Experiencing Quantum Computers

Masterpraktikum

Contact: Dr. H. Chau Nguyen and Prof. Otfried Gühne

Theoretical Quantum Optics Group
Department of Physics, University of Siegen

Email: `chau.nguyen@uni-siegen.de` and `otfried.guehne@uni-siegen.de`

Contents

The tutorial will be continuously improved for better readability. Do check out for the updated version at:

https://www.tp.nt.uni-siegen.de/+nguyen/index.html

Version October 10, 2023

This labcourse introduces students to the use of quantum computers. The labcourse consists of two days working in the computer room with access to publicly available simulated and real quantum computers. Students will learn how to write code to control and calibrate a quantum computers. As an examples, we implement the Bell experiment and the Deutsch-Jozsa algorithm.

- **Day 1:** Getting used to coding in a quantum computer (basic coding, tomography)
- **Day 2:** Implementation of the Deutsch-Jozsa algorithm and error mitigation

Day 1: Getting used to coding in a quantum computer

Background requirement
Please read the materials provided in Section 1. Be sure that you are able to answer the following questions (or completed the indicated task) before going into the lab.

(1) What is a qubit?
(2) What is a pure state and a mixed state?
(3) What does one mean by state tomography?
(4) What is the computational basis representation?
(5) What are the X,Y,Z, Hadamard, gates?
(6) What is the CNOT gate?
(7) What happens when a qubit is measured?
(8) What is a quantum circuit?
(9) Implement the python code in Section 2.3 [If you failed to install `qiskit`, you can use the `IBMQ` Website with an account]
(10) How does one carry out the tomography of a state?
(11) What is a Bell state?
(12) What is the setup of the Bell experiment with the singlet Bell state?

1. Quantum computer: qubits, gates and circuits

There are already quite several good introductions to the topics. Please read the following introductory articles:

- N. David Mermin, *'From Cbits to Qbits: Teaching computer scientists quantum mechanics,'* Am. J. Phys.**71**, 23 (2003) [`arXiv:quant-ph/0207118`].
- S. M. Barnett, *'Quantum information,'* Oxford University Press, Chap. 2, pp. 31-49.

2. The `Qiskit` interface

`Qiskit` can serve as a very simple *frontend* interface to a quantum computer *backend* (or a simulated one). That means, among other features, `qiskit` allows one to build up a quantum circuit and send it to a quantum computer (or a simulated one) for execution.

2.1. Install `qiskit`. It is also recommended that students try to install `qiskit` locally in their computer for further learning. The installation in `linux` is straightforward following the instructions from `qiskit` at:

`https://qiskit.org/documentation/stable/0.24/install.html`

The basic steps are:

(1) Install `anacoda`
(2) Create and activate an environment in `anacoda`, here named `TQO2023`.
(3) Install `qiskit`
(4) Install `jupyter notebook`

For `windows` users, follow the guidelines here:

https://techglimpse.com/install-qiskit-on-windows-10-ibmq/

2.2. Startup. In the experiment, you are provided with a computer, where `qiskit` has been installed. After starting the system, open your terminal. Go to the `TQO` folder and make a new folder with your group name and go inside the folder.

```
cd TQO
mkdir GROUPNAME
cd GROUPNAME
```

Thereafter activate the `anacoda` enviroment `TQO2023` where `qiskit` was installed. Then open the `jupyter notebook`.

```
conda activate TQO2023
jupyter notebook &
```

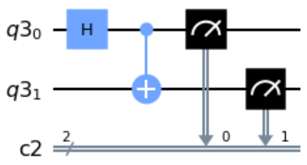
The webbrowser Firefox will be opened, redirecting you to the `jupyter notebook`. Choose `new notebook` from the drop menu.

