Institute for Theoretical Physics
University of Cologne

Prof. Dr. S. Trebst
Q. Preiss, F. Eckstein, M. Pütz

Quantum Computational Physics

Exercise Sheet 1

Winter Term 2024/25

Due date: No hand-in **Discussion**: Tuesday, 15.10.2024

Website: thp.uni-koeln.de/trebst/Lectures/2024-QuantCompPhys.shtml

This exercise sheet is a **highly entangled** system of tasks. As many quantum states, it looks quite complex (long) at first glance, but once you measure it you will see that it collapses into many very small tasks and code snippets!

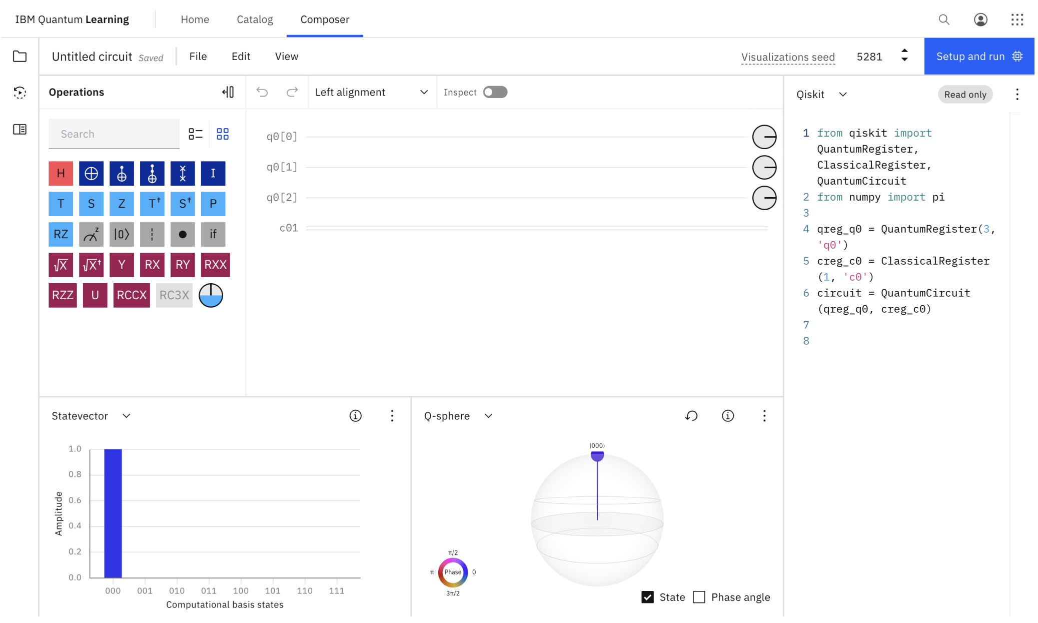
The purpose of this sheet is a superposition between two macroscopically distinct states:

- Getting you started with quantum computing using the IBM quantum composer and Qiskit.
- Providing you with the necessary tools (syntax, code snippets, etc.) to work on the following exercises.

Feel free to come back to this sheet at any time if you need to look up something. Happy coding!

Exercise 1: Hello quantum world!

To familiarize ourselves with the concept of quantum circuits and what it means to create/compose, simulate and run them, we will start with the very convenient IBM Circuit Composer (`https://quantum.ibm.com/composer`).



```
from qiskit import QuantumRegister,
ClassicalRegister,
QuantumCircuit
from numpy import pi

qreg_q0 = QuantumRegister(3,
'q0')
creg_c0 = ClassicalRegister
(1, 'c0')
circuit = QuantumCircuit
(qreg_q0, creg_c0)

```

Figure 1 – Screenshot of the IBM circuit composer.

Here you can create your first simple quantum circuits using a graphical interface (fig. 1) by simply dragging and dropping gates onto the circuit. You can also increase the number of qubits. On the bottom left you can see a visualization of a classical state vector simulation of the circuit. As you might know already such a simulation scales very poorly $\mathcal{O}(2^n)$ with the number of qubits n. That is why the IBM quantum composer limits you to 6 qubits. Using more powerful hardware it is of course possible to simulate more qubits. However since every additional qubit doubles the resources necessary, going further than 30 qubits is already a challenge for classical computers. But more on that later.

a) Experiment with the IBM circuit composer and see how the different gates affect the simulation result. For example as a starting point you can try to create a Bell state (fig. 2).

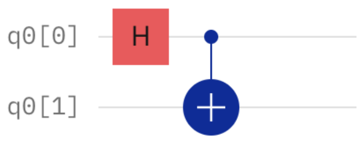


Figure 2 – Bell state circuit.

On a real quantum computer you of course don't have direct access to the state vector, i.e. the probability amplitudes of the quantum state. Thus, one needs to make use of measurements to extract information out of the quantum computer. In Qiskit the results of these measurements are stored in classical registers, i.e. bitstrings.

b) Try adding classical bits to the quantum circuit and use them to store results of measurements on every qubit at the end of your quantum circuit. For the visualization in the bottom left select `Probabilities` to show the probability distribution of the classical bits. This distribution converges to the probability amplitudes in the state vector from before if one would simulate an infinite number of `shots`, i.e. repeat the simulation an infinite number of times. However, a sufficiently large number of shots is usually enough to get a good approximation. You can see the number of shots used for the visualization on the y-axis in the bottom left plot.

