MPS et simulation quantique

Léo Durand-Köllner

Avril 2021

1 Introduction

Dans ces notes, on parle de simulation d'ordinateurs quantiques sur des ordinateurs classiques. Le but est in fine d'arriver à une simulation efficace et rapide, sans pour autant avoir une simulation parfaite comme le ferait un ordinateur quantique parfait. Les ordinateurs quantiques d'aujourd'hui sont encore très bruités et peu fiables : on est encore dans l'ère *NISQ*. En tirant parti de ces imperfections et de la décohérence, on peut accélérer la simulation des ordinateurs quantiques actuels.

2 MPS

2.1 Définition

Dans la suite, on utilise extensivement la représentation dite MPS : *Matrix Product State*. On se donne un état quantique pur à N qbit, que l'on désigne par son ket d'état $|\psi\rangle$. On commence par faire une décomposition de Schmidt en séparant le qbit de poids fort avec le reste (on suppose de manière implicite que chaque μ_j varie de 1 à χ_j) :

$$|\psi\rangle = \sum_{\mu_{N-1}} \lambda_{\mu_{N-1}} |\mu_{N-1}\rangle_{[N-1]} |\mu_{N-1}\rangle_{[N-2,\dots,0]}$$

Ensuite, on décompose le qbit de poids fort dans la base canonique :

$$|\mu_{N-1}\rangle_{[N-1]} = \sum_{i_{N-1}=0}^{1} \langle i_{N-1}|\mu_{N-1}\rangle_{[N-1]} \, |i_{N-1}\rangle$$

En notant $A(N-1)^{i_{N-1}}_{\mu_{N-1}} = \lambda_{\mu_{N-1}} \langle i_{N-1}|\mu_{N-1}\rangle_{[N-1]}$,

$$|\psi\rangle = \sum_{i_{N-1}} \sum_{\mu_{N-1}} A(N-1)^{i_{N-1}}_{\mu_{N-1}} |i_{N-1}\rangle |\mu_{N-1}\rangle_{[N-2,\dots,0]}$$

Maintenant, on répète l'opération pour le qbit $N-2$, sauf que les coefficients de Schmidt (les λ) dépendent aussi de μ_{N-1} :

$$|\mu_{N-1}\rangle_{[N-2,\dots,0]} = \sum_{\mu_{N-2}} \lambda_{\mu_{N-1},\mu_{N-2}} |\mu_{N-2}\rangle_{[N-2]} |\mu_{N-2}\rangle_{[N-3,\dots,0]}$$

$$|\mu_{N-2}\rangle_{[N-2]} = \sum_{i_{N-2}=0}^{1} \langle i_{N-2}|\mu_{N-2}\rangle_{[N-2]} \, |i_{N-2}\rangle$$

Donc en notant $A(N-2)^{i_{N-2}}_{\mu_{N-1},\mu_{N-2}} = \lambda_{\mu_{N-1},\mu_{N-2}} \langle i_{N-2}|\mu_{N-2}\rangle_{[N-2]}$,

$$|\psi\rangle = \sum_{\substack{i_{N-1} \\ i_{N-2}}} \sum_{\substack{\mu_{N-1} \\ \mu_{N-2}}} A(N-1)^{i_{N-1}}_{\mu_{N-1}} A(N-2)^{i_{N-2}}_{\mu_{N-1},\mu_{N-2}} |i_{N-1}\rangle |i_{N-2}\rangle |\mu_{N-2}\rangle_{[N-3,\dots,0]}$$

