POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

Laboratory 1 – Introduction to QisKit
058171 - Quantum Circuits and Devices

Piergiulio Mannocci

POLITECNICO MILANO 1863 | DIPARTIMENTO DI ELETTRONICA INFORMAZIONE E BIOINGEGNERIA

Contents

Contents

Library and modules

- Python programs typically rely on external collections of functions and routines contained in *modules*
- Modules must be imported before using routines and/or functions contained within them
- To import an entire module: `import module`
  - Functions and elements of the module can be called as `module.function`
  - Can be assigned an alias for convenience: `import module as mod`
    - Functions are then called using the alias: `mod.fun`
- Specific elements of a module can be loaded using: `from module import a,b,c`
- Most mathematical operators (exp, sqrt, ...) are contained in the `numpy` module

Fundamental data types in Python

- Text types stores strings and characters: *str* (string)

```
s = "hello world"
```

- Numeric data types can store numbers in different formats:
  - *int* (integer), *float* (floating-point), *complex* (complex, can be float or int)
  - Can be used to construct arrays, which are indexed using square brackets
  - Numeric arrays are compatible with mathematical operations

```
x = 1
x = 1.1
x = 1 + 1j
x = np.array([1,2,3])
```

- Sequence types:
  - *list*: similar to arrays, but can have elements of different data types; mutable; (generally) incompatible with mathematical operations
  - *tuple*: similar to arrays, but can have elements of different data types; immutable; (generally) incompatible with mathematical operations

```
l = [1,2,3]
t = (1,2,3)
```

- Mapping types: *dict* (dictionary), allows storage as key-value pairs

```
d = {"key1":1, "key2":"a"}
```

Python classes and objects

- Objects of a *class* are data structures characterized by *properties* and *methods*
  - Properties are variables stored inside the object, sort-of `struct` fields
  - Methods are functions that can act on both properties and external arguments
- Objects are typically initialized through a *constructor*

```
obj = NewObject(args)
```

- Methods are called using the dot-notation:

```
y = obj.method(args)
```

- (Public) properties can be similarly accessed using dot-notation:

```
p = obj.property
```

QisKit

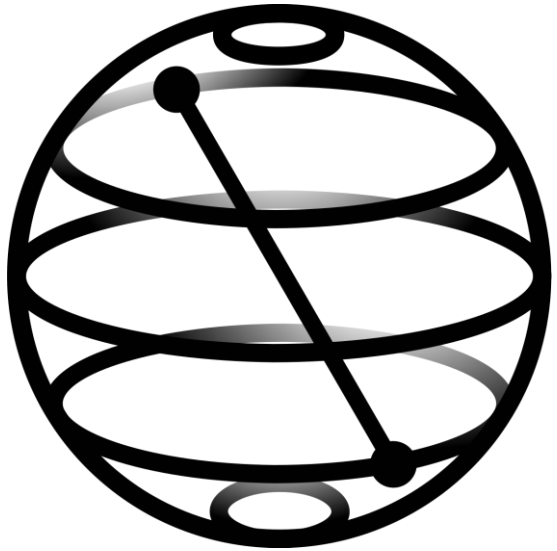
- Qiskit is an open-source software development kit (SDK) for quantum computers and hardware developed by IBM

- Allows both hardware-abstracted investigation of quantum circuits and hardware-specific evaluations of selected platforms
  - Currently limited to superconducting qubits and trapped ions

- We will use release v1.2.0

- Installation guide: https://docs.quantum.ibm.com/guides/install-qiskit

- Full documentation is available at https://docs.quantum.ibm.com/



Contents

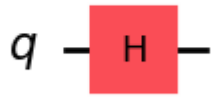
QuantumCircuit class

- Circuits are represented by objects of the QuantumCircuit class, constructor:
  - Nq number of qubits,
  - Nc number of classical bits (used for measurements)

```
from qiskit import QuantumCircuit

circ = QuantumCircuit(Nq,Nc)
```

- Gates can be appended (left-to-right) using methods of the class:
  - `.h(q)` Hadamard gate on qubit q
  - `.x(q)` NOT gate on qubit q
  - `.s(q)` S gate on qubit q
  - `.rx(theta,q)` X-rotation by theta on qubit q
  - `.ry(theta,q)` Y-rotation by theta on qubit q
  - `.rz(theta,q)` Z-rotation by theta on qubit q

```
circ = QuantumCircuit(1,0)
circ.h(0);
circ.draw("mpl")
```



