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

Day 1: Getting used to coding in a quantum computer 3
1. Quantum computer: qubits, gates and circuits 3
2. The `Qiskit` interface 3
3. State tomography 7

Day 2: The Deutsch-Jozsa algorithm and the Grover algorithm 9
4. The Deutsch–Jozsa algorithm 9
5. The Grover algorithm 11

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 8, 2024

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 and run it in a quantum computers. As an examples, we implement the Bell experiment, the Deutsch-Jozsa algorithm, and the Grover algorithm.

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

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.

(Q1) What is a qubit?
(Q2) What is a pure state and a mixed state?
(Q3) What does one mean by state tomography?
(Q4) What is the computational basis representation?
(Q5) What are the X,Y,Z, Hadamard, gates?
(Q6) What is the CNOT gate?
(Q7) What happens when a qubit is measured?
(Q8) What is a quantum circuit?
(Q9) How does one carry out the tomography of a state?
(Q10) What is a Bell state?
(Q11) 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`.

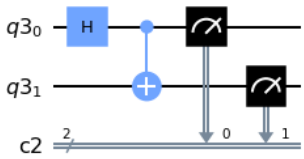
```
conda activate TQO2023
jupyter notebook &
```

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:

```
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:

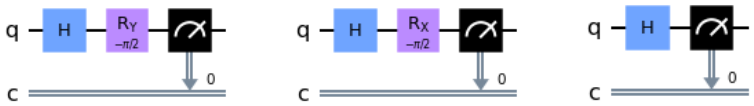
```
from qiskit import transpile
qc_bin = transpile(qc, backend=backend)
```

And run the code as usual:

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

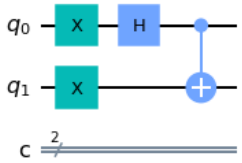
Tip: Save all the relevant codes to the `Example.ipynb` so that later you can reuse different paragraphs.

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.

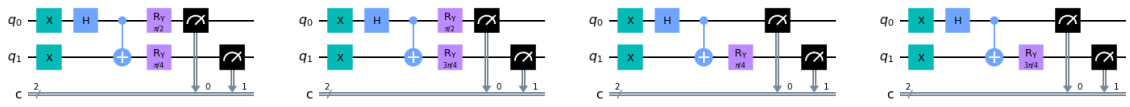


Notice that $R_\alpha(\varphi)$ denote that rotation operator by angle φ around axis $\alpha = x, y, z$. Explain why the circuits implement the designed experiments. Write the codes to construct the circuits. Run each of the circuits 5 times in a simulated quantum computer (say `lima`). Note the counts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

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

Build a circuit as usual.

```
# 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

```
# 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()
```

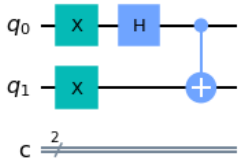
Using the results, one can compute Θ_{ij} and obtain an estimate $\hat{\rho}$ for the original state.

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

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

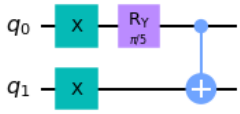
In all of our experiments, the target states are pure and this simple formula is valid.

Experiment 3: Tomography of the Bell state. Recall that the following circuit prepare the singlet Bell state



Write the code to prepare the Bell state using the above circuit. 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: The Deutsch-Jozsa algorithm and the Grover algorithm

Background requirement
Please review the material in Day 1 and read the materials provided below and be sure that you are able to answer the following questions before going into the lab. Questions marked with $*$ are optional.

- (Q12) What is the Deutsch-Jozsa problem?
- (Q13) Explain the quantum algorithm for the Deutsch-Jozsa problem.
- (Q14)$*$ Prove equation (5) for Deutsch-Jozsa problem with $n = 1$.
- (Q15) What is the problem targeted by the Grover algorithm?
- (Q16) Explain why a classical greedy algorithm requires $N - 1$ calls of the function for the unstructural search.
- (Q17) Explain the implementation of the phase flip operator from the oracle.
- (Q18) What is the circuit implements the quantum-OR oracle?
- (Q19) What is the circuit for the Grover step?
- (Q20) What is the circuit for the full Grover algorithm?
- (Q21)$*$ Derive the output state after t Grover steps in equation (11).

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,

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

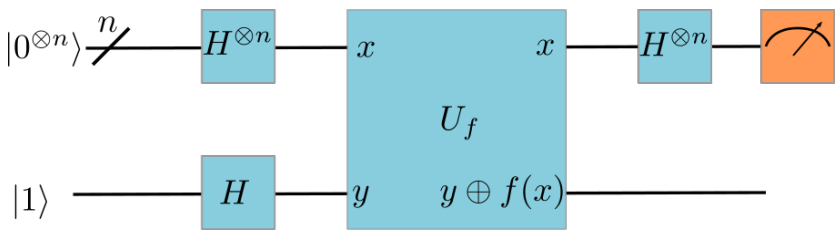
The property of the given Boolean function is that it is guaranteed to either be *balanced* or *constant*. A constant function returns always 0 or always 1 for any input, while a balanced function returns 0 for exactly half of all inputs and 1 for the other half. Our task is to determine whether the 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 f is modelled in quantum mechanics by an *oracle* U_f, which maps $U_f : |x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$.

