

PÔLE PROJET
CENTRALESUPÉLEC - 1A

Guide d'informatique quantique

November 25, 2024



CentraleSupélec

Auteur :
Gaspard BERTHELIER

Contents

1	Introduction	3
2	Notions de base	4
2.1	Notations de Dirac	4
2.2	Définition des Cbits	5
2.3	Définition des Qbits	6
2.3.1	Définition	6
2.3.2	Règle de Born	6
2.3.3	Intrication	7
2.3.4	Simulation de Qbits sur qiskit	7
2.3.4.1	Création d'un circuit	8
2.3.4.2	Initialisation	9
2.3.4.3	Mesures avec qasm_simulator	9
2.3.4.4	Mesures avec statevector_simulator	11
2.3.4.5	Sphère de Bloch	11
2.3.4.6	Q-Sphère	13
2.3.4.7	Qbits quelconques	14
2.4	Opérations de base sur les Qbits	15
2.4.1	Opérateurs sur 1 bit	15
2.4.2	Opérateurs sur 2 bits	16
2.4.3	Identités intéressantes	16
2.4.4	Initialisation quelconque	17
2.4.5	Non localité	18
2.4.6	Réversibilité	18
2.4.7	Additionneur	19
3	Algorithmes simples pour commencer	20
3.1	Deutsch	21
3.1.1	Algorithme	21
3.1.2	Implémentation qiskit	21
3.2	Bernstein Varizani	24
3.2.1	Algorithme	24
3.2.2	Implémentation qiskit	24
3.3	Grover	26
3.3.1	Algorithme	26
3.3.2	Implémentation qiskit	27
3.4	Portes Toffoli	28
3.4.1	Algorithme	28
3.4.1.1	Avec 8 CNOTS	28
3.4.1.2	Avec 6 CNOTS	29
3.4.2	Implémentation qiskit	29
4	Approfondissement	30

5	Annexes	31
5.1	Espace de Hilbert	31
5.2	Espace dual	31
5.3	Convention d'Einstein	31
5.4	Produit tensoriel	32
5.4.1	Tenseurs	32
5.4.2	Produit tensoriel	32
5.4.3	Produit contracté	32
5.4.4	Produit de Kronecker	33
5.5	Règle de Born généralisée	34
5.6	Théorème de non-clonage	34
5.7	Matrices particulières	35
5.8	Matrices de Pauli	35
5.9	Addition modulo 2	35
5.10	Portes universelles	36
5.11	Suppléments sur les opérateurs quantiques	36
5.12	Construction d'un état quelconque	37
5.13	Additionneur	38
5.14	Deutsch	40
5.15	Bernstein Varizani	40
5.16	Grover	41

1 Introduction

Ce guide a pour vocation d'introduire l'informatique quantique à ceux qui souhaitent coder rapidement tout en comprenant la théorie de fond. L'idée est de présenter les fondements théoriques les plus importants et de les appliquer sur des algorithmes quantiques simples, de façon concise mais rigoureuse. Il est préférable d'avoir certaines bases en mathématiques (en particulier les espaces vectoriels), et de savoir coder en python pour le côté applicatif. Néanmoins, des annexes sont disponibles en fin d'ouvrage pour donner quelques rappels. Une asterisk sera présente à droite d'un mot-clé* qui se réfère à l'une des annexes.

La source principale pour le contenu de ce guide est l'ouvrage de N. David Mermin [1]. Les propos ont été simplifiés ou parfois clarifiés afin d'être rendus plus digestes, mais peuvent donc comporter quelques approximations et abus de notations. Néanmoins, en cas de réelles erreurs passées inaperçues, je vous prie de les rapporter à *gberthelmer.projet@gmail.com*.

Nous utiliserons pour les codes la bibliothèque qiskit. Il s'agit d'un framework python open source proposé par IBM [2].



2 Notions de base

2.1 Notations de Dirac

Nous travaillons sur des espaces de Hilbert*. Un vecteur d'un tel espace sera nommé un "ket" et se notera sous la forme : $|\Psi\rangle$. On suppose qu'il existe une base dénombrable de cet espace. On note alors $|x\rangle$ le $x^{i\grave{e}me}$ vecteur de la base, il est donc possible d'écrire Ψ sous la forme :

$$|\Psi\rangle = \sum_x c_x |x\rangle, \quad c_x \in \mathbb{C}$$

On appelle "bra" la forme linéaire : $\langle\Psi|$, qui correspond au dual* de $|\Psi\rangle$. Dans le cas d'un espace de dimension finie et en utilisant la notation vectorielle usuelle, on a $\langle\Psi| = |\Psi\rangle^{T*} = |\Psi\rangle^\dagger$. Il faut le voir comme un vecteur "renversé" et conjugué.

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}^\dagger = (x_1^*, x_2^*, x_3^*)$$

Le produit scalaire dans une base orthonormée est alors très visuel, puisque par dualité il s'agit de:

$$(\Phi, \Psi) = \Phi(\Psi) = \langle\Phi|\Psi\rangle = \text{"}\langle\Phi|\Psi\text{"} = \Phi^\dagger\Psi = (\Phi_1^*\Psi_1, \Phi_2^*\Psi_2, \Phi_3^*\Psi_3)$$

Les guillemets servent à indiquer une nouvelle notation. Pour un opérateur A (représenté par une matrice), on définit l'opérateur adjoint $A^\dagger = A^{T*}$ qui obéit aux propriétés suivantes :

- $(A^\dagger)^\dagger = A$
- $|A\Psi\rangle^\dagger = |\Psi\rangle^\dagger A^\dagger = \langle\Psi|A^\dagger = \text{"}\langle\Psi A^\dagger\text{"}$
- $A|\Psi\rangle = \text{"}|A\Psi\text{"}$ et $\langle A\Psi| = |A\Psi\rangle^\dagger$
- $\langle\Phi|A\Psi\rangle = \langle A^\dagger\Phi|\Psi\rangle = \langle\Phi A|\Psi\rangle = \text{"}\langle\Phi|A|\Psi\text{"}$

La dernière propriété se démontre facilement puisque dans une base orthonormée finie :

$$\langle\Phi|A\Psi\rangle = \Phi^\dagger A\Psi = (A^\dagger\Phi)^\dagger\Psi = \langle A^\dagger\Phi|\Psi\rangle$$

C'est en fait la définition de l'opérateur adjoint. Toutes les propriétés se démontrent de manière rigoureuses à partir de celle-ci. Pour plus de détails, se référer à la page : Opérateur adjoint

Enfin, on note $|\Psi\rangle\langle\Psi|$ l'opérateur projection sur Ψ :

$$|\Psi\rangle\langle\Psi| : |\Phi\rangle \rightarrow \langle\Psi|\Phi\rangle|\Psi\rangle$$

2.2 Définition des Cbits

En informatique quantique, on va considérer qu'un état quantique de base ne peut exister que sous deux états uniquement : $|0\rangle$ et $|1\rangle$. Peu importe le phénomène physique que peut représenter ces deux états : un courant on/off, l'état de spin d'une particule, etc ; tant que la sortie est binaire.

On décide de représenter ces deux états par des vecteurs colonnes :

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Ces deux états représentent en fait les deux états possibles en informatique classique. Dans notre formalisme quantique, les deux états sont orthogonaux dans un plan vectoriel de dimension 2. On les appelle des "Cbits" (pour bits classiques) de dimension 1.

On généralise ensuite à des Cbits de dimensions n , un n -Cbit correspondant à n instances de 1-Cbits. Par exemple, les états possibles en dimension 2 sont :

$$|0\rangle|0\rangle = "|00\rangle" \quad |01\rangle \quad |10\rangle \quad |11\rangle$$

On utilise ici la représentation binaire* pour exprimer le $p^{ième}$ vecteur, avec $p \in \llbracket 0, 2^n - 1 \rrbracket$. Son expression change selon le nombre de bits utilisés (la dimension n). Par exemple, pour écrire le nombre 2 (i.e le deuxième vecteur de base) sur 3 bits :

$$|2\rangle_3 = |010\rangle = |0\rangle |1\rangle |0\rangle \quad (\text{car } 2 = 0 * 2^2 + 1 * 2^1 + 0 * 2^0)$$

À ne pas confondre avec $|2\rangle_2 = |10\rangle$.

On désigne souvent par "bit de rang 0", "bit de poids faible" ou "premier bit" le bit tout à droite.

Par ailleurs, pour continuer à utiliser une représentation avec des vecteurs colonnes, on utilise un produit tensoriel particulier, le produit de Kronecker* :

$$|0\rangle |1\rangle |0\rangle = |0\rangle \otimes |1\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Le vecteur colonne qui correspond à $|m\rangle_n$ est de taille 2^n et le seul coefficient à 1 sera celui à la position $m+1$. On peut ainsi coder les entiers de 0 à $2^n - 1$. Notez qu'il s'agit d'une *représentation* des états (il en existe d'autres, sous forme de matrices carrés par exemple).

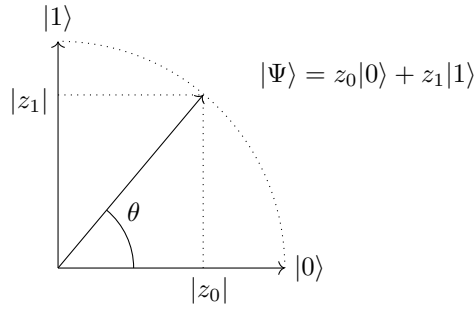
2.3 Définition des Qbits

2.3.1 Définition

Nous venons de voir comment définir vectoriellement et de manière unique les Cbits de dimension n . Maintenant, nous définissons les Qbits : un n -Qbit est un vecteur de norme 1 appartenant à l'espace vectoriel complexe engendré par les n -Cbits de dimension n . Il s'écrit donc de la forme :

$$|\Psi\rangle_n = \sum_{x=0}^{2^n-1} c_x |x\rangle, \quad \sum_x |c_x|^2 = 1$$

En dimension 1, on peut visualiser cela dans le plan 2D : un Qbit de dimension 1 sera un vecteur sur le cercle unité. Evidemment, les c_x étant complexes, on doit ajouter la phase pour avoir la description complète de l'état. Mais nous verrons dans le paragraphe suivant que ce sont en réalité seulement les $|c_x|$ qui nous intéressent, dans lesquels la phase n'intervient pas.



2.3.2 Règle de Born

Un Qbit représente une superposition d'états "classiques" incarnés par les Cbits. Ces Cbits seront appelés la "base computationnelle" ; ce sont des états dans lequel un système non quantique peut se trouver. Les amplitudes représentent alors la probabilité de *mesurer* un Qbit donné dans l'état correspondant (et non pas la probabilité qu'il *soit* dans cet état-là). Cette distinction très importante est au coeur de la mesure en physique quantique (pour plus de détails, se référer à la page : Problème de la mesure quantique).

Plus précisément, la probabilité de mesurer $|\Psi\rangle = \sum c_x |x\rangle$ dans l'état $|x\rangle$ sera $|c_x|^2$. C'est la règle de Born. On constate que puisque $|\Psi\rangle$ est normé, la somme des probabilités donne bien 1.

On suppose qu'il existe un moyen de mesurer un Qbit grâce à une porte "mesure" (notez que cette porte est nécessairement non inversible). Après la mesure dans un état donné, le Qbit restera dans cet état donné (tant qu'on n'applique pas d'autre opérateur). Il n'est plus en superposition, c'est la "décohérence" (Décohérence quantique). Retenez donc que mesurer un Qbit revient à le réinitialiser aléatoirement dans un état de la base computationnelle.

$$|\Psi\rangle_n = \sum c_x |x\rangle_n = \boxed{\text{mesure}} |x\rangle_n \quad p = |c_x|^2$$

La porte "mesure" du schéma précédent mesure n bits d'un coup. Mais il est aussi possible de mesurer un seul bit à la fois. Il faut dans ce cas recalculer les probabilités de mesure en factorisant certains états entre-eux :

$$|\Psi\rangle_{n+1} = \alpha_0|0\rangle|\Psi_0\rangle_n + \alpha_1|1\rangle|\Psi_1\rangle_n$$

La factorisation sépare les vecteurs commençant par 0 et ceux commençant par 1, ce qui correspond en probabilités à une "réunion d'événements disjoints". Par ailleurs, α_0 , α_1 , $|\Psi_0\rangle$ et $|\Psi_1\rangle$ sont construits de sorte à avoir $|\alpha_0|^2 + |\alpha_1|^2 = 1$ (détails en annexe)*. On a alors :

$$\alpha_0|0\rangle|\Psi_0\rangle_n + \alpha_1|1\rangle|\Psi_1\rangle_n \xrightarrow{\text{mesure}} |0\rangle|\Psi_0\rangle_n \quad p = |\alpha_0|^2$$

Comme expliqué plus tôt, l'un des intérêts des portes mesures sont qu'elles permettent d'initialiser un circuit. En effet, si l'on souhaite par exemple initialiser l'entrée par un $|0\rangle$, il suffit de mesurer un Qbit quelconque : si l'on mesure $|0\rangle$ on ne fait rien, si l'on mesure $|1\rangle$, on applique un NOT. On peut par ailleurs montrer qu'il est impossible de construire une porte universelle de clonage, c'est à dire qui recopie n'importe quel Qbit donné en entrée (démonstration en annexe*). On initialisera donc souvent un circuit en répétant le processus décrit précédemment.

