Pôle projet

CentraleSupélec - 1A

Crash course d'informatique quantique

June 1, 2021

CentraleSupélec

Auteurs :
Gaspard Berthelier
Maxime Wolf
Antoine Debouchage Léo
Durand–Kollner

Contents

1 Introduction

Ce guide a pour but de rapidement introduire l'informatique quantique à ceux qui souhaitent s'initier sans passer trop de temps à bachoter les (longs) cours disponibles dans la littérature. L'idée est de présenter les fondements théoriques les plus importants et de les appliquer sur des algorithmes quantiques simples. Divers sous-domaines de l'informatique quantique seront ensuite développés. Il est préférable d'avoir suivi des cours de CPGE en maths et python ou l'équivalent pour bien comprendre les propos ; des annexes mathématiques sont en revanche disponibles en fin d'ouvrage. Une asterisk sera présente à droite d'un mot-clé* qui se réfère à l'une des annexes.

Ce pdf est écrit par des élèves en première année d'école d'ingénieur et peut donc éventuellement comporter des approximations ou abus de notations.

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

2 Partie 1 - Notions de base

2.1 Notations de Dirac

Nous travaillons sur des espaces de Hilbert*. Un vecteur de cet espace sera nommé un "ket" et se notera sous cette forme : $|\Psi\rangle$. Si l'on note ensuite $|x\rangle$ le $x^{ième}$ vecteur d'une base de l'espace, il est alors 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 est le dual* de $|\Psi\rangle$. En notation vectorielle usuelle, il est égal à $\Psi^{T*} = \Psi^\dagger$. 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:

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

De même, pour un opérateur A (représenté par une matrice), on définit l'opérateur adjoint $A^\dagger = A^{T*}$ qui obéit aux propriétés suivantes :

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

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

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

(C'est en fait la définition de l'opérateur adjoint, qui devient l'opérateur transconjugué avec le produit scalaire usuel).

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

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

2.2 Définition des Cbits

On va considérer qu'un vecteur peut exister sous deux états uniquement : $|0\rangle$ et $|1\rangle$. On ne s'intéresse pas au phénomène physique qui pourra représenter ces deux états (un courant on/off, l'état de spin d'une particule, etc).

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 sont orthogonaux dans un plan vectoriel de dimension 2, mais attention, nous ne sommes pour l'instant qu'en présence de Cbits de dimension 1.

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

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

Dans le cas de n bits, on utilise la représentation binaire pour exprimer le $p^{ième}$ vecteur, $p \in [|0, 2^n - 1|]$. Par exemple, pour écrire le deuxième vecteur possible sur 3 bits :

$$|2\rangle_3 = |010\rangle = |0\rangle\,|1\rangle\,|0\rangle$$

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. Un 2 est ainsi un $|0\rangle$ en position 0, un $|1\rangle$ en position 1, et des 0 pour le reste selon la taille de l'espace.

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

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

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 (puisque l'on peut coder les décimaux de 0 à $2^n - 1$)

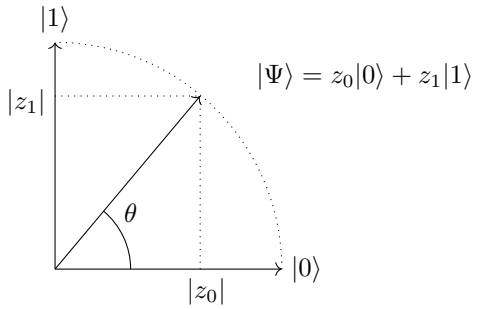
2.3 Définition des Qbits

2.3.1 Définition

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 :

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

En dimension 1, on peut visualiser cela dans le plan 2D : un Qbit de dimension 1 sera un vecteur sur le cercle unité, auquel on peut ajouter une phase sur chaque composante (les phases n'interviennent pas dans la norme des amplitudes).



2.3.2 Règle de Born

Un Qbit représente donc une superposition des 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, nous reviendrons sur cette distinction).

Plus précisemment, 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. Par ailleurs, après la mesure d'un Qbit dans un état donné, ce Qbit restera dans cet état pour tout le restant de sa vie (si l'on n'effectue aucune autre opération que des mesures). Il n'est plus en superposition, c'est la "décohérence", phénomène physique sur lequel nous ne nous attarderons pas. Retenez simplement que mesurer un Qbit revient à le réinitialiser aléatoirement dans un état de la base computationnelle.

$$|\Psi\rangle_n = \sum c_x|x\rangle_n \Longrightarrow\boxed{\quad\diagup\quad}\Longrightarrow |x\rangle_n \qquad p = |c_x|^2$$

La porte "mesure" que nous avons considéré mesure n bits d'un coup. Mais il est aussi possible de mesurer un seul bit à la fois, en factorisant certains états entre-eux :

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

Il s'agit en fait d'une factorisation avec d'une part les vecteurs commençant par 0 et d'autre part ceux commençant par 1, ce qui correspond à une "réunion d'événements disjoints" en probabilité, où $\alpha_0, \alpha_1, |\Psi_0\rangle, |\Psi_1\rangle$ sont construits de sorte à avoir $|\alpha_0|^2 + |\alpha_1|^2 = 1$ (calcul en annexe)*. On aura alors :

$$\alpha_0|0\rangle|\Psi_0\rangle_n + \alpha_1|1\rangle|\Psi_1\rangle_n \longrightarrow |0\rangle|\Psi_0\rangle_n \qquad p = |\alpha_0|^2$$

Pour conclure cette sous-partie, retenez un second intérêt des portes mesures : celle 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 montrer qu'il est impossible de construire une porte universelle de clonage, c'est à dire qui recopie n'importe quel Qbit en entrée (démo dans le Mermin*). On initialisera donc sounvent un circuit souvent en répétitant le processus décrit précédemment.

2.3.3 Intrication

Nous allons prendre l'exemple de la dimension 2 pour donner l'intuition au lecteur 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 est un sous-espace des 2-Qbits. On peut passer de la première forme à la seconde si et seulement si $c_{00}c_{11} = c_{01}c_{10}$. Un vecteur d'état qui ne peut se factoriser en un produit tensoriel de 1-Qbits est dit "intriqué". Cela a des conséquences majeurs sur lesquelles nous reviendrons dans la partie "non localité".

2.3.4 Simulation de Qbits sur qiskit

Il faut d'abord télécharger la bibliothèque et importer les modules principaux dans votre environment python :

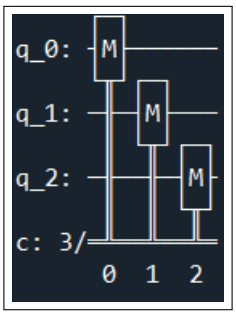
```
#dans le shell
pip install qiskit
pip install qiskit[visualization]

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

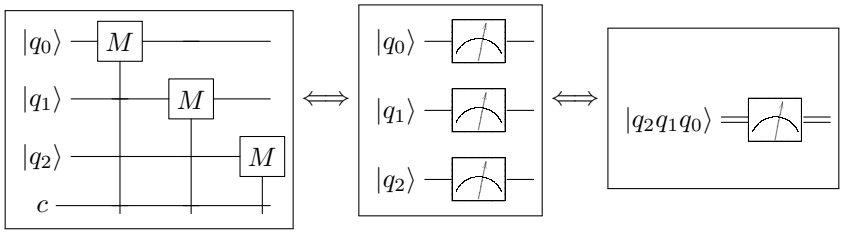
2.3.4.1 Création d'un circuit

Nous allons créer un circuit avec n Qbits. Lorsque l'on effectura des mesures sur ces Qbits, il faudra inscrire le résultat dans n 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
```

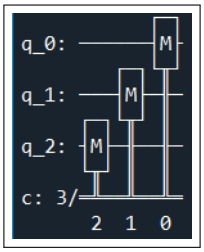
Vous voyez ci-dessous ce que renvoie le code précédent sur Anaconda Spyder (image supérieure). Par soucis d'une meilleure lisibilité, nous utiliserons parfois plutôt un module LateX pour la visualisation des circuits (images inférieures).





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()
```

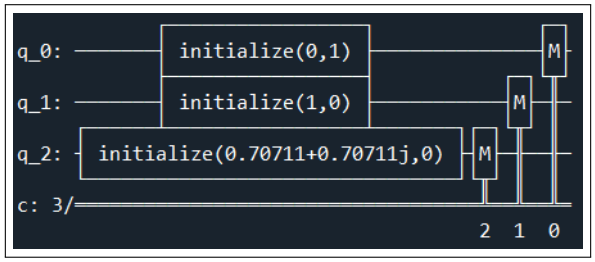


2.3.4.2 Initilisation

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>
initial_state_1 = [1,0] #état |0>
initial_state_2 = [(1/sqrt(2))*(1+1j),0] #état de même probabilité que |0>

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()
```

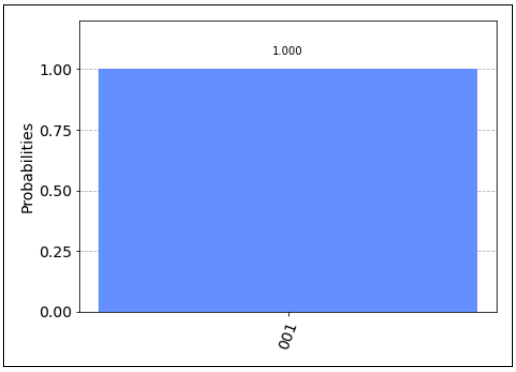
Notez qu'il a fallu réinitialiser le circuit. Si l'on avait initialisé les bits sur le circuit déjà existant, l'initialisation aurait été ajoutée après les blocs mesure d'avant. Avec le code précédent :



2.3.4.3 Mesures avec qasm_simulator

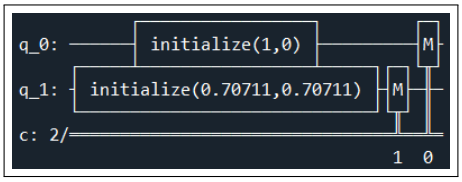
Nous allons effectuer des mesures sur les Qbits grâce au calculateur qasm_simulator qui agit comme un vrai ordinateur quantique (c'est à dire qu'il suit la règle de Born). Ce calculateur répète l'opérateur un grand nombre de fois pour obtenir une fréquence d'apparition de chaque état et permet ainsi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

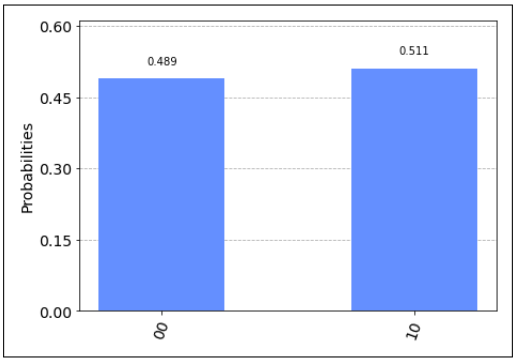
```
backend = Aer.get_backend('qasm_simulator')
counts = execute(qc, backend).result().get_counts()

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)
```





On constate ainsi 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 :

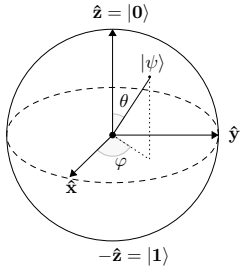
$$|\Psi\rangle = r_0 e^{i\gamma_0}|0\rangle + r_1 e^{i\gamma_1}|1\rangle, \quad r_0^2 + r_1^2 = 1$$

$$|\Psi_0\rangle = e^{i\gamma}\left[\cos(\frac{\theta}{2})|0\rangle + e^{i\varphi}\sin(\frac{\theta}{2})|1\rangle\right]$$

où les coefficients apparaissant dans cette expression sont des réels. 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 :

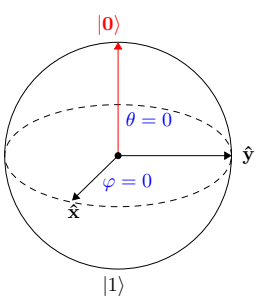
$$|\Psi\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\varphi}\sin(\frac{\theta}{2})|1\rangle$$

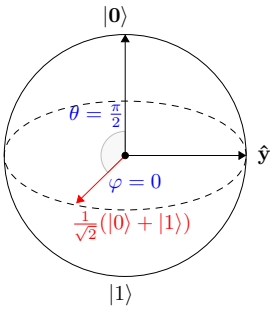
Une représentation de cet état est donné sur la figure ci-dessous :



On se rend compte qu'un état à n Qbits non intriqué peut donc se représenter avec n sphères de Bloch. Par exemple :

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

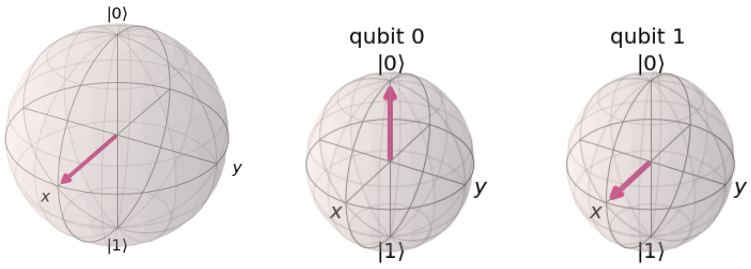




Regardons comment tracer ces sphères avec qiskit :

```
#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 :

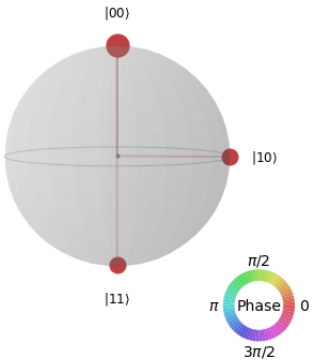


Pour la suite, nous n'utiliserons que la fonction plot_bloch_multivector.

Pour des états intriqués comme : $|\Psi\rangle = \frac{1}{\sqrt{2}}(|10\rangle + |01\rangle)$, il est impossible d'utiliser la sphère de Bloch. La partie suivante présentera une autre façon de représenter les états quantiques.

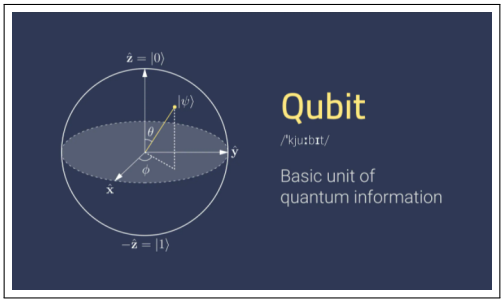
2.3.4.6 Q-Sphère

```
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 verrons dans les parties suivantes comment construire une état intriqué (nous n'avons vu sur qiskit que l'initialisation de Qbits non intriqués).



