Tutorial: Introduction to quantum software engineering

Yoann Marquer,
Domenico Bianculli

Luxembourg National Research Fund

Supported by SES and the Luxembourg National Research Fund (FNR) IPBG19/14016225/INSTRUCT.

UNIVERSITY OF LUXEMBOURG

The University of Luxembourg

The University of Luxembourg is a research university with a distinctly **international**, **multilingual** and **interdisciplinary** character.

The University's ambition is to provide the **highest quality research** and teaching in its chosen fields and to generate a positive scientific, educational, social, cultural and societal impact in Luxembourg and the Greater Region.

UNIVERSITY OF LUXEMBOURG

Ranked

20th Young University

worldwide and #4 worldwide for its "international outlook" in the Times Higher Education (THE) World University Rankings 2024

7500 students

1000+ PhDs

300 faculty members

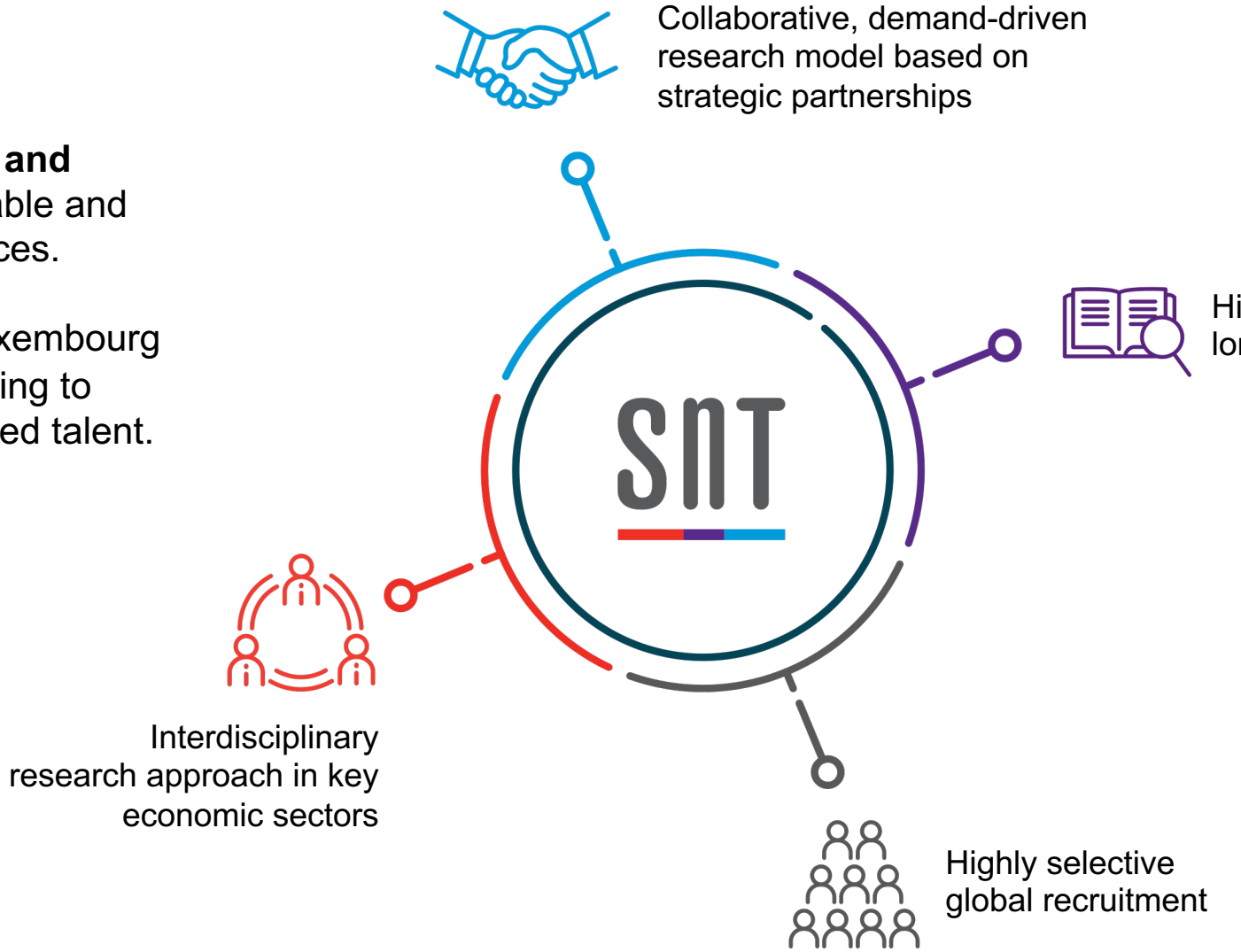
140 nationalities

60% international students

Our Vision

A **leading international research and innovation centre** in secure, reliable and trustworthy ICT systems and services.

We play an instrumental role in Luxembourg by boosting R&D investments leading to economic growth and highly qualified talent.



Software Verification and Validation (SVV) group

Engineering

Compliant, Safe, and Secure

AI-based

Cyber-physical

Quantum

Systems

- Head of the group: Prof. Domenico Bianculli
- 22 members (as of 1/12/2025)
  - 2 professors
  - 2 research scientists
  - 13 research associates (postdoc)
  - 5 PhD candidates
- Focus on applicability and scalability
- Fields of application: space, FinTech, legal, automotive, and e-government

Testing and analysis

- Functional Safety of AI-based systems
- Security testing
- Program analysis for security

Run-time Verification

- Specification languages
- Run-time monitoring
- AIOps: Log analysis and anomaly detection

Requirement Engineering

- Automated Regulatory Compliance
- AI for RE and RE for AI
- Quality assurance of requirements

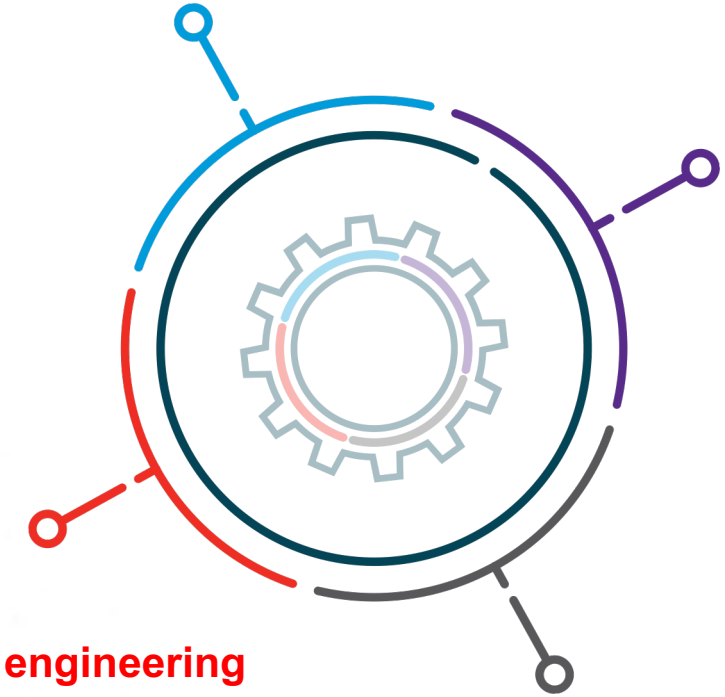
Design-time Verification

- Verification of quantum programs
- Simulation of CPS models
- Schedulability Analysis

Tutorial

Presentation

1. **Quantum computing**
2. **Quantum programming**



Exercises – part 1

3. **Simulation of quantum executions (Qiskit)**

Presentation

4. **Quantum software engineering**

Exercises – part 2

5. **Simulation of quantum executions (pyGSTi)**

   + Quantum noise analysis (bonus)

An overview of quantum computing

SnT

UNIVERSITY OF LUXEMBOURG

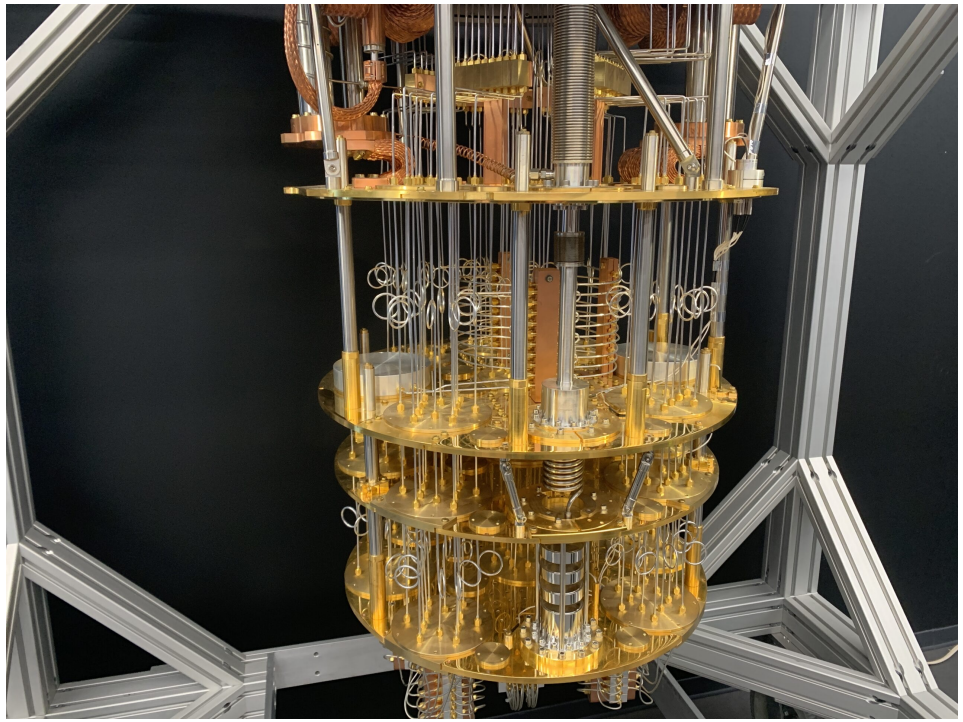
Quantum Computing

Quantum computing uses quantum bits (**qubits**) instead of the bits used in classical computation.

This enables quantum computers to process information **faster** (for some problems) than any classical computer (quantum supremacy).

Quantum computing has an impact on many emerging technologies and **industrial use-cases**, especially regarding optimization problems.

For this reason, technology giants such as IBM, Google, and Microsoft have heavily invested in and committed to quantum computing.



An IBM quantum computer

Qubit states

In classical computing, a **bit** can be **0** or **1**.

In quantum computing, a system can be in two states denoted $|0\rangle$ and $|1\rangle$, using the bra-ket notation.
A **qubit** is in a **superposition** of these states:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

α and β are **not probabilities**, as in "the cat is 50% alive and 50% dead".
α and β are complex numbers called **amplitudes** and satisfying $|\alpha|^2 + |\beta|^2 = 1$.

For instance, the following state is in equal superposition (intuition: in the "diagonal" of $|0\rangle$ and $|1\rangle$):

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

As opposed to classical bits, qubit states are **continuous**.
$|\alpha|^2 + |\beta|^2 = 1$ indicates that α and β live in a circle or a sphere…

Qubit states

The Bloch sphere

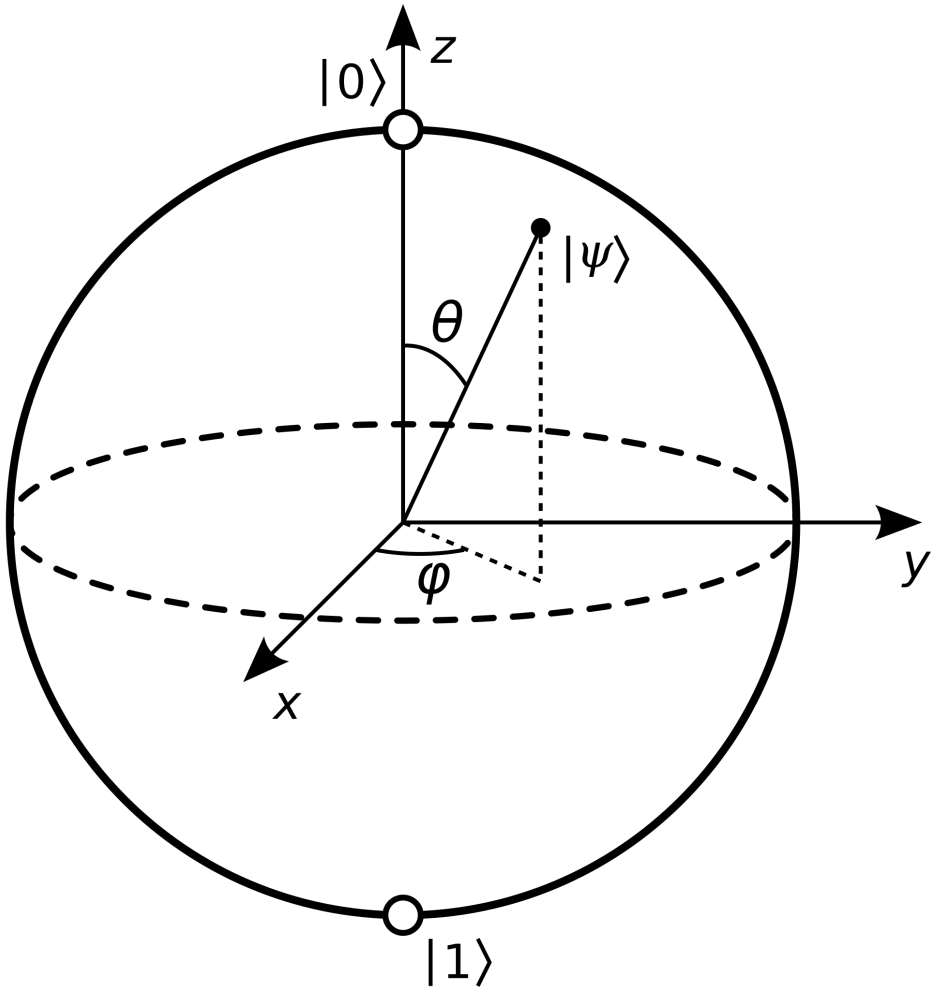
In a qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the complex numbers α and β can be written in polar coordinates

$$\alpha = |\alpha|e^{i\varphi_\alpha}$$
$$\beta = |\beta|e^{i\varphi_\beta}$$

The **Bloch sphere** is a convenient graphical representation of a qubit state:

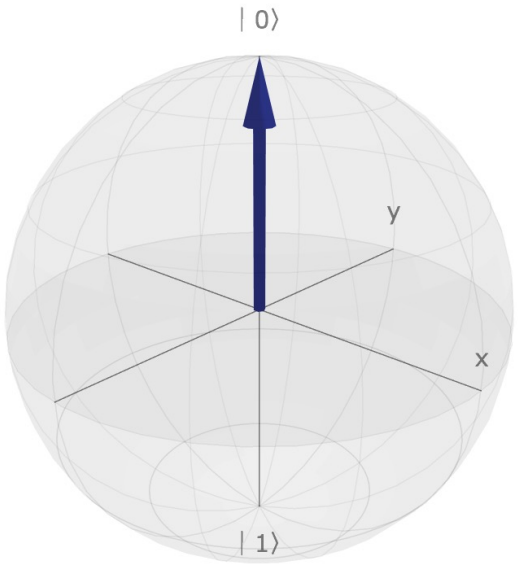
$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle$$

- Where $\varphi = \varphi_\beta - \varphi_\alpha$ is the **relative phase**
- And θ controls the repartition of the **magnitudes** $|\alpha|$ and $|\beta|$ between $|0\rangle$ and $|1\rangle$

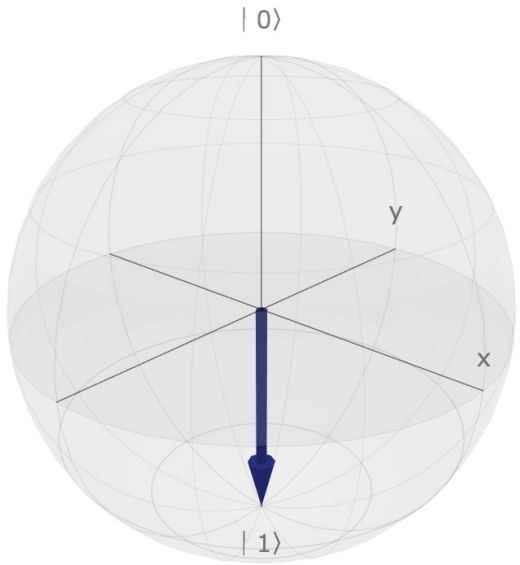


Qubit states

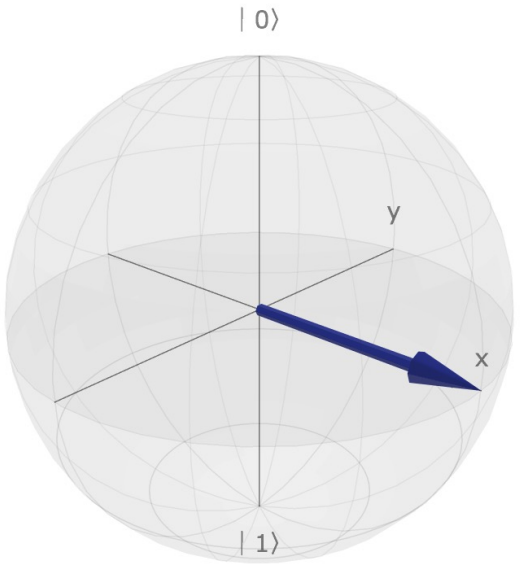
The Bloch sphere: state examples



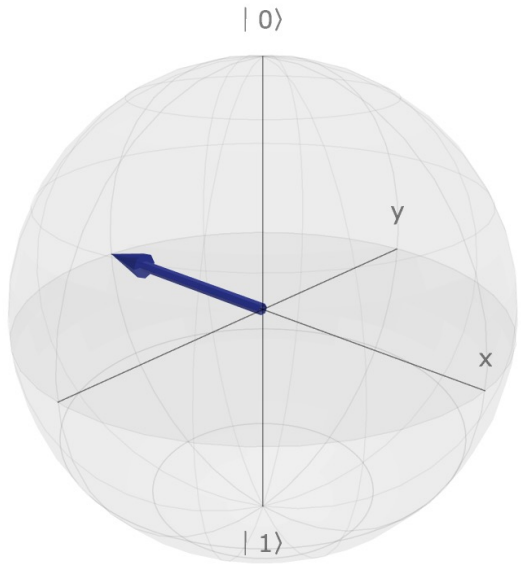
$|0\rangle$



$|1\rangle$



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



$\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle = |-\rangle$

The last two states have the same magnitudes but different relative phases, i.e., $\varphi = 0$ and π respectively

Quantum circuits

In classical computing, **logic gates** are the basic blocks of **circuits**.

In quantum computing, **quantum gates** are the basic blocks of **quantum circuits**. Qbits are represented by wires.

In **vector notation**, $|0\rangle$ and $|1\rangle$ form a basis of the space:

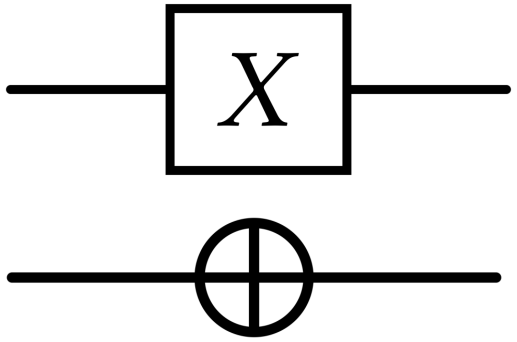
$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \; and \; |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

So, each qubit state can be represented as a vector:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Quantum gates are represented in vector notation using unitary matrices…

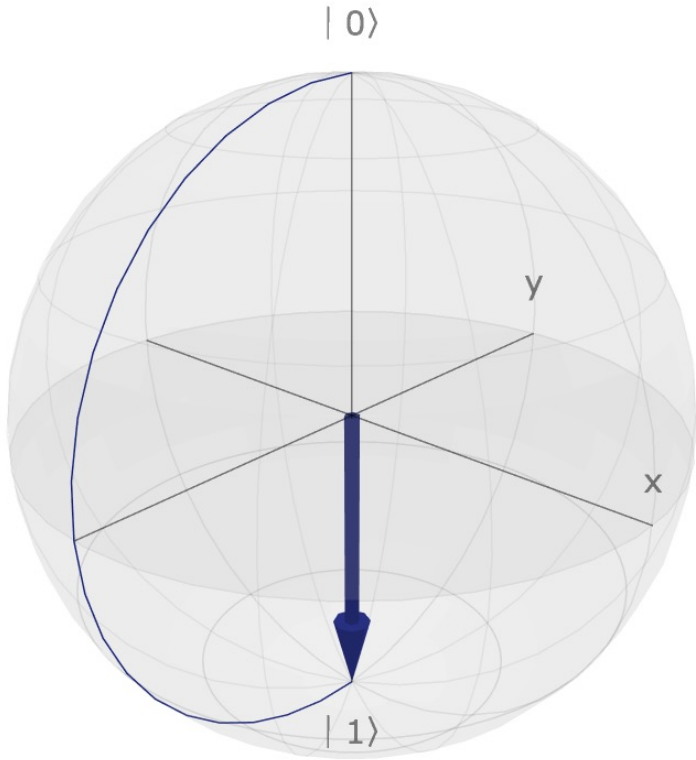
Quantum gates



Example: X gate

The Pauli **X gate** corresponds to a **rotation around the x axis** in the Bloch sphere.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$



For instance:

$$X\,|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

More generally:

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

The X gate is also called the **NOT gate**, since:

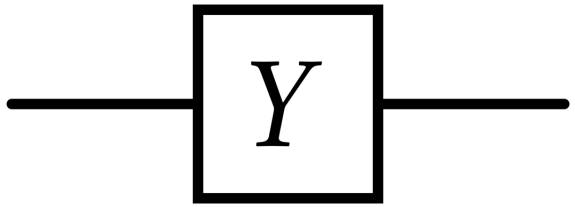
$$X\,|0\rangle = |1\rangle$$
$$X\,|1\rangle = |0\rangle$$

It is also called the **bit-flip gate**, since:

$$X\,(\alpha|0\rangle + \beta|1\rangle) = \alpha|1\rangle + \beta|0\rangle$$

Quantum gates

Example: Y gate



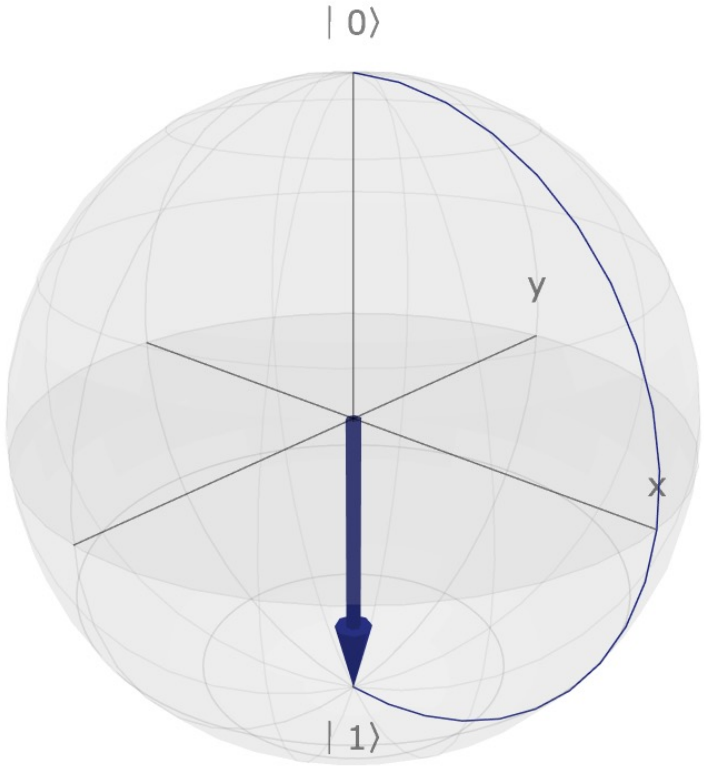
The Pauli **Y gate** corresponds to a **rotation around the y axis** in the Bloch sphere.