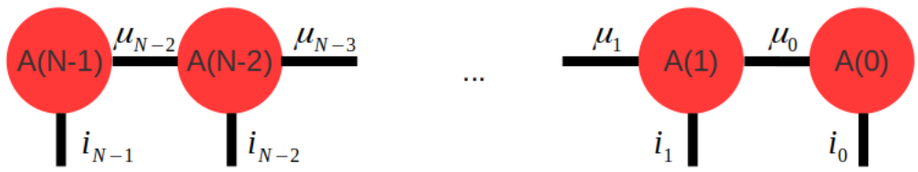

FIGURE 1 – Représentation diagrammatique d'un MPS

Si on continue à itérer le procédé, on se retrouve avec la forme MPS :

$$|\psi\rangle = \sum_{i_{N-1},\ldots,i_0} \sum_{\mu_{N-2},\ldots,\mu_0} A(N-1)^{i_{N-1}}_{\mu_{N-2}} A(N-2)^{i_{N-2}}_{\mu_{N-2},\mu_{N-3}} \ldots A(1)^{i_1}_{\mu_1,\mu_0} A(0)^{i_0}_{\mu_0} |i_{N-1}\ldots i_0\rangle$$

Au centre, on a des tenseurs de rang 3, puis aux extrémités des tenseurs de rang 2 (matrices). Comme on peut le constater, effectuer des opérations analytiquement sur une telle forme n'a rien de pratique. C'est pourquoi on préfère utiliser des diagrammes tensoriels. Dans cette représentation, le nombre de "pattes" de chaque tenseur correspond à son rang.

2.2 Contraction tensorielle

Pour deux tenseurs A et B qui partagent un indice i prenant les mêmes valeurs, la contraction tensorielle est le tenseur indexé par tous les autres indices de A et B et où l'on a sommé sur i le produit de A et B. Par exemple, si on se donne $(A)_{\alpha,\beta,\gamma}$ et $(B)_{\delta,\alpha,\epsilon}$, où α varie entre 1 et p, la contraction tensorielle de A et B est le tenseur $(C)_{\beta,\gamma,\delta,\epsilon}$ tel que $\forall \beta, \gamma, \delta, \epsilon$:

$$C_{\beta,\gamma,\delta,\epsilon} = \sum_{\alpha=1}^{p} A_{\alpha,\beta,\gamma} B_{\delta,\alpha,\epsilon}$$

Sous forme diagrammatique, cette opération se représente simplement par une interconnexion des pattes de deux tenseurs, où la patte concernée correspond à l'indice de sommation. Le tenseur résultant a un rang égal au nombre de pattes sortantes de l'ensemble des deux tenseurs contractés. La figure 1 prend tout son sens : MPS n'est en fait rien d'autre qu'une contraction tensorielle de N tenseurs, et chaque tenseur (que l'on appelle *site*) correspond à un qbit.

2.3 Calcul rapide d'opérations quantiques

A ce stade-là, on peut se demander : quel peut bien être l'intérêt de la décomposition en MPS ? On obtient une subdivision compliquée avec plein d'indices, et on ne sait pas bien manipuler un tel état... Et bien les MPS permettent en fait de simuler très efficacement les portes quantiques unitaires à 1 et 2 qbits ! Malheuresement, cela a un prix : on perd en précision et le résultat obtenu n'est pas exact. Cependant, on peut quantifier cette erreur et ajuster les paramètres de notre MPS pour essayer de mimer les ordinateurs quantiques actuels (avec leurs défauts).

Dans la suite, on impose que $\forall j, \chi_j \leq \chi$ où χ est un paramètre qu'il faudra ajuster. Intuitivement, chaque χ_j correspond au degré d'intrication entre les qbits j et $j+1$, donc notre condition revient à limiter l'intrication du système. Pour quantifier cette intrication, on utilise la mesure $E_\chi = \log_2(\chi)$.