The following circuit demonstrates the quantum algorithm[1]:

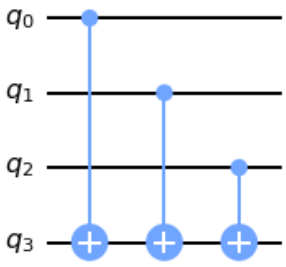
[1] Figure taken from `qiskit` tutorial.



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

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

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 n qubits is classical or balanced. Write the code according to the following structure:

```
#Deutsch Jozsa algorithm:
def DeutschJozsaCircuit(n,oracle):
    #n: number of qubits
    #oracle: the oracle in the form of instruction
    ...
  return qc
```

Fill your code in. Here `oracle` is an 'intruction', which can be appended to a circuit `qc` by

```
qc.append(oracle,qlist) #qlist: list of qubits where oracle acts
```

Use the oracle and the following code to test your implementation:

```
#Functions constructing oracles:
def constant_oracle(n):
    qc = QuantumCircuit(n+1, name='oracle')
```

```
    print("The oracle: \n"); print(qc);
    return qc.to_instruction();

def balanced_oracle(n):
    qc = QuantumCircuit(n+1, name='oracle')
    for k in range(n):
        qc.cx(k,n);
    print("The oracle: \n"); print(qc);
    return qc.to_instruction();

#Set number of qubits:
n=3;
#Fippling coin to choose oracle:
coin=randint(0,1);
if (coin==0):
    oracle= constant_oracle(n);
else:
    oracle= balanced_oracle(n);
#Construct Deutsch-Jozsa circuit:
qc= DeutschJozsaCircuit(n,oracle);
print("The Deutsch-Jozsa circuit: \n"); print(qc)
#Run:
qc_bin= transpile(qc,backend=backend);
qc_job = backend.run(qc_bin, shots=10000);
qc_result = qc_job.result();
print("Result counts:")
display(qc_job.result().get_counts());
if (coin==0):
    print("Oracle was constant!");
else:
    print("Oracle was balanced!");
```

Set the backend to a simulator for testing and then a FakeBackend of your choice. Make a table recording the counts of 10 runs of the circuit with the FakeBackend. Compute the probability of getting the correct output from the algorithm (with error bars).

5. The Grover algorithm

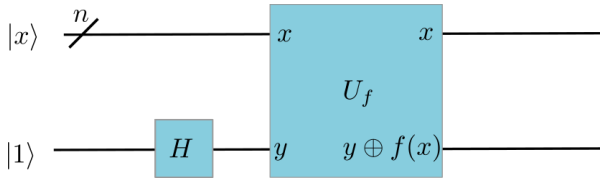
Suppose we have a function $f : \{0,1\}^n \to \{0,1\}$ which is 0 except for a single (unknown) value $x_0 \in \{0,1\}^n$ where $f(x_0) = 1$. The Grover algorithm aims at finding x_0 with low number of calls to the function.

The function f is assumed to have no *structure* to exploit (such as continuity, monotonicity, etc.) for a fast classical algorithm. In this case, a classical greedy search requires $N - 1 = 2^n - 1$ calls of the function (explain why!). Assuming the implementation of f in terms of a quantum oracle [see previous section], the Grover algorithm allows for finding x_0 in $\mathcal{O}(\sqrt{N})$ calls of the oracle. This algorithm is implemented using the following building blocks.

(1) *Implementation of the phase-flip operator.* Recall that the oracle implementation of f is an unitary U_f acting over $n+1$ qubits (including input as well as output qubits) such that $U_f(|x\rangle|y\rangle) = |x\rangle|y \oplus f(x)\rangle$. The phase-flip operator implementation for f is an unitary operator Z_f on the input qubits such that

$$Z_f(|x\rangle) = \begin{cases} |x\rangle \text{ if } f(x) = 0 \\ -|x\rangle \text{ if } f(x) = 1, \end{cases} \tag{6}$$

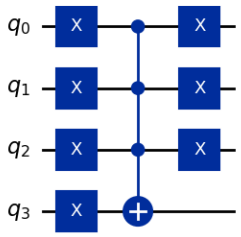
The phase-flip implementation Z_f of f can be obtained from its oracle implementation U_f using the following circuit:



(2) *Implementation of the Z_{OR}.* Notice that the OR fuction is a map $\text{OR} : \{0,1\}^n \to \{0,1\}$, which is 1 except for $x = 0$, where $\text{OR}(0) = 0$. Its phase-flip operator Z_{OR} is thus

$$Z_{\text{OR}}(|x\rangle) = \begin{cases} |x\rangle \text{ if } x = 0, \\ -|x\rangle \text{ if } x \neq 0. \end{cases} \tag{7}$$

