Introduction au Quantique
Année 3
2023-2024

Préparation de l'environnement du travail

1. *Installez Jupyter sur votre ordinateur*

- Si vous n'avez jamais installé **Python** alors installer directement la **distribution Anaconda :** **https://www.anaconda.com/products/distribution**

- Si vous avez déjà installé Python, alors vous aurez besoin d'installer le notebook **Jupyter :** **http://jupyter.org/install.html**

a. Avec la distribution Anaconda

Une fois la distribution Anaconda installée, il faut créer un environnement :

1. Dans le menu *Démarrer,* ouvrir l'*invité de commande* : **Anaconda Prompt**
2. Créer un environnement nommé ***Quantum_Env*** avec la commande :

   `conda create --name Quantum_Env python=3.9`

   Note : 3.9 est la version de votre python. Pour connaître la version de python, il suffit d'exécuter la commande ***:*** `python --version` et d'appuyer sur *Entrée*.
3. Activer votre environnement avec la commande `activate Quantum_Env`
   (Pour désactiver votre environnement, il suffit d'exécuter la commande *: **deactivate**)*
4. Dans l'environnement ***Quantum_Env*** installer :
   1. Jupyter : `conda install jupyter`
   2. myQLM : `conda install myqlm`

   Si les deux commandes ne fonctionnent pas, utilisez les commadesde la partie b
5. Pour l'affichage des circuits quantiques, il faut installer ***ImageMagick*** : https://docs.wand-py.org/en/0.6.6/guide/install.html#install-imagemagick-windows
6. **Attention** : il faut bien suivre les instructions d'installation et vérifier les variables d'environnement sur votre PC.
7. Une fois les installations terminées, vous pouvez ouvrir le navigateur ***AnacondaNavigator*** et cliquer sur **jupyter**.
8. Un redémarrage du PC peut-être nécessaire.

b. Sans la distribution anaconda

1. Assurez-vous que le programme ***pip*** est installé sur votre ordinateur.
2. Pour cela, tapez tout simplement ***pip*** dans une console **cmd**. Normalement, le programme ***pip*** s'est installé en même temps que Python.
3. Tapez ces lignes de commande une après l'autre :

```
python -m pip install --upgrade pip
python -m pip install jupyter
```

4. Ensuite, saisissez dans votre console la commande : `pip install myQLM`
5. Pour vérifier si l'installation s'est bien déroulée, tapez dans votre *cmd* la commande suivante : `jupyter notebook`
6. Pour l'affichage des circuits quantiques, veuillez installer ***ImageMagick*** : https://docs.wand-py.org/en/0.6.6/guide/install.html#install-imagemagick-windows
7. **Attention** : il faut bien suivre les instructions d'installation et vérifier les variables d'environnement de votre PC.
8. Un redémarrage du PC peut-être nécessaire.

2. Ouverture de votre notebook

- **Jupyter** s'ouvre sur cette adresse : http://localhost:8888/tree
- L'arborescence de votre PC s'affiche
- Ouvrir le fichier TD1/…/Helloworld.ipynb

Avant d'attaquer les notebooks vous devez lire ce qui suit pour comprendre ce que nous allons faire

Introduction au Quantique
Année 3
2023-2024

Ce support est une aide pour le TP1

QLM ***(Quantum Learning Machine)*** est une application ***Atos*** qui combine du Hardware (HPC) sur lequel une note Software est mise pour aborder l'informatique quantique au travers de la simulation.

myQLM (que nous allons utiliser) est une partie Software qu'***Atos*** a mis à disposition pour permettre à des écosystèmes d'utilisateurs de développer des algorithmes quantiques ***« de manière autonome ».***

myQLM *est un environnement Python conçu pour* ***développer et simuler des programmes quantiques sur un poste de travail***.