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

For instance:

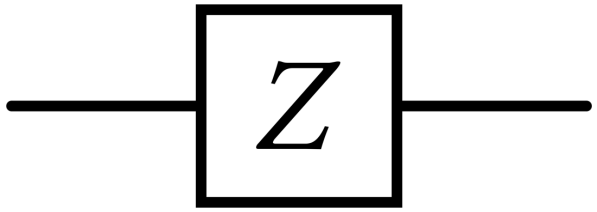
$$Y\,|0\rangle = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = i\begin{bmatrix} 0 \\ 1 \end{bmatrix} = e^{i\frac{\pi}{2}}|1\rangle$$

$e^{i\frac{\pi}{2}}|1\rangle$ and $|1\rangle$ differ only by a **global phase** of $\frac{\pi}{2}$.
In quantum mechanics, no measurement can distinguish states that differ only by a global phase, so they are seen as identical, e.g.,$e^{i\frac{\pi}{2}}|1\rangle = |1\rangle$.



Quantum gates

Example: Z gate



The Pauli **Z gate** corresponds to a **rotation around the z axis** in the Bloch sphere.

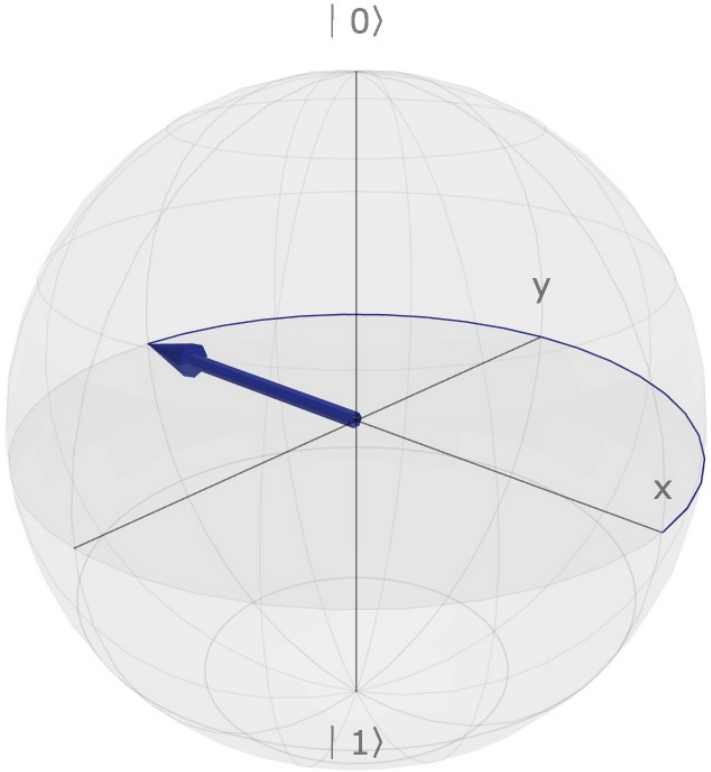
$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

For instance:

$$Z|+\rangle = Z\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle = |-\rangle$$

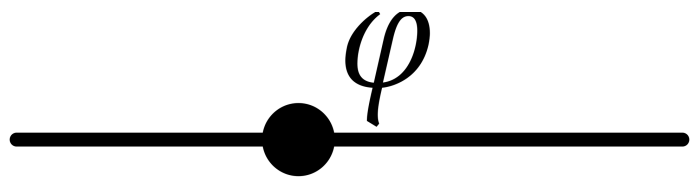
The Z gate is also called the **phase-flip gate**, since it flips the relative phase:

$$Z\left(\alpha|0\rangle + \beta|1\rangle\right) = \alpha|1\rangle - \beta|0\rangle = \alpha|0\rangle + \beta e^{i\pi}|1\rangle$$



Quantum gates

Example: phase gate

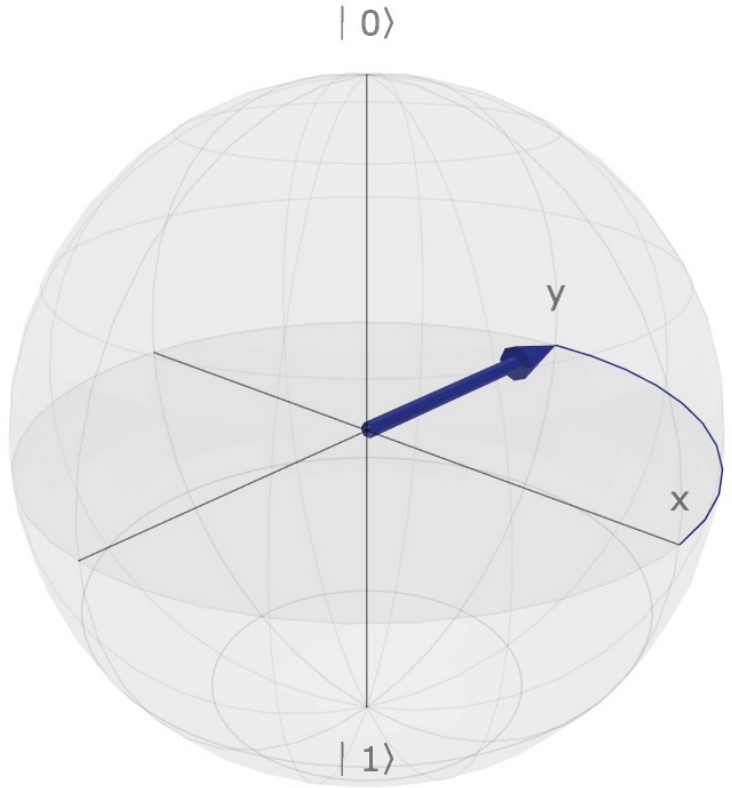


The Pauli Z gate is an occurrence of more general **phase gates** changing the relative phase:

$$P(\varphi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix}$$

For instance:

$$P(\frac{\pi}{2})\,|+\rangle = P(\frac{\pi}{2})(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ i \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle$$



bloch.kherb.io

Quantum gates

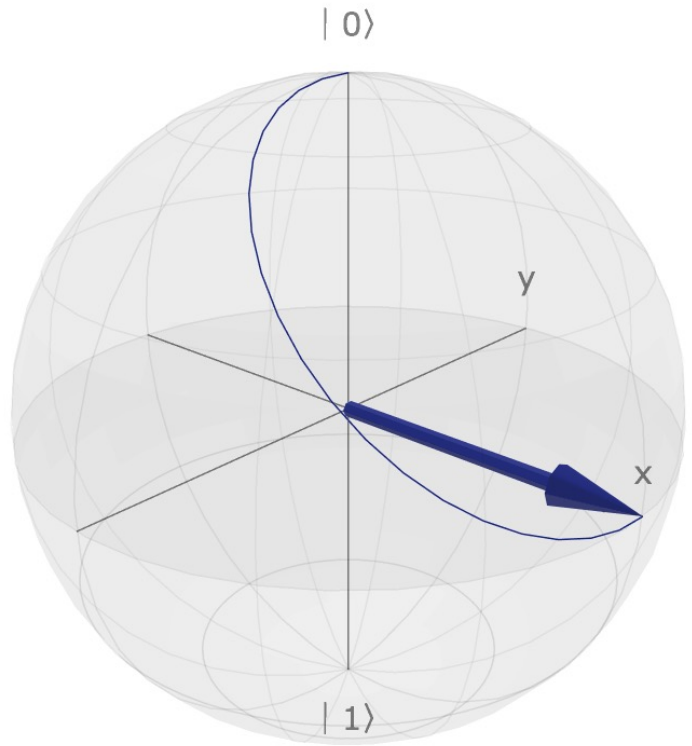
Example: Hadamard gate

Another notable example is the **Hadamard gate**:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$$

It is commonly used to produce a state in superposition from a basis state:

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\ 1\end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = |+\rangle$$



Quantum states

Individual qubits can be combined using an operation called a **tensor product** and denoted $\otimes$.

For instance, on two qubit states:

$$\begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} a \begin{bmatrix} c \\ d \end{bmatrix} \\ b \begin{bmatrix} c \\ d \end{bmatrix} \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}$$

A tensor product $|\psi_1\rangle \otimes |\psi_2\rangle$ between two quantum states $|\psi_1\rangle$ and $|\psi_2\rangle$ is simply denoted $|\psi_1\psi_2\rangle$.

For instance, $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ form a basis for a **2-qubit system**, allowing quantum states with **four amplitudes**:

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}$$

More generally, n **qubits** allow quantum states with 2^n **amplitudes**.
This is one of the reason for the speed-up obtained by quantum computing.

More complex quantum gates can be used on several qubits…

Quantum gates

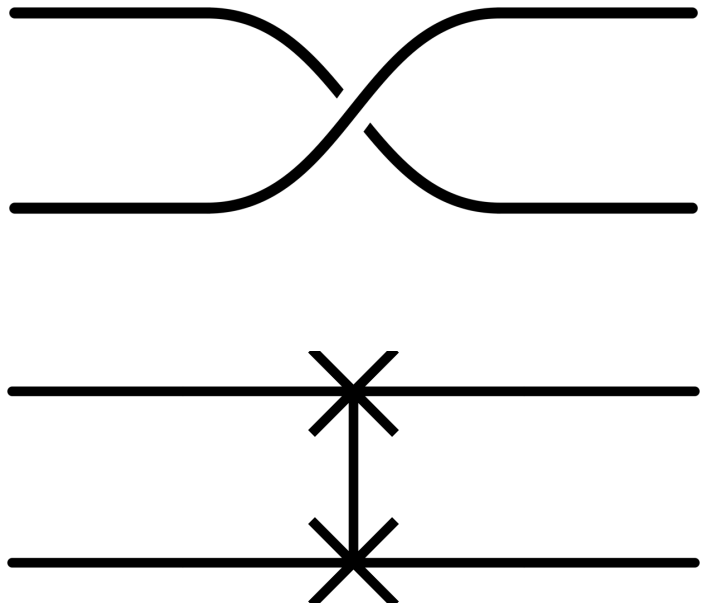
Example: swap gate

The **swap gate** is used to exchange two qubit states:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Indeed:

$$SWAP|\psi_1\psi_2\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix} = \begin{bmatrix} ac \\ bc \\ ad \\ bd \end{bmatrix} = |\psi_2\psi_1\rangle$$

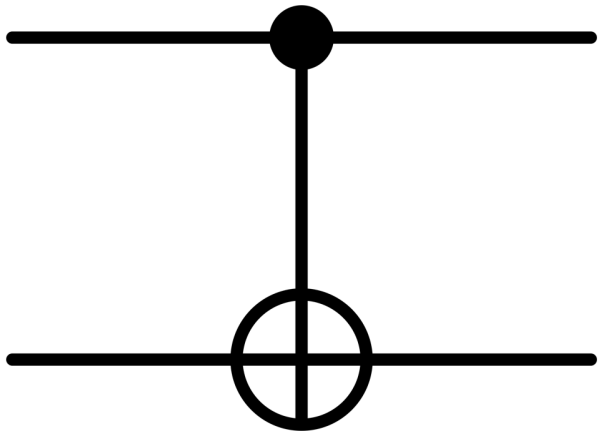


Quantum gates

Example: CNOT gate

The controlled NOT or **CNOT gate** is used to flip the bits of the second (target) qubit when the first (control) qubit is $|1\rangle$

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$



For instance:

$$CNOT|00\rangle = |00\rangle$$
$$CNOT|10\rangle = |11\rangle$$

$$CNOT\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Quantum gates

Controlled gates

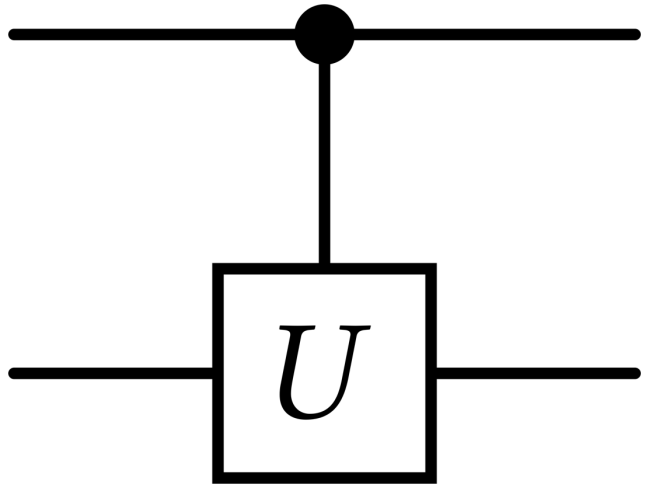
More generally, if U is a quantum gate

$$U = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

The **controlled-U gate** is:

$$CU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & c \\ 0 & 0 & b & d \end{bmatrix}$$

Common examples include the CX (the previous CNOT), CY, and CZ gates.



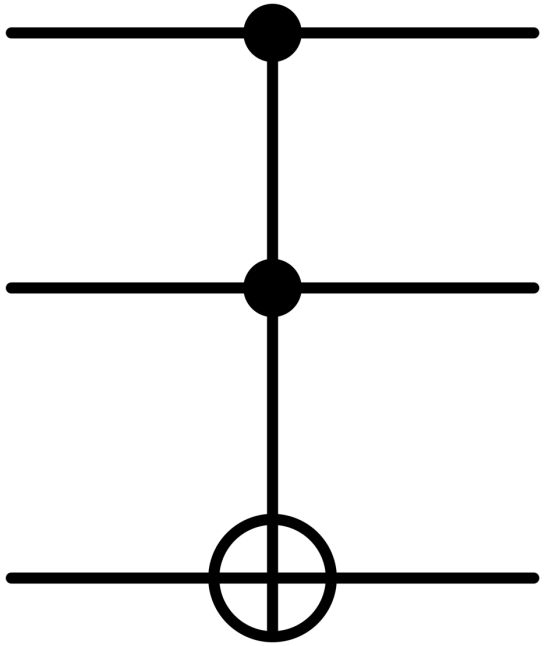
Quantum gates

Controlled gates

The process can be generalized to any number of control qubits.

For instance, the **Toffoli gate** (also known as CCNOT gate) with 2 control qubits.

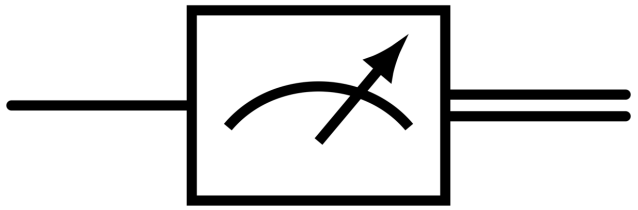
$$CU = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



Quantum measurement

A quantum state can be **measured** according to a direction, usually the z axis of the considered quantum bit(s), called the **computational basis**.

In that case, the quantum state **collapses** to a classical state, with a given probability depending on the magnitude.



For instance, if $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ is measured in the computational basis, it can collapse

- Either in state $|0\rangle$ with probability $|\alpha|^2$
- Or in state $|1\rangle$ with probability $|\beta|^2$

The sum of the probabilities is 1, since $|\alpha|^2 + |\beta|^2 = 1$.

A quantum circuit does not usually provide a single output, but an **output distribution**.

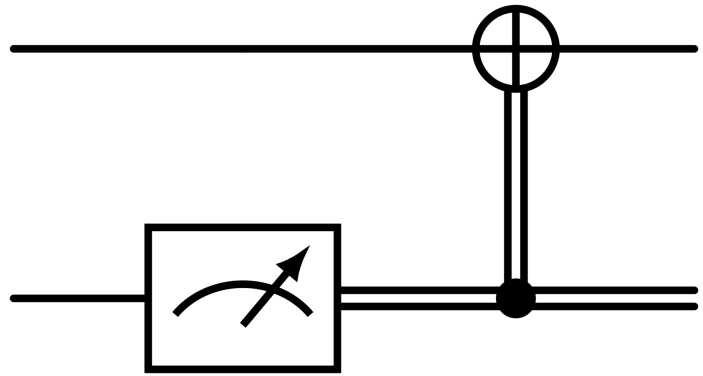
Classical bits are represented in quantum circuits with a doubled line.

Classical control

Classical bits can be used to **control** quantum gates.

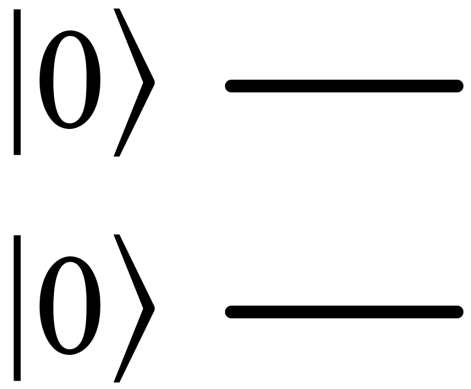
For instance:

- If the classical bit is $|1\rangle$ then the NOT gate is applied on the qubit
- Otherwise, the NOT gate is ignored



Quantum circuit

Example

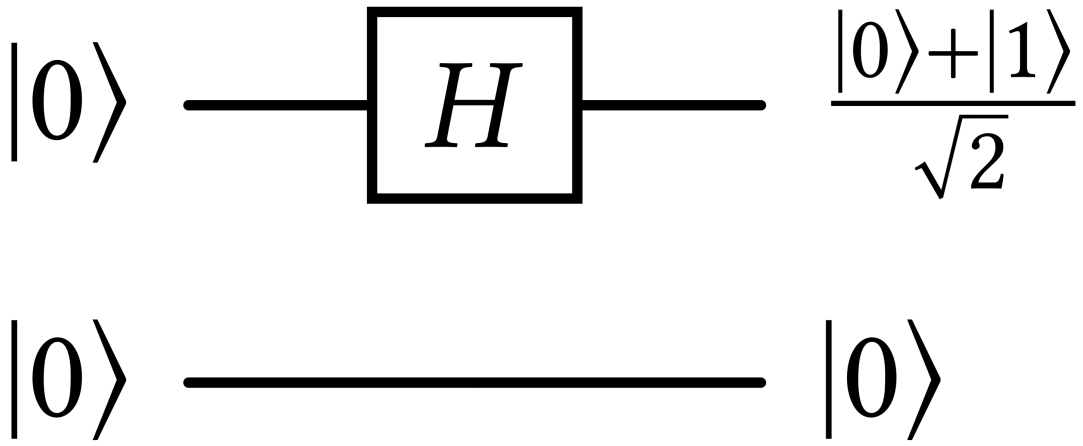


Example of a two-qubit system.

Each qubit is **initialized** to $|0\rangle$.

Quantum circuit

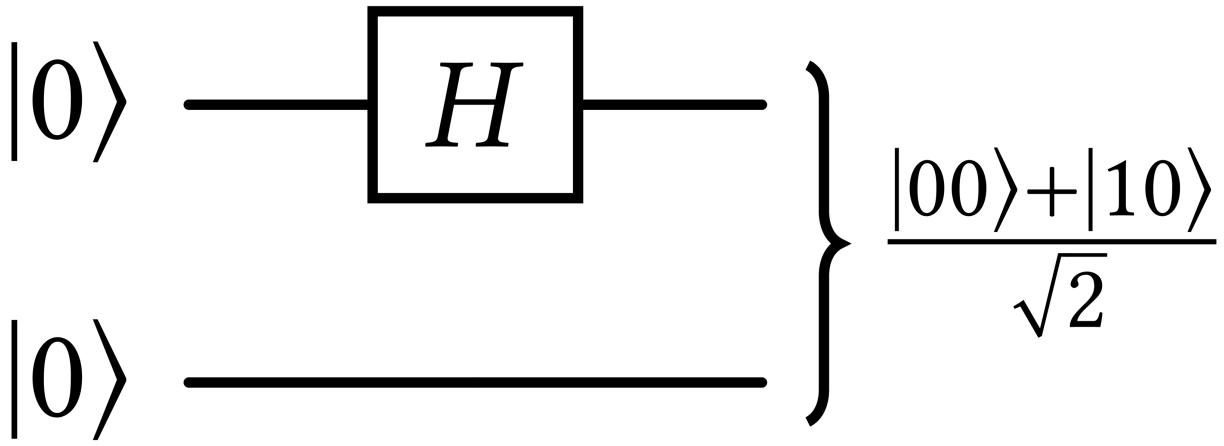
Example



After the **Hadamard** gate, the state of the first qubit is now in (uniform) **superposition**.

