
BSQ 112 - Introduction au calcul quantique

Mon simulateur quantique

Simulation de circuits quantiques

À remettre le 10 novembre 2025

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 devrez d'abord élaborer un certain nombre d'outils qui vous rendront la tâche de construction de circuit beaucoup plus facile. Vous allez également simuler la mesure en utilisant la génération de nombres aléatoires.

Vous utiliserez ensuite ces outils pour simuler l'exécution de circuits quantiques simples.

1.1 Premiers pas

Nous utiliserons GitHub ainsi et GitHub Classroom pour ce projet. Suivez les étapes suivantes pour démarrer votre projet :

1. Créer un compte GitHub si vous n'en avez pas déjà un.
2. Suivez ce [lien](#) pour créer votre équipe. Ce projet doit se faire de manière individuelle.
3. Accéder au *repo*.
4. Créez une branche pour le développement (ne travaillez pas directement dans la branche `main`).
5. Écrivez votre nom dans le fichier `README.md`, surtout si votre identifiant GitHub ne permet pas de vous identifier.
6. Commencez à coder !

1.2 Structure du projet

La structure du projet telle que vous la trouverez est illustrée à la figure 1. D'abord, le dossier `my_quantum_simulator` contient les fonctionnalités de votre projet et constitue la majeure partie du travail. Le dossier `tests` contient des tests qui permettront de vérifier vos implémentations. Vous pouvez modifier et bonifier ces tests si vous le désirez. Finalement, le dossier `usage` contient des exemples d'utilisation des fonctionnalités. Tous les fichiers en `rouge` devront être complétés.

1.3 Livrables

Lorsque vous aurez complété le projet, vous pourrez soumettre le contenu de votre branche `main` pour évaluation. Assurez-vous que cette branche possède la structure originale et que les fichiers en `rouge` sont dûment complétés. Idéalement, tous les tests devraient également pouvoir être exécutés sans problèmes.

1. Les fichiers dans `my_quantum_simulator/` avec toutes les fonctions complétées. N'hésitez pas à utiliser les tests pour valider vos implémentations.
2. Deux fichiers `.py` dans `usage` qui appliquent les fonctionnalités de `my_quantum_simulator` à deux exemples de circuits quantiques. Dans ceux-ci vous devez obtenir la matrice du circuit quantique et simuler les mesures. Basez-vous sur l'exemple fourni `bell_circuit.py`.

Vous pouvez soumettre une version préliminaire de votre code au moins une semaine avant la date de remise du projet pour qu'il soit révisé et commenté. Vous devez aviser l'enseignant lorsque vous désirez obtenir cette rétroaction. N'attendez pas à la dernière minute pour demander une révision.

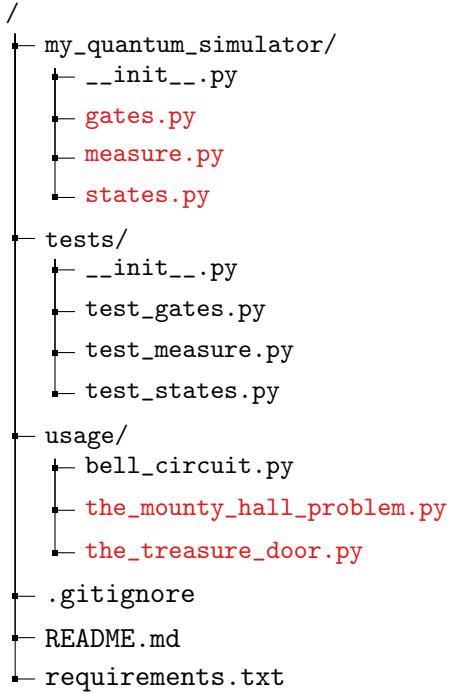


FIGURE 1 – Structure du projet `my_quatum_simulator`

1.4 Critères d'évaluation

Votre travail sera évalué selon les points suivants :

- La validité de vos solutions. Est-ce que ça fonctionne ? Est-ce que ça passe les tests ?
- La clarté de votre code. Est-ce que votre code est facile à lire et à comprendre ? Vous pouvez consulter la section 6 « Bonnes habitudes de programmation » pour vous guider.
- La clarté de vos explications si nécessaire.
- L'originalité de votre solution. Avez-vous utilisé des approches alternatives à celles proposées ?