Now that we understand the basics about the results we get from a quantum circuit it makes sense to run it on a real quantum computer. This is also easy in the IBM circuit composer. In the top right corner click on `Setup and run`. Then select the device you want to run you circuit on and the number of shots (i.e. how often the circuit is run). Then click on `Run`.

c) Run your circuit on a real quantum computer and compare the results to the simulation. Notice that the results can differ due to noise in the quantum computer. This is a very important aspect of quantum computing, and we will talk more about it in the next exercises.
Note: The IBM quantum devices are usually quite busy, so it might take a while until your job is executed (You might want to continue with next exercise while you wait). You can check the status of your job in the `Jobs` tab on the IBM quantum website.

If you got this far: Congrats on running your first quantum circuit on a real quantum computer!

Exercise 2: Qiskit

The web-based IBM quantum composer is a great tool to get started with quantum computing. However it is not as flexible and powerful as writing code in Qiskit. In this exercise we will write our first quantum circuits on our lokal machine. Using Qiskit involves a couple of generic steps that we want to break down in the following.

(i) Creating the Circuit

First of all we gotta create a circuit of course! Since we alreade did the previous exercise we can just copy the code from the IBM quantum composer. There you find the code representation of the circuit on the right side. You may have to click on `OpenQASM 2.0` and select `Qiskit` to see the Qiskit code. The code for our simple Bell state circuit should look something like this:

python
```python
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(2, 'q')

circuit = QuantumCircuit(qreg_q)

circuit.h(qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])
```

That was easy enough, right? We can now run this code in a jupyter notebook and visualize the circuit using the

python
```python
circuit.draw("mpl")
```

method. This will give us a nice visualization using the `matplotlib` library.

(ii) Loading a Backend

To run our circuit on a real quantum computer we need to load a backend. Here we use the `least_busy` method from the `QiskitRuntimeService` to get the least busy quantum computer:

python
```python
from qiskit_ibm_runtime import QiskitRuntimeService
service = QiskitRuntimeService()
backend = service.least_busy(simulator=False, operational=True)
```

In case you did not save your IBM token, you have to specify it by passing it as a keyword argument to the `QiskitRuntimeService` constructor:

python
```python
service = QiskitRuntimeService(
    channel="ibm_quantum",
    token="<MY_IBM_QUANTUM_TOKEN>"
)
```

(iii) Transpiling the Circuit

Every backend has a set of universal gates, that can be used to build a quantum circuit. Generallay these gates are not the same gates that we use to create our circuits. In order to run our circuit on a given backend, we have to transpile the circuit to the backend's gateset. We can think of this process just like the compilation of a high-level programming language to machine code.

To transpile a circuit we can use the `generate_preset_pass_manager` function from the `qiskit.transpiler.preset_passmanagers` module:

python
```python
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
pm = generate_preset_pass_manager(backend=backend, optimization_level=0)
isa_circuit = pm.run(circuit)
```

The `optimization_level` parameter can be set to 0, 1, 2 or 3. The higher the level, the more optimizations are applied to the circuit. However, higher levels also increase the compilation time. The `isa_circuit` object is the compiled circuit, which is based on the ***I**nstruction **S**et **A**rchitecture.* of the chosen backend. Note that this object is still a `QuantumCircuit`, which can be visualized using the `draw` method.

(iv) Sampler and Estimator

Awesome! Now we know how to create a quantum circuit, load a backend to run it on and transpile it to that specific backend. Before we start crushing the quantum world, let's understand the concept of the Qiskit primitives: `Sampler` and `Estimator`.

- The `Sampler` in Qiskit allows us to sample probabilities of different quantum states from a circuit. It runs quantum circuits and returns probability distributions for measurement outcomes in the computational basis. We can load the `SamplerV2` from the `qiskit_ibm_runtime` package, or from the `qiskit_aer.primitives` package (if we want to simulate the circuit classically) by running one of the following code blocks:

  python
  ```python
  # for the real quantum computer
  from qiskit_ibm_runtime import SamplerV2 as Sampler
  sampler = Sampler(backend)
  ```

  python
  ```python
  # for the classical simulator
  from qiskit_aer.primitives import SamplerV2 as AerSampler
  sampler = AerSampler()
  ```

Make sure to measure the qubits at the end of your circuit when using the `Sampler` primitive. A simple way to do this is by running

python
```python
circuit.measure_all()
```

after the last gate in your circuit.

