Utiliser un ordinateur quantique (avec Qiskit)

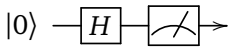
Le but est de programmer des circuits quantiques et de simuler les résultats. Mais nous allons aussi utiliser un véritable ordinateur quantique.

1. Un premier circuit quantique

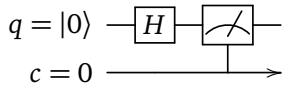
On se jette l'eau et on réalise notre premier circuit quantique. Nous utilisons le langage de programmation *Python* et la librairie *qiskit* fournie par IBM.

1.1. Le circuit

Il s'agit de programmer le circuit suivant.



On part donc de l'état initial $|0\rangle$, on applique une porte de Hadamard, l'état quantique devient donc $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. On termine par une mesure qui renvoie un bit classique 0 ou 1 avec ici chacun la probabilité $1/2$.
Pour mieux représenter la réalité de ce circuit, on l'écrit sur deux lignes. La première ligne correspond au qubit, noté q et à sa transformation, la seconde ligne correspond au bit classique, noté c, qui sert à stocker la mesure du qubit.



Remarque importante. Noter la différence avec les circuits rencontrés dans le premier chapitre : les circuits sont ici initialisés avec un état initial, $|0\rangle$ pour les qubits et 0 pour les bits classiques.

1.2. Le programme

```
import qiskit as q

### Partie A. Préparation

# On simule un ordinateur quantique
simulator = q.Aer.get_backend('qasm_simulator')

### Partie B. Construction du circuit
```

```
# Circuit quantique avec un qubit et une mesure
circuit = q.QuantumCircuit(1, 1)

# Une porte de Hadamard
circuit.h(0)

# Mesure du qubit (donne un bit classique)
circuit.measure(0, 0)

# Affichage du circuit
print(circuit.draw(output='text'))

### Partie C. Exécution

# Lancer de 1000 simulations
job = q.execute(circuit, simulator, shots=1000)

### Partie D. Résultats et visualisation

result = job.result()

# Comptage
counts = result.get_counts(circuit)
print("Nombre de '0' et de '1':",counts)

# Diagramme en barres
import matplotlib.pyplot as plt
q.visualization.plot_histogram(counts)
plt.show()
```

1.3. Explications et résultats

On reprend pas à pas le programme ci-dessus avec des explications et les résultats.

Partie A. Préparation

- Le module *Python* à importer au préalable est le module *qiskit*, on abrège son nom par la seule lettre `q`.
- Pour l'instant on n'utilise pas un véritable ordinateur quantique, mais on transforme notre machine en un simulateur avec l'option `'qasm_simulator'`.

Partie B. Construction du circuit

On commence par déclarer l'architecture du circuit :

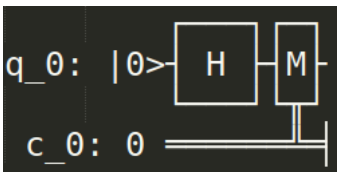
```
circuit = q.QuantumCircuit(1, 1)
```

L'instruction définit un circuit quantique, nommé `circuit`, et déclare le nombre de bits quantiques (ici 1) suivi du nombre de bits classiques (ici 1 également, pour la mesure).

On construit ensuite le circuit de la gauche vers la droite :

- on ajoute une porte H de Hadamard pour le qubit numéro 0 : `circuit.h(0)`.
- on mesure le qubit numéro 0 et on envoie le résultat sur le bit classique numéro 0 : `circuit.measure(0, 0)`.

Le programme affiche ensuite une version texte de notre circuit, ce qui permet de vérifier que tout est bien en place.



Comme l'entrée n'a pas été précisée, c'est, comme mentionné sur l'affichage ci-dessus, la valeur par défaut qui sera utilisée, à savoir $|0\rangle$.

Partie C. Éxécution

Le travail de simulation commence. Un test du circuit conduit à une mesure, avec une sortie 0 ou 1 (la valeur exacte du qubit dans le circuit n'est pas accessible). Une seule valeur ne permet pas de conclure sur la nature du circuit, c'est pourquoi on effectue une simulation avec un grand nombre de lancers (*shots*).

```
job = q.execute(circuit, simulator, shots=1000)
```