Quantum circuit

Example

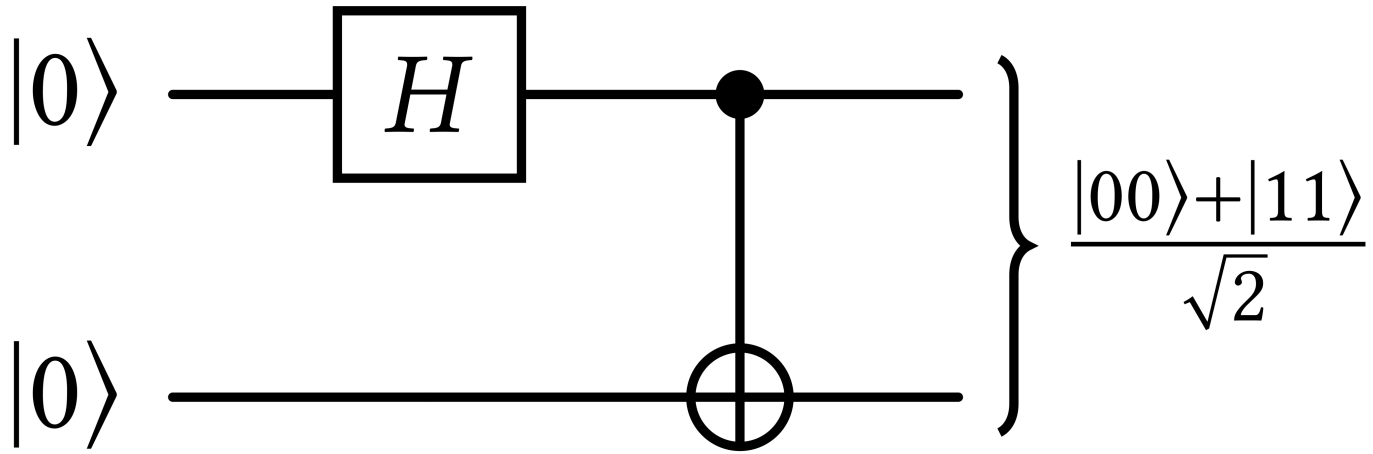


Using a tensor product, we can express the **state of the system** as whole:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$

Quantum circuit

Example

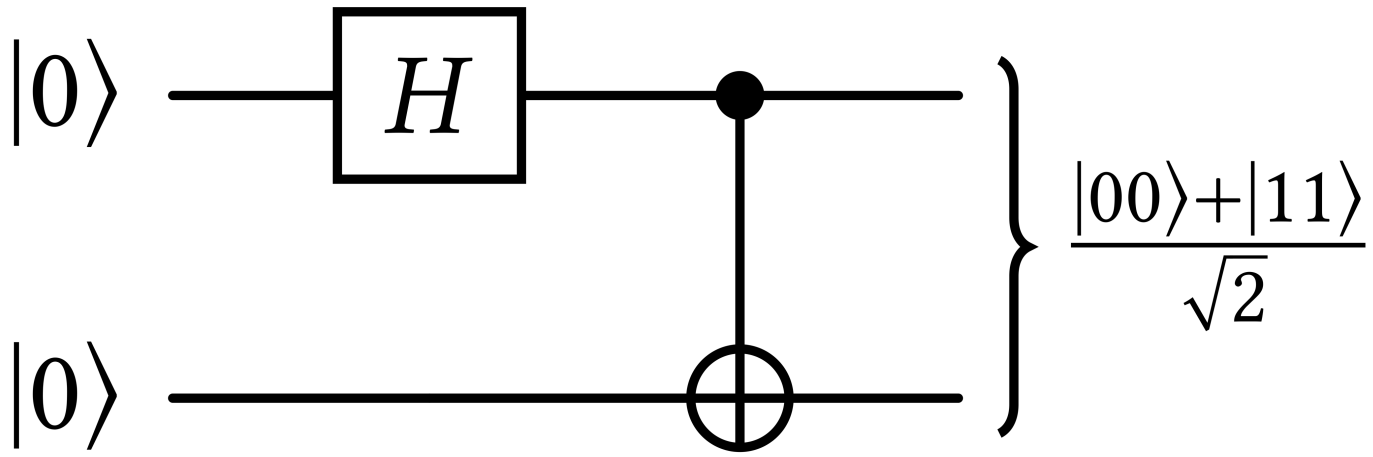


After the **CNOT** gate, the bit of the second (target) qubit is flipped when the first (control) qubit is $|1\rangle$:

$$CNOT\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Quantum circuit

Example



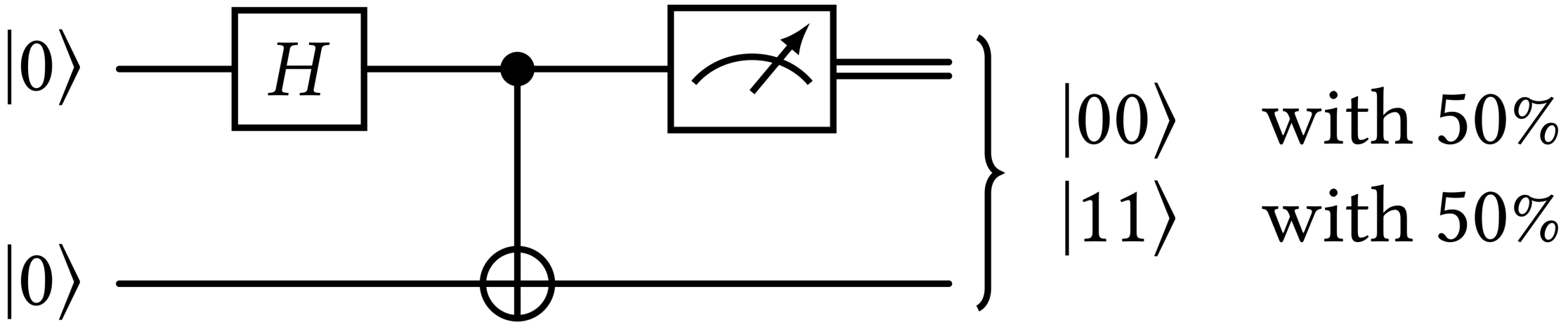
Now, the first qubit is $|0\rangle$ if and only if the second qubit is $|0\rangle$.

This is an example of **quantum entanglement**, called a Bell state.

By contrast, the previous quantum state $\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle$ was not entangled.

Quantum circuit

Example



Finally, we perform a **quantum measurement** on the first qubit, collapsing the quantum state into a classical state.

The measurement on the first qubit instantaneously **collapses** the state of the second (**entangled**) qubit as well.

The output distribution is 50%-50% so, in average, multiple runs would lead to balanced measurements between 00 and 11

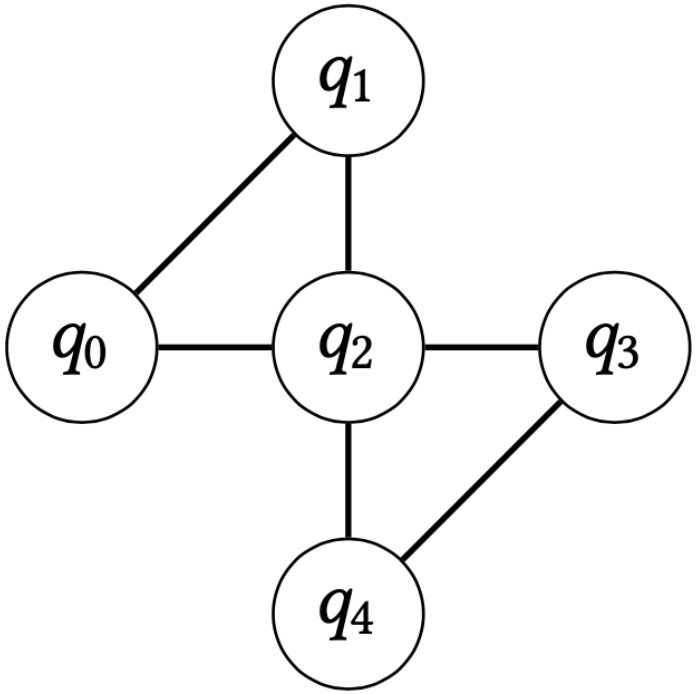
Quantum processing units

In practice, computations are performed on physical devices called **quantum processing units** (QPUs) and based on a variety of technologies (trapped ions, neutral atoms, photonics, etc.).

The **qubit connectivity** of the QPU is how qubit registers are connected to each other.

For instance, an operation between q_0 and q_3 requires
- A SWAP gate between q_2 and q_3
- The operation is made between q_0 and q_2
- A SWAP gate between q_2 and q_3

This is usually done automatically in a step called **transpilation**, which:
- Insert SWAP gates to match the QPU connectivity
- Translate unsupported gates into a combination of supported gates
- Optimize the circuit, e.g., by reducing the number of gates and the depth of the circuit



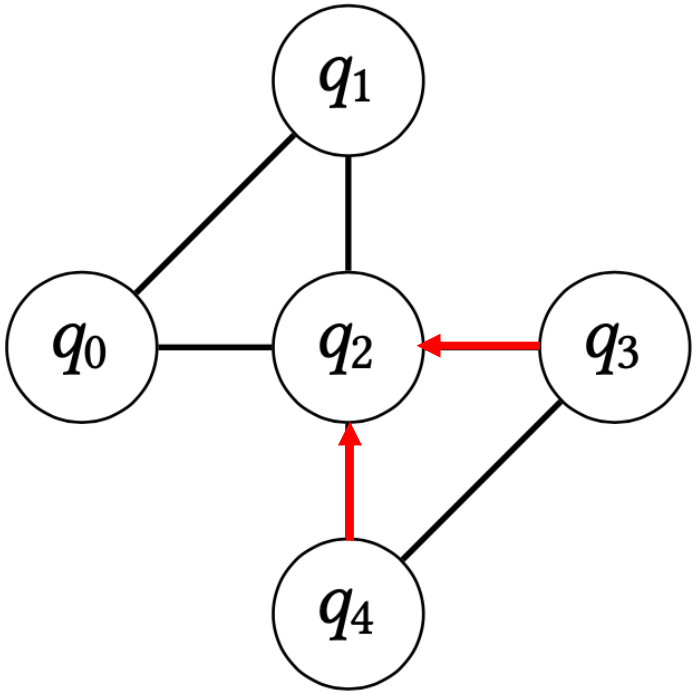
ibmqx2 (Yorktown) QPU

Quantum noise

Current QPUs are called **NISQ** (noisy intermediate-scale quantum) computers
- because they are larger than small-scale prototypes with a few qubits,
- but not large enough so that **quantum error correction** (combining several physical qubits into one less noisy, logical qubit) can be applied.

Sources of noise include:
- **Decoherence**: The spontaneous loss of qubit state information, i.e., qubits suffer from a short lifetime, e.g.,:
  - Relaxation: A qubit state changes from $|1\rangle$ to $|0\rangle$ by losing energy after interacting with the environment
  - Dephasing: A qubit loses phase information
- **Crosstalk**: The unintended interaction of qubits
  - E.g., because of qubit connectivity and operating frequency, q_2 is affected when q_3 and q_4 are used.

It is possible to analyze quantum QPUs to determine noise models.
→ Bonus exercise in part 2



Crosstalk between q3, q4, and q2

Quantum programming

SnT

UNIVERSITY OF LUXEMBOURG

Quantum programming languages

Quantum circuits are usually described using **low-level programming languages embedded in classical languages**

- C: QCL
- C++: Scaffold
- C#: Q#
- Python: ProjectQ, Qiskit, Forest
- F#: Liqui|⟩
- Scala: Chisel-Q
- Haskell: Quiper
- Etc.

Qiskit is massively used in the state of the art[1].
It automatically deals with quantum compilation issues, like:

- **Transpilation** and
- **Uncomputation**: Undo any previously built entanglement by performing inverse operations in backward order.

Propositions for higher-level programming languages have been made but they are not ready yet.

[1] https://www.ibm.com/quantum/qiskit

Quantum programming languages

Example of Qiskit code

```
tq = QuantumRegister(3)
tc0 = ClassicalRegister(1)
tc1 = ClassicalRegister(1)
tc2 = ClassicalRegister(1)
teleport = QuantumCircuit(tq, tc0,tc1,tc2)

teleport.h(tq[1])
teleport.cx(tq[1], tq[2])
teleport.ry(np.pi/4,tq[0])
teleport.cx(tq[0], tq[1])
teleport.h(tq[0])
teleport.barrier()
teleport.measure(tq[0], tc0[0])
teleport.measure(tq[1], tc1[0])
teleport.cx(tq[1], tq[2])
teleport.cz(tq[0], tq[2])
teleport.measure(tq[2], tc2[0])

backend = Aer.get_backend("qasm_simulator")
job = execute(teleport, backend, shots=1, memory=True).result()
result = job.get_memory()[0]
```

Quantum programming languages

Example of Qiskit code

```
tq = QuantumRegister(3)# instantiate a quantum register with three qubits
tc0 = ClassicalRegister(1)# instantiate a classical register
tc1 = ClassicalRegister(1)# instantiate a classical register
tc2 = ClassicalRegister(1)# instantiate a classical register
teleport = QuantumCircuit(tq, tc0,tc1,tc2)# instantiate a quantum circuit

teleport.h(tq[1])
teleport.cx(tq[1], tq[2])
teleport.ry(np.pi/4,tq[0])
teleport.cx(tq[0], tq[1])
teleport.h(tq[0])
teleport.barrier()
teleport.measure(tq[0], tc0[0])
teleport.measure(tq[1], tc1[0])
teleport.cx(tq[1], tq[2])
teleport.cz(tq[0], tq[2])
teleport.measure(tq[2], tc2[0])

backend = Aer.get_backend("qasm_simulator")
job = execute(teleport, backend, shots=1, memory=True).result()
result = job.get_memory()[0]
```

Initialization of the circuit

Quantum and classical registers are variables of the program and correspond to the wires in the circuit

Quantum programming languages

Example of Qiskit code

```
tq = QuantumRegister(3)
tc0 = ClassicalRegister(1)
tc1 = ClassicalRegister(1)
tc2 = ClassicalRegister(1)
teleport = QuantumCircuit(tq, tc0,tc1,tc2)

teleport.h(tq[1])# Hadamard gate applied to qubit 1
teleport.cx(tq[1], tq[2])# NOT gate controlled by qubit 1 and applied to qubit 2
teleport.ry(np.pi/4,tq[0])# rotation of qubit 0 by pi/4 on the y axis
teleport.cx(tq[0], tq[1])# NOT gate controlled by qubit 0 and applied to qubit 1
teleport.h(tq[0])# Hadamard gate applied to qubit 0
teleport.barrier()# separate subcircuit before and after for compilation purpose
teleport.measure(tq[0], tc0[0])# measurement of qbit 0 stored in bit 0
teleport.measure(tq[1], tc1[0])# measurement of qbit 1 stored in bit 1
teleport.cx(tq[1], tq[2])# NOT gate controlled by qubit 1 and applied to qubit 2
teleport.cz(tq[0], tq[2])#   Z gate controlled by qubit 0 and applied to qubit 2
teleport.measure(tq[2], tc2[0])# measurement of qbit 2 stored in bit 2

backend = Aer.get_backend("qasm_simulator")
job = execute(teleport, backend, shots=1, memory=True).result()
result = job.get_memory()[0]
```

Quantum operations

Added in sequence to define the circuit

Quantum programming languages

Example of Qiskit code

```
tq = QuantumRegister(3)
tc0 = ClassicalRegister(1)
tc1 = ClassicalRegister(1)
tc2 = ClassicalRegister(1)
teleport = QuantumCircuit(tq, tc0,tc1,tc2)

teleport.h(tq[1])
teleport.cx(tq[1], tq[2])
teleport.ry(np.pi/4,tq[0])
teleport.cx(tq[0], tq[1])
teleport.h(tq[0])
teleport.barrier()
teleport.measure(tq[0], tc0[0])
teleport.measure(tq[1], tc1[0])
teleport.cx(tq[1], tq[2])
teleport.cz(tq[0], tq[2])
teleport.measure(tq[2], tc2[0])

backend = Aer.get_backend("qasm_simulator")# get backend for  the Aer simulator
job = execute(teleport, backend, shots=1, memory=True).result()# execute the
    circuit with 1 run and return the result
result = job.get_memory()[0]# get the memory state of the first run
```

Execution of the circuit

The content of the memory can be retrieved to observe the result

Quantum programming languages

A more complex example

Problem:

- We want to find M “special” items amongst N items.
- We can easily test if an item is special, using an oracle f

Classical computing requires $O(N)$ steps

- By (randomly) selecting each item x, then checking $f(x)$

Grover’s algorithm is optimal (on a quantum computer) and requires… How many steps?

- $O(N)$?
- $O(\sqrt{N})$?
- $O(\log(N))$?
- $O(1)$?

Quantum programming languages

A more complex example

Problem:
- We want to find M “special” items amongst N items.
- We can easily test if an item is special, using an oracle f

Classical computing requires $\mathrm{O(N)}$ steps
- By (randomly) selecting each item x, then checking $\mathrm{f(x)}$

Grover’s algorithm is optimal (on a quantum computer) and requires… How many steps?
- ~~$\mathrm{O(N)}$~~: Quantum supremacy is possible for this problem
- $\mathrm{O}(\sqrt{N})$
- ~~$\mathrm{O(log(N))}$~~: Even if n qubits allow quantum states with 2^n amplitudes, not all speedups are exponential
- ~~$\mathrm{O(1)}$~~: Quantum computing is not just “massive parallelization”

Quantum programming languages

Example of PennyLane code[1]

An implementation of the **Grover's algorithm** using the **PennyLane** framework.

```
import matplotlib.pyplot as plt
import pennylane as qml
import numpy as np
```

[1] https://pennylane.ai/

Quantum programming languages

Example of PennyLane code

An implementation of the **Grover's algorithm** using the **PennyLane** framework.

The algorithm assumes an **oracle function** f such that
- $f(x) = -x$ if x is the index of one of the M items we search for
- $f(x) = x$ otherwise

(It flips the sign of the magnitude)

```
import matplotlib.pyplot as plt
import pennylane as qml
import numpy as np
```

```
def oracle(wires, omega):
    qml.FlipSign(omega, wires=wires)
```

Quantum programming languages

Example of PennyLane code

```
NUM_QUBITS = 5

omega = np.array([np.zeros(NUM_QUBITS), np.ones(NUM_QUBITS)])

M = len(omega)
N = 2**NUM_QUBITS
wires = list(range(NUM_QUBITS))

dev = qml.device("default.qubit", wires=NUM_QUBITS)
```

```
@qml.qnode(dev)
def circuit():
    iterations = int(np.round(np.sqrt(N / M) * np.pi / 4))

    # Initial state preparation
    equal_superposition(wires)

    # Grover's iterator
    for _ in range(iterations):
        for omg in omega:
            oracle(wires, omg)
        qml.templates.GroverOperator(wires)

    return qml.probs(wires=wires)


results = qml.snapshots(circuit)()
```

We search for $M = 2$ possible states amongst $N = 2^5 = 32$ possible ones: $|00 \ldots 00\rangle$ and $|11 \ldots 11\rangle$

Quantum programming languages

Example of PennyLane code

```
NUM_QUBITS = 5

omega = np.array([np.zeros(NUM_QUBITS), np.ones(NUM_QUBITS)])

M = len(omega)
N = 2**NUM_QUBITS
wires = list(range(NUM_QUBITS))

dev = qml.device("default.qubit", wires=NUM_QUBITS)

@qml.qnode(dev)
def circuit():
    iterations = int(np.round(np.sqrt(N / M) * np.pi / 4))

    # Initial state preparation
    equal_superposition(wires)

    # Grover's iterator
    for _ in range(iterations):
        for omg in omega:
            oracle(wires, omg)
        qml.templates.GroverOperator(wires)

    return qml.probs(wires=wires)


results = qml.snapshots(circuit)()
```

The quantum state is **initialized** in equal superposition

```
def equal_superposition(wires):
    for wire in wires:
        qml.Hadamard(wires=wire)
```

Quantum programming languages

Example of PennyLane code

```
NUM_QUBITS = 5

omega = np.array([np.zeros(NUM_QUBITS), np.ones(NUM_QUBITS)])

M = len(omega)
N = 2**NUM_QUBITS
wires = list(range(NUM_QUBITS))

dev = qml.device("default.qubit", wires=NUM_QUBITS)

@qml.qnode(dev)
def circuit():
    iterations = int(np.round(np.sqrt(N / M) * np.pi / 4))

    # Initial state preparation
    equal_superposition(wires)

    # Grover's iterator
    for _ in range(iterations):
        for omg in omega:
            oracle(wires, omg)
        qml.templates.GroverOperator(wires)

    return qml.probs(wires=wires)


results = qml.snapshots(circuit)()
```

The rest of the program is **iterated** $\frac{\pi}{4}\sqrt{\frac{N}{M}}$ times so the probability of obtaining a correct outcome is close to 1

This bound is obtained using mathematical analysis

Each step will **amplify the magnitude** of the correct answers and reduce the magnitude of the incorrect ones

Quantum programming languages

Example of PennyLane code

```
NUM_QUBITS = 5

omega = np.array([np.zeros(NUM_QUBITS), np.ones(NUM_QUBITS)])

M = len(omega)
N = 2**NUM_QUBITS
wires = list(range(NUM_QUBITS))

dev = qml.device("default.qubit", wires=NUM_QUBITS)

@qml.qnode(dev)
def circuit():
    iterations = int(np.round(np.sqrt(N / M) * np.pi / 4))

    # Initial state preparation
    equal_superposition(wires)

    # Grover's iterator
    for _ in range(iterations):
        for omg in omega:
            oracle(wires, omg)
        qml.templates.GroverOperator(wires)

    return qml.probs(wires=wires)


results = qml.snapshots(circuit)()
```

```
def oracle(wires, omega):
    qml.FlipSign(omega, wires=wires)
```