2.3.3 Intrication

Nous allons prendre l'exemple de la dimension 2 pour donner l'intuition du phénomène, mais ce qui suit se généralise à toute dimension.

Un 2-Qbit s'écrit de la forme générale suivante :

$$|\Psi\rangle = c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle$$

Par ailleurs, le produit tensoriel de deux 1-Qbits donne un 2-Qbit :

$$|\Psi\rangle = (a_0|0\rangle + a_1|1\rangle) \otimes (b_0|0\rangle + b_1|1\rangle) = a_0b_0|00\rangle + a_0b_1|01\rangle + a_1b_0|10\rangle + a_1b_1|11\rangle$$

On voit que le produit tensoriel des 1-Qbits donne un sous-espace des 2-Qbits. On peut passer de la première forme à la seconde si et seulement si $c_{00}c_{11} = c_{01}c_{10}$ (la démonstration n'étant pas évidente, surtout dans le sens réciproque, nous ne l'incluons pas dans ce papier). Un vecteur d'état qui ne peut pas se factoriser en un produit tensoriel de 1-Qbits est dit "intriqué". Inversement, un état qui peut se factoriser est "séparable". Ces deux notions ont des conséquences majeurs sur lesquelles nous reviendrons.

2.3.4 Simulation de Qbits sur qiskit

Nous allons voir maintenant comment simuler le comportement de Qbits en Python. Il faut d'abord télécharger la bibliothèque qiskit et importer les modules principaux dans votre environnement :

```
#dans le shell
pip install qiskit
pip install qiskit[visualization]

#dans le script
from qiskit import QuantumCircuit, execute, Aer
```

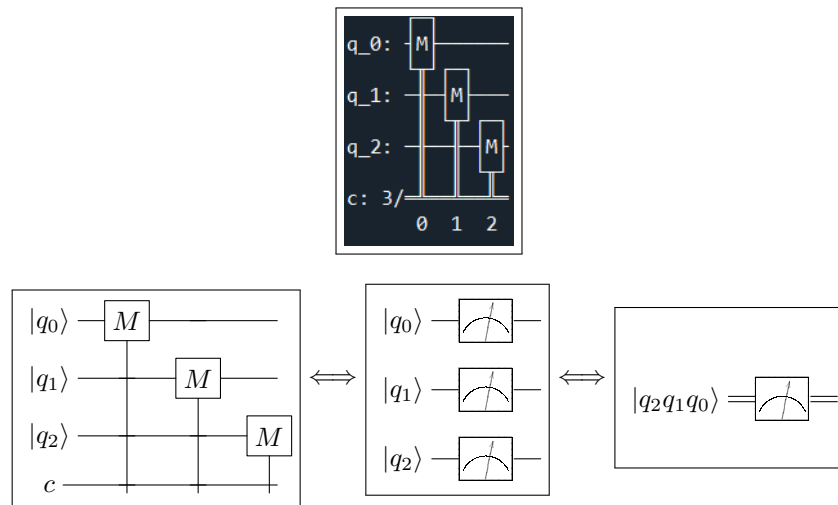
2.3.4.1 Création d'un circuit

Nous allons créer un circuit avec n Qbits. Lorsque l'on effectue des mesures sur ces Qbits, il faut inscrire le résultat dans des Cbits. La syntaxe est la suivante pour 3 Qbits :

```
n = 3
qc = QuantumCircuit(n,n) #n Qbits et n Cbits
for j in range(n):
    qc.measure(j,j) #ajout d'une mesure du jième Qbit vers le jième Cbit

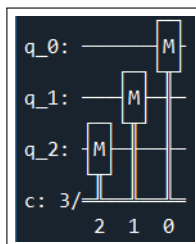
qc.draw() #Affiche le circuit construit
```

Vous voyez ci-dessous ce que renvoie le code précédent sur l'IDE Spyder (image supérieure). Par soucis de propreté, nous utiliserons différents schémas pour visualiser les circuits : les images ci-dessous sont schématiquement équivalentes.



Notez l'ordre des Qbits dans qiskit : le bit de poids faible est en haut selon l'axe vertical, et à gauche selon l'axe horizontal. Il est possible d'inverser l'ordre d'écriture des Cbits (pour respecter les conventions habituelles) en appliquant le bloc mesure au Qbit 3, puis au Qbit 2, puis au Qbit 1.

```
for j in range(n-1,-1,-1):
    qc.measure(j,j)
qc.draw()
```



2.3.4.2 Initialisation

Avec le code précédent, les Qbits sont initialisés à l'état $|0\rangle$. Il est cependant possible de les initialiser dans un autre état. Voici comment :

```
from math import sqrt

qc = QuantumCircuit(n,n)

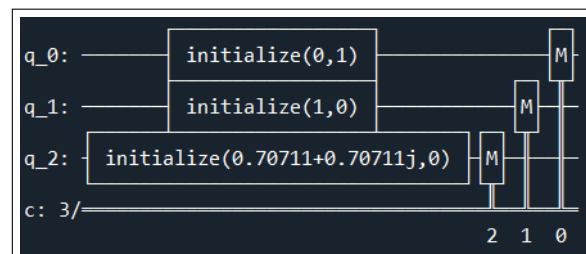
initial_state = []
initial_state_0 = [0,1] #état  $|1\rangle$ 
initial_state_1 = [1,0] #état  $|0\rangle$ 
initial_state_2 = [(1/sqrt(2))*(1+1j),0] #état de même probabilité que  $|0\rangle$ 

initial_state.append(initial_state_0)
initial_state.append(initial_state_1)
initial_state.append(initial_state_2)

for j in range(n):
    qc.initialize(initial_state[j], j) #initier le Qbit j à l'état initial_state_j

qc.measure(range(n-1,-1,-1),range(n-1,-1,-1)) #syntaxe + rapide et sens conventionnel
qc.draw()
```

Cela donne :



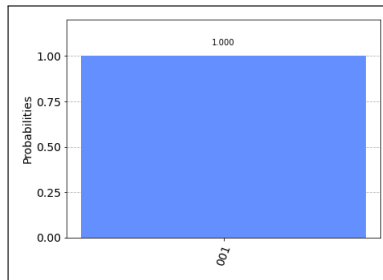
Notez qu'il a fallu réinitialiser le circuit. Si l'on avait initialisé les bits sur le circuit déjà existant, l'initialisation aurait été ajoutée après les blocs mesure d'avant.

2.3.4.3 Mesures avec qasm_simulator

Nous allons effectuer des mesures sur les Qbits grâce au calculateur "qasm_simulator" qui agit comme un vrai ordinateur quantique (c'est à dire qu'il suit la règle de Born). Ce calculateur répète l'opération un grand nombre de fois pour obtenir une fréquence d'apparition de chaque état et permet ainsi retrouver approximativement l'état du Qbit en question (l'initialisation est exactement la même à chaque répétition).

```
backend = Aer.get_backend('qasm_simulator')
counts = execute(qc, backend).result().get_counts()

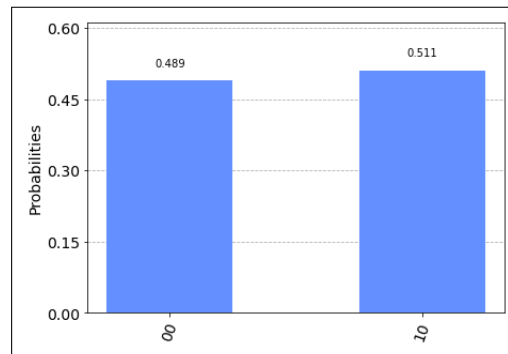
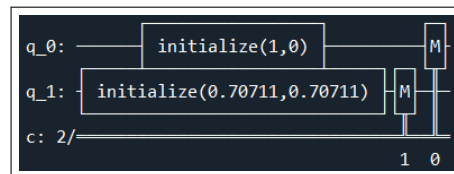
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```



On voit que le 3-Qbit est mesuré avec certitude dans l'état $|010\rangle$.
 Voyons ce qu'il se passe maintenant avec une superposition d'états :

```
qc = QuantumCircuit(2,2)
qc.initialize([1,0],0)
qc.initialize([1/sqrt(2),1/sqrt(2)],1)
qc.measure(1,1)
qc.measure(0,0)

counts = execute(qc, backend).result().get_counts()
plot_histogram(counts)
```



Puisque le bit de poids fort est dans une superposition entre les états $|0\rangle$ et $|1\rangle$, on s'attend à obtenir une probabilité de 0.5 pour les états $|00\rangle$ et $|10\rangle$. On constate néanmoins un léger décalage avec la fréquence théorique : le calculateur inclut un bruit non négligeable dans la simulation.

2.3.4.4 Mesures avec statevector_simulator

Un autre calculateur peut être utilisé : "statevector_simulator". Celui-ci n'agit pas comme un vrai ordinateur quantique car il renvoie le vrai vecteur d'état (théorique).

```
qc = QuantumCircuit(1) #juste un Qbit, pas besoin de Cbit
qc.initialize([0,1], 0)
backend = Aer.get_backend('statevector_simulator')
result = execute(qc,backend).result()
state = result.get_statevector()

print(state)
#Cela va renvoyer [0.+0.j 1.+0.j]

from qiskit_textbook.tools import array_to_latex
array_to_latex(state, pretext="\\text{Statevector} = ")
#Cette ligne de code ne fonctionne que sur un Jupyter notebook
#Cela renvoie le ket associé.
```

Remarquez que n'avons pas besoin d'ajouter un bloc mesure et donc pas de Cbits au circuit. On peut aussi utiliser une syntaxe différente :

```
from qiskit import assemble
qobj = assemble(qc) #Ceci est un nouveau type
state = backend.run(qobj).result().get_statevector()
print(state)
```

2.3.4.5 Sphère de Bloch

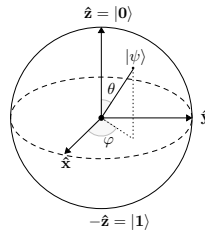
Dans cette partie, on introduit la sphère de Bloch qui est une autre façon de représenter des états quantiques à 1 Qbit. Puisque un état $|\Psi\rangle$ est de norme 1, il peut s'écrire sous la forme :

$$|\Psi\rangle = r_0 e^{i\gamma_0} |0\rangle + r_1 e^{i\gamma_1} |1\rangle, \quad r_0^2 + r_1^2 = 1$$
$$\Leftrightarrow |\Psi\rangle = e^{i\gamma} \left[\cos\left(\frac{\theta}{2}\right) |0\rangle + e^{i\varphi} \sin\left(\frac{\theta}{2}\right) |1\rangle \right]$$

où tous les coefficients sont des réels. La deuxième équation découle de la première par des factorisation sur les angles γ_i et θ correspond à (2 fois) l'argument de \vec{r} . On peut ignorer le facteur $e^{i\gamma}$ car il n'a pas d'effet observable (il n'influe pas sur les amplitudes). On peut donc écrire :

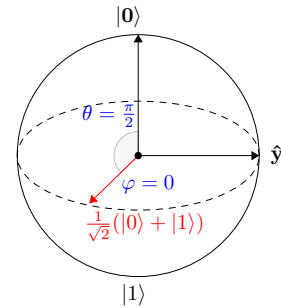
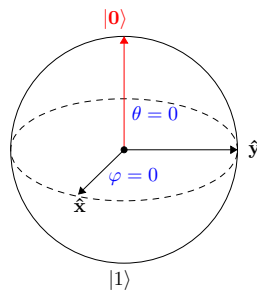
$$|\Psi\rangle = \cos\left(\frac{\theta}{2}\right) |0\rangle + e^{i\varphi} \sin\left(\frac{\theta}{2}\right) |1\rangle$$

On représente alors ces deux angles sur une sphère de Bloch :



Un état à n Qbits non intriqué peut alors se représenter avec n sphères de Bloch. Par exemple :

$$\Psi = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle) = |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$



Regardons comment tracer ces sphères avec qiskit :

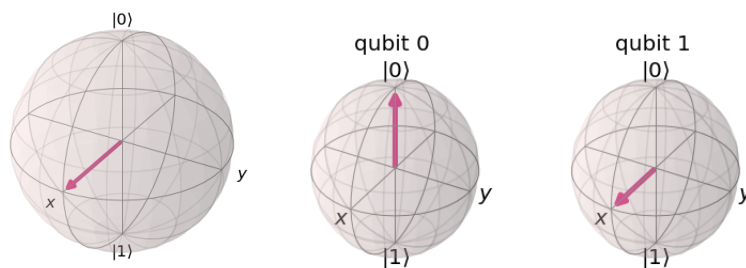
```
#Nous allons voir deux syntaxes possibles
from qiskit_textbook.widgets import plot_bloch_vector_spherical
from qiskit.visualization import plot_bloch_multivector
from math import pi

qc = QuantumCircuit(2,2)
qc.initialize([1/sqrt(2),1/sqrt(2)],1)
state = execute(qc,Aer.get_backend('statevector_simulator')).result().get_statevector()