Partie D. Résultats et visualisation

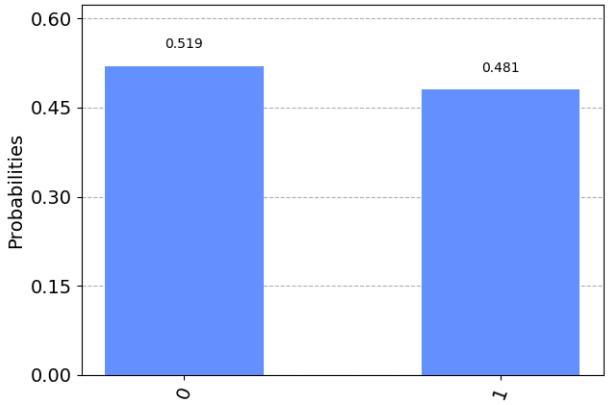
Voici un exemple de résultat renvoyé.

```
Nombre de '0' et de '1': '0': 519, '1': 481
```

Bien sûr, chaque simulation a une part d'aléatoire et conduit à des résultats différents. Cependant, selon la loi des grands nombres, par exemple avec 1000 lancers, la proportion de 0 et de 1 obtenue doit se rapprocher de la probabilité attendue. Ici les proportions de 0 et de 1 sont effectivement proches de la probabilité $\frac{1}{2}$ attendue :

$$p_0 = \frac{519}{1000} = 0.519 \qquad \text{et} \qquad p_1 = \frac{481}{1000} = 0.481.$$

On dispose aussi d'un affichage graphique.



En conclusion, notre circuit, qui réalise la mesure du qubit $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, fonctionne bien comme attendu et renvoie 0 ou 1 avec chacun une probabilité $\frac{1}{2}$.

2. Un qubit

2.1. Nombres complexes avec *Python*

Le nombre complexe $z = \frac{\sqrt{3}}{2} + \frac{1}{2}\mathrm{i}$ s'affiche avec *Python* de la manière suivante :

```
0.8660254037844386-0.5j
```

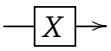
Le nombre i est représenté par `j` (ou plus exactement par `1j`). Malheureusement *Python* ne fait pas de calculs exacts, il utilise des nombres flottants pour la partie réelle et pour la partie imaginaire.
Voici comment définir et afficher ce nombre complexe :

```
import numpy as np
z = np.sqrt(3)/2 + 1/2j
print(z)
print(abs(z))
print(1/z)
```

On manipule les nombres complexes comme les nombres réels : `z1+z2`, `z1*z2`, `z**2`, `1/z`.
Le module du nombre complexe z est 1, la valeur renvoyée par la fonction `abs(z)` est `0.99999...`

2.2. Circuit

Nous allons programmer un circuit encore plus simple que précédemment :



Deux différences majeures cependant :

- Cette fois nous allons initialiser l'entrée par un qubit $|\psi\rangle$ quelconque (et non plus $|0\rangle$).
- Nous utiliserons un « faux » ordinateur quantique afin d'obtenir les états quantiques. En effet un « vrai » circuit quantique n'est pas pratique pour l'apprentissage car il ne permet d'obtenir que des probabilités approchées et pas les états quantiques de qubits.

2.3. Le programme

```
### Partie A. Préparation

simulator = q.Aer.get_backend('statevector_simulator')

### Partie B. Construction du circuit

circuit = q.QuantumCircuit(1)

# Initialisation à la main : écriture algébrique
alpha0 = 3+1j
beta0 = 1-2j
norme = np.sqrt(abs(alpha0)**2 + abs(beta0)**2)
alpha, beta = alpha0/norme, beta0/norme
etat_initial = [alpha,beta]
qubit_initial = q.extensions.Initialize(etat_initial)
circuit.append(qubit_initial, [0])
```

```
# Circuit : une porte X
circuit.x(0)

# Partie C. Exécution

job = q.execute(circuit, simulator)

# Partie D. Résultats

result = job.result()

coefficients = result.get_statevector()
print("Coefficient alpha:", coefficients[0])
print("Coefficient beta :", coefficients[1])
```

2.4. Explications et résultats

- Cette fois le simulateur appelé est `'statevector_simulator'`, ce qui nous permet de récupérer les états quantiques en plus de leur mesure. C'est pratique pour vérifier la validité de nos circuits mais cela ne correspond pas à la réalité physique !
- Cette fois, notre circuit est défini avec un seul qubit (indexé par 0). La valeur par défaut de ce qubit est $|0\rangle$.
- Mais on va changer cette valeur, on souhaite comme qubit initial :