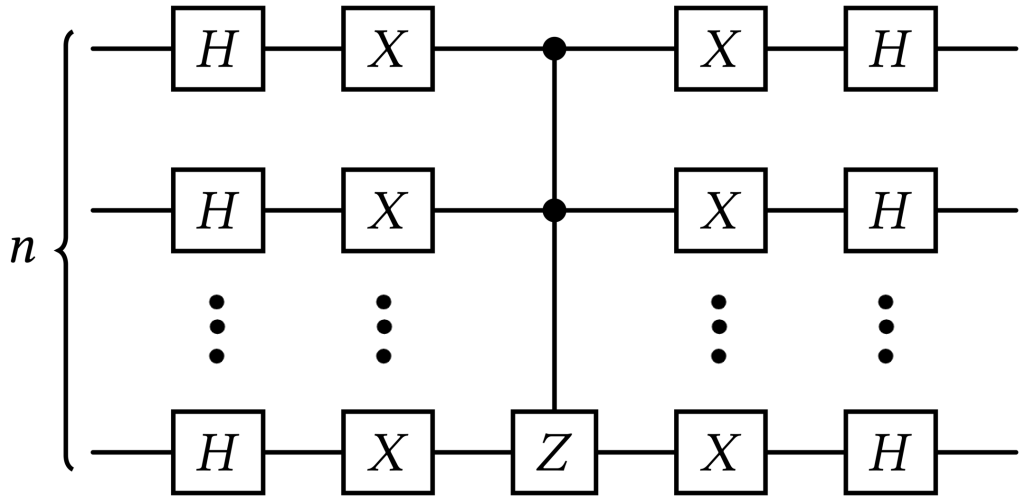
The **phases** of the correct answers **are flipped**

Quantum programming languages

Example of PennyLane code

```
NUM_QUBITS = 5

omega = np.array([np.zeros(NUM_QUBITS), np.ones(NUM_QUBITS)])

M = len(omega)
N = 2**NUM_QUBITS
wires = list(range(NUM_QUBITS))

dev = qml.device("default.qubit", wires=NUM_QUBITS)

@qml.qnode(dev)
def circuit():
    iterations = int(np.round(np.sqrt(N / M) * np.pi / 4))

    # Initial state preparation
    equal_superposition(wires)

    # Grover's iterator
    for _ in range(iterations):
        for omg in omega:
            oracle(wires, omg)
        qml.templates.GroverOperator(wires)

    return qml.probs(wires=wires)


results = qml.snapshots(circuit)()
```

GroverOperator implements the following circuit, where *n* is the number of qubits



Quantum programming languages

Example of PennyLane code

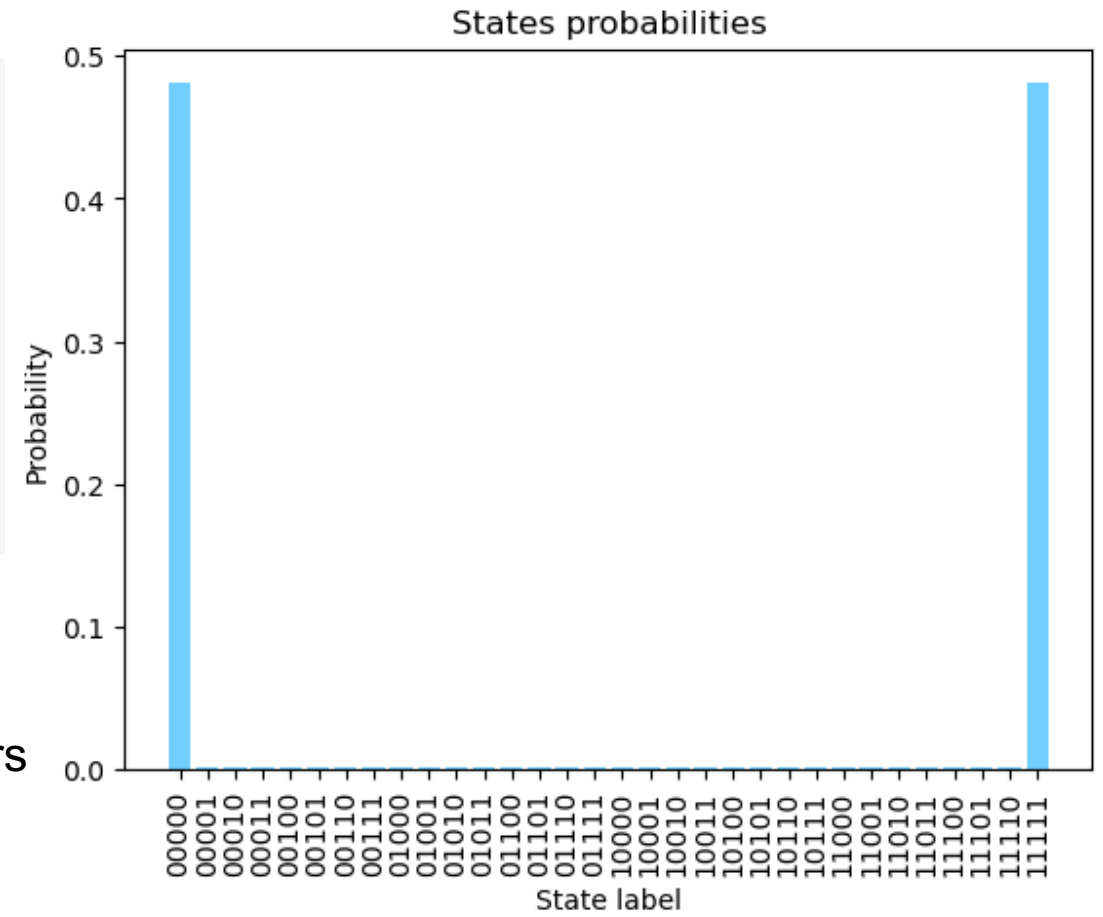
```
NUM_QUBITS = 5

omega = np.array([np.zeros(NUM_QUBITS), np.ones(NUM_QUBITS)])

M = len(omega)
N = 2**NUM_QUBITS
wires = list(range(NUM_QUBITS))

dev = qml.device("default.qubit", wires=NUM_QUBITS)

@qml.qnode(dev)
def circuit():
    iterations = int(np.round(np.sqrt(N / M) * np.pi / 4))

    # Initial state preparation
    equal_superposition(wires)

    # Grover's iterator
    for _ in range(iterations):
        for omg in omega:
            oracle(wires, omg)
        qml.templates.GroverOperator(wires)

    return qml.probs(wires=wires)


results = qml.snapshots(circuit)()
```

The **probabilities** of the different states are returned

Quantum programming languages

Example of PennyLane code

```
y = results["execution_results"]
bit_strings = [f"{x:0{NUM_QUBITS}b}" for x in range(len(y))]

plt.bar(bit_strings, results["execution_results"], color = "#70CEFF")

plt.xticks(rotation="vertical")
plt.xlabel("State label")
plt.ylabel("Probability")
plt.title("States probabilities")

plt.show()
```



These **probabilities** are plotted, revealing the correct answers $|00 \ldots 00\rangle$ and $|11 \ldots 11\rangle$ have a larger magnitude

This computation was performed using a quantum simulator…

Quantum simulators

Quantum executions can be costly and noisy, even when there are enough qubits available to run them in the first place. Hence, **quantum simulators** can be useful to test quantum circuits on classical computers.

In that case, the execution can be stopped at any moment so intermediate states as well as phase information can be observed at will, facilitating debugging.

Since quantum simulations do not benefit from the speed-up due to actual quantum superposition, they are **tractable only for small circuits**, with a few qubits.

Qiskit

Cirq

PENNYLANE

Exercises
Part 1

SnT

UNIVERSITY OF LUXEMBOURG

Installation

Requirements

- A terminal console
- Internet access
- Python 3.12 installed (venv, pip)

Instructions (monospace font -> on the console)

```
1. mkdir Exercises
2. cd Exercises
3. python3.12 -m venv venv
4. source venv/bin/activate
5. pip install --upgrade pip setuptools
6. Prepare the “requirements.txt” file
7. pip install -r requirements.txt
8. mkdir 1_QuantumSimulation
9. cd 1_QuantumSimulation
```

```
Jinja2==3.1.6
MarkupSafe==3.0.2
matplotlib==3.10.1
pygsti==0.9.14
qiskit==1.4.1
qiskit-aer==0.17.0
qiskit-ibm-runtime==0.38.0
tqdm==4.67.1
```

Part 1: Simulation of quantum executions with Qiskit

Exercises

1. Circuit definition and sampling of outputs
2. Transpilation and comparison between ideal and noisy simulations
3. Quantum errors and noise models

Qiskit

Exercise 1.1

In this exercise, we introduce **circuit definition** and a **sampling of outputs**.

Qiskit (Quantum Information Software Kit) is a very common framework for writing and executing quantum circuits, developed by IBM.

Instructions

- Prepare an empty “exercise1.py” file
- For each step:
  - Write the code
  - `python exercise1.py`

Exercise 1.1 – Step 1: Circuit definition (1)

We want to define a circuit generating and measuring a **Bell state**:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Instructions

- Declare `circuit0` with 2 qubits
- Apply the H gate on qubit 0
- Apply the CX gate on qubits 0 (control) and 1 (updated)
- Measure qubits 0 and 1
- Print the obtained circuit

Hints

```
from qiskit import QuantumCircuit
```

`QuantumCircuit` takes as input the number of qubits and generates a new circuit

The `.gate(q)` method of a circuit applies the gate `gate` to a qubit `q` or several qubits [replace `gate` with a suitable gate name, like `h` or `cx`]. For example, `.cx(q1, q2)` applies gate `cx` to qubits `q1` (control) and `q2` (updated) of the circuit

The `.measure_all()` method of a circuit measures all qubits

Exercise 1.1 – Step 1: Circuit definition (1)

```
from qiskit import QuantumCircuit
circuit0 = QuantumCircuit(2)# declaration of 2 qubits
circuit0.h(0)# H gate on qubit 0
circuit0.cx(0, 1)# X gate on qubit 1, controled by qubit 0
circuit0.measure_all()# measure all qubits
print(circuit0)
# expected output:
#
#   q_0: ┤ H ├──■──┤ M ├──
#              ┌─┴─┐└╥┘┌─┐
#   q_1: ──────┤ X ├─╫─┤ M ├
#              └───┘ ║ └╥┘
#meas: 2/═══════════╩══╩═
#                   0  1
```

The `measure_all()` method added a barrier before measurements

At the bottom right, 0 and 1 indicates classical bits that contain the result of the corresponding qubit measurements

They were defined automatically, but we can also define them when creating the circuit, see the next example

Exercise 1.1 – Step 1: Circuit definition (2)

We want to define a circuit generating and measuring a **3-qubit GHZ** (Greenberger–Horne–Zeilinger) state:

$$\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

(The Bell state generated by `circuit0` is a 2-qubit GHZ state, n-qubit GHZ states are very common in quantum communication)

Instructions

- Declare `circuit1` with 3 qubits and 3 classical bits
- Apply the H gate on qubit 0
- Apply the CX gate on qubits 0 and 1
- Apply the CX gate on qubits 1 and 2
- Measure qubits 0, 1, and 2, store the result in bits 2, 1, and 0
- Print the obtained circuit

Hints

`QuantumCircuit` takes as second input the number of classical bits

The `.measure(listQ, listC)` method of a circuit measures qubits in `listQ` and store the result in classical bits in `listC`

Exercise 1.1 – Step 1: Circuit definition (2)

```
circuit1 = QuantumCircuit(3, 3)# declaration of 3 qubits and 3 classical bits
circuit1.h(0)# H gate on qubit 0
circuit1.cx(0, 1)# X gate on qubit 1, controled by qubit 0
circuit1.cx(1, 2)# X gate on qubit 2, controled by qubit 1
circuit1.measure([0, 1, 2], [2, 1, 0])# qubit 0 is measured and the result is stored in bit 2, etc.
print(circuit1)
# expected output:
#      ┌───┐          ┌─┐
#q_0: ┤ H ├──■───────┤M├──────
#      └───┘┌─┴─┐     └╥┘┌─┐
#q_1: ─────┤ X ├──■───╫─┤M├───
#           └───┘┌─┴─┐ ║ └╥┘┌─┐
#q_2: ──────────┤ X ├─╫──╫─┤M├
#                └───┘ ║  ║ └╥┘
#c: 3/═════════════════╩══╩══╩═
#                      2  1  0
```

Since the quantum state should be $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$, we expect outputs '000' and '111' with 50% probability, respectively

Exercise 1.1 – Step 2: Sample quantum results

Qiskit Aer is a package for quantum simulations with realistic noise models

Instructions

- Generate a sampler
- Run a job for `circuit0` and `circuit1` with 128 shots
- From the job results, extract and print the output counts for both circuits

Hints

`from qiskit_aer.primitives import SamplerV2`

`SamplerV2()` generates a sampler

The `.run(circuits, shots = n)` method of a sampler takes as arguments a list of circuits and prepare a job (eventually with a given number `n` of shots)

A job has a `.result()` method, which provides a list a results

A result has a `.data()` method

When a circuit used measure_all, you can use the `.meas.get_counts()` data method

When a circuit used measure, you can use the `.c.get_counts()` data method

Exercise 1.1 – Step 2: Sample quantum results

```
from qiskit_aer.primitives import SamplerV2
sampler = SamplerV2()# sample (noiseless) quantum execution
job = sampler.run([circuit0, circuit1], shots = 128)# run the circuits for a given number of shots (if
    you want to sample outputs for only one circuit, you can use a list of circuits with one element)
result = job.result()# get results
counts0 = result[0].data.meas.get_counts()# when using measure_all as in circuit0, the Qiskit name is
    'meas'
counts1 = result[1].data.c.get_counts()# when explicitly defining identifiers for classical registers
    as in circuit 1, the Qiskit name is 'c'
print(f"{counts0 = }")
print(f"{counts1 = }")
# example of output:
#counts0 = {'00': 63, '11': 65}
#counts1 = {'111': 60, '000': 68}
```

- For **circuit0**, amongst 128 runs, 63 returned '00' and 65 returned '11'
- For **circuit1**, 60 returned '111' and 68 returned '000'

This meets our expectation of balanced output probabilities
(You can run exercise1.py several times to observe different outputs for `circuit0` and `circuit1`)

More generally, sampling is very useful to **approximate the circuit probability distribution** in the ideal case and thus to determine if the circuit performs as expected

Exercise 1.2

In this exercise, we introduce **transpilation** and a compare results from **ideal and noisy simulations**.

Instructions

- Prepare an empty “exercise2.py” file
- For each step:
  - Write the code
  - `python exercise2.py`

Exercise 1.2 – Step 1: Circuit definition

We want to compare an ideal execution with a noisy one.
The difference is very easy to observe with GHZ states, so we want to define (again) a circuit generating and measuring a **3-qubit GHZ** state:

$$\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

Instructions

- Define the appropriate `circuit` for 3 qubits and 3 classical bits
- Measure qubits 0, 1, and 2, and store the result in bits 0, 1, and 2
- Print the circuit

Hints

See exercise 1.1: start by copying the previous circuit definition and adapt it as required by the exercise

Exercise 1.2 – Step 1: Circuit definition

```
circuit = QuantumCircuit(3, 3)# declaration of 3 qubits and 3 classical bits
circuit.h(0)# H gate on qubit 0
circuit.cx(0, 1)# X gate on qubit 1, controled by qubit 0
circuit.cx(1, 2)# X gate on qubit 2, controled by qubit 1
circuit.measure([0, 1, 2], [0, 1, 2])# qubit 0 is measured and the result is stored in bit 0, etc.
print(circuit)
# Expected output:
#
#q_0: ┤ H ├──■──────────┤ M ├
#          ┌─┴─┐        └╥┘
#q_1: ─────┤ X ├──■──────╫─┤ M ├
#          └───┘┌─┴─┐    ║ └╥┘
#q_2: ──────────┤ X ├────╫──╫─┤ M ├
#              └───┘    ║  ║ └╥┘
#c: 3/═══════════════════╩══╩══╩═
#                        0  1  2
```

Exercise 1.2 – Step 2: Ideal execution

While in Exercise 1.1 we introduced a general sampler, in this exercise, we use simulators for quantum processing units (QPUs).

Instructions

- Generate an ideal simulator
- Run a job for `circuit` with 1024 shots
- From the job result, extract and print the output counts

Hints

```
from qiskit_aer import AerSimulator
```

`AerSimulator()` creates an ideal simulator, which has an `.run(circuits, shots = n)` method, like a sampler

A result simply has a `.get_counts(0)` method

Exercise 1.2 – Step 2: Ideal execution

```
from qiskit_aer import AerSimulator
ideal_simulator = AerSimulator()
# We simulate an ideal execution with 1024 shots and get the output counts
job = ideal_simulator.run(circuit, shots = 1024)
result = job.result()
counts = result.get_counts(0)
print(f"ideal {counts = }")
# example of output:
#ideal counts = {'111': 508, '000': 516}
```

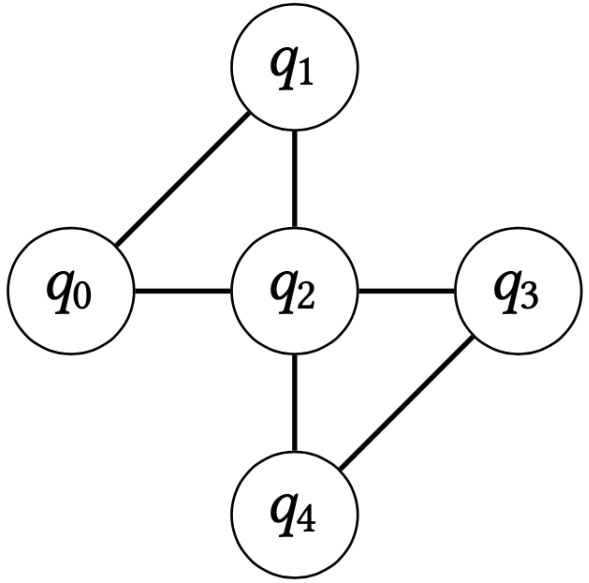
Since the output state is $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$, '000' and '111' are the only expected results

Exercise 1.2 – Step 3: Transpile the circuit (1)

Aer can generate (simplified) **noise models** for a real QPU, using the calibration information obtained on gate errors (for each gate and qubit combination), relaxation times and readout error probability (for each qubit)

For this exercise, we consider the IBM **Yorktown** QPU:

- Made of 5 qubits
- With the connectivity represented on the right
- But… what are the supported gates?



Instructions

- Generate the QPU backend and configuration
- Print the supported instructions

Hints

```
from qiskit_ibm_runtime.fake_provider import FakeYorktownV2
```

`FakeYorktownV2()` creates a backend

A backend has a `.configuration()` method

A configuration has a `.supported_instructions` attribute

Exercise 1.2 – Step 3: Transpile the circuit (1)

```
from qiskit_ibm_runtime.fake_provider import FakeYorktownV2
backend = FakeYorktownV2()
config = backend.configuration()
print(f"supported_gates = {config.supported_instructions}")# print
    the quantum gates supported by the QPU
#supported_gates = ['rz', 'sx', 'id', 'reset', 'delay', 'play', 'u1',
    'measure', 'acquire', 'u2', 'cx', 'setf', 'shiftf', 'u3', 'x']
```

The X gate is supported, but not the H gate…

Exercise 1.2 – Step 3: Transpile the circuit (2)

Since the computation is performed on a (simulated) QPU, the circuit must be **transpiled** to match the QPU connectivity and supported gates

Instructions

- Generate a noisy simulator from the backend
- Transpile the circuit with this simulator and print the result

Hints

`AerSimulator.from_backend` takes a backend as input and generates the corresponding noisy simulator

```
from qiskit import transpile
```

The `transpile` function takes a circuit and simulator as inputs, and returns a transpiled circuit

Exercise 1.2 – Step 3: Transpile the circuit (2)

```
noisy_simulator = AerSimulator.from_backend(backend)
transpiled_circuit = transpile(circuit, noisy_simulator)
print(transpiled_circuit)# print the transpiled circuit
# Expected output:
#global phase: π/4
#
#ancilla_0 -> 0 ──────────────────────────────────────────────────────────────
#
#      q_2 -> 1 ──────────────────────────────────────────────────────X──M──
#
#      q_1 -> 2 ──────────────────────────────────────────────X──────■──M──
#
#      q_0 -> 3 ─┤ Rz(π/2) ├┤ √X ├┤ Rz(π/2) ├──────────────────■──M──
#
#ancilla_1 -> 4 ──────────────────────────────────────────────────────────────
#
#          c: 3/═══════════════════════════════════════════════════0══1══2══
#                                                                  0  1  2
```

The initial circuit was defined on 3 qubits q0, q1, and q2, that are **mapped** to the Yorktown qubits 3, 2, and 1, respectively
Other Yorktown qubits (0 and 4) are not used by the circuit and are labelled as **ancillary qubits**

The H gate was decomposed into a **combination of supported gates**

- **RZ**, a rotation around the Z axis of the Boch sphere, and
- **SX**, the "square root of X", a special rotation (Up to a global phase of π/4) around the X axis such that SX performed twice is equivalent to an X gate, i.e., $SX^2 = X$

Exercise 1.2 – Step 4: Run the transpiled circuit

Similarly to running the ideal (noiseless) execution, we can now run the transpiled circuit with the noisy simulator and observe the results

Instructions

- Use the noisy simulator to run the transpiled circuit with 1024 shots
- Print the output counts
- Print the proportion of expected outputs

Hints (as in Step 2)

```
from qiskit_aer import AerSimulator
```

`AerSimulator()` creates an ideal simulator, which has an `.run(circuits, shots = n)` method, like a sampler

A result simply has a `.get_counts(0)` method

Exercise 1.2 – Step 3: Transpile the circuit (2)

```
job = noisy_simulator.run(transpiled_circuit, shots = 1024)
result = job.result()
counts = result.get_counts(0)
proportion = round(100*(counts['000'] + counts['111'])/1024, 2)
print(f"noisy {counts = }")
print(f"{proportion = }% of expected outputs")
# Example of output:
#noisy counts = {'101': 53, '001': 22, '100': 22, '110': 13, '010':
    55, '000': 412, '011': 21, '111': 426}
#proportion = 81.84% of expected outputs
```

Most (for this output, 81.84%) of the results are '000' and '111', as expected from the output state $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$, but other results may happen due to quantum noise…

Exercise 1.3

In this exercise, we give a small tour of the managerie of **quantum errors**, then we explain how to build custom **noise models** from them

Instructions

- Prepare an empty “exercise3.py” file
- For each step:
  - Write the code
  - `python exercise3.py`

Exercise 1.3 – Step 1: Pauli errors

The Qiskit Aer noise module contains Python classes to build customized noise models for simulation

```
from qiskit_aer.noise import pauli_error
```

Reminder: The X, Y, and Z gates are also called the Pauli gates and they correspond to a π-rotation around the X, Y, and Z axis of the Bloch sphere, respectively.