#Ne peut tracer qu'un seul Qbit
coords = [pi/2,0,1] # [Theta, Phi, Rayon]
plot_bloch_vector_spherical(coords)

#Plusieurs Qbits
plot_bloch_multivector(state)
```

Voici les plots renvoyés dans l'ordre :



Pour la suite, nous n'utiliserons que la fonction "plot_bloch_multivector".

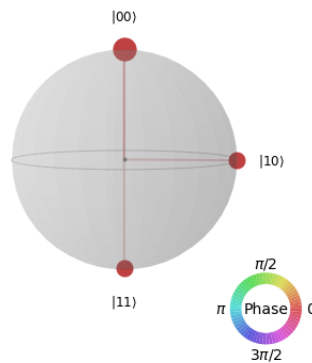
Notez que pour des états intriqués comme : $|\Psi\rangle = \frac{1}{\sqrt{2}}(|10\rangle + |01\rangle)$, il n'est malheureusement pas possible d'utiliser la sphère de Bloch. La partie suivante présente une dernière façon de représenter les états quantiques.

2.3.4.6 Q-Sphère

```
from qiskit.visualization import plot_state_qsphere
import numpy as np

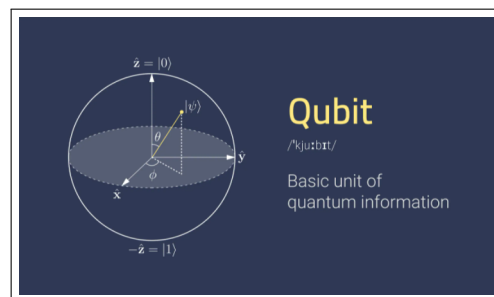
#réécriture de 1/sqrt(2) (00+01), même type que get_statevector
state = state = np.array([1,0,1/sqrt(2),1/sqrt(2)])

plot_state_qsphere(state)
```



La sphère est divisée en points équirépartis pour chaque vecteur de la base computationnelle. La taille du point est proportionnelle à son amplitude et sa couleur à la phase.

Il est alors possible de la tracer pour un vecteur d'état intriqué, tant qu'on arrive à lire son état avec `get_statevector`. Nous voyons page suivante comment construire une état éventuellement intriqué (nous n'avons pour l'instant vu sur qiskit que l'initialisation de Qbits non intriqués).

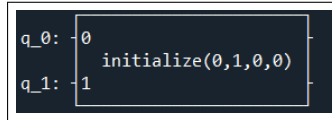


*Autre façon de dénommer les Qbits.
Source de l'image : shutterstock.com*

2.3.4.7 Qbits quelconques

Pour simuler un Qbit quelconque (c'est à dire de dimension plus grande que 1, sans forcément qu'il soit séparable), il est possible d'étendre l'initialisation sur plusieurs Qbits :

```
qc = QuantumCircuit(2)
qc.initialize([0,1,0,0], qc.qubits)
qc.draw()
```

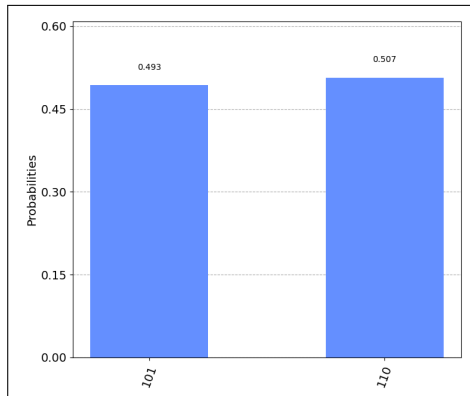
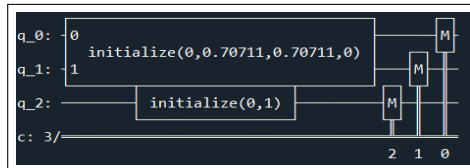


Dans le code précédent, chaque terme de la liste correspond à la composante de l'état de base associé. On produit ainsi $|01\rangle$. Voyons maintenant l'exemple d'un Qbit intriqué :

$$|\phi\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

Il s'agit bien d'un état valide puisque $c_{01}^2 + c_{10}^2 = 1$ et intriqué car $c_{00}c_{11} = 0 \neq c_{10}c_{01} = \frac{1}{2}$. On ajoute par ailleurs un dernier Qbit de sorte à avoir $|\phi\rangle = |1\rangle \otimes |\psi\rangle$.

```
qc = QuantumCircuit(3,3)
qc.initialize([0,1/sqrt(2),1/sqrt(2),0], qc.qubits[0:2])
qc.initialize([0,1],qc.qubits[2])
qc.measure([2,1,0],[2,1,0])
qc.draw()
counts = execute(qc, Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```



2.4 Opérations de base sur les Qbits

Nous allons maintenant présenter quelques opérateurs basiques que l'on peut appliquer sur les Qbits. Vous remarquerez que la plupart sont des fonctions agissant seulement sur la base computationnelle, étendues au Qbits par linéarité. Par ailleurs, on s'attend au minimum à ce que les opérateurs soient unitaires* afin que l'on reste dans l'espace des Qbits (ils sont donc au moins réversibles). Nous indiquerons comment implémenter ces opérateurs sur qiskit.

2.4.1 Opérateurs sur 1 bit

- Identité (noté $\mathbf{1}$) : $\mathbf{1}|0\rangle = |0\rangle$ et $\mathbf{1}|1\rangle = |1\rangle$ de matrice $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
- NOT (noté \mathbf{X}) : $\mathbf{X}|0\rangle = |1\rangle$ et $\mathbf{X}|1\rangle = |0\rangle$ de matrice $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Il s'agit ici des seuls opérateurs unitaires inversibles qui renvoient un vecteur de la base computationnelle vers un autre. Notez que les opérateurs projections (matrices avec un seul 1) ne sont pas unitaires, puisqu'on aurait par exemple : $\|P_{|0\rangle}|1\rangle\|^2 = \||0\rangle\|^2 = 0 \neq 1$

```
qc = QuantumCircuit(1)
qc.x(0) #porte NOT
```

Ensuite, il est possible de considérer des portes non classiques, dites "portes quantiques" qui prennent en compte les superposition d'états.

- Porte Hadamard :

$$\mathbf{H}|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \mathbf{H}|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

On peut ré-écrire ces deux équations en une seule :

$$\mathbf{H}|x\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^x|1\rangle) \quad \text{et sous forme matricielle:} \quad \mathbf{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

```
qc.h(0) #porte Hadamard
```

- Portes Y et Z :

$$\mathbf{Y} = \begin{pmatrix} 0 & -i \\ i & 1 \end{pmatrix} \quad \mathbf{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

On note $\sigma_x = X$, $\sigma_y = Y$, $\sigma_z = Z$. Ce sont les matrices de Pauli*, elles obéissent à des propriétés intéressantes données en annexe.

```
qc.y(0) #porte Y
qc.z(0) #porte Z
```

Nous mentionnerons seulement les propriétés suivantes :

$$\bullet \mathbf{X}^2 = \mathbf{Y}^2 = \mathbf{Z}^2 = \mathbf{H}^2 = \mathbf{1} \quad \bullet \mathbf{Y} = i\mathbf{XZ} \quad \bullet \mathbf{H} = \frac{1}{\sqrt{2}}(\mathbf{X} + \mathbf{Z})$$

2.4.2 Opérateurs sur 2 bits

Rappelons tout d'abord qu'on numérote les bits d'un Qbit de droite à gauche, à partir de 0. Le bit de plus à gauche est appelé "bit de poids fort".

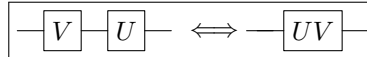
- SWAP : S_{ij} échange les bits de rang i et j . Par exemple : $S_{10}|xy\rangle = S_{01}|xy\rangle = |yx\rangle$
- CNOT : C_{ij} applique un NOT au bit de rang j si le bit de contrôle i est à 1.
Ainsi : $C_{10}|xy\rangle = |x\rangle|x \oplus y\rangle$ où $|x\rangle \oplus |y\rangle$ est addition modulo 2 (XOR)*.

(on peut montrer par ailleurs que $S_{ij} = C_{ij}C_{ji}C_{ij}$)

```
qc = QuantumCircuit(2)
qc.swap(0,1) #Porte SWAP
qc.cx(0,1) #Porte CNOT, bit de contrôle : 0
```

A ce stade, nous avons les premiers blocs élémentaires pour construire des circuits sur qiskit. En effet, la plupart des algorithmes quantique se baseront uniquement des portes à 1 ou 2 Qbits. Cela s'explique par les limites technologiques pour construire des portes d'ordre supérieur. Heureusement pour nous, il est possible de construire toutes les fonctions logiques uniquement à partir de portes d'ordre 1 ou 2, par complétude* de certains ensemble de ces portes. Si vous souhaitez découvrir de nouveaux opérateurs, n'hésitez pas à feuilleter la page wikipedia des opérateurs.

Notez que les opérateurs s'appliquent de droite à gauche sur les kets (c'est la composition matricielle), mais leur représentation est inversée sur les circuits.



Enfin, lorsque l'on applique des portes qu'à certains rangs d'un n-Qbit, il faut préciser le rang de ces Qbits en indice. Ainsi :

$$\mathbf{X}_0|11\rangle = |1\rangle \otimes \mathbf{X}|1\rangle = |10\rangle$$

Des opérateurs qui agissent sur des Qbits différents commutent. Ainsi :

$$\mathbf{X}_0\mathbf{X}_1\mathbf{X}_0|10\rangle = \mathbf{X}_1\mathbf{X}_0\mathbf{X}_0|10\rangle = \mathbf{X}_1|10\rangle = |00\rangle$$

Plus de détails sur ces opérateurs en annexe*.

2.4.3 Identités intéressantes

À partir des opérateurs défini précédemment, on peut construire des identités intéressantes qui serviront à simplifier les circuits. Pour vérifier ces identités, nous allons écrire une fonction qiskit qui permet de vérifier si deux circuits sont égaux.

```
from qiskit.quantum_info import Statevector
def compare(qc1,qc2):
    #compare si deux circuits sont équivalents
    return Statevector.from_instruction(qc1).equiv(Statevector.from_instruction(qc2))
```

Une première identité intéressante est la dualité entre **X** et **Z**, par la biais de **H**.

$$\mathbf{H}\mathbf{X}\mathbf{H} = \mathbf{Z} \quad \mathbf{H}\mathbf{Z}\mathbf{H} = \mathbf{X}$$

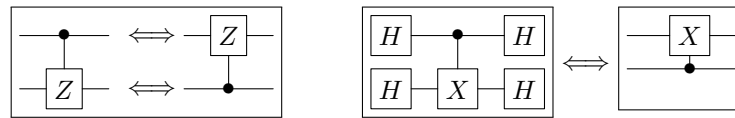
Cette identité se vérifie facilement par un calcul matriciel. Vérifions par le code.

```
qc1 = QuantumCircuit(1,1)
qc2 = QuantumCircuit(1,1)
qc1.z(0)
qc2.h(0)
qc2.x(0)
qc2.h(0)
compare(qc1,qc2)

qc1 = QuantumCircuit(1,1)
qc2 = QuantumCircuit(1,1)
qc1.x(0)
qc2.h(0)
qc2.z(0)
qc2.h(0)
compare(qc1,qc2)
```

On trouve que la fonction *compare* renvoie bien *True* dans les deux cas.