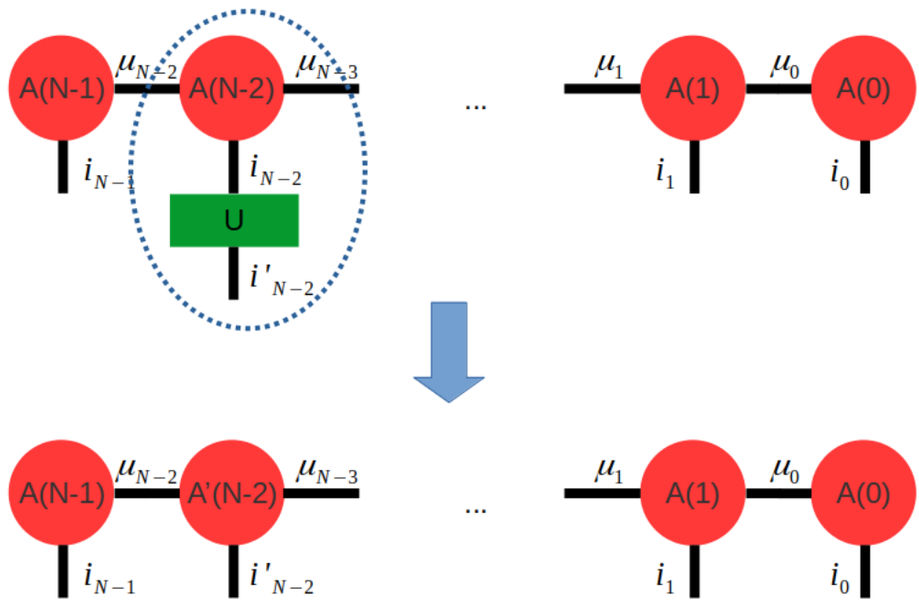

FIGURE 2 – Application d'une porte à 1 qbit sur un MPS

2.3.1 Effet d'une porte à 1 qbit

Si on considère l'unitaire U sur 1 qbit, pour obtenir le tenseur résultant il suffit de considérer la contraction tensorielle de l'unitaire et du tenseur concerné sur la branche i_k. La figure 2 illustre le processus. Ce qui est remarquable, c'est que l'on a seulement besoin de calculer les

$$A'(k)^{i'_k}_{\mu_k,\mu_{k-1}} = \sum_{i_k} U_{i'_k,i_k} A(k)^{i_k}_{\mu_k,\mu_{k-1}}$$

pour chaque μ_k et chaque μ_{k-1}, donc le coût total de l'opération est en $\mathcal{O}(\chi^2)$.

2.3.2 Effet d'une porte à 2 qbit

Pour compléter notre analyse, il faut s'attarder sur les portes à 2 qbits (les portes à 1 qbit et 2 qbit sont universelles). Ce cas est un peu plus compliqué. On considère l'unitaire U qui agit sur 2 qbits adjacents k et $k+1$. On réalise les opérations décrites ci-dessous. La figure 3 aide grandement à la compréhension.

1. On contracte $A(k)$ et $A(k+1)$ en un tenseur noté $(T^{i_{k+1},i_k}_{\mu_{k+1},\mu_{k-1}})$
2. On contracte U et T, en regardant U comme un tenseur de rang 4 :

$$(T')^{i'_{k+1},i'_k}_{\mu_{k+1},\mu_{k-1}} = \sum_{i_{k+1},i_k} \widetilde{U}_{i'_{k+1},i'_k,i_{k+1},i_k} T^{i_{k+1},i_k}_{\mu_{k+1},\mu_{k-1}}$$

   avec

$$\widetilde{U}_{a,b,c,d} = U_{2a+b,2c+d}$$

3. On fait une SVD sur T', en regardant T' comme une matrice $\widetilde{T'}$ de taille $2\chi \times 2\chi$:

$$\widetilde{T'}_{i,j} = \sum_{\mu_k} \widetilde{X}_{i,\mu_k} S_{\mu_k} \widetilde{Y}_{\mu_k,j}$$

   avec

$$(T')^{i'_{k+1},i'_k}_{\mu_{k+1},\mu_{k-1}} = \widetilde{T'}_{i'_{k+1}\chi+\mu_{k+1},i'_k\chi+\mu_{k-1}}$$