Objectif des TDs

L'objectif des TDs, est d'écrire vos premiers programmes quantiques dans le langage mis à disposition par Atos.

myQLM **propose :**

- deux langages de programmation ***AQSAM*** et ***pyAQASM,***
- un ***simulateur*** Open Source ***PyLinalg,***
- des ***QRoutine*** (fonctions déjà implémentées),
- Possibilité de réaliser des ***circuits quantiques,***
- Des ***outils d'interopérabilités*** Open Source.

Pour avoir des fonctionnalités plus avancées, comme les fonctionnalités qui sont autorisées par *Python*, nous allons utiliser le langage ***pyAQSM.***

pyAQSM est une combinaison de ***Python*** et de ***AQASM*** *(****AQASM*** est un langage proche du langage assembleur). ***pyAQSM*** *nous* donne accès à une librairie *Python* pour implémenter des circuits quantiques et pour les simuler avec le simulateur ***PyLinalg***.

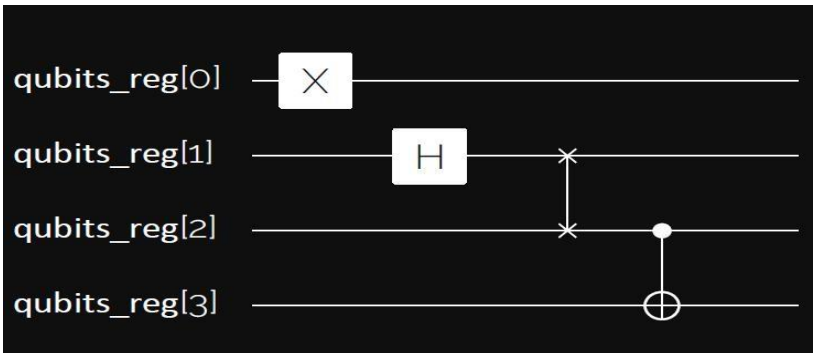
Écriture d'un circuit quantique avec pyAQASM

Pour écrire un circuit quantique avec ***pyAQASM***, nous allons commencer par importer un certain nombre de librairies (voir ci-dessous **Programme avec pyAQASAM)** :

1. L'objet central de type **Program** qui nous permettra : de définir des registres de qubit, d'appliquer les portes quantiques, de générer les circuits, etc.
2. Les 4 autres imports (**X, H, CNOT, SWAP**) sont les portes quantiques que nous allons utiliser.

Explication	Programme avec pyAQASAM
Import *functions*	**from qat.lang.AQASM import** *Program, X, H, CNOT, SWAP*
Create *your* ***program : my_program*** *est une variable*	*#Create a Program* my_program = **Program**()
Allocate *registers of qubits* *Allouer 4 qubits à notre programme* ***my_program***	*#Allocate some qubits* qubits_reg = my_program.**qalloc**(**4**)
Apply *gates* 1. ***my_program.apply(X, qubits_reg[0])*** *: appliquer la porte X sur le qubit qui se trouve à la position* ***0*** *du registre de qubits* 2. ***my_program.apply(CNOT, qubits_reg[2], ubits_reg[3]) :*** *appliquer la porte CNOT d'arité* ***2*** *sur le qubit indice* ***2*** *et* ***3****.*	#Apply some quantum Gates my_program.**apply**(**X**, qubits_reg[**0**]) my_program.**apply**(**H**, qubits_reg[**1**]) my_program.**apply**(**SWAP**, qubits_reg[**1**], qubits_reg[**2**]) my_program.**apply**(**CNOT**, qubits_reg[**2**], qubits_reg[**3**]) ***#Attention : avec la porte CNOT l'ordre des qubits est important***
Create *your* ***circuit***	*#Export this program into a quantum circuit* my_circuit = my_program.to_circ()
Display *your* ***circuit***	*#And display it!* ~~%qatdisplay my_circuit~~ %qatdisplay --svg circuit OU mu_circuit.display()

Ce programme va afficher le circuit quantique ci-dessous :