A Pauli error is a small percentage of applying a Pauli gate on a given qubit

Instructions

- Define and print a bit flip (Pauli error X) of 5%
- Define and print a phase flip (Pauli error Z) of 10%

Hints

`pauli_error` takes a list `[(gatename, p), ('I', 1 - p)]` as input, where p is the probability, represented as a float between 0 and 1

Exercise 1.3 – Step 1: Pauli errors

```
# 5% chance to perform a bit flip:
p = 0.05
bit_flip = pauli_error([('X', p), ('I', 1 - p)])
print(bit_flip)
# 10% chance of performing a phase flip::
p = 0.1
phase_flip = pauli_error([('Z', p), ('I', 1 - p)])
print(phase_flip)
# Expected output:
#QuantumError on 1 qubits. Noise circuits:
#  P(0) = 0.05, Circuit =
#
#q: ┤ X ├
#
#  P(1) = 0.95, Circuit =
#
#q: ┤ I ├
#
#QuantumError on 1 qubits. Noise circuits:
#  P(0) = 0.1, Circuit =
#
#q: ┤ Z ├
#
#  P(1) = 0.9, Circuit =
#
#q: ┤ I ├
#
```

Bit flip, with two possibilities:

- 95% chances of applying the identity, and
- 5% chances of applying the X gate on qubit q

Phase flip, with two possibilities:

- 90% chances of applying the identity, and
- 10% chances of applying the Z gate on qubit q

Exercise 1.3 – Step 2: Composing errors

Quantum errors can be combined, for instance by applying the bit flip then the phase flip in sequence to the same qubit, using **composition**

Instructions

- Define and print a `composed_error`, performing the bit flip, then the phase flip

Hints

An error has a `.compose` method, that takes as input another error

Exercise 1.3 – Step 2: Composing errors

```
composed_error = bit_flip.compose(phase_flip)
print(composed_error)
# Expected output:
#QuantumError on 1 qubits. Noise circuits:
#  P(0) = 0.005000000000000001, Circuit =
#
#q: ┤ X ├┤ Z ├
#
#  P(1) = 0.045000000000000005, Circuit =
#
#q: ┤ X ├┤ I ├
#
#  P(2) = 0.095, Circuit =
#
#q: ┤ I ├┤ Z ├
#
#  P(3) = 0.855, Circuit =
#
#q: ┤ I ├┤ I ├
#
```

Bit flip (gate X) error is applied, then phase flip (gate Z) error

There are 4 possibilities:

- Both flips XZ (0.5%),
- Bit flip alone XI (4.5%),
- Phase flip alone IZ (9.5%), and
- No flip II (85.5%)

Exercise 1.3 – Step 3: Errors on multiple qubits

Alternatively, one error can be applied to a qubit, and the other error to another qubit

This is implemented using a **tensor product**

Instructions

- Define and print a `tensor_error`, performing the bit flip on the first qubit and the phase flip on the second qubit

Hints

An error has a `.tensor` method, that takes as input another error

Exercise 1.3 – Step 3: Errors on multiple qubits

```
tensor_error = phase_flip.tensor(bit_flip)
print(tensor_error)
#QuantumError on 2 qubits. Noise circuits:
#  P(0) = 0.005000000000000001, Circuit =
#
#q_0: ┤ X ├
#     ├───┤
#q_1: ┤ Z ├
#     └───┘
#  P(1) = 0.095, Circuit =
#
#q_0: ┤ I ├
#     ├───┤
#q_1: ┤ Z ├
#     └───┘
#  P(2) = 0.045000000000000005, Circuit =
#
#q_0: ┤ X ├
#     ├───┤
#q_1: ┤ I ├
#     └───┘
#  P(3) = 0.855, Circuit =
#
#q_0: ┤ I ├
#     ├───┤
#q_1: ┤ I ├
#     └───┘
```

Bit flip (gate X) error is applied on qubit q_0, while phase flip (gate Z) error is applied to qubit q_1 in parallel

Again, there are 4 possibilities:

- Both flips XZ (0.5%),
- Phase flip alone IZ (9.5%),
- Bit flip alone XI (4.5%), and
- No flip II (85.5%)

Exercise 1.3 – Step 4: Readout errors

Readout errors are common errors occuring during measurement: a given output state is expected, but there is a small chance to record another value instead

```
from qiskit_aer.noise import ReadoutError
```

This error is expressed in terms of probabilities P(observed given expected)

Instructions

- Define and print a `readout_error` of
  - 5% to observe 1 while 0 was expected
  - 10% to observe 0 while 1 was expected

Hints

`ReadoutError` takes as input a list `[P0, P1]`, where:

- `P0` is a list containing
  - the probability of 0 given 0 and
  - the probability of 1 given 0
- `P1` is a list containing
  - the probability of 0 given 1 and
  - the probability of 1 given 1

Exercise 1.3 – Step 4: Readout errors

```
from qiskit_aer.noise import ReadoutError
# 5% chance of readout error for an expected |0>
p1given0 = 0.05# 1 is recorded while 0 was expected
p0given0 = 1 - p1given0# 0 is recorded while 0 was expected
P0 = [p0given0, p1given0]# pair for an expected |0> output state
# 10% chance of readout error for an expected |1>
p0given1 = 0.1# 0 is recorded while 1 was expected
p1given1 = 1 - p0given1# 1 is recorded while 1 was expected
P1 = [p0given1, p1given1]# pair for an expected |1> output state
# We can now define out readout error
readout_error = ReadoutError([P0, P1])
print(readout_error)
# Expected output:
#ReadoutError on 1 qubits. Assignment probabilities:
# P(j|0) =  [0.95 0.05]
# P(j|1) =  [0.1 0.9]
```

The first line is our `P0` for an expected 0 and the second line is our `P1` for an expected 1

Exercise 1.3 – Step 5: Depolarizing errors

Depolarizing errors represent cases when a qubit's state may become completely mixed, thus losing all quantum information. A depolarizing error randomly applies the identity or one of the Pauli error on the considered qubit(s).

```
from qiskit_aer.noise import depolarizing_error
```

The **parameter** associated with a depolarizing error must be between 0 and 1, but it is not a percentage of error

- if parameter = 0, then there is no error, i.e., it always applies the identity
- if parameter = 1, then it can be the identity (25%), an X-error (25%), a Y-error (25%), or a Z-error (25%)

Instructions

- Define and print a 1-qubit depolarizing error `error_unary` of 10%.

Hints

`depolarizing_error` takes as inputs the parameter and the number of qubits.

Exercise 1.3 – Step 5: Depolarizing errors

```
parameter = 0.1
nb_qubits = 1
error_unary = depolarizing_error(parameter, nb_qubits)
print(error_unary)
#QuantumError on 1 qubits. Noise circuits:
#  P(0) = 0.925, Circuit =
#
#q: ┤ I ├
#
#  P(1) = 0.025, Circuit =
#
#q: ┤ X ├
#
#  P(2) = 0.025, Circuit =
#
#q: ┤ Y ├
#
#  P(3) = 0.025, Circuit =
#
#q: ┤ Z ├
#
```

There are 4 possibilities:

- No error (92.5%),
- Pauli X error (2.5%),
- Pauli Y error (2.5%), and
- Pauli Z error (2.5%)

Exercise 1.3 – Step 6: Noise model (unary gates)

After this small tour, let's use our last (depolarizing) error to build noise models (other errors would work as well)

```
from qiskit_aer.noise import NoiseModel
```

Instructions

- Define and print a noise model containing `error_unary` for the gates 'rz' and 'sx' and qubit 0
- Define and print a noise model containing `error_unary` for the gates 'rz' and 'sx' and for all qubits

Hints

`NoiseModel()` generates an instance of noise model

A noise model has an `.add_quantum_error` method taking as inputs an error, a list of noisy gate names, and a list of noisy qubits

It has also an `.add_all_qubit_quantum_error` method taking as inputs an error and a list of noisy gate names

Exercise 1.3 – Step 6: Noise model (unary gates)

```
noisy_gates = ['rz', 'sx']
noise_model = NoiseModel()
noise_model.add_quantum_error(error_unary, noisy_gates, [0])# noise for qubit 0 only
print(noise_model)
# Alternatively, you can also add the same error to each qubit:
noise_model = NoiseModel()
noise_model.add_all_qubit_quantum_error(error_unary, noisy_gates)# noise for all qubits
print(noise_model)
# Expected output:
#NoiseModel:
#  Basis gates: ['cx', 'id', 'rz', 'sx']
#  Instructions with noise: ['sx', 'rz']
#  Qubits with noise: [0]
#  Specific qubit errors: [('rz', (0,)), ('sx', (0,))]
#NoiseModel:
#  Basis gates: ['cx', 'id', 'rz', 'sx']
#  Instructions with noise: ['sx', 'rz']
#  All-qubits errors: ['rz', 'sx']
```

Exercise 1.3 – Step 7: Noise model (binary gates)

The previous noisy gates (RZ and SX) can be applied only to one qubit at a time, but the procedure is similar for binary gates (or gates using more qubits), for instance for the CX (also known as CNOT) gate.

Instructions

- Define a 2-qubit depolarizing error `error_binary` of 10%.
- Define and print a noise model containing `error_binary` for the CX and qubits 0 and 1
- Do the same, but for qubits 1 and 0

Hints

See the last two steps ;)

Exercise 1.3 – Step 7: Noise model (binary gates)

```
parameter = 0.1
nb_qubits = 2
error_binary = depolarizing_error(parameter, nb_qubits)
noisy_gates = ['cx']
noise_model = NoiseModel()
noise_model.add_quantum_error(error_binary, noisy_gates, [0, 1])
print(noise_model)
noise_model = NoiseModel()
noise_model.add_quantum_error(error_binary, noisy_gates, [1, 0])
print(noise_model)
# Expected output:
#NoiseModel:
#  Basis gates: ['cx', 'id', 'rz', 'sx']
#  Instructions with noise: ['cx']
#  Qubits with noise: [0, 1]
#  Specific qubit errors: [('cx', (0, 1))]
#NoiseModel:
#  Basis gates: ['cx', 'id', 'rz', 'sx']
#  Instructions with noise: ['cx']
#  Qubits with noise: [0, 1]
#  Specific qubit errors: [('cx', (1, 0))]
```

For more than one qubit, the qubit order matters, since errors may be different on [1, 0] than on [0, 1]

Exercise 1.3 – Step 8: Custom noise model

Of course, you can add multiple errors to obtain complex noise models

Instructions

- Define and print a `custom_noise_model` containing:
  - `composed_error` for RZ on all qubits
  - `tensor_error` for CX on all qubits
  - `error_unary` for SX on qubit 0
  - `error_binary` for CX on qubits 1 and 0

Hints

We defined these errors in the previous steps

Exercise 1.3 – Step 8: Custom noise model

```
custom_noise_model = NoiseModel()
custom_noise_model.add_all_qubit_quantum_error(composed_error, ['rz'])
custom_noise_model.add_all_qubit_quantum_error(tensor_error, ['cx'])
custom_noise_model.add_quantum_error(error_unary, ['sx'], [0])
custom_noise_model.add_quantum_error(error_binary, ['cx'], [1, 0])
print(custom_noise_model)
# Expected output:
#NoiseModel:
#  Basis gates: ['cx', 'id', 'rz', 'sx']
#  Instructions with noise: ['sx', 'cx', 'rz']
#  Qubits with noise: [0, 1]
#  All-qubits errors: ['rz', 'cx']
#  Specific qubit errors: [('sx', (0,)), ('cx', (1, 0))]
```

Note that `NoiseModel` does not support non-local qubit errors, for instance crosstalk errors
Hence the need for more sophisticated tools like pyGSTi (second part of the exercises)

Exercise 1.3 – Step 9: Running a circuit with the custom noise model (1)

Let’s wrap up all the concepts we have seen so far…

Instructions

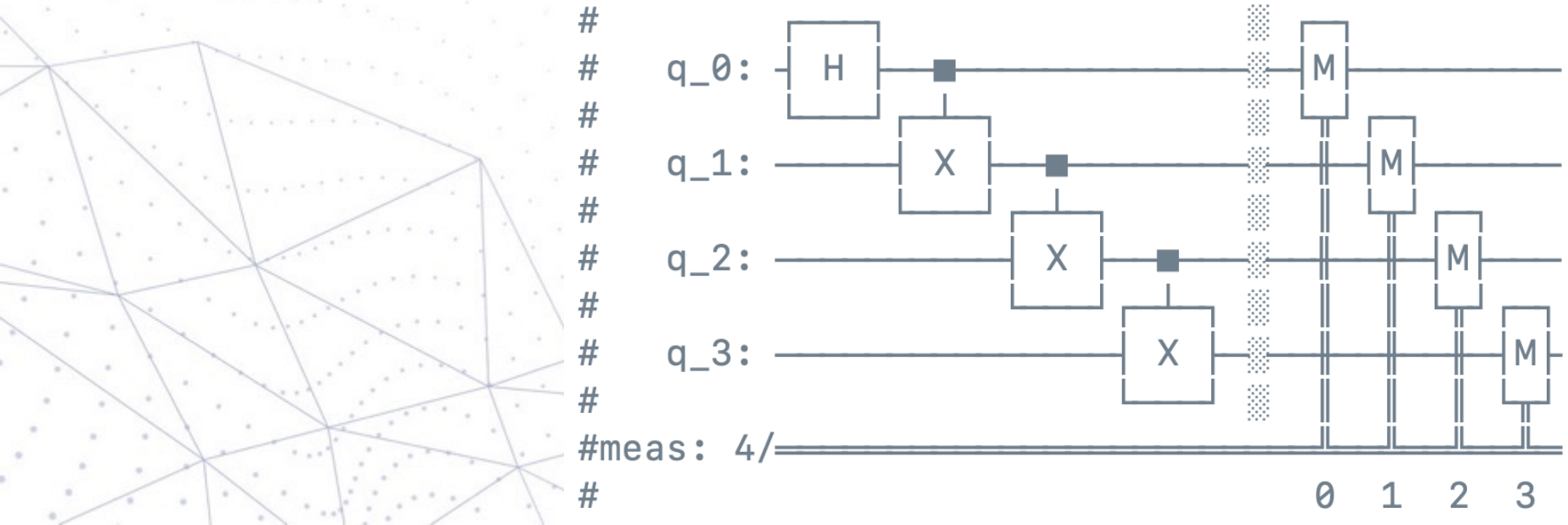
- Define and print a circuit for a 4-qubit GHZ state

Hints

To get an n-qubit GHZ state, repeat the X gate n-1 times between qubits i and i+1

Exercise 1.3 – Step 9: Running a circuit with the custom noise model (1)

```
from qiskit import QuantumCircuit
# The following circuit generates a n-qubit GHZ state
    1/sqrt(2)*(|0...0> + |1...1>)
circuit = QuantumCircuit(4)
circuit.h(0)# H gate on qubit 0
circuit.cx(0, 1)
circuit.cx(1, 2)
circuit.cx(2, 3)
circuit.measure_all()
print(circuit)
```



Exercise 1.3 – Step 9: Running a circuit with the custom noise model (1)

```python
from qiskit import QuantumCircuit
def get_GHZ_circuit(n_qubits):
    circuit = QuantumCircuit(n_qubits)
    circuit.h(0)
    for qubit in range(n_qubits - 1):
        circuit.cx(qubit, qubit + 1)
    circuit.measure_all()
    return circuit
circuit = get_GHZ_circuit(4)
print(circuit)
# Expected output:
#         ┌───┐                ░ ┌─┐         
#    q_0: ┤ H ├──■─────────────░─┤M├─────────
#         └───┘┌─┴─┐           ░ └╥┘┌─┐      
#    q_1: ─────┤ X ├──■────────░──╫─┤M├──────
#              └───┘┌─┴─┐      ░  ║ └╥┘┌─┐   
#    q_2: ──────────┤ X ├──■───░──╫──╫─┤M├───
#                   └───┘┌─┴─┐ ░  ║  ║ └╥┘┌─┐
#    q_3: ───────────────┤ X ├─░──╫──╫──╫─┤M├
#                        └───┘ ░  ║  ║  ║ └╥┘
#meas: 4/════════════════════════╩══╩══╩══╩═
#                                0  1  2  3 
```

Exercise 1.3 – Step 9: Running a circuit with the custom noise model (2)

Instructions

- For the ideal simulator
  - Transpile the circuit
  - Run it with the simulator
  - Get and print the outcome counts for 1024 shots
- For our custom noise model
  - Transpile the circuit
  - Run it with the simulator
  - Get and print the outcome counts for 1024 shots
  - Get and print the proportion of expected outputs

Hints

`AerSimulator` takes as input `noise_model = custom_noise_model`

Exercise 1.3 – Step 9: Running a circuit with the custom noise model (2)

```
from qiskit import transpile
# We use an auxiliary function to obtain the output counts for each simulator
def get_counts(circuit, simulator):
    circuit_transpiled = transpile(circuit, simulator)
    job = simulator.run(circuit_transpiled)# by default, shots = 1024
    result = job.result()
    counts = result.get_counts(0)
    return counts
# ideal simulation
from qiskit_aer import AerSimulator
ideal_simulator = AerSimulator()
ideal_counts = get_counts(circuit, ideal_simulator)
print(f"{ideal_counts = }")
# simulation with our custom noise model
noisy_simulator = AerSimulator(noise_model = custom_noise_model)
noisy_counts = get_counts(circuit, noisy_simulator)
proportion = round(100*(noisy_counts['0000'] + noisy_counts['1111'])/1024, 2)
print(f"{noisy_counts = }")
print(f"{proportion = }% of expected outputs")
#ideal_counts = {'0000': 520, '1111': 504}
#noisy_counts = {'0110': 1, '1010': 1, '1110': 26, '0100': 34, '0011': 2, '0001':
    24, '1011': 25, '0111': 1, '1111': 436, '1101': 24, '0010': 23, '0000': 427}
#proportion = 84.28% of expected outputs
```

Quantum software engineering

SnT

uni.lu
UNIVERSITY OF LUXEMBOURG

Quantum software engineering

As more powerful (with more qubits) quantum computers are built, they can support **more complex and various quantum programs** that will be deployed at larger scale.

These programs are **challenging to design, implement, and maintain**, as:

- They require expertise in various disciplines like physics, computer science, and mathematics
- They involve unintuitive concepts like superposition and entanglement
- They are implemented in low-level, error-prone programming languages

Hence the need for **quantum software engineering (QSE)** approaches

- To establish design principles and good practices in quantum computing
- Detect and fix quantum-related bugs
- Verify the quantum system satisfy desired functional / non-functional properties

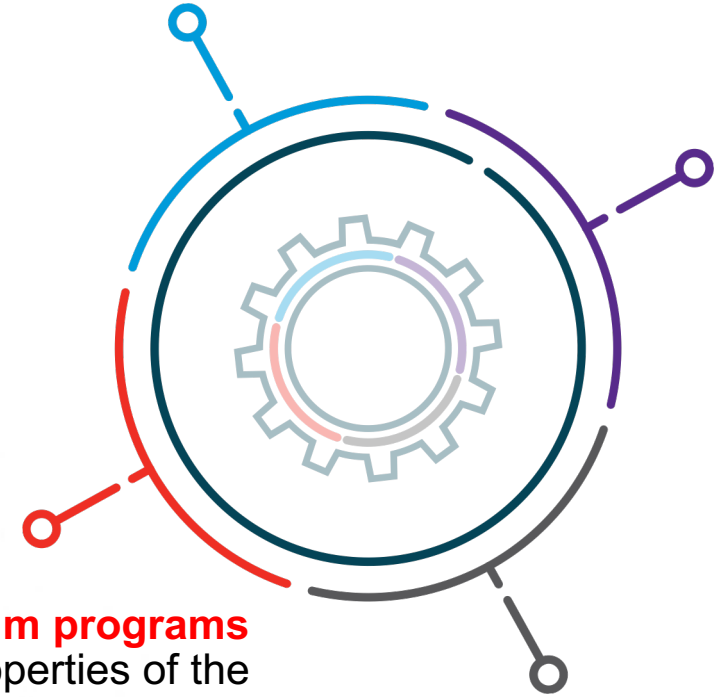
These QSE approaches need not only to adapt techniques from classical computing but to consider **specificities of quantum computing**

- For instance, traditional slicing techniques are usually not applicable in quantum computing because of entanglement

(An opinionated view of) Quantum software engineering

Quantum program analysis

White-box approaches for analyzing quantum circuits, detect and fix quantum-related bugs



Quantum program testing

Black-box approaches for generating and executing quantum inputs to discover quantum-related bugs

Specification of quantum programs

Approaches to check properties of the quantum system

Security of quantum programs

Approaches to identify and close adversarial information flows in quantum systems

Quantum bugs

A **quantum bug**

- Is a bug on a quantum algorithm (not on the language or implementation)
- That happens because of quantum considerations (as opposed to classical ones, like API usage)

Quantum bugs require domain-specific knowledge to fix

Quantum bugs

Example of bug

```
1  tq = QuantumRegister(3)
2  tc0 = ClassicalRegister(1)
3  tc1 = ClassicalRegister(1)
4  tc2 = ClassicalRegister(1)
5  teleport = QuantumCircuit(tq, tc0,tc1,tc2)
6
7  teleport.h(tq[1])
8  teleport.cx(tq[1], tq[2])
9  teleport.ry(np.pi/4,tq[0])
10 teleport.cx(tq[0], tq[1])
11 teleport.h(tq[0])
12 teleport.barrier()
13 teleport.measure(tq[0], tc0[0]) # measurement of quantum bit 0
14 teleport.measure(tq[1], tc1[0]) # measurement of quantum bit 1
15 teleport.cx(tq[1], tq[2]) # quantum operation on quantum bit 1 (now classical)
16 teleport.cz(tq[0], tq[2]) # quantum operation on quantum bit 0 (now classical)
17 teleport.measure(tq[2], tc2[0])
18
19 backend = Aer.get_backend("qasm_simulator")
20 job = execute(teleport, backend, shots=1, memory=True).result()
21 result = job.get_memory()[0]
```

The quantum measurement of qubits 0 and 1 is performed at Lines 13 and 14

Because these qubits are now classical, they cannot be used in quantum operations at Lines 15 and 16 anymore

Quantum bugs

Example of fix

```
1  tq = QuantumRegister(3)
2  tc0 = ClassicalRegister(1)
3  tc1 = ClassicalRegister(1)
4  tc2 = ClassicalRegister(1)
5  teleport = QuantumCircuit(tq, tc0,tc1,tc2)
6
7  teleport.h(tq[1])
8  teleport.cx(tq[1], tq[2])
9  teleport.ry(np.pi/4,tq[0])
10 teleport.cx(tq[0], tq[1])
11 teleport.h(tq[0])
12 teleport.barrier()
13 teleport.measure(tq[0], tc0[0]) # measurement of quantum bit 0
14 teleport.measure(tq[1], tc1[0]) # measurement of quantum bit 1
15 teleport.x(tq[2]).c_if(tc1,1) # classical control
16 teleport.z(tq[2]).c_if(tc0,1) # classical control
17 teleport.measure(tq[2], tc2[0])
18
19 backend = Aer.get_backend("qasm_simulator")
20 job = execute(teleport, backend, shots=1, memory=True).result()
21 result = job.get_memory()[0]
```

Bits obtained after quantum measurement can be used as classical control for quantum gates at Lines 15 and 16

Quantum bugs

Bug patterns

Study of 223 bugs [1]

- Most quantum bugs occur in components independent of the execution environment
  - Hence, they can be triggered in **simulators**
- Quantum bugs tend to manifest through **unexpected outputs**
  - Silent misbehavior, as opposed to crashes which are more common for classical bugs
- Most quantum bugs are API- or math-related, indicating a **lack of proficiency** by developers.