- Circuits can be visualized using the `.draw()` method

Contents

Statevector simulator

- High-level simulator to describe Hilbert-space state vectors
- Must be imported from the `qiskit.quantum_info` module
- Constructed by providing coefficients of |0⟩ and |1⟩ basis states
  - Or by "labeled" state initializers:

```
state = Statevector.from_label("0")
```

- Statevectors transformation by a QuantumCircuit can be evaluated using the `.evolve(circ)` method

```
from qiskit.quantum_info import Statevector

state = Statevector([alpha,beta])
```

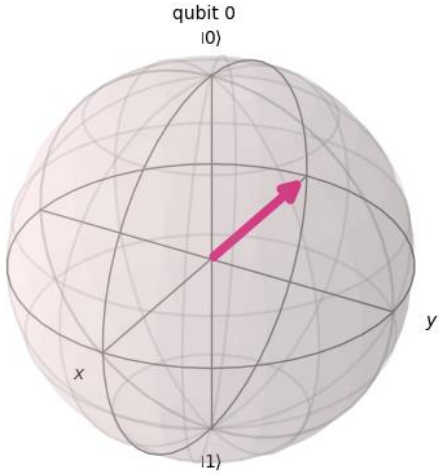
```
state = Statevector([1,0])
circ = QuantumCircuit(1,0)
circ.h(0);
state = state.evolve(circ);

print(state)
```

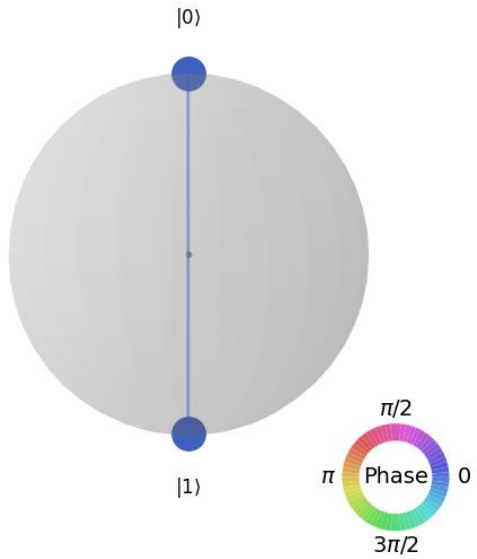
```
Statevector([0.70710678+0.j, 0.70710678+0.j],
            dims=(2,))
```

Statevector visualization

- Statevectors can be visualized using the `.draw(vis)` method:
  - "latex" returns a LaTeX visualization of the State
  - "bloch" plots the state vector on the Bloch sphere
  - "qsphere" plots the state vector on IBM's Q-sphere

$$\frac{\sqrt{2}}{2}|0\rangle + \frac{\sqrt{2}}{2}|1\rangle$$





Contents

Multiple-qubit circuits and state vectors

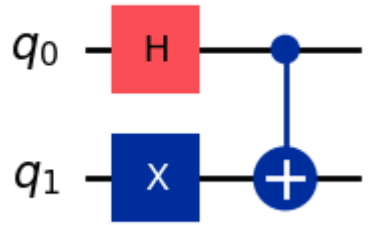
- Use an Nq number greater than 1 for multiple-qubit QuantumCircuits
  - Qiskit uses q0 as least significant qubit and qn as most significant qubit

- Multiple-qubit gates are similarly added to the circuit using methods of the QuantumCircuit class, e.g.:
  - `.cx(cq,tq)` c-NOT gate with control cq and target tq
  - `.iswap(q0,q1)` iSWAP gate between qubits q0 and q1
  - `.ccx(cq0,cq1,tq)` Toffoli gate with control cq0, cq1 and target tq

- Multiple-qubit statevectors can be constructed by providing all coefficients, or from multiple-qubit labels

- Statevectors transformation by a QuantumCircuit can be evaluated using the `.evolve(circ)` method similarly to the single-qubit case

```
circ = QuantumCircuit(2,0)

circ.h(0);
circ.x(1);
circ.cx(0,1);

circ.draw("mpl")
```