Ainsi, il est intéressant de simplifier des circuits en amont de leur implémentation. Par exemple, vérifiez que les deux circuits sont équivalents (le point relié indique une porte contrôlée, à l'instar d'un CNOT est qui est un **X** contrôlé) :



Le premier provient du fait qu'une porte contrôlée n'agit que si le bit de contrôle est à 1, et *Z* ne change l'état que si le bit cible est aussi à 1. Ainsi C_Z n'influe que sur $|11\rangle$ et ne fait qu'ajouter un terme (-1) , d'où la symétrie. On utilise ensuite cette propriété ainsi que $HXH = Z$ et $H^2 = 1$ pour le second circuit.

Nous avons maintenant les bases pour construire des circuits quantiques sur qiskit. Il y a encore quelques détails intéressants qui pourrons servir plus tard, et que nous expliquons dans les trois sous-parties suivantes. Si vous souhaitez commencer à coder dès maintenant, vous pouvez passer au premier algorithme que nous proposons : l'additionneur*.

2.4.4 Initialisation quelconque

On montre qu'il est possible d'initialiser n'importe quel état quantique superposé à 2 Qbits, à partir de l'état $|00\rangle$ et en utilisant ensuite seulement des portes d'ordre 1 et des CNOTS. Les calculs sont donnés en annexe*. Ce qu'il faut retenir, c'est qu'il existe toujours en théorie une succession simple de portes quantiques pour initialiser vos circuits dans un état souhaité. Cela justifie l'utilisation de *initialize* sur Qiskit.

2.4.5 Non localité

En admettant la propriété précédente, nous pouvons prendre un Qbit intriqué, par exemple l'état de Hardy : $|\Psi\rangle = \frac{1}{\sqrt{12}}(3|00\rangle + |01\rangle + |10\rangle - |11\rangle)$ qui aura été formé à partir de deux Qbits initiaux chacun à l'état $|0\rangle$. On suppose que Alice avait en sa possession le premier Qbit et Bob le second et qu'il est toujours possible de les mesurer séparément après les avoir intriqués.

On constate que chaque Qbit a une probabilité non nulle d'être mesuré à $|0\rangle$ ou $|1\rangle$. Maintenant, supposons que Alice et Bob s'écartent l'un de l'autre d'une très grande distance. Par ailleurs, chacun décide d'appliquer ou non une porte d'Hadamard à son Qbit avec une probabilité $\frac{1}{2}$. On peut alors lister selon l'action de chacun les états futurs possibles pour $|\Psi\rangle$.

Par exemple : $\mathbf{H}_b\mathbf{H}_a|\Psi\rangle = \mathbf{H}_1\mathbf{H}_0|\Psi\rangle = \frac{1}{\sqrt{3}}(|00\rangle + |01\rangle + |10\rangle)$ (à vérifier par le calcul)

H_a et H_b	$\dots + 0 11\rangle$
H_a et $\overline{H_b}$	$\dots + 0 01\rangle$
$\overline{H_a}$ et H_b	$\dots + 0 10\rangle$
$\overline{H_a}$ et $\overline{H_b}$	$\frac{1}{\sqrt{12}}(3 00\rangle + 01\rangle + 10\rangle + 11\rangle)$

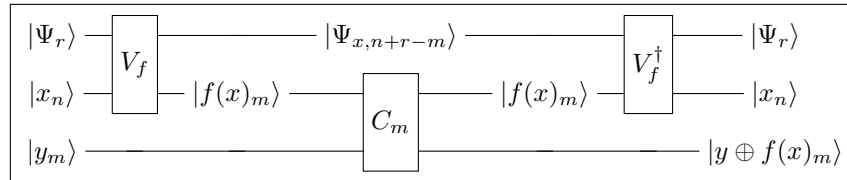
On constate que si l'un des deux au moins a appliqué une porte d'Hadamard, un des états devient impossible. Ainsi, si Alice applique effectivement une porte Hadamard à son Qbit, soit $|11\rangle$ soit $|01\rangle$ seront impossibles à mesurer, selon le choix de Bob. Le fait que Bob influe sur le Qbit d'Alice, à une distance infinie d'elle, est pour le moins surprenant. La réponse des physiciens à ce paradoxe est qu'une fois intriqué, $|\psi\rangle$ représente une entité unique, les deux Qbits de Alice et Bob ne sont alors plus dissociables (même si écartés).

2.4.6 Réversibilité

On souhaite que les opérations effectuées sur les Qbits soient toujours réversibles. Une façon de faire cela pour toute fonction quelconque f est de conserver l'entrée en mémoire dans la sortie. Si f agit sur un n -Qbit et renvoie un m -Qbit, on crée l'opérateur U_f de taille $n+m$:

$$U_f|xy\rangle = |x\rangle|y \oplus f(x)\rangle \quad \rightarrow \quad U_f|x\rangle|0\rangle = |x\rangle|f(x)\rangle$$

De cette manière, U_f est effectivement un opérateur réversible. Notez que l'implémentation d'une fonction complexe peut parfois nécessiter des Qbits supplémentaires pour effectuer les calculs et retenir en mémoire les données. On utilise alors r Qbits supplémentaires, initialement à 0, et qui reviennent à 0 à la fin du calcul.



Dans cette figure, C_m représente m CNOTS en parallèle sur chaque bit et V_f l'opérateur qui renvoie $f(x)$ ainsi que les "retenues" nécessaires pour pouvoir recalculer x .

2.4.7 Additionneur

Nous allons coder notre premier algorithme en prenant l'exemple d'un additionneur binaire. Un tel additionneur ajoute bit à bit les termes, en prenant en compte les retenues. Lors de l'addition de deux bits, on effectue une addition de type XOR*, et on conserve une retenue de 1 quand les deux bits sont à 1 (car $1 + 1 = 10$). Par exemple :

$$\begin{array}{r} 001 \\ 011 \\ \hline 100 \end{array}$$

Il faut a priori 12 bits pour cette addition : 6 pour les entrées, 3 pour la sortie, et 3 retenues.

Nous allons coder l'étape intermédiaire principale de l'additionneur : l'addition de deux bits. On s'attend à ce que la fonction n'agissent pas directement sur les deux bits en entrée, et renvoie deux sorties : le résultat de l'addition XOR et la retenue. On suppose l'existence d'une porte dite de Toffoli* qui agit sur 3 bits : si les deux premiers sont à 1, on applique NOT au troisième (c'est une CNOT à deux bits de contrôle, notée ccx sur qiskit).

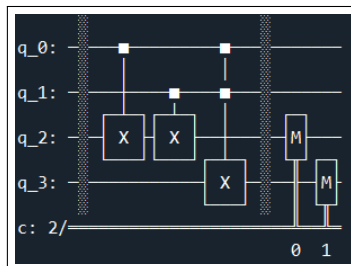
Essayez de trouver le code par vous-même.

Solution :

```
qc = QuantumCircuit(4,2)
qc.barrier() #pour la visibilité

#XOR
qc.cx(0,2)
qc.cx(1,2)
#Retenue
qc.ccx(0,1,3)

qc.barrier()
qc.measure(2,0)
qc.measure(3,1)
```



Il reste à assembler plusieurs fois cette fonction (en l'adaptant un peu) pour construire un additionneur pour plusieurs bits. Le code est donné en annexe*.

Cet algorithme illustratif aurait très bien pu s'implémenter sur un ordinateur classique. Nous allons dans la partie suivante voir des algorithmes spécifiques à l'informatique quantique.

3 Algorithmes simples pour commencer

Nous allons voir une succession d'algorithmes qui témoignent de l'intérêt de l'informatique quantique par rapport à l'informatique classique. Ces algorithmes simples permettront aussi de découvrir quelques astuces de calcul très pratiques.

Avant toute chose, on rappelle les notations particulières pour les opérateurs qui ne s'appliquent que sur certains Qbits : $\mathbf{A}_i|\Psi\rangle$ aura pour effet d'appliquer l'opérateur \mathbf{A} au $i^{\text{ème}}$ bit de $|\Psi\rangle$. Par exemple :

$$\mathbf{H}_1|000\rangle = (\mathbf{1} \otimes \mathbf{H} \otimes \mathbf{1})(|0\rangle \otimes |0\rangle \otimes |0\rangle) = |0\rangle \otimes \mathbf{H}|0\rangle \otimes |0\rangle$$

Notez que \mathbf{H}_1 est un opérateur de plus grande dimension que \mathbf{H} et dépend de la dimension considérée, par exemple en dimension 2 :

$$\mathbf{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad \mathbf{H}_0 = \mathbf{I} \otimes \mathbf{H} = \begin{pmatrix} H & (0) \\ (0) & H \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix}$$

Par ailleurs, quelques astuces de calculs qui pourront vous être utiles par la suite :

- Des opérateurs qui agissent sur des Qbits différents commutent : $\mathbf{A}_i\mathbf{A}_j = \mathbf{A}_j\mathbf{A}_i$
- Deux 1-Qbits orthogonaux $|\Phi\rangle$ et $|\Psi\rangle$ peuvent être obtenus par rotation unique à partir des états $|0\rangle$ et $|1\rangle$. Cet opérateur \mathbf{v} est unitaire : $|\Psi\rangle = \mathbf{v}|0\rangle$ et $|\Phi\rangle = \mathbf{v}|1\rangle$ avec $\mathbf{v}^\dagger\mathbf{v} = \mathbf{1}$ ie $\mathbf{v}^{-1} = \mathbf{v}^\dagger$.

D'autres astuces seront présentées dans les sections suivantes.

3.1 Deutsch

3.1.1 Algorithme

On s'intéresse aux fonctions qui agissent sur la base computationnelle à 1 dimension. Il en existe quatre différentes : $f_0 = \mathbf{I}$, $f_1 = \mathbf{X}$, $f_2 = 0$, $f_3 = 1$, les deux dernières renvoyant tout le temps respectivement 0 ou 1.

Si l'on a face à nous une boîte noire qui agit comme l'une des 4 fonctions, il faut de manière classique deux évaluations pour la connaître. Par exemple, $f(1) = 1$ permet de réduire les possibilités à f_0 et f_3 . Connaître ensuite $f(0) = 1$ permet de savoir avec certitude que $f = f_3$. De même, il faut deux évaluations si l'on veut savoir si f est constante ou non. Nous allons voir comment déterminer si f est constante ou non avec 1 seule opération quantique.

Pour cela, il suffit de considérer l'état suivant :

$$|\Psi\rangle = (H \otimes 1)U_f(H \otimes H)(X \otimes X)|00\rangle = \begin{cases} |1\rangle \frac{1}{\sqrt{2}}(|f(0)\rangle - |\overline{f(0)}\rangle) & \text{si } f(0) = f(1) \\ |0\rangle \frac{1}{\sqrt{2}}(|f(0)\rangle - |\overline{f(0)}\rangle) & \text{si } f(0) \neq f(1) \end{cases}$$

Le calcul est en annexe*. On constate donc qu'il peut se mettre sous la forme suivante :

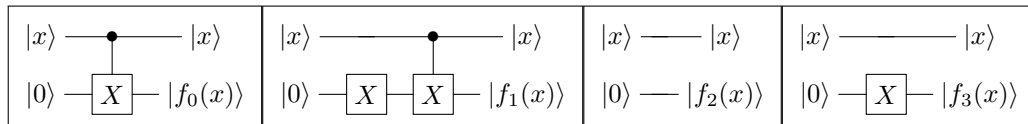
$$|\Psi\rangle = \begin{cases} |1\rangle|\Phi\rangle & \text{si } f(0) = f(1) \\ |0\rangle|\Phi\rangle & \text{si } f(0) \neq f(1) \end{cases}$$

Avec l'expression $|\Phi\rangle$ qui ne dépend pas du caractère constant de f . Ainsi, il suffit de mesurer le bit de poids fort de $|\Psi\rangle$ pour conclure si f est constante ou non. Cet algorithme est un premier exemple de calcul qui soit plus rapide en computation quantique qu'en computation classique : il effectue "plusieurs calculs en même temps", puisqu'il suffit d'une seule évaluation pour obtenir le résultat. Notez en revanche qu'il ne nous apporte pas plus d'information sur les valeurs de la fonction.