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

2.4 Opérations de base sur les Qbits

Nous allons 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. Nous indiquerons comment implémenter ces opérateurs sur qiskit.

2.4.1 Opérateurs sur 1 bit

- Identité (noté $\mathbf{1}$) : $\mathbf{1}(0) = 0$ et $\mathbf{1}(1) = 1$ de matrice $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
- NOT (noté $\mathbf{X}$) : $\mathbf{X}(0) = 1$ et $\mathbf{X}(1) = 0$ 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. De tels opérateurs non inversibles* existent aussi, mais on ne s'y intéressera pas.

```
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 transformer ces deux équations en une seule :

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

```
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 :

$$\mathbf{X^2} = \mathbf{Y^2} = \mathbf{Z^2} = \mathbf{H^2} = \mathbf{1}$$

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".

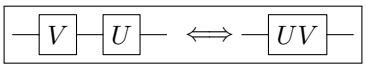
- SWAP : S_{ij} échange les bits de rang i et j. Par exemple : $S_{10}|xy\rangle = S_{01}|xy\rangle = |yx\rangle$

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

```
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 maximum. 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 systèmes.

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



Enfin, si l'on veut appliquer des portes qu'à certains Qbits d'un n-Qbit, il faudra 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 :

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

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 biai de **H**.

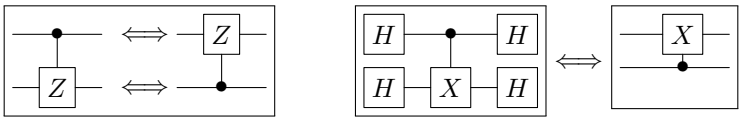
$$\mathbf{HXH} = \mathbf{Z} \quad \mathbf{HZH} = \mathbf{X}$$

Cette identité se vérifie facilement par un calcul matriciel. Vérifions plutôt avec notre fonction *compare*.

```
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 vérifie bien que la fonction renvoie *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 (l'action de Z est symétrique sur les vecteurs de base) :



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

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 dans les faits une succesion simple de portes pour initialiser vos circuits à l'état souhaité. Cela justifie l'utilisation de *iniatialize* 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 dans les états $|0\rangle$ ou $|1\rangle$. Ainsi, si on ne fait rien de plus à $|\Psi\rangle$, les deux valeurs doivent pouvoir être mesurées par Alice et Bob.

Maintenant, supposons que Alice et Bob s'écartent l'un de l'autre d'une très grande distance. Par ailleurs, chacun applique ou non une porte d'Hadamard à son Qbit uniquement, selon une probabilité de $\frac{1}{2}$. On peut alors lister selon l'action de chacun l'état futur de $|\Psi\rangle$.

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

H_a et H_b	$\cdots + 0\|11\rangle$
H_a et $\overline{H_b}$	$\cdots + 0\|01\rangle$
$\overline{H_a}$ et H_b	$\cdots + 0\|10\rangle$
$\overline{H_a}$ 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, $|11\rangle$ ou $|01\rangle$ ne sera plus possible. Ce qui est surprenant, est que le choix de Bob va influer sur les possibilités du Qbit d'Alice, alors que celui-ci peut-être à une distance infinie d'elle.

2.4.6 Réversibilité

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

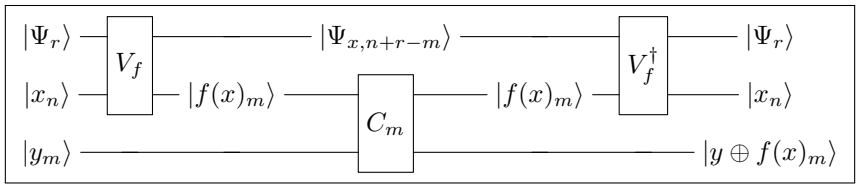
$$U_f|xy\rangle = |x\rangle|y \oplus f(x)\rangle$$

Cet opérateur est particulierement intéressant quand $y = 0$, auquel cas :

$$U_f|x\rangle|0\rangle = |x\rangle|f(x)\rangle$$

De cette manière, U_f est effectivement inversible.

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 des 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 :

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

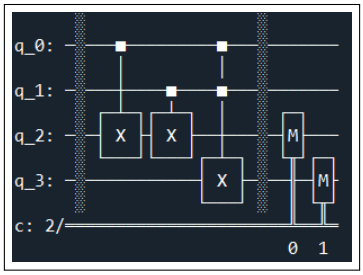
Notez qu'il faut 4 bits pour aditionner 111 et 001 par exemple.

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 visiblité

#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 Partie 2 - 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 calculs quantiques 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 A au $i^{ième}$ bit de $|\Psi\rangle$. Cela s'écrit ainsi :

$$\mathbf{H_1}|000\rangle = (\mathbf{1} \otimes \mathbf{H} \otimes \mathbf{1})(|0\rangle \otimes |0\rangle \otimes |0\rangle) = |0\rangle \otimes \mathbf{H}|0\rangle \otimes |0\rangle$$

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 :

$$\mathbf{H} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \quad \mathbf{H_0} = \begin{pmatrix}H & (0)\\ (0) & H\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1 & 0 & 0\\ 1 & -1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 1 & -1\end{pmatrix}$$

AJOUTER DES TRUCS ICI

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 : $f_0 = \mathbf{1}$, $f_1 = \mathbf{X}$, $f_2 = 0$, $f_3 = 1$, 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 fonction, il faut en temps normal deux évaluations pour la connaître. Par exemple, f(1) = 1 permet de réduire les possiblités à f_0 et f_3. Connaître ensuite, f(0) = 1 permet de savoir avec certitude f = f_3. De même, si l'on veut connaître si f est constante ou non, il faut nécessairement deux évaluations. Nous allons voir comment déterminer si f est constante ou non avec seulement 1 seule opération quantique.

Il suffit de considérer l'état suivant :

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

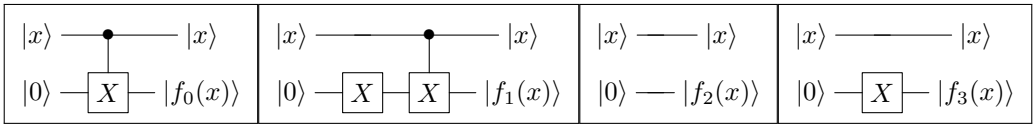
Le calcul est en annexe*. On constate donc qu'il peut se mettre sous la forme suivante :

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

$|\Phi\rangle$ est constant et ne dépend pas de la fonction. Ainsi, il suffit de mesurer le bit de poids fort de $|\Psi\rangle$ pour savoir 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". Notez en revanche qu'il ne nous apporte pas plus d'information sur les valeurs de la fonction.
Un exemple plus puissant serait de calculer si la millonième décimale de $\sqrt{2}$ est égale à celle de $\sqrt{3}$. Selon un algorithme similaire à celui de Deutsch, il est possible de réaliser cela avec la même complexité que de calculer la valeur de la millonième décimale de $\sqrt{2}$ ou celle de $\sqrt{3}$. Le gain de temps est considérable. En revanche on n'en saura pas dans ce cas davantage sur la valeur réelle de cette décimale.

3.1.2 Implémentation qiskit

Il faut d'abord coder les fonctions mystères. 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 de créer une fonction qui prend en valeur un entier entre 0 et 3 et construit le circuit de la fonction correspondante.

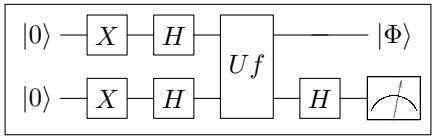
Solution :

```
    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)

    qc = qc + fonction_mystere(k_f) #on concatène les circuits
    #attention à l'ordre

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

On va enfin coder l'algorithme de Deutsch. Essayez de le faire vous même.



Solution :

```
def deutsch(k_f):
    backend = Aer.get_backend('statevector_simulator')
    qc = QuantumCircuit(2,1)
    qc.barrier()

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

    qc + fonction_mystere(k_f)
    qc.barrier()

    qc.h(1)
    qc.measure(1,0)
```

3.2 Bernstein Varizani

3.2.1 Algorithme

On s'intéresse aux fonctions de la forme :

$$f_a(x) = (a.x) = a_0 x_0 \otimes \cdots \otimes a_n x_n, \quad x = \sum_{k=0}^{n-1} x_k 2^k$$

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

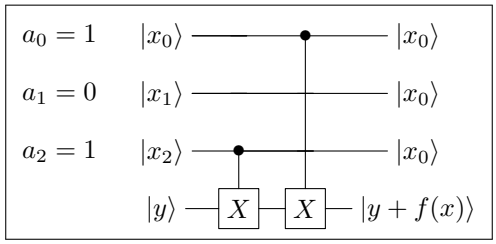
Il suffit de considérer l'état suivant (détails de calcul en annexe*) :

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

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 dés 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 dés que $x_j = 1$. Ainsi il suffit d'ajouter des portes CNOT entre x_j et la sortie dés que $a_j = 1$.

Par exemple pour $a = 101$:



On va d'abord créer une fonction qui permet de créer un tel circuit en fonction de a. Tentez de le faire par vous même. Solution :

```
fonction_bernstein(a):
    #on suppose que a est donné sous la forme d'une chaine 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,n)