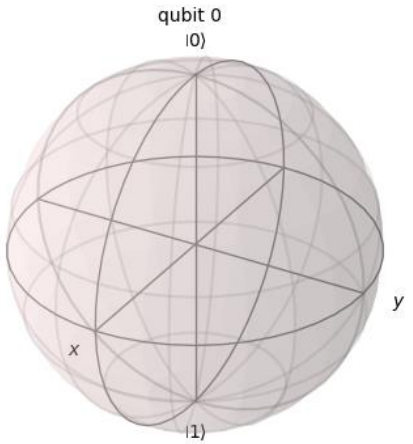


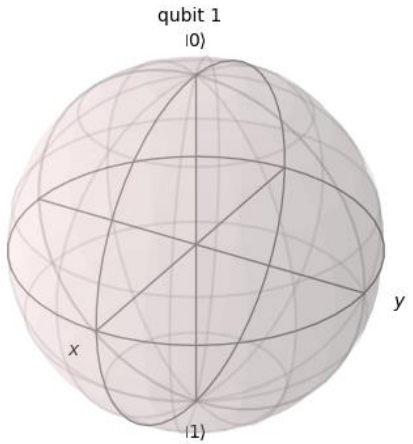
Multiple-qubit state vector visualization

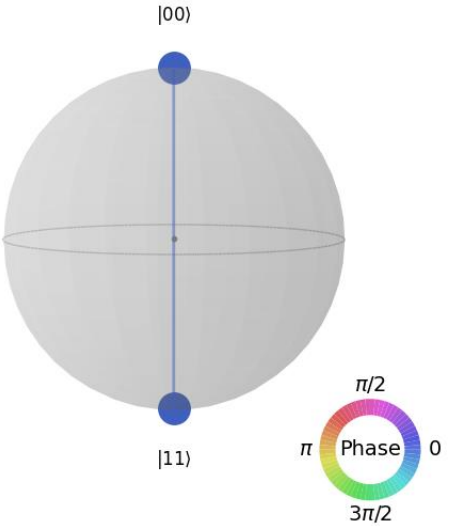
- Multiple-qubit state vectors can be visualized again with the `.draw(visualizer)` method:
  - "latex" returns a LaTeX visualization of the State
  - "bloch" plots the state vector of each qubit on a separate Bloch sphere
    - requires a *product state*, so does not work for entangled states!
  - "qsphere" plots the state vector on IBM's Q-sphere

- Example: $|\Phi^+\rangle$ state

$$\frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|11\rangle$$







Contents

Operator class

- Generic quantum operator, constructed by:
  - equivalent operator matrix: `op = Operator(numpy.matrix([[1,1],[-1,1]]));`
  - QuantumCircuit object: `op = Operator(circ)`
- Support for basic linear algebra:
  - `c*op` multiplication by a scalar (e.g. global phase)
  - `op1.tensor(op2)` tensor product `op1`⊗`op2`
  - `op1.expand(op2)` tensor product `op2`⊗`op1`
  - `op1.compose(op2)` matrix product `op2` · `op1`
  - `op1.compose(op2,front=True)` matrix product `op1` · `op2`
- Custom operators op representing quantum gates can be appended to qubit q of a QuantumCircuits using: `.append(op,q)`
- Operators can be checked for equivalence by:
  - `op1==op2` checks for *exact* equivalence (including global phase)
  - `qiskit.quantum_info.process_fidelity(op1,op2)` checks for equivalence aside from global phase

POLITECNICO MILANO 1863 | DEPARTMENT OF ELECTRONICS INFORMATION AND BIOENGINEERING

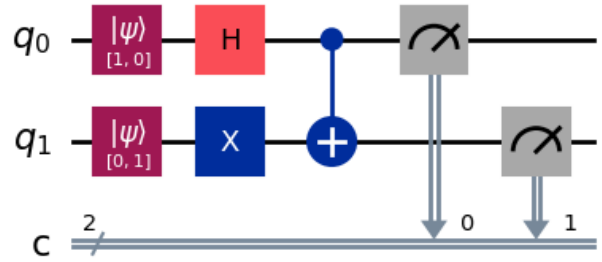
Hands-on time!
Exercise 1

Contents

Measurement

- Statevector and Operator simulators allow circuit evaluation in the quantum regime, *i.e.* before measurement
- We wish to perform a full simulation of the quantum circuit *including* projective measurement on the {0,1} basis
- Requires addition of measurement blocks to QuantumCircuit objects using the `.measure(q,c)` method:
  - `q` is the measured qubit, `c` is the classical bit where measurement result is stored
- We also used the `.initialize(state)` method to set a custom initial state
  - If no specific initialization is performed, qubits start in the |0⟩ state