1.5 Les modules

Pour ce simulateur, nous utiliserons principalement le module `numpy` pour manipuler les matrices représentant des portes quantiques et des vecteurs d'état. Consultez la section 4 pour voir comment vous pouvez utiliser `numpy` pour représenter et simuler des portes quantiques.

Nous utiliserons `pytest` pour tester notre code. Consultez la section 5 pour voir comment tester votre code avec `pytest`.

Nous utiliserons `flit` comme gestionnaire de module (*package*), surtout pour gérer l'instalation de votre module python.

1.6 Installation

Dans ce projet, vous allez construire un module python `my_quantum_simulator`. Lorsqu'on développe un module de la sorte, il est très utile de l'installer dans votre environnement python pour simplifier l'importation et l'exécution des tests. Nous allons utiliser le module `flit` pour la création du module. Il existe d'autres gestionnaires de module et vous êtes libre d'en utiliser un autre si vous le désirez. Si vous désirez utiliser `flit` vous devez l'installer. Assurez-vous d'être dans l'environnement dans lequel vous développer `my_quantum_simulator` et utiliser la commande suivante.

```
pip install flit
```

Pour initialiser votre module, naviguer jusqu'au dossier racine du module (là où se trouve le fichier `README.md`) et exécuter la commande suivante.

```
flit init
```

Répondez aux questions qui vous seront posées. Il n'est pas nécessaire de spécifier une licence si vous ne distribuez pas votre code. Cela créera un fichier `.toml` dans votre dossier racine.

Normalement, l'installation d'un module python dans un environnement implique de copier les fichiers sources dans cet environnement. Lorsqu'on développe, il est alors nécessaire de réinstaller le module après chaque modification. Pour éviter cela, on peut installer le module, toujours à partir du dossier racine du module, avec la commande suivante.

```
flit install --symlink
```

Plutôt que de copier les fichiers de votre module dans l'environnement, `flit` créer un lien symbolique vers le dossier racine de votre projet. Ainsi, chaque modification sera prise en compte sans devoir réinstaller le module.

Par contre, il sera quand même nécessaire de réimporter le module après chaque modification. Si vous travaillez en mode interactif, cela nécessite de redémarrer le Kernel.

2 Votre simulateur quantique

Votre simulateur quantique comportera quelques fonctionnalités principales :

- Construire et assembler les matrices des transformations unitaires qui représentent l'application de certains circuits quantiques ;
- Construire un état quantique initial et le modifier en appliquant une transformation unitaire ;
- Simuler la mesure aléatoire des qubits.

Pour vous permettre d'atteindre ces objectifs, vous devrez construire un certain nombre d'outils, sous la forme de fonctions, qui vous simplifieront grandement la tâche.

Les fonctions suivantes doivent être implémentées et testées. Rien ne vous empêche d'en implémenter d'autres si vous les jugez utiles.

2.1 Construction de portes quantiques

Bien que votre implémentation doit reposer sur `numpy`, l'utilisateur ne devrait avoir à invoquer `numpy` directement ou du moins le moins possible. Vous allez donc construire quelques outils qui vous permettront de manipuler des portes quantiques.

2.1.1 Porte paramétrée

Écrivez une fonction qui retourne la matrice pour une porte $R_y(\theta)$ pour un angle donné.

```
def build_ry_gate(angle: float) -> NDArray:  
    gate_matrix = ...  
    return gate_matrix
```

2.1.2 Porte contrôlée

Écrivez 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 contrôlée par le qubit 0.

```
def control(gate: NDArray) -> NDArray:  
    ...  
    return control_gate
```

Par exemple, en appliquant cette fonction sur une porte \hat{X}_0 vous devriez obtenir un $C_0\hat{X}_1$. Ensuite, en l'appliquant sur $C_0\hat{X}_1$ vous devriez obtenir la porte $C_0C_1\hat{X}_2$ aussi appelée la porte Toffoli.

```
x0_gate = np.array([[0,1],[1,0]])  
c0x1_gate = control(x0_gate)  
c0c1x2_gate = control(c0x1_gate)  
print(c0c1x2_gate)
```

```
[[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 0 0 0 1]  
 [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 1 0 0 0 0]]
```

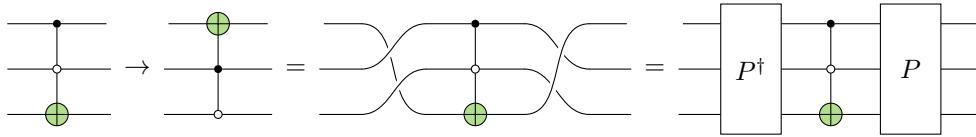


FIGURE 2 – Exemple de la transformation d'une porte à trois qubits sous une permutation dans l'ordre des qubits.