    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(fct,n):
    qc = QuantumCircuit(n+1,n)
    qc.initialize([0,1],0)
    for k in range(n+1):
        qc.h(k)
    qc = qc+fct
    for k in range(n+1):
        qc.h(k)
    for k in range(n+1,0,-1):
        qc.measure(k,k)
```

AJOUTER IMAGES RESULTATS

3.3 Grover

3.3.1 Algorithme

On s'intéresse à la fonction :

$$\begin{array}{rcccc} f & : & [|0, 2^n - 1|] & \to & \{0, 1\} \\ & & x & \mapsto & \delta_a(x) \end{array}$$

Le but est de trouver a.
On commence par préparer 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 :

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

L'opérateur V est souvent appelé opérateur de Grover.

Schéma

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

L'idée est d'appliquer WV m fois à Φ, c'est à dire effectuer m rotations de 2θ de sorte à avoir $2m\theta = \frac{\pi}{2}$

3.3.2 Implémentation qiskit

3.4 Portes Toffoli

3.4.1 Algorithme

L'objectif est de coder l'opérateur suivant :

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

Un NOT est appliqué à z si et seulement si x et y sont à 1.
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 portes de Toffoli avec uniquement des portes à 1 et 2 Qbits.

3.4.1.1 Avec 8 CNOTS

L'idée est de construire la porte :

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

Avec d'une part :

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

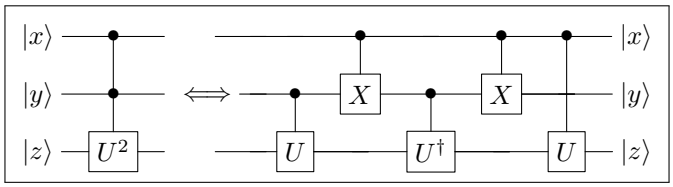
Et d'autre part pour un opérateur U quelconque :

$$C^{U^2}|xyz\rangle = U_0^{2xy}|xyz\rangle$$

Ainsi cet opérateur applique U^2 à z si et seulement si x et y sont à 1. On l'appelle "double-controlledU^2 " ou juste CC-U.

Pour pouvoir construire un CC-U, on se sert de portes controlled-U simple :

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



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.

Voyons maintenant comment implémenter la porte c-U.

3.4.1.2 Avec 6 CNOTS

L'idée est de construire la porte :

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

Avec en particulier :

$$U = e^{-i\frac{\pi}{2}\mathbf{n}}, \quad A^2 = B^2 = 1$$

3.4.2 Implémentation qiskit

4 Partie 3 - Approfondissement

Nous allons maintenant introduire plusieurs domaines intéressants et prometteurs de l'informatique quantique :

- La cryptologie : avec la Quantum Fourier Transform (QFT)
- L'optimisation : avec QAOA et les matrix product states (MPS)
- Le machine learning

4.1 Quantum Fourier Transform

4.1.1 Introduction

Il existe principalement 4 classes d'algorithmes quantiques : les algorithmes *variationnels* (*e.g. VQE*), les algorithmes de *simulation quantique*, les algorithmes de *recherche* (*e.g. Grover*), et enfin les algorithmes basés sur la *transformée de Fourier quantique* (QFT). Dans ces notes de cours, on s'attarde sur cette dernière catégorie. On présente la QFT, et deux de ses plus grandes applications : *Quantum Phase Estimation* (QPE) et *l'algorithme de Shor*.

4.1.2 QFT

La QFT est définie, de manière similaire au cas classique (voir annexe*), par l'opérateur linéaire agissant sur la base orthonormée $|0\rangle, |1\rangle, \cdots, |N-1\rangle$ comme suit :

$$|j\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} |k\rangle e^{\frac{2\pi i k j}{N}}$$

Par linéarité, on peut représenter l'action de cette transformation sur un état quelconque par :

$$\sum_{j=0}^{N-1} x_j |j\rangle \longrightarrow \sum_{k=0}^{N-1} y_k |k\rangle$$

où y_k sont les amplitudes obtenurs par application de la transformée discrèter sur les (x_j). (voir annexe) Il est important de remarquer que l'opération QFT est unitaire et donc qu'elle peut être implémentée dans un ordinateur quantique comme les autres opérateurs qu'on a vus jusque là. Dans la suite, on prendra $N = 2^n$. On écrira un état $|j\rangle$ en utilisant sa représentation binaire :

$$j = j_1 j_2 \cdots j_n$$

et on écrira aussi :

$$0.j_l j_{l+1} \cdots j_m = \frac{j_l}{2} + \frac{j_{l+1}}{4} \cdots + \frac{j_m}{2^{m-l+1}}$$

Voici une définition équivalente pour l'opérateur QFT :

$$|j\rangle \longrightarrow \frac{(|0\rangle + e^{2\pi i 0.j_n}|1\rangle)(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle) \cdots (|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n}|1\rangle)}{2^{n/2}}$$

L'utilité de cette nouvelle écriture est qu'elle permet d'interpréter l'opérateur QFT comme une composition de rotations (plus précisement, des rotations avec un contrôle sur un bit). On note $R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2i\pi/2^k} \end{pmatrix}$.

Détaillons l'obtention du circuit quantique :

- D'abord, on applique l'opérateur de Hadamard sur le premier bit de $|j_1 \cdots j_n\rangle$, on obtient $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1})|j_2 \cdots j_n\rangle$. En effet, si $j_1 = 0$, $e^{2\pi i 0.j_1}) = 1$ et si $j_1 = 1$, $e^{2\pi i 0.j_1}) = e^{2\pi i/2} = -1$.
- On applique maintenant la rotation R_2 avec un contrôle sur le 2 ème bit, on obtient alors : $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1 j_2})|j_2 \cdots j_n\rangle$.
- On applique la rotation R_3 avec contrôle sur j_3 pour obtenir $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1 j_2 j_3})|j_2 \cdots j_n\rangle$.

• ...
• Enfin, on applique la rotation R_n avec contrôle sur j_n : $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n})|j_2 \cdots j_n\rangle$.

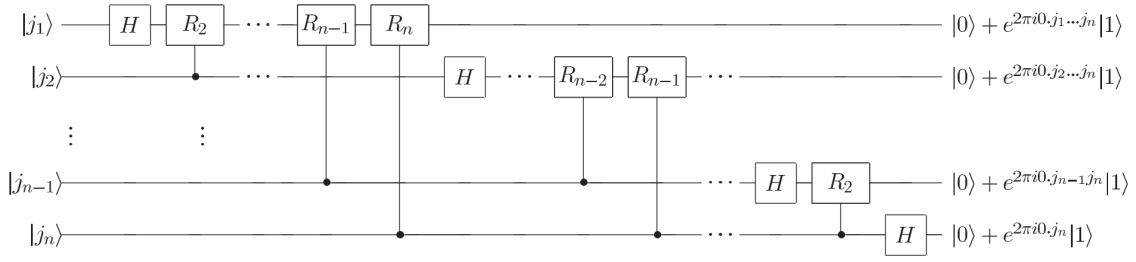
On a obtenu à l'aide d'une transformation d'Hadamard et de $n-1$ rotation l'état quantique :

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n})|j_2 \cdots j_n\rangle$$

L'idée est de réitérer ce processus sur chacun des autres bits, on effectue à chaque fois une transformation d'Hadamard sur le bit en question puis des rotations avec contrôle sur chacun des bits suivants successivement. A la fin bien :

$$\frac{(|0\rangle + e^{2\pi i 0.j_1 \cdots j_n}|1\rangle)(|0\rangle + e^{2\pi i 0.j_2 \cdots j_n}|1\rangle) \cdots (|0\rangle + e^{2\pi i 0.j_1}|1\rangle)}{2^{n/2}}$$

Pour obtenir, le bon état, il faut encore permuter les qubits. Généralement, on ne représente pas ces opérations sur le circuit pour alléger les notations. Il est nécessaire d'appliquer au plus $n/2$ portes S (swap). Pour j_k, on applique $1 + (n-k)$ opérateurs (1 porte d'Hadamard et $n-k$ rotations). En tout, on applique donc $\sum_{k=1}^{n} n + 1 - k = \frac{n(n+1)}{2}$ portes et en ajoutant les portes de permutation (réalisables avec 3 portes C-NOT), le circuit nous donne un algorithme en $O(n^2)$ pour le calcul de la QFT.



4.1.3 QPE

4.1.3.1 Présentation du problème

On peut maintenant se servir de la QFT pour résoudre des problèmes. Une des applications les plus courantes est la QPE : *Quantum Phase Estimation*.

Posons le problème : soit **U** un opérateur unitaire de vecteur propre $|u\rangle$ associé à la valeur propre λ. Comme **U** est unitaire, ses valeurs propres sont nécéssairement sur le cercle unité, donc $\lambda = e^{2i\pi\varphi}$ avec $\varphi \in [0, 1[$. Le but du jeu est de trouver φ.

Pour cela, on suppose disposer de l'attirail suivant :

- On sait implémenter l'état $|u\rangle$.
- On sait implémenter les portes U^{2^k} pour n'importe quel k.

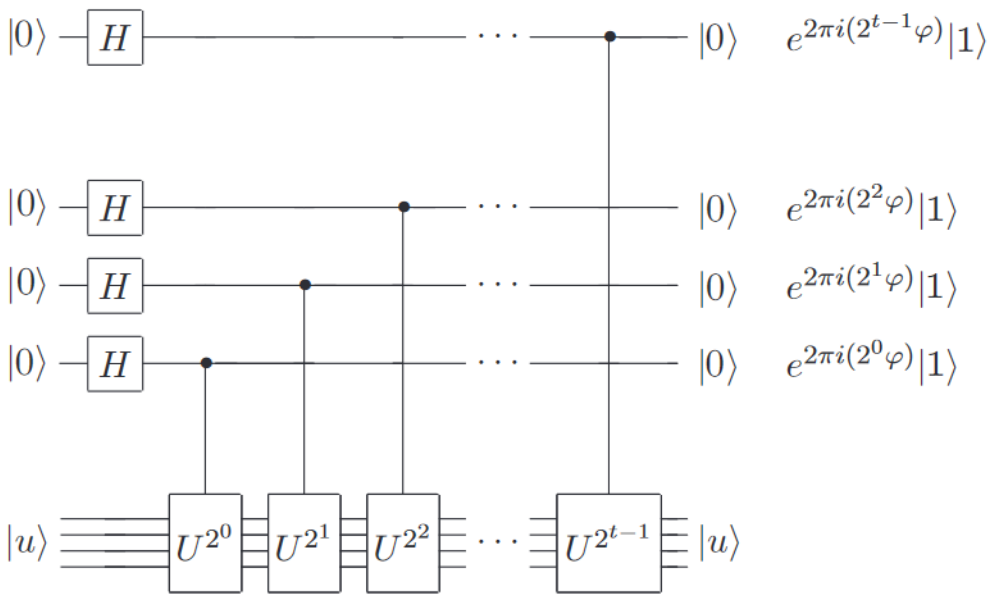


Figure 1: Première partie du circuit

4.1.3.2 Cas idéal

On se fixe un entier t, qui correspond à la précision souhaitée pour φ, et on suppose dans un premier temps que φ s'écrit de manière exacte en binaire sur t bits : $\varphi = \frac{1}{2}\varphi_1 + ... + \frac{1}{2^t}\varphi_t$. On note d le nombre de qbits sur lesquels agit $\mathbf{U}$. On dispose de deux registres : le premier est initialisé à $|0\rangle_t$, et le deuxième est initialisé à $|u\rangle_d$. On commence par appliquer des Hadamards au premier registre. Ensuite, on applique successivement des portes contrôlées $\mathbf{U}^{2^k}$ (cf figure 1).

Comme $|u\rangle$ est un vecteur propre, à chaque application de $\mathbf{U}$, on fait sortir un facteur $e^{2i\varphi\pi}$. Par conséquent l'état final du premier registre s'écrit :

$$|\psi\rangle = \frac{1}{2^{t/2}}(|0\rangle + e^{2i\pi 2^{t-1}\varphi}|1\rangle) \otimes ... \otimes (|0\rangle + e^{2i\pi 2^{0}\varphi}|1\rangle)$$

En utilisant l'écriture binaire de φ :

$$\begin{aligned}|\psi\rangle &= \frac{1}{2^{t/2}}(|0\rangle + e^{\frac{2i\pi}{2^t}2^{t-1}2^t\varphi}|1\rangle) \otimes ... \otimes (|0\rangle + e^{\frac{2i\pi}{2^t}2^{0}2^t\varphi}|1\rangle) \\ &= \frac{1}{2^{t/2}}\bigotimes_{k=0}^{t-1}(|0\rangle + \omega^{2^k(2^t\varphi)}|1\rangle)\end{aligned}$$

avec $\omega = e^{\frac{2i\pi}{2^t}}$, ce qui peut encore se réécrire, comme on l'a vu précédemment, comme :

$$|\psi\rangle = \frac{1}{2^{t/2}}\sum_{y=0}^{2^t-1}\omega^{(2^t\varphi)y}|y\rangle$$

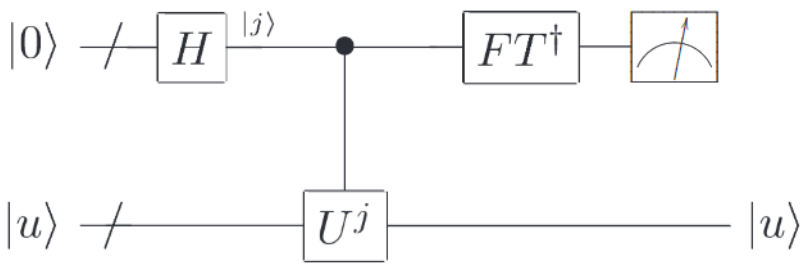


Figure 2: Circuit complet

Par conséquent, si on applique ensuite la QFT inverse à notre premier registre, on tombe sur $2^t\varphi = 2^{t-1}\varphi_1 + ... + 2^0\varphi_t$. On mesure directement l'écriture binaire de φ.

4.1.3.3 Cas général

Mais que se passe-t-il si φ ne s'écrit pas de manière exacte en binaire ? Est-on certain de trouver une bonne approximation de φ ? Il se trouve que QPE fonctionne toujours bien, mais avec une certaine probabilité. On a deux paramètres à prendre en compte :

- ϵ, notre tolérance à l'erreur
- m, la précision souhaitée

Si on choisit la taille t du premier registre comme

$$t = m + \lceil log_2(2 + \frac{1}{2\epsilon}) \rceil$$

alors QPE nous donne une approximation φ' de φ telle que $|\varphi - \varphi'| \leq \frac{1}{2^m}$ avec une probabilité d'au moins $1 - \epsilon$.

4.1.4 L'algorithme de Shor

L'algorithme de Shor est une des applications les plus fascinantes et remarquables de la QFT. Cet algorithme permet de factoriser un nombre composé en temps polynomial, contrairement aux meilleurs algorithmes classiques qui s'exécutent en temps exponentiel. Si des ordinateurs quantiques suffisamment performants et résistants au bruit venaient à voir le jour, des cryptosystèmes couramment utilisés comme *RSA* ou *ElGamal* deviendraient inefficaces. En effet, la sécurité de ces algorithmes repose sur la présumée difficulté de factoriser un nombre composé, ou, plus généralement, de calculer le logarithme discret d'un élément dans un groupe.

4.1.4.1 Rappels sur RSA

Rappels succints de RSA :

- $n = pq$ avec p et q premiers

- $e \in 1, \phi(n)$ avec $e \wedge \phi(n) = 1$
- $\exists! d \in 1, \phi(n)$ tel que $ed \equiv 1[\phi(n)]$

Message clair : $a \in 1, n$. Message chiffré : $b \equiv a^e[n]$. La question qu'on peut se poser est : Comment, à partir de b, n et e, retrouver a ?

Considérons r l'ordre de $\bar{b}$ dans $\mathbb{Z}/n\mathbb{Z}$: il peut être trouvé en déterminant la période de $f : x \mapsto \bar{b}^x$. Le sous groupe engendré par b est égal à celui engendré par a. En effet, $\forall x \in \langle \bar{b} \rangle$, $\exists k \in \mathbb{Z}, x = \bar{b}^k = \bar{a}^{ek} \in \langle \bar{a} \rangle$, et réciproquement $\forall x \in \langle \bar{a} \rangle, \exists k \in \mathbb{Z}, x = \bar{a}^k = \bar{b}^{-ke} \in \langle \bar{b} \rangle$. Donc $\bar{a}$ est également d'ordre r.

Soit maintenant $e' \in 1, r$, tel que $e \equiv e'[r]$. On a $e \wedge r = 1$ car $r|\phi(n)$ et $e \equiv e'[r]$. Donc $\bar{e} = \bar{e'}$ est inversible dans $\mathbb{Z}/r\mathbb{Z}$, ie $\exists d' \in 1, r, e'd' \equiv ed' \equiv 1[r]$, ie $\exists m \in \mathbb{Z}, ed' = 1 + mr$. On a alors $b^{d'} = a^{ed'} = aa^{mr} = a[n]$. d' peut facilement être calculé avec l'algorithme d'Euclide étendu.

On peut même faire encore plus fort que cela. On peut retrouver N, en suivant les étapes suivantes :