  $$|\psi\rangle = (3+\mathrm{i})\,|0\rangle + (1-2\mathrm{i})\,|1\rangle\,.$$

  Pour cela on définit $\alpha_0 = 3+\mathrm{i}$, $\beta_0 = 1-2\mathrm{i}$.
  Afin que l'entrée soit acceptée, il faut normaliser le qubit $|\psi\rangle$. On calcule donc la norme $\|\psi\| = \sqrt{|\alpha_0|^2 + |\beta_0|^2}$. Et on définit $\alpha = \alpha_0/\|\psi\|$ et $\beta = \beta_0/\|\psi\|$. Ce qui nous permet de définir le qubit initial à l'aide de la commande `Initialise([alpha,beta])`.
- On rajoute une porte de Pauli X (qui échange les coefficients α et β du qubit).
- On récupère les coefficients du qubit de sortie par la fonction `get_statevector()`.
- Résultats. Notre qubit normalisé en entrée est :

  $$\left|\psi'\right\rangle = (0.7746 + 0.2582\mathrm{i})\,|0\rangle + (0.2582 - 0.5164\mathrm{i})\,|1\rangle$$

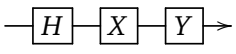
  La sortie obtenue :

  $$(0.2582 - 0.5164\mathrm{i})\,|0\rangle + (0.7746 + 0.2582\mathrm{i})\,|1\rangle$$

  est bien $X(\left|\psi'\right\rangle)$.

Toutes les portes classiques sont disponibles : la porte H de Hadamard, les portes de Pauli X, Y, Z...

Exercice.
On considère le circuit :



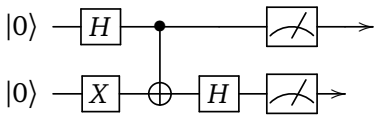
et le qubit d'entrée

$$|\psi\rangle = \frac{\sqrt{3}}{2}\,|0\rangle + \frac{1-\mathrm{i}}{2\sqrt{2}}\,|1\rangle$$

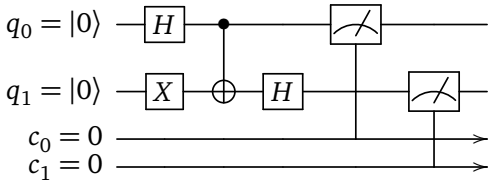
qui est un qubit de norme 1. Calculer la sortie à l'aide de la machine et vérifier vos calculs à la main.

3. Deux qubits

3.1. Le circuit



Voici une représentation plus réaliste de ce circuit, avec les deux lignes pour les bits classiques qui servent de stockage pour les mesures :



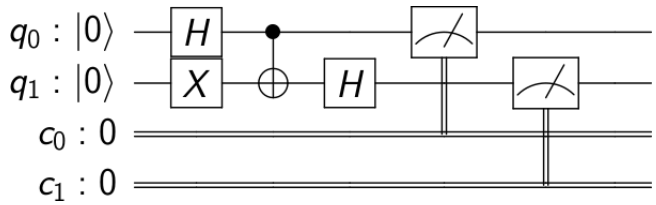
3.2. Le programme

```
### Partie A. Préparation
simulator = q.Aer.get_backend('qasm_simulator')

### Partie B. Construction du circuit
circuit = q.QuantumCircuit(2, 2)  # 2 qubits et 2 mesures

circuit.h(0)  # Porte de Hadamard sur le premier qubit
circuit.x(1)  # Porte X sur le second qubit
circuit.cx(0, 1) # CNOT
circuit.h(1)  # Porte de Hadamard sur le second qubit
circuit.measure([0,1], [0,1]) # Mesure (q0->c0,q1->c1)

# Affichage graphique du circuit
import PIL
img_circuit = circuit.draw(output='latex')
img_circuit.show()

### Partie C. Exécution
job = q.execute(circuit, simulator, shots=1000)

### Partie D. Résultats et visualisation
result = job.result()
counts = result.get_counts(circuit)
print("Nombre de '00', '01', '10' et de '11':",counts)

# Diagramme en barres
q.visualization.plot_histogram(counts)
plt.show()
```

3.3. Commentaires

Voici les quelques points nouveaux.

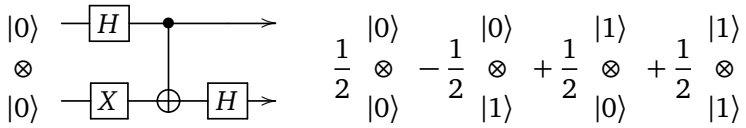
- Il faut préciser le numéro de ligne du circuit lorsque l'on ajoute une porte, par exemple `circuit.h(i)`. L'ajout se fait toujours de la gauche vers la droite.
- Pour une porte $CNOT$, il faut bien sûr préciser deux numéros de lignes.
- Pour la mesure, on précise d'abord la liste des lignes de qubits à lire et ensuite la liste des destinations.
- Ici on propose un affichage graphique du circuit construit (qui nécessite LaTeX).