Show that the following circuit implements the phase-flip operator for the OR function,



Further, one can write

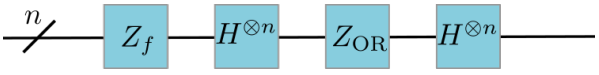
$$Z_{\text{OR}} = 2|0\rangle\langle 0| - \mathbb{I}, \tag{8}$$

where $\mathbb{I}$ is the identity operator (explain why!) Therefore

$$H^{\otimes n} Z_{\text{OR}} H^{\otimes n} = 2|u\rangle\langle u| - \mathbb{I}, \tag{9}$$

where $|u\rangle = 1/\sqrt{N} \sum_{x=0}^{N} |x\rangle$ (explain why!)

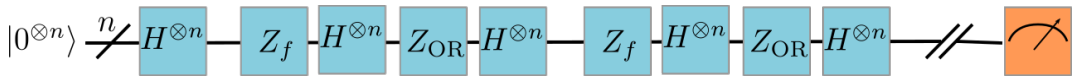
(3) *The Grover step*. The following circuit implements the Grover step:



(4) The Grover algorithm constitutes of preparing the state $|u\rangle = 1/\sqrt{N}\sum_{x=0}^{N}|x\rangle$ by Hadamard gates over every qubits, and then iteratively applying the Grover step for t_{opt} times, where

$$t_{\text{opt}} = \lfloor \pi/4\theta \rfloor \text{ with } \theta = \arcsin(1/\sqrt{N}). \tag{10}$$

The circuits assumes:



The output of the circuit gives x_0 with high probability.

Experiment 7: Implementation of the Grover algorithm. The rough structure of the program would be as follow:

```
#Quantum Or gate
def QOR(n):
  #n: number of input qubits, output last
  qc= QuantumCircuit(n+1);
  ...
  return qc.to_intruction();

#Construct the GroverStep
def GroverStep(n,oracle):
  #n: number of input qubits, output last
  qc= QuantumCircuit(n+1);
  ...
  return qc.to_intruction();

#Main function constructing full circuit
def GroverCircuit(n,oracle):
  qc= QuantumCircuit(n+1);
  ...
  return qc;

```

Fill the codes that implement the obove functions in. Using the following testing oracle

```
#Oracle:
def ConstructSingleTargetOracle(n,target):
    #Result in the last qubit (make sure in |->);
    qc= QuantumCircuit(n+1,name='q-oracle');
    for k in range(n):
        if (target[k]==0):
            qc.x(k);
    qc.mcx(list(range(n)),n)

    for k in range(n):
        if (target[k]==0):
            qc.x(k);

```

```
    print("Oracle ");print(qc);
    return qc.to_instruction();

ConstructSingleTargetOracle(3,[0,0,1])
```

and test your code with

```
#Set number of qubits and target
n=4;
N=2**n; #dim
target= [0,1,0,0];
rtarget= target[::-1]; #reverse to meet the order of qiskit

SingleTargetOracle= ConstructSingleTargetOracle(n,rtarget)
qc= GroverCircuit(n,SingleTargetOracle);
qc_bin= transpile(qc,backend=backend);
qc_job = backend.run(qc_bin, shots=10000);
qc_result = qc_job.result()

print("Original target \n", target)
plot_histogram(qc_job.result().get_counts())
```

Run your code on a simulator and make sure that it gives correct answer. Run your code on a FakeBackend (5 times) and give remark on the effect of noise in an actual quantum computer on the output. Estimate the probability of the correct answer (with error bars) from the collected data.

Let us now try to understand how the Grover algorithm works [further discussion during the experiment]. Notice that the qubits are initiated in state $|u\rangle$. Every Grover step then changes the state vector of the qubits, but it remains in the 2D-space of spanned by $|u\rangle$ and $|x_0\rangle$. In fact, every Grover step rotates the state vector in this 2D-space from $|u\rangle$ towards $|x_0\rangle$ by an angle of $2\theta = 2\arcsin(1/\sqrt{N})$. The state vector after t steps can be found to be

$$|\psi_t\rangle = \cos[(2t+1)\theta]|x_0\rangle + \sin[(2t+1)\theta]|\bar{x}_0\rangle, \tag{11}$$

where $|\bar{x}_0\rangle = 1/\sqrt{N-1}\sum_{x\neq x_0}|x\rangle$.

Experiment 8: Investigation of the state vector during Grover search. Run your code in **Experiment 7** with varying different steps t (from 0 to $4t_{\rm opt}$). From the data estimate the probability (with error bars!) of obtaining the output as $|x_0\rangle$ and compare it with eq. (11). Plot them together with the theoretical expected exact result. As usual, run the experiment with a simulator backend for testing and then a FakeBackend of your choice (5 times) for data collection.