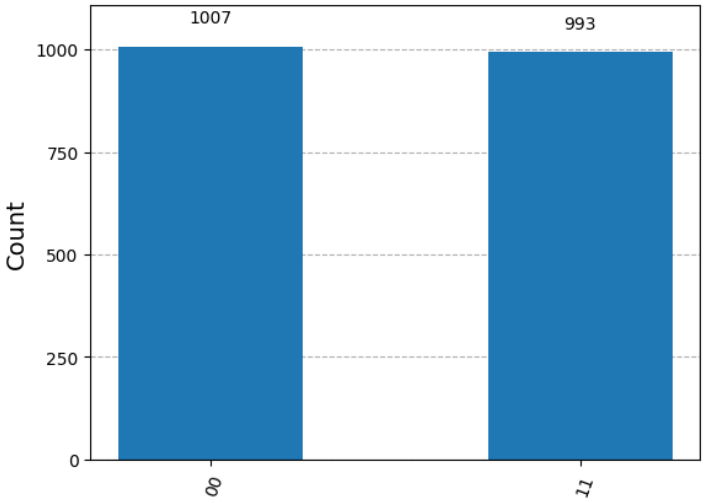
```
circ = QuantumCircuit(2,2);

circ.initialize([1,0],0);
circ.initialize([0,1],1);

circ.h(0);
circ.x(1);
circ.cx(0,1);

circ.measure(0,0);
circ.measure(1,1);

circ.draw("mpl")
```



AerSimulator

- AerSimulator is a simulation backend based on OpenQASM
- After constructing a backend object with `AerSimulator()`
- Simulations of a QuantumCircuit can be executed using the `.run(circ,shots=N)` method:
  - `shots` is the number of times the circuit is run and measured
  - returns a `Job` object that can be queried for results
- Measurement results are accessed through the `.result()` method of `Job` objects
  - Measurement counts for each basis state are accessed by the `.get_counts()` method on the Result object
- Counts can be visualized with `plot_histogram(counts)` from `qiskit.visualization`

```
from qiskit_aer import AerSimulator

backend = AerSimulator()

job = backend.run(circ,shots=2000)

result = job.result();

counts = result.get_counts();

from qiskit.visualization import
    plot_histogram

plot_histogram(counts)
```



Contents

Generic (Fake) Backends

- Real quantum processors (QPs) are plagued by nonidealities (noise, dephasing, relaxation, ...) that alter the operation of quantum circuits

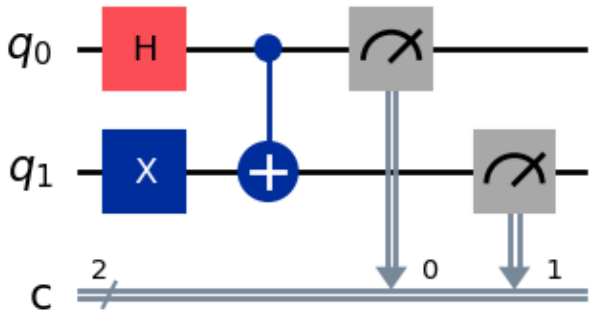
- `GenericBackend` objects provide something "halfway" between a simulator and a real QP
  - They are based on AerSimulator, so are run locally with no global queue
  - They include common nonidealities of real QPs
  - Models are based on snapshots of real IBM QPs

- Example: 5-qubit "realistic" backend

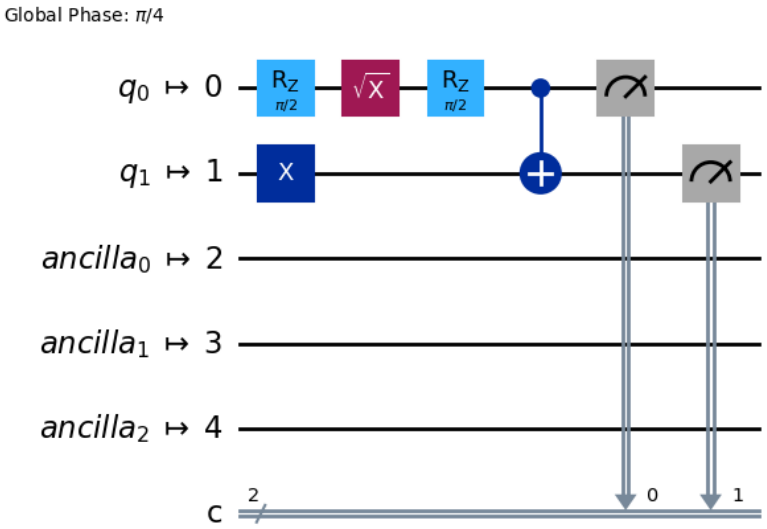
```
from qiskit.providers.fake_provider import GenericBackendV2
backend = GenericBackendV2(num_qubits=5);
```