2.1.3 Composer des portes quantiques

Nous savons que pour composer deux portes quantiques, nous devons multiplier leur matrice. Vous devez définir une fonction qui vous permette de composer plusieurs portes quantiques. Ici, la première porte fournie doit être la première porte appliquée.

```
def compose_gates(gates: tuple[NDArray]) -> NDArray:
    ...
    return composed_gate
```

Un exemple d'application de cette fonction.

```
XHZ_gate = compose_gates((Z_GATE, H_GATE, X_GATE))
print(XHZ_gate)
```

```
[[ 0.70710678  0.70710678]
 [ 0.70710678 -0.70710678]]
```

2.1.4 Réorganiser les qubits

Cette fonctionnalité est un peu plus complexe que les autres. N'hésitez pas à en discuter avec vos collègues et à demander plus d'indications.

Construisez une fonction qui réorganise l'ordre des qubits pour une porte donnée. Les arguments de cette fonction sont d'abord la porte quantique dont on veut réorganiser les qubits et le nouvel ordre des qubits. Pour une porte à n qubits, la variable `order` devrait être une liste d'entiers qui comportent tous les éléments de 0 à $n - 1$ dans un ordre donné. Chaque entier indique à quel endroit le qubit sera après la transformation.

Afin de bien comprendre le rôle de cette fonction, illustrons son application avec un exemple, en évitant de rentrer dans les détails. Nous verrons ensuite une méthode systématique qui permettra d'effectuer les mêmes opérations plus simplement.

On veut modifier l'ordre des qubits pour l'application d'une porte Toffoli modifiée comme illustré à la figure 2. Dans ce cas l'argument `order` devrait être `[1, 2, 0]` ce qui implique que le premier qubit (q_0) s'en va à la position 1, le second (q_1) s'en va à la position 2, et le dernier qubit (q_2) s'en va à la position 0. On voit que le changement d'ordre des qubits revient à appliquer une permutation des fils avant et après avoir appliqué la porte originale.

Cette réorganisation des qubits engendre une réorganisation des états de base. Le tableau 1 montre comment les états de base à trois qubits changent d'ordre lorsqu'on change l'ordre des qubits en suivant la transformation \hat{P} .

Ensuite, ce changement dans l'ordre des états de base se traduit par une permutation des lignes *et* une permutation des colonnes de la matrice qui représente la porte quantique. Pour relier cela à notre exemple, on doit d'abord établir la matrice qui représente la porte Toffoli modifiée. On peut rapidement voir que cette porte échange les deux états de base $|001\rangle$ et $|101\rangle$, c'est-à-dire les états d'indice 1 et 5. On peut

$q_2 q_1 q_0$	j	$q_1 q_0 q_2$	j'
0 0 0	0	0 0 0	0
0 0 1	1	0 1 0	2
0 1 0	2	1 0 0	4
0 1 1	3	1 1 0	6
1 0 0	4	0 0 1	1
1 0 1	5	0 1 1	3
1 1 0	6	1 0 1	5
1 1 1	7	1 1 1	7

TABLE 1 – Changement de l'ordre des états de base à trois qubits lors de l'échange des qubits $q_0 \rightarrow 1$, $q_1 \rightarrow 2$ et $q_2 \rightarrow 0$.

ensuite modifier la matrice qui représente cette transformation en échangeant d'abord ses lignes et ensuite ses colonnes. La matrice résultante devrait échanger les états de base $|010\rangle$ et $|011\rangle$, c'est-à-dire les états d'indice 2 et 3.

$$\begin{array}{c} \left(\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right) \xrightarrow{\substack{\text{lignes} \\ \text{colonnes}}} \left(\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right). \end{array}$$

Maintenant qu'on a vu les concepts clés, voyons voir comment utiliser `numpy` pour établir le changement dans l'ordre des états de base.