3.4. Résultats

Vérifier que le qubit de sortie attendu (avant mesure) est :

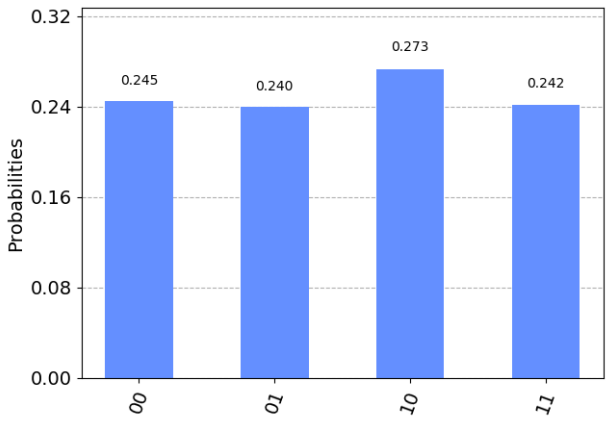
$$|\psi\rangle = \frac{1}{2}|0.0\rangle - \frac{1}{2}|0.1\rangle + \frac{1}{2}|1.0\rangle + \frac{1}{2}|1.1\rangle\,.$$



Expérimentalement, voici un exemple de ce que renvoie le programme :

```
'00': 245, '01': 240, '10': 273, '11': 242
```

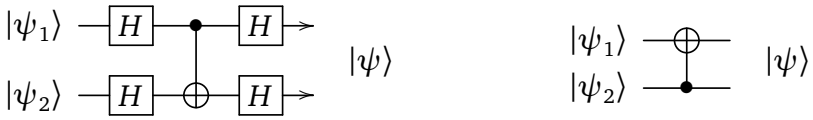
L'affichage graphique met en évidence que chacune des 4 mesures possibles 0.0, 0.1, 1.0, 1.1 est équiprobable.



Piège ! Il y a une inversion entre notre notation $|0.1\rangle$ (qui se mesure 0.1) et celle de *qiskit* `'10'`. De même $|1.0\rangle$ correspond à `'01'`. En effet, *qiskit* adopte la convention d'écriture des nombres binaires dans laquelle les bits sont écrits de droite à gauche.

Écriture d'un qubit	$\|c_0.c_1 \ldots c_k\rangle$
Écriture de sa mesure	$c_0.c_1 \ldots c_k$
Notation *qiskit*	'c_k … c_1 c_0'

Exercice.
On souhaite vérifier expérimentalement que les deux circuits suivants sont équivalents (des entrées égales donnent des sorties égales).



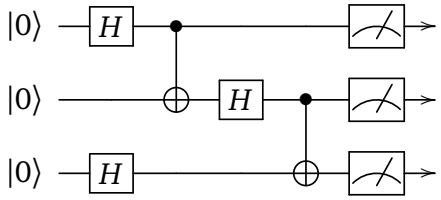
Voici les étapes.

1. Utiliser le moteur `'statevector_simulator'`.
2. Définir au hasard un qubit de norme 1, $|\psi_1\rangle$. Pour cela, on suit la méthode utilisée dans le programme de la section 2.3 :
   - définir des nombres complexes α_0 et β_0 (choisis au hasard),
   - puis calculer la norme $n = \sqrt{|\alpha_0|^2 + |\beta_0|^2}$,
   - puis $\alpha = \alpha_0/n$ et $\beta = \beta_0/n$ définissant $|\psi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ de norme 1,
   - puis utiliser la fonction `Initialize()` pour définir ce qubit.
3. Définir de même $|\psi_2\rangle$.
4. Définir le circuit `circuit1` suivant le premier schéma et calculer le 2-qubit $|\psi\rangle$ de sortie.
5. Définir le circuit `circuit2` suivant le second schéma et calculer le 2-qubit $\left|\psi'\right\rangle$ de sortie.
6. Comparer les sorties $|\psi\rangle$ et $\left|\psi'\right\rangle$.

On doit avoir $|\psi\rangle = \left|\psi'\right\rangle$ quel que soit le choix des entrées $|\psi_1\rangle$ et $|\psi_2\rangle$.