1. On choisit $a \in 1, n-1$ au hasard.
2. Si $a \wedge n \neq 1$, alors $a \wedge n$ est p ou q, on a donc gagné (mais n'arrive jamais, très très improbable, autant chercher à la main). Sinon on continue.
3. On calcule son ordre r avec l'algorithme de Shor (sections suivantes). Avec un peu de chance, $2|r$ et $a^{r/2} \not\equiv -1[n]$ (un peu plus de 50% de chances d'avoir un a qui convient). Sinon, goto 1.
4. Si on a obtenu un a qui convient, alors $n|(a^{r/2} - 1)(a^{r/2} + 1)$, mais on a aussi $n \nmid a^{r/2} + 1$ par hypothèse, et $n \nmid a^{r/2} - 1$ car l'ordre de a est r. Donc nécéssairement on a (à l'ordre de p et q près) $p|a^{r/2} + 1$ et $q|a^{r/2} - 1$. Alors on peut retrouver p et q avec $p = (a^{r/2} + 1) \wedge n$ et $q = (a^{r/2} - 1) \wedge n$. We won ! :)

4.1.4.2 Trouver l'ordre dans un groupe

Le but est donc de trouver r, ordre de a dans $\mathbb{Z}/n\mathbb{Z}$. De manière totalement inattendue et presque miraculeuse, *QPE* permet de faire cela ! Pour voir comment, on pose l'opérateur $\mathbf{U}$ qui agit de la manière suivante pour $x \in [0, n-1]$: $\mathbf{U}|x\rangle = |ax \pmod n\rangle$. On considère également les états suivants, indexés par un nombre $s \in [0, r-1]$:

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2i\pi sk}{r}} |a^k \pmod n\rangle$$

On peut remarquer qu'il s'agit de vecteurs propres de $\mathbf{U}$:

$$\begin{aligned}\mathbf{U}|u_s\rangle &= \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} e^{-\frac{2i\pi sk}{r}}|a^{k+1} \pmod n\rangle \\ &= \frac{1}{\sqrt{r}}e^{\frac{2i\pi s}{r}}\sum_{k=0}^{r-1} e^{-\frac{2i\pi sk}{r}}|a^{k} \pmod n\rangle \\ &= e^{\frac{2i\pi s}{r}}|u_s\rangle\end{aligned}$$

On peut donc utiliser QPE pour retrouver les valeurs propres associées, $\frac{2i\pi s}{r}$. On peut alors remonter à r ! Le deuxième registre doit donc être de taille $d = \lceil log_2(n)\rceil$, pour stocker le vecteur propre. Si on avait un des états $|u_s\rangle$, on pourrait déterminer $\frac{s}{r}$. On sait que QPE nous donne un résultat approché φ de $\frac{s}{r}$ avec une erreur de $\frac{1}{2^t}$ au plus :

$$|\frac{s}{r} - \varphi| \leq \frac{1}{2^t}$$

En fixant la taille du premier registre à $t = 2\lceil log_2(n)\rceil + 1$,

$$2^t \geq 2^{2log_2(n)+1} = 2n^2 \geq 2r^2$$

d'où :

$$|\frac{s}{r} - \varphi| \leq \frac{1}{2r^2}$$

Un théorème des fractions continues nous permet d'affirmer que dans ce cas, on peut trouver la valeur exacte de $\frac{s}{r}$ dans le développement en fractions continues de φ. On trouve alors s' et r' avec $s' \wedge r' = 1$ et $\frac{s}{r} = \frac{s'}{r'}$. Si s et r sont premiers entre eux, ce qui arrive très souvent, alors on connaît r. Sinon, on connaît un diviseur de r, généralement assez grand. In fine, on a réussi à casser RSA avec une probabilité assez haute :)

Maintenant, pour réussir à implémenter les états $|u_s\rangle$, il suffit de remarquer que $\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}|u_s\rangle = |1\rangle$. Ainsi en initialisant le deuxième registre à $|1\rangle$, on a en sortie une superposition de valeurs propres qui contiennent l'information sur r. En sortie QPE nous donne donc une valeur approchée d'un $\frac{s}{r}$ pour un s quelconque.

4.1.4.3 Un exemple : factorisation de 15

Dans cette section, on applique l'algorithme de Shor à la décomposition de 15. On détaille chaque étape.

Calcul de $\mathbf{CU}^{2^k}$

On a besoin de calculer $\mathbf{CU}^{2^k}$ pour n'importe quel a premier avec 15. Dans la suite, on note $\mathbf{CS_{i,j}}$ le *SWAP* contrôlé. On liste les cas possibles :

1. Cas $a = 1$: cela revient à multiplier par 1, donc à ne rien faire (identité)
2. Cas $a = 2$:
   - Cas $k = 0$: on fait une permutation circulaire.
     $$|2 \times x_3x_2x_1x_0 \pmod{15}\rangle = |x_2x_1x_0x_3\rangle$$
     (il y a une permutation car en écrivant la décomposition binaire de x : $2(2^3x_3 + 2^2x_2 + 2^1x_1 + 2^0x_0) \pmod{15} = 2^0x_3 + 2^3x_2 + 2^2x_1 + 2^1x_0$) par linéarité de mod. L'opération unitaire associée est donc $\mathbf{CS_{1,0}CS_{2,1}CS_{3,2}}$.
   - Cas $k = 1$: il faut multiplier par 4 (mod 15).
     $$|4 \times x_3x_2x_1x_0 \pmod{15}\rangle = |x_1x_0x_3x_2\rangle$$
     Autrement dit, on applique $\mathbf{CS_{3,1}CS_{2,0}}$.
   - Cas $k \geq 2$: $4^2 = 1 \pmod{15}$, donc en posant la division euclidienne de k par 2, $k = 2p+r$, on a
     $$|2^{2^k} \times x \pmod{15}\rangle = |4^{2^{2(p-1)+r}} x \pmod{15}\rangle = |x\rangle$$
     car $2(p-1) + r \geq 1$. On ne fait rien.
3. Cas $a = 4$: traité dans le cas $a = 2$
4. Cas $a = 7$:
   - Cas $k = 0$: on peut remarquer que la multiplication modulo 7 revient à appliquer un NON à une permutation circulaire :
     $$|7 \times x_3x_2x_1x_0 \pmod{15}\rangle = |\overline{x_0x_3x_2x_1}\rangle$$
     Autrement dit, on applique $\mathbf{CX_3CX_2CX_1CX_0CS_{3,2}CS_{2,1}CS_{1,0}}$
   - Cas $k = 1$: comme $7^2 = 4 \pmod{15}$, on se ramène au cas $a = 4$
   - Cas $k \geq 2$: idem $a = 2$, $k \geq 2$, on ne fait rien
5. Cas $a = 8$:
   - Cas $k = 0$: permutation circulaire vers la droite.
     $$|8 \times x_3x_2x_1x_0 \pmod{15}\rangle = |x_0x_3x_2x_1\rangle$$
     On applique donc $\mathbf{CS_{3,2}CS_{2,1}CS_{1,0}}$.

- Cas $k = 1$: $8^2 = 4 \pmod{15}$, donc on se ramène au cas $k = 0$, $a = 4$.
- Cas $k \geq 2$: toujours la même chose, on ne fait rien.

6. Cas $a = 11$:
   - Cas $k = 0$: on permute puis on fait une négation.

     $$|7 \times x_3x_2x_1x_0 \pmod{15}\rangle = |\overline{x_1x_0x_3x_2}\rangle$$

     On fait agir $\mathbf{CX_3CX_2CX_1CX_0CS_{3,1}CS_{2,0}}$.
   - Cas $k \geq 1$: $11^2 \pmod{15} = 1$, donc on ne fait rien.
7. Cas $a = 13$:
   - Cas $k = 0$: on permute puis on fait une négation.

     $$|7 \times x_3x_2x_1x_0 \pmod{15}\rangle = |\overline{x_2x_1x_0x_3}\rangle$$

     On fait agir $\mathbf{CX_3CX_2CX_1CX_0CS_{1,0}CS_{2,1}CS_{3,2}}$.
   - Cas $k \geq 1$: comme $13^2 \pmod{15} = 4$, on se ramène au cas $a = 4$.
8. Cas $a = 14$:
   - Cas $k = 0$: il s'agit simplement d'un NON.

     $$|14 \times x_3x_2x_1x_0 \pmod{15}\rangle = |\overline{x_3x_2x_1x_0}\rangle$$

     On applique $\mathbf{CX_3CX_2CX_1CX_0}$.
   - Cas $k \geq 1$: comme $14^2 = 1 \pmod{15}$, on ne fait rien.

4.1.4.4 Implémentation d'une porte de Fredkin

Ouf ! On a toutes nos opérations possibles. Le problème implicite, c'est qu'on n'a pas explicité l'expression des SWAP contrôlés, aussi appelés *portes de Fredkin* (figure 6). Pour cela, on peut décomposer un SWAP en CNOTs, pour se ramener à des portes de Toffoli. En effet,

$$\mathbf{S} = (\mathbf{C_0^1X})(\mathbf{C_1^0X})(\mathbf{C_0^1X})$$

Or, toute porte de Toffoli peut s'implémenter de la manière suivante, à l'aide de la racine carrée $\mathbf{V} = e^{i\frac{\pi}{4}}\mathbf{R_X}(\frac{\pi}{2})$ de $\mathbf{X}$:

$$\mathbf{C_0^{2,1}X} = (\mathbf{C_0^2V})(\mathbf{C_1^2X})(\mathbf{C_0^1V^\dagger})(\mathbf{C_1^2X})(\mathbf{C_0^1V})$$

ce qui est représenté figure 7. Donc au final :

$$\mathbf{C_{1,0}^2S} = (\mathbf{C_0^{2,1}X})(\mathbf{C_1^{2,0}X})(\mathbf{C_0^{2,1}X})$$

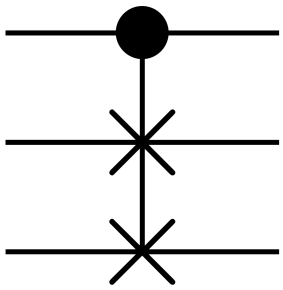


Figure 3: Porte de Fredkin

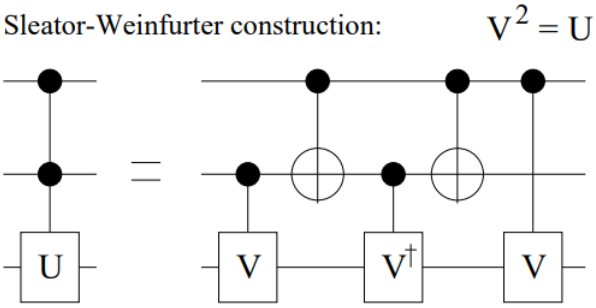


Figure 4: Construction d'une Toffoli

4.2 Théorie de l'information quantique

4.2.1 Matrice de densité

Soit un vecteur d'état normé $|\Psi\rangle$, on note sa matrice de densité $\hat{\rho} = \langle\Psi|\Psi\rangle$. On note $|u_n\rangle_n$ une base orthonormée de l'espace des états.

Propriétés :

- $|\Psi\rangle = \sum_n c_n|u_n\rangle$ avec $c_n = \langle u_n|\Psi\rangle$ et $\sum_n |c_n|^2 = 1$
- $\rho_{np} = \langle u_n|\hat{\rho}|u_p\rangle = \langle u_n|\Psi\rangle\langle\Psi|u_p\rangle = c_n c_p^*$
- $\hat{\rho} = \sum_{n,p} c_n c_p^*|u_n\rangle\langle u_p|$

4.2.2 Mélange statistique

Il s'agit d'une combinaison convexe d'états "purs" : $\hat{\rho} = \sum_i p_i\langle\Psi_i|\Psi_i\rangle$ où p_i est la probabilité de se retrouver dans l'état $\langle\Psi_i|\Psi_i\rangle$.

Propriétés :

- $\sum_i p_i = 1$ avec $p_i \geq 0$
- $\rho_{n,p} = \sum_i p_i\langle u_n|\Psi_i\rangle\langle\Psi_i|u_p\rangle = \sum_i c_n^i c_p^{i*}$
- $\hat{\rho} = \sum_{n,p,i} p_i c_n^i c_p^{i*}|u_n\rangle\langle u_p|$

On constate que l'aspect aléatoire provient de deux sources : l'une quantique (superposition d'états) et l'autre classique (distribution de probabilité sur plusieurs kets possibles).

4.3 QAOA (Quantum Approximate Optimization Algorithm)

4.3.1 Algorithme théorique

Voici les données du problème :

- On dispose de n bits et m clauses
- On définit la fonction $C(z) = \sum_{x=1}^{m} C_x(z)$ avec $z = z_1 z_2 \dots z_n$ une chaîne de bits.
- pour la $x$ième clause : $C_x(z) = 1$ si z satisfait la clause, et 0 sinon.

L'objectif est de trouver le nombre $\max(C(z))$.
L'algorithme s'appuie sur 2 opérateurs :

$$U(C, \gamma) := e^{-i\gamma C} = \prod_{x=1}^{m} e^{-i\gamma C_x}$$

Ici, C est vu comme un opérateur diagonal et on a pris γ entre 0 et 2π. On remarque que les termes de ce produit commutent car les matrices C_x (et donc les matrices $e^{-i\gamma C_x}$) sont diagonales dans la base usuelle.

Remarque : C est donc ici défini comme l'opérateur tel que : $C|z\rangle = \sum_{x=1}^{m} C_x(z)|z\rangle = f(z)|z\rangle$: il s'agit donc bien d'un opérateur diagonal.

On définit aussi :

$$U(B, \beta) := e^{-i\beta B} = \prod_{j=1}^{n} e^{-i\beta\sigma_j^x}$$

avec

$$B = \sum_{j=1}^{n} \sigma_j^x$$

Les opérateurs σ_j^x correspondent à la première matrice de Pauli, pour chacun des n bits considérés. On a cette fois pris β entre 0 et π.

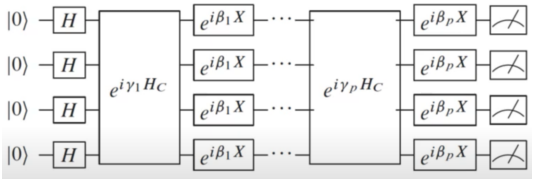
On choisit maintenant un entier p : plus p est grand, meilleure sera l'optimisation. On considère aussi une collection d'angles : $\gamma_1\gamma_2 \dots \gamma_p$ et $\beta_1\beta_2 \dots \beta_p$. On suppose que l'on dispose des C_x qui sont les clauses de notre probème d'optimisation. Toutes ces données correspondent enfait à l'entrée de notre algorithme. La sortie est une approximation de la solution du problème.

- On initialise un état : $|s\rangle = \frac{1}{\sqrt{2^n}} \sum_z |z\rangle = H^{\otimes n}|0^n\rangle$
- Ensuite, le circuit total consiste à appliquer successivement les rotations définies précédemment. A la fin du circuit l'état des qbits et donc :

  $$|\gamma, \beta\rangle := U(\beta_p, B)U(\gamma_p, C)U(\beta_{p-1}, B)U(\gamma_{p-1}, C) \dots U(\beta_1, B)U(\gamma_1, C)H^{\otimes n}|0^n\rangle$$

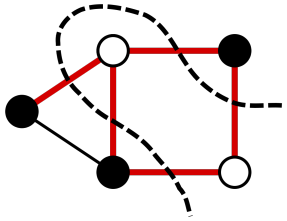
  Une approximation du problème est $\langle\gamma, \beta|C|\gamma, \beta\rangle := F_p$. En notant z la mesure de l'état $|\gamma, \beta\rangle$, on peut calculer $C(z)$ (pour rappel C est une donnée connue du problème). Plus p est grand, plus la quantité $C(z)$ sera proche de la valeur attendue F_p (tout en restant supérieure à F_p). On ne détaillera pas le choix de γ et β : l'idée est de les choisir tel que F_p soit maximal (ce qui peut se faire grâce à des algorithmes classiques comme "Grid Search").

Voici une représentation du circuit correspondant à l'algorithme :



4.3.2 Application au problème de Maximum Cut

Ce problème consiste à trouver une partition des sommets d'un graphe en deux ensembles tel que le nombre d'arêtes traversant ces deux ensembles est maximal.



Si l'on dispose d'un graphe régulier (tous les sommets ont même degré) avec un ensemble d'arêtes E et des sommets indnicés par des entiers, la fonction de coût associée au problème de coupe maximale est :

$$C(z) = \frac{1}{2} \sum_{(i,j)\in E} (1 - z_i z_j)$$