Une manière d'obtenir l'ordre des états de base est d'abord initialiser un vecteur de 2^n composantes avec l'ordre initial des états de base. On peut ensuite remodeler ce vecteur est un tenseur à n dimension de taille $2 \times 2 \times \dots \times 2$. Chaque dimension est associée à un qubit. L'échange des qubits revient alors à échanger les dimensions de ce tenseur. Avec `numpy` on peut changer l'ordre des dimensions à l'aide de la fonction `moveaxis`. *Attention*, parce qu'on utilise une convention petit-boutisme, la dernière dimension correspond au premier qubit. Une fois que cela est fait, on peut remodeler le tenseur en un vecteur de 2^n composantes qui devrait donner le nouvel ordre des états de base. Il ne reste qu'à permuter les lignes et les colonnes de la matrice de la porte quantique.

```
def reorder_qubits_gate(gate: NDArray, order: list[int]) -> NDArray:
    order = np.array(order)
    number_of_qubits = int(np.log2(gate.shape[0]))
    assert np.all(np.sort(order) == np.arange(number_of_qubits))

    from_qubit_order = np.arange(number_of_qubits)
    to_qubit_order = order

    from_dim_index = number_of_qubits - 1 - from_qubit_order
    to_dim_index = number_of_qubits - 1 - to_qubit_order

    basis_order = np.arange(2**number_of_qubits)
    tensor_shape = (2,) * number_of_qubits
    basis_reorder = ...
    reordered_gate = gate[basis_reorder, :][:, basis_reorder]
```

```
    return reordered_gate
```

Un exemple d'utilisation de cette fonction est de transformer la porte $C_0C_1\hat{X}_2$ en une porte contrôlée par les qubits 0 et 2 et appliquée sur le qubit 1 ($C_0C_2\hat{X}_1$).

```
c0x1c2_gate = reorder_qubits_gate(c0c1x2_gate, [0, 2, 1])
print(c0x1c2_gate)
```

```
[[1 0 0 0 0 0 0]
 [0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 0 0 0 0 1]
 [0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0]]
```

2.1.5 Ajout de qubits

Lorsqu'on veut appliquer des portes à un ou deux qubits sur des systèmes qui comportent plus de qubits, on doit ajouter des qubits sur lesquels aucune opération n'est effectuée. Construisez une fonction qui convertit une porte quantique à un plus grand nombre de qubits.

Cette fonction devrait prendre en entrée la porte originale `gate`, le nombre de qubits total `number_of_qubits` ainsi que sur quels qubits la porte originale devrait être appliquée `apply_on_qubits`.

Pour simplifier l'implémentation de cette fonction, vous pouvez commencer par ajouter les qubits supplémentaires à la suite de la porte originale et utiliser la fonction précédente `reorder_qubits_gate` pour utiliser l'argument `apply_on_qubits` et appliquer la porte sur les bons qubits.

```
def convert_to_more_qubits(gate: NDArray, number_of_qubits: int, apply_on_qubits: list[int]) -> NDArray:
    gate_number_of_qubits = int(np.log2(gate.shape[0]))
    assert len(apply_on_qubits) == gate_number_of_qubits
    assert number_of_qubits > gate_number_of_qubits

    qubits_to_add = number_of_qubits - gate_number_of_qubits
    i_gates = np.eye(2**qubits_to_add)

    tmp_gate = np.kron(i_gates, gate)
    ...
    return extend_gate
```

Par exemple, si vous voulez appliquer un $C\hat{X}$ depuis le qubit 0 vers le qubit 2, vous pourrez utiliser cette fonction comme ceci.

```
c0x2_gate = convert_to_more_qubits(c0x1_gate, 3, [0, 2])
print(c0x2_gate)
```

```
[[1 0 0 0 0 0 0]
 [0 0 0 0 1 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1]
 [0 0 0 0 1 0 0]]
```

```
[0 1 0 0 0 0 0]  
[0 0 0 0 0 1 0]  
[0 0 0 1 0 0 0]
```

2.1.6 Transformer un état quantique

Il est utile de pouvoir transformer un état quantique à l'aide d'une porte quantique. Cette fonction doit faire cela.

```
def apply_gate_on_state(gate: NDArray, state: NDArray) -> NDArray:  
    ...  
    return final_state
```

2.2 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.

```
def sample_state(state_vector: NDArray, shots: int) -> dict:  
    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 écrire cette fonction.

- 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` chaîne 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
```