2.3. Building a circuit with `qiskit`. The best way to get used to `qiskit` is to follow a simple example. The following simple example is adapted from the tutorial provided by `qiskit`. Type the example into the jupyter notebook. Save it as `Example.ipynb`.

```
#BUILDING A CIRCUIT--------------------------------
#Importing circuit builder
from qiskit import QuantumCircuit

#Create a Quantum Circuit acting on 2 qubits and 2 cbits
qc = QuantumCircuit(2, 2);

#Add a H gate on qubit 0
qc.h(0);

#Add a CNOT gate on control qubit 0 and target qubit 1
qc.cx(0, 1);

#Measure the qbits and write to the classical bits
qc.measure([0,1], [0,1]);

```

```
# Draw the circuit
qc.draw('mpl');
```

One should get:



2.4. Running the circuit in a simulator. To run the circuit, one first has to setup the *backend*. Here we learn to setup a simulator. Setting up the backend to be a real quantum computer will be discussed later. Type the following codes into the next scope

```
#SETUP THE BACKEND AND TRANSLATOR----------------
#Importing the simulator
from qiskit.providers.aer import QasmSimulator
# Use Aer's qasm_simulator
backend = QasmSimulator()

#Importing translator
from qiskit import transpile
#Compile the circuit so that the backend understands
qc_bin = transpile(qc, backend=backend)

#EXECUTION-------------------------------------
# Execute the circuit on the qasm simulator
qc_job = backend.run(qc_bin, shots=20000)
```

The last command typically sends the job to the backend. In general, the backend can be busy (which is not the case here for the local simulator), then the job can be put in a queue waiting for execution. The following commands query for the status of the job.

```
from qiskit.providers.jobstatus import JobStatus
print(qc_job.status())
```

2.5. Obtain the results. Once the job is done, one can retrieve the results and proceed to analysis.

```
# Grab results from the job
qc_result = qc_job.result()
#LOADING AND VISUALISE THE RESULTS-------------
# Returns counts
qc_counts = qc_result.get_counts(qc_bin)
print("\nTotal counts are:",qc_counts)
```

2.6. Changing the backend to mimic an IBMQ. Unfortunately this year IBM has limited the access to their quantum computers. It is hard to queue our small academical job on their computers. Fortunately, there are options to simulate the real backends with `qiskit`.

We can take a backend, for example:

```python
from qiskit.providers.fake_provider import FakeLima
backend = FakeLima();
```

In many aspects, the fake backend mimics the real backend at IBMQ well. This also include all the possible sources of noise in the operation of the device.

Next we need to compile the circuit constructed before to the machine code that the backend understands:

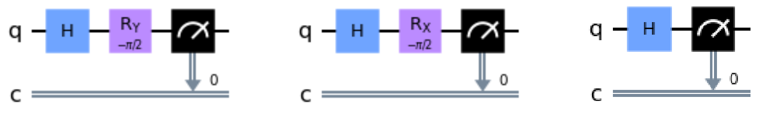
```python
from qiskit import transpile
qc_bin = transpile(qc, backend=backend)
```

And run the code as usual:

```python
from qiskit.providers.jobstatus import JobStatus
qc_job = backend.run(qc_bin,shots=20000)
print(qc_job.status())
```

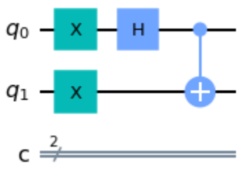
Tip: Save all the relevant codes to the `Example.ipynb` so that later you can reuse different paragraphs.

Experiment 1: The Hadamard gate. The following circuits prepare a qubit (initiated as $|0\rangle$), acting a Hadamard gate before making a measurements in three different direction, x, y and z.

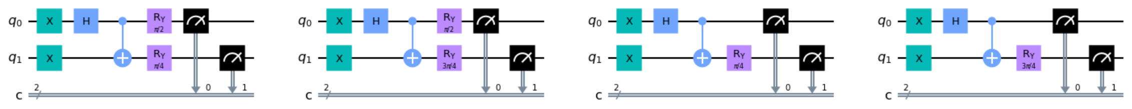


Notice that $R_\alpha(\varphi)$ denote that rotation operator by angle φ around axis $\alpha = x, y, z$. Explain why the circuits implement the designed experiments. Write the codes to construct the circuits. Run each of the circuits 5 times in a simulated quantum computer (say `lima`). Note the counts to your report. Remark whether the results are reasonable.

Experiment 2: The Bell state and Bell experiment. Show that the following circuit prepare the singlet Bell state



The following 4 circuits perform different measurement settings on the prepared Bell's state



Explain which measurements haven been simulated in terms of spin measurement directions two spin-$1/2$ particles. Relate the setting to the CHSH inequality.

Write the code that constructs the circuits. Run the code 5 times in a simulated quantum computer. Note the counts to your report. From the data, compute the value of the CHSH operator. Does it violate the CHSH inequality?