Un exemple d'application : on souhaite comparer la i ème décimale de $\sqrt{2}$ à celle de $\sqrt{3}$. Plutôt que de calculer les deux décimales, un algorithme similaire à celui de Deutsch permet de le savoir avec la même complexité que le calcul d'une seule des décimales. Le gain de temps est considérable si on imagine un i très grand ($\sqrt{2}$ étant irrationnel, on peut le prendre aussi grand que l'on veut). Notez cependant qu'en appliquant Deutsch, on apprend rien de plus sur la valeur de cette décimale.

3.1.2 Implémentation qiskit

Il faut d'abord coder les fonctions f_i . Il est possible de faire cela uniquement avec des portes NOT et CNOT. Essayez de trouver les schémas par vous-même puis vérifiez que vous trouvez les suivants :



Tentez ensuite de créer une fonction qui prend en valeur un entier entre 0 et 3 et construit le circuit de la fonction correspondante.

```
def fonction_mystere(k):
    qc = QuantumCircuit(2,2)
    if k == 0 :
        qc.cx(1,0)
    if k == 1 :
        qc.x(0)
        qc.cx(1,0)
    if k==3 :
        qc.x(0)
    return qc
```

On peut ensuite créer une fonction test qui évalue la valeur d'une fonction mystère.

```
def test(k_f,k_v):
    #k_f est l'entier qui commande la fonction mystère
    #k_v est la valeur à tester (0 ou 1)
    backend = Aer.get_backend('qasm_simulator')

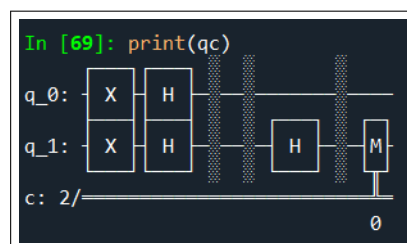
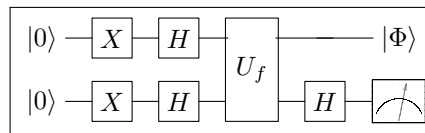
    qc = QuantumCircuit(2,2)

    if k_v == 1:
        qc.initialize([0,1],1)

    qcf = fonction_mystere(k_f)
    qcc = qc + qcf #on concatène les circuits
    #attention à l'ordre
    qcc.measure(1,1)
    qcc.measure(0,0)

    counts = execute(qcc, backend).result().get_counts()
    return(plot_histogram(counts))
```

On va enfin coder l'algorithme de Deutsch. Essayez de le faire vous même. Pour rappel, voici le circuit à implémenter :



Circuit attendu pour f_2

Solution :

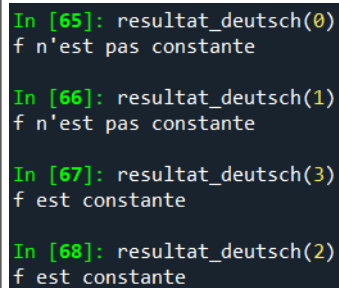
```
def deutsch(k_f):
    backend = Aer.get_backend('statevector_simulator')
    qc = QuantumCircuit(2,2)

    for k in range(2):
        qc.x(k)
        qc.h(k)
        qc.barrier()

    qcc = qc + fonction_mystere(k_f)
    qcc.barrier()

    qcc.h(1)
    qcc.barrier()
    qcc.measure(1,0)
    return qcc

def resultat_deutsch(k_f):
    qc = deutsch(k_f)
    backend = Aer.get_backend('qasm_simulator')
    counts = execute(qc,backend).result().get_counts()
    if '01' in counts.keys():
        print('f est constante')
    else :
        print("f n'est pas constante")
```



```
In [65]: resultat_deutsch(0)
f n'est pas constante

In [66]: resultat_deutsch(1)
f n'est pas constante

In [67]: resultat_deutsch(3)
f est constante

In [68]: resultat_deutsch(2)
f est constante
```

On retrouve bien les résultats attendus !

3.2 Bernstein Varizani

3.2.1 Algorithme

On s'intéresse aux fonctions de la forme :

$$f_a(x) = (a.x) = a_0x_0 \otimes \cdots \otimes a_nx_n, \quad x = \sum_{k=0}^{n-1} x_k 2^k$$

L'objectif est de déterminer a avec un minimum d'opérations possibles. L'algorithme naïf serait de déterminer $a_p = f(2^p)$. Mais on peut mieux faire avec de simples opérations quantiques.

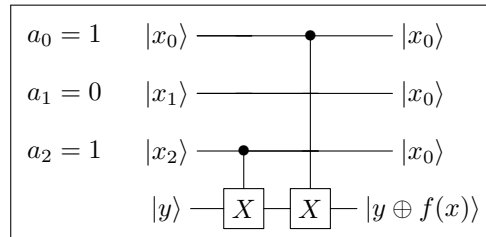
Il suffit de considérer l'état suivant (détails de calcul en annexe*) :

$$\mathbf{H}^{\otimes n+1} U_f \mathbf{H}^{\otimes n+1} |0\rangle_n |1\rangle_1 = |a\rangle_n |1\rangle_1$$

3.2.2 Implémentation qiskit

Il faut d'abord coder U_f . Puisque la sortie est une somme modulo 2, il suffit d'appliquer une porte NOT dès qu'une valeur $a_j x_j$ prend la valeur 1. Si $a_j = 0$, ce n'est jamais le cas. Si $a_j = 1$, c'est vrai dès que $x_j = 1$. Ainsi il suffit d'ajouter des portes CNOT entre x_j et la sortie quand $a_j = 1$.

Par exemple pour $a = 101$:



On peut alors créer une fonction qui renvoie un tel circuit en fonction de a . Tentez de le faire par vous même. Solution :

```
def fonction_bernstein(a):
    #on suppose que a est donné sous la forme d'une chaîne de caractère en binaire

    n = len(a)
    qc = QuantumCircuit(n+1,n)

    for k in range(n):
        if int(a[k])==1:
            qc.cx(k+1,0)

    return qc,n
```

On implémente ensuite l'algorithme de Bernstein Varizani. Tentez de le faire par vous même.

Solution :

```
a = "101"
(fct,n) = fonction_bernstein(a)

def Bernstein(a):

    qcf,n = fonction_bernstein(a)

    qc = QuantumCircuit(n+1,n)
    qc.initialize([0,1],0)
    qc.barrier()

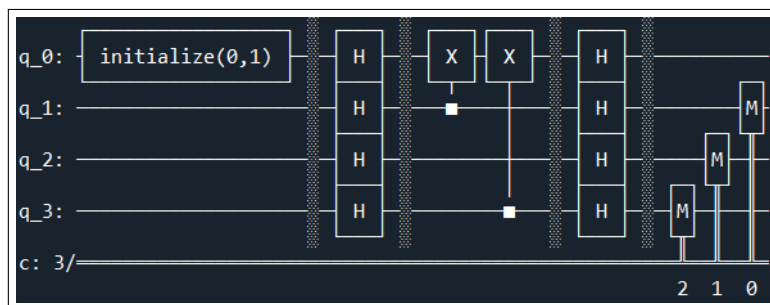
    for k in range(n+1):
        qc.h(k)
    qc.barrier()

    qcc = qc+qcf
    qcc.barrier()

    for k in range(n+1):
        qcc.h(k)
    qcc.barrier()

    for k in range(n+1,1,-1):
        qcc.measure(k-1,k-2)

    return qcc
```



```
In [44]: resultat_bernstein('101')
Out[44]: '101'
```

On obtient bien le résultat attendu !

3.3 Grover

3.3.1 Algorithme

On s'intéresse aux fonctions :

$$\begin{array}{ccc} f_a & : & \llbracket 0, 2^n - 1 \rrbracket \rightarrow \{0, 1\} \\ & & x \mapsto \delta_a(x) \end{array}$$

Ainsi : $f(x) = 1 \Leftrightarrow x = a$.

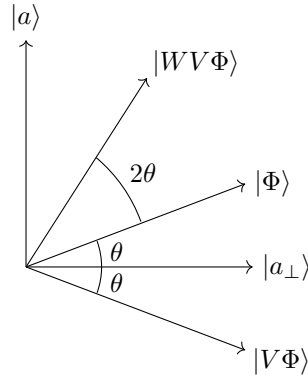
Le but est de trouver a sans avoir à calculer chaque $f(x)$ (complexité $O(2^n)$). On utilise alors l'état suivant :

$$|\Phi\rangle = H^{\otimes n} |0\rangle_n = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle$$

On va tenter construire de les opérateurs suivants :

$$V = I - 2|a\rangle\langle a| \quad W = 2|\Phi\rangle\langle\Phi| - I$$

V est une symétrie autour de $|a_\perp\rangle$ et W autour de $|\phi\rangle$. Schématiquement :



On remarque que l'on a : $\langle a|\Psi\rangle = \cos(\frac{\pi}{2} - \theta) = \frac{1}{2^{n/2}}$ car a est l'un des vecteurs de la base.

Donc plus n est grand, plus θ est faible et alors $\cos(\frac{\pi}{2} - \theta) = \sin(\theta) \approx \theta$. L'idée est alors d'appliquer $(WV)^m$ à $|\Phi\rangle$, c'est à dire m rotations de 2θ , de sorte à avoir $2m\theta \approx \frac{\pi}{2}$.

Pour cela, il faut $m = \frac{\pi}{4\theta} = 2^{n/2} \frac{\pi}{4}$ et alors : $(WV)^m |\Phi\rangle \approx |a\rangle$ (puisque θ est faible)

Il ne reste donc maintenant qu'à construire W et V avec n suffisamment grand.

Construction de V

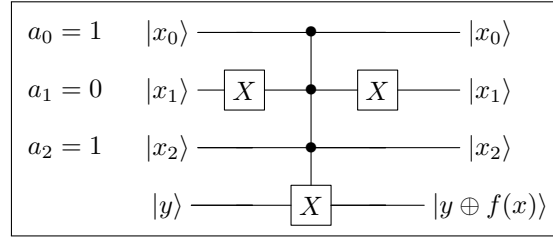
Notons $|+\rangle = \mathbf{H}|0\rangle$ et $|-\rangle = \mathbf{H}|1\rangle$, alors pour tout a :

$$U_f|x\rangle|+\rangle = |x\rangle|+\rangle \quad U_f|x\rangle|-\rangle = (-1)^{f(x)}|x\rangle|-\rangle$$

On choisit alors : $\tilde{V}|x\rangle = (-1)^{f(x)}|x\rangle$. Puisque $|a\rangle = \frac{n}{2}\langle\phi|a\rangle$, on a :

$$\tilde{V}|\phi\rangle = \frac{1}{n/2}(\sum_{x \neq a} |x\rangle - |a\rangle) = |\phi\rangle - \frac{2}{n/2}|a\rangle = (I - 2\langle a|a\rangle)|\phi\rangle = V|\phi\rangle$$

Ainsi, il suffit de savoir implémenter U_f pour obtenir le comportement de $V|\phi\rangle$. Pour construire U_f , on place des NOT à la place des 0 de a , et on applique une porte n-CNOT, où n est la dimension de a . Par exemple, pour $a = 101$:



La porte n-CNOT applique un NOT au bit cible si et seulement si tous les bits de contrôle sont à 1. La construction d'une telle porte est expliquée en annexe*.

Construction de W

Par des calculs donnés en annexe*, on peut exprimer W sous la forme :

$$W = -\mathbf{H}^{\otimes n} \mathbf{X}^{\otimes n} (\mathbf{c}^{n-1} \mathbf{Z}) \mathbf{X}^{\otimes n} \mathbf{H}^{\otimes n}$$

La construction d'une porte $\mathbf{c}^{n-1} \mathbf{Z}$ (porte \mathbf{Z} contrôlée par $n - 1$ Qbits) est similaire à celle des multiple CNOTS. Un schéma explicatif est donné en annexe*.

3.3.2 Implémentation qiskit

Pour construire le circuit, il peut être intéressant d'utiliser les portes déjà construites en qiskit. Par exemple, la porte MCMT (multi-controlled multi-target gate) : MCMT documentation.

3.4 Portes Toffoli

3.4.1 Algorithme

L'objectif est de coder l'opérateur suivant :

$$\mathbf{T}|xyz\rangle = |x\rangle|y\rangle|z \otimes xy\rangle$$

Cela équivaut à un NOT appliqué à z si et seulement si x et y sont à 1, donc un "CCNOT". Cette porte permet entre autres de construire facilement une porte AND, puisque on a :

$$\mathbf{T}|x\rangle|y\rangle|0\rangle = |x\rangle|y\rangle|xy\rangle$$

Remarquez que cette porte est réversible, alors qu'un AND usuel ne l'est pas. Cette porte est donc une porte universelle (grâce au système universel formé par les portes NOT et AND), et permet donc de construire toutes les opérations réversibles qui nous intéressent.