- The `Estimator` primitive allows us to calculate the expectation values of observables (like Pauli operators) for a quantum state prepared by a circuit. We can load the `EstimatorV2` from the `qiskit_ibm_runtime` package, or from the `qiskit_aer.primitives` package (if we want to simulate the circuit classically) by running one of the following code blocks:

python
```python
# for the real quantum computer
from qiskit_ibm_runtime import EstimatorV2 as Estimator
estimator = Estimator(backend)
```

python
```python
# for the classical simulator
from qiskit_aer.primitives import EstimatorV2 as AerEstimator
estimator = AerEstimator()
```

(v) Running the Circuit

Now we are ready to run our circuit on the chosen backend. We can do this by calling the `run` method of the `Sampler` or `Estimator` object. If we wanted to sample the probabilities of the quantum states, we would run something like:

python
```python
job = sampler.run([isa_circuit], shots=1024)
```

Once the job is done, we can get the results by calling the `result` method of the `job` object:

python
```python
result = job.result()
```

Running a job on IBM's quantum devices can take a while (anything between seconds and days). You can check the status of your job on the IBM quantum website. Usually you dont want to keep your Python session open while waiting for the job to finish. To download the results of a job - once it has finished - you can use the command

python
```python
job = service.job(job_id)
```

where `job_id` is the id of the job you want to download the results from. You can find the `job_id` either on your dashboard on the IBM quantum website or by running:

python
```python
job.job_id()
```

Amazing! We are now able to create our own circuits and run them using the Qiskit ecosystem. Let's get started with some simple examples.

a) Create some quantum circuits using Qiskit and run them using the `Sampler` primitive on your local device (i.e. using the `qiskit_aer.primitives` package). Feel free to start from the Bell state example above. The goal of this exercise is really to get started and get a feeling for the Qiskit framework. We will dive deeper into the details in the next exercises. To visualize the results you could use the following code:

python
```python
from qiskit.visualization import plot_histogram
plot_histogram(results[0].data.meas.get_counts())
```

b) If you feel comfortable with the Qiskit framework, you can submit some jobs to the IBM quantum devices by using the `Sampler` from the `qiskit_ibm_runtime` package. They will (most likely) not finish during this tutorial, but you can come back and download them later. Do you spot any differences between the results you get from the simulator and the real quantum devices?

c) Now let't calculate some observables using the `Estimator` primitive. We can define Pauli string observables as follows:

python
```python
from qiskit.quantum_info import SparsePauliOp
obs_labels = ["IZ", "IX", "IY", "ZI", "XI", "YI", "ZZ", "XX", "YY"]
obs = [SparsePauliOp(label) for label in obs_labels]
```

Note that these observables are 2-qubit observables and therefore only work for circuits with 2 qubits. You can of course define your own Pauli strings for more qubits if you like. Before running the `Estimator` primitive, we have to map the observables to the circuit. This can be done by running:

python
```python
mapped_obs = [
    ob.apply_layout(isa_circuit.layout) for ob in obs
]
```

Finally, we can calculate the expectation values of the observables by running:

python
```python
job = estimator.run([(isa_circuit, mapped_obs)])
```

Once the job is done, we can get the results by running:

```python
result = job.result()
```

You can access and plot the expectation values using the follwing code:

```python
from matplotlib import pyplot as plt

# plot the results (very ugly...)
plt.plot(obs_labels, job_result[0].data.evs, 'o')
plt.xlabel('Observables')
plt.ylabel('Values')
plt.show()
```

Go ahead and calculate some expection values of your circuits! What happens when you remove all 2-qubit gates from the circuit? Feel free to also submit these jobs to the IBM quantum devices.

Exercise 3: Can you hear the noise?

Now that we can simulate as well as run our quantum circuit on a real quantum computer, it makes sense to compare the results we get from both methods. In this exercise we will look at two circuits, one beeing a very short circuit to create a Bell state and the other beeing a longer circuit that contains a mysteric block of gates which is repeated N times.

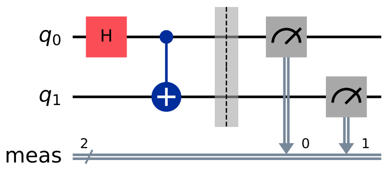


Figure 3 – Circuit 1 (short).

a) Use fig. 3 and fig. 4 as a reference to recreate these two circuits in Qiskit. Write a function that takes an integer N as an argument and returns the corresponding circuit, where the marked gate block is repeated N times. For $N = 0$ the function should return the circuit from fig. 3 and for $N \geq 1$ the circuit from fig. 4.
If you have trouble with the gates, you can always refer to the IBM circuit composer to see the Qiskit code representation of single gates.

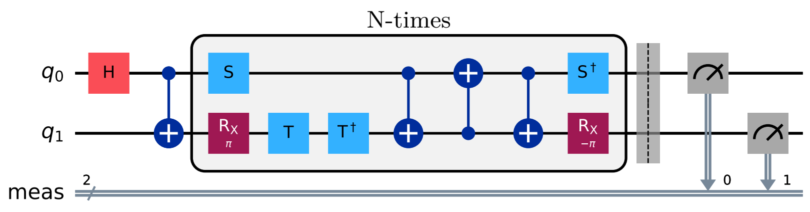


Figure 4 – Circuit 2 (long).

b) Sample both circuits using a simulator as well as a real quantum computer. When transpiling the circuit you should use `optimization_level=0` (Any idea why?). Compare the results you get from both methods for different values of N (e.g. $N = 0, 1, 10, 20, 50$). Do you have any clue what the mysteric block of gates does? What happens when you increase the number of repetitions N? Which method is correct?