3. State tomography

In an actual quantum computer, there are various sources of noise. The output of a circuit may not be exactly the theoretically expected one. We therefore want to characterise the output of the circuit by means of state tomography. In this section, we learn how to carry out state tomography.

Let us consider a two-qubit state ρ, which can always be written as

$$\rho = \frac{1}{4}\sum_{i,j=0}^{3} \Theta_{ij}\sigma_i \otimes \sigma_j, \tag{1}$$

where σ_i and σ_j are the Pauli matrices. So, in order to construct the density operator ρ, one only has to measure the mean values

$$\Theta_{ij} = \mathrm{tr}[\rho(\sigma_i \otimes \sigma_j)]. \tag{2}$$

We say we have a tomography of the state.

First we have to load the libraries like in Experiment 1. Then a library for state tomography:

```python
# Tomography functions
from qiskit_experiments.library import StateTomography
```

Build a circuit as usual.

```python
# Create the actual circuit
from qiskit import QuantumCircuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
```

We then use `StateTomography` to generate the list of circuits for state tomography

```python
# Generate the state tomography circuits.
qc_tm = StateTomography(qc);
for c in qc_tm.circuits():
  print(c)
```

Look at and explain the outcomes.

One then only need to run the circuit in an appropriate backend:

```
# Execute
backend = FakeLima();
qc_tm_bin= transpile(qc_tm.circuits(),backend=backend)
qc_job = backend.run(qc_tm_bin, shots=20000)
qc_job.result().get_counts()
```

Using the results, one can compute Θ_{ij} and obtain an estimate $\hat{\rho}$ for the original state.

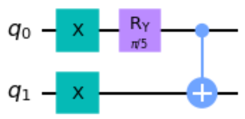
To access the quality of the reconstructed state $\hat{\rho}$ one can compute the so-called fidelity. If the target state ρ is pure, $|\psi\rangle\langle\psi|$, the fidelity is given by

$$F(\hat{\rho}, |\psi\rangle) = \langle\psi|\, \hat{\rho} |\psi\rangle. \tag{3}$$

In all of our experiments, the target states are pure and this simple formula is valid.

Experiment 3: Tomography of the Bell state. Using the code that prepare a Bell state in Experiment 2 and construct the tomography circuit set of the state. Reconstruct your state and compare it with the theoretical one using the fidelity as the quality measure. Repeat the experiments 10 times to have the statistics.

Experiment 4: Tomography of a general state. Consider the following circuit:



Write the code to implement the circuit and carry the tomography of the output state. Calculate the state theoretically and compare it with the experiment results using the fidelity as the quality measure. Repeat the experiments 10 times to have the statistics.

Day 2: Implementing the Deutsch-Jozsa algorithm and error mitigation

Background requirement
Please review the material in Day 1 and read the materials provided in Section 4 and be sure that you are able to answer the following questions (or completed the indicated task) before going into the lab.

(1) How does one write a circuit in `qiskit`?
(2) How does one carry out state tomography?
(3) What is the Deutsch-Jozsa problem?
(4) Explain the quantum algorithm for the Deutsch-Jozsa problem.
(5) Prove equation (5) for Deutsch-Jozsa problem with $n = 1$.

4. The Deutsch–Jozsa algorithm

We are given a hidden Boolean function f, which takes as input a string of bits, and returns either 0 or 1, that is,

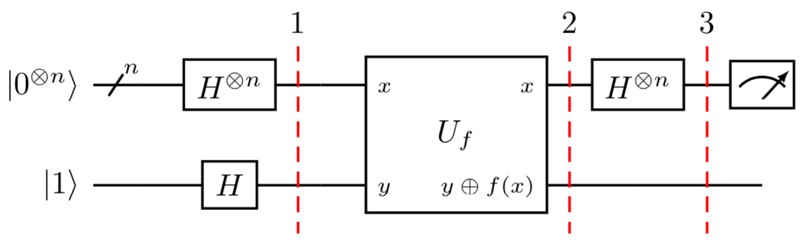
$$f : \{0,1\}^n \to \{0,1\} \tag{4}$$

The property of the given Boolean function is that it is guaranteed to either be *balanced* or *constant*. A constant function returns always 0 or always 1 for any input, while a balanced function returns 0 for exactly half of all inputs and 1 for the other half. Our task is to determine whether the given function is balanced or constant.