Portes quantiques constantes disponibles dans le langage pyAQASM

Nom de la porte	Mot clé	Arité
Hadamard	H	1
Pauli X	X	1
Pauli Y	Y	1
Pauli Z	Z	1
Identity	I	1
S gate	S	1
T gate	T	1
Controlled NOT	CNOT	2
SWAP	SWAP	2
iSWAP	iSWAP	2
√SWAP	SQRTSWAP	2
Toffoli	CCNOT	3

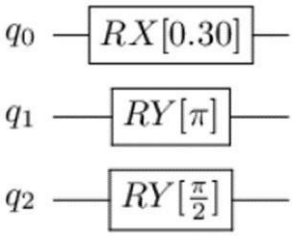
Portes quantiques paramétrées disponibles dans le langage pyAQASM

Pour rappel, l'information sur un qubit peut être représentée par la sphère tridimensionnelle de ***Bloch***. Tous les états possibles d'un qubit peuvent être définis sur cette sphère.
Sur une sphère, nous pouvons définir, par exemple, ***trois angles x, y, z***, et faire des rotations selon ces trois angles. Ces angles de rotation sont les paramètres des portes paramétrées ci-dessous :

Nom de la porte	Mot clé	Arité
Rotation x-axis	RX[θ]	1
Rotation y-axis	RY[θ]	1
Rotation z-axis	RZ[θ]	1
Phase shift	PH[θ]	1

La **Phase shift** permet d'ajouter un ***déphasage.***

Exemple d'application de portes quantiques paramétrées




```
from qat.lang.AQASM import RX, RY
Import math
#Apply some quantum Gates
my_program.apply(RX(0.3), qubits_reg[0])
my_program.apply(RY(math.pi), qubits_reg[1])
my_program.apply(RY(math.pi/2), qubits_reg[2])
```

Opérations sur les portes quantiques

Soit une porte quantique ***P***, le tableau ci-dessous décrit 4 opérations pouvant être appliquées aux portes quantiques. Les deux premières opérations sont les plus utilisées lors des créations de circuits quantiques.

Nom de l'opération	Mot clé	Exemple	Explication
control	ctrl	P.ctrl()	Ajouter un contrôle sur la porte ***P*** c.à.d la porte ***P*** sera contrôlée par un qubit.
dagger	dag	P.dag()	***Rappel*** : les portes quantiques sont des matrices unitaires. L'opération **dag** permet la création de la transposée conjuguée, d'une porte (de la matrice). Cette opération est utile lors de la manipulation d'objet plus complexe que de simples portes.
transpose	trans	P.trans()	Création de la transposée de la porte ***P***
conjugate	conj	P.conj()	Création de le conjuguée de la porte ***P***

Simulation d'un circuit avec pyAQASM

Ci-dessous un exemple d'un programme de simulation d'un circuit quantique

Explication	Programme avec pyAQASAM
Import functions Importer le simulateur de myQLM : **PyLinalg**	# import one Quantum Processor Unit Factory **from qat.qpus import** PyLinalg

Get a simulator
Stocker le simulateur dans ''une variable'' (par abus de langage) ***pqu***.
	#Create a Quantum Processor Unit,
	qpu= PyLinalg()

Create your job
Un job est l'objet soumis au simulateur pour l'exécuter. La fonction ***to_job*** *permet de simuler le circuit.*
	#Create a job
	job = my_circuit.to_job()

Submit your job *Le job est soumis avec la fonction* ***submit(job)***	**#Submit the job to the QPU** result= qpu.submit(job)
Print the result *Une boucle pour afficher les différents résultats de la simulation.*	**#Iterate over the final state vector to get all final components** sample in result: print(**"State %s amplitude %s"** % (sample.state, sample.amplitude))qubits_reg[2])

Résultat de la simulation

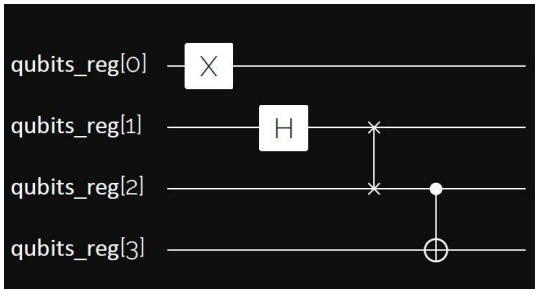
nous avons deux états possibles :