Cette quantité dénombre enfait le nombre d'arêtes qui sont des coupes. En effet, on définit : $z_i z_j = -1$ si (i, j) est une coupe et $z_i z_j = 1$ sinon. Cela définit par ailleurs les clauses : pour chaque $(i, j) \in E$, $C_{(i,j)}$ vaut 1 si (i, j) est une coupe, 0 sinon (on peut vérifier que ces opérateurs commutent bien). // Si l'on choisit une arête e de notre graphe, on peut définir le sous graphe de rayon p par rapport à e. C'est comme cela qu'est définit p. On se rend bien compte qu'en augmentant p, on aura une meilleure approximation du problème. // Tout cela montre que le problème Max Cut se ramène bien à QAOA.

4.4 Matrix Product State (MPS)

4.4.1 Principe

Nous allons voir maintenant une autre façon de représenter les états quantiques. Un système quantique de N particules peut être représenté par un tenseur* d'ordre N que l'on note $A_{i_1 i_2 \cdots i_N}$. Chaque composante de ce tenseur représente une amplitude pour le système de se retrouver dans une certaine configuration. Prenons en exemple d'un système de taille 2 :

$$A = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Cette matrice A signifie que le système peut se retrouver dans la configuration $|00\rangle$ ou $|11\rangle$ avec la même probabilité.

Dans certains cas, le tenseur peut se factoriser en des tenseurs d'ordre 1 :

$$\Psi_{\alpha\beta\gamma\cdots} = A_\alpha B_\beta C_\gamma \cdots$$

Ici, il s'agit d'une factorisation au sens du produit tensoriel. On n'écrit pas le symbole $\otimes$ pour simplifier l'écriture. Il se trouve que cette écriture n'est possible que lorsqu'il n'y a pas d'intrication entre les états.

Par exemple, le système quantique dans l'état suivant :

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

Il y a intrication donc le tenseur d'ordre 2 représenté par A ne peut pas se décomposer en 2 tenseurs d'ordre 1. En effet, il faudrait :

$$\begin{pmatrix} a \\ b \end{pmatrix} \begin{pmatrix} c & d \end{pmatrix} = A = \frac{1}{\sqrt{2}} I_2$$

$$\iff \quad ac = bd = \frac{1}{\sqrt{2}}, bc = ad = 0$$

Ceci est effectivement impossible. Cependant, il est possible d'écrire cette matrice comme un produit usuel de 2 matrices, en prenant :

$$B = C = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

De manière plus générale, quand il y a intrication, une décomposition en des tenseurs d'ordres compris entre 1 et celui d'origine est toujours possible. L'état Ψ pourrait être par exemple de la forme :

$$\Psi_{\alpha\beta\gamma\cdots} = \sum_i A_{\alpha,i} B_{i,\beta} C_\gamma \cdots$$

L'intrication fait apparaître de nouveaux indices qui apparaissent sous la forme de sommes. Physiquement, un tel indice représente un lien entre des particules dû à leur intrication. Dans le cas d'intrication maximale, il peut y avoir un lien entre chaque tenseur :

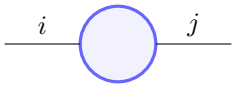
$$\Psi_{\alpha\beta\gamma\delta\cdots} = \sum_{i,j,k} A_{\alpha,i} B_{i,\beta,j} C_{j,\gamma,k} D_{k,\delta} \cdots$$

Ce qu'on appelle MPS est la représentation d'un état quantique comme un produit de matrices (tenseurs d'ordre 2).

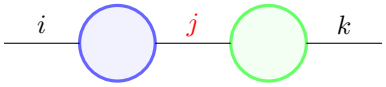
4.4.2 Diagramme / Réseau de tenseurs

Nous allons introduire une façon de représenter graphiquement les contractions de tenseurs et les MPS.

Une matrice indicée par (i, j) sera représentée par un noeud et 2 arêtes qui correspondent aux indices :

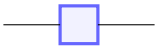


Si $(N)_{jk}$ est une autre matrice, la contraction (ie le produit matriciel) $(MN)_{ik} = \sum_j M_{ij} N_{jk}$ sera alors représenté par :

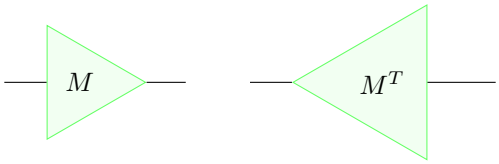


On adopte généralement d'autres conventions pour représenter certaines matrixes particulières :

- Pour une matrice symétrique :

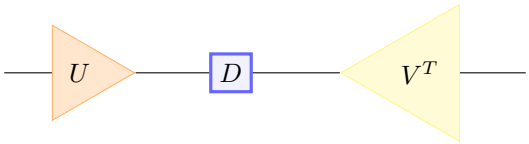


- Pour différencier une matrice et sa transposée :



La trace d'une matrice $(\mathrm{tr}(M) = \sum_i M_{ii})$ devient :

Avec les notations précédentes, la factorisation d'une matrice M à l'aide d'une SVD peut être représenté par :



4.5 Introduction au Machine Learning Quantique

L'intelligence artificielle et le machine learning sont des thèmes récurrents depuis plusieurs décennies. Avec l'essor de l'informatique quantique la question d'associer les deux domaines se pose naturellement. Nous allons voir dans cette section les concepts généraux du machine learning quantique. Quelques rappels seront fait mais nous ne détaillerons pas en détail certaines notions de machine learning classique pour nous concentrer sur son application dans le contexte quantique.

4.5.1 Encodage de données

RAM Quantique avec encodage en base

Dans le cadre du machine learning quantique, on se donne un dataset d'état binaire $\mathcal{D} = (x^m)_{m\in 1,M}$ avec pour tout $m \in I$, $x^m = (x_1^m, \ldots, x_N^m)$ avec $x_i^m \in \{0,1\}$ pour $i \in 1, N$. On cherche alors à créer une superposition de tous les états de bases $|x^m\rangle$.

$$|\mathcal{D}\rangle = \frac{1}{\sqrt{M}} \sum_{m=1}^{M} |x^m\rangle$$

Pour ce faire, on va créer une superposition de données en temps linéaire en M et N grâce à la méthode de préparation des données de Ventura et Martinez. On se donne un système quantique $|l_1, \ldots, l_N; a_1, a_2; s_1, \ldots, s_N\rangle = |l\rangle \otimes |a\rangle \otimes |s\rangle$ comprenant 3 registres :

1. le registre de chargement de N qbits $|l\rangle = |l_1, \ldots, l_N\rangle$
2. le registre auxiliaire composé de 2 qbits $|a\rangle = |a_1, a_2\rangle$
3. le registre de stockage $|s\rangle = |s_1, \ldots, s_N\rangle$

Intéressons nous au qbit a_2 du registre auxiliaire. Lorsque $a_2 = 0$, la superposition de tels états constitue la branche de mémoire tandis que lorsque $a_2 = 1$, on parle de la branche de processus.

On raisonne alors par récurrence sur $m \in 1, \ldots, M$. On prépare la première itération de l'algorithme avec l'état du système

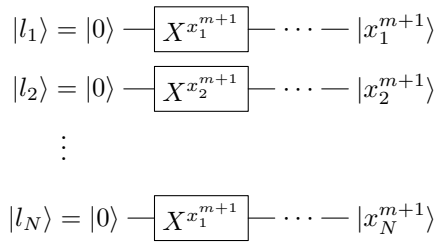
$$|\phi^{(0)}\rangle = |0, \ldots, 0; 0, 1; 0, \ldots, 0\rangle = |0\rangle_{\text{loading}} \otimes |01\rangle \otimes |0\rangle_{\text{storage}}$$

Supposons que les m premiers vecteurs d'entrainement ont été encodés en m itérations de l'algorithme. On est en présence de l'état suivant :

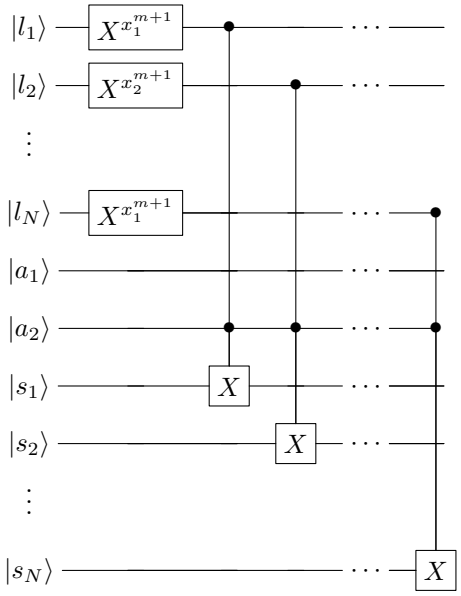
$$\begin{aligned} |\phi^{(m)}\rangle &= \tfrac{1}{\sqrt{M}} \textstyle\sum_{k=1}^{m} |0, \ldots, 0; 0, 0; x_1^k, \ldots, x_N^k\rangle + \sqrt{\tfrac{M-m}{M}} |0, \ldots, 0; 0, 1; 0, \ldots, 0\rangle \\ &= \tfrac{1}{\sqrt{M}} \textstyle\sum_{k=1}^{m} |0\rangle_{\text{loading}} \otimes |00\rangle \otimes |x^k\rangle + \sqrt{\tfrac{M-m}{M}} |0\rangle_{\text{loading}} \otimes |01\rangle \otimes |0\rangle_{\text{storage}} \end{aligned}$$

On peut expliciter plus en détail cet état, dans la branche de mémoire se trouve la superposition des m premières entrées dans le registre de mémoire alors que la branche de processus est dans son état de base. On remarque que le registre de chargement est à l'état de base ($|0, \ldots, 0\rangle$).

A partir d'un tel état, analysons l'itération pour encoder $m+1$ vecteurs d'entrées. Soit $x^{m+1} = (x_1^{m+1}, \ldots, x_N^{m+1})$, On écrit ce vecteur dans le registre de chargement en utilisant des portes $\mathbf{X}$ lorsque le bit est égal à 1.



Ensuite, on charge ce même vecteur dans le registre de mémoire en utilisant des portes double-**CNOT** sur les qbits du registre de chargement tout en contrôlant selon le deuxième qbit du registre auxiliaire en plus pour n'opérer que sur la branche de processus.



Ceci amène à l'état suivant :

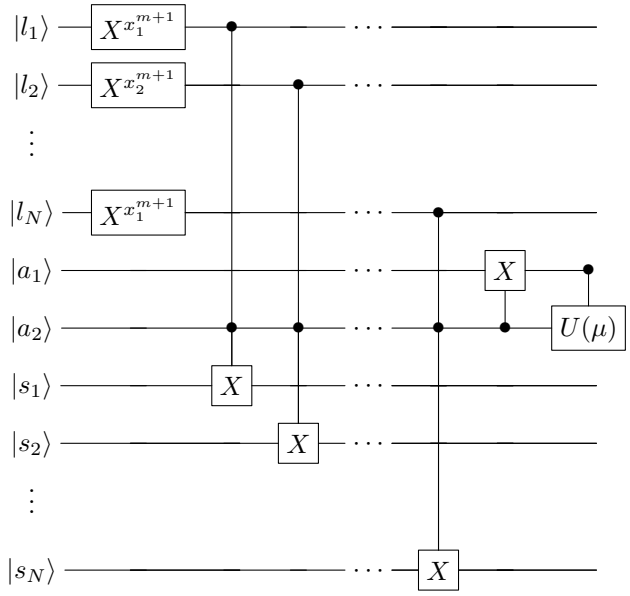
$$\begin{aligned} |\psi^{(m+1)}\rangle &= \textstyle\sum_{k=1}^{m} |x_1^{m+1},\ldots,x_N^{m+1};0,0;x_1^k,\ldots,x_N^k\rangle + \sqrt{\tfrac{M-m}{M}}|x_1^{m+1},\ldots,x_N^{m+1};0,1;x_1^{m+1},\ldots,x_N^{m+1}\rangle \\ &= \tfrac{1}{\sqrt{M}} \textstyle\sum_{k=1}^{m} |x^{m+1}\rangle \otimes |00\rangle \otimes |x^k\rangle + \sqrt{\tfrac{M-m}{M}}|x^{m+1}\rangle \otimes |01\rangle \otimes |x^{m+1}\rangle \end{aligned}$$

Ensuite, on sépare la branche de processus en deux sur la valeur de a_1 en utilisant une porte **CNOT** sur le qbit de contrôle a_2. On définit l'opérateur unitaire sur un qbit suivant avec $\mu = M + 1 - (m+1)$:

$$U(\mu) = \begin{pmatrix} \sqrt{\frac{\mu-1}{\mu}} & \frac{1}{\sqrt{\mu}} \\ -\frac{1}{\sqrt{\mu}} & \sqrt{\frac{\mu-1}{\mu}} \end{pmatrix}$$

On sépare les deux branches grâce à un control-U controllé par a_2, c'est-à-dire qu'on applique l'opération suivante sur tout le système : $\mathbb{1}_{\text{loading}} \otimes \mathbf{C}_{a_1} U_{a_2}(\mu) \otimes \mathbb{1}_{\text{storage}}$

On a donc le circuit suivant pour le moment :



Faisons le calcul sachant que $a_1 = |1\rangle$ et $a_2 = |1\rangle$,

$$V_{\text{split}}|0,\dots,0;0,1;0,\cdots,0\rangle = \mathbb{1}_{\text{loading}}\otimes\mathbf{C}_{a_1}U_{a_2}(\mu)\otimes\mathbb{1}_{\text{storage}}(|0\rangle_{\text{loading}}\otimes|01\rangle\otimes|0\rangle_{\text{storage}}) = |0\rangle_{\text{loading}}\otimes\left(\frac{1}{\sqrt{\mu}}|10\rangle + \sqrt{\frac{\mu-1}{\mu}}|\right.$$

car

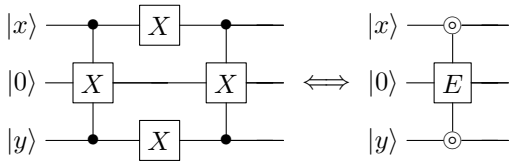
$$\begin{aligned} U(\mu)|1\rangle &= \begin{pmatrix} \sqrt{\frac{\mu-1}{\mu}} & \frac{1}{\sqrt{\mu}} \\ -\frac{1}{\sqrt{\mu}} & \sqrt{\frac{\mu-1}{\mu}} \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{\mu}} \\ \sqrt{\frac{\mu-1}{\mu}} \end{pmatrix} \\ &= \frac{1}{\sqrt{\mu}}|0\rangle + \sqrt{\frac{\mu-1}{\mu}}|1\rangle \end{aligned}$$

Ainsi, de manière générale avec $|\psi^{(m+1)}\rangle$, on tombe sur l'état suivant

$$\begin{aligned} V_{\text{split}}|\psi^{(m+1)}\rangle &= \frac{1}{\sqrt{M}}\sum_{k=1}^{m}|x^{m+1}\rangle\otimes|00\rangle\otimes|x^k\rangle + \sqrt{\frac{M-m}{M}}|x^{m+1}\rangle\otimes(\mathbf{C}_{a_1}U_{a_2}(\mu)|01\rangle)\otimes|x^{m+1}\rangle \\ &= \frac{1}{\sqrt{M}}\sum_{k=1}^{m}|x^{m+1}\rangle\otimes|00\rangle\otimes|x^k\rangle + \frac{1}{\sqrt{M}}|x^{m+1}\rangle\otimes|10\rangle\otimes|x^{m+1}\rangle \\ &\quad + \frac{\sqrt{M-(m+1)}}{\sqrt{M}}|x^{m+1}\rangle\otimes|11\rangle\otimes|x^{m+1}\rangle \end{aligned}$$

Il ne reste plus qu'à ajouter le $m+1$ ème terme à la somme en changeant $|10\rangle \longrightarrow |00\rangle$, pour ce faire il faut conditionner selon le fait que $|a_2\rangle = |0\rangle$ (ce qui permet de sélectionner seulement les deux sous-branches de processus) et que le registre de chargement et de stockage soit dans le même état.

Il est alors possible de vérifier si deux qbits sont dans le même état grâce à 2 portes de Toffoli :



On obtient

$$\mathbf{E} = \mathbf{T_{off}}(\mathbf{X} \otimes \mathbb{1} \otimes \mathbf{X})\mathbf{T_{off}}$$

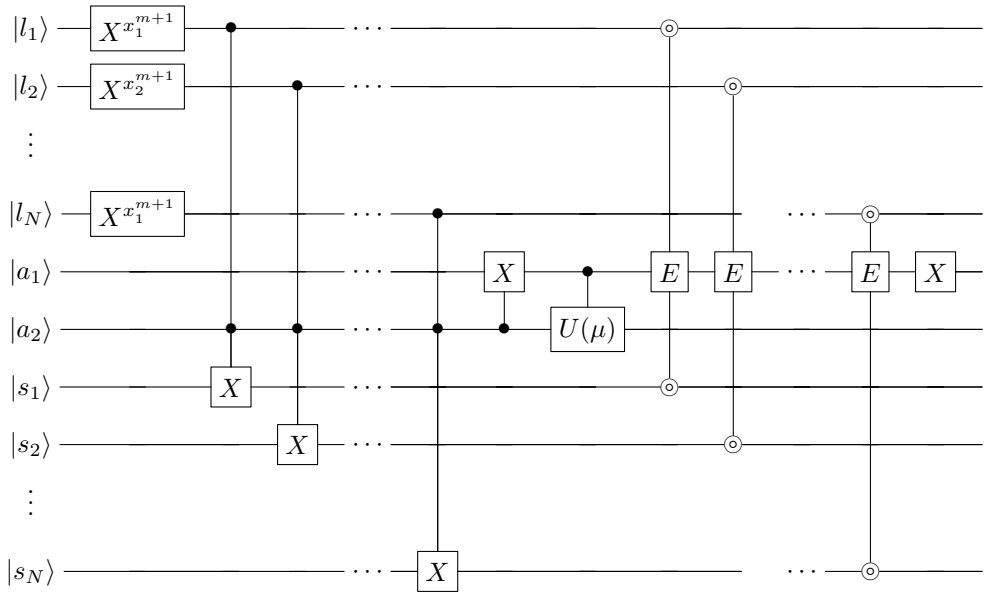
Avec la table de vérité suivante :

x ; y		s
0	0	1
0	1	0
1	0	0
1	1	1

On peut expliciter **E** matriciellement :

$$\mathbf{E} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

On construit alors le circuit total



On arrive enfin à l'état suivant

$$\frac{1}{\sqrt{M}}\sum_{k=1}^{m+1}|x^{m+1}\rangle\otimes|00\rangle\otimes|x^k\rangle+\frac{\sqrt{M-(m+1)}}{\sqrt{M}}|x^{m+1}\rangle\otimes|11\rangle\otimes|x^{m+1}\rangle$$

Il ne reste qu'une dernière étape, de remettre le registre de chargement à zéro ainsi que le registre de stockage de la branche de processus. C'est assez simple, il suffit d'inverser les opérations faites auparavant sur ces qbits. On applique les comparaisons successivement puis un $\mathbf{X}$ controllé par a_2 pour avoir $|11\rangle \longrightarrow |01\rangle$ puis le control-$\mathbf{X}$ et enfin l'inversion du registre de chargement. On a donc obtenu par récurrence l'état :

$$\phi^{(m+1)}=\frac{1}{\sqrt{M}}\sum_{k=1}^{m+1}|0\rangle\otimes|00\rangle\otimes|x^k\rangle+\frac{\sqrt{M-(m+1)}}{\sqrt{M}}|0\rangle\otimes|01\rangle\otimes|0\rangle$$