   On peut alors considérer $\widetilde{X}$ et $\widetilde{Y}$ comme des tenseurs d'ordre 3, de sorte que :

$$(T')^{i'_{k+1},i'_k}_{\mu_{k+1},\mu_{k-1}} = \sum_{\mu_k} X^{i'_{k+1}}_{\mu_{k+1},\mu_k} S_{\mu_k} Y^{i'_k}_{\mu_k,\mu_{k-1}}$$

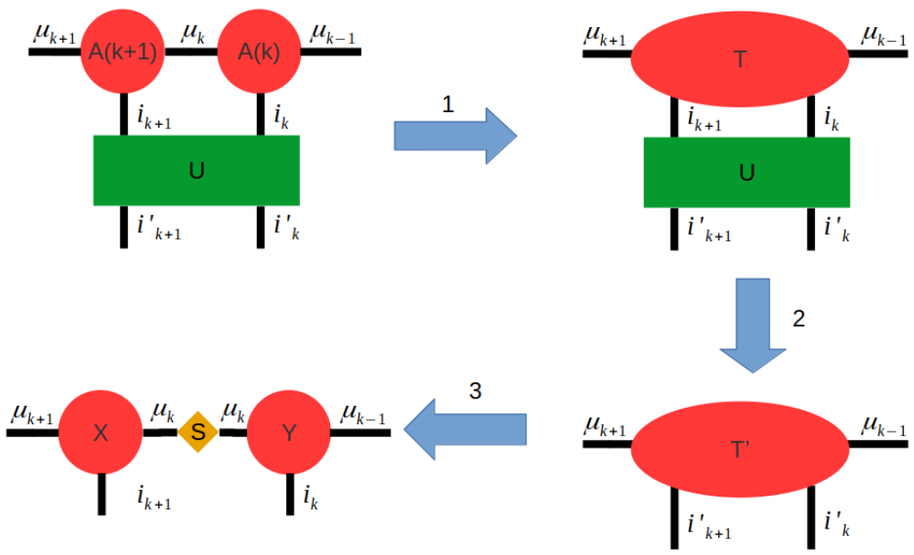

FIGURE 3 – Application d'une porte à 2 qbit sur un MPS

avec

$$X^{i'_{k+1}}_{\mu_{k+1},\mu_k} = \widetilde{X}_{i'_{k+1}\chi+\mu_{k+1},\mu_k}$$

$$Y^{i'_k}_{\mu_k,\mu_{k-1}} = \widetilde{Y}_{\mu_k,i'_k\chi+\mu_{k-1}}$$

On se retrouve avec une décomposition en 2 tenseurs séparés par une matrice, et avec μ_k qui varie au plus entre 1 et 2χ. Autrement dit, l'application de la porte unitaire U a pu augmenter le degré d'intrication entre les qbits k et $k+1$: c'est logique. C'est là où l'on fait intervenir une approximation : on ne conserve que les χ plus grandes valeurs singulières de S. De cette manière, on continue à imposer que le degré d'intrication doit être borné par χ.

4. On regroupe S avec un des deux tenseurs, par exemple X :

$$A'(k+1)^{i'_{k+1}}_{\mu_{k+1},\mu_k} = X^{i'_{k+1}}_{\mu_{k+1},\mu_k} S_{\mu_k}$$

$$A'(k)^{i'_k}_{\mu_k,\mu_{k-1}} = Y^{i'_k}_{\mu_k,\mu_{k-1}}$$

Au final, les étapes 1, 2 et 4 se font en $\mathcal{O}(\chi^2)$, mais l'étape la plus gourmande en calculs est l'étape 3, qui demande $\mathcal{O}(\chi^3)$ opérations à cause de la décomposition en valeurs singulières.

3 Implémentation

Dans cette section, on détaille une implémentation des MPS avec les techniques vues dans la section 2. Le code est en `Python`, et utilise les bibliothèques `Numpy` (pour les calculs tensoriels) ainsi que `Matplotlib` (pour afficher les courbes). Le code est disponible au lien suivant : `https://github.com/LeoDK/Quantum-Computing-Lab/blob/leo/MPS/mps.py`.