Classically, this question can be answered by evaluate the function at least twice for the best case, or $2^{n-1} + 1$ for the worse case.

Quantum mechanically, this question can be answered by a single run of the (quantum) function. The function is modelled in quantum mechanics by an oracle, which maps $|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$.

The following circuit demonstrates the quantum algorithm[1]:

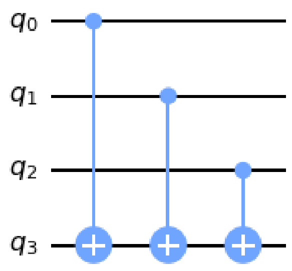


[1]Figure taken from `qiskit` tutorial.

When the output get $x = 0$, the function is constant, else it is balanced. To see that, one only has to follow the definition of the gate and show that the end state of the x-register is given by

$$\frac{1}{2^n} \sum_{y=0}^{2^n-1} \sum_{x=0}^{2^n-1} (-1)^{f(x)} (-1)^{x \cdot y} |y\rangle. \qquad (5)$$

A constant oracle is really simple. For example, the trivial circuit (no gate) outputs always 0 regardless of the inputs on the three qubits (can you think of a circuit that always output 1?). As an example of an oracle that is balanced, we have the following circuit:



Write the full table of output in your report to verify that the oracle is indeed balanced.

Experiment 5: Implementing the Deutsch-Jozsa algorithm. In this experiment we implement the Deutsch-Jozsa algorithm to determine if an oracle acting on three qubits is classical or balanced. Write the code to implement the following steps:

(1) Choose a real quantum computer backend with at least 4 qubits (3 input qubits and 1 output qubit). Checkout the physical constraints and errors.
(2) Create a 4 qubit register `qr`.
(3) Create a circuit `qc`.
(4) Write a function `add_constant_oracle()` that adds the constant oracle to a circuit `qc`.
(5) Write a function `add_balanced_oracle()` that adds the balanced oracle to a circuit `qc`.
(6) Write a function that coins a random number and call either `add_constant_oracle()` or `add_balanced_oracle()` accordingly.
(7) Implement the Deutsch–Jozsa algorithm.
(8) Transpile the circuit, run and obtain the raw counts for $|000\rangle$ in the input register.

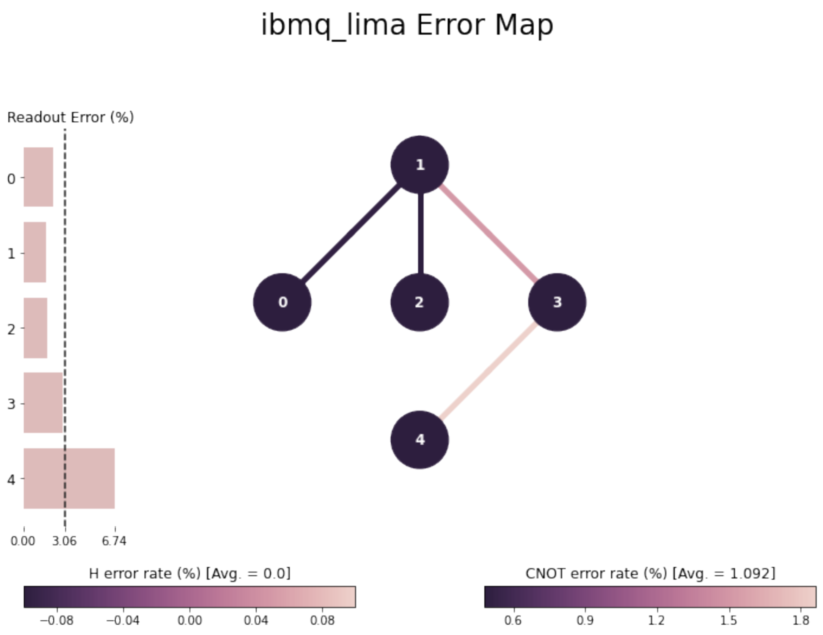
In your report, make a table recording the counts of 10 runs of the circuit with the simulated backend. Is the algorithm successful?