Several studies [1, 2, 3] investigated **quantum bug patterns**, like:

- API misuses
- Overlooked corner cases
- Number or order of qubits
- Incorrect initial state
- Usage of non-unitary matrices
- Incorrect usage of quantum gates
- Incorrect measurement handling
- Incorrect randomness handling

[1] M. Paltenghi and M. Pradel, and S.Ghosh, "Bugs in Quantum computing platforms: an empirical study" in Proc. ACM Program. Lang.
[2] P. Zhao, J. Zhao, and L. Ma, "Identifying Bug Patterns in Quantum Programs " in Proceedings of QSE”21, IEEE
[3] J. Luo, P. Zhao and Z. Miao, "A Comprehensive Study of Bug Fixes in Quantum Programs" in Proceedings of SANER”22, IEEE

Quantum bugs

Example of bug pattern

```
tq = QuantumRegister(3)
tc0 = ClassicalRegister(1)
tc1 = ClassicalRegister(1)
tc2 = ClassicalRegister(1)
teleport = QuantumCircuit(tq, tc0,tc1,tc2)

teleport.h(tq[1])
teleport.cx(tq[1], tq[2])
teleport.ry(np.pi/4,tq[0])
teleport.cx(tq[0], tq[1])
teleport.h(tq[0])
teleport.barrier()
teleport.measure(tq[0], tc0[0]) # measurement of quantum bit 0
teleport.measure(tq[1], tc1[0]) # measurement of quantum bit 1
teleport.cx(tq[1], tq[2]) # quantum operation on quantum bit 1 (now classical)
teleport.cz(tq[0], tq[2]) # quantum operation on quantum bit 0 (now classical)
teleport.measure(tq[2], tc2[0])

backend = Aer.get_backend("qasm_simulator")
job = execute(teleport, backend, shots=1, memory=True).result()
result = job.get_memory()[0]
```

Quantum measurement of quantum registers

+

Usage of quantum registers in quantum gates

Challenges

QSE approaches must face several challenges regarding quantum programs:

1. **Quantum states cannot be duplicated (no-cloning theorem)**
   - One cannot extract a value, then study it while the execution resumes
2. **Quantum states collapse when observed by quantum measurement**
   - It is challenging to investigate intermediate states to find quantum bugs
3. **Intermediate states have a large size**
   - Even in simulation (where they can be observed), this makes them hard to interpret
4. **Quantum state space is large**
   - It is challenging to identify the input able to trigger a quantum bug
5. **Quantum programs have probabilistic outcomes**
   - Quantum bugs may not manifest in the same way between two executions

Quantum program analysis

QSE approaches include **analysis techniques** that can be applied to quantum programs without executing them:

- **Slicing techniques**
  - For instance, at the gate level, to divide the circuit in small subcircuits
  - Remove qubits not involved in a particular slice

- **Static analyzer** (like Qchecker[1] or LintQ[2])
  - Based on bug patterns or quantum abstractions
  - Generate bug reports with bug location and description

[1] https://github.com/Z-928/QChecker
[2] https://github.com/sola-st/LintQ

Probabilistic cloning

A quantum state cannot be cloned using only unitary transformations (no-cloning theorem)
But probabilistic cloning = unitary transformations + quantum measurement

Probabilistic cloning of state $|\psi_1\rangle$

- Can **produce a clone** $|\psi_2\rangle$ which is not entangled with $|\psi_1\rangle$
  - Hence $|\psi_2\rangle$ can be measured
    - Without collapsing $|\psi_1\rangle$ in the rest of the computation
    - To obtain information about $|\psi_1\rangle$ (at the time of cloning)
- But it **can fail**
  - In that case, $|\psi_1\rangle$ is destroyed
  - $|\psi_1\rangle$ and $|\psi_2\rangle$ are entangled
- The probability of failure is smaller
  - When the expected state for $|\psi_1\rangle$ is known
  - And the actual state $|\psi_1\rangle$ is close to the expected state

Quantum program testing

Automatically generated test suites

In a 2022 study of 223 bugs [1], only 16 (7%) of them were detected using failing tests.
Hence a need for better testing techniques, like automatically generated test suites.

To tackle the **challenge of large quantum state spaces**, multiple approaches were applied to generate and execute input qubits able to trigger quantum bugs.

- QuanFuzz [2] mutates qubit inputs using common quantum gates
- QuBST [3] uses a genetic algorithm to determine a subset of the initial test suite with enough failing tests
- QuCAT [4] tests combinations of input qubits
- QuraTesT [5] generates small circuits to generate qubit inputs with diverse magnitude, phase, and entanglement

[1] M. Paltenghi and M. Pradel, and S.Ghosh, "Bugs in Quantum computing platforms: an empirical study" in *Proc. ACM Program. Lang.*
[2] J. Wang et al., “QuanFuzz: Fuzz Testing of Quantum Program”, *arXiv*
[3] X. Wang et al., “QuSBT: search-based testing of quantum programs”, *Proceedings of ICSE ‘22*, ACM, available: https://github.com/Simula-COMPLEX/qusbt-tool
[4] X. Wang et al., “QuCAT: A Combinatorial Testing Tool for Quantum Software”, *Proceedings of ASE ‘23*, IEEE, available: https://github.com/Simula-COMPLEX/qucat-tool
[5] J. Ye et al., “QuraTest: Integrating Quantum Specific Features in Quantum Program Testing”, *Proceedings of ASE ‘23*, IEEE, available: https://sites.google.com/view/quratest

Quantum program testing

Quantum-property-based tests

To tackle the **challenge of probabilistic outcome**, runs are repeated to determine the underlying output distribution.

A failure of a test is
- Either an unexpected output
- Or an incorrect output distribution

Also, statistical tests on the output can be used to determine if two qubits
- Are equal
- Or are entangled

To tackle the **challenge of large intermediate states**, a 2024 study [1] proposed to test qubit states after a change of basis (mixed Hadamard basis), depending on the expected state.
- This led to reduced output distributions, hence smaller test cases.
- Also, this led to reduced sample size, reducing execution time while improving fault detection.

[1] N. H. Oldfield et al., “Faster and Better Quantum Software Testing through Specification Reduction and Projective Measurements”, *TOSEM 2025*, ACM

Specifications

Formal verification

Similarly to tests on the final state, **specifications** can be written and tested on intermediate states.

A 2024 study [1] on formal verification proposed a model to check properties instead of executing the quantum program. These properties were written using the **linear temporal logic (LTL),** for instance:

$$\diamond(Prob > 0 \Rightarrow (C[0] = 0 \wedge C[1] = 0))$$

- Expresses that, at some point during the execution, if a quantum state was measured with a non-zero probability, the first two classical registers should contain the value 0

$$\diamond(Prob > 0 \Rightarrow (Q[2] = \text{init}(Q)[0] \wedge Q[0] \neq \text{init}(Q)[0]))$$

- Expresses that, at some point during the execution, if a quantum state was measured with a non-zero probability, the third quantum register should be in the same state as the first quantum register was at the start of the execution, and that the state of the first quantum register has changed during the execution.

Note: in these approaches, quantum systems are not executed.

[1] C. Minh Do and K. Ogata, ‘Symbolic model checking quantum circuits in Maude”, *PeerJ Computer Science 2024*

Specifications

Statistical assertions

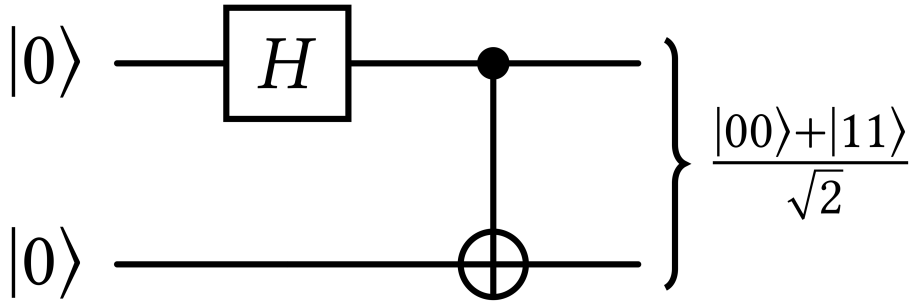
To tackle the **challenge of probabilistic outcome** of actual quantum executions, in a 2019 study [1], measurements are performed at breakpoints across multiple runs to approximate the underlying distribution.

This distribution is tested to determine if a **statistical assertion** is satisfied, for instance:

- *assert_classical(Q[0])*: After measurement, register 0 has (almost) always the same value
- *assert_superposition(Q[1])*: After measurement, register 1 can consistently take several values
- *assert_entangled(Q[0], Q[1])*: After measurement, values in registers 0 and 1 tend to be correlated

For instance:

- *assert_classical(Q[0])* fails
- *assert_superposition(Q[1])* passes
- *assert_entangled(Q[0], Q[1])* passes



But

- Each quantum measurement stops program execution
- Each assertion requires many runs to reach statistical significance

[1] Y. Huang, and M. Martonosi, “Statistical assertions for validating patterns and finding bugs in quantum programs”, *Proceedings of ISCA ‘19*, ACM

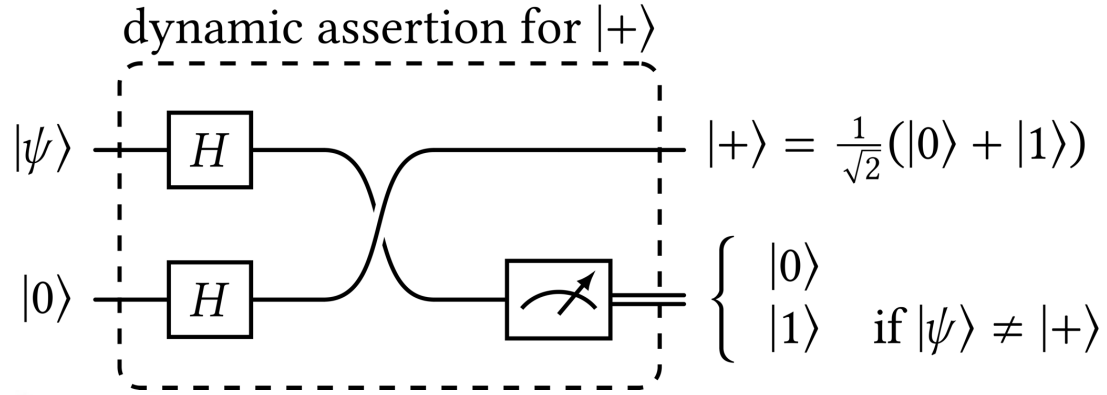
Specifications

Dynamic assertions

Two studies [1, 2] proposed to instrument the source program with circuits checking for **dynamic assertions**, covering:

- **Any state-comparison** *assert_equal(Q, X)*, where Q is a set of quantum registers and *X* is a quantum state
- Some entangled states, up to the amplitude

For instance, *assert_equal(Q[2],* $|+\rangle$*)* checks if the state in quantum register 2 is $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$



- An **ancillary qubit** (bottom) is added for the assertion
- The ancillary qubit is not entangled with $|\psi\rangle$, so can be **measured without impacting** the rest of the computation
- A measurement == $|1\rangle$ raises an **alarm**, but false negatives are possible
- In any case, the qubit $|\psi\rangle$ is **corrected** to be the desired state in the rest of the computation

[1] J. Liu, G. T. Byrd and H. Zhou, "Quantum Circuits for Dynamic Runtime Assertions in Quantum Computation", *Proc. of ASPLOS '20*, ACM
[2] J. Liu and H. Zhou, "Systematic Approaches for Precise and Approximate Quantum State Runtime Assertion", *Proc. of HPCA'21,* IEEE

Specifications

Projective assertions

A 2020 study [1] proposes a generalization of dynamic assertions with **projective predicates** *assert_projection(Q, X)*,
- where *Q* represents several qubits
- and *X* is a **projective subspace**, written in bra-ket notation

Which means that qubits Q are in a state which belongs to the projection, i.e., a subspace of the state space, X.

Intuitively, these projections can be seen as yes-no questions about where the quantum state of interest is, amongst all possible states, like:
- Is the state in qubit register Q[i] equal to $|+\rangle$?
- Is the state in registers Q[0] and Q[1] a combination of entangled states $|00\rangle$ and $|11\rangle$?
- Does the quantum state Q contain zeros in registers 1 and 2?

These predicate can be combined using NOT, AND, OR connectives (in quantum logic) to write more complex **projective assertions**.

But:
- They perform quantum measurements during execution, while they are usually performed at the end in current quantum computers.
- To our knowledge, the tool has not been implemented yet.

[1] G. Li et al. “Projection-based runtime assertions for testing and debugging Quantum programs”, *Proc. ACM Program. Lang.*

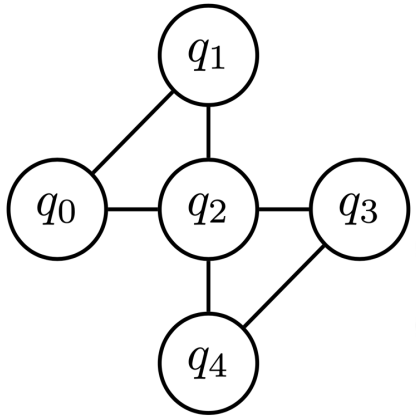
Quantum servers

- Every interaction between the quantum system and **the environment** is an uncontrolled measurement, which collapses the quantum state in an unexpected way, ending the computation.
- Current quantum computers require ultra-cold temperature, shielded environment, as well as complex wiring for controlling quantum gates, which makes quantum computing far from becoming a personal commodity.
    - Quantum computing must be delegated to **servers**.

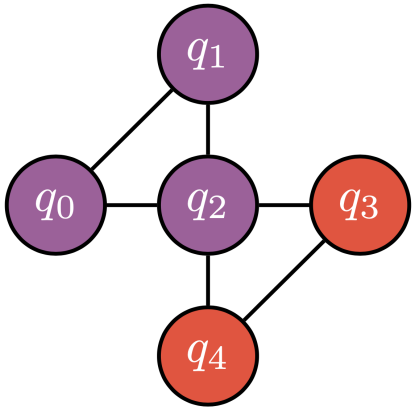
Security threats in quantum servers

- **Context**
  - To not waste quantum resources, multi-tenant **quantum servers** will be more and more common
  - Reminder: Quantum executions are noisy, **crosstalk** is the unintended interactions between qubits

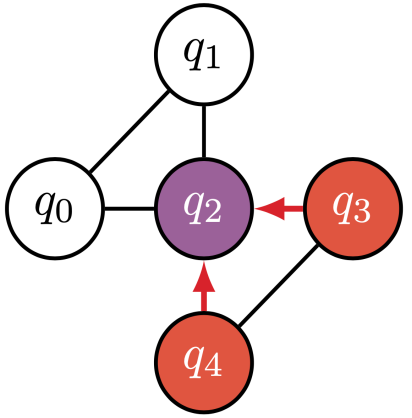
- **Problem**: Crosstalk-enabled attacks in quantum servers



Example of a quantum processor



Qubits are respectively allocated to the **attacker** and the **victim**



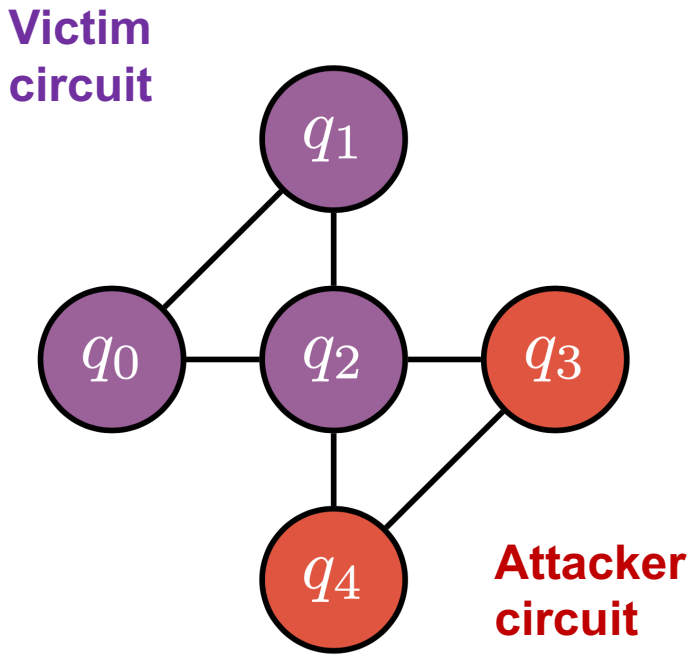
The **attacker**'s circuit execution tampers with the **victim**'s output [1]

[1] A.Ash-Saki, M.Alam, and S.Ghosh, "Analysis of crosstalk in NISQ devices and security implications in multi-programming regime," in *Proceedings of ISLPED '20*, ACM.

Crosstalk-enabled fault-injection attack

More precisely, the study proposed the following scenario:

1. The **attacker** analyzes crosstalk errors on the targeted platform
2. The **victim** requests qubits to the server
3. The **attacker** requests several copies of small quantum circuits to control the largest number possible of remaining qubits
4. The **victim** runs its circuit, e.g., with output in q_2
5. The **attacker** runs its circuit, e.g., q_3 and q_4, to induce crosstalk errors in the victim output



The study demonstrated that:

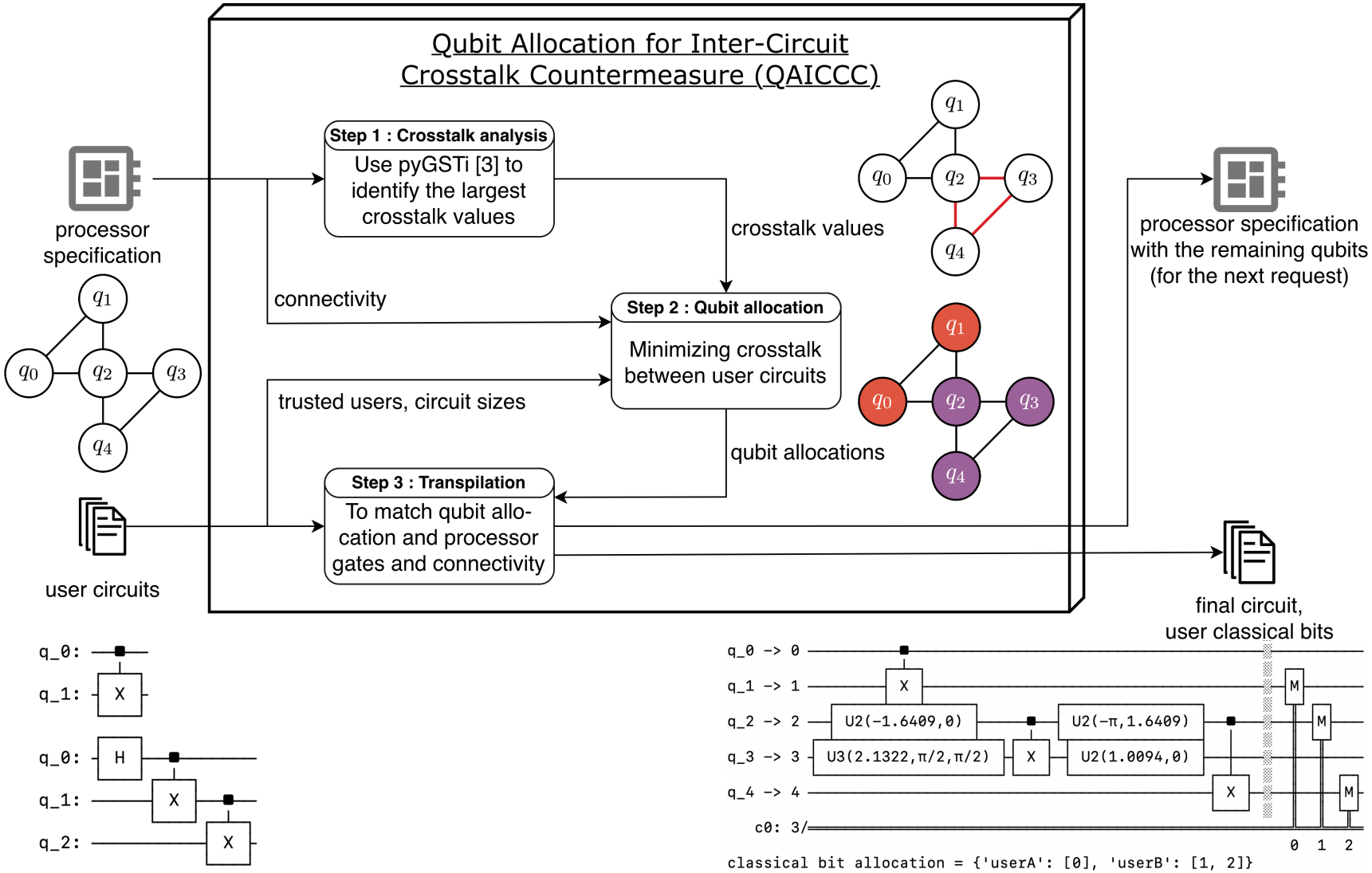
- Crosstalk can be the largest source of noise
- The magnitude of the output result can be reduced below the magnitude of other results, tampering the outcome after quantum measurement

The study suggested adding buffer qubits between circuits to prevent crosstalk-based attacks, but wasting qubit computation time is unrealistic.