Transpilation

- Real QPs implement a limited set of calibrated quantum gates
- Example: `GenericBackendV2` only implements CNOT, RZ, X, SX and I
- Quantum gates which are not part of this set are converted into combinations of calibrated gates (*transpilation*):




```
from qiskit import transpile
tcirc = transpile(circ,backend);
tcirc.draw("mpl")
```

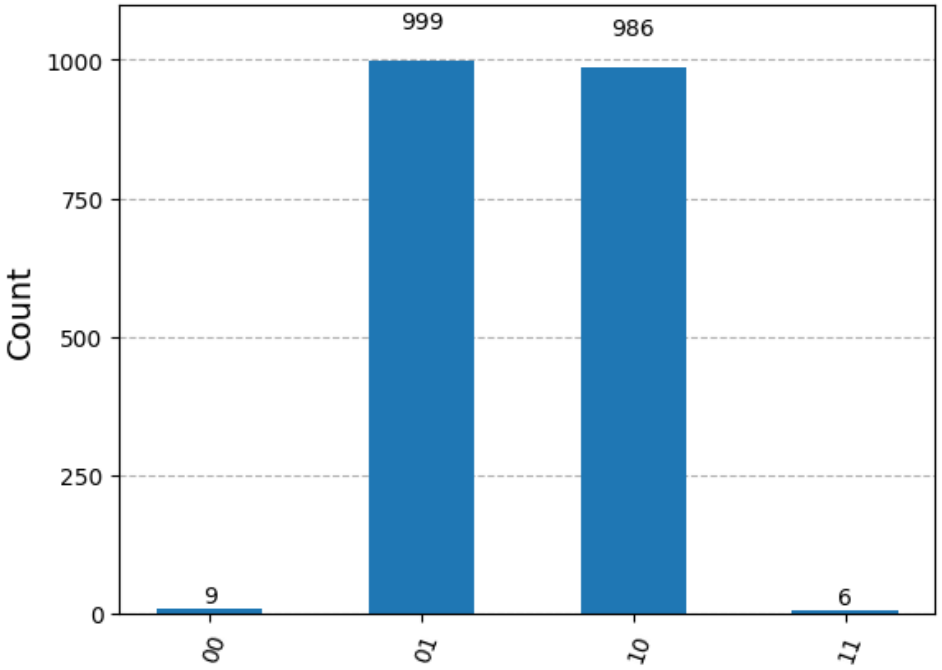


Measurement result on GenericBackend

- Circuits are executed with the same steps seen for the AerSimulator

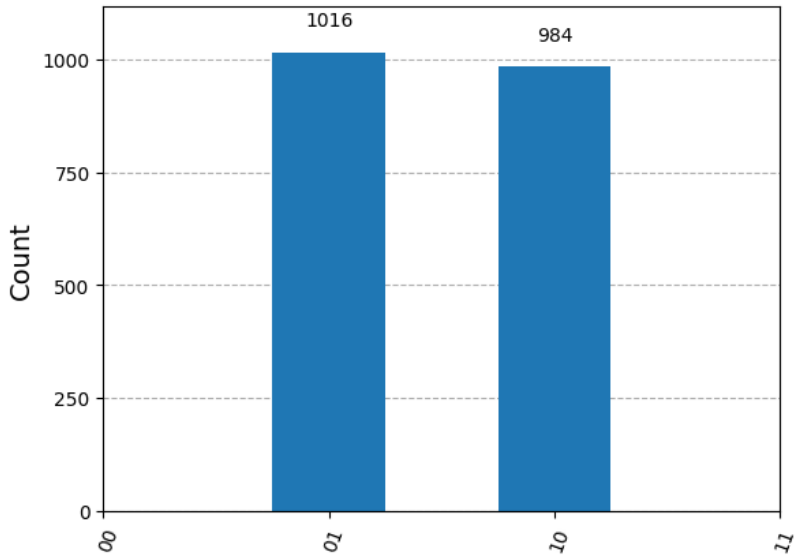
```
job = backend.run(tcirc, shots=2000)
result = job.result();
counts = result.get_counts();
plot_histogram(counts);
```