En conclusion, on peut résumer une itération de l'algorithme par les étapes suivantes : chargement du vecteur d'état dans le registre de chargement, copie de ce vecteur dans le registre de stockage, séparation de la branche de processus en deux pour isoler le vecteur, on fait basculer une des sous branches dans la somme pour stocker le vecteur avec ceux des itérations précédentes, on réinitialise le registre de chargement et celui de stockage de la branche de processus.

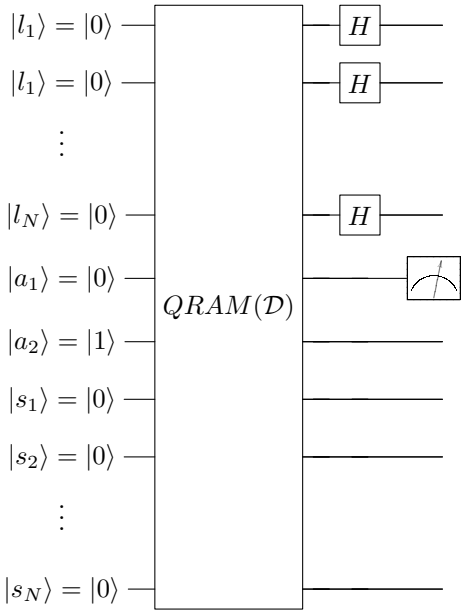
RAM Quantique avec encodage en amplitude

Dans le cadre du machine learning quantique il est parfois utile d'utiliser des entrées basées sur de l'encodage en amplitude. Il faut donc mettre en place une architecture de QRAM permettant de stocker la superposition des données.

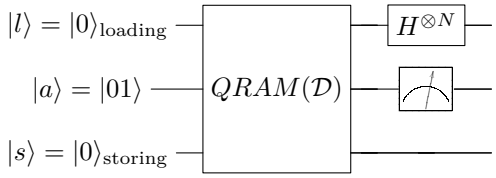
On se place dans le cas d'un dataset $\mathcal{D}=(x_k)_{k\in 1,M}$ avec $x_k\leq 1$ pour tout $k\in 1,M$ des nombres réels que l'on veut encoder en amplitude. On suppose $M=2^n$ pour simplifier les choses. L'idée est alors de réutiliser le principe de QRAM avec encodage en base en encodant les valeurs du dataset sous forme de fraction binaire. On note alors $\mathcal{D}'=(x^k)_{k\in 1,M}$ où $x^k=\mathrm{bin}_N(x_k)\in\{0,1\}^N$ et on cherche à obtenir en sortie de la RAM quantique l'état :

$$|\psi\rangle=\frac{1}{\sqrt{M}}\sum_{k=1}^{M}|k\rangle\otimes|00\rangle\otimes|x^k\rangle$$

Le circuit suivant réalise de manière simple cette étape avec probabilité $p(a_1=0)$, il faut donc répéter le processus le cas échéant. :



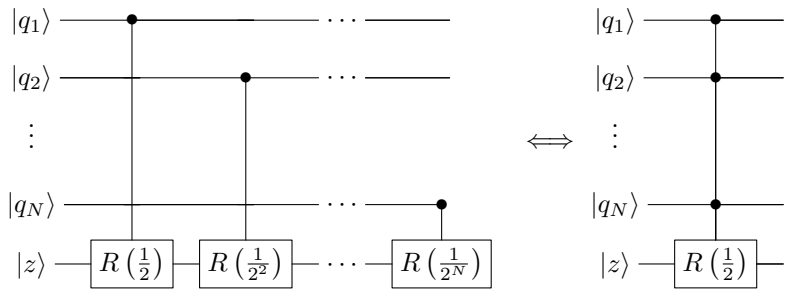
Dans la suite nous écrirons le circuit de cette manière simplifiée :



Ensuite, on cherche à obtenir le résultat suivant :

$$|\psi\rangle = \frac{1}{\sqrt{M}} \sum_{k=1}^{M} |k\rangle \otimes (\sqrt{1-|x_k|^2}|0\rangle + x_k|1\rangle)|1\rangle \otimes |x^k\rangle$$

Cette opération est faisable avec N portes control-**R** d'angle de rotation $\frac{1}{2^i}$ pour chaque qbit s_i pour $i \in 1, N$. Voici un exemple de rotation sur le qbit $|z\rangle$ controllé par les qbits $|q_1\rangle, \cdots, |q_N\rangle$.

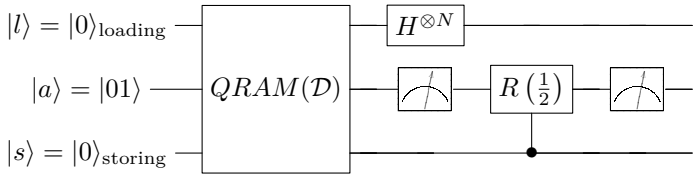


Chaque rotation approche de plus en plus l'état de $|1\rangle$ ceci étant d'autant plus vrai que le nombre de qbit N est grand. En effet, un calcul simple de somme géométrique montre que

$$\lim_{N\longrightarrow\infty}\sum_{i=1}^{N}\frac{1}{2^i}=\lim_{N\longrightarrow\infty}\frac{2^N-1}{2^N}=1$$

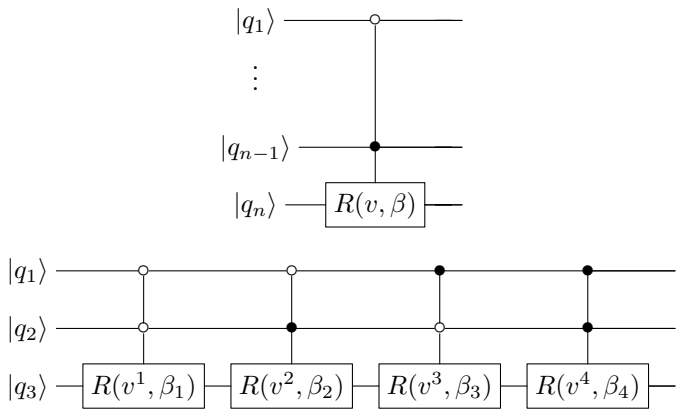
Il ne reste plus qu'à mesurer le qbit $|a_1\rangle$ pour obtenir l'état suivant (on réitère si on obtient 0 lors de la mesure). On a une probabilité p_{acc} de tomber sur 1.

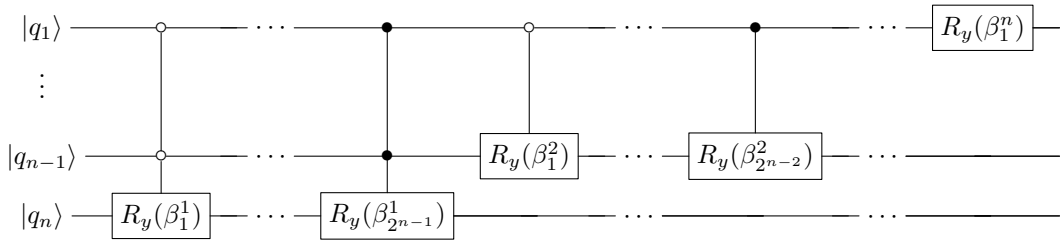
$$|\psi\rangle=\frac{1}{\sqrt{Mp_{\text{acc}}}}\sum_{k=1}^{M}|k\rangle\otimes x_k|11\rangle\otimes|x^k\rangle$$



Préparation d'état en temps linéaire

Pour un ordinateur quantique de n qbit, la profondeur théorique minimale d'un circuit permettant de préparer des états arbitraires est connu comme étant de $\frac{2^n}{n}$ avec les algorithmes connus y arrivant en un peu moins de 2^n opérations en parallèle au prix de portes à 2 qbits qui sont comme nous l'avons vu cher en terme de bruit. Ici, nous allons développé l'idée présentée par Möttönen qui consiste à partir d'un état $|\varphi\rangle$ et d'arriver à l'état de base $|0\cdots0\rangle$. Le principe est le suivant : utiliser des **rotations multi-controlées** pour contrôler la rotation d'un qbit $|q_i\rangle$ en fonction des états des qbits antérieurs $|q_1\rangle,\cdots,|q_{i-1}\rangle$





$$\beta_j^s = 2\arcsin\left(\frac{\sqrt{\sum_{l=1}^{2^{s-1}} |\alpha_{(2j-1)2^{s-1}+l}|^2}}{\sqrt{\sum_{l=1}^{2^s} |\alpha_{(j-1)2^{s-1}+l}|^2}}\right)$$

4.5.2 Encodage à base d'hamiltoniens

Au lieu d'encoder les données comme nous l'avons fait précédemment via les états quantiques, il est possible de choisir une approche plus implicite en utilisant l'évolution d'un système quantique. Ainsi, nous ne préparons plus un état quantique contenant de l'information mais définissant une évolution d'un état. L'encodage hamiltonien permet d'associer l'hamiltonien d'un système avec une matrice représentant les données d'un dataset par exemple. Nous serons amené à utiliser des techniques permettant de transformer une matrice en un opérateur hermitien, l'extraction des valeurs propres ou la multiplication par une vecteur d'amplitude est alors faisable.

Par conséquent, nous cherchons à être capable d'implémenter une évolution de la forme

$$|\psi'\rangle = e^{-iH_A t}|\psi\rangle$$

sur un ordinateur quantique où H_A représente un hamiltonien encodant une matrice hermitienne A de même dimension. L'état $|\psi'\rangle$ est l'état final du système contenant les données encodées dans l'hamiltonien. L'idée de la simulation Hamiltonienne peut se résumer par le problème suivant : Soit un hamiltonien H et un état $|\psi\rangle$. Soit $t \in \mathbb{R}+$ le temps d'évolution. Soit $\varepsilon > 0$, existe-t-il un algorithme qui implémente l'équation ci-dessus tel que l'écart entre l'état final de l'algorithm $|\bar{\psi}\rangle$ et l'état désiré $|\psi'\rangle$ soit ε-petit i.e $\||\psi'\rangle - |\bar{\psi}\rangle\| \leq \varepsilon$ pour une norme donnée sur l'espace des qbits.

Soit un hamiltonien H décomposable en hamiltoniens élémentaires facimement simulable, $H = \sum_{k=1}^{M} H_k$. Dans le cas où les H_k ne commutent pas, la formule $e^{-i\sum_{k=1}^{M} H_k t} = \prod_{k=1}^{M} e^{-iH_k t}$ n'est pas applicable. Cependant, en utilisant la formule de Suzuki-Trosker d'ordre 1, nous avons l'approximation suivante

$$e^{-i\sum_{k=1}^{M} H_k t} = \prod_{k=1}^{M} e^{-iH_k t} + O(t^2)$$

Pour des petits temps d'évolution, la formule est valide asymptotiquement. Pour des temps arbitraires, il est possible d'utiliser des pas de discrétisation Δt arbitrairement petit, en effet,

$$e^{-iHt} = (e^{-iH\Delta t})^{\frac{1}{\Delta t}} = \prod_{k=1}^{M} e^{-iH_k \Delta t}$$

Un exemple de décomposition d'un hamiltonien est celle sous forme de produit tensoriel de matrices de Pauli, tout hamiltonien H peut s'écrire sous la forme

$$H = \sum_{k_1,,k_n \in \{1,x,y,z\}} a_{k_1,\cdots,k_n}(\sigma_{k_1} \otimes \cdots \otimes \sigma_{k_n})$$

avec

$$a_{k_1,\cdots,k_n} = \frac{1}{2^n}\mathrm{tr}((\sigma_{k_1} \otimes \cdots \otimes \sigma_{k_n})H)$$

Cette décomposition est formée de 4^n termes en général mais lorsque les intéractions du système ne sont que locales (les σ_i sont presque toutes égales à I la matrice identité) nous pouvons espérer réduire le nombre de terme dans la somme ci-dessus.

Dans certains cas précis d'hamiltoniens strictement locaux, ces derniers peuvent être simulés en temps logarithmique de leur dimension et dans le cas de l'endocage hamiltonien que le dataset peut être encodé en temps logarithmique de sa dimension. En utilisant la notion d' "hamiltonien s-creux" c'est à dire que chaque ligne et colonne possède au plus s coefficients non nuls.

5 Annexes

5.1 Espace de Hilbert

Un espace de Hilbert E est un espace vectoriel complexe muni d'un produit scalaire hermitien, complet.

Le produit scalaire hermitien $(x, y) \mapsto \langle x|y\rangle \in \mathbb{C}$ est défini sur $E \times E$ ainsi :

- $\forall x, y\ \langle y|x\rangle = \overline{\langle x|y\rangle}$ (forme hermitienne)
- $\forall x, y, z\ \langle x|ay + z\rangle = a\langle x|y\rangle + \langle x|z\rangle$ (linéaire à droite)
- $\langle ax + y|z\rangle = \overline{a}\langle x|y\rangle + \langle x|z\rangle$ (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)

5.2 Espace dual

L'espace dual d'une espace vectoriel E est l'espace vectoriel E* qui est l'ensemble des forme linéaires sur E. Plus précisemment, il s'agit du dual "algébrique". Le dual topologie E' est l'ensemble des formes linéaire continues (il 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. Dans notre cadre complexe, les notations utilisées sont plus complètes et définies au paragraphe 2.1.

5.3 Convention d'Einstein

Cette convention d'écriture est très utilisée dans la littérature, il me semble important de la mentionner ici.

• il est d'abord courant d'écrire $v = v^i e_i$ à la place de $v = \sum_i v^i e_i$ (l'indice se place en exposant pour les composantes du vecteur v)

• lorsqu'on fait un produit matriciel ou une contraction (voir paragraphe suivant), on note souvent en indices les index qui apparaîssent dans la somme.

5.4 Produit tensoriel

5.4.1 Tenseurs

On utilisera la convention d'Einstein* pour écrire les formules qui suivent. 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 :

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

(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 :

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

On comprend donc mieux la notation de $\otimes^p E$ qui est en fait 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 :

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

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 :

$$a \overline{\otimes} b = a_k b_k = \langle a|b\rangle$$

$$(A \overline{\otimes} b)_i = A_{ik} b_k = (Ab)_i$$

$$(A \overline{\otimes} B)_{ij} = A_{ik} B_{kj} = (AB)_{ij}$$

5.4.4 Produit de Kronecker

Une façon de représenter un produit tensoriel dans le cas de matrices (c'est le seul cas qui nous intéresse) est le produit de Kronecker. 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 bien 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 :

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

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

Propriétés

On a avec ce produit tensoriel les propriétés suivantes :

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

5.5 Contractions de tenseurs

Il faut d'abord définir ce qu'on appelle une contraction : il s'agit simplement d'une généralisation du produit matriciel standard à des tenseurs. Cela consiste donc à faire une somme sur un ensembl d'indices muets. Prenons un exemple. On considère l'état :

$$\Psi = |\downarrow\uparrow\uparrow\uparrow\rangle + |\uparrow\downarrow\uparrow\uparrow\rangle + |\uparrow\uparrow\downarrow\uparrow\rangle + |\uparrow\uparrow\uparrow\downarrow\rangle$$

Comme souvent, on n'écrit pas les coefficients de normalisation. Ici, avec les notations habituelles : $|\uparrow\rangle = |0\rangle$ et $|\downarrow\rangle = |1\rangle$. Voici une représentation MPS de cet état (avec la convention d'Einstein) :

$$\Psi^{\alpha\beta\gamma\delta} = \sum_{ijk} A_i^\alpha B_{ij}^\beta C_{jk}^\gamma D_k^\gamma$$

où l'on fait une contraction selon les indices i, j, k avec :

$$A^\uparrow = \begin{pmatrix} 1 & 0 \end{pmatrix}, A^\downarrow \begin{pmatrix} 1 & 0 \end{pmatrix},$$

$$B^\uparrow = C^\uparrow = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, B^\downarrow = C^\downarrow = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

et

$$D^\uparrow \begin{pmatrix} 0 \\ 1 \end{pmatrix}, D^\downarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Comme expliqué dans l'introduction, $\Psi^{\alpha\beta\gamma\delta}$ donne la "probabilité" (attention, on a pas normalisé, donc il s'agit ici plutôt d'une amplitude) pour le système de se trouver dans l'état $|\alpha\beta\gamma\delta$. Il peut être intéressant de faire le calcul pour observer, par exemple, que $\Psi^{\uparrow\downarrow\uparrow\uparrow} = 1$ en développant le produit. On peut représenter cela de manière plus compact :

$$\Psi = \begin{pmatrix} \uparrow & \downarrow \end{pmatrix} \begin{pmatrix} \uparrow & \downarrow \\ 0 & \uparrow \end{pmatrix} \begin{pmatrix} \uparrow & \downarrow \\ 0 & \uparrow \end{pmatrix} \begin{pmatrix} \downarrow \\ \uparrow \end{pmatrix}$$

Remarque: En python :