5. Inside the IBMQs, physical constraints, errors, mitigation

Now let get a bit more deeper physical insights into the computer. The following commands allow one to visualise the lower level information of the backend.

```
from qiskit.visualization import *
backend = FakeLima();
plot_gate_map(backend);
plot_error_map(backend);
```

One should get:



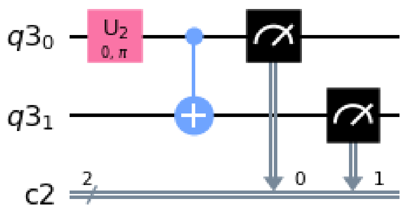
One characteristic of the currently available NISQ is that they are noisy. As physicists, we now learn about noise and how to control them (as much as we can).

A very important type of noise in current NISQ is the measurement readout noise. This means that when a measurement is made, the qubit is collapsed to state 0 or 1. However, when we actually read them out (through the physical interaction), we might misread it; the outcome can be flipped.

We are now to study this type of noise and its mitigation. To do so, we will have to go deeper into the hardware behind the command we have learned.

In execution, it is important that the virtual gates are decomposed into physical gates (supported by the computer).

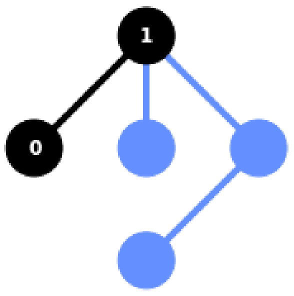
```
qc.decompose().draw(output='mpl');
```



Because of the physical constraints of the actual computer, some gate may not be directly executed between certain pair of qubits. In this case, the swap gates are used to change the qubit that contains information closer to each other; all these we do not see if we naively run the circuit as above. In reality, the virtual qubits are mapped into the physical ones in away to reduce these swapping.

Let us see an example how the the virtual qubits are layout in the the physical qubits of an actual computer.

```
from qiskit import transpile
backend = FakeLima();
qc_transpiled = transpile(qc, backend=backend,layout_method='trivial',
    optimization_level=1)
from qiskit.visualization import plot_circuit_layout
plot_circuit_layout(qc_transpiled, backend)
```



Let us now study the measurement errors associated with the qubits.

```
# Import general libraries (needed for functions)
import numpy as np
import time

# Import Qiskit classes
```

```
from qiskit_experiments.library import LocalReadoutError,
    CorrelatedReadoutError
from qiskit.tools.visualization import plot_histogram
```

Construct the calibration circuits

```
backend=FakeLima()
# Generate the calibration circuits
qubits=[0,1,2]
gcalib= CorrelatedReadoutError(qubits)
for c in gcalib.circuits():
  print(c)
```

Look at the output and explain what the circuits describe.

We then run the circuit and analyse the data:

```
#Transpile and execute:
gcalib_bin= transpile(gcalib.circuits(),backend=backend,layout_method
    = 'trivial',optimization_level =1)
calib_job = backend.run(gcalib_bin, shots=20000)
calib_result = job.result()
```

This data will allows us to construct the so-called transition matrix of the form

$$\mathrm{Prob}(\boldsymbol{s}|\boldsymbol{\sigma}). \tag{6}$$

where $\boldsymbol{\sigma}$ is the prepared state in the computational basis, and $\boldsymbol{s}$ is the measured outcome. One see that even when the state is prepare in the computational basis, due to the noise in the measurement process, the outcomes is not necessary deterministic.

This transition matrix then allows one the mitigate the distribution of the measurement outcomes. Indeed, suppose a quantum state in principle generates distribution $P(\boldsymbol{\sigma})$ upon measurement in the computational basis. The read out process in the quantum computer would actual gives a distribution $Q(\boldsymbol{s})$ given by

$$Q(\boldsymbol{s}) = \sum_{\boldsymbol{\sigma}} \mathrm{Prob}(\boldsymbol{s}|\boldsymbol{\sigma}) P(\boldsymbol{\sigma}). \tag{7}$$

In order to get the true distribution $P(\boldsymbol{\sigma})$, one thus has to invert $\mathrm{Prob}(\boldsymbol{s}|\boldsymbol{\sigma})$ as a matrix and apply onto $Q(\boldsymbol{s})$ as a vector.

Experiment 6: The Bell experiment with error mitigation. Write a code that carry out the Bell experiment. Run the code on a simulated quantum computer. Carry out the measurement error mitigation and computed the mitigated CHSH violation.