3.1 Représentation du MPS

3.1.1 Initialisation

On commence par détailler la représentation et les conventions adoptées pour représenter le MPS. On va représenter notre ordinateur quantique simulé

sous la forme d'une classe `QuantumComputer`. Pour créer une instance de cet objet, on a besoin de deux paramètres : N et χ.

```
class QuantumComputer :
    def __init__ (self, N, khi):
        self.N = N
        self.khi = khi
```

Dans la suite du constructeur, pour représenter notre MPS, on va simplement utiliser une liste (chaque élément correspond à un qbit). Petite particularité : la liste va en fait contenir $N+2$ éléments. Les qbits aux deux extrémités sont fictifs et ne sont pas utilisés. Cela permet de facilement implémenter les opérations sur les tenseurs sans se préoccuper de leur taille, puisque tous les qbits utiles sont alors représentés par des tenseurs d'ordre 4.

```
self.mps = list()
self.mps.append ( np.zeros((2, khi), dtype=complex) )
for i in range ( N ):
    self.mps.append ( np.zeros((2, khi, khi), dtype=complex) )
self.mps.append ( np.zeros((2, khi), dtype=complex) )
```

On veut démarrer l'ordinateur quantique dans l'état $|0\rangle$: pour cela on utilise une représentation simple de $|0\rangle$ avec $A(0)^{i_0}_{\mu_0} = \delta_{\mu_0,0}\delta_{i_0,0}$, $A(1)^{i_1}_{\mu_1,\mu_0} = \delta_{\mu_1,0}\delta_{\mu_0,0}\delta_{i_1,0}$, etc... (avec δ le symbole de Kronecker).

```
self.mps[0][0][0] = 1
for tensor in self.mps[1:-1]:
    tensor[0][0][0] = 1
self.mps[-1][0][0] = 1
```

3.1.2 Portes à 1 qbit

Maintenant, on s'attarde sur les portes à 1 qbit. C'est assez simple à implémenter : il suffit de faire une contraction tensorielle. Pour cela, on utilise la fonction `einsum` de `Numpy`, qui permet de faire des sommations selon la convention d'Einstein. Les indices en commun entre les deux tenseurs sont sommés, et l'ordre des indices conservés est l'ordre alphabétique. On n'oublie pas qu'il y a un offset de 1 à cause des qbits fictifs.

```
def gate_1qbit (self, U, qbit):
    self.mps[qbit+1] = np.einsum('ij,jkl', U, self.mps[qbit+1])
```

3.1.3 Portes à 2 qbits sur qbits adjacents

Pour les portes à 2 qbits, on respecte scrupuleusement les 4 étapes de la section 2.3.2. On appelle notre méthode `gate_2qbit_adj` puisque l'on fait une opération entre 2 qbits *adjacents*. La fonction prend en paramètre l'unitaire U ainsi que le premier des 2 qbits sur lesquels on applique U.

```
def gate_2qbit_adj (self, U, qbit):
```

On commence par contracter les 2 qbits en un tenseur d'ordre 4 :

```
# Step 1
T = np.einsum('ikl,jlm', self.mps[qbit+2], self.mps[qbit+1])
```

Ensuite, on met U sous forme de tenseur et on contracte $\widetilde{U}$ avec T :

```
# Step 2
U_tilde = np.zeros((2,2,2,2), dtype=complex)
for a in range (2):
    for b in range (2):
        for c in range (2):
            for d in range (2):
                U_tilde[a][b][c][d] = U[2*a+b][2*c+d]
Tp = np.einsum('ijkl,klmn', U_tilde, T)
```

