BSQ 110 - Introduction aux sciences quantiques et à leurs applications

Premier projet de programmation

Simulation d'un circuit quantique

À remettre le vendredi 11 novembre 2022

1 Introduction

Dans ce projet, vous allez utiliser la librairie Numpy pour construire les matrices unitaires qui représentent l'application de différentes portes quantiques et les composer pour obtenir les matrices qui représentent des circuits quantiques complets. Vous allez également simuler la mesure en utilisant la génération de nombres aléatoires.

Chaque équipe devra aborder deux circuits quantiques différents, lesquels partagent certaines problématiques, mais possèdent également leurs défis spécifiques. À vous de voir comment vous voulez vous séparer les différentes tâches entre les membres de votre équipe.

1.1 Critères d'évaluation

Votre travail devra être remis sous la forme de scripts écrits dans un ou plusieurs fichiers `.py` ou encore sous la forme d'un *notebook* Jupyter `.ipynb`. Votre travail sera évalué selon les points suivants :

— La validité de vos solutions. Est-ce que ça fonctionne ?

— La clarté de votre code. Est-ce que votre code est facile à lire et à comprendre ? Vous pouvez consulter la section 1.2 « Bonnes habitudes de programmation » pour vous guider.

— La clarté de vos explications.

— L'originalité de votre solution. Avez-vous utilisé des approches alternatives à celles proposées ?

1.2 Bonnes habitudes de programmation

Pourquoi faire du bon code ? Qu'est-ce que c'est du bon code ? Un élément de réponse commun à ces deux questions est la lisibilité. Lire et comprendre du code écrit par quelqu'un d'autre (ou vous-même dans le passé) est rarement évident. Il est donc essentiel de maximiser sa clarté. Cela augmentera grandement les chances que vos solutions soient réutilisées dans le futur et donne de la valeur à votre travail. Avec cet objectif en tête, voici quelques bonnes habitudes de programmation à prendre :

— Utilisez des noms de variables informatifs.

— Soyez cohérents dans vos conventions de noms de variables, fonctions, etc.

— Un code qui se comprend sans commentaire est un bon code. Utilisez des commentaires lorsqu'il est difficile ou impossible de le rendre suffisamment lisible.

— Utilisez l'encapsulation. Évitez les répétitions. Si vous avez à changer comment vous faites quelque chose, il est préférable d'avoir à corriger votre code à un seul endroit.

— Faites en sorte que votre code soit réutilisable dans un autre contexte. Les outils développés pour construire la matrice pour un circuit devraient être utiles pour le second circuit également.

— Idéalement, chaque fonction/méthode ne devrait faire qu'une seule chose.

Cela étant dit, il y a une balance entre la qualité du code qu'on produit et la quantité. Si vous tentez d'écrire un code parfait du premier coup, vous risquez de progresser très lentement. Il peut parfois être plus efficace de rapidement programmer une solution qui fonctionne et, dans un deuxième temps, repasser sur le code afin d'améliorer sa lisibilité.

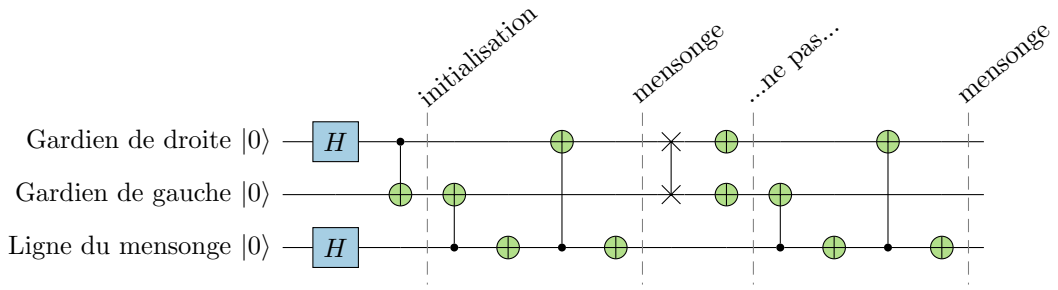

FIGURE 1 – Circuit quantique de l'énigme « La porte du trésor »

1.3 Suggestions

Nous suggérons fortement l'utilisation de GitHub pour gérer votre code et faciliter votre travail en équipe. Ensuite, tentez d'identifier les grands objectifs que votre équipe devra accomplir pour vous séparer les tâches. Partagez vos résultats à l'intérieur de votre équipe, cela vous permettra de valider vos idées et vos solutions.

2 Description du projet

Les circuits quantiques que vous allez considérer pour ce projet sont ceux des Énigmes quantiques « La porte du trésor » et « Le problème de Monty Hall ». Pour chacun d'eux, vous aurez à remplir les mêmes objectifs généraux. Ensuite, la description de chaque circuit sera suivie d'indications qui vous aideront à atteindre ces objectifs. Nous proposons également un objectif optionnel pour chaque circuit. À vous de décider si vous voulez vous y attaquer. Décrivons d'abord ces objectifs généraux avant de présenter les circuits et leurs objectifs optionnels.