- **|1000>** avec une amplitude de 0.707 ..... = $\frac{1}{\sqrt{}}$
- **|1011>** avec l'amplitude 0.707 ..... = $\frac{1}{\sqrt{}}$

```
State |1000> amplitude
(0.7071067811865475+0j)
State |1011> amplitude
(0.7071067811865475+0j)
```

Explication des résultats :

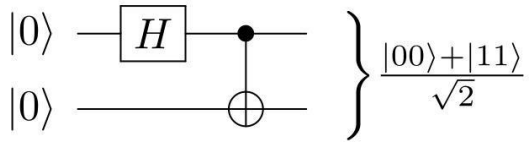
Circuit de départ :



Information importante à mettre en évidence, lors de l'allocation d'un registre de qubits, tous les qubits sont initialisés à l'état fondamental **|0>**. Dans ce circuit, les qubits d'entrée sont dans l'état **|0000>**.

Application des différentes portes :

- La première porte que nous allons appliquer est la porte **X**. Cette porte est appliquée sur le qubit |0> cela va changer son état de |0> vers l'état |1>
- Le second qubit est à |0> nous allons lui appliquer une porte **H,** cette porte transforme l'état de base |0> vers l'état de superposition $\frac{1}{\sqrt{2}}(|0>+|1>)$
- A la sortie de la porte **H** nous appliquons la porte **Swap**. Le qubit d'indice 1 va prendre la valeur du qubit d'indice 2 qui est de |0>.
- L'effet de la porte **H** sera "**Swapé**" sur le qubit d'indice 2.
- Nous appliquons un **CNOT** sur un état d'**Hadamard :**



  - Pour la partie $\frac{1}{\sqrt{2}}|0>$ nous n'allons pas appliquer la porte **CNOT** sur le qubit l'indice 3. Ce qui nous donne $\frac{1}{\sqrt{2}}|00>$.
  - Pour l'autre partie $\frac{1}{\sqrt{2}}|1>$ nous allons appliquer la porte **CNOT** sur le qubit d'indice 3. Ce qui nous donne $\frac{1}{\sqrt{2}}|11>$.

Cela explique le résultat de la simulation |1000> ou |1011> avec une 50% de chance d'obtenir l'état |1000> et 50% de chance d'obtenir |1011> avec une amplitude de $\frac{1}{\sqrt{2}}$.

Modes de simulation

Il existe trois modes de simulation avec le simulateur **PyLinalg**. Nous allons nous intéresser à seulement deux :

Pour simuler un travail nous utilisons le code suivant :

```
job = circuit.to_job(*options*) #creating job from circuit.

results= qpu.submit(job) #submitting job to QPU instance, getting results.
```

- **Simulation par défaut :** Calcul de la distribution complète, sans option dans la fonction **to_job()**.

```
job= circuit.to_job()  #creating job from circuit to get full distribution.

results= qpu.submit(job)  #submitting job to QPU instance, getting results.
```

Cette simulation permet de calculer entièrement le vecteur d'état et d'avoir accès aux **amplitudes** à travers le calcul. Dans le cas de la simulation par défaut, pour tous les états et en une seule simulation, toutes les amplitudes différentes de 0 sont calculées.

Cependant, lors de l'utilisation d'un vrai processeur quantique (un vrai QPU) nous n'aurons jamais accès à l'**amplitude.**

- **Simulation stricte**

Pour se rapprocher du comportement d'un vrai QPU, il faut utiliser *la simulation stricte*. Pour cela, il faut fixer un nombre de **shots.** Le nombre de shots est le nombre de fois que l'expérimentation est réalisée jusqu'à la mesure.