```
import numpy as np
a=np.array([[1,2],[1,3]]) #tenseur d'ordre 2 (matrice)
b=np.array([[1,3],[2,6]]) #tenseur d'ordre 2 (matrice)
c=np.array([a,b])         #tenseur d'ordre 3

res=np.einsum('ij,kj,klm',a,b,c) #einsum permet de faire des contractions selon les indices voulus
print(res)
```

Dans cette exemple, on a fait l'opération :

$$\sum_{j,k} A_j^i B_{kj} C_k^{lm}$$

(on peut vérifier par le calcul que le terme correspondant à $(i, l, m) = (1, 1, 1)$ vaut bien 21 par exemple)

5.6 Règle de Born généralisée

On avait 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. 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 au Mermin (biblio).

5.7 Matrices particulières

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 : $\langle\Phi|\Psi\rangle = \langle\Phi|U^\dagger U|\Psi\rangle = \langle U\Phi|U\Psi\rangle$

5.8 Opérations supplémentaires

Nous présentons ici des opérateurs supplémentaires à ceux déjà présentés*.

5.8.1 Opérations non inversibles sur 1 bit

- Projection sur $|1\rangle$ (noté $\mathbf{n}$) : $\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$
  Tel que $\mathbf{n}|x\rangle = x|x\rangle$

- **Xn** $\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$
  Tel que $\mathbf{Xn}|\mathrm{x}\rangle = x|\overline{x}\rangle$

- Projection sur $|0\rangle$ (noté $\overline{\mathbf{n}}$) : $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$

- **X$\overline{\mathbf{n}}$** $\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$

5.9 Matrices de Pauli

Les matrices de Pauli sont les suivantes :

$$\sigma_X = \mathbf{X} \quad \sigma_Y = \mathbf{Y} \quad \sigma_Z = \mathbf{Z}$$

Ces matrices vérifies des propriétés intéressantes, par exemple :

- $\sigma_X^2 = \sigma_Y^2 = \sigma_Z^2 = \mathbf{1}$
- $\sigma_X\sigma_Y = -\sigma_Y\sigma_X = i\sigma_Z$ que l'on peut permuter circulairement

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

$$\forall a, \forall b, (a.\sigma)(b.\sigma) = (a.b)\mathbf{1} + i(a \times b).\sigma$$

Il se trouve que $(\mathbf{1}, \sigma_X, \sigma_Y, \sigma_Z)$ est une base de l'espace des opérateurs hermitiens. Alors, toute matrice hermitienne s'écrit de la forme :

$$A = a_0\mathbf{1} + a.\mathbf{a}\acute{e}\sigma$$

5.10 Portes universelles

5.11 Addition modulo 2

L'addition modulo 2 est définie ainsi :

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

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

On vérifie aisément les propriétés suivantes :

$$x \oplus 1 = \overline{x} \quad x \oplus 0 = x \quad x \oplus x = 0 \quad x \oplus \overline{x} = 1$$

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 d'appliquer la rotation $\mathbf{R}_\theta$ 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 :

$$\mathbf{u} \otimes \mathbf{1}|\Psi\rangle = |0\rangle \otimes |\Psi_0'\rangle + |1\rangle \otimes |\Psi_1'\rangle$$

Avec $|\Psi_0'\rangle = a|\Psi_0\rangle - b^*|\Psi_1\rangle$ 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 Mermin 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 :

$$|\Psi_0''\rangle = \mathbf{R}_\theta|0\rangle \qquad |\Psi_1''\rangle = \mathbf{R}_\theta|1\rangle$$

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 :

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

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

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

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 prémisces 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.