2.1 Objectifs généraux

Ces objectifs doivent être remplis pour les deux circuits quantiques. Les outils que vous développez pour le premier circuit peuvent être réutilisés pour le second.

Objectif 1 Construire les matrices qui représentent l'application des sous-circuits quantiques (ou section). Le premier circuit comporte 4 sections (dont deux sont identiques) et le second seulement 2 sections.

Objectif 2 Assemblez la matrice qui représentent l'application des circuits quantiques complets.

Objectif 3 Simuler l'exécution de ces circuits quantiques en appliquant les matrices unitaires des circuits sur l'état quantique initial $|0\dots0\rangle$ pour obtenir les états quantiques finaux.

Objectif 4 Simulez des résultats de mesure à l'aide du module `random` de Numpy. Consultez la section 4 pour des indications détaillées.

2.2 Premier circuit : La porte du trésor

L'énigme quantique « La porte du trésor » utilise un circuit quantique à trois qubits pour représenter la solution à une énigme classique. Pour vous immerger dans le problème, vous pouvez visionner la vidéo sur YouTube. Le circuit quantique de cette énigme est illustré à la figure 1.

FIGURE 2 – Exemple de propriétés de commutation des portes NOT et CNOT.

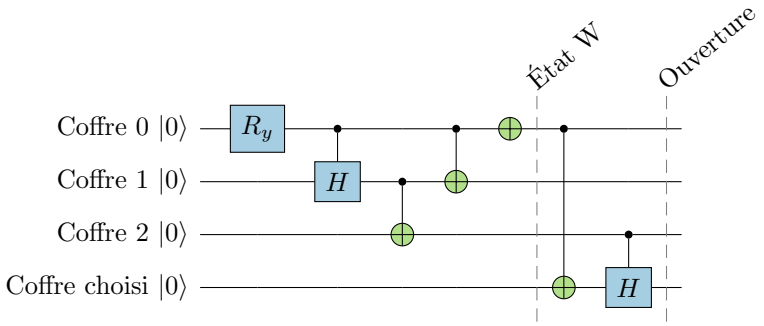

FIGURE 3 – Circuit quantique de l'énigme « Le problème de Monty Hall »

Indication 1 Utiliser le produit tensoriel pour obtenir les matrices 8×8 qui représentent l'application des portes à un qubit incluses dans ce circuit à trois qubits.

Indication 2 Obtenez les matrices 8×8 qui représentent l'application des portes à deux qubits pour ce système de trois qubits. Pour ce circuit, il y a trois CNOT différents et une porte SWAP.

Objectif optionnel Utilisez des propriétés de commutation afin de simplifier au maximum ce circuit quantique. La figure 2, illustre des exemples de propriétés de commutation. Vous aurez à en trouver d'autres. La matrice représentant le circuit simplifié devrait être identique à celui obtenu à l'objectif précédent. Illustrez chacune des étapes de votre raisonnement. Cette partie peut être remise en format papier.

2.3 Deuxième circuit : Le problème de Monty Hall

L'énigme quantique « Le problème de Monty Hall » utilise un circuit quantique à quatre qubits pour représenter la solution à une énigme classique. Pour vous immerger dans le problème, vous pouvez visionner la vidéo sur YouTube. Le circuit quantique de cette énigme est illustré à la figure 3. En particulier, la première partie du circuit prépare un état quantique à trois qubits

$$|\mathrm{W}\rangle = \frac{1}{\sqrt{3}}(|001\rangle + |010\rangle + |100\rangle).$$

Indication 1 Ce circuit débute avec une porte paramétrée $\hat{R}_y(\theta)$ dont l'angle doit être égal à $\theta = 2\arccos\left(1/\sqrt{3}\right)$. Utilisez Python pour calculer cet angle. Ensuite, définissez une fonction qui prépare la matrice 2×2 pour une rotation d'un angle donnée.

```
def ry_gate(angle):
    gate_matrix = ...
    return gate_matrix
```

Indication 2 Comme chacun des sous-circuits n'implique que trois qubits, vous pouvez les construire d'abord sous la forme de matrices 8×8. Comment arriverez vous à les convertir en circuits à quatre qubits ?

Objectif optionnel Modifiez le circuit qui prépare l'état W afin de déséquilibrer les probabilités que le trésor se cache derrière chacune des portes. Trouvez ensuite une distribution de probabilité où ce n'est plus avantageux de changer votre choix.

3 Matrices et portes quantiques avec Numpy

La librarie Numpy est un outil très utile pour vous aider à remplir les différents objectifs. L'utilisation des `ndarray` permet de traiter les portes quantiques et les états quantiques sous forme de matrices et de vecteurs.