Hence, other countermeasures have been proposed.

Securing quantum servers

- **Our approach**: QAICCC
  - Poster presented at QCE'25 [2]



[2] Y. Marquer and D. Bianculli, "A Piece of QAICCC: Towards a Countermeasure Against Crosstalk Attacks in Quantum Servers", in *Proceedings of QCE'25*, IEEE, Available: https://orbilu.uni.lu/handle/10993/64693

[3] pyGSTi, "A python implementation of gate set tomography" v0.9.13. [Online]. Available: https://github.com/sandialabs/pyGSTi

Noise reduction techniques

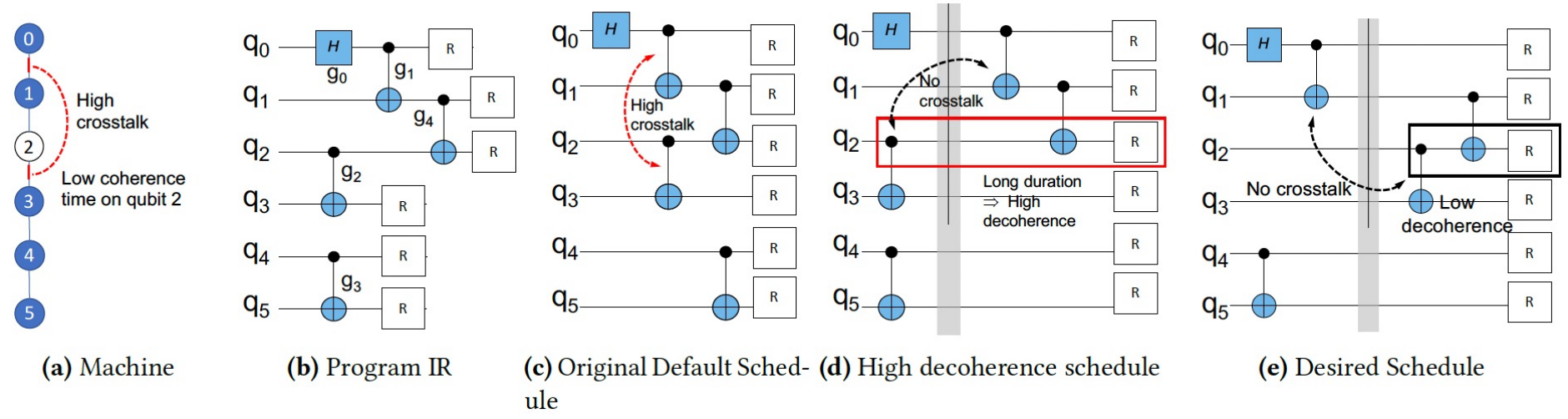
Gate scheduling

Reminder: Qbits in state |1⟩ may interact with the environment and lose energy, ending in state |0⟩, or may lose phase information. Both phenomena are called **decoherence** and are responsible for the short lifetime of qubits.

To minimize decoherence, quantum computing providers usually focus on reducing program duration.
Hence, more quantum gates run in parallel, which tends to increase crosstalk errors.

A 2020 study [1] proposed an approach:

- To reduce the time required to identify crosstalk
- To model the **trade-off between decoherence and crosstalk** and propose an optimal **schedule** for quantum gates



(a) Machine (b) Program IR (c) Original Default Schedule (d) High decoherence schedule (e) Desired Schedule

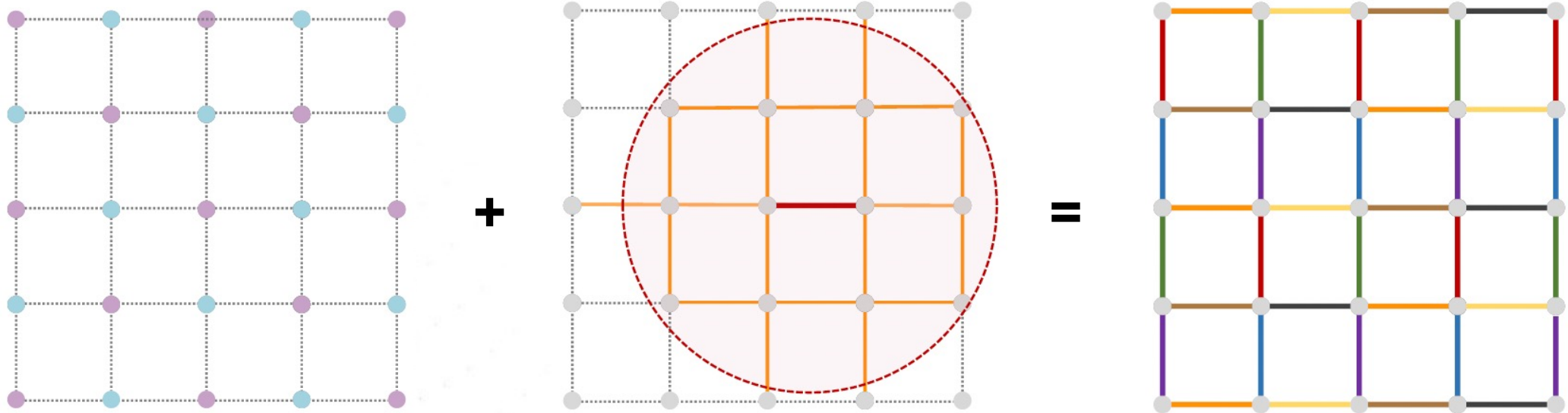
[1] P. Murali et al., "Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers", Proceedings of ASPLOS '20, ACM

Noise reduction techniques

Address frequency crowding

During execution, qubit pairs are tuned using specific frequencies to execute the desired quantum gates.
Crosstalk may occur when two pairs are tuned on the same (or close) frequency, hence are in **resonance**.

A 2020 study [1] proposes a graph-coloring approach,
- when frequencies can be controlled by the software,
- to **allocate different frequencies** to neighboring qubit pairs



Qubit connectivity | Neighboring pairs | Graph coloring

[1] Y. Ding et al., “Systematic Crosstalk Mitigation for Superconducting Qubits via Frequency-Aware Compilation”, Proceedings of MICRO’20, IEEE

Exercises
Part 2

SnT

UNIVERSITY OF LUXEMBOURG

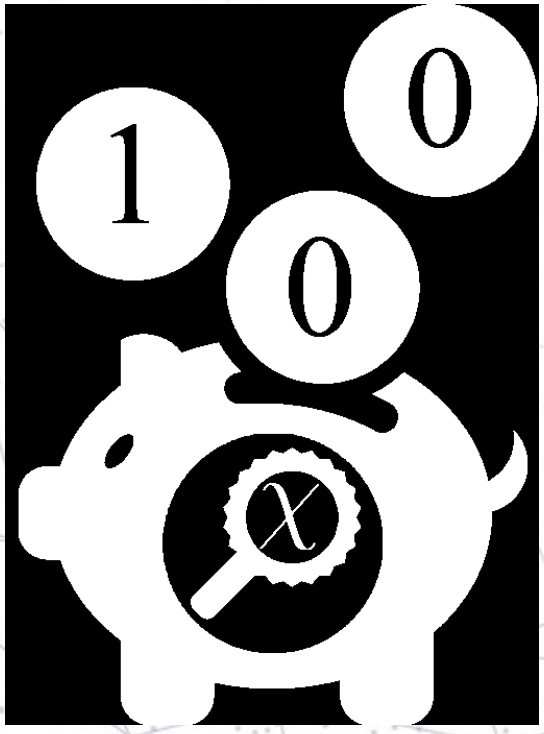
Setup

Instructions

1. `cd ..`
2. `mkdir 2_NoiseAnalysis`
3. `cd 2_NoiseAnalysis`

Part 2: Quantum noise analysis with pyGSTi

pyGSTi started as an implementation of the GST (gate set tomography) protocol.
Today it implements various noise-analysis protocols



Exercises

- pyGSTi circuits, processor specifications, models, and datasets
- Quantum noise analysis: the GST protocol

Exercise 2.1

In this exercise, as for Qiskit, we start with **circuit definition**.

Instructions

- Prepare an empty “exercise1.py” file
- For each step:
  - Write the code
  - `python exercise1.py`

Exercise 2.1 – Step 1: Circuit definition (layers)

Every pyGSTi gate name starts with ‘G’, e.g., ‘Gx’, ‘Gcnot’, ‘Gy’

Layers are list of tuples starting with the gate name, and then the qubits

Instructions

- Define a list of layers with
  - The X gate on qubit 0
  - The CNOT gate on qubits 0 and 1
  - The idle gate (how pyGSTi calls the identity gate)
  - The Y gate on qubit 3
- Use it to define and print a circuit

Hints

```
from pygsti.circuits import Circuit
```

`Circuit` takes as arguments layers and line_labels = labels, the list of qubits

The idle gate is empty

The syntax and even the gate names are different as in Qiskit, indicating a need for standards.

Exercise 2.1 – Step 1: Circuit definition (layers)

```
from pygsti.circuits import Circuit
layers = [# every pyGSTi gate name starts with 'G'
    ('Gx', 0),# X gate on qubit 0
    ('Gcnot', 0, 1),# CNOT gate
    (),# identity/idle gate
    ('Gy', 3)
]
labels = [0, 1, 2, 3]# the list of labels you use in your circuit
circuit = Circuit(layers, line_labels = labels)# instance of
    Circuit
print(circuit)
# Expected output:
#Qubit 0 ---|Gx|-|C1|-| |-|  |---
#Qubit 1 ---|  |-|T0|-| |-|  |---
#Qubit 2 ---|  |-|  |-| |-|  |---
#Qubit 3 ---|  |-|  |-| |-|Gy|---
```

Each column is a layer

Qubit labels are usually `list(range(n))` (this tends to avoid mapping issues), where n is the number of qubits

Exercise 2.1 – Step 2: Circuit definition (More complex layers)

To have multiple gates in the same layer, you can use a list of tuples instead of a tuple

Instructions

- Define a list of layers with
  - The X gate on qubit 0
  - The CNOT gate on qubits 0 and 1 and
    The Y gate on qubit 3 on the same layer
  - The idle gate
- Use it to define and print a circuit

Exercise 2.1 – Step 2: Circuit definition (More complex layers)

```
layers = [
    ('Gx', 0),
    [('Gcnot', 0, 1), ('Gy', 3)],# multiple gates at the same layer
    ()
]
circuit = Circuit(layers, line_labels = labels)
print(circuit)
# Expected output:
#Qubit 0 ---|Gx|-|C1|-| |---
#Qubit 1 ---|  |-|T0|-| |---
#Qubit 2 ---|  |-|  |-| |---
#Qubit 3 ---|  |-|Gy|-| |---
```

In this second circuit example, Gy is performed at the same layer as the Gcnot gate

Exercise 2.1 – Step 3: Circuit definition (other layer types)

On top of operation layers, you may also define **preparation layers** and **measurement layers**

- 'rho' indicates an n-qubit state preparation and
- 'Mz' indicates a measurement along the Z axis of the Bloch sphere

Instructions

- Define a list of layers with
  - Preparation 'rho'
  - The Z gate on qubit 1
  - A SWAP gate on qubits 0 and 1, and a Y gate on qubit 2
  - Measurement 'Mz'
- Use it to define and print a circuit

Hints

```
from pygsti.circuits import Circuit
```

`Circuit` takes as arguments layers and line_labels = labels, the list of qubits

The idle gate is empty

Exercise 2.1 – Step 3: Circuit definition (other layer types)

```
circuit = Circuit(['rho', ('Gz', 1), [('Gswap', 0, 1), ('Gy', 2)],
    'Mz'], line_labels = [0, 1, 2])
print(circuit)
# Expected output:
#Qubit 0 ---|rho|-|   |-|Gswap:0:1|-|Mz|---
#Qubit 1 ---|rho|-|Gz|-|Gswap:0:1|-|Mz|---
#Qubit 2 ---|rho|-|   |-|    Gy    |-|Mz|---
```

The first layer is the preparation layer, and the last is the measurement layer

Exercise 2.1 – Step 4: Circuit translation

If you need to run pyGSTi circuits (containing only unary gates) on IBM QPUs, you need to translate them from PyGSTi notation to OpenQASM (v2 and not v3, unfortunately)

```
from qiskit import qasm2
```

Instructions

- Define and print a circuit with
  - Gh gate on qubit 0
  - Gxpi2 gate on qubit 1
- Translate this circuit to an OpenQASM circuit, then print it

Hints

A pyGSTi has a method `.convert_to_openqasm`, with options `num_qubits` and `standard_gates_version = 'x-sx-rz'`, which converts the circuit to a string

The `qasm2.loads` function can convert a string to a Qiskit circuit, with option `custom_instructions = qasm2.LEGACY_CUSTOM_INSTRUCTIONS`

Exercise 2.1 – Step 4: Circuit translation

```
from qiskit import qasm2
pyGSTi_circuit = Circuit([('Gh', 0), ('Gxpi2', 1)],  line_labels = [0, 1])
circuit_string = pyGSTi_circuit.convert_to_openqasm(num_qubits = 2,
    standard_gates_version = 'x-sx-rz')# all gates are converted into X, SX,
    or RZ gates (the basis gates) that are supported by IBM QPUs
qasm2_circuit = qasm2.loads(circuit_string, custom_instructions =
    qasm2.LEGACY_CUSTOM_INSTRUCTIONS)# the custom_instructions parameter is
    necessary to fix a bug where the gate names are not correctly translated
print(pyGSTi_circuit)
print(qasm2_circuit)
# Expected output:
#Qubit 0 ---|Gh|-|     |---
#Qubit 1 ---|  |-|Gxpi2|---
#
#
# q_0: ─┤ Rz(π/2) ├─┤ √X ├─┤ Rz(π/2) ├─┤ Delay(0[dt]) ├─┤M├──
#
# q_1: ─┤ Delay(0[dt]) ├──────────────────┤ √X ├──────┤M├──
#
#cr: 2/═══════════════════════════════════════════════╩═╩═
#                                                    0 1
```

X, SX, and RZ are **basis gates**, gates that (with conditional gates) can simulate any common gate

Exercise 2.2

In this exercise, we introduce pyGSTi **processor specifications** and **models**.

Instructions

- Prepare an empty “exercise2.py” file
- For each step:
  - Write the code
  - `python exercise2.py`

Exercise 2.2 – Step 1: Processor specification (simple case)

A **processor specification** describes the number of qubits, the available gates, and the connectivity of the QPU

```
from pygsti.processors import QubitProcessorSpec
```

The connectivity is named "geometry" in pyGSTi, e.g., 'line' indicates that three qubits are connected as: 0 - 1 - 2

Instructions

- Specify a QPU with
  - 3 qubits
  - Gates X(π/2), Y(π/2), and CNOT
  - A linear geometry
- Print the qubit labels and the qubits were the X(π/2) and CNOT gates can be applied

Hints

`QubitProcessorSpec` takes as arguments `num_qubits`, `gate_names`, and `geometry`

A processor specification has an attribute `.qubit_labels`

A processor specification has a method `.resolved_availability` to check where a gate can be applied

Exercise 2.2 – Step 1: Processor specification (simple case)

```
from pygsti.processors import QubitProcessorSpec
procSpec1 = QubitProcessorSpec(num_qubits = 3, gate_names
    = ['Gxpi2', 'Gypi2', 'Gcnot'], geometry = "line")
print(f"Qubit labels are: {procSpec1.qubit_labels}")
print(f"X(pi/2) gates can be applied on qubits:
    {procSpec1.resolved_availability('Gxpi2')}")
print(f"CNOT gates can be applied on pairs of qubits:
    {procSpec1.resolved_availability('Gcnot')}")
# Expected output:
#Qubit labels are: (0, 1, 2)
#X(pi/2) gates can be applied on qubits: ((0,), (1,), (2,))
#CNOT gates can be applied on pairs of qubits: ((0, 1),
    (1, 0), (1, 2), (2, 1))
```

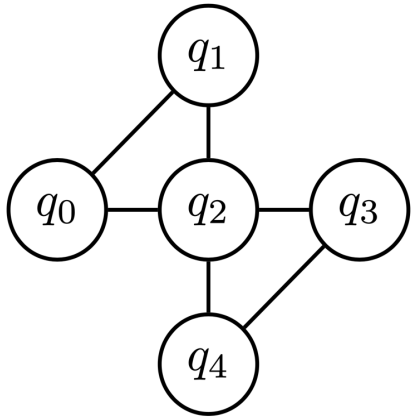
pyGSTi usually assumes the qubit labels are tuple(range(num_qubits))

Exercise 2.2 – Step 2: Processor specification (more complex connectivity)

One can also explicitly define the connectivity of the GPU using a graph

```
from pygsti.baseobjs.qubitgraph import QubitGraph
```

In this exercise we use again the Yorktown example (see on the right)



Instructions

- Specify the Yorktown QPU with gates CNOT, I, X(π/2), and ZR
- Print the qubit labels and the qubits were the X(π/2) and CNOT gates can be applied

Hints

`QubitGraph` takes as arguments `qubit_labels`, `initial_edges`, and `directed = False`

Exercise 2.2 – Step 2: Processor specification (more complex connectivity)

```
n = 5
connectivity = [
    [0, 1],
    [0, 2],
    [1, 2],
    [2, 3],
    [2, 4],
    [3, 4]
]
gates = [
    "Gcnot",
    "Gi",
    "Gxpi2",
    "Gzr"
]
graph = QubitGraph(qubit_labels = range(n), initial_edges = connectivity, directed = False)
procSpec2 = QubitProcessorSpec(num_qubits = n, gate_names = gates, geometry = graph)
print(f"Qubit labels are: {procSpec2.qubit_labels}")
print(f"X(pi/2) gates can be applied on qubits: {procSpec2.resolved_availability('Gxpi2')}")
print(f"CNOT gates can be applied on pairs of qubits: {procSpec2.resolved_availability('Gcnot')}")
# Expected output:
#Qubit labels are: (0, 1, 2, 3, 4)
#X(pi/2) gates can be applied on qubits: ((0,), (1,), (2,), (3,), (4,))
#CNOT gates can be applied on pairs of qubits: ((0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1),
    (2, 3), (2, 4), (3, 2), (3, 4), (4, 2), (4, 3))
```

Exercise 2.2 – Step 3: Explicit models

Models contain information about quantum gates, as well as state preparation and measurement (SPAM)

A model can be used to predict the outcome probability of quantum circuits

There are two types of models: **explicit** and **implicit models**. Explicit models are optimized for 1 or 2 qubits, while implicit models are optimized for 3+ qubits.

Let's start with the explicit models

```
from pygsti.models import create_explicit_model
```

Instructions

- Specify a QPU with
  - 2 qubits
  - Gates X(π/2), Y(π/2), and CNOT
  - A linear geometry
- Create an explicit model using the processor specification
- Print its preparations, measurements, and operations

Hints

An explicit model has attributes `.preps`, `.povms`, and `.operations`

Exercise 2.2 – Step 3: Explicit models

```
from pygsti.models import create_explicit_model
procSpec = QubitProcessorSpec(num_qubits = 2, gate_names = ['Gxpi2', 'Gypi2', 'Gcnot'],
    geometry = "line")
explicit_model = create_explicit_model(procSpec)
print("Preparations:", ', '.join(map(str,explicit_model.preps.keys())))
print("Measurements:", ', '.join(map(str,explicit_model.povms.keys())))
print("Operations:",   ', '.join(map(str,explicit_model.operations.keys())))
# Expected output:
#Preparations: rho0
#Measurements: Mdefault
#Operations: Gxpi2:0, Gxpi2:1, Gypi2:0, Gypi2:1, Gcnot:0:1, Gcnot:1:0
```

rho0 is the default preparation, where all the qubits are in the zero state

Mdefault is the default measurement, in the Z axis of the Bloch sphere

Exercise 2.2 – Step 4: Implicit models

Similarly, one can define implicit models using `create_crosstalk_free_model` or the (more complex) `create_cloud_crosstalk_model` constructs

```
from pygsti.models import create_crosstalk_free_model
```

Instructions

- Create an explicit (crosstalk-free) model using the processor specification
- Print its preparations, measurements, and operations

Hints

An explicit model has attributes `.prep_blks`, `.povm_blks`, and `.operation_blks`

Exercise 2.2 – Step 4: Implicit models

```
# ---- Step 4: Implicit models  ----
# Similarly, one can define implicit models using create_crosstalk_free_model or the
    (more complex) create_cloud_crosstalk_model constructs
from pygsti.models import create_crosstalk_free_model
implicit_model = create_crosstalk_free_model(procSpec)
print("Preparation building blocks:")
for label, block in implicit_model.prep_blks.items():
    print("  " + label, ":", ', '.join(map(str ,block.keys())))
print("Measurement building blocks:")
for label, block in implicit_model.povm_blks.items():
    print("  " + label, ":", ', '.join(map(str, block.keys())))
print("Operation building blocks:")
for label, block in implicit_model.operation_blks.items():
    print("  " + label, ":", ', '.join(map(str, block.keys())))
# Expected output:
#Preparation building blocks:
#  layers : rho0
#Measurement building blocks:
#  layers : Mdefault
#Operation building blocks:
#  gates : Gxpi2, Gypi2, Gcnot
#  layers : Gxpi2:0, Gxpi2:1, Gypi2:0, Gypi2:1, Gcnot:0:1, Gcnot:1:0
```

Again, rho0 is the default preparation, where all the qubits are in the zero state

Mdefault is the default measurement, in the Z axis of the Bloch sphere

Exercise 2.2 – Step 5: Probability distribution

Models can be used to predict the probability distribution of a circuit, using the `.probabilities` method

Instructions

- Create a circuit with gates
  - X(π/2) on qubit 0
  - CNOT on qubits 0 and 1
  - Y(π/2) on qubit 1
- Use both the explicit and implicit models to determine the probability of output 00

Hints

`.probabilities` takes the circuit as input

Exercise 2.2 – Step 5: Probability distribution

```
from pygsti.circuits import Circuit
circuit = Circuit([('Gxpi2', 0), ('Gcnot', 0, 1), ('Gypi2', 1)], line_labels = [0, 1])
print(circuit)
proba = explicit_model.probabilities(circuit)
print(f"Explicit model: probability of outcome 00 = {proba['00']}")
proba = implicit_model.probabilities(circuit)
print(f"Implicit model: probability of outcome 00 = {proba['00']}")
# Expected output:
#Qubit 0 ---|Gxpi2|-|C1|-|     |---
#Qubit 1 ---|     |-|T0|-|Gypi2|---
#Explicit model: probability of outcome 00 = 0.25
#Implicit model: probability of outcome 00 = 0.2499999999999997
```

Both agree, event if the implicit model would be better used for larger (with more qubits) QPUs

Exercise 2.3

In this exercise, we introduce pyGSTi **datasets**.