Exemple : pour **nbshots = 6,** faire 6 fois l'expérimentation. Nous appliquons successivement les portes du circuit jusqu'à construire la superposition d'état souhaitée puis réaliser la mesure.

```
job= circuit.to_job(nbshots= 6, aggregate_data=False) #creating job from circuit to get 6 measures.

results= qpu.submit(job)  #submitting job to QPU instance, getting results.
```

aggregate_data = False veut dire que nous allons nous placer au plus près de ce que fait un QPU.

***Ps** : pour nbshots = 0, nous remettra à la simulation par défaut.*

Exemple de résultats que nous pouvons obtenir avec **nbshots= 6, aggregate_data=False** :

```
Sample(state=|00>, probability=None, amplitude=None, intermediate_measurements=None, err=None)
Sample(state=|00>, probability=None, amplitude=None, intermediate_measurements=None, err=None)
Sample(state=|11>, probability=None, amplitude=None, intermediate_measurements=None, err=None)
Sample(state=|11>, probability=None, amplitude=None, intermediate_measurements=None, err=None)
Sample(state=|00>, probability=None, amplitude=None, intermediate_measurements=None, err=None)
Sample(state=|11>, probability=None, amplitude=None, intermediate_measurements=None, err=None)
```

Dans cet exemple, après la première expérimentation, nous obtenons la mesure **|00>**. Nous répétons cela 6 fois. A la fin, nous avons 50% de chance d'obtenir **|00>** et 50% d'obtenir **|11>.**

Sans **aggregate_data= False** : dans ce cas les données seront agrégées. A chaque fois que nous mesurons le même état, nous allons cumuler les données pour obtenir une estimation de la probabilité d'obtenir l'état.

```
job= circuit.to_job(nbshots= 6)  #creating job from circuit to aggregate 6 measures.

results= qpu.submit(job)  #submitting job to QPU instance, getting results.
```

```
Sample(state=|00>, probability=0.5, amplitude=None, intermediate_measurements=None, err=0.16666666666666666)
Sample(state=|11>, probability=0.5, amplitude=None, intermediate_measurements=None, err=0.16666666666666666)
```

L'autre information importante à noter ici, avec la simulation stricte nous n'avons pas accès à l'amplitude donc **amplitude= None**. **Avec un vrai QPU nous n'aurons jamais accès à l'amplitude**.

Si nous le souhaitons, nous pouvons définir uniquement un sous-ensemble de qubits à mesurer à la fin de notre expérimentation. Pour cela, nous utilisons l'argument **qubits =[ ]** dans le paramètre de la fonction **to_job** en spécifiant l'indice du qubit à mesurer ou une liste d'indices de qubits si nous souhaitons mesurer plusieurs qubits.

Lorsque nous mesurons un sous-ensemble de qubits, nous n'aurons pas accès à l'amplitude. Néanmoins, nous pouvons avoir accès à l'estimation de probabilité.

```
job = circuit.to_job(qubits=[0]) #creating job from circuit only on the first qubit

    results = qpu.submit(job) #submitting job to QPU instance, getting results.
```

```
Sample(state=|0>, probability=0.4999999999999999, amplitude=None, intermediate_measurements=None, err=None)
Sample(state=|1>, probability=0.4999999999999999, amplitude=None, intermediate_measurements=None, err=None)
```

Vous pouvez aussi avoir accès à l'aide :

```
help(circ.to_job)
```

```
Help on method to_job in module qat.core.wrappers.circuit:

to_job(job_type='SAMPLE', qubits=None, nbshots=0, aggregate_data=True, amp_threshold=9.094947017729282e-13,
**kwargs) method of qat.core.wrappers.circuit.Circuit instance
  Generates a Job containing the circuit and some post
  processing information.

  Args:
    job_type (str): possible values are "SAMPLE" for computational basis sampling
      of some qubits, or "OBS" for observable evaluation (see :py:mod:`qat.application.observables`
      for more information about this mode).
    qubits (optional, list<int>, list<QRegister>): the list of qubits to measure (in "SAMPLE" mode).
      If some quantum register is passed instead, all the qubits of the register will be...
```