running the binary built from source

The following will result in the same runs using the executable binary instead of the Python package:

```
./qx-simulator -c 1000 -j simulation_result.json ../tests/circuits/bell_pair.qc
```

CHAPTER

THREE

OUTPUT

3.1 Contents of the output

As described in *Usage*, the simulator can output to stdout (C++ executable version and old Python API), json (C++ executable and Python) and as a `SimulationResult` object (Python API). Those all contain the same data.

Let's simulate a bell pair 100 times:

```
version 1.0
qubits 2

h q[0]
cnot q[0], q[1]
```

This will result in:

```
Shots requested: 100
Shots done: 100
Results: {'00': 100}
State: {'00': (0.7071067811865475+0j), '11': (0.7071067811865475+0j)}
```

- `Shots requested` and `Shots done` are always equal to the number of iterations.
- The `Results` section contains how many times a given measurement register is captured when running the iterations. In this case, the circuit doesn't contain any measurement, and therefore the measurement registers are always 00, and this occurred 100 times.
- The `State` section contains the full quantum state at the end of the very last iteration. It maps quantum kets to complex amplitudes. Here you can recognize

the usual bell pair state: `1/sqrt(2) ( |00> + |11> )`.

Let's add measurements:

```
h q[0]
cnot q[0], q[1]
measure q[0:1]
```

The result is now:

```
Shots requested: 100
Shots done: 100
Results: {'00': 49, '11': 51}
State: {'00': (0.9999999999999998+0j)}
```

You can there notice that the final quantum state is collapsed to “00”, because of the measurements. The ket `|00>` doesn’t have amplitude exactly 1 because of the approximation of real numbers done by floating point arithmetic inside the simulator, something to keep in mind when interpreting the results. Again, the `State` section is the quantum state at the end of the 100th iteration. Some other iterations ended up in the state `|11>`.

The `Results` section can here be interpreted as: in 49 iterations out of 100, the final measurement register was “00” and in the remaining 51 iterations it was “11”.

3.2 Iterating a simulation

For those circuits that contain only final measurements, running multiple iterations and aggregating measuremeng registers in the `Results` section is not very useful, since you can directly obtain the probability of each measurement by taking the squared modulus of the complex numbers in the `State` section.

However, the `Results` section takes all its meaning when using for instance conditional gates, since then the quantum state varies per iteration. For example:

```
version 1.0
qubits 3

h q[0]
measure q[0]
cond(b[0]) h q[1]
measure q[1]
cond(b[1]) x q[2]
```

When simulating this circuit, the final quantum state in the `State` section is non-deterministic. However, the aggregated measurement register is very useful and the ratios like `results["001"] / shots_done` converge as the number of iterations grow. For 100 iterations:

```
Shots requested: 100
Shots done: 100
Results: {'000': 48, '001': 27, '011': 25}
State: {'000': (0.9999999999999998+0j)}
```

For 10000 iterations:

```
Shots requested: 10000
Shots done: 10000
Results: {'000': 4995, '001': 2517, '011': 2488}
State: {'001': (0.9999999999999998+0j)}
```

CHAPTER

FOUR

ERROR MODELS

Error models allow the introduction of probabilistic errors during the execution of the quantum circuit. They are useful for simulating more realistically a real quantum computer.

The error model is specified by adding an `error_model` statement in the cQasm input, followed by the model name and its parameters.

The only supported error model is currently the depolarizing channel.

4.1 Depolarizing channel

This model is implemented as described in *Quantum Computation and Quantum Information* by Nielsen & Chuang. Note that the book uses the density matrix/quantum computation formalism, while currently QX-simulator only uses state vector simulation.

This model is parametrized by a probability of error **p**. Between each gate of the circuit, an error on a uniformly randomly chosen qubit is applied with probability **p**. The error is uniformly a **X** (bit-flip), **Y** or **Z** (phase-flip) gate.

For instance, take the following circuit:

```
version 1.0
qubits 2

error_model depolarizing_channel, 0.0001

h q[0]

measure_all
```

When simulated 100000 times, it can yield:

```
Measurement register averaging
00       50000/100000 (0.50000000)
01       49994/100000 (0.49994000)
10       4/100000 (0.00004000)
11       2/100000 (0.00002000)
```

CHAPTER

FIVE

INTERNALS OF THE SIMULATOR

5.1 Sparse state vector

QX-simulator internally represents a quantum state using a hash table mapping kets (e.g. `|0010110>`) to their associated complex amplitudes, and omits zero (or near-zero) amplitudes.

You can read about this approach in *this paper <https://dl.acm.org/doi/10.1145/3491248>* by Samuel Jaques and Thomas Häner. Note however that QX-simulator was developed independently and the internal implementation differs.

This way to represent a quantum state is, in a lot of cases, very beneficial in terms of simulation runtime and memory usage.