Nous allons voir deux méthodes pour construire des Toffoli avec des portes à 1 et 2 Qbits.

3.4.1.1 Avec 8 CNOTS

L'idée est de construire la porte :

$$T = C^{\sqrt{X}^2}$$

Qui utilise les matrices suivantes :

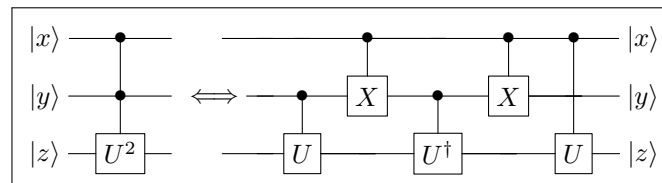
$$\sqrt{X} = H\sqrt{Z}H, \quad \sqrt{Z} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

Sachant le comportement pour un opérateur quelconque U :

$$C^{U^2}|xyz\rangle = U^{2xy}|xyz\rangle$$

Ainsi cet opérateur applique U^2 à z si et seulement si x et y sont à 1. On l'appelle "double-controlled- U^2 " ou juste CC- U^2 . Pour pouvoir construire un CC- U^2 , on se sert de portes c-U:

$$C^{U^2} = C_{10}^U C_{21} C_{10}^{U^\dagger} C_{21} C_{20}^U$$



Cherchez par vous même à comprendre pourquoi ces deux circuits sont équivalents (en testant les différentes conditions initiales possibles). Cette façon de construire des portes à 3-Qbits avec des portes à 2-Qbits est très commune. Pour la porte Toffoli, on remplace U par $H\sqrt{Z}H$.

Les H intérieurs vont se simplifier si bien qu'on se retrouve à simplement devoir implémenter $T = H_0 C^{\sqrt{Z}^2} H_0$. Or Mermin [1] (annexe B) montre en fait qu'on peut construire n'importe quelle porte c-U car il existe forcément des opérateurs unitaires V et W tels que $U = (VXV^\dagger)(WXW^\dagger)$ si bien que $C_{10}^U = V_0 C_{10} V_0^\dagger W_0 C_{10} W_0^\dagger$.

3.4.1.2 Avec 6 CNOTS

L'idée est de construire la porte :

$$T = C_{21}^U C^{(BA)^2}$$

Avec en particulier :

$$U = e^{-i\frac{\pi}{2}\mathbf{n}} = \begin{pmatrix} 0 & 0 \\ 0 & e^{-i\pi} \end{pmatrix} \quad \mathbf{n} = |1\rangle\langle 1|$$

$$\text{et} \quad A^2 = B^2 = 1 \quad A^\dagger = A \quad B^\dagger = B$$

$$\text{tels que} \quad (BA)^2 = iX$$

Alors :

$$C^{(BA)^2} = C_{10}^B C_{20}^A C_{10}^B C_{20}^A = C^{iX}$$

Ainsi, le rôle de U sera d'apporter un déphasage de $-i$ quand x est à 1. Or on aura justement un déphasage de i en cas d'application de la porte doublement contrôlée (c'est à dire quand $xy = 1$).

La construction de A et B est analogue à celle de W et Z précédemment.

3.4.2 Implémentation qiskit

On utilisera en pratique l'implémentation qiskit de la porte de Toffoli, notée "ccx".

On peut alors simuler tout un tas de portes logiques grâce à cette porte universelle.

```
#AND : renvoie 1 ssi x=y=1
def porte_and(qc,x,y,z):
    #x et y le rang des entrées, z la sortie (doit être initialisée à 0)
    qc.ccx(x,y,z)
    return qc

#NAND : renvoie 0 ssi x=y=1
def porte_nand(qc,x,y,z):
    #x et y le rang des entrées, z la sortie (doit être initialisée à 0)
    qc.x(z) #pour renvoyer z=1+xy
    qc = porte_and(qc,x,y,z)
    return qc

#OR : renvoie 0 ssi x=y=0 ie ssi Xx=Xy=1
def porte_or(qc,x,y,z):
    #x et y le rang des entrées, z la sortie (doit être initialisée à 0)
    qc.x(x)
    qc.x(y)
    qc = porte_nand(qc,x,y,z)
    qc.x(x)
    qc.x(y)
    return qc
```

Evidemment, nous allons aussi l'utiliser dans des circuits purement quantique. L'objectif de l'informatique quantique n'est pas de reformuler tous les circuits classiques, mais d'en trouver des nouveaux plus efficaces.

4 Approfondissement

Dans cette partie, nous présentons une liste de différents algorithmes ou domaines permettant d'approfondir ses connaissances en informatique quantique.

- QFT : Quantum Fourier Transform
Il s'agit d'une analogie de la transformation de fourier discrète avec des Qbits.
- QPE : Quantum Phase Estimation.
Permet d'estimer la phase de vecteurs propres.
- Algorithme de Shor : mis au point par Peter Shor en 1994.
Factorise un entier naturel de manière très efficace, permettant ainsi de briser théoriquement le système cryptographique RSA.
- QAOA : Quantum Approximate Optimization Algorithm
Domaine de l'optimization combinatoire.
- MPS : Matrix Product States
Factorisation de tenseurs.
- Machine Learning
Encoder et traiter des datasets sur des Qbits
- Code d'erreurs
Codes de détection d'erreurs lors de la transmission de Qbits.

Un futur guide faisant suite à ce présent ouvrage proposera une introduction plus approfondie sur chacun de ces thèmes. En attendant, vous avez maintenant les bases nécessaires pour comprendre de vous mêmes des articles et ouvrages sur ces divers sujets.

Je vous souhaite bonne chance pour la suite de votre aventure dans le monde de l'informatique quantique !

5 Annexes

5.1 Espace de Hilbert

Un espace de Hilbert H est un espace vectoriel complexe muni d'un produit scalaire hermitien, qui est de plus complet. Le produit scalaire hermitien $(x, y) \mapsto \langle x|y \rangle \in \mathbb{C}$ est défini sur $H \times H$ par les propriétés suivantes ($x, y, z \in H$) :

- $\langle y|x \rangle = \overline{\langle x|y \rangle}$ (forme hermitienne)
- $\forall a \in \mathbb{C}, \langle x|ay + z \rangle = a\langle x|y \rangle + \langle x|z \rangle$ (linéaire à droite)
- $\forall a \in \mathbb{C}, \langle ax + y|z \rangle = \overline{a}\langle x|z \rangle + \langle y|z \rangle$ (semi-linéaire à gauche)
cela donne avec la linéarité à droite une "forme sesquilinéaire à gauche"
- $\forall x \neq 0, \langle x|x \rangle > 0$ (forme définie)

Un espace complet est un espace métrique dans lequel toute suite de Cauchy converge. Pour davantage de détails sur ce point, se référer à la page suivante : Espace complet.

Revenir au cours*

5.2 Espace dual

L'espace dual d'un espace vectoriel E est l'espace vectoriel E^* qui contient l'ensemble des formes linéaires sur E . Plus précisément, il s'agit du dual "algébrique". Le dual topologique E' est l'ensemble des formes linéaires continues (qui coïncide avec E^* en dimension finie). Selon le théorème de Riesz, si H est un espace de Hilbert :

$$\forall u \in H', \exists! a \in H, \forall x \in H, u(x) = \langle a|x \rangle$$

Dans un espace vectoriel réel, il est courant de confondre la forme linéaire u et son représentant a .

Revenir au cours*.

5.3 Convention d'Einstein

Cette convention d'écriture est très utilisée dans la littérature scientifique, et est très pratique pour réaliser des calculs tensoriels.

L'idée est d'écrire $v = v^i e_i$ à la place de $v = \sum_i v^i e_i$.

L'indice à sommer peut être n'importe quel symbole, tant que celui-ci est répété deux fois. Il n'y a pas de règle sur lequel des deux symboles doit être en exposant et en indice, d'ailleurs on peut très bien mettre deux indices ou deux exposants. Cela n'a d'importance que lorsque nous souhaitons différencier coordonnées de vecteurs et coordonnées de leur duals. Se référer à : Tenseurs

On a par exemple: $a^k u_p^k e_p = \sum_i a^i \sum_k u_k^i e_k$ et $(Ax)_i = A_{ij} x_j$

5.4 Produit tensoriel

5.4.1 Tenseurs

Pour définir les tenseurs, on utilisera la convention d'Einstein. Ici s'agit d'une convention d'écriture décrite dans l'annexe ici*. Un tenseur est la généralisation des matrices à des "dimensions supérieures", dans le sens où un vecteur colonne est de dimension 1 et une matrice de dimension 2. La dimension correspond ainsi au nombre d'indices nécessaires pour caractériser le tenseur.

Si on note $E^p = E \times E \times \dots E$ le produit cartésien de p fois E, respectivement E^{*q} pour le dual, un tenseur est une forme multilinéaire sur $E^p \times E^{*q}$.

Ainsi en dimension 3 sur $E \times E \times E^*$, un tenseur est de la forme :

$$T(x^i e_i, y^j e_j, z_k e^k) = x^i y^j z_k T(e_i, e_j, e^k) = x^i y^j z_k T_{ij}^k$$

Les indices supérieurs et inférieurs servent à différencier vecteurs de E et de E^* , même si on confond en général E et E^* . L'ensemble des tenseurs sur E^p est noté $\otimes^p E$, et est de dimension n^p . Notez que $\otimes^0 E = K$ (scalaire qui ne dépend pas d'une base) et $\otimes^1 E = E$ (vecteurs).

5.4.2 Produit tensoriel

Pour la suite, nous considérerons à titre illustratif un tenseur P de E^3 et un tenseur Q de E^2 .

Le produit tensoriel de deux tenseurs renvoie un nouveau tenseur dont la dimension est la somme des deux autres :

$$P \otimes Q : \begin{array}{ccc} \otimes^p E \times \otimes^q E & \longrightarrow & \otimes^{p+q} E \\ (x, y, a, b, c) & \longmapsto & P(x, y)Q(a, b, c) \end{array}$$

On comprend donc mieux la notation de $\otimes^p E$ qui est en fait relié à l'espace vectoriel engendré par la base de tenseurs de taille p : $(e_i \otimes e_j \cdots \otimes e_k)_{1 \leq i, j, \dots, k \leq n}$

Ainsi si on considère deux vecteur a et b, $a \otimes b$ est un tenseur d'ordre 2 et :

$$a \otimes b(x, y) = a(x)b(y) = \langle a|x \rangle \langle b|y \rangle$$

5.4.3 Produit contracté

Cette opérations met en commun deux indices des tenseurs :

$$P \overline{\otimes} Q : \otimes^p E \times \otimes^q E \longrightarrow \otimes^{p+q-2} E$$

Par exemple :

$$(P \overline{\otimes} Q)_{ijm} = P_{ijk} Q_{km}$$

Entre autres, on reconnaît en dimension 1 et 2 les produits usuels auxquels on a déjà l'habitude :

$$\begin{aligned} a \overline{\otimes} b &= a_k b_k = \langle a|b \rangle \\ (A \overline{\otimes} b)_i &= A_{ik} b_k = (Ab)_i \\ (A \overline{\otimes} B)_{ij} &= A_{ik} B_{kj} = (AB)_{ij} \end{aligned}$$

5.4.4 Produit de Kronecker

Il s'agit d'une façon de représenter un produit tensoriel. Si vous ne savez pas ce qu'est un tenseur, vous pouvez vous référer à cette annexe : Produit tensoriel*.

Dans le cadre de l'informatique quantique, nous utilisons le produit de Kronecker pour représenter les états composés (d'où la notation avec les \otimes). Pour A et B de taille quelconque, obtenir $A \otimes B$ revient à multiplier chaque coefficient de A par la matrice B . Visuellement, chaque coefficient de A est donc remplacé par un élément de la dimension de B . La dimension de la matrice $A \otimes B$ est alors le produit des dimensions :

$$A \otimes B = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n,1} & \cdots & \cdots & a_{n,m} \end{pmatrix} \otimes \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,q} \\ b_{2,1} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ b_{p,1} & \cdots & \cdots & b_{p,q} \end{pmatrix} = \begin{pmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,m}B \\ a_{2,1}B & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ a_{n,1}B & \cdots & \cdots & a_{n,m}B \end{pmatrix}$$

Développons simplement le cas de matrices 2x2 :

$$A \otimes B = \begin{pmatrix} a_{1,1} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} & a_{1,2} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} \\ a_{2,1} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} & a_{2,2} \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & a_{1,2}b_{1,1} & a_{1,2}b_{1,2} \\ a_{1,1}b_{2,1} & a_{1,1}b_{2,2} & a_{1,2}b_{2,1} & a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & a_{2,2}b_{1,1} & a_{2,2}b_{1,2} \\ a_{2,1}b_{2,1} & a_{2,1}b_{2,2} & a_{2,2}b_{2,1} & a_{2,2}b_{2,2} \end{pmatrix}$$

Par ailleurs, notez la différence entre les produits suivants :

$$\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix} \quad \begin{pmatrix} a \\ b \end{pmatrix} \otimes (c \ d) = \begin{pmatrix} ac & ad \\ bc & bd \end{pmatrix}$$

En général, c'est le deuxième produit qui est utilisé en physique, mais c'est le premier qui nous intéressera en informatique quantique.