```
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()
```

```
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()                                    #
                                                    #
```

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 :

```
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 :

$$(\mathbf{H}\otimes\mathbf{H})(\mathbf{X}\otimes\mathbf{X})|00\rangle = \mathbf{H}|1\rangle\otimes\mathbf{H}|1\rangle = \frac{1}{2}(|0\rangle-|1\rangle)(|0\rangle-|1\rangle) = \frac{1}{2}(|00\rangle-|01\rangle-|10\rangle+|11\rangle)$$

On applique U_f à ce terme, donc par linéarité à chaque terme :

$$U_f(\mathbf{H}\otimes\mathbf{H})(\mathbf{X}\otimes\mathbf{X})|00\rangle = \frac{1}{2}(|0\rangle|f(0)\rangle-|0\rangle|\overline{f(0)}\rangle-|1\rangle f(1)+|1\rangle|\overline{f(1)}\rangle)$$

On factorise alors :

$$\frac{1}{2}(|0\rangle\pm|1\rangle)(|f(0)\rangle-|\overline{f(0)}\rangle)$$

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

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

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 :

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

On admet ensuite la propriété suivante des portes Hadamard (détails de calcul donnés dans le Mermin*(pg 51):

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

D'où l'état final :

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

On remplace alors $f(x)+x\cdot y=(a+y)\cdot x$, puis par des échanges de sommes et produits, tous les termes disparaissent sauf quand y = a (détails dans le Mermin* pg52), cela donne :

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

5.16 Transformée de Fourier discrète

On rappelle ici la définition de la TF discrète classique, pour pouvoir implémenter ensuite sa version quantique*

Soit un N-uplet de nombres complexes : $x=(x_1,x_2,\cdots x_{N-1})$. On définit la TF discrète par l'opérateur qui agit de la façon suivante sur x :

$$TFD(x)_k = y_k = \frac{1}{\sqrt{N}}\sum_{j=0}^{N-1}x_j e^{\frac{2ij\pi k}{N}}$$

Les nombres $y=(y_1,\cdots,y_{N-1})$ définis ainsi sont aussi complexes.

La TF discrète inverse est définie comme l'application réciproque de la TF discrète.

5.17 Décomposition de matrices

5.17.1 Décomposition QR

Définition 5.1. Soit $M\in\mathcal{M}_n(\mathbb{C})$. On note T_n^{++} l'ensemble des matrices carrées de taille n triangulaires supérieures et à diagonale strictement positive. On note $O_n(\mathbb{R})$ le groupe des matrices orthogonales.

Théorème 5.2. Soit $P\in\mathcal{M}_n(\mathbb{C})$ inversible, alors $\exists!R\in T_n^{++},\exists!Q\in O_n(\mathbb{R})$ tel que

$$P=QR$$

Démonstration :

• Existence :
Comme P est inversible, ses colonnes (p_i) constituent une base de $\mathbb{C}^n$. Par le procédé d'orthonormalisation de Gram-Schmidt, on peut construire une base orthonormé (c_i) à partir des colonnes de P. Notons R la matrice de passage de la base construite à celle des colonnes de P et Q la matrice des (c_i) dans la base canonique, on obtient $P=QR$ où $Q\in O_n(\mathbb{R})$. Par le procédé de Gram-Schmidt, on a aussi $\forall i\in 1\cdots n$,Vect$(p_1\cdots p_i)$=Vect$(c_1\cdots c_i)$. Donc $R\in T_n$. La strict positivité de la diagonale provient de la construction d'une base de Gram-Schmidt qui vérifie : $\langle p_i|c_i\rangle>0$

• Unicité :
Si $Q_1R_1 = Q_2R_2$, alors $R_1R_2^{-1} = Q_1^{-1}Q_2$. Comme T_n^{++} et $O_n(\mathbb{R})$ sont des groupes multiplicatifs,

$$R_1R_2^{-1} = Q_1^{-1}Q_2 \in T_n^{++} \cap O_n(\mathbb{R}) = \{I_n\}$$

d'où le résultat.

Exemple 5.1. Soit A la matrice inversible :

$$A = \begin{pmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{pmatrix}$$

On peut par exemple construire explicitement une base orthonormé grâce au procédé de Gram-Schmidt, ses vecteurs sont donnés (de manière unique) par :

$$e_1 = \frac{p_1}{||p_1||}, e_i = \frac{p_i - \sum_{j=1}^{2} \langle p_j|e_j\rangle e_j}{||p_i - \sum_{j=1}^{2} \langle p_j|e_j\rangle e_j||}$$

On trouve

$$Q = \begin{pmatrix} 6/7 & 3/7 & -2/7 \\ 3/7 & -2/7 & 6/7 \\ -2/7 & 6/7 & 3/7 \end{pmatrix}$$

Par unicité de la décomposition QR, $R = Q^{-1}P$ donc finalement :

$$P = \begin{pmatrix} 6/7 & 3/7 & -2/7 \\ 3/7 & -2/7 & 6/7 \\ -2/7 & 6/7 & 3/7 \end{pmatrix} \begin{pmatrix} 14 & 21 & -14 \\ 0 & -175 & 70 \\ 0 & 0 & 35 \end{pmatrix}$$

5.17.2 Décomposition en valeurs singulières

Une autre décomposition possible des matrices et très utiles que nous abordons dans ce guide lorsque nous parlons de Matrix Product State est la décomposition en valeurs singulières. C'est une méthode qui peut s'avérer pratique dans certains cas où la matrice n'est pas diagonalisable car elle permet d'avoir une décomposition pratique.

Définition 5.3. Soit $M \in \mathcal{M}_n(\mathbb{C})$. On appelle **valeurs singulières** de M les racines carrées valeurs propres de la matrice $M^\dagger M$. En effet, la matrice $M^\dagger M$ est une matrice de $S_n^+(\mathbb{C})$ (l'ensemble des matrices symétrique positive), elle admet donc n valeurs propres positives ou nulles comptées avec multiplicité.

Théorème 5.4. Décomposition en valeurs singulières (SVD)
Soit $M \in \mathcal{M}_n(\mathbb{C})$. M peut se décomposer de la forme suivante $M = USV$ où $U, V \in U_n(\mathbb{C})$ (l'ensemble des matrices unitaires d'ordre n) et S est une matrice diagonale dont les coefficients diagonaux sont les valeurs singulières de M.
Les colonnes de U sont les vecteurs propres orthogonaux de $MM^\dagger$ et les colonnes de V sont les vecteurs propres orthogonaux de $M^\dagger M$ à unité complexe près.

Exemple 5.2. Soit $A = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix}$ alors $A^\dagger A = \begin{bmatrix} 9 & 0 \\ 0 & 4 \end{bmatrix}$ donc les valeurs singulières de A sont $\sigma_1 = 3, \sigma_2 = 2$. Ensuite, on trouve U et V avec les vecteurs propres de $AA^\dagger$ et $A^\dagger A$ à constante près. On trouve comme vecteurs propres de $AA^\dagger$, $x_1 =^t \big(1, 0\big)$, $x_2 =^t \big(0, 1\big)$ et pour $A^\dagger A$, $y_1 =^t \big(1, 0\big)$, $y_2 =^t \big(0, 1\big)$ donc $u_i = \lambda_i x_i$, $v_i = \mu_i y_i$. Alors par SVD, $Av_i =_i u_i$ i.e $A\mu_i y_i = \lambda_i x_i$, on peut donc choisir λ_i, μ_i de façon à minimiser ou non certains signes. Ici, on prend $\lambda_i = 1$ pour $i = 1, 2$. donc

$$A\mu_1 y_1 = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix} \mu_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3\mu_1 \\ 0 \end{bmatrix} =_1 u_1$$

$$A\mu_2 y_2 = \begin{bmatrix} 3 & 0 \\ 0 & -2 \end{bmatrix} \mu_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -2\mu_2 \end{bmatrix} = \sigma_2 u_2$$

Ainsi, $\mu_1 = 1$, $\mu_2 = -1$ et on obtient

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$