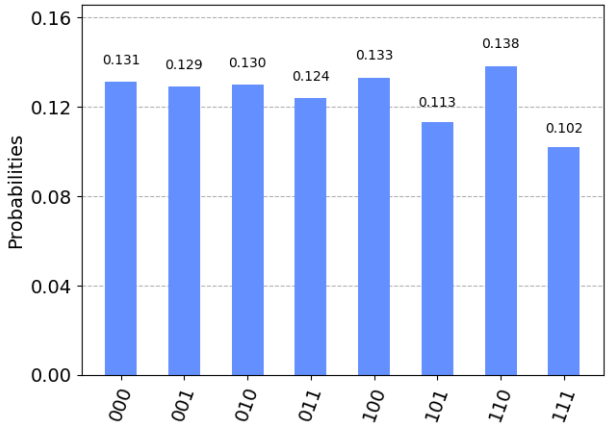
3.5. Plus de qubits

On peut bien sûr avoir davantage de qubits en entrée. Voici un exemple de circuit avec trois qubits.




```
circuit = q.QuantumCircuit(3, 3)
circuit.h(0)      # Porte de Hadamard
circuit.h(2)      # Porte de Hadamard
circuit.cx(0, 1)  # CNOT
circuit.h(1)      # Porte de Hadamard
circuit.cx(1, 2)  # CNOT
circuit.measure([0,1,2], [0,1,2]) # Mesures
```

Voici un exemple de résultat :



4. Utiliser un vrai ordinateur quantique

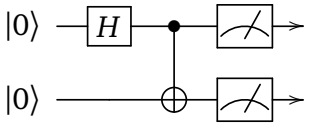
4.1. Accès à un ordinateur quantique

Un des intérêts de *qiskit* est que l'on peut exécuter ses programmes sur un ordinateur quantique ! En effet, IBM met à disposition du temps de calculs sur des véritables ordinateurs quantiques. Cet accès est gratuit et ouvert à tous. Il faut cependant s'inscrire et patienter dans une file d'attente pour lancer son programme. Voici les étapes préalables.

- Se créer un compte sur le site *quantum-computing.ibm.com*.
- Récupérer son code d'accès (*token*) qui est un un long mot de passe du genre `'ce5a6210bb21...'`. Ce code n'est nécessaire que pour la première connexion.

4.2. Programme

Voici un circuit.



Et voici le programme qui s'exécute à distance sur un vrai ordinateur quantique.

```
import qiskit as q
from qiskit.tools.monitor import job_monitor
import matplotlib.pyplot as plt

### Partie A. Préparation

# Code à donner une fois seulement, ensuite commenter la ligne
q.IBMQ.save_account('ce5a6210bb21...')

q.IBMQ.load_account()

provider = q.IBMQ.get_provider(group='open')

print(provider.backends())  # Affiche les ordinateurs disponibles

backend = provider.get_backend('ibmq_essex')  # Choix d'un ordinateur disponible
```

```
### Partie B. Construction du circuit

circuit = q.QuantumCircuit(2, 2)
circuit.h(0)
circuit.cx(0, 1)
circuit.measure([0,1], [0,1])

### Partie C. Exécution

job_exp = q.execute(circuit, backend=backend, shots=1000)
job_monitor(job_exp)

# Partie D. Résultats et visualisation

result_exp = job_exp.result()
counts_exp = result_exp.get_counts(circuit)
print(counts_exp)

q.visualization.plot_histogram(counts_exp)
plt.show()
```

4.3. Explications

Partie A. Préparation. La préparation consiste à donner son code d'accès (une seule fois), à choisir un accès libre (groupe `'open'`), puis à choisir un ordinateur quantique parmi une liste disponible (ici `'ibmq_essex'`).
Partie B. Construction du circuit. Comme d'habitude !
Partie C. Exécution. Presque comme d'habitude sauf qu'il faut être un peu plus patient pour disposer de l'accès et attendre la fin des calculs.
Voici la liste des messages qui informent l'utilisateur de l'avancement du travail :

```
Job Status: job is being validated
Job Status: job is queued
Job Status: job is actively running
Job Status: job has successfully run
```

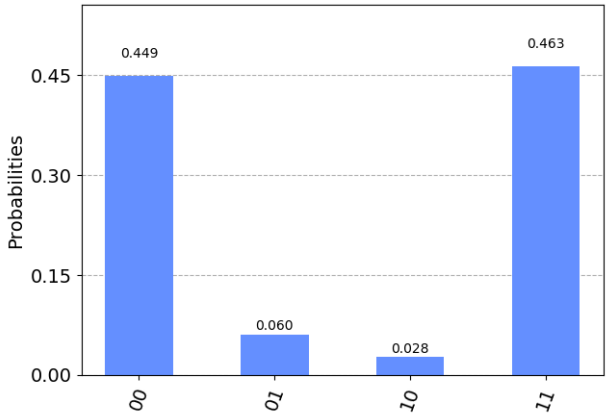
Si l'attente semble trop longue il existe un moyen de retrouver les résultats plus tard (à l'aide de la commande `retrieve_job`).