On a alors les propriétés suivantes :

- $(A \otimes B)(x \otimes y) = (Ax \otimes By)$
- $\det(A \otimes B) = \det(A)^m \det(B)^n$
- $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$
- $(A \otimes B)^T = A^T \otimes B^T$
- $\text{Tr}(A \otimes B) = \text{Tr}(A)\text{Tr}(B)$
- $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$

Revenir au cours*

5.5 Règle de Born généralisée

On a dans le cas d'une mesure sur un bit :

$$|\Psi\rangle_{n+1} = \alpha_0|0\rangle|\Psi_0\rangle_n + \alpha_1|1\rangle|\Psi_1\rangle_n$$

$$|\Psi_0\rangle = \frac{1}{\alpha_0} \sum_{x=0}^{2^n-1} c_x |x\rangle \quad |\Psi_1\rangle = \frac{1}{\alpha_1} \sum_{x=0}^{2^n-1} c_{x+2^n} |x\rangle$$

Notez le décalage d'indice sur les amplitudes pour $|\Psi_1\rangle$, dû au fait que ce sont vecteurs qui commencent par 1 et qui viennent donc *après* tout ceux ayant commencé par 0 (dans l'ordre de la représentation binaire). Par ailleurs :

$$\alpha_0 = \sqrt{\sum_{x=0}^{2^n-1} |c_x|^2} \quad \alpha_1 = \sqrt{\sum_{x=0}^{2^n-1} |c_{x+2^n}|^2}$$

On vérifie bien que la somme des carrés donne 1. Enfin, il est possible de considérer des portes qui mesurent m Qbits à la fois, en effectuant des factorisation similaires (disjonction en 4 cas si on veut mesurer les 2 bits de points fort, etc.). Les probabilités sont les mêmes peu importe le nombre et l'ordre dans lequel on mesure les Qbits. Il ne s'agit que de l'application de probabilités conditionnelles. Pour une démonstration complète de ces cas, se référer à l'ouvrage de Mermin [1].

Revenir au cours*

5.6 Théorème de non-clonage

Montrons qu'il n'existe pas de porte capable de cloner n'importe quel Qbit. Soit un opérateur unitaire U qui prend entrée deux Qbits : $|\psi\rangle|0\rangle$ et donne en sortie $|\psi\rangle|\psi\rangle$. Cela doit fonctionner pour n'importe quelle entrée, supposons que ça soit le cas pour $|\psi\rangle$ et $|\phi\rangle$:

$$U|\psi\rangle|0\rangle = |\psi\rangle|\psi\rangle \quad U|\phi\rangle|0\rangle = |\phi\rangle|\phi\rangle$$

On aurait alors :

$$\begin{aligned} \langle\psi|\langle\psi||\phi\rangle|\phi\rangle &= \langle 0|\langle\psi|U^T U|\phi\rangle|0\rangle \\ \Leftrightarrow \langle\psi|\phi\rangle^2 &= \langle\psi|\phi\rangle \quad (U^T = I \text{ et } \langle 0|0\rangle = 1) \\ \Leftrightarrow \langle\psi|\phi\rangle(\langle\psi|\phi\rangle - 1) &= 0 \end{aligned}$$

Cela signifierait que pour deux états quelconques, ceux-ci seraient toujours soit orthogonaux soit égaux (car ils sont normés), ce qui est impossible, ainsi U n'existe pas.

Revenir au cours*

5.7 Matrices particulières

Revenir au cours*

On se place dans un ensemble de matrices à coefficients complexes : $M_n(\mathbb{C})$

- Matrices normale : $A^\dagger A = AA^\dagger$
- Matrices unitaire : $A^\dagger A = AA^\dagger = I_n$, forme le groupe unitaire $U(n)$
Constatez qu'une matrice unitaire est normale, et dans le cas réel orthogonale.
- Matrice hermitienne (auto-adjointe) $A^\dagger = A$
Une matrice hermitienne est normale, et est unitaire si et seulement si $A^2 = I_n$.

L'intérêt des matrices unitaires est qu'elles représentent des isométries vectorielles (qui conservent les angles et la norme), en effet : $\langle \Phi | \Psi \rangle = \langle \Phi | U^\dagger U | \Psi \rangle = \langle U \Phi | U \Psi \rangle$. On montre que cela équivaut à $\|U\psi\| = \|\psi\|$ (voir démonstrations isométrie vectorielles et automorphismes orthogonaux).

5.8 Matrices de Pauli

Les matrices de Pauli sont les suivantes : $\sigma_X = \mathbf{X}$, $\sigma_Y = \mathbf{Y}$, $\sigma_Z = \mathbf{Z}$

Ces matrices vérifient des propriétés intéressantes, par exemple :

- $\sigma_X^2 = \sigma_Y^2 = \sigma_Z^2 = \mathbf{1}$
- $\sigma_X \sigma_Y = -\sigma_Y \sigma_X = i\sigma_Z$ (+ les permutations circulaires)

Ces identités équivalent à une seule formule vectorielle en notant $\sigma = (\sigma_X, \sigma_Y, \sigma_Z)^T$:

$$\forall a, b \in \mathbb{R}_3, \quad (a \cdot \sigma)(b \cdot \sigma) = (a \cdot b)\mathbf{1} + i(a \times b) \cdot \sigma$$

On montre que $(\mathbf{1}, \sigma_X, \sigma_Y, \sigma_Z)$ est une base de l'espace des opérateurs hermitiens (notons le \mathbb{H}):

$$\forall \mathbf{A} \in \mathbb{H}, \quad \mathbf{A} = a_0 \mathbf{1} + \vec{a} \cdot \sigma \quad \text{avec} \quad (a_0, a_1, a_2, a_3 \in \mathbb{R})$$

Cela provient du fait que tout $A \in \mathbb{H}$ est unitaire et de déterminant 1, et fait donc partie de $SU(2)$ (groupe spécial unitaire), engendré par la même base mais avec des coefficients dans \mathbb{C} . $SU(2)$ correspond aux matrices unitaires avec un déterminant de module 1, et englobe ainsi les opérateurs quantiques. On peut cependant les considérer comme égaux dans notre cas, car un simple déphasage permet de ramener le déterminant à 1 et n'influe pas sur les mesures.

5.9 Addition modulo 2

L'addition modulo 2 est définie ainsi :

$$1 \oplus 0 = 0 \oplus 1 = 1 \quad 0 \oplus 0 = 1 \oplus 1 = 0$$

C'est le résultat que renverrai une porte logique XOR.

On vérifie aisément les propriétés suivantes ($x \cdot y$ est un produit scalaire non défini) :

$$\begin{aligned} x \oplus 1 &= \bar{x} & x \oplus 0 &= x & x \oplus x &= 0 & x \oplus \bar{x} &= 1 & x \oplus y &= 0 \Leftrightarrow x = y \\ (-1)^x &= (-1)^{\sum_j x_j 2^j} = (-1)^{x_0} & (-1)^{x+y} &= (-1)^{x_0 \oplus y_0} & (-1)^{x \cdot y} &= (-1)^{x_0 y_0 \oplus \dots \oplus x_n y_n} \end{aligned}$$

5.10 Portes universelles

Revenir au cours*.

Pour qu'un ensemble de portes quantiques forme un ensemble de portes universelles, il faut que toute opération unitaire (ie. toute opération quantique) puisse être formée à partir d'une séquence finie de cet ensemble. En théorie, cela est impossible car le nombre d'opérateurs unitaires est non dénombrable, alors que l'ensemble des séquences finies d'un ensemble fini est dénombrable. Heureusement, nous n'avons besoin en pratique que d'approcher ces opérateurs, ce qui a été prouvé être possible de manière "efficace" (théorème de Solovay–Kitaev).

Exemples d'ensembles de portes universelles :

- à 2 Qbits : $(H, R_{\frac{\pi}{4}}, C_X)$ (Hadamard, rotation de $\frac{\pi}{4}$, CNOT)
- à 3 Qbits : porte de Toffoli (on montre comment la construire dans ce guide)

Cela montre que toute fonction logique classique est réalisable avec un ordinateur quantique, car la porte de Toffoli est une porte classique universelle (attention, cela n'est pour autant pas une porte quantique universelle, ce qui est impossible, mais elle permet d'en faire une bonne approximation).

5.11 Suppléments sur les opérateurs quantiques

On commence par introduire des opérateurs non réversibles qui serviront à construire nos opérateurs quantiques usuels.

- Projection sur $|1\rangle$ (noté \mathbf{n} ou $|1\rangle\langle 1|$) : $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$ • $\mathbf{Xn} : \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$
 Tel que $|1\rangle\langle 1||1\rangle = |1\rangle$ et $|1\rangle\langle 1||0\rangle = 0$ Tel que $\mathbf{Xn}|x\rangle = x|\bar{x}\rangle$
 ie. $\mathbf{n}|x\rangle = x|x\rangle$ ie. $\mathbf{Xn} = |0\rangle\langle 1|$
- Projection sur $|0\rangle$ (noté $\bar{\mathbf{n}}$ ou $|0\rangle\langle 0|$) : $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ • $\mathbf{X}\bar{\mathbf{n}} : \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} = |1\rangle\langle 0|$

On remarque qu'ajouter plusieurs de ces opérateurs de projection revient à effectuer un "ou" logique. Par exemple pour la porte CNOT, soit le bit contrôle vaut 0 auquel cas on ne fait rien, soit il vaut 1 auquel cas on effectue un NOT sur le bit target. Cela peut se résumer à :

$$C_{01} = \mathbf{I} \otimes |0\rangle\langle 0| + \mathbf{X} \otimes |1\rangle\langle 1| \quad C_{ij} = \bar{\mathbf{n}}_i + \mathbf{X}_j \mathbf{n}_i$$

Il y a en réalité une subtilité à saisir sur les différentes notations utilisées dans les deux formules précédentes. À gauche apparaît le produit tensoriel entre deux matrices de taille 2, on obtient ainsi une matrice de taille 4. Ce n'est pas un problème puisque on s'intéresse à des 2-Qbits, donc des vecteurs colonnes de taille 4. Qu'en est-il à droite des \mathbf{n}_i ? On a vu que $\mathbf{n} = |1\rangle\langle 1|$ est de taille 2, mais en ajoutant un indice de rang, on se place en fait dans un espace plus grand (puisqu'on considère plus qu'1 Qbit). Ainsi, la taille de n_0 va dépendre du nombre de Qbits n du circuit, et s'écrira $\mathbf{I} \otimes \dots$ n fois $\dots \otimes \mathbf{I} \otimes \mathbf{n}$.

5.12 Construction d'un état quelconque

On cherche à montrer qu'il est possible de construire un état quelconque* de 1 ou 2 Qbits à partir d'un nombre réduit d'opérateurs à 1 ou 2 Qbits.

Pour un 1-Qbit, il suffit d'appliquer la rotation \mathbf{R}_θ où θ est l'angle entre $|0\rangle$ et l'état $|\Psi\rangle$ souhaité. Nous savons que dans un plan de dimension 2, (formé ici par $|0\rangle$ et $|1\rangle$) une telle rotation est bien une opération unitaire.

Intéressons nous maintenant à un 2-Qbit, de la forme générale :

$$|\Psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = |0\rangle \otimes |\Psi_0\rangle + |1\rangle \otimes |\Psi_1\rangle$$

On applique ensuite l'opérateur $\mathbf{u} = \begin{pmatrix} a & -b^* \\ b & a^* \end{pmatrix}$ à gauche, ce qui donne après factorisation :

$$\mathbf{u} \otimes \mathbf{1} |\Psi\rangle = |0\rangle \otimes |\Psi'_0\rangle + |1\rangle \otimes |\Psi'_1\rangle$$

Où $|\Psi'_0\rangle = a|\Psi_0\rangle - b^*|\Psi_1\rangle$ et $|\Psi'_1\rangle = b|\Psi_0\rangle + a^*|\Psi_1\rangle$ que l'on peut rendre orthogonaux en choisissant judicieux a et b (détails dans le livre de Mermin [1] page 33).