On réécrit T' sous forme de matrice, puis on effectue une SVD dont on ne garde que les χ première valeurs singulières. Ensuite, on réécrit les deux matrices $\widetilde{X}$ et $\widetilde{Y}$ sous forme de tenseurs, puis on tronque les tenseurs obtenus (pour coller au nombre de valeurs singulières).

```
# Step 3
# Reshape Tp intro matrix of size 2*khi by 2*khi,
Tp_tilde = np.zeros((2*self.khi, 2*self.khi), dtype=complex)
for i in range (2):
    for j in range (2):
        for k in range (self.khi):
            for l in range (self.khi):
                Tp_tilde[i*self.khi + k][j*self.khi + l] = Tp[i
    ][j][k][l]

# SVD
X_tilde, S, Y_tilde = np.linalg.svd (Tp_tilde)

# Truncate S
S = S[:self.khi]

# Back to tensors
X = np.zeros((2, self.khi, 2*self.khi), dtype=complex)
for i in range (2):
    for k in range (self.khi):
        for l in range (2*self.khi):
            X[i][k][l] = X_tilde[i*self.khi + k][l]

Y = np.zeros((2, 2*self.khi, self.khi), dtype=complex)
for i in range (2):
    for k in range (2*self.khi):
        for l in range (self.khi):
            Y[i][k][l] = Y_tilde[k][i*self.khi + l]

# Truncate X and Y
X = X[:, :, :self.khi]
Y = Y[:, :self.khi, :]
```

Enfin, on contracte X et S, puis on met les tenseurs obtenus dans notre MPS.

```
# Step 4
for i in range (2):
    for k in range (self.khi):
        for l in range (self.khi):
            self.mps[qbit+2][i][k][l] = X[i][k][l] * S[l]

self.mps[qbit+1] = Y
```

3.1.4 Portes à 2 qbits générales

Pour appliquer une unitaire sur 2 qbits non adjacents, on utilise la méthode suivante : on permute, à l'aide de portes SWAP, les qbits jusqu'à ce qu'il soient adjacents, on applique l'unitaire, puis on remet les qbits à leur place par une série de SWAPs. On commence par la porte SWAP sur deux qbits adjacents :

```
def swap_adj (self, qbit):
    self.gate_2qbit_adj (QuantumComputer.SWAP, qbit)
```

où l'on a pris soin de définir l'unitaire de permutation en variable de classe :

```
class QuantumComputer :
    SWAP = np.array([[1,0,0,0],[0,0,1,0],[0,1,0,0],[0,0,0,1]])
```

On peut à présent amener les qbits côte à côte :

```
a = min(qbit1, qbit2)
b = max(qbit1, qbit2)

for i in range (a, b-1):
    self.swap_adj (i)
```

On peut ensuite appliquer U. Petite précision : si la porte n'est pas symétrique (par exemple un CNOT), on veut que `qbit1` soit bien le premier qbit et `qbit2` le deuxième, donc si `qbit2` est avant `qbit1` sur le MPS, il faut SWAPper `qbit1` et `qbit2`.

```
# In case of non symmetric gates
if qbit2 < qbit1:
    self.swap_adj (b-1)

self.gate_2qbit_adj (U, b-1)
```

Enfin, on fait exactement les mêmes SWAP, mais dans l'autre sens.

```
if qbit2 < qbit1:
    self.swap_adj (b-1)

for i in range (b-2, a-1, -1):
    self.swap_adj (i)
```

Finalement, le code obtenu est :

```
def gate_2qbit (self, U, qbit1, qbit2):
    a = min(qbit1, qbit2)
    b = max(qbit1, qbit2)

    # First swap adjacent qbits to bring qbit1 near to qbit2
    for i in range (a, b-1):
        self.swap_adj (i)

    # In case of non symmetric gates
    if qbit2 < qbit1:
        self.swap_adj (b-1)

    self.gate_2qbit_adj (U, b-1)

    # Then unswap everything
    if qbit2 < qbit1:
        self.swap_adj (b-1)

    for i in range (b-2, a-1, -1):
        self.swap_adj (i)
```