Instructions

- Prepare an empty “exercise3.py” file
- For each step:
  - Write the code
  - `python exercise3.py`

Exercise 2.3 – Step 1: Dataset from a file

A dataSet object contains mutiple rows mapping circuits to outcome counts, for instance:

```
dataset_txt = \
"""## Columns = 00 count, 01 count, 10 count, 11 count
{}                      100   0    0    0
Gxpi2:0                  55   5   40    0
Gxpi2:0Gypi2:1           20  27   23   30
Gxpi2:0^4                85   3   10    2
Gxpi2:0Gcnot:0:1         45   1    4   50
[Gxpi2:0Gxpi2:1]Gypi2:0 25   32   17   26
"""
```

In this example, for instance, circuit "**Gxpi2:0**" (applying gate Gxpi2 on qubit 0) has output **'00' in 55 of runs** (amongst 100)

{} is the identity/idle gate

Instructions

- Write it in a "dataset.txt" file

Exercise 2.3 – Step 1: Dataset from a file

```
from pygsti.io import read_dataset
```

Instructions

- Read the dataset from the txt file, then print it
- Make the dataset sparse by removing zeros, then print it

Hints

A dataset has a `.drop_zero_counts()` method

Exercise 2.3 – Step 1: Dataset from a file

```
from pygsti.io import read_dataset
from os import remove
filePath = "dataset.txt"
with open(filePath, "w") as file:
    file.write(dataset_txt)
# pyGSTi handles reading a dataset from a file
dataset = read_dataset(filePath)
remove(filePath)# remove the file after usage
print(dataset)
# Also, a dataset can be made sparse, i.e., 0-count occurrences can be dropped:
sparse_dataset = dataset.drop_zero_counts()
print(sparse_dataset)
# Expected output:
#{}  :  {('00',): 100.0, ('01',): 0.0, ('10',): 0.0, ('11',): 0.0}
#Gxpi2:0  :  {('00',): 55.0, ('01',): 5.0, ('10',): 40.0, ('11',): 0.0}
#Gxpi2:0Gypi2:1  :  {('00',): 20.0, ('01',): 27.0, ('10',): 23.0, ('11',): 30.0}
#Gxpi2:0^4  :  {('00',): 85.0, ('01',): 3.0, ('10',): 10.0, ('11',): 2.0}
#Gxpi2:0Gcnot:0:1  :  {('00',): 45.0, ('01',): 1.0, ('10',): 4.0, ('11',): 50.0}
#[Gxpi2:0Gxpi2:1]Gypi2:0  :  {('00',): 25.0, ('01',): 32.0, ('10',): 17.0, ('11',): 26.0}
#
#{}  :  {('00',): np.float32(100.0)}
#Gxpi2:0  :  {('00',): np.float32(55.0), ('01',): np.float32(5.0), ('10',): np.float32(40.0)}
#Gxpi2:0Gypi2:1  :  {('00',): 20.0, ('01',): 27.0, ('10',): 23.0, ('11',): 30.0}
#Gxpi2:0^4  :  {('00',): 85.0, ('01',): 3.0, ('10',): 10.0, ('11',): 2.0}
#Gxpi2:0Gcnot:0:1  :  {('00',): 45.0, ('01',): 1.0, ('10',): 4.0, ('11',): 50.0}
#[Gxpi2:0Gxpi2:1]Gypi2:0  :  {('00',): 25.0, ('01',): 32.0, ('10',): 17.0, ('11',): 26.0}
```

Exercise 2.3 – Step 2: Retrieve data from a dataset

```
from pygsti.io import read_dataset
```

Instructions

- Define a layer list corresponding to 'Gxpi2:0Gcnot:0:1' and print the corresponding dataset row
- Use the layer list to define a circuit and use it print the corresponding dataset row
- Use this row to get the output counts for output '00'
- Print all the outputs and counts of the row

Hints

A dataset accepts both `dataset[layers]` and `dataset[circuit]`

Exercise 2.3 – Step 2: Retrieve data from a dataset

```
from pygsti.circuits import Circuit
layers = [('Gxpi2', 0), ('Gcnot', 0, 1)]
print(f"row = {dataset[layers]} (though layers)")
circuit = Circuit(layers, line_labels = [0, 1])
print(circuit)
row = dataset[circuit]
print(f"row = {row} (through circuit)")
print(f"{row['00'] = }")
print("Iteration:")
for output, count in row.counts.items():
    print(output, count)
# Expected output:
#row = {('00',): 45.0, ('01',): 1.0, ('10',): 4.0, ('11',): 50.0} (though layers)
#Qubit 0 ---|Gxpi2|-|C1|---
#Qubit 1 ---|     |-|T0|---
#row = {('00',): 45.0, ('01',): 1.0, ('10',): 4.0, ('11',): 50.0} (through circuit)
#row['00'] = 45.0
#Iteration:
#('00',) 45.0
#('01',) 1.0
#('10',) 4.0
#('11',) 50.0
```

Note that a row does not have an `.items()` attribute, this is because a dataset is not simply a dictionary object, since it may contain timestamped data

Nevertheless, you can use the `.counts` attribute first to iterate over output counts

Exercise 2.3 – Step 3: Simulate a dataset

The `simulate_data` procedure calls `model.probabilities(circuit)` for each circuit in a given list, then samples from the corresponding output probability distribution to generate outcome counts

```
from pygsti.circuits import to_circuits
from pygsti.data import simulate_data
```

Instructions

- Specify a QPU with
  - 2 qubits
  - Gates X(π/2), Y(π/2), and CNOT
  - A linear geometry
- Use the processor specification to create an explicit model
- Use function `to_circuits` to obtain a list of circuits matching the labels of the example dataset
- Use the model and circuits to simulate then print a dataset for 100 samples, and a 'multinomial' sample error

Hints

`to_circuits` takes a list of tuples of layers, and `line_labels`

`simulate_data` takes as inputs a model, the circuits, `num_samples`, `sample_error`, and optionally a `seed`

Exercise 2.3 – Step 3: Simulate a dataset

```
from pygsti.processors import QubitProcessorSpec
from pygsti.models import create_explicit_model
from pygsti.circuits import to_circuits
from pygsti.data import simulate_data
procSpec = QubitProcessorSpec(num_qubits = 2, gate_names = ['Gxpi2', 'Gypi2', 'Gcnot'], geometry = "line")
model = create_explicit_model(procSpec)
circuit_descriptions = [
    (),
    (('Gxpi2', 0),),
    (('Gxpi2', 0), ('Gypi2', 1)),
    (('Gxpi2', 0),)*4,
    (('Gxpi2', 0), ('Gcnot', 0, 1)),
    (('Gxpi2', 0), ('Gxpi2', 1), ('Gxpi2', 0))
]
circuits = to_circuits(circuit_descriptions, line_labels = (0, 1))# translate circuit descriptions to actual circuits
simulated_dataset = simulate_data(model, circuits, num_samples = 100, sample_error = 'multinomial', seed = 8675309)
# useful options for the simulate_data function:
# - record_zero_counts = False: to generate a sparse dataset
# - sample_error = 'none': to generate simulated data following exactly the output distribution
print(simulated_dataset)
# Expected output:
#{}@(0,1)  :  {('00',): 100.0, ('01',): 0.0, ('10',): 0.0, ('11',): 0.0}
#Gxpi2:0@(0,1)  :  {('00',): 46.0, ('01',): 0.0, ('10',): 54.0, ('11',): 0.0}
#Gxpi2:0Gypi2:1@(0,1)  :  {('00',): 32.0, ('01',): 25.0, ('10',): 22.0, ('11',): 21.0}
#Gxpi2:0Gxpi2:0Gxpi2:0Gxpi2:0@(0,1)  :  {('00',): 100.0, ('01',): 0.0, ('10',): 0.0, ('11',): 0.0}
#Gxpi2:0Gcnot:0:1@(0,1)  :  {('00',): 42.0, ('01',): 0.0, ('10',): 0.0, ('11',): 58.0}
#Gxpi2:0Gxpi2:1Gxpi2:0@(0,1)  :  {('00',): 0.0, ('01',): 0.0, ('10',): 55.0, ('11',): 45.0}
```

Exercise 2.4 (bonus, more advanced exercise)

In this exercise, we introduce **quantum noise analysis**.

Instructions

- Prepare an empty “exercise4.py” file
- For each step:
  - Write the code
  - `python exercise4.py`

Exercise 2.4 – Step 1: GST circuits

GST (gate set tomography) is the main protocol of the pyGSTi framework, it builds an **error model** of the targeted QPU

- 1-qubit GST, as in this exercise, is fairly straightforward
- 2 qubit GQT is more resource-intensive
- 3-qubit (or more) GST is still an active research area

To test GST, we use a predefined 1-qubit model supporting the Idle, X(π/2) and Y(π/2) gates

```
from pygsti.modelpacks import smq1Q_XYI
```

Then, we generate circuits using this model

```
from pygsti.circuits import create_lsgst_circuits
```

Exercise 2.4 – Step 1: GST circuits

To give you an intuition, please recall our dataset example from Exercise 2.3:

```
"""## Columns = 00 count, 01 count, 10 count, 11 count
{}                        100   0   0   0
Gxpi2:0                    55   5  40   0
Gxpi2:0Gypi2:1             20  27  23  30
Gxpi2:0^4                  85   3  10   2
Gxpi2:0Gcnot:0:1           45   1   4  50
[Gxpi2:0Gxpi2:1]Gypi2:0 25  32  17  26
"""
```

Gxpi2:0 is a π/2 rotation around the X axis of the Bloch sphere of qubit 0, so repeating 4 times with Gxpi2:0^4 should be equivalent to the identity

Nevertheless, one can observe that:

- The **identity** does not change the initial '00' state, resulting in 100 shots with output '00',
- While **Gxpi2:0^4** leads to 85 shots with output '00': the other 15 outputs are due to the noise in the QPU

Repeating these rotations will increase the impact of noise in the output distribution, so one can build a **noise model**

Exercise 2.4 – Step 1: GST circuits

The GST protocol generates circuits called **fiducials** that are used to test various quantum errors. More precisely, these circuits have the form:

circuit = preparation_fiducial + repeated_germ + measurement_fiducial

- **Preparation and measurement fiducials** ensure a sufficiently diverse set of input states and measurements, i.e., which span the vector space of density matrices
- A **germ** sequence is repeated several times to amplify targeted gate errors: the more repetitions, the more the GST algorithm will be sensitive to these errors

These generated circuits have a **maximum lengths** (the larger the more precise, but also the more expensive)

Instructions

- From `smq1Q_XYI`, generate an ideal model, preparation and measurement fiducials, and germs
- Define the maximum lengths to [1, 2, 4, 8]
- Use these inputs with `create_lsgst_circuits` to generate GST circuits and print the first eight of them

Hints

`smq1Q_XYI` has the following methods: `.target_model()`, `.prep_fiducials()`, `.meas_fiducials()`, and `germs()`

Exercise 2.4 – Step 1: GST circuits

```
from pygsti.modelpacks import smq1Q_XYI
ideal_model = smq1Q_XYI.target_model()
from pygsti.circuits import create_lsgst_circuits
preparation_fiducials = smq1Q_XYI.prep_fiducials()
measurement_fiducials = smq1Q_XYI.meas_fiducials()
germs = smq1Q_XYI.germs()
maxLengths = [2**n for n in range(4)]# they are usually powers of 2
circuits = create_lsgst_circuits(ideal_model, preparation_fiducials, measurement_fiducials, germs, maxLengths)
for circuit in circuits[:8]:# there are a lot of circuits, so we only print the first ones
    print(circuit)
# Expected output:
#Qubit 0 -----
#Qubit 0 ---|Gxpi2|---
#Qubit 0 ---|Gypi2|---
#Qubit 0 ---|Gxpi2|-|Gxpi2|---
#Qubit 0 ---|Gxpi2|-|Gxpi2|-|Gxpi2|---
#Qubit 0 ---|Gypi2|-|Gypi2|-|Gypi2|---
#Qubit 0 ---|Gxpi2|-|Gypi2|---
#Qubit 0 ---|Gxpi2|-|Gxpi2|-|Gxpi2|-|Gxpi2|---
```

Note the first line corresponds to the identity and the last line corresponds to the Gxpi2^4 circuit we discussed earlier

Exercise 2.4 – Step 2: Generating the dataset

Normally, one would run these circuits on the QPU (physical device) of interest, then retrieve the dataset for noise analysis.

In the context of this exercise, we simply use a simulator.
As we did for Qiskit, we add a depolarizing error to the ideal model.

Instructions

- Depolarize the ideal model with `op_noise` = 0.01 and `spam_noise` = 0.001 to obtain a noisy model
- Use the noisy model and the GST circuits to simulate a dataset for 1000 samples and a binomial' sample error
- Print the first 8 rows

Hints

The ideal model has a method `.depolarize()`

Exercise 2.4 – Step 2: Generating the dataset

```
from pygsti.data import simulate_data
noisy_model = ideal_model.depolarize(op_noise = 0.01, spam_noise = 0.001)
dataset = simulate_data(noisy_model, circuits, num_samples = 1000, sample_error = "binomial", seed = 1234)
for circuit in circuits[:8]:
    print(dataset[circuit])
# Expected output:
#{('0',): 1000.0, ('1',): 0.0}
#{('0',): 485.0, ('1',): 515.0}
#{('0',): 509.0, ('1',): 491.0}
#{('0',): 8.0, ('1',): 992.0}
#{('0',): 487.0, ('1',): 513.0}
#{('0',): 466.0, ('1',): 534.0}
#{('0',): 500.0, ('1',): 500.0}
#{('0',): 978.0, ('1',): 22.0}
```

The last line corresponds to the **Gxpi2^4 circuit**: qubit 0 was measured '0' 978 times, compared to the 1000 '0's obtained with the **identity (idle) circuit** at the first line.

Exercise 2.4 – Step 3: run GST

We can now run the GST algorithm

```
from pygsti import run_stdpractice_gst
```

Instructions

- Run `run_stdpractice_gst` with the dataset, ideal model, preparation and measurement fiducials, germs, max lengths, `modes = ("full TP", "Target")` and `verbosity = 1` to get the results
- Print the results to investigate how they are structured
- Store the `model_estimate` in a variable

Hints

`results` has an attribute `.estimates['full TP']`, which has an attribute `.models['stdgaugeopt']`

Exercise 2.4 – Step 3: run GST

```
from pygsti import run_stdpractice_gst
results = run_stdpractice_gst(dataset, ideal_model, preparation_fiducials, measurement_fiducials,
    germs, maxLengths, modes = ("full TP", "Target"), verbosity = 1)
print(results)
model_estimate = results.estimates['full TP'].models['stdgaugeopt']
```

```
-------------------------------------------------------
---------- pyGSTi ModelEstimateResults Object ----------
-------------------------------------------------------

How to access my contents:

 .dataset    -- the DataSet used to generate these results

 .circuit_lists   -- a dict of Circuit lists w/keys:
 -----------------------------------------------------
  iteration
  prep fiducials
  meas fiducials
  germs
  final

 .estimates   -- a dictionary of Estimate objects:
 -----------------------------------------------------
  full TP
  Target
```

Exercise 2.4 – Step 4: Goodness of fit

Now that we obtained a noise model from the dataset, we want to know how well the model describes the dataset

This is usually done using the **chi-squared** and the **log-likelihood** log(L) metrics

Note that a log(L) value doesn't mean anything in isolation, but it can be compared to other log-likelihood values, for instance with log(L_max), the log-likelihood of a model that perfectly agrees with the dataset

Interestingly, with enough samples, chi-squared ≈ 2(log(L_max) - log(L)), which is denoted **2DeltaLogL** in pyGSTi and is used to quantify how well a model fits the dataset (the smaller, the better):

```
from pygsti.tools import two_delta_logl
```

Instructions

- Compute and print the 2DeltaLogL between
  - The ideal model
  - The noisy model
  - The model estimate
- And the dataset, respectively

Exercise 2.4 – Step 4: Goodness of fit

```
from pygsti.tools import two_delta_logl
TwoDeltaLogL_ideal = two_delta_logl(ideal_model, dataset)
TwoDeltaLogL_noisy = two_delta_logl(noisy_model, dataset)
TwoDeltaLogL_estimate = two_delta_logl(model_estimate, dataset)
print(f"2DeltaLogL(ideal,    data) = {TwoDeltaLogL_ideal}")
print(f"2DeltaLogL(noisy,    data) = {TwoDeltaLogL_noisy}")
print(f"2DeltaLogL(estimate, data) = {TwoDeltaLogL_estimate}")
# Expected output:
#2DeltaLogL(ideal,    data) = 125357.93910734144
#2DeltaLogL(noisy,    data) = 450.7549936014515
#2DeltaLogL(estimate, data) = 422.79090654201167
```

pyGSTi found an estimate model that fits the dataset very well, compared to the other models.

Interestingly, the estimate model fits the dataset better than the noisy model, even if the dataset was obtained from the noisy model, because the GST estimate agrees better with the finite noise sample.

Exercise 2.4 – Step 5: HTML report

pyGSTi can also generate reports in HTML format, which looks better

```
from pygsti.report import construct_standard_report
```

Instructions

- Use the results to generate a report with title "Report Example”
- Write the report in HTML format in the "Report” directory, and `auto_open = False, verbosity = 1` options

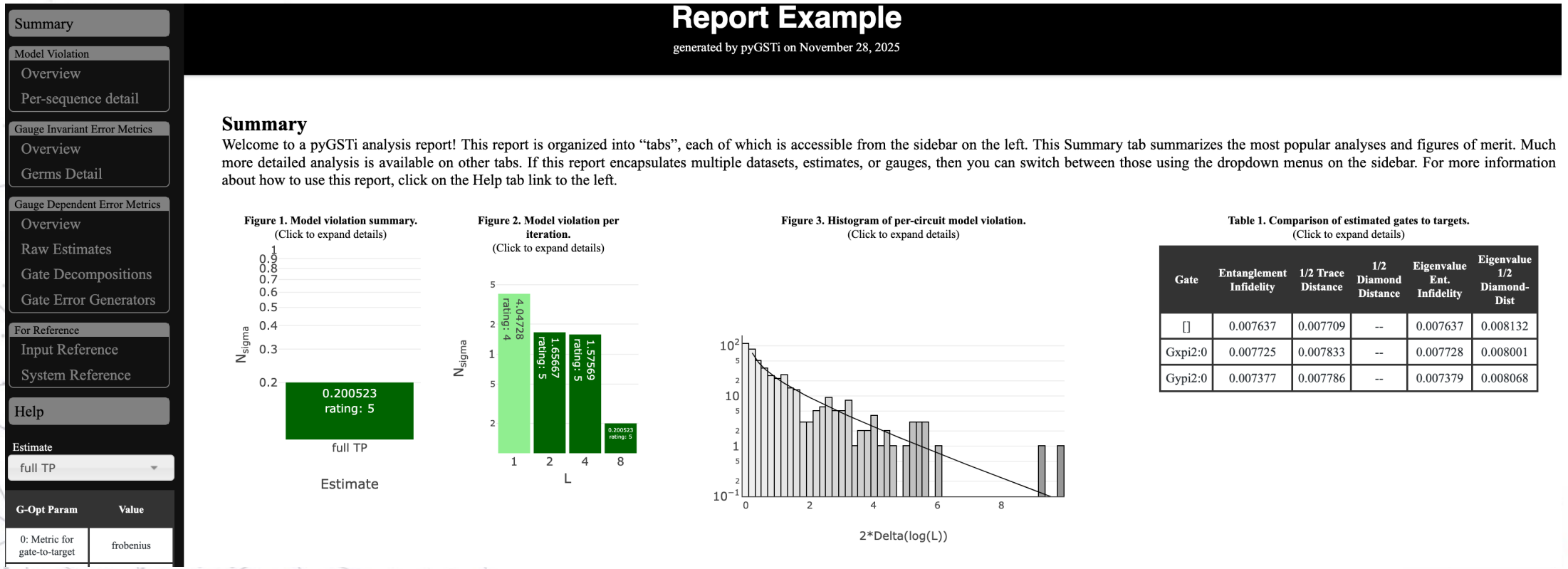
Hints

A report has a method `.write_html`

Exercise 2.4 – Step 5: HTML report

```
from pygsti.report import construct_standard_report
report = construct_standard_report(results, title = "Report Example")
directory = "Report"
report.write_html(directory, auto_open = False, verbosity = 1)
```

When ready, you can open with your favorite browser the **Report/main.html file** and take a look at the explanations and the various tabs



yoann.marquer@uni.lu
domenico.bianculli@uni.lu

SnT

uni.lu
UNIVERSITY OF LUXEMBOURG