4.4. Résultats

Voici un exemple de résultats sous forme numérique :

```
'00': 449, '01': 60, '10': 28, '11': 463
```

et sous forme graphique :



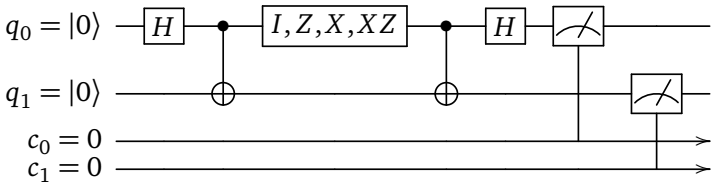
Noter que l'état quantique de sortie (avant mesure) est $\left|\Phi^+\right\rangle = \frac{1}{\sqrt{2}}|0.0\rangle + \frac{1}{\sqrt{2}}|1.1\rangle$. Les résultats devraient donc être uniquement mesurés en 0.0 ou 1.1 (avec des probabilités proches de $1/2$). Mais sur un véritable ordinateur quantique il y a des erreurs, qui ici produisent certaines mesures impossibles en théorie (ici 0.1 et 1.0).
On distingue deux causes qui font que l'on n'obtient pas exactement moitié/moitié pour les mesures 0.0 et 1.1 :

- une cause probabiliste : il peut y avoir un écart entre les mesures et les probabilités théoriques attendues, écart dû au caractère aléatoire d'une mesure (comme le lancer d'une pièce de monnaie). Avec un grand nombre de lancers, cet écart diminue.
- une cause d'erreur physique : un ordinateur quantique n'est pas parfait, il peut y avoir des erreurs.

5. Codage super-dense

5.1. Circuit

Il s'agit de transmettre une information classique composée de deux bits 0.0 ou 0.1 ou 1.0 ou 1.1 en transmettant un seul qubit. On renvoie à la fin du chapitre « Découverte de l'informatique quantique » pour les explications. On rappelle le circuit en jeu :



5.2. Programme

```
import qiskit as q

### Partie A. Préparation
simulator = q.Aer.get_backend('qasm_simulator')

### Partie B. Construction du circuit

circuit = q.QuantumCircuit(2, 2)

## B.1 Préparation de l'état de Bell
```

```
circuit.h(0)  # Porte de Hadamard
circuit.cx(0, 1) # CNOT

message_alice = '01'  # choix entre '00', '01', '10', '11'

## B.2 Porte d'Alice selon message à transmettre
if message_alice == '00':
    circuit.iden(0)  # identité
elif message_alice == '01':
    circuit.z(0)     # porte Z
elif message_alice == '10':
    circuit.x(0)     # porte X
elif message_alice == '11':
    circuit.x(0)     # porte X
    circuit.z(0)     # suivi de porte Z


## B.3 Décodage
circuit.cx(0, 1) # CNOT
circuit.h(0)  # Porte de Hadamard

## B.4 Mesures
circuit.measure([0,1], [0,1]) # Mesures

print(circuit.draw(output='text'))

### Partie C. Exécution

# Lancer de 1000 simulations
job = q.execute(circuit, simulator, shots=1000)

# Partie D. Résultats

result = job.result()

# Comptage
counts = result.get_counts(circuit)
print("Nombre de '00', '01', '10' '11':",counts)
```

5.3. Résultats

Le programme s'utilise en choisissant le message qu'Alice envoie à Bob, par exemple `message_alice = '01'`. Dans ce cas, Alice ajoute une porte Z au circuit. La sortie mesurée est alors `'01'` (dans 100% des simulations). Testez les autres messages !
On rappelle la subtilité : si la mesure du qubit q_0 est 1 et celle du qubit q_1 est 0 alors on note globalement la mesure 1.0 (correspondant à l'état quantique $|1.0\rangle$) mais *qiskit* écrit cette même mesure `'01'` (de la droite vers la gauche). Faites un choix pour votre programme et tenez-vous-y !