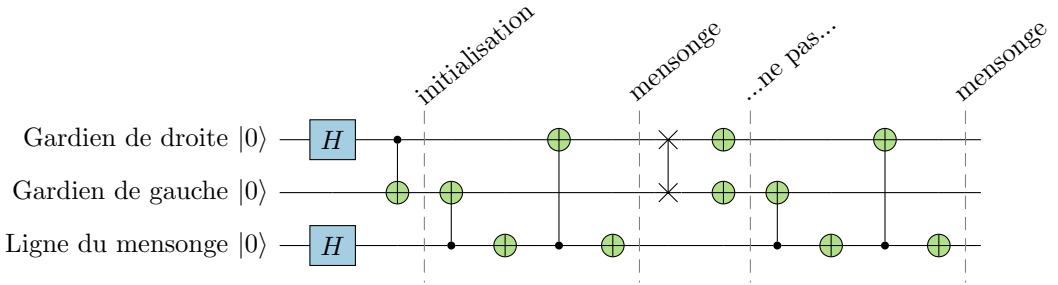


FIGURE 3 – Circuit quantique de l’énigme « La porte du trésor »

3 Les circuits quantiques

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.

3.1 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 sections). Le premier circuit comporte 4 sections (dont deux sont identiques) et le second seulement 2 sections.

Objectif 2 Développer des outils afin d’assembler la matrice qui représente l’application des circuits quantiques complets.

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

Objectif 4 Simuler des résultats de mesure à l’aide du module `random` de `numpy`. Consultez la section 2.2 pour des indications détaillées.

3.2 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 3.

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 Obtenir 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.



FIGURE 4 – Exemple de propriétés de commutation des portes NOT et CNOT.

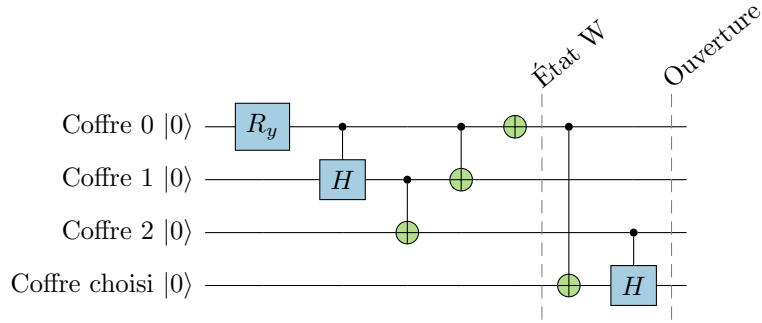


FIGURE 5 – Circuit quantique de l'énigme « Le problème de Monty Hall »

Objectif optionnel Utiliser des propriétés de commutation afin de simplifier au maximum ce circuit quantique. La figure 4, illustre des exemples de propriétés de commutation. Vous aurez à en trouver d'autres. Comparez les matrices obtenues et les vecteurs d'états finaux. Illustriez chacune des étapes de votre raisonnement.

3.3 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 5. En particulier, la première partie du circuit prépare un état quantique à trois qubits

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

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

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 Modifier 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.

4 Matrices et portes quantiques avec numpy

La librairie `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
```

4.1 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]])
```

4.2 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 6.

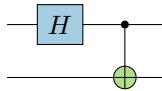


FIGURE 6 – Circuit de présentation d'état de Bell

4.2.1 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)
```

```
[[ 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.

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

4.2.2 Composition

On compose ensuite cette matrice avec la matrice du CNOT avec un produit matriciel. 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
```

4.3 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

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

5 Utilisation de `pytest`

Si vous avez installer votre module en suivant les étapes à la section 1.6 l'utilisation de `pytest` est assez simple. Si vous utilisez VSCode, dans la barre d'outil à gauche, cliquez sur la fiole. Configurez les tests en choisissant d'utiliser `pytest` et en spécifiant le dossier `tests/`. Si vous devez refaire la configuration, dans le menu Aide/Affichez toutes les commandes, saisissez Python : Configurer les tests.

Vous pouvez également lancer les tests manuellement dans un terminal. Par exemple, les tests pour les portes sont lancés avec la commande suivante.

```
pytest tests/test_gates.py
```

Au départ, parce que vos fonctions ne sont pas encore implémentées, tous les tests échoueront. Une fois que vos fonctions seront implémentées vous devriez passer tous les tests.

6 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. Assurez-vous d'avoir complété cette deuxième étape avant de remettre votre code !