On note ensuite, Ψ''_i ces deux vecteurs après normalisation (notons λ_i leurs normes). Ils forment donc une paire de vecteurs orthonormaux, et s'obtiennent ainsi à partir de la base computationnelle par une simple rotation :

$$|\Psi''_0\rangle = \mathbf{R}_\theta |0\rangle \quad |\Psi''_1\rangle = \mathbf{R}_\theta |1\rangle$$

Il reste à effectuer des factorisations successives :

$$\mathbf{u} \otimes \mathbf{1} |\Psi\rangle = |0\rangle \otimes \lambda_0 \mathbf{R}_\theta |0\rangle + |1\rangle \otimes \lambda_1 \mathbf{R}_\theta |1\rangle$$

$$\mathbf{u} \otimes \mathbf{1} |\Psi\rangle = (\mathbf{1} \otimes \mathbf{R}_\theta) (\lambda_0 |00\rangle + \lambda_1 |11\rangle)$$

Or $|11\rangle = \mathbf{C}_{10}|10\rangle$, et $\mathbf{u} \otimes \mathbf{1}$ est trivialement unitaire (car \mathbf{u} l'est) avec $(\mathbf{u} \otimes \mathbf{1})^\dagger = (\mathbf{u}^\dagger \otimes \mathbf{1})$ grâce aux propriétés du produit de Kronecker*. Ainsi :

$$|\Psi\rangle = (\mathbf{u}^\dagger \otimes \mathbf{1})(\mathbf{1} \otimes \mathbf{R}_\theta) \mathbf{C}_{10} (\lambda_0 |0\rangle + \lambda_1 |1\rangle) \otimes |0\rangle$$

Comme $|\Psi\rangle$ est unitaire, on vérifie que $\lambda_0 |0\rangle + \lambda_1 |1\rangle$ l'est aussi (les opérations unitaires préservent la norme) et s'obtient ainsi selon une nouvelle rotation $\mathbf{R}_\phi |0\rangle$. Après distribution des opérateurs on obtient enfin :

$$|\Psi\rangle = \mathbf{u}_1^\dagger \mathbf{R}_{\theta,0} \mathbf{C}_{10} \mathbf{R}_{\phi,1} |00\rangle$$

On a ainsi construit $|\Psi\rangle$ à partir de $|00\rangle$, d'une CNOT et de 3 opérations unitaires de dimension 1.

5.13 Additionneur

Nous présentons ici l'additionneur complet dont les prémises sont présentées dans la section de l'additionneur*.

On commence par adapter la fonction proposée afin qu'elle puisse s'ajouter à n'importe quel circuit. Une idée est de construire deux fonctions, l'une en particulier qui ajoute les retenues.

<pre>def add(qc,ia,ib, ic, ir): ''' ajoute les bits a et b entre eux (avec retenue) qc : le circuit complet ia : rang du bit a dans qc ib : rang du bit b dans qc ic : rang du bit du résultat de a+b ir : rang du bit de retenue ''' qc.barrier() #XOR qc.cx(ia,ic) qc.cx(ib,ic) #Retenue qc.ccx(ia,ib,ir) qc.barrier()</pre>	<pre>def add_retenue(qc,ic,ir1,ir2): ''' ajoute la retenue précédente au résultat précédent qc : circuit complet ic : rang du résultat ir1 : rang retenue précédente ir2 : rang seconde retenue ''' qc.barrier() #Retenue qc.ccx(ir1,ic,ir2) #XOR qc.cx(ir1,ic) qc.barrier() # #</pre>
---	---

Enfin, on peut créer une petite fonction qui rendra l'initialisation plus simple :

```
def state(a):  
    '''  
    fonction qui renvoie l'état initial de a  
    ex : a = '1' renvoie le ket [0,1] = |1>  
    '''  
    if int(a)==1:  
        return([0,1])  
    else :  
        return([1,0])
```

Il ne reste plus qu'à assembler le tout (code total page suivante).

```

def add_multi(a,b):
    '''
    a et b sous forme de chaine de caractère
    ex : a = '101'
    '''

    n = len(a)
    assert(len(b)==n)

    qc = QuantumCircuit(4*n,n+1)

    #initialisation des entrées
    for i in range(n):
        sa = state(a[n-1-i])
        sb = state(b[n-1-i])

        qc.initialize(sa,i)
        qc.initialize(sb,n+i)

    #additions
    for i in range(n):

        add(qc, i, n+i, 2*n+i,3*n+i)

        #addition des retenues
        if i >= 1:

            add_retenue(qc, 2*n+i,3*n+i-1,3*n+i)

    #mesures
    qc.measure(4*n-1,n)
    for i in range(n):
        qc.measure(2*n+n-i-1,n-i-1)

    return qc

#exemple :
qc = add_multi('101','011')
plot_histogram(counts_add(qc))

```

5.14 Deutsch

Voici les calculs de l'algorithme de Deutsch*.

On calcule progressivement chaque terme :

$$\begin{aligned} (\mathbf{H} \otimes \mathbf{H})(\mathbf{X} \otimes \mathbf{X})|00\rangle &= \mathbf{H}|1\rangle \otimes \mathbf{H}|1\rangle = \frac{1}{2}(|0\rangle - |1\rangle) \otimes (|0\rangle - |1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) \\ \Rightarrow U_f(\mathbf{H} \otimes \mathbf{H})(\mathbf{X} \otimes \mathbf{X})|00\rangle &= \frac{1}{2}(|0\rangle|f(0)\rangle - |0\rangle|\overline{f(0)}\rangle - |1\rangle|f(1)\rangle + |1\rangle|\overline{f(1)}\rangle) \\ &= \frac{1}{2}(|0\rangle \pm |1\rangle)(|f(0)\rangle - |\overline{f(0)}\rangle) \end{aligned}$$

Le \pm est un $-$ si $f(0) = f(1)$, et inversement. On reconnaît les deux sorties possibles d'une porte Hadamard. Or $\mathbf{H}^2 = 1$. Donc en appliquant \mathbf{H} au bit de poids fort :

$$(\mathbf{H} \otimes 1)U_f(\mathbf{H} \otimes \mathbf{H})(\mathbf{X} \otimes \mathbf{X})|00\rangle \begin{cases} |1\rangle \frac{1}{\sqrt{2}}(|f(0)\rangle - |\overline{f(0)}\rangle) & \text{si } f(0) = f(1) \\ |0\rangle \frac{1}{\sqrt{2}}(|f(0)\rangle - |\overline{f(0)}\rangle) & \text{si } f(0) \neq f(1) \end{cases}$$

5.15 Bernstein Varizani

Voici les calculs de l'algorithme de Bernstein Varizani*. Tout d'abord, il faut noter l'astuce de calcul suivante :

$$U_f|x\rangle_n(|0\rangle - |1\rangle) = |x\rangle_n(|f(x)\rangle - |\overline{f(x)}\rangle) = |x\rangle_n(-1)^{f(x)}(|0\rangle - |1\rangle)$$

Par ailleurs, il faut avoir compris l'effet de multiples Hadamard :

$$H^{\otimes n}|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \cdots \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{2^{\frac{n}{2}}} \sum_{x=0}^{2^n-1} |x\rangle_n$$

Ainsi on peut exprimer le terme suivant :

$$U_f H^{\otimes n+1}|0\rangle_n|1\rangle_1 = U_f \frac{1}{2^{\frac{n}{2}}} \sum_{x=0}^{2^n-1} |x\rangle_n \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{2^{\frac{n+1}{2}}} \sum_{x=0}^{2^n-1} |x\rangle_n (-1)^{f(x)}(|0\rangle - |1\rangle)$$

On considère ensuite la propriété suivante, qui utilise le fait que $\mathbf{H}|x\rangle = |0\rangle + (-1)^{x_j}|1\rangle$:

$$\mathbf{H}^{\otimes n}|x\rangle_n = \frac{1}{2^{\frac{n}{2}}} \sum_{y=0}^{2^n-1} (-1)^{x \cdot y} |y\rangle_n$$

En effet on garde le (-1) s'il y en un produit impaire i.e $x \cdot y = x_1 y_1 \oplus \cdots \oplus x_n y_n = 1$. Puis :

$$\mathbf{H}^{\otimes n+1} U_f \mathbf{H}^{\otimes n+1} |0\rangle_n |1\rangle_1 = \frac{1}{2^n} \sum_{x=0}^{2^n-1} \sum_{y=0}^{2^n-1} (-1)^{f(x)+x \cdot y} |y\rangle_n |1\rangle = \frac{1}{2^n} \sum_{y \in \{0,1\}^n} \sum_{x \in \{0,1\}^n} (-1)^{(a \oplus y) \cdot x} |y\rangle_n |1\rangle$$

Où $a \oplus y$ est une somme "vectorielle". Or par binarité entre les -1 et $+1$:

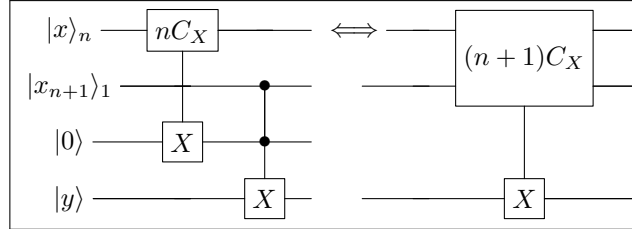
$$\sum_x (-1)^{(a \oplus y) \cdot x} = 1 \Leftrightarrow a \oplus y = 0_n \Leftrightarrow a = y \quad \Rightarrow \quad \mathbf{H}^{\otimes n+1} U_f \mathbf{H}^{\otimes n+1} |0\rangle_n |1\rangle_1 = |a\rangle_n |1\rangle$$

5.16 Grover

Porte n-CNOT

L'objectif est de construire une porte n-CNOT (que l'on notera \mathbf{nC}_X) pour l'algorithme de Grover*

Une façon de construire une telle porte se fait par récurrence :



Pour que le NOT s'applique, il faut que $x_{n+1} = 1$ et $\mathbf{nC}_X|x\rangle_n = 1$. Passer de n à $n+1$ nécessite donc l'ajout d'un bit à 0 et d'une porte de Toffoli*, dont la construction est abordée dans ce guide.

Construction de W

Montrons comment construire l'opérateur W pour l'algorithme de Grover*. Tout d'abord, sachant que $|\phi\rangle = H^{\otimes n}|0\rangle$, on a :

$$W = \mathbf{H}^{\otimes n}(2|0\rangle\langle 0| - \mathbf{I})\mathbf{H} \quad \text{car} \quad \mathbf{H}^\dagger = \mathbf{H} \quad \text{et} \quad \mathbf{H}^2 = \mathbf{I}$$

L'opérateur $\mathbf{I} - 2|0\rangle\langle 0|$ agit sur les kets de base comme l'identité sauf pour $|0\rangle$ qu'on multiplie par -1 . Cela rappelle le fonctionnement d'une porte Z (qui n'agit que sur $|1\rangle$). On obtient le comportement désiré en appliquant un NOT sur tous les 0 puis une porte Z contrôlée par les (n-1) premiers bits et qui agit sur le dernier bit. On a alors :

$$\mathbf{I} - 2|0\rangle\langle 0| = \mathbf{X}^{\otimes n}(\mathbf{c}^{n-1}\mathbf{Z})\mathbf{X}^{\otimes n}$$

$$W = -\mathbf{H}^{\otimes n}\mathbf{X}^{\otimes n}(\mathbf{c}^{n-1}\mathbf{Z})\mathbf{X}^{\otimes n}\mathbf{H}^{\otimes n}$$

Construction de $\mathbf{c}^{n-1}\mathbf{Z}$

Pour construire la porte $\mathbf{c}^n\mathbf{Z}$, on agit par récurrence, en s'inspirant de la construction des n-CNOTs et de $\mathbf{H}\mathbf{X}\mathbf{H} = \mathbf{Z}$. Le livre de Mermin [1] donne un exemple de construction page 94 et 95 (il existe en fait une multitude de façon de construire les portes n-CX ou n-CZ).

References

- [1] Mermin, N. David. *Quantum Computer Science: An Introduction*. Cambridge University Press, New York (2007).
- [2] IBM Quantum. *Getting started with Qiskit*. <https://docs.quantum.ibm.com/>