Dans cette section, nous allons voir comment utiliser cet outil pour simuler la préparation d'une paire de Bell grâce à un circuit à deux qubits. On aura d'abord besoin d'importer la librairie Numpy.

```
import numpy as np
```

3.1 Définir des portes quantiques

On peut construire les matrices représentant différentes portes quantiques grâce à la fonction `array()`. On doit fournir une liste de listes pour générer une matrice. Voici quelques exemples où on construit une matrice identité à un qubit, une porte Hadamard et une porte CNOT.

```
i_gate = np.array([[1,0],[0,1]])
h_gate = np.sqrt(0.5) * np.array([[1,1],[1,-1]])
cx_gate = np.array([[1,0,0,0],[0,0,0,1],[0,0,1,0],[0,1,0,0]])
```

3.2 Combiner des portes quantiques

Illustrons maintenant comment manipuler ces objets pour simuler l'exécution du circuit de préparation de paire de Bell présenté à la figure 4.

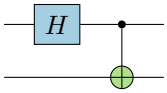

FIGURE 4 – Circuit de présentation d'état de Bell

3.2.1 Produit tensoriel

D'abord, on doit utiliser le produit tensoriel pour décrire l'effet de la porte Hadamard sur le système de deux qubits. Pour cela on utilise la fonction `kron()` [1].

```
ih_gate = np.kron(i_gate, h_gate)
```

On peut visualiser le résultat de ce produit tensoriel grâce à la fonction `print()`.

```
print(ih_gate)
```

1. Le produit tensoriel porte également le nom de produit de Kronecker, d'où le nom de la fonction.

```
[[ 0.7071 0.7071 0.      0.    ]
 [ 0.7071 -0.7071 0.    -0.    ]
 [ 0.      0.      0.7071 0.7071]
 [ 0.     -0.      0.7071 -0.7071]]
```

On obtient bien le résultat attendu.

3.2.2 Composition

On compose ensuite cette matrice avec la matrice du CNOT avec un produit matricielle. La fonction Numpy `matmul` permet d'effectuer cette opération.

```
u_circuit = np.matmul(cx_gate, ih_gate)
```

Notez que l'opérateur surchargé `@` permet d'effectuer la même opération tout en simplifiant l'écriture.

```
u_circuit = cx_gate @ ih_gate
```

3.3 Transformation d'un état quantique

Maintenant que nous avons en main la matrice qui représente l'application du circuit quantique, nous pouvons l'appliquer à un état quantique. On définit l'état quantique comme un vecteur en fournissant une liste de composantes à la fonction `array()`. Ici on débute avec l'état $|00\rangle$. On peut ensuite appliquer le circuit à cet état initial pour obtenir l'état final, qu'on affiche.

```
init_state = np.array([1,0,0,0])
bell_state = u_circuit @ init_state
print(bell_state)
```

```
[0.7071 0.    0.    0.7071]
```

On obtient bien, comme attendu, le premier état de Bell

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

4 Simuler la mesure des qubits

Pour simuler les résultats aléatoires d'une mesure de qubits, vous devez programmer une fonction qui échantillonne un vecteur d'état un certain nombre de fois et qui retourne le nombre de fois que chaque état de base a été obtenu. La définition de celle-ci pourrait ressembler à :

```
def sample_state(state_vector, shots):
    counts = ...
    return counts
```

Cette fonction doit prendre deux arguments en entrée : le vecteur d'état (`state_vector`) et le nombre d'échantillons `shots` et retourner un dictionnaire (`counts`) qui contient les états de base obtenus et le nombre de fois que chacun d'eux a été obtenu. L'utilisation de cette fonction sur l'état de Bell préparé plus haut et pour 100 échantillons pourrait ressembler à :

```
counts = sample_state(bell_state, 100)
print(counts)
```

```
{'00': 55, '11': 45}
```

Les résultats obtenus devraient être aléatoires. C'est pour cela qu'on n'obtient pas nécessairement le résultat exact attendu de 50/50. Voici quelques indications pour atteindre cet objectif :

— Commencez par obtenir le vecteur de probabilités à partir du vecteur d'état.

— Générez ensuite des résultats aléatoires à l'aide du sous-module `random` de Numpy. Il existe plusieurs fonctions qui peuvent vous aider à remplir cet objectif. La plus adaptée semble être la fonction `choice()` (voir la documentation), mais d'autres solutions sont aussi possibles.

— Comptez le nombre de fois que vous avez obtenu chacun des états de base.

— Les états de base devraient être retournés sous la forme d'une `string` représentant la chaine de bits. Une méthode efficace pour faire cela est d'utiliser le formatage de `string`. Par exemple :

```
value = 1
number_of_qubits = 3
bit_string = f"{value:0{number_of_qubits}b}"
print(bit_string)
```

```
001
```

5 Extra

5.1 Porte contrôlée

Construisez une fonction qui retourne une version contrôlée d'une porte quantique. La porte originale doit pouvoir être une porte à n qubits, de sorte que la version contrôlée est une porte à $n+1$ qubits.

```
def control(gate):
    ...
    return control_gate
```

5.2 Réorganiser les qubits

Construisez une fonction qui réorganise l'ordre des qubits pour une porte donnée.

```
def control(gate,order):
    ...
    return new_gate
```