- Counts now also include spurious measurements of the |00⟩ and |11⟩ states owing to the inclusion of QP nonidealities

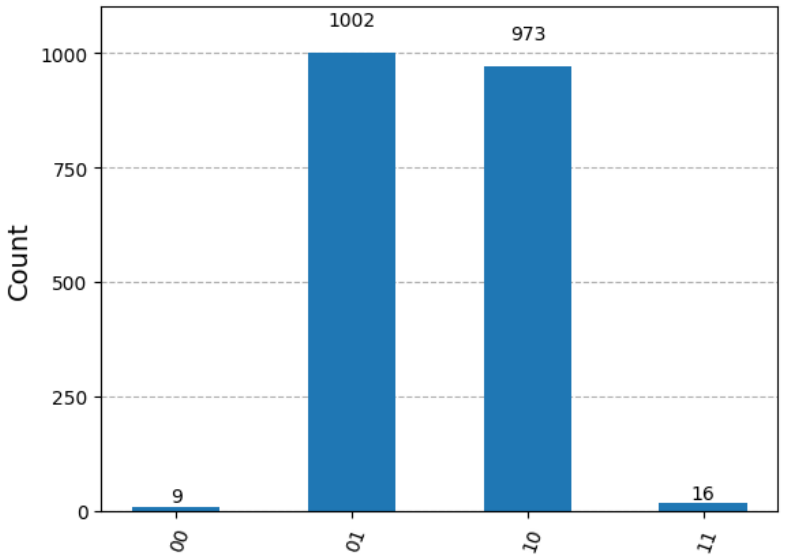


Comparison with measurements on real QP

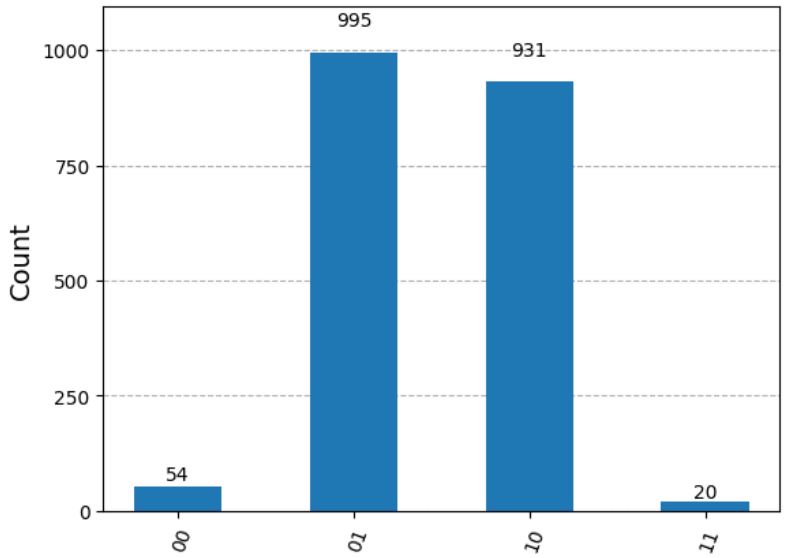
Ideal AerSimulator



GenericBackend



Real QP



- `GenericBackend`s results closely match measurements on real QPs
- Allow tuning of quantum circuit implementations in view of deployment on real hardware

POLITECNICO MILANO 1863 | DEPARTMENT OF ELECTRONICS INFORMATION AND BIOENGINEERING

Hands-on time!
Exercise 2

Contents

Recap

- Quantum circuits can be described by objects of the QuantumCircuit class
- Qubit states are described by objects of the Statevector class
- Statevector, QuantumCircuit and Operator class allow to fully characterize QCs in the quantum domain
- AerSimulator allows high-level QC simulation, including measurement (classical domain)
- GenericBackends provide insights into real-world quantum processor nonidealities

What's next?

- So far, we are treating quantum gates as black-boxes applying some equivalent operator to the state
- How are quantum gates implemented at the hardware level?
  - → topic of Laboratory 2: Hardware qubit gates

- Real quantum processors are plagued by nonidealities that affect computation results
- How can we characterize these nonidealities and fine-tune our hardware implementations?
  - → topic of Laboratory 3: Quantum system characterization